Leaflet in Practice: Create webmaps using the JavaScript Leaflet library

Samuel Gachuhi Ngugi

2023-09-01

Contents

Preface

This book was written to assist the budding GIS specialist, geographer or cartographer who has an interest in quickly learning how to create webmaps with minimal knowledge of JavaScript. The aim of this book is to bring the learner up to speed on how to create almost any kind of webmap using Leaflet. Written from the point of a self-taught programmer, this is more of a guide book on how to navigate the Leaflet JavaScript syntax at intermediate level. Nevertheless, it is the wish of this writer that the book appetites you to go deeper in using programming for not only Geographical Information Systems (GIS) purposes, but for any other purpose that fascinates your mind.

Usage

Although the author would highly recommend the reader to sequentially go through the entire book, for the hasty learner, each chapter from Chapter 3 onwards is a stand alone exercise with a link to the source code at the end of every chapter. The link can be found just before the **Summary** subsection. All the code scripts used in this book are available from this Github folder.

https://github.com/sammigachuhi/my-leaflet-project/tree/main/my-leaflet-vs

For the images, they are available here.

https://github.com/sammigachuhi/my-leaflet-project/tree/main/my-leaflet/images

Finally, this book is also available as a Portable Document File (PDF) here. For convenience purposes, this book is better read as a web version available here. This is because the web version does not clip long code along the page margins as is the case with the pdf.

About the Author

Samuel Gachuhi Ngugi is a graduate of the University of Nairobi holding a Bachelor of Arts degree in Geography and Environmental Studies. He also holds a certificate in Environmental Impact Assessment (EIA) in addition to certifications in Google Earth Engine, Site Planning and Machine Learning for Weather and Climate from the European Commission for Medium Range Weather Forecasting (ECMWF) among others. He began to gain an interest in programming in 2021. His first programming book was *A Beginner Friendly Introduction to GIS Operations in R: A practical guide* published in 2022. Prior to this he had also published a GIS tutorial book entitled *Basic Raster and Vector Operations using Qgis: A tutorial* in 2020. Both have been warmly received by the academic community.

Copyright

This book, *Leaflet in Practice: Create webmaps using the JavaScript Leaflet library* has been created under the Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

```
knitr::include_graphics(rep('copyright_image.png'))
```

This is a human-readable summary of (and not a substitute for) the license. Disclaimer.

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for commercial purposes.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Publisher

This book was generated by R Bookdown.

Chapter 1

Introduction

1.1 What is Leaflet?

Something to do with leaves? Of course not.Leaflet, when bare scrapped to its most basic definition, is simply an open source JavaScript library for interactive maps. It was developed in 2011 by Volodymyr Agafonkin, a Ukrainain with a mathematical background.

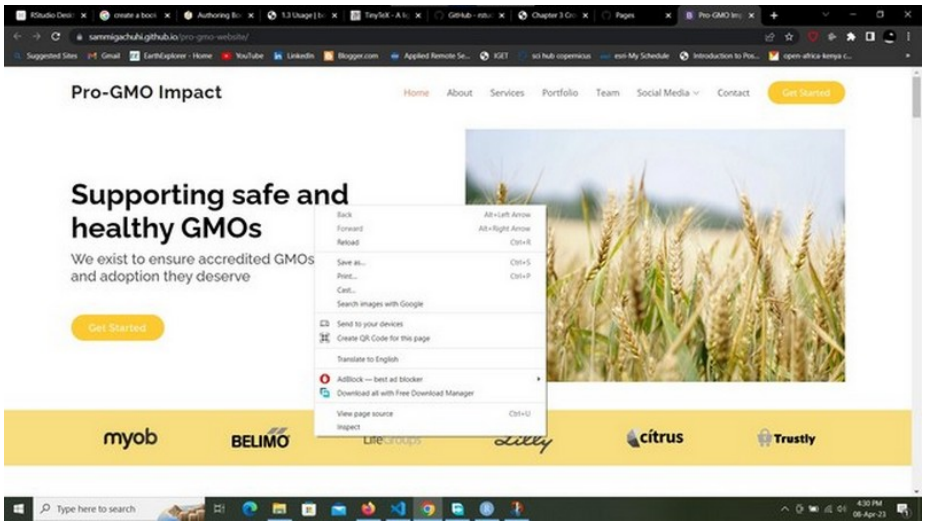
1.2 How does it work?

Leaflet can work if every line of code is inside a `html` document so long as the code appears under the `<script>` tag. However, for a neat work, especially working with complex maps, it is recommended you separate the `html` file from its other components of `main.js` and `style.css` files.

"HTML we know, but what are `main.js` and `style.css` files?", you may ask.

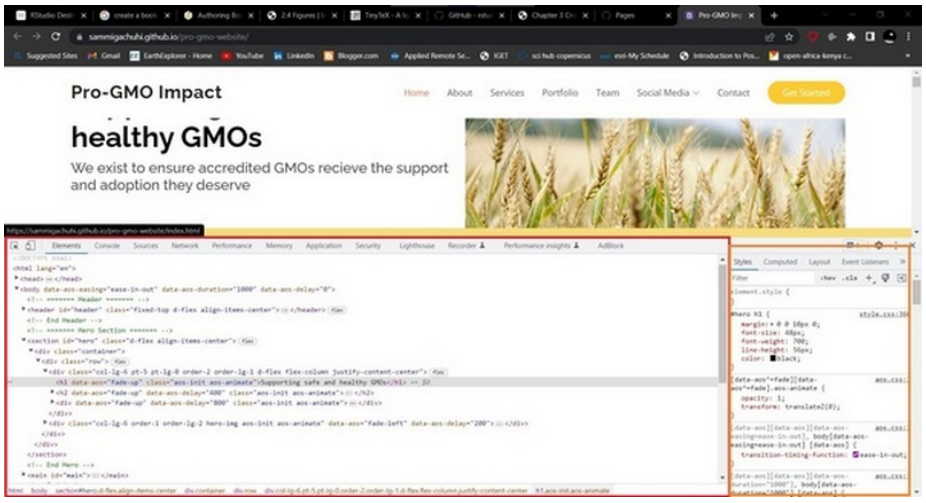
Well, beginning with `html`, which stands for **Hypertext Markup Language**, it is the language that is used in creating webpages. It is actually the standard of making static webmaps. I am yet to come across any webpage that is made up of anything apart from HTML. If you want to have a view of what HTML looks like, just right click any webpage and click *Inspect* in Google Chrome and Firefox. A toolbar will appear at the bottom or side of the webpage, depending on your settings.

```
knitr::include_graphics(rep("inspect.jpg"))
```



Scroll over to the **Element** tab and you will have something that looks like this:

```
knitr::include_graphics(rep("elements.jpg"))
```



The part encircled in red is the `html` that makes up the webpage for the ProGMO website in this case.

So, I am a GIS specialist, I want to learn how to make a HTML website so as to use Leaflet and its functionalities. Whereas this document does not provide an indepth view of all the ins and outs of a HTML document, HTML websites are made up of elements known as `tags`. Tags, normally indicated by angle brackets (`<>`) are what introduce any form of content into a webpage, be it a paragraph (`<p>`), an image (`<img>`), video (`<video>`) and even an entire

section (`<div>`, `<section>`, `<article>`). With this basic introduction, let's create a basic HTML page.

To create a HTML element along with many other programming files, such as `.js` and `.css` which we shall see later, we use a text editor. A good example of a text editor is VS code or Pycharm. Check their websites on their installation methods for your personal computer. For creating HTML and working with `.js` documents later, we shall use VS Code unless otherwise stated.

Here is a basic HTML webpage.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>A basic html webpage</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <div id="division-1">
            <p>Hello, World!</p>
        </div>
        <script src="main.js">

        </script>

    </body>
</html>
```

Let's go through each of the above tags one by one.

1. `<!DOCTYPE html>` - It is an "information" to the browser about what document type to expect.

2. `<html lang="en">` - It is the container for all other HTML elements (except for the `<!DOCTYPE>` tag). The `lang` attribute is used to assist web engines know which language the website uses.

3. `<head>` - It is not displayed on the webpage as other tags, but contains the metadata of the webpage.

4. `<title>` - Can you guess? You had it right. Defines the title of the document. In our case, if you open the webpage assuming you created it in VS Code, the webpage shall be titled *A basic html webpage* at the tab of your web-browser.

5. `<meta charset="utf-8">` - This is part of the metadata hosted by the `<head>` tag. We had mentioned earlier that the `<head>` contains the metadata of the webpage. Now here we would like to add that the `<meta>` tag found *within* the `<head>` is what *defines* the metadata. You can think of it as **README** text file that comes with any software you download. The `<meta>` tag in our case defines the encoding of our HTML5 document with the attribute `charset="utf-8"`. Don't think about this too much. HTML5 documents have `utf-8` as their encoding. You can try to look up what encoding is but it's not useful for this tutorial!

6. `<link>` - Defines the relationship between a document and an external resource. It has various attributes but `rel` and `href` have been used. The former specifies the relationship between the current document and the linked document/resource. The `rel` here references the `styles.css` file as the style sheet for our HTML. That is, the styles for our HTML are found in the `styles.css` file. `href` on the other hand points the HTML document to the path of the stylesheet –the `styles.css` file.

7. `<body>` - This is the crux of your webpage. If nothing is within the `<body>` tags, your webpage will be as empty as a blank sheet of paper. This tag is the home for all the other contents of the webpage such as headings, paragraphs, images, tables etc.

8. `<div>` - This is a special element that lets you group similar sets of content together on a web page. You can use it as a generic container for associating similar content. In the above HTML script, we have included an `<id>` attribute that is in other words, a unique identifier for this section of the webpage. `<id>s` are useful if you want to customize the appearance of a certain part of the webpage. `<class>`es behave in a similar way, but the difference between `<id>` and `<class>` is that `<id>` has to be unique, while `<class>`es can be used more than once.

9.`<script>` - It is used to embed executable code or data. In most cases it refers to JavaScript, which enhances interactivity.

If you may have noticed above, most HTML tags end with `</name-of-tag>`. With a few exceptions such as `<img>`, almost all HTML tags end this way.

1.3 JavaScript

JavaScript, shortened to `.js` is the language of the web. It introduces interactivity to HTML files. Without it our HTML files would just remain static. Have you ever clicked a link or a shiny button on a website and some visual or menu popped up? JavaScript was the engine behind all that. Think of `.js` as the life of the party while HTML is just the setting. Without `.js`, creating

webmaps would not be possible since adding JavaScript code to a HTML file using `<script>` is what makes the map appear on any website!

1.4 CSS files

CSS stands for *Cascading Style Sheet*. The CSS defines how your HTML is to appear, such as color and size of text, background color of the HTML as well as the structure of your HTML page.

CSS is quite a huge field despite appearing simple to the novice's eye. However, the html elements of a webpage are accompanied by a curly bracket containing the specified properties and values. The CSS terms 'Properties' and 'Values' are described below.

- Properties: These are human-readable identifiers that indicate which stylistic features you want to modify. For example, font-size, width, background-color.
- Values: Each property is assigned a value. This value indicates how to style the property.

Using the example of our ProGMO website, this is how we would specify the font and color of the `<body>` element of our webpage. In some cases, the property values in CSS elements can be more than one, as in `font-family` below.

```
body {
  font-family: "Open Sans", sans-serif;
  color: #444444;
}
```

The `body` in the CSS file is known as the selector. Selectors in CSS are what tags are to HTML files. However, selectors can be more specific, such as specifying the exact `<div>` that should be displayed in a particular way. Using our HTML file example, if there were other `<div>`s apart from the `<div id="division-1">` above, we would specify the one with ID `division-1` in a CSS document like so:

```
#division-1 {
  font-family: "Open Sans", sans-serif;
  color: #343a40;
}
```

We specify all specific IDs in a HTML file with a prefix of `#`. Suppose there was a `<div id="division-2">` somewhere in the HTML. We would similarly define some properties specific to it in the manner below:

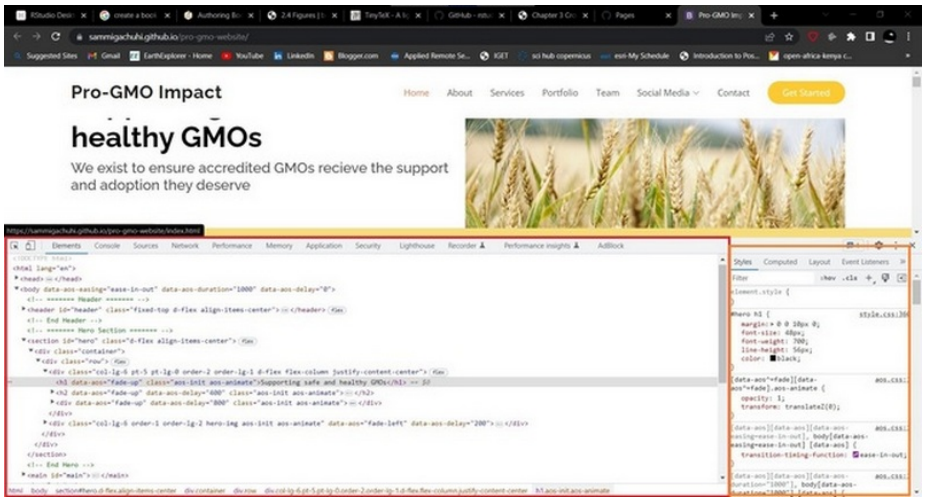
```
#division-2 {
  font-family: helvetica;
  color: #000000;
}
```

For `<classes>` and they can be several, we select each particular class using the convention:

```
.class_name {
property: value
prperty2: value2}
```

You can view the style of a particular HTML element using the styles tab found in **Element** tab of the inspect console. To get to the **Inspect** tab of your browser, right click the webpage and select **Inspect** from the list of options provided.

```
knitr::include_graphics(rep("elements2.jpg"))
```



The MDN website provides a lot of information on HTML and CSS.

1.5 Summary

This chapter was an introduction to Hyper Text Markup Language (HTML), JavaScript and Cascading Style Sheets (CSS) languages. You learnt the following:

- You can work with Leaflet in either a HTML or JavaScript file. In HTML, the JavaScript code must appear under the `<script>` tag.
- HTML files are made up of elements called tags. Tags are features that introduce any form of content into a webpage.
- JavaScript is the main language of the web. It is the language responsible for the interactivity in most websites.
- CSS stands for Cascading Style Sheet (CSS). CSS defines how your HTML is to appear, such as color and size of text, background color and even the structure of your HTML page.

Chapter 2

First Leaflet Map

2.1 Setting the superstructure

We had earlier mentioned that Javascript is the life of the party when it comes to webpages. In other words, it moves your web pages from a static sate to being interactive. You can think of it like the energy drink that makes an exhausted athlete want to do one more run.

Creating a Leaflet map is not like creating any other HTML web page. You have to set up the Leaflet essentials in your HTML page first. To begin with, create a new HTML document called `map.html`. This will be the HTML document that will act as the structure which will house our webpage to be created using JavaScript. Using VS Code, create `map.html` and paste, or preferably, type the following code.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Leaflet Maps</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="styles.css">
        <link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.3/dist/leaflet.css"
     integrity="sha256-kLaT2GOSpHechhsozzB+flnD+zUyjE2LlfWPgU04xyI="
     crossorigin=""/>
        <script src="https://unpkg.com/leaflet@1.9.3/dist/leaflet.js"
        integrity="sha256-WBkoXOwTeyKclOHuWtc+i2uENFpDZ9YPdf5Hf+D7ewM="
        crossorigin=""></script>
    </head>
    <body>
        <div id="myMap">
```

```
            <script src="main.js">
            </script>
        </div>

    </body>
</html>
```

You may be wondering why we have two `<link>` and `<script>` tags. When a browser reads a HTML script, it reads it from top to bottom. In our html script, the browser will apply the styles defined in `styles.css` to the HTML elements. To make matters clearer, the following script is what is contained in the `styles.css`:

```
#myMap {
    height: 600px;
}
```

The CSS values, those within the curly brackets, will be responsible for making our map canvas have a height of 600px on the browser.

Now to the two `<script>` tags. One refers to the online JavaScript library. The `src` attribute is in fact linking to a webpage as you can see from the protocol– `https://...`. The second, housed under the `<div>` tag, references our local JavaScript file which shall contain all the code to transform our HTML page to a webmap ninja –lines, polygons and other cool stuff.

2.2 Beautifying the house

Think of the HTML document as the superstructure, like a huge multistorey building just finished. Though the structure has the best architectural design, it just looks all grey with no life unless we call some exterior designers to add some color. That's what `main.js` file, pointed to by the `<script>` tag in the HTML document will precisely do. Through the help of JavaScript, our static HTML file will turn into an interactive map.

Open your VS Code, and assuming you had already created `main.js` already, (if not, create one now), insert the following code into the `.js` file.

```
var map = L.map('myMap').setView([-0.0884105,34.7299038], 13);
```

Take a pause.

Breath in, breath out.

You are just about to learn something very important here. In fact, it is the crux of what makes Leaflet work. Your understanding of leaflet hinges on the small code above.

The `L.map()` class we just used is what initializes the Leaflet map. Everything within the `<div>` is displayed thanks to this class function. It is referred to as a factory function because it uses the method `map` to return an object.

The `setView` method *sets the view of the map (geographical center and zoom) with the given animation options*. It's properties are Latitude-Longitude, zoom number and other options. All Leaflet methods and functions are explained in the [Leaflet reference. Not that we have inserted at Latitude, Longitude and zoom level respectively within the `setView` method.

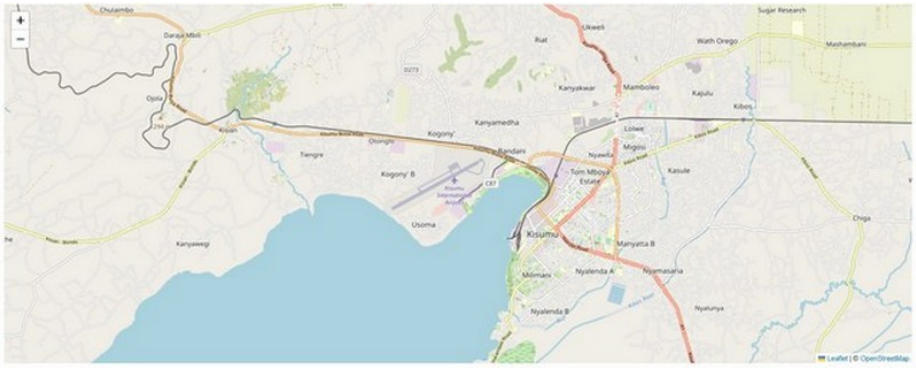
Try loading your `map.html`. All you see is a grey canvas with zoom options. This is because we haven't added a tilelayer yet. A tileLayer is a set of web-accessible tiles that reside on a server. A tile consists individual images or vector files from a server which are collectively joined together to form a webmap. If you've zoomed into a webmap, say Google Maps and noticed boxes appearing as you zoomed in or out, those are *tiles*.

Let's load an example of a common tile layer–the Open StreetMap–into Leaflet.

```
L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
}).addTo(map);
```

Reload your HTML page again. You should see an web map like the one below.

```
knitr::include_graphics(rep("kisumu-leaflet.jpg"))
```



What `L.tileLayer` has done is retrieve the web tiles from the Uniform Resource Locator (url) source provided, and within the dictionary that follows the url,

zoom level (`maxZoom`) and map attribution (`attribution`) have been provided. When working with Leaflet, the dictionary, indicated by the curly braces `{}` houses most of the additional class options other than the key one(s). In this case we used the additional options of `maxZoom` and `attribution`. Finally, the method `addTo` adds the layer to the given map or layer group. Here, our webtile is added to the `var map` which only contains the `setView` properties.

A very influential person said Kisumu located in Kenya is a town with great potential. How about dispalying it to the whole world to realise it!

2.3 Summary

In this chapter, we created our first Leaflet map. Here are a couple of things that you have learnt at the first step of this web mapping journey.

- Browsers read code scripts from top to bottom, much like skimming down a page.
- The styles defined in the CSS style sheet, the `styles.css` in this case, will apply to the HTML elements in the `map.html` file.
- We use the `src` attribute to link a different file other than HTML, such as a JavaScript file, to your HTML file. For example, the online script `leaflet.js` is connected to the HTML file through the `src` attribute of the `<script>` to enable the HTML file execute Leaflet functionalities such as web map creation, rendering and controls.
- If HTML is the magnificent building, JavaScript acts like a good exterior designer.
- The `L.map()` class is what initializes the Leaflet map.

Chapter 3

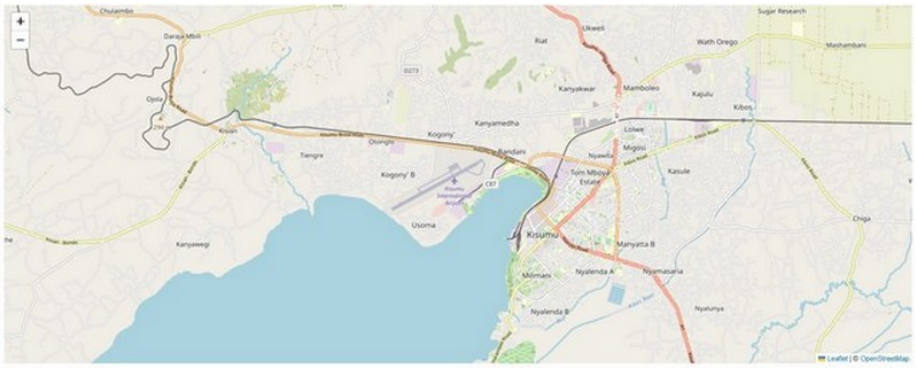
Add ons

3.1 Not just a plain map

Like in the ultimate finale of a series where the episode begins with the statement- "Previously on…", this chapter shall be a continuation of Chapter 2.

So we have a plain looking webmap like the one shown below.

```
knitr::include_graphics(rep("kisumu-leaflet.jpg"))
```



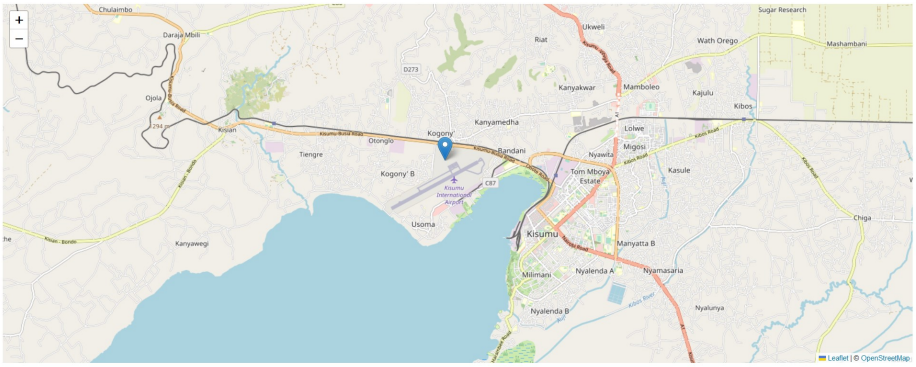
However, despite being a cool looking webmap, it offers no other additional information to the user. In order to pass some information, such as showing the location of Kisumu and *inter alia*, markers are one way of displaying such content. As a side note, there are other basemap layer servers compatible with Leaflet available here.

3.2 A marker

Many people could possibly hardly know where Kisumu, is, so lets indicate its location with a simple pin marker. To be more specific, let's pinpoint Kisumu International Airport.

```
// Location of Kisumu International Airport
var marker = L.marker([-0.0819301, 34.7260167]).addTo(map);
```

```
knitr::include_graphics(rep("kisumu-international-airport.jpg"))
```



As a simple exercise, can you try creating a marker for your home location using the `setView` method you have learned about so far?

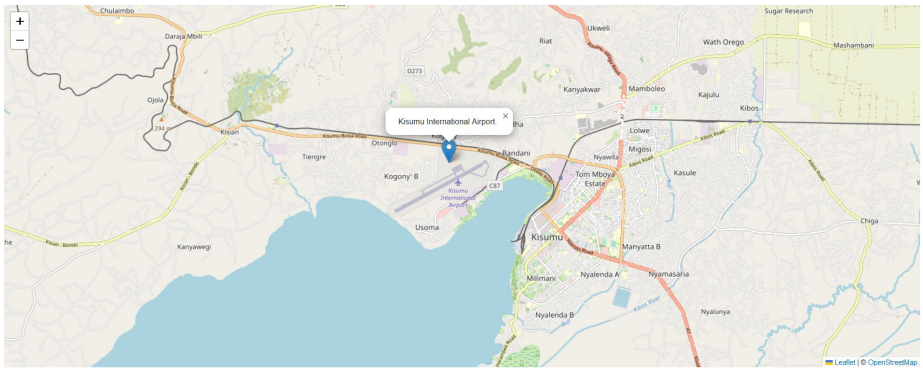
Alright, we have a marker. But what's so special about it apart from it being a lone pin in the middle of somewhere? Let's try to make this marker have some information, otherwise called attributes in GIS. Let's say the attributes we want to add are the name of the airport and other auxillary data.

3.3 A marker with a popup

To create popups, Leaflet provides the `bindPopup` method. You just *chain* it to the variable, more like how you would add an extension to a browser to perform new functions but now in this case, these extra functions are added to the variable. In the below code, `bindPopup` is chained to the `marker` variable using a dot `.`.

```
// Create popup of Kisumu international Airport
marker.bindPopup("Kisumu International Airport.").openPopup();
```

```
knitr::include_graphics(rep("kisumu-airport-popup.jpg"))
```

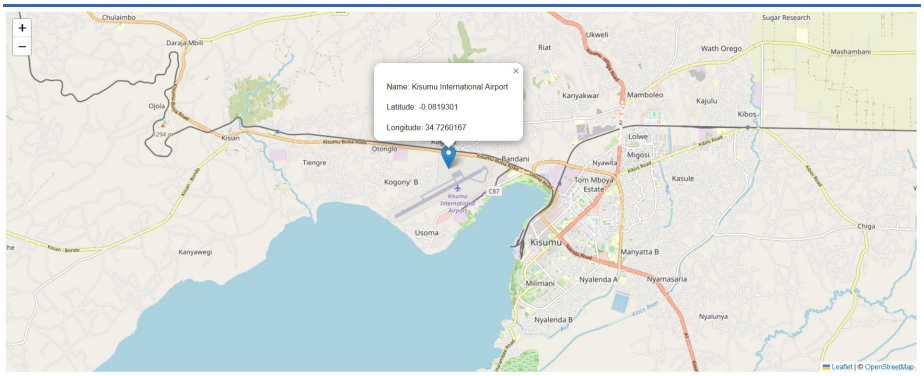


What has just happened is that `bindPopup` binds the popup content– "Kisumu International Airport" to the marker. In the below code, we have added another method, `openPopup` which *open* the popup at that specified latitude longitude. If you remove, or comment `//` out the `popUp` method, you will have to click the marker to see the popup content. Try it out.

Markers can also work with HTML elements, such as when you want to display additional metadata, say the owner of the place, size of land et cetera. In the below case, we have added the lat-lon coordinates of Kisumu airport location. It is highly advised not to include lengthy information in an HTML marker element.

```
// With html content
marker.bindPopup("<br>Name: Kisumu International Airport</br><br>Latitude: -0.0819301</br><br>Lor
```

```
knitr::include_graphics(rep("marker-html.jpg"))
```



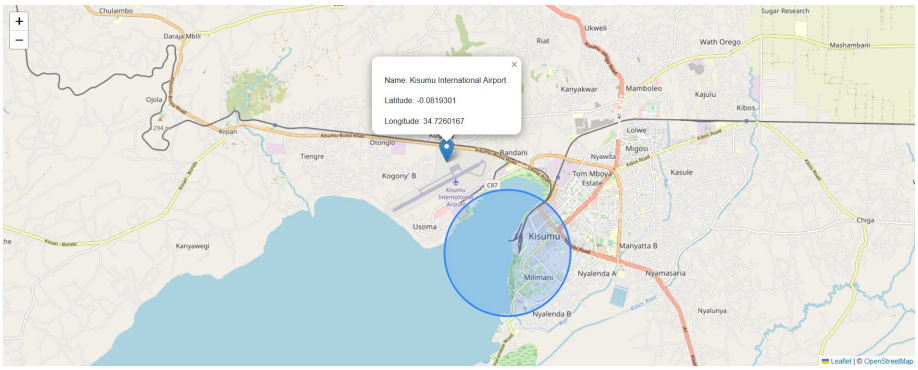
3.4 Different markers and popups

So far you have seen pin markers, but there are also other kinds of markers, such as circles and rectangles. Unlike the pin markers we have been experimenting with, these other markers require additional options, such as radius value for circle and lat-long coordinates for rectangles. Let's have a go with each type.

Starting with a circle, let's start by drawing a radius around the location of Kisumu Museum.

```
// Circle over Kisumu Museum
var circle = L.circle([-0.107637, 34.7435975]).setRadius(2000).addTo(map);
```

```
knitr::include_graphics(rep("kisumu-museum-circle.jpg"))
```



The below code will also create a slightly similar circle marker, the only difference is that in the preceding one we didn't insert `{options}` and we set radius using the `setRadius` method. In the second one below, we have been very specific in what we want –our specifications going into the curly brackets {} before eventually adding the circle marker to our map. Brackets in JavaScript indicate you are dealing with a dictionary. A dictionary in JavaScript and even in Python is used to denote key-value pairs.

```
var circle = L.circle([-0.107637, 34.7435975], {
    color: 'blue',
    fillColor: 'blue',
    fillOpacity: .5,
    radius: 2000
}).addTo(map);
```

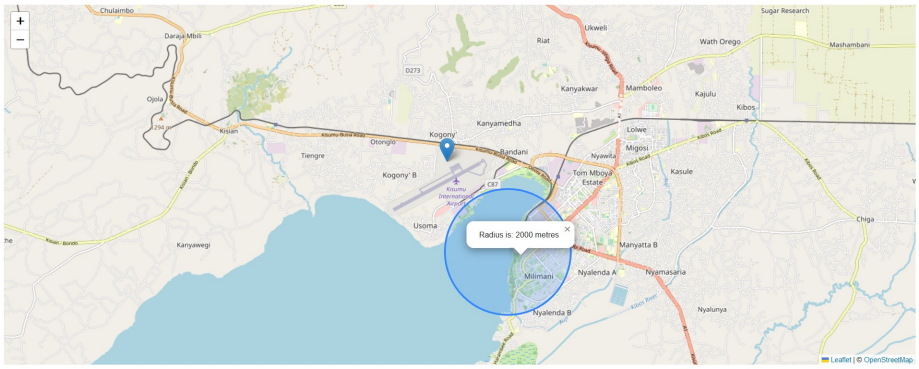
As we had mentioned earlier, other marker elements such as circles and rectangles can have popups attached to them. Ready for something cool? We will

attach a popup into our circle. Not just any other ordinary hard coded popup but one which relies on other Leaflet JavaScript methods to generate an output. In our case, we want a pop up that shows the radius of our circle, without us typing it out into the code.

```
// Circle marker pop up for Kisumu Museum
var getRadius = circle.getRadius();
circle.bindPopup("Radius is: " + getRadius.toString() + " metres");
```

In our above code, `getRadius` gets the radius of our circle marker. `bindPopup` as has already been explained before *binds* the popup content to our circle marker. But there is a catch. The variable `getRadius` is used to print out the results, which is 2000 of course. However, `bindPopup` only understands strings so we convert our variable result to a string using `toString()`. We also added other strings to give the popup a wholesome result that is understandable to every Tom, Dick, Harry and Harriet.

```
knitr::include_graphics(rep("circle-radius.jpg"))
```



Finally, let's try with a rectangle. Actually, Leaflet allows us to create polygons. Let's work with the polygon class to create a rectangle bounding a given location.

Copy the following coordinates.

```
// Draw rectangle around Kisumu Wildlife Impala Park
var impalaParkCoordinates = [
    [-0.1144753, 34.743418],
    [-0.115097, 34.745242],
    [-0.114238, 34.745071],
    [-0.114002, 34.746101],
    [-0.115054, 34.746787],
    [-0.115998, 34.745586],
    [-0.118444, 34.746208],
```

```
    [-0.121255, 34.744684]
]
```

Now using the `L.polygon` class and a few optional parameters, let's showcase where the Kisumu Wildlife Impala Park is situated.

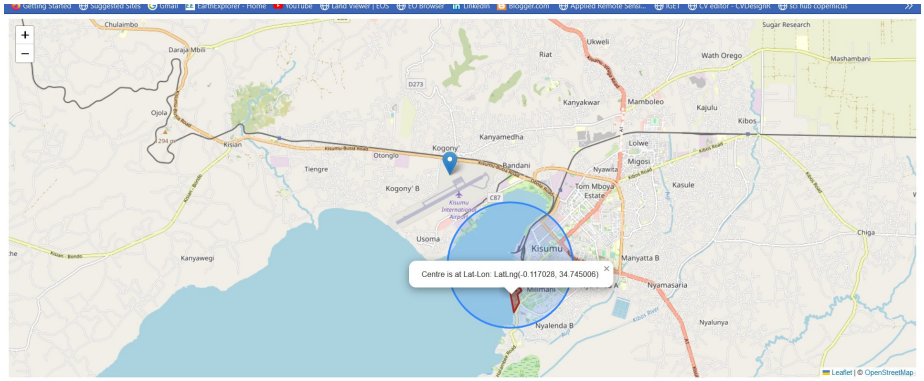
```
// Create a polygon using the above coordinates
var impalaParkPolygon = L.polygon(impalaParkCoordinates, {
    color: 'brown',
    fillOpacity: 0.4
}).addTo(map);
```

Just like we did for the circle marker, we will make our popup content rely on another variable, in this case `getCenter` which gets the centroid coordinates of our polygon. We were looking for something cooler such as `getArea` in Leaflet, one that automatically prints out the area of a polygon in a popup. Unfortunately, we were unable to find it.

```
// Add popup to the polygon of Kisumu Impala Park
var getCenter = impalaParkPolygon.getCenter();
impalaParkPolygon.bindPopup("Centre is at Lat-Lon: " + getCenter.toString()).openPopup
```

If you find the circle marker too much of an obstraction to the rectangle marker, feel free to comment it out using `//`.

```
knitr::include_graphics(rep("polygon-marker.jpg"))
```



You can get the files used in this exercise here.

3.5 Summary

This chapter took you further in enriching the content that can be displayed in a webmap. You have seen that a webmap can offer far more useful information than just mere markers and symbols on a web canvas. Popups are one way of displaying information, and they too can be customized further. Through the practicals in this chapter, you have learnt the following:

- To create popups in Leaflet, we use the `bindPopup` method.
- `openPopup` automatically opens the popups once the Leaflet map is loaded. They only disappear once you close them.
- Markers can also work with HTML elements.
- Apart from location pins, markers can also be circles and rectangles.
- There exist methods in Leaflet that can automatically parse out information in popups without requiring any hardcoding from the programmer. For example, we used `getRadius` to display the circle radius in the popup without necessarily typing it out in the `bindPopup` method.
- You comment out JavaScript code with `//`!

Chapter 4

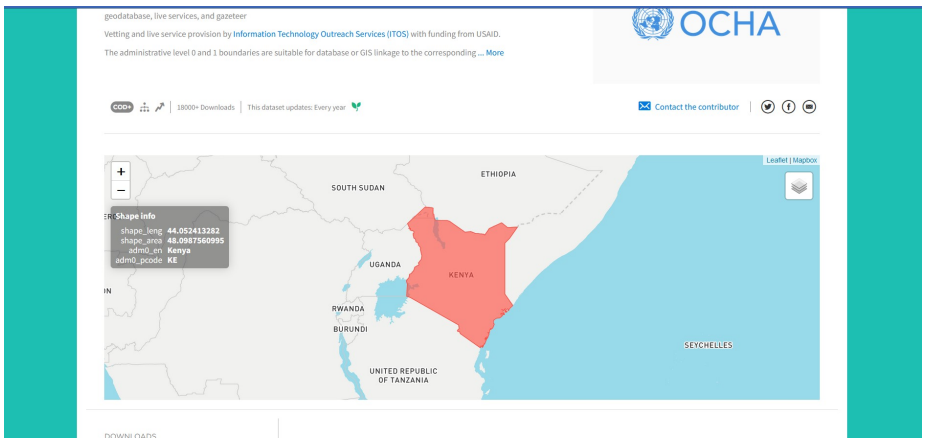
Embedding leaflet map to an external website

4.1 A website with a sense of direction

Now, we have succeeded in making a stand alone leaflet map. However, we want to do something that will quickly upscale you from a novice to a pro. That is, embeding a Leaflet map into a website.

An example of what we want is shown below, which is a snapshot from the HDX website.

```
knitr::include_graphics(rep("webmap-in-web.jpg"))
```



For this exercise, we shall embed a leaflet map to a simple HTML webpage. This

webpage doesn't look grand, but it serves the purpose of our exercise. Let's get on to it. Here are the files.

4.2 The HTML webpage

Create a HTML page with the following code.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Pro-GMO Alliance</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="example-styles.css">
        <link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.3/dist/leaflet.css"
        integrity="sha256-kLaT2GOSpHechhsozzB+flnD+zUyjE2LlfWPgU04xyI="
        crossorigin=""/>
        <script src="https://unpkg.com/leaflet@1.9.3/dist/leaflet.js"
        integrity="sha256-WBkoXOwTeyKclOHuWtc+i2uENFpDZ9YPdf5Hf+D7ewM="
        crossorigin=""></script>
    </head>
    <body>
    <div id="div-for-article">
        <article id="introduction">
            <h2>Introduction</h2>
            <q>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusanti
            doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore
            quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam volupta
            sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos
            voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia
            consectetur, adipisci velit, sed quia non numquam eius modi tempora incidun
            dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis no
            ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequat
            vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molest
            vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?</q>
        </article>
    </div>
    <div id="div-for-section">
        <section id="Products">
            <div class="row">
                <h2>Our Products</h2>
                <div class="column">
                  <img src="https://images.unsplash.com/photo-1632125941710-35a9d2fcc7
                </div>
                <div class="column">
```

```
                    <img src="https://images.unsplash.com/photo-1595615636850-3292eb0a95b0?ixlib=
                  </div>
                <div class="column">
                  <img src="https://images.unsplash.com/photo-1600333859399-228aa03f7dba?ixlib=rb
                </div>
                <div class="column">
                  <img src="https://images.unsplash.com/photo-1630145398476-853543b02843?ixlib=rb
                </div>
              </div>
        </section>
    </div>
    <div>
        <br>
        <h2>Our Branches</h2>
        <br>
    </div>
      <div class="container">
        <div id="map">
          <script src="example-main.js"></script>
        </div>
        <div class="text">
          <h1>Address</h1>
          <p>
            P.O. Box 55044, Nakuru
        </p>
        </div>
      </div>
    </body>
</html>
```

Since this is a geospatial book, we shall not go through every line of the HTML script above. It just a webpage containing some text, some pictures and a webmap. The webmap is the centre of our interest in this chapter. We at least do know how to create a Leaflet map, but how do we fit it inside a webpage?

Before we head there, let's insert the CSS file, which looks like this.

```
/* Three image containers (use 25% for four, and 50% for two, etc) */
.column {
    float: left;
    width: 33.33%;
    padding: 5px;
  }

  /* Clear floats after image containers */
  .row::after {
```

```
    content: "";
    clear: both;
    display: table;
  }


/* Styling the map */
  .container {
    display: flex;
    align-items: center;
    justify-content: center
  }

#map {
    height: 300px;
    width: 90%
}

.text {
    font-size: 15px;
    padding-left: 20px;
}
```

4.3 A simple `for` loop for webmap display

Back to the Leaflet map of our dummy Pro-GMO Alliance webpage. How did we put the Leaflet in there? In just under a minute, parsing the JavaScript Leaflet file to the `<script>` tag and referencing it using the `src` attribute makes our webmap appear at its placed position in the HTML file. The JavaScript file we parsed to our HTML file is called `example-main.js`. It contains the following code:

```
var map = L.map('map').setView([-0.302765, 36.146147], 12);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMa
}).addTo(map);

var branches = [
    ["Potatoes",-0.328858, 36.008474],
    ["Maize",-0.302765, 36.146147],
    ["Sunflower",-0.224832, 36.159880],
    ["Cotton", -0.214189, 36.135847]
```

```
    ];

for (var i = 0; i < branches.length; i++) {
    marker = new L.marker([branches[i][1], branches[i][2]])
        .bindPopup(branches[i][0])
        .addTo(map);
}
```

For the first time in this book, we are introducing the `for` loop. As is the case in other languages such as R and Python, JavaScript also uses the `for` loop to iterate over items. In our case, and remembering that indexing in arrays begins from 0, the marker popups will read the latitudes and longitudes which are at index 1 and 2 respectively. The lat-lon are indicated by `([branches[i][1], branches[i][2]])`. The popup strings, which appear as the first elements in the `branches` array, are at index 0. The popup strings are indicated by `branches[i][0]`. At the end of the chain the markers are added to the map with `.addTo`.

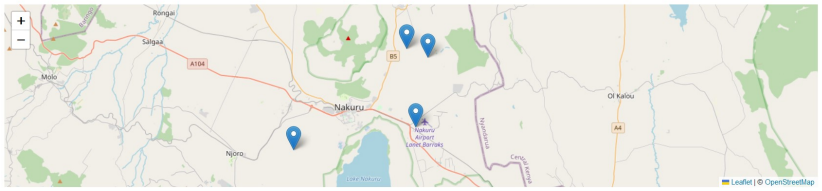
Alright. How about the keyword `new`?

The `new` keyword is a constructor. That is, it creates an empty object. Many instances of the variable `marker` can be created from the instance of `new` object. For more information on the `new` keyword constructor, see this website. Be rest assured that the `new` keyword isn't mandatory to make the webmap to work in our case but its helpful to learn another JavaScript trick to add to your hat.

Below is a snapshot of how the dummy Pro-GMO website looks like with our newly embedded webmap.

```
knitr::include_graphics(rep("embedded.jpg"))
```

Our Branches



Address

P.O. Box 55044,
Nakuru

"How are we able to align the webmap to the left and also make other text stand aside to it?" This is all thanks to the CSS property `display: flex`. `display` is a CSS property that deals with how HTML elements are displayed.

The `display` property aligns a HTML element to fill or shrink according to the space available within its assigned portion in the webpage. On the other hand, the CSS property `padding-left` just creates space all around the element, thus creating a neat space between the Leaflet map and the address text. Removing this will just make the address text and the Leaflet map touch each other edge to edge. The inspiration to use all these CSS properties and values in placing HTML elements side-by-side emanated from this example.

Having done the above, you can consider you are as good a Leaflet mapper to undertake any task! Later on, in Chapter 13, we shall see how to insert a Leaflet map in a more sophisticated website. This was just a gentle introduction.

4.4 Summary

This chapter has introduced you on how you can use CSS to customize the appearance and positioning of your webmap. You have also encountered the use of `for` loop in JavaScript code to retrieve geospatial information, particularly from arrays. Here are some of the take aways from this chapter.

- Leaflet maps can be embedded inside a website as demonstrated in the dummy Pro-GMO webpage.
- One can use `for` loops to iterate over elements from an array and retrieve geospatial information. In this chapter, the `for` loop was used to retrieve both latitude-longitude coordinates and text from the `branches` array variable.
- We can use CSS elements, such as `display` and `padding-left` to position and define how a webmap shall be displayed on our webpage.

Chapter 5

Using GeoJSON in Leaflet

5.1 Creating a `.geojson` file

So far, we have created a Leaflet map, added some aesthetics such as markers, and even embedded a map into a dummy website. Alright, the website wasn't even close to good, but the methodology should be the same when working with other fully functional and better looking websites. That may be enough to give you confidence to start as a webmapper but not so fast! There is still more territory to cover! I would like to introduce another format of storing geospatial information –the use of `.geojson` files.

5.2 What are `.geojson` files?

`.geojson` files, according to the GIS leader ESRI, are an open standard geospatial data interchange format that represents simple geographic features and their non-spatial attributes. GeoJSON is based on the JavaScript Object Notation (JSON) file format which is a lightweight data exchange format that is easily interpretable by both man and machine. In very few instances does any data format please both sides of the divide but JSON does, and this site provides examples. Anyway, just like you can tell from the name, JSON is based on the JavaScript programming language. If you have worked with JavaScript before, it looks very much like a data format based on dictionaries. In essence, JSON is a large dictionary holding other *dictionaries* of data within it. A GeoJSON is a JSON file that follows a certain structure and has spatial index and geometry specifications in it. See this website for a GeoJSON example.

An example of a GeoJSON file format structure is shown below:

```
{
```

```
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "City": "Nairobi",
        "Population": "4, 300, 000"
      },
      "geometry": {
        "coordinates": [
          36.80617598261199,
          -1.2868825246637812
        ],
        "type": "Point"
      }
    },
    {
      "type": "Feature",
      "properties": {
        "City": "Kisumu",
        "Population": "610, 082"
      },
      "geometry": {
        "coordinates": [
          34.738718987625106,
          -0.10390483386935045
        ],
        "type": "Point"
      }
    },
    ---snip----
```

Below is an example of a json file structure.

```
{

"Influencers" :   [
{
 "name" : "Jaxon",
 "age" : 42,
 "Works At" : "Tech News"
}

{
 "name" : "Miller",
```

```
 "age" : 35
 "Works At" : "IT Day"
}

]
}
```

5.3 Why geojson?

Yours truly could be wrong, but one advantage of `geojson` and `json` is that it's minimal on size compared to shapefiles and it is also more portable. Shapefiles are dependent on other data formats that accompany it, such as `.shx`, `.dbf`, `.prj` which provide auxillary geospatial orientation, metadata, and attributes. On the other hand, GeoJSON and JSON formats will come as single files but will still as much data as all the components of a shapefile (`.shx`, `.dbf`, and `.prj`) put together.

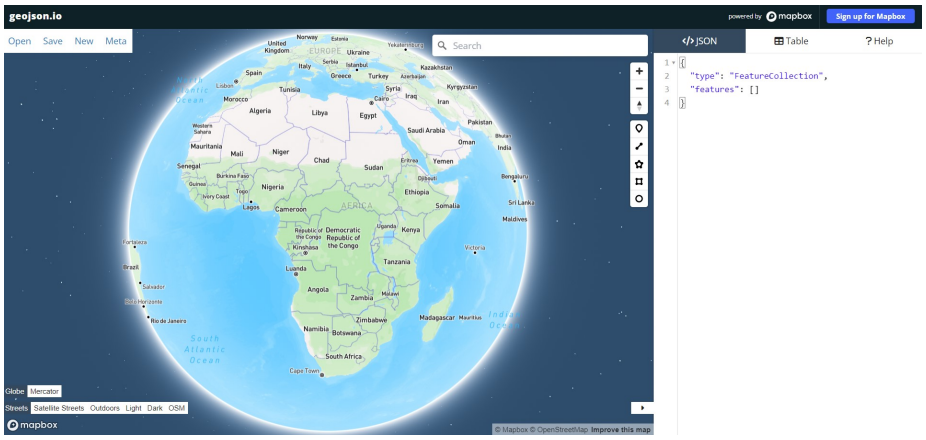
So when do I use shapefiles vis a vis GeoJson? If you want to work with geospatial data in a web interface, `geojson` is the way to go. Period.

5.4 Creating a geojson file

It may look intimidating to create a GeoJSON file without hardly making any errors, and it actually is, but luckily the geojson.io website does the heavy lifting for us. We shall head over to it and create a `geojson` file of some cities and their population.

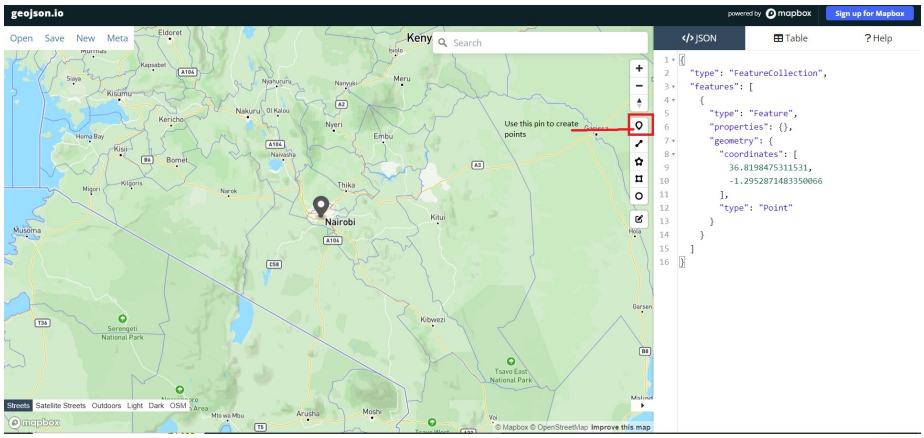
First, of all, the website looks like below. Talk of a cool global map powered by Mapbox.

```
knitr::include_graphics(rep("mapbox-front.jpg"))
```



On your right, under the **</>JSON** tab, the Mapbox folks have already given you a head start by indicating what feature type and features will go into your `geojson` file. These are important as any website uses these keywords when parsing information from the GeoJSON file. Zoom to Kenya and click a point on top of the Nairobi dot pin, like shown below. Use the highlighted pin in the image below to create a marker over Nairobi.

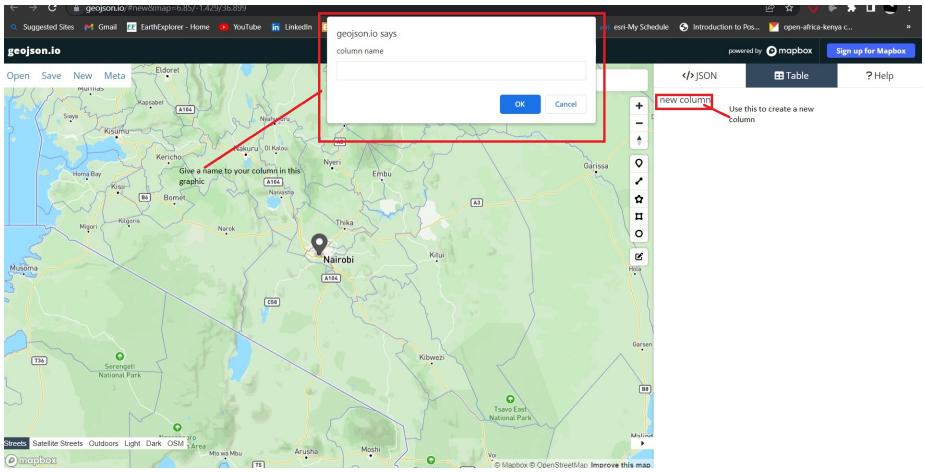
```
knitr::include_graphics(rep("nairobi-pin.jpg"))
```



By doing so, you will realize that a new dictionary of `type`, `properties` and `geometry` appears within the **features** list. These new keys provide the additional spatial and geometry data in their values which a website uses to place them at their appropriate locations on the webmap.

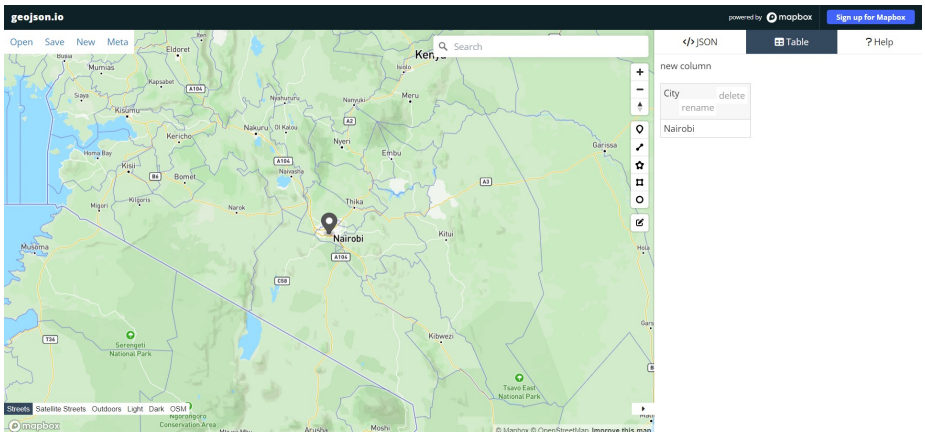
Now move over to the **Table** tab and click new column as shown below.

```
knitr::include_graphics(rep("table.jpg"))
```



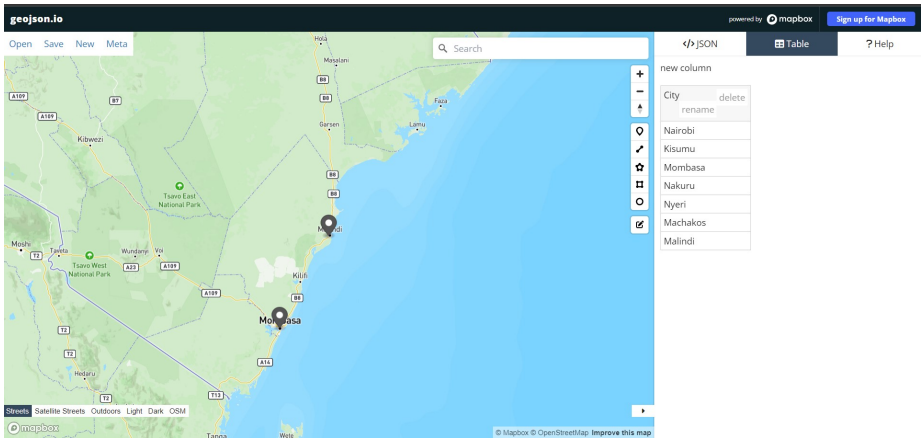
Click on it and in the graphic that appears, give your column the name **City**. Click **Ok** and in the table cell that appears, type **Nairobi**.

```
knitr::include_graphics(rep("nairobi-named.jpg"))
```



Now create a new pin over *Kisumu* and two things will happen: a new dictionary will appear below that of Nairobi and a new table row will appear in the **Table** tab. Create pins over the following cities: Mombasa, Nakuru, Nyeri, Machakos and Malindi. Legally speaking, only the first four are cities by law, the rest are just towns but for the sake of this tutorial, let's corporately refer to them as cities.

```
knitr::include_graphics(rep("cities-named.jpg"))
```



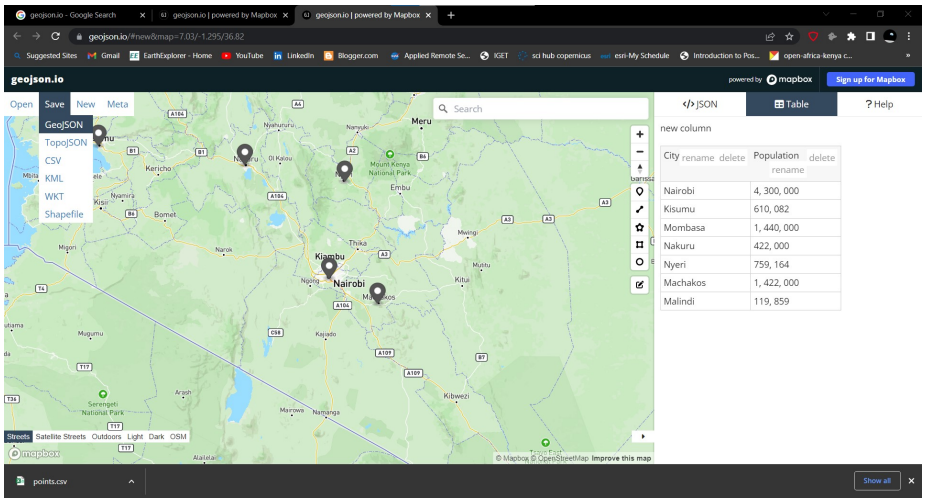
Alright, one process down, one more to go. We will fill these cities with their population statistics. Create a new column with the heading **Population**. Fill each of the cities with the following statistics.

```
read.csv("data/points.csv")[ , 1:2]
```

```
##       City  Population
## 1  Nairobi 4, 300, 000
## 2   Kisumu    610, 082
## 3  Mombasa 1, 440, 000
## 4   Nakuru    422, 000
## 5    Nyeri    759, 164
## 6 Machakos 1, 422, 000
## 7  Malindi    119, 859
```

Once done, head over to the top left of the geojson.io website, and click **Save**. A list of options will appear, click on save as **geojson**. It should appear somewhere in your *Downloads* directory.

```
knitr::include_graphics(rep("geojson-save.jpg"))
```



5.5 Saving the Geojson to Github

Now, based on experience, loading a local Geojson file (one within your computer directory) to JavaScript is a painful, if not near impossible experience. The resulting errors had to with servers or something. To get a way around this and still be able to display `.geojson` data in Leaflet, the GeoJson file had to be stored on an online server, in this case, Github. We would love to show you how to save data on Github, but this would make this chapter way too long. Therefore, and with sincere apologies, it would be best if you googled it out. Nevertheless, here is the GeoJSON file, and it will come in handy in the last two sub-chapters of this exercise.

5.6 Loading the GeoJSON into Leaflet

As they always say, there are many ways of killing a rat. There are around three ways in which to load GeoJson data into Leaflet, at least from our discovery. We shall start with the easiest and most unreliable to what we consider the best. Let start with the easy one, loading a `.geojson` file from within our Javascript file itself.

5.6.1 The easy way

First of all create a blank JavaScript file called `geojson.js`. Thereafter, go to your `map.html` file which you had created last in Chapter 2. Open it. Within the `<script>` tag of the `<body>` element, change the `src` attribute to read `"geojson.io"` like below.

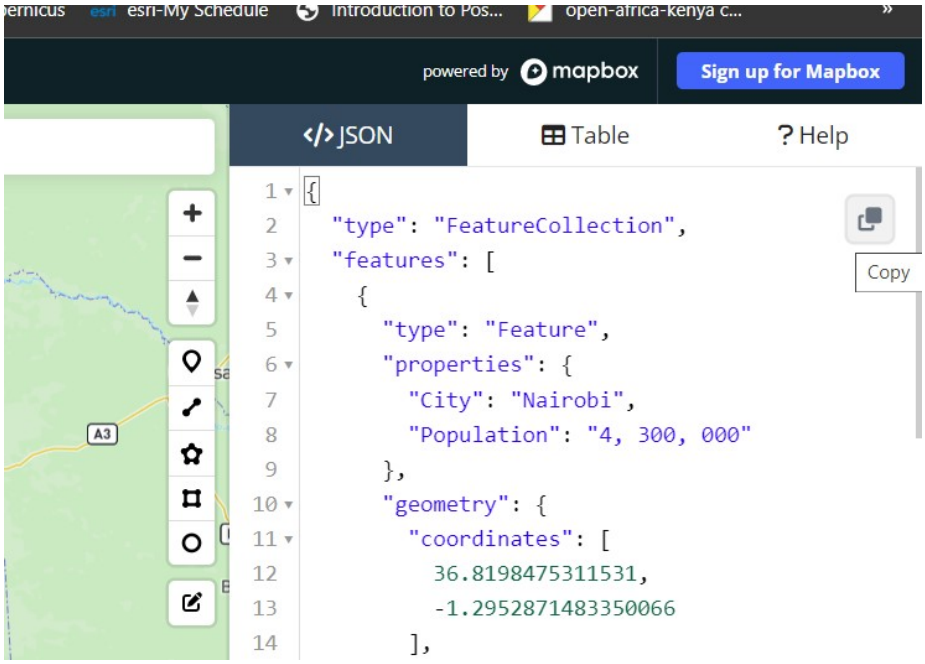
```
<!DOCTYPE html>
<html lang="en">
    <head>
        -- snip ---
    </head>
    <body>
        <div id="myMap">
            <script src="geojson.js">
            </script>
        </div>

    </body>
</html>
```

Alright. Head over to your `geojson.js` and as always, add the Leaflet classes `L.map` and `L.tileLayer`. We set the view of our new Leaflet map to that of Nairobi. Your blank `geojson.js` should now be filled with the below code.

```
var map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMa
}).addTo(map);
```

Okay. Head over to geojson.io website and right under the **</>JSON** you will see a copy icon.

```
knitr::include_graphics(rep("geojson-copy.jpg"))
```



Click it and paste the json code into your `geojson.js` file right above the other `L.map` and `L.tileLayer` classes. The contents of your `geojson.js` should look like below.

```
var cities = {
    "type": "FeatureCollection",
    "features": [
      {
        "type": "Feature",
        "properties": {
          "City": "Nairobi",
          "Population": "4, 300, 000"
        },
        "geometry": {
          "coordinates": [
            36.8198475311531,
            -1.2952871483350066
          ],
          "type": "Point"
        }
      },
      -- snip ---
      {
        "type": "Feature",
        "properties": {
```

```
          "City": "Malindi",
          "Population": "119, 859"
        },
        "geometry": {
          "coordinates": [
            40.10521499751357,
            -3.2138767356491655
          ],
          "type": "Point"
        }
      }
    ]
  }
```

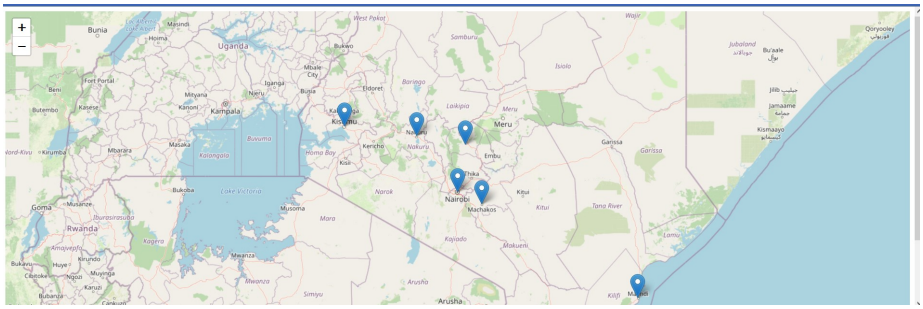
```
var map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap
}).addTo(map);
```

Refresh your `map.html`. It's a map of Kenya alright, but none of our `.geojson` features appear yet. We are about to change that. Leaflet offers the `L.geoJSON` class to add GeoJSON data to a map. The class name speaks for itself and therefore let's use it to add our GeoJSON features. Add the following code below the other Leaflet map class layers.

```
L.geoJSON(cities).addTo(map);
```

Refresh your `map.html`. The GeoJSON features should now appear at their exact locations.

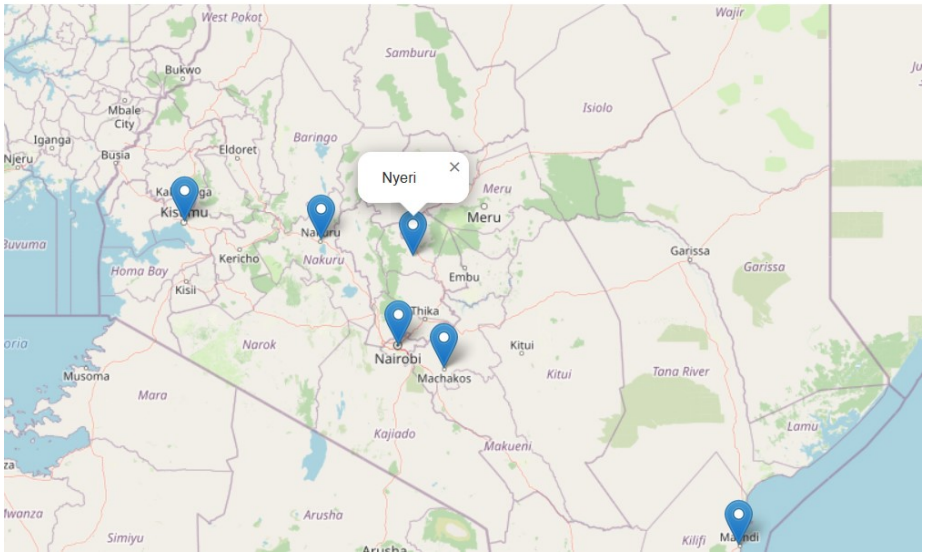
```
knitr::include_graphics(rep("geojson-leaflet.jpg"))
```



How about if we made the markers more interactive? Say, like they display popups as we did in Chapter 3? We would like the city names to appear upon a user's click on the markers. Easy. Just create a function that does so as in the logic provided here. We customized it to our case to make sure it references to the `City` key which is part of the dictionary attached to the `properties` key.

```
L.geoJSON(cities).bindPopup(function (layer) {
    return layer.feature.properties.City;
}).addTo(map);
```

```
knitr::include_graphics(rep("geojson-names.jpg"))
```



There is one issue with this method. If we have a very long GeoJSON data structure, it will clatter our JavaScript file. We only worked with seven cities, but it is very common to work with data holding hundreds and even thousands of dictionaries. That would make your JavaScript file stretch to *ad infitum*.

This brings us to the other two methods, that of using the **Ajax** plugin and using the **Fetch** Application Programming Interface (API). Don't let the words scare you. Take a break, grab a glass of water and come back.

5.6.2 Using the Ajax Plugin

As the term 'plugin' suggests, this is an extension that offers additional functions to the core Leaflet plugin. The Ajax plugin is available from this link. Download it to your directory and preferably within the same directory as your

`map.html` and `geojson.js`. Alright. Right under the `src` for `leaflet.js` in your `map.html`, add the following `<script>` tag.

```
<head>
    -- snip --
    <script src="leaflet-ajax-gh-pages\dist\leaflet.ajax.js"></script>

</head>
-- snip --
```

This file will allow you to add `.geojson` files to your Leaflet map. However, there is a catch. Both the Ajax and Fetch APIs only work with GeoJson file formats saved on a web server. Based on experience, they will not work with local GeoJson files. As a work around, we saved our GeoJSON file to Github. For convenience, here is the link again to the raw file we had initially created from geojson.io.

We shall call our GeoJSON file from Github using Ajax as shown in the code below. Please remember to comment out your `var cities` and `L.geoJSON` using `\\` because they are irrelevant in this particular case.

```
var geojsonLayer = new L.geoJson.ajax("https://raw.githubusercontent.com/sammigachuhi/g
```

Your map should now show the markers of our cities. If you are hawk eyed, you may have noticed that the syntax for Ajax is different from that of the earlier `L.geoJSON` from Leaflet. Starting with the latter, we have instead used `L.geoJson` and unlike in the Ajax creator's website where they used `var geojsonLayer = new L.GeoJSON.AJAX(<your-geojson-file>)` we used the following syntax: `L.geoJson.ajax()` (ajax and geoJson beginning with small case). Actually, that's what worked after quite a lengthy web search.

Just like using `L.geoJSON`, we can also add popups after calling the `.ajax` method.

```
var geojsonLayer = new L.geoJson.ajax("https://raw.githubusercontent.com/sammigachuhi/g
    .bindPopup(function (layer) {
        return layer.feature.properties.City;
    }).addTo(map);
```

Doing so should make the city names appear on clicking the markers on your Leaflet map.

5.6.3 Using Fetch API

Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers. In other words, it searches for a resource over the web, retrieves it, and brings it to you. Think of it as a dog in which you throw a saucer and tell your faithful hound: "*Sabretooth*, fetch!". The dog runs after the saucer, grips it with its canines mid-air and quickly brings it back to you. Same case with Fetch API!

I had mentioned I will show you how to retrieve our Github stored GeoJSON data and I shall stick to the script! I shall also attempt to explain how this *mysterious* `fetch` works.

First things first. The `fetch` function shall be called and passed to our Github url containing our GeoJSON text. Since `fetch` is an API, it retrieves data from Github.io –the server in this case–and brings it to our laptop the –the client. Enough IT. Let's write it down.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojson.geojson")
```

Okay, next step. Once the server beams back the data to us, what do we do with it? According to Digital Ocean, the response is not *actually* the data in the original format but rather *a series of methods that can be used depending on what you want to do with the information.* This sounds downright confusing. Nevertheless, it is *at least* crystal clear that the end goal is to have our fetched data in JSON format. Converting an object to JSON is done using the `json()` method. We parse `.json()` to a `then` method which is an asynchronous function[1]. After the `fetch` function is successful, the object is stored in the `response` argument, which is in brackets, and then the function returns the result as a `json` object.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojson.geojson")
    .then(function(response) {
        return response.json()
    })
```

After converting our response to JSON, it still needs to be processed further. Processed to what? To a GeoJSON file and subsequently add it to our Leaflet Map.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojson.geojson")
```

[1] Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result. See Mozilla

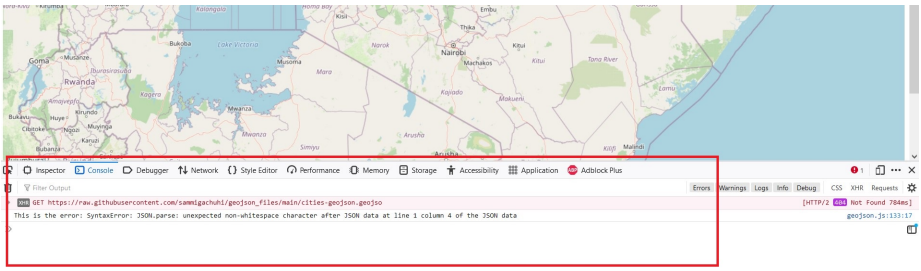
```
    .then(function(response) {
        return response.json()
    })
    .then(function(data) {
        L.geoJson(data).addTo(map);
    })
```

We shall also add one more function –the `catch()` method. `catch()` is a method that returns an action if our response to the server has been rejected. Let's demonstrate all this.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojso
    .then(function(response) {
        return response.json()
    })
    .then(function(data) {
        L.geoJson(data).addTo(map);
    })
    .catch(function(error) {
        console.log(`This is the error: ${error}`)
    })
```

To see the `catch()` in action, omit the last letter in our url so that it reads `cities-geojson.geojso`. Yeah, you read it right. Just omit the letter 'n' for now and reload your `map.html`. Right click the webmap page and click on **Inspect** in the small interface that appears. Head over to the console tab and see the error response. It should read like in the below image.

```
knitr::include_graphics(rep("catch-error.jpg"))
```



Restore the omitted letter 'n' and reload your `map.html`. Your Leaflet map should have the city markers overlain just like in the case of using Ajax plugin or hardcoding the Geojson data into `var cities`. To stretch your Javascript skills further, we can shorten our code further by retaining the arguments `response`, `data` and `error` and using the arrow function `=>` to pass on the return statements, like so:

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojson.geojson")
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        L.geoJson(data).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

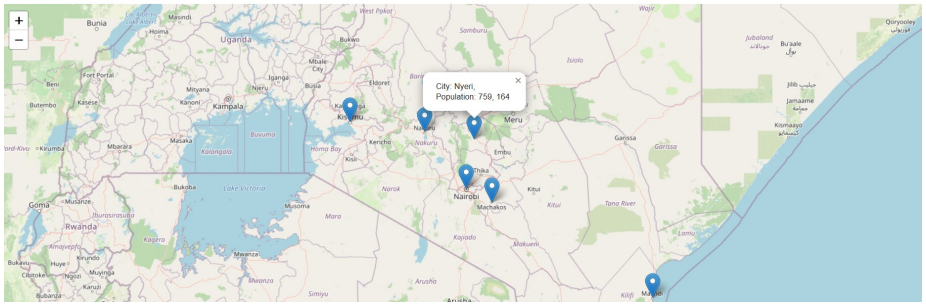
It looks cryptic but don't cry! We have only ommitted the `function()` keyword and instead added `=>` between `function()` and the curly brackets `{<code-to-run>}` which is the function body aka where the magic happens. Just like when using Ajax, we can also add other functionalities to `L.geoJSON` within the `fetch` plugin. In here, and thanks to the use of template literals (``), we can even add statements and refer to our GeoJson keys (and in some cases, even variables) using `${}`. Whatever is within the `${}` is executed and passed out as a string to the template literals.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojson.geojson")
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        L.geoJson(data).bindPopup((layer) => {
            return `City: ${layer.feature.properties.City},<br>
            Population: ${layer.feature.properties.Population}`}).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

In the `return` statement above, we added the HTML tag `<br>` to separate the `City` and `Population` keys from our GeoJSON. Doing so will return neat marker texts where the city name and population figures are in two separate lines.

Enough Javascript for a day!

```
knitr::include_graphics(rep("city-population.jpg"))
```



Here are the full files used in this chapter.

5.7 Summary

Who thought that simple GeoJSON files would be so complicated to load and see? Lets recap what we've learnt.

- GeoJSON files are an open standard geospatial data interchange format that represents simple geographic features and their nonspatial attributes.
- GeoJSON is based on the JavaScript Object Notation (JSON) file format known for its lightweight nature.
- GeoJSON files are the *lingua franca* of working with spatial information on the web.
- The geojson.io website allows a user to create a GeoJSON file interactively and intuitively.
- There are three main ways of loading a GeoJSON file in Leaflet. One is by hardcoding the dictionary of spatial attributes in a JavaScript file. This was the first method that we explored.
- The other method is by using the Ajax plugin. We call Ajax into Leaflet by inserting into the `<head>` element the path to the extracted Ajax files.
- The third and last method is through use of the `fetch` API. The `fetch` API searches for a resource over the web, retrieves it, and brings it to your computer.

Chapter 6

Create your own custom markers

6.1 Setting the base

Hope you didn't trash away the cities we created in the last chapter. In this chapter, we shall focus on creating your own custom markers. We love a clean job, so we will create a new JavaScript file and name it `custom-markers.js`. We understand the previous chapter was quite long but believe you me, although creating custom markers sounds easier, it took us way longer to get the hang around it. Good news, we received enough punches on the face to teach you how to dodge the pain points.

The very first thing to do is to create a basemap.

```
var map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
}).addTo(map);
```

We will create a new variable called `cities` that mimics the GeoJson file saved to Github only this time round the population values have been tweaked a bit. Paste the below code to your `custom-markers.js`.

```
var cities = {
    "type": "FeatureCollection",
    "features": [
```

```
{
  "type": "Feature",
  "properties": {
    "City": "Nairobi",
    "Population": 4300000
  },
  "geometry": {
    "coordinates": [
      36.8198475311531,
      -1.2952871483350066
    ],
    "type": "Point"
  }
},
{
  "type": "Feature",
  "properties": {
    "City": "Kisumu",
    "Population": 610082
  },
  "geometry": {
    "coordinates": [
      34.74657469430895,
      -0.10402992528247523
    ],
    "type": "Point"
  }
},
{
  "type": "Feature",
  "properties": {
    "City": "Mombasa",
    "Population": 1440000
  },
  "geometry": {
    "coordinates": [
      39.66358575335434,
      -4.041883912902392
    ],
    "type": "Point"
  }
},
{
  "type": "Feature",
  "properties": {
    "City": "Nakuru",
```

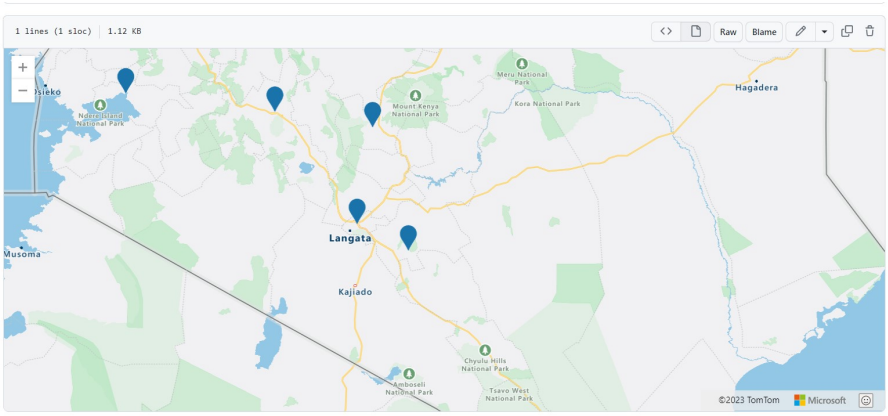
```
      "Population": 422000
    },
    "geometry": {
      "coordinates": [
        36.06412271026528,
        -0.2754534004690896
      ],
      "type": "Point"
    }
  },
  {
    "type": "Feature",
    "properties": {
      "City": "Nyeri",
      "Population": 759164
    },
    "geometry": {
      "coordinates": [
        36.957036675396154,
        -0.42345404217887506
      ],
      "type": "Point"
    }
  },
  {
    "type": "Feature",
    "properties": {
      "City": "Machakos",
      "Population": 1422000
    },
    "geometry": {
      "coordinates": [
        37.25780808801821,
        -1.518874011494134
      ],
      "type": "Point"
    }
  },
  {
    "type": "Feature",
    "properties": {
      "City": "Malindi",
      "Population": 119859
    },
    "geometry": {
      "coordinates": [
```

```
            40.10521499751357,
            -3.2138767356491655
          ],
          "type": "Point"
        }
      }
    ]
}
```

Can you notice any difference on the `Population` property compared to the code in Chapter 5? If you are hawkeyed, you will notice that the **Population** values this time round are integers compared to the string values used in the previous chapter. It sounds superflous to create population values as strings only to convert them to integers now, but please do remember the geojson.io site did that for us, not this author. Here is the raw geojson script customized for this chapter. It is also available here.

Just a small note before going on. When the GeoJSON file has the population values enclosed in strings `""`, they are automatically rendered on a map on the Github server as shown below.

```
knitr::include_graphics(rep("geojson-webmap.jpg"))
```



However, when the strings are removed, and the population values remain as integers, they are no longer rendered on a webmap as shown below.

```
knitr::include_graphics(rep("geojson-nowebmap.jpg"))
```

No map rendered, just a dictionary containing other dictionaries.

6.2 The icons

Alright. Let's create a map of our cities but with custom markers this time round. The below code creates our custom markers.

```
// Yellow Icon
var yellowIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/img/marker-i
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shadow.png',
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });

// Orange Icon
var orangeIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/img/marker-i
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shadow.png',
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });

// Red Icon
var redIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/img/marker-i
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shadow.png',
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });
```

We created three markers in order of importance: yellow, orange, red. You will see the significance (not so much) of these colors later.

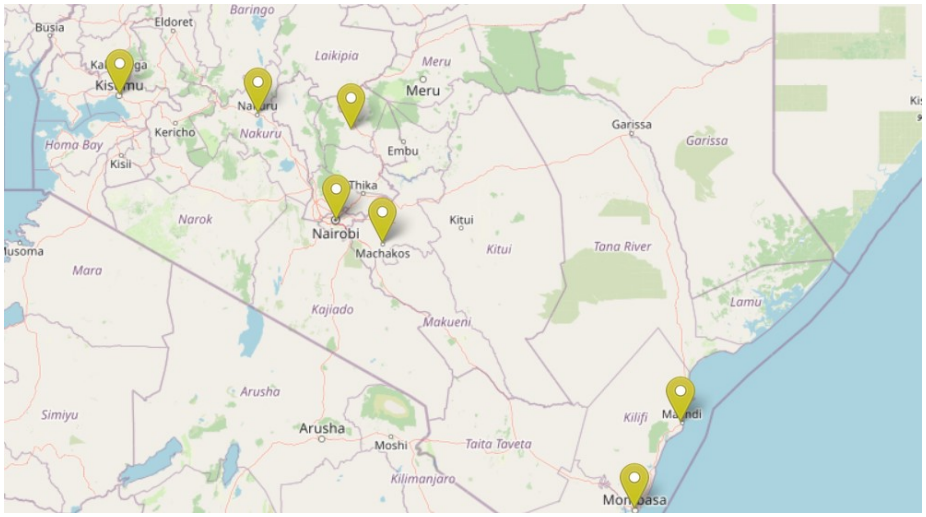
Time to create GeoJSON markers out of this.

```
L.geoJSON(cities, {
    pointToLayer(feature, latlng) {
        return L.marker(latlng, {icon: yellowIcon});
    }
}).bindPopup(function (layer) {
```

```
    return layer.feature.properties.City;
}).addTo(map);
```

We got this.

```
knitr::include_graphics(rep("geojson-markers.jpg"))
```



All cities are marked yellow irrespective of their population or jurisdictional significance. From a cartographer's perspective, the webmap needs more styling but before we get there, what was the purpose of the `pointToLayer()` function? According to the Leaflet guide, the `pointToLayer()` function is a special function for GeoJSON variables that specifies how they should be drawn. To be more descriptive, the function parses the `return L.marker...` function to every Lat-Lon coordinate to make a marker appear at that point.

6.3 Differentiate custom markers on a webmap

Now to the city markers. We would appreciate if the markers would differentiate the cities based on a particular variable, say population. By the way, size of cities is generally determined by population. The below code shall categorize cities by the size of circle markers which shall be based on the city's population. If you've seen point symbols in Qgis, get ready to see them in action in Leaflet!

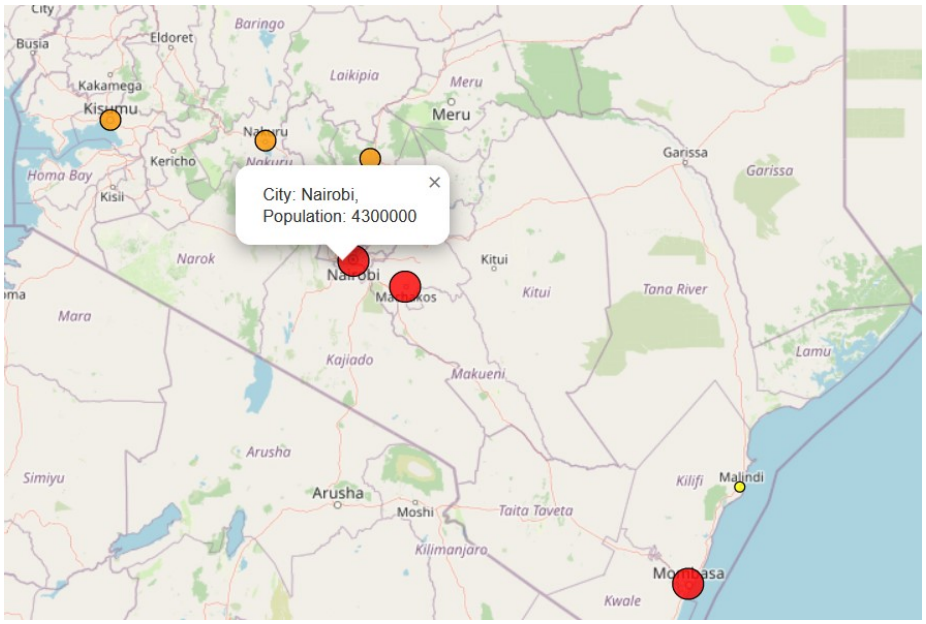
Comment out the earlier code and insert this:

```
L.geoJSON(cities, {
  pointToLayer: function (feature, latlng) {
```

```
    if (feature.properties.Population <= 250000) {
      return L.circleMarker(latlng, {
        radius: 4,
        fillColor: '#FFFF00',
        color: '#000',
        weight: 1,
        opacity: 1,
        fillOpacity: 0.8
      });
    } else if (feature.properties.Population <= 800000) {
      return L.circleMarker(latlng, {
        radius: 8,
        fillColor: '#ff9900',
        color: '#000',
        weight: 1,
        opacity: 1,
        fillOpacity: 0.8
      });
    } else {
      return L.circleMarker(latlng, {
        radius: 12,
        fillColor: '#FF0000',
        color: '#000',
        weight: 1,
        opacity: 1,
        fillOpacity: 0.8
      });
    }

  }
}).bindPopup(function (layer) {
  return `City: ${layer.feature.properties.City},<br>
  Population: ${layer.feature.properties.Population}`;
}).addTo(map);
```

What you get is a map where the size of the circle markers has been defined by the city's population. This time round, the `pointToLayer()` function body worked with **if/else if** statement to differentiate the radius and color of each circle marker. Under the **if/else if** code statement block, different circle marker specifications of radius and fillColor were inserted for each population category. Any city with a population beyond 800000 was fitted into the `else` block.

```
knitr::include_graphics(rep("geojson-diff1.jpg"))
```



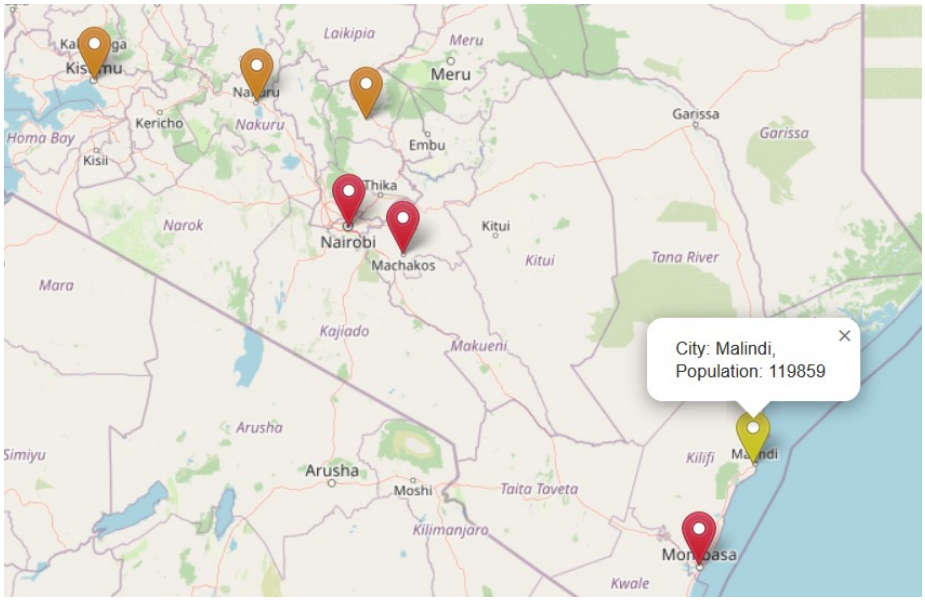
This brings us to why we changed the population values from strings to integers. If we were to work with the original string values (like "1000000" in quotes), any value beyond 1, 000, 000 would receive the settings of `feature.properties.Population <= 250000`. That is, it would be displayed in the same color scheme of yellow as a city with a population say, 25, 000 people and below. That would be passing a wrong message. The explanation for this glaring error arising from use of string values is this: when ordering numbers enclosed in strings in JavaScript, they will be ordered by their first character irrespective of the size of the value. In other words, 1 is greater than 9 even though the latter is more. Thus a city of **"800, 000"** people will be treated as bigger than a city of **"1, 000, 000"**. Now you see the logic of working without the quotes around the **Population** variable, as is the case with our GeoJSON file.

In the next code sample, we shall create city marker icons whose colors shall be determined by the size of the city's population. We will work with three colors: yellow, orange and red. Yellow shall mark the smallest city while red shall indicate the largest. The below code shows how this categorization is worked out.

```
L.geoJSON(cities, {
  pointToLayer: function (feature, latlng) {
    if (feature.properties.Population <= 250000) {
      return L.marker(latlng, {
        icon: yellowIcon
      });
```

```
    } else if (feature.properties.Population <= 800000) {
      return L.marker(latlng, {
        icon: orangeIcon
      });
    } else {
      return L.marker(latlng, {
        icon: redIcon
      });
    }

  }
}).bindPopup(function (layer) {
  return `City: ${layer.feature.properties.City},<br>
  Population: ${layer.feature.properties.Population}`;
}).addTo(map);
```

```
knitr::include_graphics(rep("geojson-diff2.jpg"))
```



In the above map, large cities with populations above 800, 000 have been categorized with a red marker, those with populations below 250, 000 with a yellow marker and those in between with an orange marker. In all cases, our `bindPopup()` still contains the same settings of showing both the city name and population size.

The value to the `pointToLayer()` key was assigned a function that checks if a city's population is within a specific range. If a city falls within the range spec-

ified by the `if` statement, a particular icon –whether `yellowIcon`, `orangeIcon` or `redIcon` is returned by the `L.marker()` function.

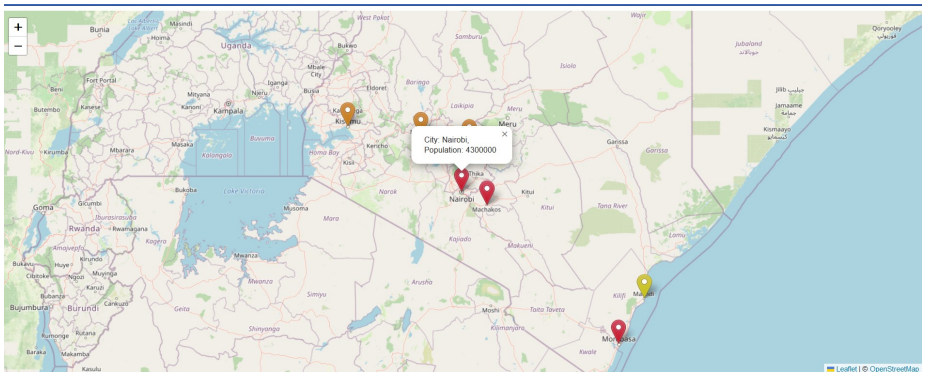
6.4 Using fetch

Remember how `fetch` helped us retrieve data from a server in a previous chapter? Whereas we won't repeat the entire process again (you can breathe a sigh of relief), the same iterations of differentiating a marker icon can also be inserted into the `fetch` API. All this is done right within the options of `L.geoJson(data, {options})`.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/cities-geojso
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        L.geoJson(data, {
          pointToLayer: function (feature, latlng) {
            if (feature.properties.Population <= 250000) {
              return L.marker(latlng, {
                icon: yellowIcon
              });
            } else if (feature.properties.Population <= 800000) {
              return L.marker(latlng, {
                icon: orangeIcon
              });
            } else {
              return L.marker(latlng, {
                icon: redIcon
              });
            }

          }
        }).bindPopup((layer) => {
            return `City: ${layer.feature.properties.City},<br>
            Population: ${layer.feature.properties.Population}`}).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

The resulting map should look like the previous one.

```
knitr::include_graphics(rep("cities_colored.jpg"))
```



6.5 Unique custom markers

This part may not be necessary, but it has been inserted to show you that there are various markers apart from the defaults provided by Leaflet. One can create custom markers outside of Leaflet using the Leaflet.Awesome.Markers plugin. Just like in the case of Ajax, you will need to provide the path to the plugin's dependencies using the `<script>` tag. Insert the following `<script>` tags into `map.html`.

```
<script src="Leaflet.awesome-markers-2.0-develop\Leaflet.awesome-markers-2.0-develop\dist\leaflet
```

```
<script src="Leaflet.awesome-markers-2.0-develop\Leaflet.awesome-markers-2.0-develop\dist\leaflet
```

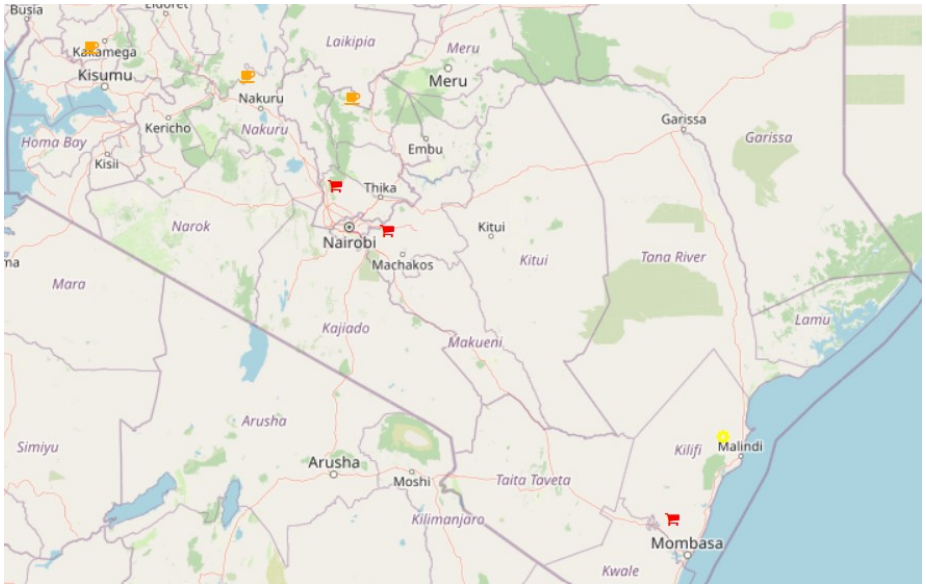
And also this `<link>` tag:

```
<link href="http://netdna.bootstrapcdn.com/font-awesome/4.0.0/css/font-awesome.css" rel="styleshe
```

To make the best use of time, following the example on their site, we simply replace the value of our `icon` keys with the new `L.AwesomeMarkers.icon`. We also tweaked the colors for each icon to match those used previously. Because these markers signify various amenities, we made some overly simplified assumptions for the sake of demonstrating their use ie. big cities with a population above 1, 000, 000 have the best malls, those with a population between 250, 000 and 800, 000 must be having good coffee places, and those with populations below 250, 000 obviously have respectable industries. We assume fair play has been exercised in our assumptions. So here is the code.

```
L.geoJSON(cities, {
    pointToLayer: function (feature, latlng) {
      if (feature.properties.Population <= 250000) {
        return L.marker(latlng, {
          icon: L.AwesomeMarkers.icon({icon: 'cog', prefix: 'fa', markerColor: 'purple
        });
      } else if (feature.properties.Population <= 800000) {
        return L.marker(latlng, {
          icon: L.AwesomeMarkers.icon({icon: 'coffee', prefix: 'fa', markerColor: 'red
        });
      } else {
        return L.marker(latlng, {
          icon: L.AwesomeMarkers.icon({icon: 'shopping-cart', prefix: 'fa', markerColo
        });
      }

    }
  }).bindPopup(function (layer) {
    return `City: ${layer.feature.properties.City},<br>
    Population: ${layer.feature.properties.Population}`;
  }).addTo(map);
```

```
knitr::include_graphics(rep("extra-markers.jpg"))
```



The custom markers are also clickable!

6.6 Image overlays

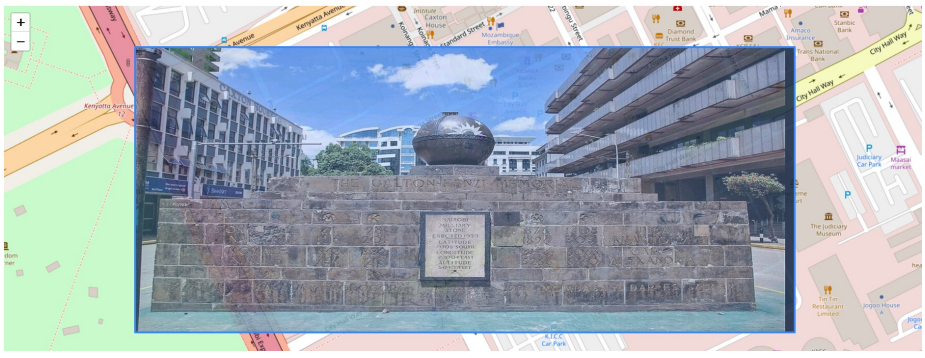
Sometimes an image can act as good a marker. Overlaying images on a map is fairly easy. The below code inserts an image above a historical monument located in the Kenyan capital, Nairobi.

```
// Image overlays
var imageUrl = 'https://pbs.twimg.com/media/DddQBk5WsAAlbdJ?format=jpg&name=large';
var errorOverlayUrl = 'https://pbs.twimg.com/media/DddQBk5WsAAlbdJ?format=jpg&name=large';
var altText = 'The Galton - Fenzi Memorial: Source: Google and Twitter';
var latLngBounds = L.latLngBounds([[-1.2861259,36.8172709], [-1.2886193,36.8230413]]);

var imageOverlay = L.imageOverlay(imageUrl, latLngBounds, {
    opacity: 0.8,
    errorOverlayUrl: errorOverlayUrl,
    alt: altText,
    interactive: true
}).addTo(map);
```

However, finding an image of just one location over the wide earth can be laborious, so we envelope it with a rectangle. The `map.fitBounds` function enables the browser to automatically zoom to where our image is placed.

```
L.rectangle(latLngBounds).addTo(map);
map.fitBounds(latLngBounds);
```

```
knitr::include_graphics(rep("geltan-fenzi.jpg"))
```



Mind you that landmark was set up in 1939 in honour of Lionel Douglas Galton-Fenzi; the first motorist to drive from Nairobi to Mombasa as early as 1926. The landmark is also inscribed with bearings to various East African cities but enough history for today.

Anyway, here is a quick breakdown of the attributes used in `L.imageOverlay`: a) the `var latLngBounds` uses the `L.latLngBounds` class to set the lat-lon coordinates. Notice they are two coordinate lists bound with a single `[]`. Failing to enclose the two coordinates with `[]` results in an error. b) `var ImageUrl` is the image source.

For the additional options parsed to the `L.imageOverlay()` class, they are described below:

1. `opacity` - defines the opacity of the image overlay, it equals to 1.0 by default. Decrease this value to make an image overlay transparent and to expose the underlying map layer.
2. `errorOverlayUrl` - is a URL to the overlay image to show in place of the overlay that failed to load.
3. `alt` - sets the HTML alt attribute to provide an alternative text description of the image. It is quite helpful in describing an image in text form just in case it fails to load due to poor network connectivity. Moreover, it can improve the Search Engine Optimization (SEO) of the website it is hosted in.
4. `interactive` - is false by default. If true, the image overlay will emit mouse events when clicked or hovered.

Full codes and files are here.

6.7 Summary

This was quite a long chapter that rendered justice on how to customize GeoJSON markers in Leaflet. Through the various exercises you encountered in this chapter, you have learnt the following:

- GeoJSON files, if in the correct format, can be rendered as a standalone map from Github. An example is here.
- One can customize marker colour or size based on the GeoJSON's file attributes. In this chapter, the city population values were used to define the marker's size and colour.
- The `pointToLayer` key, when used as an option in `L.geoJSON`, will parse a particular function to every Lat-Lon coordinate of the GeoJSON object.
- It is also possible to customize markers in the `fetch` API by simply specifying how they should appear, in form of functions, within the `L.geoJson` options environment.

- There exist plenty of custom made icons outside of, but compatible with Leaflet. An example of such a library is Leaflet Awesome Markers.
- Images can also be overlayed on a Leaflet map. This was done in the sub-chapter of Image Overlays.

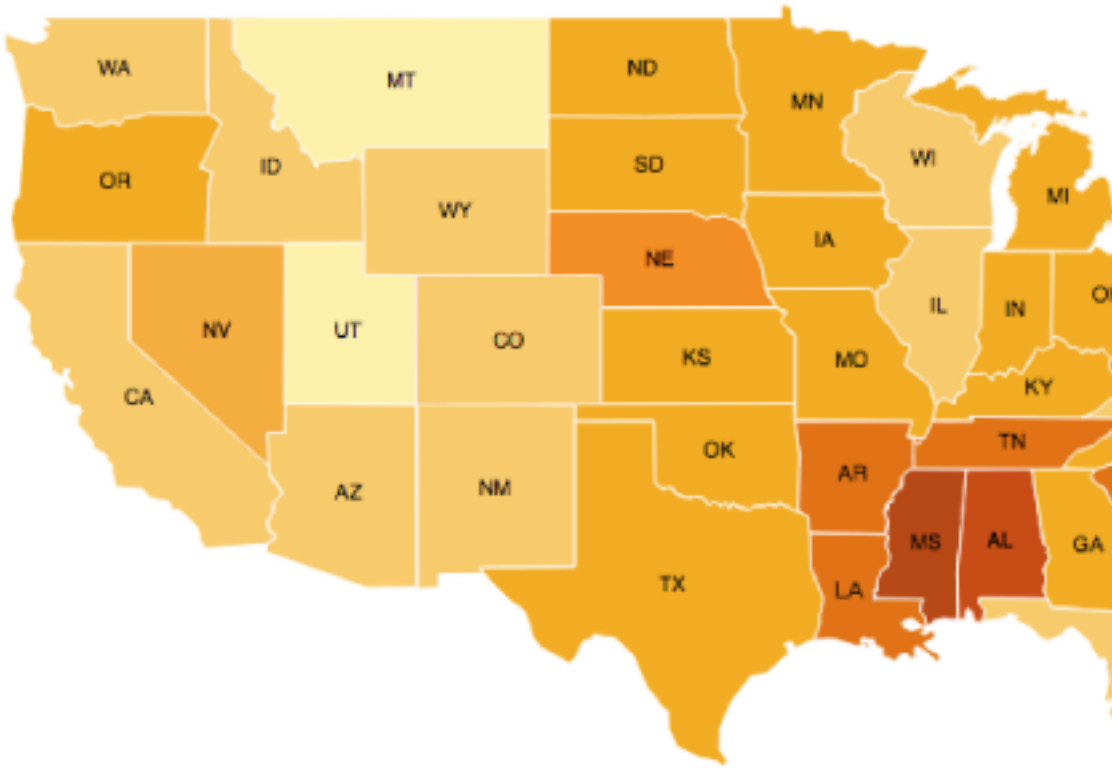
Chapter 7

Creating an interactive choropleth map

7.1 What is a choropleth map?

We will now move from markers to something larger than life –choropleth maps. *What the heck are choropleth maps?* You may ask. Geographers will roll their eyes over this term because they have come across it innumerable times throughout their career. However, for the sake of new readers, a choropleth map is a map whose geographical areas or regions are colored, shaded or patterned in relation to a data variable. If during election time you have seen a map that has draped the winning candidate in (a) particular state(s) with their party theme colors, then that's a choropleth map. Most population maps are also choropleth maps. In this chapter, we will create a choropleth map of Kenyan counties, and make it interactive by leveraging the area and population characteristics of each county.

```
knitr::include_graphics(rep("choropleth.png"))
```



Source: The Data Visualization Catalogue

7.2 Creating a choropleth map: the start

Obviously by now, without going into much details, you can now create a basic Leaflet map blindfolded. Anyway, seeing is believing, so lets start it right away anyway. Create another new JavaScript file called `interactive-choropleth.js`. To break the monotony, we shall use a different tile layer. Remember that Leaflet has several tile layer servers, and CyclOSM, used in this chapter is just one among many.

```
var map = L.map('myMap').setView([0.3556, 37.5833], 6.5);

L.tileLayer('https://{s}.tile-cyclosm.openstreetmap.fr/cyclosm/{z}/{x}/{y}.png', {
    maxZoom: 20,
    attribution: '<a href="https://github.com/cyclosm/cyclosm-cartocss-style/releases"
}).addTo(map); // the CyclOSM tile layer available from Leaflet servers
```

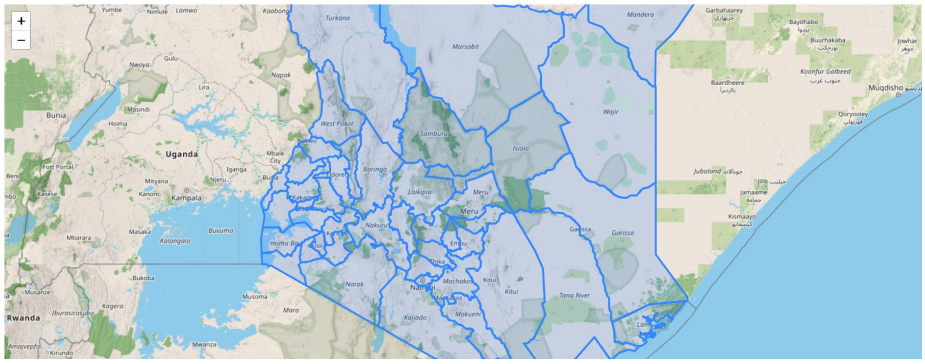
Now to the big part –the GeoJSON files. We would like to mention it was quite a hustle to set up the GeoJSON file to be in a format accessible by JavaScript's Fetch API. Only when we converted the GeoJSON to `.json` were we able to successfully view it using `fetch`. The raw json file for our Kenyan counties which we shall use in creating a choropleth map is available from here or here.

Let's fetch the counties json file.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/counties_json.json")
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        L.geoJson(data, {style: style}).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

So far, you should get a result like below.

```
knitr::include_graphics(rep("kenya_json.jpg"))
```



7.3 Coloring the counties

Alright, we have been able to load our json file to a Leaflet map. However, it looks dull and provides no meaningful information to the casual observer. When making maps, aim to provide information at lightning speed. That is, inform the reader at first glance. The code snippets that follow have been heavily borrowed from Volodymyr's interactive choropleth tutorial.

First, let's create a function that sets a hex colour code for each population category. We used color brewer for this.

Take a look at this code and we shall explain.

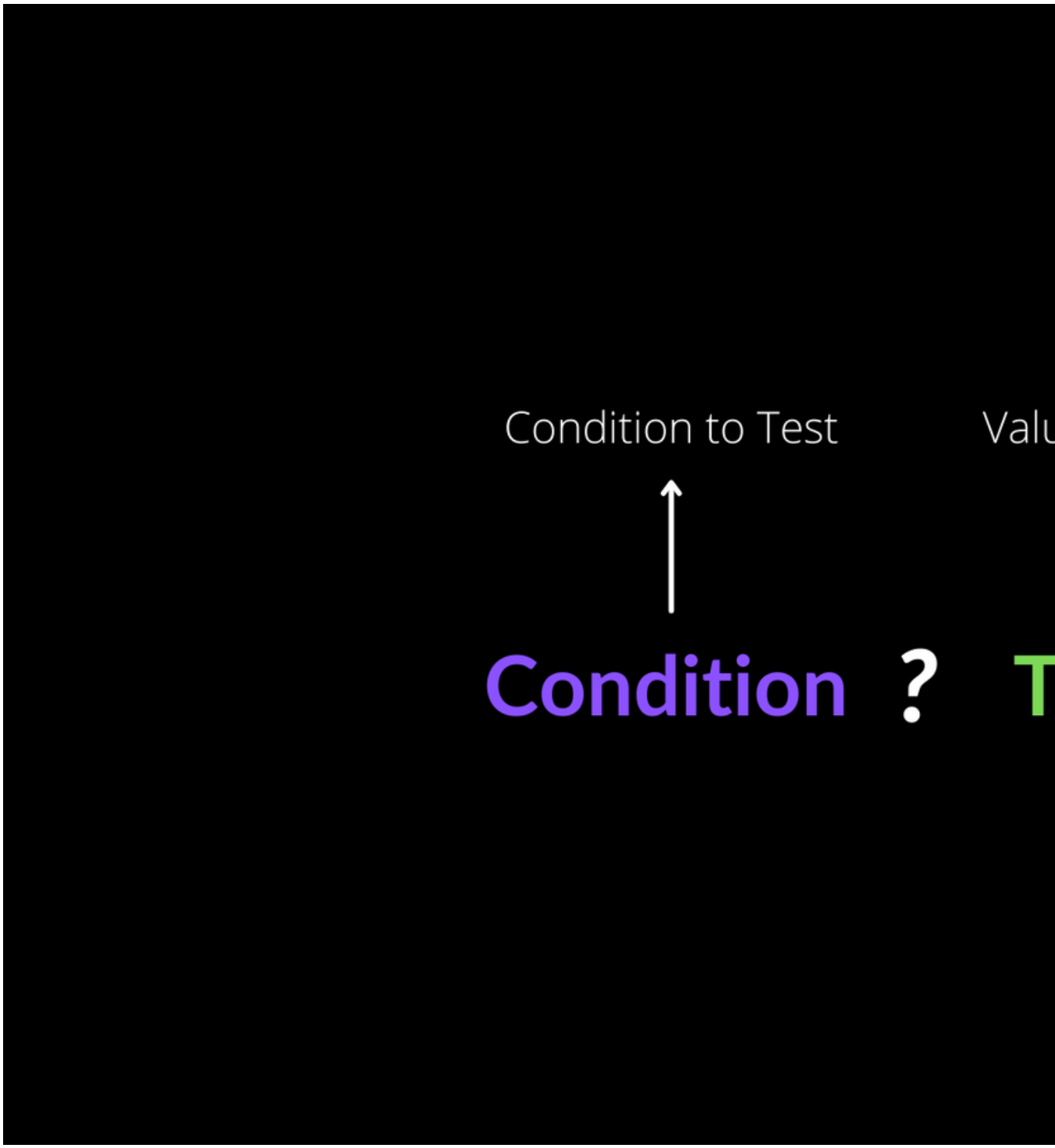
```
//// Adding some color
function getColor(d) {
    return d > 1400 ? '#8c2d04' :
            d > 700 ? '#cc4c02' :
            d > 400 ? '#ec7014' :
            d > 100 ? '#fe9929' :
            d >  50 ?  '#fec44f':
            d > 25 ?   '#fee391':
                       '#ffffd4';
}
```

The above function uses a question mark `?` to act as the `if...else` statement. In JavaScript, this is known as a ternary operator. A ternary operator? What's that? How does it work?

For a ternary operator, any statement to the right of the `?` is returned as true if it agrees with the value to the left of the `?`. Reread that last statement. If the value to the right of the `?` is false, then the value after the colon `:` is returned. Reread that last statement for this ternary operator concept to sink in.

If it still sounds fuzzy, the below image adapted from FreeCodeCamp should help.

```
knitr::include_graphics(rep("ternary.png"))
```


Condition to Test
Valu
Condition ?
T

In other words, the ternary operator is a short form of writing the `if...else` statement over several lines. Look at the below code which is simply the `if...else` version of the ternary operator we've used above. Definitely, the ternary operator wins the day in terms of brevity.

```
function getColor(d) {
  if (d > 1400) {
    return '#8c2d04';
  } else if (d > 700) {
    return '#cc4c02';
  } else if (d > 400) {
    return '#ec7014';
  } else if (d > 100) {
    return '#fe9929';
  } else if (d > 50) {
    return '#fec44f';
  } else if (d > 25) {
    return '#fee391';
  } else {
    return '#ffffd4';
  }
}
```

Now that we've created the function of setting colors to our json file on Leaflet, we next also have to *create* a function that applies the color designation to the counties in the GeoJSON file itself. Luckily, we have the **style** option from Leaflet which is a function for precisely styling GeoJSON lines and polygons alone. We saw it in Chapter 5 and we shall also use it here.

```
var style = ((feature)=> {
    return {
        fillColor: getColor(feature.properties.Pop_Density),
        weight: 2,
        opacity: 1,
        color: 'gray',
        fillOpacity: 0.5
    }
})
```

The above is an arrow function. Unlike regular JavaScript function declarations which are in the format of `function <name of function> (<parameter>)`, we remove the `function` keyword, enclose everything in brackets and put an arrow `=>` between the parameter brackets and function body. That's just it. Arrow functions aren't so hard!

Remember we assign the arrow function to a variable called `var style` since we will parse it to the `L.geoJson` class.

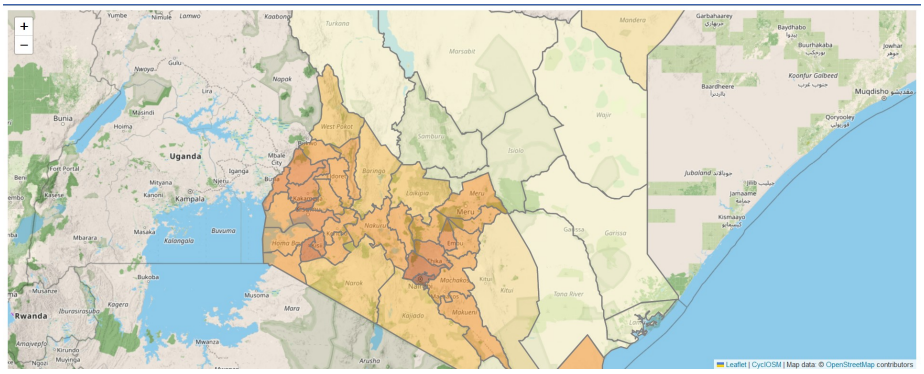
Finally, we add the `style` variable to the **style** option of `L.geoJson` class.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/counties_json.json")
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        L.geoJson(data, {style: style}).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

Since the **style** option is a key (and also a function), the value will be the `var style` which we created. This value is in and by itself a function that iterates over every county because of the `getColor(feature.properties.Pop_Density)` contained in it as the `fillColor` value!

Enough JavaScript for one day!

```
knitr::include_graphics(rep("choropleth-map.jpg"))
```



Our choropleth map is beginning to take shape.

7.4 Highlight features

Going on from where we last left, we would like the choropleth map to highlight any county that the screen pointer (or mouse) hovers over. Additionally, upon

clicking, the map should zoom to the clicked county as well as display its attributes. The counties should also be reset to their default characteristics when the screen pointer hovers out.

Alright. It seems like we have our hands full.

Let's start simple, and scale upwards in terms of complexity.

Remember the `fetch` API we had used in retrieving our json file? We will tweak it a bit by adding the `var geojson` just before calling the `L.geoJson` class. We shall parse the variable `var geojson` to the `L.geoJson` class. We shall explain why we are assigning `L.geoJson` to a variable (`var geojson`) when we know very well it can work alone, as seen in the last two previous chapters.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/counties_json
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        var geojson;
        geojson = L.geoJson(data, {
            style: style,
            --snip--
    })
```

We had initially mentioned that we wanted the county hovered over to be highlighted. The following code will highlight with a white hue the county hovered over while also seemingly popping it out above the rest.

```
        geojson = L.geoJson(data, {
            style: style,
            onEachFeature: ((feature, layer) => {
                layer.on('mouseover', ((e) => {
                    var layer = e.target;

                    layer.setStyle({
                        weight: 5,
                        color: '#FFFFFF',
                        dashArray: '',
                        fillOpacity: 0.7
                    });

                    layer.bringToFront();
                }))
```

The purpose of `on` method is to add an event listener. Event listeners in JavaScript are functions that run a code when the browser user interacts with the browser in a specific way. Now what `on` method does is that it triggers the event listener `mouseover` for each county in our GeoJSON layer. The `mouseover` event is what makes our counties be highlighted and 'pop' out when a screen pointer is *over* them.

Since the change in state of an element in HTML is known as an event (denoted as `e` in our case), the `e.target` property returns the element on which the event is occurring on. Since it's a particular county in our case, we proceed to change its symbology through the parameters in the `setStyle` function. Thereafter we use the `bringToFront` method to make the element in which the event has happened on to 'pop' out above the rest.

Remember we had mentioned we also want the counties to be reset to their default status when one hovers out to some other county or outside the map altogether. The following code does the job.

```
layer.on('mouseout', function() {
                    geojson.resetStyle(this);
                })
```

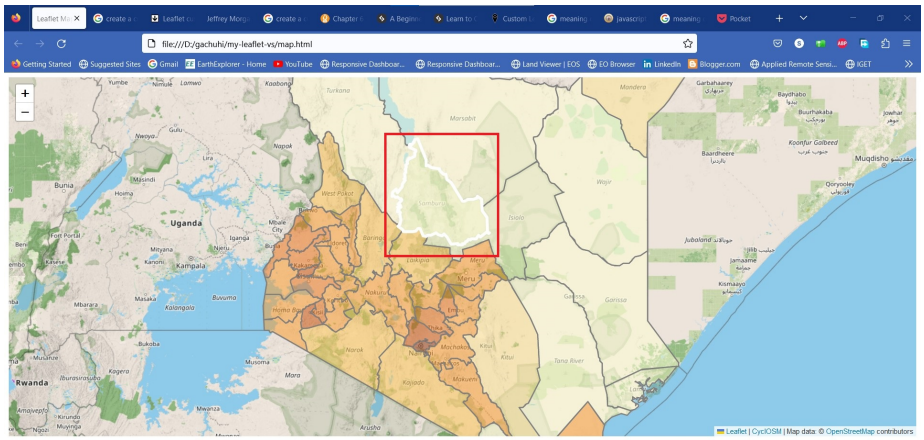
We use the `resetStyle` function to return the layer to default settings but there is a twist. In the `resetStyle` case, we add the argument `this` in parenthesis to refer to the element that was received. In other words, when the mouse 'hovers out' of a county, the element will revert to its original symbology.

Before, we end this monologue, we ensure we pass the variable `geojson` to `resetStyle` function or else it won't work. This appears to be the *modus operandi* for resetting styles in Leaflet as shown in this Stack Overflow question and also here.

Finally, we mentioned we want to zoom to a particular county upon clicking it. The following code fits our map to the bounds of the particular county that was clicked. Note that `fitBounds` is parsed `getBounds` which gets the boundaries of the county clicked upon. The browser is able to identify the county clicked upon through the help of the `e` parameter and `e.target`.

```
layer.on('click', ((e) => {
                    map.fitBounds(e.target.getBounds())
                }))
```

```
knitr::include_graphics(rep('highlightable-map.jpg'))
```



Your code within the `fetch` API should look like this.

```
fetch("https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/counties_json
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        var geojson;
        geojson = L.geoJson(data, {
            style: style,
            onEachFeature: ((feature, layer) => {
                layer.on('mouseover', ((e) => {  // highlight county on mouse hover
                    var layer = e.target;

                    layer.setStyle({
                        weight: 5,
                        color: '#FFFFFF',
                        dashArray: '',
                        fillOpacity: 0.7
                    });

                    layer.bringToFront();
                }))

                layer.on('mouseout', function () { // return to original symbology upo
                    geojson.resetStyle(this);
                })

                layer.on('click', ((e) => {  // Zoom to county upon clicking it
                    map.fitBounds(e.target.getBounds())
```

```
                })

            })
        }).addTo(map);
    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

7.5 Creating a custom info

Our choropleth map works perfectly but lacks some interactivity. How about complementing it with an information interface next to it? Custom info controls are one way to do that. Think of a control as an UI element that allows interactivity. For our choropleth map, we want the custom control info to provide details of the name, total population and population density of each county hovered over.

```
// Add control
var info = L.control();

info.onAdd = function (map) {
    this.div = L.DomUtil.create('div', 'info');
    this.update();
    return this.div;
};

// Method that we will use to update the control based on feature properties passed
info.update = function (props) {
    this.div.innerHTML = '<h4>Kenya Population Density</h4>' + (props ?
        '<b>' + props.ADM1_EN + '</br><br />' + 'Total Population' + '<br>' + props.County_pop +
        props.Pop_Density + ' people / km<sup>2</sup>': 'Hover over state')
};

info.addTo(map);
```

Okay. Let's go through the above code bit by bit as best as we (hopefully!) can.

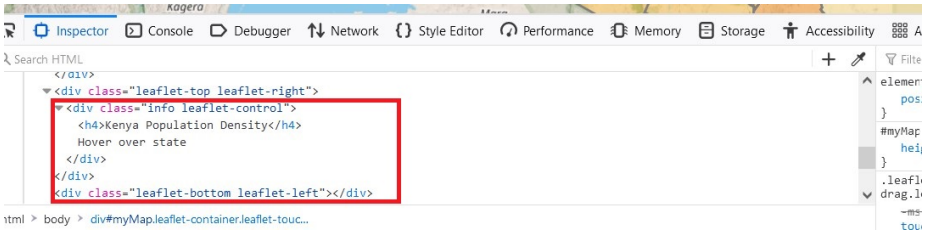
```
var info = L.control();
```

The above creates a variable `info` that holds the base class `L.control()` for all map controls. For example, in `L.control.zoom`, the `L.control` class creates a zoom control in the map.

```
info.onAdd = function (map) {
    this.div = L.DomUtil.create('div', 'info');
    this.update();
    return this.div;
};
```

The above code returns the DOM element for the control and creates a `<div>` of class `info`. This is done through the help of `L.DomUtil` which, according to the Leaflet website, provides utility functions to work with the Document Object Model (DOM)[1]. Actually, this new `<div>` of class `info` is created when you fire up your browser but will only exist in your browser. Don't expect it to magically appear in your static `map.html` file.

```
knitr::include_graphics(rep('div-info-element.jpg'))
```



In the aboe image, the `<div>` class bounded in red wasn't there before introducing the function containing `L.DomUtil`. Moving on forward, `this.update()` simply updates the content of the custom info control with the attributes of every newly clicked county.

```
// Method that we will use to update the control based on feature properties passed
info.update = function (props) {
    this.div.innerHTML = '<h4>Kenya Population Density</h4>' + (props ?
        '<b>' + props.ADM1_EN + '</br><br />' + 'Total Population' + '<br>' + props.Cou
        props.Pop_Density + ' people / km<sup>2</sup>': 'Hover over state')
};

info.addTo(map);
```

The above function updates the Leaflet map with the name, population and population density for each county. This function is passed to the variable `info.update` and thereafter added to the map using the method `addTo`. Inside this function exists the `this.div.innerHTML`. The purpose of `innerHtml` is to return the HTML content of an element. Since our map is rendered in a HTML page, the features will be returned as HTML.

[1]The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web.

Because the custom control info is a UI element, we need to set up its symbology in our CSS file. Paste the following to your `styles.css` file.

You may ask, "I thought I don't have to put CSS styles to Leaflet because it seems to come with all batteries included, style 'n all".

I get your point, but remember we created a new `<div>` called `info` that appears when our browser is powered up. And since this `<div class="info ...">` must appear when the browser is powered up, CSS styles must be used to define its looks as well.

```
.info {
    padding: 6px 8px;
    font: 14px/16px Arial, Helvetica, sans-serif;
    background: white;
    background: rgba(255,255,255,0.8);
    box-shadow: 0 0 15px rgba(0,0,0,0.2);
    border-radius: 5px;
}

.info h4 {
    margin: 0 0 5px;
    color: #777;
}
```

7.6 Create a legend

Having created a custom control info, the following code creates the legend.

```
var legend = L.control({position: 'bottomright'});

legend.onAdd = function (map) {
    var div = L.DomUtil.create('div', 'info legend'),
    grades = [0, 25, 50, 100, 400, 700, 1400],
    labels = [];

    // loop through our density intervals and generate a label with a colored square for each int
    for (var i = 0; i < grades.length; i++) {
        div.innerHTML +=
            '<i style="background:' + getColor(grades[i] + 1) + '"></i> ' +
            grades[i] + (grades[i + 1] ? '&ndash;' + grades[i + 1] + '<br>' : '+');
    }
    return div;
}
```

```
legend.addTo(map);
```

Obviously the position of our legend is set using the `position` option in `L.control({position: 'bottomright'});`.

Apart from the other elements we discussed in our custom control info, we set the color interval of our legend through the `grades` array. The `for` loop that follows iterates through the grades array creating a color box for each interval.

Honestly, the code to create the legend color scheme is quite complicated. It has actually been copy pasted from the one in the Leaflet choropleth tutorial. The strangely looking `&ndash` within the `for` loop of creating colors is simply how a hyphen (-) is written in HTML. But anyway, getting back to the `for` loop, it essentially creates a range within each interval, such as 0 - 25, 25 - 50 and so on. This takes place after the `?` ternary operator discussed earlier. Once it reaches the end of the loop, that is, it goes out of the range, the `+` is appended to the last value from our `grades` array. This last bit is made possible due to the `:` which returns values that are false in JavaScript.

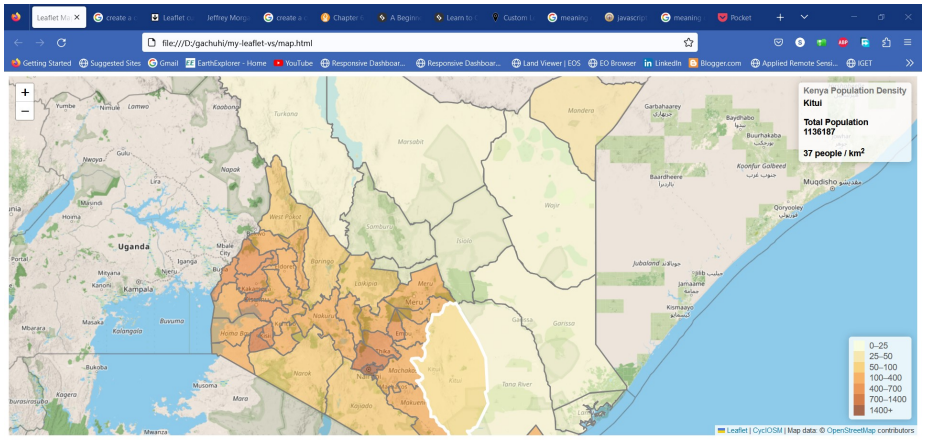
The legend also needs its CSS properties or else it will not appear in the browser. The `<div>` class of `info legend` is also created when the browser fires up. You may be wondering why `info legend` and not just `legend`. Well, dear Golden Eye, the properties of `.info` class, such as background color and others are also inherited by the `legend` class by virtue of the browser, as in the case of HTML files, reading the CSS in a top-down format. The `legend` class will only override those properties that also appear in the `info legend` class. We also specify the CSS properties for the color intervals and text through the `.legend i` class.

Here's the CSS anyway.

```
.legend {
    line-height: 18px;
    color: #555;
}
.legend i {
    width: 18px;
    height: 18px;
    float: left;
    margin-right: 8px;
    opacity: 0.7;
}
```

The legend is done and is finally added to the map.

```
knitr::include_graphics('choropleth-legend.jpg')
```



The full code files are available from here.

When creating choropleth maps, aim for challenging and impacting the reader rather than merely informing.

7.7 Summary

This chapter was long and hard if not confusing. Nevertheless, here's our take home from this chapter:

- Choropleth maps are maps whose whose geographical areas or regions are colored, shaded or patterned in relation to a data variable. Good examples of choropleth maps are national population maps.
- Ternary operators are short forms of the `if...else` statement. In a ternary operator, any statement to the right of the `?` is returned as true if it agrees with the value to the left of the `?`. Conversely, any value after the colon `:` is returned if the value to the right `?` is false.
- `style` is a special key in the `L.geoJson` class that applies custom styles to each layer in the GeoJSON file.
- `mouseover` is an event listener that triggers a certain action when a screen pointer is over a certain feature.
- We can reset features back to their defaults using `resetStyle`.

- The `L.control()` class is the base class for enabling most Leaflet controls. For example, the `L.control.zoom` creates the zoom controls.
- `L.control()` can also be used to customize the appearance and information displayed by the legend.

Chapter 8

Layer groups and controls

8.1 Purpose of layer groups and controls

Sometimes, one may wish for their webmap to consist of several baselayers or overlay maps. Suppose you want your Leaflet to have two basemap layers, and an additional overlay with the option of switching to any of them, how would you proceed?

8.2 Set up the basemaps

In order to create controls, we have to parse the objects holding our map variables into a `L.control.layers` class. Thereafter, the `L.controls.layers` class is used to parse the object values to the Leaflet map and create a UI control. To demonstrate this, open a new JavaScript file and name it `groups_controls.js`. Insert the following code which will save our basemaps to the respective variables of `osm` and `cyclOSM`.

```
var osm = L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
});

var cyclOSM = L.tileLayer('https://{s}.tile-cyclosm.openstreetmap.fr/cyclosm/{z}/{x}/{y}.png', {
    maxZoom: 20,
    attribution: '<a href="https://github.com/cyclosm/cyclosm-cartocss-style/releases" title="Cyc
}); // the CyclOSM tile layer available from Leaflet servers
```

We will pass the above two variables of `osm` and `cyclOSM` to the `L.map` class which has an option of `layers` in which one can parse the layers they wish to be displayed on the map.

```
var map = L.map('myMap', {
    layers: [osm, cyclOSM]
}).setView([-1.295287148, 36.81984753], 7);
```

However, that will only add the first basemap variable –that for `osm` while blocking out that of `cylcOSM`. This is shown below.

```
knitr::include_graphics(rep('no-control.jpg'))
```

8.3 Creating the controls

However, in order to give `cyclOSM` a fair chance, we need to store them in an object say `var basemaps` and parse it to `L.controls.layer` which shall create a checkbutton for each basemap. The below code does just that.

```
// Set object for the basemaps
var basemaps = {
    "OpenStreetMap": osm,
    'cycleOsm': cyclOSM,
}

L.control.layers(basemaps).addTo(map);
```

This is the result you get.

```
knitr::include_graphics(rep('controls.jpg'))
```

8.4 Adding overlay maps

Now we have seen how to add two or more basemaps to Leaflet and make all of them appear in the layer control. Adding an overlay map follows the same procedure as well.

The first overlay we would like to create is a choropleth map displaying the percentage of conventional households with access to main sewers as per the 2019 census. For simplicity purposes and to bypass errors we faced, we shall reuse the Ajax plugin for fetching GeoJSON files from online servers. As a reminder, we load Ajax into Leaflet by inserting it to the following `<script>` tags of our `map.html`.

```
<script src="leaflet-ajax-gh-pages\dist\leaflet.ajax.js"></script>

<script src="leaflet-ajax-gh-pages\dist\leaflet.ajax.min.js"></script>
        <script src="leaflet-ajax-gh-pages\example\leaflet.spin.js"></script>
        <script src="leaflet-ajax-gh-pages\example\spin.js"></script>
```

Using `fetch` API to load the json file brought up several errors requiring out-of-the-box thinking to solve, but using the alternative Ajax plugin was a safe landing. The following chunks of code will add the color function and styling for our countrywide sanitation map. The resulting coloring function shall be parsed to the Ajax function.

```
//// Adding some color
function getColor(d) {
    return d > 20 ? '#3288bd' :
            d > 10 ? '#99d594' :
            d > 6 ? '#e6f598' :
            d > 4 ? '#fee08b' :
```

```
            d >  2 ? '#fc8d59':
                     '#d53e4f';

}

// Function for setting color (using arrow function)
var style = ((feature)=> {
    return {
        fillColor: getColor(feature.properties.Human_waste_disposal),
        weight: 2,
        opacity: 1,
        color: 'gray',
        fillOpacity: 0.9
    }
})
```

Now let's add the overlay map that will display the population's accessibility to main sewer sanitation services. Spoiler alert: the statistics are quite grim.

```
// Adding the first overlay - map of household access to main sewer
var countySanitation = new L.geoJson.ajax("https://raw.githubusercontent.com/sammigachu
    style: style
})
.bindPopup(function (layer) {
            return `<b>County Name: </b> ${layer.feature.properties.ADM1_EN} <br>
            <b>Total County Population: </b><br>
            ${layer.feature.properties.County_pop.toString()} <br><br>
            <b>% of Conventional Households with access to main sewer: </b><br><br>
            ${layer.feature.properties.Human_waste_disposal.toString()}`
        }).addTo(map);
```

Let's add an accompanying legend for the above map. If you worked through Chapter 7 that dealt with interactive choropleths, some of the below code should look familiar.

```
// Create a legend
var legend = L.control({position: 'bottomright'});

legend.onAdd = function (map) {
    var div = L.DomUtil.create('div', 'info legend'),
    grades = [0, 2, 4, 6, 10, 20],
    labels = [];

    // loop through our density intervals and generate a label with a colored square fo
```

```
    for (var i = 0; i < grades.length; i++) {
        div.innerHTML +=
            '<i style="background:' + getColor(grades[i] + 1) + '"></i> ' +
            grades[i] + (grades[i + 1] ? '&ndash;' + grades[i + 1] + '<br>' : '+');
    }
    return div;
}

legend.addTo(map);
```

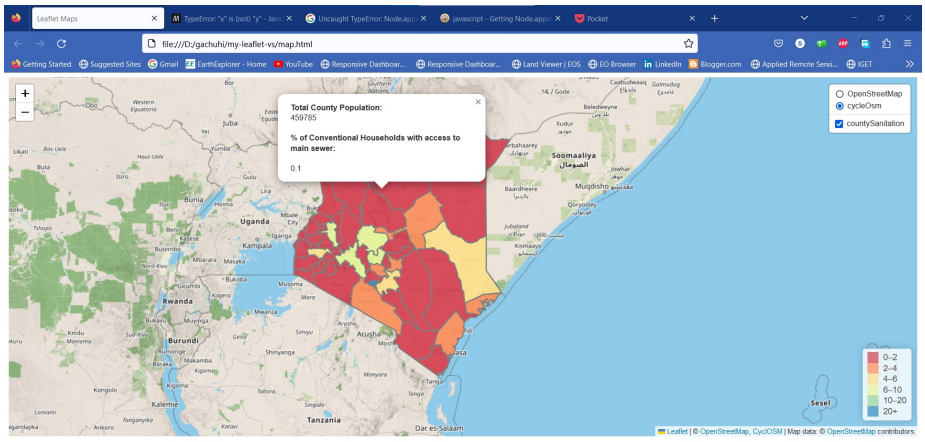
Now set an object to hold one of our two overlay maps.

```
var overlays = {
    'countySanitation': countySanitation,
}
```

Finally parse the `overlays` object to the `L.controls.layer` class.

```
// Add layer controls
L.control.layers(basemaps, overlays).addTo(map);
```

```
knitr::include_graphics(rep('controls-overlay.jpg'))
```



If you click in any one of the counties, you will see popups appear.

There is one more overlay we will add to our display to bring our experimentation with layer controls back full circle. Remember the GeoJSON of our `cities` variable? Let's call it back to action. But first let's load the custom icon markers that differentiate our cities based on population.

```
// Color icons
// Yellow Icon
var yellowIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/ir
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shac
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });

// Orange Icon
var orangeIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/ir
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shac
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });

// Red Icon
var redIcon = new L.Icon({
    iconUrl: 'https://raw.githubusercontent.com/pointhi/leaflet-color-markers/master/ir
    shadowUrl: 'https://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/images/marker-shac
    iconSize: [25, 41],
    iconAnchor: [12, 41],
    popupAnchor: [1, -34],
    shadowSize: [41, 41]
  });
```

Let's load the `cities` GeoJSON with its icons.

```
var cities = L.geoJson.ajax("https://raw.githubusercontent.com/sammigachuhi/geojson_fil
    pointToLayer: function (feature, latlng) {
            if (feature.properties.Population <= 250000) {
              return L.marker(latlng, {
                icon: yellowIcon
              });
            } else if (feature.properties.Population <= 800000) {
              return L.marker(latlng, {
                icon: orangeIcon
              });
            } else {
              return L.marker(latlng, {
```

```
                icon: redIcon
              });
            }

          }
}).bindPopup(function (layer) {
      return `City: ${layer.feature.properties.City},<br>
      Population: ${layer.feature.properties.Population}`;
    }).addTo(map);
```

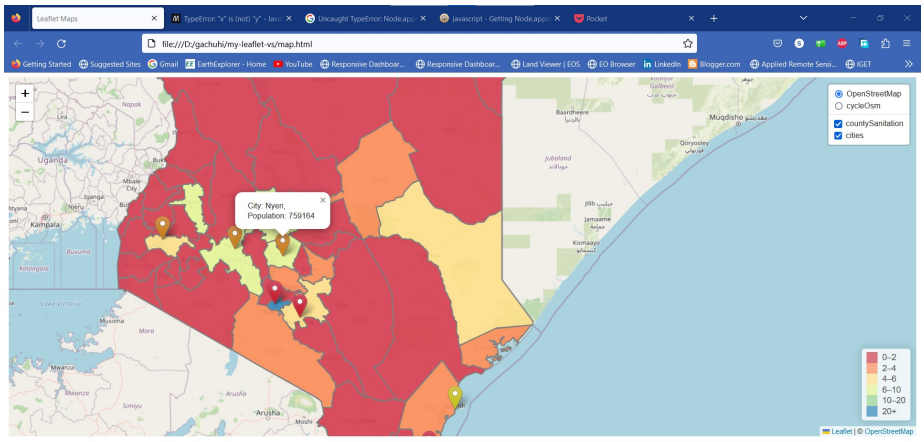
Add the `cities` variable as one of the keys to our `overlays` variable and finally parse the `overlays` object to `L.control.layers`.

```
// Set object for the overlay maps
var overlays = {
    'countySanitation': countySanitation,
    'cities': cities
}

// Add layer controls
L.control.layers(basemaps, overlays).addTo(map);
```

In the chapter 7, we ended by saying that we strive to make our choropleths map challenge rather than merely inform. This looks like a sketchy map, but it clearly shows the discrepancy of access to basic sanitation services in a large portion of the population.

Finally, it seems changing the position of the `L.controls.layer` from its default top-right position is impossible. For example, setting the position to left, in the `L.controls.layer(basemap, overlaymap, {position: 'topleft'})` results in the UI layers control disappearing completely off the map. This may likely be a yet to be reported bug.

```
knitr::include_graphics(rep('controls-all.jpg'))
```

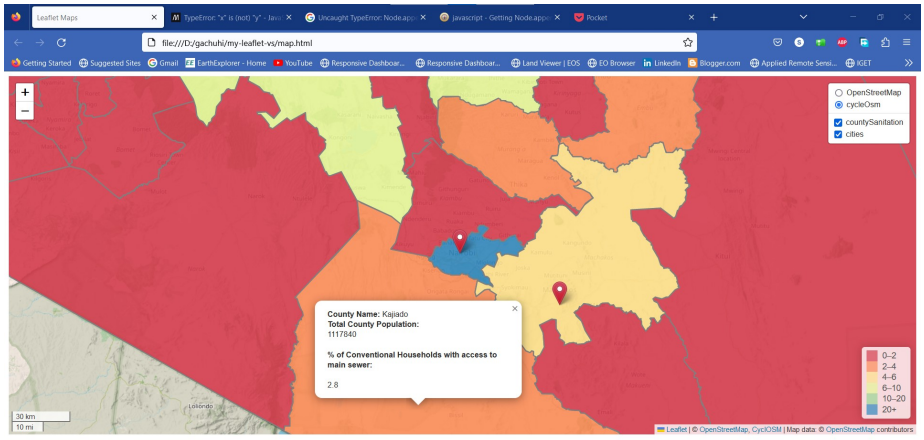


8.5 Add a scale bar

Scales are one of the key elements of any map. One may wonder what's their use in a webmap, but they are useful in estimating size and length of features. Adding a small reactive scale will not hurt!

```
// Add scale
L.control.scale({position:'bottomleft'}).addTo(map);
```

```
knitr::include_graphics(rep('controls-scale.jpg'))
```



All the files and scripts used in this chapter can be accessed here.

8.6 Summary

Unlike paper maps, webmaps provide us the ability to include several basemaps and overlays all in one platform. Here's is what you have learnt from this chapter.

- Several basemaps and overlays can be parsed into Leaflet by holding the basemap and overlay variable names in a JavaScript dictionary object.
- To create a layers control, parse the JavaScript object containing the basemap and overlay names to the `L.control.layers()` class.
- GeoJSON features can also be parsed to the layer control. All that is needed is to save the function calling the GeoJSON file into a variable. For example, the Ajax function `var cities` was parsed to the `L.control.layers()` class.
- Scales are useful for size and length estimation, even in webmaps. Leaflet provides the `L.control.scale` class to add custom scales to your webmap.

Chapter 9

Heatmaps

9.1 What are heatmaps?

Heatmaps are a type of maps that geographically visualize locations with patterns of higher than average occurrence of particular variables say crime, disease, service centres and et cetera.

9.2 Loading the heatmap plugin

Leaflet does not have a prepackaged tool for drawing heatmaps. Instead, we have to use an external plugin by the name of Leaflet.heat. The link to the zipfile is available here. Download and extract the folder to the same directory as your `map.html`.

Once you have extracted the zip file contents, open your `map.html` and add the following `<script>` tag within your `<head>` element.

```
<script src="Leaflet.heat-gh-pages\Leaflet.heat-gh-pages\dist\leaflet-heat.js"></script>
```

Thereafter, we will have to think of a way of loading the GeoJson to a new JavaScript file called `heatmap.js`. This is where we shall write our heatmap code. You may think of using `fetch` or `L.geoJson.ajax` for this purpose but not so fast! The aforementioned methods failed in creating a heatmap. According to the documentation, heatmaps are created using `L.heatLayer`. Parsing `fetch` or `L.geoJson.ajax` into `L.heatLayer` to supposedly create a heatmap out of our GeoJSON points was not working. This is despite the two GeoJSON methods working well for various purposes in previous chapters. Thus, a new approach was needed.

After much internet searching, a code in Stack Overflow came to the rescue. However, it only works after inserting another Ajax plugin. Luckily no download is needed so insert this new `<script>` into your `map.html`.

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script
```

9.3 Creating the Leaflet heatmap

Let's call the usual suspects of adding a basemap.

```
var map = L.map('myMap').setView([0.3556, 37.5833], 6.5);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMa
}).addTo(map);
```

Thereafter, we add the following large code chunks.

```
var geoJsonUrl = "https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/sel

    var geojsonLayer = $.ajax({
    url : geoJsonUrl,
    dataType : 'json',
    jsonpCallback: 'getJson',
    success : console.log("Data successfully loaded!"),
    });

geoJson2heat = ((geojson) => {
        return geojson.features.map(function(feature) {
        return [parseFloat(feature.geometry.coordinates[1]),
                parseFloat(feature.geometry.coordinates[0])];
        });
        });


$.when(geojsonLayer).done(function() {
        // var kill = L.geoJSON(geojsonLayer.responseJSON);
        var layer = geoJson2heat(geojsonLayer.responseJSON, 4);
        var heatMap = L.heatLayer(layer, {
                radius: 40,
                blur: 10,
                gradient: {
                        '0': 'Navy', '0.25': 'Navy',
```

```
                    '0.26': 'Green',
                    '0.5': 'Green',
                    '0.51': 'Yellow',
                    '0.75': 'Yellow',
                    '0.76': 'Red',
                    '1': 'Red'
                  },
            maxZoom: 13});
        map.addLayer(heatMap);
        });
```

To give credit where it is due, the code chunks were taken from this Stack Overflow question and modified a bit. Let's do everyone some justice by going through the preceding code bit by bit.

```
var geoJsonUrl = "https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/selected_hosp

    var geojsonLayer = $.ajax({
    url : geoJsonUrl,
    dataType : 'json',
    jsonpCallback: 'getJson',
    success : console.log("Data successfully loaded!"),
    });
```

The `var geoJsonUrl` holds the link to our hospitals json file. The `var geojsonLayer` uses the Asynchronous Javascript And Xml (Ajax) method to load data from our Github server. If successful, we get the message "Data successfully loaded!" in our browser console.

```
geoJson2heat = ((geojson) => {
        return geojson.features.map(function(feature) {
        return [parseFloat(feature.geometry.coordinates[1]),
                parseFloat(feature.geometry.coordinates[0])];
        });
        });
```

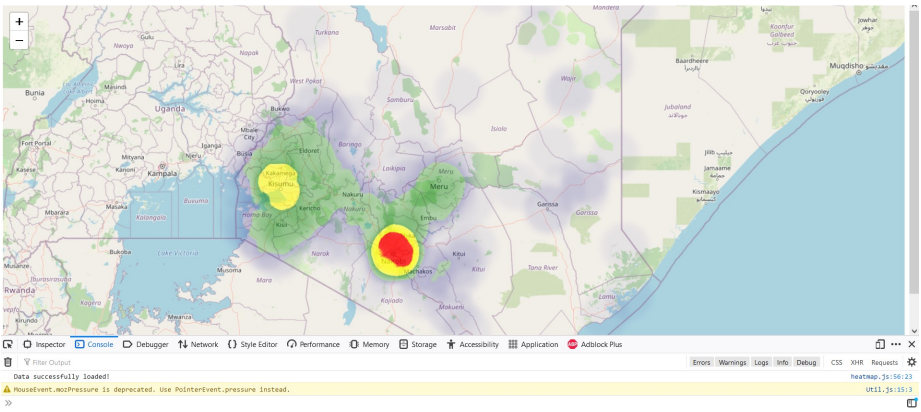
The above code chunk is fairly easy to understand. The function passed to `geoJson2heat` uses the `map` method to iterate over every element, such as cities in the json file and retrieve the longitude and latitude coordinates. Notice the format has been inverted. Rather than [0]… [1] for Lat-Lon, we use [1]…[0].

The final is a jQuery function that will return the desired heatmap layer.

```
$.when(geojsonLayer).done(function() {
        // var kill = L.geoJSON(geojsonLayer.responseJSON);
        var layer = geoJson2heat(geojsonLayer.responseJSON, 4);
        var heatMap = L.heatLayer(layer, {
                radius: 40,
                blur: 10,
                gradient: {
                        '0': 'Navy', '0.25': 'Navy',
                        '0.26': 'Green',
                        '0.5': 'Green',
                        '0.51': 'Yellow',
                        '0.75': 'Yellow',
                        '0.76': 'Red',
                        '1': 'Red'
                      },
                maxZoom: 13});
        map.addLayer(heatMap);
        });
```

Honestly the last was a bit of a stretch since jQuery is hardly used in Leaflet or in normal JavaScript programming nowadays.

```
knitr::include_graphics(rep('heatmap.jpg'))
```



The full code files are available from here.

NB: It should be noted that all manner of efforts were employed to create a heatmap using `L.heatLayer`. The most promising seemed nesting the `L.heatLayer` within the simple `L.geoJson.ajax` class encountered numerous times so far. All attempts, including asking ChatGPT for a plausible solution, were unsuccessful. ## Summary

Heatmaps are one of the simplest forms of maps in geography. They use color and hue to visualize the concentration of phenomena across space. Here are the take home notes from this chapter.

- Heatmaps are a type of maps that geographically visualize locations with patterns of higher than average occurrence of particular variables.
- In order to draw heatmaps using Leaflet, we have to download the `Leaflet.heat` plugin. Thereafter, the path to the plugin is parsed to the `script` tag of your HTML file.
- Apart from using L.heatLayer, the other alternative of creating heatmaps from GeoJSON data is through a combination of Ajax and jQuery.

Chapter 10

Cluster to reduce the clutter

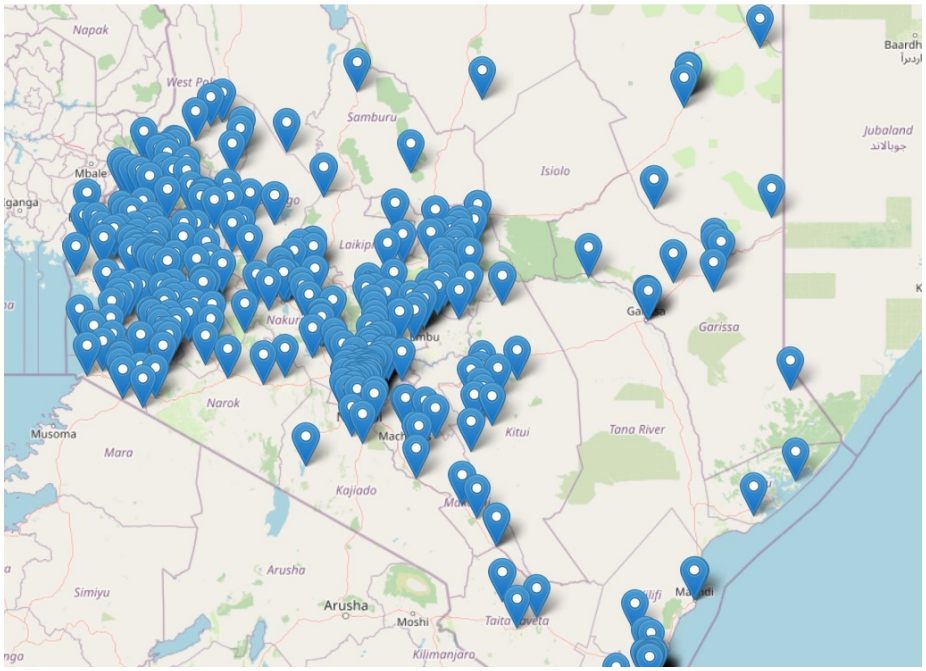
10.1 A map full of clutter

There comes a time when it is convenient to coalesce several points into a single multi-cluster point. Consider the following example: you have a GeoJSON file with over 10000 points of houses within a densely populated island of 5km by 5km, if by any chance such a scenario exists. Will you want to display such a gigantic number of points within such a small area? That would be an overkill! Moreover, the map would be incomprehensible to the viewer. Consider the following example we set up in a new JavaScript file which we fondly called `cluster-markers.js`.

```
var map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
}).addTo(map);


url = 'https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/selected_hospitals.json'

L.geoJson.ajax(url).addTo(map);

var markers = L.markerClusterGroup();
```

The following is the result.

```
knitr::include_graphics(rep('clutter-points.jpg'))
```



Not good at all. The `Leaflet.markercluster` plugin is what transforms a clutter map into one of neatly arranged clustered marker points.

10.2 Preparations

Creating a cluster marker map is fairly easy. You will first have to insert the `Leaflet.markerCluster` plugin into `map.html`. The plugin is available from here. Insert the following `<script>` tag into the `<head>` element of your `map.html`.

```
<script src="Leaflet.markercluster-1.4.1\Leaflet.markercluster-1.4.1\dist\leaflet.marke
```

You will also have to insert the `Leaflet.markercluster` CSS properties via the `<link>` tag too. Add the following `<link>` tag for `Leaflet.markercluster`. For legibility and clean code, place below the other `<link>` properties.

```
<link rel="stylesheet" href="Leaflet.markercluster-master\Leaflet.markercluster-master\
```

Don't underestimate them. These CSS properties are necessary to style your cluster points in a nice way that's easy on the eye. It is also assumed you have already inserted the name of your JavaScript file, the `cluster-marker.js` within the `script` tag enclosed by the `<body>` element of your `map.html` file.

If you had done the small exercise at the beginning of the chapter, the following code should be present.

```
var map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
}).addTo(map);


url = 'https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/selected_hospitals.json'
```

10.3 Behold, a cluster marker map!

Delete the `L.geoJson.ajax(url).addTo(map);`, we won't need it now. Our real work of creating a cluster marker map begins with the `markerClusterGroup` class. Let's proceed!

```
var markers = L.markerClusterGroup();
```

We shall use Ajax again but this time round we shall parse in some functions to customize the appearance and functionalities of our GeoJson markers.

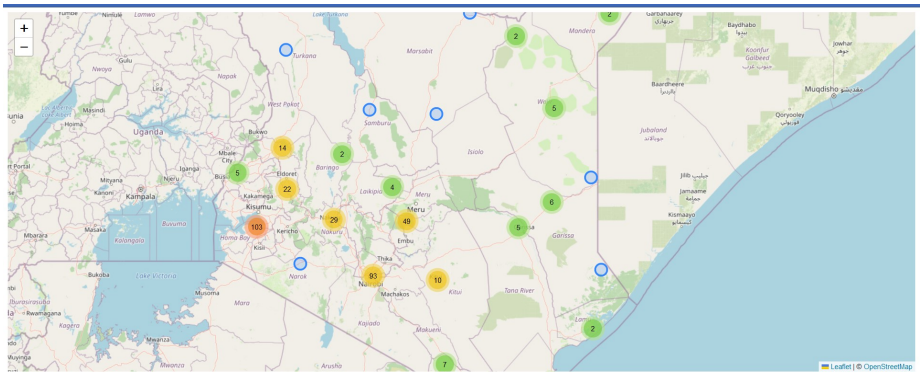
```
L.geoJson.ajax(url, {
    pointToLayer: ((feature, latLng) => {
        return markers.addLayer(L.circleMarker(latLng));
    }),

    onEachFeature: ((feature, layer) => {
        layer.bindPopup(`<b>Facility Name:</b> ${feature.properties.Facility_N} <br>
        <b>Type:</b> ${feature.properties.Type}`)
    })
}).addTo(map);
```

Remember `pointToLayer` of Leaflet GeoJSON files? It defines how the GeoJson file will appear. The `pointToLayer` retrieves the Latitude-Longitude coordinates before finally creating circle markers out of them as enabled by `...return markers.addLayer(L.circleMarker(latLng))`,

How about for `onEachFeature`? You can guess. It simply means– *O*n *E*ach *F*eature, *do this and that*. In our case we bind a popup of facility name and type which will appear when a circle marker is clicked upon.

Actually, the above code sort of finished the work for us.

```
knitr::include_graphics(rep('cluster-marker-map.jpg'))
```



Zoom in and out and watch the circle markers *spidefy* the individual points with a popup. I have restrained from tweaking the markers because the defaults are already good enough. See them from this Github Page.

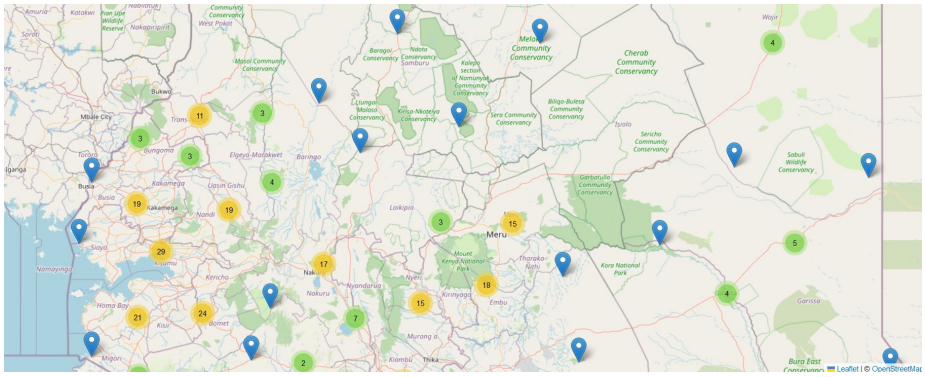
However, as good practice, we need to add the `markers` global variable to the map. We do so by using the following code.

```
map.addLayer(markers)
```

`addLayers`, just like the name suggests, adds the given layer to the map.

The `L.circleMarker` in the `pointToLayer` key of the Ajax function can be replaced with `L.marker()`. Below is how some spiderfied hospital markers look like, but they are less aesthetically pleasing than the circle markers.

```
knitr::include_graphics(rep('cluster-marker-plain.jpg'))
```



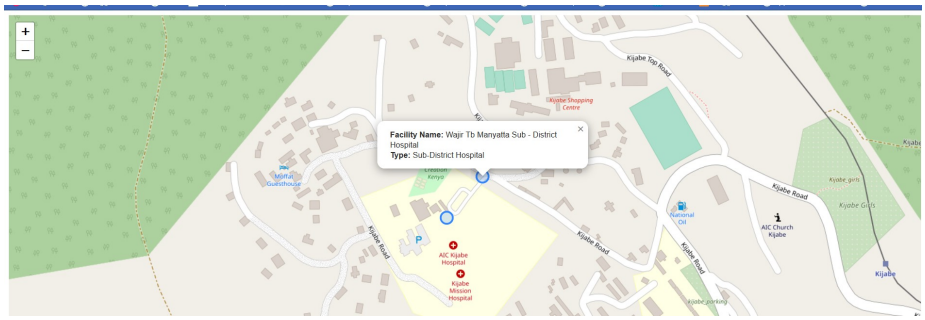
Coming back full circle, circle markers are way better.

Most of the Ajax code was inspired by this video.

But wait! Hold your horses, there is a bug. Try to click on any of the spiderfied or lone hospitals and you will notice something that will raise eyebrows. All hospitals display the following popup when clicked:

```
Facility Name: Wajir Tb Manyatta Sub - District Hospital
Type: Sub-District Hospital
```

```
knitr::include_graphics(rep('cluster-marker-bug.jpg'))
```

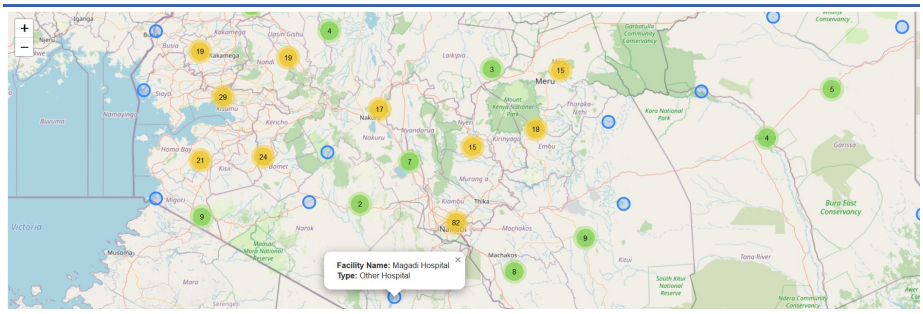


Unless a developer would like to be left with an egg on the face for assigning the same place names to all hospitals, this should be dealt with expeditiously. Our code is alright, since it works in other scenarios such as here. However, it is unnacceptable to assign wrong place names in the world wide web.

Time to try a different strategy: using `fetch` API. We have worked with `fetch` before so we will not explain it that much here. Feel free to google about it as a refresher. Comment out the earlier code beginning from `var markers` to `map.addLayer(markers)` and replace it with the following code chunk.

```
fetch(url)
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        var markers = L.markerClusterGroup();

        var geojsonGroup = L.geoJSON(data, {
            onEachFeature : function(feature, layer){
                layer.bindPopup(`<b>Facility Name:</b> ${feature.properties.Facility_N
                <b>Type:</b> ${feature.properties.Type}`);
            },
            pointToLayer: function (feature, latlng) {
                return L.circleMarker(latlng);
            }
        });

        markers.addLayer(geojsonGroup);
        map.addLayer(markers);

    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

```
knitr::include_graphics(rep('cluster-marker-map-fixed.jpg'))
```



Now all points have their rightful and respective names.

10.4 Additional features of Cluster marker plugin

The official documentation of the plugin lists many other features that come along with the tool. We can't go through all of them but let's surmise just one

important functionality: the `mouseover` event. A `mouseover` event triggers an action when a mouse hovers of a feature in Leaflet.

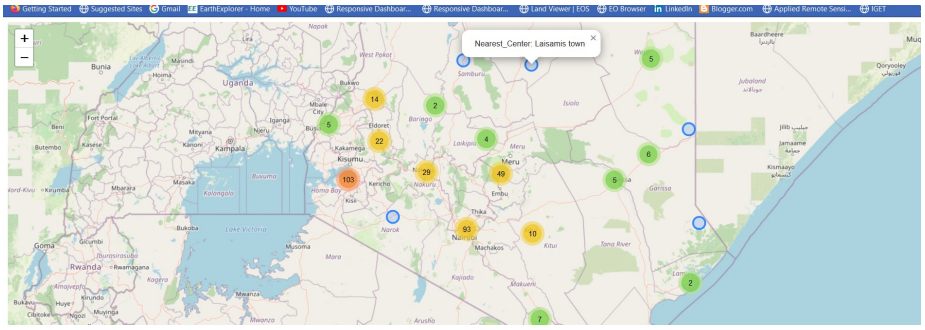
Let's demonstrate adding a hover event to our cluster marker map, thanks to this answer.

```
/////// Added `mouseover` event

fetch(url)
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        var markers = L.markerClusterGroup({chunkedLoading: true}); // Splits the add layers to s

        var geojsonGroup = L.geoJSON(data, {
            onEachFeature : function(feature, layer){
                layer.bindPopup(`<b>Facility Name:</b> ${feature.properties.Facility_N} <br>
                <b>Type:</b> ${feature.properties.Type}`);
            },
            pointToLayer: function (feature, latlng) {
                return L.circleMarker(latlng).on('mouseover', function(){
                    this.bindPopup(`Nearest_Center: ${feature.properties.Nearest_To}`).openPopup(
                });
            }
        });

        markers.addLayer(geojsonGroup);
        map.addLayer(markers);

    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

From the above code, we added the `mouseover` event to each marker point using the `on` method. The `this` keyword in `this.bindPopup()` ties popups of selected variables to coordinates in the `var geojsonGroup` variable. We also sped the rendering of our Leaflet map by adding `chunkedLoading: true` parameter to `L.markerCluster()` function.

Hovering over each marker point will show a value of the `Nearest_To` attribute which is an actual town centre in closest proximity to the hospital marker.

```
knitr::include_graphics(rep('cluster-marker-mouseover.jpg'))
```



The code files used for this chapter are available from here.

10.5 Summary

Cluster marker points are a neat way of displaying several points. Upon zooming out, the spatially sparsed points coalesce to a single unit displaying a figure standing for the number of markers it holds. Upon zooming in, the points spiderfy, (think of spreading out) to their appropriate locations. Here are the lessons from this chapter.

- Clustering points is at times useful for map neatness. In Leaflet, this is made possible with the `Leaflet.markercluster` plugin.
- It is possible to customize how the spiderfied marker points will appear. For example, instead of settling for the default styles of individual markers, we used the circle markers to show individual points.
- It is also possible to add events to the spiderfied marker points. When clicked, the spiderfied marker points can show a popup, or trigger any other event as specified in its code block.

Chapter 11

Mobile Friendly Webapps

11.1 The need for mobile friendly web apps

Short story. Not too long ago I was the proud owner of a famous phone brand on the decline. One time, when taking a photo of the iconic Ngong Hills for Wikipedia's Africa Climate photo contest, the phone just died. That was it. A quick visit to the authorized dealer was greeted with the unbelievable and bemusing words of, "We no longer ship the motherboard to the country anymore." Some healing has taken place, but I was totally heartbroken, and occassionaly suffer some nostalgia of the 'good times' I had with my phone.

Now back to business. Webapps can be heavy, and they can load slowly on smaller devices such as smartphones. Apps that load slow can put off your web app users, so it is prudent to customize your webapp for your user's phones.

For this chapter, we will work on making our cluster marker app mobile friendly. We shall also add other functionalities to make the web app 'heavier' in order to test to destruction if our ambitions of making our app load faster have worked.

In order to create a mobile friendly Leaflet experience, insert the below code within the `<head>` element of your `map.html`. The below `meta` tag tells the browser to disable unwanted scaling of the page and instead set it to its actual size.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-sca
```

11.2 The basemaps

If you have gone through Chapter 8 where we created controls, the following will look familiar. We will add some basemaps and later on create their control

widgets.

```
// Basemaps
var osm = L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap
});

var cyclOSM = L.tileLayer('https://{s}.tile-cyclosm.openstreetmap.fr/cyclosm/{z}/{x}/{y
    maxZoom: 20,
    attribution: '<a href="https://github.com/cyclosm/cyclosm-cartocss-style/releases"
}); // the CyclOSM tile layer available from Leaflet servers
```

Let's add our basemaps to Leaflet.

```
// Add the Leaflet basemaps
var map = L.map('myMap', {
    layers: [osm, cyclOSM]
}).setView([-1.295287148, 36.81984753], 7);
```

11.3 Adding the features

Remember our hospital json layer? Let's call it again and transform it to a cluster marker with `fetch`.

```
// Add hospital dataset

url = 'https://raw.githubusercontent.com/sammigachuhi/geojson_files/main/selected_hospi

var cluster = fetch(url)
    .then((response) =>{
        return response.json()
    })
    .then((data) => {
        var markers = L.markerClusterGroup();

        var geojsonGroup = L.geoJSON(data, {
            onEachFeature : function(feature, layer){
                var popupContent =  `<b>Facility Name:</b> ${feature.properties.Facilit
                <b>Type:</b> ${feature.properties.Type}`;
                layer.bindPopup(popupContent)
            },
            pointToLayer: function (feature, latlng) {
```

```
                return L.circleMarker(latlng);
            }
        });

        markers.addLayer(geojsonGroup);
        map.addLayer(markers);

    })
    .catch((error) => {
        console.log(`This is the error: ${error}`)
    })
```

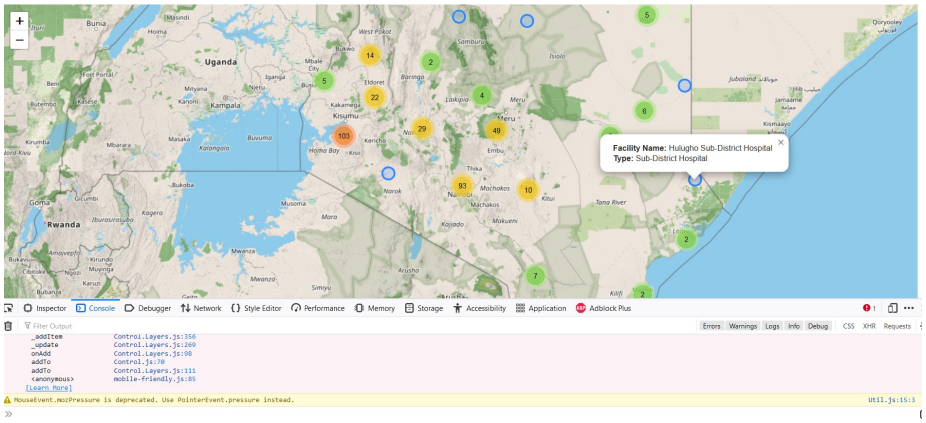
Why was the `fetch` code being parsed to `var cluster`? Well, we were aiming for the stars. We wanted to have a layer control for our `cluster` variable too but unfortunately this plan failed.

Let's put our `basemaps` and `cluster` variables into JavaScript objects in order to create a layer control for each.

```
// Set object for the basemaps
var basemaps = {
    "OpenStreetMap": osm,
    'cycleOsm': cyclOSM,
}

////Don't add the 'overlays' object. For demonstration purposes only
// Set object for the overlay maps
var overlays = {
    'Hospitals': cluster
}
```

Before you head on any further, inserting the `overlays` object into the `L.control.layers()` class results in several errors. This is why we were unable to create a control for the markers held in `var cluster`. The image below shows the errors appearing in the console after inserting the `overlays` object into `L.control.layers()`.

```
knitr::include_graphics(rep('mobile-friendly-error.jpg'))
```



To get rid of the error showcased above, just comment out the `overlays` object and remove it from `L.control.layers()`. The `L.control.layers()` class should only contain the `basemap` object.

11.4 Zooming to the mobile user's location

According to the Leaflet official documentation, Leaflet has a handy shortcut of zooming in to the user's location. If for some reason it will not pinpoint the exact coordinates, it will create a buffer around the mobile user's approximate location.

```
// Zoom to your location
map.locate({setView: true, maxZoom: 16});
```

11.5 Add marker to mobile user's geolocation

Even if the location is off by a couple of miles, at least a marker to show the triangulated position will help. At least you will not be *all over* the map! The following code adds a marker to the mobile user's triangulated Latitude-Longitude coordinates, and displays a message showing the radius in which the mobile user is most likely to be found from the marker point.

```
// Add marker at your location
function onLocationFound(e) {
    var radius = e.accuracy;
```

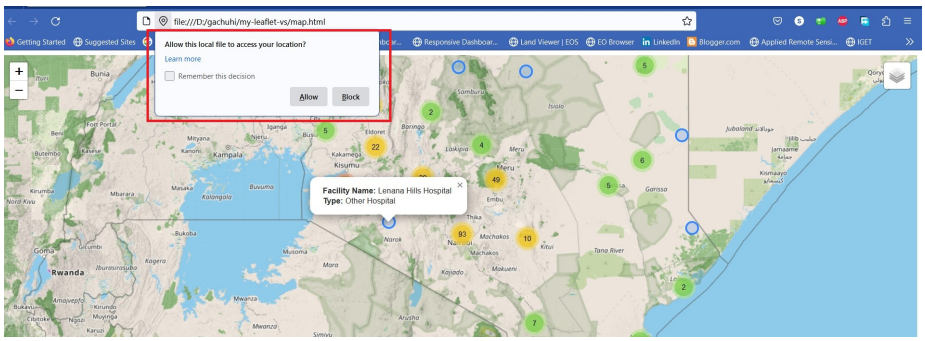
```
    L.marker(e.latlng).addTo(map)
        .bindPopup("You are within " + Number((radius/1000).toFixed(2)) + " kilometers from this

    L.circle(e.latlng, radius).addTo(map);
}
```

Incase you forgot, the `on` method adds listeners. As a gentle reminder, listeners are codes that run when an event is triggered, such as the simple hovering of a mouse over a feature. In the below code, the listener `'locationfound'` triggers the `onLocationFound` function in case Leaflet successfully approximated the user's location.

```
map.on('locationfound', onLocationFound);
```

The `locationfound` listener is responsible for the message bounded in red below when a browser loads a Leaflet map. Clicking **Allow** will give the browser the heads up to zoom to the user's location as it best can.

```
knitr::include_graphics(rep('location-found.jpg'))
```



What if, getting the mobile user's geolocation is unsuccessful? We will create a function that outputs the error event to our console, as shown below.

```
// Error displayed after finding location failed
function onLocationError(e) {
    alert(e.message);
}
```

Actually, `message` is an error event that displays the error message of a parameter. The `message` event is parsed to the `onLocationError` function. If the browser fails to approximate the user's location, the `onLocationError` function returns 'true' which triggers an error alert on the browser.

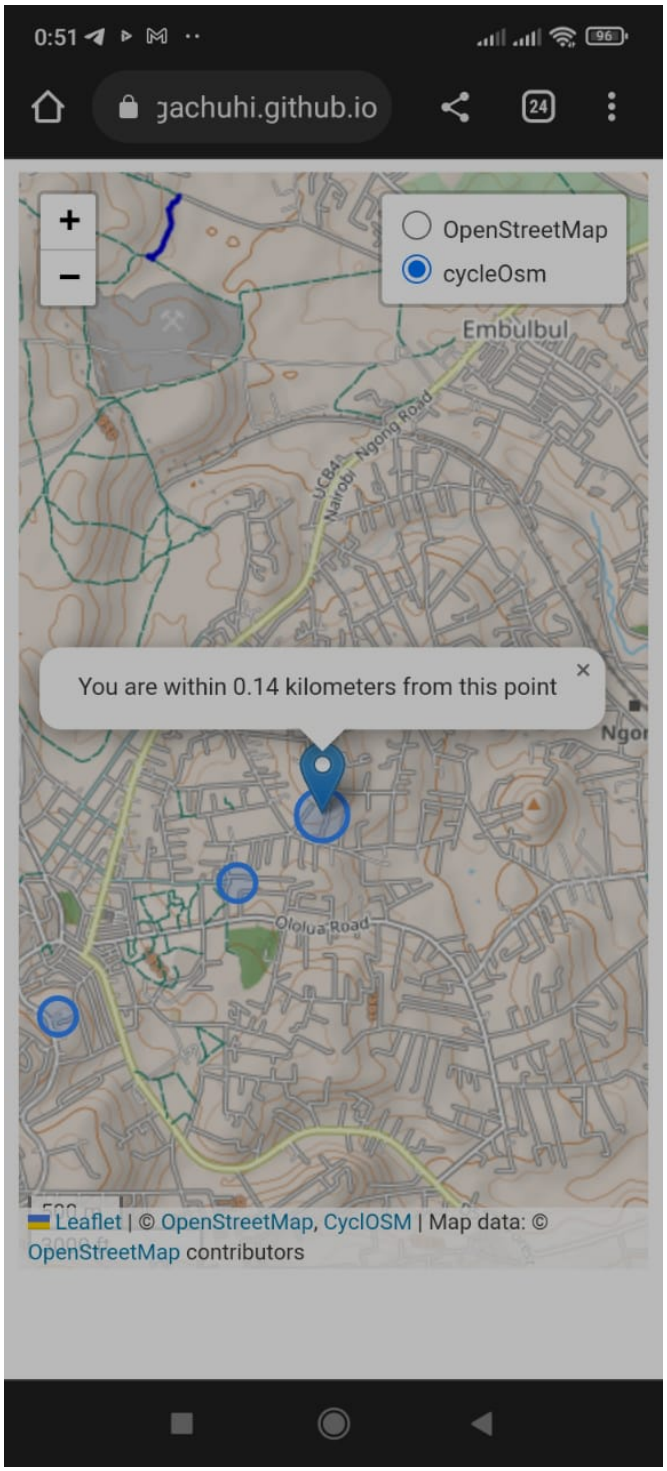
```
map.on('locationerror', onLocationError);
```

11.6 The mobile webmap app

Yours truly has saved you the hustle of detailing how this chapter's files have been saved to Github and subsequently converted to a webpage. The key thing to note is that the HTML file has to be named as `index.html` and not any other name such as `map.html` since the name 'index' is the easiest way to render a file on the fly on Github. Some additional steps exist to launch our Leaflet map to the global web but to ensure brevity in this chapter, we suggest you visit this authoritative Github page for further guidance. The link below should nevertheless allow you to view the webapp on your phone.

https://sammigachuhi.github.io/hospitals_webapp/

```
knitr::include_graphics(rep('mobile-app.jpg'))
```


0:51
gachuhi.github.io
OpenStreetMap
cycleOsm
Embulbul
Ngong Road
You are within 0.14 kilometers from this point
Ololua Road
Leaflet | © OpenStreetMap, CyclOSM | Map data: © OpenStreetMap contributors

We had initially aimed for the stars by wanting to create a web app that in addition to the basemap layers, it would also feature some layer controls. However, it seemed like we landed on the skies instead. Nevertheless, this looks like a good hospital locations app. The sky is only the baseline for what further features can be built on top of this app.

The full code script is available from here.

11.7 Summary

Enabling a Leaflet map to be mobile friendly allows the Leaflet map to load fast as well as scale efficiently on a smartphone. Here is what you've learnt.

- A special `meta` tag is inserted in the `<head>` element of your HTML file to enable the browser scale smoothly when a user is viewing a Leaflet map on a smartphone.
- Leaflet has a special function, the `map.locate` that geolocates and zooms to the user's exact coordinates. If for some reason precision fails, it creates a buffer around the mobile user's approximate location.
- In case Leaflet is unable to approximate the user's location, one can resort to the `message` event which throws back an error on the browser.

Chapter 12

Web Map Service Layers

12.1 What are Web Map Service (WMS) Layers?

A Web Mapping Service (WMS) consists of geospatial data hosted through the internet with standards set by the Open Geospatial Consortium (OGS).

A WMS enables the exchange of spatial information and viewing over the web in the form of a map or image to your browser. The most common formats of Web Mapping Services are Web Map Services (WMS), Web Feature Services (WFS), Web Coverage Services (WCS), Web Processing Services (WPS), Web Map Tile Services (WMTS), and Web Coverage Processing Services (WCPS). However, the Web Map Services (WMS) layer is the most used. It offers basic panning, zooming and quick rendering speeds.

12.2 Loading a WMS server

To load a WMS layer into Leaflet, we use the `L.tileLayer.wms` class. As simple as that. So let's set up our Leaflet map. Create a new JavaScript file named `wms_layers.js` and insert the following code:

```
let map = L.map('myMap').setView([-1.295287148, 36.81984753], 7);

let tileLayer = L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
});
```

We have added an OSM layer because we want to create a layer control widget that also contains WMS layers so that you can switch between an Open Street Map (OSM) layer and a WMS. For this exercise, we shall add two WMS layers.

```
let wmsLayerTopo = L.tileLayer.wms('https://www.gmrt.org/services/mapserver/wms_merc_ma
    layers: 'topo',
    format: 'image/png'
})

let wmsLayerTopomask = L.tileLayer.wms('https://www.gmrt.org/services/mapserver/wms_mer
    layers: 'topo-mask',
    format: 'image/png'
})
```

12.3 Adding WMS to layer control

Let's insert the above web map layers into a JavaScript object before parsing it to a layer control.

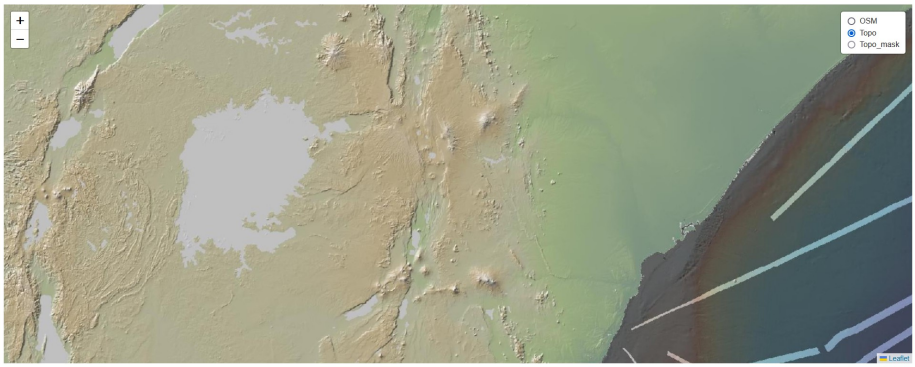
```
var basemaps = {
    OSM: tileLayer,
    Topo: wmsLayerTopo,
    Topo_mask: wmsLayerTopomask,
}
```

Now let's parse the object into a control. If you did Chapter 8 then this should look familiar.

```
L.control.layers(basemaps).addTo(map);

basemaps.Topo.addTo(map);  // To have the Topo as the default map layer
```

The last line `basemaps.Topo.addTo(map)` serves to set up the default layer that will appear when the map is loaded. In this case it is the `Topo` key in `var basemaps`. If this last line is omitted, Leaflet will only show a blank canvas unless one of the radioitems is selected.

```
knitr::include_graphics(rep('wms_layer.jpg'))
```



And that's it!

Even though we wanted to work with African-based WMS layers, of which Digital Earth Africa provides plenty of them, for some reason this was not possible despite diligently following their documentation. It is quite ironic that they can be easily loaded in Qgis and not Leaflet.

Interested in getting other WMS layers? Go to Spartineo.com.

The files used in this chapter are available here.

12.4 Summary

A Web Mapping Service (WMS) layer in many respects looks and acts like a basemap. Just like Open Street Map and other similar servers, WMS layers host data through the internet. Here are the take homes from this chapter.

- A Web Mapping Service consists of geospatial data hosted through the internet with standards set by the Open Geospatial Consortium (OGS).
- The most common formats of Web Mapping Services are Web Map Services (WMS), Web Feature Services (WFS), Web Coverage Services (WCS), Web Processing Services (WPS), Web Map Tile Services (WMTS), and Web Coverage Processing Services (WCPS).
- The `L.tileLayer.wms` class is used to load WMS layers into Leaflet.
- Just like any other basemap, one can parse WMS layer into Leaflet controls, as seen with `wmsLayerTopo` and `wmsLayerTopomask`.

Chapter 13

Standard Website with Leaflet Project

13.1 Get the HTML Template

Back in Chapter 4 we inserted a Leaflet map into a pale looking website. Taking that exercises a little further, we would like to demonstrate how Leaflet can be added to a standard, professional looking HTML website. This chapter will train you how to insert a Leaflet map into your company's, client's or even colleague's website.

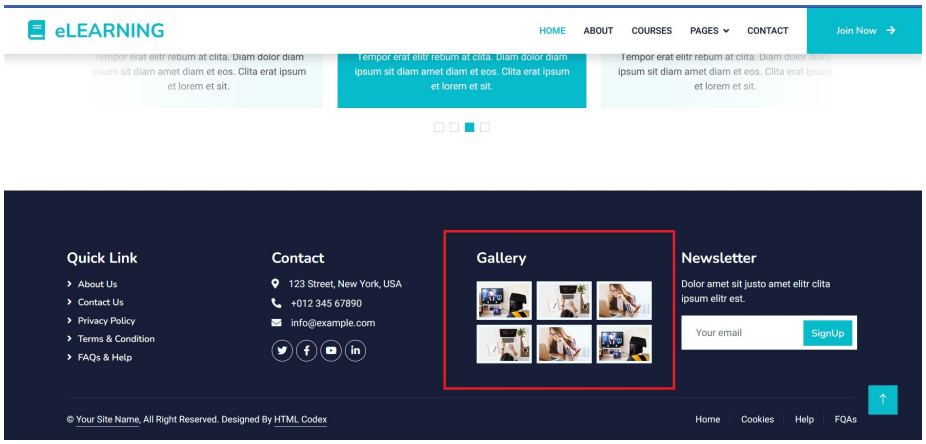
The template used in this exercise was acquired from Free CSS.com and is available here. Extract the files and take a look at the HTML documents. That is, the `index.html`, `about.html`, `contact.html`, `courses.html`, `team.html` and `testimonial.html` files. Pay special attention to the `<head>` element and the `<div>` containing the Gallery section of the website. The Gallery `<div>` looks like so:

```
<div class="col-lg-3 col-md-6">
                    <h4 class="text-white mb-3">Gallery</h4>
                    <div class="row g-2 pt-2">
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-1.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-2.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-3.jpg" alt="">
                        </div>
```

```
                    <div class="col-4">
                        <img class="img-fluid bg-light p-1" src="img/course-2.jpg"
                    </div>
                    <div class="col-4">
                        <img class="img-fluid bg-light p-1" src="img/course-3.jpg"
                    </div>
                    <div class="col-4">
                        <img class="img-fluid bg-light p-1" src="img/course-1.jpg"
                    </div>
                </div>
            </div>
```

Here is where the Gallery in the eLearning webpage is situated (bounded in red).

```
knitr::include_graphics(rep('gallery.jpg'))
```



Why do we want to change the gallery, or rather, to insert a Leaflet map in its place? It seemed a good idea to replace the pretty looking pictures with a webmap since this website is for demonstration purposes only.

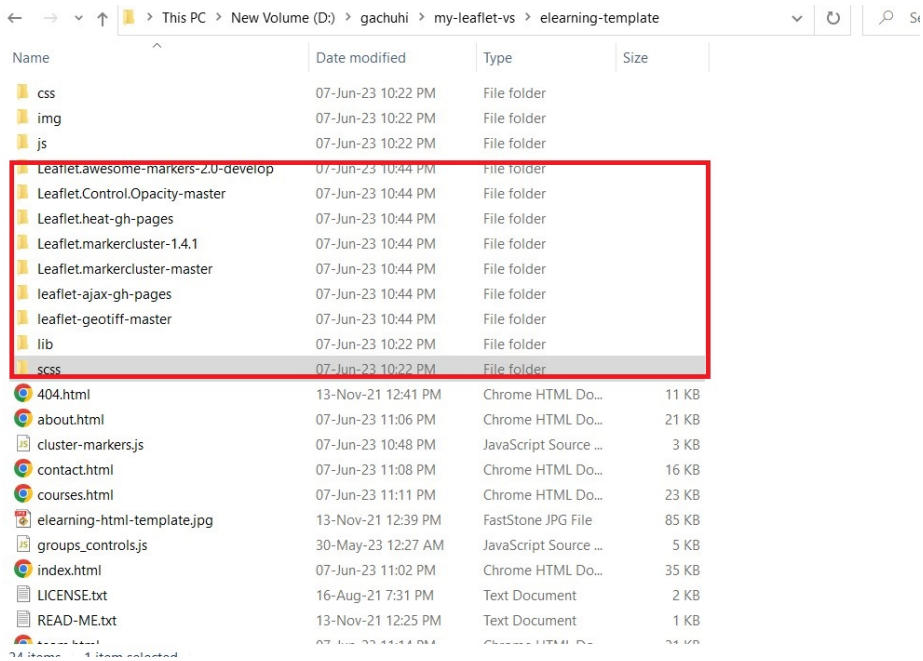
13.2 Embed Leaflet to standard html website

Since we want to integrate a Leaflet map to the eLearning website, we will have to load the requisite Leaflet links and scripts to the eLearning dummy website's `<head>` element, just like we used to do to our `map.html`. Thereafter, we shall replace the entire gallery section with just one `<div>` that references one of our many Leaflet JavaScript files.

Ready? Let's go.

Assuming you are working on VSCode, copy all the plugin files for Leaflet, such as the `Leaflet.markercluster-master` folder and all the others you have extracted at one point or another in previous chapters. The names of these folders are enclosed with a red border in the image below.

```
knitr::include_graphics(rep('elearning-plugins.jpg'))
```



Why are we ensuring that the Leaflet plugin folders are in the same directory as our eLearning template files? It is because we want to avoid the need of tinkering with the relative path, which has to change if the (plugin) folders were located in a different directory from that of our eLearning website.

Go to the `index.html` file of your eLearning template. From the `<head>` to `</head>` tags, replace the existing code with the following:

```
<meta charset="utf-8">
    <!-- For leaflet and mobile compatibility -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user

    <title>eLEARNING - eLearning HTML Template</title>
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <meta content="" name="keywords">
    <meta content="" name="description">

    <!-- Favicon -->
```

```
<link href="img/favicon.ico" rel="icon">

<!-- Google Web Fonts -->
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Heebo:wght@400;500;600&family=

<!-- Icon Font Stylesheet -->
<link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.10.0/css/all.min
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.4.1/font/bootstrap-icons

<!-- Libraries Stylesheet -->
<link href="lib/animate/animate.min.css" rel="stylesheet">
<link href="lib/owlcarousel/assets/owl.carousel.min.css" rel="stylesheet">

<!-- Customized Bootstrap Stylesheet -->
<link href="css/bootstrap.min.css" rel="stylesheet">

<!-- Template Stylesheet -->
<link href="css/style.css" rel="stylesheet">

<!-- For leaflet -->
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.3/dist/leaflet.css"
    integrity="sha256-kLaT2GOSpHechhsozzB+flnD+zUyjE2LlfWPgU04xyI="
    crossorigin=""/>
<link rel="stylesheet" href="Leaflet.Control.Opacity-master\Leaflet.Control.Opacity
<!-- For Leaflet marker clusters -->
<link rel="stylesheet" href="Leaflet.markercluster-master\Leaflet.markercluster-mas

<!-- For leaflet scripts -->
<script src="https://unpkg.com/leaflet@1.9.3/dist/leaflet.js"
    integrity="sha256-WBkoXOwTeyKclOHuWtc+i2uENFpDZ9YPdf5Hf+D7ewM="
    crossorigin=""></script>
<script src="leaflet-ajax-gh-pages\dist\leaflet.ajax.js"></script>
<script src="leaflet-ajax-gh-pages\dist\leaflet.ajax.min.js"></script>
<script src="leaflet-ajax-gh-pages\example\leaflet.spin.js"></script>
<script src="leaflet-ajax-gh-pages\example\spin.js"></script>
<script src="Leaflet.heat-gh-pages\Leaflet.heat-gh-pages\dist\leaflet-heat.js"></s
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></s
<!-- For leaflet cluster markers -->
<script src="Leaflet.markercluster-1.4.1\Leaflet.markercluster-1.4.1\dist\leaflet.
<script src="Leaflet.markercluster-1.4.1\Leaflet.markercluster-1.4.1\dist\leaflet.
```

The first alteration we will do to our eLearning website's HTML folder is to add the relevant Leaflet `<link>` and `<script>` tags. These `<link>` and `<script>`

tags have just been copy pasted from the `map.html` file we have been dealing with so far.

Now that we have the requisite `<link>` and `<script>` tags to embed Leaflet into our eLeaflet demo website, we only need to insert the path to our JavaScript file containing the Leaflet code. The JavaScript is of course responsible for firing up the Leaflet map in our eLearning website. For this exercise, we shall use the `cluster_markers.js` file we have had so much fun with. Copy paste this file into the same folder containing your eLeaflet website's folders.

Alright.

Time to add the leaflet map. Where? At the gallery section.

As a reminder, it looks like this:

```
<div class="col-lg-3 col-md-6">
                    <h4 class="text-white mb-3">Gallery</h4>
                    <div class="row g-2 pt-2">
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-1.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-2.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-3.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-2.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-3.jpg" alt="">
                        </div>
                        <div class="col-4">
                            <img class="img-fluid bg-light p-1" src="img/course-1.jpg" alt="">
                        </div>
                    </div>
                </div>
```

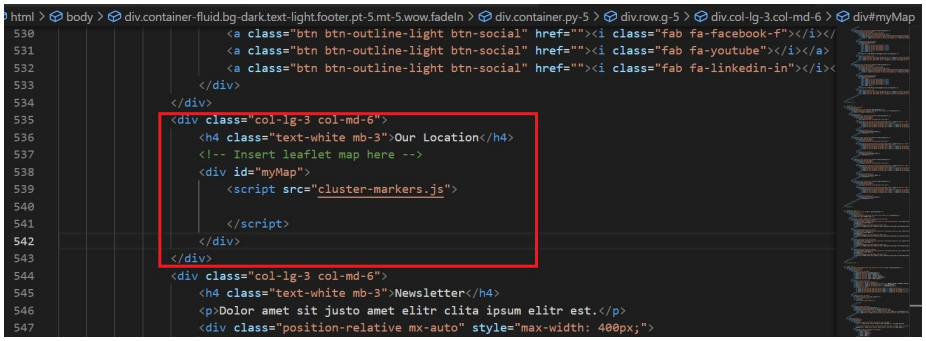
You may have to scroll *really* down to locate it. Right under the `<h4 class="text-white mb-3">Gallery</h4>` tag, replace it with the following code, all the way to the `<div>` tag after `<img class="img-fluid bg-light p-1" src="img/course-1.jpg" alt="">`.

```
<div id="myMap">
                        <script src="cluster-markers.js">
```

```
            </script>
        </div>
```

Perhaps an image will clarify matters a bit. This is where you are to insert the path to your `cluster-markers.js` file.

```
knitr::include_graphics(rep('div_replace.jpg'))
```



Also as a heads up, remember to replace the `<h4>` tags with the statement `Our Location`. It's no longer a gallery but a webmap!

As if that was not enough, we also have to set the CSS properties for our Leaflet map.

13.3 Editing the CSS

Open the `style.css` file of your eLearning template in your text editor (such as VS code). The path to the CSS file is as follows:

```
elearning-template\css\style.css
```

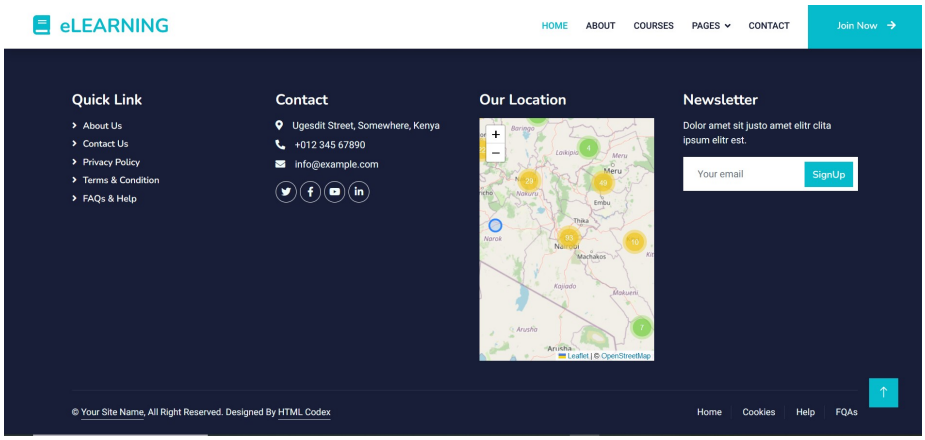
Scroll down to the very bottom of `style.css` and paste the following CSS properties for our Leaflet map.

```
#myMap {
    height: 400px;
    width: auto;
}
```

Remember the `#` and the text that follows references the specific name of that `ID` attribute from the countless HTML tags in `index.html`. The above CSS selector references the `<div>` with the ID– `#myMap`.

After reloading `index.html`, this is how our Leaflet map looks inside the eLearning template.

```
knitr::include_graphics(rep('location_elearning.jpg'))
```



Good.

13.4 Embedding Leaflet to every webpage

Now click the **About**, **Courses**, **Pages/Our Team**, **Pages/Testimonial** and **Contact** web pages. Is the Leaflet map there in each case? No. It's the same ol' gallery. Imagine you will have to replace the `<head>` and Gallery `<div>` elements of the respective `about.html`, `courses.html`, `team.html`, `testimonial.html` and `contact.html` webpages here as well with the relevant Leaflet paths! Luckily, since you have hopefully got the gist of it, the following link contains the eLearning folder with Leaflet map embedded in all the aforementioned webpages.

elearning_template_demo

13.5 Posting the Html website to the world

This chapter will leave out the details of saving your files to Github. Nevertheless, as mentioned in Chapter 11, the name `index` in your HTML file is automatically considered the root HTML file by Github. With that said, here is the link to our eLearning website with a Leaflet webmap embedded therein.

eLearning website

13.6 Summary

Having the necessary Leaflet folders in the same directory as your HTML website's files prevents the Leaflet map from failing to load due to a broken path. Here are other lesson's we have encountered.

- In order to embed a Leaflet map into your website, you have to load the requisite Leaflet `<link>` and `<script>` tags into your `<head>` and where necessary, `<div>` elements.
- Insert the JavaScript code that fires up your Leaflet using the `<script>` tags. This was the method used to insert the JavaScript file `cluster-markers.js` into the eLeaflet dummy website.
- It is also important to ensure your Leaflet map appears in the right place in all the child webpages. For example, Leaflet also had to be embedded in the webpages of `about.html`, `courses.html`, `team.html`, `testimonial.html` and `contact.html` at the Gallery section just as it was done for their parent file– `index.html`.

Chapter 14

ESRI and Leaflet

14.1 An overview of ESRI

Alright. If you are a GIS practitioner, you have probably heard about ESRI, one of the world's leading geospatial software and services provider. You (might) have also come across various ESRI basemap servers, such as ESRI topographic, ESRI streets and ESRI Imagery layers. They also have plugins that allow Leaflet users to access the ArcGIS functionalities. For example, the ESRI plugins for Leaflet allow you to access some ESRI basemaps and products, while also allowing you to become an ESRI ArcGIS JavaScript developer.

To use ESRI Leaflet, you have to create an ArcGIS Developer account and also get an API key. Kindly do so before proceeding.

14.2 ESRI Leaflet plugins

As we had mentioned earlier, you need ESRI Leaflet plugins to experience all the ArcGIS functionalities possible with your Application Programming Interface (API) key. Remember your `map.html` file? To experience Leaflet in ESRI, add the following `<script>` tags to the `<head>` element of your `map.html` file.

```
<!-- Load Esri Leaflet from CDN -->
        <script src="https://unpkg.com/esri-leaflet@3.0.10/dist/esri-leaflet.js"></script>
        <script src="https://unpkg.com/esri-leaflet-vector@4.0.2/dist/esri-leaflet-vector.js"></s
```

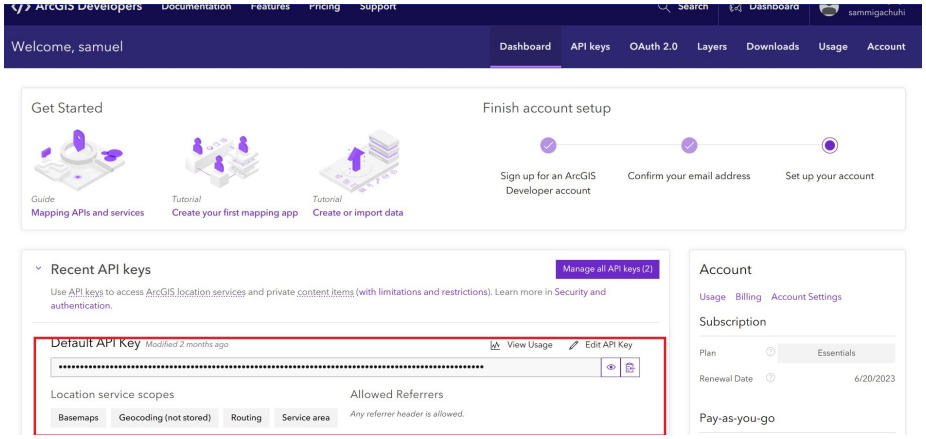
Create a new JavaScript file called `esri_leaflet.js` and get ready to enjoy ESRI services!

14.3 Creating an ESRI Leaflet map

On your blank `esri_leaflet.js` file, create a variable to store your API Key. This key is important to access your ESRI benefits, much like the magic phrase "Open Sesame" which would open a doorway to a cave full of treasures in the legendary story of *Ali Baba and the Forty Thieves*!

Below is an example of how to access your API key from your ArcGIS Developer account.

```
knitr::include_graphics(rep('esri_leaflet_api.jpg'))
```



Copy paste that key to your `esri_leaflet.js` file.

```
const apiKey = "Your key";
```

Let's proceed to create a basemap. To create one, just create a variable called `basemapEnum` that stores the ESRI basemap identifier. In ESRI Leaflet, a basemap is called by parsing the provider name and the desired style, like so: `{Provider}:{Style name}` or `{Provider}:{Style name}:{Component}`. In our case, we want the ArcGIS streets basemap layer.

```
const basemapEnum = "ArcGIS:Streets";
```

Alright. It's about time we fired up our mapping power. Create an ESRI Leaflet map instance much like we have always been doing.

```
const map = L.map("myMap", {
    minZoom: 2
}).setView([0.3556, 37.5833], 6.5);
```

We had created a basemap variable earlier, so we will parse it to the `L.esri.Vector.vectorBasemapLayer` class which is responsible for creating basemaps. Actually one can also parse the `{Provider}:{Style name}` to the `L.esri.vector.vectorBasemapLayer()` class but our method of using variable names looks cleaner.

```
L.esri.Vector.vectorBasemapLayer(basemapEnum, {
    apiKey: apiKey,
}).addTo(map);
```

Now open your `map.html` file. It should look like below.

```
knitr::include_graphics(rep('esri_leaflet_basemap.jpg'))
```

Now imagine there are other basemap styles as shown in this webpage.

14.4 Geocode search

Of course there are other ESRI Leaflet functionalities. Out of curiousity, a brief search was made to find out an ESRI plugin that loads raster layers in the hope it would be simpler in operation compared to the georaster plugin of Leaflet. Although both have a plugin for downloading `.jpeg` and `.png` image formats, none is good enough for loading geospatial raster files such as `.tiff`. Nevertheless, there are other cool services that ESRI Leaflet offers, such as geocode search[1].

For the rest of this chapter, we shall create a geocoding service using ESRI Leaflet.

[1] Geocoding is the process of converting address or place text into a location. The geocoding service provides address and place geocoding as well as reverse geocoding

14.5 Add search bar

Add the following `<link>` and `<script>` to your `<head>` element.

```
<!-- Load Esri Leaflet Geocoder from CDN -->
        <link rel="stylesheet" href="https://unpkg.com/esri-leaflet-geocoder@3.1.4/dist
        <script src="https://unpkg.com/esri-leaflet-geocoder@3.1.4/dist/esri-leaflet-g
```

We will first add a search control widget to the topright of our ESRI Leaflet map. Search control widgets are created using the `L.esri.Geocoding.geosearch` class. In the below code, we have parsed the `L.esri.Geocoding.geosearch` to the variable `searchControl`.

```
const searchControl = L.esri.Geocoding.geosearch({
        position: "topright",
        placeholder: "Enter an address or place e.g. 1 York St",
        useMapBounds: false,

      }).addTo(map);
```

Just like in Leaflet, ESRI Leaflet constructors have their options. Please refer to the options for this specific constructor here.

Another small thing; change your `const basemapEnum` to read `ArcGis:Navigation`.

```
const basemapEnum = "ArcGIS:Navigation";
```

Upon refreshing your `map.html` file, apart from having a new basemap style, you will see a new search bar at the top right. However, it is non-functional. It takes us nowhere.

```
knitr::include_graphics(rep('search_bar.jpg'))
```

14.6 Make the search bar functional

In order to make the search bar do what it says, we will set the value of the `providers` key to a `arcgisOnlineProvider`. The latter is instantiated with the constructor `L.esri.Geocoding.arcgisOnlineProvider`. ESRI has various open source tutorials on its platform that provide succinct and concise explanation for most of its tools.

```
const searchControl = L.esri.Geocoding.geosearch({
    position: "topright",
    placeholder: "Enter an address or place e.g. 1 York St",
    useMapBounds: false,

    // Add provider
    providers: [
        L.esri.Geocoding.arcgisOnlineProvider({
          apikey: apiKey,
          nearby: {
            lat: 0.3556,
            lng: 37.5833
          }
        })
      ]

  }).addTo(map);
```

Now try your search bar again. Some location names appear as you type them in, and if you press **Enter** it takes you to the exact location. This is good except for one thing: there is no marker to pinpoint that specific address. The following section outlines how to enable ESRI create a marker on the fly at any searched location name.

```
knitr::include_graphics(rep('searchable.jpg'))
```

14.7 Adding an auto-generated location pin

First add a layer group to store the geocoding results.

```
const results = L.layerGroup().addTo(map);
```

We will create an event handler to access the `data` from the search results. We shall also add a `clearLayers` call to remove the previous data from the layer group. This is beginning to sound complicated, but if the reader desires further clarity, they should refer to the ESRI Leaflet geocoding tutorial.

```
searchControl.on("results", (data) => {
        results.clearLayers();

      });
```

The following loop adds the coordinates of the searched location to the marker. This loop goes into the `searchControl` code block.

```
for (let i = data.results.length - 1; i >= 0; i--) {
          const marker = L.marker(data.results[i].latlng);

          results.addLayer(marker);

        }
```

The `lngLatString` variable below will store the rounded coordinates, and our familiar `bindPopup` and `openPopup` will show the coordinates and our address.

```
const lngLatString = `${Math.round(data.results[i].latlng.lng * 100000) / 100000}, ${
            Math.round(data.results[i].latlng.lat * 100000) / 100000
          }`;
          marker.bindPopup(`<b>${lngLatString}</b><p>${data.results[i].properties.Longl

          results.addLayer(marker);

          marker.openPopup();
```

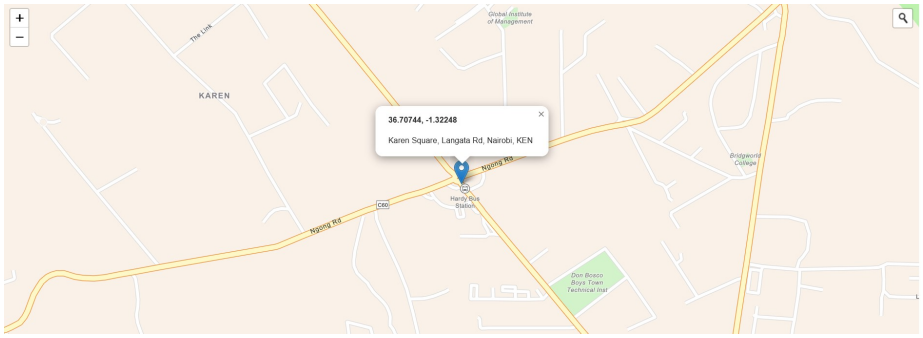
The entire `searchControl` code block should look like below.

```
searchControl.on("results", (data) => {
    results.clearLayers();
```

```
    for (let i = data.results.length - 1; i >= 0; i--) {
        const marker = L.marker(data.results[i].latlng);

        const lngLatString = `${Math.round(data.results[i].latlng.lng * 100000) / 100000}, ${
            Math.round(data.results[i].latlng.lat * 100000) / 100000
          }`;
          marker.bindPopup(`<b>${lngLatString}</b><p>${data.results[i].properties.LongLabel}</p>`

        results.addLayer(marker);

        marker.openPopup();

      }

  });
```

So now when you search for a particular place in the search bar and press **Enter**, a marker and with a pop up showcasing the address and longitude-latitude coordinates of the location should appear.

```
knitr::include_graphics(rep('search_success.jpg'))
```



The ArcGIS Developers website provides several more tutorials on using ESRI Leaflet.

Here are the files used in this exercise.

14.8 Summary

Using ESRI Leaflet brings many of the ArcGIS functionalities to the open source Leaflet plugin.

Here are the lessons learnt in this chapter.

- An ArcGIS Developer account is needed to use ESRI Leaflet features.
- An Application Programming Interface (API) key serves like a real life key in granting the user access to ESRI Leaflet functionalities and features.
- The standalone Leaflet and ESRI Leaflet are similar in many respects. ArcGIS features can be built on top of Leaflet via various ESRI Leaflet constructors such as `L.esri.Geocoding.arcgisOnlineProvider`.

Chapter 15

Conclusion

Now here comes the end of our book. If you have reached this far, then congratulations! You now have the power to continue using Leaflet to create cool webmaps, even far better than those you've encountered in this book.

I hope this book has been helpful. For anything that you found difficult to understand, or the text lacked sufficient clarity, it had nothing to do with you. Totally. The author wishes they could explain everything concisely, succinctly, sweetly and clearly as though they were composing a poem. Nevertheless, it is the author's desire he could intuitively explain concepts from the computing world as easily as he could geographical and GIS matters.

Finis!