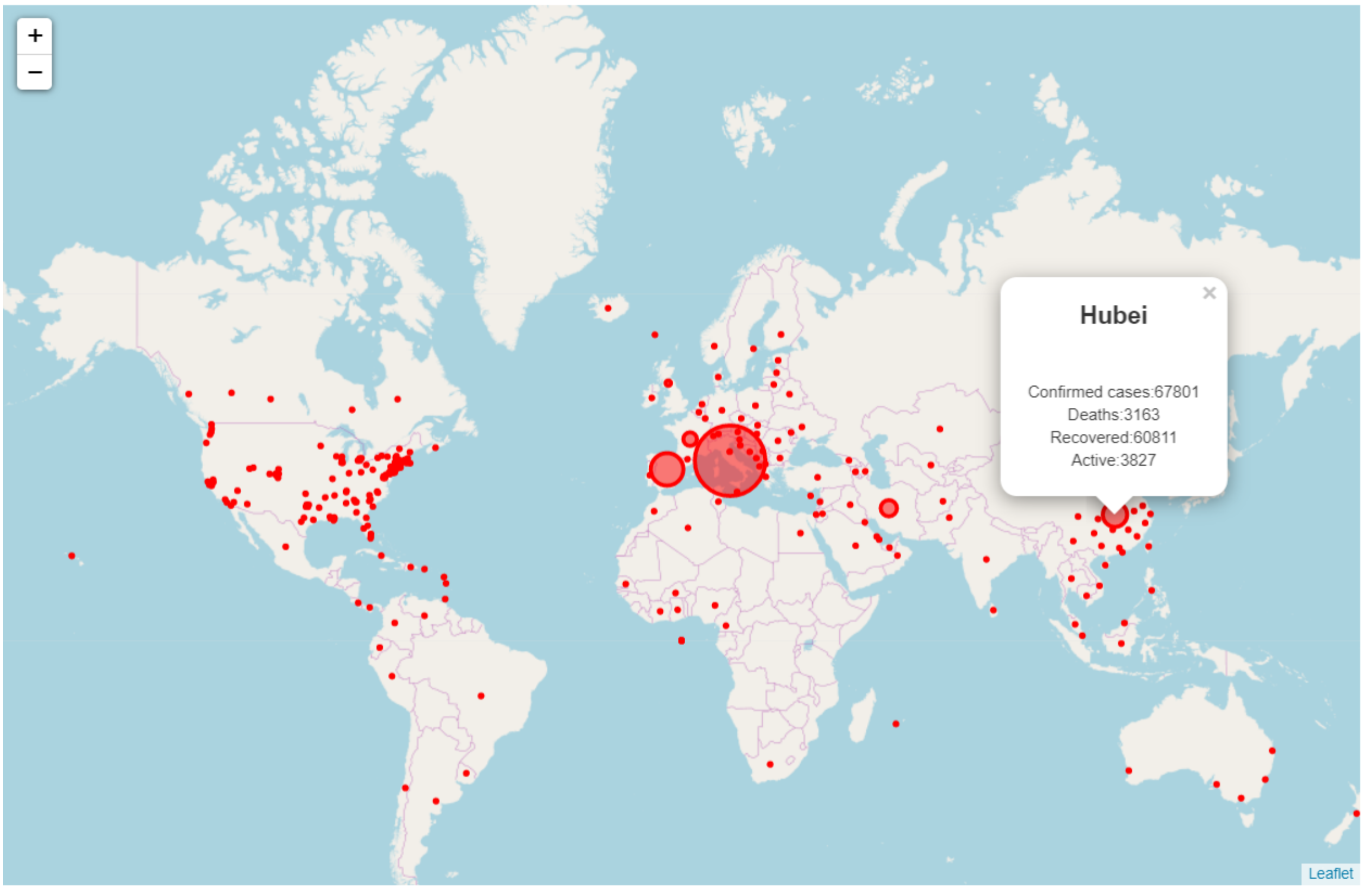
Make your first web map of the Coronavirus outbreak in 5 easy steps - for free!



Data is cool

When presented well, data can highlight some of the most profound facts about our world. From highlighting how far we've yet to go with gender equality, to how small we are in relation to the rest of the universe; and from the changing economic fates of nations to the devastating statistics of suicide, data can tell stories that change our view of the world and beyond. There are infinite ways to present data, allowing different stories to be told and facts to be highlighted. And the proliferation of the internet, access to data, and free data science tools and software is allowing greater reach of interesting data science than ever before. In this tutorial, I'll introduce Leaflet.js for making interactive web maps to visualize data, and use html and JavaScript to explore the stories in the ongoing global Coronavirus pandemic.

Some data is inherently spatial

Spatial data is data that has some sort of a location recorded, and it can be fascinating. The "spatial" bit of spatial data is most commonly seen as a location attached to each individual record in a dataset. This location might commonly be a latitude and longitude, or an x and y location in another coordinate reference system. Spatial data is normally represented as points, lines or polygons depending on the features the data is measuring, and may come in specialist file formats such as shapefiles. For this tutorial we will use a csv format that I've handily converted to JSON, so allow us to focus on learning to make maps rather than wrangle data.

5 steps to your first interactive web map

Making your first web map is far easier than you have probably realized. We're going to use Leaflet.js, a brilliant open source library for creating interactive web maps. All of the code for this tutorial can be found on Github, but this tutorial will walk your through the steps. There are 5 steps to creating your first web map and displaying it in a web browser. First, we'll import Leaflet.js and make a `<div>` to hold our map canvas. Then we'll add some code to create a map canvas and add a base map to it, to give our data some context. Then we'll loop over our data to add it to the map. After that, we'll alter the code to style the data and to draw out the story hidden in the numbers. Lastly, we'll add popups to each data point on our map, to give the user access to the data if they are interested in a specific point.

1. Import Leaflet.js and make the map canvas

Importing Leaflet.js is easy. We start with a boilerplate html webpage, and add the following two lines of text to our webpage's `<head>`.

```
<link
    rel="stylesheet"
    href="https://unpkg.com/leaflet@1.6.0/dist/leaflet.css"
    integrity="sha512-
xwE/Az9zrjBIphAcBb3F6JVqxf46+CDLwfLMHloNu6KEQCAWi6HcDUbeOfBIptF7tcCzusKFjFw2yuvEpD
L9wQ=="
    crossorigin=""
/>
<script
    src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
    integrity="sha512-
gZwIG9x3wUXg2hdXF6+rVkLF/0Vi9U8D2Ntg4Ga5I5BZpVkVxlJWbSQtXPSiUTtC0TjtGOmxa1AJPuV0CP
thew=="
    crossorigin=""
></script>
```

And just like that, we now have access to Leaflet.js functions in our code. This is accessable through the leaflet object, stored in a variable `L` and manipulated with a series of methods, as we will see in the coming sections.

Now that we have imported Leaflet.js, we need a map canvas. In Leaflet.js, a map canvas is simply a

than contains our map. To make this, add

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

in our html's `<body>` tag. A Leaflet subtlty is that this `<div>` must have a height to display properly. We'll go a step further by specifying a width and centering the div on our page by adding

```
<style>
#mapid {
margin: 0 auto;
height: 650px;
width: 1000px;
```

```
  }
</style>
```

to the `<head>` of our page. We're now ready to make our map canvas in our `#mapid`

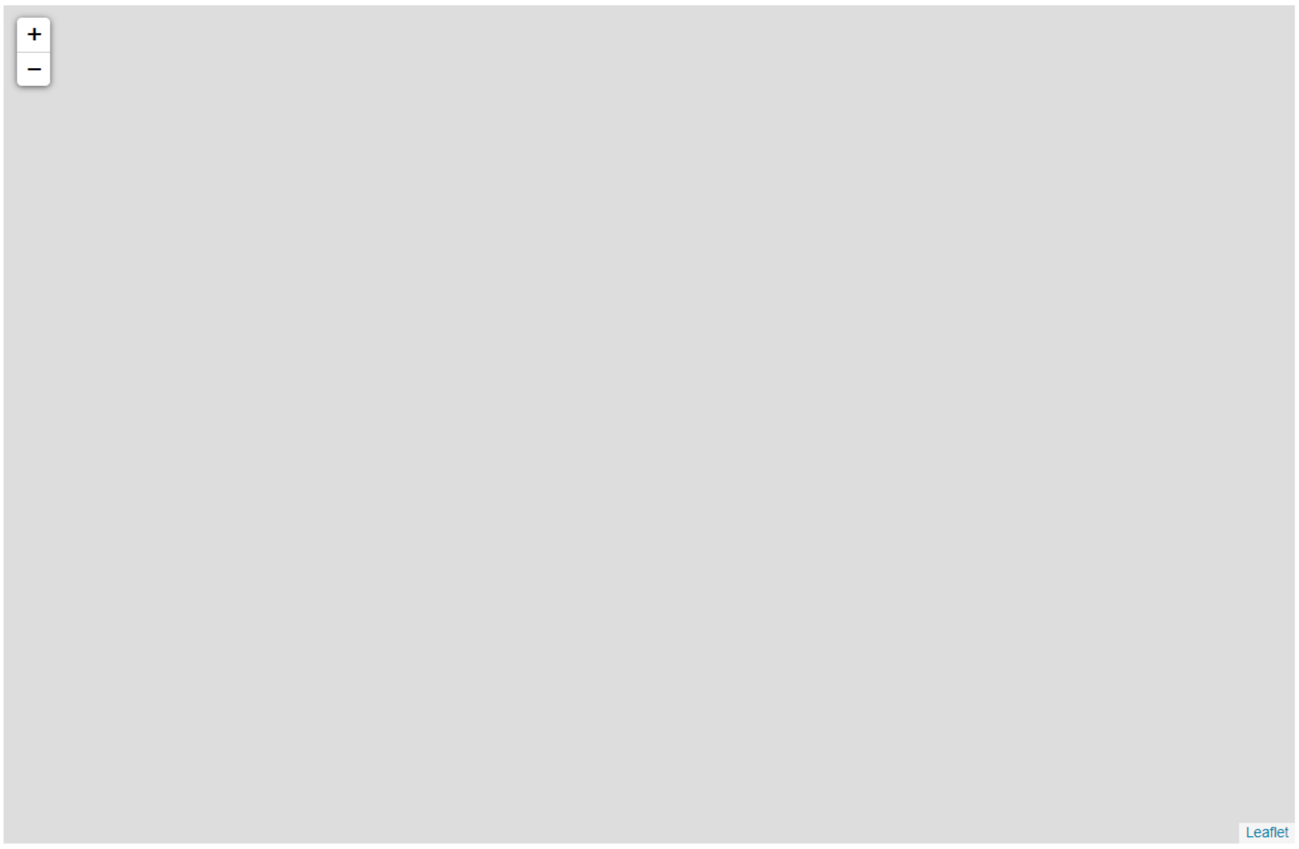
. To do this, make a new JavaScript file, and link to it with a script tag at the bottom of the body in our html page. For example, I created a file called `main.js`, and at the bottom of the `<body>` added a script tag to link to it:

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

Then, inside our new main.js file, I created the map canvas in our `#mapid` div by adding:

```
const myMap = L.map("mapid").setView([45, 0], 2);
```

Let's unpack what this line is doing. This line creates a variable called myMap, and stores the map canvas in it. The map canvas is created with the `L.map()` call, which takes the id of our `<div>` as it's argument. Also required is the chained `.setView()` method, which takes two arguments. The first argument is an array with a latitude and longitude that defines the centre view of the map, which here we have centered on 45° latitude and 0° longitude, i.e. over Europe, where the Coronavirus outbreak is currently especially bad. The second argument is the zoom level of the map, which ranges between 1 and 18. Here I've chosen 2 for a view of the whole world. Here is what our map looks like so far:

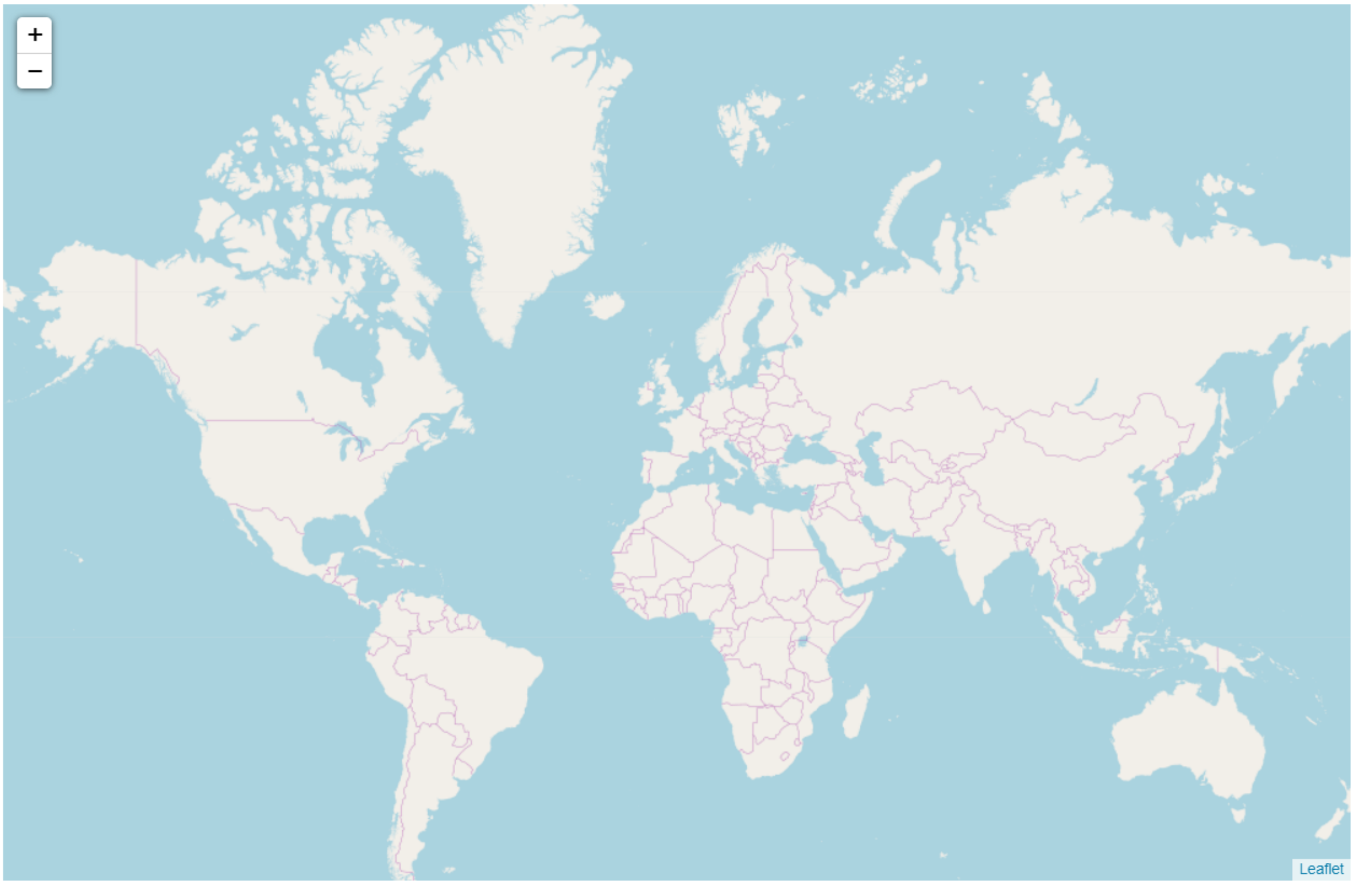


2. Add a base map

A basemap is the background to your data. It is normally a recognizable map - such as Google Maps, Bing Maps or Openstreetmaps background - which we add to our map to help the reader to orient themselves. Let's add an Openstreetmap background by adding the following code below our map canvas in our main.js file:

```
L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png").addTo(myMap);
```

The `L.tileLayer()` fetches the map tile from the URL provided to it. I found the URL by reading the Leaflet.js docs, which also handily explain the {z}, {x} and {y} found in the url correspond to the zoom level and x and y locations of the map view. Note the `.addTo()` method referencing the variable holding our map. We should now have a map that looks like this:



3. Add data

Now we have an interactive web map, lets actually add data to it. I've pulled some up to date data from Johns Hopkins University's repository, where they have lots of data avaliable as .csv files. I converted this into JSON using Python, but you could use libraries such as Papa Parse, or online tools such as csvjson. If you don't want to do this yourself, you can just grab a copy my JSON from my github here. For simplicity I imported it by adding

```
```

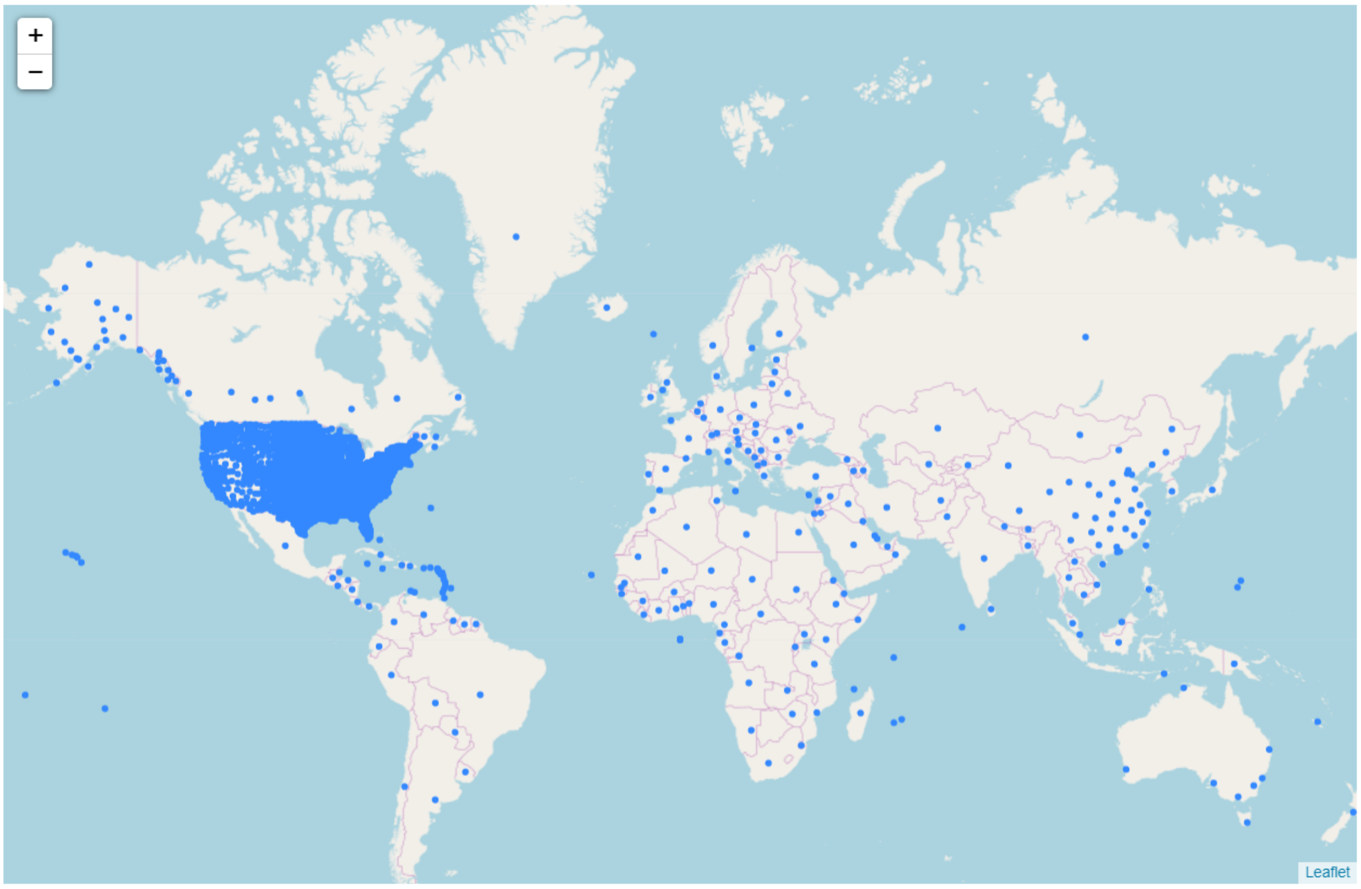
```
<script src="./data/03-25-2020.js"></script>
```

to my html file, after saving the JSON file in a folder called data. Then in main.js, I added:

```
data.forEach(d => {const marker = L.circle([d.Lat, d.Long_]).addTo(myMap)});
```

The data variable is accessable because we imported it in the preceeding script tag. In this line of code, we use forEach to loop over our data and add a circle to our map for each data record. L.circle takes an array with a latitude and a longitude as argumentments, which define the centre of the circle. The .addTo() method, unsuprisingly, adds each marker to our map.

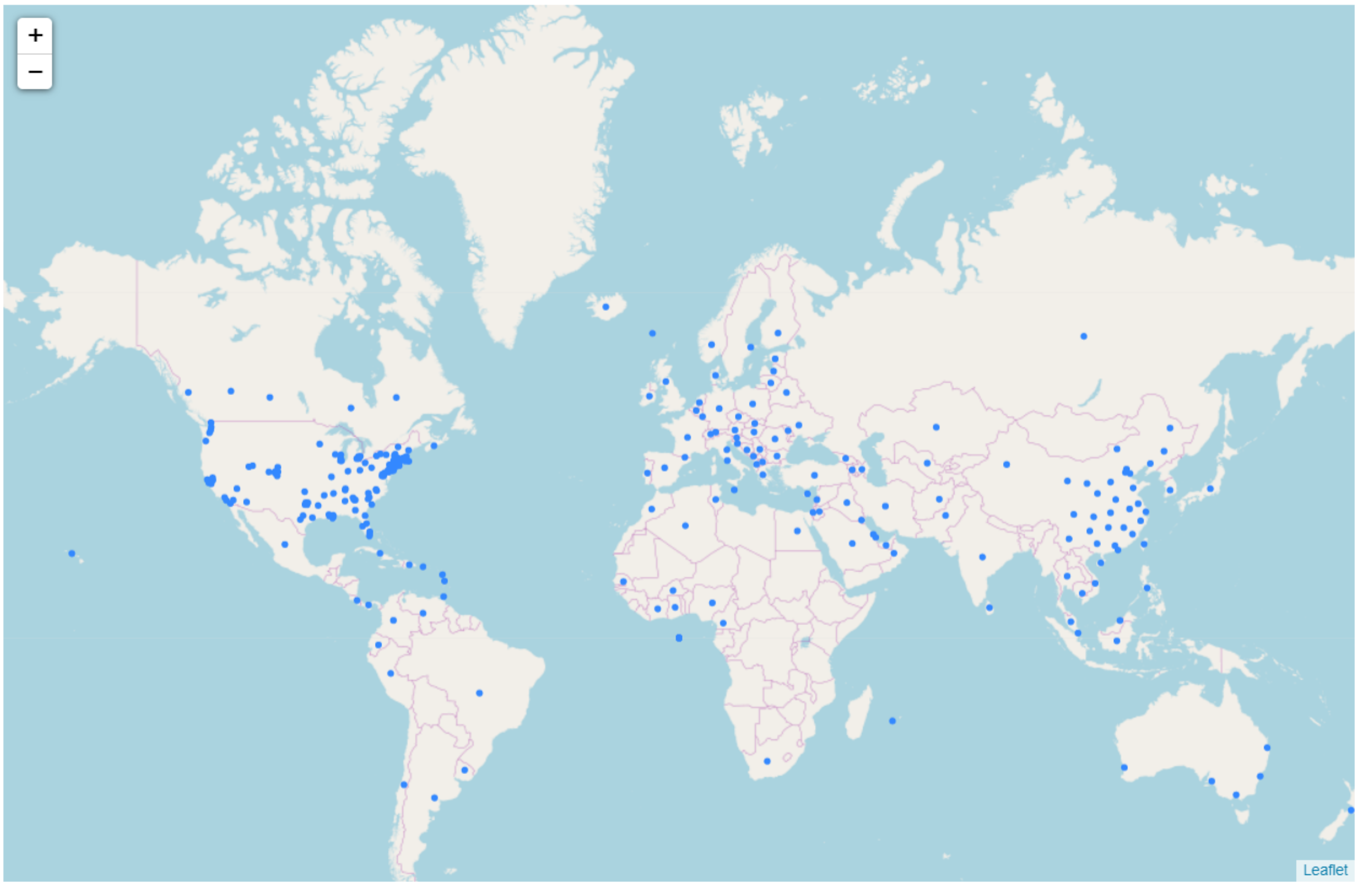
Our map should now look like this:



But here we have already run into a common problem in data science - having too much data to be able to see any realationships. The USA appears to have data split by state which leads to the whole country looking like a big blue blob. Let's cut down our data and filter out any data with less than 50 cases by changing our code to:

```
data.filter(d => d.Confirmed > 50).forEach(d => {const marker = L.circle([d.Lat,
d.Long_]).addTo(myMap)});
```

This cuts out a lot of datapoints:

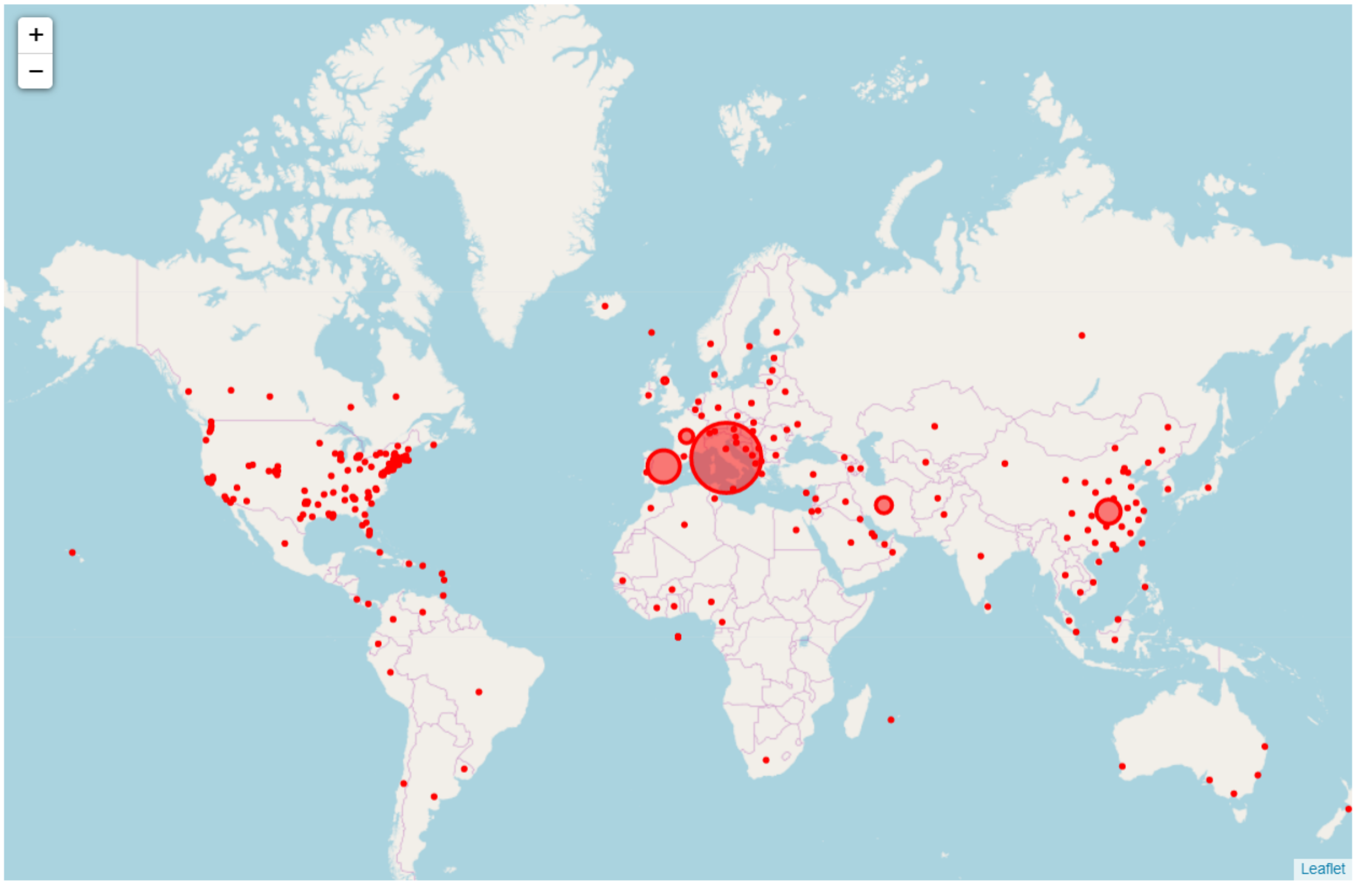


4. Style data

How to style the data depends on what you want to show. For this map we might be most interested in which countries have the largest number of deaths. One way of showing this is to scale the markers to be big if there are a large number of cases, and small if there are a small nubmer of cases. To help us see the data when there is a lot of it let's also make the circles semi-transparent. Finally for a dangerous disease, red is probably a more appropriate colour than the default blue.

```
data
  .filter(d => d.Confirmed > 50)
  .forEach(d => {
    const marker = L.circle([d.Lat, d.Long_], {
      color: "red",
      fillColor: "red",
      fillOpacity: 0.5,
      radius: d.Deaths * 100
    }).addTo(myMap);
```

Here, the important bit is that we are scaling the marker radius as a function of each datapoints Deaths key, in this case scaling it by 100.

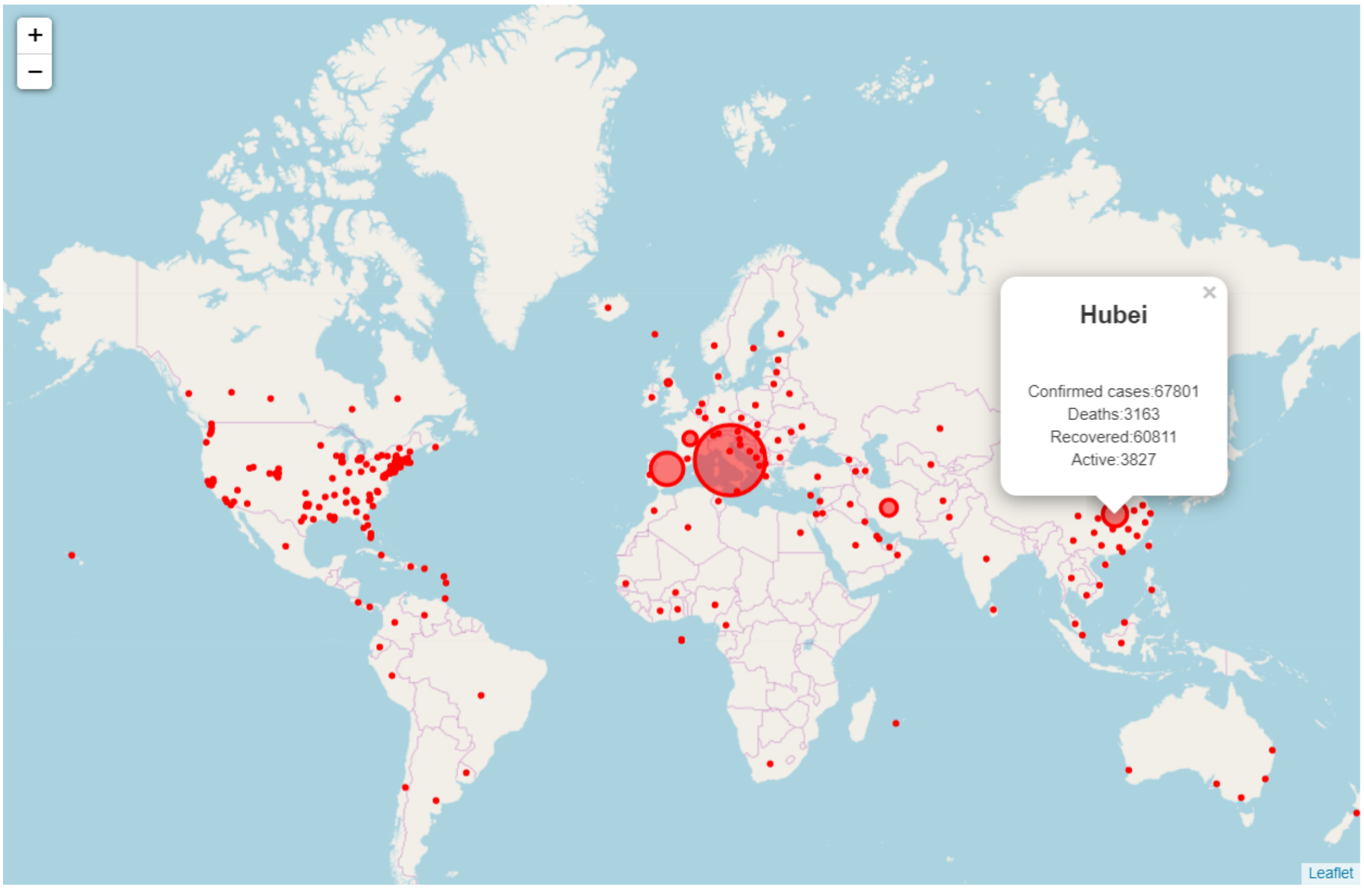


5. Add popups

Finally, popups on your web map give the user more to interact with. The visuals of our map are now pretty good, and the eye is naturally drawn to the large markers. However, that does increase the chance of an interested user clicking on those large circles. Let's add popups to each data point to allow interested users to dig into the data themselves.

```
data
  .filter(d => d.Confirmed > 50)
  .forEach(d => {
    const marker = L.circle([d.Lat, d.Long_], {
      color: "red",
      fillColor: "red",
      fillOpacity: 0.5,
      radius: d.Deaths * 100
    }).addTo(myMap);
    marker.bindPopup(
      `<h2>${d.Province_State ||
        d.Country_Region}</h2><p class="popup"><br>Confirmed cases:${
        d.Confirmed
      }<br>Deaths:${d.Deaths}<br>Recovered:${d.Recovered}<br>Active:${
        d.Active
      }</p>`
    );
  });
```

The bindPopup() method adds a popup to each marker as we loop over the data. Note the need for specifying `d.Province_State || d.Country_Region` to handle data records corresponding to individual states within a country rather than at the country level.



Further ideas

This basic map only scratches the surface of what you can do, both with web maps and with this dataset. You may have noticed a number of problems with the data - one immediatly obvious problem is found in the middle of the Atlantic Ocean, where you can find a point corresponding to New Jersey. Mistakes in the primary dataset are a common problem in data science. The normal approach to such problems is not to alter the data file, but to write code to check for errors and fix them before plotting. Another idea would be to check for duplicates in the data records. Another cool idea would be to make a time-series animation by making a map for each day of the outbreak. Or maybe a dashboard of daily updates. This dataset has many other stories to tell when combined with data from other sources - for example, you could start to look into how effective different government's policy is in countaining the outbreak by combining the numbers of cases and normalizing by population and money spent on combating the virus.

Finally as inspiration, here are a couple of examples of great web maps using Leaflet.js:

- http://donutholes.ch/
- http://www.triprisk.com.au/
- https://nuclearsecrecy.com/nukemap/
- https://birdmigrationmap.vogelwarte.ch/
- Lots of other cool examples on Leaflet.js's twitter page

And that's the end. Thanks for reading! Any questions and you can find me on twitter @murrayhoggett, or on linkedin. All of the code and data to follow this tutorial can be found on Github.