INSTANT

Short | Fast | Focused

Interactive Map designs with Leaflet JavaScript Library How-to

An intuitive guide to creating animated, interactive maps with the Leaflet JavaScript library in a series of straightforward recipes

Jonathan Derrough

[PACKT] PUBLISHING

Instant Interactive Map Designs with Leaflet JavaScript Library How-to

An intuitive guide to creating animated, interactive maps with the Leaflet JavaScript library in a series of straightforward recipes

Jonathan Derrough

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Interactive Map Designs with Leaflet JavaScript Library How-to

First published: May 2013

Production Reference: 1160513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-520-0

www.packtpub.com

Credits

Author
Jonathan Derrough

Reviewer
Michael Markieta

Acquisition Editor
Vinay Argekar

Commissioning Editor
Harsha Bharwani

Technical Editor
Akshata Patil

Project Coordinator
Esha Thakker

Proofreader
Clyde Jenkins

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Jonathan Derrough has been working with Leaflet to build backend applications for location-based mobile video games and travel and tourism apps for several years. He is also a minor contributor to Leaflet on GitHub.

Jonathan is employed by companies big and small to build great applications with maps, from data mining, spatial database integration, tile rendering, and caching to displaying on various client platforms, desktop, and mobiles.

I would like to thank Vladimir Agafonkin for his tremendous work on Leaflet and my family and friends for their everyday support.

About the Reviewer

Michael Markieta has been a self-employed GIS consultant specializing in spatial analytics and programming since his senior years in the Geographic Analysis program at Ryerson University. Michael works confidently in many open source programming languages and libraries, incorporating python scripting into GIS workflows, and JavaScript for custom web mapping applications. He also writes on his personal blog (`http://spatialanalysis.ca`) about GIS, web mapping and cartography, and participates on the GIS StackExchange Network (`http://gis.stackexchange.com/users/3773/michael-markieta`) answering and contributing questions about all things spatially relevant.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Instant Interactive Map Designs with Leaflet JavaScript Library How-to will show you how to create markers and vector shapes, display data from different types of providers, deploy on mobile, and more. It will take you through detailed recipes using a step-by-step approach, from the simplest web map to a full-fledged interactive map fed with GeoJSON data.

Leaflet is the new alternative to web map pioneers, such as OpenLayers or the Google Maps API. It is lightweight, open source, and aims at helping developers create beautiful maps compatible across desktops and mobiles without sacrificing on performance.

What this book covers

Getting started with Leaflet will show you how to setup a Leaflet development environment and create your first map, which will be used as our code base throughout the recipes. Furthermore, you will be able to take an in-depth look into the map's options, learn how to build Leaflet from source and display tiles from different providers.

Using Leaflet on mobile follows by describing how to tweak our code base to make it suitable for mobile browsers and leverage the location features of Leaflet. Location options—probing, high accuracy, and so on—and events—location update and errors—are detailed to make out the best to meet our requirements.

Creating markers with popups and handling events will take you through the steps to add custom markers to your maps and bind popups to them. Tips on how to use HD markers for retina screens and handling click or touch events, and finally interesting popup options will be presented at the end of this recipe.

Creating layers and layer groups focuses on different kinds of layers, how to add them to the map, and organize them in groups to be interacted with. As layers implement the same interface, you will be shown some common attributes and specifics.

Using Leaflet map controls is the recipe looking into how to take advantage of Leaflet's standard map controls such as zoom buttons, scale display, and layers toggling. It will also address layout options and the attributions control.

Using GeoJSON to create stylish Map objects will help you create colorful vector objects using GeoJSON, with points as circles or markers, lines, and polygons. It will also explain how to filter and customize the shapes based on their GeoJSON properties.

Designing interactive choropleth maps details the process to build a rich interactive map of Europe, from getting the data to a fully functional and beautiful map with colors, based on population information, countries highlighting on mouse hover, and a custom control to display the legend and another as an info box updated with the selected object's data.

What you need for this book

The reader will need a computer with a release of Leaflet, an Internet browser and a text or code editor. He or she will also need a smartphone to complete one of the recipes.

Who this book is for

The reader should be familiar with JavaScript programming and can have little or no prior experience with other mapping APIs.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The `onEachFeature` function will add mouse events to each layer created for our features."

A block of code is set as follows:

```
map.locate({
    setView: true
});
```

Any command-line input or output is written as follows:

```
cd leaflet/src
npm install -g jake
npm install jshint
npm install uglify-js
```

New terms and **important words** are shown in bold.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Interactive Map Designs with Leaflet JavaScript Library How-to

Welcome to *Instant Interactive Map Designs with Leaflet JavaScript Library How-to* set up a development environment and build **Leaflet** from source, before quickly moving on to creating your first map with tile layers; and then adding custom markers with popups, layer groups, and controls; and using **GeoJSON** to create vector objects. You will also learn to deploy your maps on mobile and use location. Along with these recipes, you will be shown how to handle events. Finally, combining all the knowledge gathered in this book, you will be shown how to create a complex and colorful interactive choropleth map.

Leaflet is an open-source JavaScript library created by Vladimir Agafonkin (`http://agafonkin.com/en`) aimed at creating interactive maps for desktops and mobiles alike. Web mapping libraries have been around for some time—the Google Maps API was released back in 2005 and OpenLayers in 2006. Leaflet is quite a newcomer but fits neatly next to its competitors by aiming at helping developers create beautiful maps efficiently by being both lightweight and easy to get on with.

Getting started with Leaflet (Simple)

Leaflet is a JavaScript library so there are a few things we will need before we get started, namely an Internet browser, a text editor, and a stable release of Leaflet. We will then be on our way to create our first map.

Getting ready

First, we need to get an Internet browser, if we don't have one already installed. Leaflet is tested with modern desktop browsers: Chrome, Firefox, Safari 5+, Opera 11.11+, and Internet Explorer 7-10. Internet Explorer 6 support is stated as not perfect but accessible. We can pick one of them, or all of them if we want to be thorough.

Then, we need an editor. Editors come in many shapes and flavors: free or not free, with or without syntax highlighting, or remote file editing. A quick search on the Internet will provide thousands of capable editors. Notepad++ (`http://notepad-plus-plus.org/`) for Windows, Komodo Edit (`http://www.activestate.com/komodo-edit`) for Mac OS, or Vim (`http://www.vim.org/`) for Linux are among them.

We can download Leaflet's latest stable release (v0.5.1 at the time of writing) and extract the content of the ZIP file somewhere appropriate. The ZIP file contains the sources as well as a prebuilt version of the library that can be found in the `dist` directory.

Optionally, we can build from the sources included in the ZIP file; see this recipe's *Building Leaflet from source* section.

Finally, let's create a new project directory on our hard drive and copy the `dist` folder from the extracted `Leaflet` package to it, ensuring we rename it to `leaflet`.

How to do it...

Note that the following code will constitute our code base throughout the rest of the book.

1. Create a blank HTML file called `index.html` in the root of our project directory.
2. Add the code given here and use the browser installed previously to execute it:

```
<!DOCTYPE html>
<html>

    <head>
        <link rel="stylesheet" type="text/css" href="leaflet/
leaflet.css" />
        <!--[if lte IE 8]>
            <link rel="stylesheet" type="text/css" href=" leaflet/
leaflet.ie.css" />
        <![endif]-->
        <script src="leaflet/leaflet.js"></script>
        <style>
            html, body, #map {
                height: 100%;
            }
            body {
                padding: 0;
```

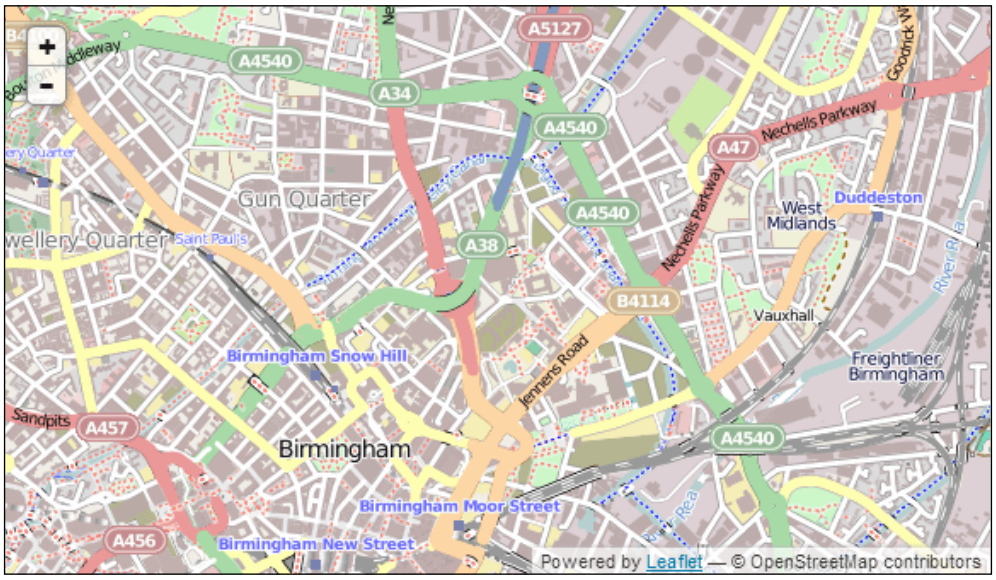
```
                    margin: 0;
                }
            </style>
            <title>Getting Started with Leaflet</title>
        </head>

        <body>
            <div id="map"></div>
            <script type="text/javascript">
                var map = L.map('map', {
                    center: [52.48626, -1.89042],
                    zoom: 14
                });
                L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/
{x}/{y}.png', {
                    attribution: '&copy; OpenStreetMap contributors'
                }).addTo(map);
            </script>
        </body>

</html>
```

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.

The following screenshot is of the first map we have created:



How it works...

The `index.html` file we created is a standardized file that all Internet browsers can read and display the contents. Our file is based on the **HTML doctype standard** produced by the **World Wide Web Consortium** (**W3C**), which is only one of many that can be used as seen at `http://www.w3.org/QA/2002/04/valid-dtd-list.html`. Our index file specifies the doctype on the first line of code as required by the W3C, using the `<!DOCTYPE HTML>` markup.

We added a link to Leaflet's main CSS file in the `head` section of our code:

```
<link rel="stylesheet" type="text/css" href="leaflet/leaflet.css" />
```

We also added a conditional statement to link an Internet Explorer 8 or lower only stylesheet when these browsers interpret the HTML code:

```
<!--[if lte IE 8]>
  <link rel="stylesheet" type="text/css" href="leaflet/leaflet.ie.css"
/>
<![endif]-->
```

This stylesheet mainly addresses Internet Explorer specific issues with borders and margins.

Leaflet's JavaScript file is then referred to using a `script` tag:

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

We are using the compressed JavaScript file that is appropriate for production but very inefficient for debugging. In the compressed version, every white space character has been removed, as shown in the following bullet list, which is a straight copy-paste from the source of both files for the function `onMouseClick`:

- compressed:

```
_onMouseClick:function(t){!this._loaded||this.dragging&&
this.dragging.moved()||(this.fire("preclick"),this._
fireMouseEvent(t))},
```

- uncompressed:

```
_onMouseClick: function (e) {
    if (!this._loaded || (this.dragging &&
    this.dragging.moved())) { return; }

    this.fire('preclick');
    this._fireMouseEvent(e);
},
```

To make things easier, we can replace `leaflet.js` with `leaflet-src.js`—an uncompressed version of the library.

We also added styles to our document to make the map fit nicely in our browser window:

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

The `<div>` tag with the `id` attribute `map` in the document's body is the container of our map. It must be given a height otherwise the map won't be displayed:

```
<div id="map" style="height: 100%;" ></div>
```

Finally, we added a `script` section enclosing the map's initialization code, instantiating a `Map` object using the `L.map(…)` constructor and a `TileLayer` object using the `L.tileLayer(…)` constructor. The `script` section must be placed after the map container declaration otherwise `Leaflet` will be referencing an element that does not yet exist when the page loads.

When instantiating a `Map` object, we pass the id of the container of our map and an array of `Map` options:

```
var map = L.map('map', {
    center: [52.48626, -1.89042],
    zoom: 14
});
```

There are a number of `Map` options affecting the state, the interactions, the navigation, and the controls of the map. See the documentation to explore those in detail at `http://leafletjs.com/reference.html#map-options`.

Next, we instantiated a `TileLayer` object using the `L.tileLayer(…)` constructor and added to the map using the `TileLayer.addTo(…)` method:

```
L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; OpenStreetMap contributors'
}).addTo(map);
```

Here, the first parameter is the URL template of our tile provider—that is `OpenStreetMap`—and the second a noncompulsory array of `TileLayer` options including the recommended attribution text for our map tile's source.

The `TileLayer` options are also numerous. Refer to the documentation for the exhaustive list at `http://leafletjs.com/reference.html#tilelayer-options`.

There's more...

Let's have a look at some of the `Map` options, as well as how to build Leaflet from source or use different tile providers.

More on Map options

We have encountered a few `Map` options in the code for this recipe, namely `center` and `zoom`. We could have instantiated our OpenStreetMap `TileLayer` object before our `Map` object and passed it as a `Map` option using the `layers` option. We also could have specified a minimum and maximum zoom or bounds to our map, using `minZoom` and `maxZoom` (integers) and `maxBounds`, respectively. The latter must be an instance of `LatLngBounds`:

```
var bounds = L.latLngBounds([
    L.latLng([52.312, -2.186]),
    L.latLng([52.663, -1.594])
]);
```

We also came across the `TileLayer` URL template that will be used to fetch the tile images, replacing {s} by a subdomain and {x}, {y}, and {z} by the tiles coordinate and zoom. The subdomains can be configured by setting the `subdomains` property of a `TileLayer` object instance.

Finally, the `attribution` property was set to display the owner of the copyright of the data and/or a description.

Building Leaflet from source

A Leaflet release comes with the source code that we can build using `Node.js`. This will be a necessity if we want to fix annoying bugs or add awesome new features.

The source code itself can be found in the `src` directory of the extracted release ZIP file. Feel free to explore and look at how things get done within a Leaflet.

First things first, go to `http://nodejs.org` and get the `install` file for your platform. It will install `Node.js` along with `npm`, a command line utility that will download and install **Node Packaged Modules** and resolve their dependencies for us. Following is the list of modules we are going to install:

- **Jake**: A JavaScript build program similar to make
- **JSHint**: It will detect potential problems and errors in JavaScript code
- **UglifyJS**: A mangler and compressor library for JavaScript

Hopefully, we won't need to delve into the specifics of these tools to build Leaflet from source.

So let's open a command line interpreter—`cmd.exe` on Windows, or a `terminal` on Mac OSX or Linux—and navigate to the Leaflet's `src` directory using the `cd` command, then use `npm` to install Jake, JSHint and UglifyJS:

```
cd leaflet/src
npm install -g jake
npm install jshint
npm install uglify-js
```

We can now run Jake in Leaflet's directory:

```
jake
```

What about tile providers?

We could have chosen a different tile provider as OpenStreetMap is free of charge but has its limitations in regard of a production environment. A number of web services provide tiles but might come at a price depending on your usage: **CloudMade**, **MapQuest**.

These three providers serve tiles use the `OpenStreetMap` tile scheme described at `http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames`.

Remember the way we added the OpenStreetMap layer to the map?

```
L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; OpenStreetMap contributors'
}).addTo(map);
```

Adding a Cloudmade layer or a MapQuest layer would have been very similar:

- Cloudmade:

```
L.tileLayer(' http://{s}.tile.cloudmade.com/API-key/997/256/{z}/
{x}/{y}.png', {
    attribution: ' Map data &copy; <a href="http://openstreetmap.
org">OpenStreetMap</a> contributors, <a href="http://
creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, Imagery ©
<a href="http://cloudmade.com">CloudMade</a>'
}).addTo(map);
```

- MapQuest:

```
L.tileLayer('http://{s}.mqcdn.com/tiles/1.0.0/map/{z}/{x}/{y}.
png', {
    attribution: ' Tiles Courtesy of <a href="http://www.mapquest.
com/" target="_blank">MapQuest</a> <img src="http://developer.
mapquest.com/content/osm/mq_logo.png">',
    subdomains: ['otile1', 'otile2', 'otile3', 'otile4']
}).addTo(map);
```

You will learn more about the `Layer` URL template and `subdomains` option in the documentation at `http://leafletjs.com/reference.html#tilelayer`.

Leaflet also supports **Web Map Service** (**WMS**) tile layers—read more about it at `http://leafletjs.com/reference.html#tilelayer-wms`—and GeoJSON layers—learn about it in our *Using GeoJSON to create stylish map objects (Intermediate)* recipe or in the documentation at `http://leafletjs.com/reference.html#geojson`.

Using Leaflet on mobile (Simple)

Deploying a Leaflet map on mobile requires a bit of preparation. We will also look at how to turn on location tracking using Leaflet's utilities.

Getting ready

Create a simple map as described in the previous recipe *Getting started with Leaflet (Simple)*.

We might also want to install a web server or a WAMP—for example, EasyPHP (`http://www.easyphp.org/`), MAMP (`http://www.mamp.info/`), or LAMP environment on our development computer so that our test project can be accessed via HTTP instead of the file protocol. The reason for this is that some browsers—Chrome at least—won't authorize the use of the location API when accessing local files. We will also be able to test our project across our local network with our mobile.

Leaflet supports a wide range of mobile browsers on iOS, Android, webOS, Blackberry, and Windows Phone 8. The complete feature list can be seen at `http://leafletjs.com/features.html`.

How to do it...

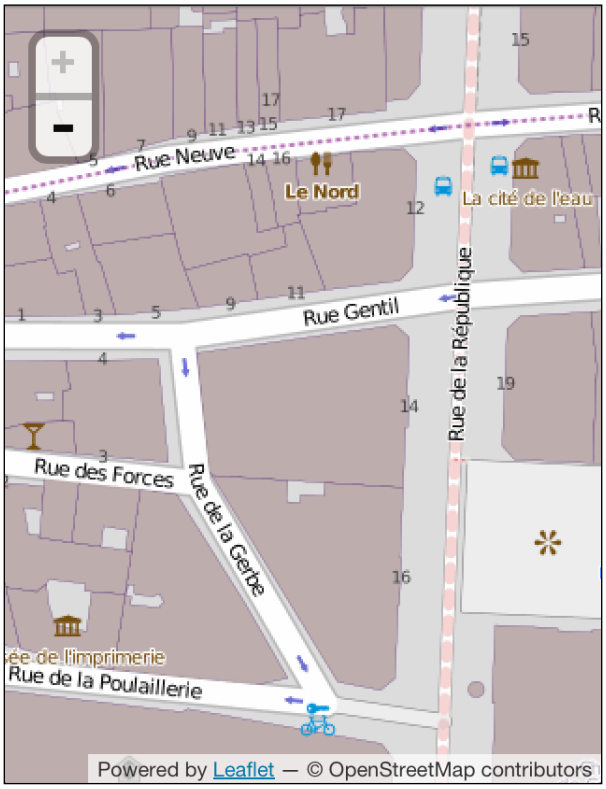
1. First, we need to add a `meta` tag to the head section of our previously created `index.html` file:

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

2. Then add the following code right after the `TileLayer` object instantiation and use the browser installed previously, or a mobile, to execute it:

```
map.locate({
    setView: true
});
```

The next screenshot shows the map centered on an example location of a mobile user. Map controls, attribution text, and window are resized according to the browser and screen dimensions of the mobile phone.



How it works...

The first thing we did using the `meta` tag ensures that our map container won't be scaled by the browser or the user.

Leaflet comes with the JavaScript location API support out of box and makes it really easy for us to take an advantage of it.

In our example, we enable the location with the `Map.locate(...)` method and tell `Leaflet` to center the view on our current location by passing the `setView` Locate option set to `true`.

There's more...

Learn more about location events and `Map.locate(...)` options.

More on location events

Note that without using the `setView` Locate option, the map's view won't be updated to be centered on our location but it will fire the `Map.locationfound` event all the same:

```
map.on('locationfound', function (e) {
  alert('Location found: ' + e.latlng.lat + ' ' + e.latlng.lng);
});
```

Another important location event is the `Map.locationerror` event fired when something goes wrong:

```
map.on('locationerror', function (e) {
  alert('Location error: ' + e.message + '[' + e.code + ']');
});
```

More on Map.locate(...) options

`Map.locate(...)` accepts a number of interesting Locate options:

- `watch`: (Boolean type) It probes location changes; use the `Map.stopLocate()` method to stop watching (default is false)
- `enableHighAccuracy`: (Boolean type) It enables high accuracy; getting the location might be slower and more power consuming (default is false)
- `maxZoom`: (Integer type) It is used when the `setView` option is set to `true` (default is Infinity)

Creating markers with popups and handling events (Intermediate)

The most essential addition to a map is markers. We will see how to add markers to a Leaflet map, customize the markers' display, and handle mouse or touch events to open popup balloons.

Getting ready

Create a simple map as described in the *Getting started with Leaflet (Simple)* recipe.

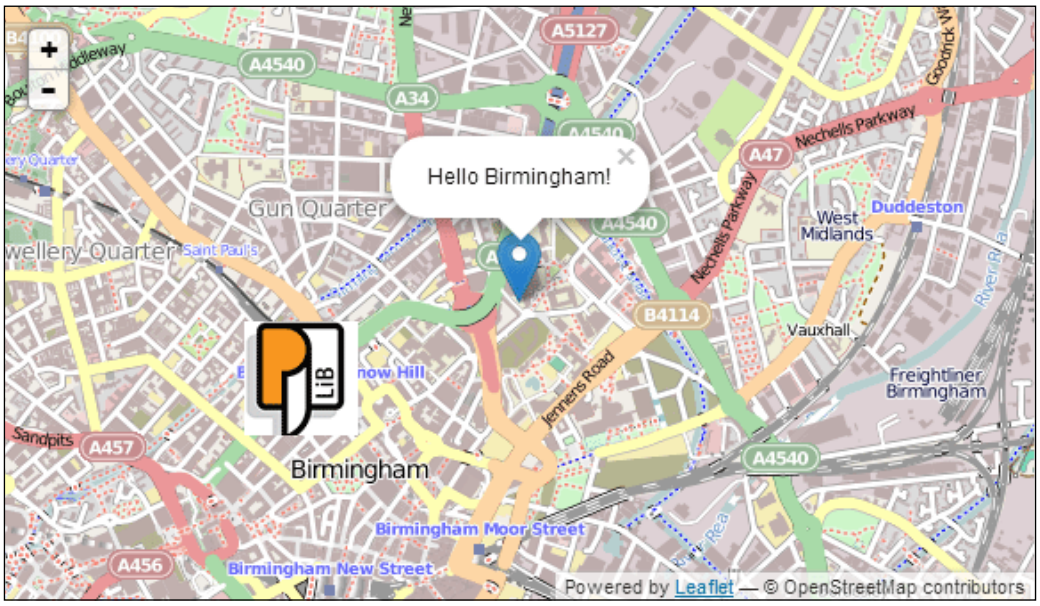
Create an `images` subdirectory in your project's `root` directory and put an image file to be used as a custom marker icon in it. The image should have a size and weight suitable for web usage; it should not have the need to be shrunk nor stretched by the browser for best performance and quality. But if we really have to use an undersized or oversized image, the `iconSize` option can be used to modify its display resolution.

How to do it...

1. Add the following code to our previously created `index.html` file, right after the `TileLayer` instantiation:

```
L.marker([52.48626, -1.89042], {
    clickable: true
})
    .bindPopup('Hello Birmingham!')
    .addTo(map);
L.marker([52.484103, -1.900850], {
    clickable: true,
    draggable: true,
    icon: L.icon({
        iconUrl: 'images/packtpub.png',
        iconAnchor: [32, 32],
        popupAnchor: [32, 10]
    })
})
    .bindPopup('Hello custom!')
    .addTo(map);
```

The next screenshot shows our map with the markers we added:



How it works...

The first marker was instantiated using the `L.marker(…)` constructor to which we passed a set of coordinates—the center of our map—and an array of Marker options. The only `Marker option` set here was the `clickable` option, which enables the marker to respond to mouse or touch events.

As we did not provide an instance of `Icon` object—which basically is a Leaflet representation of an image that can be bound to a marker—as a `Marker option`. Leaflet will use the default `marker-icon.png` image file found in the `leaflet/images` directory.

The second marker however displays a custom image, passing an instance of an Icon object as the `icon` Marker option. The `L.icon(…)` constructor takes an array of `Icon options`. Here, `iconUrl` was set to designate the image we put in the `images` subdirectory, `iconAnchor` to ensure the marker's center is placed on its coordinate—otherwise the top left corner is placed—and `popupAnchor` to offset the popup's position so that the tip of the arrow is centered horizontally and overlaps the marker a bit. This custom marker can also be dragged on the map which was made possible by setting the `draggable` Marker option to `true`.

The `Marker.bindPopup(…)` will create an instance of a `Popup` object and bind it to the marker for us. It takes an HTML string and an optional array of `Popup options` as parameters. The marker is then added to the map using the `Marker.addTo(…)` method.

The `Marker.bindPopup(…)` and `Marker.addTo(…)` methods—as well as most object methods in `Leaflet`—returns the calling instance, which is a very efficient pattern since it enables us to fiddle our method calls.

There's more...

Once again, Leaflet made things really easy for us. Here is a way to achieve the popup mechanism:

```
var marker = L.marker([52.48626, -1.89042], {
    clickable: true
})
    .on('click', function (e) {
    var point = map.latLngToLayerPoint(marker.getLatLng());
    point.y -= marker.options.icon.options.iconSize.y;
    point = map.layerPointToLatLng(point);
    L.popup()
        .setContent('Hello Birmingham!')
        .setLatLng(point)
        .openOn(map);
})
    .addTo(map);
```

What we did here was binding an anonymous function to the `click` event of the marker that when fired will instantiate a `Popup` object with the same content as before and opening it on the map. We had to offset its vertical position on screen by the height of the marker's icon so that the popup doesn't hide the marker, converting back and forth the latlong coordinate to onscreen coordinate. Latlong coordinates are often expressed as pairs of floating point values in WGS84 decimal format as used by the GPS system.

Other events can be bound on `Map` and `Marker` object instances. You can have a look at Leaflet's documentation for the complete list of events available for each object:

- `http://leafletjs.com/reference.html#map-events`
- `http://leafletjs.com/reference.html#marker`

What about retina screens?

We could also have provided a retina version of our custom image using the `iconRetinaUrl` Icon option. This image should be in higher definition. More about computer displays and pixel density can be found at `http://en.wikipedia.org/wiki/Pixel_density`.

More on Popup options

The `Popup` object constructor can take a number of interesting `Popup options`:

- `autoPan`: (Boolean) It will prevent the map from panning automatically to make the popup visible (default is true)
- `autoPanPadding`: (Point) The padding applied between the popup and the map's borders after the autopan was performed

See the documentation for the complete list of `Popup options` at `http://leafletjs.com/reference.html#popup-options`.

Creating layers and layer groups (Intermediate)

Interactive maps often present several layers that can be toggled on events. Let's see how to create layers, add them to layer groups and interact with them.

Getting ready

Get familiar with layers and markers—see the *Getting started with Leaflet (Simple)* and *Creating markers with popups and handling events (Intermediate)* recipes.

Create a simple map as described in the *Getting started with Leaflet (Simple)* recipe.

How to do it...

1. Modify the map's instantiation from our previously created `index.html` file to match the following code:

```
var map = L.map('map', {
    center: [52.4807, -1.972],
    zoom: 12
});
```

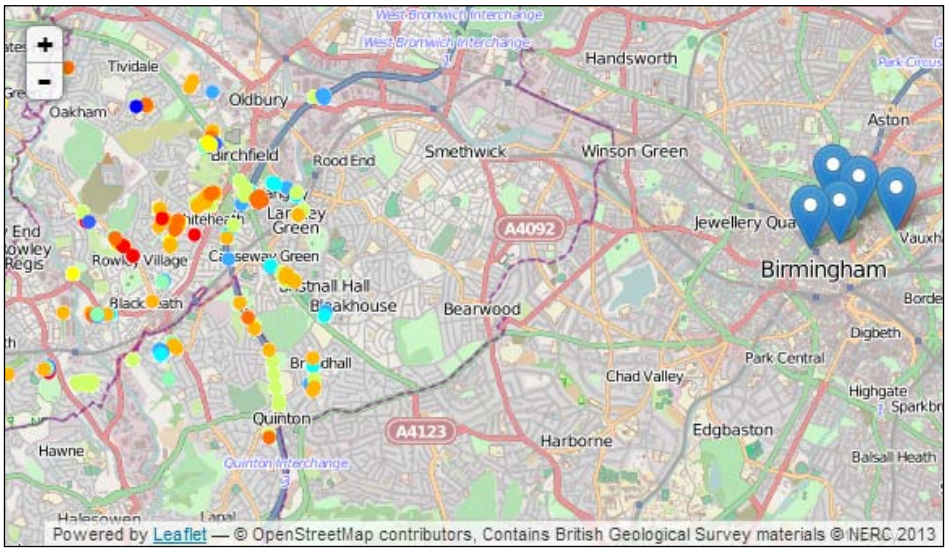
2. Add the following code right after the `TileLayer` instantiation:

```
var hydro = L.tileLayer.wms('http://mapapps.bgs.ac.uk/arcgis/
services/HydroMap/HydroMap/MapServer/WMSServer', {
    layers: 'Hydrogeology',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials ©
NERC 2013',
    opacity: 0.25
});
var soil = L.tileLayer.wms('http://mapapps2.bgs.ac.uk/ArcGIS/
services/SoilPortal/SoilPortal/MapServer/WMSServer', {
    layers: 'Soil.depth.from.boreholes',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials ©
NERC 2013'
});
var firstGroup = L.layerGroup([
    L.marker([52.48626, -1.89042]),
    L.marker([52.48857, -1.88733]),
    L.marker([52.49063, -1.89263]),
    L.marker([52.48349, -1.89894]),
    L.marker([52.48295, -1.88613]),
    hydro
]).addTo(map);
var secondGroup = L.layerGroup([
    L.marker([52.484103, -1.900850]),
    L.marker([52.48788, -1.89066]),
    L.marker([52.48930, -1.89617]),
    L.marker([52.48475, -1.89482]),
    L.marker([52.48626, -1.88289]),
    soil
]);
map.on('click', function (e) {
    if (map.hasLayer(firstGroup)) {
        map.removeLayer(firstGroup);
        map.addLayer(secondGroup);
    } else {
```

```
            map.removeLayer(secondGroup);
            map.addLayer(firstGroup);
        }
    });
```

The next screenshot shows our first group—a hydrological WMS layer and a few random markers:

The next screenshot is our second group—a WMS layer showing the soil depth from bore holes and another set of random markers:



How it works...

We saw in the previous recipe how to create markers and add them to the map. Here we used the `LayerGroup` class to easily show and hide two sets of markers. It might seem a bit odd but the `Marker` class implements the `ILayer` interface that enables us to add them to a `LayerGroup`. It would work all the same if we added `TileLayer` instances to a layer group.

First, we created two new layers, specifically WMS layers using the `L.tileLayer.wms(…)` constructor:

```
var hydro = L.tileLayer.wms('http://mapapps.bgs.ac.uk/arcgis/services/
HydroMap/HydroMap/MapServer/WMSServer', {
    layers: 'Hydrogeology',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials © NERC
2013',
    opacity: 0.25
});
var soil = L.tileLayer.wms('http://mapapps2.bgs.ac.uk/ArcGIS/services/
SoilPortal/SoilPortal/MapServer/WMSServer', {
    layers: 'Soil.depth.from.boreholes',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials © NERC
2013'
});
```

Tile images from providers, such as OpenStreetMap or CloudMade, are likely to be pregenerated and cached. A Web Map Service lets us request imagery from a map server targeting a specific data set—specified here by the `layers` TileLayer.WMS option—and bounding box calculated on the fly by Leaflet for each tile. The `transparent` TileLayer.WMS option tells the map server to send back transparent pixels where no data appears which is highly recommended when using layers as overlay. We also want to consider using the `opacity` TileLayer option if our data covers large areas like the hydro layer.

Next, we created two instances of `LayerGroup` using the `L.layerGroup(…)` constructor, which takes an array of instances implementing the `ILayer` interface, adding only the first one to the map:

```
var firstGroup = L.layerGroup([
    L.marker([52.48626, -1.89042]),
    L.marker([52.48857, -1.88733]),
    L.marker([52.49063, -1.89263]),
    L.marker([52.48349, -1.89894]),
    L.marker([52.48295, -1.88613]),
    hydro
```

```
]).addTo(map);
var secondGroup = L.layerGroup([
    L.marker([52.484103, -1.900850]),
    L.marker([52.48788, -1.89066]),
    L.marker([52.48930, -1.89617]),
    L.marker([52.48475, -1.89482]),
    L.marker([52.48626, -1.88289]),
    soil
]);
```

Each `Marker` instance is created using the `L.marker(…)` constructor. See the previous recipe or the documentation for more information. We also added the `hydro` layer to the first group and the `soil` layer to the second.

Finally, we bound an anonymous function to the map's `click` event:

```
map.on('click', function (e) {
    if (map.hasLayer(firstGroup)) {
        map.removeLayer(firstGroup);
        map.addLayer(secondGroup);
    } else {
        map.removeLayer(secondGroup);
        map.addLayer(firstGroup);
    }
});
```

The anonymous function tests if the first group was added to the map. If so, it is removed from the map and the second group is added. Otherwise, the second group is removed and the first one is added.

Using Leaflet map controls (Intermediate)

Controls are a powerful way to expose interactions and crucial information to the user. Let's see what Leaflet has up its sleeves when it comes to map and layers controls.

Getting ready

Learn about markers, layers and layer groups in the *Getting started with Leaflet (Simple)*, *Creating markers with popups and handling events (Intermediate)*, and *Creating layers and layer groups (Intermediate)* recipes.

Create a simple map as described in the *Getting started with Leaflet (Simple)* recipe.

How to do it...

1. Modify the map's instantiation from our previously created `index.html` file to match the following code snippet:

```
var map = L.map('map', {
    center: [52.4807, -1.972],
    zoom: 12
});
```

2. Modify the `TileLayer` instantiation to match the following code snippet:

```
var osm = L.tileLayer('http://{s}.tile.openstreetmap.org/{z}/{x}/
{y}.png', {
    attribution: '&copy; OpenStreetMap contributors'
}).addTo(map);
```

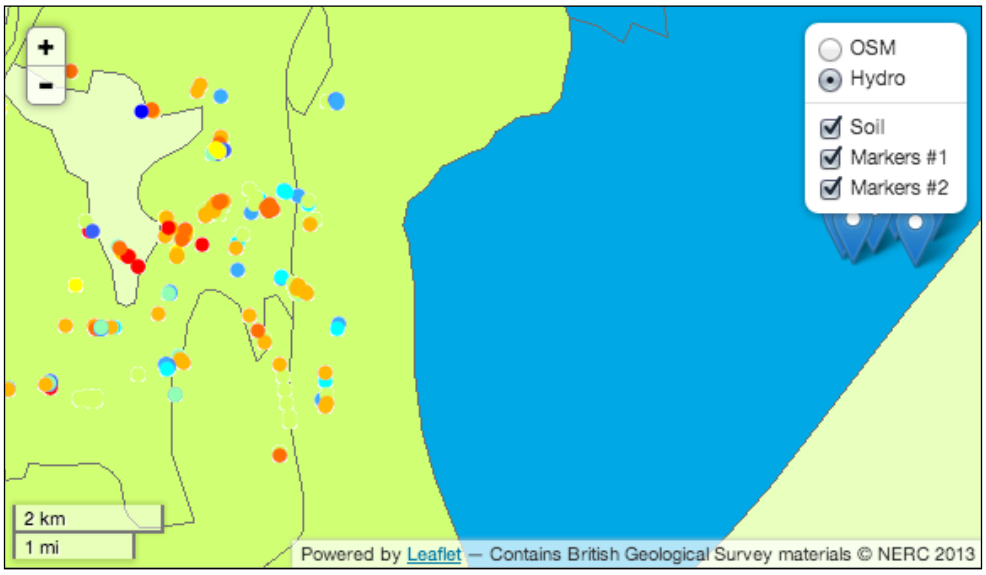
3. Add the following code right after the `TileLayer` instantiation:

```
var hydro = L.tileLayer.wms('http://mapapps.bgs.ac.uk/arcgis/
services/HydroMap/HydroMap/MapServer/WMSServer', {
    layers: 'Hydrogeology',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials ©
NERC 2013',
    opacity: 0.25
});
var soil = L.tileLayer.wms('http://mapapps2.bgs.ac.uk/ArcGIS/
services/SoilPortal/SoilPortal/MapServer/WMSServer', {
    layers: 'Soil.depth.from.boreholes',
    format: 'image/png',
    transparent: true,
    attribution: 'Contains British Geological Survey materials ©
NERC 2013'
});
var firstGroup = L.layerGroup([
    L.marker([52.48626, -1.89042]),
    L.marker([52.48857, -1.88733]),
    L.marker([52.49063, -1.89263]),
    L.marker([52.48349, -1.89894]),
    L.marker([52.48295, -1.88613])
]);
```

```
var secondGroup = L.layerGroup([
    L.marker([52.484103, -1.900850]),
    L.marker([52.48788, -1.89066]),
    L.marker([52.48930, -1.89617]),
    L.marker([52.48475, -1.89482]),
    L.marker([52.48626, -1.88289])
]);
L.control.layers({
    'OSM': osm,
    'Hydro': hydro
}, {
    'Soil': soil,
    'Markers #1': firstGroup,
    'Markers #2': secondGroup
}).addTo(map);
L.control.scale().addTo(map);
```

The next screenshot is our map displaying solely an OpenStreetMap tile layer:

The next screenshot shows that we have enabled a hydrological WMS layer as background and three layer groups we saw previously.



How it works...

Controls are interface elements implementing the `IControl` interface that are added to specific positions: top-left, top-right (default), bottom-left, and bottom-right. It is specified with the position Control option.

A number of standard controls are available and one is visible by default on a Leaflet map, the zoom control of type `Control.Zoom`.

Back to our code, we modified the two `LayerGroup` instantiations so that the `hydro` and `soil` layers are not part of them any more since we passed them as base layers when creating and adding a standard `control.layers` control to the map:

```
L.control.layers({
    'OSM': osm,
    'Hydro': hydro
}, {
    'Soil': soil,
    'Markers #1': firstGroup,
    'Markers #2': secondGroup
}).addTo(map);
```

The `L.control.layers(…)` constructor takes an array of base layers which act as mutually exclusive background layers, with names and `ILayer` instances as keys and values. It also takes an optional array of overlays that can be shown or hidden using the control's interface. The available `Control.Layers` options are the `position` inherited from the Control options, `collapsed`—a Boolean specifying if the control should be collapsed automatically into an icon and expanded when the mouse is over it, and `autoZIndex`—another Boolean that will tell the control to assign z-indices to the layers it handles.

Next, we added another standard control, the `scale` control:

```
L.control.scale().addTo(map);
```

It shows the scale of the map depends on the zoom in miles and kilometers.

There's more...

Leaflet defines a specific control type to display the map's data copyrights, ownership and other attributions.

More on attributions

The map attribution is also a standard control of type `Control.Attribution`. The `L.control.attribution(…)` takes an array of `Control.Attribution options` of which there is one besides `position Control option`:

- `prefix`: (String type) Lets us configure the HTML prefix of the attribution (default: Powered by Leaflet)

Using GeoJSON to create stylish map objects (Intermediate)

In this recipe we will learn how to display custom data on top of our tiles by creating map objects, such as points, lines, and polygons. We will also learn how to use GeoJSON with Leaflet to create vector objects with style(s).

Getting ready

Get familiar with GeoJSON, a human readable format inspired by the JSON format aimed at describing geographic features. The full specification is available at `http://www.geojson.org/geojson-spec.html`.

Create a simple map as described in the *Getting started with Leaflet (Simple)* recipe.

How to do it...

1. Modify the map's instantiation from our previously created `index.html` file to match the following code:

```
var map = L.map('map', {
    center: [52.48626, -1.89042],
    zoom: 15
});
```

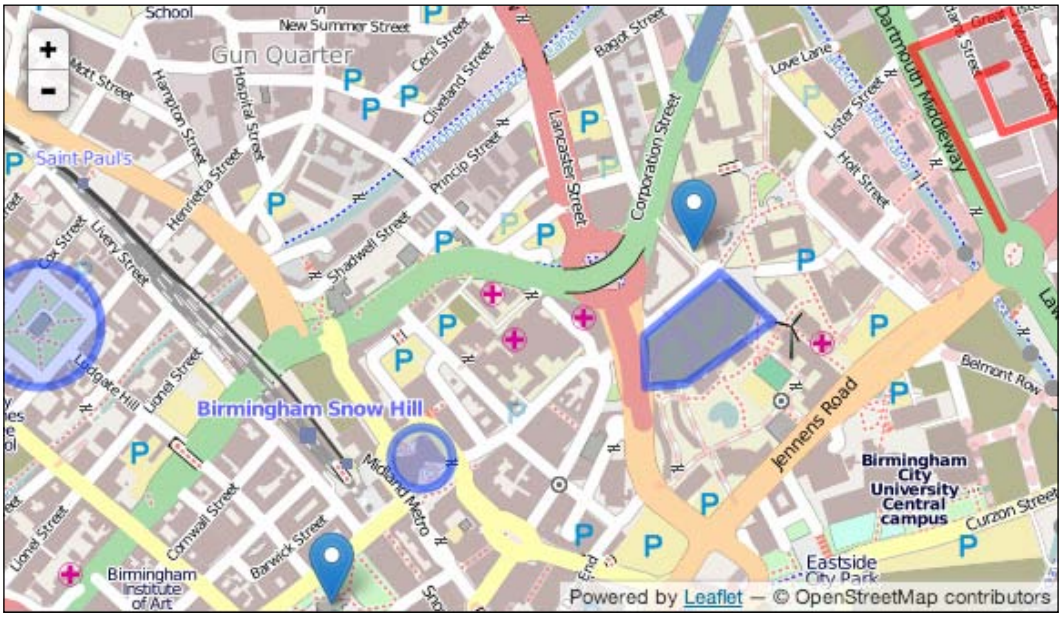
2. Add the following code right after `TileLayer` object instantiation:

```
var point = {
    'type': 'Point',
    'coordinates': [-1.89042, 52.48626]
};
L.geoJson(point).addTo(map);
var linestring = {
    'type': 'LineString',
    'coordinates': [
        [-1.88313, 52.48659],
        [-1.88525, 52.48918],
        [-1.88296, 52.48973],
        [-1.88185, 52.48824],
        [-1.88313, 52.48799],
        [-1.88368, 52.48880],
        [-1.88305, 52.48894]
    ]
};
var redlinestyle = {
    style: {
        'color': 'red',
        'weight': 5,
        'opacity': 0.65
    }
};
L.geoJson(linestring, redlinestyle).addTo(map);
var polygon = {
    'type': 'Polygon',
    'coordinates': [
        [
```

```
                [-1.89058, 52.48438],
                [-1.89152, 52.48433],
                [-1.89156, 52.48506],
                [-1.88976, 52.48598],
                [-1.88852, 52.48527]
            ]
        ]
};
L.geoJson(polygon).addTo(map);
var points = [{
        'type': 'Feature',
        'geometry': {
            'type': 'Point',
            'coordinates': [-1.89896, 52.48119]
        }
    }, {
        'type': 'Feature',
        'properties': {
            'circle': 'true',
            'radius': '50'
        },
        'geometry': {
            'type': 'Point',
            'coordinates': [-1.89682, 52.48329]
        }
    }, {
        'type': 'Feature',
        'properties': {
            'circle': 'true',
            'radius': '100'
        },
        'geometry': {
            'type': 'Point',
            'coordinates': [-1.90583, 52.48521]
        }
    }
];
```

```
L.geoJson(points, {
    pointToLayer: function (feature, latlng) {
        if (typeof (feature.properties) != 'undefined' && typeof
(feature.properties.circle) != 'undefined' && feature.properties.
circle == 'true') {
            return L.circle(latlng, feature.properties.radius);
        }
        return L.marker(latlng);
    }
}).addTo(map);
```

The following screenshot shows our map with the different shapes we created:



How it works...

Before we delve into details, let's talk about **map panes** for a bit. A Leaflet map stores each layer in one of its map panes depending on its type—tile layer, overlay, marker, and so on. They act as rendering groups with specific depth orders. See the documentation for more details on how to access the map panes at `http://leafletjs.com/reference.html#map-panes`.

Here we made extensive use of the `L.geojson(…)` constructor which takes a `GeoJSON` object as first parameter and an optional `GeoJSON options` array as second.

First, we added a point at the center of our map. Note that the coordinates in GeoJSON follow the **OGC** (**Open Geospatial Consortium**) convention—longitude first in the following code snippet:

```
var point = {
    'type': 'Point',
    'coordinates': [-1.89042, 52.48626]
};
L.geoJson(point).addTo(map);
```

The display of a point without style is the default Leaflet marker.

Next we created an array of `GeoJSON options` affecting the display of the line we created next:

```
var redlinestyle = {
    style: {
        'color': 'red',
        'weight': 5,
        'opacity': 0.65
    }
};
```

The style is specified either as an array of vector graphics properties or as a function returning such an array and taking a feature parameter of type `GeoJSON`. Styling would have to be done another way with points; let's see how.

Finally we added a set of points taking advantage of the `pointToLayer` GeoJSON option in the form of a function taking a `GeoJSON` object instance and a `LatLng` object instance as parameters:

```
L.geoJson(points, {
    pointToLayer: function (feature, latlng) {
        if (typeof (feature.properties) != 'undefined' && typeof
(feature.properties.circle) != 'undefined' && feature.properties.
circle == 'true') {
            return L.circle(latlng, feature.properties.radius);
        }
        return L.marker(latlng);
    }
}).addTo(map);
```

The `properties` field of a `GeoJSON` feature enables us to pass information as a key-value pair. Here we added a `circle` property to some of the points to render them as circle, specifying a `radius` by the same means. Our `pointToLayer` function tests for the presence and value of the `circle` property and returns a `Circle` instance created with the `L.circle(…)` constructor, passing the `latlng` parameter as coordinate and the `radius` property. Otherwise it returns a simple `Marker` object instance styled with the default icon.

There's more...

Here is how to bind events, customize, and filter each of our features.

More on GeoJSON options

Two more `GeoJSON options` exist:

- `onEachFeature`: A function receiving a `GeoJSON` object instance—or feature—and an instance of `ILayer`, which is called at the creation of each feature as a layer so that you can bind events or take actions on properties.
- `filter`: A function also receiving a `GeoJSON` object instance and an instance of `ILayer` and returning a Boolean indicating if the feature should be visible (true) or not (false).

Designing interactive choropleth maps (Advanced)

Create a colorful interactive choropleth map of the population density in Europe.

Getting ready

Create a simple map as described in the *Getting started with Leaflet (Simple)*, *Using Leaflet map controls (Intermediate)*, and *Using GeoJSON to create stylish map objects (Intermediate)* recipes.

We will also need data and GeoCommons is a good place for that since it houses thousands of data sets available in several formats such as KML, ESRI Shapefile, and CSV. Additional formats are also available but might not be suitable.

Population density data in Europe can be found by running a quick search at `http://geocommons.com/search?model=&query=europe+density`.

Unfortunately, JSON is not GeoJSON, so downloading a data set in that format won't provide us with what we really need: country names, polygons, and population densities. GeoJSON derives from JSON by its format and syntax and provides data structures to define geospatial features. The format is best described at `http://www.geojson.org/geojson-spec.html`.

An alternative would be to convert the KML version of the data with an online service such as MyGeodata Converter (`http://converter.mygeodata.eu/`) or a tool like Quantum GIS (`http://www.qgis.org/`). Beware as converters might not be able to guess what to do with the parts of the data that are not geography related, so you might need to address those in another way.

How to do it...

Note that the `GeoJSON` data won't be shown here for brevity purpose. Refer to the recipe's `index.html` file for the full code.

First we need to add a few styles to our HTML document, so just after the `body` selector definition, add the following code:

```
.info {
  padding: 6px 8px;
  font: 14px/16px Arial, Helvetica, sans-serif;
  background: white;
  background: rgba(255,255,255,0.8);
  box-shadow: 0 0 15px rgba(0,0,0,0.2);
  border-radius: 5px;
  height: 100px;
}
.info h4 {
  margin: 0 0 5px;
  color: #777;
}
.legend {
  line-height: 18px;
  height: auto;
}
.legend td {
  opacity: 0.75;
}
```

Next, modify the map's instantiation from our previously created `index.html` file to match the following code snippet:

```
var map = L.map('map', {
    center: [56.8, 10.4],
    zoom: 3
});
```

Then add the following code snippets to our previously created `index.html` file, right after the `TileLayer` instantiation:

1. Our abbreviated data:

```
var europe = { … };
```

2. A custom map control to present information:

```
var infobox = L.control({
    position: 'bottomright'
});
infobox.onAdd = function (e) {
    this._div = L.DomUtil.create('div', 'info');
    this.refresh();
    return this._div;
};
infobox.refresh = function (properties) {
    this._div.innerHTML = '<h4>Europe Population Density 2009</
h4>';
    if (typeof (properties) != 'undefined') {
        this._div.innerHTML += '<b>' + properties.name + '</
b><br/>' + 'Population: ' + properties.population + '<br/>' +
'Density: ' + properties.density + '<br/>' + '<b>Click to zoom.</
b>';
    } else {
        this._div.innerHTML += '<b>Hover a country.</b>';
    }
};
infobox.addTo(map);
```

3. A couple of arrays to associate thresholds to colors:

```
var densitythresholds = [
    [0, 'rgb(237, 248, 233)'],
    [40, 'rgb(186, 228, 179)'],
    [100, 'rgb(116, 196, 118)'],
    [200, 'rgb( 49, 163,  84)'],
    [300, 'rgb(  0, 109,  44)']
];
var populationthresholds = [
    [0, 'rgb(239, 243, 255)'],
    [40000, 'rgb(189, 215, 231)'],
    [1000000, 'rgb(107, 174, 214)'],
    [10000000, 'rgb( 49, 130, 189)'],
    [50000000, 'rgb(  8,  81, 156)']
];
```

4. A function to tap into our previous arrays easily:

```
var colorByThresholds = function (thresholds, value) {
  for (var i = 0; i < thresholds.length - 1; i++) {
    if (value < thresholds[i + 1][0])
      return thresholds[i][1];
  }
  return thresholds[thresholds.length - 1][1];
};
```

5. Another custom control to present the map's legends:

```
var legendbox = L.control({
    position: 'bottomleft'
});
legendbox.onAdd = function (e) {
    this._div = L.DomUtil.create('div', 'info legend');
    var innerHTML = '<table><tr>';
    innerHTML += '<td>Population:</td>';
    for (var i = 0; i < populationthresholds.length; i++) {
        innerHTML += '<td style="background: ' + colorByThresh
olds(populationthresholds, populationthresholds[i][0]) + '">' +
populationthresholds[i][0] + (typeof (populationthresholds[i + 1])
!= 'undefined' ? '-' + populationthresholds[i + 1][0] : '+') + '</
td>';
    }
    innerHTML += '</tr><tr><td>Density:</td>';
    for (var i = 0; i < densitythresholds.length; i++) {
        innerHTML += '<td style="background: ' + colorByThre
sholds(densitythresholds, densitythresholds[i][0]) + '">' +
densitythresholds[i][0] + (typeof (densitythresholds[i + 1]) !=
'undefined' ? '-' + densitythresholds[i + 1][0] : '+') + '</td>';
    }
    innerHTML += '</tr></table>';
    this._div.innerHTML = innerHTML;
    return this._div;
};
legendbox.addTo(map);
```

6. Our first `GeoJSON` layer:

```
var densitylayer = L.geoJson(europe, {
    style: function (feature) {
        var density = feature.properties.density;
        return {
            fillColor: colorByThresholds(densitythresholds,
density),
            fillOpacity: 0.75,
            weight: 1,
            color: 'black'
        };
    },
    onEachFeature: function (feature, layer) {
        layer.on({
            'mousemove': function (e) {
                e.target.setStyle({
                    weight: 4
                });
```

```
                infobox.refresh(feature.properties);
            },
            'mouseout': function (e) {
                densitylayer.resetStyle(e.target);
                infobox.refresh();
            },
            'click': function (e) {
                map.fitBounds(e.target.getBounds());
            }
        });
    }
}).addTo(map);
```

7. And the second one:

```
var populationlayer = L.geoJson(europe, {
    style: function (feature) {
        var population = feature.properties.population;
        return {
            fillColor: colorByThresholds(populationthresholds,
population),
            fillOpacity: 0.75,
            weight: 1,
            color: 'black'
        };
    },
    onEachFeature: function (feature, layer) {
        layer.on({
            'mousemove': function (e) {
                e.target.setStyle({
                    weight: 4
                });
                infobox.refresh(feature.properties);
            },
            'mouseout': function (e) {
                populationlayer.resetStyle(e.target);
                infobox.refresh();
            },
            'click': function (e) {
                map.fitBounds(e.target.getBounds());
            }
        });
    }
}).addTo(map);
```

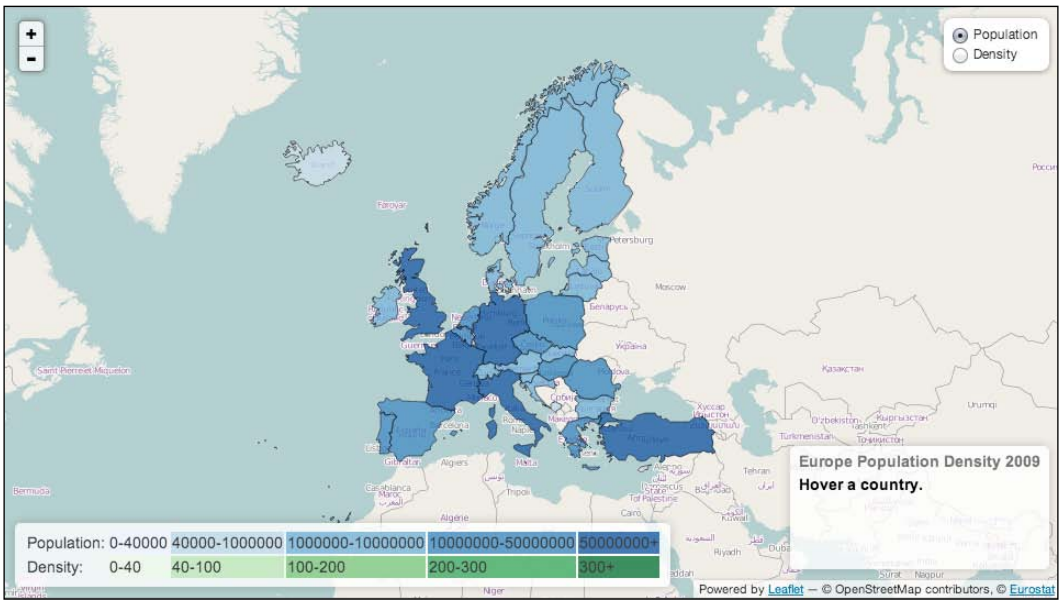
8. A control to switch `GeoJSON` layers:

```
L.control.layers({
    'Density': densitylayer,
    'Population': populationlayer
}).addTo(map);
```

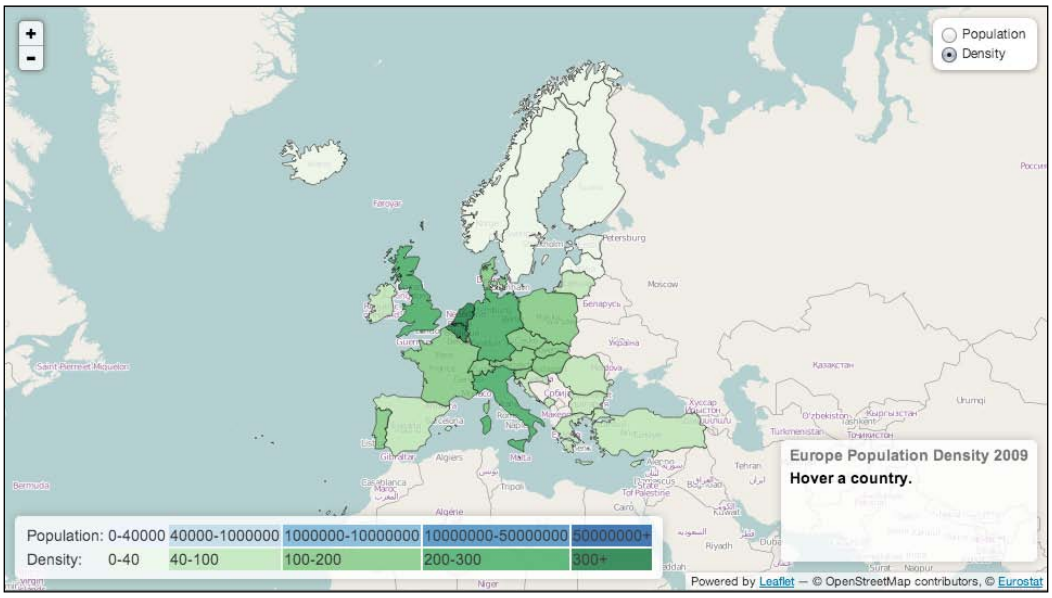
9. And last, our map attribution:

```
map.attributionControl.addAttribution('&copy; <a href="http://
epp.eurostat.ec.europa.eu/portal/page/portal/eurostat/
home/">Eurostat</a>');
```

The population layer should look like the next screenshot:



The population density layer looks like the one in the following screenshot:



How it works...

In preparation of displaying information related to the countries on the map as well as a legend, we needed to define a few styles:

```
.info {
  padding: 6px 8px;
  font: 14px/16px Arial, Helvetica, sans-serif;
  background: white;
  background: rgba(255,255,255,0.8);
  box-shadow: 0 0 15px rgba(0,0,0,0.2);
  border-radius: 5px;
  height: 100px;
}
```

The `info` class defines a box with rounded corner:

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

The `info h4` selector will grey out our info header:

```
.legend {
  line-height: 18px;
  height: auto;
}
```

The `legend` class will add some wiggle room between the lines and make its height automatic or it would have been constrained by the fixed `info` class height:

```
.legend td {
  opacity: 0.75;
}
```

The `legend td` selector will make our table cells transparent.

Next was our `GeoJSON` data describing the polygons for countries of Europe.

Then we instantiated a `Control` object as our `infobox` in which we will display information related to the hovered country:

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

We used `position Control option` to display the `infobox` at the bottom right of the map.

By itself, the control doesn't have a display, so we overrided the `onAdd` method, called when it is added to the map in which we create a `div` using the `DomUtil` JavaScript class and assign the `info` CSS class to it:

```
infobox.onAdd = function (e) {
    this._div = L.DomUtil.create('div', 'info');
    this.refresh();
    return this._div;
};
```

We also added a `refresh` method to our `infobox` that we called in the `onAdd` method to refresh the div's content:

```
infobox.refresh = function (properties) {
    this._div.innerHTML = '<h4>Europe Population Density 2009</h4>';
    if (typeof (properties) != 'undefined') {
        this._div.innerHTML += '<b>' + properties.name + '</b><br/>'
+ 'Population: ' + properties.population + '<br/>' + 'Density: ' +
properties.density + '<br/>' + '<b>Click to zoom.</b>';
    } else {
        this._div.innerHTML += '<b>Hover a country.</b>';
    }
};
```

The `infobox` method takes a `property` object instance as parameter that reflects the properties of our `GeoJSON` features. We use the properties to display the name, population, and population density in our `infobox`, or to reset its content if no properties are passed.

And last, we added the `infobox` to the map:

```
infobox.addTo(map);
```

In order to display our data with colorful visual feedback, we created two arrays of numeric thresholds, linked to shades of the same color. Color Brewer 2.0 (`http://colorbrewer2.org/`) is a nice online tool to help you create color grades for maps:

```
var densitythresholds = [
    [0, 'rgb(237, 248, 233)'],
    [40, 'rgb(186, 228, 179)'],
    [100, 'rgb(116, 196, 118)'],
    [200, 'rgb( 49, 163,  84)'],
    [300, 'rgb(  0, 109,  44)']
];
var populationthresholds = [
    [0, 'rgb(239, 243, 255)'],
    [40000, 'rgb(189, 215, 231)'],
    [1000000, 'rgb(107, 174, 214)'],
    [10000000, 'rgb( 49, 130, 189)'],
    [50000000, 'rgb(  8,  81, 156)']
];
```

Every country with a density under 40 inhabitants per square kilometer will be displayed using a color described as `rgb(237, 248, 233)`. So densities will be represented as shades of a color, and it goes the same for the population but with different thresholds linked to a different color so there is no confusion when switching layers.

Next we defined a utility function that returns a color string based on a value looked up in an array, both (value and array) passed as parameters:

```
var colorByThresholds = function (thresholds, value) {
  for (var i = 0; i < thresholds.length - 1; i++) {
    if (value < thresholds[i + 1][0])
      return thresholds[i][1];
  }
  return thresholds[thresholds.length - 1][1];
};
```

Now, our legend is also a simple control:

```
var legendbox = L.control({ position: 'bottomleft' });
```

The legendbox's `div` will also be created when it is added to the map with a table inside it, the first line being the population and the second the density. Using our threshold arrays, we loop through every field to display the values in the cells, also coloring the backgrounds:

```
legendbox.onAdd = function (e) {
    this._div = L.DomUtil.create('div', 'info legend');
    var innerHTML = '<table><tr>';
    innerHTML += '<td>Population:</td>';
    for (var i = 0; i < populationthresholds.length; i++) {
        innerHTML += '<td style="background: ' + colorByThresh
olds(populationthresholds, populationthresholds[i][0]) + '">' +
populationthresholds[i][0] + (typeof (populationthresholds[i + 1]) !=
'undefined' ? '-' + populationthresholds[i + 1][0] : '+') + '</td>';
    }
    innerHTML += '</tr><tr><td>Density:</td>';
    for (var i = 0; i < densitythresholds.length; i++) {
        innerHTML += '<td style="background: ' + colorByThresholds(den
sitythresholds, densitythresholds[i][0]) + '">' + densitythresholds[i]
[0] + (typeof (densitythresholds[i + 1]) != 'undefined' ? '-' +
densitythresholds[i + 1][0] : '+') + '</td>';
    }
    innerHTML += '</tr></table>';
    this._div.innerHTML = innerHTML;
    return this._div;
};
```

It is then added to the map:

```
legendbox.addTo(map);
```

Let's move on to our first `GeoJSON` layer showing the population density in Europe by countries. We used the `L.geoJson(…)` constructor, passing our `GeoJSON` data and an array of `GeoJSON options`:

```
var densitylayer = L.geoJson(europe, {
    style: function (feature) {
        var density = feature.properties.density;
        return {
            fillColor: colorByThresholds(densitythresholds, density),
            fillOpacity: 0.75,
            weight: 1,
            color: 'black'
        };
    },
```

```
        onEachFeature: function (feature, layer) {
            layer.on({
                'mousemove': function (e) {
                    e.target.setStyle({
                        weight: 4
                    });
                    infobox.refresh(feature.properties);
                },
                'mouseout': function (e) {
                    densitylayer.resetStyle(e.target);
                    infobox.refresh();
                },
                'click': function (e) {
                    map.fitBounds(e.target.getBounds());
                }
            });
        }
    });
```

The `style` function returns an array of graphics options, especially the color of the polygon based on the density found in the feature properties using our `colorByThresholds` utility function and our `densitythresholds` array.

The `onEachFeature` function will add mouse events to each layer created for our features. The `mousemove` event will add weight to our polygon's borders, making the country stand out, and will refresh the content of the `infobox`. The `mouseout` will reset the polygon's style and the infobox's content. The `click` event will zoom into the map to the country using the `Map.fitBounds(…)` method with the bounds of the polygon.

The population layer is handled mostly the same way, changing the `populationthresholds` array instead of the `densitythresholds` array and the `population` property instead of the `density` property as parameters to the `colorByThresholds` function in the `style` function and adding it to the map to make it the default base layer.

Then, we added the standard layers control to the map to make our population and density layers mutually exclusive:

```
L.control.layers({
    'Density': densitylayer,
    'Population': populationlayer
}).addTo(map);
```

And finally, we added the standard attribution control to the map referring to the data provider:

```
map.attributionControl.addAttribution('&copy; <a href="http://epp.
eurostat.ec.europa.eu/portal/page/portal/eurostat/home/">Eurostat</
a>');
```

Thank you for buying Instant Interactive Map Designs with Leaflet JavaScript Library How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

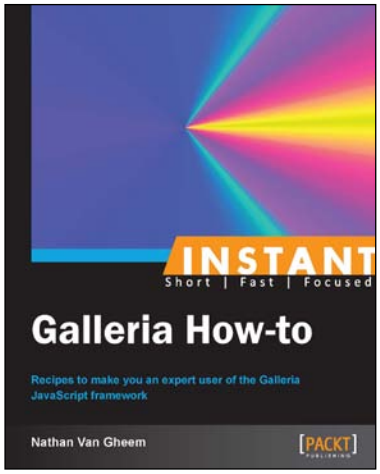
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING

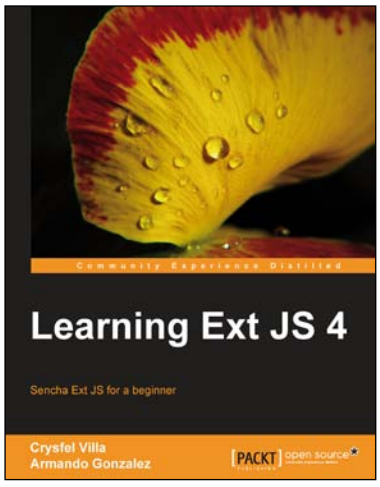


Instant Galleria How-to

ISBN: 978-1-849696-60-9 Paperback: 70 pages

Recipes to make you an expert user of the Galleria JavaScript framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Integrate and create themes
3. Extend Galleria using the API
4. Learn how to create mobile friendly galleries



Learning Ext JS 4

ISBN: 978-1-849516-84-6 Paperback: 434 pages

Sencha Ext JS for a beginner

1. Learn the basics and create your first classes
2. Handle data and understand the way it works, create powerful widgets and new components
3. Dig into the new architecture defined by Sencha and work on real world projects

[PACKT]
PUBLISHING

Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-849516-86-0 Paperback: 488 pages

Over 110 easy-to -follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

---

Instant PhpStorm Starter

ISBN: 978-1-849693-94-3 Paperback: 86 pages

Learn professional PHP development with PhpStorm

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Learn PHPStorm from scratch, from downloading to installation with no prior knowledge required
3. Enter, modify, and inspect the source code with as much automation as possible
4. Simple, full of easy-to-follow procedures and intuitive illustrations, this book will set you speedily on the right track