

Time Series Proportional Symbol Maps with Leaflet and jQuery

Richard G. Donohue
University of Kentucky
rgdonohue@uky.edu

Carl M. Sack
University of Wisconsin–Madison
cmsack@wisc.edu

Robert E. Roth
University of Wisconsin–Madison
reroth@wisc.edu

ASSUMED SKILLS AND LEARNING OUTCOMES

THE FOLLOWING TUTORIAL describes how to make a time series proportional symbol map using the Leaflet (leafletjs.com) and jQuery (jquery.com) code libraries. The tutorial is based on a laboratory assignment created in Spring of 2013 for an advanced class on Interactive Cartography and Geovisualization at the University of Wisconsin–Madison (www.geography.wisc.edu/courses/geog575). This is the first of two *On the Horizon* tutorials on the topic of web mapping, with the next tutorial covering multivariate choropleth mapping using the D3 library. Commented source code for the tutorial is available through a Creative Commons license at geography.wisc.edu/cartography/tutorials.

The tutorial assumes a basic understanding of the open web platform, particularly the HTML, CSS, and JavaScript standards. It also assumes that you are familiar with the manipulation of JavaScript objects as well as jQuery-style DOM element selection. Tutorials and reference documentation for HTML, CSS, and JavaScript are available at

such resources as developer.mozilla.org, www.lynda.com, www.codecademy.com, and www.w3schools.com. Further, it is assumed that you are familiar with in-browser development tools such as Chrome Developer Tools (developers.google.com/chrome-developer-tools) or Firebug (getfirebug.com). Finally, the tutorial assumes that you have access to a web server, either running remotely or as a local host; MAMP for Mac (www.mamp.info/en) and WAMP for Windows (www.wampserver.com/en) are useful for this.

After completing the tutorial, you will be able to:

- Work with the GeoJSON data format
- Use the Leaflet library to publish a time series proportional symbol map to the web
- Create interactivity using mouseover popup windows and range sliders

GETTING STARTED WITH LEAFLET

LEAFLET IS ONE OF MANY code libraries now available for publishing slippy maps to the web. Leaflet is a JavaScript library pioneered and maintained by Vladimir Agafonkin (agafonkin.com/en), and quickly is growing in popularity within the web development community because it is both lightweight (only 33kb of code at the time of this writing) and open source (meaning you can both view how it functions and extend it to fit your needs). Maps produced using Leaflet can load a variety of basemap tile services and can draw vector features atop these tiles using the SVG (Scalable Vector Graphics) standard. Leaflet is also

bundled with Mapbox.js (www.mapbox.com/mapbox.js), allowing for simple loading and manipulation of custom tilesets. Because of the small file size and support of touch-based interactions, Leaflet is considered among the best web mapping libraries when designing for mobile devices.

The Leaflet library is an open-source project on GitHub (github.com/Leaflet) and can be extended through numerous open plugins (leafletjs.com/plugins.html). The goal of this tutorial is to provide you with a broad introduction to using Leaflet for Web Cartography. The following tutorial



© by the author(s). This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

extends the reference and tutorials available at leafletjs.com/examples.html. There also are supportive communities for the Leaflet library on Twitter (<https://twitter.com/search?q=%23leaflet>) and Stack Overflow (stackoverflow.com/search?q=leaflet). Refer to these materials for additional background and guidance as you complete the tutorial.

com/search?q=leaflet). Refer to these materials for additional background and guidance as you complete the tutorial.

1. FINDING AND FORMATTING TIME SERIES INFORMATION

THE FIRST STEP is the assembly of appropriate *time series* information (i.e., geographic information that changes over time) to portray on your proportional symbol map. Because proportional symbol maps leverage the visual variable size, you only should map ordinal, or, preferably, numerical data using this thematic map type (i.e., do not collect categorical information).

Use your preferred spreadsheet (e.g., as a *.csv* file) or GIS (e.g., as a *.shp* file) software to prepare your dataset. Format the dataset with the unique map features (e.g., cities, regions) included as rows and the unique timestamps (generically describing either a single moment in time or a time interval) included as columns. Be sure to use logical header names (such as 2005, 2006, etc.), as these terms serve as attribute keys for referencing the time series information using JavaScript and will be used to create a temporal legend in the map itself. Because Leaflet natively understands the geographic coordinate system, you need to include a pair of columns for the latitude and longitude of the proportional symbol anchor (e.g., the city center, the centroid of the region). For this tutorial save the latitude value as *lat* the longitude value as *lon*. Finally, include an additional pair of columns at the start of your file for a unique *id* number and name field. **Figure 1** provides an example time series dataset for fifteen major cities in the United States.

Next, convert your dataset into the GeoJSON format. *JSON* stands for JavaScript Object Notation and has become a standard format for information loaded into and interpreted by a browser. *GeoJSON* is one geographic variant of JSON that structures each map feature as an array

	A	B	C	D	E	F	G	H	I	J	K
1	id	name	latitude	longitude	2005	2006	2007	2008	2009	2010	2011
2	1	Atlanta	33.7489	-84.3881	85	38	75	30	9	15	38
3	2	Chicago	41.85	-87.65	28	29	38	26	15	12	10
4	3	Dallas	32.7828	-96.8039	18	59	22	60	82	42	18
5	4	Denver	39.7392	-104.9842	35	45	31	26	14	9	15
6	5	Houston	29.7631	-95.3631	12	31	15	22	28	38	31
7	6	Kansas City	39.0997	-94.5783	25	50	25	25	25	25	100
8	7	Los Angeles	34.0522	-118.2428	88	46	56	15	12	25	46
9	8	Miami	25.7738	-80.1924	52	51	46	68	75	85	96
10	9	Minneapolis	44.98	-93.2636	7	12	18	11	9	9	4
11	10	New York	40.7142	-74.0064	23	18	16	24	26	28	30
12	11	Philadelphia	39.9522	-75.1642	32	28	29	25	22	15	8
13	12	Phoenix	33.4539	-112.0746	8	15	22	25	29	28	32
14	13	San Francisco	37.775	-122.4183	82	74	72	10	85	88	74
15	14	Seattle	47.6097	-122.3331	9	16	14	23	45	66	85
16	15	Washington	38.89	-77.03	33	45	68	96	102	82	74
17											

Figure 1: An Example Time Series Dataset.

of *nodes* (lat/long coordinate pairs) defining the complete outer boundary of the polygon.

There are multiple ways to convert your dataset to the GeoJSON format, including:

- GIS applications such as ArcGIS (www.esri.com/software/arcgis) or QGIS (www.qgis.org);
- Open utilities such as GDAL/OGR (www.gdal.org/ogr2ogr.html);
- Free web services, such as MapShaper (www.mapshaper.org), ShpEscape (www.shpescape.com), ToGeoJSON (togejson.com), and GeoJSON.io.

Leaflet also includes several methods to load formats other than GeoJSON, although these are not discussed in this tutorial. An example GeoJSON file (extension *.json*) for the **Figure 1** time series dataset is included in the online code repository.

2. PREPARING YOUR DIRECTORY STRUCTURE AND BOILERPLATE

WITH YOUR TIME SERIES information processed, it is now time to start building your map! Create a directory that includes folders named “data,” “css,” “img,” and “js.” Because you will be using AJAX requests, we strongly

recommend that you set up a development server on your machine and place this directory on the server, accessing it as a localhost. Using your preferred text editor, create three new files named *index.html* (root level), *style.css* (css folder),

```

1<!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>Leaflet Prop Symbol Map</title>
6
7      <!--[if IE]>
8          <script src="html5shiv.googlecode.com/svn/trunk/html5.js">
9          </script>
10     <![endif]-->
11
12     <!--stylesheet-->
13     <link rel="stylesheet" href="css/leaflet.css">
14     <link rel="stylesheet" href="css/style.css">
15 </head>
16 <body>
17     <!--scripts-->
18     <script src="js/jquery.js"></script>
19     <script src="js/leaflet.js"></script>
20     <script src="js/main.js"></script>
21 </body>
22 </html>

```

Example 1: Basic HTML5 Boilerplate, with References to Styles/Scripts (in: index.html).

and *main.js* (*js* folder). Copy your newly created GeoJSON file into the *data* folder.

Next, add the boilerplate text provided in **Example 1** into the *index.html* file. The boilerplate comprises the minimum markup of a valid HTML 5 document, with one conditional tag to handle older versions of Internet Explorer (**EX1: 7–10**¹). The boilerplate also includes references to the stylesheets (**EX1: 12–14**) and scripts (**EX1: 17–20**) that you will be using in your time series proportional symbol map. Before moving on, change the content of the `<title>` element (**EX1: 5**) to something logical for your map.

After configuring your directory, acquire the most recent, stable version of the Leaflet source code from leafletjs.com/download.html. Uncompress the downloaded *.zip* file and place the *leaflet.css* file into the *css* folder, and the Leaflet *images* folder, *leaflet.js*, and *leaflet-src.js* files into the *js* folder. While the html boilerplate links to minified *leaflet.js* file (**EX1: 19**), it is recommended that you reference

the un-minified, human-readable *leaflet-src.js* file when interpreting Leaflet functionality.

In addition to the Leaflet source code, you also need to acquire the source code for the *jQuery* library. jQuery (jquery.com) is a JavaScript plug-in that simplifies accessing and manipulating DOM elements for both representation and interaction. Additionally, jQuery handles many of the browser compatibility issues that otherwise require specific JavaScript solutions. If you are unfamiliar with jQuery, review the jQuery tutorial available at learn.jquery.com/about-jquery/how-jquery-works/. Download the jQuery source code from jquery.com/download/, uncompress the downloaded *.zip* file, and place the *jquery.js* file in your *js* folder.

Before moving onto the next step, check to see if your file structure and webpage files are properly configured. The primary method for debugging scripts is by printing a message to the *error console* using the `console.log()` method in JavaScript. To demonstrate its utility, and confirm that your webpage is properly configured, add a script to print to the console in the *main.js* file (**Example 2**).

1. This notation is used in the following tutorial for brevity; for example, "Example 1: Lines 7–10" will be displayed as "EX1: 7–10."

```
1 console.log("hello world!");
```

Example 2: Debugging Scripts with the Console (in: main.js).

Once added, open *index.html* in Firefox; at this point, it should be a blank webpage (**Figure 2**). Activate Firebug by clicking the Firebug icon; you may need to *Enable All Panels* in the Firebug dropdown option, if not already enabled. Once activated, click the console tab and reload the page.

3. LOADING A BASEMAP USING LEAFLET

YOU ARE NOW READY to load basemap tiles into your webpage using Leaflet. Leaflet allows you to load tiles from a variety of sources. An overview of public tile services is available on the UW–Madison GIS Collective blog: giscollective.org/tutorials/web-mapping/wmsthree/. For Leaflet to use a public tile service, you need to reference the URL using the following syntax:

```
{s}.acetate.geoiq.com/tiles/acetate/{z}/{x}/  
{y}.png
```

Every tile in a slippy map is a separate 256 x 256 pixel image—a *.png* file in the above example syntax. The {s} indicates possible server instances from which the map can draw tiles. For each loaded tile, the {z} indicates its zoom level, the {x} indicates its horizontal coordinate, and {y} indicates its vertical coordinate. Near all public tile services use this z/x/y directory format, which was pioneered by Google. The example syntax above loads the minimalist *Acetate* tile service (developer.geoiq.com/tools/acetate) from GeoIQ (now Esri); a minimalist tile design is recommended when adding thematic content atop the basemap tiles.



Figure 2: Debugging using the Error Console in Firebug.

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>Leaflet Prop Symbol Map</title>
6
7      <!--[if IE]>
8          <script src="html5shiv.googlecode.com/svn/trunk/html5.js">
9          </script>
10     <![endif]-->
11
12     <!--stylesheet-->
13     <link rel="stylesheet" href="css/leaflet.css">
14     <link rel="stylesheet" href="css/style.css">
15 </head>
16 <body>
17     <div id="wrapper">
18         <div id="map"></div>
19     </div><!-- end wrapper -->
20
21     <!--scripts-->
22     <script src="js/jquery.js"></script>
23     <script src="js/leaflet.js"></script>
24     <script src="js/main.js"></script>
25 </body>
26 </html>

```

Example 3: Adding a `<div>` Element for the Map (in: `index.html`).

To load a tileset into Leaflet, first add a `<div>` element to the `<body>` of `index.html`, giving it the `id` attribute `map` for referencing by stylesheets and scripts (EX3: 18). While not required, it is good practice to place this `<div>` within a second `<div>` element named `wrapper` containing any additional page elements you add to your design (EX3: 17–19).

Next, edit the `style.css` file to apply style rules for the pair of `<div>` elements, as well as to define the `<body>` element within the `index.html` document (Example 4). For the tutorial example, the wrapper `<div>` is given a width of 960px (EX4: 6), a conventional width in web design for non-mobile devices, and its left and right margin values are set to `auto` in order to center the wrapper `<div>` within the webpage (EX4: 7). It is necessary to set the height attribute of the map `<div>` in order for Leaflet to draw the map within this container. Note that the width of the map `<div>` will automatically fill 100% of its parent container, in this example the wrapper `<div>`. The height is set to

```

1  body {
2  font-family: sans-serif;
3  }
4
5  #wrapper {
6  width: 960px;
7  margin: 15px auto;
8  }
9
10 #map {
11 height: 450px;
12 margin: 15px auto;
13 }

```

Example 4: Styling the `<div>` Element Containing the Map (in: `style.css`).

```

1   $(document).ready(function() {
2
3       var cities;
4       var map = L.map('map', {
5           center: [37.8, -96],
6           zoom: 4,
7           minZoom: 4
8       });
9
10      L.tileLayer(
11          '{s}.acetate.geoiq.com/tiles/acetate/{z}/{x}/{y}.png', {
12              attribution: 'Acetate tileset from GeoIQ'
13          }).addTo(map);
14  });

```

Example 5: Loading a Basemap using Leaflet (in: main.js).

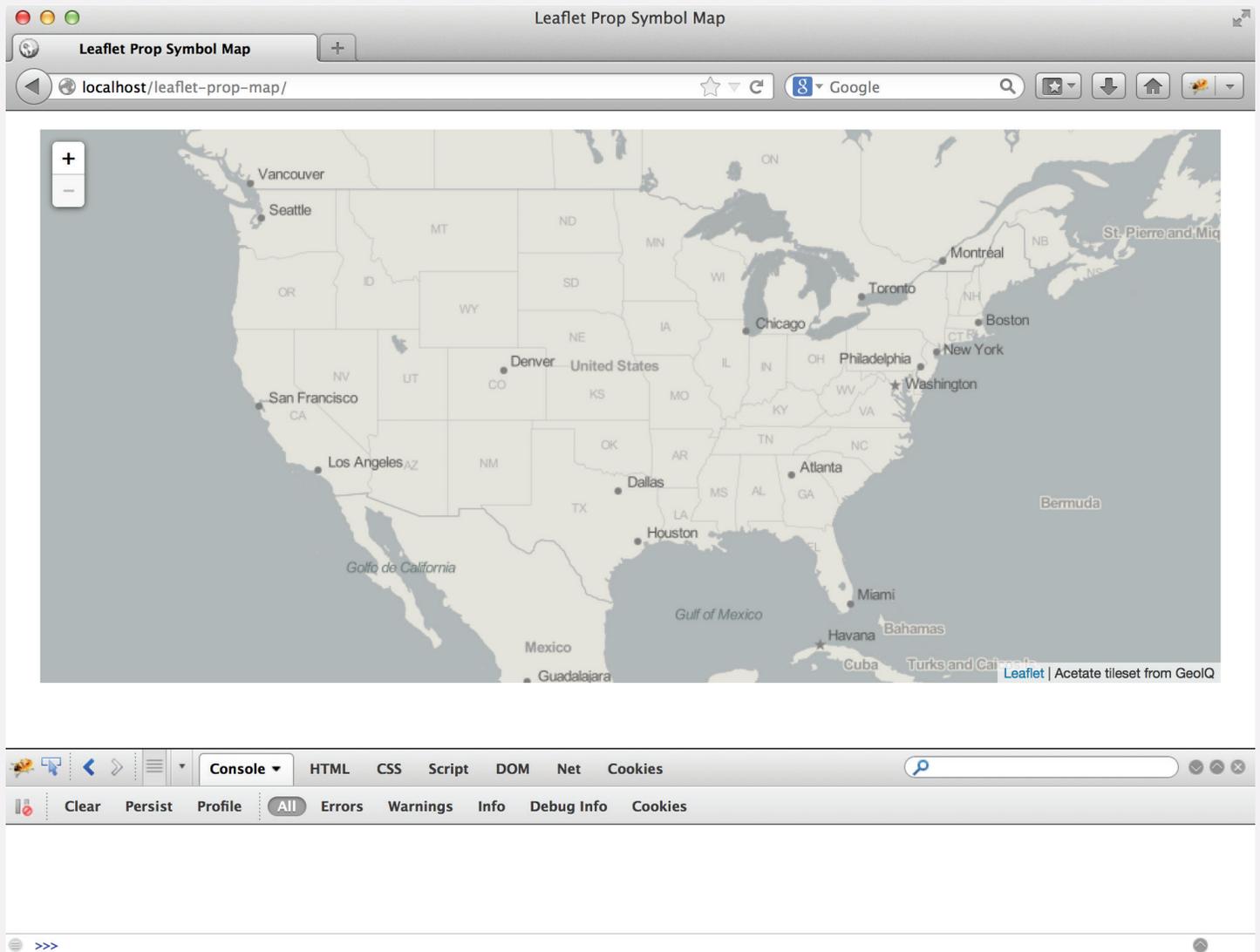


Figure 3: Loading the Acetate tileset.

450px (EX4: 11), and a margin again is used to keep the map away from other page elements as well as to center it within the page (EX4: 12). The values used for these styles are arbitrary, and should be adjusted given the layout of the map within your webpage.

With the page elements in place for the map, you now can add code to the *main.js* file for loading a tileset. The *main.js* file begins with the jQuery method `ready()`, which is used to ensure that the entire page has finished loading before the script begins executing (Example 5). The callback function() within the `ready()` method (EX5: 1) must be closed at the bottom of the *main.js* page (EX5: 14). All subsequent JavaScript code in the tutorial is written within the `ready()` callback function().

Declare two variables within the `ready()` callback function(): (1) `cities` and (2) `map` (EX5: 3–8). The unsigned `cities` variable will reference the proportional

symbols added atop the basemap, as explained later in the tutorial. The map object references the Leaflet map class (`L.Map`) itself, to which all mapped data and controls are added. The map object allows for configuration of basic map parameters, such as the map center (EX5: 5), the zoom scale on loading (EX5: 6), and constraints in zooming interaction (EX5: 7). Review the Leaflet documentation to learn about additional map parameters that can be set using the `L.map` class (leafletjs.com/reference.html#map-usage). The `cities` and `map` variables are declared with a global scope and therefore are accessible within all subsequent method definitions. After the map object is declared and defined, a `tileLayer` of your choosing can be added to the map (EX5: 10–13). Example 5 makes use of the aforementioned Acetate tileset.

Save your changes to the *main.js* file and refresh the *index.html* page in the browser. The map `<div>` element now should be populated with the Acetate tileset, including basic slippery map interactivity (Figure 3).

4. LOADING THE GEOJSON

ONCE YOU HAVE SUCCESSFULLY loaded a tileset into your map `<div>`, the next step is to load the time series dataset you prepared in the GeoJSON file into your webpage; the file is named *cityData.json* in Example 6. Once loaded, this information is used to draw and resize the proportional symbols atop the tile service.

Use the jQuery function `getJSON()` to load the GeoJSON file (EX6: 1); this code block should be placed within the `ready()` callback function(), after adding the `tileLayer`. The `getJSON()` method makes an AJAX request to a specified file (*cityData.json*). After the GeoJSON file is

loaded completely, the data contained in the file is accessible through the `done()` method chained to the `getJSON()` method (EX6: 2–4). In this case, the data object, which is itself a JSON object, is passed as an argument to the callback function(). Use `console.log()` to confirm that the GeoJSON file is loaded correctly (EX6: 3). Finally, an alert is sent if the GeoJSON fails to load (EX6: 5).

Save your changes to the *main.js* file and refresh the *index.html* page in the browser. The contents of the GeoJSON file now should be logged to the console (Figure 4).

```
1 $.getJSON("data/cityData.json")
2   .done(function(data) {
3     console.log(data);
4   })
5   .fail(function() { alert("There has been a problem loading the data.");});
```

Example 6: Loading the GeoJSON (in: *main.js*).

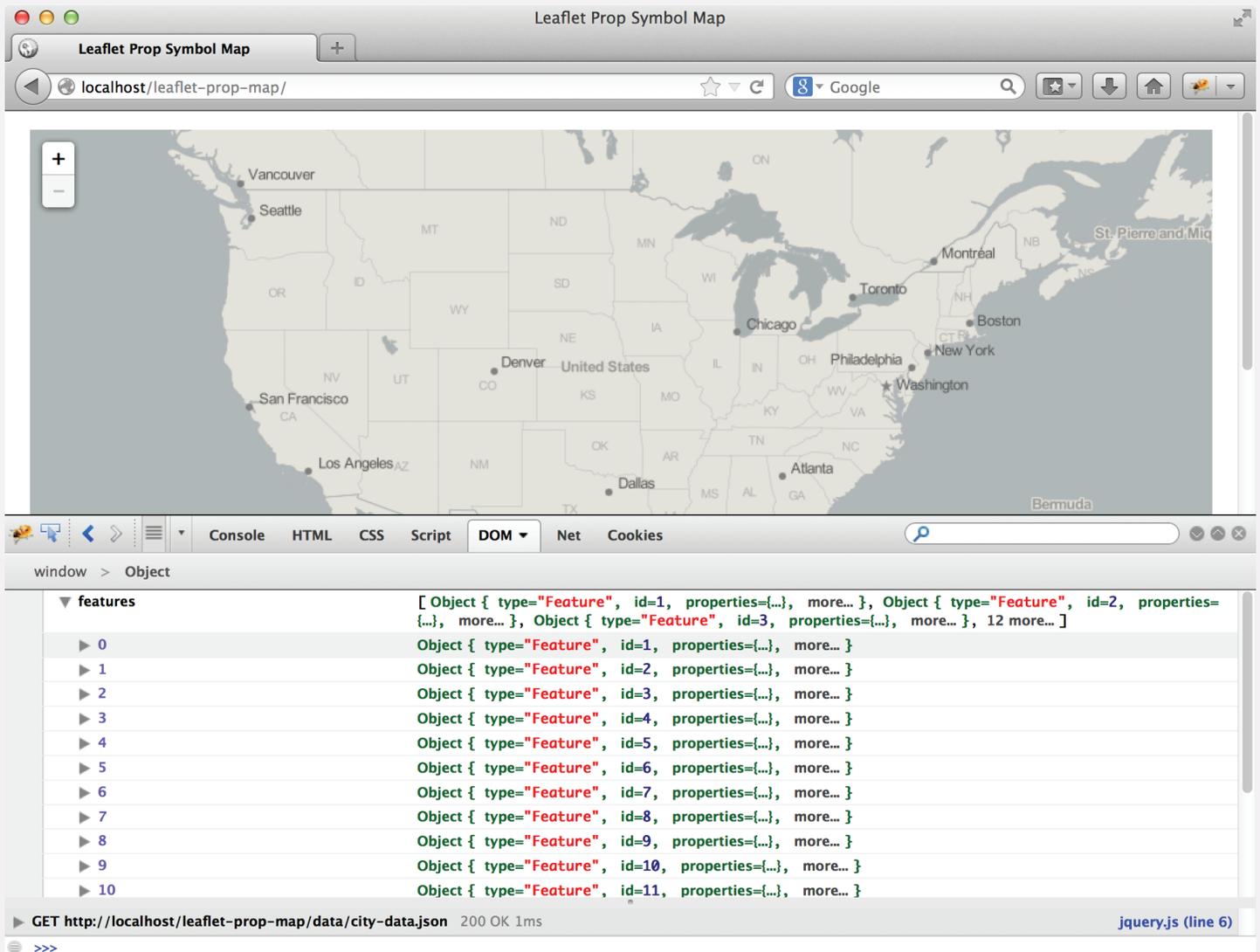


Figure 4: Printing the GeoJSON to the console.

5. PROCESSING THE GEOJSON

ONCE THE GEOJSON is loaded, you can use Leaflet to immediately draw the geographic linework as SVG markers atop the tileset (see Step 6 below). However, to improve the script's efficiency, first process the data to derive values that will be useful later on. In this tutorial, several pieces of information are derived dynamically from the GeoJSON so they need not to be hardcoded into the script. These data include the timestamp names (i.e., the name of each column) for use in a temporal legend and the minimum/maximum values across the time series for use in a map legend.

First remove the `console.log()` call within the `done()` function (EX6: 3) and replace it with a call to a new function named `processData()` (EX7: 3). The function takes

the data variable holding the loaded GeoJSON as a parameter. As described below, the `processData()` function returns the derived information as three key value pairs stored within a JavaScript object. Declare a new variable called `info` to store the returned values for future use in the temporal and map legends.

Next, define the `processData()` function (Example 8). The `processData()` function begins by defining three local variables used to store the derived information (EX8: 2–4): 1) `timestamps` (an array holding the time series headers from the GeoJSON), 2) `min` (a number holding the lowest value across the time series), and 3) `max` (a number holding the highest value across the time series).

```

1 $.getJSON("data/cityData.json")
2   .done(function(data) {
3     var info = processData(data);
4   })
5   .fail(function() { alert("There has been a problem loading the data.");});

```

Example 7: Calling the `processData()` function (in: `main.js`).

```

1 function processData(data) {
2   var timestamps = [];
3   var min = Infinity;
4   var max = -Infinity;
5
6   for (var feature in data.features) {
7
8     var properties = data.features[feature].properties;
9
10    for (var attribute in properties) {
11
12      if ( attribute != 'id' &&
13          attribute != 'name' &&
14          attribute != 'lat' &&
15          attribute != 'lon' ) {
16
17        if ( $.inArray(attribute,timestamps) === -1) {
18          timestamps.push(attribute);
19        }
20
21        if (properties[attribute] < min) {
22          min = properties[attribute];
23        }
24
25        if (properties[attribute] > max) {
26          max = properties[attribute];
27        }
28      }
29    }
30  }
31
32  return {
33    timestamps : timestamps,
34    min : min,
35    max : max
36  }
37 }

```

Example 8: Processing the GeoJSON (in: `main.js`).

The `processData()` function then makes use of a nested looping structure to determine the values for these three local variables. First, a `for` loop is used to traverse each of the features in the `data` variable, treating each of the map features included in the GeoJSON one at a time (EX8: 6). Next, the properties associated with the given feature (i.e., the header names for all attributes in the GeoJSON) are stored in a local variable called `properties` (EX8: 8). A second `for` loop then is used to traverse through each attribute in the `properties` variable (EX8: 10). In other words, this nested looping structure accesses each map feature in the GeoJSON individually, and then accesses each of the attributes associated with a given feature individually before moving onto the next map feature.

Once a single attribute of a single map feature is isolated using the nested looping structure, the attribute is evaluated according to four `if` statements to determine if it influences the derived information (e.g., if it is a new timestamp name or the min/max value):

1. An `if` statement first is included to test if the current attribute is one of the included timestamps (e.g., 2005, 2006), or if it instead is the `id`, `name`, `lat`, or `long` column in the GeoJSON (EX8: 12–15). You will need to modify the set of conditions included in the `if` statement if you added additional columns to your GeoJSON, or gave the columns different header names.

2. If the attribute is one of the timestamps (i.e., if it conforms to the aforementioned conditions), then it is appended to the end of the `timestamps` array (EX8: 18). This `push()` call is encapsulated within an `if` statement that checks if the given timestamp name already has been added to the `timestamps` array (i.e., if 2005 already exists in the array) (EX8: 17–19); if it does not exist (`===`), a value of -1 is returned and the attribute name is appended to the `timestamp` array.
3. Next, an `if` statement is used to check if the value of the current attribute for the current feature is smaller than the current value assigned to the `min` variable (EX8: 21–23). If the value is smaller, then the `min` value is replaced with the attribute value of the current feature.
4. Finally, an `if` statement is used to check if the value of the current attribute for the current feature is larger than the current value assigned to the `max` variable (EX8: 25–27). The logic in this `if` statement is conceptually opposite to that used to update the `min` variable.

Once the nested looping structure works through all properties of all features, the `timestamp`, `min`, and `max` variables are returned (EX8: 32–36), concluding the `processData()` function.

6. DRAWING THE PROPORTIONAL SYMBOLS

WITH THE GEOJSON loaded and processed, it is now time to add the proportional symbols to the map. Leaflet supports the overlay of map symbols, or *markers*, using either pre-rendered iconic point symbols (e.g., in `.png` format) or dynamically drawn scalable vector graphics (SVG). Because SVG is a vector image format rendered in the browser, SVG markers can be easily resized with a change

in the time series data. This advantage makes SVG the preferred format for thematic web mapping generally.

To add markers to the map, first return to the `done()` function and note that there are now two local variables based on the GeoJSON: (1) the `data` object containing the GeoJSON and (2) the `info` array containing the three

```
1 $.getJSON("data/cityData.json")
2   .done(function(data) {
3       var info = processData(data);
4       createPropSymbols(info.timestamps, data);
5   })
6   .fail(function() { alert("There has been a problem loading the data.");});
```

Example 9: Calling the `createPropSymbols()` Function (in: `main.js`).

```

1 function createPropSymbols(timestamps, data) {
2
3     cities = L.geoJson(data).addTo(map);
4
5 }

```

Example 10: Adding Teardrop Markers to the Map (in: main.js).

derived variables returned by the `processData()` function (Example 9). Following the call to `processData()`, call a new functional named `createPropSymbols()`, passing `info.timestamps` and `data` as parameters (EX9: 4).

Next, define the `createPropSymbols()` function, adding it after the `processData()` definition (Example 10). Because of the popularity of the GeoJSON format, Leaflet offers the method `L.geoJson()` to create a new GeoJSON FeatureGroup from the geographic information contained within a GeoJSON file. A GeoJSON FeatureGroup is a specialized type of FeatureGroup, a Leaflet class that is used to group multiple map layers together, allowing for the group to be treated as one whole programmatically. When using Leaflet, a “layer” refers to a single map feature (e.g., a

point marker, a polygon), meaning that the FeatureGroup is closer to the concept of a “layer” or “geometry Collection” in GIS software. Refer to the Leaflet documentation for additional details about the FeatureGroup class: leaflet-js.com/reference.html#featuregroup. Create a GeoJSON FeatureGroup and assign it to the previously declared `cities` variable using `L.geoJson()`, passing the data object as the parameter (EX10: 3). Then call the `addTo()` function, passing the global map variable as the parameter in order to place the markers onto the map.

Save your changes to the `main.js` file and refresh the `index.html` page in the browser. You now should see your map features added to the map as inverted teardrop markers, the Leaflet default for point features (Figure 5).

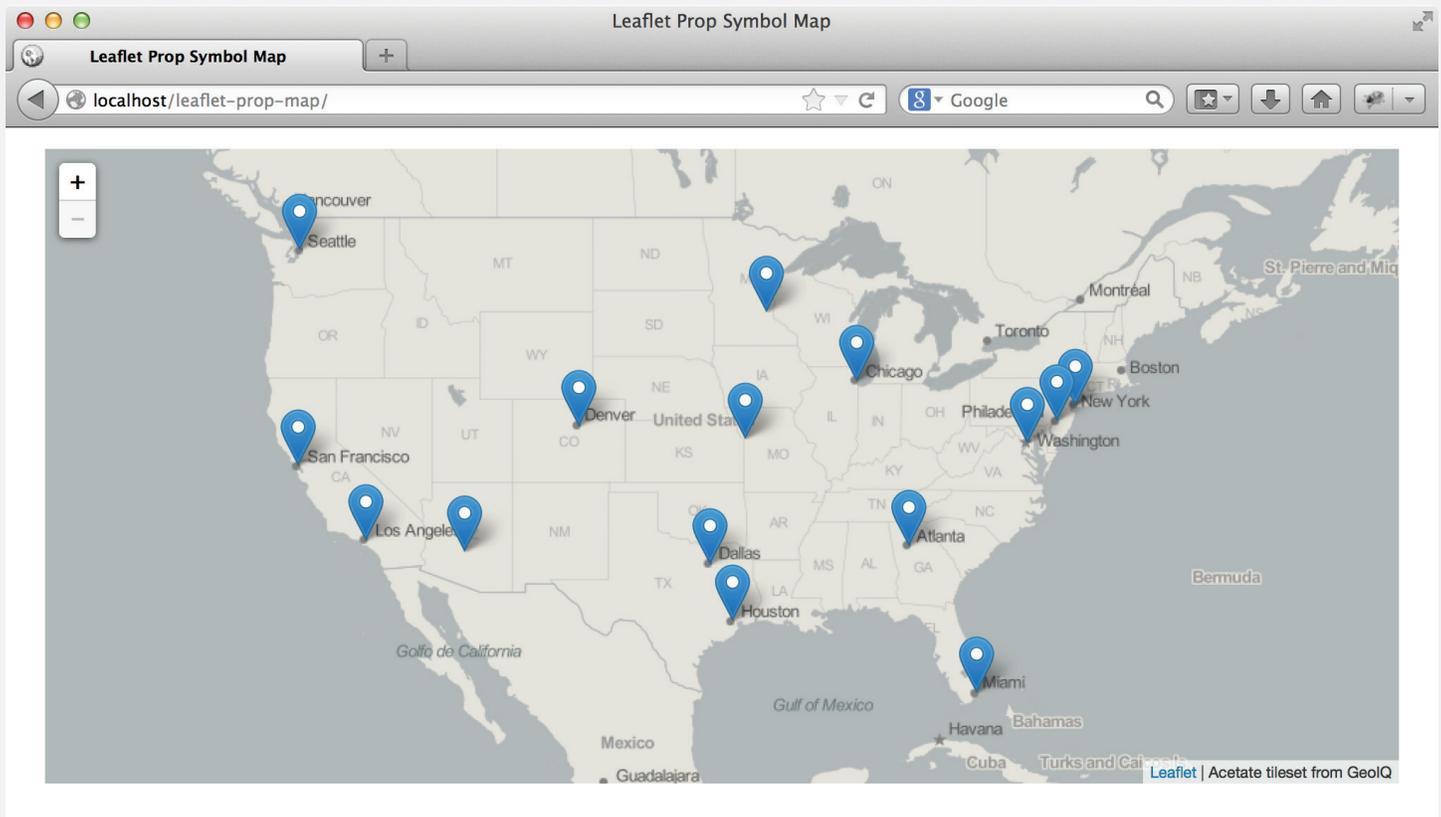


Figure 5: Adding markers to the map.

```

1  function createPropSymbols(timestamps, data) {
2
3      cities = L.geoJson(data, {
4
5          pointToLayer: function(feature, latlng) {
6
7              return L.circleMarker(latlng, {
8                  fillColor: "#708598",
9                  color: '#537898'
10                 weight: 1,
11                 fillOpacity: 0.6
12             }).on({
13
14                 mouseover: function(e) {
15                     this.openPopup();
16                     this.setStyle({color: 'yellow'});
17                 },
18                 mouseout: function(e) {
19                     this.closePopup();
20                     this.setStyle({color: '#537898'});
21                 }
22             });
23         });
24     });
25     }).addTo(map);
26 }

```

Example 11: Replacing teardrop markers with circle markers and adding event listeners for a popup window (in: main.js).

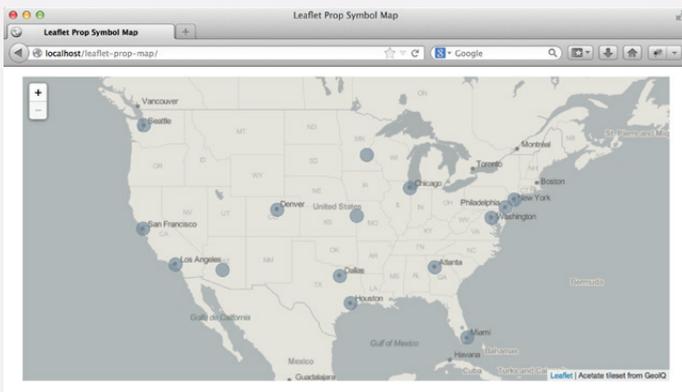


Figure 6: Drawing circles on the map.

A non-compact, teardrop symbol is not ideal for proportional symbol mapping. Make use of Leaflet’s `pointToLayer()` function to draw custom SVG markers for the proportional symbols, rather than using the teardrop images (Example 11). Using `pointToLayer()`, you are able to draw each proportional symbol as a Leaflet

`CircleMarker`, giving you control over the styling of the map symbol. The tutorial example manipulates the color (EX11: 8–9), stroke width (EX11: 10), and opacity (EX11: 11) of the proportional symbols. Additional details about the `CircleMarker` class, are available at: leafletjs.com/reference.html#circlemarker.

Again save your changes to the `main.js` file and refresh the `index.html` page in the browser. You now should see your map features added as partially transparent blue symbols, centered upon the lat/long location of the map feature (Figure 6).

Before moving on, append an additional method `on()` to the `pointToLayer()` callback function, which adds a pair of event listeners to open and close the popup window upon `mouseover` and `mouseout`, respectively (EX11: 12–23). Note that the popup window and content is not yet bound to these symbols, which we will do in the subsequent steps (see EX13: 14).

7. SCALING THE PROPORTIONAL SYMBOLS

AFTER DRAWING the SVG markers to the map, you now need to add the functionality to resize each marker according to a value in the time series. Such a function needs to be applied uniquely to each `CircleMarker` layer in the newly created `cities` `GeoJson` `FeatureGroup` layer, as each proportional symbol on your map has a different set of attributes values (i.e., a differently sized proportional symbol), and these values vary different over the time series.

To resize the proportional symbols, first return to the `createPropSymbols()` function and add a call to a new function named `updatePropSymbols()`. The call to this new function should come at the end of the

`createPropSymbols()` definition, after the `cities` `FeatureGroup` is added to the map (**EX12: 26**). The `updatePropSymbols()` function takes as a parameter the value stored in the first index position of the timestamps array (i.e., the first date in the time series).

Next, declare and define two new functions: 1) the aforementioned `updatePropSymbols()` function used to resize each proportional symbol individually and 2) a `calcPropRadius()` function providing the math to compute the appropriate size of a proportional symbol given its attribute value (**Example 13**). These pair of functions should be defined in `main.js` after the `createPropSymbols()` function.

```
1 function createPropSymbols(timestamps, data) {
2
3     cities = L.geoJson(data, {
4
5         pointToLayer: function(feature, latlng) {
6
7             return L.circleMarker(latlng, {
8
9                 fillColor: "#708598",
10                color: '#537898'
11                weight: 1,
12                fillOpacity: 0.6
13            }).on({
14
15                mouseover: function(e) {
16                    this.openPopup();
17                    this.setStyle({color: 'yellow'});
18                },
19                mouseout: function(e) {
20                    this.closePopup();
21                    this.setStyle({color: '#537898'});
22                }
23            });
24        });
25    }
26    }).addTo(map);
27
28    updatePropSymbols(timestamps[0]);
29
30 }
```

Example 12: Updating the proportional circles by timestamp (in: main.js).

```

1    function updatePropSymbols(timestamp) {
2
3        cities.eachLayer(function(layer) {
4
5            var props = layer.feature.properties;
6            var radius = calcPropRadius(props[timestamp]);
7            var popupContent = "<b>" + String(props[timestamp]) +
8                " units</b><br>" +
9                "<i>" + props.name +
10               "</i> in </i>" +
11               timestamp + "</i>";
12
13            layer.setRadius(radius);
14            layer.bindPopup(popupContent, { offset: new L.Point(0,-radius) });
15        });
16    }
17    function calcPropRadius(attributeValue) {
18
19        var scaleFactor = 16;
20        var area = attributeValue * scaleFactor;
21        return Math.sqrt(area/Math.PI)*2;
22    }

```

Example 13: Scaling the proportional circles (in: main.js).

The `updatePropSymbols()` function begins by calling the Leaflet `eachLayer()` method on the `cities` GeoJson `FeatureGroup` (EX13: 3). The `eachLayer()` is a simplified loop offered by Leaflet that applies the same logic (here a new function definition) to every layer included in a `FeatureGroup`. Within the new `function()` definition, three local variables are defined: 1) `props`, storing the complete set of attributes for the given proportional symbol (EX13: 5), 2) `radius`, storing the attribute value of the proportional symbol for the current `timestamp` (EX13: 6), and 3) `popupContent`, storing the markup text used to populate an information window popup (EX13: 7–11). The latter variable can be modified to include whatever information you wish to present to the user upon probing a proportional symbol.

Note that assignment of the `radius` variable makes use of the custom `calcPropRadius()` method (EX13: 17–22), which takes the attribute value of the proportional symbol and multiplies it against an arbitrary `scaleFactor` (here, the value of 16 is hardcoded given this particular sample dataset) to determine the area of proportional circle (EX13: 19–20). Experiment with the `scaleFactor` to

find a value that works well with your dataset; the larger the `scaleFactor`, the larger all proportional symbols will be. The `radius` then is calculated and returned to the `updatePropSymbols()` function, as the `CircleMarker` class scales a marker by a `radius` value rather than an area value. This geometry logic is included in a separate `calcPropRadius()` function, rather than embedded in the `updatePropSymbols()` function, so that it also can be used to resize the symbols included in the map legend.

The `updatePropSymbols()` function proceeds by calling two methods from the `CircleMarker` class on the currently treated layer: 1) `setRadius()`, which adjusts the size of the proportional symbol (EX13: 13) and 2) `bindPopup()`, which binds the aforementioned `popupContent` markup text to the proportional symbol (EX13: 14).

Save your changes to the `main.js` file and refresh the `index.html` page in the browser. You now should see your proportional symbols scaling according to the first timestamp (Figure 7). Your proportional symbols also should have popup functionality to retrieve details about the probed symbol.

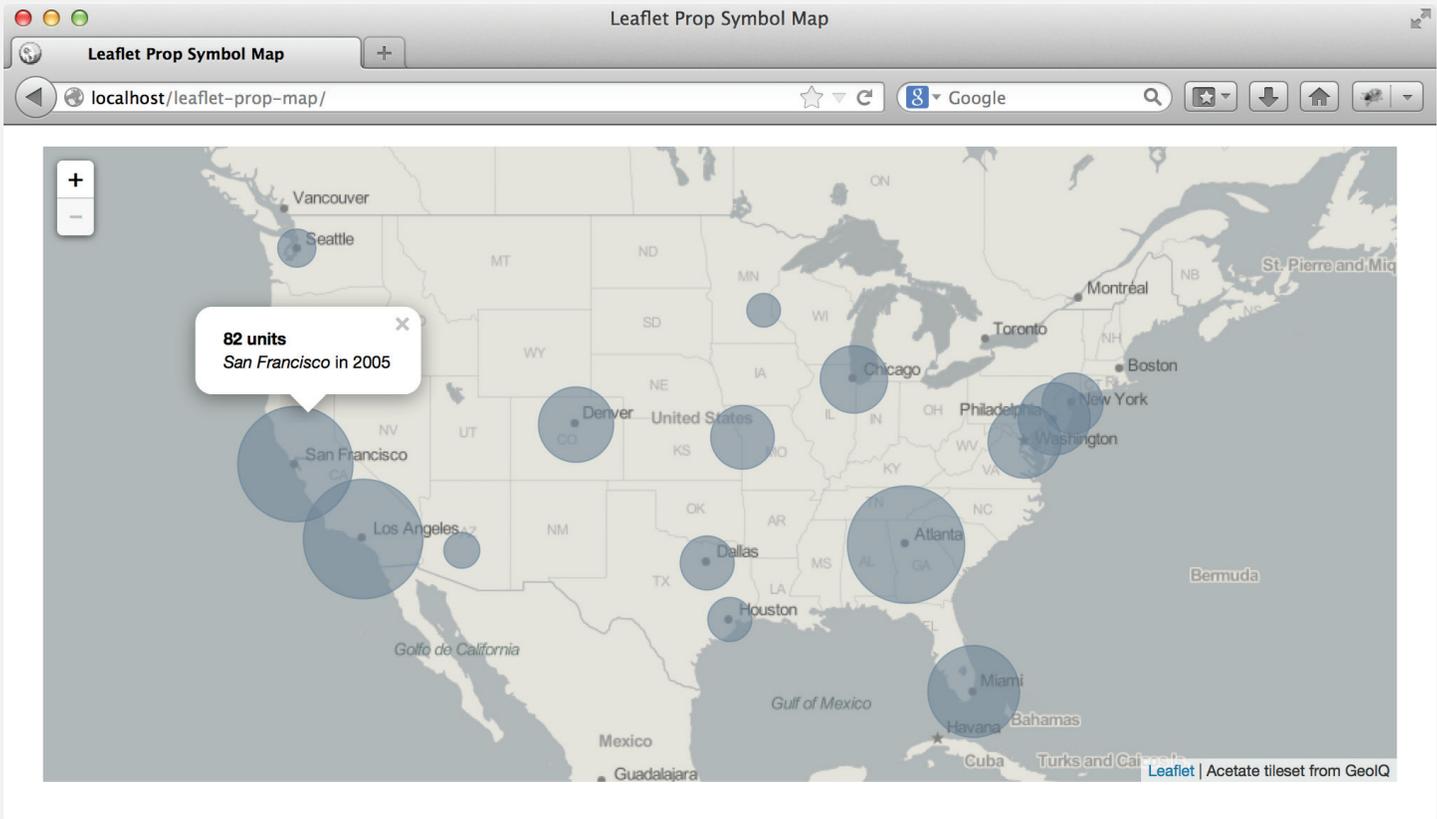


Figure 7: Scaling the proportional symbols and binding a popup window to symbols.

8. CREATING A MAP LEGEND

WHILE THE POPUP WINDOW provides a way to determine the specific value of each proportional symbol, it is conventional also to include a persistent map legend indicating the values of several example symbols. The following tutorial creates a map legend using HTML elements, in this case simple `<div>` elements styled with the CSS property `border-radius` value of 50%, which rounds the corners of the `<div>` elements to make them appear as circles (EX16: 19). The `calcPropRadius()` method then is used to dynamically resize the legend symbols.

First return to the `done()` function and add a call to a new function named `createLegend()`. The call to this new

function should follow the existing call to the `createPropSymbols()` function (EX14: 5). The `createLegend()` function takes as parameters the minimum and maximum values across the time series, as identified through the `processData()` function.

Next, define the `createLegend()` function, placing the function definition after the `calcPropRadius()` definition (Example 15). The `createLegend()` function makes use of the Leaflet `L.control()` method, which adds a new UI element to the map, and the `L.DomUtil()` method for creating a new DOM entity. Read more about these methods in the API reference at leafletjs.com/reference.

```

1    $.getJSON("data/cityData.json")
2        .done(function(data) {
3            var info = processData(data);
4            createPropSymbols(info.timestamps, data);
5            createLegend(info.min, info.max);
6        })
7    .fail(function() { alert("There has been a problem loading the data.");});

```

Example 14: Calling the `createLegend()` Function (in: `main.js`).

[html#icontrol](#) and [leafletjs.com/reference.html#domutil](#).

The `createLegend()` function begins by assessing the `min` value passed as a parameter, changing it to a value of 10 if below 10 so that the small legend symbol remains visible in the webpage (EX 15: 3–5). You also can add logic here to place a ceiling on the `max` value, if desired. A local function named `roundNumber()` then is defined that rounds the input value to the nearest increment of 10 (EX15: 7–10); again, you can adjust this function to round to a different value (25, 100, etc.) depending on your time series dataset.

The `createLegend()` function proceeds by using `L.control()` to create a new control named `legend` that contains the map legend, setting the `position` style to `bottomright` on the map (EX15: 12). Leaflet's `onAdd()` event listener then is attached to the `legend` control so that the legend is configured only after added to the map. The `onAdd()` event listener defines a new `function()` that first declares seven local variables:

1. `legendContainer`, a wrapper `<div>` that holds the graphic and text elements in the legend (EX15: 16); note that the `DomUtil()` function is evoked to add the `<div>` as a page element;
2. `symbolsContainer`, a `<div>` that contains the example proportional symbols in the legend (EX15: 17); again, `DomUtil()` is used to add the `<div>` to the webpage;
3. `classes`, an array holding the values of the `min`, `max`, and a third value in the middle of the attribute range (EX15: 18); this solution produces a legend with three example proportional symbols, but can be modified to include a different number of example symbols;
4. `legendCircle`, an unassigned variable used repeatedly to restyle each of the three legend proportional symbols (EX15: 19);
5. `lastRadius`, a variable assigned an initial value of zero, which will be used to store the value of the previous symbol's radius while looping through the `classes` array (EX15: 20);
6. `currentRadius`, a unassigned variable used to store the current symbol's radius while looping through the `classes` array (EX15: 21);

7. `margin`, an unassigned variable used to store the relative pixel distance of each of the legend's proportional symbols from the left side of their parent container, in order to horizontally align them with each other (EX15: 22).

Before adding the logic needed to draw the legend, first disable the panning of the tiled basemap underneath the legend. To do this, pass the `mousedown` event into the callback `function()` and use the `stopPropagation()` method to prevent the click behavior from being applied to the legend's parent object, the Leaflet map (EX15: 24–26).

Once the `mousedown` event is disabled, the `createLegend()` function continues by selecting the newly created `legendContainer` element and adding an `h2` header element to it (EX15: 28). A `for` loop then is used to iterate through the three values within the `classes` array and to add new `<div>` elements to the `legendContainer` and provide widths that are proportionate to twice the values in the `classes` array (EX15: 30–47). The `for` loop first creates a new `<div>` element for the given legend symbol, storing it in the previously declared `legendCircle` variable so that CSS rules can be applied to the `<div>` (EX15: 32). The radius of the proportional symbol then is calculated (EX15: 34). With each iteration through the `for` loop, the value assigned to `margin` is calculated using the values of `currentRadius` and `lastRadius` (EX15: 34). Because each of the three `legendCircle` `<div>` elements is given a display property of `inline-block` (EX16: 22) within the external style sheet, they will normally flow alongside each other within the layout of the `legendContainer`. Deriving a negative value for the left margin of each allows them to be stacked on top of one another and vertically aligned, producing a nested display result. As the loop iterates from the smallest symbol to the largest, the negative left margin value is calculated using the current symbol's width, the previous symbol's width, as well as two additional pixel value units to account for the 1px border applied to the symbols within the external style sheet (EX16: 20). The newly created `legendCircle` element is then selected using `jQuery`, given a `width` attribute based on its current radius (multiplied by two in this case to fill the full width or diameter of the `div` element), a `height` attribute of the same value, and the calculated `margin` value to offset the symbol's `margin-left` property (EX15: 38–40). The `legendCircle` then is appended to the `symbolsContainer` and the `lastRadius` is set to the `currentRadius` value before the loop iterates again.

```

1  function createLegend(min, max) {
2
3      if (min < 10) {
4          min = 10;
5      }
6
7      function roundNumber(inNumber) {
8
9          return (Math.round(inNumber/10) * 10);
10     }
11
12     var legend = L.control( { position: 'bottomright' } );
13
14     legend.onAdd = function(map) {
15
16         var legendContainer = L.DomUtil.create("div", "legend");
17         var symbolsContainer = L.DomUtil.create("div", "symbolsContainer");
18         var classes = [roundNumber(min), roundNumber((max-min)/2), roundNumber(max)];
19         var legendCircle;
20         var lastRadius = 0;
21         var currentRadius;
22         var margin;
23
24         L.DomEvent.addListener(legendContainer, 'mousedown', function(e) {
25             L.DomEvent.stopPropagation(e);
26         });
27
28         $(legendContainer).append("<h2 id='legendTitle'># of somethings</h2>");
29
30         for (var i = 0; i <= classes.length-1; i++) {
31
32             legendCircle = L.DomUtil.create("div", "legendCircle");
33
34             currentRadius = calcPropRadius(classes[i]);
35
36             margin = -currentRadius - lastRadius - 2;
37
38             $(legendCircle).attr("style", "width: " + currentRadius*2 +
39                 "px; height: " + currentRadius*2 +
40                 "px; margin-left: " + margin + "px" );
41             $(legendCircle).append("<span class='legendValue'>" + classes[i] + "</span>");
42
43             $(symbolsContainer).append(legendCircle);
44
45             lastRadius = currentRadius;
46
47         }
48

```

Example 15: Creating a map legend (in: main.js). Continued next page.

```

49     $(legendContainer).append(symbolsContainer);
50
51     return legendContainer;
52
53     };
54
55     legend.addTo(map);
56
57 } // end createLegend()

```

Example 15, continued.

After adding all three proportional symbols to the `symbolsContainer` `<div>` and concluding the for loop, the `symbolsContainer` `<div>` is appended to the `legendContainer` `<div>` (EX15: 49). Finally, the `legendContainer` `<div>` is returned to the `legend` variable through the `onAdd()` callback function() (EX15: 51). The

`createLegend()` function concludes by adding the legend control to the map (EX15: 55).

Save your changes to the `main.js` file and refresh the `index.html` page in the browser. You now should see a map legend in the bottom, right corner of the map (Figure 8).

```

1     .legend, .temporal-legend {
2         padding: 6px 10px;
3         font: 14px/16px Arial, Helvetica, sans-serif;
4         background: white;
5         background: rgba(255,255,255,0.8);
6         box-shadow: 0 0 15px rgba(0,0,0,0.2);
7         border-radius: 5px;
8     }
9     #legendTitle {
10        text-align: center;
11        margin-bottom: 15px;
12        font-variant: small-caps;
13    }
14    .symbolsContainer {
15        float: left;
16        margin-left: 50px;
17    }
18    .legendCircle {
19        border-radius:50%;
20        border: 1px solid #537898;
21        background: rgba(113, 133, 152, .6);
22        display: inline-block;
23    }
24    .legendValue {
25        position: absolute;
26        right: 8px;
27    }

```

Example 16: Style rules for creating a nested proportional symbol legend using `div` elements (in: `style.css`).

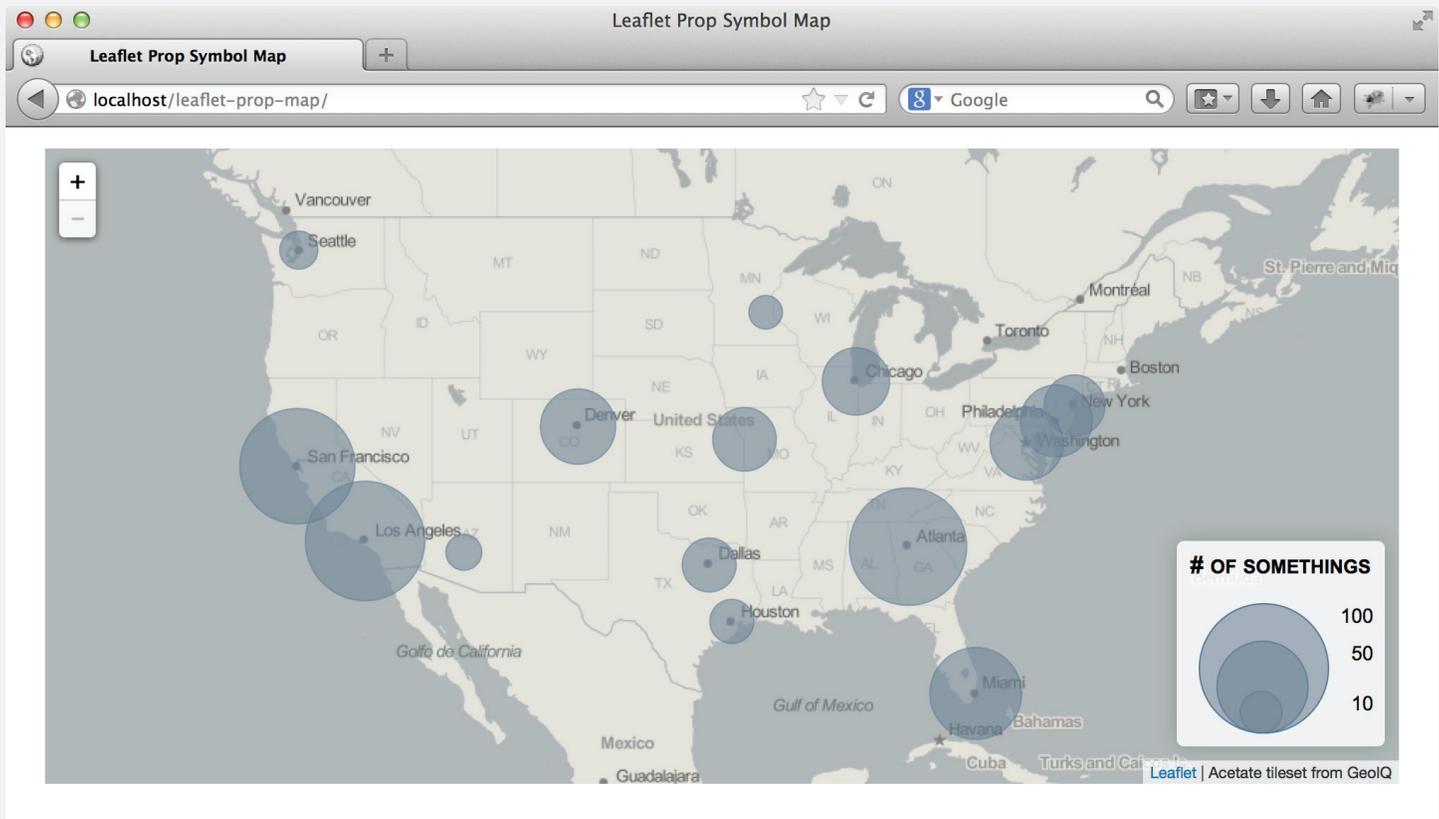


Figure 8: Adding a map legend.

9. ADDING A TEMPORAL SLIDER

THE NEXT STEP in completing the time series proportional symbol map is implementation of a temporal slider for displaying the time series. A *slider* is a UI widget that allows users to set the value of an ordinal or, more commonly, numerical variable; checkboxes (allowing compound selection of multiple values) or radio buttons (constraining selection to a single value in a set) are used for categorical variables. A temporal slider thus allows the user to change the current timestamp, updating the map to any point in the time series. A slider widget works best for depictions of linear time rather than cyclical time, following a timeline metaphor rather than a clock metaphor. Several options exist for implementing a slider widget within a web page (we will implement option 3 in this tutorial):

1. **jQueryUI** is a plugin library for jQuery that supports a range of common UI widgets. The jQueryUI plugin includes default graphics needed for the interface widgets as well as associated events and effects for implementing these widgets. Before getting started with jQueryUI, review the jQueryUI API Documentation and the jQueryUI demo pages at jqueryui.com/.

2. **noUiSlider** is a smaller jQuery plugin written specifically to create a range slider element, rather than the host of UI widgets supported by jQueryUI. The code for this plugin is available at: refreshless.com/nouislider/.
3. Finally, the HTML5 specification now includes a range type for the `<input>` element: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Input>. The range type makes it easy to create a simple slider and apply basic styles to it. Note that the range type is a W3C recommendation and still is in the process of gaining support among web browsers. If support among older browsers is important, you may wish to use one of the first two plugins mentioned above. Their implementation will be very similar to the process described here.

To implement the temporal slider, return to the `done()` function and add a call to a new function named `createSliderUI()` (EX17: 6). Pass the `timestamps` array as the argument with this function call. This is the last

```

1 $.getJSON("data/cityData.json")
2   .done(function(data) {
3     var info = processData(data);
4     createPropSymbols(info.timestamps, data);
5     createLegend(info.min,info.max);
6     createSliderUI(info.timestamps);
7   })
8   .fail(function() { alert("There has been a problem loading the data.");});

```

Example 17: Calling the `createSliderUI()` function (in: `main.js`).

function invoked from within the `done()` callback function().

Next, define the `createSliderUI()` function, placing the function definition after the `createLegend()` definition (**Example 18**). The `createSliderUI()` function first uses the `L.Control()` method to add a new control named `sliderControl` to the map (**EX18: 3**). Note that there are now two Leaflet controls added to the map: one for the

map legend and one for the slider control, with the latter positioned in the bottom, left corner of the map.

Leaflet's `onAdd()` event listener then is attached to `sliderControl` to configure the temporal slider after it is added to the map (**EX18: 5–24**). The `onAdd()` event listener first adds a new `<input>` element to the DOM named `slider` using the `L.DomUtil()` function (**EX18: 7**); the `<input>` element is given the class name `range-slider` so that it can be styled. As with the map legend above, the

```

1 function createSliderUI(timestamps) {
2
3   var sliderControl = L.control({ position: 'bottomleft' });
4
5   sliderControl.onAdd = function(map) {
6
7     var slider = L.DomUtil.create("input", "range-slider");
8
9     L.DomEvent.addListener(slider, 'mousedown', function(e) {
10       L.DomEvent.stopPropagation(e);
11     });
12
13     $(slider)
14       .attr({'type': 'range',
15            'max': timestamps[timestamps.length-1],
16            'min': timestamps[0],
17            'step': 1})
18       .on('input change', function() {
19         updatePropSymbols($(this).val().toString());
20       });
21     return slider;
22   }
23
24   sliderControl.addTo(map)
25 }

```

Example 18: Creating a temporal slider (in: `main.js`).

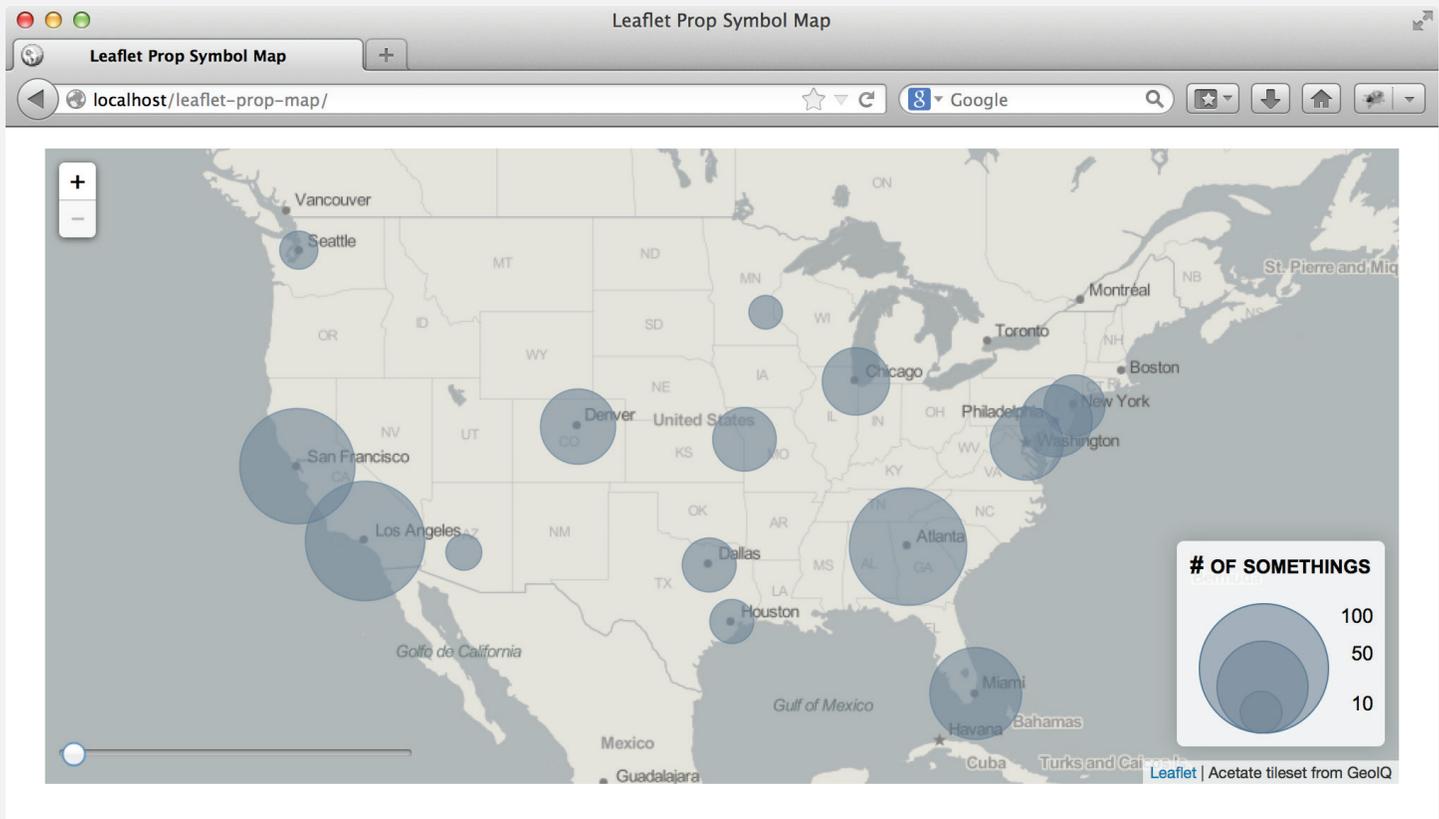


Figure 9: Adding a temporal slider.

`stopPropagation()` method is applied to the slider to prevent the click behavior from being propagated to the slider's parent object, the Leaflet map (EX18: 9–11).

Next, two methods are called on the newly created slider `<input>` element using jQuery. First, the `attr()` method is called to set four properties of the slider element:

1. the `type`, using the aforementioned range type (EX18: 14);
2. the maximum value of the slider, using the last value in the timestamps array (EX18:15)
3. the minimum value of the slider, using the first value in the timestamps array (EX18: 16);
4. the `step` interval, set to 1 to increment by one year for the tutorial example (EX18: 17).

The `on()` method then is called to listen for any change to the slider `<input>` element (EX18: 18–29). Traditionally this is behavior that needed to be written explicitly with the jQueryUI or noUiSlider JavaScript plugin, but is now supported within the browser itself. On any change to the slider (i.e., when the user interacts with it), the `updatePropSymbols()` function is called, passing the current value of the slider (i.e., the new timestamp value) to the `updatePropSymbols()` function (EX18: 19). Finally, the slider is returned to the `onAdd()` callback function() (EX18: 21). The `createSliderUI()` function concludes by calling the `addTo()` function, adding the `sliderControl` to the Leaflet map (EX18: 24).

Save your changes to the `main.js` file and refresh the `index.html` page in the browser. You now should see the slider widget in the bottom, left corner of the map (Figure 9). Take a second to play with the slider widget to ensure your complete time series dataset is loaded and mapped correctly.

10. CREATING A TEMPORAL LEGEND

FINALLY, THE TEMPORAL SLIDER requires a legend to alert the user to the current timestamp portrayed in the map. The temporal legend should update as the user interacts with the temporal slider. To create a temporal legend, make several modifications to the `createSliderUI()` function (**Example 19**):

1. First declare a fifth property named `value` for the slider `<input>` element (**EX19: 18**). This variable will store the name of the current timestamp (i.e., the header from your time series dataset). Assign the first timestamp as a default.
2. Next, add logic to the `on()` method to update this `value` property when the user changes the position of the slider `<input>` element (**EX19: 21**). Note that this logic actually changes the text of a page

element with the class name `temporal-legend`, which is an `<output>` element added to the DOM in the subsequently defined `createTemporalLegend()` function.

3. Finally, add a call to this new `createTemporalLegend()` function, passing the first value in the `timestamps` array as a parameter (**EX19: 27**).

Next, declare the `createTemporalLegend()` function at the bottom of the `main.js` file. This function is similar to the `createSliderUI()` function. First, a new control named `temporalLegend` is added to the bottom, left corner of the map using the `L.control()` method (**EX20: 3**). The `onAdd()` event listener then is called on the `temporalLegend` control, which creates an `<output>` element (a new HTML5 element used to represent the result of

```
1 function createSliderUI(timestamps) {
2
3     var sliderControl = L.control({ position: 'bottomleft' });
4
5     sliderControl.onAdd = function(map) {
6
7         var slider = L.DomUtil.create("input", "range-slider");
8
9         L.DomEvent.addListener(slider, 'mousedown', function(e) {
10             L.DomEvent.stopPropagation(e);
11         });
12
13         $(slider)
14             .attr({'type': 'range',
15                 'max': timestamps[timestamps.length-1],
16                 'min': timestamps[0],
17                 'step': 1,
18                 'value': String(timestamps[0])})
19             .on('input change', function() {
20                 updatePropSymbols($(this).val().toString());
21                 $(".temporal-legend").text(this.value);
22             });
23         return slider;
24     }
25
26     sliderControl.addTo(map)
27     createTemporalLegend(timestamps[0]);
28 }
```

Example 19: Updating the temporal slider (in: main.js).

```

1  function createTemporalLegend(startTimestamp) {
2
3      var temporalLegend = L.control({ position: 'bottomleft' });
4
5      temporalLegend.onAdd = function(map) {
6          var output = L.DomUtil.create("output", "temporal-legend");
7          $(output).text(startTimestamp)
8          return output;
9      }
10
11     temporalLegend.addTo(map);
12 }

```

Example 20: Creating a temporal legend (in: main.js).

a calculation or user action) with the class name temporal-legend in the DOM named output (EX20: 6). The output variable storing the element is then passed to the jQuery .text() method to place the first timestamp as its content (EX20: 7), then returned to the callback function() (EX20: 8). Finally, the temporalLegend control

is added to the Leaflet map using the addTo() function (EX20: 11).

For one last time, save your changes to the main.js file and refresh the index.html page in the browser. Congratulations, you have made a time series proportional symbol map using Leaflet and jQuery (Figure 10)!

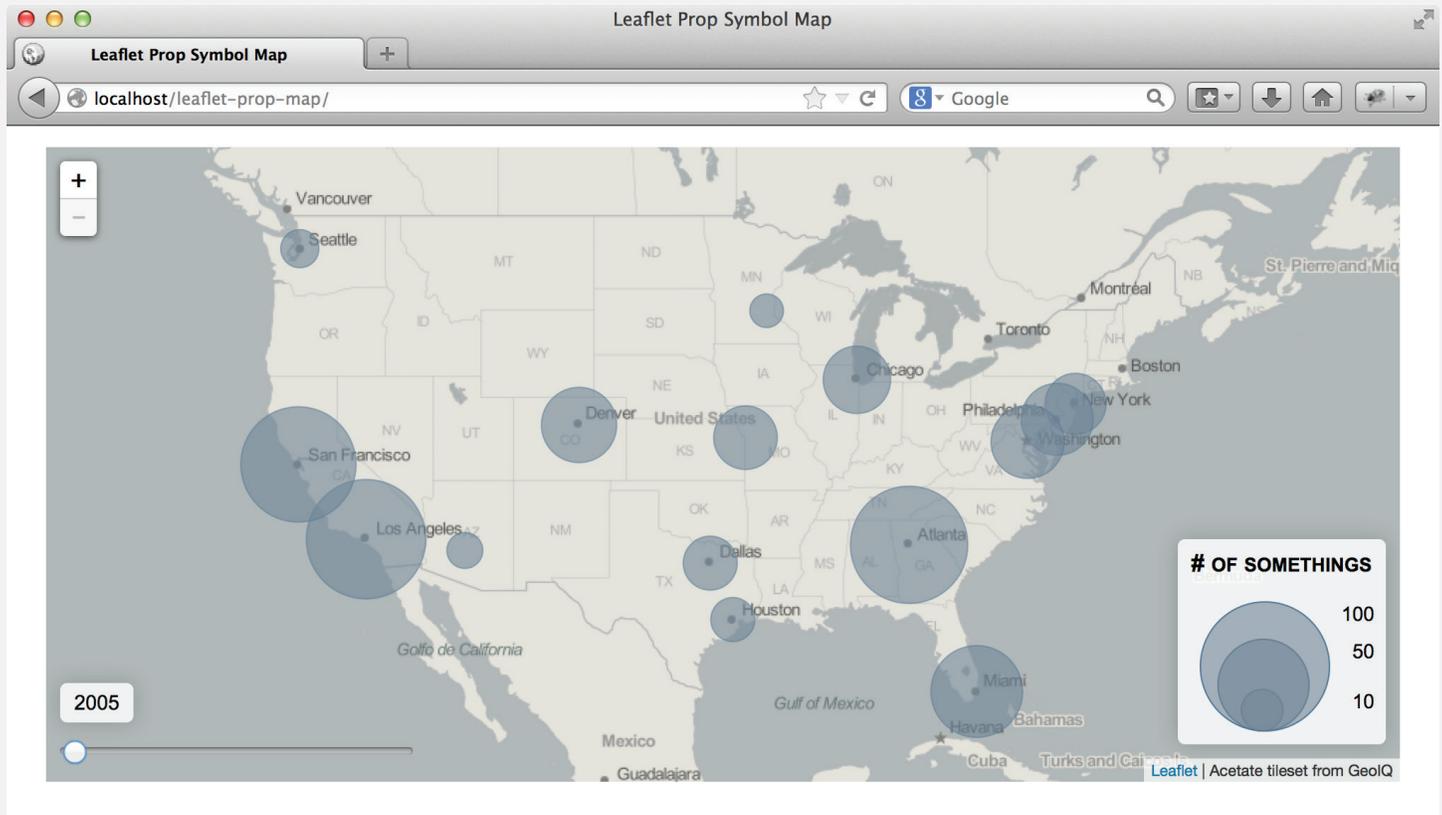


Figure 10: Adding a temporal legend.

THIS TUTORIAL PRESENTS the process and the tools to make a fairly impressive web map depicting time series information using proportional symbols. However, there are many ways you can improve your web map from this baseline. You may wish to enhance the existing map by contextualizing it within a webpage and providing pertinent supplemental content. Give the map a good title, cite your data sources, and consider how you can use additional text to help the map tell a meaningful story. How can the design of the webpage complement the objectives *and* aesthetics of the map?

You also may wish to extend the existing map and code, either through the representation of additional geographic information or by the support of additional user interaction. For example, consider how you could modify the script to add a second dataset of a different nominal type (and perhaps style with a different color)? How could you then use Leaflet's built-in methods (leafletjs.com/reference.html#map-stuff-methods) to provide the user with a

layer control to toggle these layers on and off? Another option would be to load in a second areal dataset and provide a choropleth map layer beneath the proportional symbols (see the Leaflet tutorial for creating an interactive choropleth: leafletjs.com/examples/choropleth.html).

Finally, you may wish to experiment with some of the additional features and functionality provided by the many Leaflet plugins (see leafletjs.com/plugins.html and <https://www.mapbox.com/mapbox.js/plugins>). If you have many point values to visualize, challenge yourself to implement the Leaflet heat map solution (www.mapbox.com/mapbox.js/example/v1.0.0/leaflet-heat) or the Markercluster plugin (github.com/Leaflet/Leaflet.markercluster). Reading and understanding other examples and solutions will greatly improve your ability to customize a map to create compelling graphic narratives. Most importantly, remember to have fun!