Assignment 7 (2 parts)

80 Points, in two parts

Part I: Time Sliders with ArcGIS JavaScript API (40 points)

The first section of this assignment will walk you through the steps of working with time-enabled data from Esri. Normally, you would be creating data in ArcGIS Pro or ArcMap desktop, setting the time properties for the map layers, and then publishing those data as a Feature Layer or feature service (to either an ArcGIS for Server instance or ArcGIS Online). In the interest of time (and an absence of ArcGIS Online accounts), we will be skipping those steps and working with an existing data layer, global earthquakes.  
  
In your head section, you load in Esri’s scripts and set the styles. Make SURE to replace any slant quotes with straight quotes if you copy-paste this:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport"

content="initial-scale=1,maximum-scale=1,user-scalable=no" />

<title>Lab 7</title>

<link rel="stylesheet"href="https://js.arcgis.com/4.18/esri/themes/light/main.css" />

<script src="https://js.arcgis.com/4.18/"></script>

<style>

html,

body,

#viewDiv {

padding: 0;

margin: 0;

height: 100%;

width: 100%;

}

#timeSlider {

position: absolute;

left: 5%;

right: 5%;

bottom: 20px;

}

</style>

<**script**>

This final tag on the above section begins the “script” that will fire all the functions and layering for your map. Virtually all the code will go in this section. The script should begin with the dojo require section (you may recall from assignment 4) – this loads in the resources for all the widgets we’ll use. Here you will load in the map and map view features, as well as the feature layer and legend widgets you’ve already used in previous examples. But notice that “Time Slider” also appears here, too (a new feature we haven’t used before). Please add this section after the script tag, but before you add anything else:

require([

"esri/Map",

"esri/views/MapView",

"esri/layers/FeatureLayer",

"esri/widgets/TimeSlider",

"esri/widgets/Legend"

], function (Map, MapView, FeatureLayer, TimeSlider, Legend) {

Notice a few things here:

1) The require section begins with a (. That means at the end of your script section, before the </script> tag to close, you will need to end with a matching ) to bind it all together.

2) Likewise, your functions list opens with a { which means that you’ll also need to close with a matching } when you’re done with the script for the map that fires functions. Taken together, this explains why the code usually has a concluding line that looks something like this: **});**

3) Also, the *order* in the list of resources matches the *order* in functions list that follows it. This is not only good practice, it’s also often necessary to make the page to function properly.

This list also tells us something about the expectations of the following code we’ll be writing. We’ll want to define a map, a map view, a feature layer, a time slider, and a legend. However, just because we’ll be defining/adding all those things doesn’t mean we’ll be adding them ***in that order***. Rather, the map feature expects to know what layers will be loaded in, so the feature layers have to be defined first; the map view expects to know the map’s properties, so the map variable has to come before the view; the legend expects to know the properties of the map view before it can be fired, so the view has to come before the legend, et cetera…  
  
So, we will begin by defining the thing that has to come first – the map layer itself. We are adding just one layer here first. This item has time properties that are recognized by Esri, and it is a Feature Layer:

const layer = new FeatureLayer({

url: "https://services9.arcgis.com/RHVPKKiFTONKtxq3/ArcGIS/rest/services/USGS\_Seismic\_Data\_v1/FeatureServer/0"

});

Next, we can add the map that will use the feature layer:

const map = new Map({

basemap: "hybrid",

layers: [layer] //if you add more layers, separate by a comma

});

The map view will be next:

const view = new MapView({

map: map,

container: "viewDiv",

zoom: 2, //zoomed almost all the way out to see the globe

center: [-150, 0] //set this somewhere in the middle of the Pacific

});

Now we have the map starting position set, we have a base map, and we have a layer drawn onto the map. It is time to begin adding map elements like the Legend as well as interactive functions, in this case, the time slider -

const timeSlider = new TimeSlider({

container: "timeSlider",

view: view,

timeVisible: true, // show the time stamps on the timeslider

loop: true

});

view.whenLayerView(layer).then(function (lv) {

// this just rounds up the time extent to full hours

timeSlider.fullTimeExtent = layer.timeInfo.fullTimeExtent.expandTo("hours");

});

const legend = new Legend({

view: view

});

view.ui.add(legend, "top-left"); //you can position the legend here

});

</script>

</head>

Now we have the entire script section completed for the basics of what we want. It’s still not done! This is just the head section of the HTML document where we’ve preemptively loaded in all the scripts for functionality. We still need to give the map a place on the physical page the user sees. So, in the “body” section, we’ll put in a div, essentially the window where our map will live. We will also create a separate div for the time slider. Notice that these div items link up with their corresponding names in the style section on page 1.

<body>

<div id="viewDiv"></div>

<div id="timeSlider"></div>

</body>

</html>

At this point, the basics of the page are in place. But let’s experiment with adding another interactive element to the page: a pop up for the earthquake data.

In the ArcGIS JS API, pop ups do not need to be declared in the top/require section if you’re using them only to derive data from existing feature layers. However, you will need to modify the feature layer properties in order to make them work properly. We return to the feature layer variable to make some additions (note the new comma after the URL):

const layer = new FeatureLayer({

url: "https://services9.arcgis.com/RHVPKKiFTONKtxq3/ArcGIS/rest/services/USGS\_Seismic\_Data\_v1/FeatureServer/0"**,**

outFields: ["depth", "mag"], // used to specify what attribute fields to query

popupTemplate: popupTemplate

});

Here we specify the “outFields” as any of the attributes in the feature layer that we want to query for information. If you click on the link, you will see that each attribute field name is listed. Notice also that the feature layer property “popupTemplate” is set to a variable called “popupTemplate.” We haven’t defined that yet! So **before** the layer variable (but after the require section), we’ll need to do that, much like you did with renderers in Lab 4.

So add this section to the location I’ve just described:

const popupTemplate = {

// autocasts as new PopupTemplate()

title: "this event took place at {depth} feet deep",

content: [

{

type: "fields",

fieldInfos: [

{

fieldName: "mag",

label: "Magnitude",

format: {

places: 0,

digitSeparator: true

}

}]}]};

I have simplified this substantially, so there’s only one field being populated into the table of data (the magnitude of the earthquake) that pops up. You could use commas here to extend this template further and include additional attributes for each point on the map, when clicked. Notice also that there is a pop up TITLE field. It draws from the attributes as well. In this case, I’ve got it set to {depth} so that it pulls the recorded depth of the quake and populates it into the pop up.

Now you have everything you need for a functional, time-enabled map with pop ups (10 points)

On your own, do the following steps:  
1) change the pop-up template so that **two more attributes** from the earthquake layer are loaded into the window when you click on a point [look at the feature layer URL for ideas] (10 points)

2) change the **pop-up template title** in any way you’d like, as long as it dynamically loads in an attribute [e.g. “depth”] as part of the title (10 points)

3) change out the **base map** with a different option [consult lab 4 if needed] (10 points)

Upload your finished work to your GitHub pages account and submit the link.

Assignment 7, Part II: Leaflet and Sliders (40 points)

There are a number of instances in which you may want to use a time slider with data, but your data are not necessarily formatted as dates or number types. For example, imagine you are working with historical datasets where you have “fuzzy” information about time, but you still have a general order in which you’d like events to appear on a map (like an occurrence on 05/1885 and something in 1886 and an event on the day 02/15/1887 in a spreadsheet; or even something more general than that, such as features in the Cretaceous and Jurassic eras). Most mapping frameworks will force you to choose a specific date format. Furthermore, at present, major mapping software usually will insist that you use some derivative of MM/YYYY or YYYY-DD or DD/MM/YYYY, forcing you to use a date format that isn’t conducive to ancient and prehistorical dates. This second part of the exercise provides a framework for dealing with these types of data situations as well as handling standard dates/times using an open-source framework, Leaflet.JS.

We begin with a standard template for a Leaflet map application, just a tile layer with no data loaded in:

<!DOCTYPE html>

<head>

<meta charset="utf-8">

<title>Lab 7 part 2</title>

<link rel="stylesheet" href="https://unpkg.com/leaflet@1.3.4/dist/leaflet.css"/>

<script src="https://unpkg.com/leaflet@1.3.4/dist/leaflet.js"></script>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>

<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>

<style>

html, body { margin: 0; padding: 0; height: 100%; }

#map { min-height: 100%; }

</style>

</head>

<body>

<div id="map"></div>

<script>

var map = L.map('map', {

center: [38.1625228, -96.9633171],

zoom: 4

});

L.tileLayer('https://stamen-tiles-{s}.a.ssl.fastly.net/toner-lite/{z}/{x}/{y}{r}.{ext}', {

attribution: 'Map tiles by <a href="http://stamen.com">Stamen Design</a>, <a href="http://creativecommons.org/licenses/by/3.0">CC BY 3.0</a> &mdash; Map data &copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',

subdomains: 'abcd',

minZoom: 0,

maxZoom: 20,

ext: 'png'

}).addTo(map);

</script>

</body>

</html>

Once again, be careful that you are using straight quotes and not *slanted*, formatted quotes if you are copying code.

Notice that some new scripts are added relating to jQuery, another JavaScript library for data processing and manipulation. Rather than loading in .JS files with data saved into arrays that look very much like GeoJSON files, this time we’ll actually be calling in and processing a true GeoJSON file. This process complicates the code slightly – this will become apparent as you step through the exercise and it will be explained later. The gist of it is that you’re not “extending” your HTML document to include your data as a variable as you’ve done in the past, but rather you’ll be selectively loading it into the document separately using an AJAX call. So in the meantime, before we get there, we will need to make a slight change to our head section to make sure we don’t run into cross-domain issues:

<link rel="stylesheet" href="https://unpkg.com/leaflet@1.3.4/dist/leaflet.css" integrity="sha512-puBpdR0798OZvTTbP4A8Ix/l+A4dHDD0DGqYW6RQ+9jxkRFclaxxQb/SJAWZfWAkuyeQUytO7+7N4QKrDh+drA==" crossorigin=""/>

<script src="https://unpkg.com/leaflet@1.3.4/dist/leaflet.js" integrity="sha512-nMMmRyTVoLYqjP9hrbed9S+FzjZHW5gY1TWCHA5ckwXZBadntCNs8kEqAWdrb9O7rxbCaA4lKTIWjDXZxflOcA==" crossorigin=""></script>

<script src="https://code.jquery.com/jquery-3.3.1.min.js" integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=" crossorigin="anonymous"></script>

<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js" integrity="sha256-VazP97ZCwtekAsvgPBSUwPFKdrwD3unUfSGVYrahUqU=" crossorigin="anonymous"></script>

Now, we’ll load in that data somewhat differently than we’ve been doing before. We use the getJSON() function built into jQuery to acquire the geojson file, parse it as JSON (read it like a JavaScript object), and finally we have a little function that fires when that’s done so that those data are turned into proportional symbols and processed into groups based on timestamps, and then format a time slider with the information it pulls in… all of which we’re referring to as a variable named “info” to store it on a user’s browser. This isn’t magic stuff – we actually haven’t yet defined what a lot of this means yet. Essentially, we’re just saying that this process *should* happen when jQuery grabs the data, but we still need to articulate a lot of this in the rest of the code. For example, *how* should the proportional symbols look? We’ll be doing that in subsequent steps. Put this after your tile layer gets added, but make sure it’s within the <script> tags in the body of your HTML document:

$.getJSON("https://rshepard2.github.io/city.geojson")

.done(function(data) {

var info = processData(data);

createPropSymbols(info.timestamps, data);

createSliderUI(info.timestamps);

});

By the way, that dollar sign is common in scripts using jQuery. Essentially, it’s shorthand for saying this is a jQuery function. Now, it is time to explain to the browser what exactly all that means. We start by defining the processData function and articulating how exactly information for timestamps should be extracted from the dataset. Add this after your getJSON() call above:

function processData(data) {

// First, initialize the variables to hold the timestamps and min/max population values

var timestamps = []; // square brackets to define an array of data

// because there are multiple timestamps

var min = Infinity; // for the min, begin with largest possible value - infinity

var max = -Infinity;// for the max, smallest possible value - negative infinity

// Go through each row/feature of the data table

// Note data is the variable name in the function definition - processData(data)

for (var feature in data.features) {

var properties = data.features[feature].properties;

// At each row, go through the columns/attributes to get the values

for (var attribute in properties) {

if ( attribute != 'id' &&

attribute != 'name' &&

attribute != 'latitude' &&

attribute != 'longitude' ) // != means NOT EQUAL TO

// These 3 columns are NOT recorded. Add more if you have non-data fields.

{

if ( $.inArray(attribute,timestamps) === -1) { // JQuery in.Array() method searches for value within an array and return its index (or -1 if not found)

// here, the new timestamp is only added when it is not already in the array

// triple equals === compares both type and value to see if there’s a match

timestamps.push(attribute);   
 // The JS push() method adds new items to the end of an array

// and returns the new length of the array

}

if (properties[attribute] < min) {

min = properties[attribute]; // record/update current smaller values as min

}

if (properties[attribute] > max) {

max = properties[attribute]; // update current larger values as the max

}

}

}

}

return {

timestamps : timestamps,

min : min,

max : max

}

}

Note that if you use your own data to add more columns of information for each data point, you’re going to want to add the field names that you want to exclude from the processing function (so that it ignores any quantitative data in those sections). The above code can process a geojson or csv file and separate out any actual data fields you want to use (and if you’re using numbers for headings, it should be able to parse that out and arrange it in a slider), but it still doesn’t tell the browser how to draw the data at each time. So the next block of code here is a function that creates proportional symbols, for each “column” or attribute field in your dataset. But if you look closely at the code, you’ll see that – within the function – this looks remarkably similar to previous lab exercises with regard to adding a geoJSON layer and then using the pointToLayer process to draw in circle markers based on data values (we did this with the COVID cases by Chinese provinces, for example). You will set the default circle icon and also set its fill and border colors in the same location.

We’ll also write in some code here to define a “hover over” or “mouseover” effect, which is similar to a pop up, however it doesn’t actually require the user to *click* on the dataset to return an informative window. Rather, whenever the cursor hovers over the data points, their information will be returned. Notice how you can also set the style of markers dynamically during a hover over event. You can specify a color with text for standard colors like blue or green, or use hex colors if you want to specify a particular hue.

To do all of that, we add this code after the data processing code provided above:

function createPropSymbols(timestamps, data) {

cities = L.geoJson(data, {

// By default, Leaflet draws geojson points as simple markers

// To alter this, the pointToLayer function needs to be used

pointToLayer: function(feature, latlng) {

return L.circleMarker(latlng, { // we use circle marker for the points

fillColor: "#501e65", // fill color of the circles

color: '#501e65', // border color of the circles

weight: 2, // circle line weight in pixels

fillOpacity: 0.5 // fill opacity (0-1)

}).on({

mouseover: function(e) {

this.openPopup();

this.setStyle({fillColor: 'green'}); // fill color turns green when mouseover

},

mouseout: function(e) {

this.closePopup();

this.setStyle({fillColor: '#501e65'}); // fill turns original color when mouseout

}

});

}

}).addTo(map);

updatePropSymbols(timestamps[0]); // this function is defined below

// When loaded, the map will first show proportional symbols with the first timestamp's data

}

At this point, we have a mechanism for importing and processing data, and the only remaining steps would be to create a slider, make sure that the data get loaded into that slider (for querying information by date) and then we’ll also want to set a starting time for the slider as well. In the example code, I use “1950” below because this is the first column in my dataset that I want to use to represent data, and I want it to step through the subsequent columns in increments of 10 years (so I have the “step” part of the slider set to 10). If you use your own layers with different years, you will want to change those parts accordingly:

Go ahead and add this code after the previous section to put in a slider that will step through the dataset:

function createSliderUI(timestamps) {

var sliderControl = L.control({ position: 'bottomleft'} ); // position of the slider

// Another use of L.control :)

sliderControl.onAdd = function(map) {

//initialize a range slider with mousedown control

var slider = L.DomUtil.create("input", "range-slider");

L.DomEvent.addListener(slider, 'mousedown', function(e) {

L.DomEvent.stopPropagation(e);

});

// Define the labels of the time slider as an array of strings

// Modify this for your data

var labels = ["1950", "1960", "1970", "1980", "1990", "2000", "2010"];

<!-- $(slider).mousedown(function () { -->

<!-- map.dragging.disable(); -->

<!-- }); -->

<!-- $(document).mouseup(function () { -->

<!-- map.dragging.enable(); -->

$(slider)

.attr({

'type':'range',

'max': timestamps[timestamps.length-1],

'min':timestamps[0],

'step': 10, // Change this to match the interval between adjacent timestamps

'value': String(timestamps[0])

})

.on('input change', function() {

updatePropSymbols($(this).val().toString()); // updates map for timestamp

var i = $.inArray(this.value,timestamps);

$(".temporal-legend").text(labels[i]); // updates label for the timestamp

})

.mousedown(function () {

map.dragging.disable();

})

return slider;

}

sliderControl.addTo(map);

createTimeLabel("1950"); //The starting timestamp label

}

Also, we want to add labels to the time slider and give it a location on the map area:

// Add labels to the time slider when the map first loaded

function createTimeLabel(startTimestamp) {

var temporalLegend = L.control({position: 'bottomleft' }); // same position as the slider

// One more use of L.control !!

temporalLegend.onAdd = function(map) {

var output = L.DomUtil.create("output", "temporal-legend");

$(output).text(startTimestamp);

return output;

}

temporalLegend.addTo(map);

}

We are almost done. Finally, we want to define how to scale the symbols based on quantities in the data. Our very last piece of code before we close the script is based around updating the symbols for each time. This seems to be an exception to some of the APIs we’ve used in the past (which normally expect you to describe styles first) in that we can actually explain how to scale (the calcPropRadius function) after we call the function. This time, though, we’re dynamically loading in a data layer with jQuery rather than being rendered when the page loads, so the code here fires first. You can put this right before you close the script section with the </script> closing tag:

// function to update/resize each circle marker according to value in the time series

function updatePropSymbols(timestamp) {

cities.eachLayer(function(layer) { // eachLayer() is an Leaflet function to iterate over the layers/points of the map

var props = layer.feature.properties; // attributes

var radius = calcPropRadius(props[timestamp]); // circle radius, calculation function defined below

// pop-up information (when mouseover) for each city is also defined here

var popupContent = props.name + ' ' + timestamp + ' population: ' + String(props[timestamp]);

layer.setRadius(radius); // Leaflet method for setting the radius of a circle

layer.bindPopup(popupContent, { offset: new L.Point(0,-radius) }); // bind the popup content, with an offset

});

}

// calculate the radius of the proportional symbols based on area

function calcPropRadius(attributeValue) {

var scaleFactor = 0.001; // the scale factor is used to scale the values; the units of the radius are in meters

// you may determine the scale factor accordingly based on the range of the values and the mapping scale

var area = attributeValue \* scaleFactor;

return Math.sqrt(area/Math.PI); // the function return the radius of the circle to be used in the updatePropSymbols()

}

Now you’ve reached the end of the build. Your code is very long, but also very simple – a series of small functions pieced together that tell your document how to display and change data based on user selections. You can apply this to any dataset. In my GeoJSON, I have population data, for 1950 to 2010, from the cities proper for the 15 largest U.S. cities in the United States. Download and process this or transform it using skills you’ve acquired in previous exercises.

Your challenge will be to add 2020 data to your map, using skills you’ve acquired in previous exercises. You can download the GeoJSON and transform it within QGIS. Or do it all as a CSV and then convert it into a GeoJSON with Ogre or QSGIS. Alternatively, use your own dataset, just make sure to closely follow comments and steps I’ve outlined here and make the requisite changes. It’s entirely up to you. Complete this task and share a working URL for **20 points**.

Part 2.2 is next – it is a much shorter part of the exercise overall. It also uses a time slider, but does not proportionally scale symbols and it does not require you to sort each “step” of your slider into separate columns. This outlines another way to handle “fuzzy” dates in your data but still keep them sorted in some kind of order (which might be relevant to some of your final projects!). We’ll also introduce a search/filtering tool via Fuse.JS, another library that can parse and search through your GeoJSON and return only the results the user selects.

Begin by adding in the scripts for Leaflet, as well as a few new ones. We’re also going to be adding scripts for “Leaflet Slider” and “Leaflet Fuse” as well as “Leaflet Fuse Search.” These are standalone scripts that are being hosted by University of Iowa, but any of them can be downloaded and hosted on your own GitHub, if you want further customization. We’ll also load in jQuery scripts as well as a dataset, the geocoded 1870 census for the city of Omaha. Finally, we’ll load in some stylesheets. Begin with this:

<!DOCTYPE html>

<html>

<head>

<title>Lab 7 part 2.2</title>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="stylesheet" href="https://unpkg.com/leaflet@1.3.4/dist/leaflet.css" crossorigin=""/>

<script src="https://unpkg.com/leaflet@1.3.4/dist/leaflet.js" crossorigin=""></script>

<script src="https://dsps.lib.uiowa.edu/placing/public/leafletSlider-1.0.2/leaflet.SliderControl.min.js"></script>

<script src="https://dsps.lib.uiowa.edu/placing/public/fuse-1.2.1/fuse.min.js"></script>

<script src="https://dsps.lib.uiowa.edu/placing/public/leafletFuseSearch-noVersion/leafletfuse.js"></script>

<link rel="stylesheet" href="https://dsps.lib.uiowa.edu/placing/public/leafletFuseSearch-noVersion/leafletfuse.css" type="text/css"/>

<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>

<script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>

<link rel="stylesheet" href="http://code.jquery.com/ui/1.9.2/themes/base/jquery-ui.css" type="text/css">

<script src="https://dsps.lib.uiowa.edu/placing/public/jqueryUiTouchPunch-0.2.2/jquery.ui.touch-punch.min.js"></script>

<script src="https://dsps.lib.uiowa.edu/placing/omadata.js"></script>

<link rel="stylesheet" type="text/css" href="https://dsps.lib.uiowa.edu/placing/css/style.css">

<link rel="stylesheet" type="text/css" href="https://dsps.lib.uiowa.edu/placing/css/navwrap.css">

</head>

Now, once the head section is complete we can turn our attention to the scripts in the body of the document. You will notice that this section looks very familiar to the code we’ve already used in Lab 2 – we make a div window in which the map we’ll live, and the rest is a Leaflet map loading in a custom tile layer and then also setting some rules for a pop up feature. (Remember, this takes advantage of Leaflet’s “onEachFeature” functionality so that every time a user clicks on a feature, they will return a set of attribute data that we customize here). By default, this is set to pull out layers from Omaha’s census records, like name, race, and occupation. After the head section, you will begin the body with this code:

<body>

<div id="map" style="width: 100%; height: 650px"></div>

<script>

var basemap = L.tileLayer('https://stamen-tiles-{s}.a.ssl.fastly.net/toner-lite/{z}/{x}/{y}{r}.{ext}', {

attribution: 'Map tiles by <a href="http://stamen.com">Stamen Design</a>, <a href="http://creativecommons.org/licenses/by/3.0">CC BY 3.0</a> &mdash; Map data &copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',

subdomains: 'abcd',

minZoom: 0,

maxZoom: 20,

ext: 'png'

});

var map = L.map('map', {

center: [41.259944, -95.937057],

zoom: 15,

layers: [basemap]

});

// onEachFeature generate a popup for each location

var onEachFeature = function(feature, layer) {

if (feature.properties) {

var prop = feature.properties;

// make a cool little html popup with properties

var popup = '<h3>'+prop['FNAME']+' '+prop['LNAME']+'</h3>'+'<strong>Race/Gender: </strong>'+prop['RACE']+' '+prop['GENDER']+'<br><strong>Occupation: </strong>'+prop['OCCUPATION']+'<br><strong>Age: </strong>'+prop['AGE']+'<br><strong>Family Estate: $</strong>'+prop['PERSREAL']+'<br> <strong>Fam. Real Estate: $</strong>'+prop['FAMREAL']+'<br><strong>Address: </strong>'+prop['ADDRESS']+'<br><br><i>details from Census Bureau and city directories</i>';

// add known info about event to the description

// you must create a layer property on each feature or else

// the search results won't know where the item is on the map / layer

feature.layer = layer;

layer.bindPopup(popup, {maxWidth: "auto"});

}

}; // end onEachFeature

You will be able to change that var popup definition to point to any attributes you like – just note that the field names will be loaded in here through prop[‘FieldName’], because we have set “prop” to be feature.properties, essentially grabbing the JSON properties of each feature and searching them for corresponding field attributes. Something like prop[‘AGE’] will return the value in the field “AGE” in the original JSON data.

Next we will style some of the markers before we use them. Leaflet “circleMarkers” is a known style type that will create circle markers to specifications. We’ll create three types of circle markers – orange, green and blue ones – but note that the variable names are whatever we want to call them. That means “orange” circle markers can have a yellow hex value, and it will draw yellow circles, rather than orange.   
Here is the style section that follows the above code:

var blueCircleMarkers = {

radius: 4,

fillColor: "#00AEEF",

color: "#3978BF",

weight: 1,

opacity: 0.8,

fillOpacity: 0.5

};

var orangeCircleMarkers = {

radius: 5,

fillColor: "#ED1C24",

color: "#ED1C24",

weight: 1,

opacity: 1,

fillOpacity: 0.7

};

var greenCircleMarkers = {

radius: 5,

fillColor: "#ffff00",

color: "#286933",

weight: 1,

opacity: 1,

fillOpacity: 0.7

};

Next, we do something a bit unique. We’ll add our layer and set it to one of the styles we just added, but we’ll also want to take control over how the layer appears if we activate the time slider or use the filtering/search tool. We can do that by adding the layer (omaJson) three times as different variables, and we’ll associate each with a unique style, but use the same pop up:

var firstlayer = L.geoJson(omaJson, {

// add popups

onEachFeature: onEachFeature,

pointToLayer: function(feature, latlng) {

return L.circleMarker(latlng, blueCircleMarkers);

}

}).addTo(map);

var timelineLayer = L.geoJson(omaJson, {

onEachFeature: onEachFeature,

pointToLayer: function(feature, latlng) {

return L.circleMarker(latlng, orangeCircleMarkers);

}

});

// now make a timeline layer that the slider can manipulate

var searchLayer = L.geoJson(omaJson, {

onEachFeature: onEachFeature,

pointToLayer: function(feature, latlng) {

return L.circleMarker(latlng, greenCircleMarkers);

}

});

Finally, we add slider controls and define the parameters of what should appear in a search field if somebody uses the search feature. We’ll define that next:

var sliderControl = L.control.sliderControl({

position: "topright",

layer: timelineLayer,

range: true // gives it a bottom and a top slider

});

map.addControl(sliderControl);

sliderControl.startSlider(); // initialize

var searchOptions = {

position: 'topleft',

title: 'Search',

placeholder: 'Example: John Smith',

maxResultLength: 10,

caseSensitive: false,

showInvisibleFeatures: true,

layerToToggle: searchLayer,

threshold: 0.5, // default is .5, will match imperfect results

showResultFct: function(feature, container) {

props = feature.properties;

var name = L.DomUtil.create('b', null, container);

name.innerHTML = props.LNAME;

container.appendChild(L.DomUtil.create('br', null, container));

var cat = props.LNAME ? props.FNAME : props.OCCUPATION,

info = cat;

container.appendChild(document.createTextNode(info));

}

};

var searchControl = L.control.fuseSearch(searchOptions);

map.addControl(searchControl);

searchControl.indexFeatures(omaJson, ['LNAME']);

displayFeatures(omaJson, searchLayer);

function displayFeatures(features, layer) {

var popup = L.DomUtil.create('div', 'tiny-popup', map.getContainer());

for (var id in features) {

var feat = features[id];

var cat = feat.properties.categorie;

var site = L.geoJson(feat, {

pointToLayer: function(feature, latLng) {

var marker = L.marker(latLng, {

icon: orangeCircleMarkers,

keyboard: false,

riseOnHover: true

});

if (! L.touch) {

marker.on('mouseover', function(position) {

}).on('mouseout', function(position) {

L.DomUtil.removeClass(popup, 'visible');

});

}

return marker;

},

onEachFeature: onEachFeature

});

if (layer !== undefined) {

layer.addLayer(site);

}

}

return layer;

};

And last, we set the parameters for the toggle box and also add that to Leaflet’s “control” options for the user:

var overlays = {

"People of Color": firstlayer

};

var baseMaps = {

"Historic": basemap

};

L.control.layers(overlays, baseMaps, {collapsed:false}).addTo(map);

</script>

</body>

</html>

Now that you have a working example, your will make this work with another data layer instead. You can use any data layer you’d like, or you can use my provided layer (Brinton traveling motion picture shows [provided as an Excel file]). For **20 points**, you will need to:

* Refocus the map to center on your area of interest (5 points)
* Change the pop-up text to correspond to the dataset (5 points)
* Make sure the slider works with the order in which you’d like your data to appear (5 points) – *note, name your time field “time” and sort it in ascending order BEFORE transforming it into a GeoJSON / JS file to ensure that it draws in chronological order*
* Change the text in search control so that it indexes features based on Name or some other attribute in the data (5 points)

Share your .HTML file with me on Canvas to complete this assignment, Assignment 7 Part 2.2 -