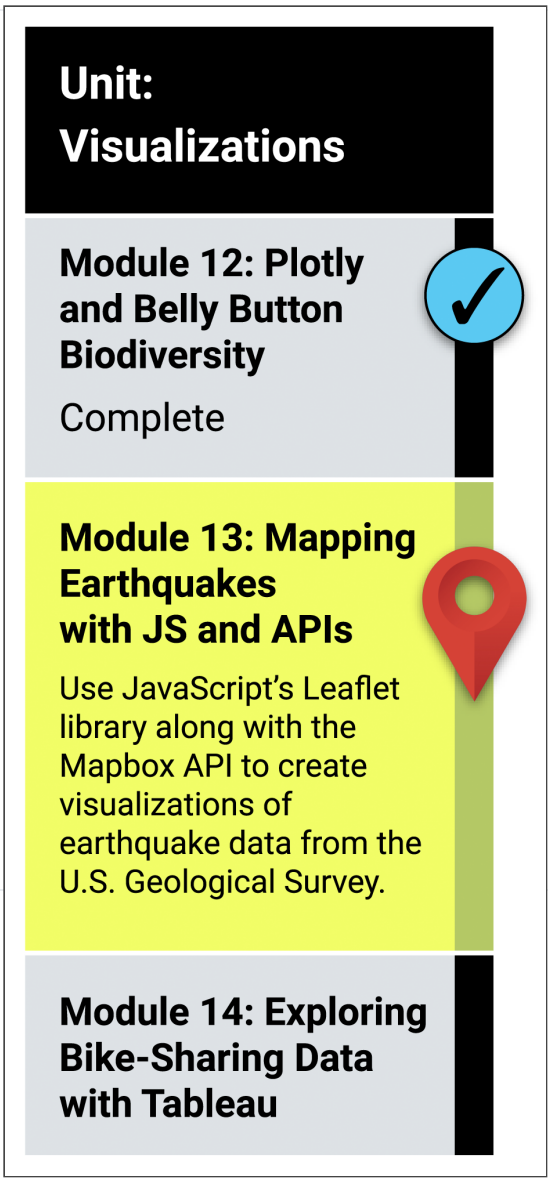
13.0.1: Visualizing Earthquake Data

One way we can tell stories with data is through interactive maps, which is what you'll create in this module. The following video explains the technical tools you'll work with and the roles they play in enhancing data visualizations.

13.0.2: Module 13 Roadmap

Looking Ahead

In this module, you will use the Leaflet.js Application Programming Interface (API) to populate a geographical map with GeoJSON earthquake data from a URL. Each earthquake will be visually represented by a circle and color, where a higher magnitude will have a larger diameter and will be darker in color. In addition, each earthquake will have a popup marker that, when clicked, will show the magnitude of the earthquake and the location of the earthquake.



What You Will Learn

By the end of this module, you will be able to:

- Create a branch from the master branch on GitHub.
- Add, commit, and push data to a GitHub branch.
- Merge a branch with the master branch on GitHub.

- Retrieve data from a GeoJSON file.
- Make API requests to a server to host geographical maps.
- Populate geographical maps with GeoJSON data using JavaScript and the Data-Driven Documents (D3) library.
- Add multiple map layers to geographical maps using Leaflet control plugins to add user interface controls.
- Use JavaScript ES6 functions to add GeoJSON data, features, and interactivity to maps.
- Render maps on a local server.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to Module 13 (15 minutes)
- The Earthquake Mapping Project (30 minutes)
- Create Your First Map (1 hour)
- Don't Mess with the Master Branch (1 hours)
- Map Geographical Features (2 hours)
- Map GeoJSON Data (2 hours)
- Map Earthquakes (2 hours)
- Application (5 hours)

13.0.3: Welcome to Mapping Earthquakes

You know you'll be creating interactive maps using GeoJSON data—but how and for what purpose? The following video covers the details of the specific project you'll be working on as you explore earthquakes around the world.

13.1.1: Create and Clone a New GitHub Repository

Before you get started, Basil wants to make sure you have access to GitHub. Once you have logged in to GitHub, create a new repository and clone the repository to your computer.

Create a GitHub repository for your project and name it “Mapping_Earthquakes.” Make the repository public. Then add a README for the repository and clone it on your computer.

13.1.2: Project Overview

In your first team meeting, Basil assigns Sadhana as your mentor on this project, since she has created a similar map for severe weather. When you get back to your desk, you and Sadhana will go over the basic project plan and what you'll need to do.

You're all set up in your office space and have cloned the GitHub repository onto your computer. Now it’s time to review the plan for your project.

Basic Project Plan

Purpose

The purpose of this project is to visually show the differences between the magnitudes of earthquakes all over the world for the last seven days.

Tasks

To complete this project, use a URL for GeoJSON earthquake data from the USGS website and retrieve geographical coordinates and the magnitudes of earthquakes for the last seven days. Then add the data to a map.

Approach

Your approach will be to use the JavaScript and the D3.js library to retrieve the coordinates and magnitudes of the earthquakes from the GeoJSON data. You'll use the Leaflet library to plot the data on a Mapbox map through an API request and create interactivity for the earthquake data.

Now that you have an overview of the project plan, let's set up a Mapbox account and get the API token you'll need to create geographical maps.

13.2.1: The Mapbox API

Before you create any maps, you'll need to create a Mapbox account and get your access token. Sadhana is going to give you access to the company Mapbox account so you can create your own API key.

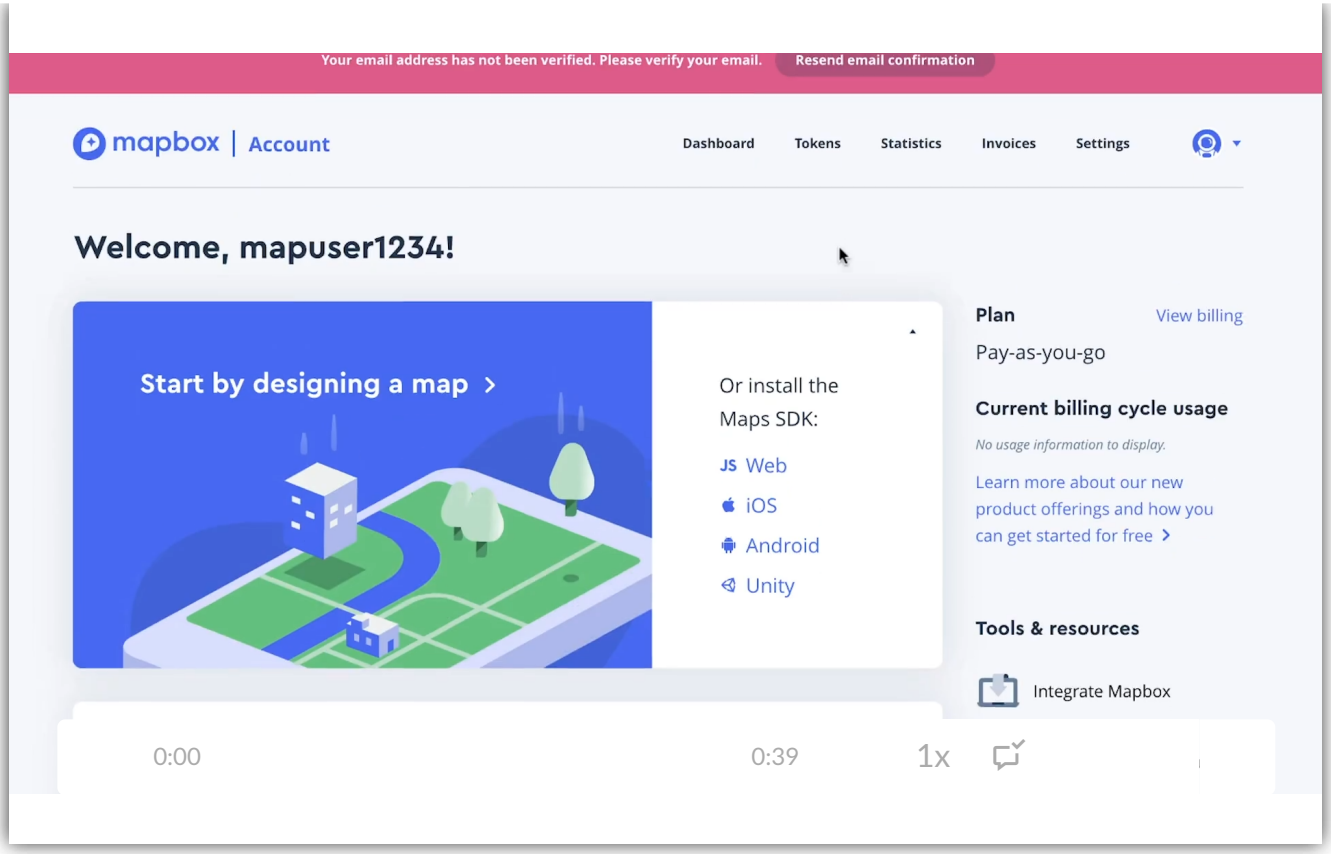
Mapbox provides custom maps for websites and applications such as Strava, Facebook, the *Financial Times*, The Weather Channel, Snapchat, and Instacart. Since October 2019, Mapbox has been generating up to 14 billion individual sensor readings daily across 100,000 map updates on connected devices.

To get started, register for a Mapbox account and get an API key.

Create a Mapbox Account

Follow the steps below to register for a Mapbox account and generate your API key so that you can render maps on the browser. Once you create an account, copy the access token and keep it somewhere safe.

The following video walks you through the process of creating your Mapbox account and generating your API key.



> **Note**
>
> Refer to **Mapbox's pricing (https://www.mapbox.com/pricing/?utm_medium=blog&utm_source=mapbox-blog&utm_campaign=blog%7Cmapbox-blog%7Cpricing%7Cnew-pricing-46b7c26166e7-19-05&utm_term=pricing&utm_content=new-pricing-46b7c26166e7)** for more options.

Now that you have your Mapbox API key, Sadhana will walk you through how to create a branch off the master branch where you'll add the code to create a simple map.

13.2.2: Create a Branch on Your Repository

Before you add your folders and files to the Mapping_Earthquakes folder, Basil would like you to create a GitHub branch for the basic map. Basil informs you that creating a branch off the master branch allows you to create a new feature, or change some part of the original code. Once he has reviewed your code on your branch, Sadhana will show you how to merge it with the master branch.

Branching makes it possible to create an isolated environment when creating a new feature, or to correct an issue in the main code no matter the scope.

Creating branches off the master branch is where Git shines. A branch allows you to make changes to the main code “off to the side,” like a branch on a tree. Once the code on the branch has been reviewed and approved, only then can the branch be merged into the master branch. This ensures that the master branch always contains production-quality code.

Before we create a branch and add files to the new branch, make sure your copy of the master branch on your computer is up to date with the master branch on GitHub.

What command do you use to ensure your version of the master branch on your computer is up to date the GitHub master branch? (Select all that apply.)

- [ ] `git pull`
- [ ] `git pull master`
- [ ] `git checkout master`
- [ ] `git pull origin master`
- [ ] `pull`

Check Answer

Finish ▸

It's a best practice to either “pull” from the master branch to get the latest changes to the master branch, or pull “upstream” of the branch you are creating. This is important because your version of the master branch on your computer needs to be up to date with the master branch on GitHub.

Follow these steps to create a branch:

1. Navigate to the repository folder on your computer.
2. In Terminal on macOS or Git Bash on Windows, type `git pull` or `git pull origin master` and press Enter.
3. Type `git checkout -b Simple_Leaflet_Map` and press Enter.
   - `git checkout` lets us navigate between branches.
   - `-b` indicates we are creating a new branch.
   - The name of the new branch follows `-b`.

After pressing Enter, your terminal or Git Bash should return the following:

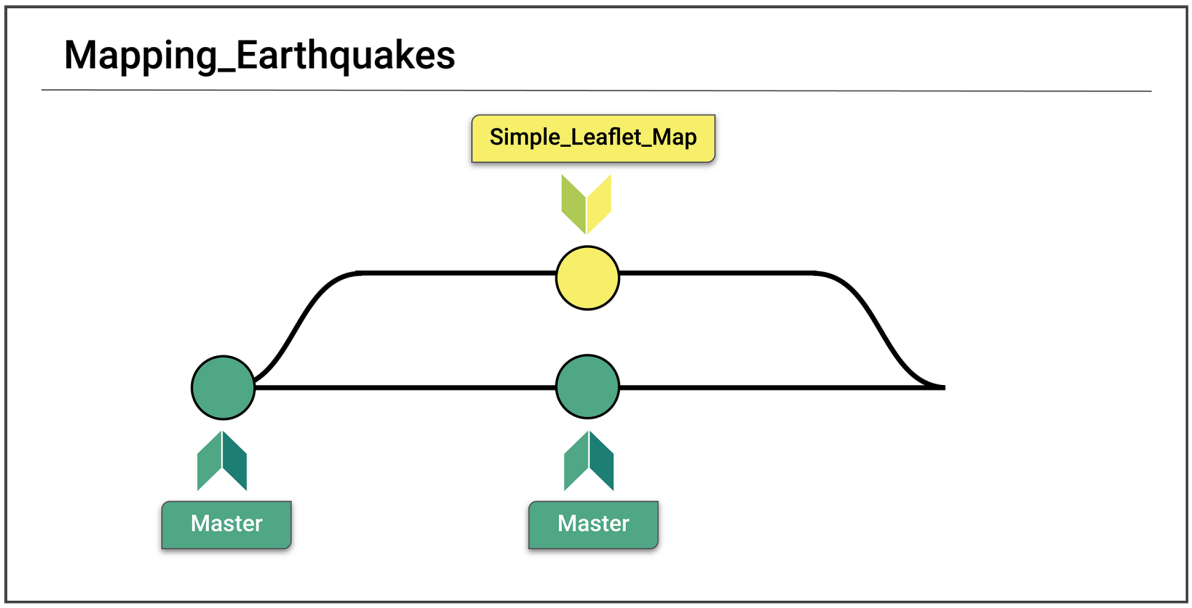
```
Switched to a new branch 'Simple_Leaflet_Map'
```

Now we are in the Simple_Leaflet_Map branch. Confirm this by typing `git branch` and pressing Enter.

The output in Terminal or Git Bash should look as follows, with an asterisk next to the branch name:

```
* Simple_Leaflet_Map
  master
```

At this point, the folder structure on your computer hasn't changed. The files you had in your master branch are now in your Simple_Leaflet_Map branch. Visually, the repository should look like the following:



Next, we'll create a folder structure and write the code to create a simple map. When we are done, Sadhana will show us how to add folders and files to the Simple_Leaflet_Map branch.

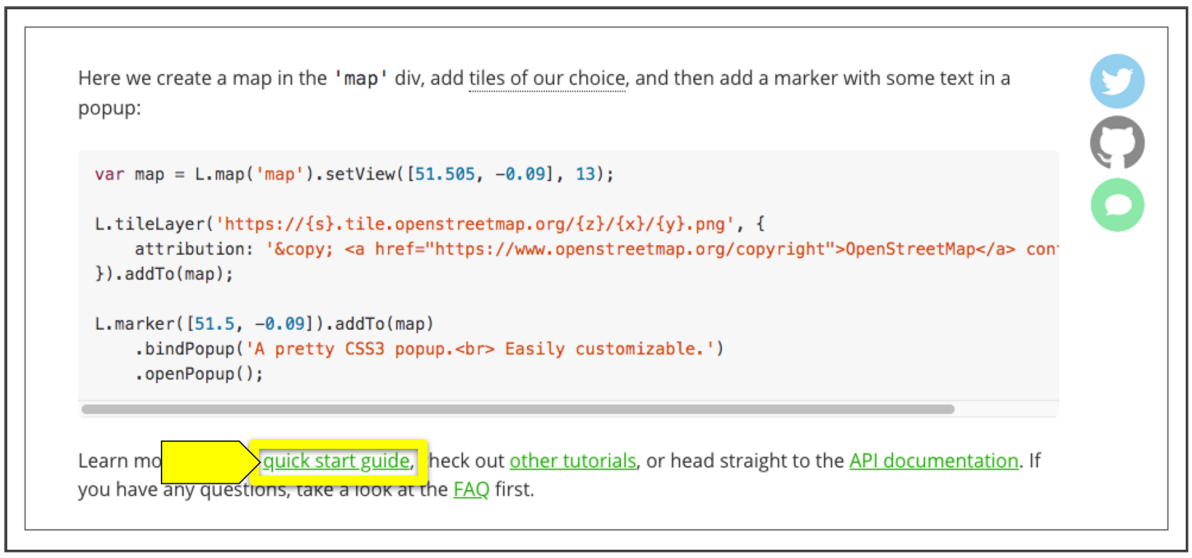
13.2.3: Create an HTML Page and CSS File

Now that you have your API token and your branch is set up, Sadhana will let you get familiar with the documentation for using Leaflet. Then, she's suggested you jump in and create an HTML and CSS file and add links to some stylesheets in the HTML file.

Let's get familiar with Leaflet. Leaflet has a few links and simple HTML code that we will add to an `index.html` page.

The Leaflet JavaScript Library

On the **Leaflet (https://leafletjs.com/index.html)** homepage, scroll midpage and click the **Leaflet Quick Start Guide (https://leafletjs.com/examples/quick-start/)** link:

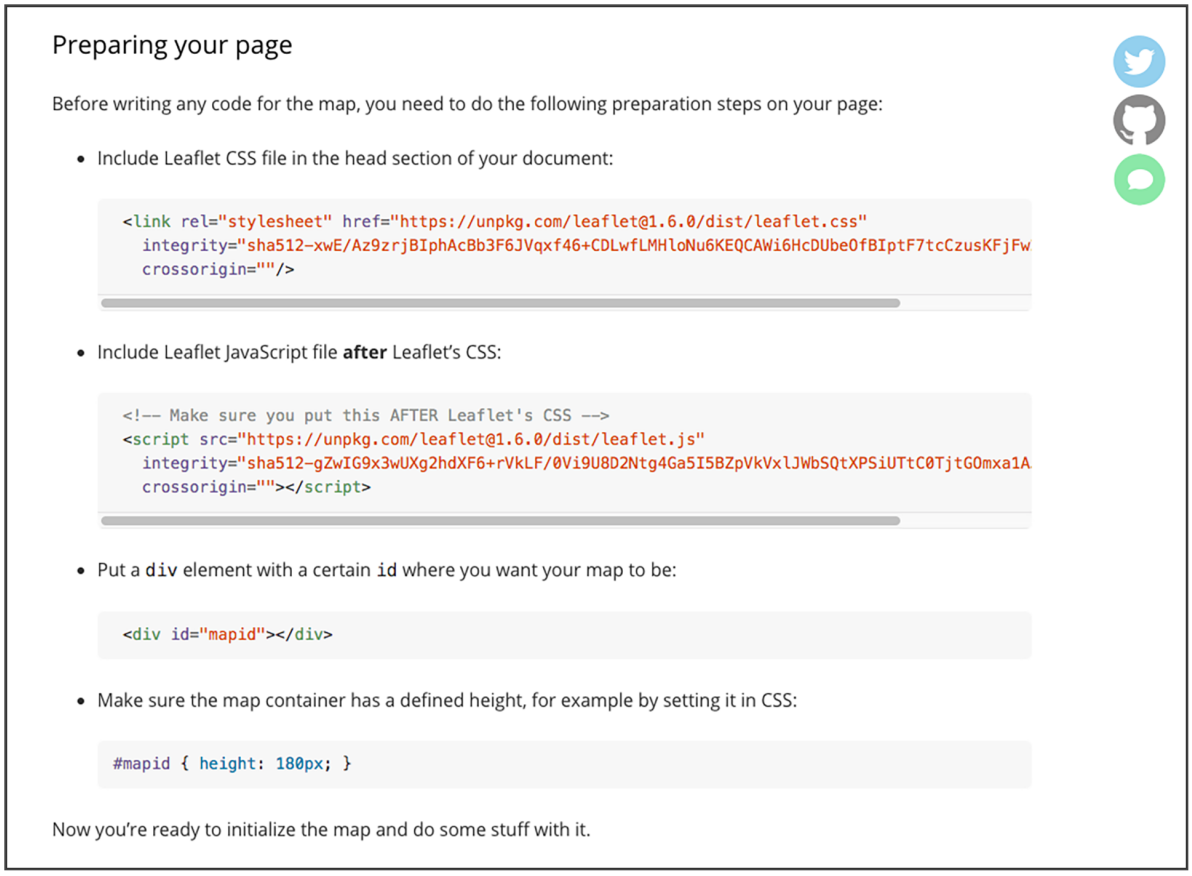


Here we create a map in the `'map'` div, add tiles of our choice, and then add a marker with some text in a popup:

```
var map = L.map('map').setView([51.505, -0.09], 13);

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

L.marker([51.5, -0.09]).addTo(map)
    .bindPopup('A pretty CSS3 popup.<br> Easily customizable.')
    .openPopup();
```

Learn mo quick start guide, heck out other tutorials, or head straight to the API documentation. If you have any questions, take a look at the FAQ first.

The Leaflet Quick Start Guide provides steps for setting up a Leaflet map. To begin, scroll midpage to the “Preparing your page” section:



Preparing your page

Before writing any code for the map, you need to do the following preparation steps on your page:

- Include Leaflet CSS file in the head section of your document:

```
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.6.0/dist/leaflet.css"
  integrity="sha512-xwE/Az9zrjBIphAcBb3F6JVqxf46+CDLwfLMHloNu6KEQCAWi6HcDUbeOfBIptF7tcCzusKFjFw
  crossorigin=""/>
```

- Include Leaflet JavaScript file **after** Leaflet’s CSS:

```
<!-- Make sure you put this AFTER Leaflet's CSS -->
<script src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
  integrity="sha512-gZwIG9x3wUXg2hdXF6+rVkLF/0Vi9U8D2Ntg4Ga5I5BZpVkVxlJWbSQtXPSiUTtC0TjtGOmxa1A
  crossorigin=""></script>
```

- Put a `div` element with a certain `id` where you want your map to be:

```
<div id="mapid"></div>
```

- Make sure the map container has a defined height, for example by setting it in CSS:

```
#mapid { height: 180px; }
```

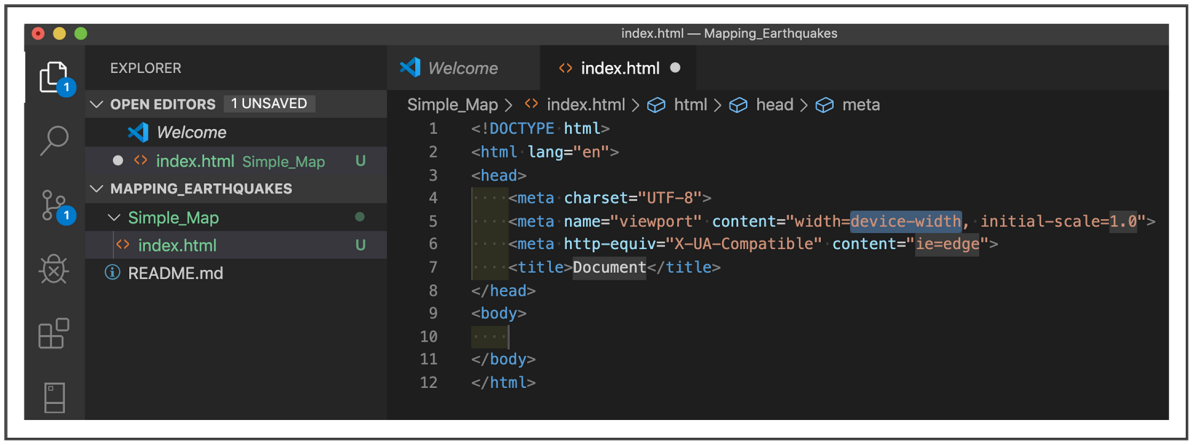
Now you’re ready to initialize the map and do some stuff with it.

The “Preparing your page” section includes links and HTML code that we’ll add to our `index.html` page. First, we’ll create a folder structure for our earthquake map webpage.

Create an HTML Page

Open your Mapping_Earthquakes repository folder in Visual Studio Code (VS Code). Create a new folder called “Simple_Map.” Inside the folder, create a new `index.html` file. Next, create a template for your `index.html` file.

> **Rewind**
>
> To create a template for an `html` file, type an ! (exclamation point) and press Tab or Enter, and the empty file will be filled with the complete HTML5 boilerplate code.



In our `index.html` file, we’ll change the document title and add the code from the “Preparing your page” section.

1. Change the document title in the `<title>` element in the `<head>` section (`<title>Document</title>`) to “Leaflet-Basic-Map.”
2. In the `<head>` section, just below the `<title>` element, add the following Leaflet CSS script from the "Preparing your page" section:

```
<!-- Leaflet CSS -->
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.6.0/dist/lea
   crossorigin=""/>
```

3. In the body of our `index.html` file, add the Leaflet JavaScript script and id tag for the map inside a `<div>` element, as shown in the “Preparing your page” section:

```
<!-- Leaflet JavaScript -->
<script src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"integrit
   crossorigin=""></script>
```

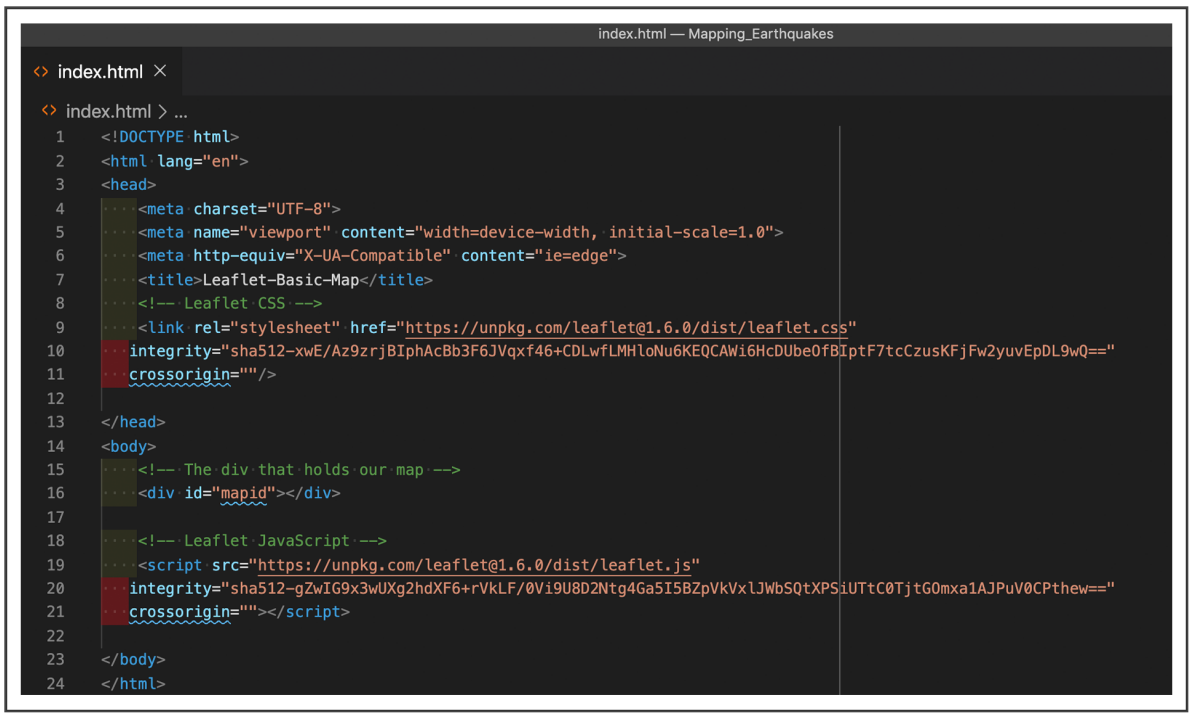
   The Leaflet CSS and JavaScript files we added to the `index.html` file are referred to as content delivery networks (CDNs). Using CDNs has a security risk. To avoid the security risk, it's a best practice to include an integrity value with the CDN. Each file we added has its own integrity value, which is a Base64-encoded cryptographic hash of a resource that prevents the CDN from being hacked.

   > **Note**
   > For more about the Subresource Integrity value, please see the documentation on the **Download Leaflet webpage (https://leafletjs.com/download.html)** and Mozilla Developers' **Subresource Integrity webpage (https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity)**.

4. Above the Leaflet JavaScript link script, add the following `<div>` element with the id tag for the map:

```
<!-- The div that holds our map -->
<div id="mapid"></div>
```

After adding the Leaflet CSS file, JavaScript file, and the `<div>` element with the `id` tag for the map, our `index.html` file should look like the following:

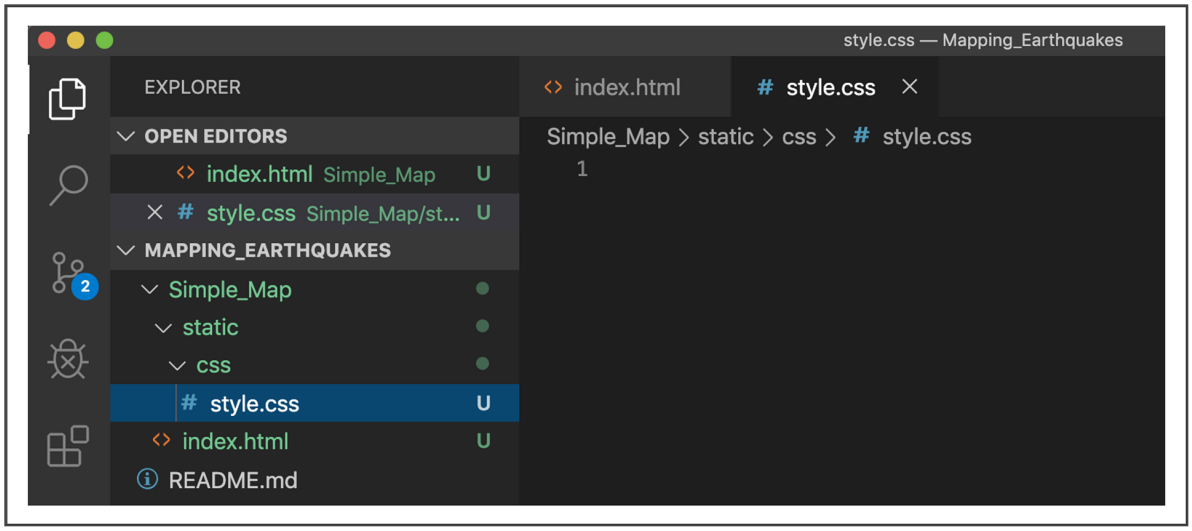


> **Important!**
>
> Make sure you copy and paste the Leaflet CSS file and JavaScript script from the website as they appear. Do not edit the script by deleting empty spaces. This will prevent that script from working on the `index.html` file, resulting in the map not being shown on the webpage.

Next, we will modify our `#mapid` to be set at a specific height using CSS code. To do this, we'll need to create a `style.css` file.

Create a CSS File

Before we create a `style.css` file, we'll need to create a folder for the file. In VS Code, create a new subfolder called "static" in our Simple_Map folder. In this folder, create another subfolder named "css." In the css folder, create a new file and name it `style.css`:

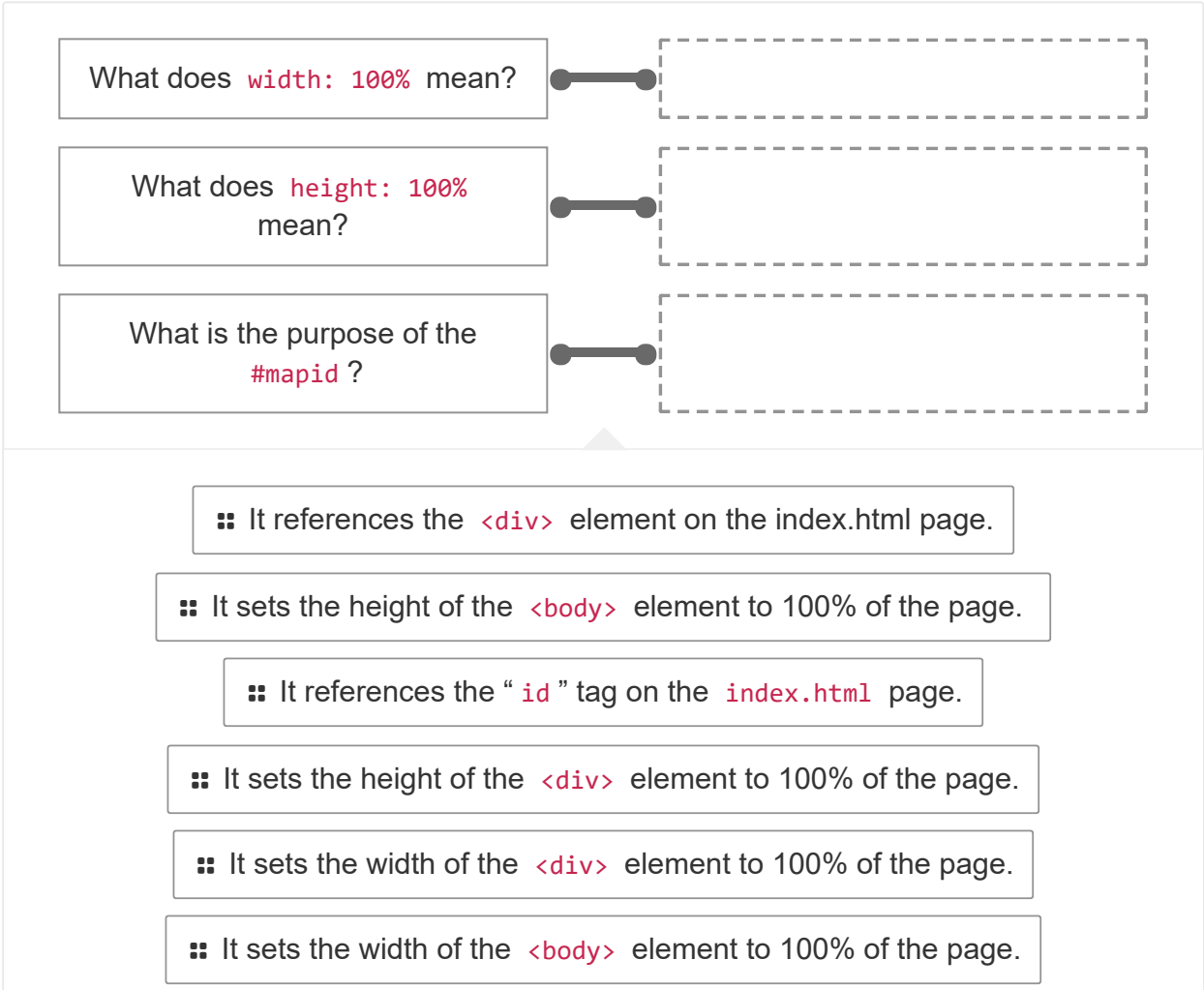


Next, add the following CSS code to our `style.css` file to set the style for our map on our `index.html` page and save the file:

```
html,
body,
#mapid {
  width: 100%;
  height: 100%;
  padding: 0;
  margin: 0;
}
```

Match the best answer for each question below.

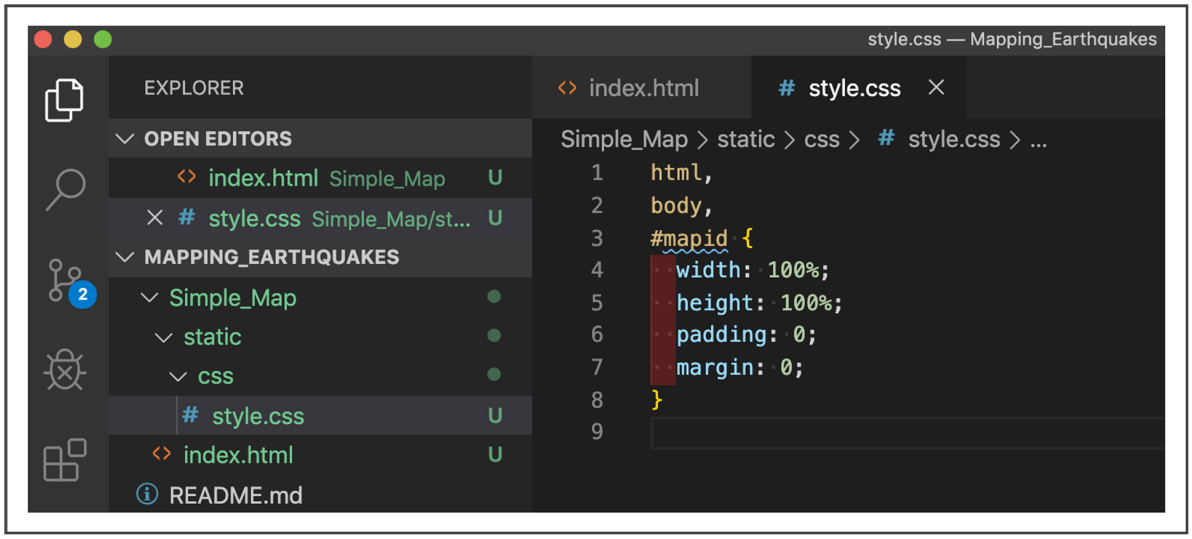
For the `style.css` file:



Check Answer

Finish ▸

At this point, your folder should look like the following:



Finally, we need to tell our `index.html` page to use the `style.css` file we created.

In the `<head>` section of our `index.html` page, add the following CSS link script below the Leaflet CSS and before the closing `</head>` element:

```
<!-- Our CSS -->
  <link rel="stylesheet" type="text/css" href="static/css/style.css">
```

Next, we'll create the code for a simple map.

13.2.4: Create a Simple Map

Now that you have added stylesheet links to your HTML page and CSS file, Sadhana will help you add the last two files you'll need to create a simple map—the `config.js` file for your API token and the `logic.js` file for your JavaScript and Leaflet code.

To create a map, we'll need to add two final pieces to our Simple_Map folder:

- The `config.js` file, which will hold our Mapbox API key.
- The `logic.js` file, which will contain all the JavaScript and Leaflet code to create the map and add data to the map.

Add the config.js File

In our Simple_Map folder, add a new folder in the static folder called "js." In the js folder, create a new file and name it `config.js`. This file will hold our Mapbox API key.

> **Rewind**
> The `config.js` file is like the `config.py` file we used to hold the OpenWeatherMap API and Google API keys.

On the first line of the `config.js` file, add the following code:

```
// API key
const API_KEY = "";
```

Add your Mapbox API key between the quotations and save the file.

Add the logic.js File

Next, in the js folder, create a new file and name it, `logic.js`. Now we'll add some boilerplate code to the `logic.js` file. Most of this code can be reused for many of the maps we'll create later on in this module.

On the first line, add the following code:

```
// Add console.log to check to see if our code is working.
console.log("working");
```

The `console.log()` function with the phrase "working" inside the parentheses will help us confirm that our `logic.js` file is being accessed in the console on Chrome.

Add a Map Object

Next, we'll add the map object, as shown on the Leaflet Quick Start Guide page with some slight modifications. We'll change the geographical center of the map to the approximate geographical center of the United States.

```
// Create the map object with a center and zoom level.
let map = L.map('mapid').setView([40.7, -94.5], 4);
```

In the code block above:

1. We're assigning the variable `map` to the JavaScript class "L," an acronym for Leaflet, and we'll instantiate the map object with the given string `'mapid'`.
2. The `mapid` will reference the `id` tag in our `<div>` element on the `index.html` file.
3. The `setView()` method sets the view of the map with a geographical center, where the first coordinate is latitude (40.7) and the second is longitude (-94.5). We set the zoom level of "4" on a scale 0–18.

An alternative to using the `setView()` method is to modify each attribute in the map object using the curly braces notation as follows:

```
// Create the map object with a center and zoom level.
let map = L.map("mapid", {
  center: [
    40.7, -94.5
  ],
  zoom: 4
});
```

This method is useful when we need to add multiple tile layers, or a background image of our map(s), which we will do later in this module.

Add a Tile Layer for Our Map

Now, we will add the map tile layer method to the `logic.js` file. The tile layer is used to load and display a tile layer on the map. We have two options to create a tile layer.

Use the Leaflet Documentation

The **Leaflet Quick Start Guide** **(https://leafletjs.com/examples/quick-start/)** provides the `tileLayer()` code:

```
L.tileLayer('https://api.mapbox.com/styles/v1/{id}/tiles/{z}/{x}/{y}?acce
ss_token={accessToken}', {
    attribution: 'Map data &copy; <a
href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a
href="https://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>,
Imagery © <a href="https://www.mapbox.com/">Mapbox</a>',
    maxZoom: 18,
    id: 'mapbox/streets-v11',
    accessToken: 'your.mapbox.access.token'
}).addTo(mymap);
```

We can copy this tile layer code and assign it to the `streets` variable, since the tile layer will create a street-level map. Add the following code block to your `logic.js` file:

```
// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{
attribution: 'Map data © <a href="https://www.openstreetmap.org/">OpenStr
    maxZoom: 18,
    id: 'mapbox.streets',
    accessToken: API_KEY
});
// Then we add our 'graymap' tile layer to the map.
streets.addTo(map);
```

Let’s break down what's happening in this code block:

1. We assign the `streets` variable to the `tileLayer()` method, as shown in the Quick Start Guide’s “Setting up the map” section. Leaflet doesn't provide a tile layer. Instead, it offers various tile layer APIs.
2. The following URLs appear in the parentheses of our `tileLayer()` method:
    - The API URL with a reference to the `accessToken`
    - The `OpenStreetMap` URL inside the curly braces of our `tileLayer()` method
3. We add the `maxZoom` attribute and assign it a value of 18.

4. We add the `id` attribute and assign it `mapbox.streets`, which will show the streets on the map.
5. We add the `accessToken` attribute and assign it the value of our `API_KEY`.
6. Finally, we call the `addTo()` function with our map object, `map` on our graymap object tile layer. The `addTo()` function will add the graymap object tile layer to our `let map`.

To change the map's style, change the map `id` using the list of Mapbox ids below:

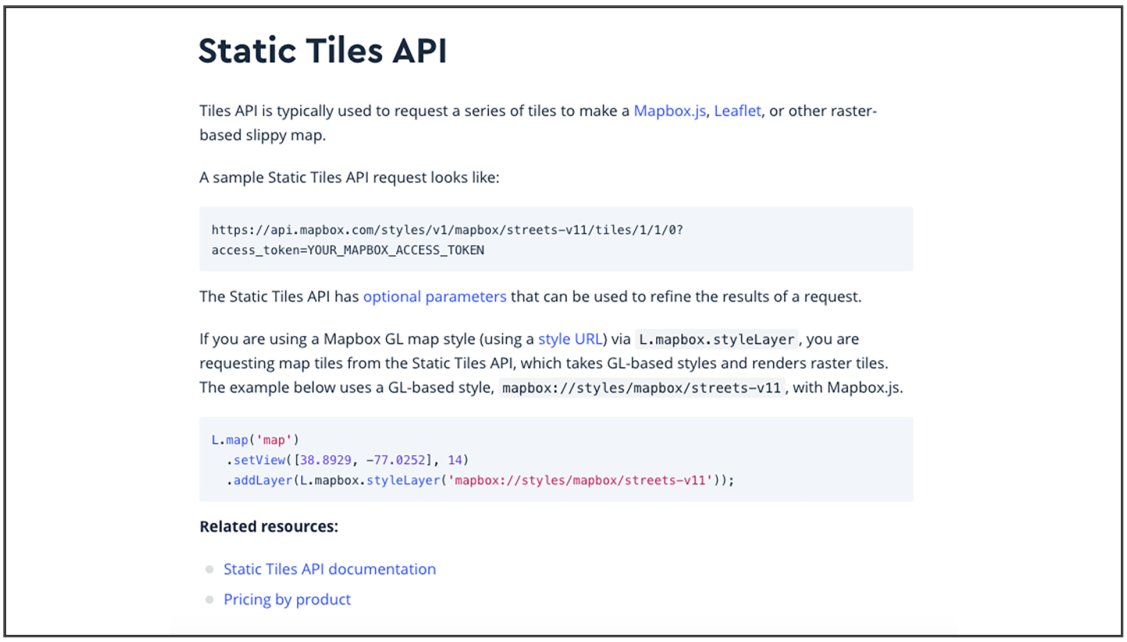
- mapbox.streets
- mapbox.light
- mapbox.dark
- mapbox.satellite
- mapbox.streets-satellite
- mapbox.wheatpaste
- mapbox.streets-basic
- mapbox.comic
- mapbox.outdoors
- mapbox.run-bike-hike
- mapbox.pirates
- mapbox.emerald
- mapbox.high-contrast

> **Note**
> Although this API URL still works with the different map styles above, the user documentation is no longer on the Mapbox website. Instead, edit the URL and use the Styles API for the type of map you want. Refer to the instructions below and on the Mapbox website.

Use the Mapbox Styles API

To use the Mapbox Styles API, edit the URL in the Leaflet `tilelayer()` method, as detailed on the Leaflet website:

1. First, navigate to the **Mapbox Glossary (https://docs.mapbox.com/help/glossary/)** .
2. Search the **Static Tiles API (https://docs.mapbox.com/help/glossary/static-tiles-api/)** .



Static Tiles API

Tiles API is typically used to request a series of tiles to make a Mapbox.js, Leaflet, or other raster-based slippy map.

A sample Static Tiles API request looks like:

```
https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/1/1/0?
access_token=YOUR_MAPBOX_ACCESS_TOKEN
```

The Static Tiles API has optional parameters that can be used to refine the results of a request.

If you are using a Mapbox GL map style (using a style URL) via `L.mapbox.styleLayer`, you are requesting map tiles from the Static Tiles API, which takes GL-based styles and renders raster tiles. The example below uses a GL-based style, `mapbox://styles/mapbox/streets-v11`, with Mapbox.js.

```
L.map('map')
  .setView([38.8929, -77.0252], 14)
  .addLayer(L.mapbox.styleLayer('mapbox://styles/mapbox/streets-v11'));
```

Related resources:

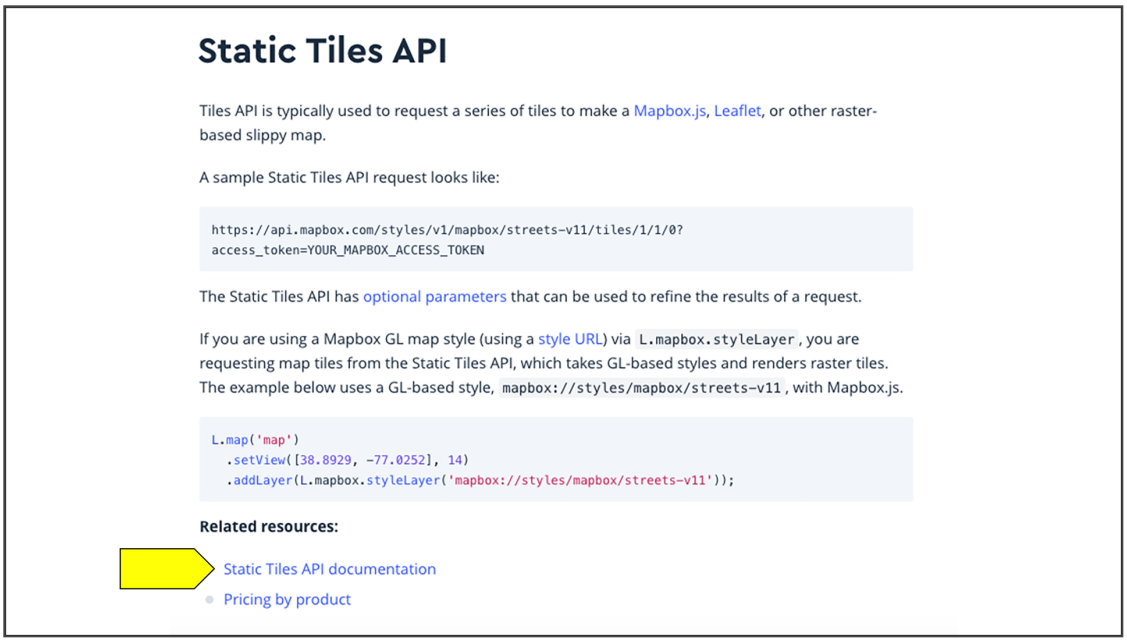
- Static Tiles API documentation
- Pricing by product

3. Copy this part of the URL: `https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/`.
4. In your `tileLayer()` code, replace this part of the URL, `https://api.tiles.mapbox.com/v4/{id}/`, with the Mapbox Styles API URL you copied.
5. Remove the `id` attribute and the map style reference.
6. The code for the `tileLayer()` should look like the following:

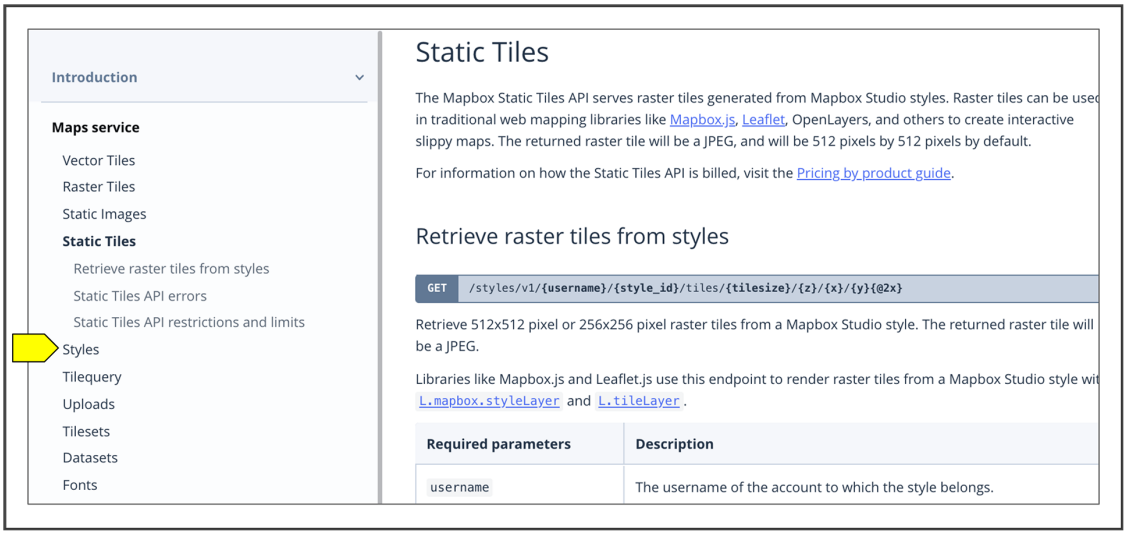
```
// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/st
attribution: 'Map data © <a href="https://www.openstreetmap.org/">Ope
    maxZoom: 18,
    accessToken: API_KEY
});
```

```
// Then we add our 'graymap' tile layer to the map.
streets.addTo(map);
```

7. To change your map style, click on the **Static Tiles API documentation (https://docs.mapbox.com/api/maps/#static-tiles)** link on the Static Tiles API page.



8. On the left sidebar, click on the **Styles (https://docs.mapbox.com/api/maps/#styles)** link.



9. Below the Styles subheading, find a list of different Mapbox styles.

Mapbox styles

The following Mapbox styles are available to all accounts using a valid access token:

- mapbox://styles/mapbox/streets-v11
- mapbox://styles/mapbox/outdoors-v11
- mapbox://styles/mapbox/light-v10
- mapbox://styles/mapbox/dark-v10
- mapbox://styles/mapbox/satellite-v9
- mapbox://styles/mapbox/satellite-streets-v11
- mapbox://styles/mapbox/navigation-preview-day-v4
- mapbox://styles/mapbox/navigation-preview-night-v4
- mapbox://styles/mapbox/navigation-guidance-day-v4
- mapbox://styles/mapbox/navigation-guidance-night-v4

10. To change the map style, use the style given in the URLs (e.g., “streets-v11,” “dark-v10,” etc.).

> **Note**
>
> For the rest of this module, we’ll use the Static Tiles API format in the Leaflet `tileLayer()` method.

After adding all of the code, your `logic.js` file should look like the following:

```
// Add console.log to check to see if our code is working.
console.log("working");

// We create the map object with options.
let map = L.map('mapid').setView([40.7, -94.5], 4);

// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecom
    maxZoom: 18,
    accessToken: API_KEY
});
// Then we add our 'streets' tile layer to the map.
streets.addTo(map);

```

Next, we’ll add the `logic.js` and `config.js` scripts to the HTML page.

Add JavaScript Tags to HTML Page

The last step is to allow our `index.html` file to use the `logic.js` and `config.js` scripts. To do this, open up the `index.html` file and add the following `<script>` elements below the Leaflet JavaScript link script and above the closing `</body>` element, and save the file.

```
<!-- API key -->
<script type="text/javascript" src="static/js/config.js"></script>
<!-- Our JavaScript -->
<script type="text/javascript" src="static/js/logic.js"></script>
```

Your `index.html` file should look like the following after adding all the elements to the head and body sections:

Open the Map in the Browser

Now, open the `index.html` file.

> **Rewind**
>
> To open the `index.html` file in the browser:

1. Open the command line and navigate to your Mapping_Earthquakes folder. The `index.html` file should be in the top-level of that folder.
2. On the command line, type `python -m http.server` and press Enter.

The basic map should look like the following in your web browser:

Congratulations on creating your first map using Leaflet and the Mapbox API!

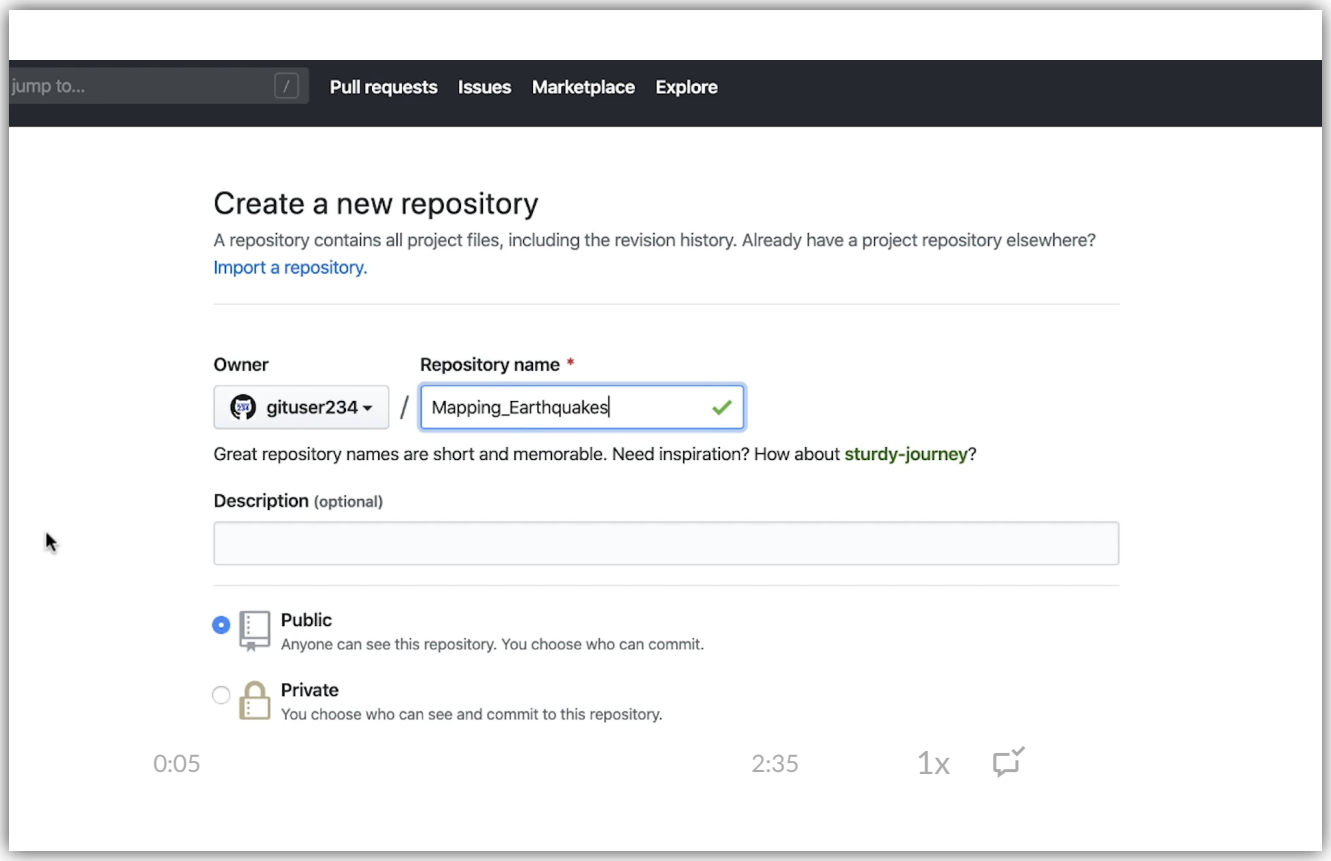
Note

For more information, see the **Leaflet documentation on methods using the map object and tile layer (https://leafletjs.com/reference-1.5.0.html#map-example)** .

13.3.1: Add, Commit, and Push to a Branch

Basil wants you to commit your changes to the Simple_Leaflet_Map branch so the team can review. If he says there are no changes to be made to your branch, then you can merge it with the master branch.

Watch the following video to learn more about Git Branching.



What command do you use to see what branch you are on? (Select all that apply.)

- [ ] `git check branch`
- [ ] `git check branch`
- [ ] `git checkout branch`
- [ ] `git branch`
- [ ] `git status`

Check Answer

Finish ▸

First, we'll check to see if you're in the Simple_Leaflet_Map branch. In the command line, navigate to your Mapping_Earthquakes folder and type `git branch`. If the following output is returned with an asterisk next to the branch name, you're in the Simple_Leaflet_Map branch:

```
* Simple_Leaflet_Map
  master
```

Just as we have checked the master branch status of other repositories, we'll do the same with the Simple_Leaflet_Map branch.

What command do you use to check the status of a branch on the terminal or Git Bash?

- `git branch`
- `git show`
- `git checkout branch`
- `git show status`
- `git status`

Check Answer

Finish ▸

After typing `git status`, the command line should show the following:

```
On branch Simple_Leaflet_Map
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Simple_Map/

nothing added to commit but untracked files present (use "git add" to track)
```

To see all the folders and files in the Simple_Map folder, type `git status -u` and press Enter. The output in the command line should look like the following:

```
On branch Simple_Leaflet_Map
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Simple_Map/index.html
        Simple_Map/static/css/style.css
        Simple_Map/static/js/config.js
        Simple_Map/static/js/logic.js

nothing added to commit but untracked files present (use "git add" to track)
```

Rewind

> We don't want to add `config.js` file to the repo because our repository is public. This means the Mapbox API key would be available for anyone to copy and use, possibly causing us to incur charges.

What is the best practice to prevent the `config.js` file from being added (tracked) on GitHub?

- ○ Don't add it with `git add.`
- ○ Remove it from the folder before you use `git add.`
- ○ Add `config.js` to the `.gitignore` file.

Check Answer

Finish ▸

If you didn't initialize the repository with a `.gitignore` file, we'll create one. Then we'll add `config.js` to the `.gitignore` file so that GitHub doesn't track that file.

In VS Code, create a new file in the Simple_Map folder and name it `.gitignore`. Your repository folder should look like the following:

Next, add the following lines to the `.gitignore` file and save the file:

```
# Ignore the config.js file.
config.js
```

In the command line, type `git status - u` and press Enter. The output should have the `.gitignore` file and all the files as before, except the `config.js` file. All the files in red will be tracked when we use `git add`, with the exception of the `config.js` file.

```
On branch Simple_Leaflet_Map
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        Simple_Map/index.html
        Simple_Map/static/css/style.css
        Simple_Map/static/js/logic.js

nothing added to commit but untracked files present (use "git add" to track)
```

Next, we'll add these folders and files to the Simple_Leaflet_Map branch as we did for the master branch.

What is the order you would use to add files or folders to a GitHub branch?

Source	Target
≡ git checkout <branch_name>	
≡ git status	
≡ git status	
≡ git push	
≡ git add .	
≡ git commit -m	

◀ ▶ ▲ ▼

Check Answer

Finish ▸

After typing `git push` and pressing Enter, the output message will look like the following:

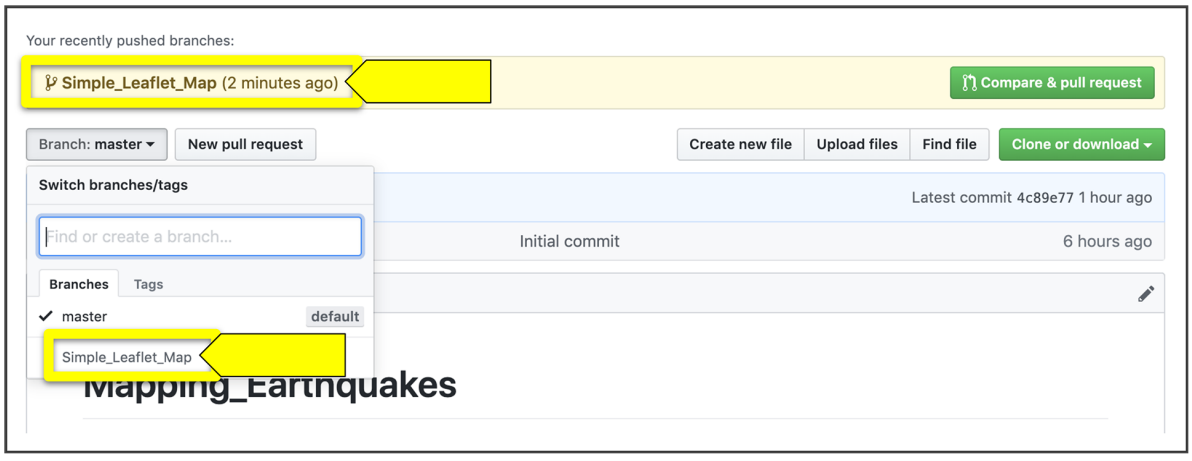
```
fatal: The current branch Simple_Leaflet_Map has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin Simple_Leaflet_Map
```

To push the folders and files to the Simple_Leaflet_Map branch for the first time, type `git push --set-upstream origin Simple_Leaflet_Map` in the command line and press Enter.

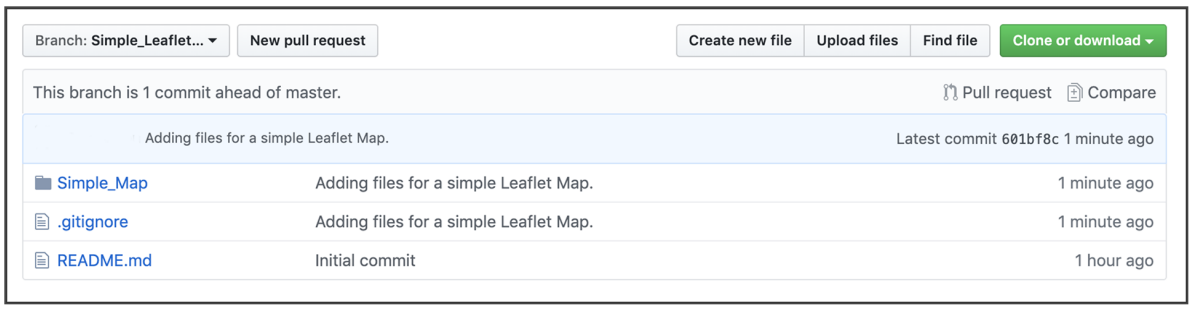
> **Note**
>
> To add any files after this initial push, type `git push`.

Now all the folders and files are in the Mapping_Earthquakes repository in the Simple_Leaflet_Map branch. Confirm this by navigating to the repository and refreshing the page.

We can see that we have created a new branch, indicated at the top. If we click the dropdown menu "Branch: **master**" in our repository, the Simple_Leaflet_Map branch appears.



To check if our folders and files are in the Simple_Leaflet_Map branch, select the branch from the dropdown menu. You should see all the files we added.



Congratulations on adding your files to a new branch!

13.3.2: Compare Branch to the Master Branch

Basil has reviewed your code and is impressed with the comments you made; comments to your code allow others to understand what the code does. Now that your code has Basil's approval, Sadhana will walk you through the steps to merge your branch with the master branch. The first step when merging a branch is to compare the changes between the Simple_Leaflet_Map branch and the master branch.

To merge your Simple_Leaflet_Map branch with the master branch, we will:

1. Compare the changes between the Simple_Leaflet_Map branch and the master branch so that we can merge them.
2. If we are able to merge the Simple_Leaflet_Map branch into the master branch, we need to create a pull request on GitHub.
3. After we review the pull request, we can merge the branch into the master branch.

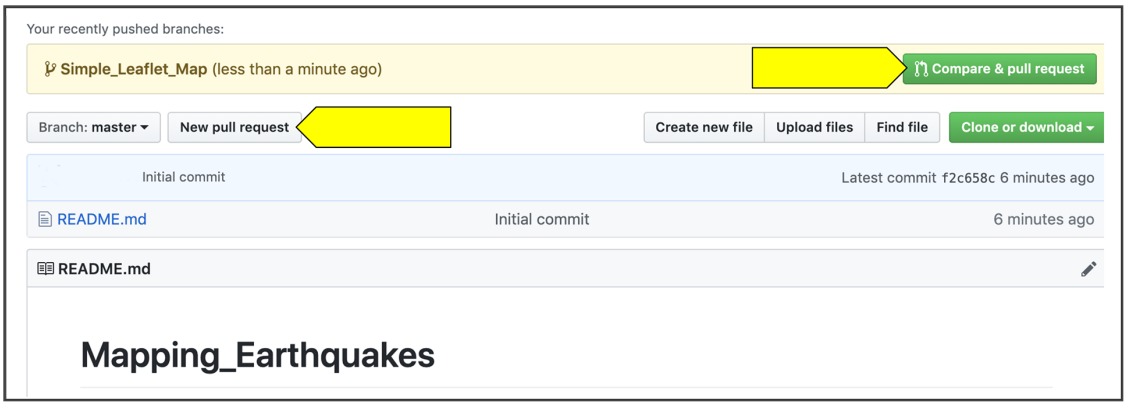
Let's compare the changes between our Simple_Leaflet_Map branch and the master branch:

1. Navigate to your Mapping_Earthquake GitHub repository.
2. To start merging your branch into the master branch, compare the branches to confirm they can be merged.

> **Note**
>
> If changes have been made in the master branch after you added your files to a branch, you might not be able to merge.

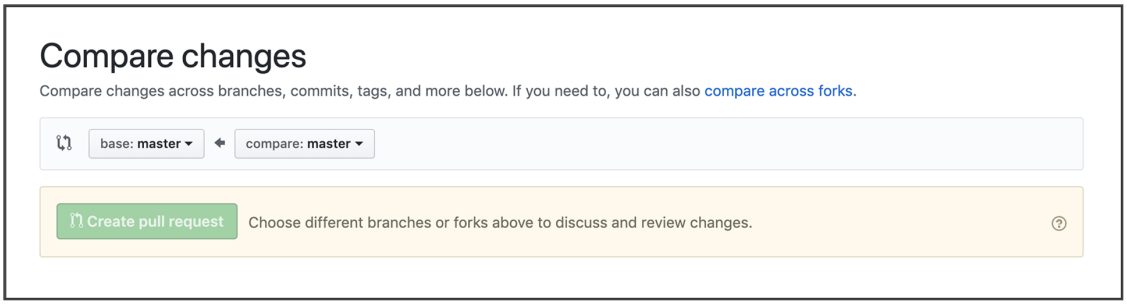
3. You have two ways to compare your branch with the master branch:
   - Click the green "Compare & pull request" button, or
   - Click the gray "New pull request" button.

   (If you click the “Compare & pull request” button, skip to Step 5.)

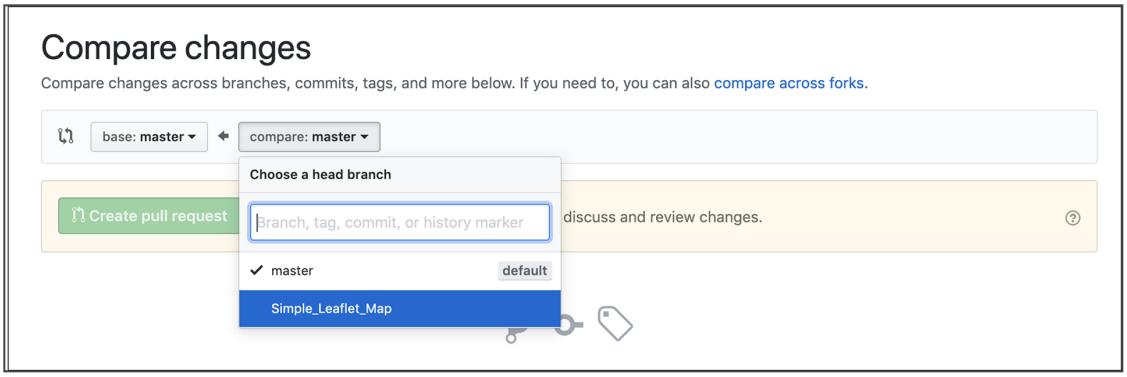


4. If you click the “New pull request” button, a new page will open with two buttons: "base: **master**" and "compare: **master**." Compare the Simple_Leaflet_Map branch with the base or master branch.

> **Note**
>
> Developers and team members often refer to pull requests as “PRs.”



5. Click on the "compare: **master**" button and select the branch you want to compare (the Simple_Leaflet_Map branch).



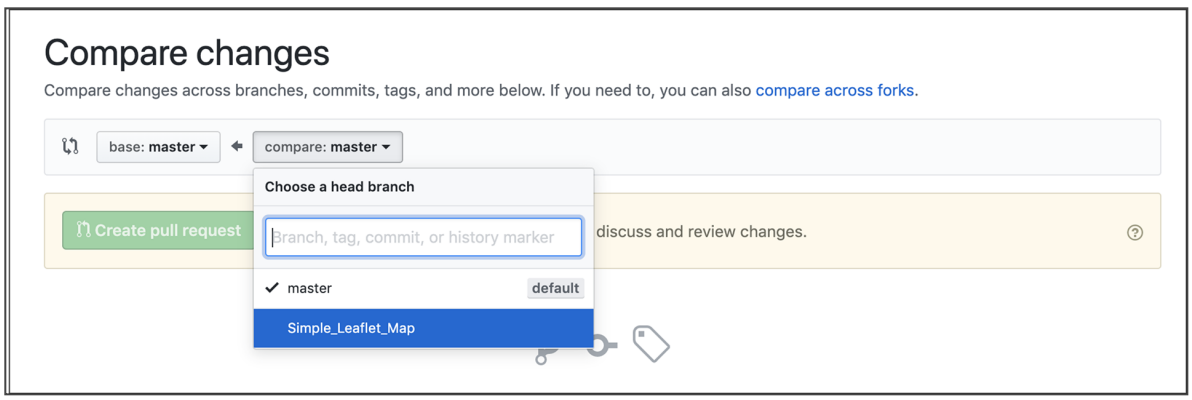
Next, Sadhana will show you how to create a pull request required to merge a branch with the master branch.

13.3.3: Create a Pull Request

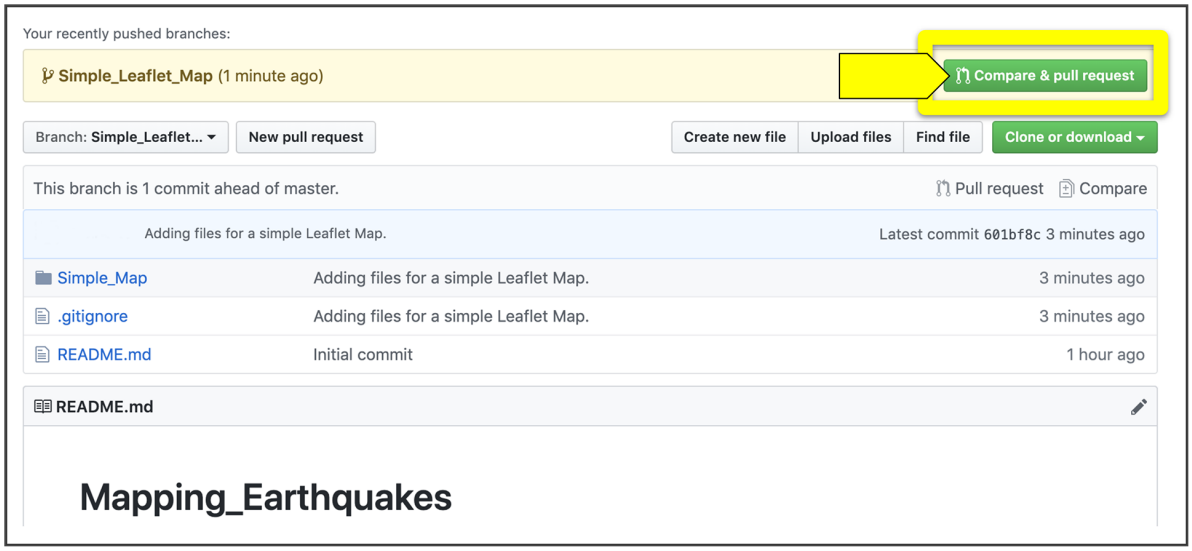
You made it through the first step. Now, Sadhana will show you how to create a pull request. This step is important when working on a team because it allows other team members to check your code on your branch, and then either ask you to make changes or approve it so it can be merged with the master branch.

The next step is to create a pull request on GitHub. This step is essential when working with your Disaster Reporting Network team. It will allow Basil and Sadhana to review and run your code to verify there are no errors. If there are errors in your code, or if the code can be modified to be easier to read, then they will ask you to make changes. If your code looks good and there are no changes to be made, then you can merge your branch with the master branch.

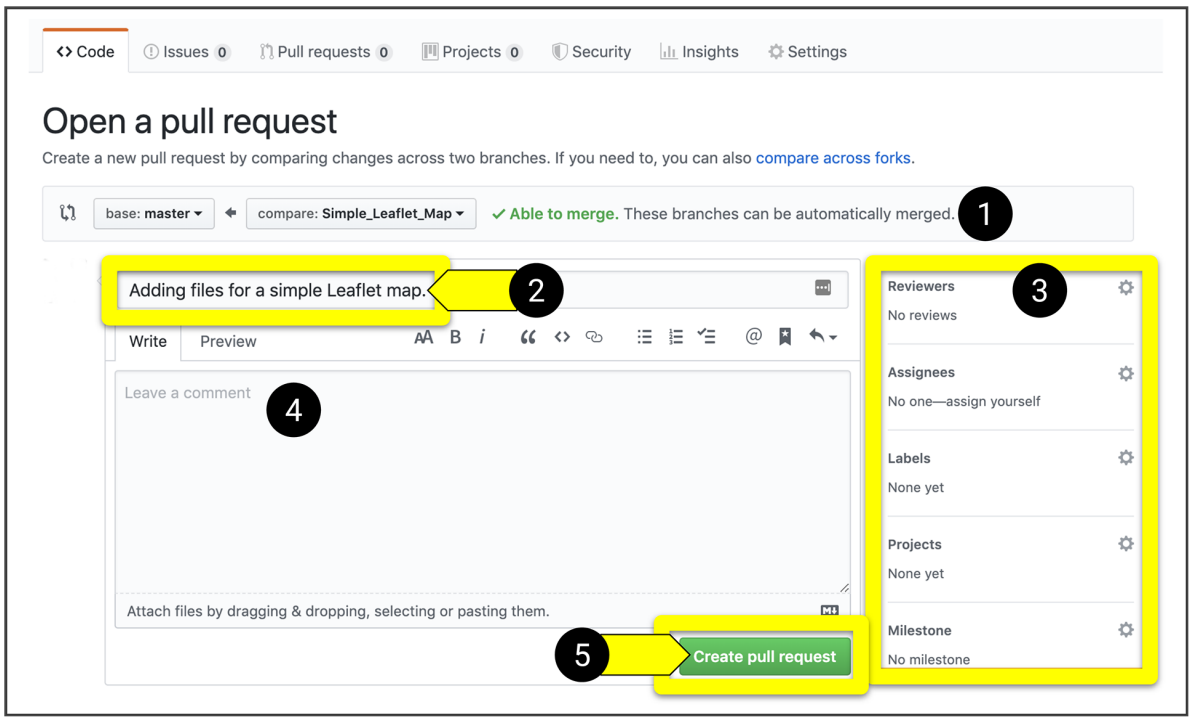
Let's walk through the process of creating a pull request. First, select the Simple_Leaflet_Map branch.



Or, you can select the green "Compare & pull request" button.



A new page called “Open a pull request” will launch.



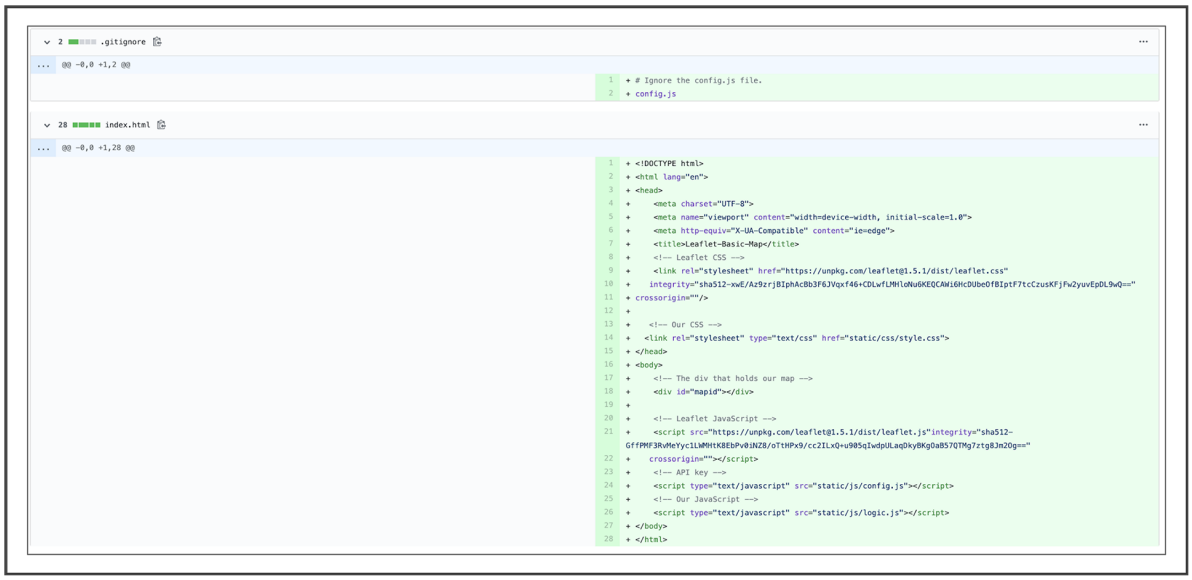
The page shows the five key elements of a pull request:

- #1: Near the top of the page is a green checkmark and text stating “Able to merge.” If you’re unable to merge, GitHub will display an explanatory message.
- #2: Next is the commit message you made.
- #3: “Reviewers” and “Assignees” are members, such as yourself, with access to the repository. You can assign the pull request to yourself. Labels, Projects,

and Milestones are completed by team members and owners of the repository. Click on a gear wheel to modify any of these items.

- #4: In the “Leave a comment” field, describe what you’re adding to the master branch in the pull request.
- #5: After entering a comment, click the green "Create pull request" button.

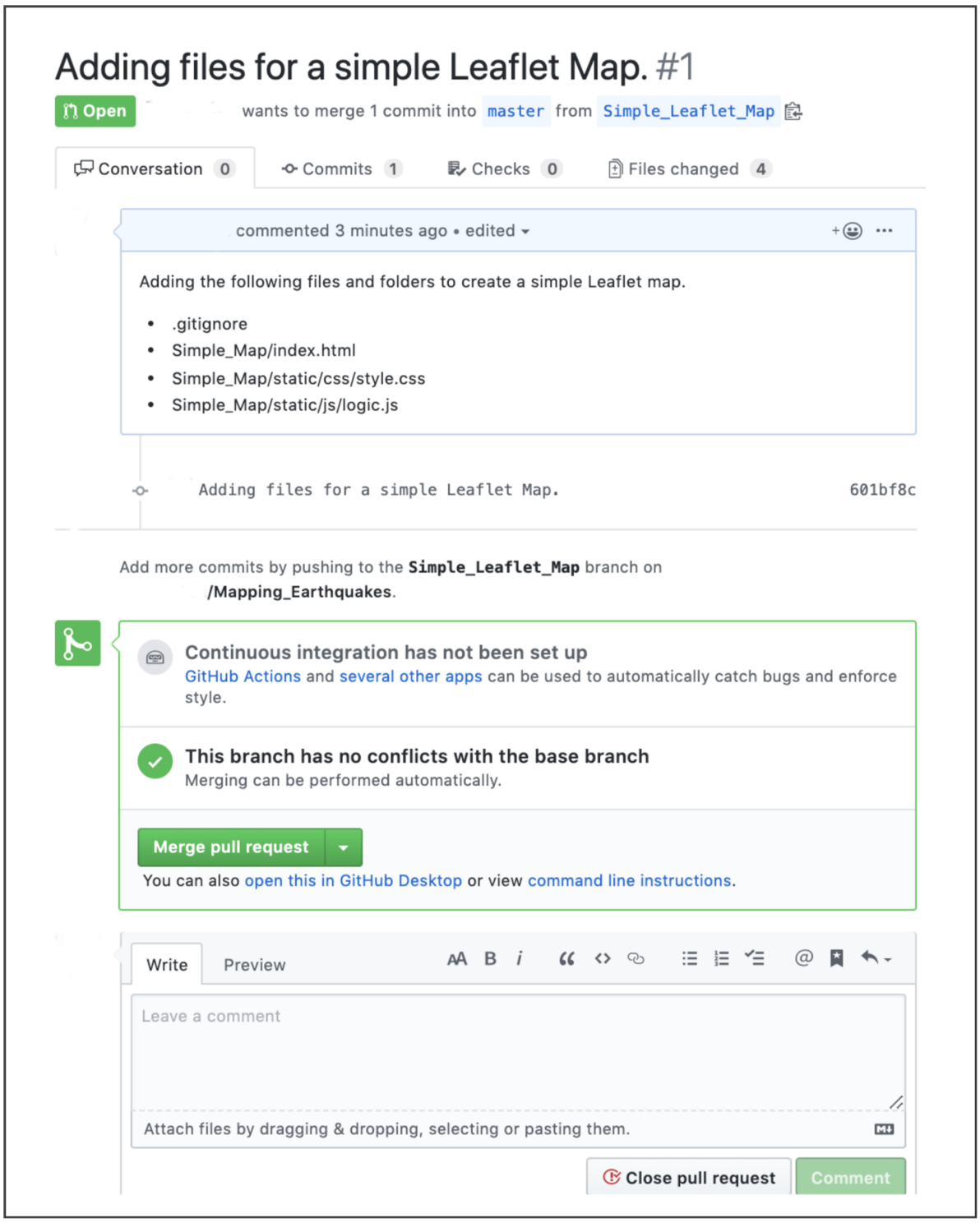
Scroll below “Open a pull request” to see the files you’re adding to the pull request.



Write a message and create a pull request. In the “Leave a comment” field, type the following message:

```
Adding the following folders and files to create a simple Leaflet map.
- .gitignore
- Simple_Map/index.html
- Simple_Map/static/css/style.css
- Simple_Map/static/js/logic.js
```

After you click the green "Create pull request" button, the page should look similar to the following:



Reviewers can add a comment in the “Leave a comment” field. For example, Sadhana or Basil could type approval and further instructions: “Looks good to me. You can merge.” Once a reviewer approves your pull request in a comment, you can click the green Merge pull request button.

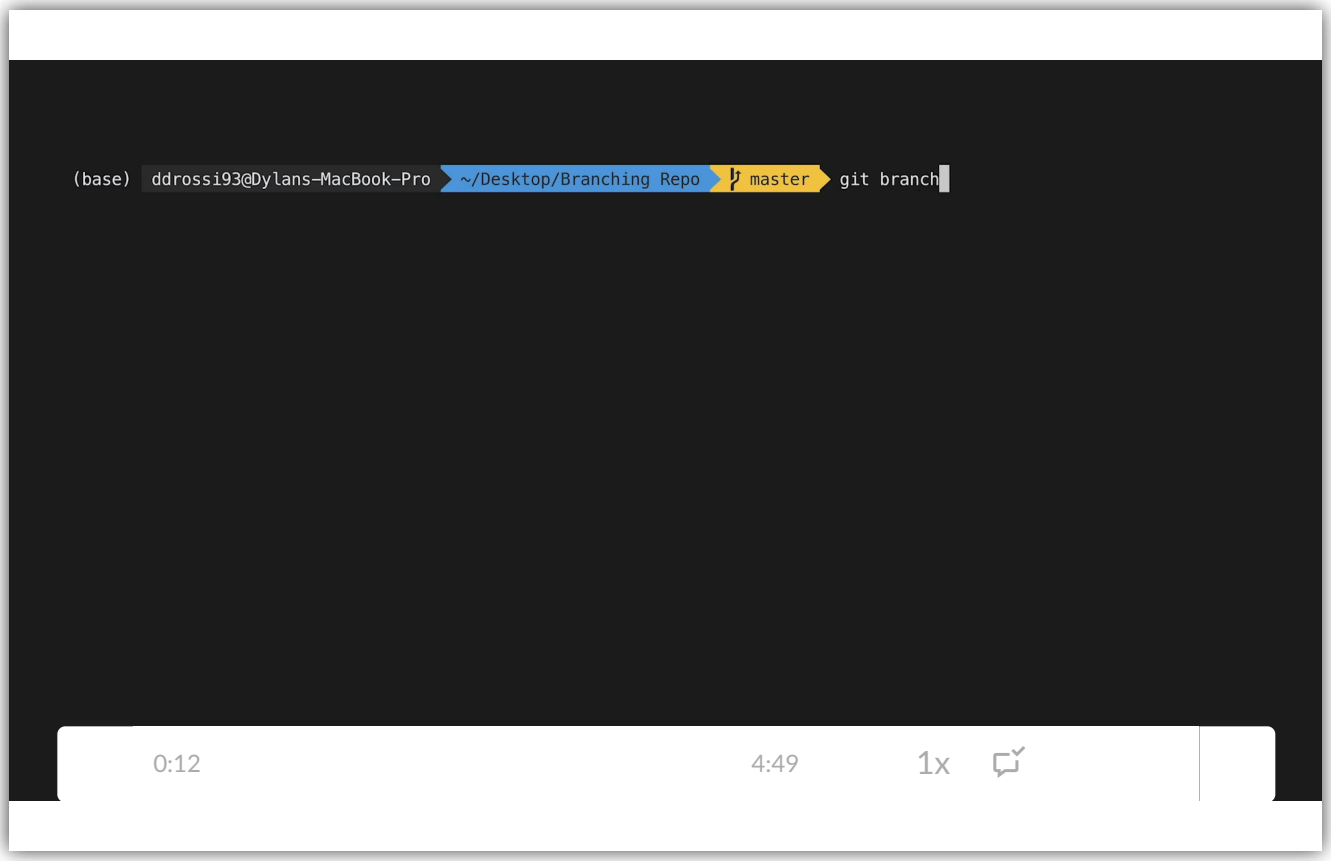
If your code looks good, click the “Merge pull request” button.

Don’t navigate away from GitHub. Next, Sadhana will walk you through the final steps of merging your branch with the master branch.

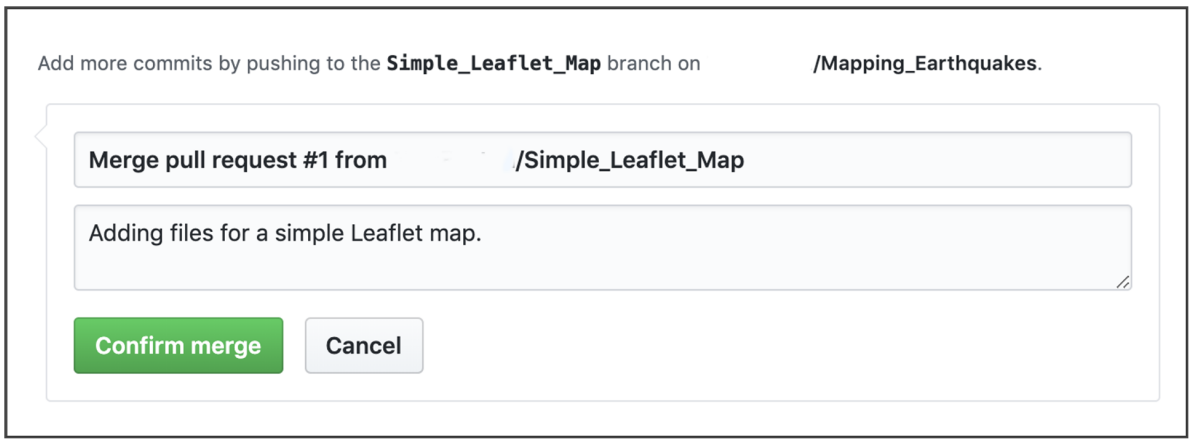
13.3.4: Merge Branch Into Master Branch

Congratulations on creating a pull request, which can be daunting sometimes. The final step will be to merge the Simple_Leaflet_Map branch into the master branch.

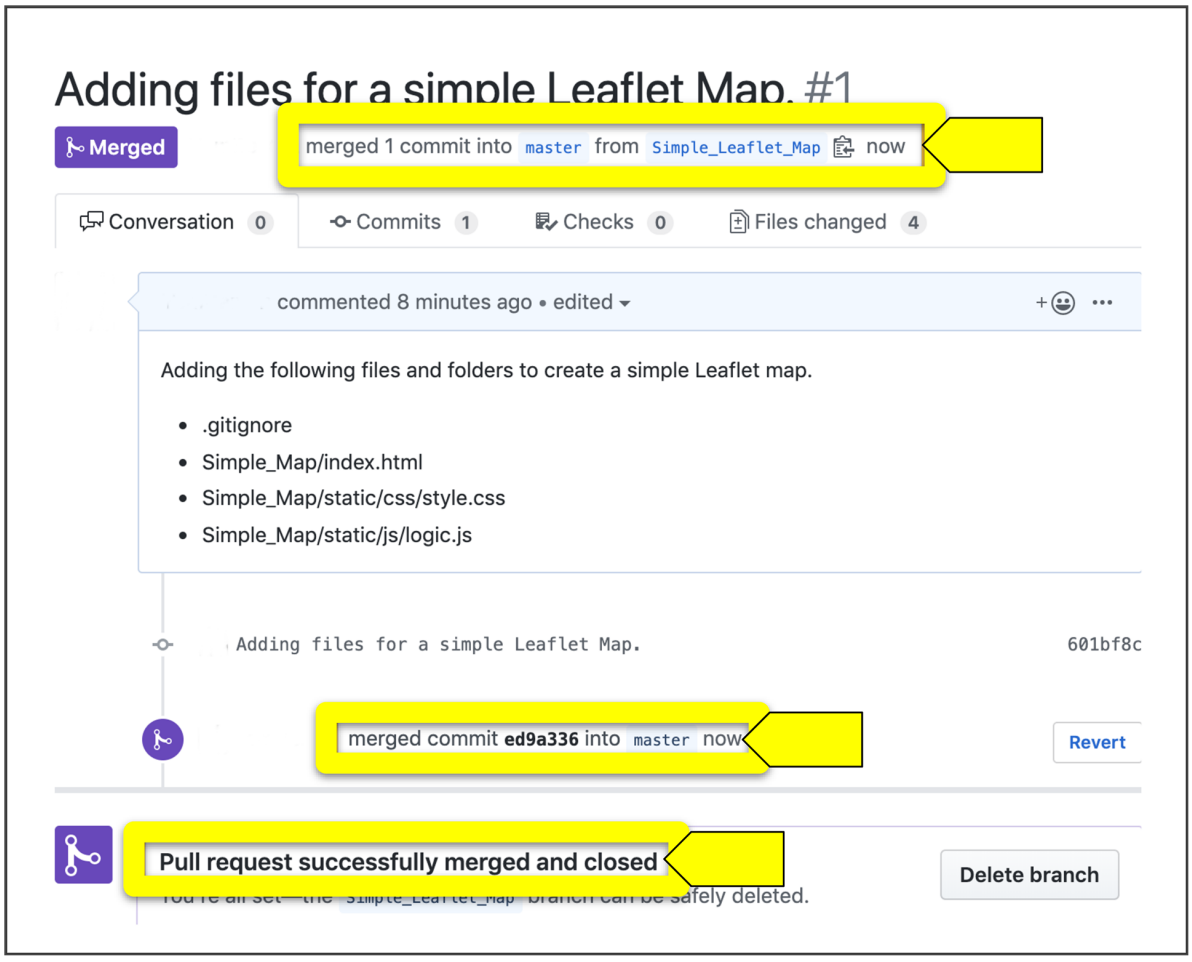
The following video walks you through the process of merging a branch into the master branch.



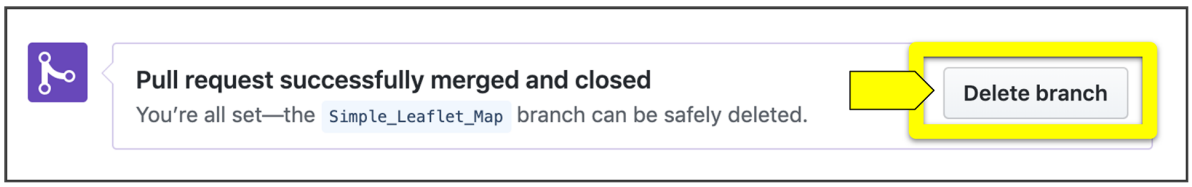
After clicking the "Merge pull request" button, the page will refresh with two options: "Confirm merge" or "Cancel."



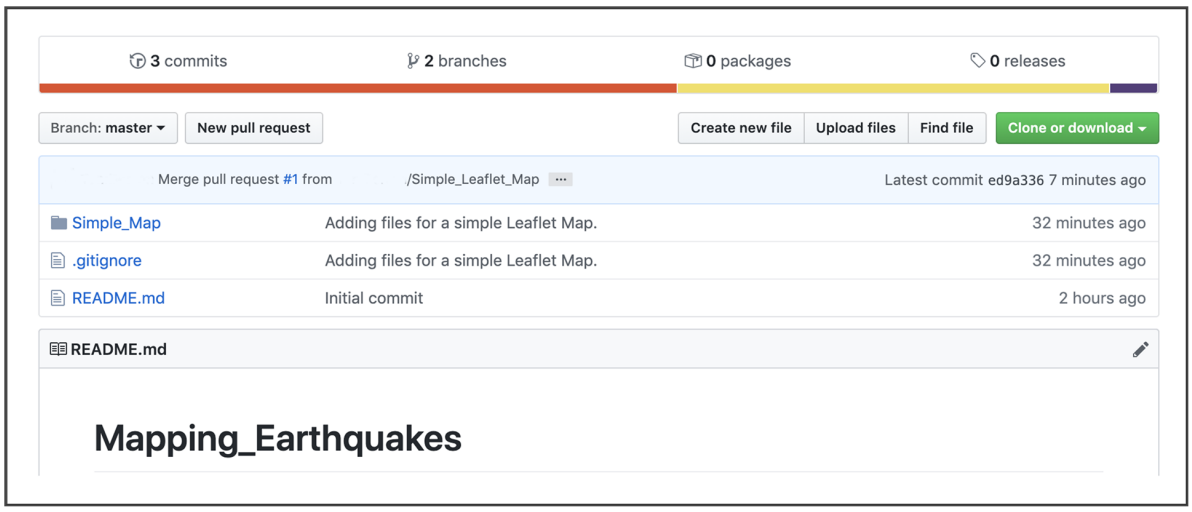
Click the green “Confirm merge” button. The page refreshes to confirm, in three places, that the pull request has been merged into the master branch.



This page gives the option of deleting the branch. However, don’t delete it so that future interns can use it to create a simple Leaflet map.



Navigate to the master branch on GitHub to confirm all the folders and files have been merged from the Simple_Leaflet_Map branch.



We are almost done!

Next, pull the latest changes on the master branch on our computer because the master branch on our computer is not up to date. Follow these steps.

1. Open the terminal or Git Bash and type `git checkout master` and press Enter.
2. Type `git status` and press Enter. The output might tell you that you are up to date, which can be misleading, or it might say you are “1” commit behind the Simple_Leaflet_Map branch.
3. Type `git pull` and press Enter. The output should look like the following:

```
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From https://github.com/         /Mapping_Earthquakes
   dc0f6f4..ed9a336  master     -> origin/master
Updating dc0f6f4..ed9a336
Fast-forward
 .gitignore                    |  2 ++
 Simple_Map/index.html         | 29 ++++++++++++++++++++++++++++++
 Simple_Map/static/css/style.css |  8 ++++++++
 Simple_Map/static/js/logic.js   | 16 ++++++++++++++++
 4 files changed, 55 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Simple_Map/index.html
 create mode 100644 Simple_Map/static/css/style.css
 create mode 100644 Simple_Map/static/js/logic.js
```

4. Next, type `open .` to launch your folder on your computer, or open the folder using VS Code to confirm that the files are in the master branch.

Congratulations on merging your Simple_Leaflet_Map branch into the master branch!

13.4.1: Map a Single Point

Creating a simple Leaflet map was relatively straightforward. Now Sadhana will show you how to add a single marker to a map and change the radius of the marker. However, she would like you to create a branch for adding points to a map for the GitHub repository so that new interns and employees can use this as a tutorial.

Now that we can create a simple Leaflet map, we can plot data on the map. First, let's create a new branch. Sadhana suggests that we name this branch "Mapping_Single_Points" since we'll map single points.

> **Rewind**
>
> Follow these steps to create a branch off of the master branch:
>
> 1. Navigate to your repository on your computer.
> 2. Make sure you're on the master branch by typing:
>   `git branch`
> 3. If you're not on the master branch, type:
>   `git checkout master`
> 4. Pull the changes from the master branch by typing:
>   `git pull`
> 5. Create a new branch by typing:
>   `git checkout -b [name_of_your_new_branch]`

In your new branch, we'll add a new folder inside the Mapping_Earthquakes folder. Since we're going to work with the same file names in the same folder structure, we'll use the same folder structure as we did for the Simple_Leaflet_Map branch.

Set up the folder structure as follows:

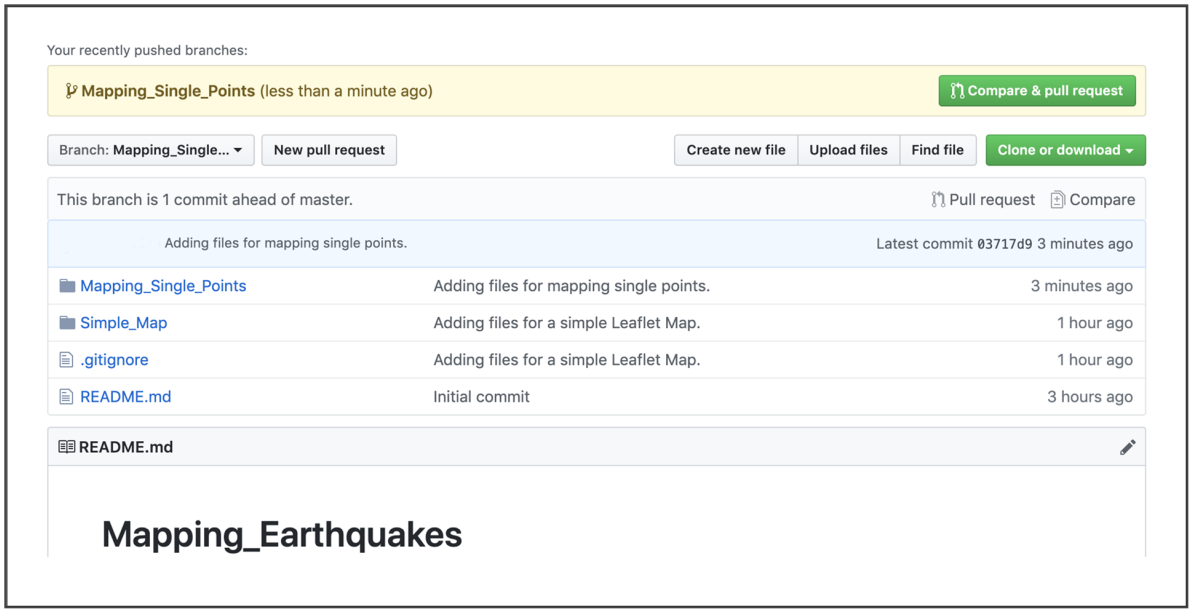
- Mapping_Single_Points
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

The two files that we'll change most often are the `index.html` and the `logic.js` files. Also, we might add an external file in the js folder. After checking out the new Mapping_Single_Points branch, copy all files from your Simple_Leaflet_Map folder and add them to a new Mapping_Single_Points folder..

Next, push the latest changes to the Mapping_Single_Points branch to GitHub.

> **Rewind**
>
> Follow these steps to push changes to a new branch:
>
> 1. Type:
>   `git status`
> 2. Add the folders and files by typing:
>   `git add .`
> 3. Confirm the correct files will be added by typing:
>   `git status`
> 4. Commit the changes by typing:
>   `git commit -m`
> 5. Push the changes to the branch by typing:
>   `git push --set-upstream origin Mapping_Single_Points`

After adding the folders and files to your Mapping_Single_Points branch, your repository should look like the following:



Next, we'll edit the `logic.js` file to add single points or markers to the basic map.

Add a Marker to the Map

Adding a marker to our simple map requires only one line of code, found in the **Leaflet Quick Start Guide (https://leafletjs.com/examples/quick-start/)**, under the "Markers, circles and polygons" subheading. Below the map is a line of code that reads as follows:

```
var marker = L.marker([51.5, -0.09]).addTo(map);
```

We're going to edit this line of code with the latitude and longitude for Los Angeles, California, and add it to our `logic.js` file that we used to create a simple map.

Open up the `logic.js` file using VS Code and add the following line of code before our `tileLayer()` code, and save the `logic.js` file:

```
//  Add a marker to the map for Los Angeles, California.
let marker = L.marker([34.0522, -118.2437]).addTo(map);
```

Next, open the `index.html` file in your browser. Your map should look like the following:

Next, we'll change the marker to a circle.

Add a Circle to the Map

To change the marker on our map to a point or dot, we'll use the `circle()` function. The `circle()` function will place a circle on the map at the given coordinates. The syntax for using the `circle()` function follows:

```
L.circle([34.0522, -118.2437], {
   radius: 100
}).addTo(map);
```

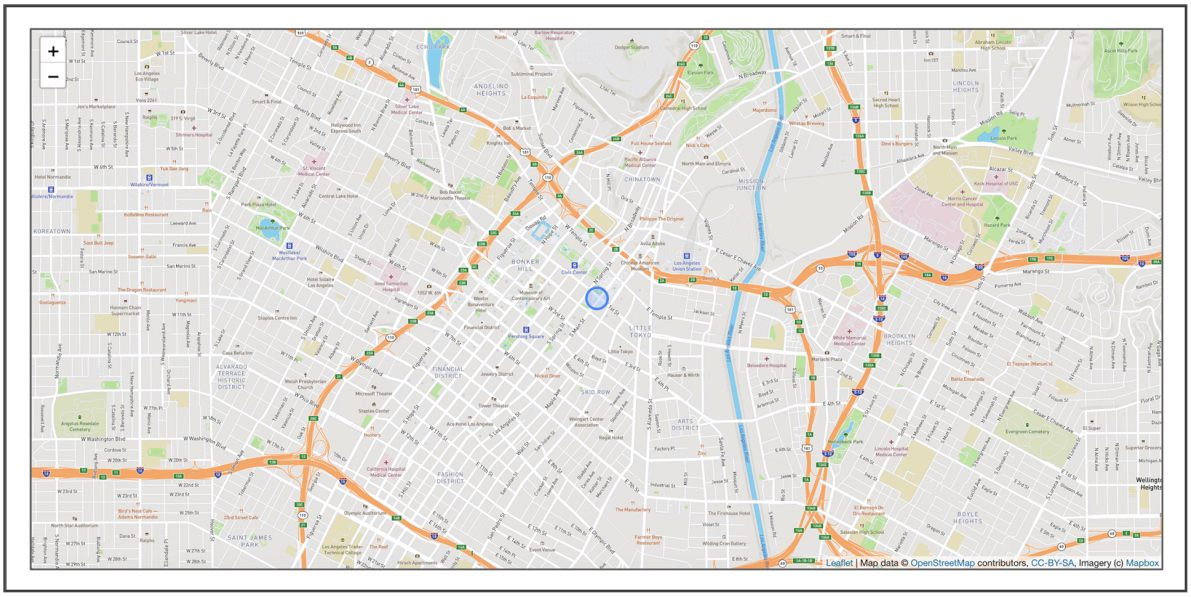
When using the `circle()` function, the default is just a small dot on the map, but we want to adjust the radius so that it's bigger and easier to see. The radius for the `circle()` function is measured in meters.

For the code above, add a circle with a 100-meter radius over Central Los Angeles when we assign a value to the `radius` key in the `circle()` function.

Copy the code for the `circle` function and replace it with the `marker()` function we used previously. We're also going to zoom in to a level of 14 on the `setView()` method. After editing your `logic.js` file, it should look like the following:

```
// Create the map object with a center and zoom level.
let map = L.map('mapid').setView([34.0522, -118.2437], 14);

// Add a marker to the map for Los Angeles, California.
L.circle([34.0522, -118.2437], {
    radius: 100
}).addTo(map);
```

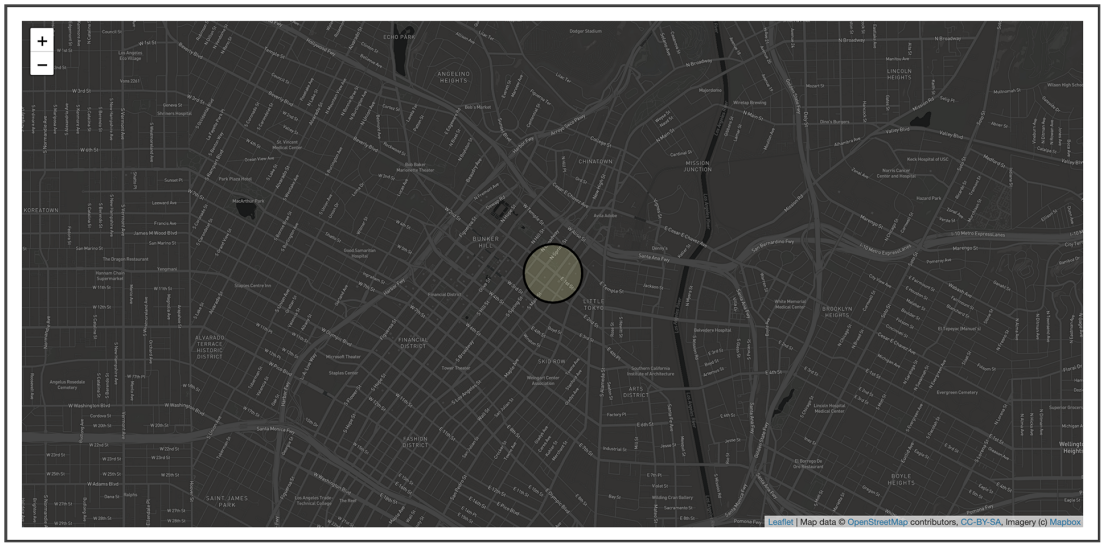
When we open our `index.html` file in our browser, it will show a circle over Central Los Angeles.



Now test your skills in the following Skill Drill:

> **Skill Drill**
>
> Using the Leaflet documentation, create a light-yellow circle with black lines indicating a 300-meter radius of Central Los Angeles on a dark map.

Your map should look like the following:



Alternatively, we can create a circle using the `circleMarker()` function. The `circleMarker()` function measures the radius of the circle in pixels, with the default radius set at 10 pixels. The syntax for using the `circleMarker()` function follows:

```
L.circleMarker([34.0522, -118.2437]).addTo(map);
```

Let's create a light-yellow circle with black lines indicating a 300-pixel radius on a dark map. Edit your `logic.js` file from the previous Skill Drill by changing your `circle()` function to a `circleMarker()` function. Your `logic.js` file should now look like the following:

```
// Add a circle to the map
L.circleMarker([34.0522, -118.2437], {
    radius: 300,
    color: "black",
    fillColor: '#ffffa1'
}).addTo(map);
```

If you didn't get the correct map style in the Skill Drill, replace the "streets-v11" in our `tileLayer()` code with "dark-v10" to look like the following:

```
// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/dark-v
```

Save your `logic.js` file and open your `index.html` file in our browser. The circle will show a 300-pixel radius of Central Los Angeles.

Wow! What a big difference between the `circle()` and `circleMarker()` functions.

Remember, it's a best practice to commit early and often! Before you commit your code for the Mapping_Single_Points branch to GitHub, check to see if all the files will be tracked in the branch.

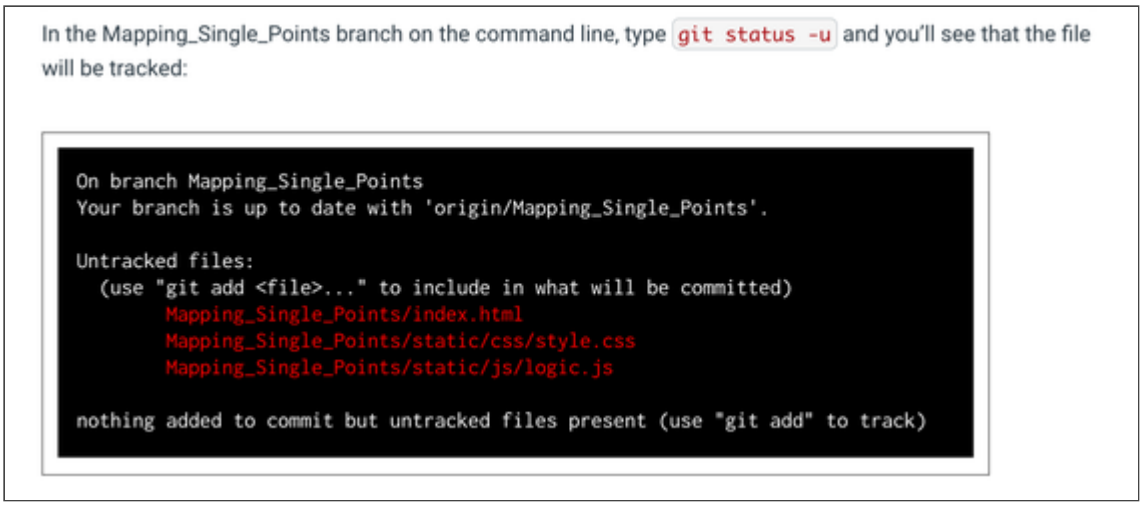
How would you check to see which files will be tracked in the branch? (Select all that apply)

- [ ] `git status`
- [ ] `git status -i`
- [ ] `git status -u`

Check Answer

Finish ▸

In the Mapping_Single_Points branch on the command line, type `git status` and you'll see that the `logic.js` file will be tracked:



Great job! Now, commit and push these files to the Mapping_Single_Points branch. Don't delete the branch, so that others can use it to learn how to map single points.

Next, Sadhana is going to show you how to add multiple locations to a map and change the radius of each marker.

> **Note**
>
> Use the links below to learn more about these Leaflet functions:
>
> - **marker() function (https://leafletjs.com/reference-1.5.0.html#marker)**

- **circle() function (https://leafletjs.com/reference-1.5.0.html#circle)**
- **circleMaker() function (https://leafletjs.com/reference-1.5.0.html#circlemarker)**

13.4.2: Map Multiple Points

Now that you have added a single marker to a map and changed some of the features of the marker, Sadhana wants you to iterate through an array of objects and map them. Other employees really like the branches you created, so Sadhana would like you to create a new branch for adding multiple points to a map.

Before we plot multiple markers and points, Sadhana wants you to create a new branch for mapping multiple points.

Create a new branch called “Mapping_Multiple_Points” with the following folder structure:

- Mapping_Multiple_Points
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

Copy the necessary folders and files from your Mapping_Single_Points branch and add them to the Mapping_Multiple_Points folder.

Add Multiple Markers

When we added a single marker to our simple map, we assigned our `marker` variable to the Leaflet class `marker()` function. This function will only add one latitude and longitude to the map. To add more markers to the map, the latitudes and longitudes are usually nested in an array. To add a marker for each location, we have to iterate through the array and add each latitude and longitude to the map.

First, in the `logic.js` file, replace the `marker` variable (which we used to map one location) with the `cities` variable that references the five most populous cities array in the following code block. Then save the file:

```
// An array containing each city's location, state, and population.
let cities = [{
  location: [40.7128, -74.0059],
  city: "New York City",
  state: "NY",
  population: 8398748
},
{
  location: [41.8781, -87.6298],
  city: "Chicago",
  state: "IL",
  population: 2705994
},
{
  location: [29.7604, -95.3698],
  city: "Houston",
  state: "TX",
  population: 2325502
},
{
  location: [34.0522, -118.2437],
  city: "Los Angeles",
  state: "CA",
  population: 3990456
```

```
},
{
  location: [33.4484, -112.0740],
  city: "Phoenix",
  state: "AZ",
  population: 1660272
}
];
```

Next, we need to iterate through each `city` object and add each city location to the `marker()` function, which will, in turn, be added to the map.

How would you iterate through the cities array? (Select all that apply.)

- [ ] `for (let i = 0; i < cities; i++)`
- [ ] `for (let i = 0; i < cities.length; i++)`
- [ ] `for (let i = 0, i < cities.length, i++)`
- [ ] `cities.forEach(function(city))`

Check Answer

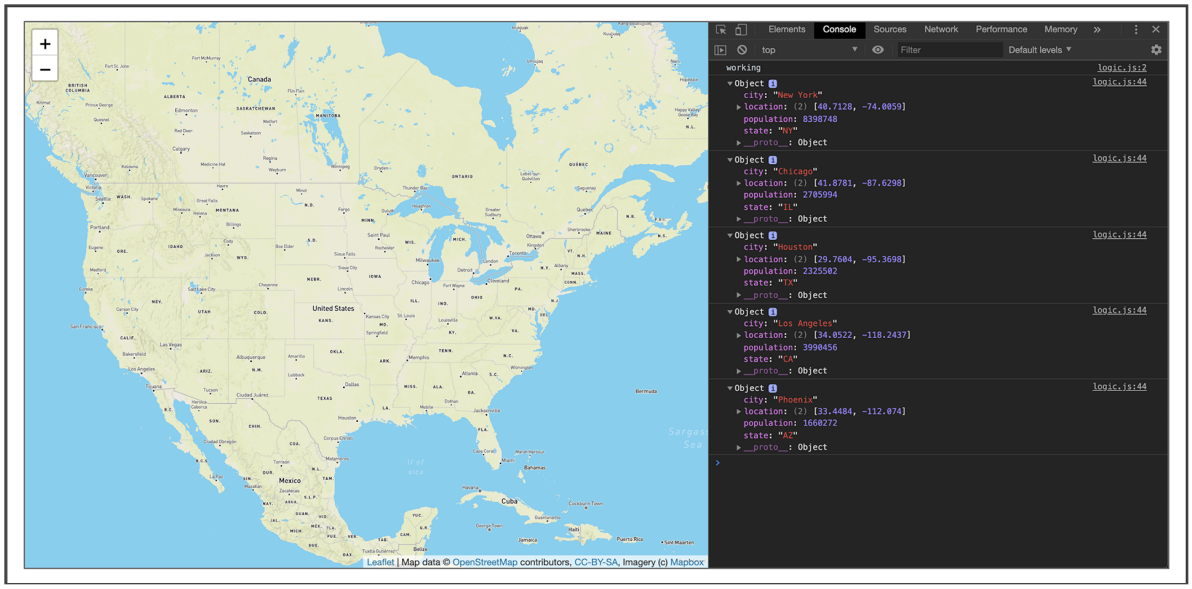
Finish ▸

Below the cities array, add the following code to iterate through the array. Inside the brackets, use the `console.log()` function to print each object in the array to the console:

```
// Loop through the cities array and create one marker for each city.
cities.forEach(function(city) {
 console.log(city)
});
```

Save the `logic.js` file and open the `index.html` file in your browser.

If we look at the console tab, we'll see that each object, or city, of the `cities` array is printed to the console.



Now, add each city's location to the map by adding the location to the `marker()` function.

Complete the code that will add the location of each city to the map when you iterate through the `cities` array.

```
cities.forEach(function(city) {

    console.log(city)

    L.marker(                ).addTo(map);

});
```

Check Answer

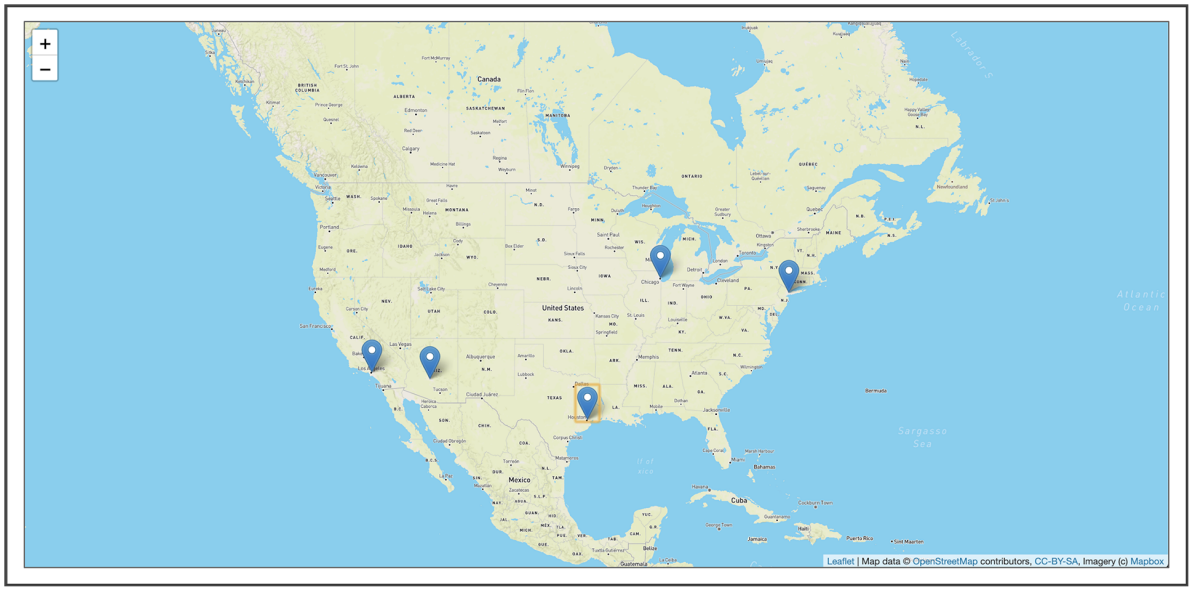
Finish ▸

In the `forEach()` function, assign the `city` variable to each object of the `cities.js` file. Then, get the coordinates of each city by adding `city.location` in the `L.marker()` function. We can then add each location to the map with the `addTo()` function and pass the `map` object as the argument.

Add the following code to your `logic.js` file and save it:

```
// Loop through the cities array and create one marker for each city.
cities.forEach(function(city) {
    console.log(city)
    L.marker(city.location).addTo(map);
});
```

When you open the `index.html` file in your browser, the map will show a marker on each city in the `cities` array.



When handling large datasets, it's a best practice to have the data in an external file and refer to that file and dataset in the `logic.js` file.

Even though our `cities` array is not that large, let's create a new file in the "js" folder called `cities.js`. Cut the `cities` array data from the `logic.js` file, place it in the `cities.js` file, and save the file.

Next, in the `logic.js` file, where the `cities` array was located, add a variable and assign it to the `cities` array. Add the following code to the `logic.js` file:

```
// Get data from cities.js
let cityData = cities;
```

Now the `cities` array is assigned to the `cityData` variable, which means we'll need to replace `cities` with `cityData` in our `forEach()` function. Edit the `forEach()` function so that it looks like the following and save the `logic.js` file:

```
// Loop through the cities array and create one marker for each city.
cityData.forEach(function(city) {
    console.log(city)
    L.marker(city.location).addTo(map);
});
```

Now open the `index.html` in your browser to confirm these changes worked.

Uh-oh! Something went wrong, as shown in the following image:

How would you determine why your web browser didn't show a map?

- Clear the cache and history in the web browser and open the `index.html` file again.
- Open the `index.html` file in another browser.
- Inspect the page using the DevTools.

Check Answer

Finish ▸

After you inspect the page using the DevTools, the console might have an error message that says `Uncaught ReferenceError: cities is not defined`. This means the `cities` array data can't be found.

To correct this error, in the body of the `index.html` file and before the path to the `logic.js` script, add a `<script>` file with the path to the JavaScript `cities.js` file, like this:

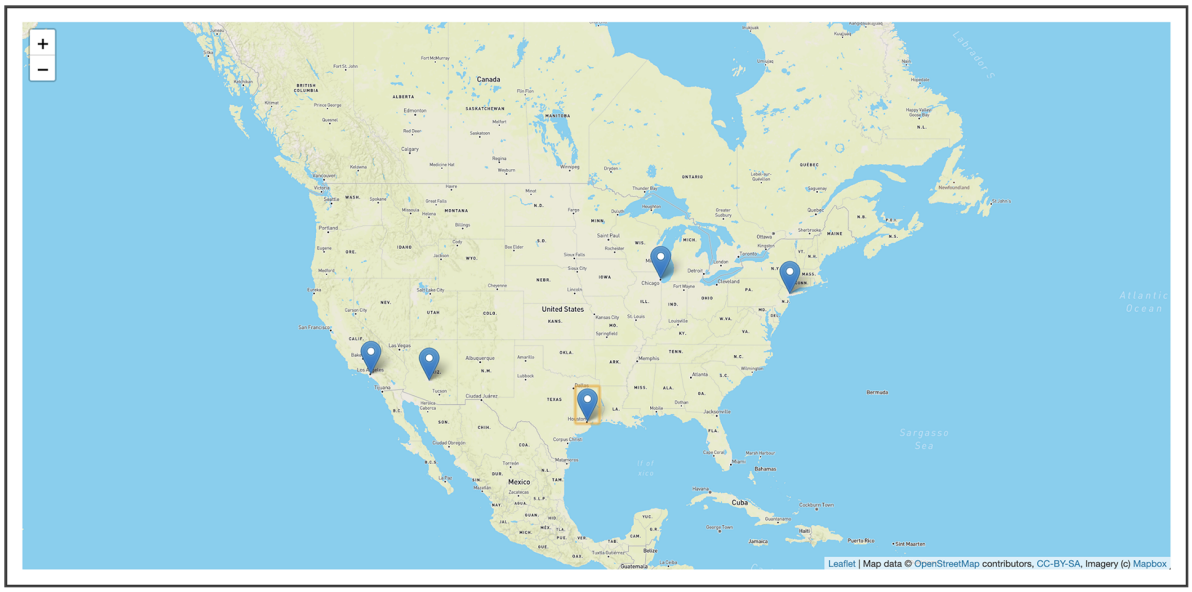
```
<script type="text/javascript" src="static/js/cities.js"></script>
```

After adding the `<script>` file, the body of our `index.html` file should look like the following:

```
<body>
    <!-- The div that holds our map -->
    <div id="mapid"></div>

    <!-- Leaflet JavaScript -->
    <script src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
integrity="sha512-gZwIG9x3wUXg2hdXF6+rVkLF/0Vi9U8D2Ntg4Ga5I5BZpVkVxlJWbSQtXPSiUTtC0TjtGOmxa1AJPuV0CPthew=="
crossorigin=""></script>
    <!-- API key -->
    <script type="text/javascript" src="static/js/config.js"></script>
    <!-- Our JavaScript -->
    <script type="text/javascript" src="static/js/cities.js"></script>
    <script type="text/javascript" src="static/js/logic.js"></script>

</body>
```

Now, when we open up the `index.html` in our browser, the map should look like it did before we created the `cities.js` file and edited the `logic.js` and

`index.html` files.



Bind a Popup to the Marker

To add data from each object in the cities array, we'll use Leaflet's `bindPopup()` method on the `marker()` function. According to the guidance in the **Quick Start Guide (https://leafletjs.com/examples/quick-start/)**'s "Working with popups" section, we only need to add HTML code inside the parentheses of the `bindPopup()` method:

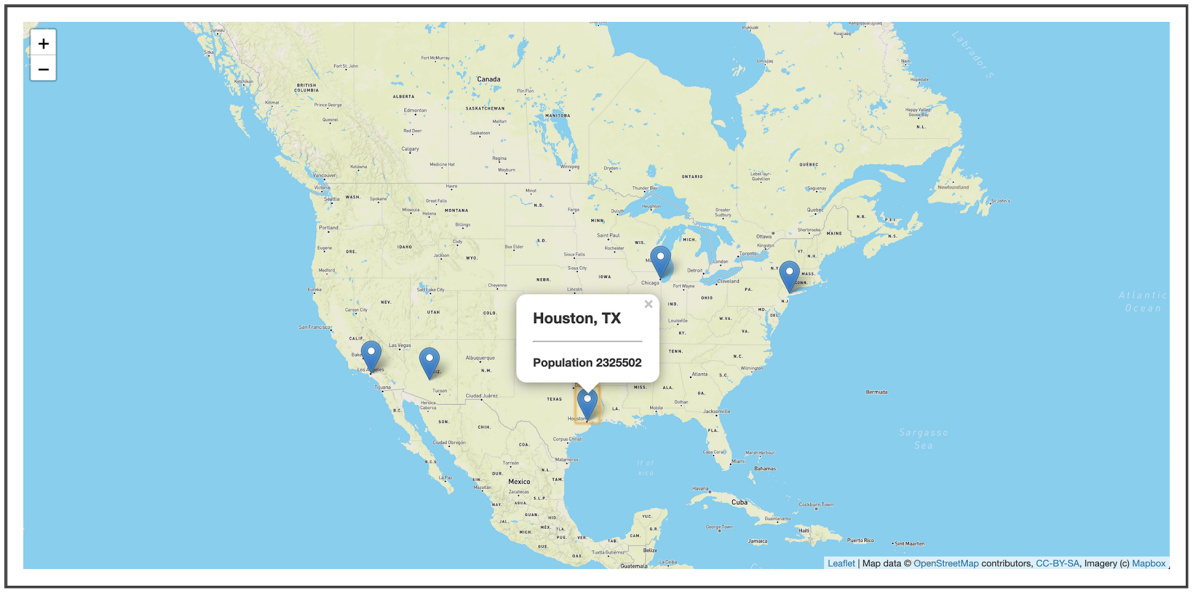
> Popups are usually used when you want to attach some information to a particular object on a map. Leaflet has a very handy shortcut for this:
>
> ```
> marker.bindPopup("Hello world!<br>I am a popup.") .openPopup();
> circle.bindPopup("I am a circle.");
> polygon.bindPopup("I am a polygon.");
> ```
>
> Try clicking on our object. The `bindPopup` method attaches a popup with the specified HTML content to your marker so the popup appears when you click on the the object, and the `openPopup` method (for markers only) immediately opens the attached popup.

In the `logic.js` file, edit the `forEach` function and add the `bindPopup()` method. Inside the parentheses of the `bindPopup()` method, we'll retrieve the name of the city, state, and population.

Edit the `forEach` function to look like the following, save the `logic.js` file, and open the `index.html` file in your browser:

```
// Loop through the cities array and create one marker for each city.
cityData.forEach(function(city) {
    console.log(city)
    L.marker(city.location)
    .bindPopup("<h2>" + city.city + ", " + city.state + "</h2> <hr> <h3>P
  .addTo(map);
});
```

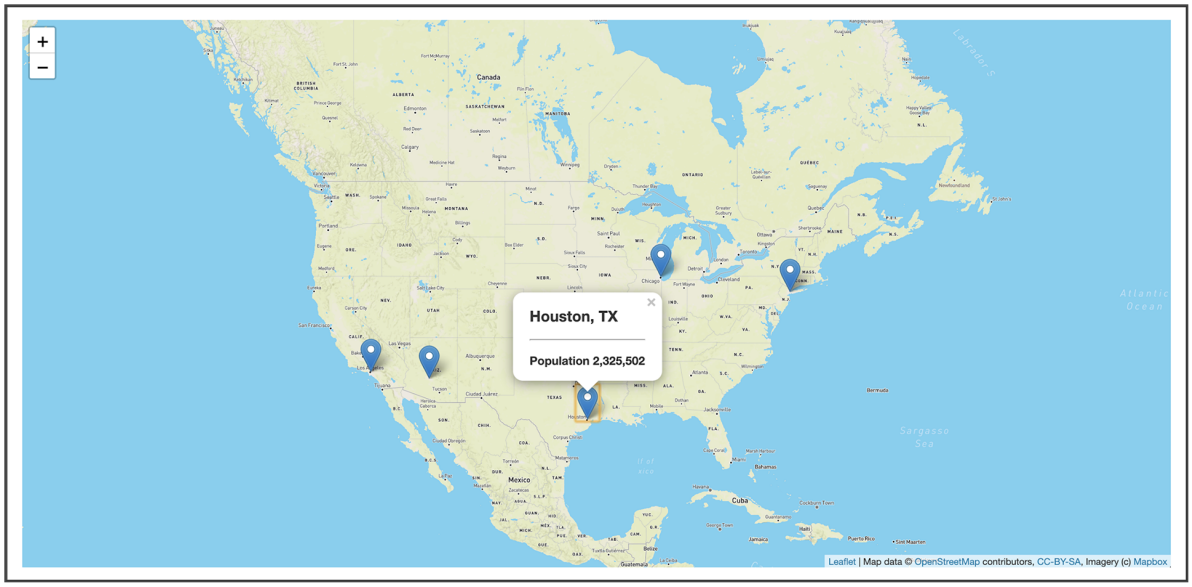
Now, when we click on each marker, it will show the name, state, and population of the city.



Let's format the population with a thousands separator by using the `toLocaleString()` method on the `city.population` in the `bindPopup()` method, like this:

```
// Loop through the cities array and create one marker for each city.
cityData.forEach(function(city) {
    console.log(city)
    L.marker(city.location)
    .bindPopup("<h2>" + city.city + ", " + city.state + "</h2> <hr> <h3>Population " + city.population.toLocaleString() + "</h3>")
    .addTo(map);
});
```

Now our popup markers have the population formatted with a thousands separator.



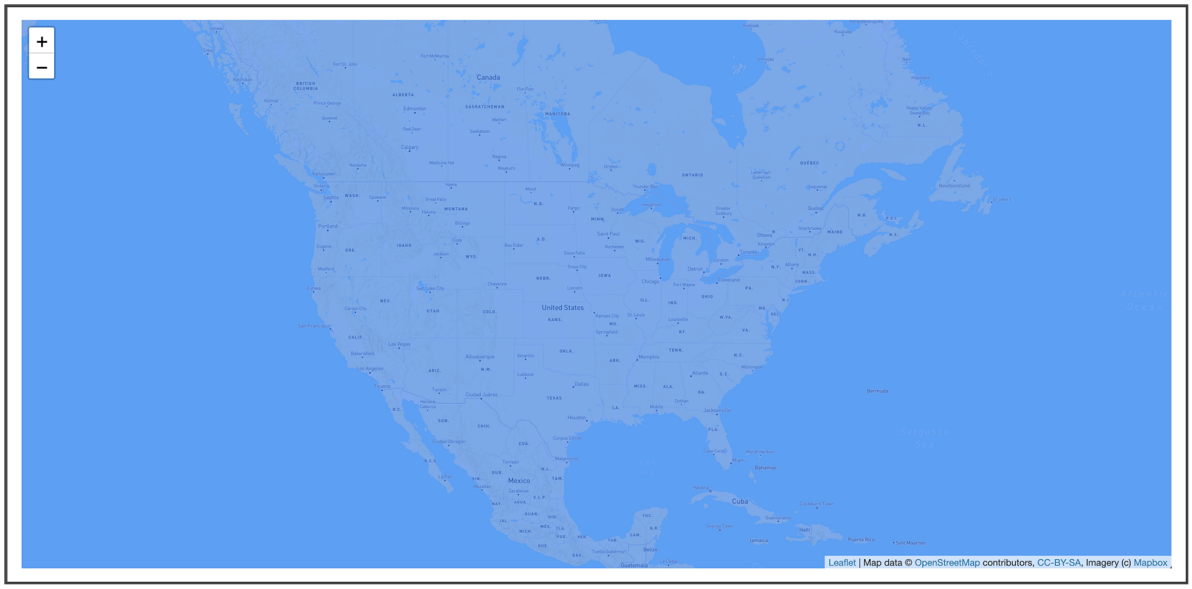
Next, change the marker for each city to a circle that has a radius equivalent to the city's population.

In the `logic.js` file, we'll replace the `marker()` function with the `circleMarker()` function in the `forEach()` function. Then we'll assign the "radius" key to the population by using `city.population`.

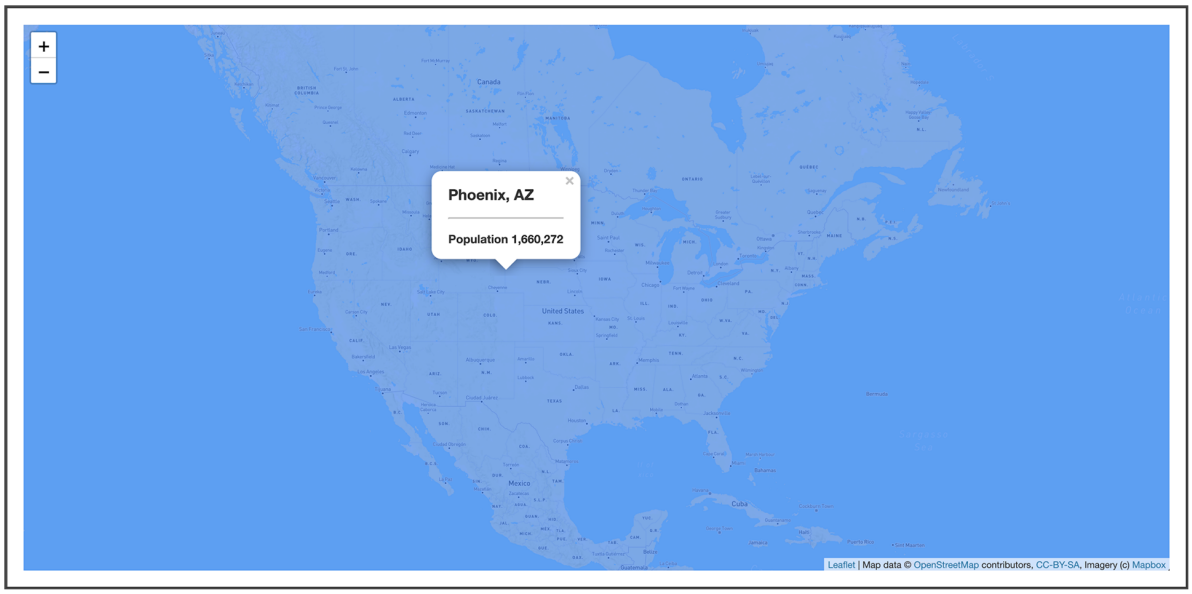
The `forEach()` function in our `logic.js` file should look like the following:

```
// Loop through the cities array and create one marker for each city.
cityData.forEach(function(city) {
  console.log(city)
  L.circleMarker(city.location, {
    radius: city.population
  })
  .bindPopup("<h2>" + city.city + ", " + city.state + "</h2> <hr> <h3>Population " + city.population.toLocaleString() + "</h3>")
  .addTo(map);
});
```

After you save the `logic.js` file and open the `index.html` file in your browser, your map will look like the following:



Well, that doesn't look like the map from before! If we click on that map, "Phoenix, AZ" and its population appear in a popup.



We know that the data is being loaded onto the map, but what is the problem?

Why is the map covered in the color of the circle marker, and why does “Phoenix, AZ” pop up when you click anywhere on the map?

- The color of the circle is incorrect.
- We need to use the `circle()` function.
- The radii are too large.

Check Answer

Finish ▸

The problem with the map is that the radii are too large and don’t fit on the map. To fix this, we’ll have to decrease each city’s radius so the circle markers fit on the map. In the `logic.js` file, divide the `city.population` value by “100000” to look like this:

```
radius: city.population/100000
```

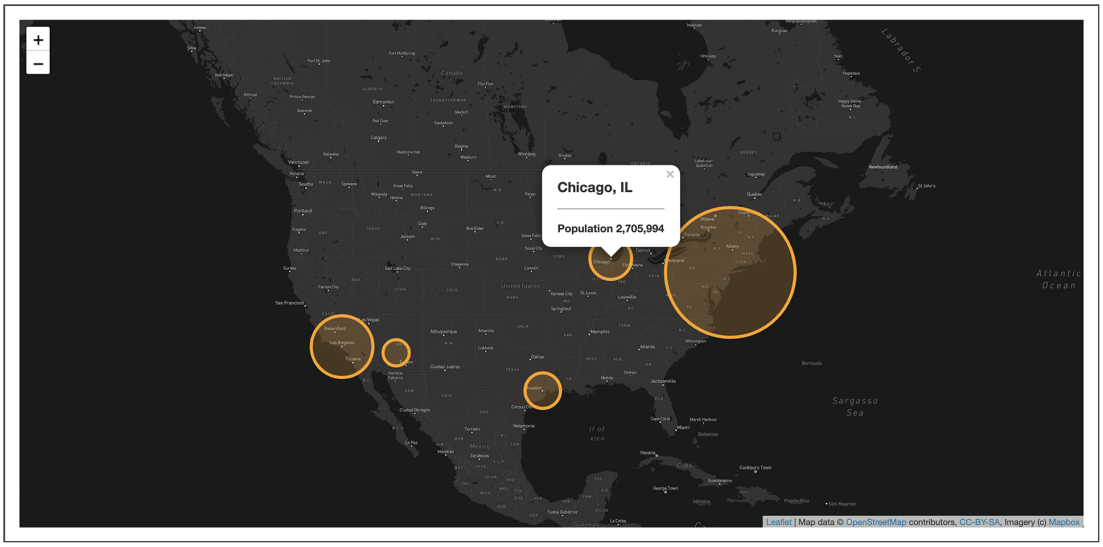
Now when we open the map in our browser, the radius for each city looks proportional to the population.

Congratulations on creating varying size circle markers with popup information!

Skill Drill

Edit the `logic.js` file to create an orange circle popup marker for each city, with a lineweight of 4, a radius where the population number is decreased by 200,000, that's on a dark map. When you click on the circle, the popup should display the city, state, and the population formatted with a thousands separator.

Your map should look similar to the following:



Next, Sadhana will show you how to plot lines on a map.

Add, Commit, Push

Add, commit, and push your changes to the Mapping_Mulitple_Points branch. Don't delete the branch so that others can use it to learn how to map multiple points with popup markers.

Note

For more information, see the **Leaflet documentation on the bindPopup() method (https://leafletjs.com/reference-1.5.0.html#popup)**.

13.4.3: Map Lines

Basil and Sadhana are ecstatic that you can add multiple locations to a map. This will be highly beneficial when you need to add the earthquake data to a map. Now, Sadhana will walk you through how to add lines to a map.

On our Leaflet map, we can plot coordinates to create lines between locations, like transportation routes.

Before we plot lines on a map, let's create a new branch called "Mapping_Lines" that has the following folder structure:

- Mapping_Lines
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

Copy the necessary folders and files from one of your Mapping_Mulitple_Points branches and add them to the Mapping_Lines folder.

Map a Single Line

Adding lines to a map requires that the coordinates for the starting and ending points be a one-dimensional array with two elements: latitude and longitude. To illustrate how lines are mapped, let's map the airline route from Los Angeles to San Francisco. Mapping airline routes will help us understand how tectonic plate data is added to a map.

The starting point for our line will be the Los Angeles International Airport (LAX), with the coordinates `[33.9416, -118.4085]`. The ending point for our line will be the San Francisco International Airport (SFO), with the coordinates `[37.6213, -122.3790]`.

When we create a line in Leaflet, the starting and ending points and all coordinates along the route need to be in an array. We can assign the array to the `line` variable like this:

```
// Coordinates for each point to be used in the line.
let line = [
  [33.9416, -118.4085],
  [37.6213, -122.3790]
];
```

Let's edit our `logic.js` file to create a line from LAX to SFO.

- First, change the coordinates for the center of the map to somewhere between LAX and SFO by adding `[36.1733, -120.1794]` in the `setView()` method.
- Change the zoom level in the `setView()` method to 7.
- Add the code above for our line below the `map` variable for the center of the map.
- Lastly, create a line on a map using the Leaflet `polyline()` function. Add the following line of code after the `line` variable:

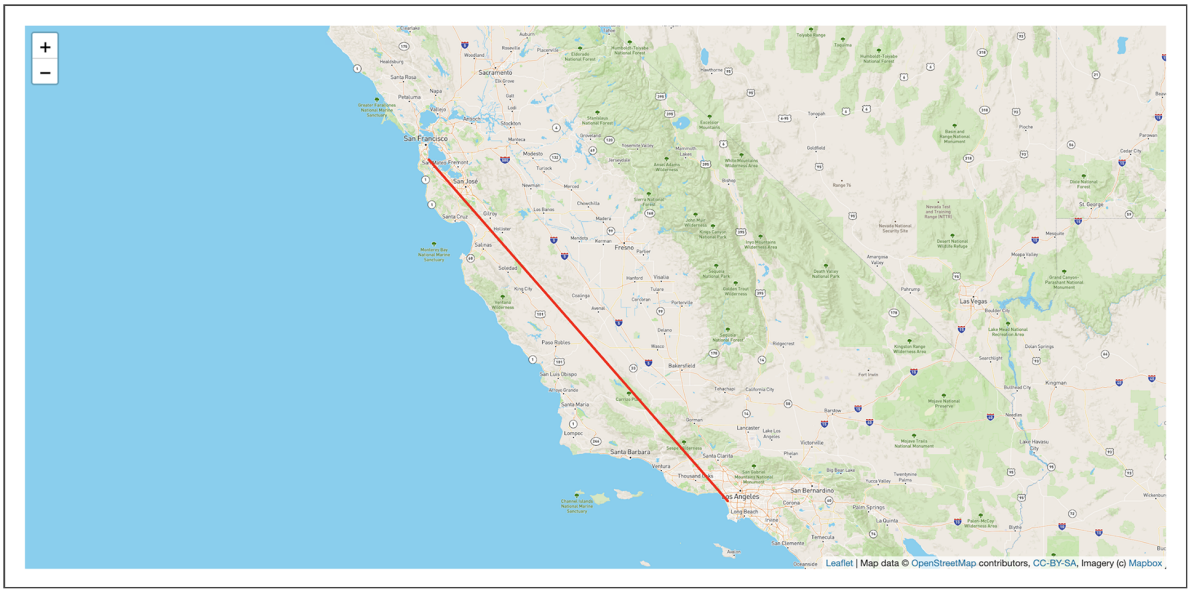
```
// Create a polyline using the line coordinates and make the line red.
L.polyline(line, {
  color: "red"
}).addTo(map);
```

In the `polyline()` function, we pass the line coordinates and the key-value pair `color: "red"` to make the line red.

Save the `logic.js` file with the changes. It should look like the following:

```
// We create the map object with options.
let map = L.map('mapid').setView([36.1733, -120.1794], 7);

// Coordinates for each point to be used in the polyline
let line = [
  [33.9416, -118.4085],
  [37.6213, -122.3790]
];

// Create a polyline using the line coordinates and make the line red.
L.polyline(line, {
  color: "red"
}).addTo(map);
```

When you open the `index.html` file in your browser, your map should have a red line between LAX and SFO.



Now we'll add a few more stops on our airline route.

Map Multiple Lines

Let's edit the `logic.js` file and add two more airport stops to our `line` variable: Salt Lake City International Airport (SLC) and Seattle-Tacoma International Airport (SEA). Follow these steps:

1. Edit the `line` variable in the `logic.js` file so that it includes the two new sets of coordinates.

```
// Coordinates for each point to be used in the polyline.
let line = [
  [33.9416, -118.4085],
  [37.6213, -122.3790],
  [40.7899, -111.9791],
  [47.4502, -122.3088]
];
```

2. Make the line yellow by editing the value for the "color" key in the `polyline()` function to `yellow`.

```
// Create a polyline using the line coordinates and make the line bla
L.polyline(line, {
   color: "yellow"
}).addTo(map);
```

3. Change the map style to "satellite-streets-v11."
4. Finally, change the center of the map to SFO and change the zoom to 5 so that we can see the line.

```
// Create the map object with center at the San Francisco airport.
let map = L.map('mapid').setView([37.6213, -122.3790], 5);
```

After you save the `logic.js` file and open the `index.html` file in your browser, your map should look like the following, showing the route from LAX, SFO, SLC, and SEA:

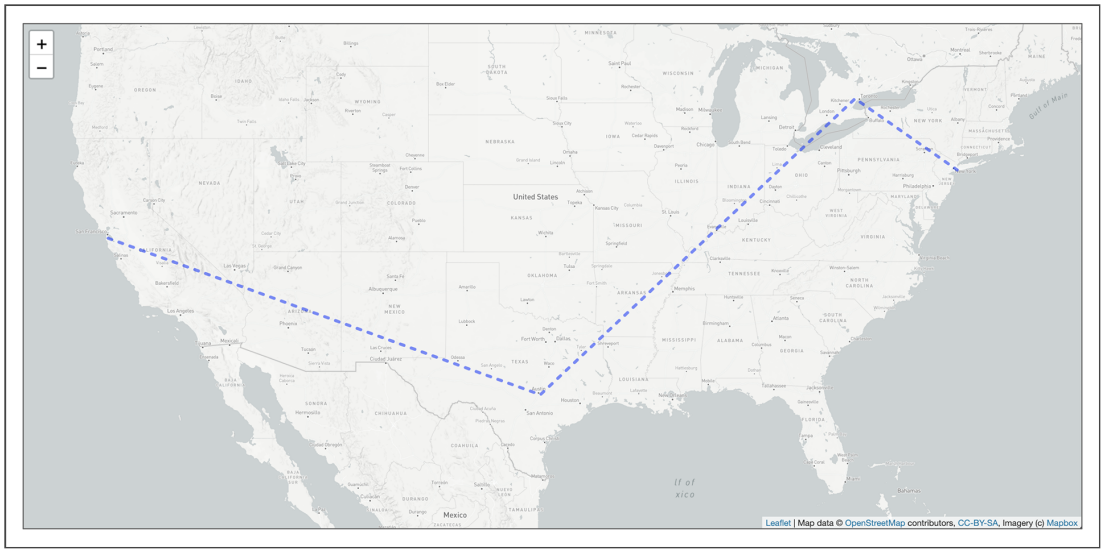
Skill Drill

Edit your `logic.js` to create an airline route from SFO to John F. Kennedy International Airport (JFK) with two stops, Austin-Bergstrom International Airport (AUS) and Toronto Pearson International Airport (YYZ). Make the route a blue dashed line, with a weight of 4 and opacity of 0.5 on the light map.

Hint: You'll need to find the coordinates for some of these airports.

Bonus: Add your city or another city as a stopping point.

Your map should look similar to the following:



Great job on mapping routes on your map!

Add, Commit, Push

Add, commit, and push your changes to your Mapping_Lines branch. Don't delete the branch so that others can use it to learn how to map lines.

After you push your changes to the branch, Sadhana will show you how to plot data from a GeoJSON (`.json`) file.

Note

For more information, see the **Leaflet documentation on the polyline() function (https://leafletjs.com/reference-1.5.0.html#polyline)** .

13.5.1: Overview of GeoJSON Data

Sadhana and Basil are impressed with your mapping skills. Now that you know how to map points and lines from an array, Basil wants you to become familiar with how to map points, LineStrings, and polygons from GeoJSON data.

> **Rewind**
> GeoJSON data is a type of JavaScript Object Notation (JSON) data that is specifically designed to host geographical information.

GeoJSON data consists of a single object, which can be represented by a geometry object, features object, or collection of features (FeatureCollection object). Let's review these GeoJSON objects in more detail.

Geometry Object

A GeoJSON geometry object is where the type member's value is one of the following strings: Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, or GeometryCollection.

Geometry object types include the following:

- Point has a single set of coordinates, like when we mapped a single marker to the map.

```
"geometry":{
  "type":"Point",
  "coordinates": [-118.4, 33.9]
  }
```

- MultiPoint is an array of point coordinates, like when we mapped multiple cities with their population to a map.

```
"geometry":{
  "type":"MultiPoint",
  "coordinates": [-118.4, 33.9], [-118.5, 34.0]
  }
```

- LineString has an array of Point coordinates, like when we mapped the airline route from LAX to SFO.

```
"geometry":{
  "type":"LineString",
  "coordinates": [[-118.4, 33.9],[-122.4, 37.6]]
  }
```

- MultiLineString are an array of LineString coordinates, like when we mapped the LAX-SFO-SLC-SEA airline route.

```
"geometry":{
  "type":"MultiLineString",
  "coordinates":
     [[-118.4, 33.9],[-106.4, 31.8]],
     [[-118.4, 33.9],[-123.2, 44.1]]
  }
```

- Polygon has an array of LineString coordinates. We'll map polygons later in this module.

```
"geometry": {
   "type": "Polygon",
    "coordinates":
   [
    [ [ -122.446, 37.861 ], [ -122.438, 37.868 ], [ -122.430, 37.872
   ]
```

- MultiPolygon has an array of polygon coordinates. We'll map multiple polygons later in this module.

```
"geometry": {
   "type": "MultiPolygon",
   "coordinates": [
    [ [ -122.446, 37.861 ], [ -122.438, 37.868 ], [ -122.430, 37.872
    [ [ -122.378, 37.826 ], [ -122.377, 37.830 ], [ -122.369, 37.832
]
```

- GeometryCollection has a geometry array. Each element in the geometry array can be one of the geometry objects above.

```
{
   "type": "GeometryCollection",
   "geometries": [
       {
         "type":"Point",
       "coordinates": [-118.4, 33.9]
      }
       },
       {
          "type":"LineString",
          "coordinates": [
             [-118.4, 33.9],[-122.4, 37.6]
           ]
       }
   ]
}
```

Features Object

GeoJSON features object contains a geometry object, like those above, and additional properties, such as magnitude, address, and time as shown below for an earthquake features object.

```
{
    type: "Feature",
    properties: {
    mag: 1.88,
    place: "6km SE of Pahala, Hawaii",
    time: 1573766377230,
    type: "earthquake",
    title: "M 1.9 - 6km SE of Pahala, Hawaii"
    },
    geometry: {
    type: "Point",
    coordinates: [
    -155.4329987,
    19.1634998
    ]},
  }
```

FeatureCollection Object

A FeatureCollection object contains an array of features objects. In the following FeatureCollection of airline routes, there is an array of features objects where each feature has its own properties and geometry:

```
{"type":"FeatureCollection","features":
[
  {
    "type":"Feature","properties":{
"airline":"AA","airline_id":"24","src":"LAX","dst":"ABQ","dst_id":"4019",
```

```
"type":"LineString",
"coordinates":[[-118.4079971,33.94250107],[-106.609001,35.040199]]}
},
{
"type":"Feature","properties":{
"airline":"AA","airline_id":"24","src":"LAX","src_id":"3484","dst":"ANC",
"type":"LineString",
"coordinates":[[-118.4079971,33.94250107],[-149.99600219726562,61.1744003
  },
{
  "type":"Feature","properties":{
"airline":"AA","airline_id":"24","src":"LAX","src_id":"3484","dst":"AUS",
"type":"LineString",
"coordinates":[[-118.4079971,33.94250107],[-97.6698989868164,30.194499969
  }
]

}
```

Now that you are familiar with the different GeoJSON objects, practice using these to map data to our map.

> **Note**
>
> For more information on GeoJSON format, refer to the **GeoJSON Standard (https://tools.ietf.org/html/rfc7946)** .

13.5.2: Map GeoJSON Point Type

You meet with Basil and Sadhana to discuss your project. Basil informs you that the earthquake data you'll map will have the geometry type Point. Basil thinks it would be a good idea to learn to parse GeoJSON data that is similar to the earthquake data.

Sadhana wants you to practice mapping GeoJSON data that she will give you to add to your `logic.js` file. This will be a good introduction on learning how to access the data from a JSON file.

Before we map any data, let's create a new branch called "Mapping_GeoJSON_Points" and create the following folder structure:

- Mapping_GeoJSON_Points
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

Copy the necessary folders and files from one of your previous branches and add them to the Mapping_GeoJSON_Points folder.

Map a GeoJSON Point

First, we'll add single point on our map using GeoJSON data. The following GeoJSON data is a FeatureCollection object that has properties and geometry for the San Francisco Airport:

```
// Add GeoJSON data.
let sanFranAirport =
{"type":"FeatureCollection","features":[{
    "type":"Feature",
    "properties":{
        "id":"3469",
        "name":"San Francisco International Airport",
        "city":"San Francisco",
        "country":"United States",
        "faa":"SFO",
        "icao":"KSFO",
        "alt":"13",
        "tz-offset":"-8",
        "dst":"A",
        "tz":"America/Los_Angeles"},
        "geometry":{
            "type":"Point",
            "coordinates":[-122.375,37.61899948120117]}}
]};
```

Since we are going to add the San Francisco Airport to our map, let's change the center to the San Francisco Airport. Add the following code to our `logic.js` file to create the center of the map at the airport with a zoom level of "10."

```
// Create the map object with center at the San Francisco airport.
let map = L.map('mapid').setView([37.5, -122.5], 10);
```

In the **GeoJSON example (https://leafletjs.com/examples/geojson/)** given on the Leaflet page, we can see that the simple GeoJSON feature is similar to our

`sanFranAirport`.

```
var geojsonFeature = {
    "type": "Feature",
    "properties": {
        "name": "Coors Field",
        "amenity": "Baseball Stadium",
        "popupContent": "This is where the Rockies play!"
    },
    "geometry": {
        "type": "Point",
        "coordinates": [-104.99404, 39.75621]
    }
};
```

GeoJSON objects are added to the map through a GeoJSON layer, `L.geoJSON()`. In “The GeoJSON Layer” section, it says to create the GeoJSON layer and add it to our map. We can use the following code to do that:

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

Let’s edit this GeoJSON layer as follows:

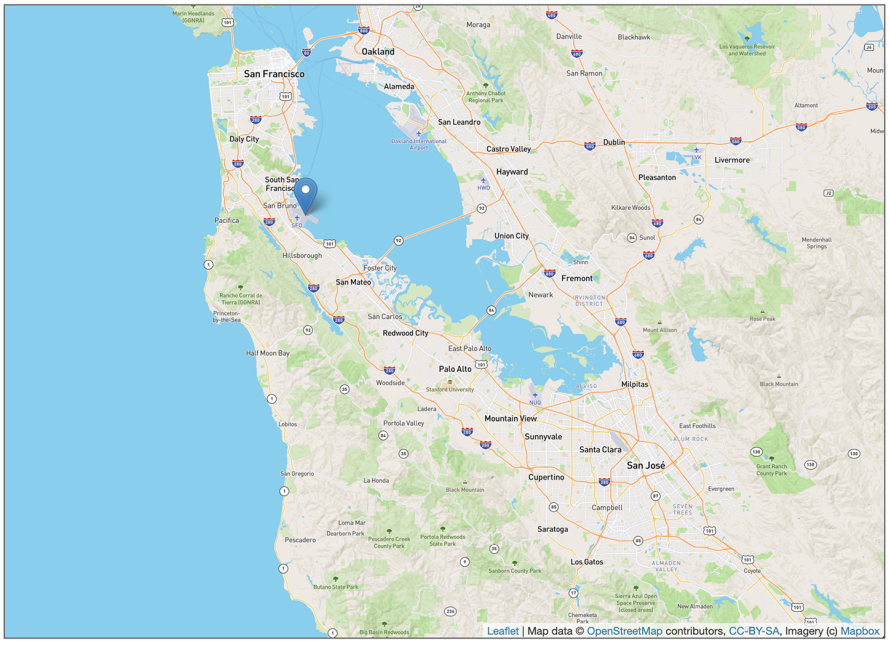
```
// Grabbing our GeoJSON data.
L.geoJSON(sanFranAirport).addTo(map);
```

Also, add it to our `logic.js` file below the GeoJSON airport data and above the `tileLayer()` method. After you save the `logic.js` file, it should look like the following:

```
// Create the map object with center at the San Francisco airport.
let map = L.map('mapid').setView([37.5, -122.5], 10);

// Add GeoJSON data.
let sanFranAirport =
{"type":"FeatureCollection","features":[{
    "type":"Feature",
    "properties":{
        "id":"3469",
        "name":"San Francisco International Airport",
        "city":"San Francisco",
        "country":"United States",
        "faa":"SFO",
        "icao":"KSFO",
        "alt":"13",
        "tz-offset":"-8",
        "dst":"A",
        "tz":"America/Los_Angeles"},
    "geometry":{
        "type":"Point",
        "coordinates":[-122.375,37.61899948120117]}}
]};

// Grabbing our GeoJSON data.
L.geoJSON(sanFranAirport).addTo(map);
```

> ## Note
>
> Please note that the coordinates appear in reverse order `[-122.375, 37.61899948120117]`, compared to their order in the `setView()` method. This is because the GeoJSON data coordinates are set with the first parameter as X (longitude) and the second parameter as Y (latitude), as documented in the **GeoJSON Standard.** **(https://tools.ietf.org/html/rfc7946)** The `L.geoJSON()` layer reverses the coordinates to plot them on the map.

Open the `index.html` file in your browser. Your map should have a marker at SFO.



Later in this module we'll be using a URL to access a larger GeoJSON dataset to plot more points.

Bind a Popup to the Marker

> **Rewind**
> To display data on a map with a popup marker, we have to bind the marker with the GeoJSON layer, `L.geoJSON()`, using a callback function.

Our options to add data to a marker are to use the `pointToLayer` or `onEachFeature` callback functions. With either of these functions, we can add data to a map from each GeoJSON object. The major difference between the two functions is that the `pointToLayer` callback function adds markers to a map, whereas the `onEachFeature` callback function allows you to add styling and bind data to a popup marker.

Let's look at these two functions more closely.

The pointToLayer Function

For the `pointToLayer` callback function, the basic syntax for adding functionality to a marker follows:

```
L.geoJson(data, {
    pointToLayer: function(feature, latlng) {
      return L.marker(latlng);
     }
});
```

Let's break down what is happening in the `L.geoJSON()` layer:

1. We add two arguments: the `data` and the `pointToLayer`c allback function.
2. The `data` will be our `sanFranAirport` data.

3. For the `pointToLayer` callback function, we are first going to call a `function()` where we pass each GeoJSON feature as `feature`, and its latitude and longitude as `latlng`.
4. Then we add a marker for each feature with a latitude and longitude in the `pointToLayer` callback function argument by using `return L.marker(latlng)`.

If you want to create a circle marker instead of a marker on the map, what method would you use?

- Use `circleMarker()`.
- Use `circlemarker()`.
- Use `circle()`.

Check Answer

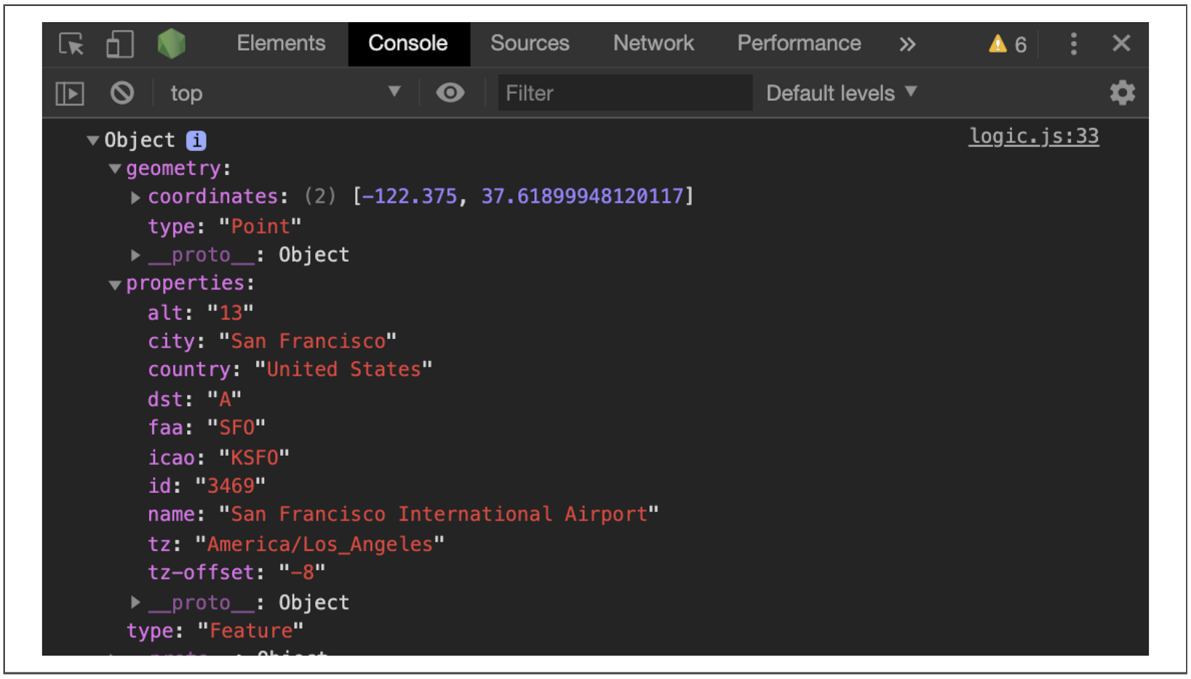
Finish ▸

Even though we have a marker on the previous map, let's edit our `logic.js` file to add a marker using the `pointToLayer` function and add data to a popup marker.

First, let's edit the `logic.js` file to add the `pointToLayer` callback function to the `L.geoJSON()` layer. To better understand what is passed with the `feature` argument in the `function()`, we will add `feature` in the `console.log()` function. Edit your `L.geoJSON()` layer code to look like the following:

```
// Grabbing our GeoJSON data.
L.geoJson(sanFranAirport, {
    // We turn each feature into a marker on the map.
    pointToLayer: function(feature, latlng) {
      console.log(feature);
      return L.marker(latlng);
    }

  }).addTo(map);
```

Save your `logic.js` file and open the `index.html` file in your browser. The map should look the same as it did before the edits. However, if we open the console on our developer tools, we will see that the `feature` is the JavaScript object `geometry` and `properties` of our GeoJSON object.



Now, we'll add the data in the JavaScript objects to a popup marker.

> **Rewind**
> The properties in each JavaScript object can be accessed using the dot notation.

What Leaflet method is used to make a popup marker?

- `bindPopUp()`
- `bindPopup()`
- `popUp()`

Check Answer

Finish ▸

To add a popup marker, we need to use the `bindPopup()` method to the `pointToLayer` callback function. This will add a popup marker for each object in our GeoJSON data even though we only have one object in our data, SFO.

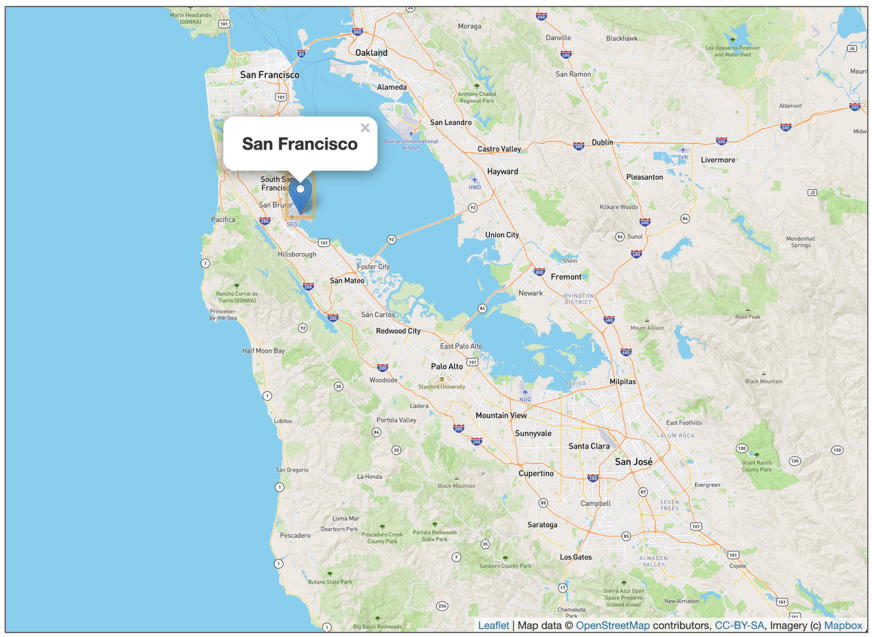
Let's add the city to the popup marker. In our `logic.js` file, after the `return L.marker(latlng)` in our `L.geoJSON()` layer, add the following code on the next line:

```
.bindPopup("<h2>" + feature.properties.city + "</h2>")
```

Using the dot notation, we can traverse through the JSON object to get the city by using `feature.properties.city`. Now, your `logic.js` file with `L.geoJSON()` layer should look like the following:

```
// Grabbing our GeoJSON data.
L.geoJson(sanFranAirport, {
    // We turn each feature into a Marker on the map.
    pointToLayer: function(feature, latlng) {
      console.log(feature);
      return L.marker(latlng)
      .bindPopup("<h2>" + feature.properties.city + "</h2>");
    }

  }).addTo(map);
```

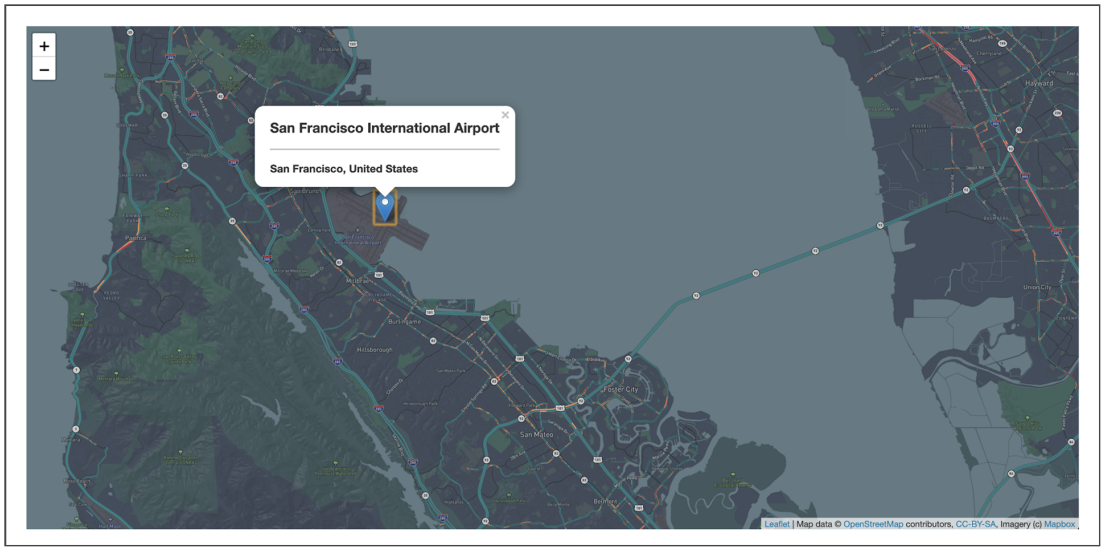
Our map should look like the following, where a marker, when clicked, shows a city name:



Skill Drill

Edit your `logic.js` to create a popup marker for San Francisco Airport on a night preview navigation map. When you click on the popup, it will display the city, state, and the name of the airport.

Your map should look like the following:



The onEachFeature Function

When we use the `onEachFeature` callback function we can add a popup marker for each feature and add data from the properties of the JavaScript object. The basic syntax for adding functionality to a marker follows:

```
L.geoJson(data, {
    onEachFeature: function(feature, layer) {
      layer.bindPopup();
     }
});
```

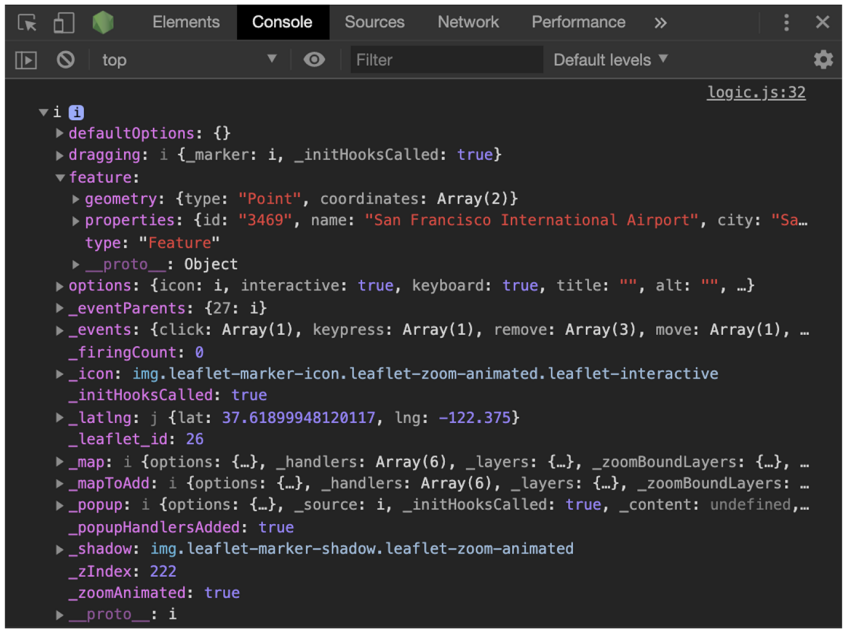
Let's break down what is happening in the `L.geoJSON()` layer:

1. First, we add two arguments: the `data` and the `onEachFeature` callback function.
2. The `data` will be our `sanFranAirport` data.
3. With the `onEachFeature` callback function we are first going to call an anonymous function, `function()`, where we pass each GeoJSON feature as `feature`, and any properties to the second argument, `layer`.

Let's edit our `logic.js` file to add a popup marker using the `onEachFeature` function. First, edit the `logic.js` file to add the `onEachFeature` callback function to the `L.geoJSON()` layer. To see what is passed with the `layer` argument in the anonymous `function()`, we'll pass `layer` in the `console.log()` function. Edit your `L.geoJSON()` layer code to look like the following:

```
// Grabbing our GeoJSON data.
L.geoJson(sanFranAirport, {
  onEachFeature: function(feature, layer) {
    console.log(layer);
    layer.bindPopup();
  }
}).addTo(map);
```

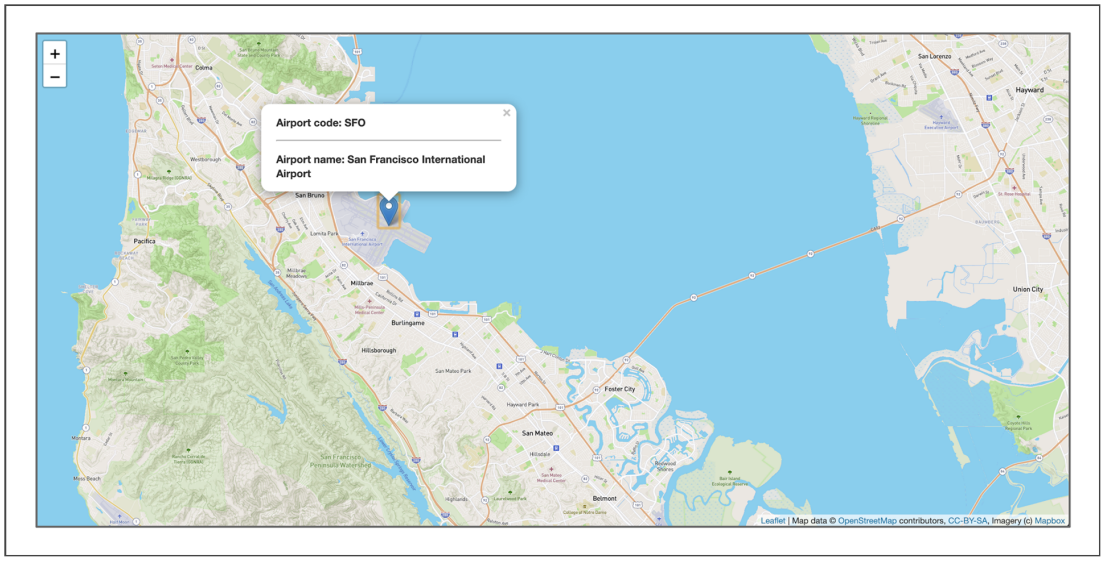
When we open our `index.html` file, the map will display a popup marker for SFO. When we open the console on our DevTools, we'll see that the `layer` returns many JavaScript methods that can be accessed and used, including the `geometry` and `properties` of our GeoJSON object.



Skill Drill

Edit your `logic.js` to create a popup marker for the San Francisco Airport on the outdoor map. When you click on the popup, it will display the airport code and name of the airport.

Your map should look like the following:



Great job on adding GeoJSON data to your map!

> **Note**
>
> For more information, see the **Leaflet documentation on the L.geoJSON() layer. (https://leafletjs.com/reference-1.6.0.html#geojson)** .

Next, we'll map multiple point type geometry from a JSON file.

13.5.3: Map Multiple GeoJSON Points

Now that you have a handle on how to map GeoJSON point type and add data to a popup marker, Basil and Sadhana want you to fetch GeoJSON data from a URL. After all, this is how GeoJSON data is usually accessed, and this is how you will access the earthquake data.

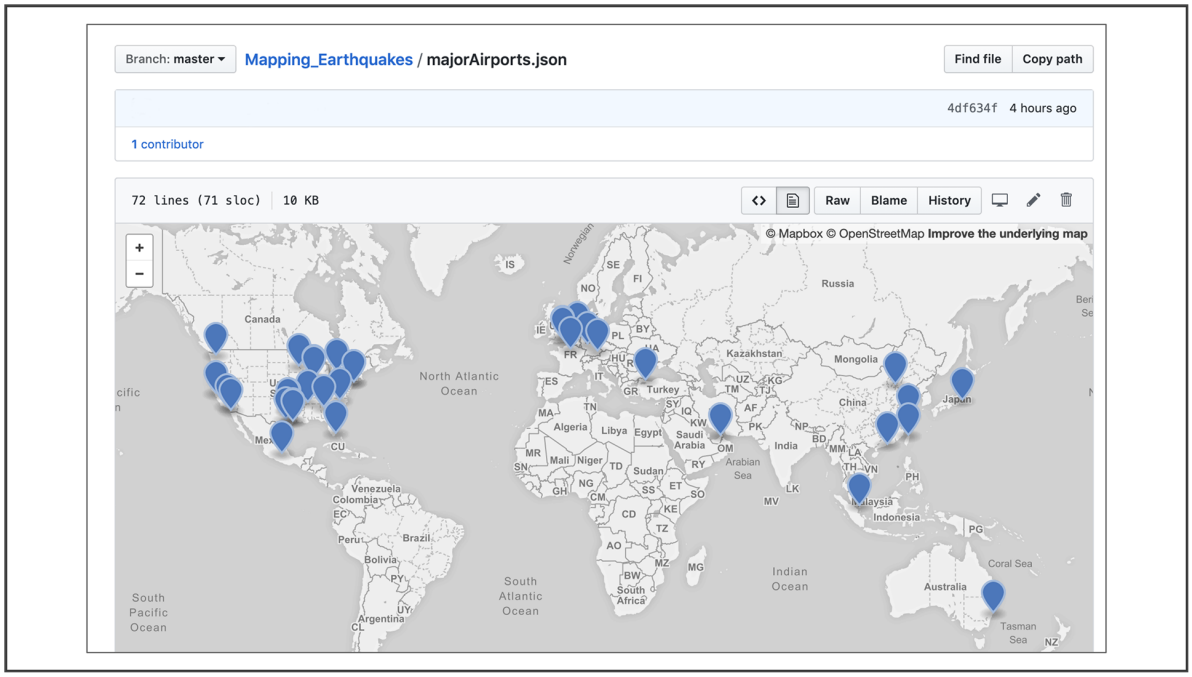
When mapping points, lines, and polygons, the data we use is accessed from a URL because this data is usually inaccessible for download or maybe too large to store on your computer and add as an external file.

Download the `majorAirports.json` file and put it on the Mapping_Earthquakes repository.

Download majorAirports.json (https://courses.bootcampspot.com/courses/138/files/13670/download?wrap=1)

Using the URL for the `majorAirports.json` file in your GitHub repository, we'll add multiple points onto a map.

When you click on the `majorAirports.json` file on GitHub, you should see an OpenStreetMap populated with major airports. Our map will look similar to this after we are done.



Click the Raw button and the GeoJSON data will be loaded in the browser.

If the file size is large, it could take awhile to load on the page. Once it loads, it should look like the following:


```
{
   type: "FeatureCollection",
 - features: [
    - {
        type: "Feature",
      - properties: {
            id: "3550",
            name: "George Bush Intercontinental Houston Airport",
            city: "Houston",
            country: "United States",
            faa: "IAH",
            icao: "KIAH",
            alt: "97",
            tz-offset: "-6",
            dst: "A",
            tz: "America/Chicago"
        },
      - geometry: {
            type: "Point",
          - coordinates: [
                -95.34140014648438,
                29.984399795532227
            ]
        }
    },
    - {
        type: "Feature",
      - properties: {
            id: "3566",
            name: "William P Hobby Airport",
            city: "Houston",
            country: "United States",
            faa: "HOU",
            icao: "KHOU",
            alt: "46",
            tz-offset: "-6",
            dst: "A",
            tz: "America/Chicago"
        },
      - geometry: {
            type: "Point",
          - coordinates: [
                -95.27890015,
                29.64539909
            ]
        }
    },
    - {
```

To begin adding the data to the map, first we need to read the external `majorAirports.json` file.

> **Rewind**
>
> To read an external `.json` file, we need to use the `d3.json()` method. To use the `d3.json()` method, we need to have the `<script src="https://d3js.org/d3.v5.min.js"></script>` file in the `index.html` page.

Open the `index.html` file, and in the `<head>` section above the CSS link, add the following D3.js library file script:

```
<!-- d3 JavaScript -->
<script src="https://d3js.org/d3.v5.min.js"></script>
```

The `<head>` section of your `index.html` file should look like the following:

Next, we'll edit the `logic.js` file.

1. Change the geographical center of the map to the geographical center of the Earth and set the zoom level as follows:

   ```
   // Create the map object with center and zoom level.
   let map = L.map('mapid').setView([30, 30], 2);
   ```

2. Next, we'll access the `majorAirports.json` file on GitHub with the following `airportData` variable. Your URL may be different, but it should begin with `https://raw.githubusercontent.com`.
3. Add the following code after your `tileLayer()` method:

```
// Accessing the airport GeoJSON URL
let airportData = "https://raw.githubusercontent.com/<GitHub_name>/Ma
```

> **Note**
>
> Having the `tileLayer()` method before accessing large datasets ensures that the map gets loaded before the data is added to it.

4. Next, we'll add the `d3.json()` method, which returns a promise with the `then()` method and the anonymous `function()`.
   - Inside the `d3.json()` method we'll add the `airportData` variable.
   - Inside the anonymous `function()` we'll add the `data` parameter, which references the `airportData`.
   - We'll pass this `data` to the `L.geoJSON()` layer and then it'll be added to the map with `addTo(map)`.

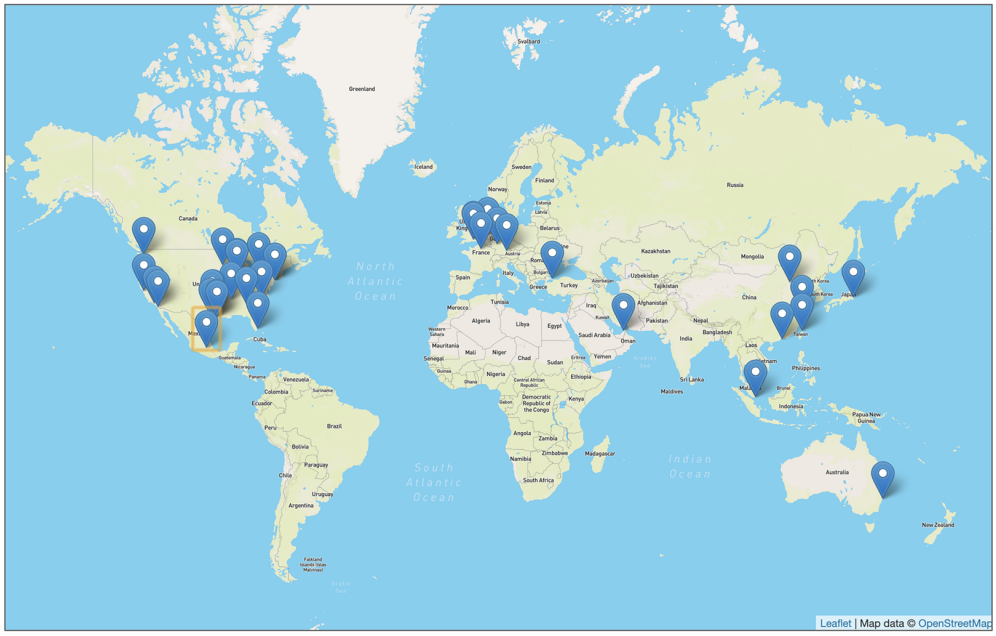
```
// Grabbing our GeoJSON data.
d3.json(airportData).then(function(data) {
    console.log(data);
  // Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data).addTo(map);
});
```

Your `logic.js` file should look like the following:

```
// Create the map object with center and zoom level.
let map = L.map('mapid').setView([30, 30], 2);

// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecom
    maxZoom: 18,
    accessToken: API_KEY
});
// Then we add our 'streets' tile layer to the map.
streets.addTo(map);

// Accessing the airport GeoJSON URL
let airportData = "https://raw.githubusercontent.com/         /Mapping_Earthquakes/master/majorAirports.json";

// Grabbing our GeoJSON data.
d3.json(airportData).then(function(data) {
    console.log(data);
    // Creating a GeoJSON layer with the retrieved data.
    L.geoJson(data).addTo(map);
});
```

Let's see how our map looks now. Open your `index.html` file in your browser using the command `python -m http.server`—just to be sure that the data is accessible through the Python server.

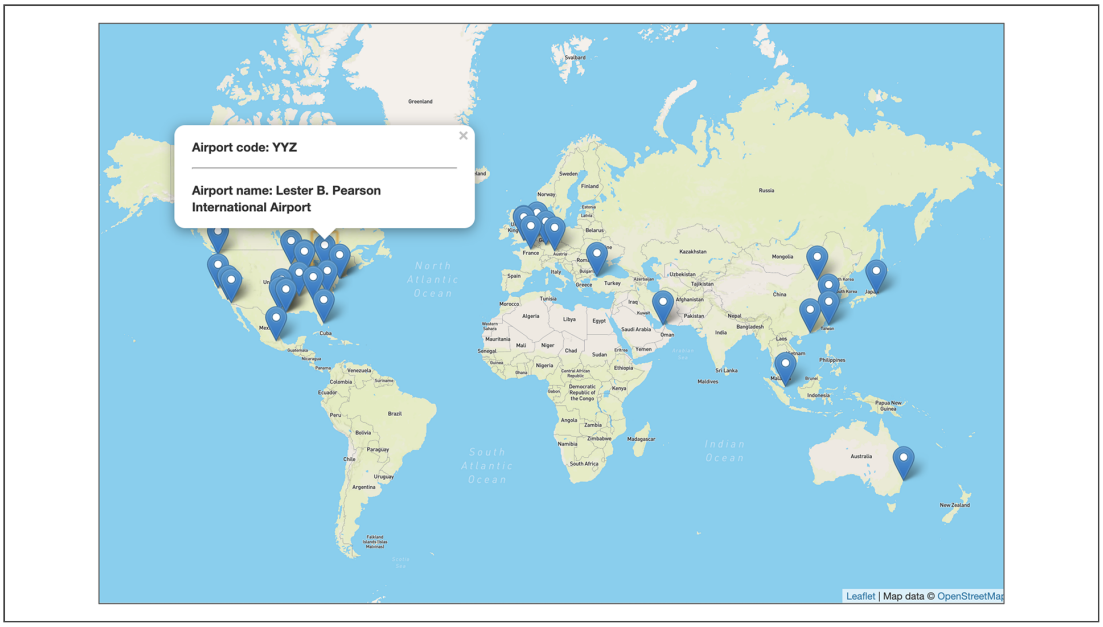
Your map should look like the following:



Skill Drill

Edit your `L.geoJson()` layer to add a popup marker that displays all airports' codes and names.

Your map should look like the following:



Great job on adding multiple-point type GeoJSON data to your map. Next, Sadhana is going to show you how to add another map to the `index.html` file so you can toggle between two different maps.

13.5.4: Add Multiple Maps

Before Sadhana shows you how to map GeoJSON LineString data, she's going to walk you through adding another map so that you can toggle between maps to visualize the data. One of the features you'll be working on for the earthquake data is different map styles, which will allow individuals to choose how they want to visualize the data.

Adding another map to showcase the data is a nice feature for a map. Most mobile apps with a mapping service use two different styles, or layers: a satellite layer and a dark layer. This makes the app visually appealing and brings functionality to the map.

To create two map choices, we'll edit the `logic.js` file for mapping the major airports without the popup markers. We'll move some code to make it more readable, and we'll add more code to the `logic.js` file.

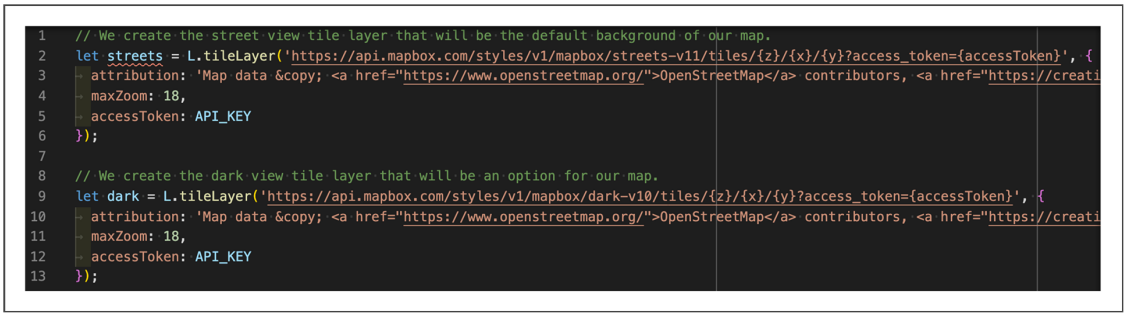
To add another map, we'll use the Leaflet Layers Control (see the **documentation (https://leafletjs.com/examples/layers-control/)**). The Layers Control allows us to control which layers, or styles, we'll see on our map. For this task, we'll work with the streets and dark layers.

1. First, we'll add another `tileLayer()` to create a dark map. If you don't have the `tileLayer()` code for the dark map, add the following code block below the code for the streets map.

```
// We create the dark view tile layer that will be an option for our
let dark = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/dark-
```

```
attribution: 'Map data © <a href="https://www.openstreetmap.org/">Ope
    maxZoom: 18,
    accessToken: API_KEY
});
```

2. Don't add the `addTo(map)` at the end of your streets or dark `tileLayer()` code. We'll add it later.
3. At this point, your `logic.js` file should look like the following:

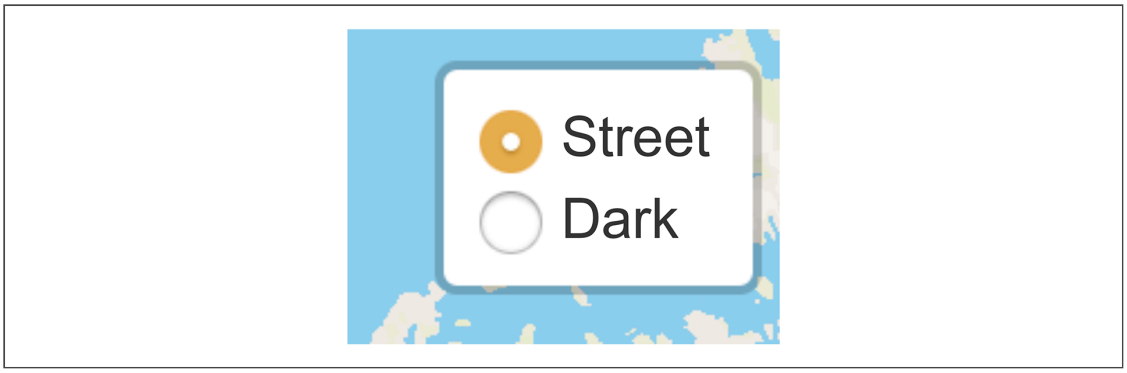


```
// We create the street view tile layer that will be the default background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creati
    maxZoom: 18,
    accessToken: API_KEY
});

// We create the dark view tile layer that will be an option for our map.
let dark = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/dark-v10/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creati
    maxZoom: 18,
    accessToken: API_KEY
});
```

4. Next, we'll add both map variables to a new variable, `baseMaps`. This variable will be used as our base layer, which we'll reference later.
5. After the code for the dark map, add the following variable to reference the base layer:

```
// Create a base layer that holds both maps.
let baseMaps = {
  Street: streets,
  Dark: dark
};
```

6. In the base layer code, the Street and Dark keys set the text, which we'll see in the `index.html` file, while the corresponding values reference the tile layers. Street and Dark can be used to toggle between styles in the `index.html` file and will look like the following:



7. Modify the `map` object to change the center and zoom level, and add the base layer with the default map. For the `map` object, we won't use the `setView()` method; instead, we'll apply the alternative method that we used earlier in this module.

> **Rewind**
>
> An alternative to using the `setView()` method is to modify each attribute in the map object using the curly braces notation as follows:
>
> ```
> // Create the map object with a center and zoom level.
> let map = L.map("mapid", {
>   center: [40.7, -94.5],
>   zoom: 4
> });
> ```

8. Add the following code after the base layer code:

```
// Create the map object with center, zoom level and default layer.
let map = L.map('mapid', {
    center: [30, 30],
    zoom: 2,
    layers: [streets]
})
```

9. To complete the code for the map layers, use the Leaflet `control.layers`, which will control the layers we'll see on the map. Add the following code

below the `map` object:

```
// Pass our map layers into our layers control and add the layers cor
L.control.layers(baseMaps).addTo(map);
```

10. When creating the Layers Control, the argument passed, `baseMaps`, is the base layer object, which will allow the two different map styles to be shown on the `index.html` file. The Layers Control will look like the following before it is clicked to show the Street and Dark options:

11. Your `logic.js` file should look like the following:

```
// We create the street view tile layer that will be the default background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creat
    maxZoom: 18,
    accessToken: API_KEY
});

// We create the dark view tile layer that will be an option for our map.
let dark = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/dark-v10/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creat
    maxZoom: 18,
    accessToken: API_KEY
});

// Create a base layer that holds both maps.
let baseMaps = {
  Street: streets,
  Dark: dark
};

// Create the map object with center, zoom level and default layer.
let map = L.map('mapid', {
    center: [30, 30],
    zoom: 2,
    layers: [streets]
});

// Pass our map layers into our layer control and add the layer control to the map.
L.control.layers(baseMaps).addTo(map);
```

12. Finally, add the `airportData` variable to the `d3.json()` method as shown below:

```
// Accessing the airport GeoJSON URL
let airportData = "https://raw.githubusercontent.com/          /Mapping_Earthquakes/master/majorAirports.json";

// Grabbing our GeoJSON data.
d3.json(airportData).then(function(data) {
    console.log(data);
  // Creating a GeoJSON layer with the retrieved data.
    L.geoJson(data).addTo(map);
});
```

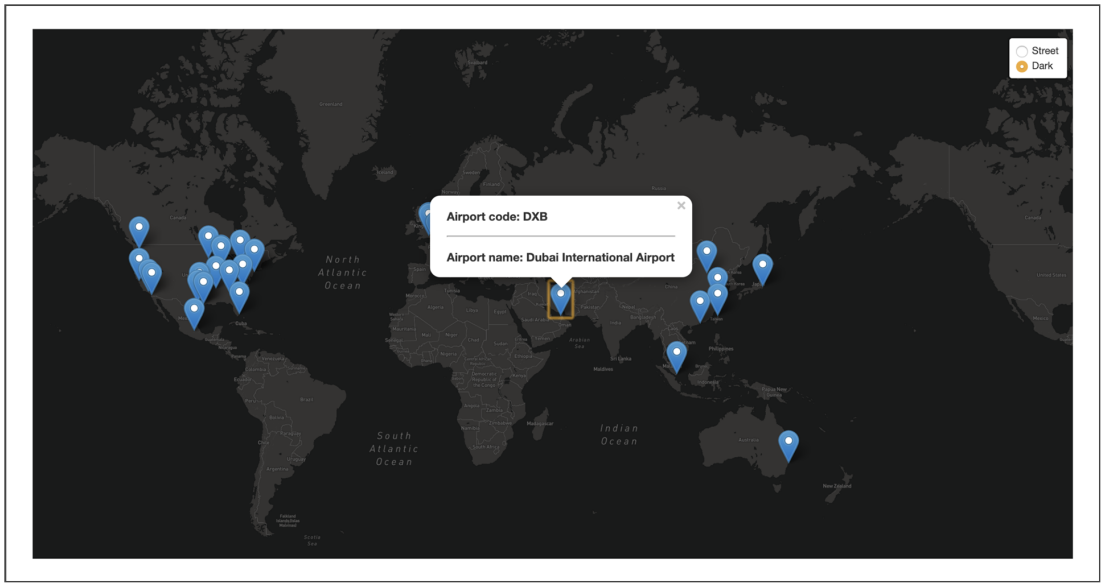
Save the `logic.js` file and open `index.html` in your browser. Your map should look like the following with a Layers Control in the top right corner of the map:

When we hover over the Layers Control, the map options are shown.

Skill Drill

Edit your `L.geoJson()` layer to add a popup marker that displays all airports' codes and names for both the Street and Dark layers.

Your map should look like the following:



Great job adding multiple-point type GeoJSON data to your map! Next, Sadhana will show you how to add map LineString type GeoJSON data.

Add, Commit, Push

Add, commit, and push your changes to your Mapping_GeoJSON_Points branch. Don't delete the branch so that others can use it to learn how to map GeoJSON points.

13.5.5: Map GeoJSON LineStrings

Since you did so well on fetching and mapping the GeoJSON point type objects, Basil would like you to map GeoJSON LineString type objects.

Sadhana likes the work you have been pushing to the branches on the work repository. These are becoming a big hit with the other members of your team. So, Sadhana would like you to repeat the process for mapping GeoJSON LineStrings.

Create a new branch called “Mapping_GeoJSON_Linestrings,” with the following folder structure. Copy the folders and files from your Mapping_GeoJSON_Points branch and add them to the Mapping_GeoJSON_Linestrings folder:

- Mapping_GeoJSON_Linestrings
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

Next, download the `torontoRoutes.json` file and put it on the Mapping_Earthquakes repository.

Download torontoRoutes.json
(https://courses.bootcampspot.com/courses/138/files/13656/download?wrap=1)

Let's take a look at the `torontoRoutes.json` file, which contains a majority of the nonstop airline routes from Toronto, Canada.

Looking at the first object in the `torontoRoutes.json` file, we can see that there is a properties and geometry object. In the geometry object, we can see the type is `LineString` and the `coordinates` are an array of point coordinates, like when we mapped the airline route from LAX to SEA earlier in this module.

Now, map the nonstop routes from Toronto on two map styles: light and dark. First, we'll need to create both map styles.

1. For the dark map: Use the code from the Mapping_GeoJSON_Points `logic.js` file to get the API for the dark map.

For the following Mapox API URL, what is the code for the light map?

- `https://api.mapbox.com/styles/v1/mapbox/lightv11/tiles/{z}/{x}/{y}?access_`
- `https://api.mapbox.com/styles/v1/mapbox/light-v10/tiles/{z}/{x}/{y}?acces`
- `https://api.mapbox.com/styles/v1/mapbox/light-v11/tiles/{z}/{x}/{y}?acces`

Check Answer

Finish ▸

2. For the light map:
   - In the streets variable in the `logic.js` file (from the Mapping_GeoJSON_Points), replace `streets-v11` with `light-v10` for the `streets` map.
   - Change the variable for the `streets` tile layer to `light`.

Where else would you have to edit your `logic.js` file to ensure the light map is shown? Select all that apply.

- [ ] The key-value pair in the `baseMaps` base layer
- [ ] The Layers Control
- [ ] The default layer in the `map` object.
- [ ] The variable for the `streets` tile layer

Check Answer

Finish ▸

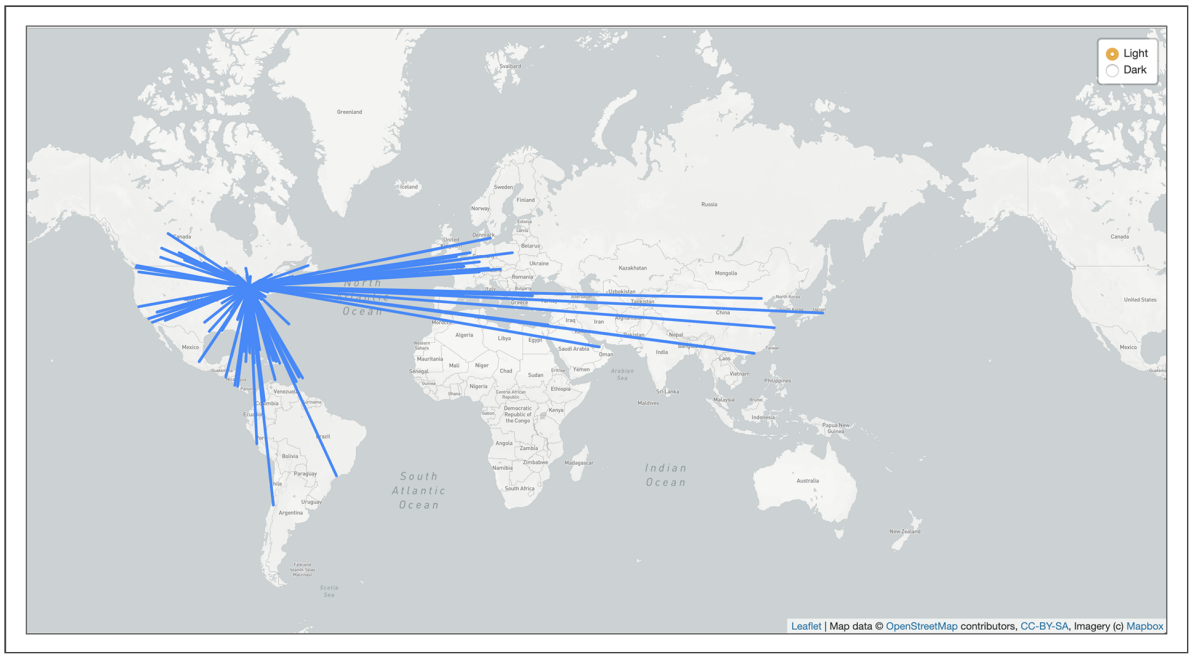
3. Change the center of the map to Toronto with a zoom level of 2. Toronto's coordinates are listed first in the array: `[44.0, -80.0]`.
4. Create a `torontoData` variable in the `logic.js` file and assign it to the URL for the `torontoRoutes.json` file on GitHub.

```
// Accessing the Toronto airline routes GeoJSON URL.
let torontoData = "https://raw.githubusercontent.com/<GitHub_name>/Ma
```

5. Edit the `d3.json()` method to look like the following:

```
// Grabbing our GeoJSON data.
d3.json(torontoData).then(function(data) {
    console.log(data);
  // Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data).addTo(map);
});
```

When you open `index.html` in your browser, your map should look like the following:

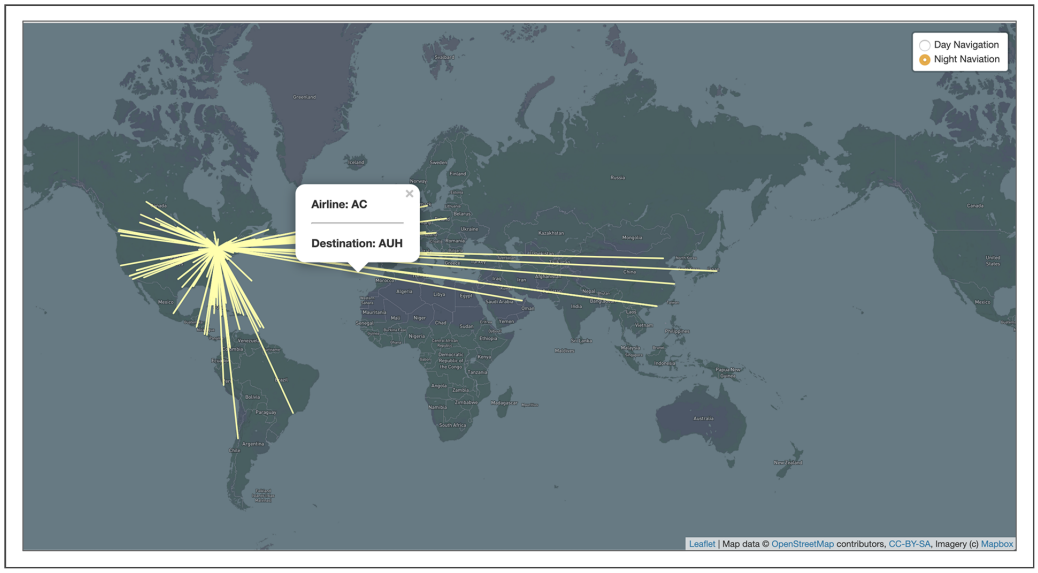


Great job on mapping the nonstop routes from Toronto!

> ### Skill Drill
>
> Edit the `L.geoJson()` layer so that it displays the following:
>
> 1. The default map layer as night navigation with day navigation as another option.

2. The airline routes are in light yellow with a `weight` of 2.
3. Each airline route has a popup marker that shows the airline code and destination.

   Your map should look like the following:



The code in your `logic.js` file specifies a dark map showing nonstop flight routes, styled in light yellow with a `weight` of 2. Your file should look like the following:

```
// Grabbing our GeoJSON data.
d3.json(torontoData).then(function(data) {
  console.log(data);
// Creating a GeoJSON layer with the retrieved data.
L.geoJson(data, {
  color: "#ffffa1",
  weight: 2,
  onEachFeature: function(feature, layer) {
    layer.bindPopup("<h3> Airline: " + feature.properties.airline + "</h3> <hr><h3> Destination: "
    + feature.properties.dst + "</h3>");
  }
})
.addTo(map);
});
```

To make this code easier to read, now we'll create an object with the style parameters for the lines and assign it to a variable, `myStyle`. Add the following code before `d3.json()`.

```
// Create a style for the lines.
let myStyle = {
    color: "#ffffa1",
    weight: 2
}
```

Then, in the `L.geoJson()` layer, the `style` key will be assigned to the `myStyle` object as shown:

```
// Grabbing our GeoJSON data.
d3.json(torontoData).then(function(data) {
  console.log(data);
// Creating a GeoJSON layer with the retrieved data.
L.geoJson(data, {
  style: myStyle,
  onEachFeature: function(feature, layer) {
    layer.bindPopup("<h3> Airline: " + feature.properties.airline + "</h3> <hr><h3> Destination: "
    + feature.properties.dst + "</h3>");
  }
})
.addTo(map);
});
```

Next, Sadhana will show you how to map polygon type geometry from a JSON file.

> **Add, Commit, Push**
>
> Add, commit, and push your changes to your Mapping_GeoJSON_LineStrings branch. Don't delete the branch so that others can use it to learn how to map GeoJSON LineStrings.

13.5.6: Map GeoJSON Polygons

The last example that Sadhana will walk you through is mapping GeoJSON polygon type objects. Polygons are more complex than Points or LineStrings. Sadhana is going to review the structure of a polygon's type object and then walk you through mapping these objects.

First, Sadhana wants you to create a branch on the master repository for mapping polygons. Create a branch called “Mapping_GeoJSON_Polygons” with the following folder structure. Copy the folders and files from your Mapping_GeoJSON_Linestrings branch and add them to the Mapping_GeoJSON_Polygons folder.

- Mapping_GeoJSON_Polygons
  - `index.html`
  - static
    - css
      - `style.css`
    - js
      - `config.js`
      - `logic.js`

Next, download the `torontoNeighborhoods.json` file and put it in the Mapping_Earthquakes repository.

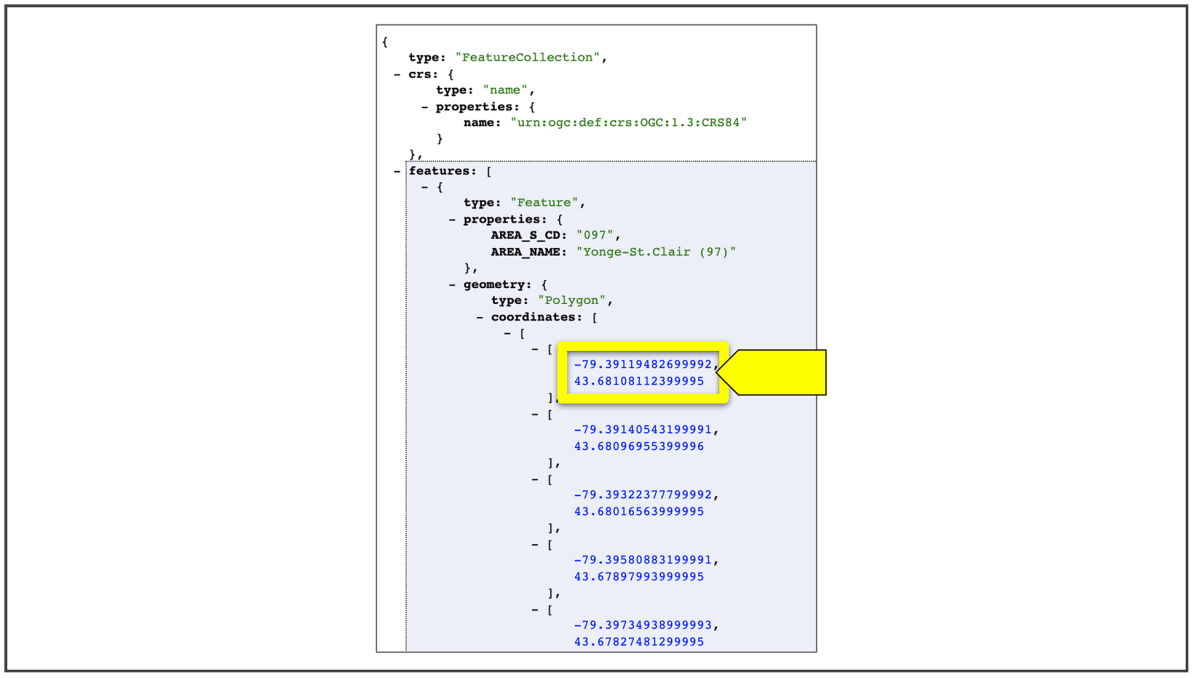
Download torontoNeighborhoods.json
(https://courses.bootcampspot.com/courses/138/files/13664/download?wrap=1)

Let's take a look at the `torontoNeighborhoods.json` file, which contains the coordinates for each neighborhood in Toronto, Canada.

> **Note**
> Depending on your computer's processor, the `torontoNeighborhoods.json` data may take awhile to load.

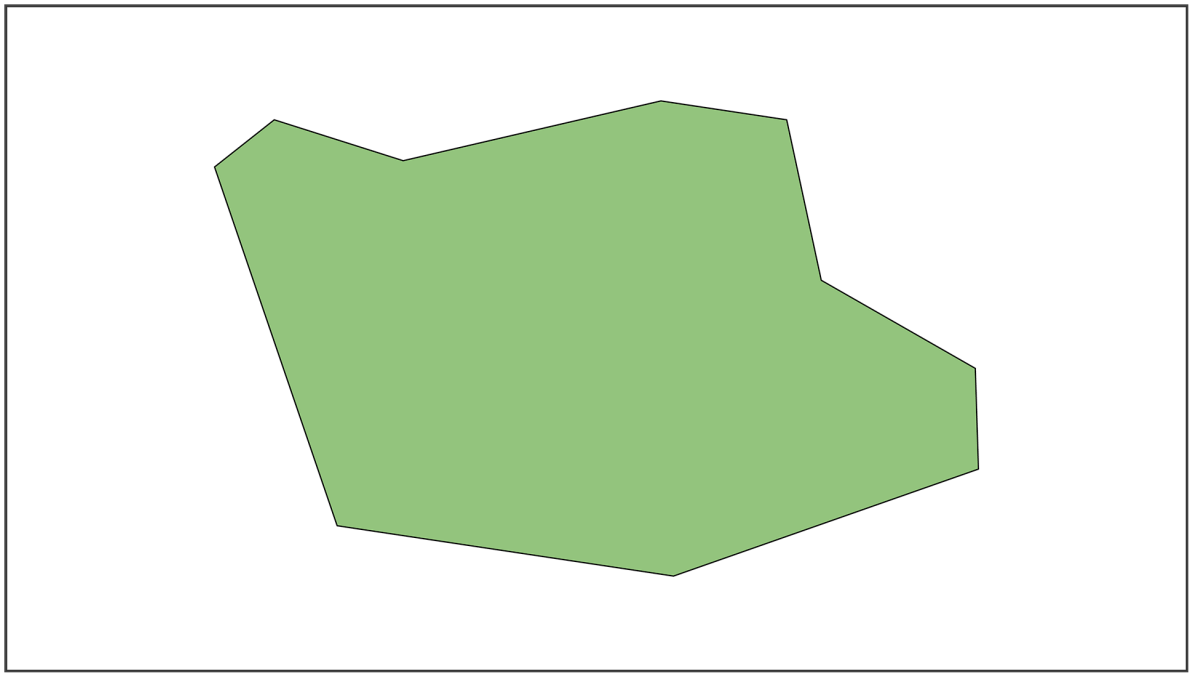
In the JSON data, we can see that the geometry type is "Polygon." To form a polygon, the coordinates have to be an array of linear ring (LinearRing) coordinate arrays. A LinearRing is a LineString with at least four or more sets of coordinates, where the starting and end points have the same coordinates.

In our first object in the `torontoNeighborhoods.json` file, the starting point coordinates are `[-79.39119482699992, 43.68108112399995]`.



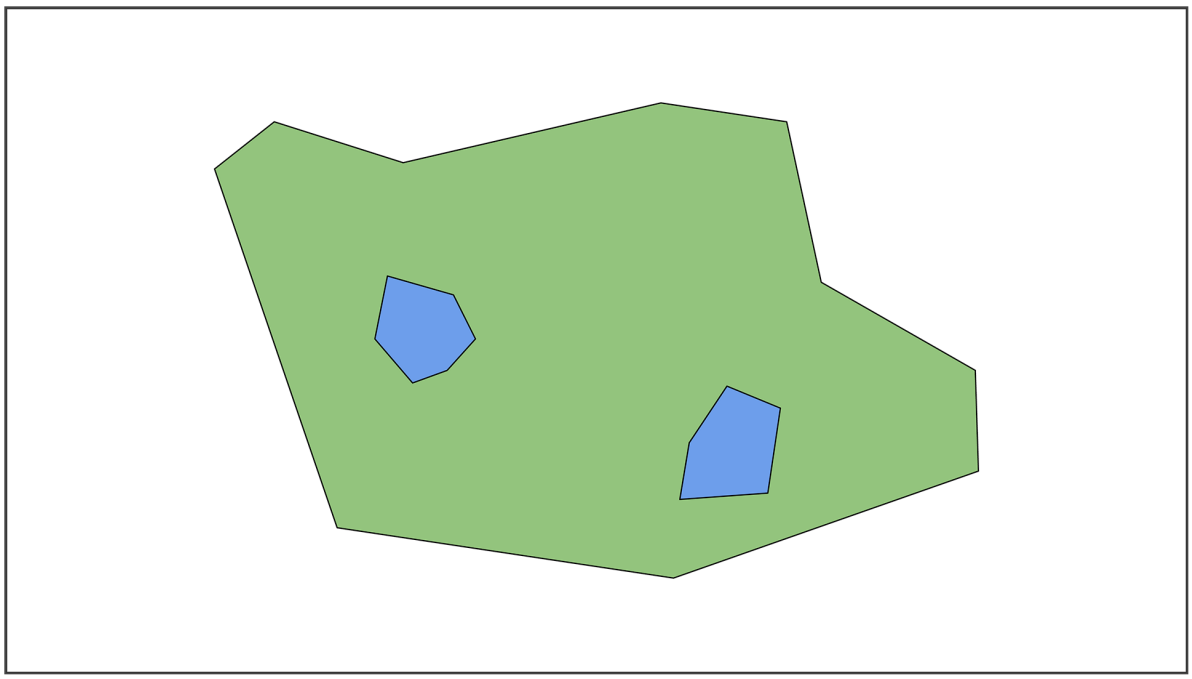
If we scroll to the end of the first object, the end point coordinates are also `[-79.39119482699992, 43.68108112399995]`.

In addition, the polygon can have multiple LinearRings, each with a different shape. The outer boundary of the polygon is one LinearRing shape, and there can be one or more LinearRing shapes inside that polygon. Consider the following solid polygon:



This is a solid polygon, like the neighborhood boundaries we'll map. There are no LinearRing shapes inside. However, if we add boundaries around the livable areas

of a neighborhood with two bodies of water, like a lake or pond, then the polygon might look like the following, where the blue areas represent those bodies of water. This polygon would have multiple LinearRing shapes inside the outer boundary LinearRing:



Now that we know more about polygons, let's map this data. Since we have copied the `logic.js` file from the Mapping_GeoJSON_Linestrings and added it to our Mapping_GeoJSON_Polygons folder, we'll make some changes to the `logic.js` file.

First, we'll use the `Streets` and `Satellite Streets` map styles, so we'll need to change the variables and the API code for each style:

- Use `streets` as the variable for the `Streets` style.
- Use `satelliteStreets` as the variable for the `Satellite Streets` style.

Since we have changed the variables for our map styles, complete the code for the base layer that holds both maps.

```
let baseMaps = {
 “Streets”: [                    ],
 “Satellite Streets”: [                    ],
};
```

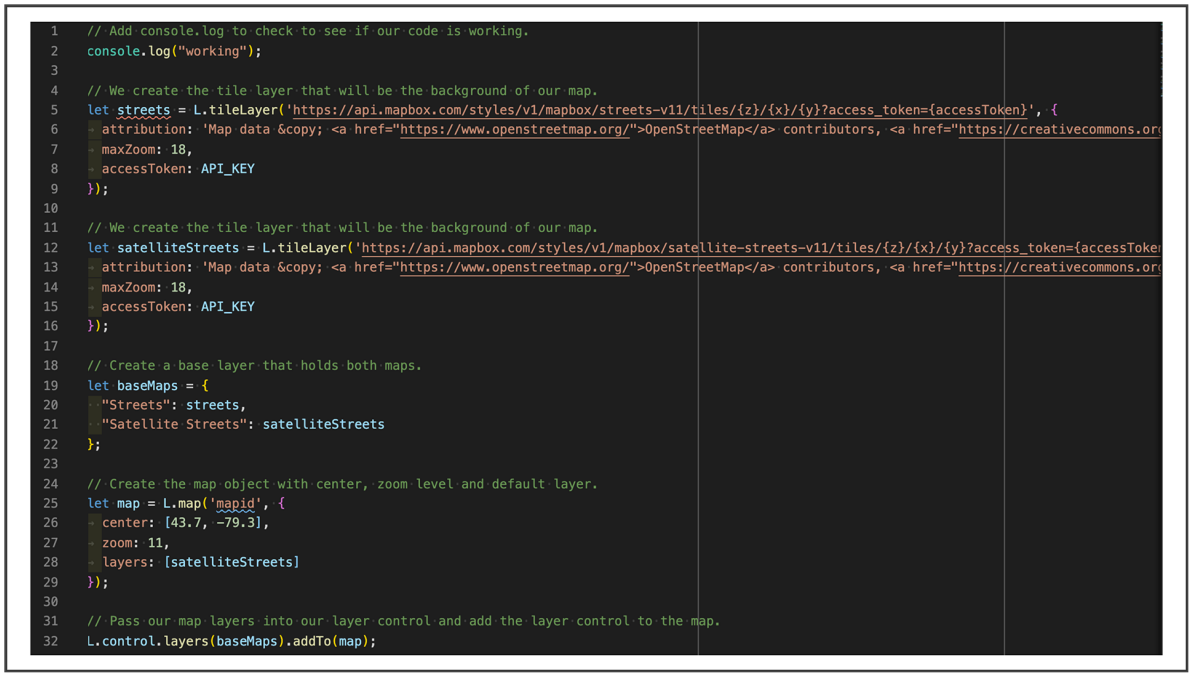
streets | satelliteStreets | Streets | Satellite Streets

Check Answer

Finish ▸

Next, change the center of our map to Toronto with the coordinates `[43.7, -79.3]`, with a zoom level of 11, and we’ll make the default layer `satelliteStreets`.

Our `logic.js` file should look like the following:



```
// Add console.log to check to see if our code is working.
console.log("working");

// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.or
    maxZoom: 18,
    accessToken: API_KEY
});

// We create the tile layer that will be the background of our map.
let satelliteStreets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v11/tiles/{z}/{x}/{y}?access_token={accessToke
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.or
    maxZoom: 18,
    accessToken: API_KEY
});

// Create a base layer that holds both maps.
let baseMaps = {
  "Streets": streets,
  "Satellite Streets": satelliteStreets
};

// Create the map object with center, zoom level and default layer.
let map = L.map('mapid', {
    center: [43.7, -79.3],
    zoom: 11,
    layers: [satelliteStreets]
});

// Pass our map layers into our layer control and add the layer control to the map.
L.control.layers(baseMaps).addTo(map);
```

Now we'll create a `torontoHoods` variable in our `logic.js` file and assign it to the URL for the `torontoNeighborhoods.json` file on GitHub, which should look like the following:

```
// Accessing the Toronto neighborhoods GeoJSON URL.
let torontoHoods = "https://raw.githubusercontent.com/<GitHub_name>/Mappi
```

Finally, we'll need to edit the `d3.json()` method that includes the `L.geoJSON()` layer to plot the data.

Complete the following code in the `d3.json()` method that includes the `L.geoJSON()` layer to map the polygons for the Toronto Neighborhoods.

```
d3.json(                                    ).then(function(data) {
    console.log(                                );
                                            ;
});
```

- `L.geoJSON(data).addTo(map)`
- `torontoHoods`
- `data`
- `L.geoJSON(torontoHoods).addTo(map)`
- `torontoNeighborhoods`
- `L.geoJson(data).addTo(map)`

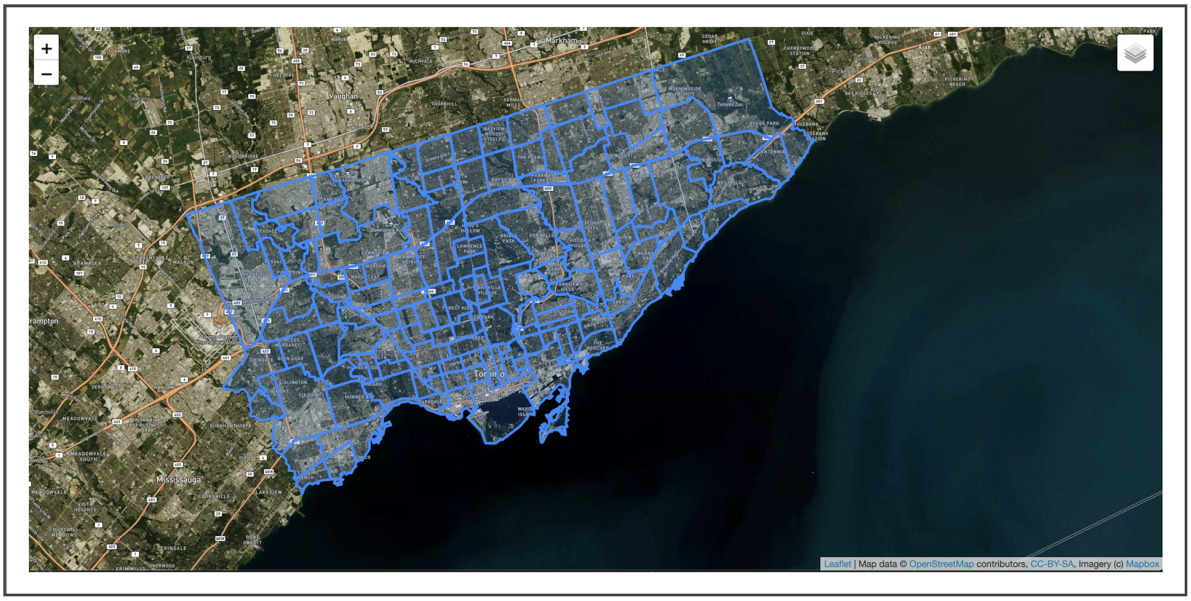
Check Answer

Finish ▸

After we add the code to access and parse the `torontoNeighborhoods.json` data, the `logic.js` file should look like the following:

```
33
34  // Accessing the Toronto neighborhoods GeoJSON URL.
35  let torontoHoods = "https://raw.githubusercontent.com/          /Mapping_Earthquakes/master/torontoNeighborhoods.json";
36
37  // Grabbing our GeoJSON data.
38  d3.json(torontoHoods).then(function(data) {
39    console.log(data);
40    // Creating a GeoJSON layer with the retrieved data.
41    L.geoJSON(data).addTo(map);
42  });
```

After you save your `logic.js` file and open your `index.html` file in your browser, your map should look like the following:



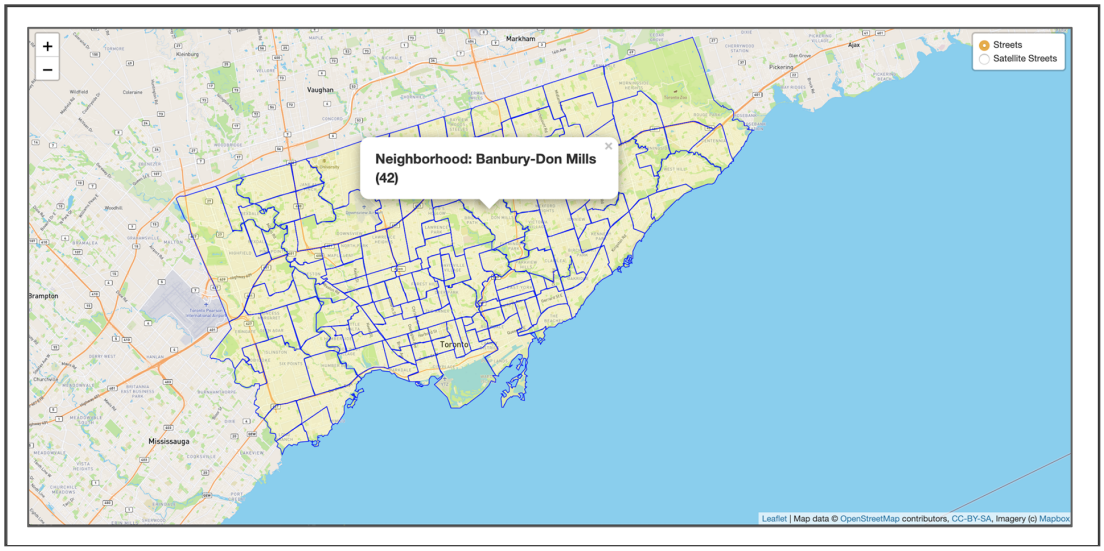
Great job! You're getting really good at mapping GeoJSON data!

Skill Drill

Edit your `logic.js` file with the following changes:

1. Make the lines blue, with a `weight` of 1.
2. Make the polygon fill color yellow.
3. Add a popup to show each neighborhood.
4. Make the default map layer `Streets` with `Satellite Streets` the second option.

Your map should look like the following:



Note

For more information, see the **Leaflet documentation on changing color options and other features of polygons (https://leafletjs.com/reference-1.6.0.html#path-color)** .

Add, Commit, Push

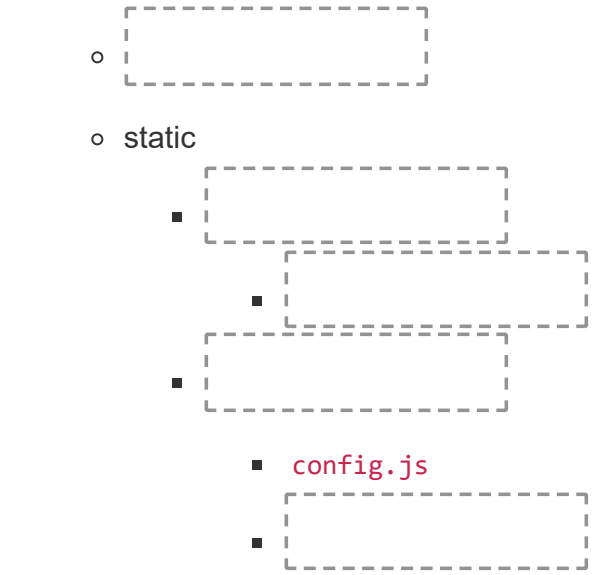
Add, commit, and push your changes to your Mapping_GeoJSON_Polygons branch. Don't delete the branch so that others can use it to learn how to map lines.

13.6.1: Add Earthquake Data to a Map

Now that you know how to access GeoJSON data, parse the data, and add it to a map, Sadhana would like you to map all recorded earthquakes in the past seven days. Once you get the data, you'll add some features to the map to showcase the severity of earthquakes for viewers.

As before, we need to set up a folder structure for our project in a new branch. Create a branch called "Earthquakes_past7days." Copy the folders and files from one of your previous branches and add them to the Earthquakes_past7days folder.

What is the structure of our Earthquakes_past7days folder?



css | js | style.css | CSS | logic | index.html | style | logic.js

Check Answer

Finish ▸

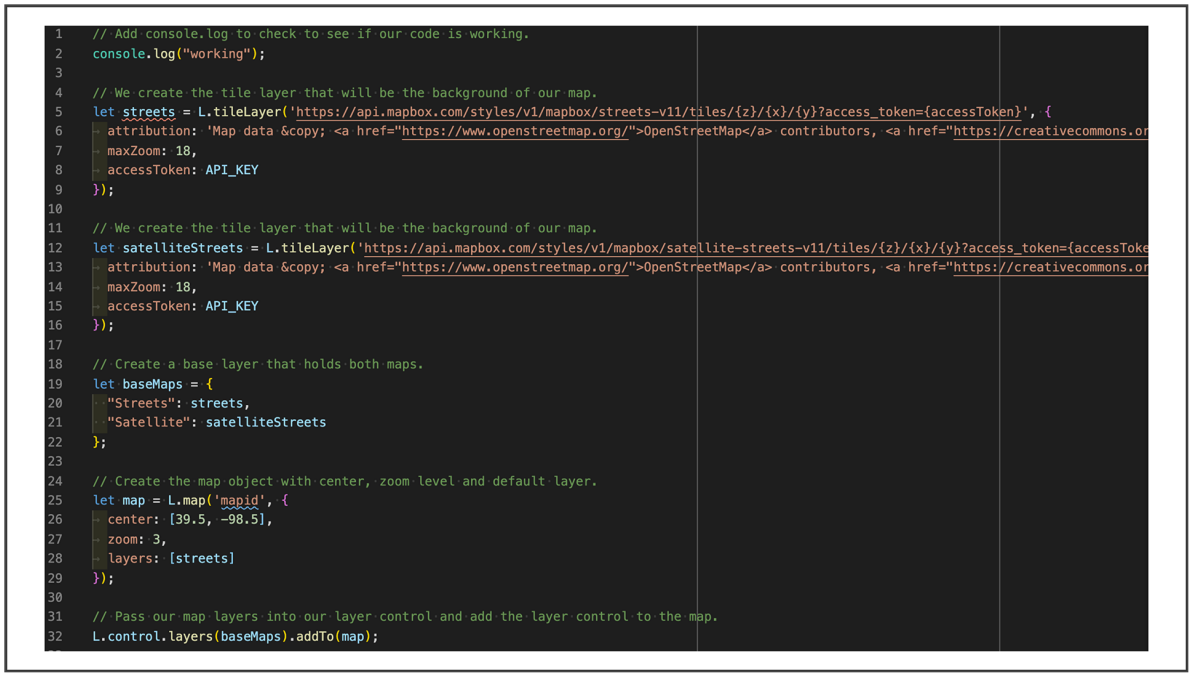
First, Sadhana wants you to rename the `logic.js` file to `logicStep1.js`. This way, each step has its own `logic.js` file that can be used by other interns in the future.

Now we'll edit our `logicStep1.js` file to create a map with all recorded earthquakes from the past seven days.

First, apply the `streets` and `satelliteStreets` map styles used for the GeoJSON polygon mapping. Change the text for the maps on the base layer to read as "Streets" and "Satellite" to look like the following:

```
// Create a base layer that holds both maps.
let baseMaps = {
  "Streets": streets,
  "Satellite": satelliteStreets
};
```

Change the center of our map to the geographic center of the United States using the coordinates `[39.5, -98.5]`, with a zoom level of 3 and default layer `streets`. Our `logicStep1.js` file should look like the following:

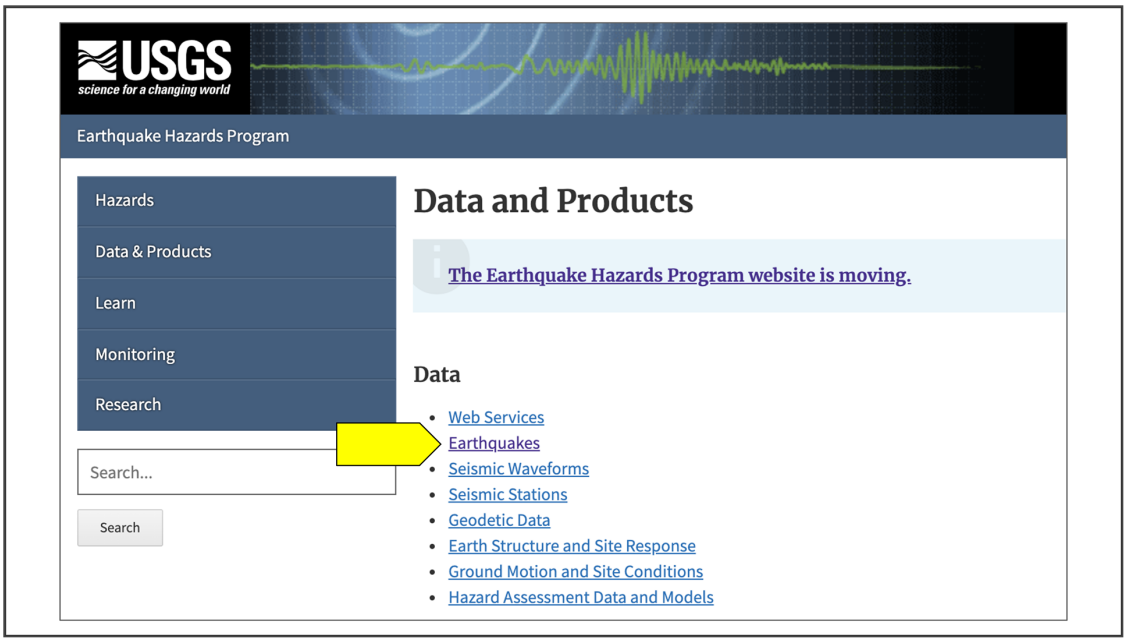


```
// Add console.log to check to see if our code is working.
console.log("working");

// We create the tile layer that will be the background of our map.
let streets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/streets-v11/tiles/{z}/{x}/{y}?access_token={accessToken}', {
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.or
    maxZoom: 18,
    accessToken: API_KEY
});

// We create the tile layer that will be the background of our map.
let satelliteStreets = L.tileLayer('https://api.mapbox.com/styles/v1/mapbox/satellite-streets-v11/tiles/{z}/{x}/{y}?access_token={accessToke
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.or
    maxZoom: 18,
    accessToken: API_KEY
});

// Create a base layer that holds both maps.
let baseMaps = {
  "Streets": streets,
  "Satellite": satelliteStreets
};

// Create the map object with center, zoom level and default layer.
let map = L.map('mapid', {
    center: [39.5, -98.5],
    zoom: 3,
    layers: [streets]
});

// Pass our map layers into our layer control and add the layer control to the map.
L.control.layers(baseMaps).addTo(map);
```

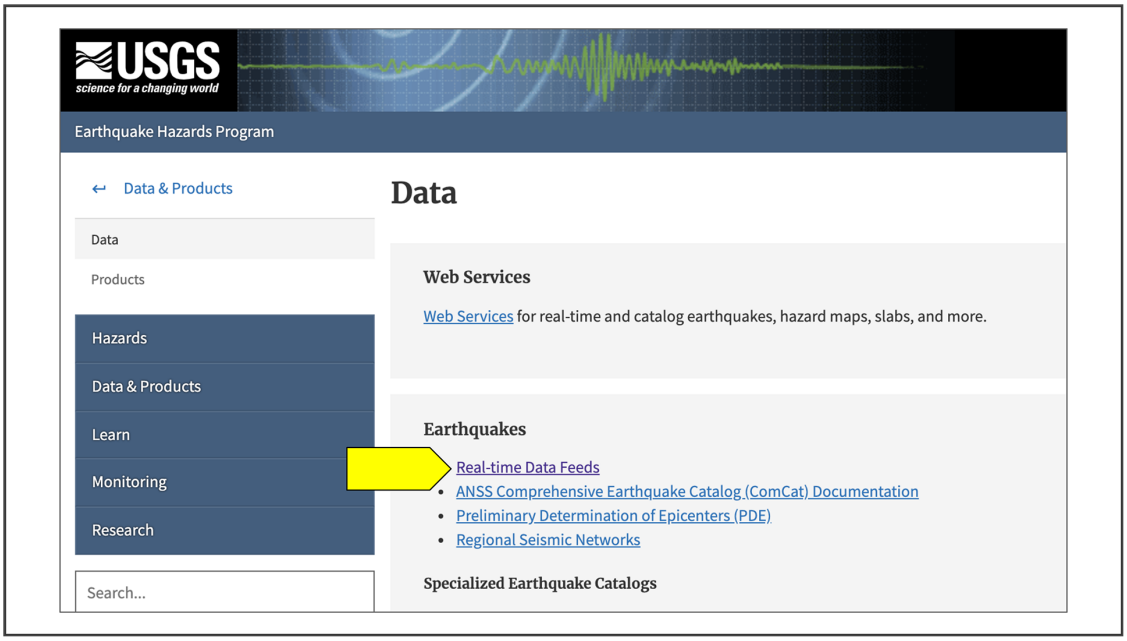
Add the USGS URL for earthquake data by following these steps:

1. Navigate to the **USGS Hazards Program (https://earthquake.usgs.gov/data/)**. You should see a page that looks like the following:

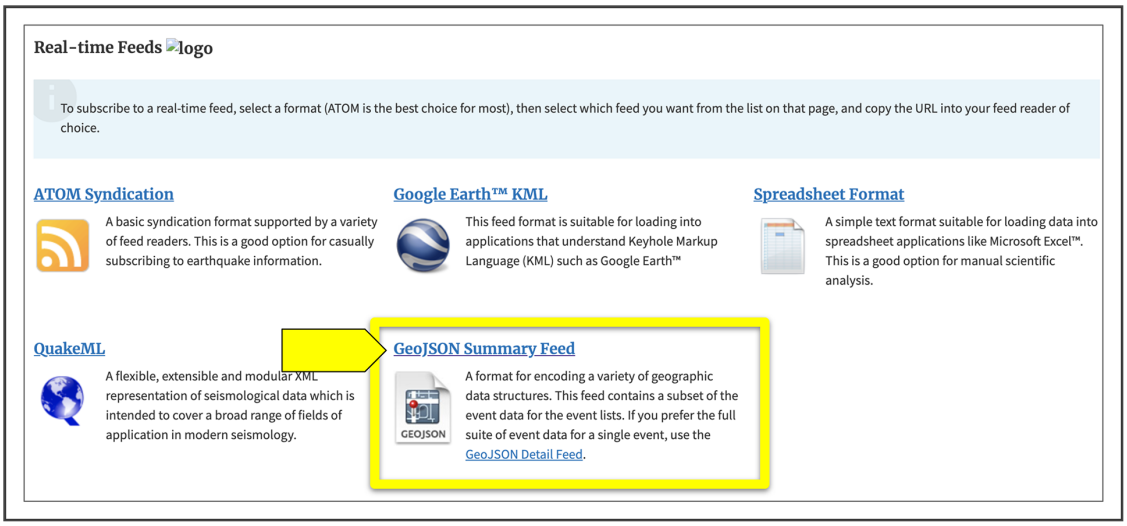
2. Click the Earthquakes link to open the following page:



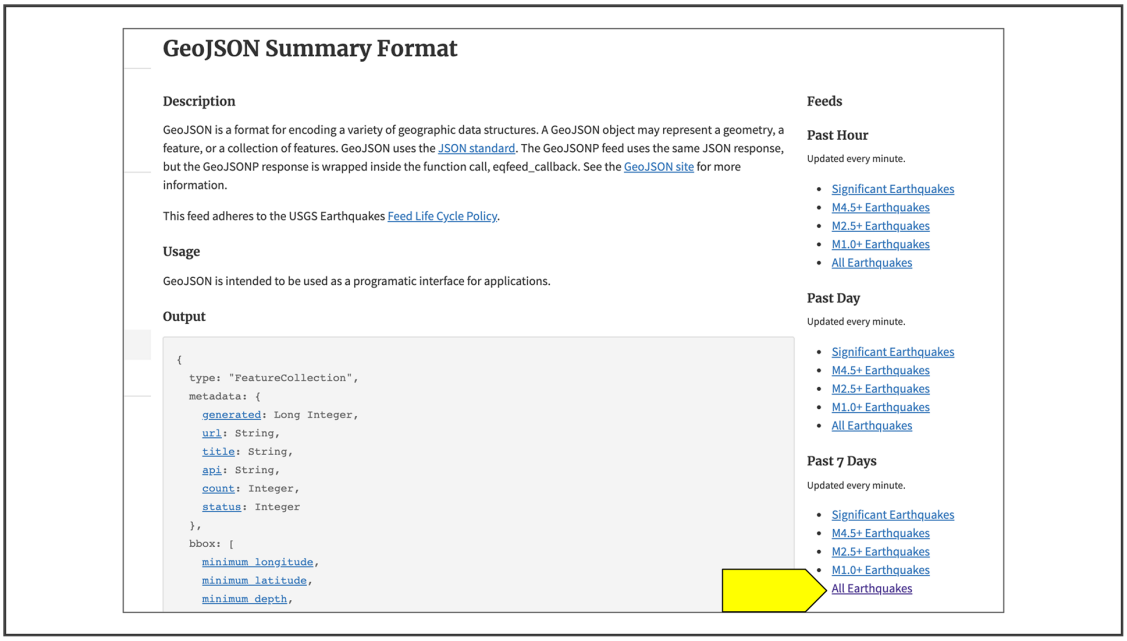
3. Click the Real-time Data Feeds link and scroll down until you see “GeoJSON Summary Feed":



4. Click the GeoJSON Summary Feed link:



5. On the right-hand side, click the All Earthquakes link under the “Past 7 Days” heading:



Nice work! The GeoJSON data will launch in your browser:

Review the GeoJSON earthquake data and answer the following questions.

1. What type of geometry is the earthquake data?

2. How would you get the magnitude of the earthquake?

3. How would you get the location (place) of the earthquake?

`features.properties.mag` `features[0].properties.place` `features[0].properties.mag` Point Points `features.properties.place`

Check Answer

Finish ▸

If we look closer at the geometry object, we'll see an additional data point in the coordinates object, 3.91, which is the depth of the earthquake in kilometers:

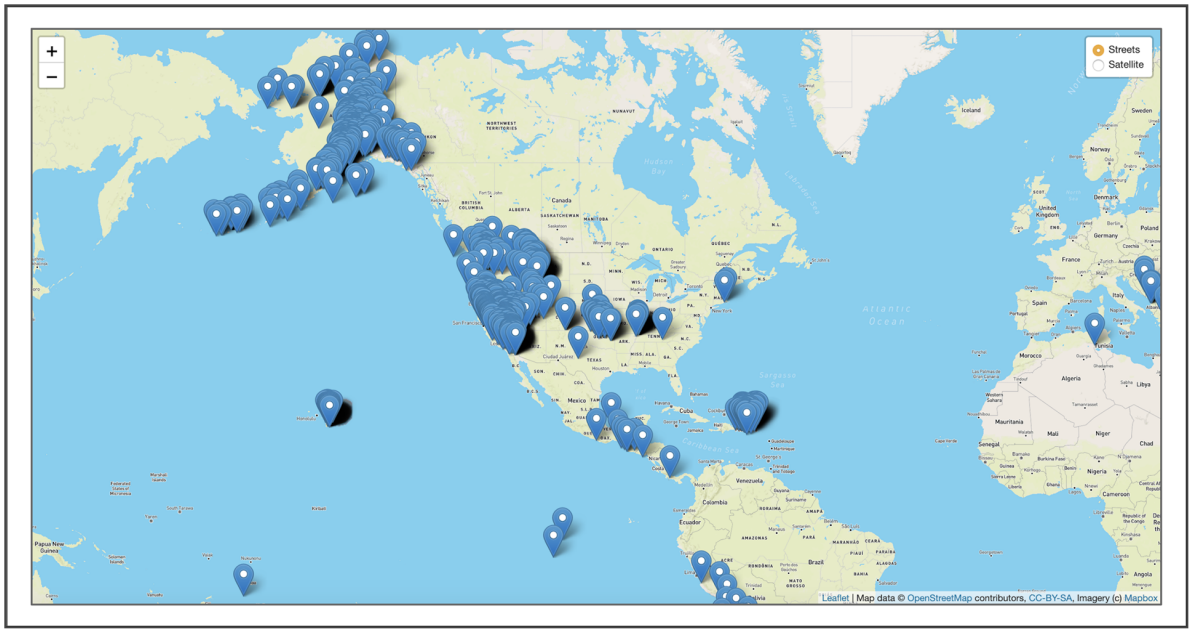
```
- geometry: {
    type: "Point",
  - coordinates: [
      -116.3505,
      33.9515,
      3.91
    ]
  };
```

> **Note**
>
> For more information on earthquake depth and other terms, see the **Event Terms (https://earthquake.usgs.gov/data/comcat/data-eventterms.php)**.

Copy the URL for the earthquake JSON data recorded for the past seven days, and add it in place of the previous URL in the `d3.json()` method. It should look like the following:

```
// Retrieve the earthquake GeoJSON data.
d3.json("https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_we
  // Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data).addTo(map);
});
```

After saving the `logicStep1.js` file and opening the `index.html` file in your browser, the map should look like the following. Make sure you are referring to the correct `logic` file in your `index.html` file:



Great job adding the earthquake data to our maps!

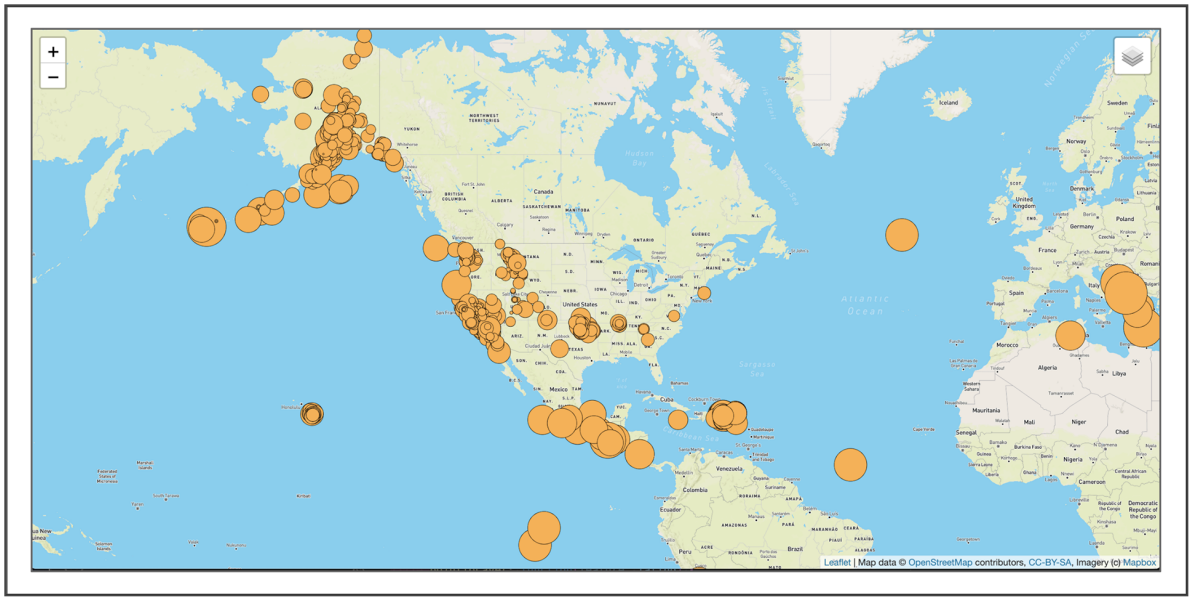
> ### Add, Commit, Push
>
> Add, commit, and push your changes to your Earthquakes_past7days branch.

Let’s make this data visually interesting by changing the marker to a circle with a radius representing the earthquake’s magnitude, and then we’ll style each earthquake data point.

13.6.2: Add Style to the Earthquake Data

As a first step in making the earthquake data more visually appealing, Sadhana would like you to add some styling to the earthquake data and vary the radius of each earthquake based on the magnitude.

After styling and modifying the radius of the circle for each earthquake's magnitude, our map should look like the following:



Before we write the code to create this map, make a copy of the `logicStep1.js` file and name it `logicStep2.js`. Now let's edit the file.

First, we'll change the basic marker to a circleMarker by using the `pointToLayer` function.

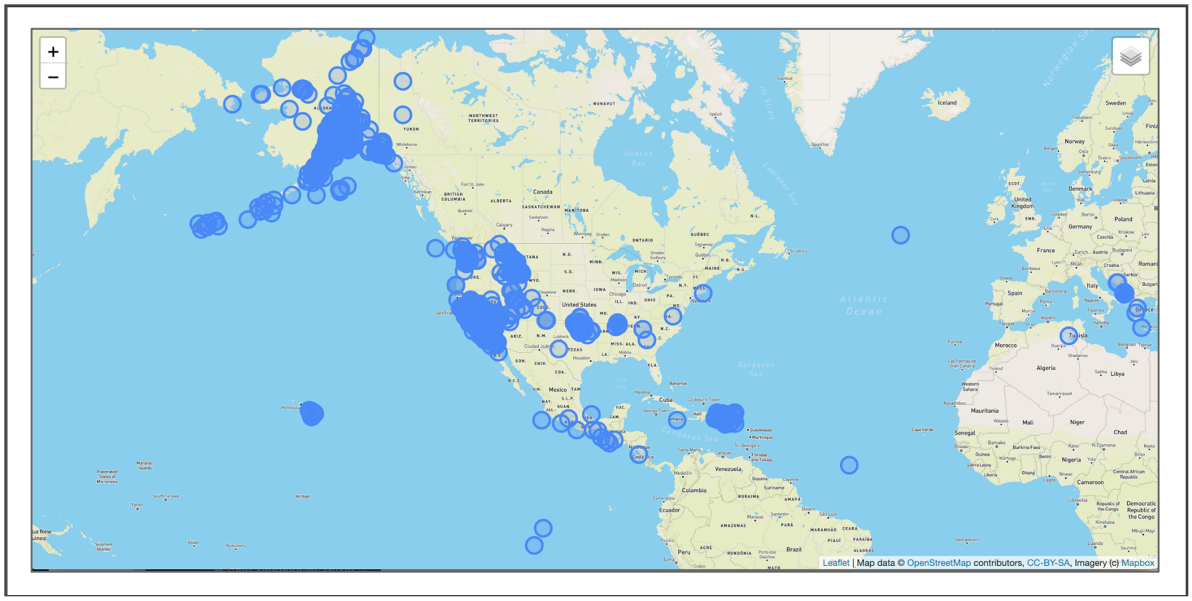
```
{
```

> **Rewind**
>
> For the `pointToLayer` callback function, the basic syntax for adding functionality to a marker is:
>
> `L.geoJson(data, {`
> `pointToLayer: function(feature, latlng) {`
> `return L.marker(latlng);`
> `}`
> `});`

For our purposes, we'll use `circleMarker` instead of `marker` in the above code. Edit your GeoJSON layer code to look like the following:

```
// Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data, {

// We turn each feature into a circleMarker on the map.

pointToLayer: function(feature, latlng) {
            console.log(data);
            return L.circleMarker(latlng);
        },
    }).addTo(map);
});
```

Save the file and let's see what the data looks like on the map. The `index.html` file should look like the following:



Next, we'll create a style for each earthquake by adjusting the line color, fill color, opacity, fill opacity, stroke, weight, and radius.

> **Rewind**
> When we defined the line style for the nonstop flight routes from Toronto, we created a style variable like the following:
>
> ```
> let myStyle = {
> color: "#ffffa1",
> weight: 2
> }
> ```

We'll create a `function styleInfo()`, which will contain all the style parameters for each earthquake plotted. Within this function, we'll create a `getRadius()` function to calculate the radius for each earthquake.

Add the following `function styleInfo()` inside the `d3.json()` method:

```
// This function returns the style data for each of the earthquakes we pl
// the map. We pass the magnitude of the earthquake into a function
// to calculate the radius.
function styleInfo(feature) {
  return {
    opacity: 1
```

```
    fillOpacity: 1,
    fillColor: "#ffae42",
    color: "#000000",
    radius: getRadius(),
    stroke: true,
    weight: 0.5
  };
}
```

Let's review the style we're creating for each earthquake:

- In the `styleInfo()` function, we passed the argument `feature` to reference each object's features.
- The `opacity` and `fillOpacity` are set at 1, the `stroke` is "true," and the `weight` is 0.5.
- The `fillColor` is light orange, and the color is `"#000000"` (black).
- The `getRadius()` function retrieves the earthquake's magnitude. Next, we'll create the `getRadius()` function to calculate the radius of the circle from the magnitude.

What dot notation code would you add inside the `getRadius()` function to get the magnitude?

- `features.properties.mag`
- `feature.properties.mag`
- `magnitude`

Check Answer

Finish ▸

In the `getRadius()` function for our `styleInfo()` function, add the following code to retrieve the earthquake's magnitude: `feature.properties.mag`.

Next, we'll create the `getRadius()` function. Add the following code below the `styleInfo()` function:

```
// This function determines the radius of the earthquake marker based on
// Earthquakes with a magnitude of 0 will be plotted with a radius of 1.
function getRadius(magnitude) {
  if (magnitude === 0) {
    return 1;
  }


  return magnitude * 4;
}
```

In the `getRadius()` function, we'll pass the `magnitude` argument that will reference the `feature.properties.mag` in the `styleInfo()` function. Then we'll use a conditional statement that sets the magnitude to 1 if the magnitude of the earthquake in the JSON file is 0 so that the earthquake is plotted on the map. If the magnitude is greater than 0, then the magnitude is multiplied by 4.

Now, that we created our style, let's add it to the map.

What code would you add in the `L.geoJSON()` layer to style the map?

```
L.geoJson(data, {

    pointToLayer: function(feature, latlng) {

                console.log(data);

                return L.circleMarker(latlng);

        },

    style: [                    ]

    }).addTo(map);

});
```

styleInfo | styleInfo(feature) | Styleinfo | mapStyle | style

Check Answer

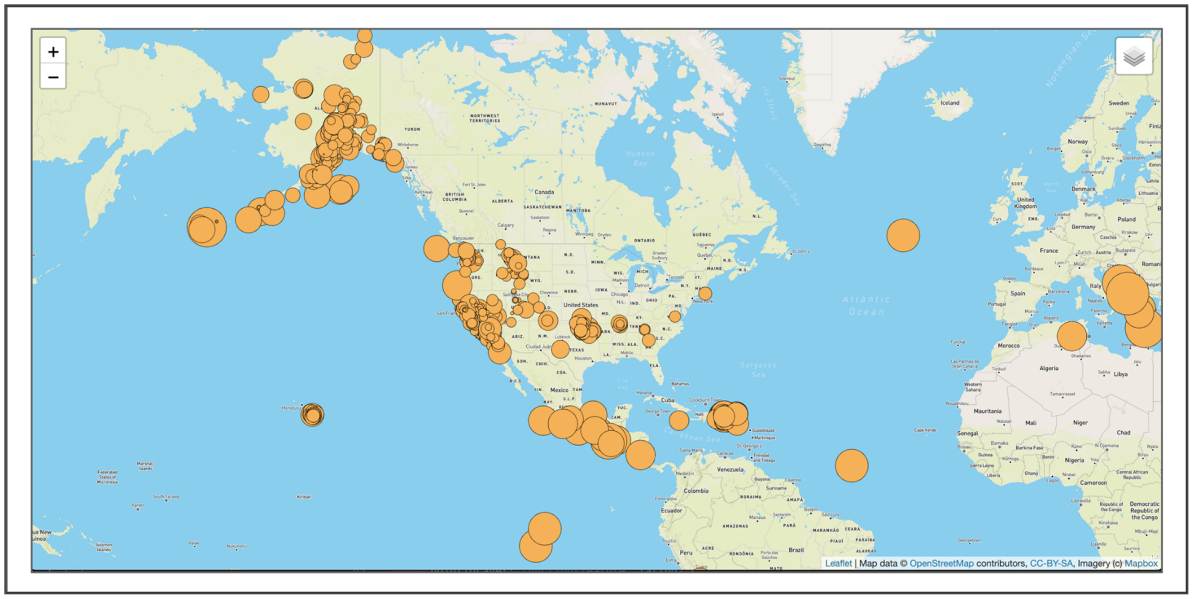
Finish ▸

To add style to the `L.geoJson()` layer, the `style` key will be assigned to the `styleInfo` function we created. Make sure the code for your `L.geoJson()` layer looks like the following:

```
// Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data, {

// We turn each feature into a circleMarker on the map.

pointToLayer: function(feature, latlng) {
            console.log(data);
            return L.circleMarker(latlng);
        },
```

```
      // We set the style for each circleMarker using our styleInfo funct
    style: styleInfo
    }).addTo(map);
});
```

When you save your `logicStep2.js` file and open `index.html` in your browser, your map will look like the following:



Great job styling each earthquake on our map!

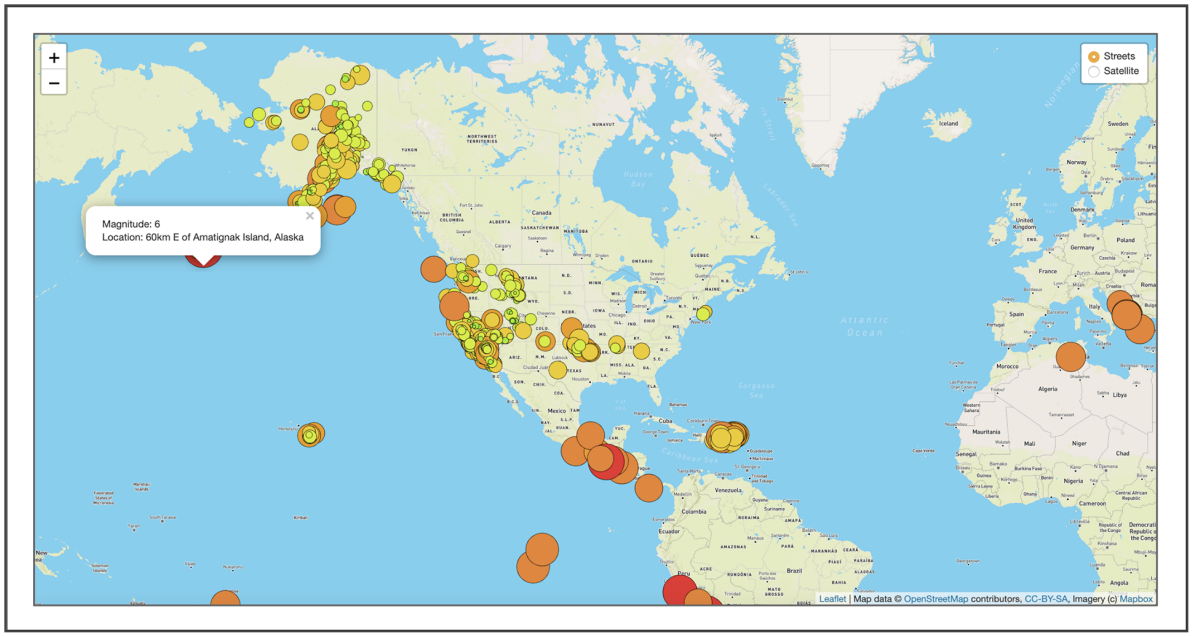
> ### Add, Commit, Push
>
> Add, commit, and push your changes to your Earthquakes_past7days branch.

Let's continue making the earthquake data visually appealing by styling colors to represent magnitudes as well as by adding informational popups.

13.6.3: Add Color and a Popup for Each Earthquake

Sadhana thinks that the size of the earthquake data based on magnitude looks great, but it’s hard to tell the difference between earthquakes within the same area. As you toss around ideas to make this data more accessible to the viewer, you come up with the idea to color-code the earthquakes based on magnitude. You aren't quite sure how to do this yet, but you know that it should be possible based on your experience with JavaScript thus far. Basil loves the idea, so you get back to coding to figure out how to make it happen. And while you're working on changing the color code for magnitude, Basil and Sadhana suggest that you add the magnitude and location as a popup for each earthquake.

After we're done adding a color range for the magnitude and a popup for each earthquake, our map should look like the following:



Before we write the code to create this map, make a copy of the `logicStep2.js` file and name it `logicStep3.js`. Now let's edit the file.

First, we'll create a fill-color range for the magnitude. In the `styleInfo()` function, our `fillColor` was set with `fillColor: "#ffae42"`. We'll replace the hexadecimal color code with the function `getColor()`. Inside the parentheses, we'll add the dot notation code to get the magnitude as we did for the `getRadius()` function, since we'll change the color of each earthquake marker based on the magnitude.

Add the `getColor(feature.properties.mag)` function for the `fillColor` so that our `styleInfo()` function looks like the following:

```
// This function returns the style data for each of the earthquakes we pl
// the map. We pass the magnitude of the earthquake into two separate fur
// to calculate the color and radius.
function styleInfo(feature) {
  return {
    opacity: 1,
    fillOpacity: 1,
    fillColor: getColor(feature.properties.mag),
    color: "#000000",
    radius: getRadius(feature.properties.mag),
    stroke: true,
    weight: 0.5
```

```
  };
}
```

Now we need to write code for the `getColor()` function to change the marker's color based on the magnitude. For example, if the magnitude is greater than 5, it will be a certain color, if the magnitude is greater than 4, it will be a different color, and so on.

What type of expression would we use for the `getColor()` function?

- Logical expression
- Conditional expression
- Boolean expression

Check Answer

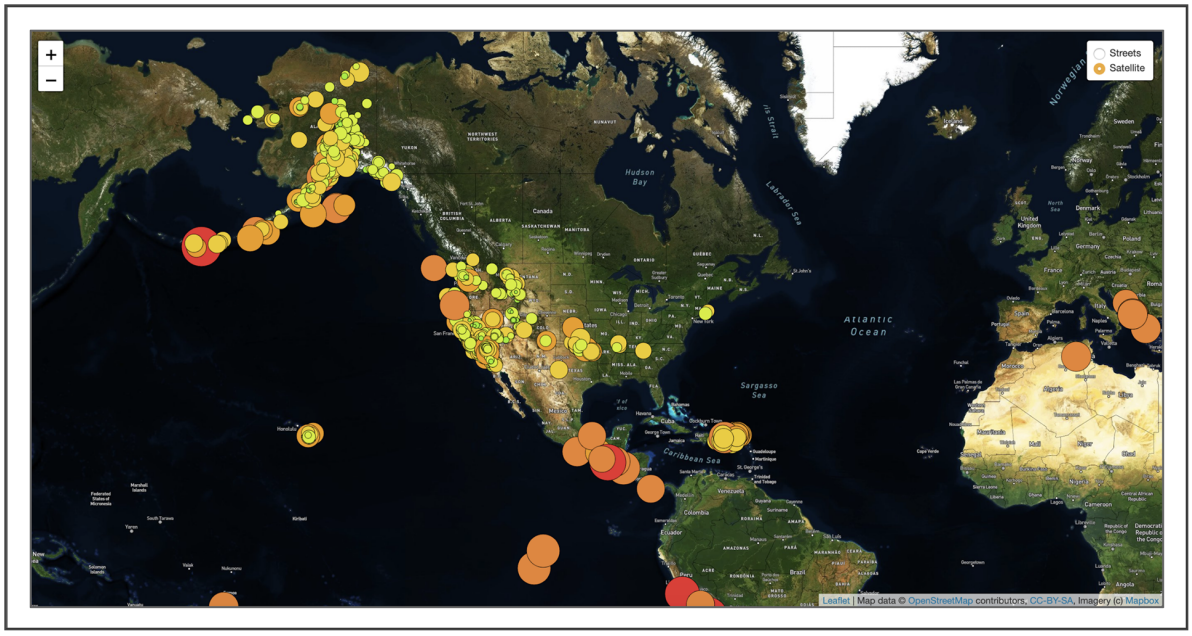
Finish ▸

For the `getColor()` function, we'll write a conditional expression with logical operators for the magnitudes. Add the following `getColor()` function below the `styleInfo()` function and above the `getRadius()` function. Sadhana suggests using the following colors for the magnitudes since they'll be visible on the Satellite map:

```
// This function determines the color of the circle based on the magnitud
function getColor(magnitude) {
  if (magnitude > 5) {
    return "#ea2c2c";
  }
  if (magnitude > 4) {

    return "#ea822c";
  }
  if (magnitude > 3) {
```

```
    return "#ee9c00";
  }
  if (magnitude > 2) {
    return "#eecc00";
  }
  if (magnitude > 1) {
    return "#d4ee00";
  }
  return "#98ee00";
}
```

Let's save our `logicStep3.js` file and open the `index.html` file in the browser to confirm our code is working. When we select the dark map, our map should look like the following:



Now we need to edit the GeoJSON layer code to add the popup for the magnitude and location.

What function are we going to use to add a popup for the `circleMarker`?

- `onEachFeature`
- `bindPopup()`
- `filter`

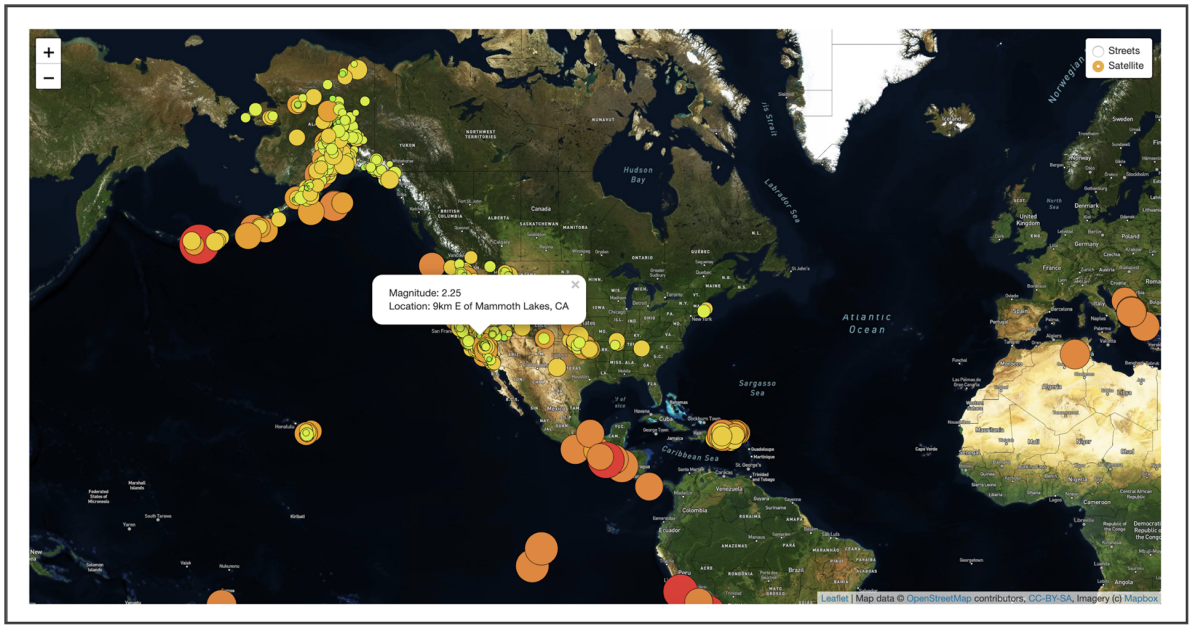
Check Answer

Finish ▸

In the `geoJson` layer, we'll add the `onEachFeature` function to add a popup for each circle marker. Edit the `L.geoJson()` layer code to include the `onEachFeature` function with the `bindPopup()` method:

```
// Creating a GeoJSON layer with the retrieved data.
L.geoJson(data, {
    // We turn each feature into a circleMarker on the map.
    pointToLayer: function(feature, latlng) {
        console.log(data);
        return L.circleMarker(latlng);
      },
    // We set the style for each circleMarker using our styleInfo functic
  style: styleInfo,
    // We create a popup for each circleMarker to display the magnitude a
    //  location of the earthquake after the marker has been created and
    onEachFeature: function(feature, layer) {
    layer.bindPopup("Magnitude: " + feature.properties.mag + "<br>Locatic
  }
}).addTo(map);
```

When you save the `logicStep3.js` file and open your `index.html` file in your browser, the Satellite map option will look like the following:



Great job on adding color and a popup marker to each earthquake!

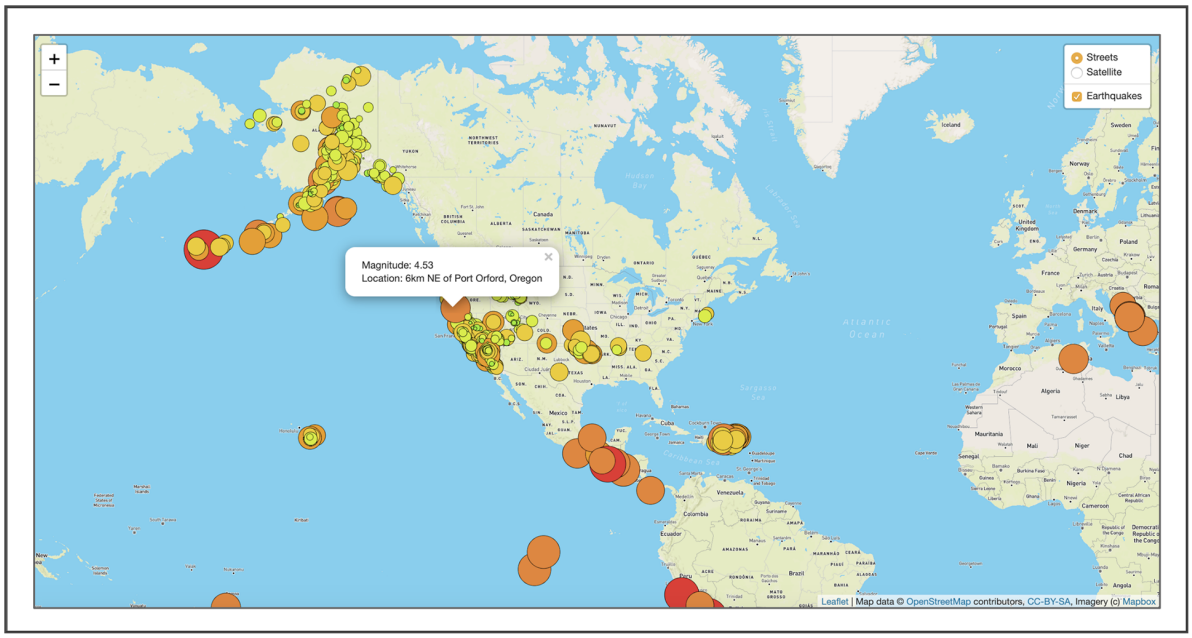
> ### Add, Commit, Push
>
> Add, commit, and push your changes to your Earthquakes_past7days branch.

Next, Sadhana will show you how to add the earthquake data as an overlay to the tile layer so that the data can be turned on and off by the viewer.

13.6.4: Add Earthquake Data as an Overlay

The earthquake map is looking great. Sadhana thinks that having the earthquake data as an overlay on both the Streets and Satellite tile layers, would be a nice added feature so users can turn the data on and off.

After adding an overlay for the earthquakes, our map should look like the following, allowing the viewer to toggle off and on the earthquake data. The default setting will always show the data:



Before we write the code to create this map, make a copy of the `logicStep3.js` file and name it `logicStep4.js`. Now, let's edit the file.

Refer to **Layer Groups and Layers Control (https://leafletjs.com/examples/layers-control/)** for guidance on how to add data as an overlay to the map.

> **Note**
> The base layers or tile layers, the Streets and Satellite, are mutually exclusive, and only one can be visible at a time on our map. Whereas, overlays are anything that you want to add to the map, which are "laid over" all the base layers and are visible all the time.

In the example below, from the **Layer Groups and Layers Control (https://leafletjs.com/examples/layers-control/)** page, we can add data to a LayerGroup class. In the example given, the `cities` variable is assigned to the `layerGroup()`. For our purposes, we'll use the earthquake data:

> Instead of adding them directly to the map, you can use the following LayerGroup class:
>
> ```
> var cities = L.layerGroup([littleton, denver, aurora, golden]);
> ```

Let's create an overlay layer for our earthquake data. Add the following code to your `logicStep4.js` file below the code for the base layer that holds the two different map styles:

```
// Create the earthquake layer for our map.
let earthquakes = new L.layerGroup();
```

Next, define the overlay object to add it to the map. Add the following code below the earthquake layer group:

```
// We define an object that contains the overlays.
// This overlay will be visible all the time.
let overlays = {
  Earthquakes: earthquakes
};
```

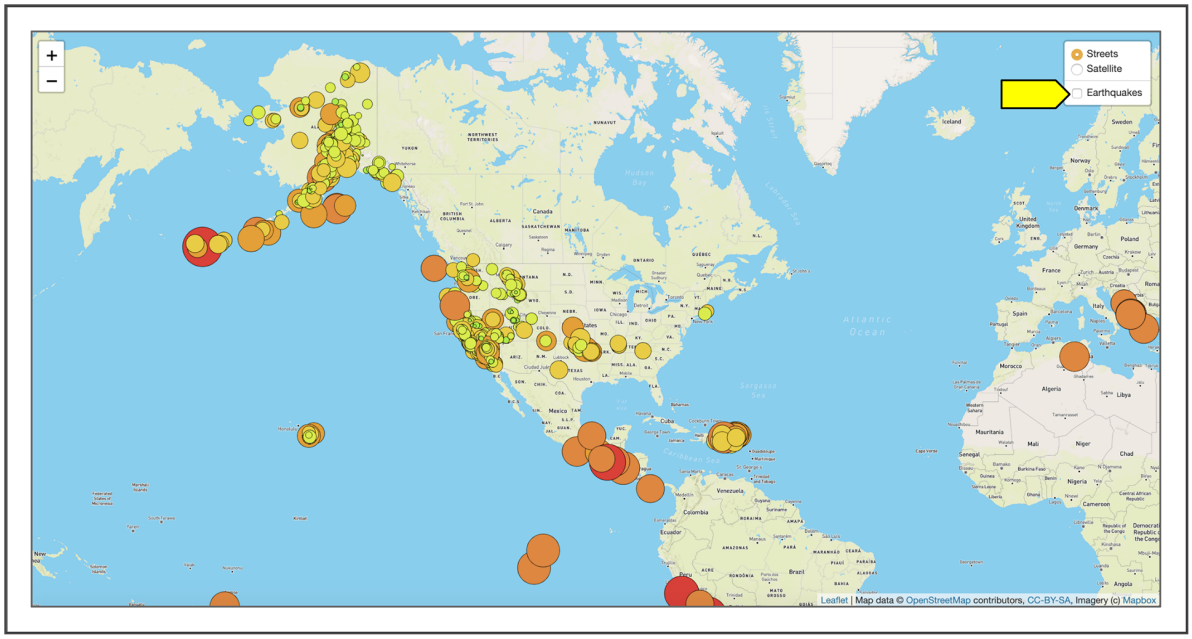
To add the overlay to the map, add the variable `overlays` to the Layers Control object. Edit the Layers Control object so that the `overlays` object will show up on the tile layers control:

```
// Then we add a control to the map that will allow the user to change
// which layers are visible.
L.control.layers(baseMaps, overlays).addTo(map);
```

Your `logicStep4.js` file should look like the following with the added code:

```
// Create a base layer that holds both maps.
let baseMaps = {
  "Streets": streets,
  "Satellite": satelliteStreets
};

// Create the earthquake layer for our map.
let earthquakes = new L.LayerGroup();

// We define an object that contains the overlays.
// This overlay will be visible all the time.
let overlays = {
  Earthquakes: earthquakes
};

// Then we add a control to the map that will allow the user to change which
// layers are visible.
L.control.layers(baseMaps, overlays).addTo(map);

```

If we open the `index.html` file in our browser, we see that the earthquake data has loaded, but the earthquake overlay button is not on:



Our `L.geoJSON()` layer code looks like this at this point:

```
// Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data, {
      // We turn each feature into a circleMarker on the map.
      pointToLayer: function(feature, latlng) {
          console.log(data);
          return L.circleMarker(latlng);
      },
      // We set the style for each circleMarker using our styleInfo function.
    style: styleInfo,
    // We create a popup for each circleMarker to display the magnitude and location of the earthquake
    //  after the marker has been created and styled.
    onEachFeature: function(feature, layer) {
      layer.bindPopup("Magnitude: " + feature.properties.mag + "<br>Location: " + feature.properties.place);
    }
  }).addTo(map);

});
```

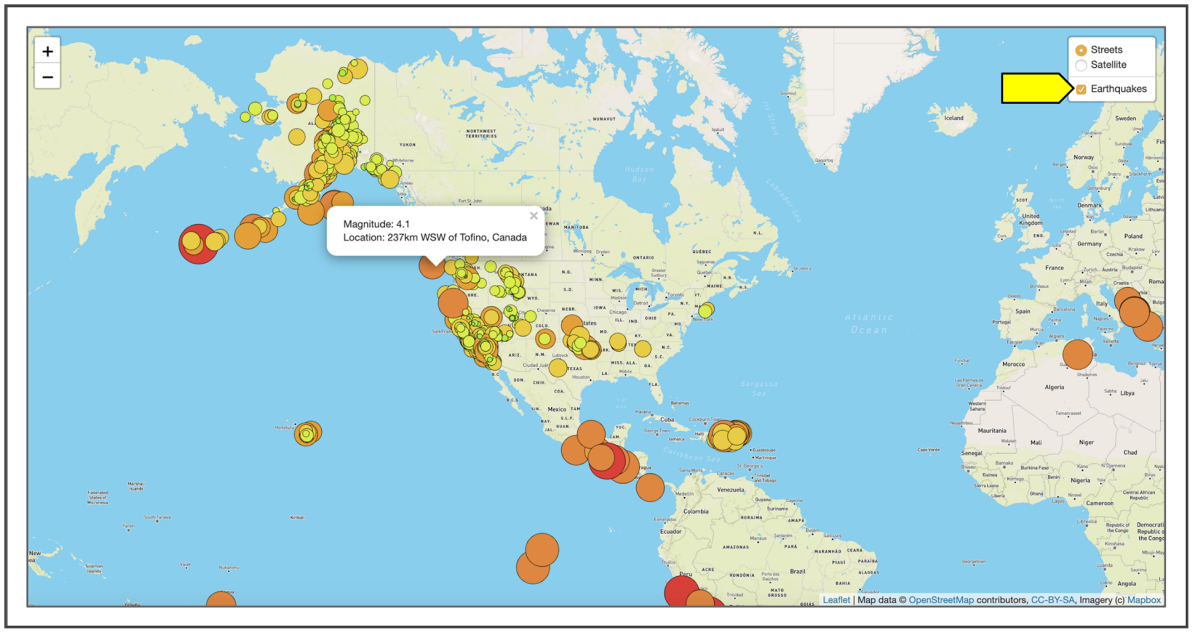
To have the Earthquakes overlay button “on,” we need to:

- Replace the `map` variable in the `addTo(map)` function with `earthquakes`.
- Before the closing bracket and parenthesis of the `d3.json()` method we add the earthquake layer to the map, with `earthquakes.addTo(map);`.

Edit your `addTo(map)` function at the end of your `L.geoJSON()` layer code, as shown in the image to look like the following:

```
// Creating a GeoJSON layer with the retrieved data.
  L.geoJson(data, {
      // We turn each feature into a circleMarker on the map.
      pointToLayer: function(feature, latlng) {
          console.log(data);
          return L.circleMarker(latlng);
      },
      // We set the style for each circleMarker using our styleInfo function.
    style: styleInfo,
      // We create a popup for each circleMarker to display the magnitude and location of the earthquake
      //  after the marker has been created and styled.
      onEachFeature: function(feature, layer) {
        layer.bindPopup("Magnitude: " + feature.properties.mag + "<br>Location: " + feature.properties.place);
      }
  }).addTo(earthquakes);

      // Then we add the earthquake layer to our map.
      earthquakes.addTo(map);
});
```

Now, when we open the `index.html` file in our browser, we can see that the earthquake data has loaded *and* the earthquake overlay button is “on":



Nice job adding the earthquake data as an overlay to the map!

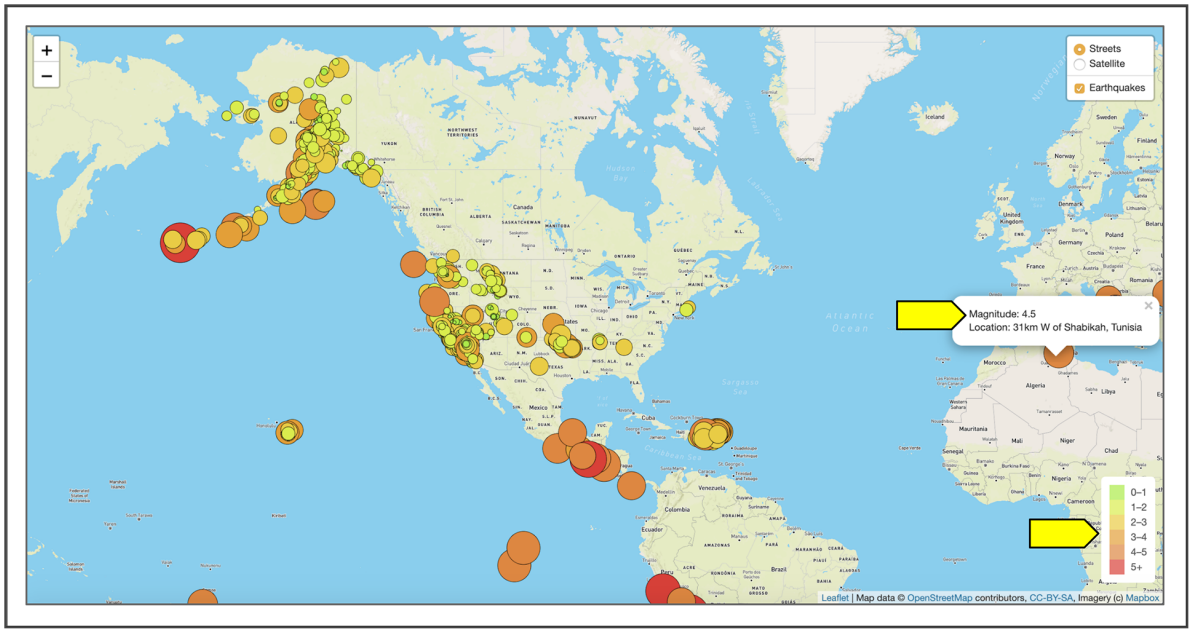
> ### Add, Commit, Push
>
> Add, commit, and push your changes to your Earthquakes_past7days branch.

Sadhana loves the map, but she thinks having a legend to indicate what magnitude is represented by each color would be helpful when viewing the map.

13.6.5: Add a Legend to the Map

There is one final piece to add to the map: a legend for the color range of the earthquakes. Basil and Sadhana think a legend will provide information needed for the colors of the earthquakes to make sense to the viewer without having to click on each marker.

After we add a legend, our earthquake map should look like the following, with the legend at the bottom right of the map:



Sadhana tells you that we can use the Leaflet **choropleth examples page (https://leafletjs.com/examples/choropleth/)** to help us write the code for the legend. On the **choropleth examples page (https://leafletjs.com/examples/choropleth/)**, scroll down to the “Custom Legend Control” section:

Custom Legend Control

Creating a control with a legend is easier, since it is static and doesn't change on state hover. JavaScript code:

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

legend.onAdd = function (map) {

      var div = L.DomUtil.create('div', 'info legend'),
        grades = [0, 10, 20, 50, 100, 200, 500, 1000],
        labels = [];

      //loop through our density intervals and generate a label with a colored square for each interval
      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);
```

Before we write the code to create this map, make a copy of the `logicStep4.js` file and name it `logicStep5.js`. Now let's edit the file.

On the choropleth examples page, copy the code for the Custom Legend Control and paste it below the `L.geoJSON()` layer, where we add the earthquake layer to the map, `earthquakes.addTo(map)`. Now, we'll edit the legend control object to suit our needs.

First, edit the code for the Leaflet `control()` object to look like the following. With this code, we'll place the legend at the indicated position—the bottom right:

```
// Create a legend control object.
let legend = L.control({
  position: "bottomright"
});
```

Next, remove the argument "map" from the legend function to look like the following:

```
// Then add all the details for the legend.
legend.onAdd = function() {
  let div = L.DomUtil.create("div", "info legend");
};
```

With this code, we're going to add a legend to the map with `legend.onAdd`. The legend will be added to a `div` element on the `index.html` file using the `DomUtil` utility function.

Next, we're going to change the `grades` array in the Leaflet documentation to a `magnitudes` array, and we'll add a `colors` array that holds the colors for our magnitudes. Add the following code inside our `legend.onAdd` function:

```
const magnitudes = [0, 1, 2, 3, 4, 5];
const colors = [
  "#98ee00",
  "#d4ee00",
  "#eecc00",
  "#ee9c00",
  "#ea822c",
  "#ea2c2c"
];
```

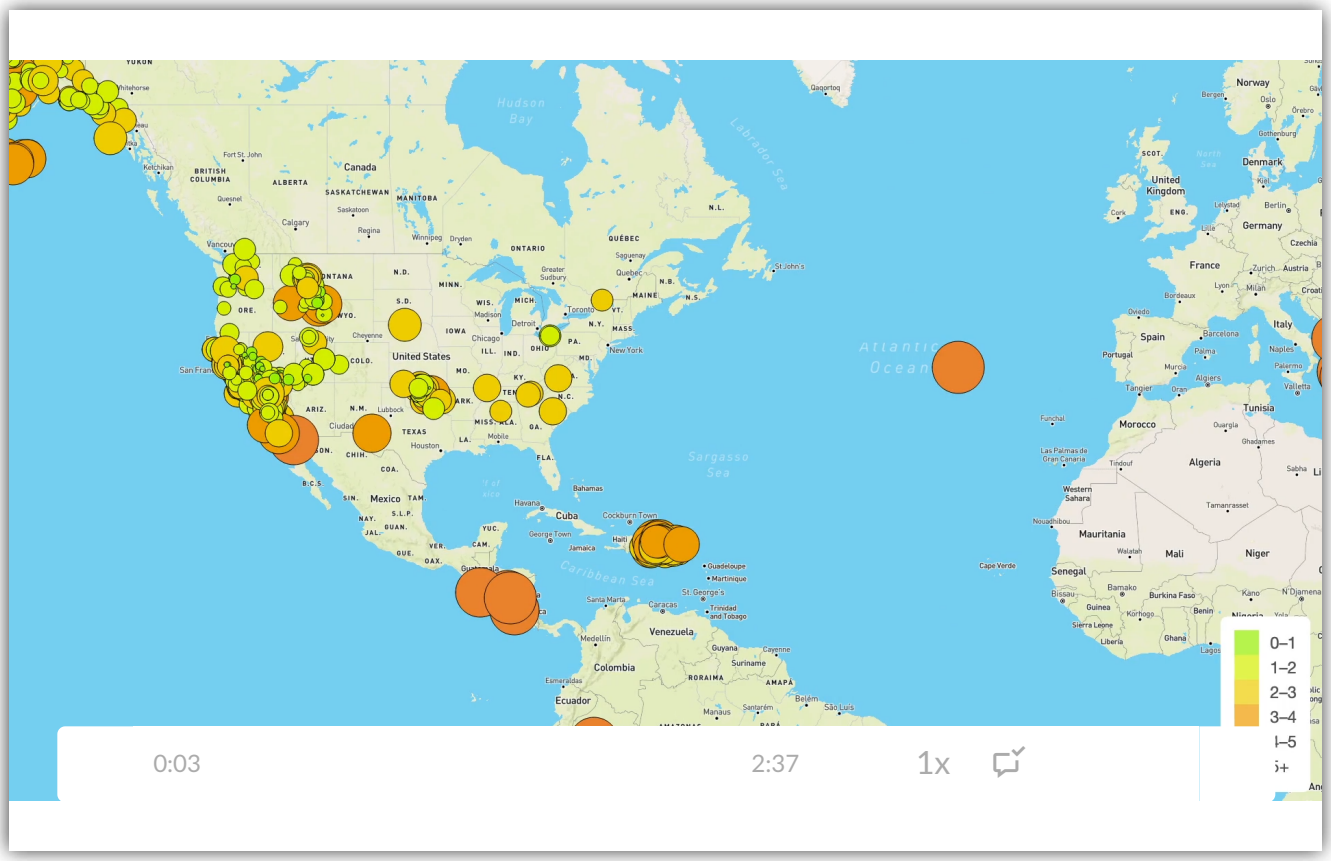
The final piece is to edit the `for` loop. The `for` loop will add the color choices from our `colors` array as a small box for the color of earthquakes and place the text of the magnitude range next to the box. Edit the `for` loop code to look like the following:

```
// Looping through our intervals to generate a label with a colored squar
   for (var i = 0; i < magnitudes.length; i++) {
     console.log(colors[i]);
     div.innerHTML +=

       " " +
       magnitudes[i] + (magnitudes[i + 1] ? "–" + magnitudes[i + 1] + ""
  }
   return div;
 };

 legend.addTo(map);
```

Let's review what's happening in this `for` loop:

1. After we iterate through the `magnitudes`, we'll add the color and text to the `div` element using `div.innerHTML +=`.
2. For each iteration, we'll add a color from the `colors` array by styling the background of an `<i>` tag with color options.
3. Next, we'll add the interval between earthquake magnitudes for our colors with the following code:
   `magnitudes[i] + (magnitudes[i + 1] ? "&ndash;" + magnitudes[i + 1] + "<br>" : "+")`.

This code is quite complex. For a deeper explanation of this code, watch the following video:



> **Note**
>
> If that seemed a bit complex, that's good! Encountering, unpacking, and using other people's complex code is a critical part of being a developer.

The last thing we need to do is style the legend using CSS. Below the JavaScript code for the legend is the CSS code. Copy the CSS code and add it our `style.css` file:

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

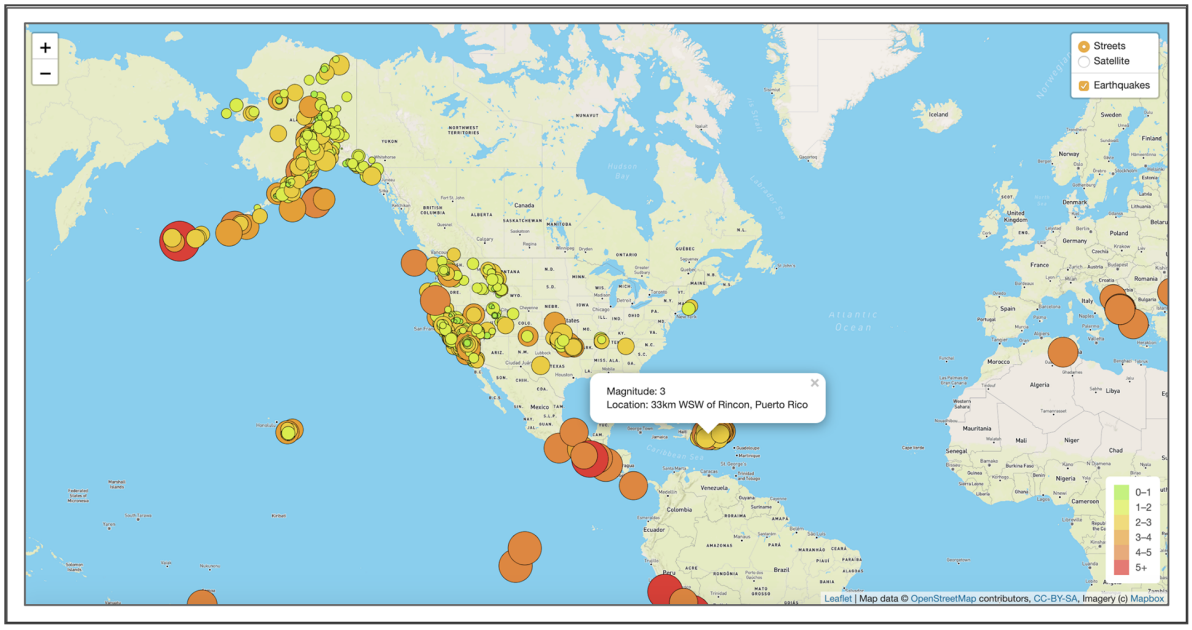
We won't edit the `<i>` tag of the `.legend` class (`.legend i`) in the `style.css` file, but let's edit our `.legend` class to create some padding around the legend, add a white background, and add a border radius. Edit your `.legend` class to look like the following:

```
.legend {
  padding: 10px;
  line-height: 18px;
  color: #555;
  background-color: #fff;
  border-radius: 5px;
}
```

Our `style.css` file should now look like the following:

```css
html,
body,
#mapid {
  width: 100%;
  height: 100%;
  padding: 0;
  margin: 0;
}
.legend {
  padding: 10px;
  line-height: 18px;
  color: #555;
  background-color: #fff;
  border-radius: 5px;
}

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

Save your `logicStep5.js` and `style.css` files. When you open `index.html` in your browser, your map should have a legend on the bottom right:



Congratulations on completing your earthquake map!

> ### Add, Commit, Push
>
> Add, commit, and push your changes to your Earthquakes_past7days branch.

13.6.6: Merge the Earthquake Branch with the Master Branch

Basil and Sadhana want to review your code for the Earthquakes_past7days branch. Once the code meets their approval, you can merge the branch with the master branch.

You've been creating a lot of GitHub branches in this module. However, you've only merged one branch into the master branch. Can you recall the steps for merging a branch into the master branch?

What is the order for merging a branch with the master branch?

1.
2.
3.
4.
5.

- Create the issue.
- Merge the pull request.
- The issues says you are “Able to merge.”
- Compare your branch with master with an issue.
- Compare your branch with master with a pull request.
- Create the pull request.
- Merge the issue.
- The pull request says you are “Able to merge.”
- Someone reviews the code and approves the pull request.

Merge Your Branch with the Master

Create the pull request and merge your Earthquakes_past7days branch with the master branch.

Module 13 Challenge

Submit Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Basil and Sadhana like how you created your earthquake map with two different maps and the earthquake overlay. Now Basil and Sadhana would like to see the earthquake data in relation to the tectonic plates' location on the Earth. In addition, they would like to see data on a third map.

In this challenge, you'll add a third map style as an additional tile layer. You'll also add tectonic plate GeoJSON data to the map to illustrate the relationship between the location and frequency of seismic activity and tectonic plates.

Background

To illustrate the severity of the earthquakes in relation to the tectonic plates, you'll need to log in to GitHub and access the tectonic plate data from this **GitHub repository. (https://github.com/fraxen/tectonicplates)** You will also need to make an API call to the tectonic plate data using `d3.json()`, and then add the data as an overlay to the map using the `L.geoJSON()` layer. In addition to the `streets` and `satelliteStreets` maps, you'll need to add a third map style of your choosing. All map styles must be added to the base layer so that they show up on the map to allow the user to change which layers are visible.

Objectives

The goals of this challenge are for you to:

- Use `d3.json()` to get tectonic plate data and add the data using the `L.geoJSON()` layer.
- Style the tectonic plate LineString data to stand out on the map.
- Add the tectonic plate data as an overlay with the earthquake data.
- Add a third map style to allow the user to select from three different maps.

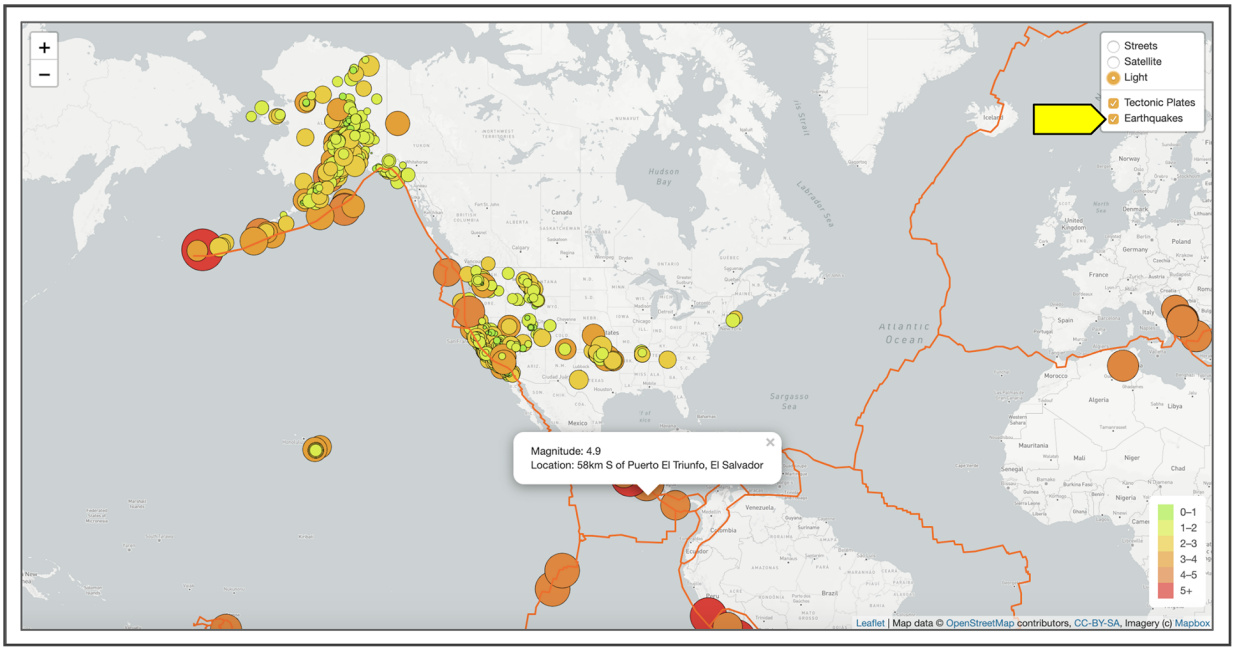
Instructions

To complete this challenge, follow these steps:

1. Create a new folder on your Mapping_Earthquakes repository and name it "Earthquake_Challenge."
2. Copy the folders and files from your Earthquakes_past7days branch and add them to the Earthquake_Challenge folder. The folder should have this structure:
    - Earthquake_Challenge folder
        - `index.html`
        - static
            - css
                - `style.css`
            - js
                - `config.js`
                - `logic.js`
3. Use the `GeoJSON/PB2002_boundaries.json` data from the **GitHub repository (https://github.com/fraxen/tectonicplates)** for the tectonic plate data. You'll need to log into GitHub to access the GeoJSON data.
4. Place the `d3.json()` call with the `L.geoJSON()` layer for the tectonic plate data at the end of your code from your Earthquakes_past7days branch.
5. Style the lines with a strong, bright color so the lines show up on the satellite map and are not too light to be seen on the lighter maps.
6. Create the tectonic plate layer for the map.

7. Add the tectonic plate layer to the `overlays` so that they show up in the tile layer, as shown in the image below.
8. Add the tectonic plate and earthquake data to the map for any map style choice.
9. Edit the base layer so that it holds all three maps.
10. Make the `streets` map the default map.

Your final map should look similar to the following image:



> **Add, Commit, Push**
>
> When you're done with the assignment, commit your work to your Mapping_Earthquakes GitHub repository.

Submission

To submit your challenge assignment, click Submit, and then provide the URL of your Mapping_Earthquakes GitHub repository for grading.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be

dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Rubric

Please **download the detailed rubric** to access the assessment criteria.

Module 13 Rubric						
Criteria	**Ratings**					**Pts**
Tectonic Plate Data Is on the Map Please see detailed rubric linked in Challenge description.	**70.0 pts Mastery**	**50.0 pts Approaching Mastery**	**30.0 pts Progressing**	**15.0 pts Emerging**	**0.0 pts Incomplete**	70.0 pts
A Third Map Is Added for the User to Choose From Please see detailed rubric linked in Challenge description.	**30.0 pts Mastery**	**20.0 pts Approaching Mastery**	**10.0 pts Progressing**	**5.0 pts Emerging**	**0.0 pts Incomplete**	30.0 pts
						Total Points: 100.0

Module 13 Career Connection

Welcome back to another Career Connection! This week you spent more time developing your skills in interactive data visualization using D3.js, Leaflet.js, and more JavaScript to engage with the GeoJSON file type. We're excited to spend a little time with you again to show you how this material is applicable to your future career in data.

It's time for you to boost your profile and make yourself more marketable by employing some **Employer Competitive** strategies—consider how you can use the material you've learned this week to become more employer competitive.

1. **Write an article on Medium.com (https://medium.com/) :**
    - This doesn't need to be a comprehensive tutorial. It could be a walkthrough of an activity you did this week, or it could be an overview of your journey in learning data visualization and analysis so far. Most important, though, is that you keep it clean and well-written, short, and a tool to market your own portfolio—so don't forget to add links to your portfolio, your GitHub, and maybe even your email address at the end of the article.

      Doing this will help you build your visibility as well as help you reflect on what you have learned so far.

2. **Update your resume with your new skillset:**
    - If you haven't already done so, add Leaflet.js and GeoJSON to your resume in the "Technical Skills" section. Display it loud and proud!

Technical Interview Preparation

Remember how we at Career Services keep saying that technical interviewing is a skill that requires constant practice and development? Now you get to practice this a little more… again! Remember, remember, remember: Employers are looking not only for the right answer, but they're looking to see your process and how you approach a topic—so even if you don't know the right answer, it's not the end of the world! Use the opportunity to demonstrate how you tackle questions that you don't readily know the answer to.

Jot down the things you might say in response to one of these questions before viewing the answer.

1. **What is Leaflet?**

   Leaflet is an extremely popular, open-source JavaScript library for mobile-friendly interactive maps. It helps us display and visualize data in ways that are user-friendly and meaningful for data analysts and nonexperts alike.

2. **Tell me about your experience using Leaflet?**

   There is no concrete answer here, but the "tell me about" format is extremely popular in interviews. It is an open-ended question that invites you to sell your skills! Write down your experiences using Leaflet this week. What were the challenges? How did you overcome them? What did you learn that is applicable to the job you're applying for?

3. **What is the GeoJSON data format?**

   GeoJSON is an open standard format, which is essentially a type of JSON (JavaScript Object Notation) for representing geographical features along with their nonspatial attributes. It is commonly used with libraries like Leaflet.js.

Case Study: Mind the Gap

If you've ever travelled to London, England, you likely stepped foot onto the London Underground and heard the iconic "Mind the Gap" announcement over the loudspeaker system. You'd also know that the London Underground is a complex web of train lines that have grown over the last century with no great planning or grid system—it is, quite simply, a masterful maze of tunnels.

Locals are usually very good at knowing where to go, which train lines start and stop, and where and when they're going to arrive. But for literally millions of tourists visiting London each year, it can be a mind-boggling adventure. So the Mayor of London has posted a job searching for a data analytics professional who can create a mobile-friendly interactive map.

The specs state that a user must be able to:

- Use a mobile device to access the map on-the-go.
- View each of the London Underground lines with a different color.
- Click on a line and get more information about a particular train, its stops, and arrival intervals.
- View each Underground station as a point on the map.
- Click each Underground station point and get more information about the trains that pass through it.

You applied and successfully convinced the astute HR Recruiter that you know what you're talking about. Now you're meeting with the development team for your technical interview. For each of the following questions that you might field during your interview, write down your answers *before* viewing our suggestions!

1. **In thinking about this map, what data format and libraries might you use to create an interactive mobile-friendly application?**

If you didn't guess it, Leaflet.js and GeoJSON are the go-to technologies here. Consider these technologies for each of the following questions.

2. **What would you use to map each of the train lines?**

   A GeoJSON LineString might be mapped as a Leaflet vector to represent the train lines.

3. **What would you use to map each of the station points of interest?**

   A GeoJSON point might be mapped as a Leaflet vector to represent each of the stations.

4. **If you wanted to highlight a whole neighborhood on the map, how would you do it?**

   A GeoJSON MultiPolygon might be mapped as a Leaflet vector to represent the different neighborhoods.

Continue to Hone Your Skills

If you're interested in learning more about the technical interviewing process and practicing algorithms in a mock interview setting, check out our **upcoming workshops.** **(https://careerservicesonlineevents.splashthat.com/)**

Career Services Next Step: Milestone 4 (https://courses.bootcampspot.com/courses/138/assignments/1734)