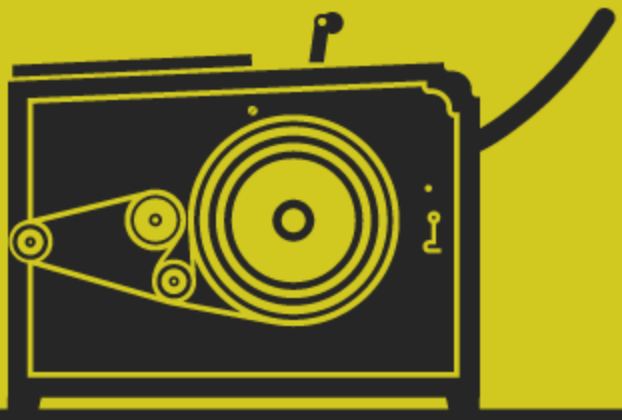


Leaflet.js

Succinctly

by Mark Lewin

Syncfusion | Technology Resource Portal

Leaflet.js Succinctly

By
Mark Lewin

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Zoran Maksimovic

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Mark Lewin has been developing, teaching, and writing about software for over 16 years. His main interest is web development generally and web mapping in particular. Working for ESRI, the world's largest GIS company, he acted as a consultant, trainer and course author and frequent speaker at industry events. He has subsequently expanded his knowledge to include a wide variety of open-source mapping technologies and a handful of relevant JavaScript frameworks including Node.js, Dojo and JQuery.

Mark currently teaches ArcGIS Server development for Geospatial Training LLC (http://www.geospatialtraining.com) and is the author of the Google Maps API:Get Started course for Pluralsight (https://www.pluralsight.com/courses/google-maps-api-get-started).

He blogs about web map development at http://www.appswithmaps.net and can be reached at mark@appswithmaps.net or on Twitter at @gisapps.

Introduction

There has always been something magical about maps. Ever since the first human beings scratched out an image in the dirt to share the location of the good hunting grounds, we have become fascinated with maps and just how much information they can impart. Maps have progressed hugely in both accuracy and sophistication over the centuries, but remained largely paper-based until very recently, with the advent of the Internet. The Internet is the perfect medium for sharing maps and for finding lots of interesting spatial data from which to create maps.

Not so long ago, the prospect of putting any form of interactive map on the web was enough to give even the most seasoned developer nightmares. Nowadays, we have some amazing tools that make it easy for even inexperienced developers to create highly sophisticated web-mapping applications with little more than a text editor and an Internet connection. One of the newest and best technologies for creating stunning and highly functional web mapping applications is Leaflet.js. With just a little knowledge of HTML, CSS, and some basic JavaScript, along with this guide, you have all you need to get started.

Chapter 1 Getting to Know Leaflet.js

What is Leaflet.js?

Leaflet.js is a super-lightweight mapping library. It consists of approximately 34KB of compressed JavaScript with absolutely no external dependencies. What makes Leaflet.js really remarkable is that it manages to do this without sacrificing features. Whereas many frameworks quickly become bloated and complex over time, Leaflet.js is still relatively new on the scene, and was designed from the ground up by developer Vladimir Agafonkin to be simple, intuitive, and easy to use.

Leaflet.js is open source and has a very vibrant and active developer community. Although the core library is very small by design (to avoid your application having to download lots of code it does not rely on), there are numerous plugins you can use to extend the functionality of your web-mapping applications. So if you are looking at another web mapping framework and thinking, “Wouldn’t it be cool if Leaflet.js did that?” you can be almost certain that someone, somewhere thought the same thing and wrote a plugin for it.

I could go on about the great design of Leaflet.js and its amazing capabilities, but this is a *Succinctly* guide, and we want you to get up and running straight away—and help you discover these things for yourself. So let’s get started.

What do you need to get started?

Basically, not much. You need a text editor to write your code; any text editor that you are familiar with will work just fine. I’m using a cross-platform editor called Brackets that is specifically designed for web development by Adobe, but you can use anything you like. Try and get one that supports syntax highlighting for HTML, CSS, and JavaScript, because it will help you make sense of your code and quickly spot any syntax errors. Notepad++ is a good choice for Windows users.

You’ll need a web browser. Again, use your favorite, but I’d suggest choosing either Mozilla Firefox or Google Chrome, simply because they have such excellent developer tools. These can really help when you’re scratching your head over why some feature of your application does not work and JavaScript is not being forthcoming about the nature of the error.

I also recommend that you use a web server to host your HTML pages and JavaScript code. Although it’s possible to run most of the samples in this book by opening the code file directly in the browser, you’ll want to use a web server for anything more than trivial experimentation. I use XAMPP, which is super-easy to configure and use, and available on both Windows and Mac. XAMPP provides an installer to set up a local Apache web server, as well as other components you might find useful as you start to expand upon your knowledge of Leaflet.js. These include MySQL (the world’s most popular open source web database) and MariaDB, a drop-in replacement for MySQL, and the PHP language for server-side scripting.

Other options include WAMP (for Windows) and MAMP (for Mac OS).

Just run the installer for your chosen solution, start it up, and note where the server's *document root* is, and which *port* the web server is running on. The document root is the location on your file system where you will put the HTML and JavaScript (and PHP, if you follow along with the examples in Chapter 5) files you create. When you try to access a specific page via your web browser, the web server looks in the document root for the corresponding file. You will need to enter the port number as part of the URL so that your chosen web server can intercept any requests.

Launch the application's GUI control panel and make sure the Apache and MySQL services are running. (MySQL is optional here, but you'll need it later on if you want to run the database samples in Chapter 5.) This is what it looks like in XAMPP:

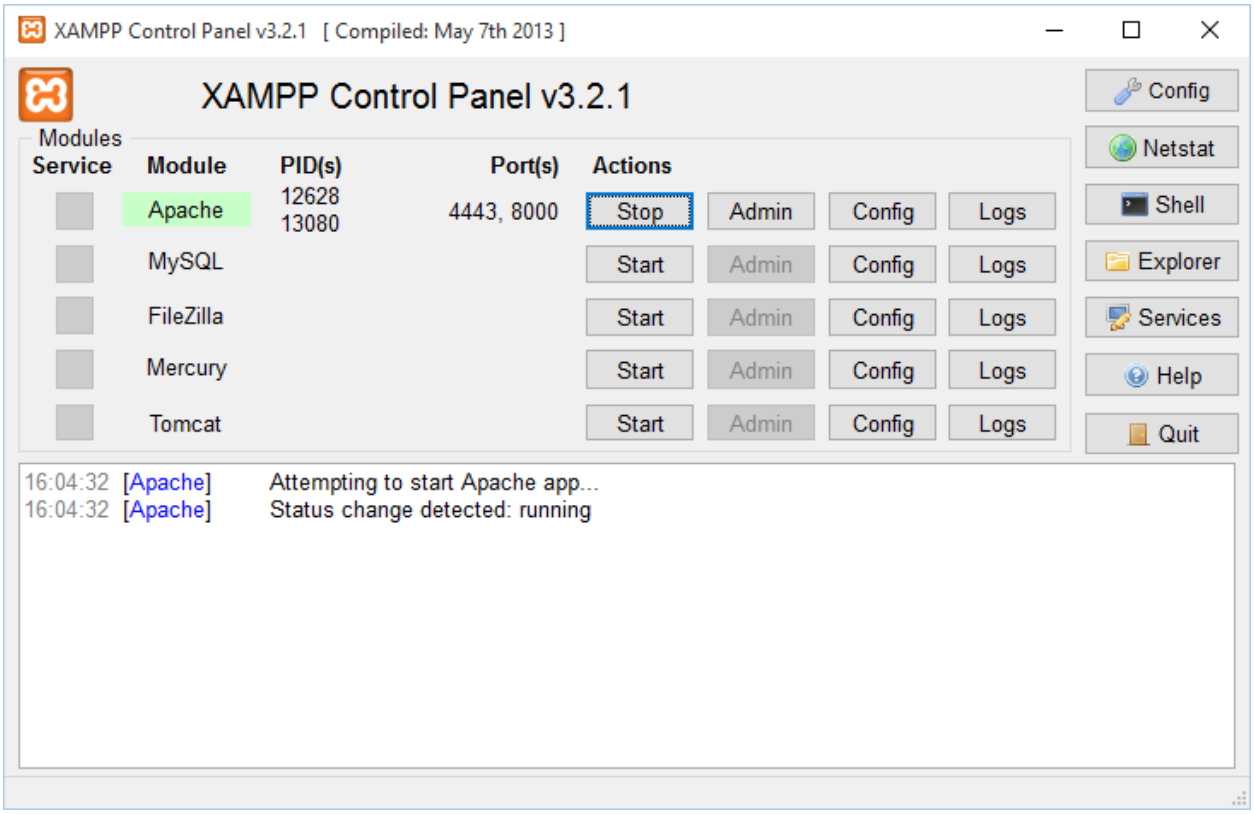


Figure 1: XAMPP control panel

I'm using a slightly older version of XAMPP. The most recent version of XAMPP contains the MariaDB database system rather than MySQL. The two DB systems are binary-compatible, so you'll have no trouble using MariaDB if you want to follow the code examples in this e-book. In my environment, I have Apache running and listening on port 8000. The default port for Apache is 80, but I have another web server using that port, so I used the "Config" option to change the port number. I can visit http://localhost:8000 in my web browser and see the confirmation page:

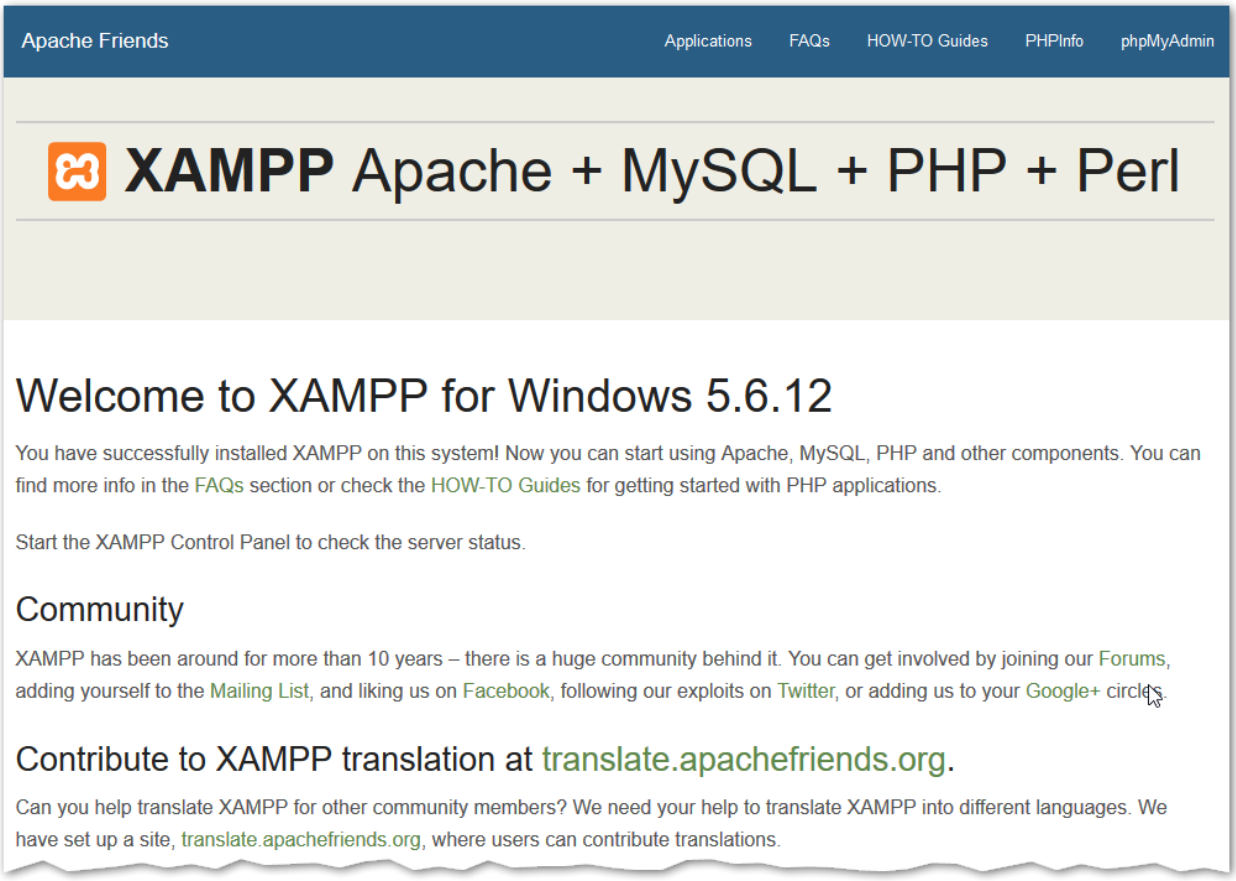


Figure 2: XAMPP/Apache confirmation page

Apache requires that you place your web pages in its **`htdocs`** folder, which, in my XAMPP environment, is under **`C:\xampp\`**:

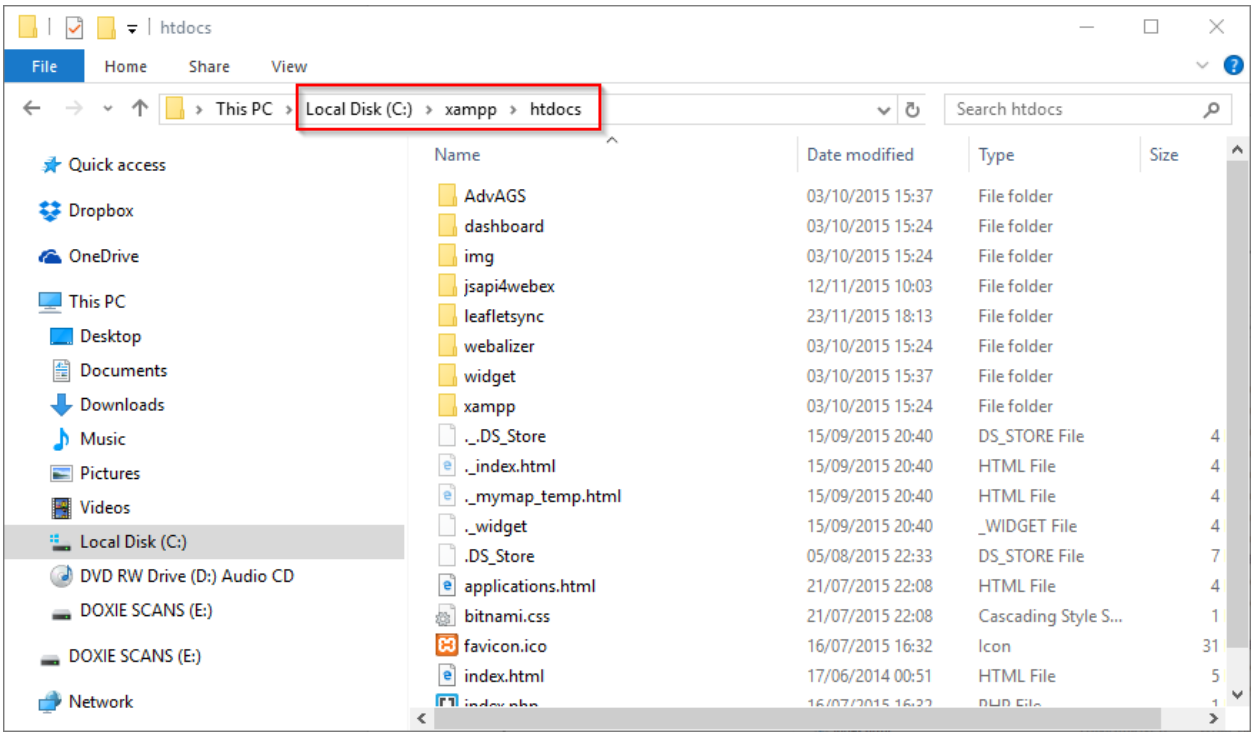


Figure 3: Where to put your code files

Any **`.html`**, **`.js`**, or **`.php`** pages you place in the root directory can be accessed via your browser at *http://localhost:<port number>/<page name>*.

The source code for this book

The source code for all the code listings in this book, along with any relevant images and other files, can be downloaded here.

Each code listing in the book relates to a file that is numbered accordingly, within a folder named after the chapter it appears in. For example, the source code for *Code Listing 1* can be found in the directory called **`Ch01_gtk`**, and is called **`listing01.html`**.

I have also included the source code for the two "helper" applications I refer to in this book. They can be found in the **`extentfinder`** and **`mapdraw`** directories within the zip file, or accessed online at the following locations:

- Extent Finder
- Map Draw Tool

Creating a simple map in a web page

You are going to create a simple interactive web map in just a few lines of code. Because this is a web map, it needs to be hosted in a web page. So open up your text editor of choice and create a file called **mymap.html**. Enter the following HTML markup into the file:

Code Listing 1: Basic HTML Web Page

```
<!DOCTYPE html>
<html>
<head>
  <title>My Leaflet.js Map</title>
</head>
<body>
</body>
</html>
```

Steps to create the map

Perform the following steps to create your first map:

1. Reference the Leaflet.js library and CSS.
2. Create a ** ** element on your page where the map will appear.
3. Create a **map** object.
4. Add a layer to the map.

Referencing Leaflet.js in your code

There are two ways to use Leaflet.js in your applications:

1. Reference a hosted version of the library via a Content Delivery Network (CDN).
2. Download the Leaflet.js library and host it on your own web server.

Accessing Leaflet.js via a CDN

To access Leaflet.js via a CDN (Content Delivery Network), reference its CSS by using a **<link>** tag, and the JavaScript library by using a **<script>** tag. Both of these tags should appear in the **<head>** section of your HTML page, right after the **<title>** tag:

Code Listing 2: Accessing Leaflet.js via a CDN

```
...
<head>
  <title>My Leaflet.js Map</title>
    <link rel="stylesheet"
      href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css
" />
```

```
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js">
    </script>
</head>
...
```

Note: The URLs shown in Code Listing 2 should reference the latest stable version of Leaflet.js, which at the time of writing, is 0.7.3. Check the Leaflet.js website for the latest available stable version.

We'll be using a CDN for the examples in this book.

Accessing Leaflet.js from your own web server

If you want to host Leaflet.js yourself, you have a couple of options:

1. Download a pre-built package from the Leaflet.js website to your document root, and set the appropriate path in the **<script>** and **<link>** tags. The package archive contains the minified JavaScript code (**leaflet.js**) and an unminified version called **leaflet-src.js**, which can help when debugging. It also includes the required CSS.
2. Download the source code from GitHub and build it using Node.js. This gives you access to absolutely everything, including the original source code, unit tests, and so on. You'll need to do this if you want to get a good look at the internals of Leaflet.js, and perhaps even add new features to make it even better for the rest of us! See the Leaflet.js website for full instructions.

Creating a element for your map

Before you write any Leaflet.js code, you need to create a space on your page to host the map. Just create a ** ** element, and give it a suitable **id** attribute so that you can refer to it in your code later on. You must also specify (at a minimum) a **height** attribute for the ** **. If you don't specify a **height** attribute, the map won't display. Specify a **width** attribute as well, if you want. If you choose not to, then the map will span the entire width of its parent element (which in the following example is the **<body>** tag).

Code Listing 3: Creating and sizing the map

```
<!DOCTYPE html>
<html>
<head>
  <title>My Leaflet.js Map</title>
    <link rel="stylesheet"
      href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css
" />
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js">
    </script>
    <style>
```

```
      #map {
        height: 400px;
      }
    </style>
</head>
<body>
  <div id="map"></div>
</body>
</html>
```

But that's a lot of empty page and a tiny map, right? What if you want to make the map occupy the entire page? Just make sure the map ** ** and its parent elements have their **height** attribute set to 100%.

Code Listing 4: Making the map full screen

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

Creating the map object

Now that we have a place for the map on our page, we need to create the Leaflet.js **map** object. To do that, we need to write a JavaScript function and wait for all the elements in the page to load before we call it. If we don't wait, then our map ** ** might not yet exist when we attempt to reference it in our code, and we'll get an error.

Create a **<script>** tag to host your JavaScript code, and stub out a function that will be called when the DOM has finished loading:

Code Listing 5: Creating a function that waits for all page elements to load

```
...
    <script type="text/javascript">
        function init() {
          alert("I'm ready!");
        }
    </script>
</head>
<body onload=init()>
      <div id="map"></div>
</body>
</html>
```

Save it, launch it in your web browser, and confirm that the **`init()`** function gets called and displays the "I'm ready!" message. If that's working correctly, we can use it to create the **`map`** object and tell it which **`<div>`** to display in. Replace the code within the **`init()`** function with the following:

Code Listing 6: Creating the map object

```
...
    <script type="text/javascript">
        function init() {
          var map = L.map('map').setView([51.73, -1.28], 12);
        }
    </script>
...
```

So what's going on here?

- **`L.map('map')`** is the Leaflet.js map constructor. All Leaflet.js objects are prefixed with **`L`**. The constructor takes two arguments: the HTML element where the map will display (in our example, that is **`map`**), and (optionally) an object literal that defines specific options for the map's appearance and behavior. We haven't specified a map options object in this example. See the full range of map options in the Leaflet.js docs.
- **`.setView(…)`** Once we have the map object, we call its **`setView()`** method to pass in the *extent* of the map. The extent is the visible area we want to display. The **`setView()`** method takes two parameters. The first parameter is an array that represents the latitude and longitude of the map's center, and the second is a number that represents the zoom level.
- The **`map`** object we defined with these two methods is then stored in our local JavaScript variable **`map`**.

If we save our file and launch it then we see…precisely nothing. That's because although we have a map, we haven't yet told it what data to display. We'll do that next.

Add a layer to the map

Our map object is just a container. To actually have our map display something we need to add one or more *layers*.

Typically, a web map will have multiple layers. These will include base maps, which show terrain, roads, or other contextual information; and operational layers, which contain any data of particular interest to our users, such as the location of coffee shops, restaurants, or hiking trails. Usually, base maps are hosted by a third party. Operational data can also come from other sources, but is often something we provide ourselves, and can be either hard-coded into the application, loaded at runtime from a database, or entered by our users.

Considering maps as discrete layers allows a great deal of flexibility, because it allows us to show or hide information depending on what our users want to see. For example, if our users only want to see restaurants, then we can hide our “coffee shops” layer. And if they are not in the market for refreshment but want to head for the hills instead, then we can hide our “restaurant” and “coffee shops” layers and swap the street map for some satellite imagery so they can better understand where those trails lead.

Unlike, for example, Google Maps, where you only have access to Google’s own base maps, we can use an astonishing array of different base maps with Leaflet.js. Some are widely available; others need more effort to track down. With the aid of some specialist software like GeoServer or MapServer, we can even create and host our own base maps.

We’ll get more into map layers in the next chapter. For now, let’s just add a single base map layer so we can have something to play with. We’ll use one from OpenStreetmap, a wonderful map made entirely from crowd-sourced map data.

Enter the following code into the **`init()`** function to add an OpenStreetMap layer to the map:

Code Listing 7: Adding a base map layer

```
...
  <script type="text/javascript">
    function init() {
      var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>"
          var map = L.map('map').setView([51.73, -1.28],12);
          L.tileLayer(
              'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
              attribution: 'Map data &copy; ' + osmLink,
              maxZoom: 18,
          }).addTo(map);
      }
  </script>
...
```

Here we define a new layer of type **`TileLayer`**. We use a **`TileLayer`** because most base maps consist of a massive image that is sliced and diced into tiles so that the browser only has to download the image tiles it needs for a particular extent. We’ll talk about layers in more detail in the next chapter.

The **`TileLayer`** object’s constructor takes as its first parameter a URL that tells our application where to get the tile data from. The **`{s}`**, **`{z}`**, **`{x}`**, and **`{y}`** placeholders in the URL represent the following:

- **`{s}`** allows one of the subdomains of the primary domain to serve tile requests, enabling our application to make multiple requests simultaneously, and thereby download tiles much faster.
- **`{x}`** and **`{y}`** define the tile coordinates.
- **`{z}`** represents the zoom level.

Thankfully, we don't have to worry about populating those placeholders ourselves; Leaflet.js takes care of that for us.

The second parameter is a JavaScript object literal that defines any options we want to set to configure the layer. In this instance we have provided some attribution data and the maximum zoom level we want our users to be able to zoom into when this layer is active.

Finally, we call the **`TileLayer.addTo()`** method and pass in the **`map`** object that we want this layer to appear in.

Save your code and navigate to the **`mymap.html`** page in your web browser. If you're using the same set up as me, that will be at http://localhost:8000/mymap.html. You should see the following:

Figure 4: Your first Leaflet.js map

The map shows OpenStreetMap data for an area around Oxford in the United Kingdom. To interact with the map:

Zooming:
- Use the plus/minus buttons in the top left-hand corner of the map.
- Use your mouse scroll wheel.
- Shift+click and drag with the mouse (this is known as a "box zoom").

Panning:
- Click on the map and drag the mouse.

Congratulations! You have created your first Leaflet.js map and are now officially a neogeographer. (See Wikipedia for more information on the club you have just joined.)

But Oxford, England is probably not a location that means very much to you. So, as an exercise, change the extent of the map to focus on your hometown. This requires knowing the latitude and longitude of the center of your desired extent. Since most people don't carry that sort of information around in their heads, I have written a little helper application you can access here. Just locate your desired extent and click **Get Map Code**. When the dialog box appears, copy the code it contains and use it to replace the map constructor code in your application.

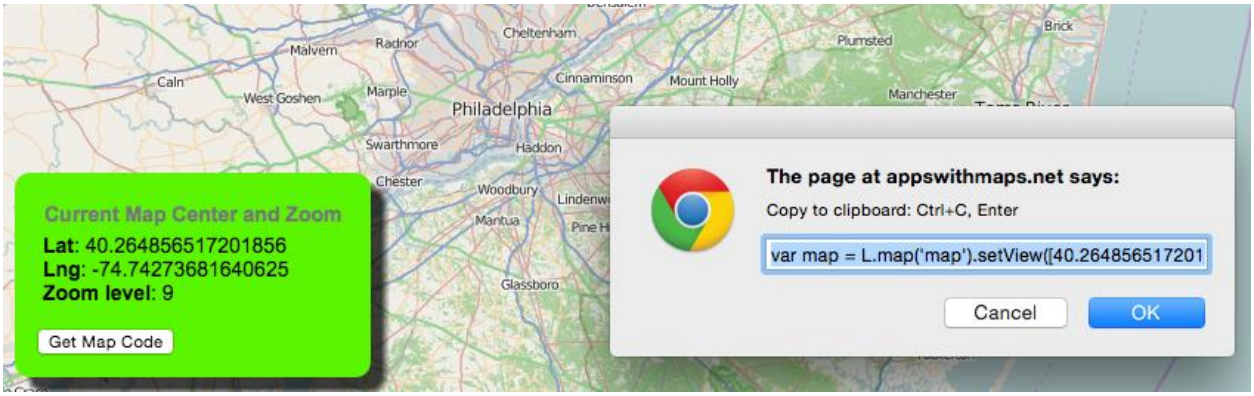


Figure 5: Using the extent finder application

Note: The extent finder application is accurate to far too many decimal places. You don't normally need that kind of accuracy; anything more than three decimal places for a latitude or longitude coordinate is probably overkill.

In the next chapter we'll look at map layers in more detail.

Chapter 2 Working with Base Layers

Leaflet.js distinguishes between two basic types of map layers: *base layers* and *overlays*. The tile layer we added to our map in the previous chapter is an example of a base layer.

Base layers are maps, typically sourced from external providers like OpenStreetMap or MapBox, that often contain rich and complex cartography and serve to set the geographical context for the other data you add to the map. That "other data" is usually an overlay. There is only ever one base map layer visible at any one time, but you can have multiple overlays "draped" over the base layer.

To understand the difference between the two types of layers, you need to think about the ways in which a computer deals with graphics. The two basic data structures for storing and manipulating graphical data on a computer are *rasters* and *vectors*. So far we have seen how to create a map control and add a single tile layer to use as a base map layer. A tile layer is an example of a raster layer. Rasters are comprised of individual pixels and are what we typically refer to as an "image."

Overlays are typically vectors. Vectors are made up of information that describes points and lines that are geometrically and mathematically related to one another. They don't really become images until something draws them.

We'll talk about vectors when we cover overlays in Chapter 3. For now, let's have a look at some of the different types of base layers, starting with the **`L.TileLayer`** we saw in the previous chapter.

Basic tile layers (TileLayer)

These layers consist of image tiles that are requested by your application when a user focuses on a particular extent of the map. At large scales (user zoomed in to a small geographic area), there are a large number of tiles, because there is more detail to display per square mile. At smaller scales (user zoomed out to a wide geographic area), there are fewer tiles, because there is less detail to display per square mile. Leaflet.js and the tile layer provider calculate which tiles need to be shown for a particular extent based on the URL you specify in the **`TileLayer`** constructor, which contains the **`{x}`**, **`{y}`**, and **`{z}`** placeholders. These placeholders correspond to the tile coordinates and zoom level that the tile service uses to locate a specific tile. Some providers also accept a **`{s}`** placeholder, which can be used to switch between subdomains, and therefore download tiles in parallel for better performance.

What follows is a roundup of some of the more popular tile providers, together with the URL you must use in the **`TileLayer`** constructor. Some providers will have usage policies to constrain bandwidth. Just about all of them will require some sort of attribution, which you can set with the **`attribution`** property. This property accepts just about any valid HTML, so you can hyperlink back to the tile provider's website if you wish. See the providers' individual websites for specific attribution details.

OpenStreetMap

OpenStreetMap is an amazing resource, consisting of crowd-sourced mapping for much of the world's surface.

OSM Mapnik

OSM Mapnik is the standard way of viewing OpenStreetMap data, which we used in Chapter 1.

URL template: `http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png`

Example:

Code Listing 8: OSM Mapnik tile layer

```
...
// OSM Mapnik
var osmLink =
"<ahref='http://www.openstreetmap.org'>OpenStreetMap</a>";
L.tileLayer(
    'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
    attribution: '&copy; ' + osmLink,
    maxZoom: 18,
}).addTo(map);
...
```

OSM Black and White

This is a black and white/grayscale version of OSM Mapnik, and it's great if you want to superimpose your own data points on the map and make them easily visible.

URL template: **`http://{s}.www.toolserver.org/tiles/bw-mapnik/{z}/{x}/{y}.png`**

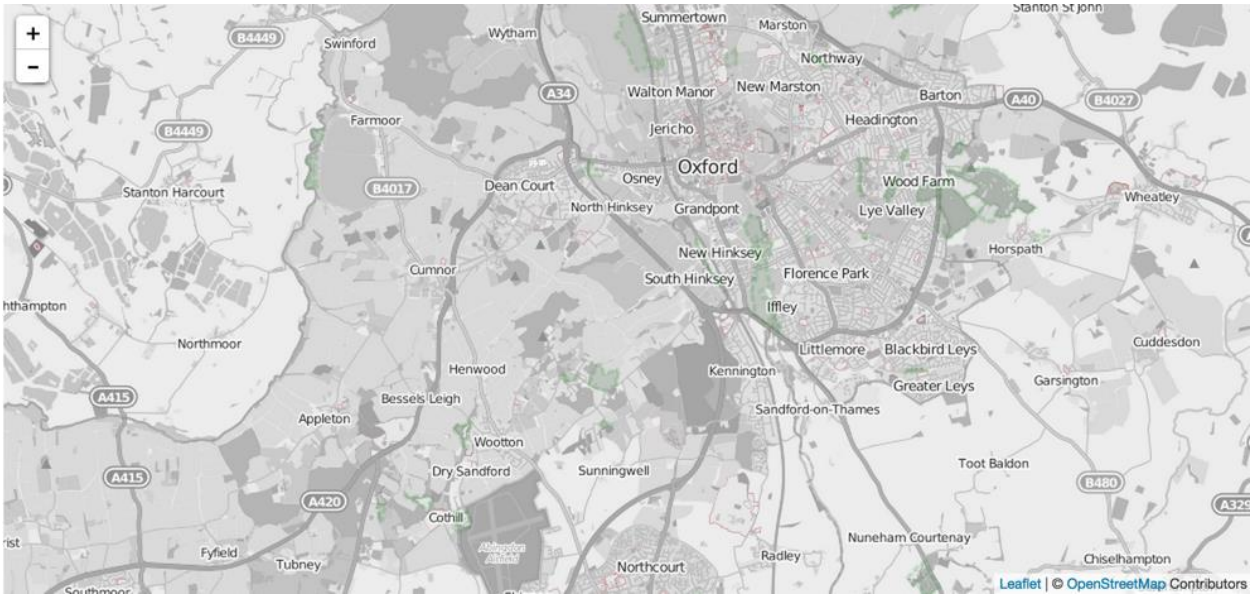


Figure 6: OSM Black and White

Code Listing 9: OSM Black and White

```
...
// OSM Black & White
var osmLink = "<a href='http://www.openstreetmap.org'>Open StreetMap</a>";
L.tileLayer(
    'http://{s}.www.toolserver.org/tiles/bw-mapnik/{z}/{x}/{y}.png', {
    attribution: '&copy; ' + osmLink,
   maxZoom: 18
}).addTo(map);
...
```

Thunderforest

Thunderforest is owned by Andy Allan, a digital cartographer best known for OpenCycleMap. Thunderforest now offers a range of different tile layers which, like OpenCycleMap, are based on OpenStreetMap. Check out the Docs page here.

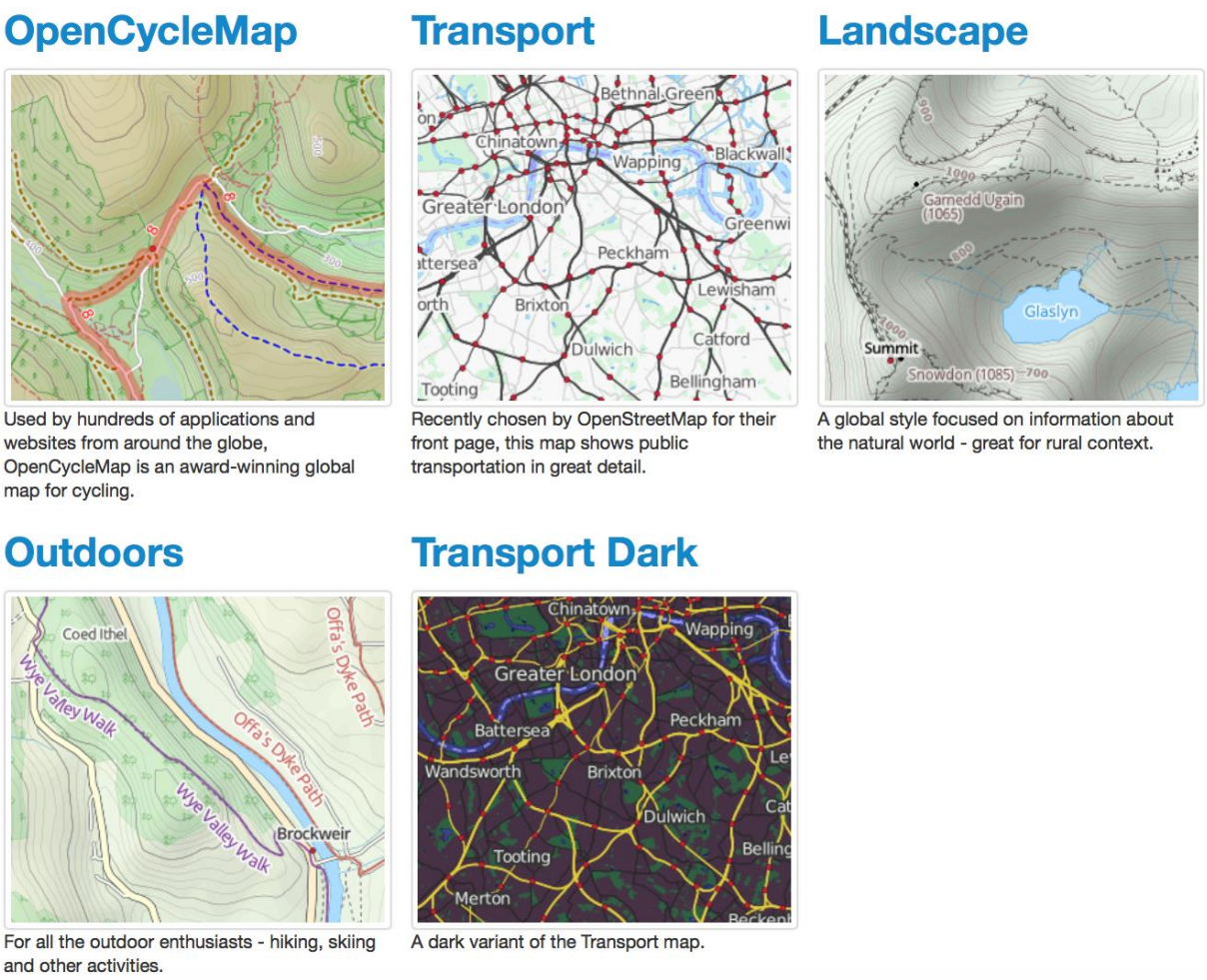


Figure 7: Tile Layers provided by www.thunderforest.com

URL template: **`http://{s}.tile.thunderforest.com/<tilelayer>/{z}/{x}/{y}.png`**

Where possible, entries for **`<tilelayer>`** are:

- **`cycle`**
- **`transport`**
- **`landscape`**
- **`outdoors`**
- **`transport-dark`**

Stamen

Stamen provides a number of gorgeous tile layers. Some of these are widely useful, such as their beautifully-shaded Terrain (USA only) and high-contrast Toner layers. Others, such as their sublime Watercolor map, are perhaps less immediately useful but very good fun to play with.

URL template: **`http://{s}.tile.stamen.com/<tilelayer>/{z}/{x}/{y}.<jpg/png>`**

We'll have a look at some of Stamen's tile layers later in this chapter.

Other tile layer providers

There are several other tile providers we don't space to do justice to here. However, hopefully you will have seen by now that the process for using tile layers is very similar, regardless of the provider.

Check out these providers for starters, and look at their documentation for the correct method to add their tile layers to your Leaflet.js maps:

- Esri**:** The world's largest Geographical Information Systems company. They have a selection of very high-quality tile layers, available here.

- **MapQuest**: MapQuest offers a couple of very useful tile layers. The MapQuest Open Aerial tiles consist of aerial imagery that covers the globe at various different levels of detail. The MapQuest OSM tiles are a variation on OpenStreetMap data with different styling. MapQuest makes these tile layers easy to use via a Leaflet plug-in.

- **Cloudmade** and **MapBox**: Both of these companies offer tile layers. Leaflet's creator, Vladimir Agafonkin, now works for MapBox, and the Leaflet.js framework is a central part of MapBox's services.

WMS (Web Map Service) tile layers (TileLayer.WMS)

Another option you can consider when you are looking for basemaps for your application is a WMS (Web Map Service) layer. WMS is a recognized standard for serving map imagery over the web, defined by the Open Geospatial Consortium. Unlike the tile layers we have seen so far, WMS imagery is generated dynamically by a map server, such as ArcGIS Server, MapServer, or GeoServer. The map server queries a spatial database that contains all the data required to generate the map dynamically, creates the required images and then sends them down to the browser for display. The benefits of dynamically generating imagery in this way (as opposed to using pre-created map tiles) include not being confined to the zoom levels at which the tiles were created, being able to choose which data (layers) within the map service to display, and the fact that you can request transparent images. Using transparency lets you stack layers without the top-most layer obscuring the ones beneath it.

With it being an industry standard, there are a lot of WMS services out there, but admittedly some of them are quite hard to find. The exciting thing about this approach is that, with a bit of know-how and freely available software such as QGIS (for map creation), PostGIS (for the spatial database) and GeoServer (to act as a WMS server), you can create your own basemap services. So, if you've ever wanted to create a web map of Westeross in the *Game of Thrones* series, knock yourself out!

One of the best places I've found on the web to locate WMS services is the Spatineo Directory. You can search by keyword, geographic area, and layer type (WMS). Note that there is a definite U.S. bias here. The United States seems keener to share geographic information than, for example, Europe, where I live.

Here is the code for using a WMS layer (**`TileLayer.WMS`**) in Leaflet.js:

Code Listing 10: WMS tile layer

```
...
// Esri World Imagery Service
L.tileLayer(
    'http://server.arcgisonline.com/ArcGIS/rest/services/\
    World_Imagery/MapServer/tile/{z}/{y}/{x}', {
    attribution: '&copy; ' + mapLink + ', ' + sourcesLink,
    maxZoom: 18,
}).addTo(map);
// US States WMS Layer
L.tileLayer.wms("http://demo.opengeo.org/geoserver/wms", {
    layers: 'topp:states',
    format: 'image/png',
    transparent: true,
    attribution: "<a href='http://www.boundlessgeo.com'>OpenGeo</a>"
}).addTo(map);
...
```

The main differences between **`TileLayer.WMS`** and **`TileLayer`** are:

- The URL does not use placeholders like the standard tile layer URLs. Each URL is different depending on who served it and how, so you'll need to do a bit of research into a WMS layer before you use it. (This typically involves issuing a **`GetCapabilities`** request on the service and deciphering the XML it returns. See this tutorial for more information.)
- You need to find out which layers are available *within* the WMS tile layer and list the ones you want in the **`layers`** property, separated by commas. (Again, this information is available via a **`GetCapabilities`** request.)
- You can set the **`transparent`** property to **`true`** so that you can see your other map layers in regions where there is no data for the WMS layer. For this to work, the image **`format`** property must be set to **`image/png`**.

In *Figure 6*, I am using a sample GeoServer WMS layer from an instance hosted by Boundless showing U.S. state polygons, and superimposing this on the Esri World Imagery tile layer.

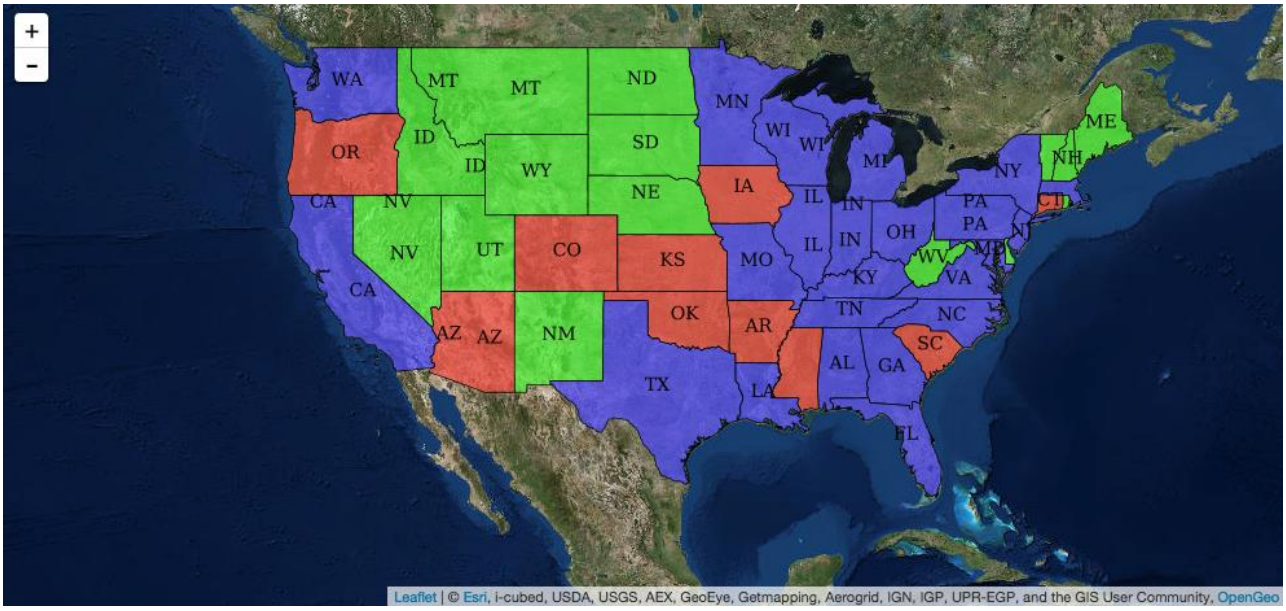


Figure 8: WMS layer showing US states

Switching between multiple tile layers

Wouldn't it be nice to allow your users to select from a range of base maps, rather like Google Maps does? Well, Leaflet.js makes it easy. We can use one of the framework's built-in controls: the **`layers`** control. Let's do that.

If you want to follow along, make a new file in your web server document root called **`basemaps.html`**. Copy the code from **`mymap.html`** and paste it into **`basemaps.html`**. Remove the script block that contains your **`init()`** function declaration and replace it with the code in Listing 10.

Code Listing 11: Basemap switching

```
...
        function init() {
            map = L.map('map').setView([37.42, -122.05], 12);
            attrLink = 'Map tiles by <a href="http://stamen.com">Stamen
Design</a>, under <a href="http://creativecommons.org/licenses/by/3.0">CC
BY 3.0</a>. Data by <a href="http://openstreetmap.org">OpenStreetMap</a>,
under <a href="http://creativecommons.org/licenses/by-sa/3.0">CC BY
SA</a>.'
            attrLinkToner = 'Map tiles by <a
href="http://stamen.com">Stamen Design</a>, under <a
href="http://creativecommons.org/licenses/by/3.0">CC BY 3.0</a>. Data by <a
href="http://openstreetmap.org">OpenStreetMap</a>, under <a
href="http://www.openstreetmap.org/copyright">ODbL</a>.';
            var terrainMap = L.tileLayer(
                'http://{s}.tile.stamen.com/terrain/{z}/{x}/{y}.jpg', {
```

```
                    attribution: attrLink,
                    maxZoom: 18,
                }).addTo(map);
            var tonerMap = L.tileLayer(
                'http://{s}.tile.stamen.com/toner/{z}/{x}/{y}.png', {
                    attribution: attrLinkToner,
                    maxZoom: 18,
                }).addTo(map);
            var watercolorMap = L.tileLayer(
                'http://{s}.tile.stamen.com/watercolor/{z}/{x}/{y}.jpg', {
                    attribution: attrLink,
                    maxZoom: 18,
                }).addTo(map);

            var baseLayers = {
                "Stamen Terrain": terrainMap,
                "Stamen Toner": tonerMap,
                "Stamen Watercolor": watercolorMap
            };
            L.control.layers(baseLayers).addTo(map);
        }
...
```

The things to note here are as follows:

1. We have added three tile layers from Stamen Design.
2. We have assigned each tile layer constructor to a variable so that we can reference it in later code.
3. We created a simple JavaScript object literal called **`baseLayers`**, which consists of three key/value pairs: the name of the layer as we want it to appear in our **`layers`** control followed by the layer's variable name.
4. We then created a layers control, passing in the **`baseLayers`** object to tell it which layers to work with, and then immediately called .**`addTo()`** to add the **`layers`** control to our map.

View the page in the browser. You'll notice that a little control appears in the top right-hand corner of the map. Click the control and the layer selection list appears, configured as you specified in the **`baseLayers`** object. You can now choose which basemap layer to display.

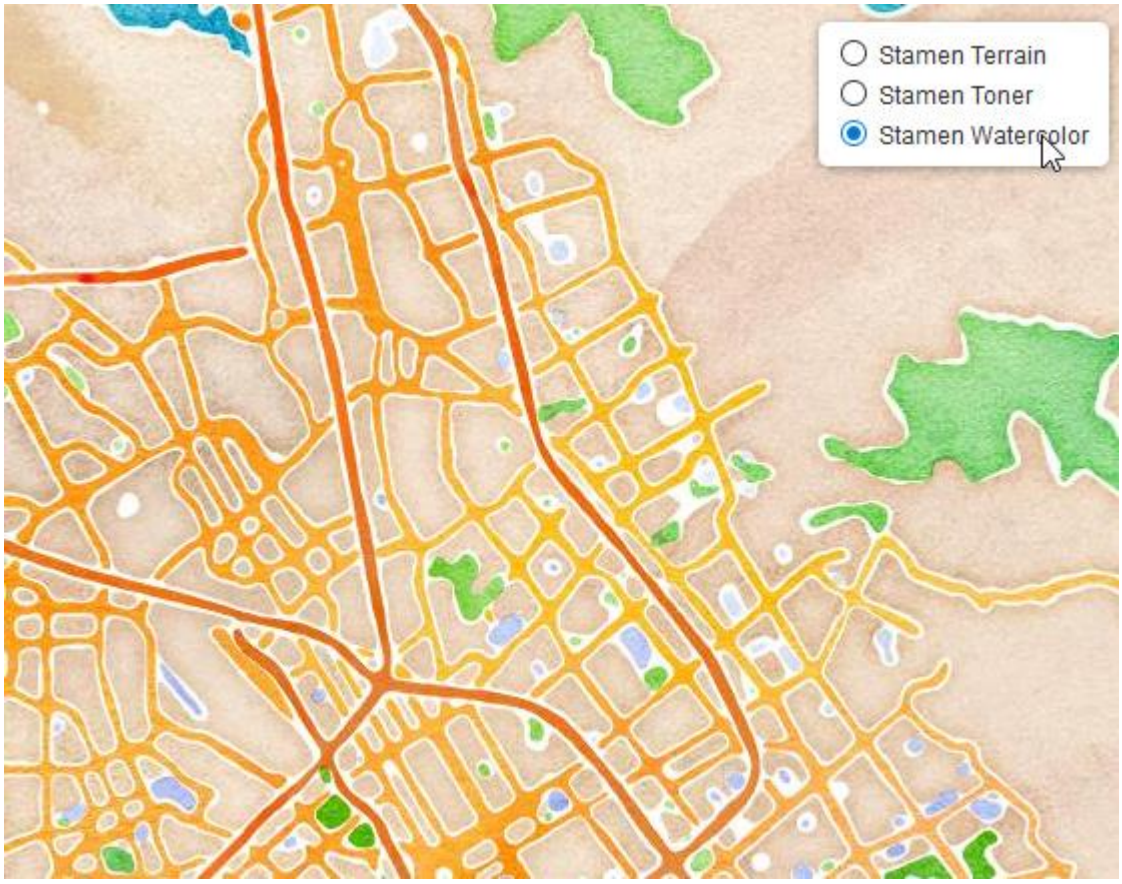


Figure 9: The layers control in action, showing the Stamen Watercolor tiled map service

Chapter 3 Adding Overlays

In the last chapter, we made the distinction between base maps and overlays. Base maps provide a "background" for the data that forms the basis of your application, which is typically in the form of an overlay.

This data can come from anywhere. You might supply it yourself (for example, if you run a chain of bicycle stores and want your users to be able to locate the nearest one), or use data from other sources (such as demographic data from the government or from remote servers via a Web API, such as Yahoo Weather). You might even want your users to create the data themselves (for instance, if you are building an application that allows users to report graffiti in their local neighborhood).

In any event, your application data is likely to be in the form of points, lines, or polygons. This data is rendered dynamically on your map at runtime. Leaflet.js provides a number of different classes to help you do this, and in this chapter, we're going to dive in and start using some of the more common ones.

Markers

In GIS terms, a point is the simplest spatial type, and is defined purely by its x and y coordinates. Leaflet.js has a **`Point`** class, but it's really only used to define locations on the map in screen, rather than map coordinates. Instantiating an instance of Point(x,y) refers to a point on the screen:

```
var pt = L.Point(300,400);
```

To display a point on the map, you need to think in terms of "markers" rather than "points." A marker is an example of what Leaflet.js refers to as a **UI Layer**. The **`Marker`** class accepts a **`LatLng`** object, which defines the latitude and longitude where the marker should display, and (optionally) an **`options`** object, which allows you to set various properties on the marker to specify its appearance and behavior. The following example demonstrates creating a marker and setting its **`title`** property so that it displays a tooltip when the user hovers over it with the mouse pointer. The **`alt`** property provides alternative text for screen readers:

Code Listing 12: Creating a marker with a tooltip

```
...
       var pubLatLng = L.latLng(51.757230,-1.260269);
       var pubMarker = L.marker(pubLatLng,
       {
           title: "The Eagle and Child: JRR Tolkien and CS Lewis supped
beer here",
           alt: "The Eagle and Child public house"
       }).addTo(map);
...
```

Figure 10: Marker with tooltip

To avoid creating any unnecessary variables, we can create the **`LatLng`** object on the fly, by passing in the latitude and longitude coordinates as an array:

Code Listing 13: Passing the L.latLng object into the L.marker constructor

```
...
       var pubMarker = L.marker([51.757230,-1.260269],
       {
           title: "The Eagle and Child: JRR Tolkien and CS Lewis supped
beer here",
           alt: "The Eagle and Child public house"
       }).addTo(map);
...
```

The **`Marker`** class has a whole bunch of different properties that you can set by specifying them in an options object. Consult the Leaflet.js documentation for a full list, but some of the more interesting ones include **`icon`**, **`draggable`**, and **`opacity`**.

Using custom marker icons

By default, Leaflet.js denotes a marker on the map with a simple blue pushpin. You can define your own custom marker icons with the **`L.icon`** class. The **`L.icon`** class has several properties that let you fine-tune the appearance of your icon, including properties for retina display, icon size, and the positioning of tooltips and shadows, but as a bare minimum, you must set the **`iconUrl`** property where the icon graphic is located:

Code Listing 14: Defining a custom icon using the L.icon class

```
...
        var pubIcon = L.icon({
            iconUrl: "./images/beer.png",
            iconSize: [40,50]
        });
        var pubMarker = L.marker([51.757230,-1.260269],
        {
            icon: pubIcon,
            title: "The Eagle and Child: JRR Tolkien and CS Lewis supped
beer here",
            alt: "The Eagle and Child public house"
        }).addTo(map);
...
```

Figure 11: Marker with custom icon

Making markers interactive

By default, the map marker's **clickable** property is **true**, which means that it can respond to mouse events, including (but not limited to) **click**.

By setting the **draggable** property to **true**, you can allow your users to move your markers around the map. This could be handy in certain use cases, such as an asset-tracking application. But in order to be really useful, you need to figure out when a marker has been moved and where it has been moved to.

We take a look at events for map and map objects like markers in Chapter 4.

Adjusting marker transparency

You can make your marker more or less transparent by adjusting the value of the **opacity** property in the marker options object. The default value is 1.0 (fully opaque), and a value of 0.0 is completely transparent.

Polylines

In geospatial terms, a line is just a collection of points. To draw one on our map, we need to create an object of the **PolyLine** class. A **PolyLine** is a type of **vector layer**, in that Leaflet.js renders the line it describes it as vector graphics. All Leaflet.js vector graphics are derived from an abstract class called **Path**, which you never instantiate directly, but whose properties, methods, and events are available in subclasses.

Polylines are created from collections of **LatLng** pairs, so to define a **PolyLine** we must supply at least two such pairs: the start and end point of a line. We can add as many other pairs as we wish if we need our line to meander. If the line consists of many segments, consider using the **PolyLine** class **smoothFactor** property, which simplifies the line and makes it faster for Leaflet.js to draw. You'll need to experiment to get the right trade-off between accuracy and performance.

You can change the appearance of the **PolyLine** by using properties from the parent **Path** class. These include **color** (for the color of the line) and **weight** (for the line width in pixels). The following example creates a line denoting an imaginary walk (or stagger?) from the Eagle and Child pub made up of five points. Note the nesting of arrays. Each point on the line is represented as a latitude/longitude pair within an array, and these point arrays are passed to the polyline constructor within a single "parent" array. The **color** property of the line is red, and the **weight** is eight pixels.

Code Listing 15: Defining a polyline

```
...
        var walkLine = L.polyline([
            [51.757276, -1.260129],
            [51.756831, -1.260054],
            [51.756154, -1.259700],
```

```
            [51.756074, -1.259453],
            [51.755636, -1.259346]
        ], {
            color: "red",
            weight: 8
        }).addTo(map);
...
```

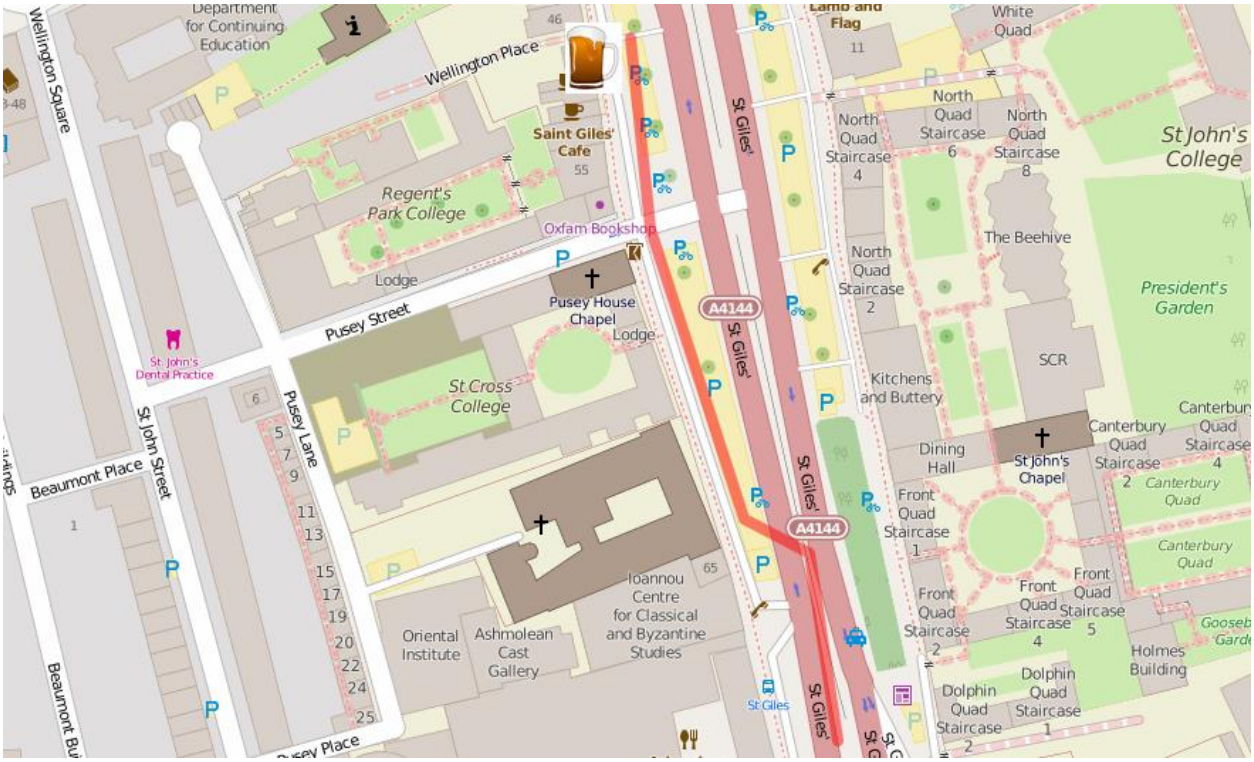


Figure 12: Polyline displayed on map

Polygons

A polygon is just a line where the start and end points are the same. You can create an object of the **`L.polygon`** class and pass in the coordinate pairs in the same way as you did for **`L.polyline`**. You don't need to specify the end point—Leaflet.js will "close" the polygon for you. Because **`L.polygon`** is a subclass of **`L.path`**, you use the same properties **`color`** and **`weight`** to set the color and thickness of the polygon's outline that you used for the polyline. But because you also have the area enclosed by the polygon to play with, you can use the **`L.path.fillColor`** and **`L.path.fillOpacity`** properties to determine how that inner region is rendered.

For example, the following code defines a polygon with six vertices, a blue outline, and a red fill with 50 percent transparency:

Code Listing 16: Defining a polygon

```
...
        var buildingPoly = L.polygon([
            [51.756633, -1.258688],
            [51.756416, -1.258618],
            [51.756454, -1.258323],
            [51.756592, -1.258371],
            [51.756584, -1.258443],
            [51.756663, -1.258473]
        ], {
            color: "blue",
            weight: 5,
            fillColor: "red",
            fillOpacity: 0.5
        }).addTo(map);
...
```

Figure 13: Polygon displayed on map

Certain polygons, such as circles and rectangles, are very common in web mapping applications, often denoting geographical boundaries, and therefore, Leaflet.js provides a couple of utility classes for creating those shapes. These are, unsurprisingly, the **`L.circle`** and **`L.rectangle`** classes.

Rectangles

The **`L.rectangle`** class is a subclass of **`L.polygon`**, so you can use all of **`L.polygon`**'s properties, methods, and events, as well as those of *its* parent class, **`L.path`**.

To create a rectangle, instantiate an object of the **`L.rectangle`** class, providing the latitude and longitude coordinates for both the upper-left and lower-right corners as parameters. Set the outline and fill in exactly the same way as any other polygon.

Code Listing 17: Defining a rectangle

```
...
        var parkRectangle = L.rectangle([
            [51.761539, -1.258820],
            [51.760995,-1.256974]
        ], {
            color: "red",
            weight: 5,
            fillColor:"blue"
        }).addTo(map);
...
```

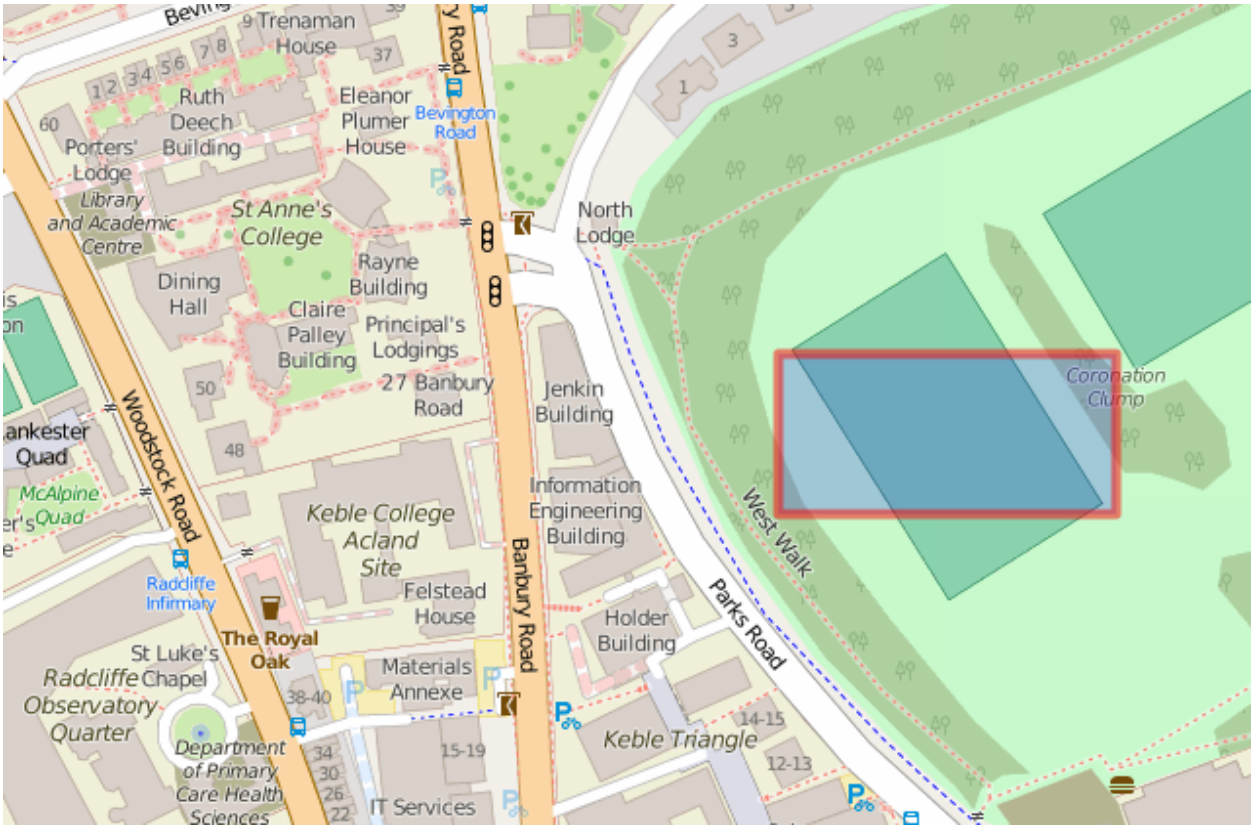


Figure 14: Rectangle displayed on map

Circles

Like **`L.rectangle`**, **`L.circle`** is subclassed from **`L.polygon`**. To create a circle, provide a latitude and longitude for the center and a radius in meters.

Code Listing 18: Defining a circle

```
...
    var areaCircle = L.circle(
        [51.759806, -1.264173],
        100,
        {
            color: "red",
            weight: 5,
            fillColor:"green"
        }
    ).addTo(map);
...
```

Figure 15: Circle displayed on map

Treating multiple polylines or polygons as single objects

Often when you're building up a layer of data, you'll want to give your users the ability to toggle all the features in that layer on or off with a single action. For that to happen, you need a way to group those features.

If the features you want to group in this way are all of the same type, and that type is either **`L.polyline`** or **`L.polygon`**, then you can use the **`L.multiPolyline`** and **`L.multiPolygon`** classes. For all intents and purposes, the way you create these objects is the same as creating individual polylines and polygons. The only difference is that you specify "sets" of latitudes and longitudes—one for each polyline or polygon. The following example uses **`L.multiPolygon`**, but you use exactly the same approach for **`L.multiPolyline`**. It also uses a layer control (which we covered in Chapter 2) that you can use to turn the overlay on or off as a whole.

Code Listing 19: Defining a multiPolygon

```
...
    function init() {
        var map = L.map('map').setView([51.76, -1.26],16);;

        // OSM Mapnik
        var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
        var osm = L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; ' + osmLink,
            maxZoom: 18,
        }).addTo(map);

        var multipolygon = L.multiPolygon([
            [
                [51.756633, -1.258688],
                [51.756416, -1.258618],
                [51.756454, -1.258323],
                [51.756592, -1.258371],
                [51.756584, -1.258443],
                [51.756663, -1.258473]
            ],
            [
                [51.756578, -1.259225],
                [51.756601, -1.259026],
                [51.756342, -1.258940],
                [51.756313, -1.259144]
            ]
        ], {
            color: "blue",
            weight: 5,
            fillColor: "red",
            fillOpacity: 0.5
        }).addTo(map);
```

```
        var baseLayers = {
            "OpenStreetMap": osm
        };

        var overlays = {
            "University Buildings": multipolygon,
        };

        L.control.layers(baseLayers, overlays).addTo(map);

    }
...
```

All the polylines and polygons within a **`L.multiPolyline`** or **`L.multiPolygon`** share the same symbology, so you'll only want to use them to group features that are related to each other, such as town boundaries or water features.

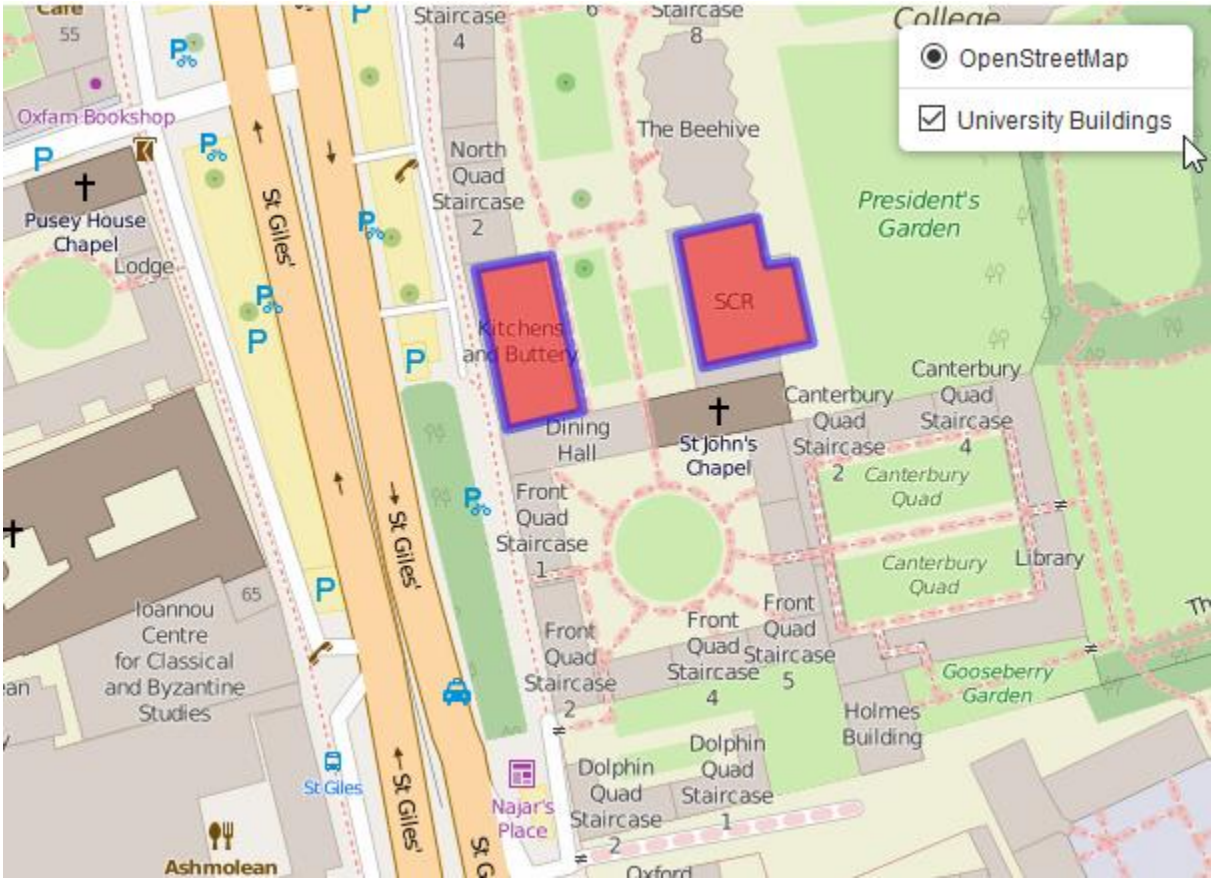


Figure 16*: L.MultiPolygon shown as discrete layer in Layer Control*

Treating features of different types as a single group layer

If you want to group features of different types, such as a polygon with a set of markers, or even just a set of markers (which don't have the equivalent of a **`L.multiPolyline`** or **`L.multiPolygon`** class), then you need to create a *layer group*.

Define the individual features first, but don't add them to the map. Then create an object of **`L.layerGroup`**, passing in the variable names that you have assigned to the features that will comprise the group. Then call the layer group's **`.addTo()`** method to add the group itself to the map instead of the individual layers.

The following example uses a layer control to demonstrate that the features within the layer group are considered as a single overlay. This gives you the ability to work with each of the features in the layer group collectively (for example, allowing your users to hide or display them all at once instead of individually).

Code Listing 20: Grouping several different features

```
...
    function init() {
        var map = L.map('map').setView([51.76, -1.26],16);;

        // OSM Mapnik
        var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
        var osm = L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; ' + osmLink,
            maxZoom: 18,
        }).addTo(map);

        var pubIcon = L.icon({
            iconUrl: "./images/beer.png",
            iconSize: [40,50]
        });
        var pubMarker = L.marker([51.757230,-1.260269],
        {
            icon: pubIcon
        });

        var walkLine = L.polyline([
            [51.757276, -1.260129],
            [51.756831, -1.260054],
            [51.756154, -1.259700],
            [51.756074, -1.259453],
            [51.755636, -1.259346]
        ], {
            color: "red",
            weight: 8
        });
```

```
        var buildingPoly = L.polygon([
            [51.756633, -1.258688],
            [51.756416, -1.258618],
            [51.756454, -1.258323],
            [51.756592, -1.258371],
            [51.756584, -1.258443],
            [51.756663, -1.258473]
        ], {
            color: "blue",
            weight: 5,
            fillColor: "red",
            fillOpacity: 0.5
        });

        var myLayerGroup = L.layerGroup(
            [
                pubMarker,
                walkLine,
                buildingPoly
            ]
        ).addTo(map);

        var baseLayers = {
            "OpenStreetMap": osm
        };

        var overlays = {
            "My Layer Group": myLayerGroup,
        };

        L.control.layers(baseLayers, overlays).addTo(map);
    }
...
```

If you want to add or remove features from the group programmatically, use the group's **`.addLayer()`** and **`.removeLayer()`** methods, respectively:

Code Listing 21: Programmatic addition and removal of layers from the layer group

```
...
    var myLayerGroup = L.layerGroup(
        [
            walkLine,
            buildingPoly
        ]
    ).addTo(map);

    var baseLayers = {
```

```
        "OpenStreetMap": osm
    };
    var overlays = {
        "My Layer Group": myLayerGroup,
    };
    L.control.layers(baseLayers, overlays).addTo(map);

    myLayerGroup.addLayer(pubMarker);
    myLayerGroup.removeLayer(walkLine);

...
```

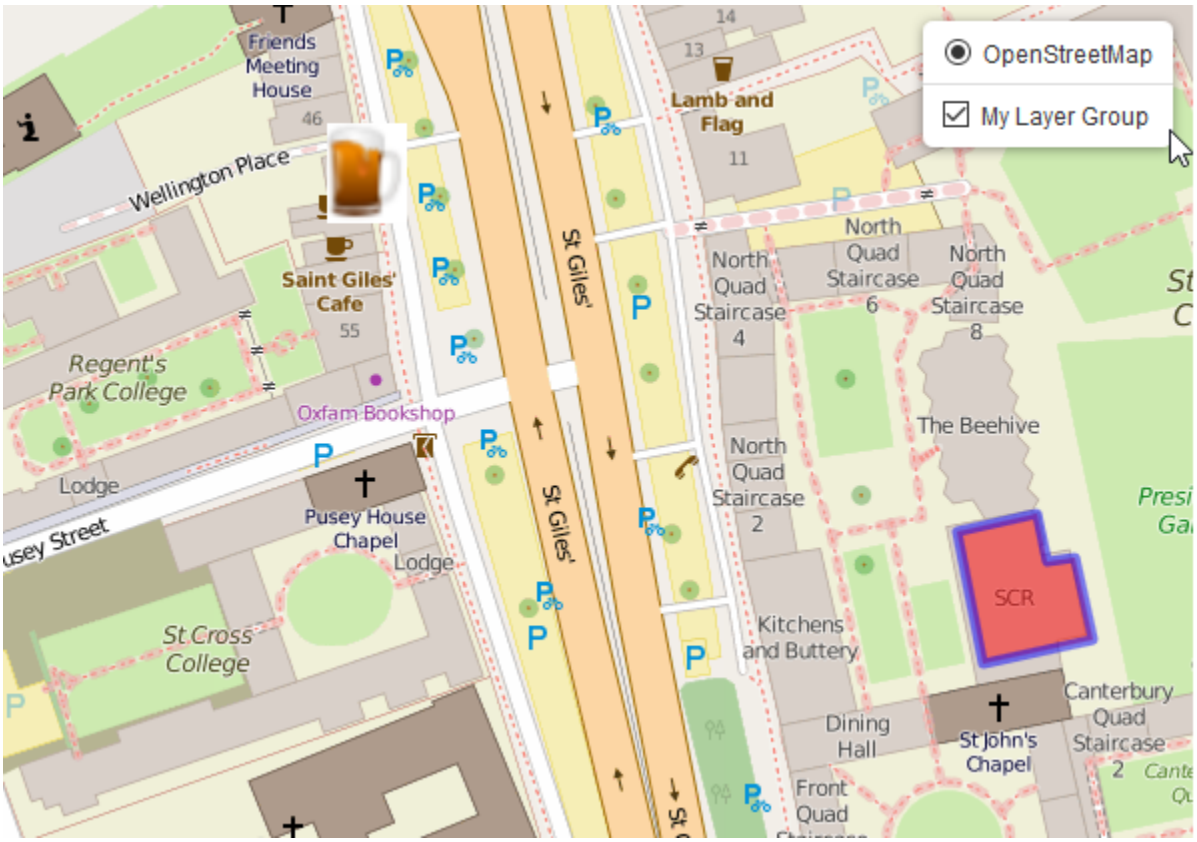


Figure 17: Layer Group with walking route removed and pub added

One thing to note about the **`L.layerGroup`** class is that it does not support popups (covered next) or mouse events. If this functionality is important to you, then use a **`L.featureGroup`** instead. The construction of a **`L.featureGroup`** object is identical to the **`L.layerGroup`**, but offers extra methods and events.

Adding popups to graphics

Now that you have all these nice graphics on your map, you're probably looking for a way for your users to interact with them to find out what they represent. A very simple way of doing this is to use another UI layer element called a **popup**. You can use a popup with any of the layers we have seen so far in this chapter, and the popup can include any valid HTML. This makes popups great for displaying formatted information about a geographical feature, and also displaying links, images, and videos.

To display a popup when a user clicks on a marker, polyline, or polygon, use that object's **`.bindPopup()`** method, pass in a string of HTML, and optionally, an options object. Valid options properties for popups include:

- **`.maxWidth, .minWidth, .maxHeight`**: To control the size of the popup
- **`.keepInView`**: Set to **`true`** to keep the popup in view if the user pans the screen away from the feature in question
- **`.closeButton`**: Set to **`false`** if you don't want a close button to appear in the popup
- **`.closeOnClick`**: Set to **`false`** if you don't want the user to be able to close the popup by clicking on the map
- **`.zoomAnimation`**: Whether the popup should be animated when the user zooms in (true by default)

Code Listing 22: Binding a popup to the polyline

```
...
    var walkLine = L.polyline([
        [51.757276, -1.260129],
        [51.756831, -1.260054],
        [51.756154, -1.259700],
        [51.756074, -1.259453],
        [51.755636, -1.259346]
    ], {
        color: "red",
        weight: 8
    }).bindPopup("Our route");
...
```

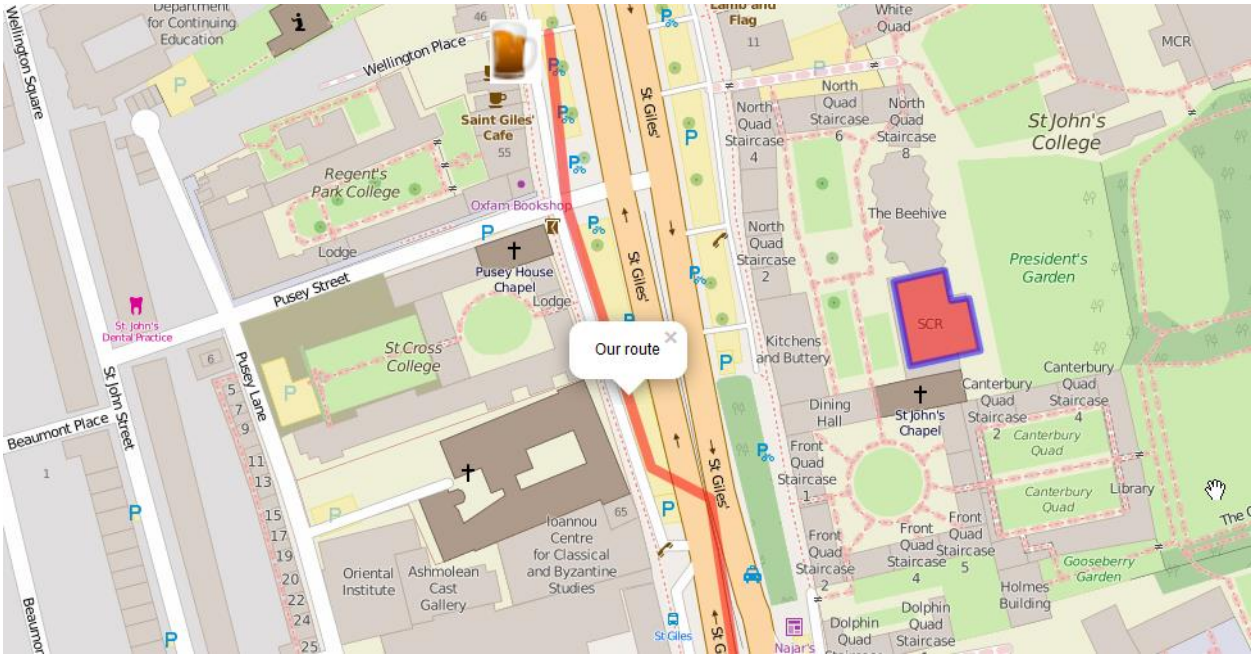


Figure 18: Clicking on the route polyline displays a popup

To have all the features in a feature group display a popup when clicked, create a **`L.featureGroup`** object and call its **`.bindPopup()`** method. The following example demonstrates adding HTML content to the popup contents. Every feature in the feature group displays exactly the same details, including a link to the Wikipedia entry for Oxford:

Code Listing 23: Binding a popup to a feature group

```
...
    var popupContent = "<h2>Walking Tour of Oxford</h2>" +
        "<p>Lots of <b>cool things</b> to see in " +
        "<a href='https://en.wikipedia.org/wiki/Oxford'>Oxford</a>";
    var myFeatureGroup = L.featureGroup(
        [
            pubMarker,
            walkLine,
            buildingPoly
        ]
    ).bindPopup(popupContent).addTo(map);

...
```

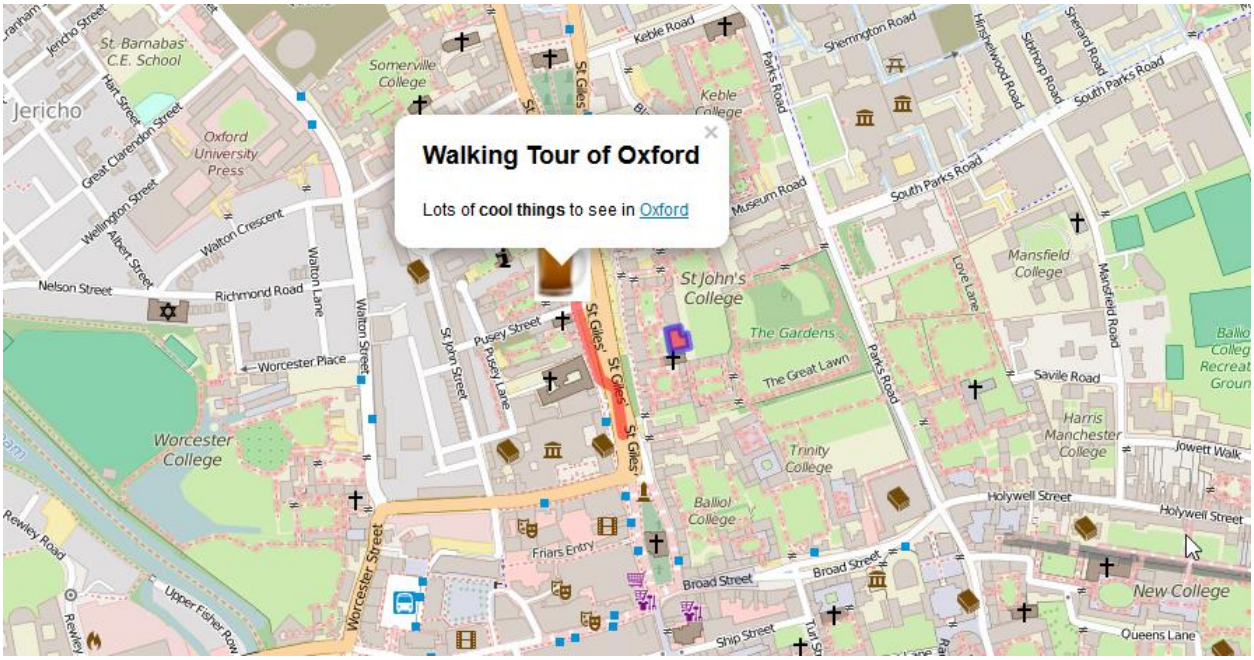


Figure 19: Popup for feature group with HTML formatted contents

You can enable popups on **`L.multiLine`** and **`L.multiPolyline`** features in exactly the same way:

Code Listing 24: Binding a popup to a L.multiPolygon

```
...
    var multipolygon = L.multiPolygon([
        [
            [51.756633, -1.258688],
            [51.756416, -1.258618],
            [51.756454, -1.258323],
            [51.756592, -1.258371],
            [51.756584, -1.258443],
            [51.756663, -1.258473]
        ],
        [
            [51.756578, -1.259225],
            [51.756601, -1.259026],
            [51.756342, -1.258940],
            [51.756313, -1.259144]
        ]
    ], {
        color: "blue",
        weight: 5,
        fillColor: "red",
        fillOpacity: 0.5
    }).bindPopup("This is a multipolygon").addTo(map);
...
```

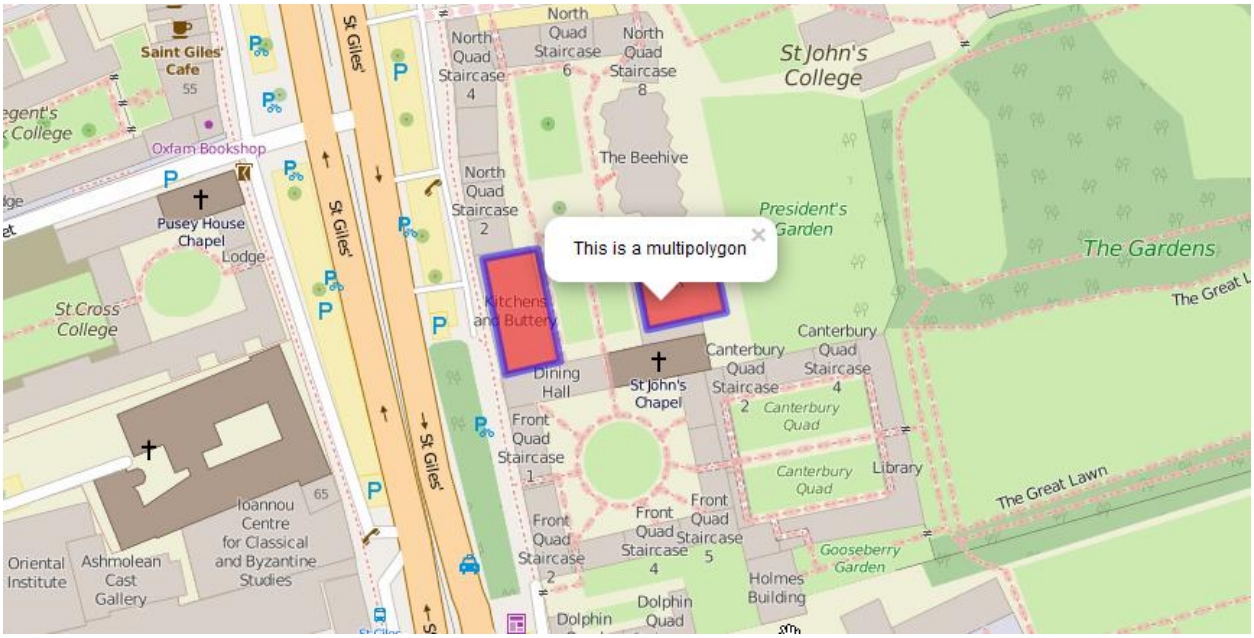


Figure 20: Each polygon in the L.multiPolygon displays the same popup

The Map Draw utility

Because it can be a pain trying to discover the coordinates for the locations and geometries you want to represent in your Leaflet.js applications, I have written a simple tool that you can access here.

To use the tool, zoom in to the location in which you want to create features, and then use the draw toolbar on the left-hand side of the map and follow the instructions to create the desired geometry. The Leaflet.js code to create that geometry will appear in the left-hand pane. Click **Copy to Clipboard** and the code is ready to be pasted into your editor or IDE of choice.

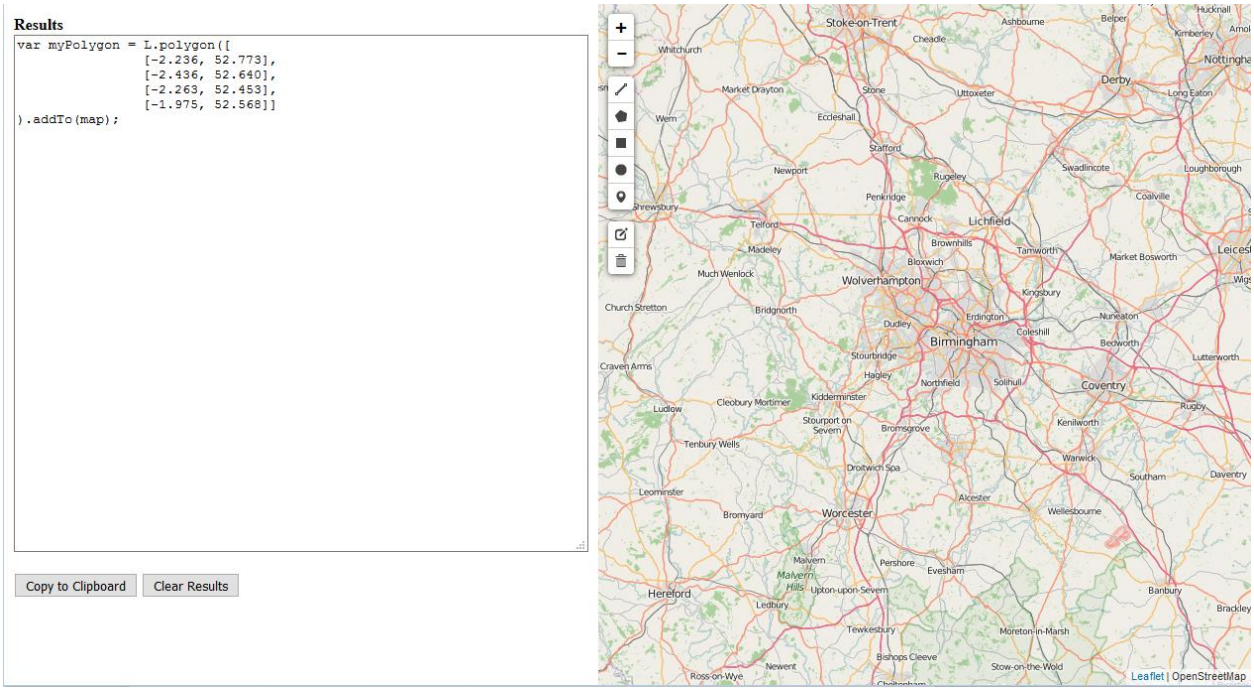


Figure 21: The Map Draw utility

GeoJSON

If you know your JavaScript, then you have doubtless heard of JSON. JSON stands for JavaScript Object Notation, and it's the syntax you use to specify object literals in JavaScript. In recent years, JSON has outgrown JavaScript, and is now used as a human-readable lightweight data interchange format, often in preference to XML, which is seen as heavyweight by comparison, and more difficult to interpret by human readers.

For example, the following JSON example defines a "takeaways" object, which comprises an array of three objects representing individual takeout restaurants and the types of cuisine they offer:

Code Listing 25: "Standard" JSON describing takeaway restaurants

```
{"takeaways":[
    {"name":"Aziz", "cuisine":"Indian"},
    {"name":"Dak Bo", "cuisine":"Chinese"},
    {"name":"Luigi's", "cuisine":"Italian"}
]}
```

GeoJSON is an extension of standard JSON that allows you to describe geometries. You can specify points, lines, and polygons using GeoJSON, and you can group those geometries into multipoints, multilines, and multipolygons.

Note: You can view the full GeoJSON specification here.

GeoJSON is a real boon to web map developers like us, because it provides a convenient way of describing geospatial features—their geometries and attributes—in a highly portable way.

Adding GeoJSON to the map

The simplest way to add GeoJSON to your Leaflet.js map is to hardcode it as a JavaScript variable and then add it using an instance of the **`L.geoJson`** class. If we take our initial JSON, we can turn it into GeoJSON by adding adding **`type`**, **`geometry`** and **`properties`** objects to each of our takeaway objects:

Code Listing 26: Takeaways represented in GeoJSON format

```
var geoJSON = [{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [-1.155, 51.896]
    },
    "properties": {
        "name": "Aziz",
        "cuisine": "Indian"
    }
}, {
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [-1.150, 51.897]
    },
    "properties": {
        "name": "Dak Bo",
        "cuisine": "Chinese"
    }
}, {
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [-1.153, 51.897]
    },
    "properties": {
        "name": "Luigi's",
        "cuisine": "Italian"
    }
}];
```

Note: GeoJSON requires that points are specified in the order of `[longitude, latitude]` rather than the `[latitude, longitude]` that Leaflet.js expects.

We can then use the **`L.geoJson`** layer type to add this geoJson to our map just like any other layer:

Code Listing 27: Adding GeoJSON to the map

```
var geoJSON = [{
    "type": "Feature",
    "geometry": {…},
    "properties": {…}
}, {
    "type": "Feature",
    "geometry": {…},
    "properties": {…}
}, {
    "type": "Feature",
    "geometry": {…},
    "properties": {…}
}];

var geoJSONLayer = L.geoJson(geoJSON).addTo(map);
```

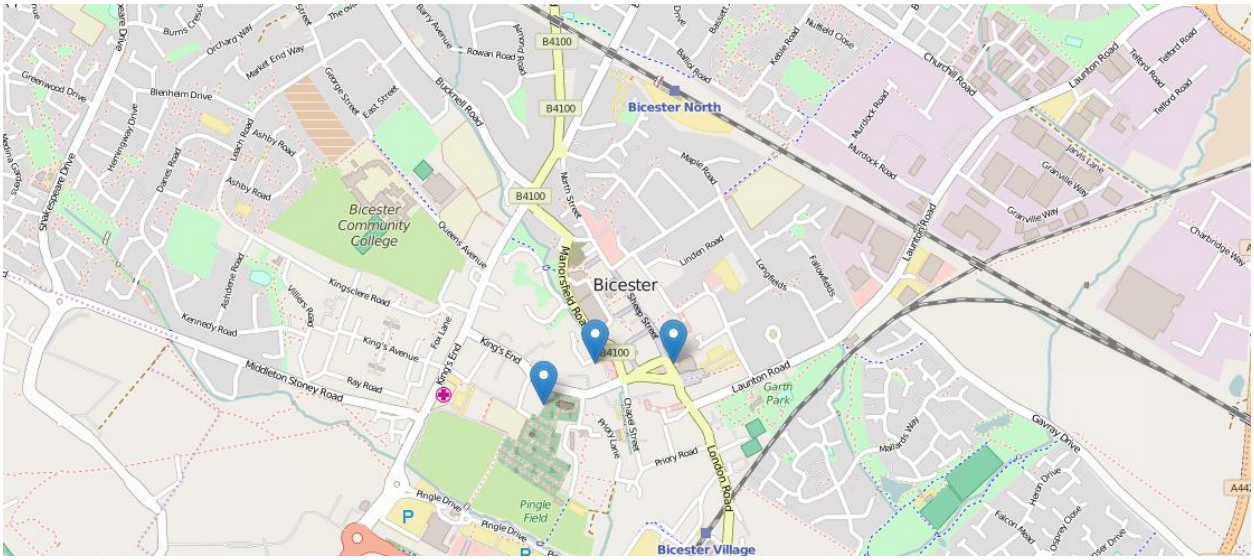


Figure 22: GeoJSON data displayed on the map

We can mix points, lines, and polygons in GeoJSON, too. Consider the following GeoJSON:

Code Listing 28: Mixing different geometries in GeoJSON

```
var geoJSON = [{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [-1.145, 51.898]
    },
    "properties": {
        "name": "My Point",
        "title": "This is a point"
    }
}, {
    "type": "Feature",
    "geometry": {
        "type": "LineString",
        "coordinates": [
            [-1.161, 51.904],
            [-1.159, 51.909],
            [-1.157, 51.903],
            [-1.149, 51.905],
            [-1.144, 51.904]
        ]
    },
    "properties": {
        "name": "My LineString",
        "title": "This is a polyline"
    }
}, {
    "type": "Feature",
    "geometry": {
        "type": "Polygon",
        "coordinates": [
            [
                [-1.176, 51.905],
                [-1.176, 51.902],
                [-1.165, 51.899],
                [-1.163, 51.902]
            ]
        ]
    },
    "properties": {
        "name": "My Polygon",
        "title": "This is a polygon"
    }
}];

var geoJSONLayer = L.geoJson(geoJSON).addTo(map);
```

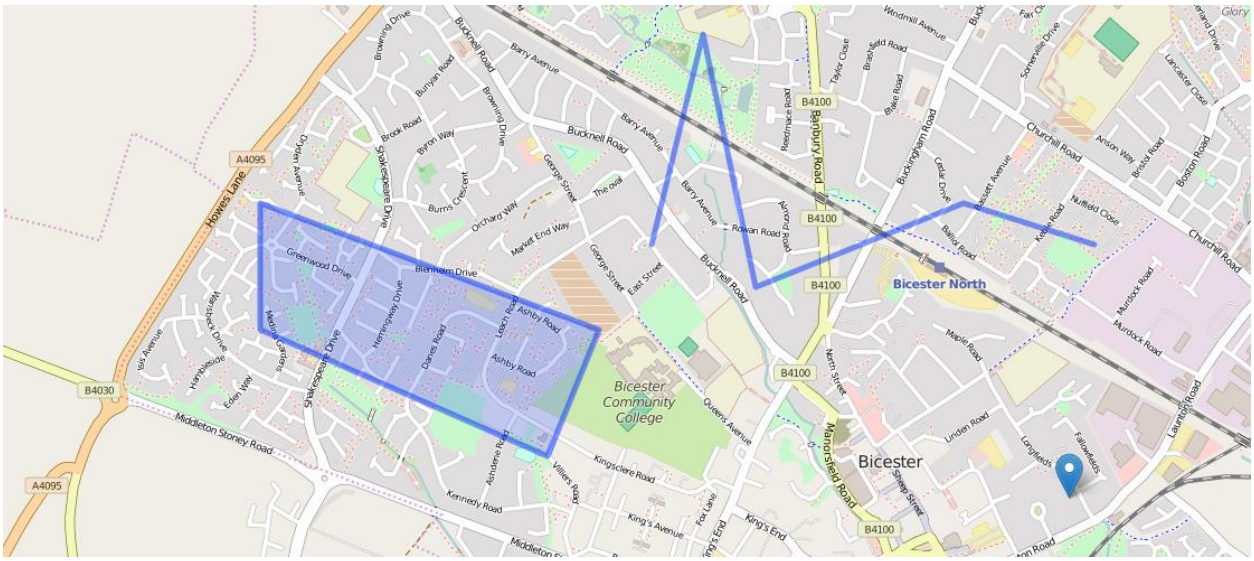


Figure 23: Mixed GeoJSON geometries displayed on the map

> ***Note: GeoJSON polygon coordinates are encoded with one more level of nested arrays than we have seen in previous examples. This is to allow for the creation of "rings"—polygons within polygons. Using rings, you can create polygons with "holes" (think of the shape of a donut). The `L.polygon` class also allows you to specify polygon geometries in this way.***

Rendering GeoJSON: polylines and polygons

You'll note from the previous two examples that Leaflet.js has provided default symbology for our GeoJSON. Usually you'll want to specify this yourself.

You can control the symbology of geoJSON polylines or polygons either by specifying a **`style`** property in an options object that you pass to the **`L.geoJson`** constructor as its second argument, or by calling the layer's **`setStyle()`** method. Generally speaking, you use the first method (setting the style in the constructor) to specify an initial style for when the layer first displays, and the second method (calling **`setStyle()`**) to programmatically change the symbology later on, perhaps in response to a user action, such as hovering over a feature with the mouse. We'll talk about responding to events in the next chapter.

Let's look at the first method: adding a **`style`** property within the options object. The value of the **`style`** property is a function that gets called when the features are loaded and can optionally be used to symbolize based on one of that feature's attributes. In this example we are symbolizing based on the value of the **`name`** property:

Code Listing 29: Symbolizing GeoJSON based on feature properties

```
...
var geoJSONLayer = L.geoJson(geoJSON, {
        style: function(feature) {
            switch(feature.properties.name) {
```

```
                case 'My LineString':
                    return {
                        color: "#F115CA",
                        weight: 8
                    }
                    break;
                case 'My Polygon':
                    return {
                        color: "red",
                        fillColor: "#488D52",
                        fillOpacity: 0.5
                    }
                    break;
            }
        }
    }).addTo(map);
...
```

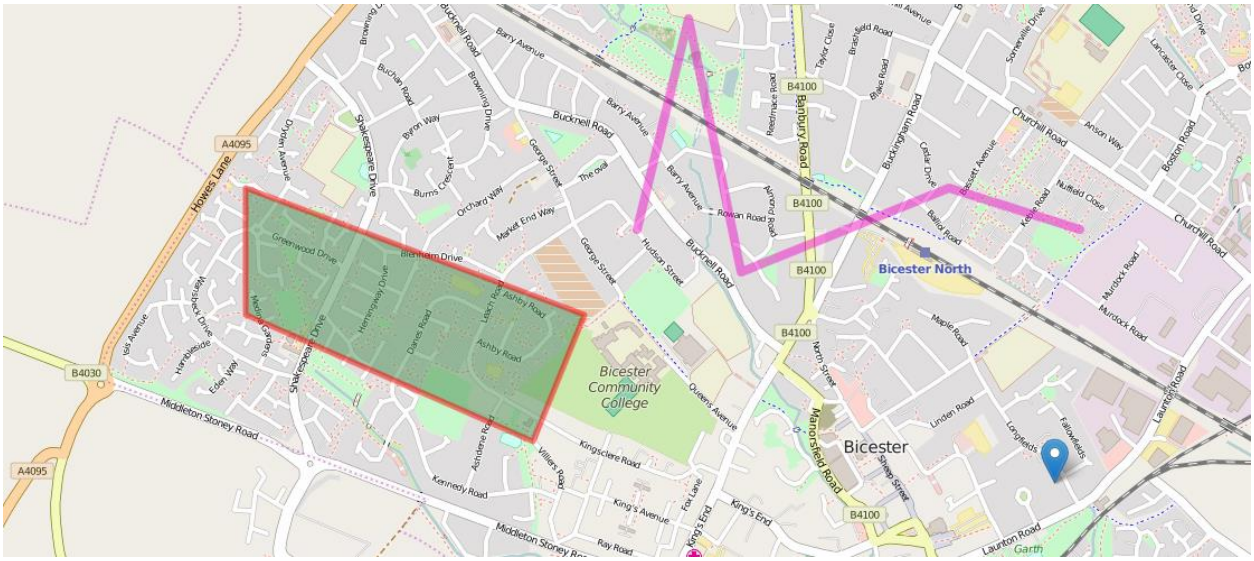


Figure 24: The GeoJSON polyline and polygon symbolized by name

If we wanted to style either multiple polylines or polygons exactly the same way without considering any differences between them, we would just define our symbology and pass it directly to the **`style`** property in the options object. The following example demonstrates setting an identical symbology for two different polygons:

Code Listing 30: Styline two different polygons identically:

```
...
    var schoolSymbol = {
        "color": "#CE4A93",
        "weight": 5,
        "fillColor": "#FF002B",
        "fillOpacity": 0.65
```

```
    };

    var schoolsGeoJSON = [{
        "type": "Feature",
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [
                    [-1.173, 51.907],
                    [-1.174, 51.906],
                    [-1.173, 51.905],
                    [-1.173, 51.905],
                    [-1.172, 51.905],
                    [-1.172, 51.905],
                    [-1.173, 51.905],
                    [-1.172, 51.906]
                ]
            ]

        }
    }, {
        "type": "Feature",
        "geometry": {
            "type": "Polygon",
            "coordinates": [
                [
                    [-1.163, 51.902],
                    [-1.164, 51.901],
                    [-1.164, 51.901],
                    [-1.165, 51.900],
                    [-1.159, 51.899],
                    [-1.158, 51.899],
                    [-1.158, 51.899],
                    [-1.158, 51.900],
                    [-1.158, 51.900],
                    [-1.158, 51.900],
                    [-1.162, 51.902]
                ]
            ]

        }
    }];

    var geoJSONLayer = L.geoJson(schoolsGeoJSON, {
        style: schoolSymbol
    }).addTo(map);
...
```

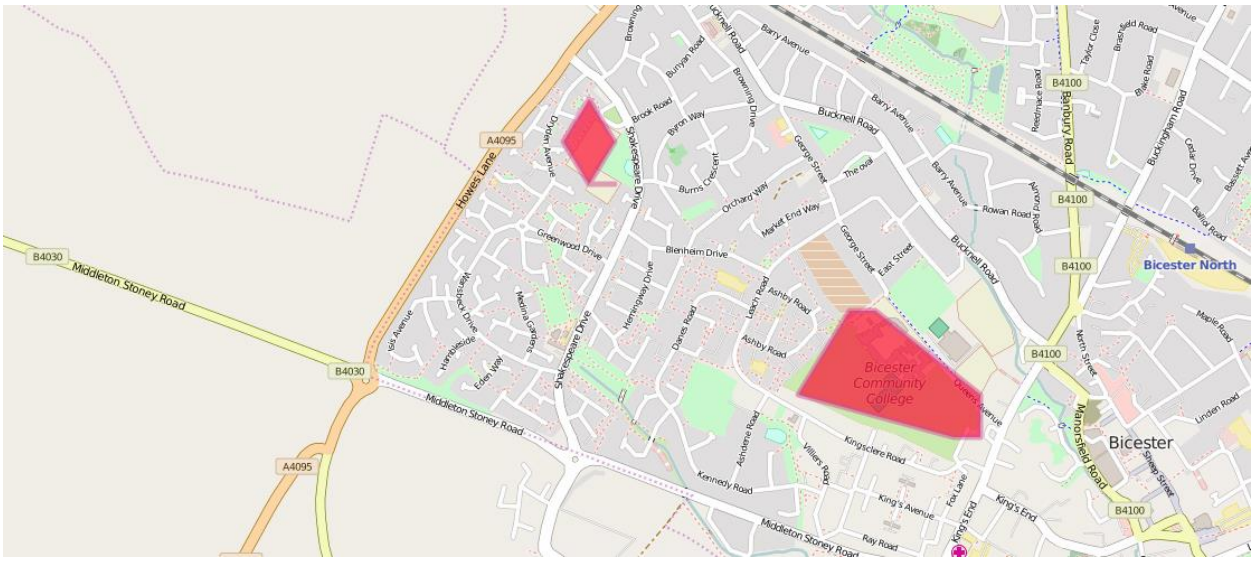


Figure 25: Two different polygons sharing the same symbology

If you want to perform other actions on each feature before it's loaded, add a property to the options object in the constructor called **`onEachFeature().`** This property exists on points as well as lines and polygons. Its value is a function that receives a feature and the layer it came from. This can be really useful if, for example, you want to bind a popup to each feature based on its properties:

Code Listing 31: Displaying popups on all features using onEachFeature()

```
...
    var geoJSONLayer = L.geoJson(geoJSON, {
        style: function(feature) {
            switch(feature.properties.name) {
                case 'My LineString':
                    return {
                        color: "#F115CA",
                        weight: 8
                    }
                    break;
                case 'My Polygon':
                    return {
                        color: "red",
                        fillColor: "#488D52",
                        fillOpacity: 0.5
                    }
                    break;
            }
        },
        onEachFeature: function(feature, layer) {
            layer.bindPopup(feature.properties.name);
        }

    }).addTo(map);
```

```
...
```

Rendering GeoJSON: points

Because **`L.polyline`** and **`L.polyline`** both inherit from **`L.path`**, they can both be styled in the same way. Markers, representing GeoJSON point features, need to be treated differently. Instead of using the **`style`** property of the options object, we use the **`pointToLayer`** property. Going back to our takeaway restaurants, we can demonstrate this in the following example, where we symbolize markers depending on the type of cuisine available at each. We also take the opportunity to bind a popup to each marker so we can display the name of the restaurant it represents:

Code Listing 32: Symbolizing markers by GeoJSON properties

```
...
    var indianIcon = L.icon({
        iconUrl: 'images/India.png',
        iconSize: [48, 48],
    });
    var chineseIcon = L.icon({
        iconUrl: 'images/China.png',
        iconSize: [48, 48],
    });
    var italianIcon = L.icon({
        iconUrl: 'images/Italy.png',
        iconSize: [48, 48],
    });

    var indian, chinese, italian;
    var geoJSONLayer = L.geoJson(geoJSON, {
        pointToLayer: function(feature, latlng) {
            switch(feature.properties.cuisine) {
                case 'Indian':
                    indian = L.marker(latlng, {
                        icon: indianIcon
                    }).bindPopup(feature.properties.name);
                    return indian;
                    break;
                case 'Chinese':
                    chinese = L.marker(latlng, {
                        icon: chineseIcon
                    }).bindPopup(feature.properties.name);
                    return chinese;
                    break;
                case 'Italian':
                    italian = L.marker(latlng, {
                        icon: italianIcon
                    }).bindPopup(feature.properties.name);
```

```
                    return italian;
                    break;
                }
            }
        }).addTo(map);
...
```

Figure 26: Takeaways symbolized by cuisine

Filtering GeoJSON

Sometimes it can be useful to filter the data within GeoJSON. Perhaps the GeoJSON comes from an external source and we are only interested in particular features. We might even extend this ability to our users and build an application that allows them to choose which features to display.

Just use the **`filter`** property in an options object and give it a function that tests the values passed to it for whatever your criteria are. Return **`true`** to display features and **`false`** to hide them. In this example, we have decided that we don't want Italian food tonight (we stuffed ourselves full of pizza last night):

Code Listing 33: Filtering GeoJSON based on feature attributes

```
...
```

```
    var indian, chinese, italian;
    var geoJSONLayer = L.geoJson(geoJSON, {
        pointToLayer: function(feature, latlng) {
            switch(feature.properties.cuisine) {
                case 'Indian':
                    indian = L.marker(latlng, {
                        icon: indianIcon
                    }).bindPopup(feature.properties.name);
                    return indian;
                    break;
                case 'Chinese':
                    chinese = L.marker(latlng, {
                        icon: chineseIcon
                    }).bindPopup(feature.properties.name);
                    return chinese;
                    break;
                case 'Italian':
                    italian = L.marker(latlng, {
                        icon: italianIcon
                    }).bindPopup(feature.properties.name);
                    return italian;
                    break;
            }
        },
        filter: function(feature, latlng) {
            switch(feature.properties.cuisine) {
                case 'Indian':
                    return true;
                    break;
                case 'Chinese':
                    return true;
                    break;
                case 'Italian':
                    return false;
                    break;
            }
        }
    }).addTo(map);
...
```

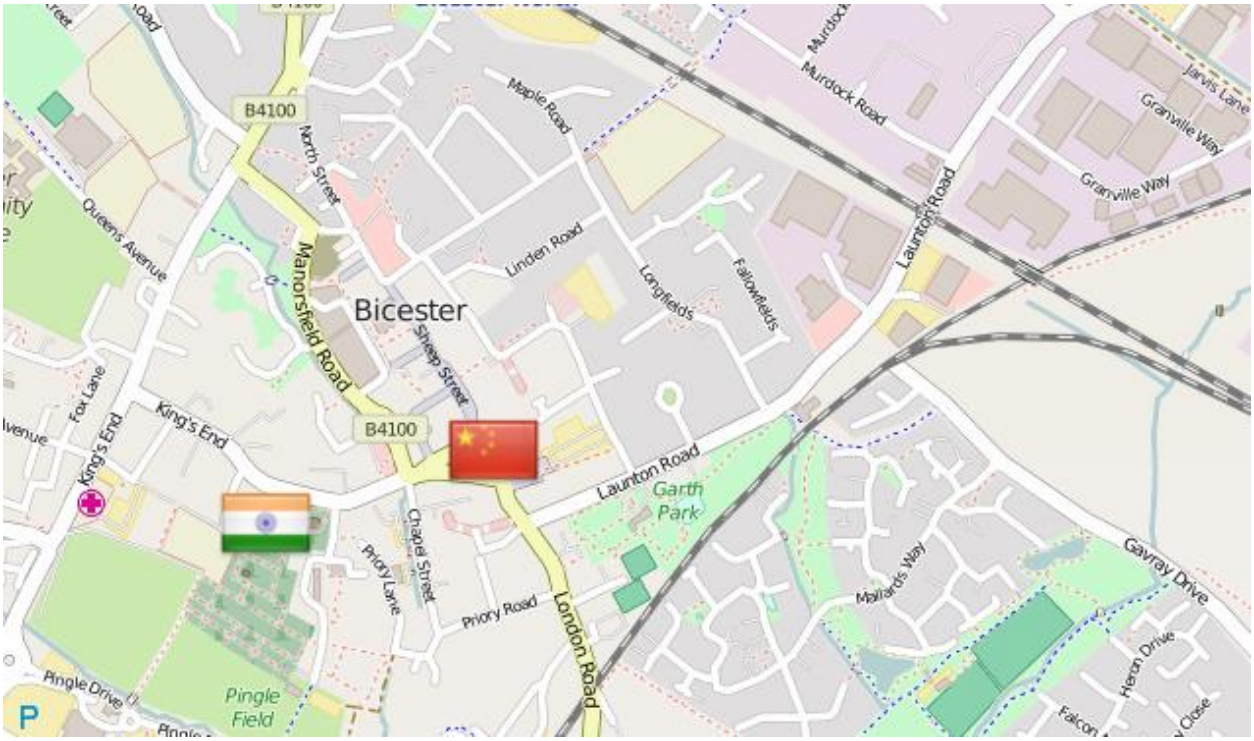


Figure 27: We don't fancy Italian food tonight

Chapter 4 Handling Events

Now that you have an application that contains a map and some data, you'll often want a way for your users to interact with it. So far, the only interaction you have enabled is the ability for a user to click on a feature in your overlay and see a popup that you bound to it using the **`.bindPopup()`** method.

Just about every control in Leaflet.js emits events to alert your application when things happen. For example, the **`L.Map`** class supports 34 events at the time of writing. These range from the sort of event that you expect just about any control in any programming environment to have, such as **`focus`**, **`mouseover`**, and **`click`**, to ones that are very specific to the functionality it provides, such as **`overlayadd`** (when a new overlay is added to the map) and **`move`** (when the map extent changes).

Map control events

This table lists the events supported in **`L.Map`**. It shows the name of the event, the object returned when the event fires, and a description.

Table 1: Map Control Events

L.Map Events		
Name	**Return Type**	**When Fired**
`click`	**`MouseEvent`**	When the user clicks (or taps) the map
`dblclick`	**`MouseEvent`**	When the user double-clicks (or double-taps) the map.
`mousedown`	**`MouseEvent`**	When the user pushes the mouse button while over the map.
`mouseup`	**`MouseEvent`**	When the user releases the mouse button while over the map.
`mouseover`	**`MouseEvent`**	When the mouse pointer enters the map.
`mouseout`	**`MouseEvent`**	When the mouse pointer leaves the map.
`mousemove`	**`MouseEvent`**	While the mouse pointer is moving over the map.
`contextmenu`	**`MouseEvent`**	When the user pushes the right mouse button while over the map (or uses a

L.Map Events		
Name	**Return Type**	**When Fired**
		long press on a mobile device). If you handle this event, it prevents the usual context menu from appearing.
focus	**Event**	When the map control gains the focus (via clicking, panning, or tabbing to it).
blur	**Event**	When the map control loses the focus.
preclick	**MouseEvent**	Prior to the click event occurring. This is useful if you want to pre-empt the click behavior.
load	**Event**	When the map is initialized and its center and zoom level is fixed for the first time.
unload	**Event**	When the map is destroyed using the **.remove()** method.
viewreset	**Event**	When the map needs to redraw.
movestart	**Event**	When the map extent starts to change (such as when the user starts dragging the map).
move	**Event**	When the map extent (center point or zoom level) changes.
moveend	**Event**	When the map extent stops changing (such as when the user stops dragging the map).
dragstart	**Event**	When the user starts dragging the map.
drag	**Event**	While the user is dragging the map.
dragend	**DragEndEvent**	When the user stops dragging the map.
zoomstart	**Event**	When the map zoom level is about to change (before zoom animation occurs).
zoomend	**Event**	When the map zoom level changes.

L.Map Events		
Name	**Return Type**	**When Fired**
`zoomlevelschange`	`Event`	When the number of zoom levels in the map changes due to the addition or removal of a layer.
`resize`	`ResizeEvent`	When the map control is resized.
`autopanstart`	`Event`	When the map begins to autopan as a result of a popup being opened.
`layeradd`	`LayerEvent`	When a new layer is added to the map.
`layerremove`	`LayerEvent`	When a layer is removed from the map.
`baselayerchange`	`LayerEvent`	When the layer control changes the current base layer.
`overlayadd`	`LayerEvent`	When the layer control selects an overlay.
`overlayremove`	`LayerEvent`	When the layer control deselects an overlay.
`locationfound`	`LocationEvent`	When geolocation is successful.
`locationerror`	`ErrorEvent`	When geolocation fails.
`popupopen`	`PopupEvent`	When a popup is opened via `.openPopup()`
`popupclose`	`PopupEvent`	When a popup is closed via `.closePopup()`

This is the list of events for just one of the controls in Leaflet.js (although arguably the most complex). All the other controls emit their own events, and this is what makes it possible to create highly interactive web mapping applications.

In this chapter, we're going to be using the map control's events to demonstrate event handling in Leaflet.js. For information about the events belonging to the other controls, check out the Leaflet.js documentation for the control you are interested in.

Handling events

In order to work with events, you need to know:

- Which event you want your application to start noticing

- The name of the event
- The type of data returned by the event when it fires

Let's demonstrate this with an example. Suppose we want to build an application that allows a user to click on the map and display the map coordinates at the point they clicked, in a popup.

Looking at Table 1, we can see that the event we want to monitor is the map's **`click`** event. In order to start capturing click events, we need to create an event handler for that event. We can do this using the map's **`.on()`** method, which takes as its first parameter the name of the event that we want to handle (**`click`**), and a function (the *callback*) that will respond to the event as its second parameter.

We can wire up the event handler and test that it is working using the following simple code:

Code Listing 34: Handling the map's click event

```
...
<script type="text/javascript">
    function init() {
        var map = L.map('map').setView([52.187, -1.274],7);
        // OSM Mapnik
        var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
        L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; ' + osmLink,
            maxZoom: 18,
        }).addTo(map);

        map.on("click", function() {
            alert("You clicked the map!");
        })
    }
</script>
...
```

If your code is correct and free of syntax errors, you can launch the app and click on the map, and your handler should fire and display a message box:

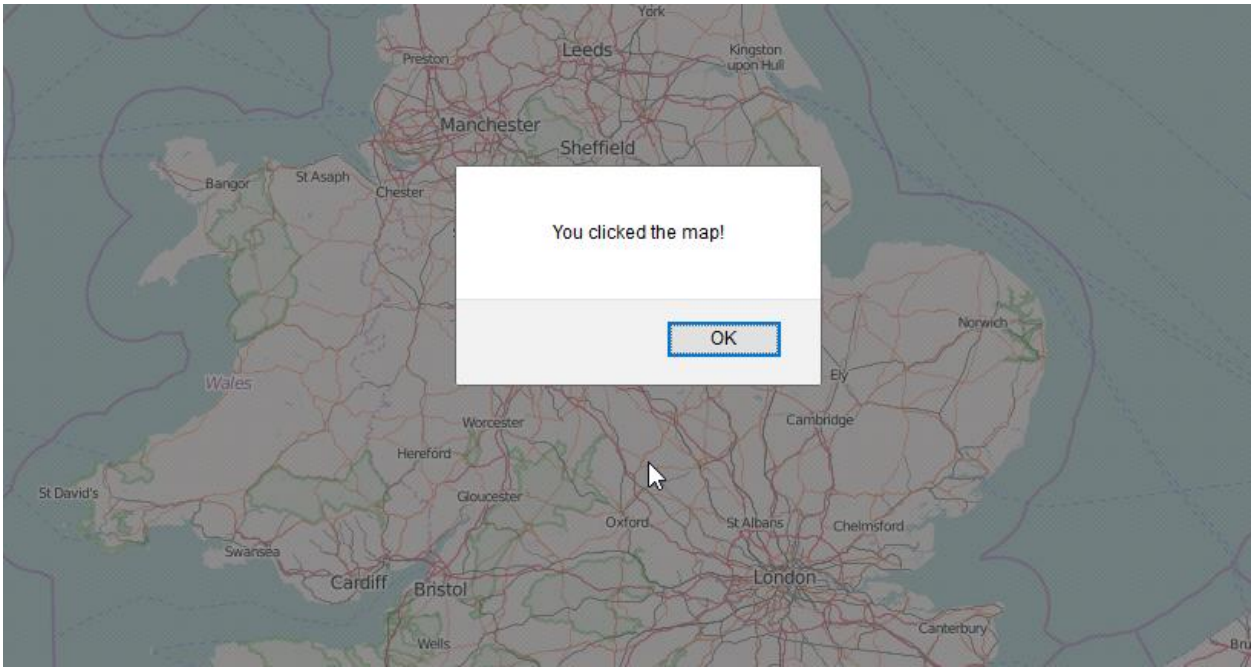


Figure 28: The map captures the click event

If you have gotten this far, then you know that your application is listening to the map click event. Great! The next thing to do is to capture the information that the map click event is just itching to pass to your event handler, but that is currently being ignored.

In order to do something with the event data, you need to know what type of data is being returned and then adjust your callback function to intercept and process it.

Looking at Table 1, you can see that the map's click event returns an object of type **`MouseEvent`**. The Leaflet.js documentation for **`MouseEvent`** provides the following information:

Table 2: MouseEvent properties:

MouseEvent		
property	**type**	**description**
`latlng`	**`LatLng`**	The geographical point where the mouse event occurred.
`layerPoint`	**`Point`**	Pixel coordinates of the point where the mouse event occurred relative to the map layer.
`containerPoint`	**`Point`**	Pixel coordinates of the point where the mouse event occurred relative to the map container.

MouseEvent		
property	**type**	**description**
`originalEvent`	**`DOMMouseEvent`**	The original DOM mouse event fired by the browser.

Looking at this table, you can see that **`MouseEvent`** extends **`DOMMouseEvent`**, which is the original event object for a mouse click on any DOM element in a web page. It just adds a few new properties relevant to the map. If we need the information from the original event, we can drill into the **`MouseEvent`**'s **`originalEvent`** property.

However, for our purposes, we don't need to do that. Our goal is to display the geographical coordinates of the point at which the user clicked, and that information is available in the **`MouseEvent`**'s **`latlng`** property. With that in mind, we can adjust our callback to handle the event and report the information back to the user.

Code Listing 35: Displaying map click coordinates in a message box

```
...
    map.on("click", function(e) {
        var lat = e.latlng.lat;
        var lng = e.latlng.lng;
        alert("You clicked the map at " + lat + "," + lng);
    })
...
```

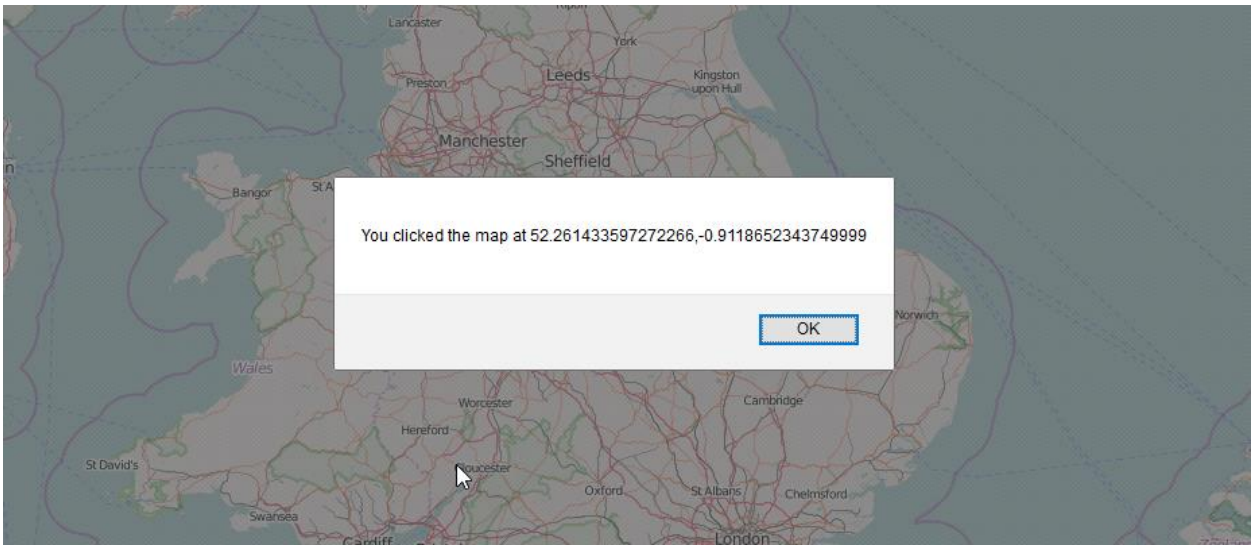


Figure 29: Displaying the map click coordinates in a message box

That's all well and good, but the use of a message box is a bit intrusive because it's modal and prevents us from interacting with the map. Besides which, our original task was to display the information in a popup. So why don't we use the knowledge we gained in the previous chapter to add a new marker object at the point where we clicked, so that we can see where it is on the map and perhaps refer to it again later once we've clicked elsewhere? While we're at it, we'll use a running counter of map clicks and format the output nicely in HTML:

Code Listing 36: Display map coordinate data as a marker with a popup

```
...
    var clickCount = 0;
    map.on("click", function(event) {
        var lat = event.latlng.lat;
        var lng = event.latlng.lng;
        ++clickCount;
        var popupContent = "<h2>Marker " + clickCount + "</h2>"
            + "<p>Map coordinates:</p>"
            + "<ul><li>Latitude: " + lat + "</li>"
            + "<li>Longitude: " + lng + "</li></ul>";

        var myMarker = L.marker([lat, lng],
        {
            title: "Click to see map coordinates"
        }).bindPopup(popupContent).addTo(map);
    })
...
```

Figure 30: Markers placed at click coordinates

Let's try something a little more complex to really get the hang of working with events. We're going to implement a very simple *geofence*. A geofence is basically a "virtual barrier." When something enters a specified area, an alert is triggered.

In our example, the geofenced area is going to be a rectangle that has appeared mysteriously in the middle of the Irish countryside, and our "something" is going to be one or more markers.

Figure 31: Our "geofence" and markers

Were going to make our markers draggable so that the the user of our application can move them around the countryside. Making a marker draggable is as simple as setting its **`draggable`** property to **`true`**.

Then we're going to wire up two event handlers to monitor the marker's **`dragstart`** and **`dragend`** events. As the names suggest, these events are fired when the user commences and finishes dragging the marker, respectively.

Because we have three markers, we're going to create just one event handler for each of the **`dragstart`** and **`dragend`** events, and the markers will share them. We'll dig into the **`Event`** object that gets passed into our callback to work out which marker raised the event.

Here's our starting point:

Code Listing 37: Geofence code, showing event handler stubs

```
...
<script type="text/javascript">

    var theRectangle;

    function init() {
```

```
        var map = L.map('map').setView([52.96228865548326, -
7.499542236328124],10);

        var mapLink = "Esri";
        var sourcesLink ="Contributors";
        // Esri World Imagery Service
        L.tileLayer(
                'http://server.arcgisonline.com/ArcGIS/rest/services/. . .
{
                attribution: '&copy; ' + mapLink + ', ' + sourcesLink,
                maxZoom: 18,
        }).addTo(map);

        theRectangle = L.rectangle(
            [[52.947728, -7.820206],[53.013874, -7.637558]],
            {
                color: "red",
                weight: 5,
                fillColor:"yellow",
            }
        ).addTo(map);

        var marker1 = L.marker([53.085694, -7.459030],
                               {draggable:true}).addTo(map);
        marker1.on("dragstart", dragStartHandler);
        marker1.on("dragend", dragEndHandler);
        var marker2 = L.marker([52.875678, -7.772141],
                               {draggable:true}).addTo(map);
        marker2.on("dragstart", dragStartHandler);
        marker2.on("dragend", dragEndHandler);
        var marker3 = L.marker([53.033699, -7.903976],
                               {draggable:true}).addTo(map);
        marker3.on("dragstart", dragStartHandler);
        marker3.on("dragend", dragEndHandler);
    }

    function dragStartHandler(e) {
        // starting marker drag
    }

    function dragEndHandler(e) {
        // ending marker drag
    }
</script>
...
```

At this point you might want to make sure that the **dragStartHandler()** and **dragEndHandler()** events fire when you drag the markers around. You can log something to the browser console using **console.log("message")**, which you can view in your browser's developer tools, or log something to the browser console using **console.log("message")**, which you can view in your browser's developer tools. I use Firefox with the amazing Firebug extension, but all the main browsers these days have pretty good developer tools you can use to debug your scripts.

The first thing we're going to do is create a nice little visual effect that makes the markers partially transparent while they're being dragged, and restores them to full opacity when they reach their destination. To achieve this, we need to know which marker raised the events, so we can set the opacity on that marker and leave the others alone.

Looking at the documentation, we can see that the **dragstart** and **dragend** events both pass different event objects into their callbacks:

dragstart	Event	Fired when the user starts dragging the marker.
drag	Event	Fired repeatedly while the user drags the marker.
dragend	DragEndEvent	Fired when the user stops dragging the marker.

Figure 32: The marker's drag events

Let's dig further into the documentation to look at those event objects. First, the **Event** object for the **dragstart** event:

property	type	description
type	String	The event type (e.g. 'click').
target	Object	The object that fired the event.

Figure 33: The Event object

The **Event** object, as the documentation helpfully points out, is the basis of all the other event classes, and they include its properties. One of those properties, **target**, is absolutely vital for us to be able to work out which marker raised an event.

Now the **DragEndEvent** object, for the **dragend** event:

property	type	description
distance	Number	The distance in pixels the draggable element was moved by.

Figure 34: The DragEndEvent object

DragEndEvent includes a **distance** property, so we can work out how far a particular marker was dragged. That's interesting for other use cases, but not relevant for what we are trying to achieve in this example. So for the **dragend** event, we'll just be using properties on **DragEndEvent**'s parent class: **Event**.

We want to work out which marker raised the **dragstart** event and set its opacity to 0.5, which is half-transparent. We also want to know which marker raised the **dragend** event so we can reset its opacity to 1.0 (fully opaque). We can use the event object's **target** property to retrieve the marker that raised the event, and then call its **setOpacity()** method:

Code Listing 38: Setting the marker transparency during the drag operation

```
...
    function dragStartHandler(e) {
        // starting marker drag
        e.target.setOpacity(0.5);
    }

    function dragEndHandler(e) {
        // ending marker drag
        e.target.setOpacity(1.0);
    }
...
```

Figure 35: Marker becoming partially transparent during the drag operation

That's great. We can now move our markers around our map, including into the geofenced area, and we've got some nice transparency effects.

What we now need to work out is if any of our markers end up in the geofence. If they do, then we can change their appearance to highlight that fact.

First, we need to work out the extent of the rectangle. The **`Rectangle`** class implements the abstract class **`Path`**, which gives it access to all of **`Path`**'s properties, events, and methods. This includes the **`getBounds()`** method, which returns a **`LatLngBounds`** object representing the bounding rectangle that encloses our rectangle. In the case of a rectangle, the bounding rectangle that encloses the rectangle is exactly the same size and shape as the rectangle it encloses! But the other overlays we have covered, such as the **`Polygon`**, also implement **`getBounds()`**, and their enclosing rectangles could be a very different shape:

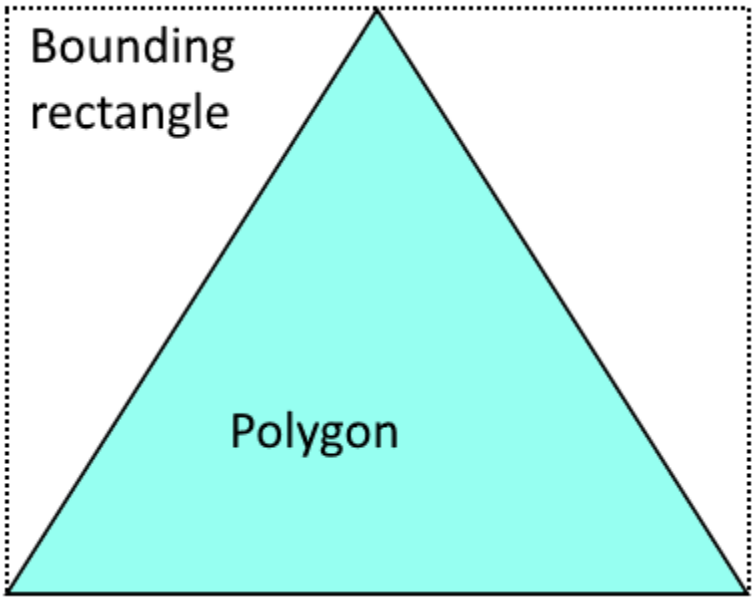


Figure 36: The bounding rectangle of a polygon

Having retrieved the **`LatLngBounds`** object, we can then call one of its **`contains()`** methods, passing in the marker's map coordinates, to establish whether the rectangle encloses the marker:

`contains( <LatLngBounds> otherBounds )`	`Boolean`	Returns `true` if the rectangle contains the given one.
`contains( <LatLng> latlng )`	`Boolean`	Returns `true` if the rectangle contains the given point.
`intersects( <LatLngBounds> otherBounds )`	`Boolean`	Returns `true` if the rectangle intersects the given bounds.
`equals( <LatLngBounds> otherBounds )`	`Boolean`	Returns `true` if the rectangle is equivalent (within a small margin of error) to the given bounds.

Figure 37: The LatLngBnds object's contains, intersect,s, and equals methods

You can see from the extract from the Leaflet.js API reference that **`LatLngBnds`** contains two versions of **`contains()`**: one for points (such as the location of our marker), and one for other rectangles. It also includes methods to check whether one rectangle intersects another one, or for the equivalency of two rectangles.

We need to implement this check in our **`dragend`** event handler. If **`contains()`** returns **`true`**, then we know that our marker has entered the geofence, and we need some way to alert the user. In this example, I'm just using a bit of CSS to change the marker icon background color, but you could switch the marker's icon instead, to achieve a different effect:

Code Listing 39: Checking if the new marker position falls within the geofence rectangle

```
...
    function dragEndHandler(event) {
        // ending marker drag
        event.target.setOpacity(0.5);
        if (theRectangle.getBounds().contains(event.target.getLatLng())) {
            event.target.valueOf()._icon.style.backgroundColor = 'red';
        }

    }
...
```

Figure 38: Geofenced marker

Chapter 5 Accessing External Data Sources

So far we've seen how to add data by hardcoding it into our mapping applications. If you're building anything that's not completely trivial, then that approach is just not going to cut it. Nobody wants to create marker code for 10,000 different locations! And we haven't even discussed the possibility that a user might want to change data via your application.

Thankfully, there are many different ways of getting data into your Leaflet.js applications. In this chapter, we look at a few of the most popular techniques you can use.

Accessing data in a database

Databases are the go-to platform for storing data of any description, so let's consider how we can store and retrieve data with a spatial component for use in our web maps.

We have been creating client-side applications that run on the browser. Browser-based applications don't typically access database data directly. They either involve some server-side code, or some sort of middleware—normally a web service of some description—which accesses the database and makes it available to the page.

One of the most common methods of building database-driven web applications is to use PHP for the server-side scripting and MySQL as the back-end database. Both have been around for a long time and are freely available, so even though there are newer, fancier platforms available, we'll stick with PHP and MySQL for our first example.

The database

You can download MySQL directly from Oracle's website, or, as I have done, use a "friendlier" package to do the installation and configuration for you. These are bundled programs with some sort of administration GUI that bundle the installation of a LAMP stack (Linux, Apache, MySQL, and PHP). Although Linux is a popular choice for hosting web servers, for development purposes you'll often want to install a Windows or Mac-specific flavor of the LAMP stack. As I mentioned in Chapter 1, I use XAMPP for this. You need to ensure that the MySQL service is running.

In my environment, I have MySQL running and listening to requests on the default port (3306):

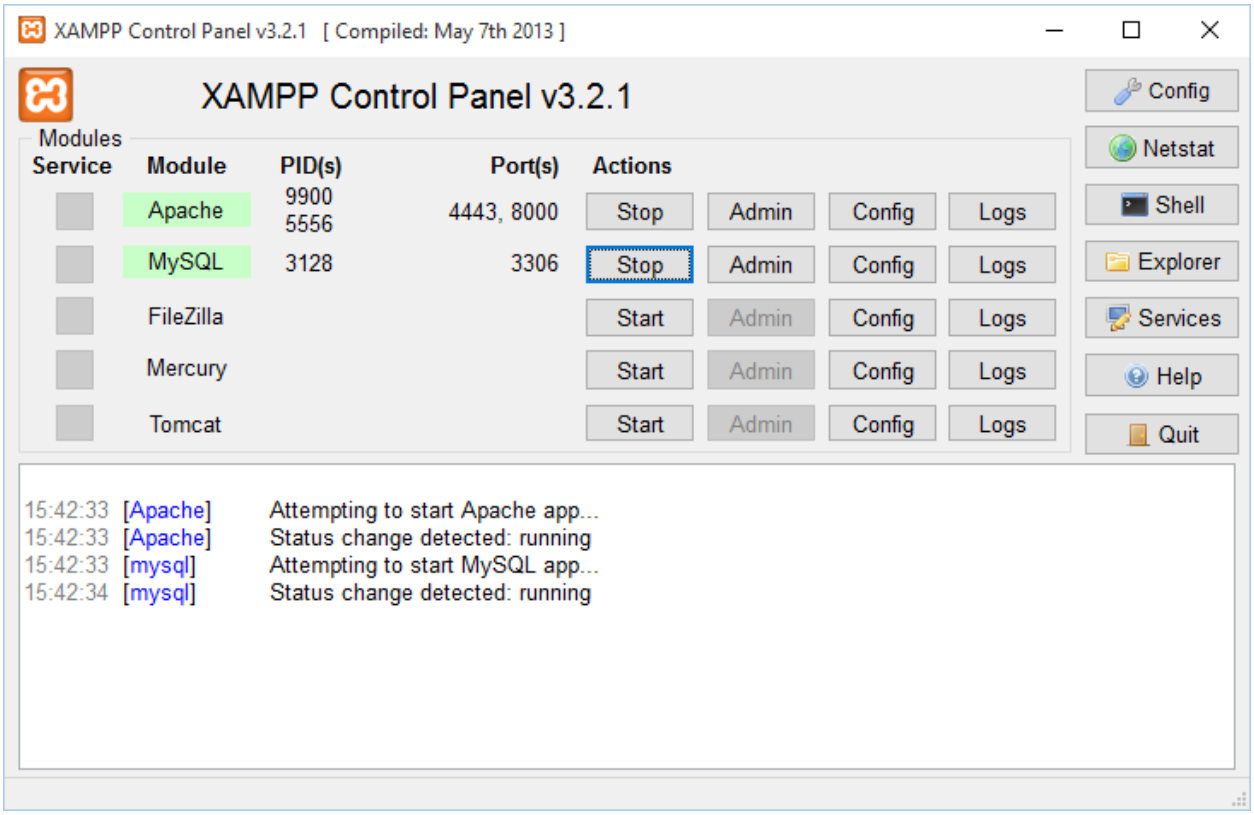


Figure 39: XAMPP control panel showing the MySQL service running

Now that I've got a running database server, I need to create a database and populate it with some data I can then consume within my Leaflet.js application. XAMPP bundles a graphical user interface to MySQL called myPhpAdmin, but I like working at the command line, so that's what I'll demonstrate here.

First, I need to log into the MySQL server. I can do this from the Shell button in the XAMPP Control Panel, or I could add the MySQL binaries to my PATH and access it from the standard terminal. XAMPP configures MySQL with a root account, but no password, so let's give the root user a password of **leaflet**.

```
# mysqladmin.exe -u root password leaflet
```

For this example, I'm going to use the locations of various coffee shops around the United States. I have created a "dump" file called **`CoffeeShops.sql`**, which I have included within the **`ch05_consumingdata`** folder in this book's source code (see Chapter 1 for details of where to download this file from). If you execute the following command at the terminal prompt, it will create and populate the **`leafletDB`** database from the **`coffee.csv`** file (also provided):

```
# mysql -uroot -pleaflet < "C:\CoffeeShops.sql";
```

Note: The '#' character represents the command prompt. Do not enter it as part of the statement.

Once this process is complete, verify that the **`CoffeeShops`** table contains records by executing the following commands at the terminal prompt:

Code Listing 40: Verifying that the coffee shop data is in the database

```
# mysql -uroot -pleaflet
Warning: Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 151
Server version: 5.6.26 MySQL Community Server (GPL)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> USE leafletDB;
Database changed
mysql> SHOW TABLES;
+---------------------+
| Tables_in_leafletdb |
+---------------------+
| coffeeshops         |
+---------------------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM coffeeshops;
+----------+
| COUNT(*) |
+----------+
|     7664 |
+----------+
1 row in set (0.00 sec)

mysql>
```

The server-side code

That's the database end of things; now we need some server-side code to access the database and return results in a format suitable for our client application to read. I'm going to code this in PHP, using PHP Data Objects (PDO) to handle the interaction with the database. The code for this is either incredibly simple or really obtuse, depending on whether or not you know PHP.

Here's the code, which I'm saving in a file called **`coffee.php`** in the same directory I'll use to create the HTML page that will display the map.

Code Listing 41: The server-side PHP code that reads the database

```
# mysql -uroot -pleaflet
<?php
    $db = new PDO("mysql:host=localhost;dbname=leafletDB", "root",
"leaflet");
    $sql = "SELECT * FROM CoffeeShops WHERE City='Boston'";
    $rs = $db->query($sql);
    if (!$rs) {
        echo "An SQL error occured.\n";
        exit;
    }
    $rows = array();
    while($r = $rs->fetch(PDO::FETCH_ASSOC)) {
        $rows[] = $r;
    }
    print json_encode($rows);
    $db = NULL;
?>
```

Here's how this code works:

```
    $db = new PDO("mysql:host=localhost;dbname=leafletDB", "root", "leaflet");
```

This line provides the connection to the database.

```
    $sql = "SELECT * FROM CoffeeShops WHERE City='Boston'";
    $rs = $db->query($sql);
    if (!$rs) {
        echo "An SQL error occured.\n";
        exit;
    }
```

This code issues a query looking for all the coffee shops in the Boston area (there are a lot of coffee shops in our database, so let's focus in on a specific area) and checks to see if any records are returned. If not, it reports an error and quits.

```
    $rows = array();
    while($r = $rs->fetch(PDO::FETCH_ASSOC)) {
        $rows[] = $r;
```

```
}
```

This code fetches each row in the result set into an associative array.

```
print json_encode($rows);
```

This line of code prints the entire record set in JSON (JavaScript Object Notation). JSON is a lightweight data interchange format that is ideal for our client application to process. If you launch the **`coffee.php`** file in the browser, then you will see what this looks like:

Figure 40: The database data in JSON format

It might look like a load of garbled nonsense to you. Even though JSON is a recognized data interchange format in its own right, its roots are in JavaScript, where it is the syntax used to create a JavaScript object literal. Yes, that thing is just one great big JavaScript object, which makes it trivial to bring into our client application.

The line of PHP code that performs the magic of taking a result set from a database and converting it into JSON that can be streamed into our web page is this:

```
print json_encode($rows);
```

This sort of capability is not unique to PHP. With JSON fast eclipsing XML as the preferred way of slinging data around in a web environment, other server-side scripting languages, such as Ruby, VBScript, Perl, and C# have equivalent functionality.

The client application

We need a way for our web page to access the JSON data returned by **`coffee.php`**. The best way to do this is asynchronously so that we don't have to hold up our users by refreshing the page.

To achieve this, we need our web page to make an AJAX request to the **`coffee.php`** script. AJAX stands for Asynchronous JavaScript and XML. Wait a minute…XML?

Well, when the AJAX technique first surfaced, XML was the preferred data interchange format. But now JSON is preferred, so technically, AJAX should now really be called AJAJ. But AJAX as an acronym is ubiquitous regardless of the fact that, more often than not, JSON is the preferred format. Plus, it's easier to say.

Now, there are ways of achieving this in vanilla JavaScript using **`XMLHttpRequest`**, but they are pretty ugly. A much nicer, cleaner way of working with AJAX is to use JQuery. JQuery is a JavaScript framework that builds upon basic JavaScript to make certain tasks easier for developers. And AJAX is one of those tasks. If you want to learn more about JQuery (and you really should) check out *Succinctly JQuery*, by Cody Lindley.

In order to access JQuery, you must reference it in a **`<script>`** tag in your page, whether that points to a local version of the framework, or one hosted by a CDN (Content Delivery Network). Both Microsoft and Google make JQuery available via CDN, so I'm just going to reference Microsoft's (because Google gets all the love these days).

Here's the starting code for our client application. I have highlighted the **`<script>`** tag that references the JQuery CDN:

Code Listing 42: Client application, referencing the JQuery framework via a CDN

```
<!DOCTYPE html>
<html>

<head>
    <title>My Leaflet.js Map</title>
    <link rel="stylesheet"
href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css" />
    <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-
1.11.3.min.js"></script>
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js">
    </script>
    <style>
        html,
        body,
        #map {
            height: 100%;
        }
    </style>

    <script type="text/javascript">
        function init() {
            var map = L.map('map').setView([42.362, -71.085], 13);

            // OSM Mapnik
            var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
            L.tileLayer(
                'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
                    attribution: '&copy; ' + osmLink,
```

```
                    maxZoom: 18
                }).addTo(map);

            showCoffeeShops(map);
        }

        function showCoffeeShops(map) {

            // Retrieve and display coffee shop data

        }
    </script>
</head>

<body onload="init()">
    <div id="map"></div>
</body>

</html>
```

You can see from the code in *Code Listing 43* that we have built a very simple mapping application with an OSM Mapnik base layer. The **`init()`** function creates the map, adds the layer, and then calls the **`showCoffeeShops()`** function, which is where we turn our attention to next.

Code Listing 43: The completed showCoffeeShops() function

```
...
    function showCoffeeShops(map) {

        var mugIcon = L.icon({
            iconUrl: "./images/mug.png",
            iconSize: [25,25]
        });
        $.getJSON("coffee.php", function(data) {
            for (var i = 0; i < data.length; i++) {
                var location = new L.LatLng(data[i].Latitude,
data[i].Longitude);
                var name = data[i].Name;
                var addr1 = data[i].Address1;
                if(data[i].Address2.length > 1) {
                    addr2 = data[i].Address2 + "</br>";
                } else {
                    addr2 = "";
                }
                var cityzip = data[i].City + ", " + data[i].Zip;
                var visits = data[i].Visits;
```

```
                var marker = new L.Marker(location, {
                    icon: mugIcon,
                    title: name
                });
                var content = "<h2>" + name + "</h2>"
                    + "<p>" + addr1 + "</br>" + addr2
                    + cityzip + "</p>"
                    + "<p>Visits: <b>" + visits + "</b></p>";

                marker.bindPopup(content, {
                    maxWidth: '200'
                });
                marker.addTo(map);
            }
        });
    }
...
```

Let's have a look at the important bits.

First, let's take a look at the call to the JQuery function **`getJson()`**. The **`getJson()`** function is a wrapper around JQuery's **`ajax()`** function, with the **`dataType`** option set to **`json`**. This accesses our **`coffee.php`** script asynchronously and passes its output (the JSON representation of our **`CoffeeShops`** table in MySQL) into a callback as the parameter **`data`**. We then iterate through that data to access the individual result:

```
$.getJSON("coffee.php", function(data) {
    for (var i = 0; i < data.length; i++) {

     ...

    }
});
```

The next several lines just pull out the column values from each row in the result set. About the only thing we're doing with these values is formatting them for later display in a popup. So, for example, if the **`Address2`** field is blank, we're leaving it out so as not to have a big gap in our popup content. We're also concatenating the **`City`** and **`Zip`** column values so that they display on the same line:

```
var location = new L.LatLng(data[i].Latitude, data[i].Longitude);
var name = data[i].Name;
var addr1 = data[i].Address1;
if(data[i].Address2.length > 1) {
    addr2 = data[i].Address2 + "</br>";
} else {
    addr2 = "";
}
var cityzip = data[i].City + ", " + data[i].Zip;
var visits = data[i].Visits;
```

Finally, we're creating a marker for each record from the Latitude and Longitude column values, and binding a popup to display the other data we have gathered:

```
var marker = new L.Marker(location, {
    icon: mugIcon,
    title: name
});
var content = "<h2>" + name + "</h2>"
    + "<p>" + addr1 + "</br>" + addr2
    + cityzip + "</p>"
    + "<p>Visits: <b>" + visits + "</b></p>";
marker.bindPopup(content, {
    maxWidth: '200'
});
marker.addTo(map);
```

Note that we have also created a custom icon called **`mugIcon`** to show each location as a coffee cup on the map:

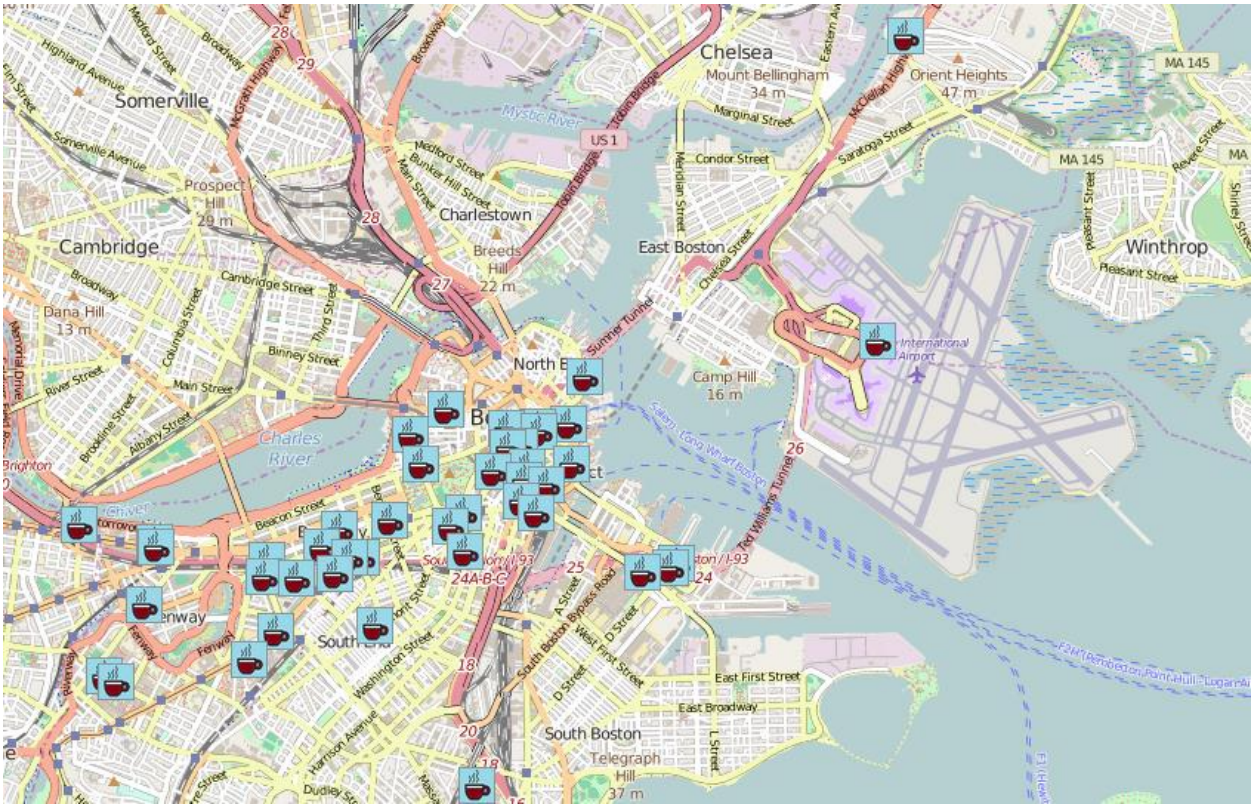


Figure 41: The finished application

Mashing up data with an API

Another way we can get ahold of data is by using a third-party API. This technique is what the cool kids refer to as creating a "mashup."

The number of APIs that you can access freely or via a paid subscription is phenomenal: just check out this website for inspiration. Many of these provide location data, and for those that don't (such as the Twitter APIs), it's pretty easy to geocode any content with place names. We'll have a look at geocoding in Chapter 6.

For now, let's use an API that does store location data when it's available, and one such API is Flickr Photo Search. Our task for this exercise is to create an application centered on the city of London in England, and display any Flickr photos that have been geotagged as being taken there.

In order to access the Flickr Photo Search API, we first need to specify what sort of information we want the API to return to us. Once we've done that, we get a URL consisting of various query parameters that specifies the request our application will make of the Flickr service.

First, visit the Flickr API App Garden page, just to get an idea of the huge range of APIs available from just this single provider. Then, click on the **flickr.photos.search** link.

This will take you to a page with a form where you need to specify the information you want the Flick Photo Search API to return to you. Make the following entries in the form:

- Set the **`text`** field to **London**. We're looking for all photos with "London" in the description somewhere.
- Set the **`has_geo`** field to **`1`**. This will return only photos with spatial coordinates.
- Set the **`extras`** field to **`geo, url_s`**. These two options return the spatial data and a small thumbnail of each image. We'll use the first to plot the location as a marker on the map and the second to display in that marker's popup.
- Finally, set the **`Output`** option at the bottom of the page to **`JSON`**. We like JSON.
- Leave all the other options alone.

Once you have entered all this information, click the **Call Method…** button. If everything works correctly, you should see a sample output and below it, a scary-looking custom URL you will need to paste into the application we are about to build.

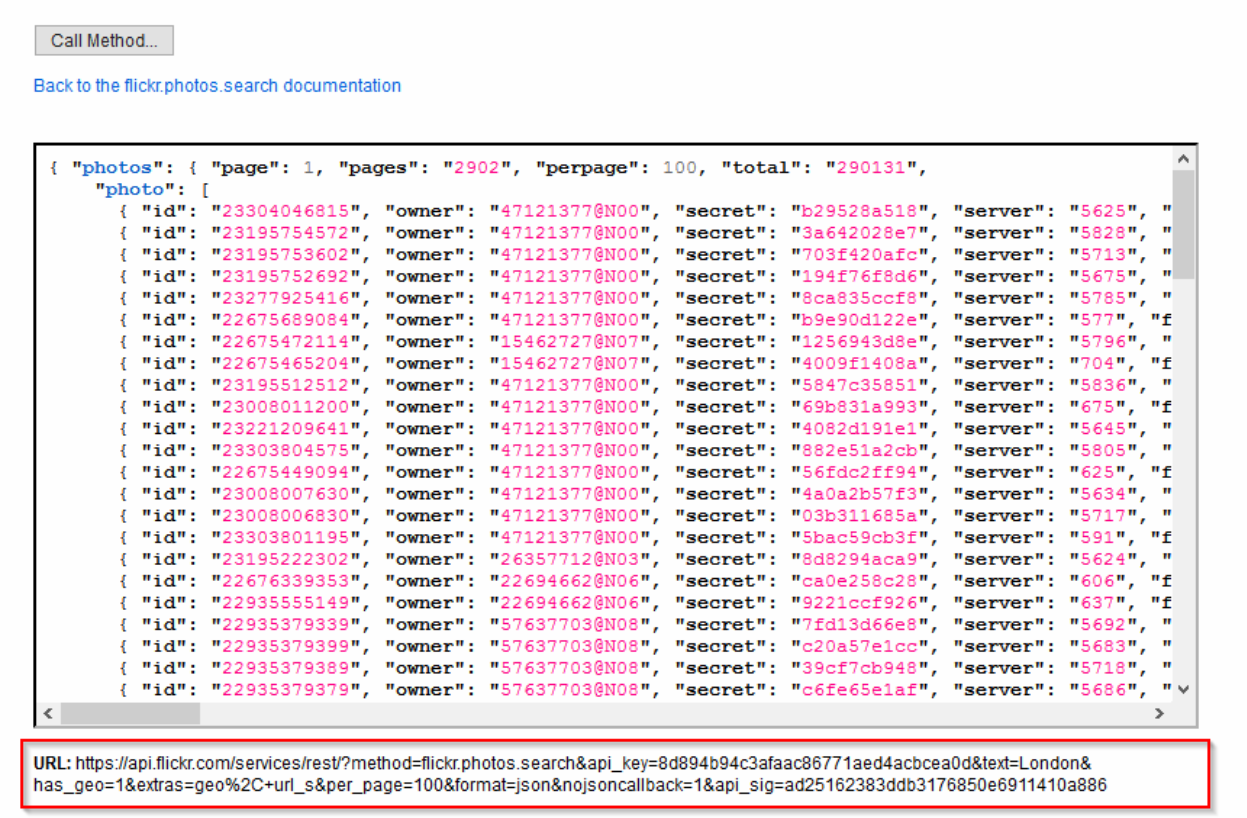


Figure 42: Successful configuration of the Flickr Photo Search API

So here is our starting point. We have a map centered on London with OSM Mapnik as a base layer, and we're calling a function called **`showPhotos()`**, where we are going to work our Flickr magic.

Code Listing 44: The starting point for our Flickr API application

```
<!DOCTYPE html>
<html>
<head>
    <title>My Leaflet.js Map</title>
    <link rel="stylesheet"
href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css" />
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></scri
pt>
    <script src="js/leaflet-layerjson.min.js"></script>
    </script>
    <style>
        html,
        body,
        #map {
            height: 100%;
        }
```

```
    </style>

    <script type="text/javascript">
        function init() {
            var map = L.map('map').setView([51.505, -0.106],14);

            // OSM Mapnik
            var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
            L.tileLayer(
                'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
                    attribution: '&copy; ' + osmLink,
                    maxZoom: 18
                }).addTo(map);
            showPhotos(map);
        }

        function showPhotos(map) {

            // Display Flickr photos
        }
    </script>
</head>

<body onload="init()">
    <div id="map"></div>
</body>
</html>
```

All is not quite the same as before in our MySQL example, however. In that example, we used JQuery to make our AJAX call to a PHP script that returned JSON data from the contents of a table in a database. In this example, we're going to use a Leaflet.js plug-in to make the AJAX request and handle the response.

Plugins are part of what makes Leaflet.js really cool. The Leaflet.js framework itself is tiny and extremely robust, and it has stayed that way because the maintainers only include core functionality in the framework. However, there is a rich base of community-developed plugins that extend Leaflet in many exciting ways. We'll have a look at other plugins in the next two chapters, but in this example we're going to use a plugin called **leaflet-layerJSON**, created by Stefano Cudini. See the project's GitHub page for more information.

The great thing about this plugin is that it will read the JSON data coming from the Flickr Photo Search API and automatically create markers and their associated popups from the data it returns. How cool is that?

Start by downloading the latest release from the **releases** tab on the leaflet-layer-JSON project page:

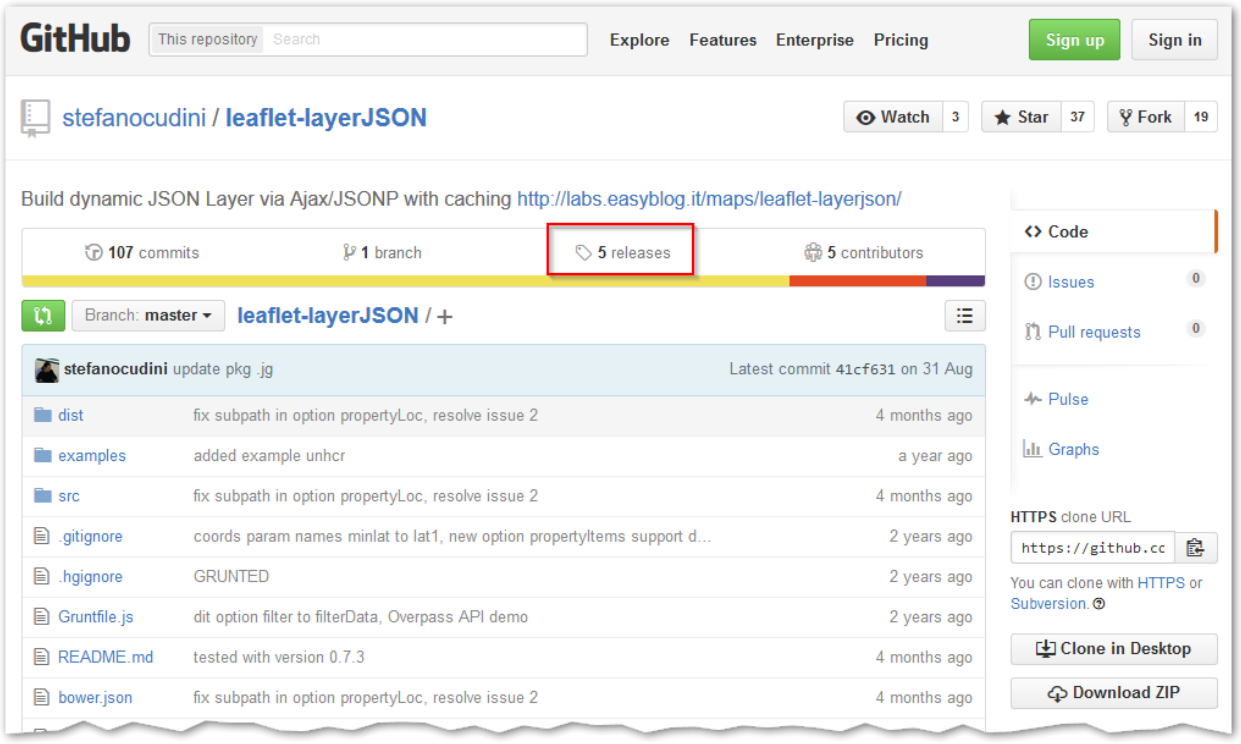


Figure 43: The leaflet-layerJSON project page on GitHub

This will give you a zip file that contains all the source code and build files for leaflet-layerJSON. There's a lot of stuff in there, but we only need the minified JavaScript file, which you can find in the zip file's **dist** folder. Create a folder in the same directory as your HTML page called **`js`**, and put the **`leaflet-layerjson.min.js`** file in there. Then, make sure you include a reference to that file in a **`<script>`** tag, as shown in *Code Listing 45*.

This is how it looks in Windows Explorer on my PC:

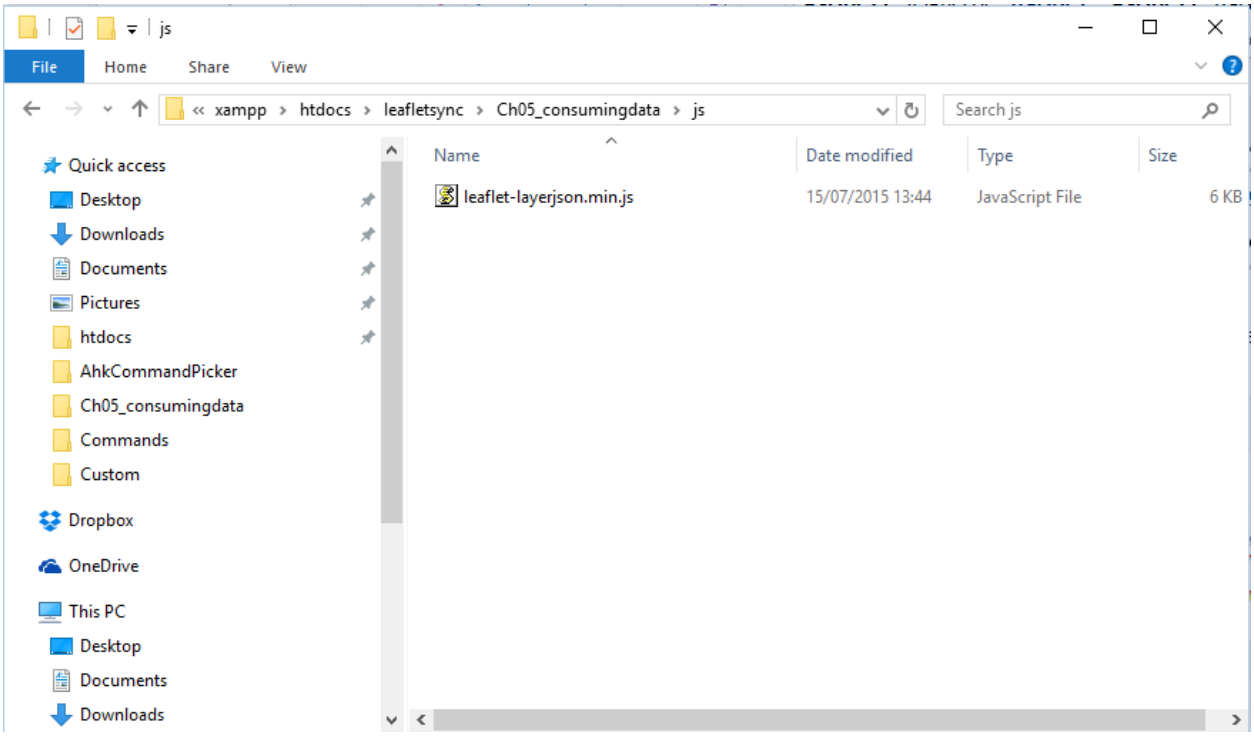


Figure 44: Location of the leaflet-layerJSON plugin

Now we can write the code for our **`showPhotos()`** function, as shown in *Code Listing 45*. Notice that I have used placeholders for the Flickr API and signature values. These are specific to the Photo Search API URL Flickr generated for you earlier.

Code Listing 45: Completing the showPhotos() function

```
...
function showPhotos(map) {

    var url1 =
"https://api.flickr.com/services/rest/?method=flickr.photos.search";
    var url2 = "&api_key=[your value]";
    var url3 = "&text=London&has_geo=1&extras=geo%2C+url_s"
    var url4 = "&per_page=100&format=json&nojsoncallback=1"
    var url5 = "&api_sig=[your value]"

    var popupContent = function(data,markers) {
                return "<strong>"
                    + data.title
                    + "</strong><br><img src='"
                    + data.url_s+"'>"|| null;
    };

    jsonLayer = new L.LayerJSON({
        url: url1 + url2 + url3 + url4 + url5,
        propertyItems: 'photos.photo',
        propertyLoc: ['latitude','longitude'],
```

```
        buildPopup: popupContent
    });

    map.addLayer(jsonLayer);
} ...
```

I have split the long URL provided by the Flickr Photo Search API into several different strings, just so you can see what it consists of.

```
var url1 = "https://api.flickr.com/services/rest/?method=flickr.photos.search";
var url2 = "&api_key=[api key value]";
var url3 = "&text=London&has_geo=1&extras=geo%2C+url_s"
var url4 = "&per_page=100&format=json&nojsoncallback=1"
var url5 = "&api_sig=[api sig value]"
```

Next, I created a **popupContent** variable that contains a function. This function will create the content of the popups that the **L.LayerJSON** object (provided by leaflet-layerJSON) will render for us. Note how I am including the **title** and **url_s** properties that we requested when we created the API URL in the flickr.photos.search page. These correspond to the title of any photos returned by the API and a small thumbnail image, respectively.

```
var popupContent = function(data,markers) {
            return "<strong>"
                + data.title
                + "</strong><br><img src='"
                + data.url_s+"'>"|| null;
};
```

What follows is our call to the Flickr Photo Search API, neatly handled by the **L.LayerJSON** object in the leaflet-layerJSON plugin. The **propertyItems** property is the element in the JSON that stores the data we're interested in. The **propertyLoc** property returns the fields in the JSON that contain coordinate data, and the **buildPopup** property uses the function we stored in the **popupContent** variable to create the popup contents:

```
jsonLayer = new L.LayerJSON({
    url: url1 + url2 + url3 + url4 + url5,
    propertyItems: 'photos.photo',
    propertyLoc: ['latitude','longitude'],
    buildPopup: popupContent
});
```

Once we have instantiated the **L.LayerJSON** object, we can add it to our map in the normal way using **map.addLayer(jsonLayer)**.

Let's run the application and click on a few markers. Note that this is a live feed, so your results will probably be significantly different from mine:

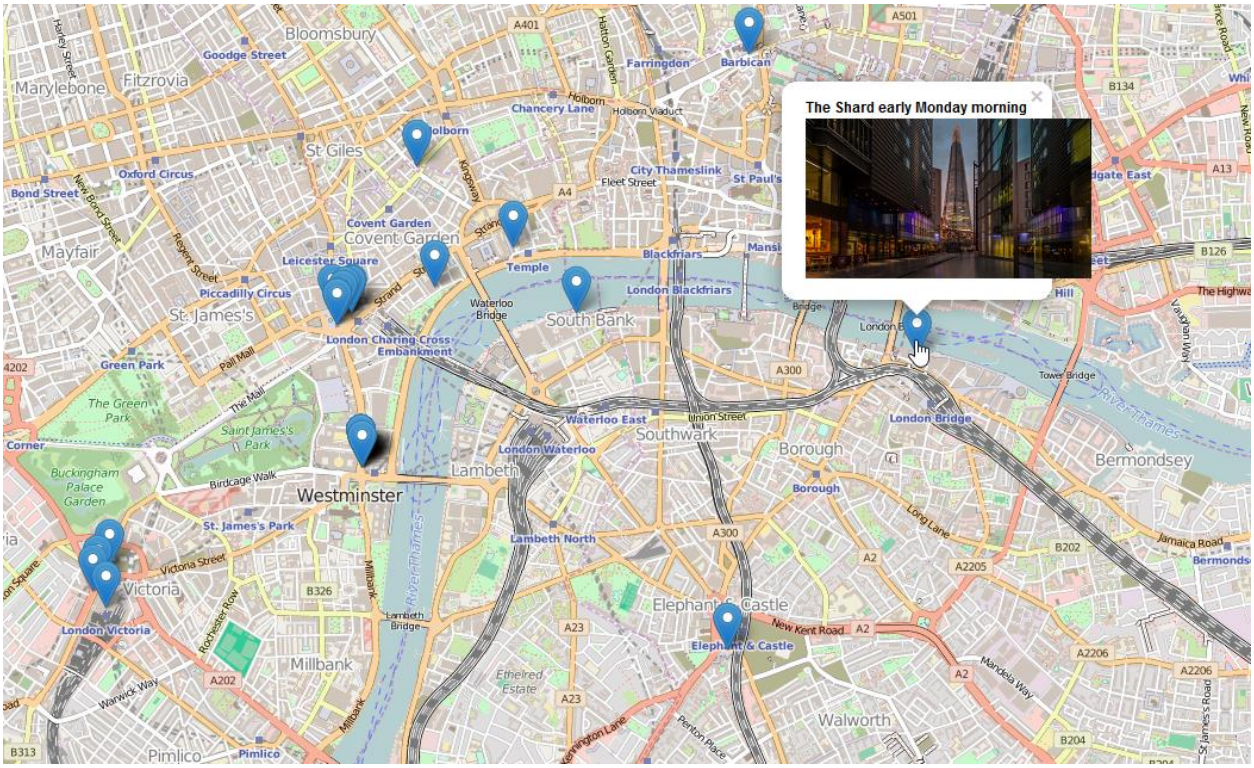


Figure 45: The Flickr API application, with dodgy-looking popups

Well it all appears to be working, but our popups aren't big enough to contain the images returned by Flickr. We can fix that with a bit of simple CSS. I used my browser's developer tools to find out which classes in the Leaflet.js API were responsible for styling the popup, and then made a small change in the **<style>** section at the head of my page:

Code Listing 46: Auto-sizing the popups with CSS

```
...
    <style>
        html,
        body,
        #map {
            height: 100%;
        }
        .leaflet-popup-content {
            width:auto !important;
        }
    </style>
...
```

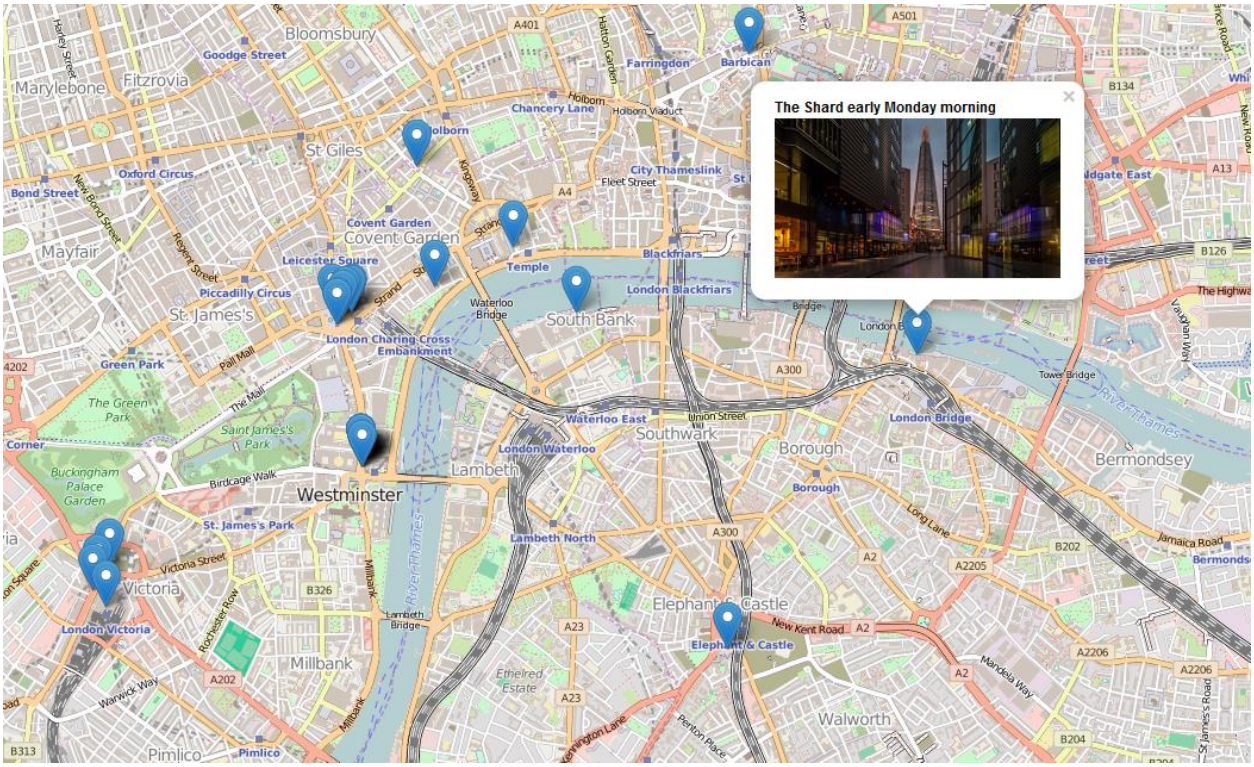


Figure 46: The final Flickr Photo Search API application

Loading KML data into your Leaflet application

As I'm sure you're beginning to appreciate, there are many different ways of getting data into your Leaflet.js application. But let's consider one more.

KML stands for Keyhole Markup Language. The full name doesn't really mean much to anyone anymore, but the acronym certainly does. It's an XML-based format for representing geographic data that Google has used extensively in its Google Earth and Google Maps applications, and has now found a considerable following elsewhere. KML has now been recognized as an official international standard of the Open Geospatial Consortium, an organization that has more than a passing interest in how we store and share geographic data.

And the Keyhole bit? That's just a hangover from the days when Google Earth was called Keyhole Earth Viewe,r until Google acquire Keyhole Inc. in 2004.

Anyway, KML is pretty cool because you can represent just about any geographic data with it in either two or three dimensions. And there's a ton of it out there, just waiting for you to suck into your Leaflet.js mapping applications. Just do a Google search for **`filetype:kml`**, and you should find plenty—that's how I found the USGS Earthquakes Google Earth/KML Files page. Let's use their Real-Time Earthquakes KML feed in a Leaflet.js application.

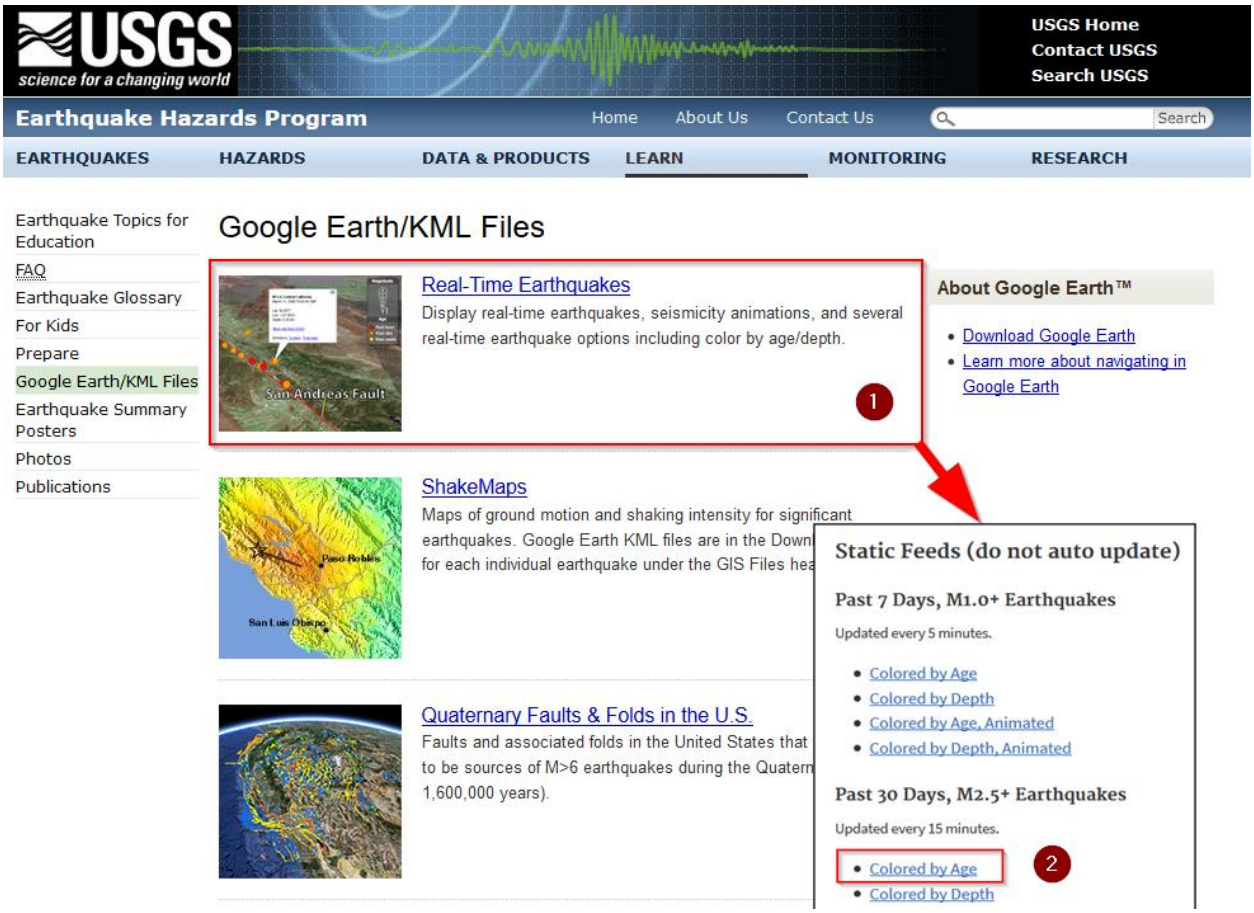
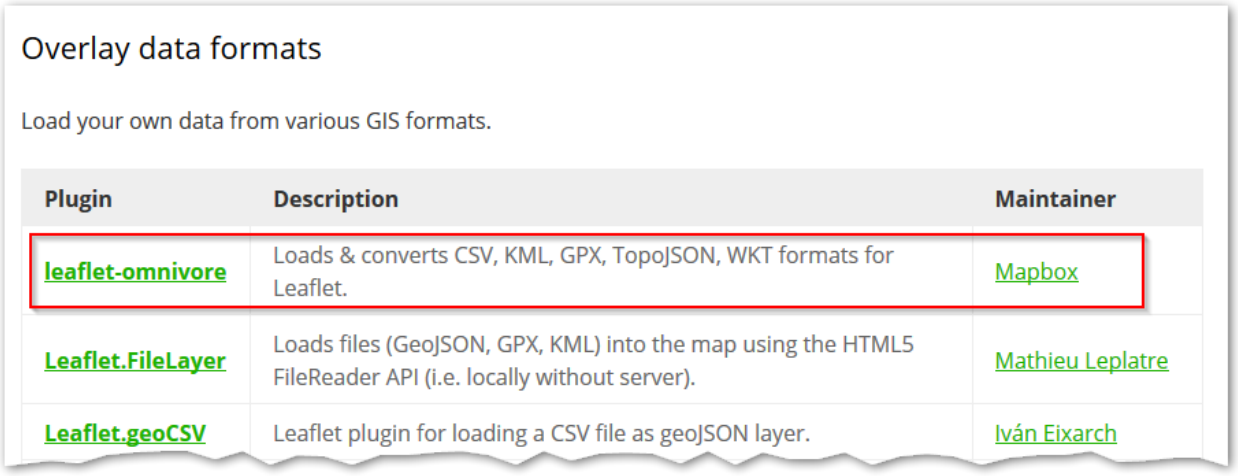


Figure 47: The USGS repository for KML files

I'm going for the **Past 30 Days, M2.5+ Earthquakes Colored by Age** KML file from **Static Feeds**, shown in *Figure 47*.

If you want to follow along, create a folder called **`kml`** in the same directory as the web page you are about to create, and put the **`2.5_month_age.kml`** file in there.

The next thing we need is something in Leaflet.js that recognizes and can deal with KML. There's nothing in the core framework, so again we're looking for a plugin. The Leaflet.js Plugins page lists "official" plugins, so that's a good place to start looking. If I search for "KML," I find something that looks very suitable: leaflet-omnivore, which reads in all sorts of different formats as overlays, including KML.



Overlay data formats

Load your own data from various GIS formats.

Plugin	Description	Maintainer
leaflet-omnivore	Loads & converts CSV, KML, GPX, TopoJSON, WKT formats for Leaflet.	Mapbox
Leaflet.FileLayer	Loads files (GeoJSON, GPX, KML) into the map using the HTML5 FileReader API (i.e. locally without server).	Mathieu Leplatre
Leaflet.geoCSV	Leaflet plugin for loading a CSV file as geoJSON layer.	Iván Eixarch

Figure 48: leaflet-omnivore, on the Leaflet.js plugins page

With a quick hop across to GitHub, I can download the latest release as a zip file, extract **`leaflet-omnivore.min.js`**, and place it in my **`js`** directory, just as we did in the MySQL example. The way leaflet-omnivore works is to download the KML data and format it as a Leaflet GeoJSON layer.

A bit of reading on the GitHub page gives me all the information I need to know to create my application. Here's the code.

Code Listing 47: Loading KML data with the leaflet-omnivore plugin

```
<!DOCTYPE html>
<html>

<head>
<title>My Leaflet.js Map</title>
<link rel="stylesheet"
href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css" />
<script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></scri
pt>
<script src='js/leaflet-omnivore.min.js'></script>
</script>
<style>
    html,
    body,
    #map {
        height: 100%;
    }
</style>

<script type="text/javascript">
    function init() {
        var map = L.map('map').setView([0,0], 2);
```

```
        // OSM Mapnik
        var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>";
        L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
                attribution: '&copy; ' + osmLink,
                maxZoom: 18
            }).addTo(map);


        var runLayer = omnivore.kml('kml/2.5_month_age.kml')
            .on('ready', function() {
                map.fitBounds(runLayer.getBounds());

                runLayer.eachLayer(function(layer) {
                    layer.bindPopup(layer.feature.properties.name);
                });
            }).addTo(map);
    }
</script>
</head>

<body onload="init()">
    <div id="map"></div>
</body>

</html>
```

Let's pick that code apart and see how it works.

First, we're referencing the leaflet-omnivore plugin code that we downloaded into our **`js`** folder:

```
<script src='js/leaflet-omnivore.min.js'></script>
```

Then we're creating the map in the usual way, not being too bothered about the initial coordinates because we're going to zoom to the extent of the KML layer when it appears on our map.

We create a new layer called **`runLayer`**, an object of class **`omnivore.kml`**, which references the earthquake data KML file we downloaded to our **`kml`** folder. We're immediately wiring up an event handler to listen for the KML layer's **`ready`** method, which will tell us when it is loaded.

In the **`callback`** function for the **`ready`** event, we first get the extent of the KML layer by calling its **`getBounds()`** method. We pass the resulting **`LatLngBounds`** object into the map's **`fitBounds()`** method, which sets the extent of the map to the extent of the KML layer.

We then iterate through each feature in the KML layer using the layer's utility method **`eachLayer()`**, and set the contents of the popup to be the name field associated with that feature. (Don't be confused by the use of the term "layer" in **`eachLayer`**; that's how KML views its contents, but we can think of them as individual features within a single KML layer.) We add the result to the map in the usual way.

```
var runLayer = omnivore.kml('kml/2.5_month_age.kml')
    .on('ready', function() {
        map.fitBounds(runLayer.getBounds());

        runLayer.eachLayer(function(layer) {
            layer.bindPopup(layer.feature.properties.name);
        });
    }).addTo(map);
```

How did I come up with that **`feature.properties.name`** path to the data we needed for the popup? It wasn't magic. I just used my browser's developer tools to put a breakpoint on that line and examine the contents of the layer variable:

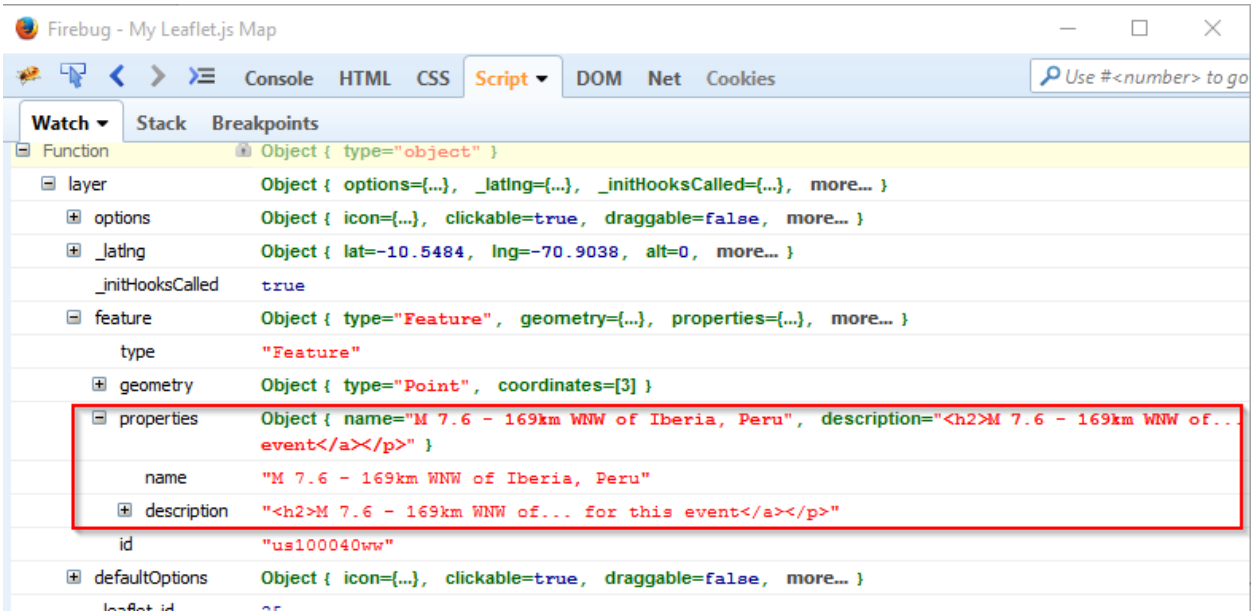


Figure 49: Examining the contents of the earthquake data in Firebug to get the right information for our popups

And what we get as a result is a bunch of markers representing the last 30-days worth of earthquakes of magnitude 2.5 and above, with a popup providing details of the location and actual magnitude of each earthquake:

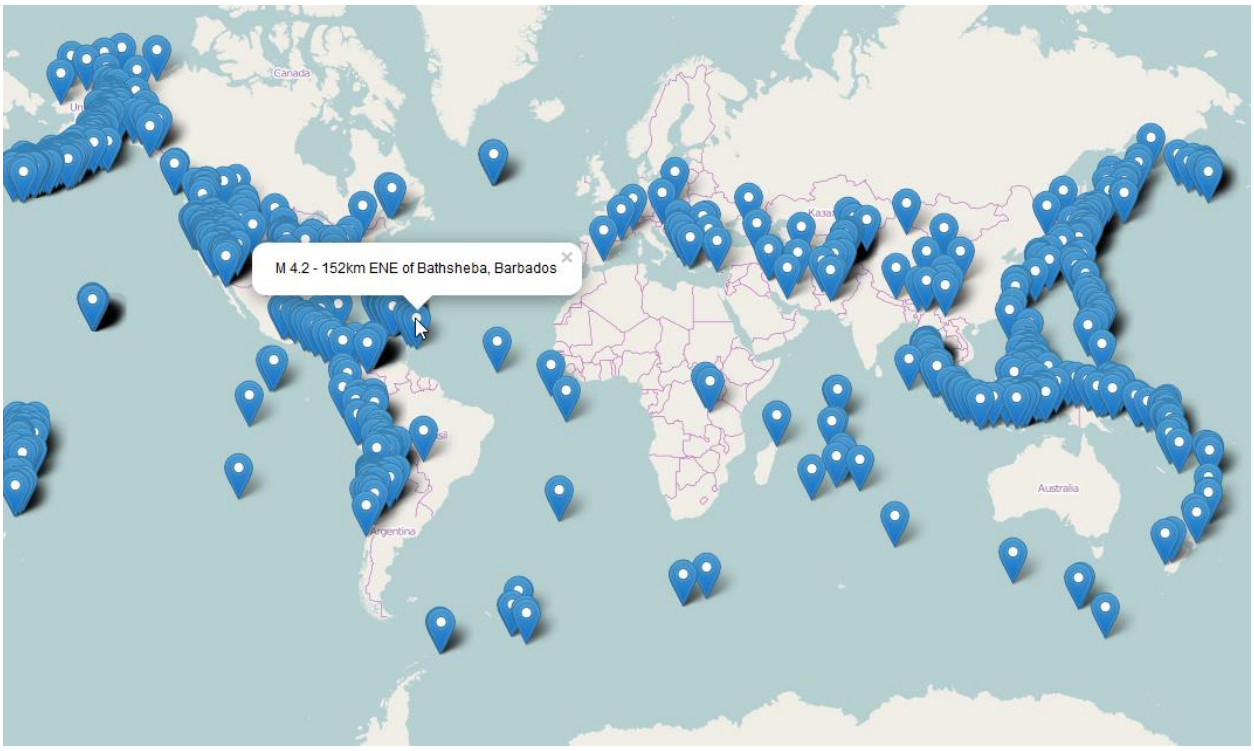


Figure 50: USGS Earthquake KML data visualized in Leaflet.js

In this chapter, we've seen how to add data to Leaflet maps from SQL databases and third-party providers. We've also seen how Leaflet plugins can greatly extend the types of data you can add to your maps, while keeping the core Leaflet code base small.

Chapter 6 Geocoding

Often you want your users to be able to enter a location such as an address, landmark, or national park, and have your map instantly zoom to the correct location. GIS people have a term for that: geocoding. Leaflet.js, in case you haven't already guessed, has a plugin for it. In fact, it has several. But the one I like, and which we're going to look at in this chapter, is **`L.GeoSearch`**.

The accuracy of any geocoding operation is dependent entirely on the quality of the data available, and one of the reasons I really like **`L.GeoSearch`** is that it gives us the ability to geocode against the data of several different providers: Google, Esri, Bing, Nokia, and OpenStreetmap. The other reason is that, regardless of which provider you decide to use, it is really easy to implement, and even provides its own perfectly adequate user interface control. Let's go ahead and use it.

First, visit the **`L.GeoSearch`** page on GitHub at: https://github.com/smeijer/L.GeoSearch and download the zip file:

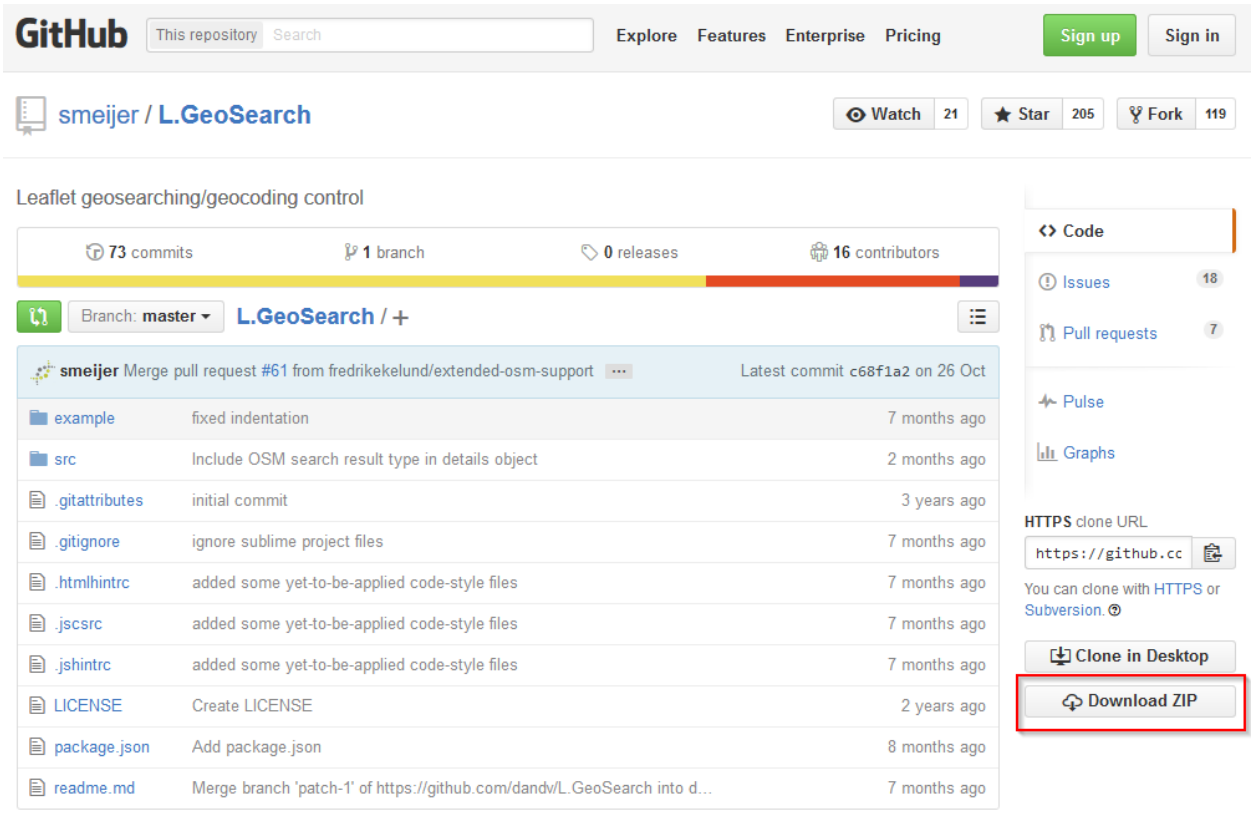


Figure 51: The L.GeoSearch page on GitHub

Like most Leaflet.js plugins, L.GeoSearch consists of some JavaScript files and some CSS to style the control, so you can ignore most of the contents of the zip file. If you want to follow along with this example, just copy the **`.js`** files from the **`src\js`** folder in the zip file to a directory called **`js`** within the same directory that you're going to write your code in. In my environment, I'm writing my code in **`Ch06_geocoding`**, which is a subdirectory of **`leafletsync`** on my local web server:

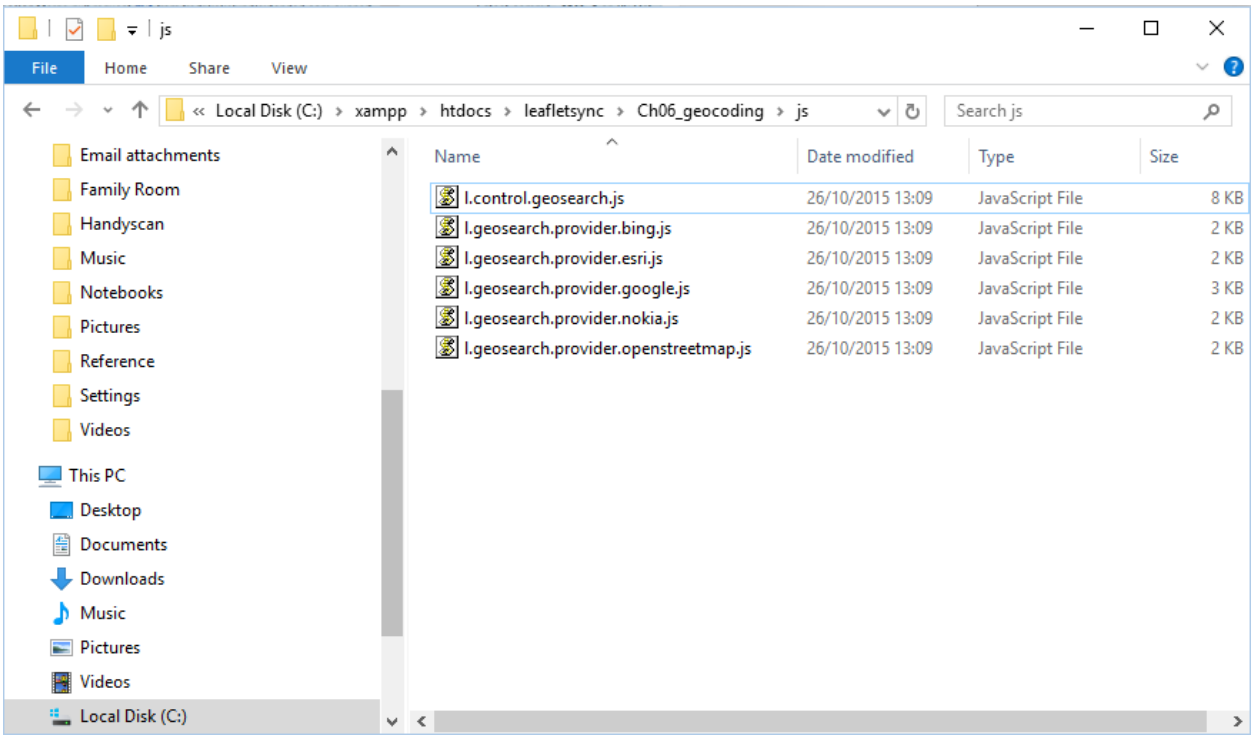


Figure 52: Placement of the L.GeoSearch JavaScript files

Then copy the **`l.geosearch.css`** file into a directory called **`css`** within the same directory that you are going to write your code:

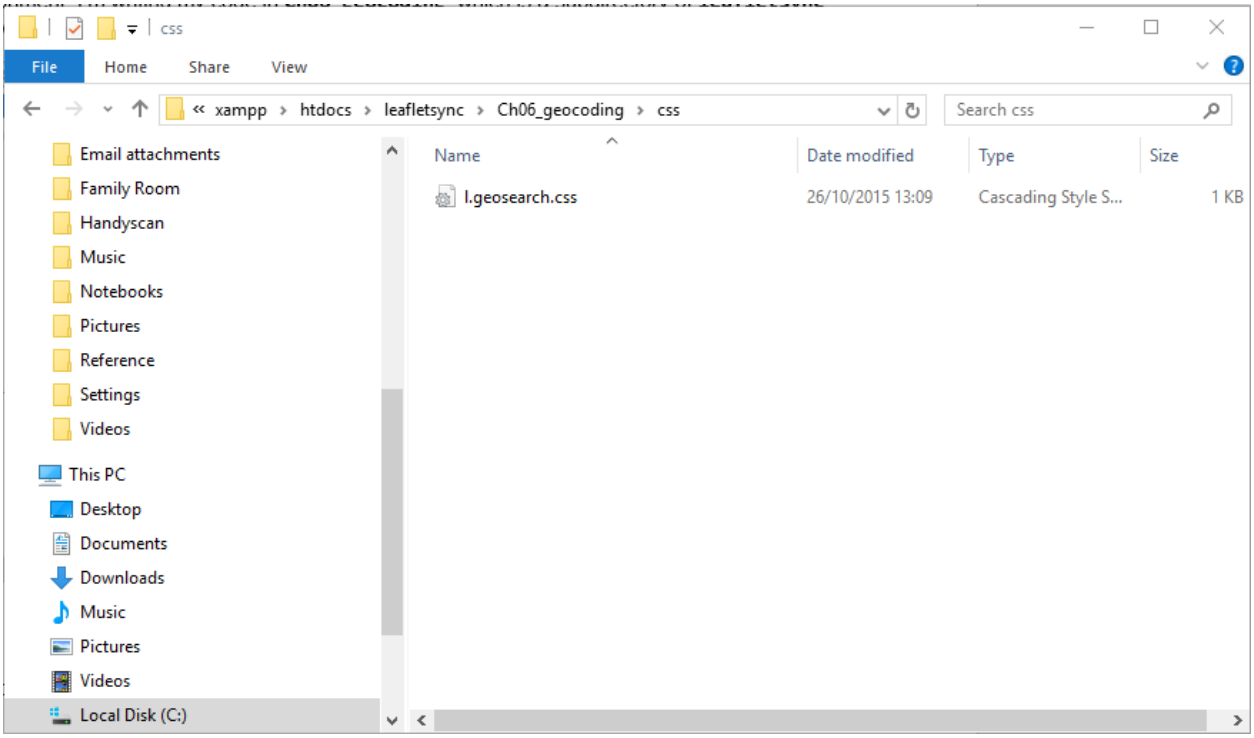


Figure 53: Placement of the L.GeoSeach CSS file

We now need to create our HTML page, being sure to reference the L.GeoSearch files we added previously. In this example, I'm going to use OpenStreetmap as my provider, so I need a **`<script>`** tag reference to both the **`l.control.js`** and **`l.geosearch.provider.openstreetmap.js`** files.

If I wanted to swap providers at any stage, or even allow my users to choose the provider at runtime, then I'd need to reference the relevant **`l.geosearch.provider.[name].js`** file(s).

I also need to reference the **`l.geosearch.css`** file in a **`<link>`** ref, which will be used to style the control.

After creating a map, setting its extent to the U.S., and adding an **`OpenStreetmap`** basemap layer, this is our starting point:

Code Listing 48: Starting point for the Geocoding application

```
<!DOCTYPE html>
<html>

<head>
<title>My Leaflet.js Map</title>
<link rel="stylesheet"
href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css" />
<script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></script>
<script src='js/leaflet-omnivore.min.js'></script>
</script>
```

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

<script type="text/javascript">
    function init() {
        var map = L.map('map').setView([0,0], 2);

        // OSM Mapnik
        var osmLink = "<a href='http://www.openstreetmap.org'>Open StreetMap</a>";
        L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
                attribution: '&copy; ' + osmLink,
                maxZoom: 18
            }).addTo(map);


        var runLayer = omnivore.kml('kml/2.5_month_age.kml')
            .on('ready', function() {
                map.fitBounds(runLayer.getBounds());

                runLayer.eachLayer(function(layer) {
                    layer.bindPopup(layer.feature.properties.name);
                });
            }).addTo(map);
    }
</script>
</head>

<body onload="init()">
    <div id="map"></div>
</body>

</html>
```

Note: The choice of basemap layer is irrelevant. Just because you are using OpenStreetmap for your geocoding operations, you are not restricted to using OpenStreetmap tiles in your map.

Now we just need to create a new instance of the class for our chosen provider:

Code Listing 49: Creating a new instance of the appropriate provider's L.GeoSearch class

```
...
    new L.Control.GeoSearch({
        provider: new L.GeoSearch.Provider.OpenStreetMap()
    }).addTo(map);
...
```

And that's it! Now, when we run the application, we get a nice text entry field at the top of the page. We can enter the name of a place, or part of an address and, depending on the quality of the provider's data, get a more or less useful result.

Let's try searching for Google's headquarters in Mountain View, California:

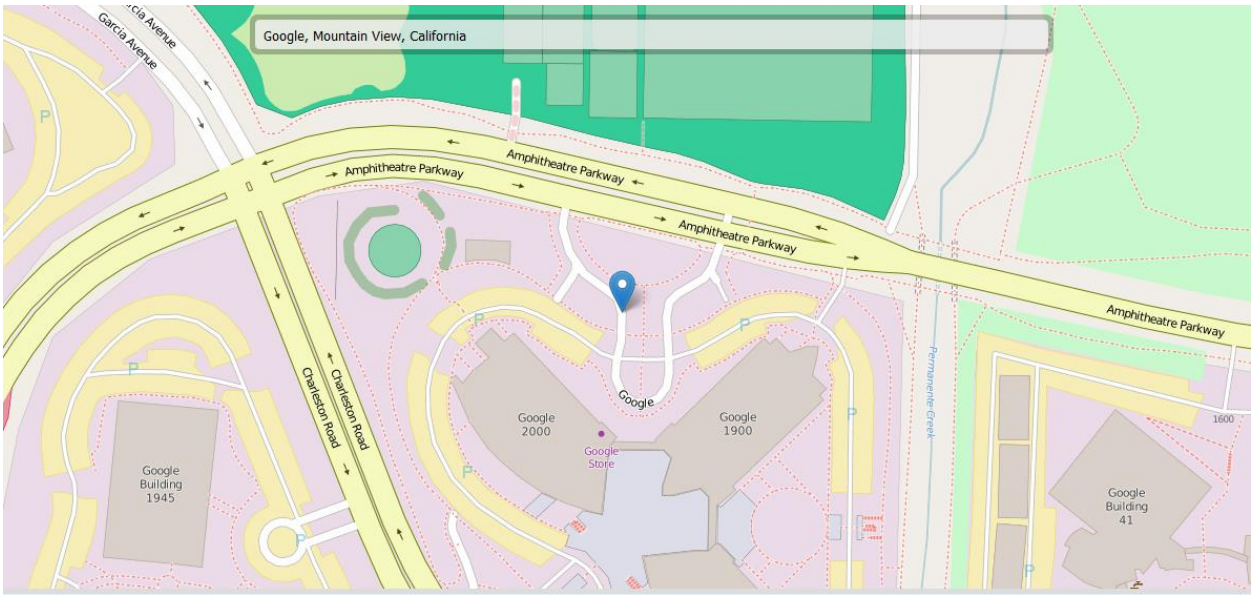


Figure 54: Searching for Google HQ

There we are, right in the middle of the Googleplex. Note how **`L.GeoSearch`** instantly positions the map to the location of the search results, zooms in, and displays a marker based on the coordinates it has stored for the address.

Let's see if it works on my address (no hate mail, please):

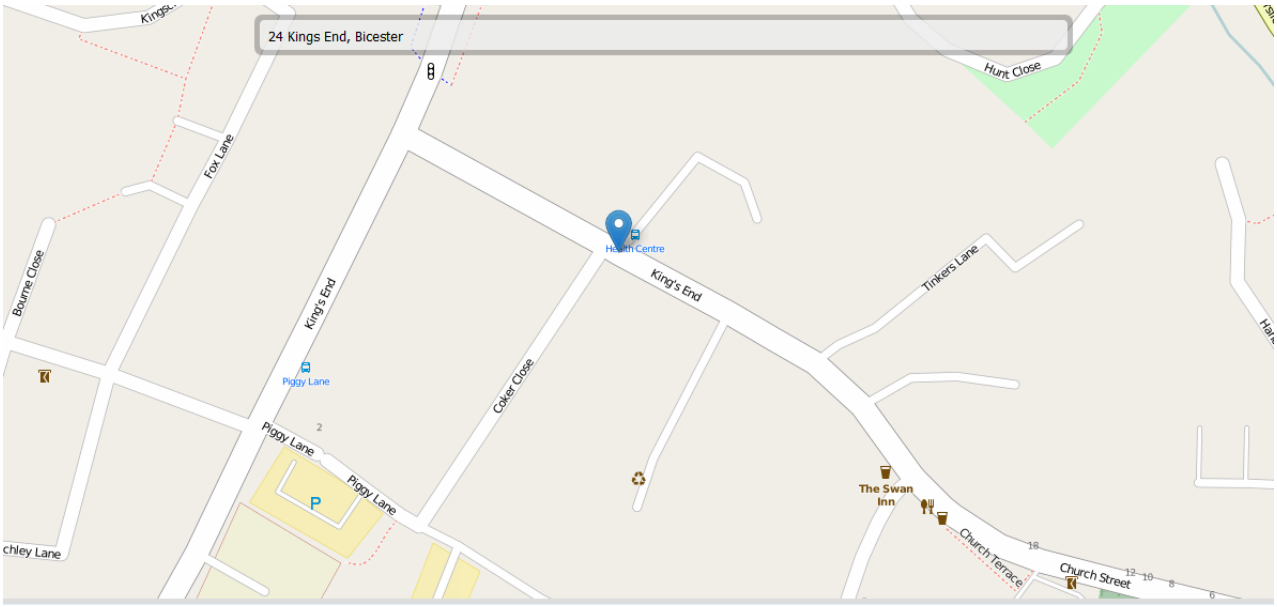


Figure 55: Searching for my home address

That's pretty poor, actually. Google's geocoding result puts me about 200 feet away from where I actually live. Let's swap providers and give Esri a chance to beat Google. I just need to change the file I'm referencing in my **<script>** tag and the name of the class I'm using in the **provider** property of the **L.GeoSearch** control:

Code Listing 50: Using Esri as a provider

```
<!DOCTYPE html>
<html>
<head>
  <title>My Leaflet.js Map</title>
    <link rel="stylesheet"
      href="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.css"
/>
    <link rel="stylesheet" href="css/l.geosearch.css" />
    <script
src="http://cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.3/leaflet.js"></script>
    <script src="js/l.control.geosearch.js"></script>
    <script src="js/l.geosearch.provider.esri.js"></script>
    <style>
      html, body, #map {
        height: 100%;
      }
    </style>
    <script type="text/javascript">
        function init() {
            var osmLink = "<a href='http://www.openstreetmap.org'>Open
StreetMap</a>"
            var map = L.map('map').setView([34.525, -97.778],5);
            L.tileLayer(
```

```
                'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
                attribution: 'Map data &copy; ' + osmLink,
                maxZoom: 18,
            }).addTo(map);

            new L.Control.GeoSearch({
                provider: new L.GeoSearch.Provider.Esri()
            }).addTo(map);
        }
    </script>
</head>
<body onload=init()>
      <div id="map"></div>
</body>
</html>
```

This is what Esri came back with:

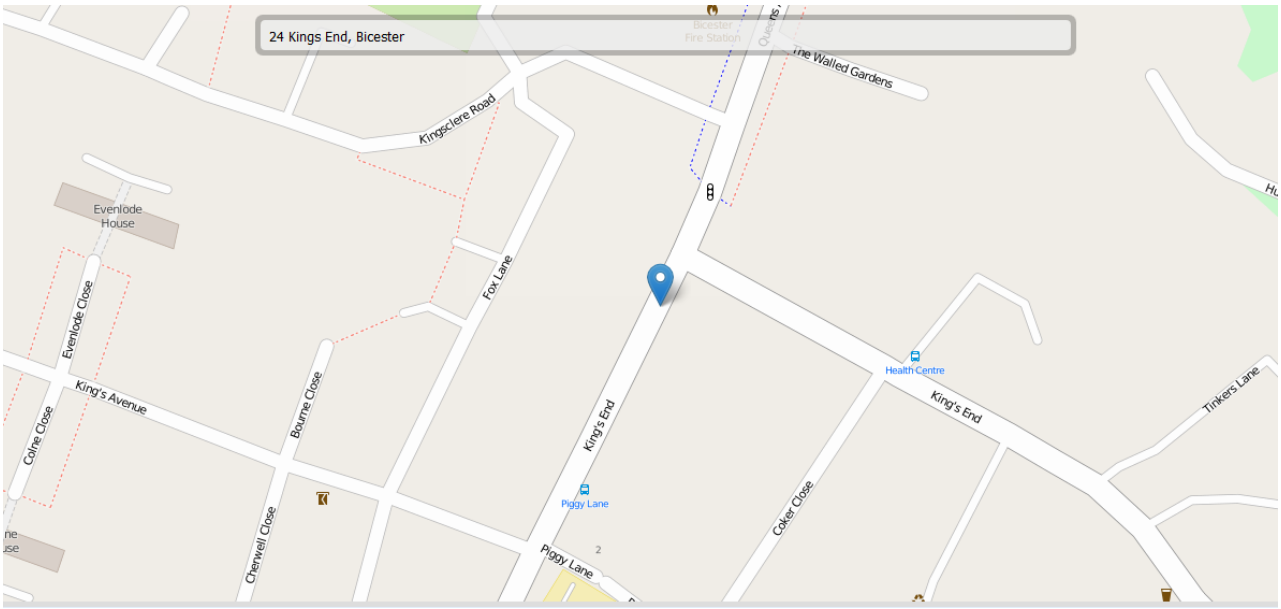


Figure 56: Esri's geocoding result on my address

Much better! (Even though I don't live in the middle of the road.) The moral of this story is that all geosearch providers are not created equal, especially if you are searching for addresses outside of the USA.

Chapter 7 Conclusion

So now that you've had a chance to play with Leaflet.js, I hope you're as excited about its possibilities as I am. I can say that it is, hands down, the nicest web mapping API I have ever used, and I've tried most of them.

By keeping the core of Leaflet.js small and extending its functionality via plugins, the API is a pleasure to work with, and without the bloat that mars the efforts of some of its competitors. Things just work the way you expect them to, which is unusual in this space. A lot of mapping APIs are created by geographers, and they tend to want features and terminology in their APIs that don't mean a great deal to ordinary folk. Leaflet's creator, Vladimir Agafonkin, is first and foremost a web developer—and it shows, because he has built an API that other web developers can relate to.

In this book, we covered most of the core API, and even had a look at some plugins. If you want to greatly extend the functionality of your web mapping applications, then you should have a look at the sheer range of plugins out there. The official plugin page on Leaflet.js lists plenty, covering just about every sort of functionality you could even imagine.

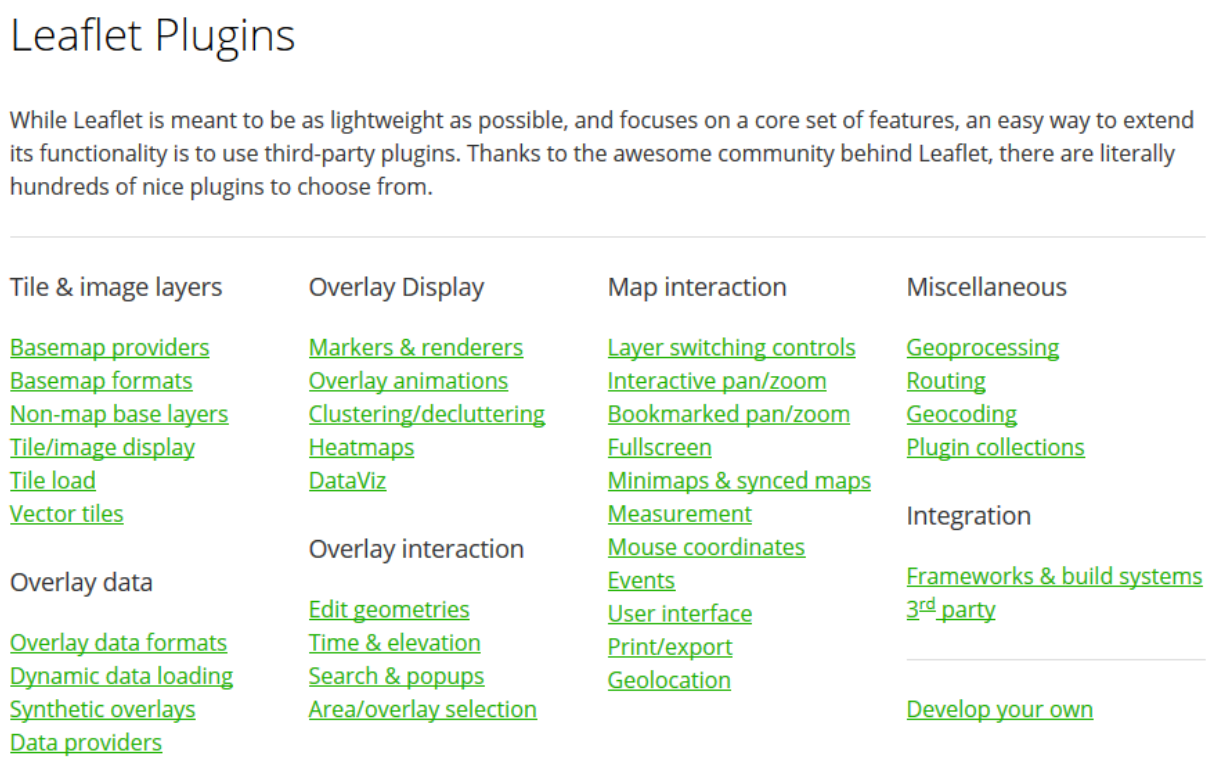


Figure 57: The Leaflet.js plugins page

There are plugins that enable animations, bookmarking of locations, route navigation, and more.

For example, if you want to create heat maps where areas of high point density are shown as “hot” and areas low point density are shown as “cold,” then you should consider the heatmap.js plugin by Patrick Wied:

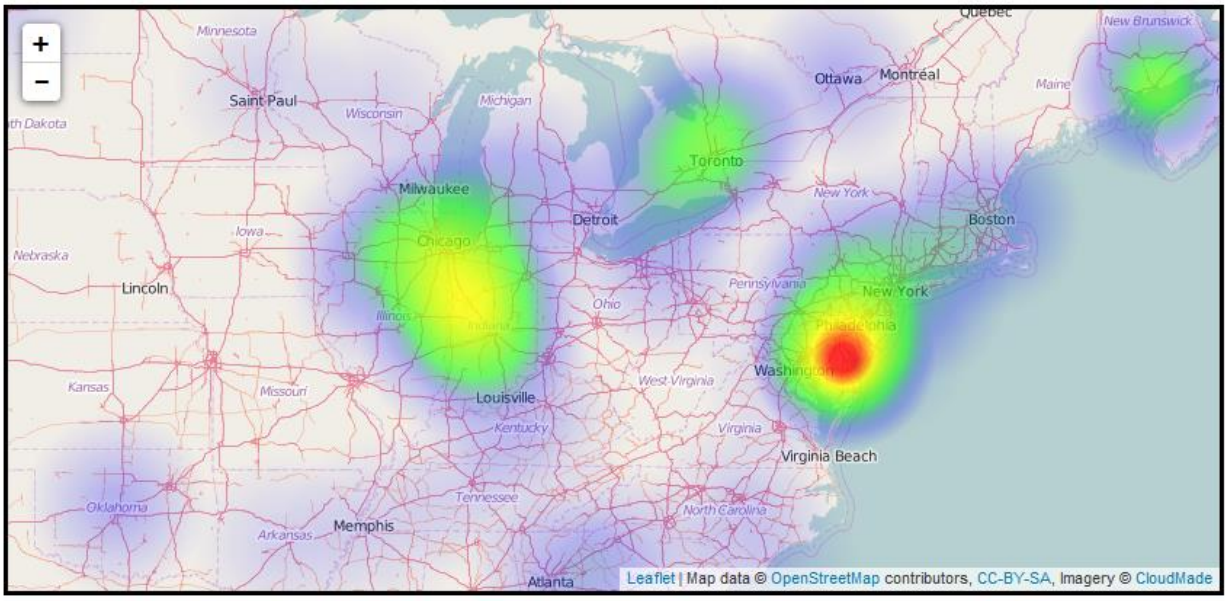


Figure 58: The heatmap.js plugin by Patrick Wied

If you want to use the D3.js to create dynamic, interactive data visualizations in conjunction with Leaflet.js, you should have a look at Kirill Zhuravlev’s Leaflet.D3SvgOverlay plugin, which allows you to combine the power of D3.js with the interactivity of SVG:

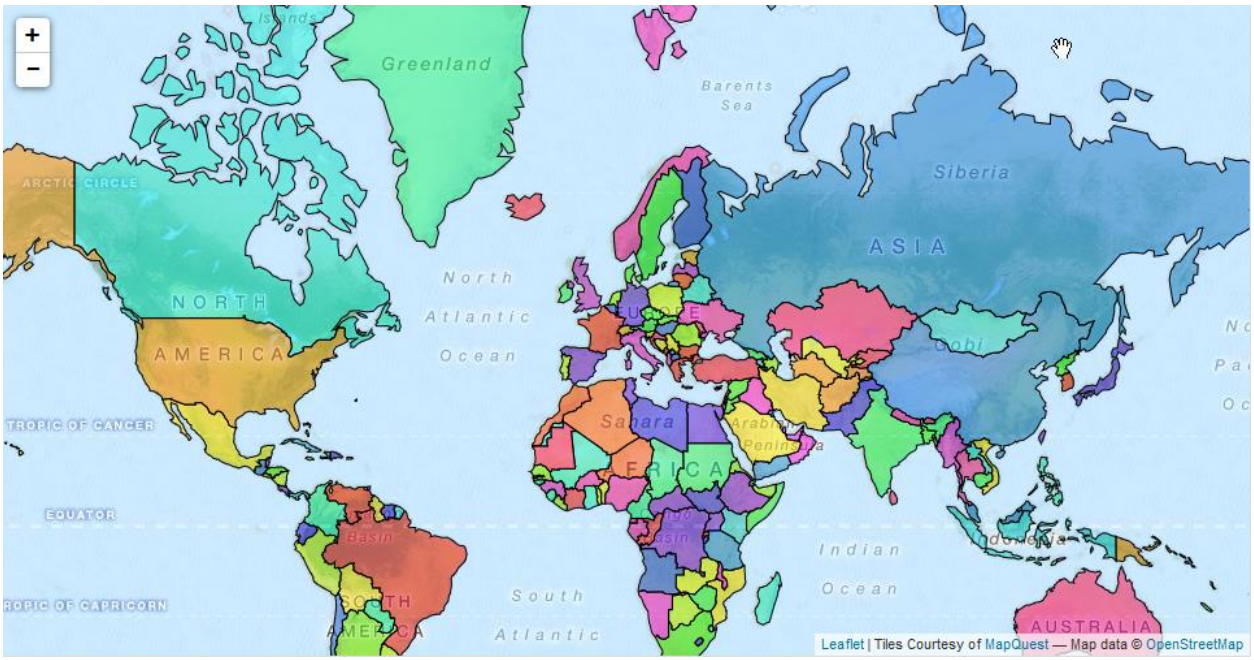


Figure 59: The Leaflet.D3SvgOverlay plugin by Kirill Zhuravlev

If you want to visualize temporal data in Leaflet.js and allow your users to use a time slider to alter the period for which they want to see data, try the **`Leaflet.TimeDimension`** plugin by SOCIB:

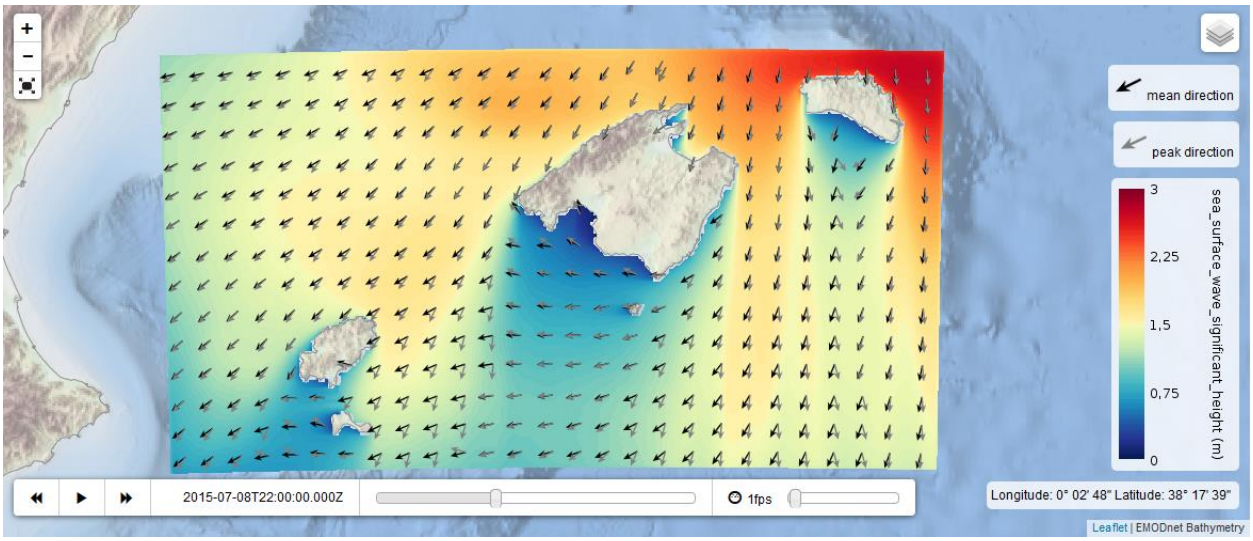


Figure 60: Visualizing temporal data with Leaflet.TimeDimension

Or if you're all about usability, check out the handy little overview map (**`Leaflet.miniMap`**) provided by Robert Nordan. It provides a control that allows you to see the larger extent to which the current map view belongs, and even navigate directly from it:

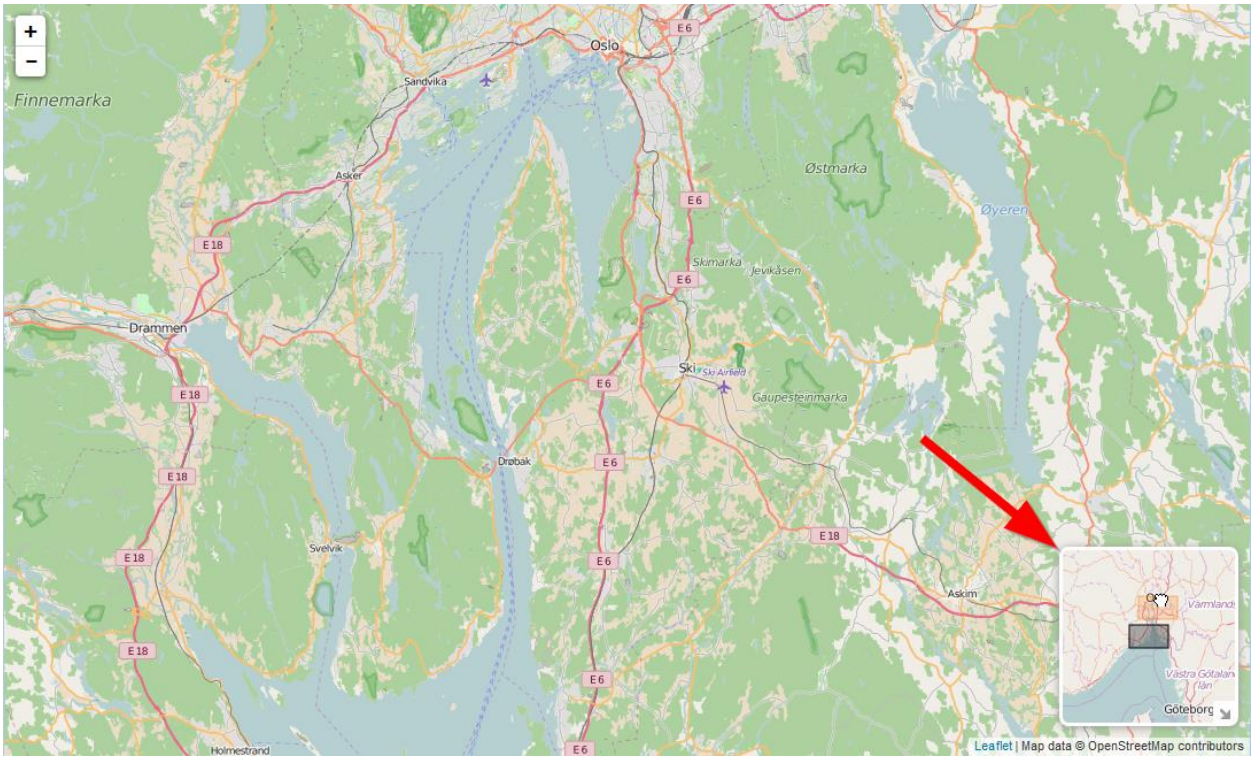


Figure 61: Robert Nordan's Leaflet.MiniMap

If there's some functionality you want that doesn't appear to be served by one of Leaflet's official plugins, then a search on GitHub will usually give you an amazing array of possibilities. Be warned, though: the quality through unofficial channels is very variable:

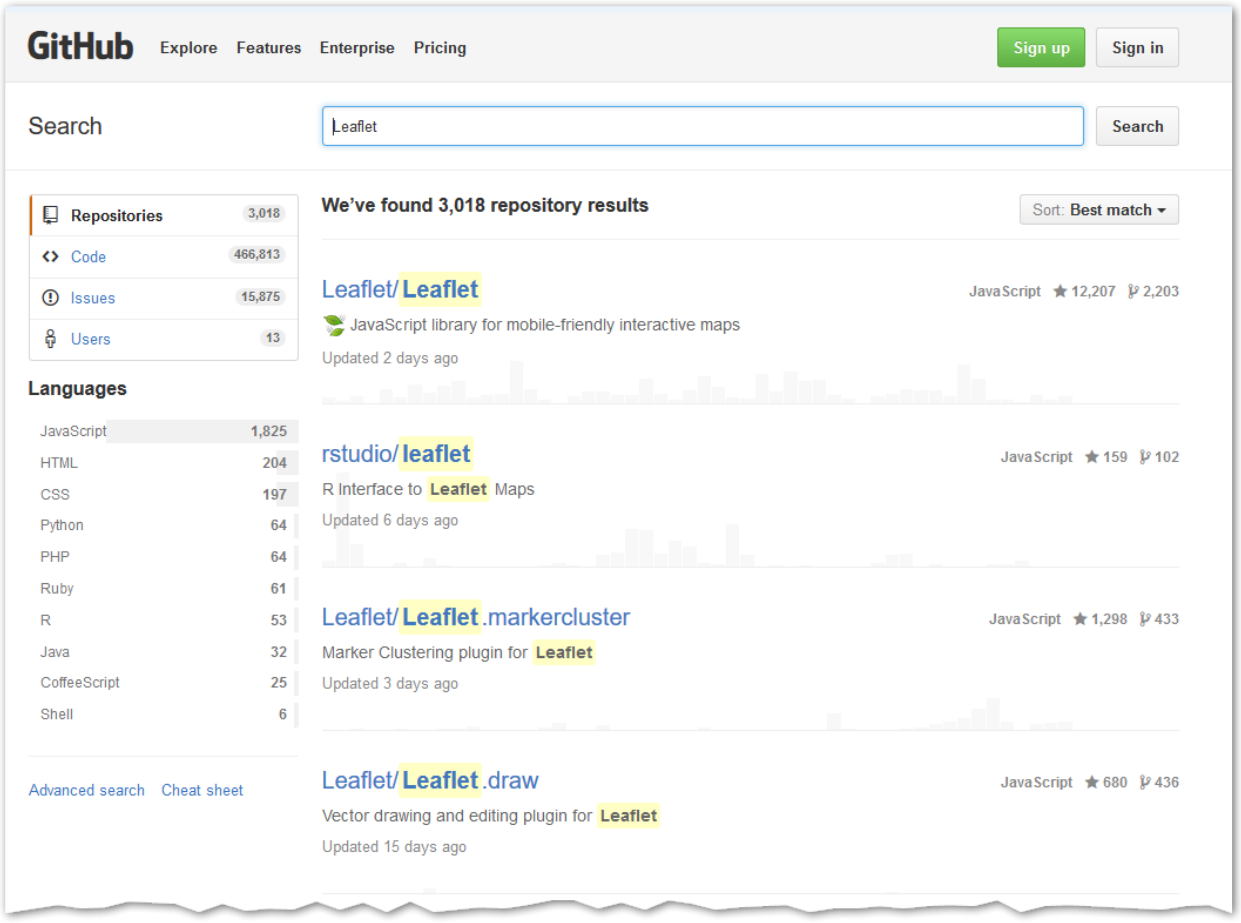


Figure 62: A huge array of plugins can be found on GitHub, as well as the Leaflet.js code itself

Finally, if there's nothing there that fits the bill, why not create your own plugin and share it with the Leaflet.js community? Leaflet maintains a very comprehensive document on GitHub called the Leaflet Plugin Authoring Guide.

Figure 63: The Leaflet Plugin Authoring Guide

Contacting the Author

I hope you found this Syncfusion Succinctly guide useful. Feel free to stay in touch with me, Mark Lewin, on my blog or via @gisapps on Twitter. Thanks for your time!