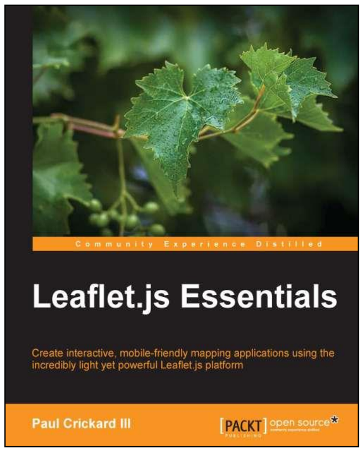
Leaflet.js Essentials

Paul Crickard III


Community Experience Distilled
Leaflet.js Essentials
Create interactive, mobile-friendly mapping applications using the incredibly light yet powerful Leaflet.js platform
Paul Crickard III
PACKT open source

Chapter No.1
"Creating Maps with Leaflet"

In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Creating Maps with Leaflet"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Paul Crickard III has been programming for over 15 years and has focused on GIS and geospatial programming for 7 years. He spent 3 years working as a planner at an architecture firm, where he combined GIS with Building Information Modeling (BIM) and CAD, and built web-based GIS applications to display and modify architectural data. He has given presentations to the New Mexico Public School Facilities Authority on BIM and GIS integration and on the use of GIS for Facility Planning, and the BIM505 Users Group on GIS as an interactive frontend to BIM and editing BIM data via web applications.

Currently, Paul works as a programmer analyst in Albuquerque, specializing in the design, maintenance, and the implementation of geospatial applications. He has written plugins and extensions for ArcMap and ArcGIS Explorer Desktop to utilize NoSQL databases and send data using the Advanced Message Queuing Protocol (AMQP). Paul has built applications using OpenLayers and Leaflet.js and is currently utilizing the ESRI JavaScript API in production.

Paul tries to incorporate Python in geospatial development wherever possible. From building plugins, toolboxes, and the Field Calculator functions in ArcMap to coding standalone desktop and web applications, pyshp is his favorite library for geospatial Python applications.

When he is not coding, Paul enjoys relaxing with his wife and son, cooking, and brewing beer.

Leaflet.js Essentials:

Making maps used to require an extensive knowledge of cartography, expensive software, and technical know-how. Today, there are numerous tools available, many of which are free, that have simplified the map-making process. This book is about using one such library, Leaflet.js.

Leaflet.js is a JavaScript library that although small, is packed with almost every feature you could need. If a feature is not available in the core library, it may be available as one of the many plugins that have become available. The largest map-making software vendor, Environmental Systems Research Institute (ESRI), has even released a plugin for Leaflet.js. If you are interested in making maps or in data visualization, Leaflet.js is the library to learn.

Whether you are looking to build simple maps or advanced mapping applications, this book will build on your JavaScript knowledge to help you reach your goal. This book was designed to be accessible to individuals who are new to map making and also to those who may know maps but are just learning to code.

What This Book Covers

Chapter 1, Creating Maps with Leaflet, walks you through the basics of making maps in Leaflet.js. You start by creating an HTML fi le with the minimum JavaScript code required to display a map. You are going to learn how to select different basemaps and providers and different basemap formats. Then, you will learn how to display geographic features such as points, polylines, and polygons.

Chapter 2, Mapping GeoJSON Data, introduces you to a geographic version of the JSON data format. You will learn how to create your own GeoJSON data as well as consume data from other sources. In this chapter, you will learn how to style the data and iterate through features to add pop ups.

Chapter 3, Creating Heatmaps and Choropleth Maps, moves away from simply displaying points and towards displaying the significance or comparisons of the data. It builds on what you have learned so far and teaches you how to use different plugins to create heatmaps. You will also learn how to use your knowledge of styling GeoJSON to create choropleth maps.

Chapter 4, Creating Custom Markers, guides you through the customization of the markers you use in your maps. You will learn how to draw your own image or modify an existing image to use it as a marker in your map. You will be introduced to several plugins that offer premade markers that are customizable. Also, you will learn how to animate markers and combine plugins for added effects.

Chapter 5, ESRI in Leaflet, opens up the most commonly used data formats and server endpoints in mapping. This chapter will teach you how to load shapefiles in your maps. You will also learn how to connect to an ESRI server that has an exposed REST service. Using the ESRI-Leaflet plugin, you will learn how to geocode and reverse geocode addresses, filter data from a server, and query by location.

Chapter 6, Leaflet in Node.js, Python, and C#, expands on everything you have learned in order to teach you how to build applications in other frameworks and languages. This chapter teaches you how to build both the frontend and the backend. You will build servers in JavaScript and Python. You will be introduced to NoSQL databases and AJAX to display and update data without refreshing your web page. Lastly, you will learn how to create a Windows desktop application by embedding Leaflet in C#.

1
Creating Maps with Leaflet

Web-based mapping has evolved rapidly over the last two decades, from MapQuest and Google to real-time location information on our phones' mapping apps. There have been open source projects to develop web-based maps in the past, such as MapServer, GeoServer, and OpenLayers. However, **Environmental Systems Research Institute** (**ESRI**) includes the Flex and Silverlight APIs; these create web-based maps from their ArcServer services.

Over the last few years, JavaScript has taken the online mapping world by storm. In 2013, there was a JS.geo conference. The library at the center of attention was Leaflet. This is a JavaScript library used to create interactive, web-based maps. With it, you can create a simple map in as little as three lines of JavaScript, or you can create complex, interactive, editable maps with hundreds of lines of code.

[You can find more information on Leaflet at `http://leafletjs.com`.]

This book assumes that you have a basic understanding of HTML and CSS, primarily of how to link external `.js` and `.css` files and how to name and size a `<div>` element. It also assumes that you have a working knowledge of JavaScript.

In this chapter, we will cover the following topics:

- Tile layers
- Vector layers
- Pop ups
- Custom functions / Responding to events
- Mobile mapping

Creating a simple basemap

To create a map with Leaflet, you need to do the following four things:

- Reference the JavaScript and **Cascading Style Sheet** (**CSS**) files
- Create a `<div>` element to hold the map
- Create a `map` object
- Add a tile layer (base layer)

Before we get into the details of building the map, let's set up an HTML file that we can use throughout the book. Open a text editor and enter the following HTML:

```
<!DOCTYPE html><html>
<head><title>Leaflet Essentials</title>
</head>
<body>
</body>
</html>
```

Save the file as `LeafletEssentials.html`. We will add to this file throughout the rest of the book.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Referencing the JavaScript and CSS files

There are two ways to load Leaflet into your code: you can either reference a hosted file or download a copy to your local machine and reference that copy. The next two sections will cover how you can set up your environment for a hosted copy or for a local copy.

Using a hosted copy

We will not be making any changes to the original CSS or JS files, so we will link to the hosted version.

In a text editor, open `LeafletEssentials.html`. In the `<head>` element, and after the `</title>` element, add the following code:

```
<link rel="stylesheet" href="http://cdn.leafletjs.com/leaflet-
0.7.3/leaflet.css"0.7.3 />
```

After the `<body>` tag, add the following code:

```
<script src="http://cdn.leafletjs.com/leaflet-
0.7.3/Leaflet"></script>
```

The links are standard HTML for `<link>` and `<script>`. Open either link in your browser and you will see the contents of the files.

Using a local copy

Using a local copy is the same as a hosted copy, except the path to the files is different. Download `Leaflet.js` from `http://leafletjs.com/download.html` and extract it to your desktop. If you downloaded `Leaflet-0.7.3.zip`, you should have a folder with the same name. In the folder, you will find a subfolder named `images` and the following three files:

- `Leaflet.css`: This is the cascading style sheet
- `Leaflet`: This is a compressed version of Leaflet
- `Leaflet-src.js`: This is the full version of Leaflet for developers

Add the following code in the `<head>` tag of `LeafletEssentials.html`:

```
<link rel="stylesheet"href="\PATH TO DESKTOP\leaflet-
0.7.3\leaflet.css" />
```

Add the following code in the `<body>` tag of `LeafletEssentials.html`:

```
<script src="\leaflet-0.7.3\Leaflet"></script>0.7.3Leaflet
```

You now have local references to the Leaflet library and CSS. We are using the `Leaflet` file because it is smaller and will load faster. As long as you do not need to add any code to the file, you can delete the `Leaflet-src.js` file.

Creating a <div> tag to hold the map

You need a place to put the map. You can accomplished this by creating a `<div>` tag with an ID that will be referenced by a `map` object. The `<div>` tag that is holding the map needs a defined height. The easiest way to give the tag a height is to use CSS in the `<div>` tag that you created. Add the following code to the `<body>` tag of `LeafletEssentials.html` after the `<script>` reference to the `Leaflet` file:

```
<div id="map" style="width: 600px; height: 400px"></div>
```

Style the `<div>` tag in the HTML file and *not* the `Leaflet.css` file. If you do this, the map `<div>` size will be global for every page that uses it.

Creating a map object

Now that you have the references and a place to put the map, it is time to start coding the map using JavaScript. The first step is to create a `map` object. The `map` class takes a `<div>` tag (which you created in the previous step) and `options`: `L.map(div id, options)`. To create a map object named `map`, add the following code after the `<script>` element in `LeafletEssentials.html`:

```
var map = L.map('map',{center: [35.10418, -106.62987],
zoom: 10
});
```

Alternatively, you can shorten the code using the `setView()` method, which takes the `center` and `zoom` options as parameters:

```
var map = L.map('map').setView([35.10418, -106.62987],10);
```

In the preceding code, you created a new instance of the `map` class and named it `map`. You may be used to creating new instances of a class using the keyword `new`; this is shown in the following code:

```
var map = new L.Map();
```

Leaflet implements factories that remove the need for the `new` keyword. In this example, `L.map()` has been given the `<div>` map and two options: `center` and `zoom`. These two options position the map on the screen with the latitude and longitude in the center of the `<div>` element and zoomed in or out at the desired level. The `center` option takes the `[latitude, longitude]` parameters, and `zoom` takes an integer; the larger the number, the tighter the zoom.

[It is good practice to always assign the `center` and `zoom` options. There is nothing worse than seeing a map of the world when all of the data is located Albuquerque, NM.]

Adding a tile layer

The last step to create your first map in Leaflet is to add a tile layer. A tile layer can be thought of as your basemap. It is the imagery that you will add points, lines, and polygons on top of later in the book. Tile layers are a service provided by a tile server. A tile server usually breaks up the layer into 256 x 256 pixel images. You retrieve the images needed based on your location and zoom through a URL that requests `/z/x/y.png`. Only these tiles are loaded. As you pan and zoom, new tiles are added to your map.

The tile layer, at a minimum, requires the URL to a tile server. In this book, we will use OpenStreetMap for our tile layer.

> You need to abide by the terms of service to use OpenStreetMap tiles. The TOS is available at `http://wiki.openstreetmap.org/wiki/Tile_usage_policy`.

The URL to the OpenStreetMap tile server is shown in the following code:

```
L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);
```

In the code, we provide the URL template to OpenStreetMaps. We also call the `addTo()` method so that the layer is drawn. We need to pass `L.map()` as a parameter to the `addTo()` function. We named our `L.map()` instance map in the previous section (`var map = L.map()`).

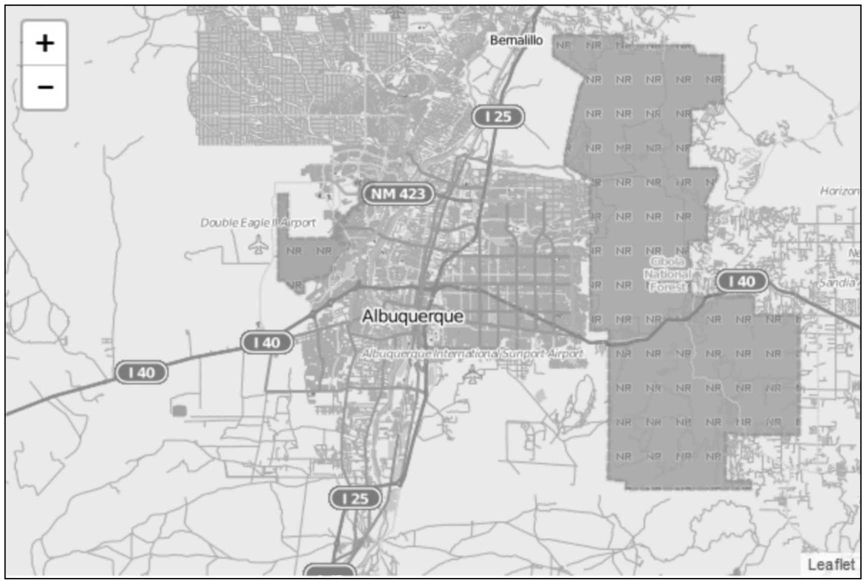
> Leaflet allows method chaining: the calling of multiple methods on an object at the same time. This is what we did when we put `.addTo(map)` at the end of the line, creating the instance of `L.tileLayer()`. The longer way of adding the layer to the map without chaining is to assign the instance to a variable and then call `addTo()` from the variable, as shown in the following code:

```
var x =
L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.
png');
x.addTo(map);
```

You now have a complete map that allows you to pan and zoom around the world. Your `LeafletEssentials.html` file should look like the following code:

```
<html>
<head><title>Leaflet Essentials</title>
<link rel="stylesheet" href="http://cdn.leafletjs.com/leaflet-
0.7.3/leaflet.css" />
</head>
<body>
<script src="http://cdn.leafletjs.com/leaflet-
0.7.3/Leaflet"></script>
<div id="map" style="width: 600px; height: 400px"></div>
<script>
var map = L.map('map',
{
center: [35.10418, -106.62987],
zoom: 10
});
L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addTo(map);
</script>
</body>
</html>
```

Even with liberal spacing, you were able to build a fully functional map of the world with pan and zoom capabilities in six lines of JavaScript. The following screenshot shows the finished map:



Tile layer providers

Now that you have created your first map, you are probably wondering how to change the tile layer to something else. There are several tile layer providers, some of which require registration. This section will present you with two more options: Thunderforest and Stamen. Thunderforest provides tiles that extend OpenStreetMap, while Stamen provides more artistic map tiles. Both of these services can be used to add a different style of basemap to your Leaflet map.

Thunderforest provides five tile services:

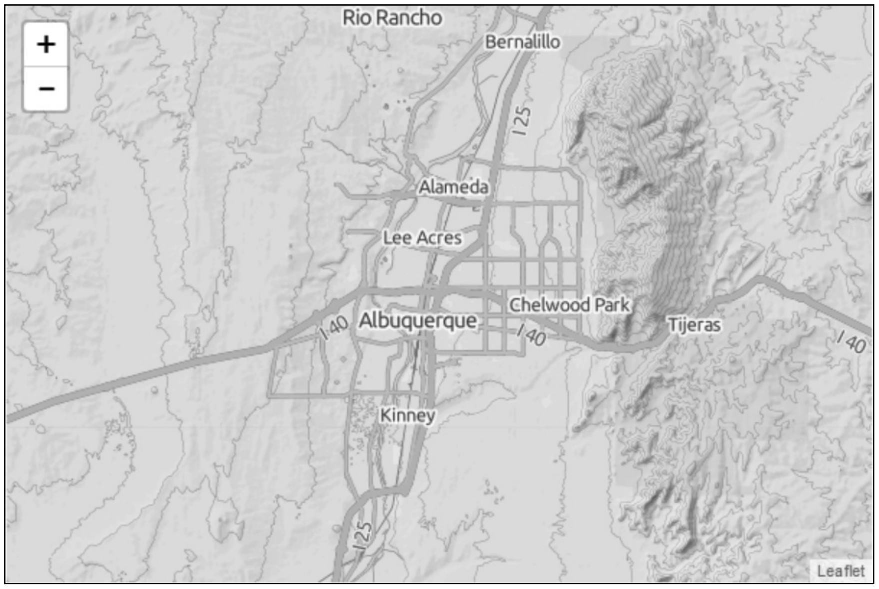
- OpenCycleMap
- Transport

- Landscape
- Outdoors
- Atlas (still in development)

To use Thunderforest, you need to point your tile layer to the URL of the tile server. The following code shows how you can add a Thunderforest tile layer:

```
var layer = new L.TileLayer('http://{s}.tile.thunderforest.com/
landscape/{z}/{x}/{y}.png');
map.addLayer(layer);
```

The preceding code loads the landscape tile layer. To use another layer, just replace `landscape` in the URL with `cycle`, `transport`, or `outdoors`. The following screenshot shows the Thunderforest landscape layer loaded in Leaflet:



Stamen provides six tile layers; however, we will only discuss the following three layers:

- Terrain (available in the United States only)
- Watercolor
- Toner

The other three are Burning Map, Mars and Trees, and Cabs & Crime. The Burning Map and Mars layers require WebGL, and Trees and Cabs & Crime are only available in San Francisco. While these maps have a definite wow factor, they are not practical for our purposes here.

Learn about the Stamen tile layers, including Burning Map, Mars and Trees, and Cabs & Crime, at `http://maps.stamen.com`.

Stamen requires you to follow the same steps as Thunderforest, but it includes an additional step of adding a reference to the JavaScript file. After the reference to your Leaflet file, add the following reference:

```
<script type="text/javascript"
src="http://maps.stamen.com/js/tile.stamen.js?v1.2.4"></script>
```

Instead of `L.TileLayer()`, Stamen uses `L.StamenTileLayer(tile set name)`. Replace the tile set name with `terrain`, `watercolor`, or `toner`. Lastly, add `addLayer()` to the map as shown in the following code:

```
var layer = new L.StamenTileLayer("watercolor");
map.addLayer(layer);
```

Stamen's tile layers are not your typical basemap layers; they are works of cartographic art.

Stamen has an online tool to edit map layers and save the output as an image. To create your own artistic map images, go to `http://mapstack.stamen.com`.

The following screenshot shows the Stamen watercolor layer loaded in Leaflet. As you zoom in, you will see more detail:

+
−
Leaflet | Map tiles by Stamen Design, under CC BY 3.0. Data by OpenStreetMap, under CC BY SA.

Adding a Web Mapping Service tile layer

Another type of tile layer that can be added to a Leaflet map is a **Web Mapping Service** (**WMS**) tile layer. WMS is a way to request and transfer map images over the Web through HTTP. It is an **Open Geospatial Consortium** (**OGC**) specification.

For detailed technical information on the WMS specification, see the OGC website: `http://www.opengeospatial.org/standards/wms`.

With an understanding of how to add tile layers, and having seen several examples, you may have noticed that none of the examples were of satellite imagery. The first WMS layer you will add to your map is the **United States Geological Survey** (**USGS**) Imagery Topo.

Like the `L.tileLayer()` function that we used previously, the `L.tileLayer.wms()` function takes a URL and a set of options as parameters. The following code adds the WMS layer to your map:

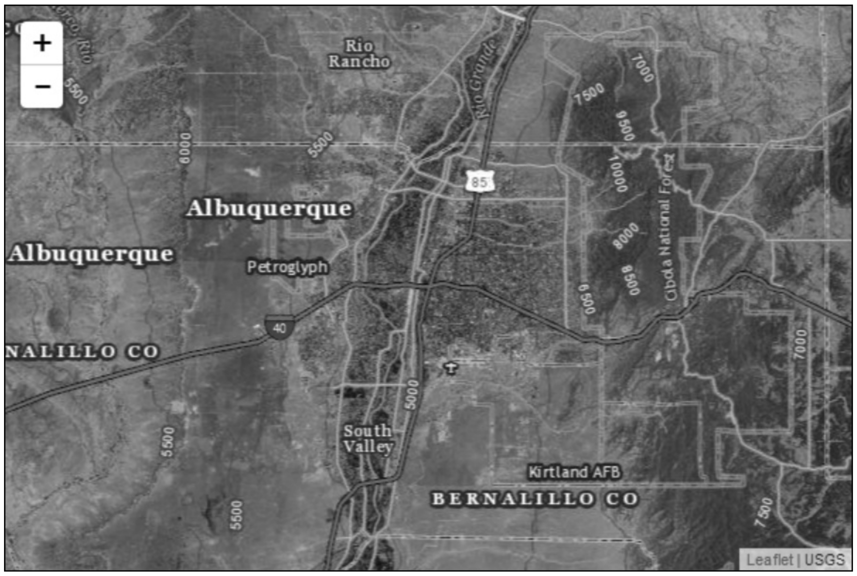
```
varusgs =
L.tileLayer.wms("http://basemap.nationalmap.gov/ArcGIS/services/US
GSImageryTopo/MapServer/WMSServer", {
layers:'0',
format: 'image/png',
transparent: true,
attribution: "USGS"
}).addTo(map);
```

The URL for the WMS was taken from the USGS website. You can find other WMS layers at `http://basemap.nationalmap.gov/arcgis/rest/services`. The options specified are the layer name, the format, the transparency, and the attribution. The layer name will be provided on the information page of the service you are connecting to. The format is an image, and the transparency is set to `true`. Since this layer covers the globe, and we are not putting any other layers underneath it, the transparency could be set to `false`. In the next example, you will see how setting the transparency to `true` allows another layer to become visible. Lastly, there is an attribution set to USGS. When you assign an attribution to a layer, Leaflet adds the text value in the lower-right corner of the map. It is important to use an attribution as it is similar to citing a source in text. If it is not your data, it is accepted practice to give credit where credit is due. Many times, it is also required by copyright. Since this layer is from the USGS, it is accredited in the attribution property of the layer.

The attribution value can contain hyperlinks, as shown in the following code:

```
attribution: "<a
href='http://basemap.nationalmap.gov/arcgis/rest/
service
s'>USGS</a>"
```

Insert the WMS layer code into `LeafletEssentials.html`, and you should now have a map with satellite imagery. The following screenshot shows the satellite imagery loaded into Leaflet:



Multiple tile layers

In the previous example, you added a WMS layer and set the transparency to `true`. The reason you need to do this is because you can add multiple tile layers on top of each other, and with the transparency set to `true`, you will be able to see them all at the same time. In this example, you need to add the **National Weather Service** (**NWS**) radar mosaic WMS on top of the USGS satellite imagery.

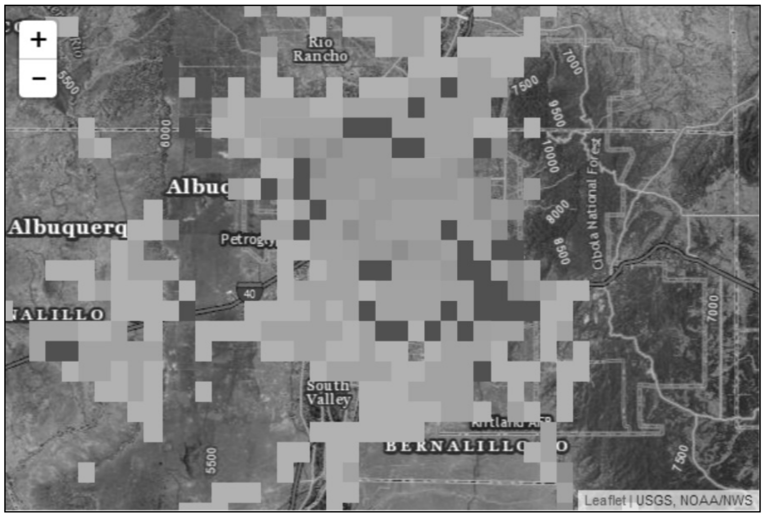
The **National Oceanic and Atmospheric Administration** (**NOAA**) provides a list of several WMS layers; they are available at the following link:

`http://nowcoast.noaa.gov/help/mapservices.shtml?name=mapservices`

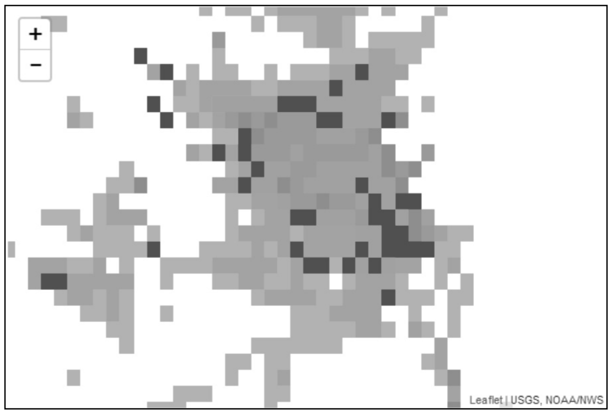
The adding of extra WMS layers follows the same format as the previous example, but with a different URL, layer name, and attribution. Add the following code after the code for the satellite imagery in `LeafletEssentials.html`:

```
Varnexrad =
L.tileLayer.wms("http://nowcoast.noaa.gov/wms/com.esri.wms.Esrimap
/obs", {
layers: 'RAS_RIDGE_NEXRAD',
format: 'image/png',
transparent: true,
attribution: "NOAA/NWS"
}).addTo(map);
```

This code adds the NOAA WMS layer for the NWS radar mosaic. Note that the URL and layer have changed and the attribution is set to `NOAA/NWS`. The `RAS_RIDGE_NEXRAD` layer is a grid that displays values when they begin to exist. The name of the layer can be found on the NOAA website; you are not expected to remember that `RAS_RIDGE_NEXRAD` is the weather radar layer. There are large portions of the map with no data, and since we set the transparency to `true`, these blank spaces allow the satellite imagery to become visible. Your map should now show the satellite imagery with the radar mosaic overlaid, as in the following screenshot:



If you set the transparency to `false`, you allow the layer to draw on the entire map. Areas with no data are displayed as white squares and cover the satellite imagery underneath, as shown in the following screenshot:



WMS layers do not need to serve as base layers only; they can be used as additional data. This was shown in the previous example where you overlaid the radar on the satellite imagery. In this example, you used a satellite image. You can also use the OpenStreetMap tile layer from the first map. Again, just set the transparency to `true`. WMS layers can be added just like points, lines, and polygons, which is discussed in the following sections.

Adding data to your map

So far, you have learned how to add tile layers to a map. In the previous example, you added a WMS layer on top of a base tile layer. Now, you will learn how to draw your own layers that need to be added on top of a tile layer. The three geometric primitives of vector data that you can add to a map are often referred to as points, lines, and polygons.

In this section, you will learn how to add markers, polylines, and polygons to your map.

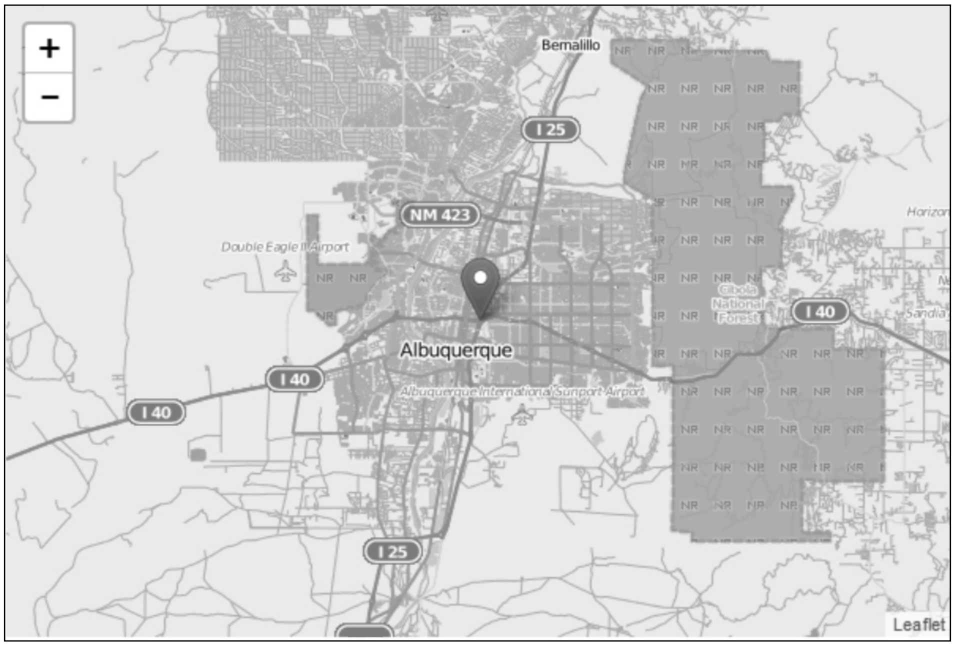
Points

So far, your map is not that interesting. You often draw a map to highlight a specific place or point. Leaflet has a `Point` class; however, it is not used to simply add a point on the map with an icon to specify the place. In Leaflet, points are added to the map using the `Marker` class. At minimum, the `Marker` class requires a latitude and longitude, as shown in the following code:

```
Var myMarker = L.marker([35.10418, -106.62987]).addTo(map);
```

> You can create a marker by simply calling `L.marker([lat,long]).addTo(map);`, but assigning the marker to a variable will allow you to interact with it by name. How do you delete a specific marker if it does not have a name?

In the preceding code, you created a marker at point `[35.10418, -106.62987]`, and then, as with the tile layer, you used the `addTo(map)` function. This created a marker icon at the specified latitude and longitude. The following screenshot shows the marker on the map:



The preceding example is a simplified, and almost useless, marker. The `Marker` class has options, events, and methods that you can call to make them more interactive and useful. You will learn about methods—specifically the `bindPopup()` method— and events later in this chapter.

There are 10 options you can specify when creating a marker, as follows:

- `icon`
- `clickable`
- `draggable`
- `keyboard`
- `title`
- `alt`
- `zIndexOffset`
- `opacity`
- `riseOnHover`
- `riseOffset`

The options `clickable`, `draggable`, `keyboard`, `zIndexOffset`, `opacity`, `riseOnHover`, and `riseOffset` are all set to a default value. In *Chapter 4, Creating Custom Markers*, you will learn about the `icon` option in detail. Two options that you should set are `title` and `alt`. The `title` option is the tooltip text that will be displayed when you hover over the point with the cursor, and the `alt` option is the alternative text that is read using screen readers for accessibility. These options are used in the following code:

```
varmyMarker = L.marker([35.10418, -106.62987],
{title:"MyPoint",alt:"The Big I",draggable:true}).addTo(map);
```

The code extends the original marker example by adding a title and alt text and making the marker draggable. You will use the `draggable` options with an event in the last section of this chapter. The options are set the same as when we created our map instance; use curly braces to group the options, and separate each option with a comma. This is how options will be set for all objects.

Polylines

The first vector layer you will learn to create is aLine. In Leaflet, you will use the `Polyline` class. A polyline can represent a single line segment or a line with multiple segments. Polylines and polygons extend the `path` class. You do not call `path` directly, but you have access to its methods, properties, and events. To draw a polyline, you need to provide at least a single longitude and latitude pair. The option for a polyline is set as default, so you need not specify any values unless you want to override the default. This is shown in the following code:

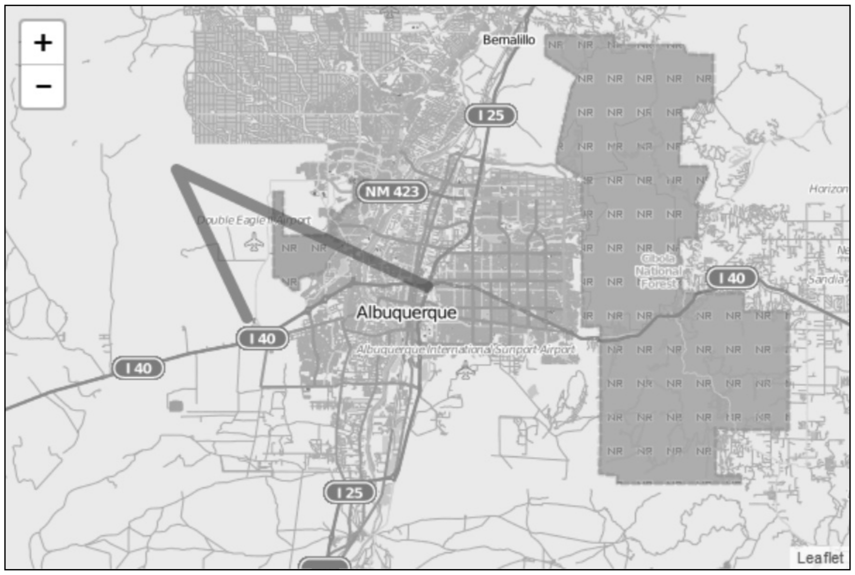
```
var polyline = L.polyline([[35.10418, -106.62987],[35.19738, -
106.875]], {color: 'red',weight:8}).addTo(map);
```

In this example, the polyline is `red` and has a weight of `8`. The `weight` option defaults to `5`. If you want a thicker line, increase the number. For a thinner line, decrease the number. To add more segments to the line, just add additional latitude and longitude values as shown in the following code:

```
var polyline = L.polyline([[35.10418, -106.62987],[35.19738, -
106.875],[35.07946, -106.80634]], {color:
'red',weight:8}).addTo(map);
```

You need to first provide a latitude and longitude pair because a line consists of at least two points. Afterwards, you can declare additional latitudes and longitudes to extend your line.

The following screenshot shows the polyline added to the map:



Polygons

A polygon is a polyline that is closed. Polygons tend to be classified by the number of sides, as follows:

- Triangle (3)
- Hexagon (6)
- Octagon (8)

Leaflet has a class for drawing two common polygons: a circle and a rectangle. When drawing a polygon, you will specify a minimum of three coordinates. A triangle is the simplest polygon that you can draw. That is why you need to provide at least three points. You do not need to specify the starting point at the end of the list. Leaflet will automatically close the polygon for you. To draw a polygon, simply copy the code for the polyline with three points and change the class to `L.polygon()`, as shown in the following code:

```
var polygon = L.polygon([[35.10418, -106.62987],[35.19738, -
106.875],[35.07946, -106.80634]], {color:
'red',weight:8}).addTo(map);
```

Since Leaflet automatically closes the polygon, our three-point polyline can become a polygon. Since `polyline` and `polygon` inherit from `path`, the options `color` and `weight` apply to both. You will notice that `color` and `weight` refer to the outline of the polygon. Two options that you will find useful when drawing polygons are `fillColor` and `fillOpacity`:

```
var polygon = L.polygon([[35.10418, -106.62987],[35.19738, -
106.875],[35.07946, -106.80634]], {color:
'red',weight:8,fillColor:'blue',fillOpacity:1}).addTo(map);
```

The preceding code draws a `red` triangle with a weight of `8`. The additional options of `fillColor` and `fillOpacity` are set to `blue` and `1`. The fill color of a polygon will be set to the default if no `fillColor` option is selected. You only need to use `fillColor` if you want a different fill color than the outline.

> Opacity is a value between `0` and `1`, where `0` is 100 percent opacity and `1` is no opacity (solid).

The following screenshot shows the red triangle with a blue fill added to the map:

Bemalillo
I 25
NM 423
Albuquerque
I 40
Albuquerque International Sunport Airport
Cibola National Forest
Horizon
Sandia
Leaflet

Rectangles and circles

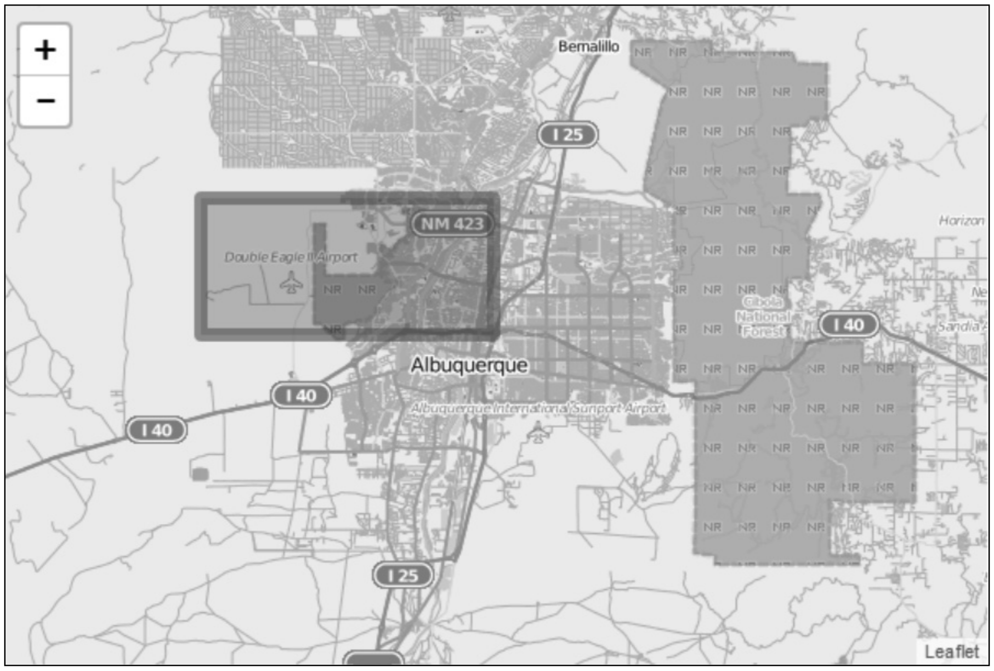
Circles and rectangles are common polygons that have built-in classes in Leaflet. You can also draw them manually using polygon and by specifying all of the line segments, but that would be a difficult route to take.

Rectangles

To create a rectangle, you need an instance of the class `L.rectangle()` with the latitude and longitude pair for the upper-left corner and lower-right corner as a parameter. The class extends `L.polygon()`, so you have access to the same options, methods, and events:

```
var myRectangle = L.rectangle([[35.19738, -106.875],[35.10418, -
106.62987]], {color: "red", weight:
8,fillColor:"blue"}).addTo(map);
```

The preceding code uses the first two points in the polyline and triangle, but in reverse order (upper left and lower right). The options are the same as the polygon, but with opacity removed. The following screenshot shows the rectangle added to the map:

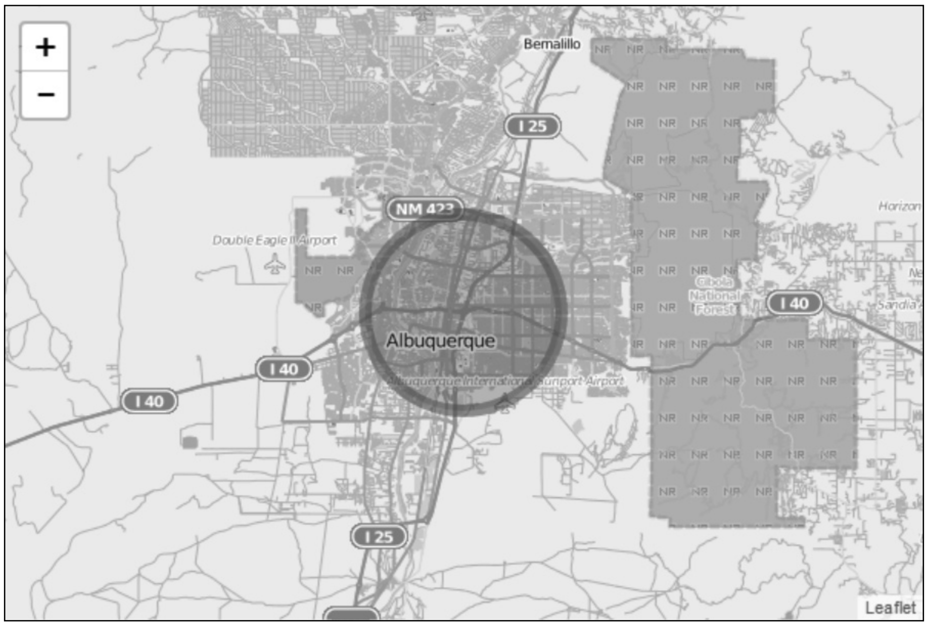


Circles

To create a circle, you need an instance of `L.circle()` with the center point and a radius (in meters) as parameters. You can specify the same options as you used in your rectangle because the `circle` class extends the `path` class. This is shown in the following code:

```
L.circle([35.10418, -106.62987], 8046.72,{color: "red", weight:
8,fillColor:"blue"}).addTo(map);
```

The preceding code specifies the center point, a radius of 5 miles (`8046.72` meters), and the same options as the rectangle in the previous example. The following screenshot shows the circle added to the map:



MultiPolylines and MultiPolygons

In the previous examples, you created each polyline and polygon as its own layer. When you start creating real data, you will find that you want multiple polylines or polygons on a single layer. For starters, it is more realistic, and it also makes it possible to deal with similar features as a single entity. If you want to map parks and bike trails on a single map, it makes sense to add the parks as MultiPolygon and the bike trails as MultiPolyline. Then, you can provide the user with the option of turning either layer on or off.

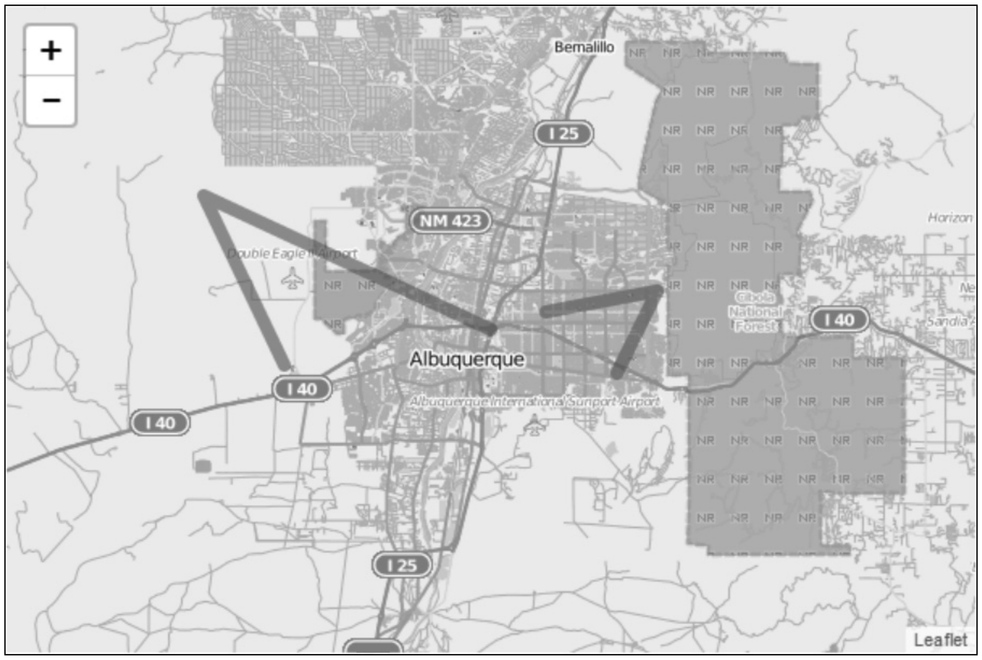
Bracketing for MultiPolylines and MultiPolygons can get confusing. You need brackets to hold the MultiPolyline or MultiPolygon, brackets for each polyline or polygon, and brackets for each latitude and longitude.

MultiPolylines

Creating a MultiPolyline is functionally the same as a single polyline, except that you pass multiple longitudes and latitudes; a set for each polygon. This is shown in the following code:

```
var multipolyline = L.multiPolyline([[[35.10418, -
106.62987],[35.19738, -106.875],[35.07946, -
106.80634]],[[35.11654, -106.58318],[35.13142, -
106.48876],[35.07384, -106.52412]]],{color:
'red',weight:8}).addTo(map);
```

In the preceding code, the first polyline is the same as the polyline example. A second polyline is added, and the options are also the same as the first polyline example. The following screenshot shows the MultiPolyline added to the map:

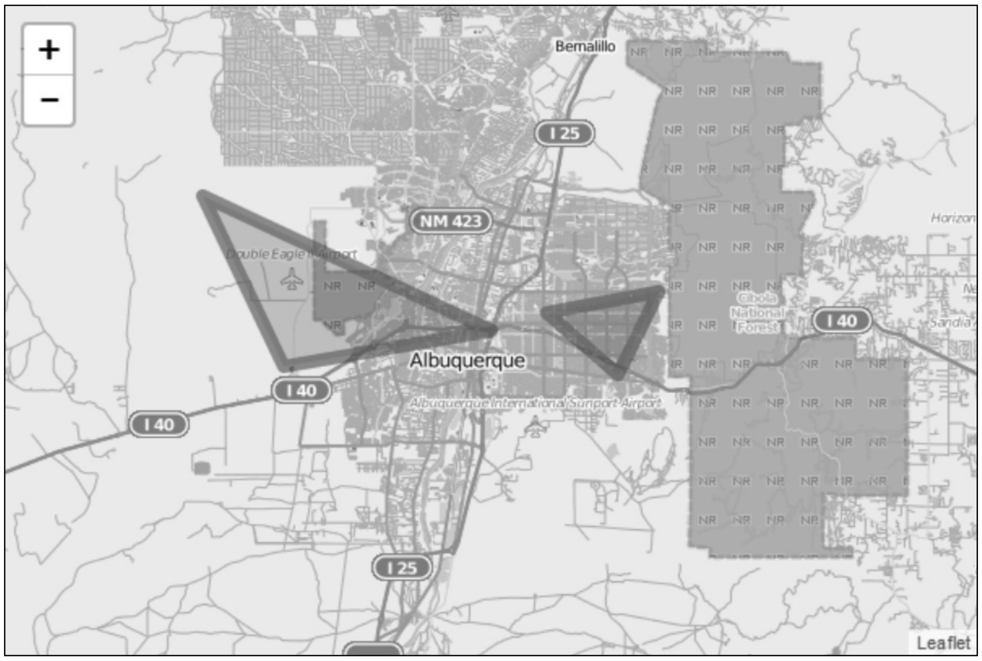


MultiPolygons

Creating a MultiPolygon is the same as creating a MultiPolyline. Since Leaflet will automatically close the polyline, as long as our polylines have three or more points, we can use them. This is shown in the following code:

```
var multipolygon = L.multiPolygon([[[35.10418, -
106.62987],[35.19738, -106.875],[35.07946, -
106.80634]],[[35.11654, -106.58318],[35.13142, -
106.48876],[35.07384, -106.52412]]],{color:
'red',weight:8}).addTo(map).bindPopup("We are the same layer");
```

In the preceding code, you can see that the parameters used are identical to those used in the MultiPolyline example earlier. When we create a MultiPolygon or MultiPolyline, the options will apply to every polygon or polyline in the collection. This means that they all have to be the same color, weight, opacity, and so on. There is a new method in the preceding code: `.bindPopup("We are the same layer")`. MultiPolygons and MultiPolylines also share the same pop up. Pop ups will be discussed later in this chapter. Also, note the use of method chaining in the line `L.multiPolygon().addTo().bindPopup()`. The following screenshot shows the MultiPolygon added to the map:



Groups of layers

MultiPolyline and MultiPolygon layers allow you to combine multiple polylines and polygons. If you want to create group layers of different types, such as a marker layer with a circle, you can use a layer group or a feature group.

The layer group

A layer group allows you to add multiple layers of different types to the map and manage them as a single layer. To use a layer group, you will need to define several layers:

```
var marker=L.marker([35.10418, -106.62987]).bindPopup("I am a
Marker");
var marker2=L.marker([35.02381, -106.63811]).bindPopup("I am
Marker 2");
var polyline=L.polyline([[35.10418, -106.62987],[35.19738, -
106.875],[35.07946, -106.80634]], {color:
'red',weight:8}).bindPopup("I am a Polyline");
```

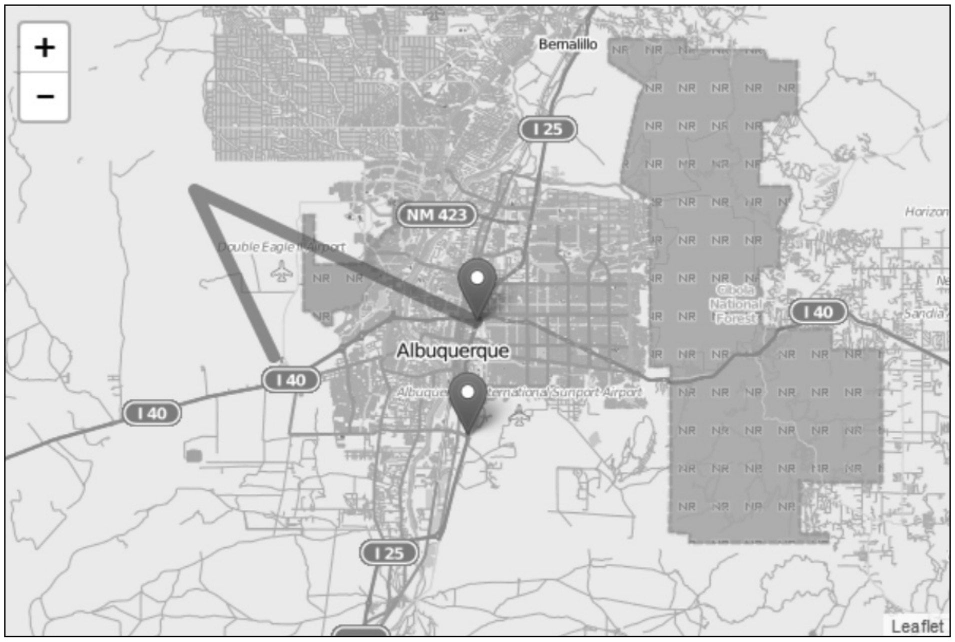
The preceding code creates two markers and a polyline. Note that you will not use the `addTo(map)` function after creating the layers, like you did in the previous examples. You will let the layer group handle adding the layer to the map. A layer group requires a set of layers as a parameter:

```
var myLayerGroup=L.layerGroup([marker, polyline]).addTo(map);
```

In the previous code, an instance of `L.layerGroup()` was created as `myLayerGroup`. The layers passed as a parameter were `marker` and `polyline`. Finally, the layer group was added to the map. The earlier code shows three layers, but only two were added to the layer group. To add layers to a layer group without passing them as a parameter during creation, you can use the layer group `addLayer()` method. This method takes a layer as a parameter, as shown in the following code:

```
myLayerGroup.addLayer(marker2);
```

Now, all three layers have been added to the layer group and are displayed on the map. The following screenshot shows the layer group added to the map:



If you want to remove a layer from the layer group, you can use the `removeLayer()` method and pass the layer name as a parameter:

```
myLayerGroup.removeLayer(marker);
```

If you remove a layer from the group, it will no longer be displayed on the map because the `addTo()` function was called for the layer group and not the individual layer. If you want to display the layer but no longer want it to be part of the layer group, use the `removeLayer()` function, as shown in the preceding code, and then add the layer to the map as shown in the earlier examples. This is shown in the following code:

```
marker.addTo(map);
```

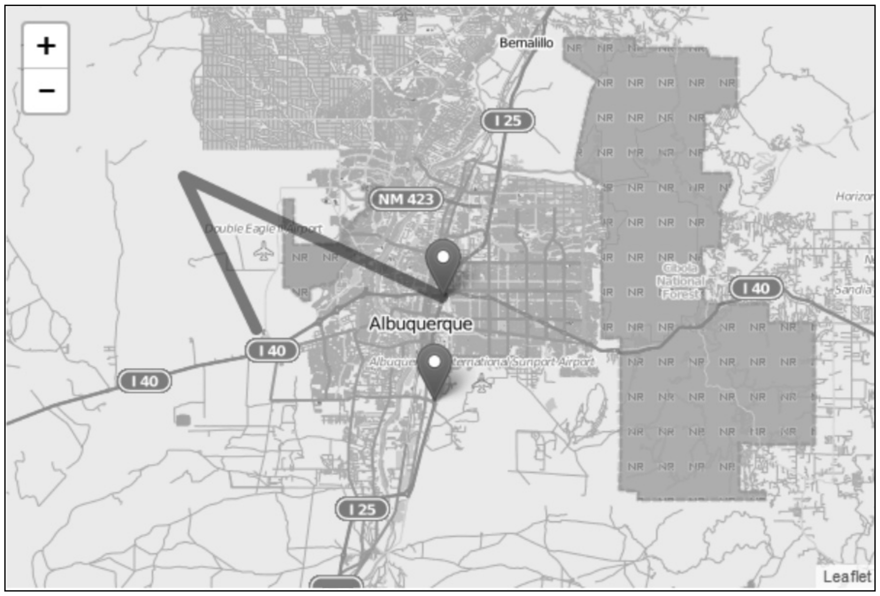
All style options and pop ups need to be assigned to the layer when it is created. You cannot assign a style or pop ups to a layer group as a whole. This is where feature groups can be used.

Feature groups

A feature group is similar to a layer group, but extends it to allow mouse events and includes the `bindPopup()` method. The constructor for a feature group is the same as the layer group: just pass a set of layers as a parameter. The following code displays an example of a feature group:

```
VarmyfeatureGroup=L.featureGroup([marker, marker2, polyline])
      .addTo(map).setStyle({color:'purple',opacity:.5})
      .bindPopup("We have the same popup because we are a group");
```

In the preceding code, the layers added are the same three that you added in the layer group. Since the feature group extends the layer group, you can assign a style and pop up to all of the layers at once. The following screenshot shows the feature group added to the map:



When you created the polyline in the previous example, you set the color to `red`. Note now that since you passed style information to the feature group by setting the color to `purple`, the polyline took the information from the feature group and discarded its original settings. If you removed the polyline from the feature group, it will be removed from the map as well. If you try to add the polyline to the map using `addTo()`, as in the previous examples, it will still be purple and have the new pop up. The markers are still blue even though you passed style information to the feature group. The `setStyle()` method only applies to layers in the feature group that have a `setStyle()` method. Since a polyline extends the `path` class, it has a `setStyle()` method. The markers do not have a `setStyle()` method, so their color did not change.

Pop ups

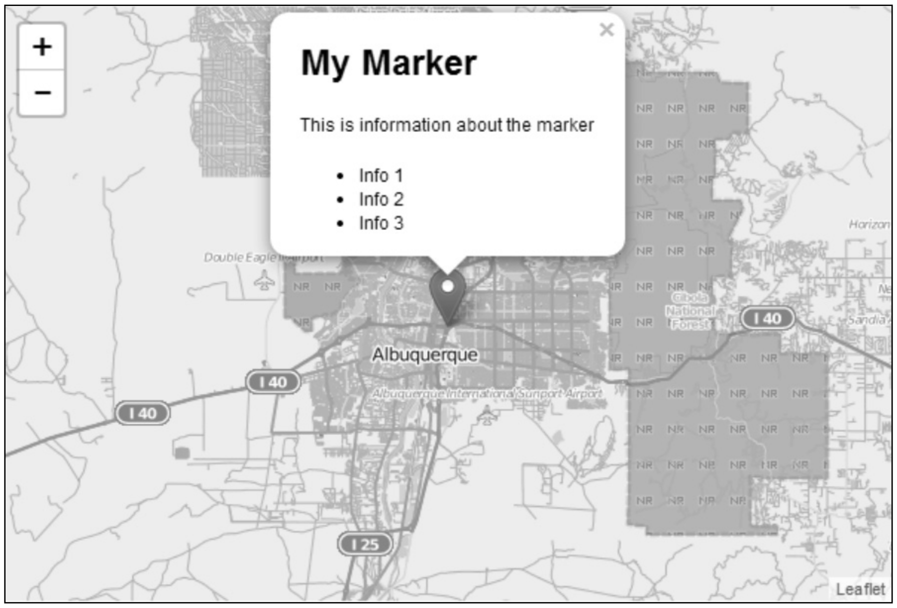
The last few examples introduced pop ups. A pop up provides a way to make your layers interactive or provides information to the user. The simplest way to add a pop up to a marker, polyline, or polygon is to use the `bindPopup()` method. This method takes the contents of the pop up as a parameter. Using the `marker` variable we created earlier, we bind the pop up to it with the following code:

```
marker.bindPopup("I am a marker");
```

The `bindPopup()` method allows you to enter HTML as the content. This is shown in the following code:

```
marker.bindPopup("<h1>My Marker</h1><p>This is information about
the marker</p><ul><li>Info 1</li><li>Info 2</li><li>Info
3</li></ul>")
```

The ability to use HTML in a pop up comes in handy when you have a lot of details to add. It allows the use of images and links in pop ups. The following screenshot shows the HTML-formatted pop up added to a marker on the map:



You can also create an instance of the `popup` class and then assign it to multiple objects:

```
var mypopup = L.popup({keepInView:true,closeButton:false})
.setContent("<h1>My Marker</h1><p>This is information about the
marker</p><ul><li>Info 1</li><li>Info 2</li><li>Info 3</li></ul>");
marker.bindPopup(mypopup);
marker2.bindPopup(mypopup);
```

In the preceding code, you create an instance of the `L.popup()` class and assign it to the variable `mypopup`. Then, you can call the `bindPopup()` method on `marker` and `marker2` with `mypopup` as the parameter. Both markers will have the same pop up content and options.

In the last section of this chapter, you will learn how to create a function that allows you to create a pop up with options and pass the content as a parameter.

Mobile mapping

The maps you have made so far have been tested on the desktop. One of the benefits of mapping in JavaScript is that mobile devices can run the code in a standard web browser without any external applications or plugins. Leaflet runs on mobile devices, such as iPhone, iPad, and Android devices. Any web page with a Leaflet map will work on a mobile device without any changes; however, you probably want to customize the map for mobile devices so that it works and looks like it was built specifically for mobile.

Lastly, the `L.map()` class has a `locate()` method, which uses the W3C Geolocation API. The Geolocation API allows you to find and track a user's location using the IP address, the wireless network information, or the GPS on a device. You do not need to know how to use the API; Leaflet handles all of this when you call `locate()`.

HTML and CSS

The first step in converting your Leaflet map to a mobile version is to have it display properly on mobile devices. You can always tell when you open a website on your phone whether the developer took the time to make it mobile-accessible. How many times have you been on a website where the page loads and all you can see is the top-left corner, and you have to zoom around to read the page. It is not a good user experience. In `LeafletEssentials.html` in the `<head>` tag after the `<link>` tag for the CSS file, add the following code:

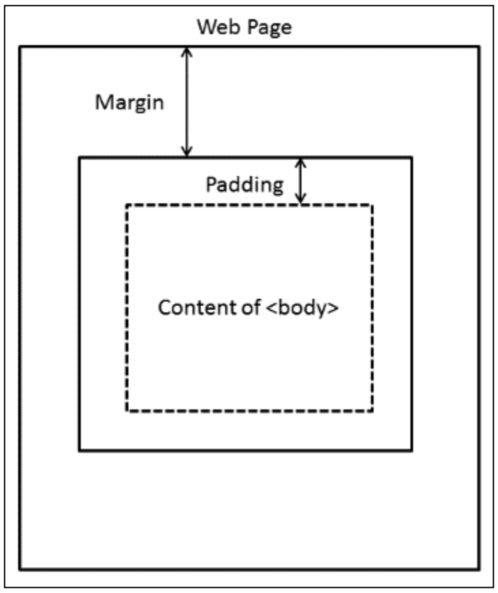
```
<style>

body{
padding: 0;
margin: 0;
    }
html, body, #map {
height: 100%;
        }
</style>
```

In the preceding CSS code, you set the `padding` and `margin` values to `0`. Think of a web page as a box model, where each element exists in its own box. Each box has a margin, which is the space between it and other boxes, and also padding, which is the space between the content inside the box and the box border (even if a border is not physically drawn). Setting the `padding` and `margin` values to `0` makes the `<body>` content fit to the size of the page. Lastly, you set the `height` value of the `<html>`, `<body>`, and `<div id = 'map'>` elements to `100%`.

> In CSS, `#` is the ID selector. In the code, `#map` is telling us to select the element with the `id = 'map'` line. In this case, it is our `<div>` element that holds the map.

The following diagram shows an overview of the settings for the web page:



The last step is to add the following code in the `<head>` section and after the `</title>` element:

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0, maximum-scale=1.0, user-scalable=no">
```

The preceding code modifies the viewport that the site is seen through. This code sets the viewport to the width of the device and renders it by a ratio of 1:1. Lastly, it disables the ability to resize the web page. This, however, does not affect your ability to zoom on the map.

Creating a mobile map with JavaScript

Now that you have configured the web page to render properly on mobile devices, it is time to add the JavaScript code that will grab the user's current location. For this, perform the following steps:

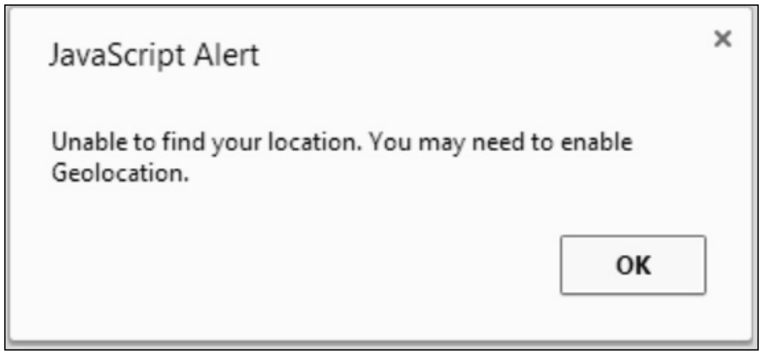
1. Create the map instance, but do not use `setView`:
   ```
   var map = L.map('map');
   ```
2. Add a tile layer:
   ```
   L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png').addT
   o(map);
   ```
3. Define a function to successfully find the location:
   ```
   Function foundLocation(e){}
   ```
4. Define a function to unsuccessfully find the location:
   ```
   functionnotFoundLocation(e){}
   ```
5. Add an event listener for `foundLocation()` and `notFoundLocation()`:
   ```
   map.on('locationfound', foundLocation);
   map.on('locationerror', notFoundLocation);
   ```
6. Use locate() to set the map view:
   ```
   map.locate({setView: true, maxZoom:10});
   ```

The code creates the map and adds a tile layer. It then skips over the functions and event listeners and tries to locate the user. If it is able to locate the user, it runs the code in `foundLocation()` and sets the view to the latitude and longitude of the user. If it does not locate the user, it executes the code in `notFoundLocation()` and displays a zoomed-out world map.

To make this example more usable, add the following code to `notFoundLocation()`:

```
function notFoundLocation(e){
alert("Unable to find your location. You may need to enable
Geolocation.");}
```

The `alert()` function creates a pop up in the browser with the message passed as a parameter. Anytime that the browser is unable to locate the user, they will see the following message. While some devices do not have location capabilities, at times, they need to be allowed in their security settings:

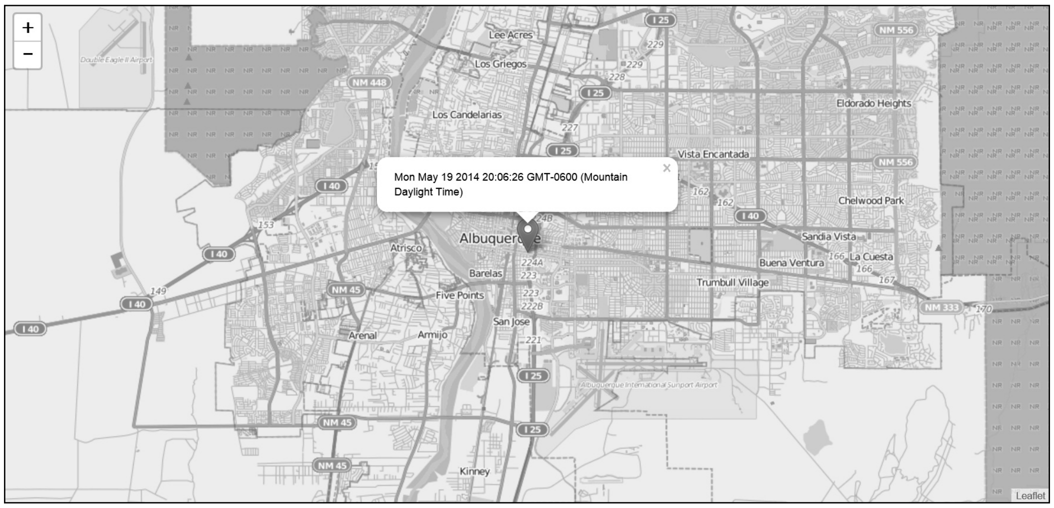


Unable to find your location. You may need to enable Geolocation.

Now, add the following code to `foundLocation()`:

```
function foundLocation(e){
varmydate = new Date(e.timestamp);
L.marker(e.latlng).addTo(map).bindPopup(mydate.toString());
     }
```

The preceding code will run when the user's location is found. The `e` in `foundLocation(e)` is an event object. It is sent when an event is triggered to the function that is responsible for handling that specific event type. It contains information about the event that you will want to know. In the preceding code, the first event object we grab is the `timestamp` object. If you were to display the timestamp in a pop up, you would get a bunch of numbers: **1400094289048**. The timestamp is the number of milliseconds that have passed since January 1, 1970 00:00:00 UTC. If you create an instance of the `date` class and pass it to the `timestamp` object, you receive a human-readable date. Next, the code creates a marker. The latitude and longitude are stored in `e.latlng`. You then add the marker to the map and bind a pop up. The pop up needs a string as a parameter, so you can use the `toString()` method of the `date` class or use `String(mydate)` to convert it. The following screenshot shows the pop up with the date and time when the user clicked on it:


Mon May 19 2014 20:06:26 GMT-0600 (Mountain Daylight Time)

Events and event handlers

So far, you have created maps that display data and added a pop up that displayed when the user clicked on a marker. Now, you will learn how to handle other events and assign these events to event handler functions to process them and do something as a result.

You will first learn how to handle a `map` event. There are 34 events in the `map` class that can be subscribed to. This example will focus on the `click` event. To subscribe to an event, you use the event method `.on();` so, for a `map` event, you use the `map.on()` method and pass the parameters as the event and function to handle the event. This is shown in the following code:

```
map.on('click', function(){alert("You clicked the map");});
```

The code tells Leaflet to send an alert pop-up box with the text **You clicked the map** when the user clicks on the map. In the mobile example, you created a listener that had a named function that executed `foundLocation()`. In the preceding code, the function was put in as a parameter. This is known as an anonymous function. The function has no name, and so, it can only be called when the user clicks on the map.

Remember `e` from the mobile example? If you pass `e` to the function, you can get the `longlat` value of the spot that the user clicked on, as shown in the following code:

```
map.on('click',function(e){
var coord=e.latlng.toString().split(',');
var lat=coord[0].split('(');
var long=coord[1].split(')');
alert("you clicked the map at LAT: "+ lat[1]+" and LONG:"+long[0])
});
```

The preceding code is spaced in a way that is more readable, but you can put it all on a single line. The code displays the longitude and latitude of the spot where the user clicked on the map. The second line assigns the variable `coord`, the value of `e.latlng`. The next two lines strip the latitude and longitude from the value so that you can display them clearly.

You can build on this example by adding a marker when the user clicks on the map by simply replacing the code with the following:

```
L.marker(e.latlng).addTo(map);
```

The preceding code is identical to the code in the mobile example. The difference is that in the mobile example, it was only executed when `locate()` was successful. In this example, it is executed every time the user clicks on the map.

In the section on markers, you created a marker that had the property `draggable:true`. Markers have three events that deal with dragging: `dragstart`, `drag`, and `dragend`. Perform the following steps to return the longitude and latitude of the marker in a pop up on `dragend`:

1. Create the marker and set the draggable property to `true`:

   ```
   varmyMarker = L.marker([35.10418, -
   106.62987],{title:"MyPoint",alt:"The Big
   I",draggable:true}).addTo(map);
   ```

2. Write a function to bind a pop up to the marker and call the method `getLatLong()`:

   ```
   myMarker.bindPopup("I have been moved to:
   "+String(myMarker.getLatLng()));
   ```

3. Subscribe to the event:

   ```
   myMarker.on('dragend',whereAmI);
   ```

Open the map and click on the marker. Hold down the left mouse button and drag the marker to a new location on the map. Release the left button and click on the marker again to trigger the pop up. The pop up will have the new latitude and longitude of the marker.

Custom functions

You subscribed to an event and handled it with a function. So far, you have only passed `e` as a parameter. In JavaScript, you can send anything you want to the function. Also, functions can be called anywhere in your code. You do not have to call them only in response to an event. In this short example, you will create a function that returns a pop up and is triggered on a call and not by an event.

First, create a marker and bind a pop up to it. For the content of the pop up, enter `createPopup(Text as a parameter)`. Add the marker to the map as shown in the following code:

```
var marker1 = L.marker([35.10418, -
106.62987]).addTo(map).bindPopup(createPopup("Text as a
parameter"));
```

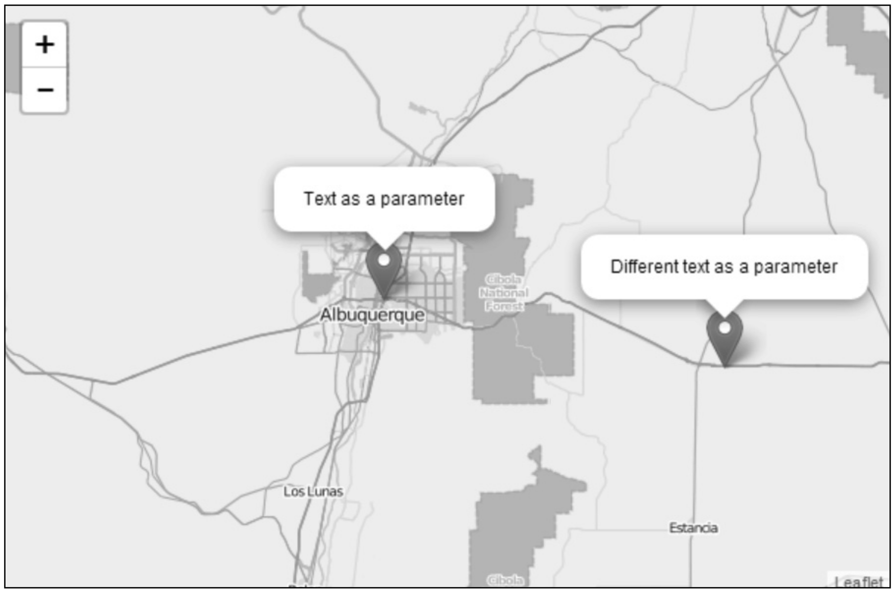
Create a second marker and set the content of the pop up to `createPopup (Different text as a parameter)`:

```
var marker2 = L.marker([35, -
106]).addTo(map).bindPopup(createPopup("Different text as a
parameter"));
```

In the previous examples, you created a pop up by passing text or a pop-up instance. In this example, you call a function, `createPopup()`, with a string as a parameter, as shown in the following code:

```
functioncreatePopup(x){
return
L.popup({keepInView:true,closeButton:false}).setContent(x);functio
n createPopup(x){
returnL.popup({keepInView:true,closeButton:false}).setContent(x);
}
```

The function takes a parameter called `x`. In the marker, when you call the function, you pass a string. This is sent to the function and stored as `x`. When the pop up is created, the `setContent()` method is given the value of `x` instead of a hardcoded string. This function is useful if you have a lot of options set on your pop ups and want them all to be the same. It limits the number of times that you need to repeat the same code. Just pass the text of the pop up to the function, and you get a new pop up with the standardized formatting options. The following screenshot shows both of the pop ups generated by the custom function:



Summary

This chapter covered almost every major topic required to create a Leaflet map. You learned how to add tile layers from multiple providers, including satellite imagery. You can now add points, lines, and polygons to the map, as well as collections of polylines and polygons. You can group layers of different types into layer or feature collections. This chapter covered the styling of objects and adding pop ups. You learned how to interact with the user by responding to events and created custom functions to allow you to code more by writing less.

In the next chapter, you will learn how to add GeoJSON data to your map.