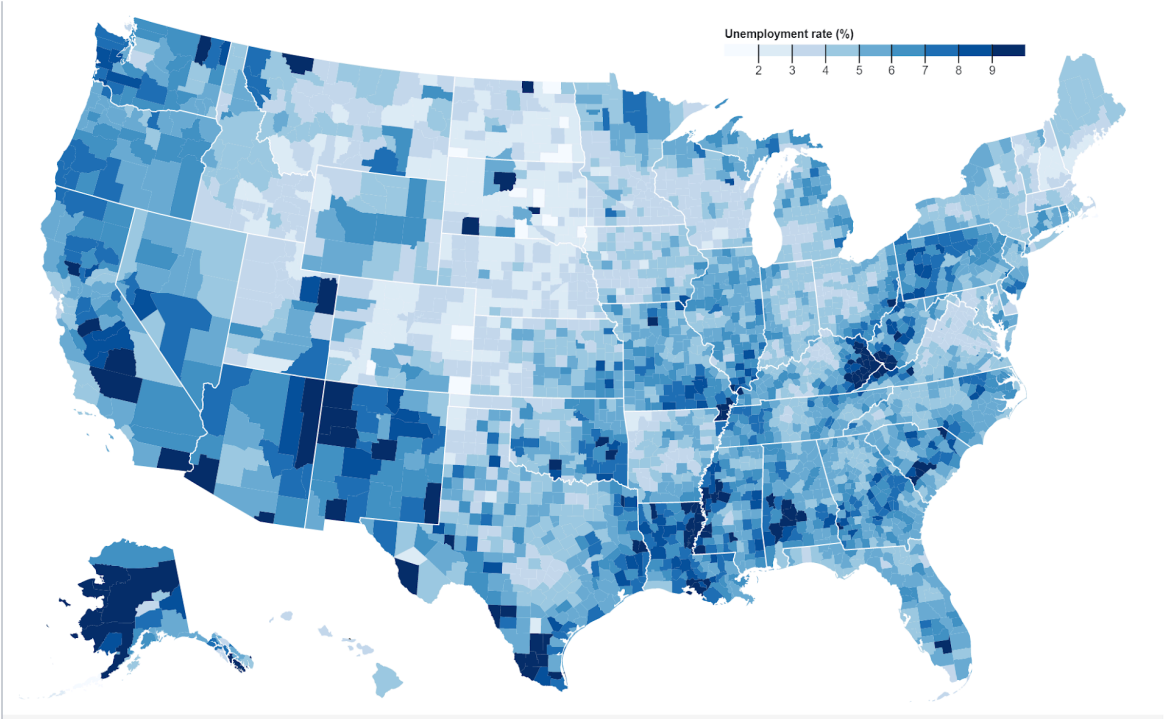
Week 10 - Intro to D3.js: Mapping Data with D3

D3 and Web Mapping: Map your Data!

It might not surprise you, given the robust visualization capabilities of D3, that it is also great for creating interactive web maps and mapping datasets. Mapping in D3, in fact, is easier than you might think. Drawing a basic map doesn't take any more code than a bar chart, line chart, or scatter plot. Web mapping in D3 can allow for animation, visualization, and interaction. You can also coordinate charts and plots with your map. As we go along, you will learn D3 supports topology and even projections, making it very powerful! In this session, let's introduce mapping with D3 by going over some fundamentals, getting some data on the map, and then showing one example of how we can do a time slider interaction. Lastly, we'll look at TopoJSON, a GeoJSON that supports topology, and some examples of choropleth maps and how you can create those.

The map below is a nice example of a choropleth map from Mike Bostock, the creator of D3. It uses a number of features you might be familiar with if you have done some web mapping. The county geometry is stored as a JSON (a TopoJSON in fact, but more on this later...), and the data in a Tab-separated format (TSV). The rest of the implementation is D3 JavaScript. One important observation, note that the map is not in a Web Mercator projection, but rather an Albers projection. D3 has a large built in projection library that you can reference when mapping data.

Unemployment rate by county, August 2016. Data: Bureau of Labor Statistics



U.S. Unemployment by County, 2016

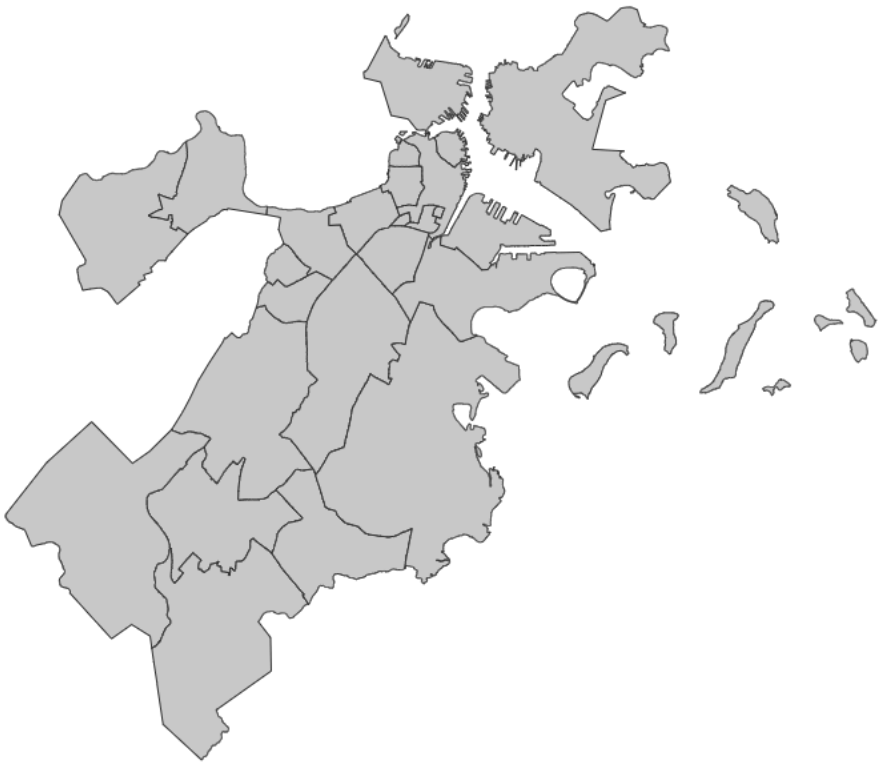
(View this example on its own.) - (View this on bl.ocks.)

So how does this work?

The following tutorial will explore, in two parts, how mapping works with D3.

Part 1: D3 Mapping from the Ground Up

Open a new blank page in your text editor. We'll write our D3 map code here. Let's start a bit more simple and revisit our rat example and create a basic map of Boston neighborhoods. This will let us see what is happening under the hood, and we can learn some of the fundamentals and build from them to create better maps and spatial visualizations. To start, we'll create the following.



A Simple D3 Map: Boston Neighborhoods (Click to view this example on its own.)

In your blank document, input the following.

1. HTML Template

A simple HTML template for our page is set up as follows, this is provided for you in the *Week_9_D3_Map_part1.html* file. *Note: we are using d3.v5 here*.

```
<html lang='eng'>
    <head>
        <title>
            Mapping wiht D3
        </title>
        <script src = "https://d3js.org/d3.v5.min.js" charset="utf-8"></script>
    </head>
    <body>
        <!-- page elemets and content go here. -->
        <script>
            //Our D3 code will go here.
        </script>
    </body>
</html>
```

2. Store Geographic Dataset URL

Load the GeoJSON file of Boston neighborhoods into our document. Make sure the path is correct.

The data url path you should be stored is here:

https://gist.githubusercontent.com/jdev42092/5c285c4a3608eb9f9864f5da27db4e49/raw/a1c33b1432ca2948f14f656cc14c7c7335f78d95/boston_neighborhoods.json

Copy this entire line and put it inside the body>>script tag as shown here:

Notice that this dataset is hosted online. We will use d3.json() to load it later.

```
<script>
    //2. Load the data url:
    var bosNeighborurl='https://gist.githubusercontent.com/jdev42092/5c285c4a3608eb9f9864f5da27db4e49/
```

var bosNeighborurl='https://gist.githubusercontent.com/jdev42092/5c285c4a3608eb9f9864f5da27db4e49/raw/a1c33b1432ca2948f14f656cc14c7c7335f78d95/boston_neighborhoods.json'

3. Create the SVG Canvas

Between the script tags, add the following to create our SVG canvas.

```
//3.Create SVG canvas
//Width and height of the whole visualization
var width = 700
var height = 580


var svg = d3.select("body")
            .append("svg")
            .attr("width", width)
            .attr("height", height)

//Append empty placeholder g element to the svg
//g will contain geometry elements
var g=svg.append("g")
```

4. Projections

D3, Projections, and GeoJSON

D3 has some internal functionality that can turn GeoJSON data into screen coordinates based on the projection you set. This is not unlike other libraries such as Leaflet, but the result is much more open-ended, not constrained to shapes on a tiled Mercator map.[1] So, yes, D3 supports projections. Jason Davies has a fantastic illustration of various projections in this example.

D3 has a handful of projections built in, but there are tons more supported in an external plugin.

In our example, we set properties for our projection by passing them to a projection object (d3.geoAlbers()) and save them as a variable. We can refer to this when we use the D3 object for generating map linework. The properties are **scale**, **rotate**, **center**, and **translate**.

- **Scale** sets the scale of the map (ie 1 is the smallest, the larger the number the more zoomed in you are, you can fiddle with this until it works). Rotate and center set parameters for the project. A full global map fits nicely around 100. The extent of Boston lands at about 190,000 with our extents.
- **Rotate** the map sets the longitude of origin for our Albers projection. **Center** sets a single standard parallel at 42.313, about the latitude of Boston.
- Lastly, **translate** is a pixel offset, commonly specified to ensure that the center of the projection is in the center of the viewing area.

This centers our map on Boston, zooming it into the extent of the city. Set up projection Parameters after your width and height setup.

```
//4.Set Projection Parameters
var albersProjection = d3.geoAlbers()
    .scale( 190000)
    .rotate([71.057, 0])
    .center([0,42.313])
    .translate([width/2, height/2])
```

5. Set up the Path Generator

Path Generators - Geo Paths

Next in our code, we create something called a Path Generator, or Geo Path. The primary mechanism for displaying data in D3 is d3.geoPath. This class is similar to d3.svg.line and the other SVG shape generators: given a geometry or feature object, it generates the path data string suitable for the "d" attribute of an SVG path element.

Basically, the Geo Path generator is a function. One of the methods it takes is the projection you define. It reads lat/lon coordinates from a GeoJSON feature, algorithmically turns them into screen

coordinates according to your specification in the projection method, and returns an SVG path string. This can then be drawn on your screen. We set the parameters in our code already, so this code is simple.

```
//5.Create GeoPath function that uses built-in D3 functionality to turn
//lat/lon coordinates into screen coordinates;
var geoPath = d3.geoPath()
    .projection(albersProjection)
```

6. Load the Data to the SVG Canvas

Classic D3 Binding - Use GeoPath

We use **d3.json(url)** to load the data. Note that the variable name **bosNeighbor** is in the function. This is typically how d3.v5 loads data and stores it in a variable.

The rest is just back to regular D3, use a selector for the elements, load the data from the JSON, bind it to the SVG using enter and append, this apply styling. The one part that might be confusing is the setting of the **d** attribute. This is the attribute that defines the coordinates of a path. We pass a function to it that draws the path according to the coordinates defined by the function, and the coordinates are provided by our GeoJSON.

```
//6. Load the data and bind data to svg canvas

d3.json(bosNeighborurl).then(function(bosNeighbor){
    //6. Bind the data to svg canvas
    //Classic D3, select non-existent elements, bind the data, append the elements, and apply attributes
    g.selectAll("path")
        .data(bosNeighbor.features)
        .enter()
        .append("path")
        .attr("fill", "#ccc")
        .attr("stroke", "#333")
        .attr("d",geoPath)

})
```

Using a GeoJSON of Boston Neighborhoods, The result is the following map. Set up your map with only one layer, then the rest comes easy.

Now, refresh your html online, it should show a map for you.

Part 1.2. Add Data to your Map

To add data to the map, follow the process above and repeat it. Let's make a map using our sample rodent incident dataset.

Rodents! Using 311 Data

For a sample dataset, we will use **boston_rodents.json**. This is a JSON dataset of rodent incidents based on 311 data from the Boston Open Data portal.

To add this to our map, it is a two step process. It follows the exact same principles of the previous step, except we do not have to set up our projection information or create the map.

1. Link the Data to your Document

In your HTML, we add rodent data url to the script under the bosNeighborurl. We give the rodent url a variable name **rodenturl**

Here is the url:

```
https://gist.githubusercontent.com/jdev42092/ee94f6d469d7084e8dca4e8533817e0e/raw/7cfd6c34c974d3a86ff19c6180cfa22ee9ce3946/boston_rodents.json
```

```
//2. Load the data url:
var bosNeighborurl='https://gist.githubusercontent.com/jdev42092/5c285c4a3608eb9f9864f5da27db4e49/raw/a1c33b143

// Load the rodent data url:
var rodenturl="https://gist.githubusercontent.com/jdev42092/ee94f6d469d7084e8dca4e8533817e0e/raw/7cfd6c34c974d3
```

var rodenturl="https://gist.githubusercontent.com/jdev42092/ee94f6d469d7084e8dca4e8533817e0e/raw/7cfd6c34c974d3a86ff19c6180cfa22ee9ce3946/boston_rodents.json"

2. Add the Data

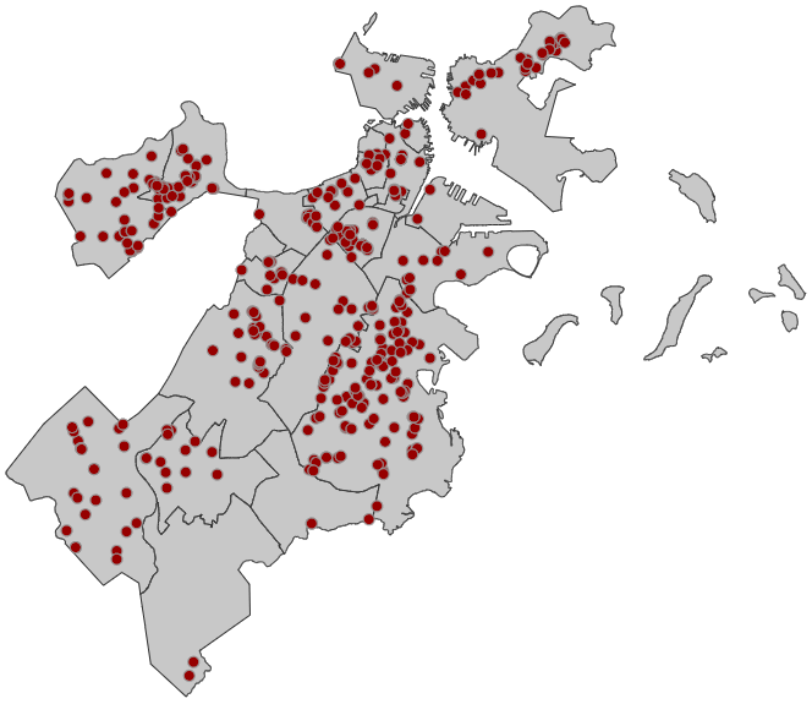
Using the same method as before, add the dataset. In your D3 script, place the following at the bottom, beneath where we load the neighborhoods.

Here we use console.log(rodent) to make sure we load the correct data. You will only see it in the console.

```
// 7. add rodent data
d3.json(rodenturl).then(function(rodent){
    //
    console.log(rodent)
    var rodents = svg.append("g")
    rodents.selectAll("path")
        .data(rodent.features)
        .enter()
        .append("path")
        .attr("fill","#900")
        .attr("stroke","#999")
        .attr("d", geoPath)

})
```

Here is how the map should look like



3. Changing the Point Symbol

Now, we are using a default point symbol that we gave a fill and stroke to so we can see it. We could specify a radius if we wanted, or we could change this and create custom symbols.

To change the color of the point, there are two ways of doing this. You can change the properties of the page element using D3 javascript, or alternately, you can write CSS and create a class. To complete the latter method, we need to do two simple things to the code. First, add some CSS styling to the head of your document. In the head section, enter the following. This will create an element class called "incident".

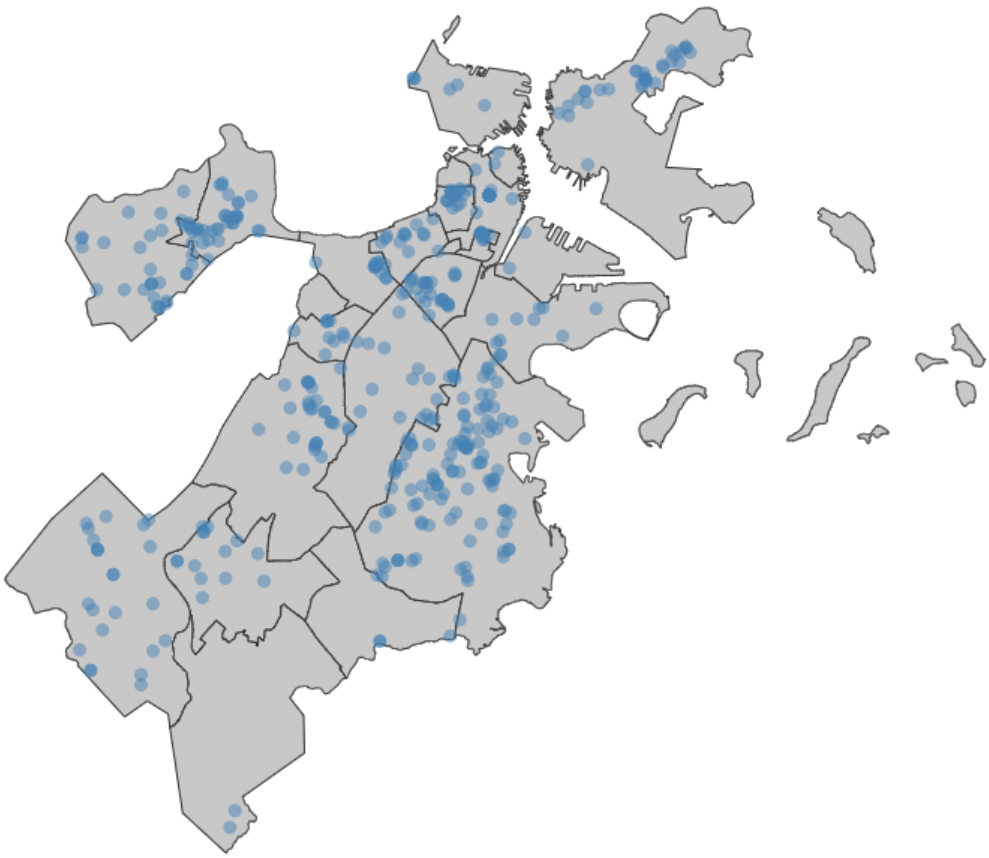
```
<!-- add a style here -->
<style>
    .incident {
        fill: steelblue;
    }
</style>
```

Next, we need to apply the class 'incident' to our data. To complete this, locate the data loading method from the step above, and add a single line of code. Make sure you put it before the end semi-color. Enter the following.

```
// 7. add rodent data
d3.json(rodenturl).then(function(rodent){
    //
    console.log(rodent)
    var rodents = svg.append("g")
    rodents.selectAll("path")
        .data(rodent.features)
        .enter()
        .append("path")
        .attr("fill","#900")
        .attr("stroke","#999")
        .attr("d", geoPath)
        //apply style here
        .attr( "class", "incident");

})
```

Save and refresh your map. You'll see the CSS properties applied to data elements.



Part 1.3. Adding Map Interaction

Once you understand how you load data into D3, adding interaction to your map is actually quite easy. Simply put, you use D3 to change the properties of page elements and incorporate event listeners, such as mouseovers and clicks. To illustrate this, let's set up our map of rodent incidents to display some information, like an address, when you interact with the map. For the example, let's set it up to show a property of the data when you hover over an incident.

In addition to our map title, we can add an element that will hold some information that can change when we hover over an incident. We will style this with the CSS we added in the last step, adding a line of code or two. Stay in our current file for these additions.

1. Add Page Elements

To start, we need page elements that can be populated to hold a title and some popup information on our map. The concept is that we want to create a page element that we can update the language

in according to a data value. Put these in the **body** section, above the script tags (not in between them!).

```
<body>
    <!-- page elemets and content go here. -->
    <h1>Rodent Incidents in Boston</h1>
    <h2></h2>

    <script>
```

2. Add Event Listeners and Change Properties of Page Attributes by adding Classes

To start, we need page elements that can be populated to hold a title and some popup information on our map. The concept is that we want to create a page element that we can update the language in according to a data value. To do this, we can append the following lines of code to the end of the rodents.selectAll("path") method. **Make sure you remove the semicolon and only have one at the end of the chained selectAll("path") method.**

```
// 7. add rodent data
d3.json(rodenturl).then(function(rodent){
    //
    console.log(rodent)
    var rodents = svg.append("g")
    rodents.selectAll("path")
        .data(rodent.features)
        .enter()
        .append("path")
        .attr("fill","#900")
        .attr("stroke","#999")
        .attr("d", geoPath)
        //apply style here
        .attr( "class", "incident")
        .on("mouseover", function(d){
            d3.select("h2").text(d.properties.LOCATION_STREET_NAME);
            d3.select(this).attr("class","incident hover");
        })
        .on("mouseout", function(d){
            d3.select("h2").text("");
            d3.select(this).attr("class","incident");
        });
```

Note the mouseover and mouseout event listeners, and the changing of the classes.

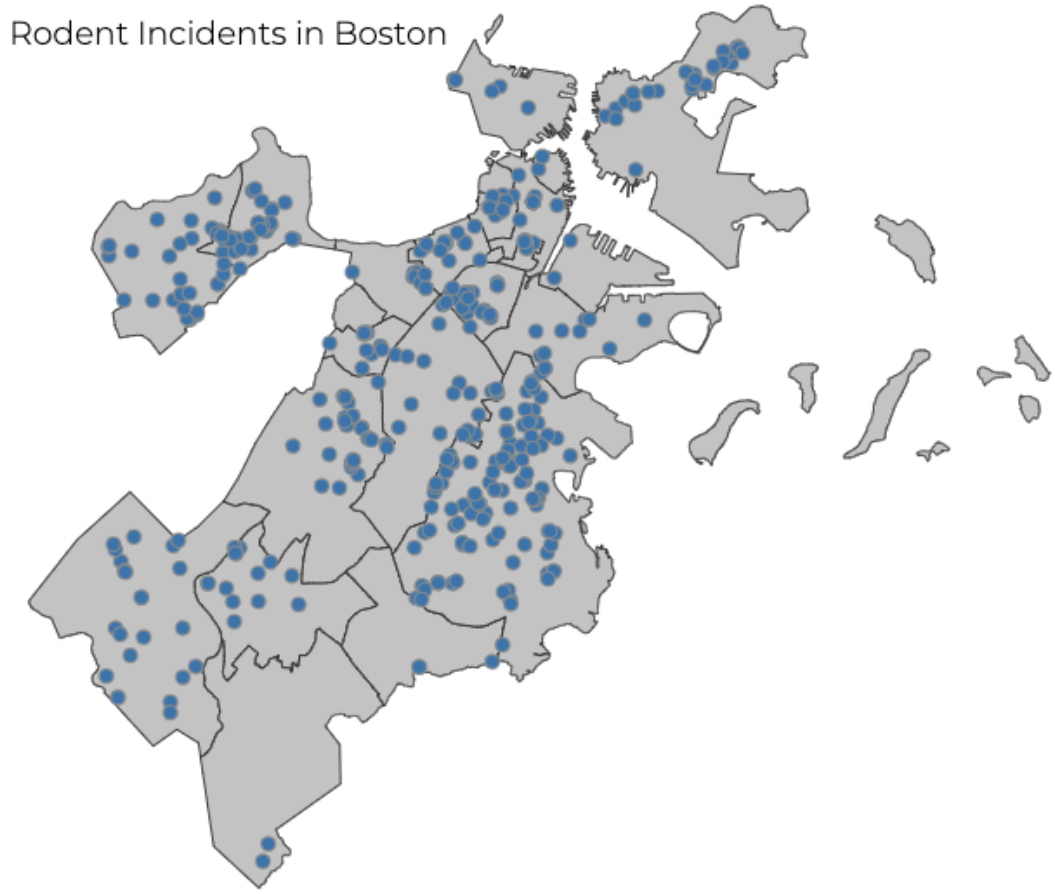
About chaining - method chaining is basically placing methods one after another in d3 code (this is similar to chaining in jQuery). You use periods (.) to chain methods together. For example, in the above code, the .data(), .enter(), .append() and other methods are chained together. Chaining is very important when writing D3 code.

3. Update the Cascading Style Sheets

Lastly, update the style sheets to change colors and set fonts and other style elements. Replace the entire block in the style tags with the following. This will simply position some of the elements on our page, and provide some simple styling.

```
        body {
            position: absolute;
            font-family: "Proxima Nova", "Montserrat", sans-serif;
        }
        h1, h2 {
            position: absolute;
            left: 10px;
            font-size: 1.3em;
            font-weight: 100;
        }
        h2 {
            top: 30px;
            font-size: 1em;
        }
        .incident {
            fill: steelblue;
        }
        .hover {
            fill: yellow;
        }
```

Our map, with this new code added, and some nice new interactivity. It should look like the following. See the page source for the full example.



Using Update and Add a Time Swiper

Let's continue to animate our map by adding some additional user interaction, showing Boston rodent incidents by month over the years the data was collected. Our map will look like the following.

To get a slider on our map, we can use a couple of different HTML features. The steps we take will be the following.

1. Add Time Slider HTML Component
2. Position the Component
3. Create global variables for Time Slider input and Months of the Year
4. Write Update Function that gets Value from Slider and sets Attribute
5. Write function (dateMatch) that returns a Color
6. Set Initial State of Map

1. Add Time Slider HTML Component

To start, let's add a time slider component to our map. HTML5 has a nice built in range slider we can use. This slider has parameters that can be set that declare the range and steps the slider takes within that range. In the body of your page, above your script tags but below your headers, add the range input. Put it in a new **div** and call it *sliderContainer*. Give the range input an id of **timeslide** and the span element an id of **range**. You have learned a lot already about the DOM. Where do you think this div goes?

```
<div id="sliderContainer">

    <input id="timeslide" type="range" min="0" max="11" value="0" step="1"/><br>

    <span id="range">January</span>

 </div>
```

2. Position the Component

Let's adjust the CSS a bit now to position the component. Also, let's remove the color we set for the **incidents** class. We will set the color based on the input from the slider, not in the CSS here. Because we named our **div** *sliderContainer*, we can style it in the CSS.

IMPORTANT: Remove the CSS for the Incidents class. Otherwise it will override things

```
<style>
    body {
        position: absolute;
        font-family: "Proxima Nova", "Montserrat", sans-serif;
    }
    h1, h2 {
        position: absolute;
        left: 10px;
        font-size: 1.3em;
        font-weight: 100;
    }
    h2 {
        top: 30px;
        font-size: 1em;
    }
    /* REMOVE。incident FILL CSS */
    /* .incident {
        fill: steelblue;
    } */
    .hover {
        fill: yellow;
    }

    /* ADD CSS FOR #sliderContainer */
    #sliderContainer{
        text-align: center;
        position: relative;
        top:600px
    }
</style>
```

Our slider will now be at the bottom of our map, top at 600px from the top relative to the container.

3. Create Global Variables

We now want two global variables, one that will hold the input value received from the time slider, we can call it inputValue and one that holds an array (months) that represents what the input values mean. For example, since position 0 in our slider represents January, we can write an array that returns "January" when we call **months[0]**, with 0 being the inputValue.

Here are the variables, **put this in your script at the top, outside of any functions**.

```
// Global Variables

var inputValue = null;

var month = ["January","February","March","April","May","June",
"July","August","September","October","November","December"];
```

These variables will handle our inputs and help us with the slider values.

4. Write an Update function and timeslide event listener

Next, we need to write two pieces of code, one that listens for when the value of the time slider changes, and one that updates the SVG elements. We are going to use some D3 code to listen for an input change on the #timeslide element, and then pass the value to a function named update that will use a selectAll on incidents to update the fill. When we change the slider position, it will change the month. We'll also set the range element to show the month, so the user can see it.

Put this at the end of the script, making sure you are within the script tags.

```
// when the input range changes update the value
d3.select("#timeslide").on("input", function() {
    update(+this.value);
});
// update the fill of each SVG of class "incident" with value
function update(value) {
    document.getElementById("range").innerHTML=month[value];
    inputValue = month[value];
    d3.selectAll(".incident")
        .attr("fill", dateMatch);
}
```

You'll notice we set fill to be 'dateMatch'. This is a function that will check the inputValue for a match in the data, then return a color if there is a match. Let's write that up in the next step

5. Write Function to Return a Color

Our color return function will look like the following. It takes our data and a value as arguments, grabs the open date of the incident from the dataset and sets it to a JavaScript data object and gets the month, then checks the inputValue against the month. If there is a match, it returns **red**, it not, **grey (#999)**. We also use **this.parentElement.appendChild(this)** to help with layering. The way D3 draws is in order of drawing, so this appends the current element to the parent, making it draw last. Check it out, add this to your script.

```
// function to match date
function dateMatch(data, value) {
```

```
            var d = new Date(data.properties.OPEN_DT);

            var m = month[d.getMonth()];

            if (inputValue == m) {

                this.parentElement.appendChild(this);

                return "red";

            } else {

                return "#999";

            };

        }
```

Note in the function, that all this does is return a color we can set to our D3 attribute fill.

6. Set the Initial State

Lastly, when the map loads, we want our initial map state to be set to January. To accomplish this, write up an **initialDate** function that returns a color based on month match, and then sets the fill in the path creation to **initialDate**. When the user visits, they will see incidents filed in January on the map. Put this at the bottom of your script.

```
        //Set Initial State

        function initialDate(d,i){

            var d = new Date(d.properties.OPEN_DT);

            var m = month[d.getMonth()];

            if (m == "January") {

                this.parentElement.appendChild(this);

                return "red";

            } else {

                return "#999";

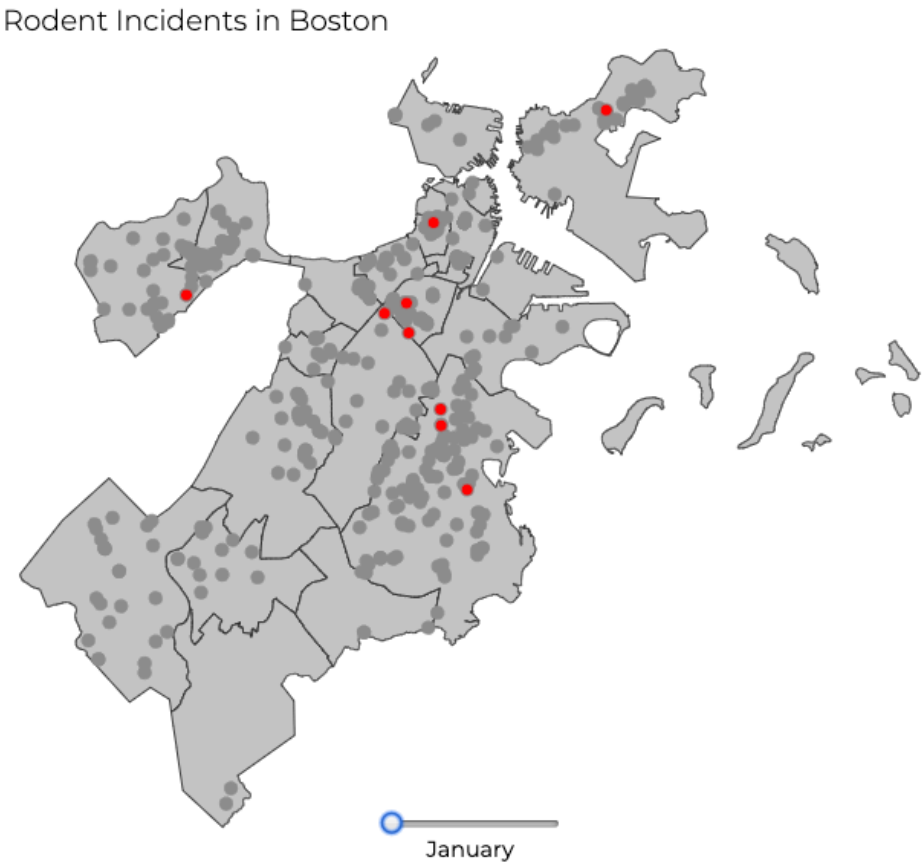
            };

        }
```

Once this is in your script, call it in your **path creation method**. Set the fill to be the return of the initialDate function

```
// 7. add rodent data
d3.json(rodenturl).then(function(rodent){
    //
    console.log(rodent)
    var rodents = svg.append("g")
    rodents.selectAll("path")
        .data(rodent.features)
        .enter()
        .append("path")
        .attr("fill",initialDate)//replace original value with initialDate here
        .attr("stroke","#999")
        .attr("d", geoPath)
        //apply style here
        .attr( "class", "incident")
        //add mouseover effects here:
        .on("mouseover", function(d){
            d3.select("h2").text(d.properties.LOCATION_STREET_NAME);
            d3.select(this).attr("class","incident hover");
        })
        .on("mouseout", function(d){
            d3.select("h2").text("");
            d3.select(this).attr("class","incident");
        });

})
```



From here, perhaps we can add a coordinated bar chart or scatterplot, that updates using the same triggers, or continue to expand up functionality of the map by adding more filtering options.

Add More Supplemental Information to the Map

At this point, our map is nice, but we could make it more informative. We don't have a date, legend, or any supplemental information on here. CHALLENGE, can you add a date to the changing popup text?

Part 2: Using TopoJSON and 'Joining Data'

Introducing TopoJSON

D3 works with two types of geographic JSON, GeoJSON, and a format called TopoJSON.

GeoJSON vs. TopoJSON

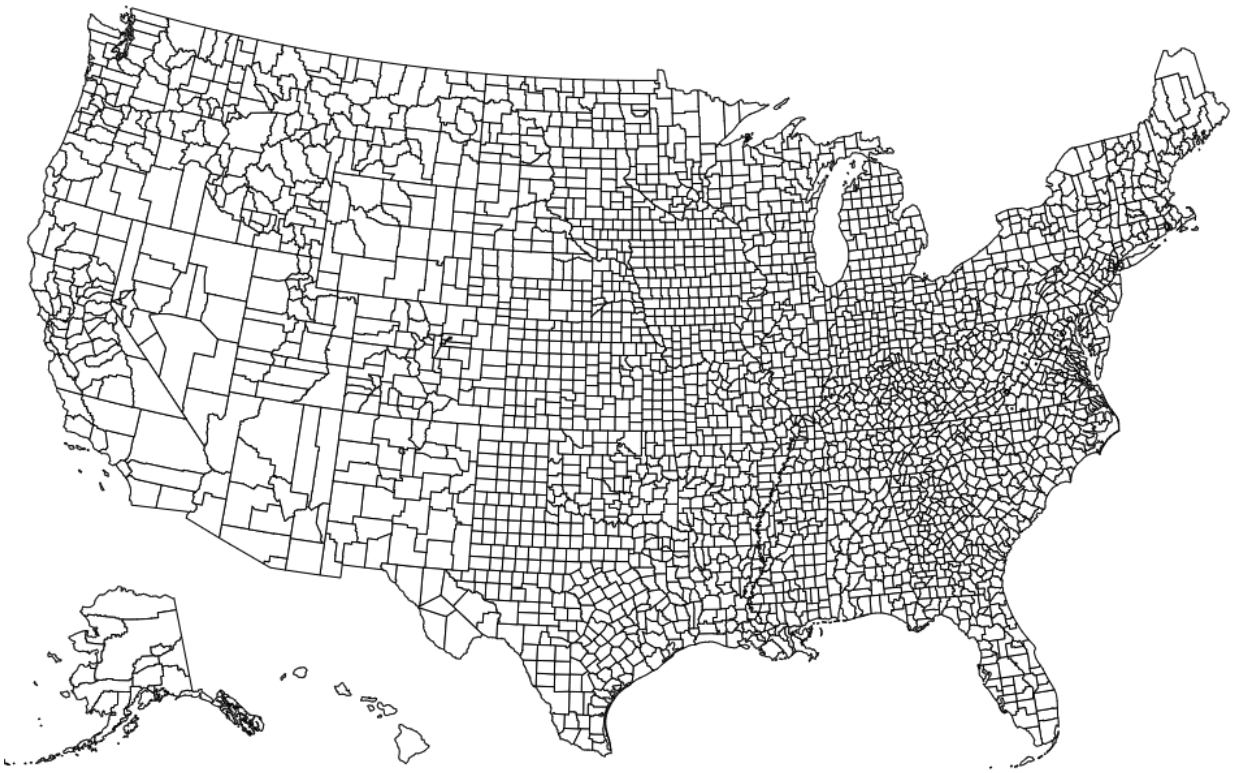
TopoJSON is an extension of GeoJSON that encodes topology. Rather than representing geometries discretely, geometries in TopoJSON files are stitched together from shared line segments called arcs. This technique is similar to Matt Bloch's MapShaper and the Arc/Info Export format, .e00. TopoJSON eliminates redundancy, allowing related geometries to be stored efficiently in the same file. For example, the shared boundary between California and Nevada is represented only once, rather than being duplicated for both states. A single TopoJSON file can contain multiple feature collections without duplication, such as states and counties. (TopoJSON wiki)

As a result, TopoJSON is substantially more compact than GeoJSON. The above shapefile of U.S. counties is 2.2M as a GeoJSON file, but only 436K as a boundary mesh, a reduction of 80.4% even without simplification.[3]

In order to use TopoJSONs, you have to add an extra TopoJSON library to your document that contains the necessary components and methods. In the head, after you load D3, load the TopoJSON library using the following.

```
<script src="https://d3js.org/topojson.v2.min.js"></script>
```

Structure of a TopoJSON



Sample TopoJSON Code

For example TopoJSON code, check out the following TopoJSON documentation.

TopoJSON Specification Documentation

Make a Choropleth Map of U.S. Counties

The following will detail, from the ground up, using TopoJSON and building a choropleth map with 'joined data' from the bottom up.

Let's look at the code.

In your folder, find the Week2_D3_Map_part2*.html*

In your blank document, input the following.

1. HTML Template

A simple HTML template for our page.

```
<!DOCTYPE html>

<html lang="en">

<head>

   <title>Mapping with D3</title>

   <script src="https://d3js.org/d3.v5.min.js" charset="utf-8"></script>

</head>

<body>

   <!-- Page elements and content go here. -->

   <script>

       // Our D3 code will go here.

   </script>

</body>

</html>
```

2. Add Additional D3 Modules for Mapping

TopoJSON is an extra addon of D3, so we need to add it to our code. We are also going to use something called d3-queue. This will load datasets one by one, and then run a function when they are all loaded. This easily allows us to work with multiple datasets from different sources in the same visualization.

Add them using the following, in the head part of your HTML.

```
<!DOCTYPE html>

<html lang="en">

<head>

   <title>Mapping with D3</title>

   <script src="https://d3js.org/d3.v5.min.js" charset="utf-8"></script>

   <script src="https://d3js.org/d3-scale-chromatic.v1.min.js"></script>

   <script src="https://d3js.org/topojson.v2.min.js"></script>

</head>

<body>
```

```
    <!-- Page elements and content go here. -->
    <script>
        // Our D3 code will go here.
    </script>
</body>
</html>
```

3. Set up the Map

Use D3 to set up your map. We did this above, so you have an understanding for what is happening now. Enter the following code into your document in the script section of the code.

```
        var width = 1200,
            height = 900;

        var path = d3.geoPath()
        var svg = d3.select("body").append("svg")
                    .attr("width", width)
                    .attr("height", height);
```

4. Load county data using d3.json

Set up our uscountyurl here:

```
var uscountyurl='https://unpkg.com/us-atlas@1/us/10m.json'

Var
unemploymenturl="https://gist.githubusercontent.com/jdev42092/9a314291805640a097e16e58
248627eb/raw/365c199c5a173a0018608615b6823b5cdcaa6bae/unemployment.tsv"
```

Once the resources are loaded, we can call a function that runs when ready using await. This is because we need to wait until the data is loaded. Set the **ready** function to read our data as arguments. We can do a lot with them, including joining them based on like attributes.

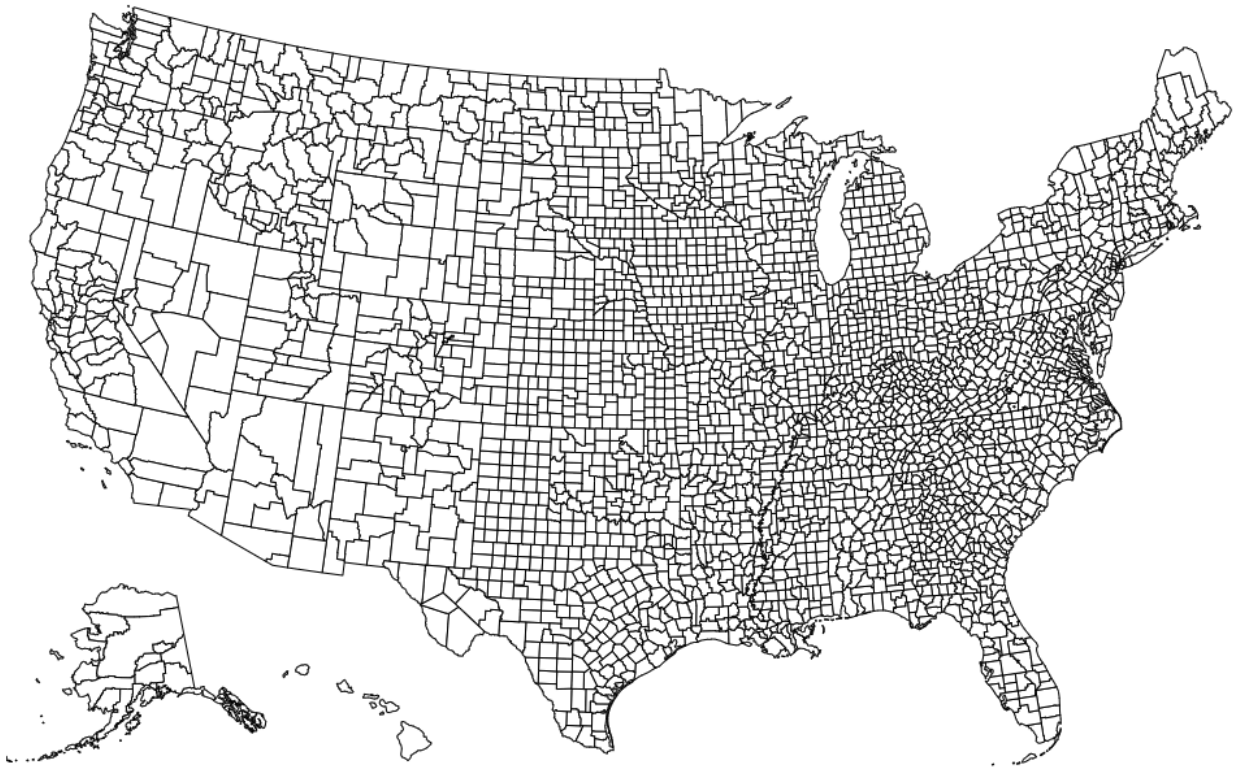
Here is how to set up ready and use d3. json to call ready:

```
        d3.json(uscountyurl).then(ready)
```

```
function ready(us) {
    svg.append("g")
        .selectAll("path")
        .data(topojson.feature(us, us.objects.counties).features)
        .enter()
        .append("path")
        .style("fill","white")
        .style("stroke","black")
        .attr("d", path)
        .append("title");
}
```

In this function, we pass our data (the TopoJSON) as an argument, then create SVG elements using a classic D3 append. Selecting all paths, the TopoJSON is bound in the data method. From here, we can perform work on each element. In this circumstance, we apply a fill and stroke.

You should see your map loaded:



Joining Data: Create a Choropleth

Continuing with this example, let's join a TSV of unemployment data to our counties. Joining in D3 is a bit different than traditional GIS, but will have some similarities. We need to add our dataset to the queue,

1. Using Promise to queue two Datasets

In **d3.v5,** we use **Promise** method to queue datasets. This method allows scripts to wait until all outside resources are loaded into your visualization. Waiting for everything to load prevents errors from occuring. We can use d3 maps to group the data, then we will set the "fill" returning our "grouped" data when we create SVG elements, calling the property we want to join on.

Remove the previous d3.json(uscountyurl) as we are using Promise to load both datasets.

```
var unemployment = d3.map();

var promises = [

                d3.json(uscountyurl),

                d3.tsv(unemploymenturl)
```

```
                        ]


        //d3.json(uscountyurl).then(ready) REMOVE THIS LINE
        Promise.all(promises).then(ready)
```

This passes our unemployment dataset to the ready function, where we can parse it and send values to the polygons using the fill property.

2. Create Object for the Tabular Dataset

We need to create an object for our tabular dataset, then, in this object, create properties for each county ID. We want to give the value of each property the corresponding rate. The D3 TSV object loads our data as an array. We revise the previous code into:

```
        var promises = [
                        d3.json(uscountyurl),
                        d3.tsv(unemploymenturl, function(d) { unemployment.set(d.id,
+d.rate); })
                                                                        ]



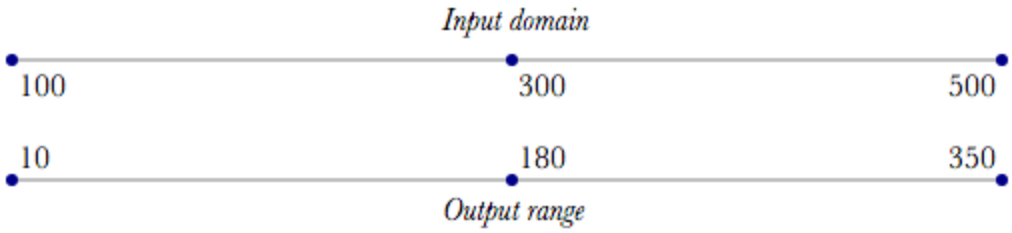
        //d3.json(uscountyurl).then(ready) REMOVE THIS LINE
        Promise.all(promises).then(ready)
```

We now have a JavaScript object with each county as a property, and the rate as the value of the property.

3. Classify by creating a Color function using d3.scale

Next we classify our unemployment data into categories. D3 has a couple of different methods for doing this that are written into the d3-scale module. These include continuous, quantize, quantile, ordinal, and threshold methods. The threshold method allows us to set our own class breaks, so we'll use that. (NOTE: You'll probably want to look at a histogram of your data to check out your breaks.) There are also some classic geographic methods, such as Jenks Natural Breaks, that can be implemented using a library called Simple Statistics. An example for this will be found at the end of this exercise.

To use scale in D3, we create a function expression using the d3.scaleThreshold() method. Separate from your **queue** and **ready** functions, implement your scale. This takes the domain and range principles learned in the first session, and applies them using color. As a reminder:



Scales: Domain and Range Scott Murray

The code for our domain and range using the threshold method looks like the following. The class breaks are taken from a look at a histogram, and using cartographic principles of logical round numbers that will make your map easier to read for the viewer. Here is the code.

```
        var color = d3.scaleThreshold()

            .domain(d3.range(2, 10))

            .range(d3.schemeBlues[9]);
```

4. Use the Color Function to set the Fill Value for each Polygon

Since we are using two datasets which are returned in a list, change the *ready* function's argument from **ready(us)** to **ready([us])**. Adjust the fill property of the append statement to use the color function. The color function takes the value found in the unemployment array for the property that matches the ID value of the counties dataset (**d.id**). To illustrate this, throw in a console.log on **d**, or **d.id**. The return value is a number that is then run through our threshold scale, return a hex code for color based on what value is received. Here is the code, note the fill property.

```
        function ready([us]) {

            svg.append("g")

                .selectAll("path")

                .data(topojson.feature(us, us.objects.counties).features)

                .enter().append("path")

                .attr("fill", function(d) { return color(d.rate =
unemployment.get(d.id)); })

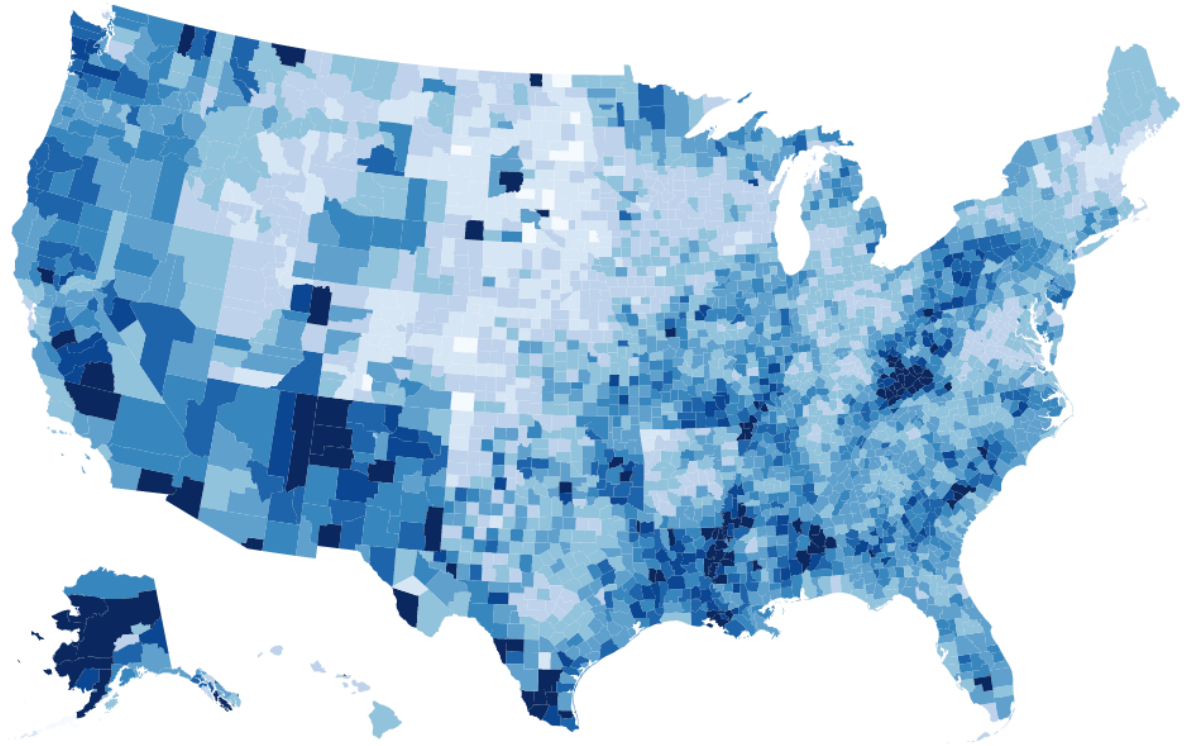
                // .style("fill","white") REMOVE THIS LINE

                // .style("stroke","black") REMOVE THIS LINE
```

```
            .attr("d", path)

            .append("title");

    }
```



5. Add Legend (bonus exercise, you will need this for the pset)

Add the following code to your scripts before calling the **Promise**.

Make sure the **g.selectAll()** is after you declare variable **g.**

This is simply making a couple rectangles.

```
    var x = d3.scaleLinear()

        .domain([1, 10])

        .rangeRound([600, 860]);

    var g = svg.append("g")

        .attr("class", "key")

        .attr("transform", "translate(0,40)");
```

```
g.selectAll("rect")
    .data(color.range().map(function(d) {
        d = color.invertExtent(d);
        if (d[0] == null) d[0] = x.domain()[0];
        if (d[1] == null) d[1] = x.domain()[1];
        return d;
    }))
    .enter().append("rect")
    .attr("height", 8)
    .attr("x", function(d) { return x(d[0]); })
    .attr("width", function(d) { return x(d[1]) - x(d[0]); })
    .attr("fill", function(d) { return color(d[0]); });

g.append("text")
    .attr("class", "caption")
    .attr("x", x.range()[0])
    .attr("y", -6)
    .attr("fill", "#000")
    .attr("text-anchor", "start")
    .attr("font-weight", "bold")
    .text("Unemployment rate");

g.call(d3.axisBottom(x)
.tickSize(13)
.tickFormat(function(x, i) { return i ? x : x + "%"; })
.tickValues(color.domain()))
.select(".domain")
.remove();
```

Your final map should look like:

