GOOGLE MAPS API

SUCCINCTLY

BY MARK LEWIN

SUCCINCTLY E-BOOK SERIES

Syncfusion

www.dbooks.org

Google Maps API Succinctly

By
Mark Lewin

Foreword by Daniel Jebaraj



Copyright © 2018 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

www.dbooks.org

Table of Contents

The Story Behind the *Succinctly* Series of Books **7**

About the Author **9**

Chapter 1 Introduction **10**

Chapter 2 Getting Started **11**

API keys 11

Getting an API key 11

Creating your first map 14

Creating the webpage 14

Writing the map initialization function 17

Setting the extent 18

Trying different base maps 20

Chapter 3 Customizing the Map Controls **24**

The default map controls 24

Hiding the default controls 25

Adding a scale bar 27

Changing control display and positioning 27

Creating custom controls 29

Chapter 4 Adding Data to the Map **32**

Adding markers 32

Labelling the marker 34

Changing the marker icon 34

Animating the marker 35

Moving the marker 36

Info windows 37
Adding shapes 40
The Overlay Creator tool....40
Polylines 42
Polygons....44
Circles 46
Rectangles....47
Editing and moving shapes....48
Ground overlays....49
KML and GeoRSS 50
Getting your users to do the work....56
A word about libraries 57
Adding the Drawing Manager....58
Chapter 5 Working with Events....62
Event-handling basics 62
Displaying mouse-click coordinates....63
Syncing two maps 65
Displaying the maps 65
Handling the map events 69
Bi-directional sync....70
Events on non-API objects 70
Cancelling edits 72
Chapter 6 Geocoding....77
Using the Geocoder 77
The GeocoderRequest class 78
The GeocoderResult object 78

www.dbooks.org

The GeocoderStatus constants....79
Geocoding an address....80
Reverse geocoding....86
Chapter 7 Getting Directions....90
Enabling the Directions API....90
Creating the page....92
Geocoding the start and end points....95
Calculating the route....99
Chapter 8 Conclusion....104

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.

www.dbooks.org

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 more than 18 years. His main interests are web development in general, and web mapping in particular. While working for ESRI, the world's largest GIS company, he acted as a consultant, trainer, and course author. He has been a frequent speaker at industry events, and he currently works with a wide variety of open-source mapping technologies and a handful of relevant JavaScript frameworks, including Node.js, Dojo, and JQuery.

Based in the UK, Mark currently teaches ArcGIS Server/JavaScript development for Geospatial Training LLC and is the author of the *Google Maps API: Get Started* course for Pluralsight. By day, he writes MySQL courseware for Oracle.

Mark can be reached at mark@marklewin.com or on Twitter at @gisapps.

This is Mark's fourth e-book for Syncfusion. The first, *Leaflet.js Succinctly*, was published in March 2016, and two e-books on the Go programming language were published in 2017.

Chapter 1 Introduction

Welcome to *Google Maps API Succinctly*! In this book, I aim to give you a hands-on, beginner-level introduction to developing mapping applications using the Google Maps JavaScript API.

Unlike with many beginner-level technical books I have written, I can be confident that you know what Google Maps is, and you might even be using it on a daily basis. As well as accessing Google Maps on Google's own site, you will doubtless have also stumbled across other sites that have Google's maps embedded within them. These might have been as simple as a website for a retail business where the purpose of the map is to show the locations of each of their stores, or more sophisticated and useful (?) examples, such as the now-infamous Zombie Outbreak Simulator (see *Figure 1*).

These embedded maps have all been created using Google's APIs, which provide not only access to Google's extensive map data, but also a simple way of visualizing and otherwise working with that data.

Maps are fun. Putting maps on the web is even more fun. But creating web maps where there is so much freely available and useful data to play with is just awesome. This book will teach you the ins and outs of the Google Maps JavaScript API so that you can harness the power of Google Maps within your own websites and applications.

Like all *Succinctly* books, this book is light on theory and heavy on the action! To get the most out of it, you'll need to complete all the coding activities rather than just read about them. All you need—apart from a basic knowledge of HTML, CSS, and JavaScript and the willingness to follow some simple instructions—is an up-to-date browser (I recommend either Firefox or Chrome) and a text editor.

Tip: If you get stuck, you can download the solution code from GitHub. There, you'll also find the image files and other resources referred to in this book.



Figure 1: Maps and zombies. What's not to like?

Chapter 2 Getting Started

API keys

Before you can embed a Google map in your page, you need to request an API key. An API key is just a string of characters you need to embed in your code to use Google Maps in your own webpages. This enables Google to keep an eye on how many map requests you are serving and, if your application becomes super popular, to start charging you for access.

Google has a couple of different pricing plans, but you'll want to start off on their Standard plan, which is free until you incur over 25,000 map loads over a period of 24 hours. You are also limited as to the number of free requests for services such as Directions and Geocoding, which we'll talk about later in this book. After that, you'll need to start paying Google for the privilege.

Rest assured that 25,000 map loads and 2,500 service requests is very generous, and you're unlikely to hit that quota until your app becomes quite popular (and is probably earning you money). If you're just following along with the examples in this book, those limits are not going to be a problem for you.

After that, you can either "pay as you go" or join their Premium plan, which enables you to purchase credits in advance and unlocks other benefits, such as support, a service-level agreement, and so on. You can find out more about their pricing options here.

Getting an API key

To get an API key, you must first have a Google account. Then:

1. Visit http://console.developers.google.com.
2. Sign in with your Google account user name and password.
3. In the dashboard, under the list of APIs (there are loads), locate the library page on the left-hand side.

www.dbooks.org

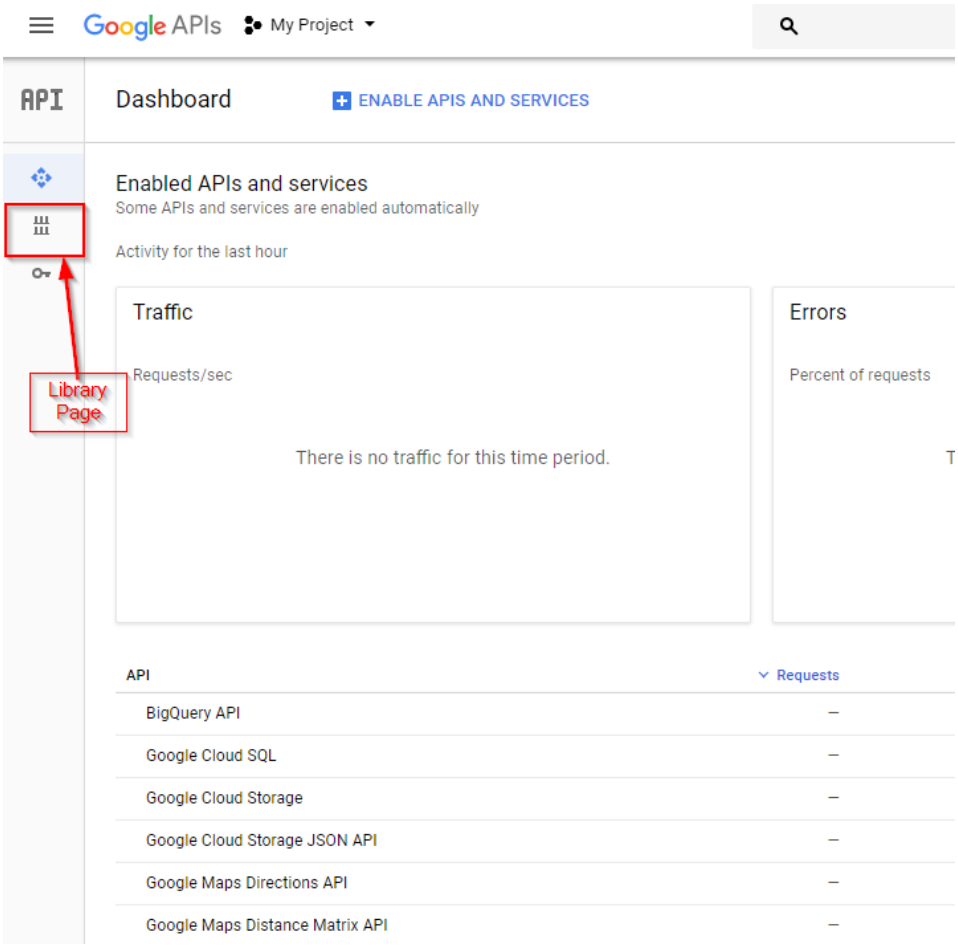


Figure 2: The Developer Console.

4. Locate the **Maps** section of the page and click **Google Maps JavaScript API**. Or use the search dialog.
5. When the summary page for the API displays, click **Enable**.

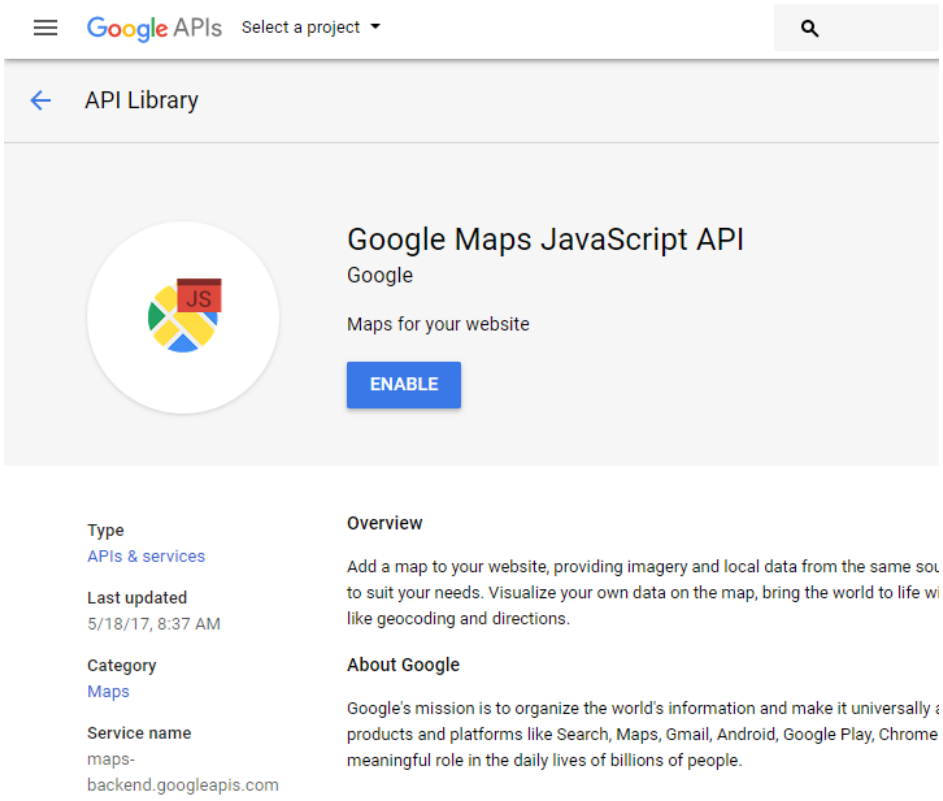


Figure 3: The Google Maps JavaScript API summary page.

6. A dialog box appears, asking you to select a project. You don't have one yet, so go ahead and click the blue button to create one. I'm going to call mine **gmaps-succinctly**. After you have created the project, the summary page appears again. But this time, the name of the project that you created in the preceding step is shown at the top of the page.
7. Click the blue **Enable** button again. You are redirected to the dashboard page, where you will be able to view usage information once your app is out there. Right now, you're probably being prompted to create credentials. If so, click **Create Credentials**.
8. In the Credentials wizard, ensure that the **Google Maps JavaScript API** is selected in the drop-down list under the **Which API are you using?** header, and then click **Which Credentials Do I Need?**.
9. The Credentials Wizard presents you with an API key—a string of alphanumeric characters (such as “AIzaTyD-2vJ3p_IpqyWPYmXtkFZwbI0juvKhBg4”) that you must embed in your code to use the API. Make a note of it somewhere, because you'll need it to follow along with the code examples in this book. Note the warning about restricting your key before using it in production. We won't do that in this book, but it is certainly something to keep in mind.

www.dbooks.org

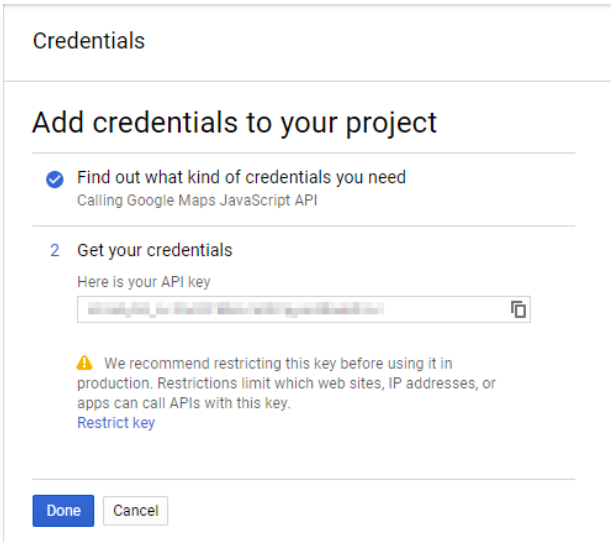


Figure 4: Your API key.

10. Once you have made a note of your API key, click **Done**. If you ever lose the key, you can find it in the Credentials area of the Developer Console.

Creating your first map

Now that you have your API key, you are ready to build your first Google Maps JavaScript API application. Your application will be hosted in a webpage that contains the following elements:

- A **`<script>`** tag that loads the API itself using your API key.
- A **`<div>`** that acts as a placeholder for the map.
- Another **`<script>`** tag that contains some JavaScript code that you write yourself. This executes when the page loads to create the map and sets some initial options for its display.
- Some CSS to make sure all this stuff appears correctly.

Creating the webpage

Create a new HTML page with the code in Code Listing 1 and save it as Ch2_FirstMap.html in a directory that's associated with your web server (such as C:\inetpub\wwwroot\Gmaps on a Windows machine). This is “boilerplate” code that you will use as a starting point for many of the examples in this book, so you might want to create a copy of it somewhere.

Code Listing 1: Initial page HTML

```
<!DOCTYPE html>
<html>

<head>
    <title>Ch2 First Map</title>
```

```
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        #map {
            height: 100%;
        }

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

<body>
    <div id="map"></div>
    <script>
        // The code to create the map goes here.
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=
                YOUR API KEY&callback=initMap" async defer></script>
</body>

</html>
```

Most of this is standard webpage stuff, so we'll just cover the interesting bits.

The first thing to note is the ** ** with an **id** of **map**. When we create a map, the API needs to know where in the page to put it, and that location needs to be a ** ** element.

The first **<script>** block is where we will write the JavaScript that will interact with the Google Maps API to create the map. We haven't written this code yet.

The second **<script>** block contains the bootstrap URL, which downloads the Google Maps API code to the browser. For this to work, you need to replace **YOUR API KEY** with your API key (including the square brackets).

To wire everything up, we use the **callback** query parameter within the Google Maps API bootstrap URL to specify the name of our JavaScript function that we want to execute when the API has loaded (we haven't written this yet, but we'll call it **initMap()**). We also add a couple of attributes to our **<script>** tag that specify *how* the API should be loaded.

The **async** attribute tells the browser to load the script behind the scenes (asynchronously), while the page itself is loading. This means that two things happen at once, resulting in better performance for our users. When the script has loaded, it is executed immediately.

www.dbooks.org

The **defer** attribute is similar, but is put there as a safeguard for older browsers that might not recognize **async**. If **async** is not supported, then **defer** doesn't execute the script until the page is loaded.

The point of doing all this is to allow the Maps API code to run the map initialization code as soon as it can, resulting in a better experience for your users.

Finally, we have a **<style>** tag where we provide a height for the map ** **, which in this case is 100 percent of the page height, thereby occupying the entire page.



Tip: If your map does not display and there are no errors on your page, ensure that you have specified a `height` for the `<div>` element that hosts the map. If you don't specify a `height`, the map will not appear.

Let's make sure the page is coded correctly by adding the **initMap()** function to the script block in which we'll be writing our Google Maps API code. Add the code in Code Listing 2 within the first **<script>** block.

Code Listing 2: Testing the callback function

```
<body>
    <div id="map"></div>
    <script>
        // The code to create the map goes here.
        function initMap() {
            alert("I'm ready to make a map!");
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=
                    [YOUR API KEY]&callback=initMap" async defer></script>
</body>
```

Make sure that your web server is running, and then open the page in your browser. If everything is working correctly, the **initMap()** function should execute when the API has finished downloading, and you should see the message in Figure 5.

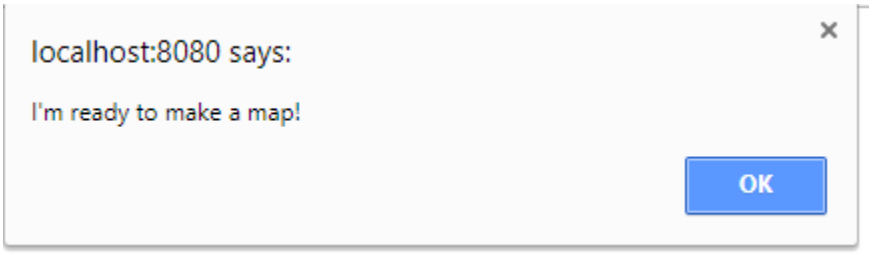


*Figure 5: The **initMap()** function is working correctly.*



Note: Most of the samples shown in this book will work just by opening the webpage itself, without hosting it on a web server. However, it is good practice to do so. If you don't yet have a web server, consider installing XAMPP, which includes the popular Apache web server. See this link to install XAMPP.

Writing the map initialization function

Now that we have access to the API and a function on our page that will be called when the API has loaded, it's time to create the map.

To create a map, we need to instantiate a new object of the **`google.maps.Map`** class, telling it *where* on our page it should appear, and providing it with some options that tell it *how* to display. We accomplish both of those tasks by passing parameters to the **`google.maps.Map`** constructor.

Let's have a look at the code in Code Listing 3 first, and then we'll talk about what it's doing.

Code Listing 3: Instantiating the google.maps.Map class

```
<div id="map"></div>
<script>
    // The code to create the map goes here.
    function initMap() {
        var map;
        map = new google.maps.Map(document.getElementById('map'), {
            center: {
                lat: 51.503,
                lng: -0.135
            },
            zoom: 12
        });
    }
</script>
```

The first thing we did is declare a variable called **`map`** to store a reference to the new **`google.maps.Map`** object. We then created the **`google.maps.Map`** object, passing in two parameters.

The first parameter uses plain JavaScript to access the DOM element on our page named **`map`**, which is the **`<div>`** where we want the map to display.

The second is a map options object literal that contains the **`center`** and **`zoom`** properties for the map. There are many different properties that you can use to modify the way the map looks and works, and we'll consider many of them in this book. For our first map, however, we are only interested in two of those properties. The first is **`center`**, which is the geographic coordinate at

www.dbooks.org

the center of the map area that we want to show our users. The second is **`zoom`**, which represents the map scale at discrete intervals.

The **`center`** property's value is itself an object that has two properties, **`lat`** and **`lng`**, that represent the latitude and longitude of the map's center point, respectively. The **`zoom`** property accepts an integer value that can be between zero (for "zoomed all the way out") and a maximum value that depends on what imagery is available for the geographic area you are showing, and which represents "zoomed all the way in." Most roadmap imagery is between 0 and 18, for example, but the availability of satellite imagery varies widely. In this example, the **`zoom`** property is set to **`12`**.

If you open the page in your browser, you'll see a map that occupies the entire page (as specified in the **`<style>`** attribute) and is focused on London, in the United Kingdom, as shown in Figure 6.

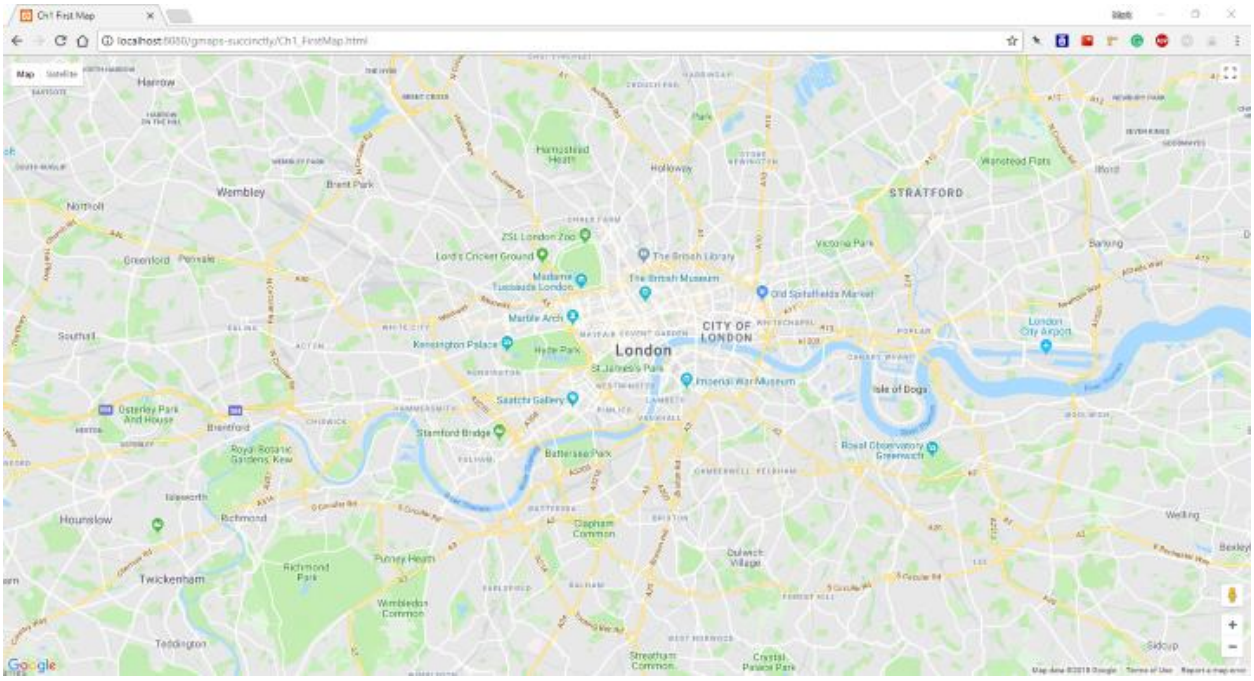


Figure 6: Your first map!

Congratulations! You just created your first map. Feel free to experiment with it, noting the default behavior: the availability of satellite imagery and street view, and the zoom in and out buttons in the bottom right-hand corner. We'll look at various ways of customizing this map as we progress through the book.

Setting the extent

The first customization you might want to make is changing the geographic extent of the map. To do that, you need to know the latitude and longitude of the map's center and the zoom level at which you want to display the area.

There are several websites that will help you determine what the latitude and longitude of a location is. The API itself provides a way to determine the zoom levels available for any location via the **`MaxZoomService`** (see this article for details).

However, I have an easier way for you to determine these details—one that will even create the code for you. If you follow this link, you'll find a little application of mine called Extent Finder. It enables you to pan and zoom the map around until you find the area you want to work with. You can then click **Create MapOptions for this Location** to access the Maps API code that you need to represent this area.

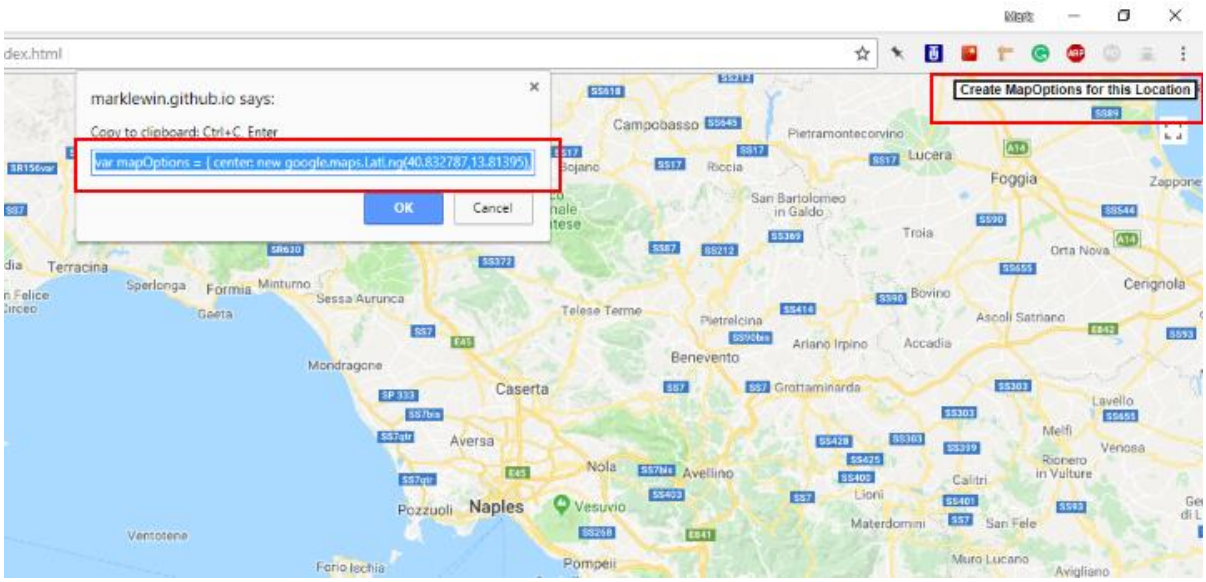


Figure 7: The Extent Finder application.

Press **Ctrl + C** to copy the highlighted code and paste it into your **`initMap()`** function just below the line of code that declares the map variable.

Now, you'll need to make a change to the way that you instantiate the map object for this to work. In our example, we used an object literal to define the map options object right within the call to the **`google.maps.Map`** class constructor. The code you generated in the Extent Finder application stores this options object within a variable called **`mapOptions`**, instead. And instead of using an object literal to specify the center coordinates, it creates an instance of the **`google.maps.LatLng`** class.

Generally, it doesn't really matter how you do it, but I prefer this approach in anything other than trivial use cases because I believe it makes the code easier to read.

Change your code to use the new **`mapOptions`** object for your area of choice as follows, and then test it to verify that the area displays correctly.

Code Listing 4: Setting the custom map extent

```
<script>
    // The code to create the map goes here.
    function initMap() {
```


www.dbooks.org

```
        var mapOptions = { center: new google.maps.LatLng(40.832787,
13.81395), zoom: 9 };
        var map;
        map = new google.maps.Map(document.getElementById('map'),
                                  mapOptions);
    }
</script>
```

Trying different base maps

The Google Maps API gives you four different base map layers to choose from. They are **`roadmap`**, **`satellite`**, **`hybrid`** (a satellite map with labels), and **`terrain`**. If you don't specify one at startup, you get **`roadmap`** by default.

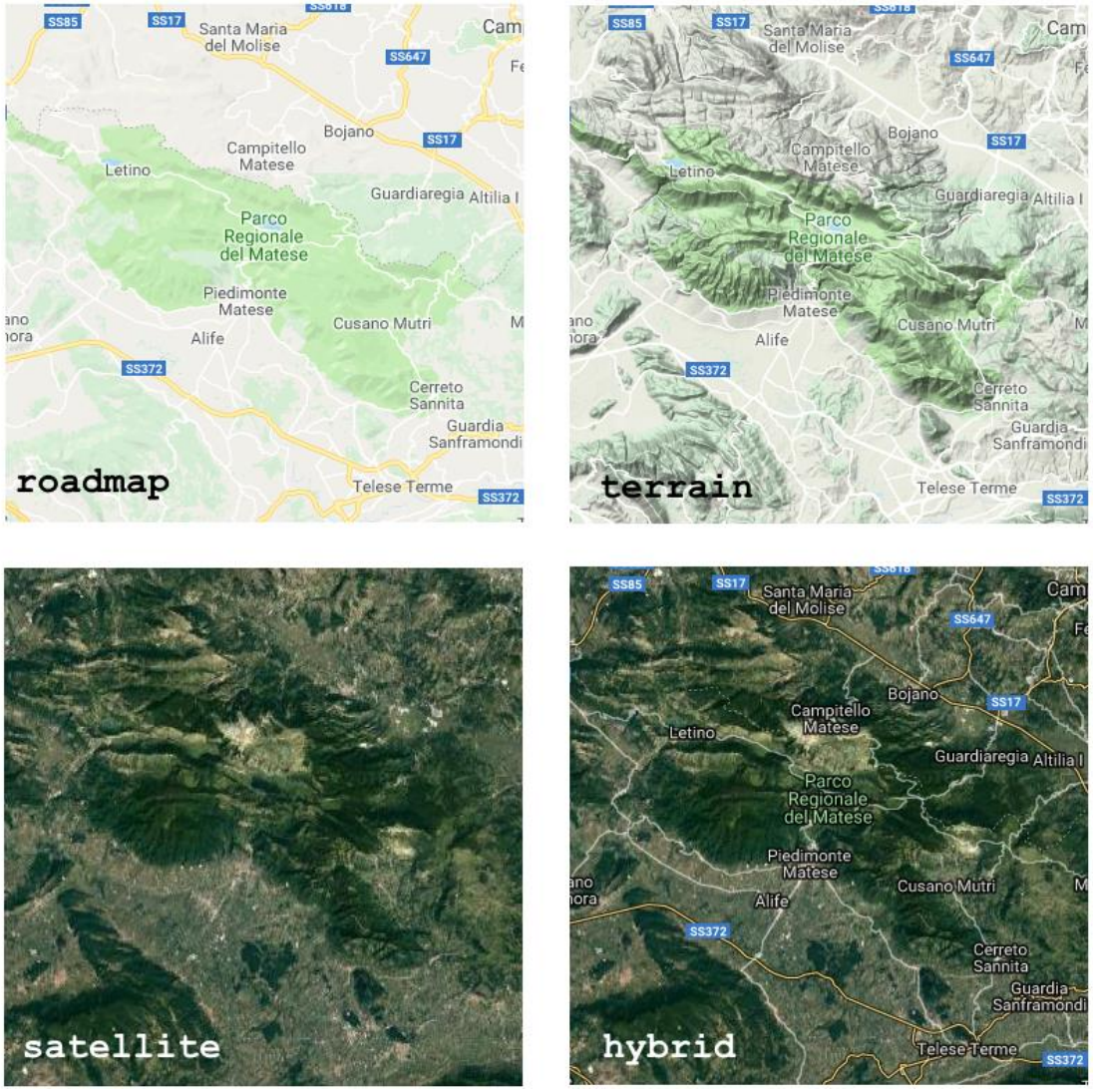


Figure 8: The available base map layers.

If you want to specify a different base map for your application, you can set the **mapTypeId** option in the **mapOptions** object. For example, to use the hybrid base map, change your code as follows:

```
var mapOptions = {
    center: new google.maps.LatLng(40.832787, 13.81395),
    zoom: 9,
    mapTypeId: 'hybrid'
};
```

You can also refer to a specific **mapTypeId** by the name of its constant, as follows:

```
var mapOptions = {
    center: new google.maps.LatLng(40.832787, 13.81395),
    zoom: 9,
    mapTypeId: google.maps.MapTypeId.HYBRID
};
```

However, by default, the map displays the **MapTypeControl**, which allows your user to select any base map they want. This control is shown in *Figure 9*.



Figure 9: The MapTypeControl.

Suppose that you didn't want users to have access to this control. Well, like all the controls in Google Maps, you can choose whether they are enabled or not. We'll talk more about controls in the next section, but for now, just be aware that the visibility of controls can also be specified in the **mapOptions** object. To disable **MapTypeControl**, just set the **MapTypeControl** property to **false** in the map options object:

```
var mapOptions = {
    center: new google.maps.LatLng(40.832787, 13.81395),
    zoom: 9,
```


www.dbooks.org

```
        mapTypeId: 'hybrid',
        mapTypeControl: false
    };
```

Now, when you launch the application, the base map is set to **hybrid**, and there's no way for the user to change it to anything else. Of course, you might want to provide an alternative method for the user to achieve this and, potentially, enable some base maps while hiding others. Let's say, for example, that you wanted the user to choose the map type from a simple drop-down list instead of Google's own **MapTypeControl**.

Make a copy of the Ch2_FirstMap.html file and call it Ch2_MapTypes.html. Change the contents of the **<title>** tag to reflect the new file name and add the **mapTypeControl: false** property assignment to the **mapOptions** object to hide Google's base map switcher control.

Then, just above the map ** **, create an HTML **<select>** element, as shown in Code Listing 5, and give it an **id** property so that you can refer to the element from your JavaScript code. You can then use the value of the item that the user selects in the drop-down list and pass it to the map's **setMapTypeId()** function to change the base map at runtime.

Code Listing 5: Changing the base map programmatically

```
<select id="selMapType" onChange="setBaseMap(this.value)">
    <option value="roadmap">Road Map</option>
    <option value="satellite">Satellite Map</option>
    <option value="hybrid">Hybrid Map</option>
    <option value="terrain">Terrain Map</option>
</select>
<div id="map"></div>
<script>
    var map;
    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(40.832787, 13.81395),
            zoom: 9,
            mapTypeControl: false
        };
        map = new google.maps.Map(document.getElementById('map'),
mapOptions);
    }

    function setBaseMap(mapType) {
        map.setMapTypeId(mapType);
    }
</script>
```

Selecting a different value in the drop-down list causes its **onChange** event to fire, which we handle by calling **setBaseMap()**. We pass in to **setBaseMap()** the name of the **MapType** that corresponds to the user's choice of map type, and use that in the map's **setMapTypeId()** function to change the map. Note also that we have moved the **map** variable declaration outside of the **initMap()** function so that we can access the map object within our **setBaseMap()** function (and any other functions we might want to add later on).



Note: Find out more about the different map types and their capabilities in the API reference documentation here.

www.dbooks.org

Chapter 3 Customizing the Map Controls

At the end of the previous chapter, we showed you how your Google Maps API applications include the **`MapTypeControl`** by default, which allows users to select different base maps at runtime. We also demonstrated how you can hide this control if you don't want users to have this option by disabling it within the map options object. In this chapter, we'll have a look at some of the other controls that Google provides, and how you can hide, show, and change how they are displayed.

The default map controls

If you don't configure your application otherwise, the controls shown in Figure 10 are included in your application.

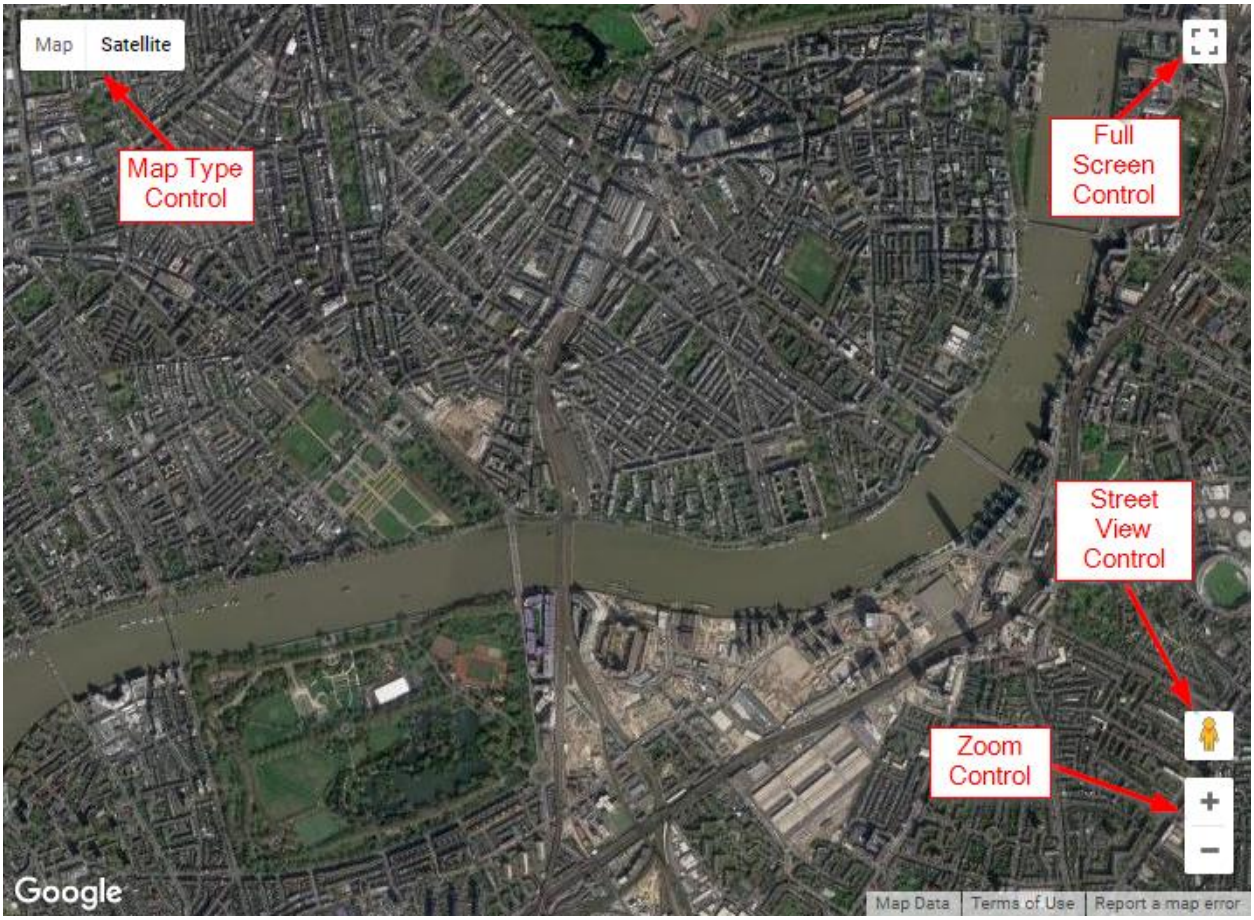


Figure 10: The default map controls.

- **Map Type control**: We've already discussed this control. It allows you to choose a different base layer for your map from **`roadmap`**, **`satellite`**, **`hybrid`**, and **`terrain`**.

- **Full Screen control**: Lets you open the map in full screen mode on desktop and Android devices (this feature is not supported, and therefore does not appear, on iOS devices).
- **Street View control**: Enables Street View mode for areas where street view imagery is visible. To use it, just drag the “peg man” to the area of interest.
- **Zoom control**: Provides buttons for zooming in (+) and out (-) of the map.

Hiding the default controls

If you don’t want these controls to appear in your application, copy Ch2_MapTypes.html to a new file called Ch3_HideMapControls.html. Remove the **`<select>`** element and the **`setBaseMap()`** function, and then amend your **`mapOptions`** object, as shown in Code Listing 6.

Code Listing 6: Hiding the default map controls

```
<div id="map"></div>
<script>
    var map;
    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12,
            mapTypeControl: false,
            zoomControl: false,
            streetViewControl: false,
            fullscreenControl: false
        };
        map = new google.maps.Map(document.getElementById('map'),
mapOptions);
    }
</script>
```

The map now appears without any controls, as seen in Figure 11.

www.dbooks.org



Figure 11: The map with no default controls.

A quicker way to disable all the default controls is to set the **`disableDefaultUI`** property to **`true`**:

```
var mapOptions = {
    center: new google.maps.LatLng(51.503, -0.135),
    zoom: 12,
    disableDefaultUI: true
};
```

If you use **`disableDefaultUI`** to remove the default controls, you can explicitly enable any controls individually by setting the control property to **`true`**. For example, to add the zoom control back in:

```
var mapOptions = {
    center: new google.maps.LatLng(51.503, -0.135),
    zoom: 12,
    disableDefaultUI: true,
    zoomControl: true
};
```

Adding a scale bar

In addition to the default controls, you can add a scale bar to your Google Maps API applications with **`scaleControl`**:

```
var mapOptions = {
    center: new google.maps.LatLng(51.503, -0.135),
    zoom: 12,
    scaleControl: true
};
```



Figure 12: Scale control

Note: If you worked with the Google Maps API in earlier versions, you might be wondering where other controls, like Overview Control and Pan Control, have gone. They were deprecated in September 2015, leaving a veritable paucity of out-of-the-box controls in recent versions. The good news is that you can create your own controls, although this is beyond the scope of this book. Consult the reference documentation for details here.

Changing control display and positioning

Earlier in this chapter, we decided that we didn't like the way the **`MapType`** control looked and went about creating our own version as a drop-down list. It turns out, however, that we need not have bothered. The **`MapType`** control, like some of the other controls, allows us to specify different display options, including a drop-down list. We can also change the location of the control on the map by specifying a **`ControlPosition`** constant in each control's options object's **`position`** property.

Figure 13 shows the available **`ControlPosition`** constants, and to which part of the map they refer.

www.dbooks.org

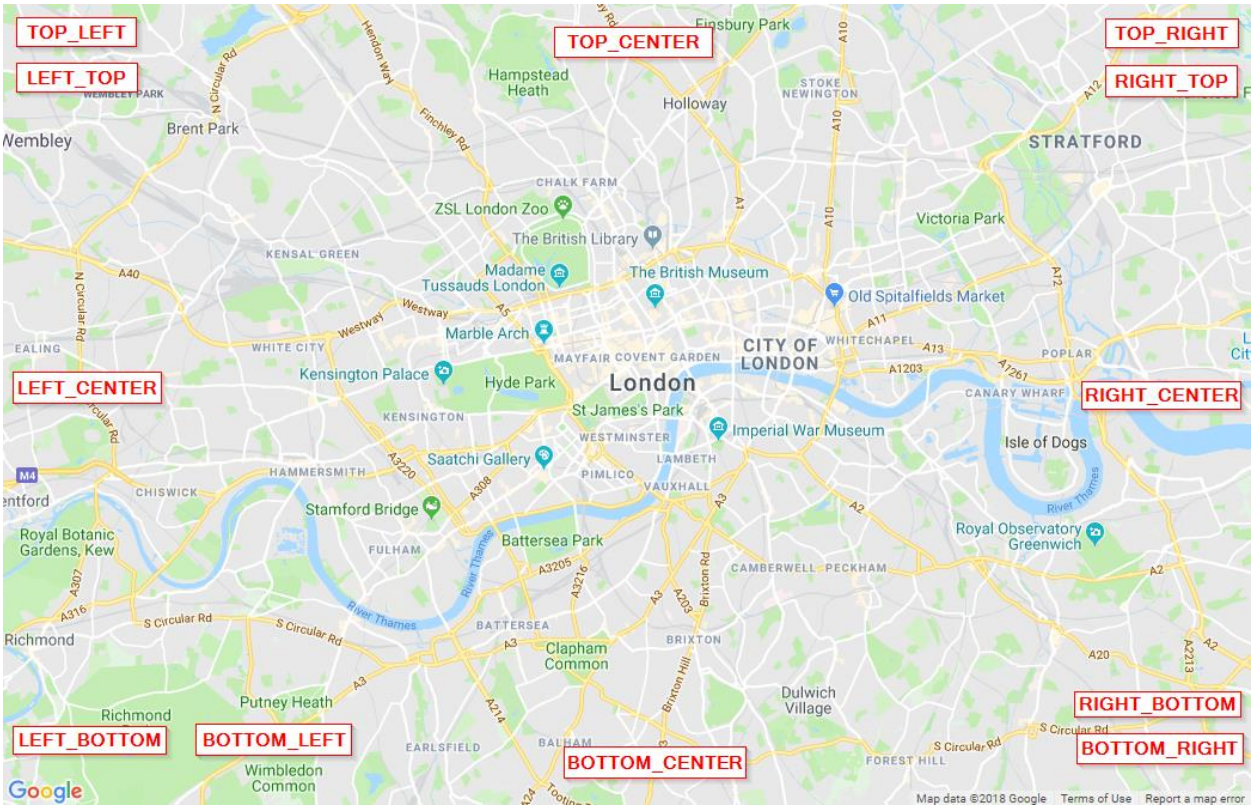


Figure 13: ControlPosition constants.

To demonstrate how this works, let's configure the Map Type control as a drop-down list and place it in the top center of the map. Let's also move the zoom and fullscreen controls so that they are in the bottom-left and bottom-right corners of the map, respectively. Copy Ch3_HideMapControls.html to a new file called Ch3_ConfiguringMapControls.html, and make the changes shown in Code Listing 7 to your **`initMap()`** code.

Code Listing 7: Changing the appearance and position of map controls

```
<script>
    var map;
    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12,
            disableDefaultUI: true,
            mapTypeControl: true,
            mapTypeControlOptions: {
                mapTypeIds: [google.maps.MapTypeId.ROADMAP,
                                google.maps.MapTypeId.HYBRID],
                style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
                position: google.maps.ControlPosition.TOP_CENTER
            },
            zoomControl: true,
```

```
            zoomControlOptions: {
                position: google.maps.ControlPosition.BOTTOM_LEFT
            },
            fullscreenControl: true,
            fullscreenControlOptions: {
                position: google.maps.ControlPosition.BOTTOM_RIGHT
            }

        };
        map = new google.maps.Map(document.getElementById('map'),
mapOptions);
    }
</script>
```

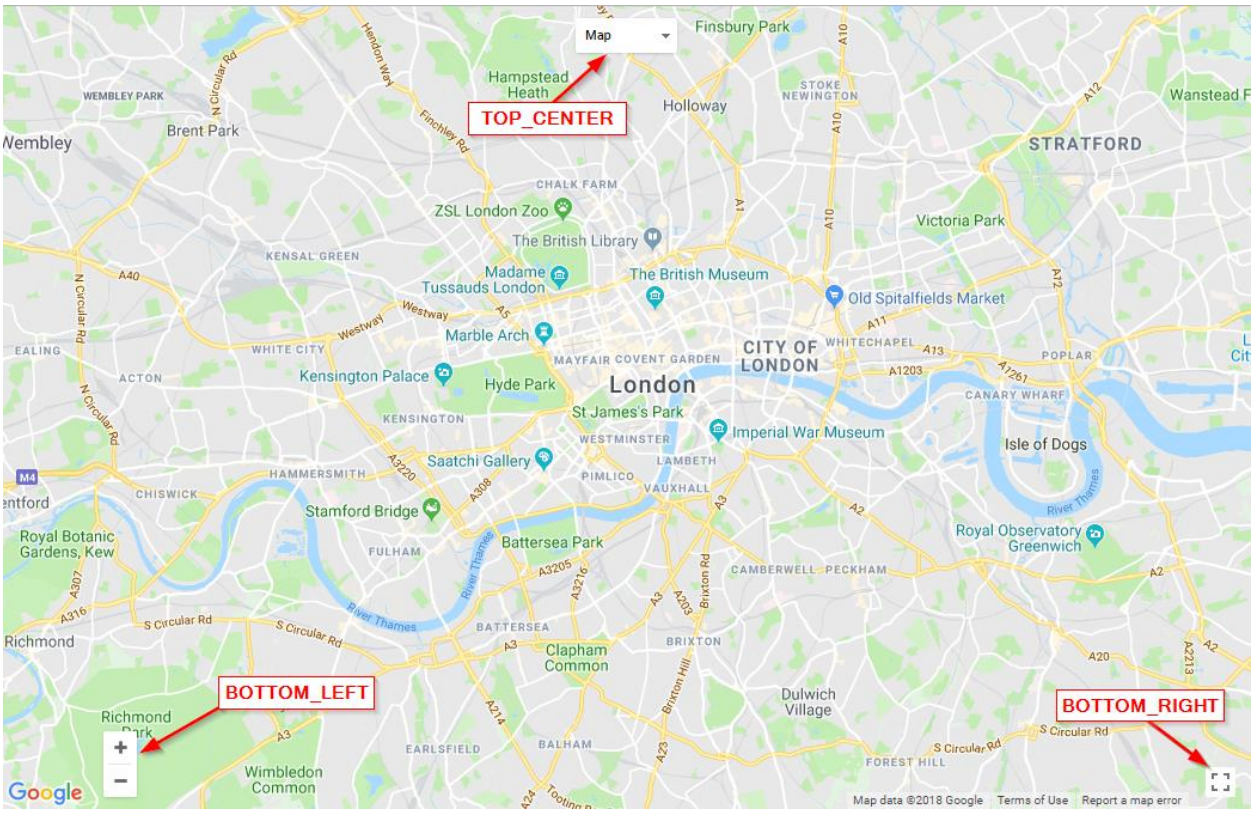


Figure 14: The relocated and restyled map controls.

Creating custom controls

Of course, there will come a time when the out-of-the-box controls are not sufficient for our purposes. The great thing is that we can use any HTML element as a custom map control.

www.dbooks.org

Since our drop-down list of base maps now appears to be a bit redundant, let's create a button that returns the map to the default center and zoom level. We'll complete the functionality for this control later, when we talk about events. But, for now, let's just create and position the button.

First, copy Ch3_ConfiguringMapControls.html to a new file called Ch3_CustomControl.html. Then, in the **<style>** tag, add CSS for a button with an ID of **btnReset**, as shown in Code Listing 8.

Code Listing 8: The CSS for our custom button

```
<style>
    #map {
        height: 100%;
    }

    #btnReset {
        background: aqua;
        font-family: "Times New Roman", Times, serif;
        font-size: 2em;
        text-align: center;
        height: 40px;
        width: 130px;
        border: 5px solid blue;
        padding: 5px;
    }

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

Then, add the code that creates a ** ** element called **btnReset** and positions it on the map, as shown in Code Listing 9.

Code Listing 9: Creating and positioning the custom control

```
<script>
    var map;
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12,
            disableDefaultUI: true,
```

```
        };
        map = new google.maps.Map(document.getElementById('map'),
                                                    mapOptions);

        var myResetControl = document.createElement('div');
        myResetControl.id = "btnReset"
        myResetControl.innerHTML = 'Reset';

map.controls[google.maps.ControlPosition.TOP_LEFT].push(myResetControl);
    }
</script>
```

Note how, in the last line of code, we add the control to the map's **`controls`** property. We can't use the usual approach of configuring the **`MapOptions`** object, because the API does not know anything about our control. The map's **`controls`** property consists of an array of each **`ControlPosition`**, which in turn is an array that contains all the controls in that location. We are using the standard JavaScript **`push()`** method to add our control to that array.

When you launch the page in the browser, you should see the control appear, as shown in Figure 15.

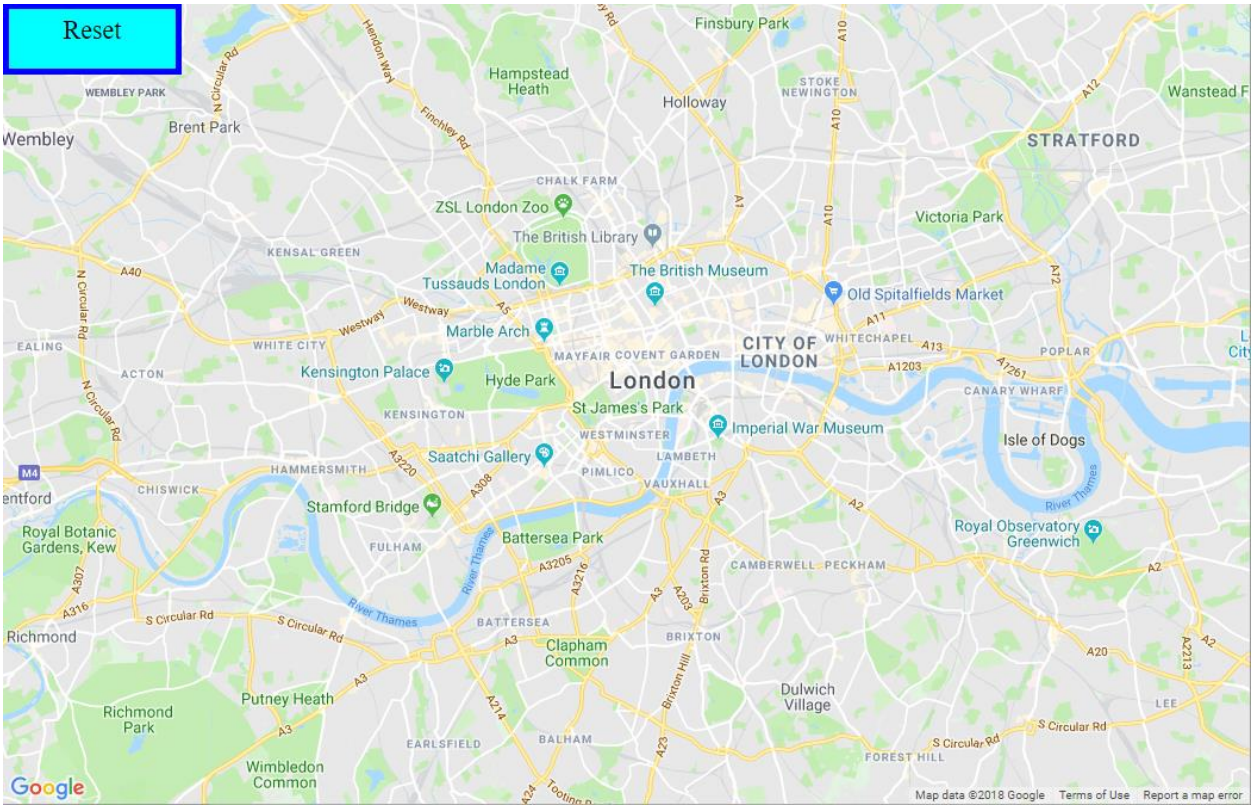


Figure 15: Our custom control.

Unfortunately, the control doesn't do anything yet. Save your work in Ch3_CustomControl.html, because we'll be returning to this example in the chapter titled "Working with Events."

www.dbooks.org

Chapter 4 Adding Data to the Map

So far, we have played around with the basic appearance of the map and the various controls we can use to interact with it. Now, it's time for us to start adding our own data to the map and making it our own.

Google calls the graphics that are presented on top of the base map *overlays.* In this chapter, we'll consider the different overlays that are available and give you access to a tool written specifically for this book that will make creating simple map graphics…well, *simple*.

We will cover:

- Markers
- Info windows
- Shapes: lines, circles, rectangles, and polygons
- Ground overlays
- KML layers and GeoRSS
- User-generated data

Adding markers

The simplest overlay we can show on the map is a *marker*. A marker represents a single point on the map, defined by a latitude/longitude coordinate (referred to as a **`LatLng`** in this book) and represented by an icon. Markers are typically used to show the location of points of interest, such as a business, address, and so on.

Like many other elements in the Google Maps API, a marker consists of a class that you need to instantiate to create the marker (**`google.maps.Marker`**), and an object literal that you pass to it that lets you configure its appearance and behavior (**`google.maps.MarkerOptions`**).

To create a marker, you must specify at a minimum its **`position`** (**`LatLng`**) property and the map that it should appear on. This is done by setting properties on the **`MarkerOptions`** object. You can optionally specify many other properties, including **`title`** (for rollover text) and **`icon`** (if you want to use an icon other than Google's iconic pushpin).

Copy Ch2_FirstMap.html to a new file called Ch4_SimpleMarker.html and enter the code in Code Listing 10 to place a marker at the location of Big Ben in London.

Code Listing 10: Adding a marker to the map

```
<script>
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135), zoom: 12 };
        var map;
```

```
        map = new google.maps.Map(document.getElementById('map'),

mapOptions);
        var marker = new google.maps.Marker({
            position: new google.maps.LatLng(51.510357, -0.116773),
            map: map,
            title: "Big Ben"
        });
    }
</script>
```

All we're doing in this code is creating a new instance of the **`google.maps.Marker`** class and setting its **`position`**, **`map`**, and **`title`** properties. When we open the page in a browser, the standard Google Maps marker icon appears, showing the location of Big Ben.



Figure 16: Marker showing the location of Big Ben

Tip: If you want to define your marker but then have it appear later, perhaps in response to the user doing something, then leave out the `map` property value

www.dbooks.org

assignment. When you are ready to display the marker, call *marker.setMap(map)* ***instead. To remove the marker from the map, call*** *marker.setMap(null).*

Labelling the marker

When you move the mouse over the marker, the text you entered in the marker's **`title`** property appears as a tooltip. But perhaps you'd like some clue as to what the marker represents on the marker icon itself. Add the **`label`** property assignment to the code that creates the marker, as shown in Code Listing 11.

Code Listing 11: Setting the label property on the marker

```
<script>
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135), zoom: 12 };
        var map;
        map = new google.maps.Map(document.getElementById('map'),
                                                            mapOptions);
        var marker = new google.maps.Marker({
            position: new google.maps.LatLng(51.510357, -0.116773),
            map: map,
            title: "Big Ben",
            label: "B"
        });
    }
</script>
```



Figure 17: Labelling the marker.

Changing the marker icon

Suppose we don't like the standard icon? Perhaps we are displaying several different markers and want our users to be able to easily distinguish among the types of features they represent based on their appearance. We can supply our own via the **`icon`** property. First, remove the

label property you added earlier (it won't show up very well on the icon), and then specify the location of the custom icon, as shown in Code Listing 12.

Code Listing 12: Marker with custom icon

```
<script>
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135), zoom: 12 };
        var map;
        map = new google.maps.Map(document.getElementById('map'),
                                                        mapOptions);
        var marker = new google.maps.Marker({
            position: new google.maps.LatLng(51.510357, -0.116773),
            map: map,
            title: "Big Ben",
            icon: "images/big-ben.png"
        });
    }
</script>
```

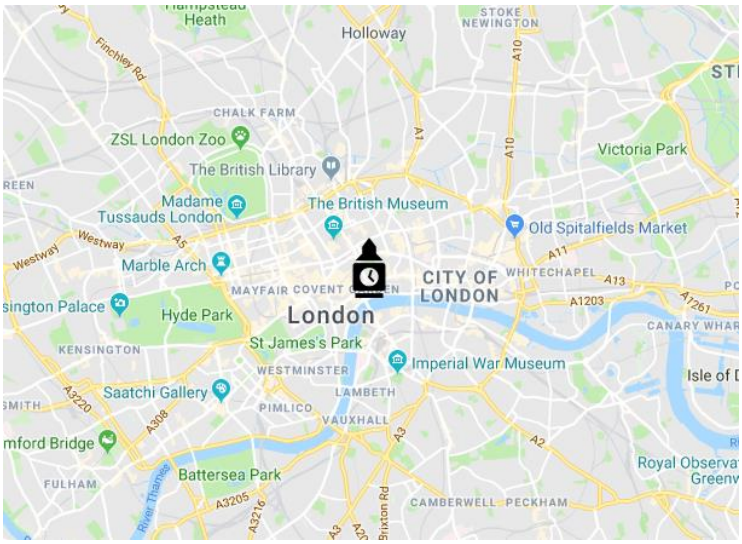


Figure 18: Custom Big Ben icon.

Animating the marker

Maybe you think that the marker still doesn't draw the user's attention sufficiently. Why not animate it? Modify Ch4_IconMarker.html by adding an **`animation`** property to the marker with a value of **`google.maps.Animation.DROP`**. Enjoy the effect when you reload the page.

www.dbooks.org

Code Listing 13: "Dropping" the icon on the map

```
<script>
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135),
                                                                zoom: 12 };
        var map;
        map = new google.maps.Map(document.getElementById('map'),

mapOptions);
        var marker = new google.maps.Marker({
            position: new google.maps.LatLng(51.510357, -0.116773),
            map: map,
            title: "Big Ben",
            icon: "images/big-ben.png",
            animation: google.maps.Animation.DROP

        });
    }
</script>
```

Still not obvious enough for you? Try changing the **`animation`** property value to **`google.maps.Animation.BOUNCE`** and enjoy the show!

Moving the marker

Maybe it's not the appearance of the marker that bothers you, as much as its location. Set the marker's **`draggable`** property to **`true`**, and you'll be able to drag the marker to a new position on the map.

Click the marker to select it, keep the left mouse button down, and drag it across the map. Release the mouse button when you are satisfied with the new location (if you're working with the Big Ben marker, please put it back where you found it after you have finished experimenting: we Brits will be mightily displeased if one of our most famous monuments suddenly disappears!).



Note: There are many other options for configuring the display of markers that we have not covered here. Consult the reference documentation for more ideas.

Info windows

So far, we have a little tooltip that displays information about our marker when the user hovers the mouse cursor over it. That's okay for tiny amounts of information, but sometimes you want to show more. That's where an info window comes in.

An info window is like a speech bubble in a comic book. The main area shows your chosen content and there is a tapered stem that anchors itself to a geographic point on the map.

Markers are good things to attach info windows to, because a single geographic coordinate defines them. If you wanted to attach an info window to a polygon (more on that later), then you would need to find a point within that polygon (typically the center point) to which to anchor it.

You can include pretty much any content you like within the info window, including text, HTML, or a DOM element. Once you have created an instance of **`InfoWindow`** and configured its appearance within an associated **`InfoWindowOptions`** object, then you have to figure out a way to trigger its display. Usually this is in response to a user action, like clicking the associated marker.

Let's add an info window to our Big Ben marker, to show users more information about this iconic London landmark. Start by copying Ch4_IconMarker.html to a new file called Ch4_InfoWindow.html. You'll need to add some CSS to the **`<style>`** tag and change the **`initMap()`** function, as shown in Code Listing 14.

Code Listing 14: Adding an Info Window

```
<!DOCTYPE html>
<html>

<head>
    <title>Ch4 Info Window</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        #map {
            height: 100%;
        }

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

        .iwContent {
            width: 300px;
        }
```


www.dbooks.org

```
        .iwHeader {
            font-family: "verdana";
            font-weight: bold;
            font-size: 2em;
            color: blue;
        }

        .iwBody {
            font-family: "arial";
            font-size: 1em;
        }

        .iwImage {
            float: left;
            padding: 5px;
        }
    </style>
</head>

<body>
    <div id="map"></div>
    <script>
        var strContent = "<div class='iwContent'><div
class='iwHeader'>Big Ben</div>" +
            "<img class='iwImage' src='images/big-ben-iw.jpg'/>" +
            "<div class='iwBody'<p>Big Ben is the nickname for the Great
Bell of the clock at the north end of the Palace of Westminster in
London.</p>" +
            "<p>It is usually extended to refer to both the clock and the
clock tower.</p>" +
            "<p><a
href='https://en.wikipedia.org/wiki/Big_Ben'>Wikipedia</a></p></div>";

        function initMap() {
            var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135), zoom: 12 };
            var map;
            map = new google.maps.Map(document.getElementById('map'),
mapOptions);
            var infowindow = new google.maps.InfoWindow({
                content: strContent
            });
            var marker = new google.maps.Marker({
                position: new google.maps.LatLng(51.510357, -0.116773),
                map: map,
```

```
                title: "Big Ben",
                label: "B"
            });
            marker.addListener('click', function () {
                infowindow.open(map, marker);
            });
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]" async
        defer></script>
</body>

</html>
```

In Code Listing 14, the following code creates the **InfoWindow** and sets the content to be the HTML fragment we stored in the **strContent** variable:

```
var infowindow = new google.maps.InfoWindow({
    content: strContent
});
```

The following code adds an event listener to the marker's **click** event:

```
marker.addListener('click', function () {
    infowindow.open(map, marker);
});
```

When the user clicks on the marker, the info window displays at the coordinates of the marker, as shown in Figure 19.



Figure 19: Info Window.

www.dbooks.org

The fact that we're associating this info window with a marker makes positioning the info window doubly easy. We can just pass the marker itself as the second parameter to the **infowindow.open([*map, anchor*])** function, and the Google Maps API will figure it out. Sometimes, however, you'll want to use an info window for something else, in which case you'll need to set the **position** property of the info window before calling **infowindow.open()** to display it. You can do this either in the **InfoWindowOptions** object:

```
var infowindow = new google.maps.InfoWindow({
    content: strContent
    position: new google.maps.LatLng(-34, 151)
});
```

Or specify the position at runtime by calling **setPosition(*LatLng*)**:

```
infoWindow.setPosition(new google.maps.LatLng(-34, 151));
```

Adding shapes

If you want to display a county boundary or the path of a winding river, then the humble marker won't cut it. You need a shape. The difference between a shape and a marker is that a marker is a single, one-dimensional geographic coordinate, whereas a shape occupies 2D space.

The Google Maps API supports the following shapes:

- Polyline
- Polygon
- Circle
- Rectangle

In this section, we'll add all these shapes to our map. Start by copying Ch2_FirstMap.html to a new file called Ch4_Shapes.html.

The Overlay Creator tool

Since it can be laborious to work out the geographic coordinates that define a shape, I have created a tool that will let you draw the shapes you want on a map and generate the code to represent that shape in the API. You can see a screenshot of the tool in action in Figure 20.

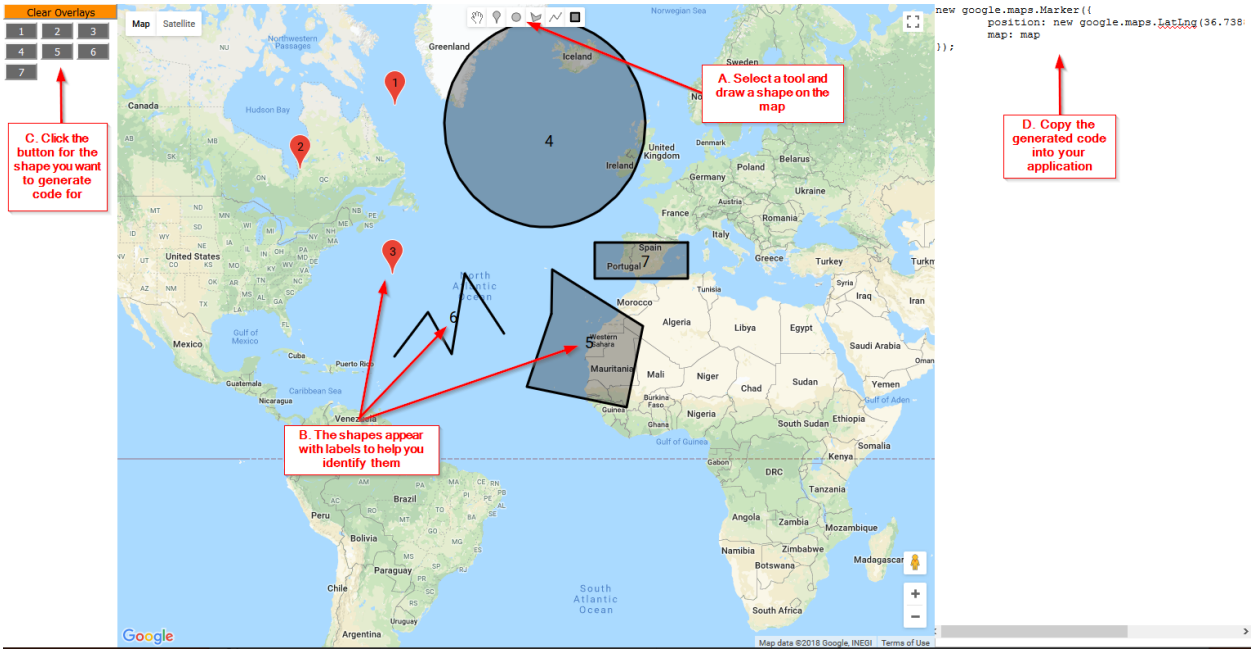


Figure 20: The Overlay Creator application.

Note: You can access the Overlay Creator tool here.

To use the Overlay Creator application:

1. Select a tool from the Drawing Toolbar at the top of the map and draw the overlay you want (marker, polyline, polygon, circle, or rectangle).



Figure 21: The Drawing Toolbar

2. Each overlay is given a numerical label that corresponds to a button in the left-hand pane.
3. Click the button for any overlay to generate the code that creates that overlay in the right-hand pane.
4. Copy the generated code into your application, making any necessary changes to variable names. For example, the Overlay Creator assumes that your **`google.maps.Map`** object is called **`map`**, and will give an error if your **`map`** is called something else, like **`myMap`**.
5. Clear the map canvas at any time by clicking **Clear Overlays**.

www.dbooks.org

Polylines

A *polyline* is a collection of connected line segments, defined by an array of **`LatLng`** objects.

Use the Overlay Creator application to create the polyline. First, zoom the focus of the map on London in the United Kingdom.

Next, select the **`Polyline`** tool and click a point on the map where you want to start your line. Move the cursor to another location and click again. Repeat two or three more times to create a meandering line, and then double-click the final location to complete the line.

Then, click the button that corresponds to the label on the line you just created. This will populate the code area. Copy and paste that code into the Ch4_Shapes.html file at the bottom of the **`initMap`** function, as demonstrated in Code Listing 15.

Code Listing 15: Adding a polyline

```
<script>
    // The code to create the map goes here.
    function initMap() {
        var mapOptions = { center: new google.maps.LatLng(51.503, -
0.135), zoom: 12 };
        var map;
        map = new google.maps.Map(document.getElementById('map'),
mapOptions);

        var polyLinePath = [
            { lat: 51.52013885206369, lng: -0.17030430061413426 },
            { lat: 51.526120070846396, lng: -0.17905903083874364 },
            { lat: 51.530178307607414, lng: -0.1677293799598374 },
            { lat: 51.53455652579403, lng: -0.1747674964149155 },
            { lat: 51.53647853735119, lng: -0.1761407874305405 },
            { lat: 51.54331169894711, lng: -0.1747674964149155 }
        ];

        new google.maps.Polyline({
            path: polyLinePath,
            map: map
        });
    }
</script>
```

In Code Listing 15, we have a variable called **`polyLinePath`** that stores an array of latitude/longitude pairs. We then create an instance of **`google.maps.Polyline`** and set the **`path`** property in the **`PolylineOptions`** object to refer to that array. Just like a marker, the polyline needs to be associated with a **`google.maps.Map`** object before it will display, and we make that association by setting the **`map`** property of the **`PolylineOptions`** object.

Open the page in a browser to verify that your polyline matches the one that you created in the Overlay Creator, as shown in Figure 22.

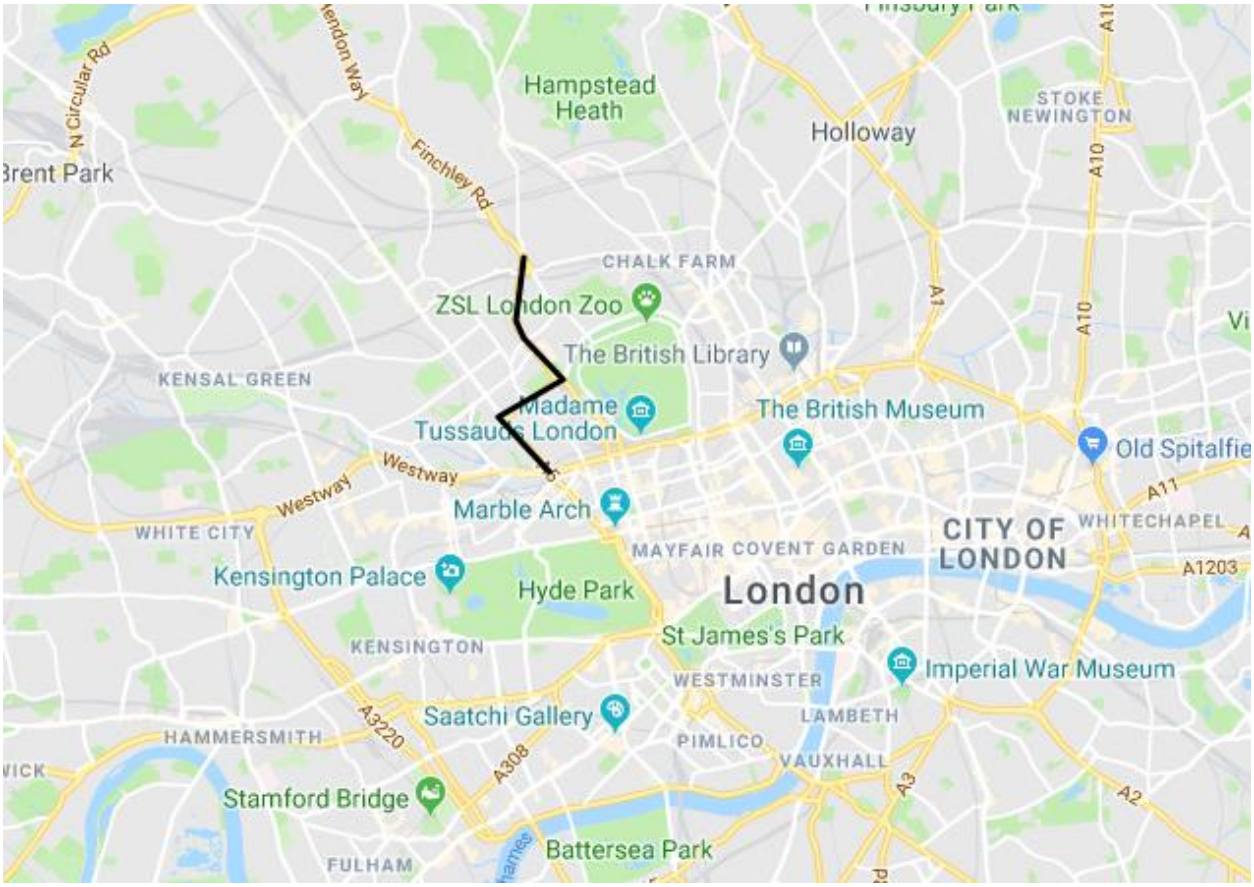


Figure 22: The polyline added to the map.

We haven't specified any symbology, so the polyline uses the default symbology, which is a bit dark and sinister-looking. Let's change that by setting some more options on the **`PolylineOptions`** object:

- **`strokeColor`** is a hexadecimal HTML color of the format "**`#FFFFFF`**".
- **`strokeOpacity`** is a numerical value between 0.0 (fully transparent) and 1.0 (fully opaque) that determines the opacity of the line's color. The default is **`1.0`**.
- **`strokeWeight`** is the width of the line in pixels.

For example, the settings in Code Listing 16 result in a polyline that looks like the one shown in Figure 23.

Code Listing 16: Specifying custom line symbology

```
new google.maps.Polyline({
    path: polyLinePath,
    map: map,
    strokeColor: "#0000FF",
```


www.dbooks.org

```
    strokeOpacity: 0.5,
    strokeWeight: 5
});
```

The result is a fatter, blue polyline that is partially transparent.

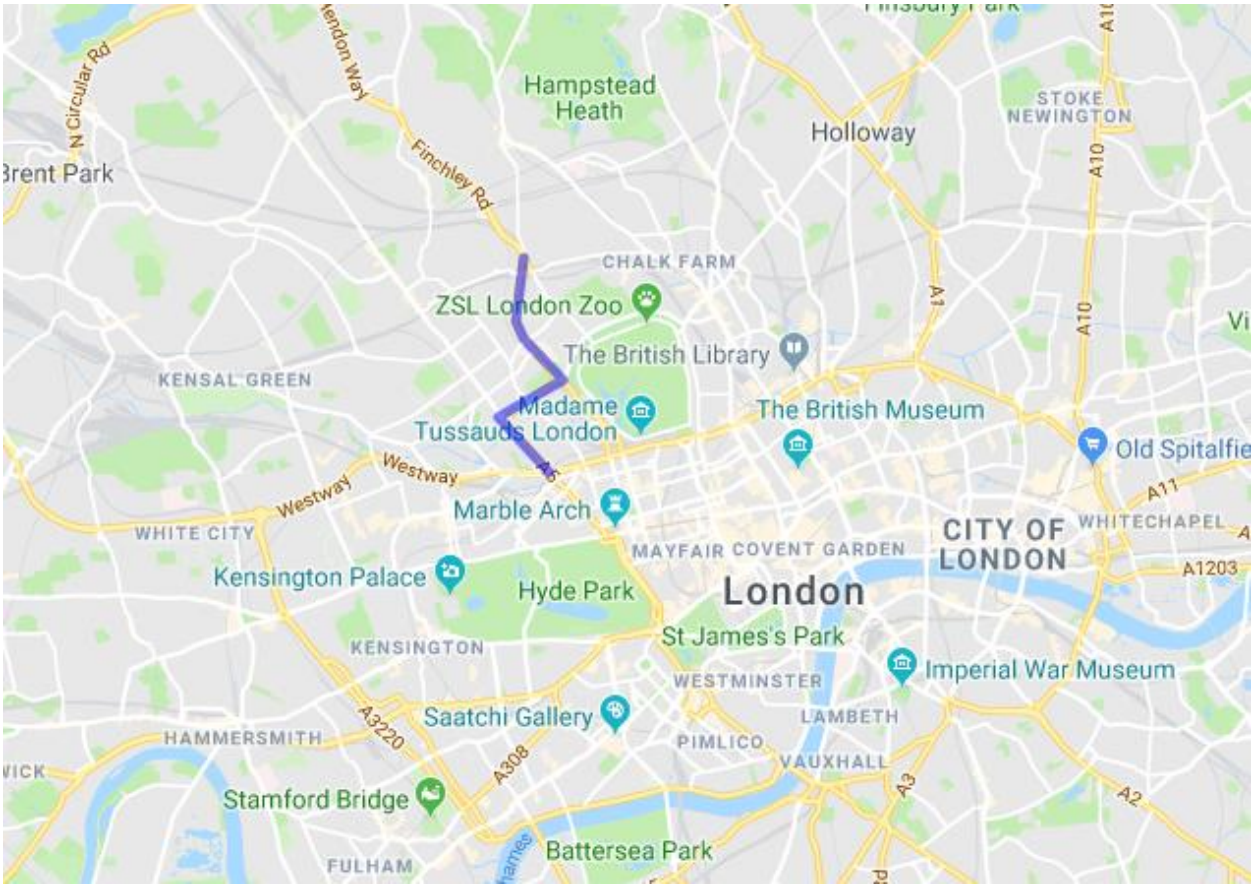


Figure 23: Polyline with custom symbology.

Polygons

A polygon is very similar to a polyline, in that it is defined by a **`path`** that contains an array of latitude/longitude pairs. The major difference, however, is that the polygon encloses an area, so the first point in the path is the same as the last. Note that you do not have to specify the last point: because you are creating an instance of **`google.maps.Polygon`**, the API infers that for you.

Create a polygon using the **Overlay Creator** tool. Click points on the map to create a vertex and double-click the last point to close the polygon. Paste the generated code into Ch4_Shapes.html underneath the polyline that you created earlier.

Because the polygon encloses an area, we have access not only to the **strokeColor**, **strokeOpacity**, and **strokeWeight** properties that define the outline (or perimeter) of the polygon, but also **fillColor** and **fillOpacity**. Set different values for these properties until you're happy with the results.

Code Listing 17: The code to create a polygon

```
var polyPath = [
    { lat: 51.5355175417175, lng: -0.1574296973426499 },
    { lat: 51.534129401093, lng: -0.1624078772742905 },
    { lat: 51.53060546938201, lng: -0.16549778205944676 },
    { lat: 51.52708126487564, lng: -0.1608629248817124 },
    { lat: 51.523877205867315, lng: -0.15657139045788426 },
    { lat: 51.524411231359004, lng: -0.1514215491492905 },
    { lat: 51.52547926355398, lng: -0.1457567237098374 },
    { lat: 51.531139415963985, lng: -0.1464433692176499 },
    { lat: 51.53573109805619, lng: -0.1485033057410874 },
    { lat: 51.53679886472031, lng: -0.152108194657103 }
];

new google.maps.Polygon({
    path: polyPath,
    map: map,
    strokeColor: "#FF2222",
    strokeOpacity: 1.0,
    strokeWeight: 3,
    fillColor: "#21212C",
    fillOpacity: 0.3,

});
```

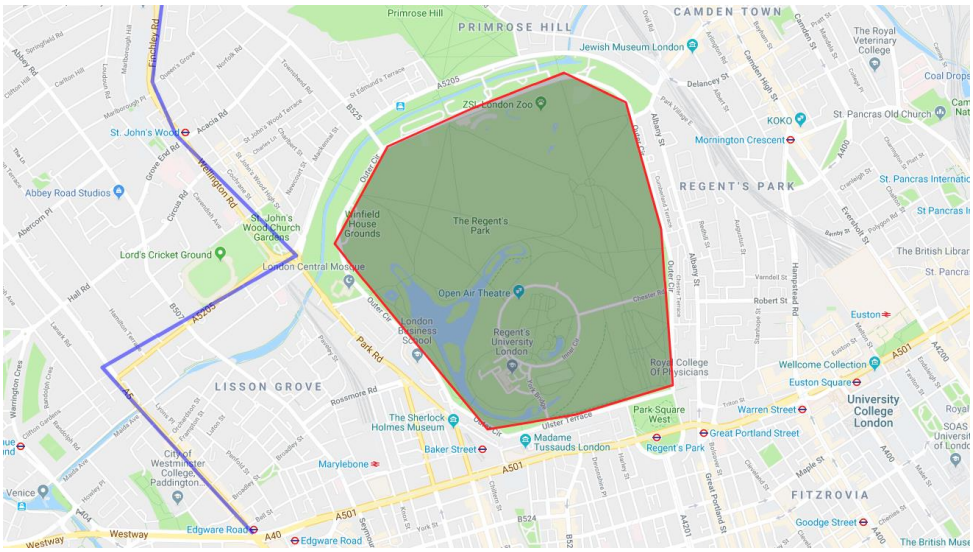


Figure 24: The polygon added to the map.

www.dbooks.org

Circles

A center point and a radius define a circle. Because a circle encloses an area, we can symbolize it in exactly the same way as a polygon.

Use the **Overlay Creator** to create a circle. Select the circle tool and click on the map where you want the center of the circle to be. Hold down the left mouse button while moving the mouse cursor and release it when the circle is the size you want it to be.

Code Listing 18: Adding a circle

```
new google.maps.Circle({
    center: new google.maps.LatLng(51.52126039042774,
                                   -0.15931797248913426),
    radius: 537.1837860475924,
    strokeColor: "#00FF00",
    strokeOpacity: 0.9,
    strokeWeight: 2,
    fillColor: "#4286F4",
    fillOpacity: 0.5,
    map: map
});
```

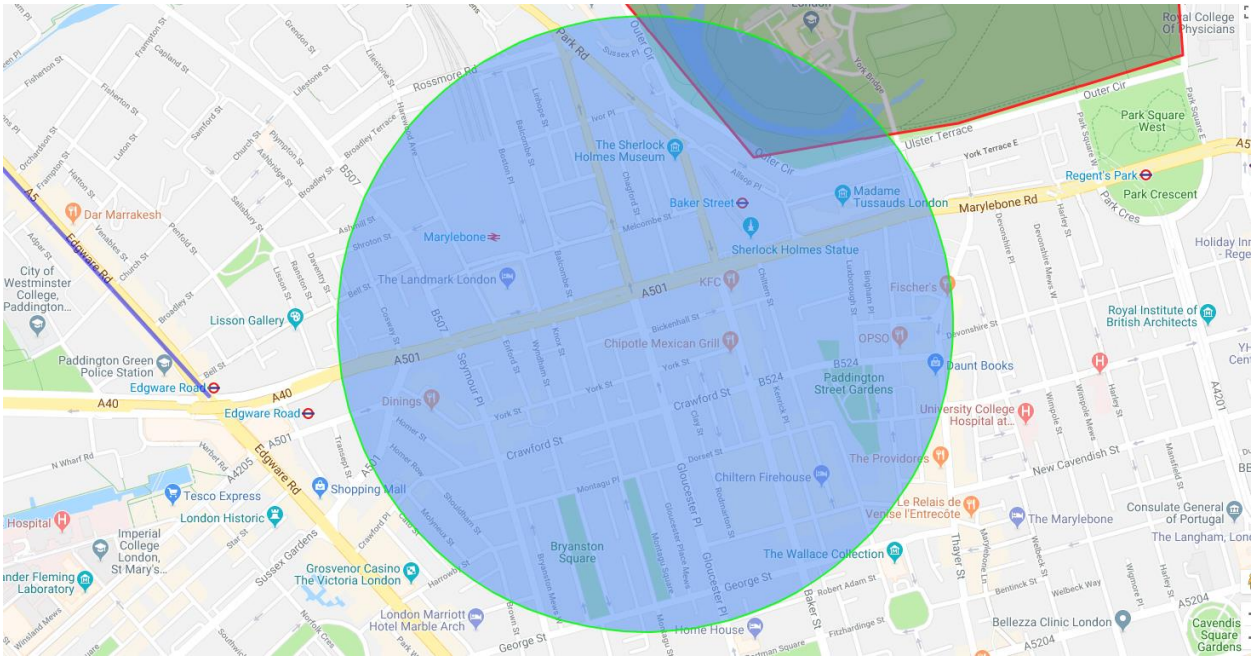


Figure 25: The circle added to the map.

Rectangles

A rectangle is a special type of polygon with four sides at 90-degree angles. You create a **google.maps.Rectangle** object by specifying the **bounds** property in its associated **RectangleOptions** object. The bounds are the location of the rectangle's corners. They can be specified either as individual points:

```
bounds: {
  north: 33.685,
  south: 33.671,
  east: -116.234,
  west: -116.251
}
```

Or as a **google.maps.LatLngBounds** object. A **LatLngBounds** object defines a rectangular area based on the points at its southwest and northeast corners. This is the approach that the Overlay Creator tool uses, as shown in Code Listing 19.

Code Listing 19: Adding a rectangle using LatLngBounds

```
var rectBounds = new google.maps.LatLngBounds(
    //southwest corner
    new google.maps.LatLng(51.525319060321756, -0.1443834326942124),
    //northeast corner
    new google.maps.LatLng(51.52937736847714, -0.13391208870007176)
);

new google.maps.Rectangle({
    bounds: rectBounds,
    strokeColor: "#4242f4",
    strokeOpacity: 1.0,
    strokeWeight: 4,
    fillColor: "#F4F141",
    fillOpacity: 0.6,
    map: map
});
```


www.dbooks.org

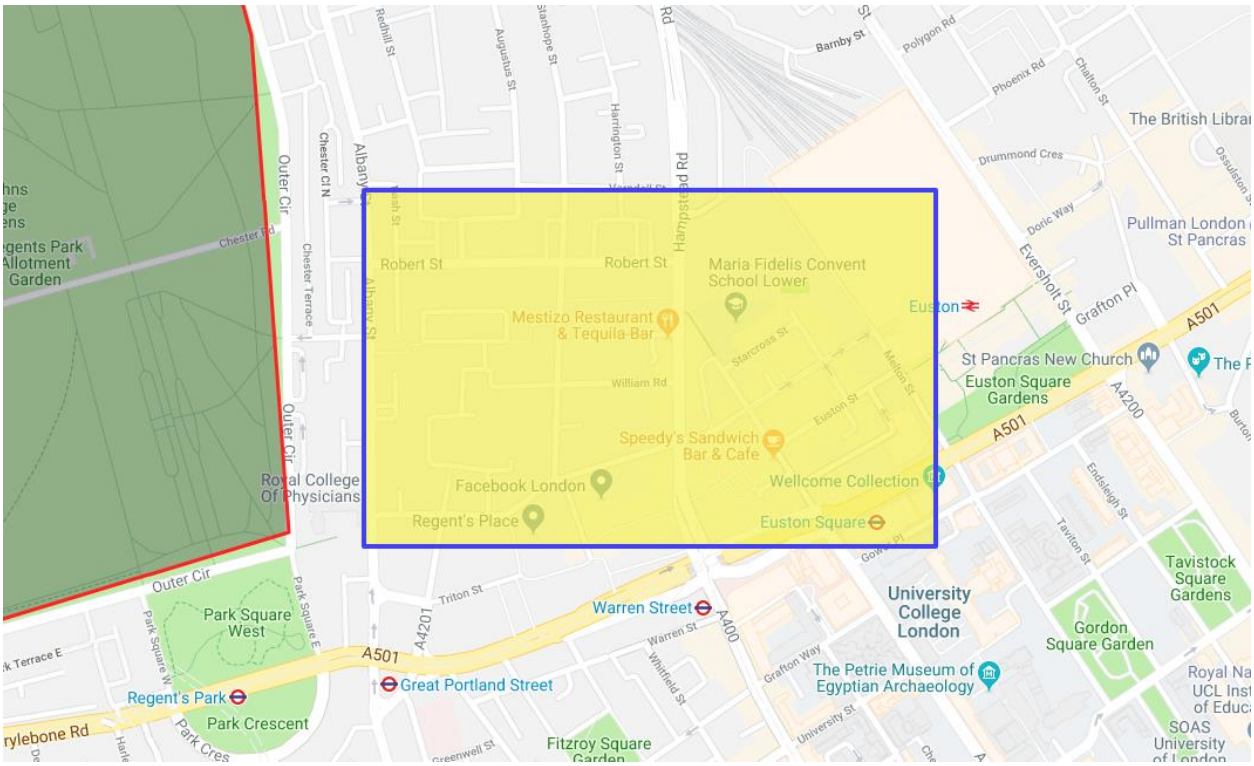


Figure 26: The rectangle added to the map.

Editing and moving shapes

You saw earlier how you could move a marker by setting its **`draggable`** property to **`true`**. You can do this with the other shapes, too.

Furthermore, you can edit a shape by setting its **`editable`** property to **`true`**, or calling **`setEditable()`** on the shape. Try this with the polygon you added earlier.

In Code Listing 20, I'm using **`setEditable()`**. To do this, I must first assign the polygon to a variable, which I call **`myPoly`**.

Code Listing 20: Making a polygon editable

```
var myPoly = new google.maps.Polygon({
    path: polyPath,
    map: map,
    strokeColor: "#FF2222",
    strokeOpacity: 1.0,
    strokeWeight: 3,
    fillColor: "#21212C",
    fillOpacity: 0.3,
});
myPoly.setEditable(true);
```

When you make an overlay editable in this way, a little point appears at each vertex, as shown in Figure 27. You can drag any of these points to change the shape. A little Undo button appears at each vertex you manipulated, so that you can revert to the previous position if you want to.

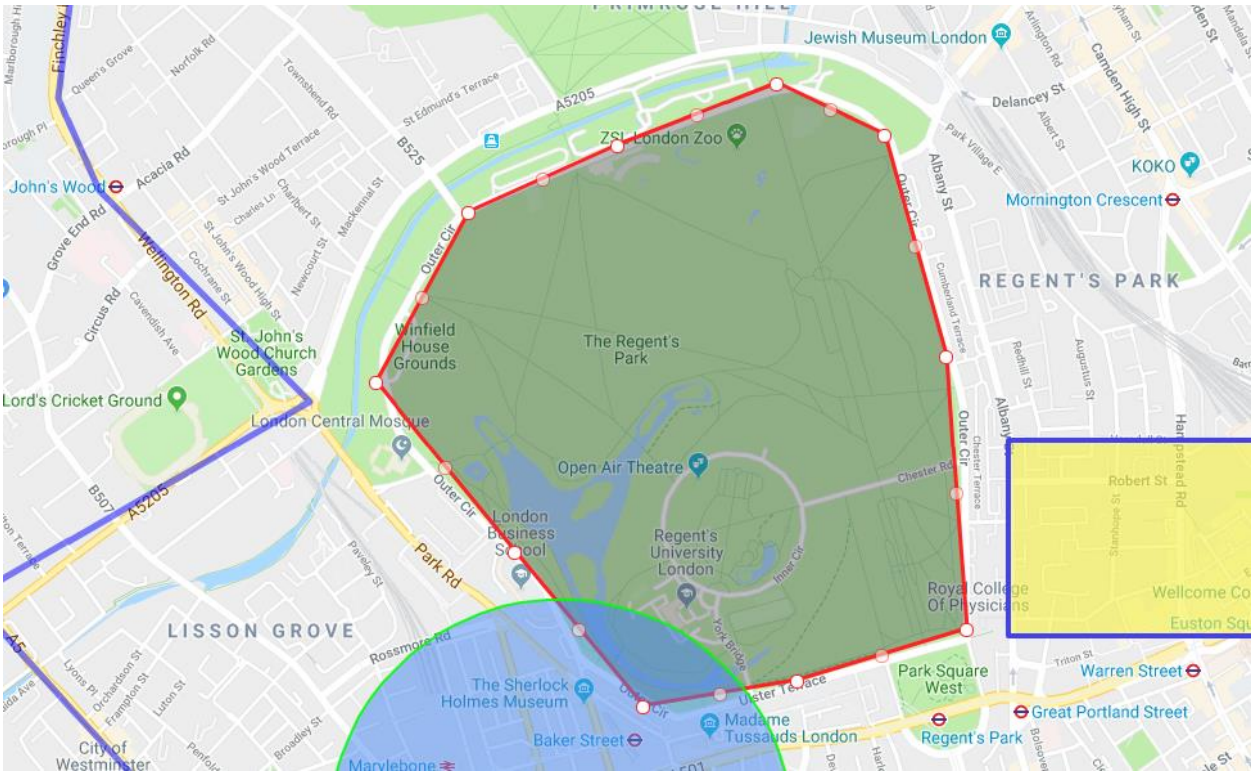


Figure 27: An editable polygon

Ground overlays

So far in this chapter, we've only looked at vector graphics and not images. You can add an image to the map in the form of a ground overlay that, like all other overlays, is geographically referenced and moves with the map. This can be useful to show things like historical map data, or different satellite imagery for a specific area.

In this example, we're going to use a ground overlay to show the floor plan at 10, Downing Street in London, where the British prime minister resides.

First, copy Ch2_FirstMap.html to a new file called Ch4_GroundOverlay.html. Then, change the map extent to focus on the Downing Street area in London and add the code that creates the ground overlay, as shown in Code Listing 21, changing the link to the image if necessary.

Code Listing 21: Creating a ground overlay

```
function initMap() {
    var mapOptions = {
```


www.dbooks.org

```
        center: new google.maps.LatLng(51.50334151281109,
                                       -0.1273725998895543), zoom: 19 };
    var map;
    map = new google.maps.Map(document.getElementById('map'),
mapOptions);

    // southwest and northeast corners where the image should be placed.
    var imageBounds = new google.maps.LatLngBounds(
        new google.maps.LatLng(51.503260954928486, -0.12832746629885605),
        new google.maps.LatLng(51.503729273890755, -0.1272773814695256)
    );
    var historicalOverlay = new google.maps.GroundOverlay(
        "images/downing-street.gif",
        imageBounds);
    historicalOverlay.setMap(map);
}
```

When you open the page in the browser, you should see the floor plan appear as shown in Figure 28 and be able to zoom and pan around as if it was part of the map. Which, because you have georeferenced it, it is!

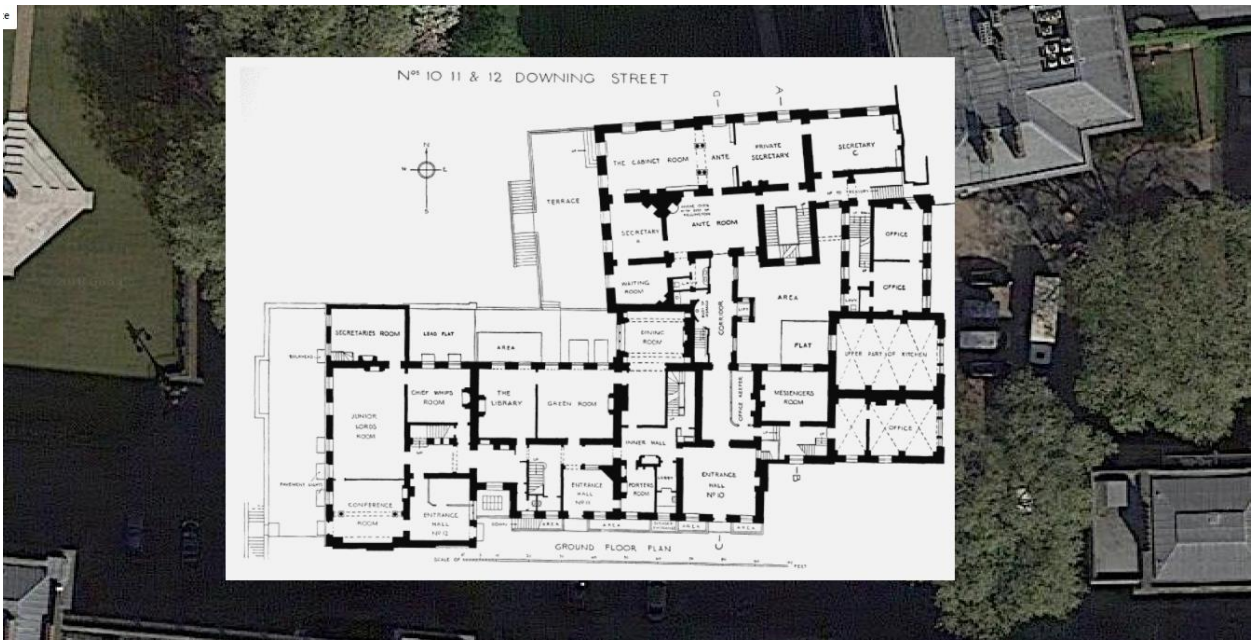


Figure 28: Floor plan of 10 Downing St., using a ground overlay.

KML and GeoRSS

KML and GeoRSS are common methods used to share geographic data. Since you're likely to find a lot of geographic datasets in these formats, you'll want to know how to use them in your Google Maps API applications.

KML stands for Keyhole Markup Language. It is an XML-based markup language specifically for geographic data and is widely used in Google Earth. The "keyhole" part of the name came from the company that originally developed Google Earth, before Google acquired it.

GeoRSS is a way of encoding geographic features as part of a web feed in the same way that your blog feed reader does.

The process for using either KML or GeoRSS is identical. Non-intuitively, the **`KmlLayer`** class handles both, rendering them into a Google Maps API overlay.

Creating a **`KmlLayer`** object is as simple as passing the URL of the GeoRSS feed or KML file to the **`google.maps.KmlLayer`** constructor.

For our first example, we're going to use a GeoRSS feed provided by the USGS (United States Geological Survey), which shows earthquake data for the past 30 days for all earthquakes with a magnitude of 4.5 and above.

The feed we're going to use is located here.

If you visit that link, you can see the raw feed data, which looks like Figure 29.



Figure 29: Extract of the USGS earthquake GeoRSS feed.

Copy Ch2_FirstMap.html to a new file called Ch4_GeoRSS.html, and amend the **`initMap`** function as shown in Code Listing 22.

www.dbooks.org

Code Listing 22: Adding the GeoRSS earthquakes feed

```
function initMap() {
    var mapOptions = { center: new google.maps.LatLng(51.503, -0.135),
                                                        zoom: 12 };
    var map;
    map = new google.maps.Map(document.getElementById('map'), mapOptions);


    var georssLayer = new google.maps.KmlLayer({
        url:
"https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/4.5_month.atom",
    });
    georssLayer.setMap(map);
}
```

When you open the page in the browser, you'll see something a bit like Figure 30. It won't be identical, as this is a live feed.



Figure 30: Live earthquake data from USGS

What just happened? The extent of the map in the **MapOptions** object has been completely ignored and the map automatically expands to show all the points in the **KmlLayer**. This is the default behavior. If you want to stop this from happening, you can set **preserveViewport: true** in the **KmlLayerOptions** object.

Also, what's with those funky blue markers? When you consume a GeoRSS feed or KML file, you're basically getting what the author intended to show you, using the symbology they defined. You're stuck with it, basically!

Let's try and load KML using the same technique. I've got a sample KML file you can use here.



Tip: Make sure the KML file you want to load is at a publicly available URL, and not on your local machine. Otherwise, it won't load.

Copy the Ch4_GeoRSS.html file to Ch4_Kml.html. Change the code that reads the GeoRSS layer to read the KML file instead, as shown in Code Listing 23.

Code Listing 23: Reading a KML file

```
function initMap() {
    var mapOptions = { center: new google.maps.LatLng(51.503, -0.135),
                                                          zoom: 12 };
    var map;
    map = new google.maps.Map(document.getElementById('map'), mapOptions);

    var kmlLayer = new google.maps.KmlLayer({
        url: "http://marklewin.github.io/gmaps/kml/cities.kml"
    });
    kmlLayer.setMap(map);
}
```


www.dbooks.org

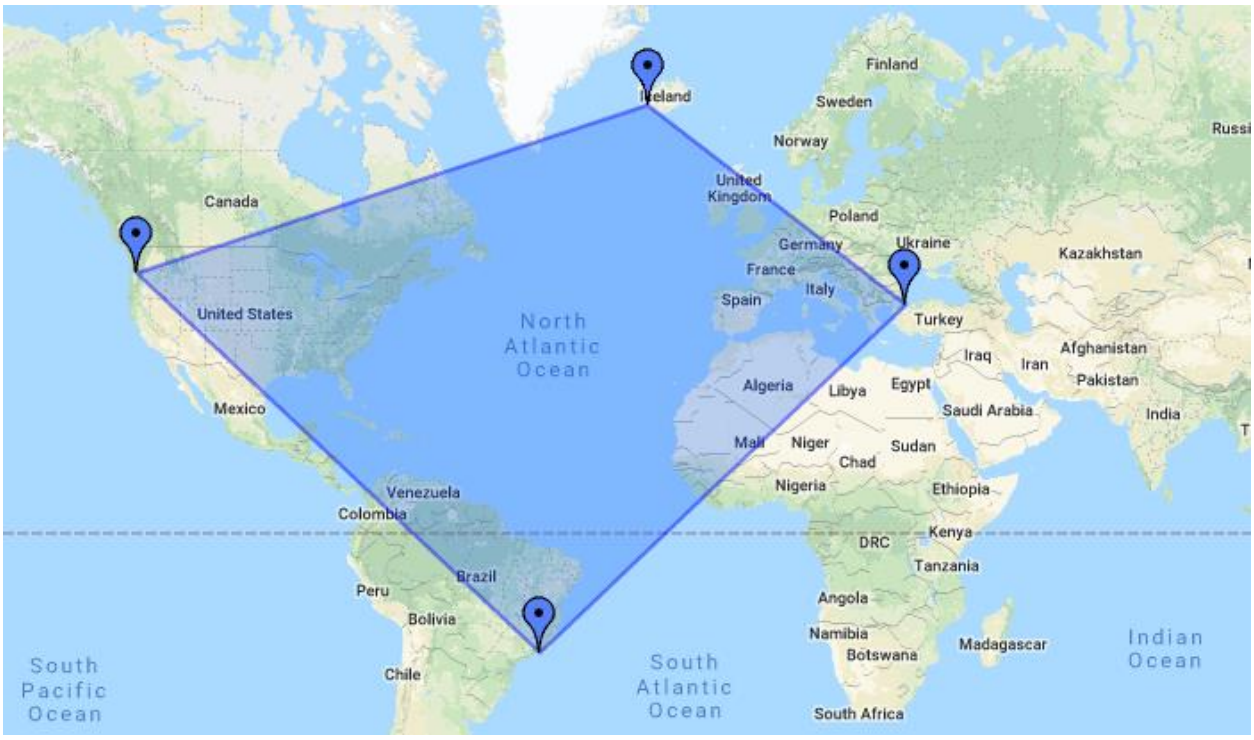


Figure 31: The KML file, displayed on the map

In case you're interested in what the contents of the KML file look like, see Code Listing 34. Note that KML uses a simple, XML-based syntax for encoding geographic features. Each feature is defined in a **<Placemark>** tag and has a **<name>** and a geometry which, in our example, is either **<Point>** or **<Polygon>**.

Code Listing 24: The cities.kml file

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <Placemark>
      <name>Portland</name>
      <Point>
        <coordinates>-122.681944,45.52,0</coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <name>Rio de Janeiro</name>
      <Point>
        <coordinates>-43.196389,-22.908333,0</coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <name>Istanbul</name>
```

```
      <Point>
        <coordinates>28.976018,41.01224,0</coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <name>Reykjavik</name>
      <Point>
        <coordinates>-21.933333,64.133333,0</coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <name>Simple Polygon</name>
      <Polygon>
        <outerBoundaryIs>
          <LinearRing>
            <coordinates>-122.681944,45.52,0
            -43.196389,-22.908333,0
            28.976018,41.01224,0
            -21.933333,64.133333,0
            -122.681944,45.52,0</coordinates>
          </LinearRing>
        </outerBoundaryIs>
      </Polygon>
    </Placemark>
  </Document>
</kml>
```

There is a lot of KML available on the web. Search for useful and interesting KML files by specifying the **`filetype:kml`** option in the Google search dialog box.

www.dbooks.org

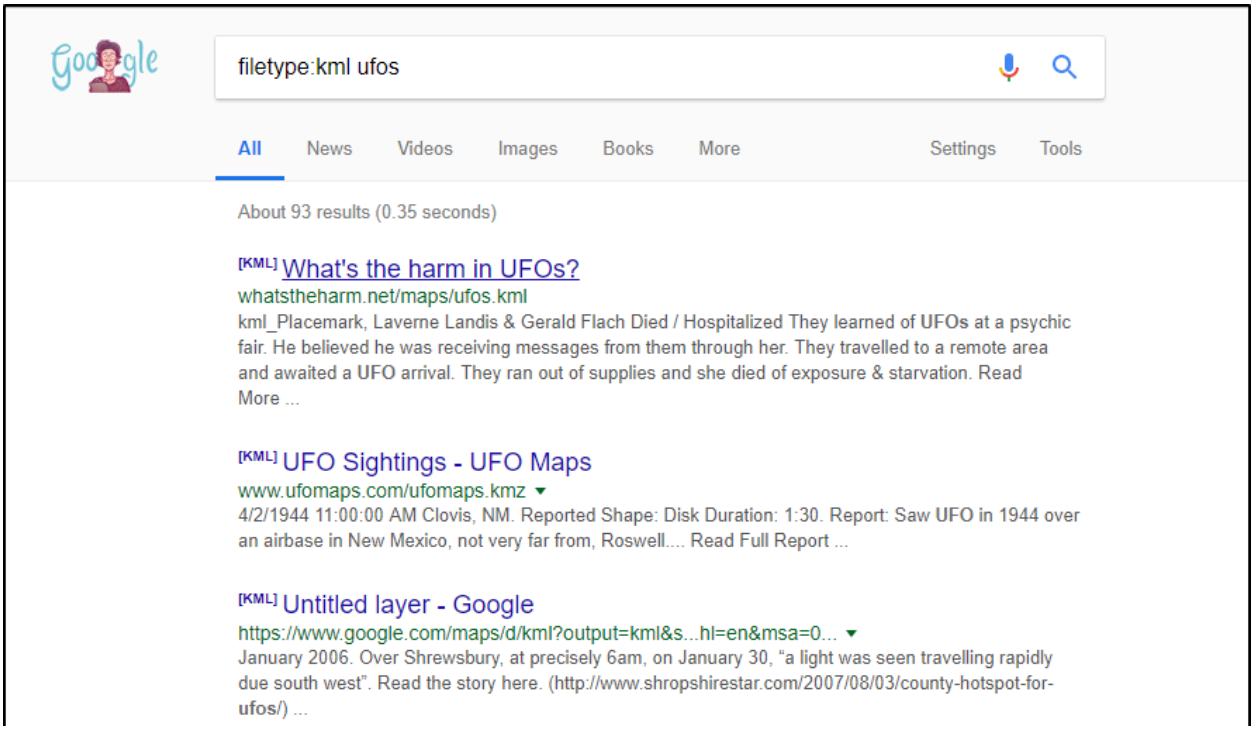


Figure 32: Searching for KML files using Google.

Tip: Learn to create your own KML files by reading Google's KML tutorial here.

Getting your users to do the work

So far, we've forced users to see our overlays. But what if we want users to create their own? This could have several potential uses. For example, we might be building an application for local citizens to report instances of unlawful garbage disposal. Our users would click the map at the location where they have observed unlawful rubbish disposal and fill in a quick form, and we would submit this to a database so that the administrators and subsequent users of the application could also see this on the map. Or, we might want to allow our users to create temporary overlays as inputs to other operations, such as drawing a polygon to define an area of interest.

Thankfully, we don't have to write the code for this ourselves. Google has anticipated this requirement and given us the Drawing Library to make adding this functionality a snap.

If you have used the Overlay Creator tool, then you have already seen the Drawing Tool in action. Essentially, it adds a toolbar to your application that includes any or all the tools shown in Figure 33.



Figure 33: The Drawing Toolbar.

A word about libraries

The Drawing Library is not part of the core Google Maps API. It's a *library* and, as such, needs to be requested separately. Google packages some functionality in this way to make the core API much quicker to download. Not every application will require drawing capability, so why force those that don't require it to download and parse all the required code? However, it's a fair bet that a Google Maps API application will require a map and overlays, so it makes sense to include that sort of stuff in the core API.

Table 1 lists the libraries that are available.

Table 1: Google Maps JavaScript API Libraries

Library	Description
`drawing`	Provides a graphical interface for users to draw polygons, rectangles, polylines, circles, and markers on the map.
`geometry`	Provides utility functions for calculating scalar geometric values (such as distance and area) on the surface of the Earth.
`places`	Enables your application to search for places such as businesses, geographic locations, or prominent points of interest, within a defined area.
`visualization`	Enables your application to create heat maps to show the density of data.

Tip: We only cover the Drawing Library in this book, but it's well worth checking out the others here.

To use a library within your application, you specify a library parameter in your bootstrap URL and provide a comma-separated list of the libraries you require. For example, to request the **`drawing`** and **`geometry`** libraries, you would use a bootstrap URL in the following format:

```
<script src="https://maps.googleapis.com/maps/api/js?key=[YOUR_API_KEY]
   &callback=initMap&libraries=drawing,geometry"></script>
```


www.dbooks.org

Adding the Drawing Manager

If you want to follow along, start by copying Ch2_FirstMap.html to create a new file called Ch4_Drawing.html.

Change the initial map extent to display the world at a smaller scale:

```
var mapOptions = { center: new google.maps.LatLng(15.5, 10.5), zoom: 3 };
```

Then, request the **drawing** library in the bootstrap URL and create an instance of **google.maps.drawing.DrawingManager**. Call **setMap()** on the **DrawingManager** object to tell the toolbar which map to appear on. Your code should look similar to that shown in Code Listing 25.

Code Listing 25: Adding the drawing library and creating the DrawingManager

```
<body>
    <div id="map"></div>
    <script>
        function initMap() {
            var mapOptions = {
                center: new google.maps.LatLng(15.5, 10.5),
                zoom: 3
            };
            var map;
            map = new google.maps.Map(
                document.getElementById('map'), mapOptions);

            var drawingManager = new google.maps.drawing.DrawingManager();
            drawingManager.setMap(map);
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]&callback=initMap&libraries=drawing"
        async defer></script>
</body>
```

That's all it takes to add drawing capability to your application. Open the page in a browser and experiment with the different tools available.

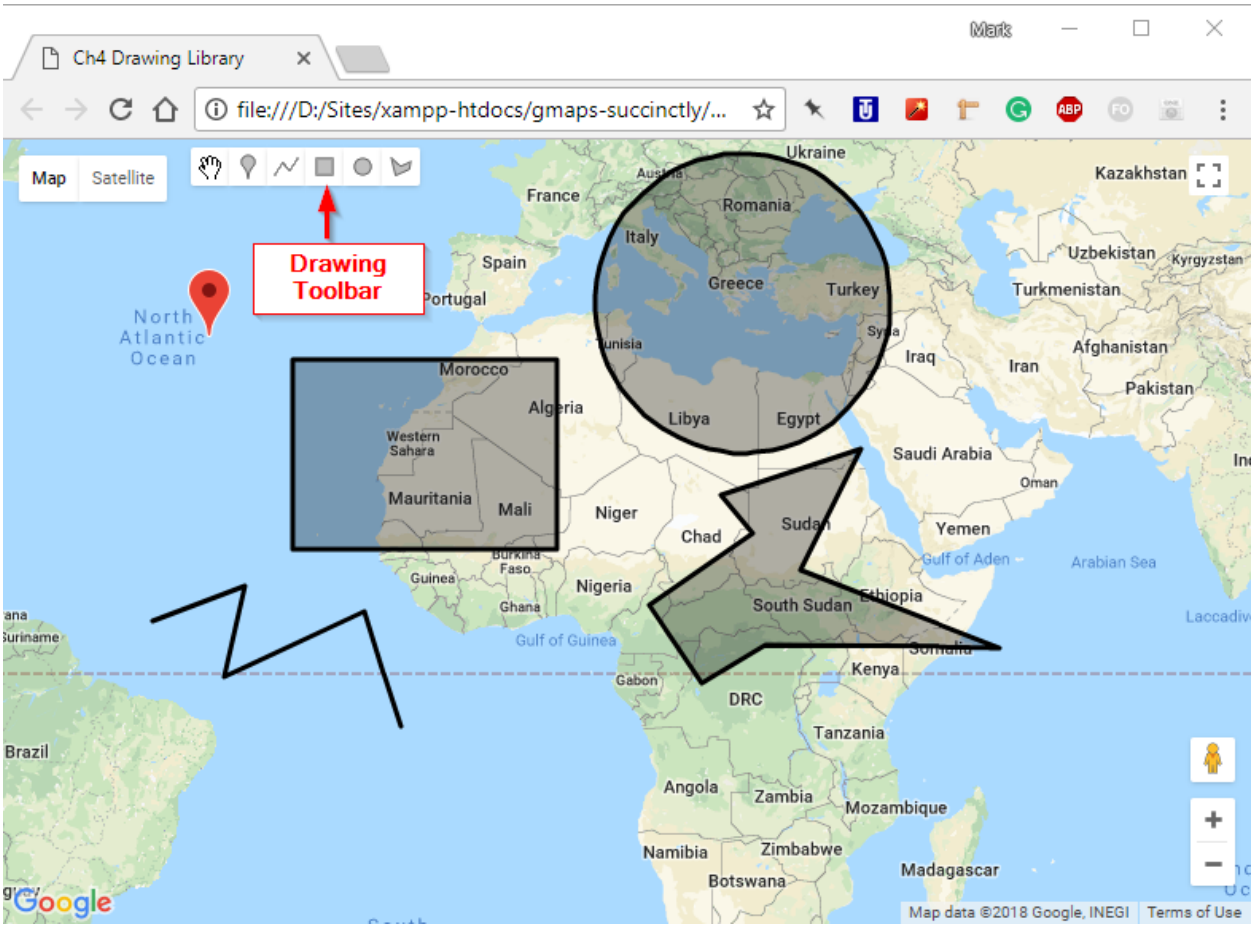


Figure 34: The Drawing Toolbar in action.

Let's tweak things a little to demonstrate how to configure the Drawing Manager. First, we want the control to be in the top center of the map. Second, we only want the marker, polyline, and polygon tools—we don't want our users to draw rectangles or circles. The default drawing mode should be marker. And finally, we want our polylines to be red, and our polygons to have a blue outline with a semitransparent yellow fill. The default marker symbol is fine.

Enter the code in Code Listing 26 to create the required **`DrawingManagerOptions`** object that will satisfy our requirements.

Code Listing 26: Configuring the DrawingManager

```
<script>
    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(15.5, 10.5),
            zoom: 3
        };
        var map;
        map = new google.maps.Map(
            document.getElementById('map'), mapOptions);
```


www.dbooks.org

```
        var drawingManager = new google.maps.drawing.DrawingManager({
            drawingMode: google.maps.drawing.OverlayType.MARKER,
            drawingControl: true,
            drawingControlOptions: {
                position: google.maps.ControlPosition.TOP_CENTER,
                drawingModes: ['marker', 'polyline', 'polygon']
            },
            polylineOptions: {
                strokeColor: "#FF0000",
                strokeWeight: 5
            },
            polygonOptions: {
                fillColor: "yellow",
                fillOpacity: 0.5,
                strokeWeight: 5,
                strokeColor: "blue"
            }
        });

        drawingManager.setMap(map);
    }
</script>
```

Now our application should appear as shown in Figure 35.

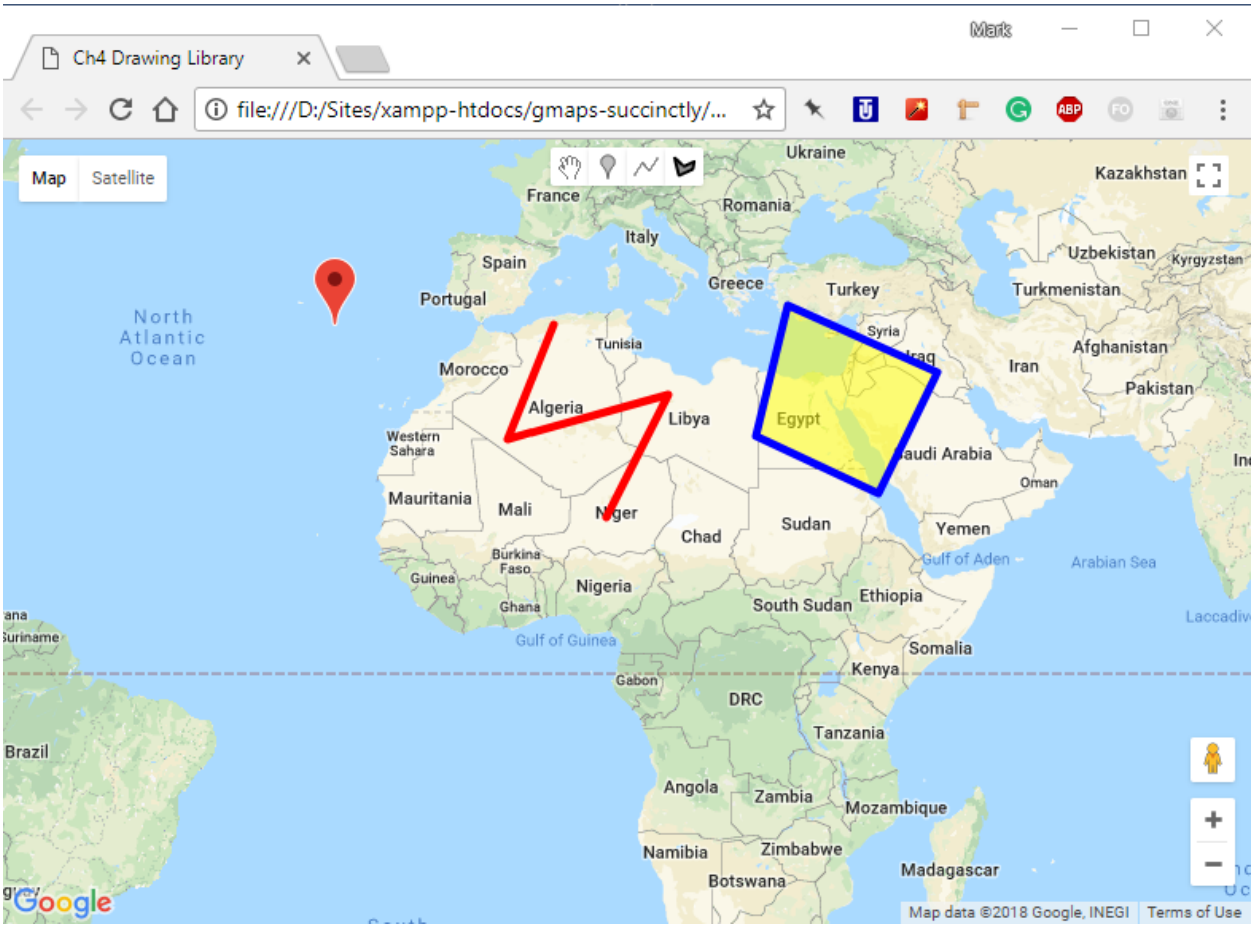


Figure 35: The customized Drawing Manager.

www.dbooks.org

Chapter 5 Working with Events

At some point, you're going to want to respond to something your user is doing, or to some state change within your application. To do that, you'll need to understand how to work with events.

JavaScript supports event-driven programming. That is, it responds to things happening by *raising* events. Events are being raised all the time and can be either from the webpage itself (when the page loads, when the user clicks a button on the webpage) or from within the Google Maps API (when the map's zoom level changes, when the map tiles are loaded, when the user clicks on a marker… you get the idea). Many of these might be of no interest to us for the application we are building, but others will be. The ones we want our application to respond to, we need to handle. We do this by writing an *event handler*, which is essentially a function that gets called automatically when a specific event is raised.

In this chapter, we will look at how to handle events in the Google Maps API, both those generated by user actions, and those arising from state changes within the API itself.

Event-handling basics

We've already handled a couple of events in our code. Quite frankly, it's hard to create any sort of meaningful application without doing so.

Look at Ch4_InfoWindow.html, where we displayed an info window when the user clicked on a marker that represented Big Ben in London. An extract of that code is shown in Code Listing 27. In that example, we handled the **`click`** event of the **`google.maps.Marker`** class. This event gets raised when the user clicks on the marker. We registered interest in this event by calling the marker's **`addListener`** method, passing in the name of the event (**`click`**) and the function that will handle it. In this instance, we provided an anonymous inline function, but we could just have easily provided the name of another function elsewhere in our code.

Code Listing 27: Handling a marker's click event

```
var marker = new google.maps.Marker({
    position: new google.maps.LatLng(51.510357, -0.116773),
    map: map,
    title: "Big Ben",
    label: "B"
});
marker.addListener('click', function () {
    infowindow.open(map, marker);
});
```

If you check out the API reference documentation for the **`google.maps.Marker`** class, you will see that it emits a number of events other than **`click`**. Other events include:

- **dblclick**
- **drag**
- **mousedown**
- **mousover**
- **rightclick**
- **position_changed**

This is typical of many classes in the Google Maps API. Each class emits its own events, and you handle them by registering interest with the class **addListener()** method and providing a handler function.

In the rest of this chapter, we'll look at several different examples of working with Google Maps API events. As you work through them, check out the API reference documentation for each of the classes to see what other events they support.

Displaying mouse-click coordinates

Let's use Google Maps API events to allow our users to click on the map and see the geographical coordinates at that location in an info window.

Start by copying Ch2_FirstMap.html to create a new file called Ch5_ClickCoords.html.

Change the initial map extent to display the world at a smaller scale:

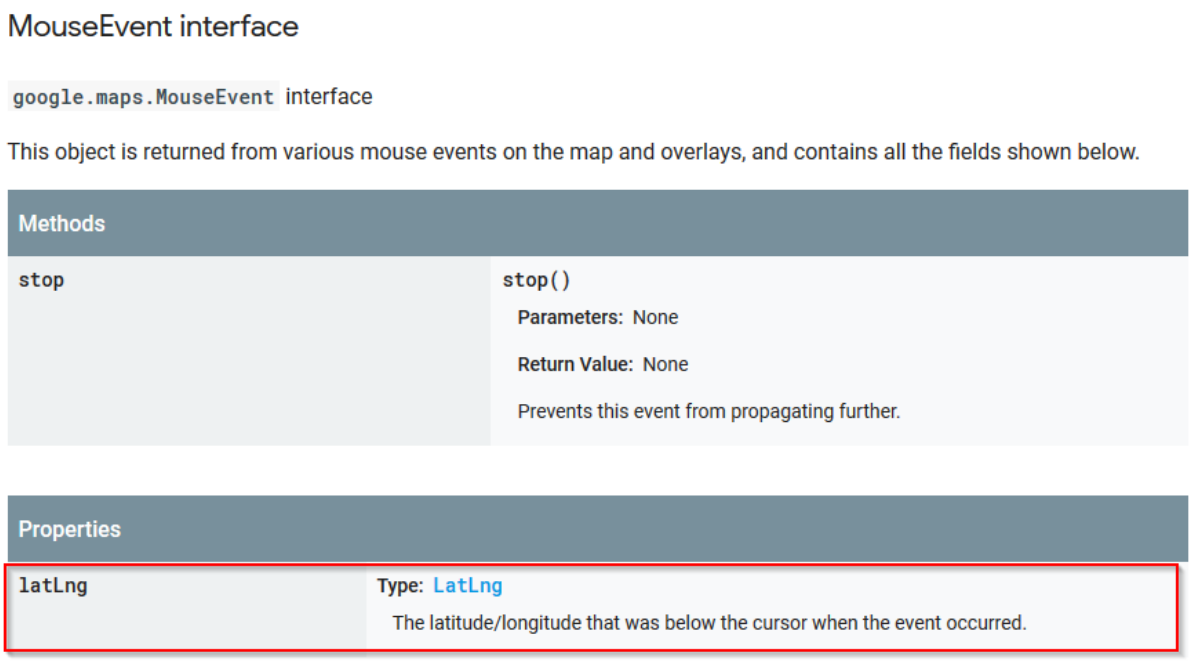
```
var mapOptions = { center: new google.maps.LatLng(15.5, 10.5), zoom: 3 };
```

Then, after the code that creates the map, add a new event listener that handles any **click** event on the **map** object. Use the **google.maps.event.addListener()** method to achieve this:

```
google.maps.event.addListener(map, 'click', function(event) {
    // do something
});
```

Every time a user clicks on the map, the anonymous function supplied as the third parameter to **addListener** executes. This function receives an input parameter that we have called **event**, which provides our handler function with details of the event that occurred. The object that gets passed depends on the type of event that occurred. The map's **click** event results in an object of type **MouseEvent**. This event object has a property called **latlng**, which contains details of the location at which the mouse click occurs, as can be seen in the reference documentation (see Figure 36).

www.dbooks.org



MouseEvent interface

google.maps.MouseEvent interface

This object is returned from various mouse events on the map and overlays, and contains all the fields shown below.

Methods	
stop	stop() **Parameters:** None **Return Value:** None Prevents this event from propagating further.

Properties	
latLng	**Type:** LatLng The latitude/longitude that was below the cursor when the event occurred.

Figure 36: The MouseEvent object.

To display this information to the user, all we need to do is create an info window that references this property on the **`MouseEvent`** object.

Putting it all together, we end up with the solution shown in Code Listing 28.

Code Listing 28: Positioning and populating the info window

```
function initMap() {
    var mapOptions = { center: new google.maps.LatLng(15.5, 10.5), zoom: 3
};
    var map;
    var infowindow;
    map = new google.maps.Map(document.getElementById('map'), mapOptions);

    google.maps.event.addListener(map, "click", function (event) {
        if (infowindow != null) {
            infowindow.close();
        }

        var strContent = "<b>Coordinates : </b>"
        strContent += "<br><b>Latitude : </b>" + event.latLng.lat();
        strContent += "<br><b>Longitude: </b>" + event.latLng.lng();

        infowindow = new google.maps.InfoWindow({
            content: strContent,
            position: event.latLng
        });
```

```
        infowindow.open(map);
    });
}
```

Note how we're using the **`MouseEvent`** object's **`latLng`** property to populate the info window (via its **`content`** property) and anchor it to the map (using its **`position`** property). When you open the page in a browser and click the map, you should see the latitude and longitude of that location in the info window, as shown in Figure 37.



Figure 37: Displaying the coordinates.

Syncing two maps

Putting two maps together on a page and keeping them synchronized might seem like a contrived example, but it could have real-world applications. For example, you might want to create an application that has a main map and an inset "overview" map that shows the area of the main map at a smaller scale. In fact, the Google Maps API used to have such a control but, for reasons of their own, they dropped it in September 2015. So, if you want such a feature, use this example as a basis for your own!

Displaying the maps

Start by copying Ch2_FirstMap.html to create a new file called Ch5_Sync2Maps.html.

www.dbooks.org

Then, change the page HTML to create two map ** ** elements and add CSS to the **<style>** tag to tell the page to display them side-by-side. Create a variable and an initialization function for each map in a **<script>** block, and a main function called **initMaps** that calls them both when the page has loaded. Also create a global variable to store the **MapOptions** object that both maps will share. Ensure that you change the name of the callback function in the **<script>** block that loads the Google Maps API to **initMaps**. Refer to Code Listing 29 to complete these steps.

Code Listing 29: Making space for two maps

```
<!DOCTYPE html>
<html>

<head>
    <title>Ch5 Sync 2 Maps</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        .map {
            width: 600px;
            height: 600px;
            background-color: red;
            margin: 5px;
            float: left;
        }

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

<body>
    <div id="mapOne" class="map"></div>
    <div id="mapTwo" class="map"></div>
    <script>
        var map1, map2;
        var mapOptions;

        function initMaps() {
           initMap1();
           initMap2();
        }
```

```
        function initMap1() {

        }

        function initMap2() {

        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]&callback=initMaps" async defer></script>
</body>

</html>
```

If you have coded the page exactly as shown in Code Listing 29, then it looks like Figure 38 when you display it in a browser.



Figure 38: The HTML page for displaying two maps.

We have set the background color to red just so we can verify that the ** ** elements appear on the page correctly.

Now, write the code that initializes the two maps. In Code Listing 30, I have set the extent to the extent of the UK, but you can focus your maps on whatever part of the world you like.

Tip: Use the Extent Finder application to quickly create the `MapOptions` object for your chosen extent.

www.dbooks.org

Code Listing 30: Initializing both maps

```
<script>
    var map1, map2;
    var mapOptions;

    function initMaps() {

        mapOptions = {
            center: new google.maps.LatLng(54.584455, -2.503702),
            zoom: 6,
            mapTypeId: google.maps.MapTypeId.TERRAIN
        };

        initMap1();
        initMap2();
    }

    function initMap1() {
        map1 = new google.maps.Map(
                            document.getElementById("mapOne"), mapOptions);
    }

    function initMap2() {
        map2 = new google.maps.Map(
                            document.getElementById("mapTwo"), mapOptions);
    }
</script>
```

Check that both maps are loaded and display the same details before proceeding, as shown in Figure 39.

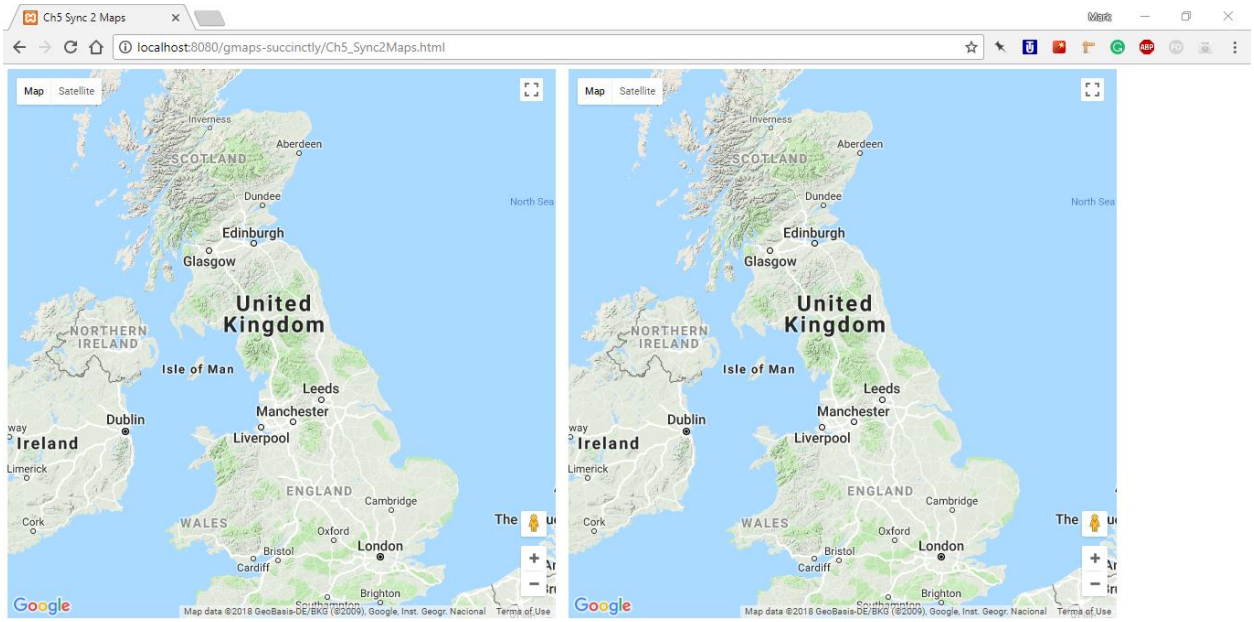


Figure 39: Two identical maps.

Handling the map events

Our first attempt at this application will let the user pan and move the left-hand map (**map1**), and the right-hand map (**map2**) will remain in sync. Now we must consider which map events to handle so that we can achieve this goal.

When we specify an initial extent for the map, we need a center point and zoom level, so it makes sense that if either of these change in **map1**, then the map extent has changed and we need to update **map2**.

Looking at the API reference documentation for the **google.maps.Map** class, we can see that it raises events for both center and zoom changes, called **center_changed** and **zoom_changed**, respectively. So we need to register handlers for each of these events in **map1**. When called, the handlers will change the relevant **center** or **zoom** properties on the other map to keep both maps in sync.

Create these two event handlers for **map1** **center_changed** and **zoom_changed** events in the initialization function for **map2**, as shown in Code Listing 31.

Code Listing 31: The center_changed and zoom_changed event handlers

```
function initMap2() {
    map2 = new google.maps.Map(
                document.getElementById("mapTwo"), mapOptions);

    google.maps.event.addListener(map1, 'center_changed', function () {
        map2.setCenter(map1.getCenter());
```


www.dbooks.org

```
    });
    google.maps.event.addListener(map1, 'zoom_changed', function () {
        map2.setZoom(map1.getZoom());
    });
}
```

Open the page in your browser and ensure that when you pan and zoom in the left-hand map, the right-hand map updates accordingly.

Bi-directional sync

What would it take to make the sync work both ways? Well, on the face of it, it might seem like adding **`center_changed`** and **`zoom_changed`** handlers to **`map1`** would do the trick, but in fact that would cause further problems.

Consider what would happen when **`map1`**'s extent changes. It would raise an **`extent_changed`** event that would be handled by **`map2`**. But in that event handler, **`map2`** would update its extent, which in turn would cause **`map1`**'s **`extent_changed`** handler to fire, and so on. In fact, the application would end up in an infinite loop that would eventually crash the browser. Not good.

A quick hack to get around this problem would be to introduce a short timeout before each map updates, as follows:

```
google.maps.event.addListener(map2, 'zoom_changed', function() {
  setTimeout(function() {
    map1.setZoom(map2.getZoom());
  }, 10);
});
```

But a much better way of keeping two maps in sync is to use the Google Maps API own **`bindTo`** function. The **`bindTo`** function syncs two properties on different objects as follows:

```
map1.bindTo('center', map2, 'center');
map1.bindTo('zoom', map2, 'zoom');
```

Much better! So why didn't I mention it in the first place? Well, because in this chapter we are talking about events, and learning about some of the things that can go wrong with event-driven programming will help you when you encounter similar problems in the wild!

Events on non-API objects

The **`Map`** is not the only Google Maps API class with events by a long shot. Just about all classes in the API raise events in response to either a state change or the user doing something with them. And of course, other page elements (like buttons, links, and so on) that are not part of the Google Maps API also raise events.

You can create events for these other DOM elements by using standard JavaScript techniques. For example, the code in Code Listing 32 handles a button being clicked and displays a message box to the user.

Code Listing 32: Event handling using vanilla JavaScript

```
<!DOCTYPE html>
<html>

<body>

    <button id="btn">Click me</p>

        <script>
            document.getElementById("btn").onclick = myFunction;

            function myFunction() {
                alert("YOU CLICKED ME!");
            }
        </script>

</body>

</html>
```

However, the Google Maps API provides its own convenience method so that you can listen to these events. It's called **`google.maps.event.addDomListener()`**, and you can see it in action in Code Listing 33.

Code Listing 33: Using google,maps.event.addDomListener

```
<!DOCTYPE html>
<html>

<head>
    <title>DomListener</title>
</head>

<body>
    <button id="btn">Click Me</button>
    <div id="map"></div>
    <script>
        function init() {
            google.maps.event.addDomListener(btn, "click",
                function (event) {
                    alert("YOU CLICKED ME!");
                }
```


www.dbooks.org

```
            );
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]&callback=init" async defer></script>
</body>

</html>
```

Cancelling edits

In this example, we are going to combine the use of **google.maps.event.addListener()** and **google.maps.event.addListener()** to create a very simple editing application.

You may recall in Chapter 4 that we mentioned a property on the **google.maps.Marker** class called **draggable** that, if enabled, lets the user move the marker to a different location on the map. In our example, we are going to listen to the marker's **dragstart** and **dragend** properties to report the initial and new location of the marker, and provide a cancel button that will reset the marker's position if the user changes their mind.

Copy the Ch2_SimpleMarker.html page to create a new file called Ch5_MarkerEdit.html, and change the code to that shown in Code Listing 34.

Code Listing 34: The marker editing code

```
<!DOCTYPE html>
<html>

<head>
    <title>Ch5 Marker Edit</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        #map {
            height: 100%;
        }

        #pos {
            display: none;
        }

        html,
        body {
            height: 100%;
            margin: 0;
```

```
            padding: 0;
        }
    </style>
</head>

<body>
    <div id="pos">
        <button id="btnCancel">Cancel</button>
        <p id="coords"></p>
    </div>

    <div id="map"></div>
    <script>
        function initMap() {
            var mapOptions = {
                center: new google.maps.LatLng(51.503, -0.135),
                zoom: 12
            };
            var map;
            map = new google.maps.Map(document.getElementById('map'),
                                                              mapOptions);

            var currentMarkerPos;

            var marker = new google.maps.Marker({
                position: new google.maps.LatLng(51.510357, -0.116773),
                map: map,
                draggable: true,
                title: "Big Ben"
            });

            google.maps.event.addListener(marker, "dragstart",
                function (event) {
                    currentMarkerPos = event.latLng;
                    document.getElementById("pos").style.display = "none";
                }
            );

            google.maps.event.addListener(marker, "dragend",
                function (event) {
                    var str = "<b>Initial Position:</b> lat: " +
                                                    event.latLng.lat();
                    str += ", lng: " + event.latLng.lng();
                    str += "<br><b>New Position:</b> lat: " +
                                                    event.latLng.lat();
                    str += ", lng: " + event.latLng.lng();
```


www.dbooks.org

```
                    document.getElementById("coords").innerHTML = str;
                    document.getElementById("pos").style.display =
"block";
                }
            );

            google.maps.event.addDomListener(btnCancel, "click",
                function (event) {
                    marker.setPosition(currentMarkerPos);
                    document.getElementById("pos").style.display = "none";
                }
            );
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]&callback=initMap" async defer></script>
</body>

</html>
```

Let's consider each of the changes we made.

First, we've added a new ** ** on the page with an **id** of **pos**. Within that ** ** are a button (**btnCancel**) and a paragraph element (**coords**). By wrapping those two elements within a ** **, we can deal with them collectively. In the **<style>** tag, we've changed the CSS display property of the ** ** to **none**. This hides the ** ** and stops it from taking up page space when the page first loads.

Then, in our **<script>** block, we have added a new variable called **currentMarkerPos**. When the user has indicated that they want to move the marker to a new location, we will store the current location first, in case they want to cancel the move.

After that, we have three event handlers:

- **dragstart**: This is fired when the user first starts to drag the marker and supplies a **MouseEvent** object to the handler function.
- **dragend**: This is fired when the user has finished dragging the marker and supplies a **MouseEvent** object to the handler function.
- **btnCancel.click**: This is a DOM event that fires when the user clicks the Cancel button.

The operations performed by each of the event handlers are as follows:

- In **dragstart**, we capture the current location of the marker in the **currentMarkerPos** variable.

- In **`dragend`**, we capture the final location of the marker and display the **`pos <div>`** to present the user with the initial and new map coordinates, and a button that allows them to cancel.
- In **`btnCancel.click`**, we cancel the move operation by setting the marker's position to the original position stored in **`currentMarkerPos`**.

When the page loads, it should look like the screenshot in Figure 40. When the user has finished moving the marker, they should be presented with the initial and new coordinates and the option to cancel, as shown in Figure 41. If they click the Cancel button, the marker is restored to its original location (Figure 42).

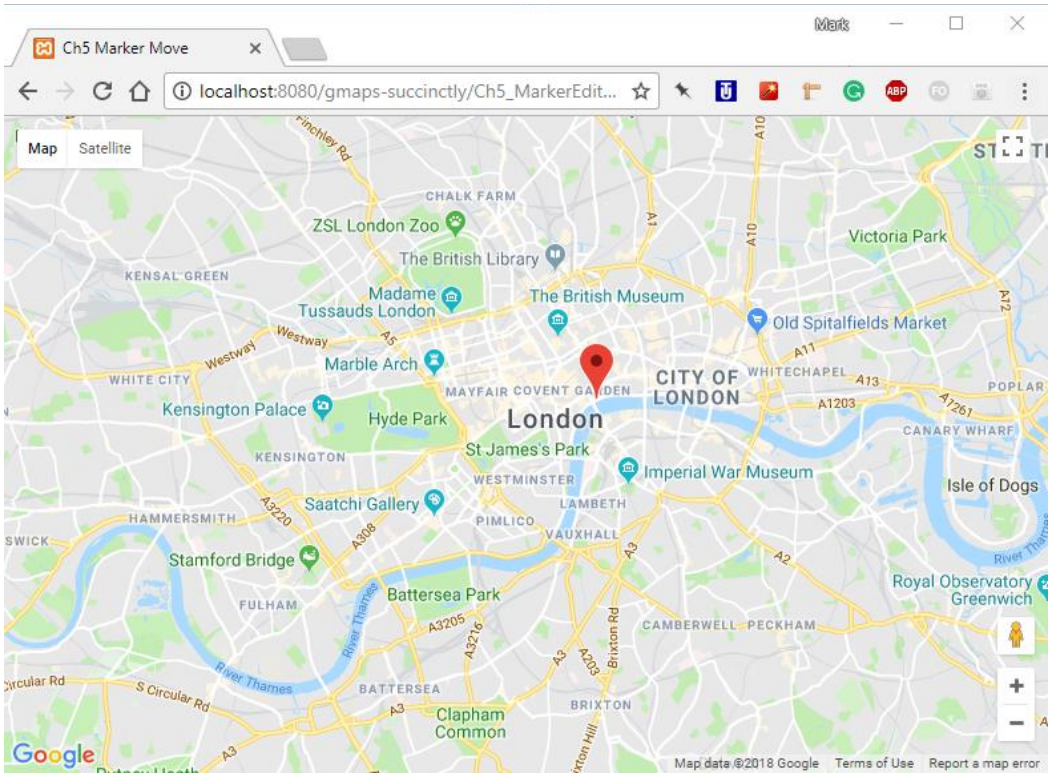


Figure 40: Initial position of the marker.

www.dbooks.org

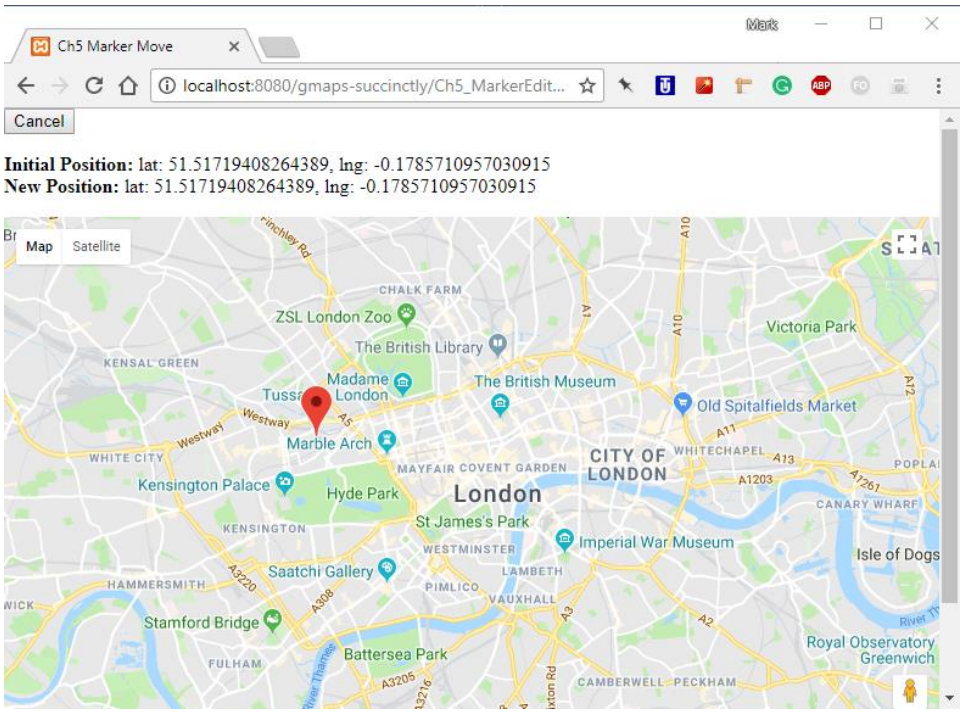


Figure 41: New position of the marker with option to cancel.

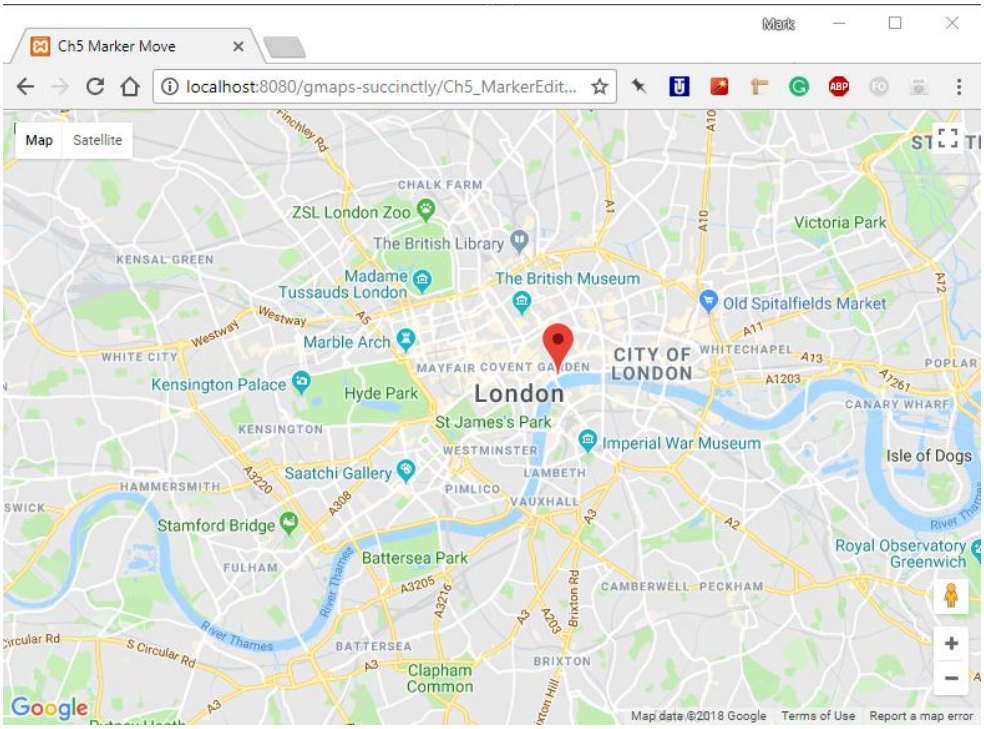


Figure 42: Marker restored to its original position after user cancels.

Chapter 6 Geocoding

Geocoding is the process of turning an address into geographic coordinates that you can use to pinpoint the location of that address on a map. *Reverse geocoding* is the process in reverse: that is, starting with a map coordinate and trying to work out which physical address is at, or close to, that location.

Note that "address" in this context can be either a full address or just partial information, such as a ZIP code, building name, or so on. The quality of the match depends on the quality of the underlying data, and Google has some of the best data available!

In this chapter, we are going to build two applications. The first will enable users to search for addresses and, if matched, place a marker at that location. The second requires them to click on the map, and it will report the nearest address to where they clicked.

In order to use geocoding, you must enable the Geocoding API using the same process as you used to enable the Google Maps JavaScript API. See Chapter 2 for details.

Using the Geocoder

Before we start building our first application, let's first talk about the class that makes it all happen: the **google.maps.Geocoder** class. Like many of the classes in the Google Maps API, the Geocoder is easy to use for simple use cases, but can become more complex by configuring options if your needs are more sophisticated.

To start geocoding, you first need an instance of the **geocoder** class:

```
geocoder = new google.maps.Geocoder();
```

Then you call the object's **geocode()** method, passing it a **GeocodeRequest** object that defines the nature of your request as the first parameter. The very simplest request you can make just sets the address property of the **GeocodeRequest** object with the address you are looking for.

In the second parameter, you provide a *callback* function. This is the function that will execute when the results of the geocoding operation are available.

```
geocoder.geocode( { address: "Big Ben, London" },
   function (results, status) {
         // process results
   }
);
```

This callback function receives two parameters from the geocoding operation: an array of **GeocoderResult** objects which, as you have probably guessed, contains the results, and a **GeocoderStatus** constant, which tells you whether the operation was successful or encountered some type of error.

www.dbooks.org

The GeocoderRequest class

Before we talk about how we unpack the results of the geocoding option, let's look at some of the other properties you can set on the **`GeocoderRequest`** object to configure your search. You can see these summarized in Table 2.

Table 2: GeocoderRequest object properties

Property	Data Type	Description
`address`*	**`string`**	The address to geocode.
`bounds`*	**`LatLngBounds`**	An area to search within.
`componentRestrictions`	**`GeocoderComponentRestrictions`**	An object with properties that you can use to restrict the area for the search (**`route`**, **`locality`**, **`administrativeArea`**, **`postalCode`**, **`country`**).
`location`	**`LatLng`**	A point on the map. This is how the Geocoder does reverse geocoding.
`placeId`*	**`string`**	An entry in the Google Places database. This is another reverse geocoding method.
`region`	**`string`**	Bias the search using the supplied country code.



Note: Only one of the `address`, `bounds`, and `placeId` properties can be set in a single instance of the `GeocoderRequest` object.

In this book, we'll only be using the **`address`** and **`location`** properties.

The GeocoderResult object

The results are returned to your callback function in an array of **`GeocoderResult`** objects. The first element in the array is the best match, with the API having decreasing confidence in any subsequent entries.

Each **`GeocoderResult`** object has the properties listed in Table 3.

Table 3: GeocoderResult object properties

Property	Data Type	Description
address_components	Array of **GeocoderAddressComponent**	Contains **long_name**, **short_name**, and **type** for each component of the address (route, locality, etc.).
formatted_address	**string**	Contains the formatted address data.
geometry	**GeocoderGeometry**	The geometry of the result for plotting on the map.
partial_match	**boolean**	**True** if this result was not an exact match.
place_id	**string**	The place ID for this location if it is in the Google Places database.
postcode_localities	Array of **string**	Breaks down postal codes with multiple localities.
types	Array of **string**	The types of each address component (route, locality, etc.) returned in the results.

Again, we'll only be using some of these properties in this chapter's examples, but be aware of the wealth of information being returned by the geocoding operation, some of which might be extremely useful in a future application.

The GeocoderStatus constants

Your callback also receives a value that tells you whether your geocoding operation was successful or, if not, what went wrong. The following are the more common status values you are likely to see:

- **OK**: The operation was successful.
- **ERROR**: There was a problem connecting to the service.
- **INVALID_REQUEST**: The **GeocoderRequest** object was invalid.
- **OVER_QUERY_LIMIT**: You have exceeded the number of requests allowed in the period.
- **REQUEST_DENIED**: The webpage is not allowed to use the geocoder.
- **UNKNOWN_ERROR**: A geocoding request could not be processed due to a server error. The request may succeed if you try again.
- **ZERO_RESULTS**: No result was found for this **GeocoderRequest** object.

www.dbooks.org

Geocoding an address

In this first example, we create a simple address field for the user to enter an address to search for, execute the geocoding operation, and place a marker at the best address candidate returned.

Before you start, you must enable the Geocoding API, which is not enabled by default. You do this in the Google Developer Console. Select the project, search for the **Geocoding API**, enable it, and then proceed with the following steps.

Begin by copying Ch2_FirstMap.html to create a new file called Ch6_Gecoding.html. Then, add the markup in Code Listing 35 to create an address text box and a button to submit its contents.

Code Listing 35: Creating the Geocoding UI

```
<body>
    <input id="txtAddress" type="text" size="60" placeholder="Enter your
Address" />
    <button id="btnSearch">Search</button>
    <div id="map"></div>
    <script>
        var map;

        function initMap() {
            var mapOptions = {
                center: new google.maps.LatLng(51.503, -0.135),
                zoom: 12
             };

            map = new google.maps.Map(document.getElementById('map'),
                                                              mapOptions);
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=[YOUR API
KEY]&callback=initMap" async
        defer></script>
</body>
```



Tip: Move the map variable outside of `initMap()` so that it can be easily accessed by other functions you will create for this example.

Create and instantiate the **`google.maps.Geocoder`** object, as shown in Code Listing 36.

Code Listing 36: Instantiating the Geocoder

```
<script>
```

```
    var map;
    var geocoder;

    function initMap() {

        var mapOptions = {

            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12
        };

        map = new google.maps.Map(document.getElementById('map'),
            mapOptions);

        var geocoder = new google.maps.Geocoder();
    }
</script>
```

Add an event listener for the **btnSearch** button. Because this is not a Google Maps API control, you will need to use either the standard JavaScript **addEventListener()** approach or the Google Maps API **addDomListener()**. In this example, instead of defining my event handler function inline, I have created a new function called **findAddress()**. This makes the code easier to read, as shown in Code Listing 37.

Code Listing 37: Responding to the user clicking Search button

```
<script>
    var map;
    var geocoder;

    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12
        };

        map = new google.maps.Map(document.getElementById('map'),
            mapOptions);

        geocoder = new google.maps.Geocoder();

        google.maps.event.addDomListener(btnSearch, "click", findAddress);
    }

    function findAddress() {

    }
```


www.dbooks.org

```
</script>
```

Within the **findAddress()** function:

1. Create a **GeocodeRequest** object called **geocodeRequest**. This has a single property called **address**, which contains the text within the **txtSearch** text box.
2. Execute the **Geocoder.geocode()** method, passing in the **geocodeRequest** as the first parameter, and a callback function that accepts two parameters, **results** and **success**, as the second parameter.
3. Within the callback function, check that the **status** is **OK**. If so, call a function called **showMarker()**, passing it the first element of the **results** array. We will create the **showMarker()** function next.

You can see the effect of these changes on the code in Code Listing 38.

Code Listing 38: Executing the geocode operation

```
<script>
    var map;
    var geocoder;

    function initMap() {
        var mapOptions = {
            center: new google.maps.LatLng(51.503, -0.135),
            zoom: 12
        };

        map = new google.maps.Map(document.getElementById('map'),
            mapOptions);

        geocoder = new google.maps.Geocoder();

        google.maps.event.addDomListener(btnSearch, "click", findAddress);
    }

    function findAddress() {
        var geocodeRequest = {
            address: document.getElementById("txtAddress").value
        }
        geocoder.geocode(geocodeRequest, function (results, status) {
            if (status == google.maps.GeocoderStatus.OK) {
                showMarker(results[0]);
            } else {
                alert('Could not gecode: ' + status);
            }
        });
    }
```

```
    function showMarker(result) {

    }
</script>
```

Implement the **showMarker()** function using the skills you learned in Chapter 4. Use the **geometry** property of the **GeocodeResult** object that **showMarker** receives to plot the marker on the map and create an info window for the marker to display the **formatted_address** property, as shown in Code Listing 39.

Code Listing 39: Displaying the results of the geocoding operation

```
function showMarker(result) {
    var marker = new google.maps.Marker({
        map: map,
        position: result.geometry.location
    });

    map.setCenter(result.geometry.location);
    map.setZoom(16);

    var strContent = '<b>Address: </b> ' + result.formatted_address;
    var infowindow = new google.maps.InfoWindow({
        content: strContent
    });

    google.maps.event.addListener(marker, 'click', function () {
        infowindow.open(map, marker);
    });
}
```

Open the page in a browser, type **Big Ben** into the search box, and then click **Search**. The geocoding operation executes and successfully identifies and zooms into the geographic coordinates of this famous London monument. If you click the marker, its full address displays, as shown in Figure 43.

www.dbooks.org

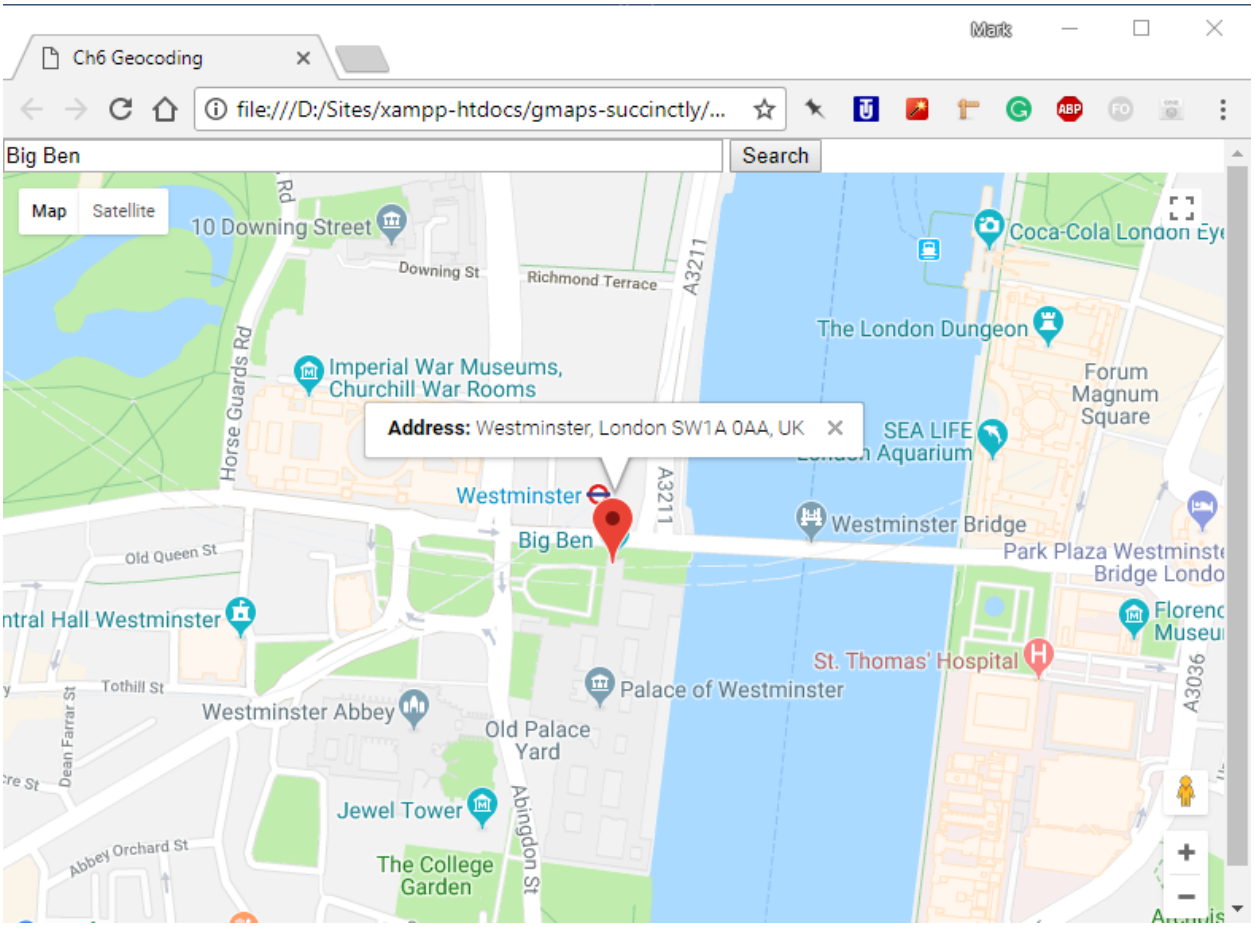


Figure 43: Geocoding Big Ben.

Try the application with the following London addresses, or set the extent to an area you know well and test it with some local addresses:

- 10, Downing Street, London
- Madame Tussaud's, London
- London, WC1
- 47 Fleet St, London EC4Y 1BJ

For extra credit, populate the info window with all the other matches from the geocoding operation. This is demonstrated in Code Listing 40.

Code Listing 40: Changing the solution to display all matches

```
function findAddress() {
    var geocodeRequest = {
        address: document.getElementById("txtAddress").value
    }
    geocoder.geocode(geocodeRequest, function (results, status) {
        if (status == google.maps.GeocoderStatus.OK) {
            showMarker(results)
```

```
        } else {
            alert('Could not gecode: ' + status);
        }
    });
}

function showMarker(results) {

    var bestMatch = results[0];

    var marker = new google.maps.Marker({
        map: map,
        position: bestMatch.geometry.location
    });

    map.setCenter(bestMatch.geometry.location);
    map.setZoom(16);

    var strContent = '<b>Address: </b> ' +
                                bestMatch.formatted_address + "<ul>";
    for (var i = 1; i < results.length; i++) {
        strContent += "<li>" + results[i].formatted_address + "</li>";
    };
    strContent += "</ul>";
    var infowindow = new google.maps.InfoWindow({
        content: strContent
    });

    google.maps.event.addListener(marker, 'click', function () {
        infowindow.open(map, marker);
    });
}
```

If you enter a search term with ambiguous results, you should see any other matches displayed in the info window, as shown in Figure 44.

www.dbooks.org

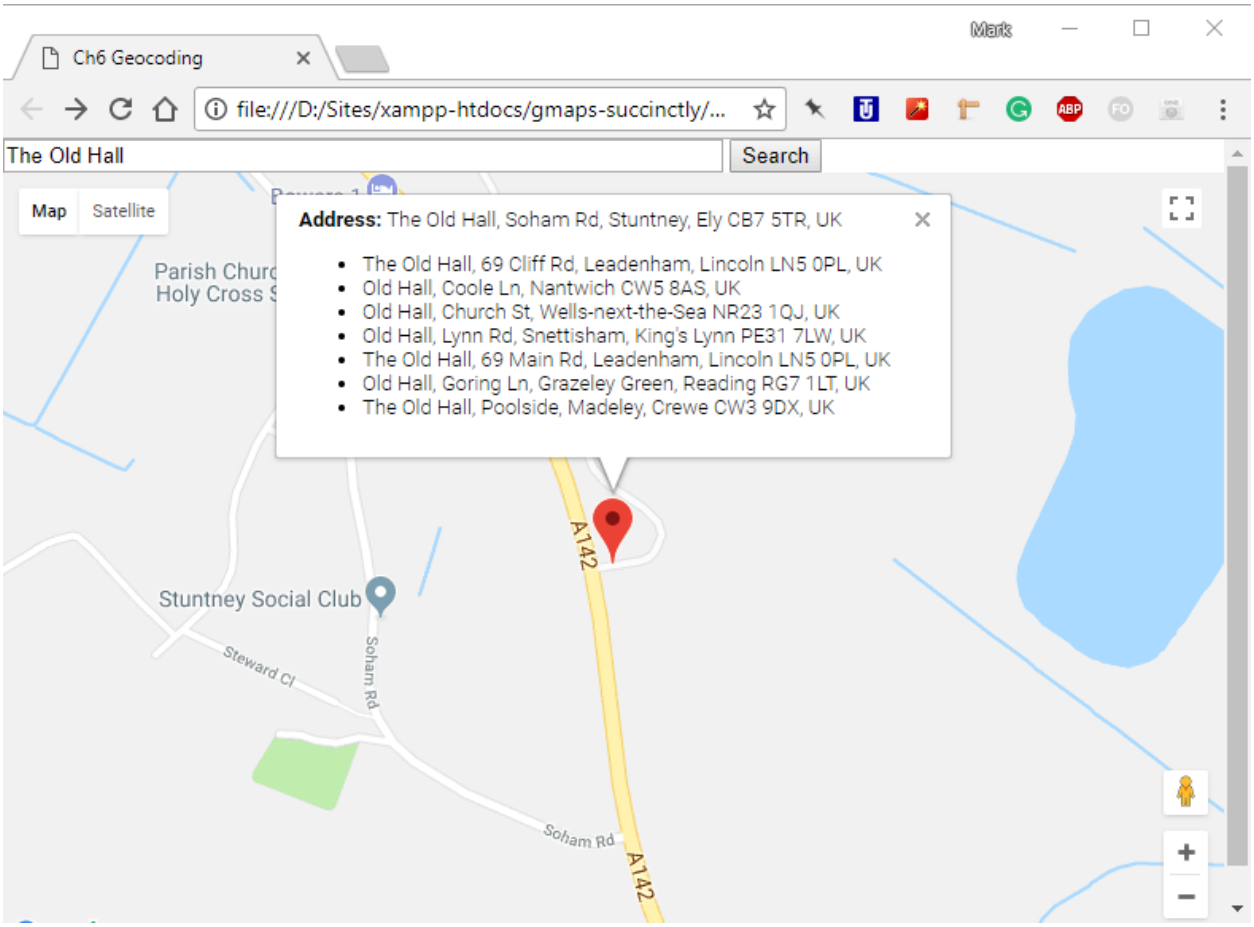


Figure 44: Listing the matches that missed the cut.

Reverse geocoding

In the previous example, we started with an address and ended up with a latitude and longitude. With reverse geocoding, you start with a latitude and longitude and (hopefully) end up with an address.

To demonstrate this, we will create an application that places a marker on the map wherever a user clicks. We'll reverse geocode that location to see what the nearest address is and display it in the marker's info window.

Copy Ch6_Geocoding.html to use as a starting point and save it as Ch6_ReverseGeocoding.html. Change the code so that it appears as shown in Code Listing 41.

Code Listing 41: Reverse geocoding

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Ch6 Reverse Geocoding</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        #map {
            height: 100%;
        }

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

<body>
    <div id="map"></div>
    <script>
        var geocoder;
        var map;

        function initMap() {
            var mapOptions = {
                center: new google.maps.LatLng(51.503, -0.135),
                zoom: 12
            };

            map = new google.maps.Map(document.getElementById('map'),
                mapOptions);

            geocoder = new google.maps.Geocoder();

            google.maps.event.addListener(map, "click", findAddress);
        }

        function findAddress(event) {
            var geocodeRequest = {
                location: event.latLng
            }
            geocoder.geocode(geocodeRequest, function (results, status) {
                if (status == google.maps.GeocoderStatus.OK) {
                    showMarker(results)
                } else {
```


www.dbooks.org

```
                    alert('Could not geocode: ' + status);
                }
            });
        }

        function showMarker(results) {

            var bestMatch = results[0];

            var marker = new google.maps.Marker({
                map: map,
                position: bestMatch.geometry.location
            });

            var strContent = '<b>Address: </b> ' +
                                    bestMatch.formatted_address + "<ul>";

            var infowindow = new google.maps.InfoWindow({
                content: strContent
            });

            google.maps.event.addListener(marker, 'click', function () {
                infowindow.open(map, marker);
            });
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=YOUR API
KEY&callback=initMap" async
        defer></script>
</body>

</html>
```

The main thing to notice about this solution is that, instead of using the **address** property of the **GeocodeRequest** object, it uses **location**, setting it to the point at which the user clicked on the map (as supplied in the **MouseEvent** object sent to the map's **click** event handler).

When you open the page in a browser and click a point on the map, a marker appears in that location and the closest address is displayed in the marker's info window when the marker is clicked. (See Figure 45.)

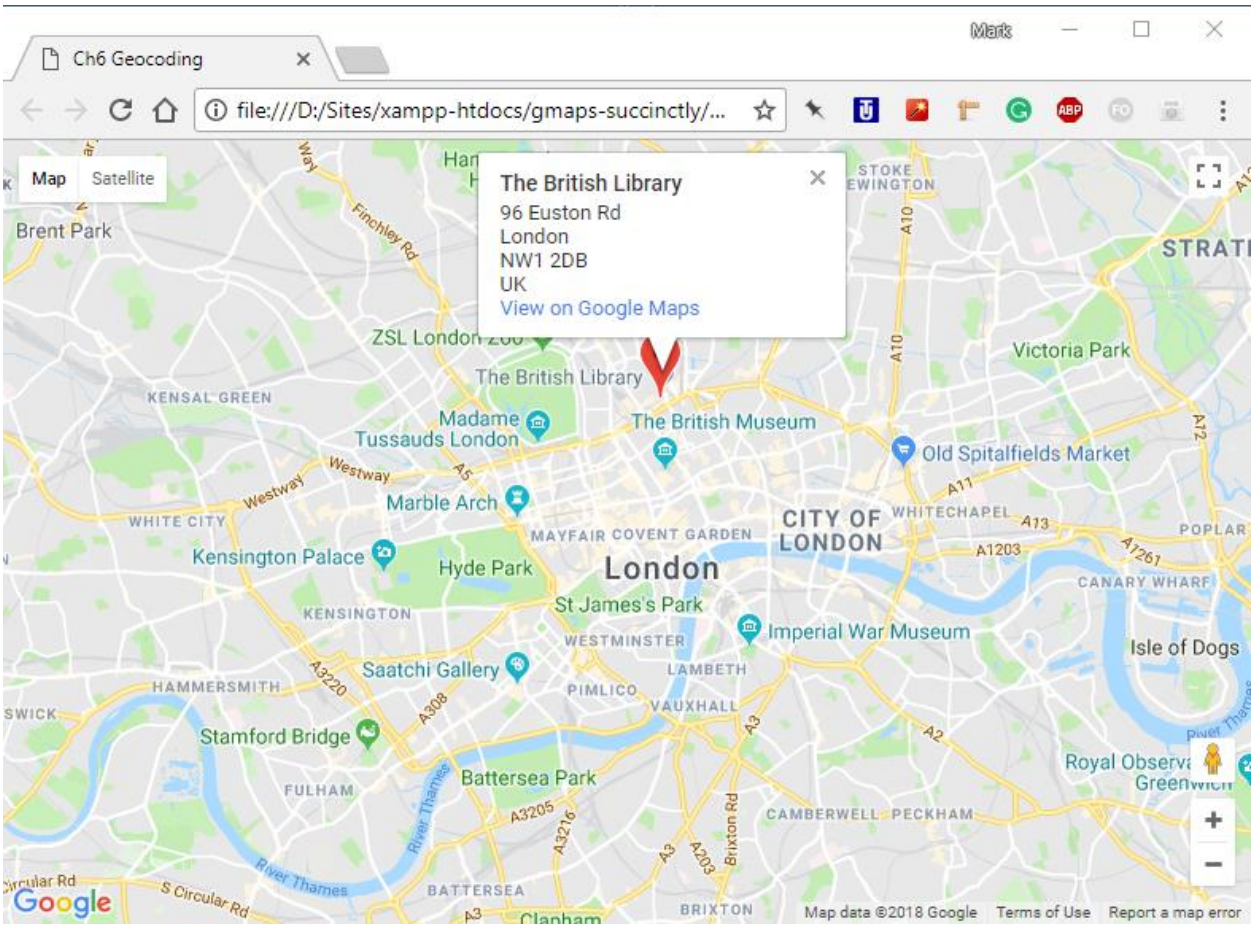


Figure 45: Reverse geocoding in action.

www.dbooks.org

Chapter 7 Getting Directions

Now that you know how to let your users find places on the map, why not show them how to get there? Sure, you could give them the address of your amazing new retail outlet/furry convention/LAN party and send them off to the Google Maps main site to get directions, but how much cooler would it be to do this entirely from within your own webpage, and be able to customize the experience? You can do all this and more with the Google Maps Directions API.

Enabling the Directions API

To work with directions in your Google Maps JavaScript API applications, you must enable the Directions API in the Developer Console. The Directions API works with other Google APIs, not just the Google Maps JavaScript API. To enable the Directions API, complete the following steps:

1. Visit http://console.developers.google.com.
2. Sign in with your Google account user name and password.
3. On the **Dashboard**, check that the project you created in Chapter 2 is selected. If not, select it from the drop-down list.

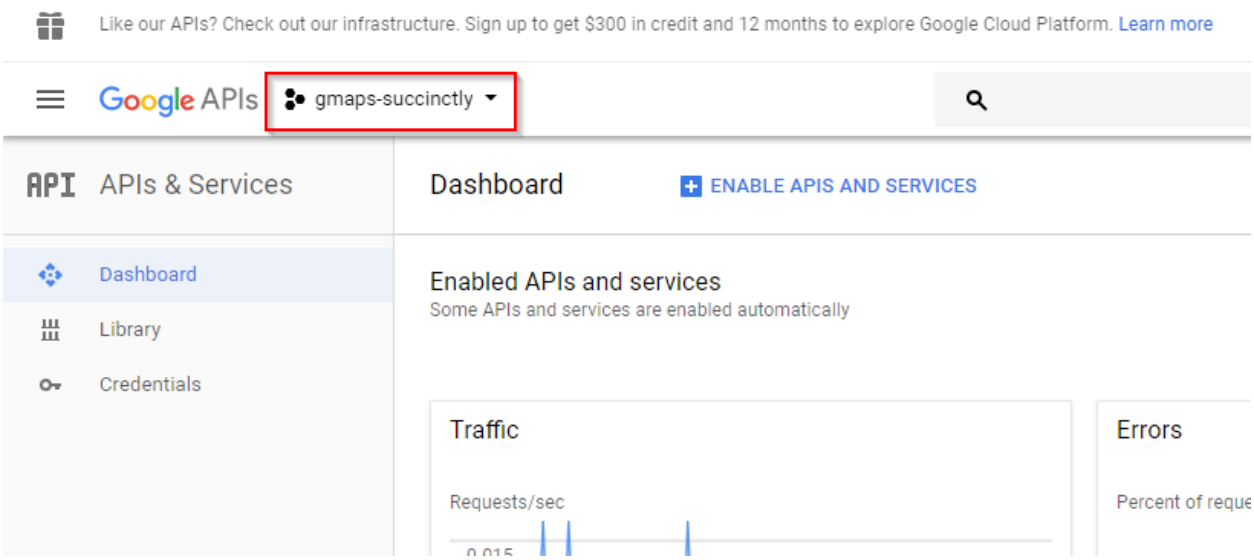


Figure 46

4. In the list of APIs shown at the bottom of the Dashboard, check if the Directions API is enabled. If it appears in the list already (item 1 in the screenshot), then you can ignore the remaining steps. Otherwise, click **Enable APIs and Services** (item 2 in the screenshot).

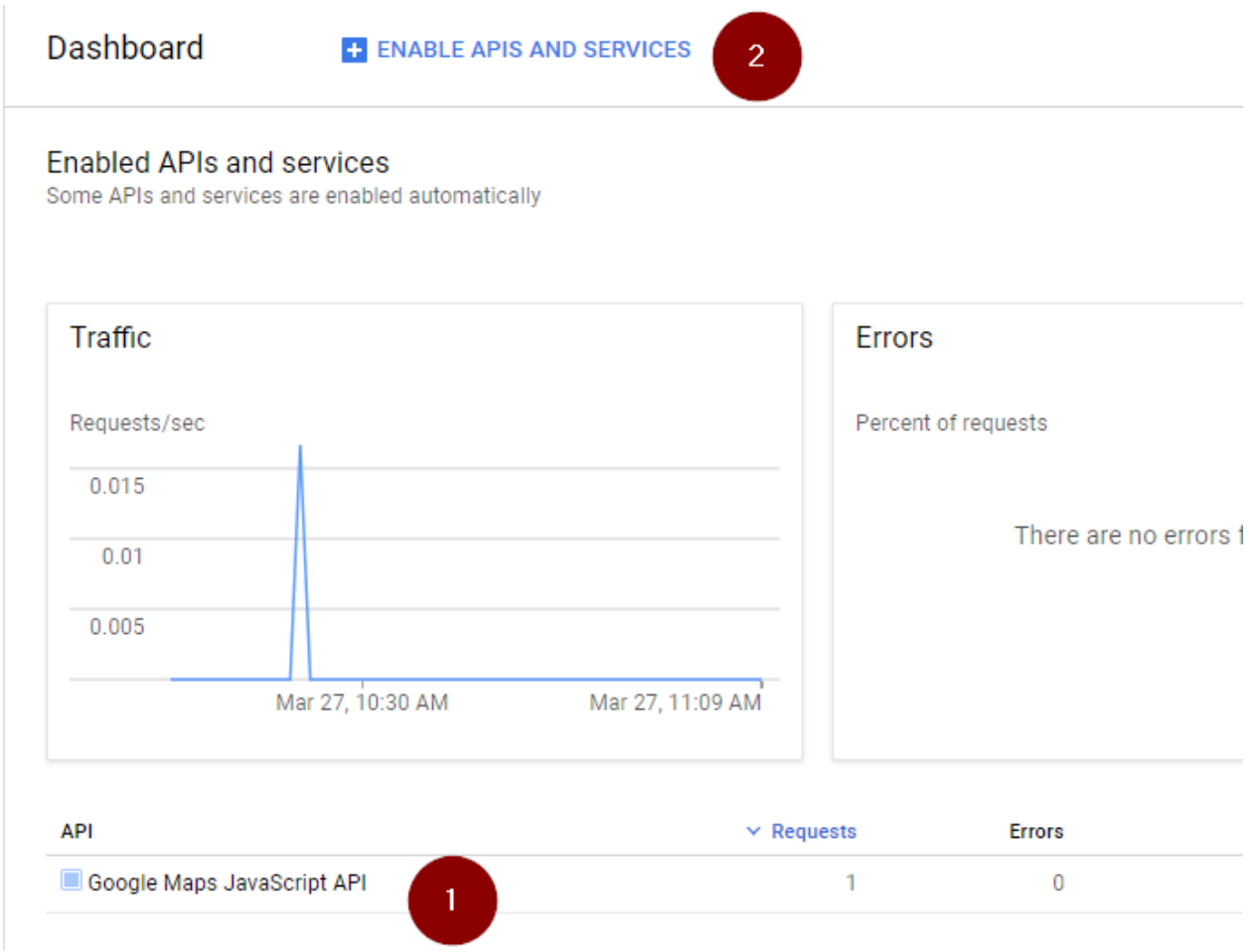
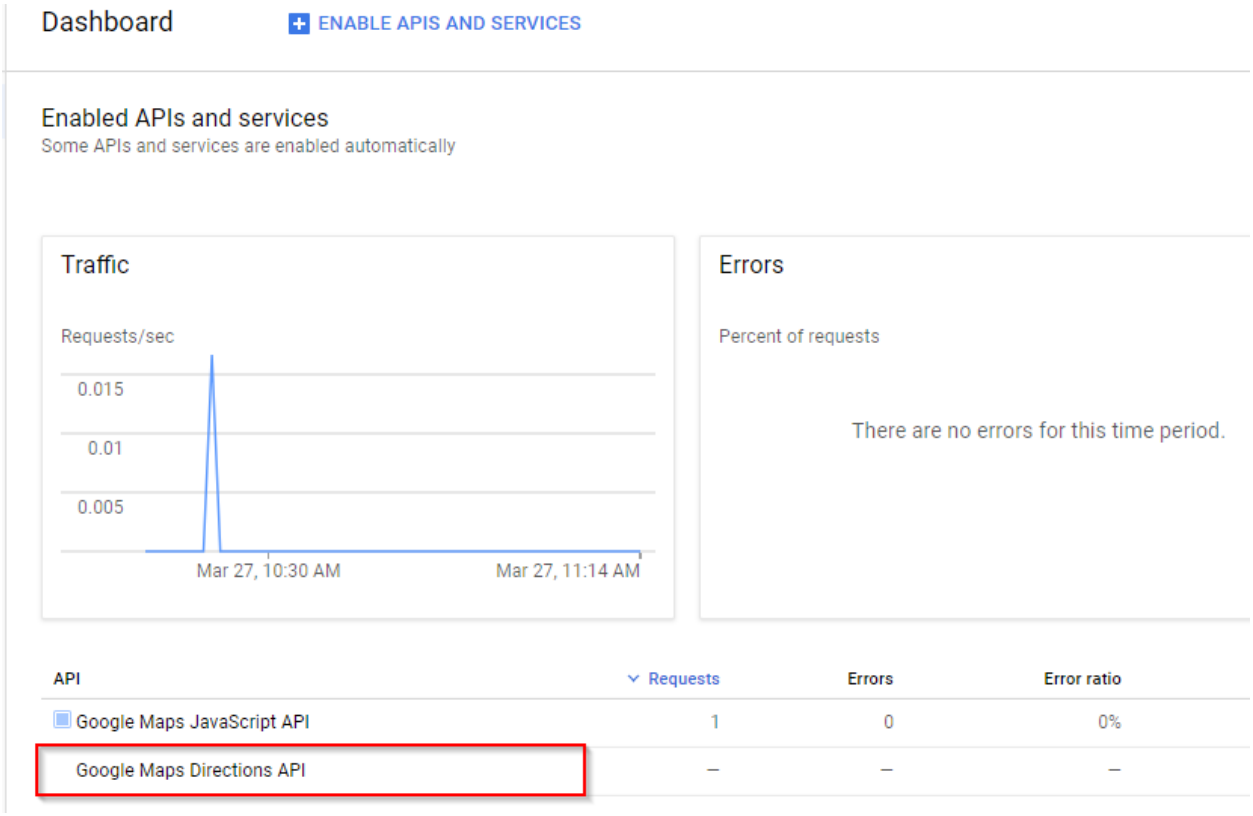


Figure 47

5. In the API Library, search for **Google Maps Directions API**. Click the Google Maps Directions API tile that appears, then click **Enable**. When the Directions API summary page appears, click the Dashboard link and verify that the Directions API is included in the list of enabled APIs at the bottom of the page.

www.dbooks.org



You are now ready to use the API.

> ***Note: Like the Google Maps JavaScript API, the Directions API is only free for a limited number of requests (see usage limits here).***

Creating the page

In this example, we are going to create an application that allows users to select an origin and destination, geocodes them both, and then displays the route on the map and turn-by-turn driving directions in another panel on the page. In building this application, we will bring together a number of techniques that we learned in earlier chapters.

> ***Note: Strictly speaking, you don't have to geocode the addresses before calculating the route if your geocoding requirements are as straightforward as they are in this example. The `DirectionsService` class can do this for you. However, doing the geocoding yourself does serve to teach a useful lesson about the asynchronous nature of JavaScript, which makes you a better Google Maps API developer!***

We'll start with the two-pane layout we used in Chapter 5 for the "Sync Two Maps" example. Copy the Ch5_Sync2Maps.html file to create a new file called Ch7_Directions.html.

Make the following changes to the page:

1. Rename the **map1** and **map2** ** **s to **map** and **directions**, respectively. Change the **class** attribute to **panel** and make the necessary adjustments to the CSS.
2. Remove the red background from both ** **s.
3. Remove all the map initialization and other code from the first **<script>** block and replace it with the **initMap()** function from Ch2_FirstMap.html.
4. Move the declaration of the **map** variable to the top of the **<script>** block, outside **initMap()**.
5. Change the callback function in the bootstrap URL from **initMaps** to **initMap**.
6. Add two text boxes at the top of the page called **txtOrigin** and **txtDestination**, and a button called **btnSubmit**.

As a result of making these changes, the initial code resembles that shown in Code Listing 42.

Code Listing 42: Initial page layout

```
<!DOCTYPE html>
<html>

<head>
    <title>Ch6 Directions</title>
    <meta name="viewport" content="initial-scale=1.0">
    <meta charset="utf-8">
    <style>
        .panel {
            width: 600px;
            height: 600px;
            margin: 5px;
            float: left;
        }

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

<body>
    <div>
        <label for="txtOrigin">Origin: </label>
        <textarea id="txtOrigin" placeholder="Enter origin"></textarea>
        <label for="txtDestination">Destination: </label>
        <textarea id="txtDestination" placeholder="Enter
origin"></textarea>
        <button id="btnSubmit">Submit</button>
    </div>
```


www.dbooks.org

```
    <div id="map" class="panel"></div>
    <div id="directions" class="panel"></div>
    <script>
        var map;

        function initMap() {
            var mapOptions = {
                center: new google.maps.LatLng(51.503, -0.135),
                zoom: 12
            };
            map = new google.maps.Map(document.getElementById('map'),

mapOptions);
        }
    </script>
    <!-- Reference the API -->
    <script src="https://maps.googleapis.com/maps/api/js?key=YOUR API
KEY&callback=initMap" async
        defer></script>
</body>

</html>
```

When you open the page in a browser, it should appear as shown in Figure 46.

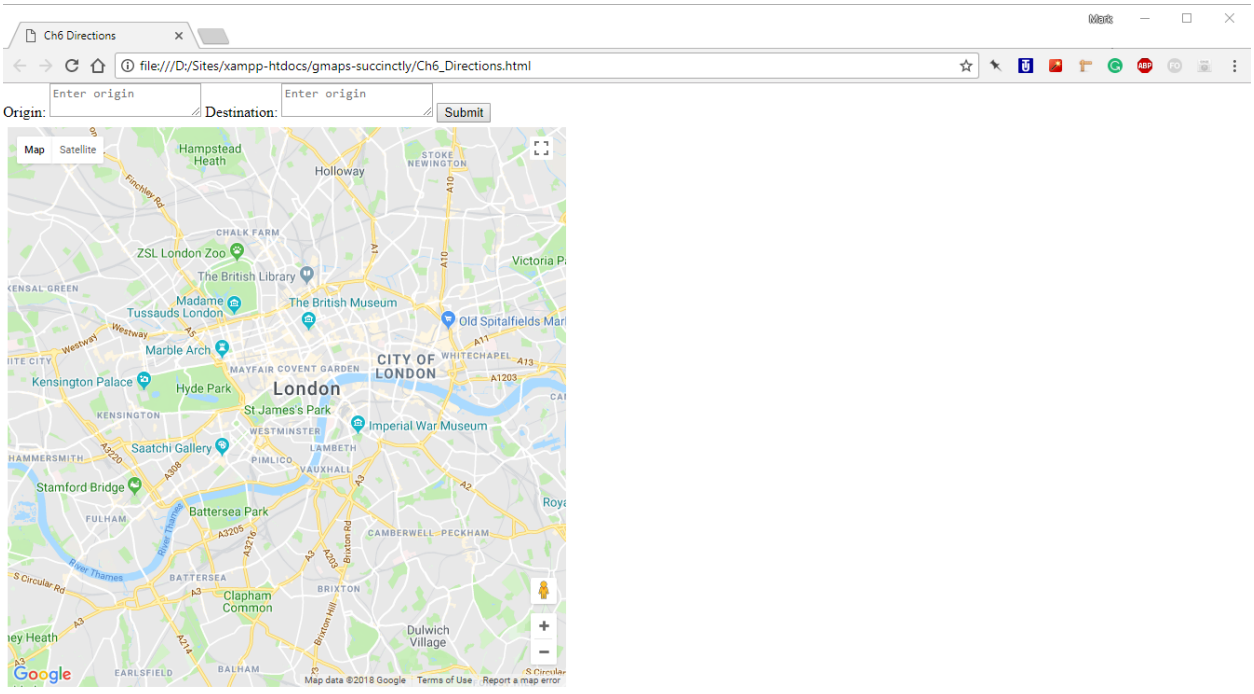


Figure 48: Initial page layout

Geocoding the start and end points

The next thing we need to do is geocode the origin and destination addresses that the user provides. We know how to do this from our explorations in Chapter 6, but there is added complexity in this situation.

We need to ensure that both addresses are correctly geocoded before we can start to calculate the route between them, but because of the asynchronous nature of JavaScript, we can't just do something like the following:

```
geocoder.geocode({address: startAddress, function(results, status) {
   origin = results[0];
}
geocoder.geocode({address: startAddress, function(results, status) {
   destination = results [0];
}
getDirections(origin, destination);
```

The problem is that JavaScript won't wait for the first call to **`Geocoder.geocode()`** before proceeding to the second, and won't wait for the second call to **`Geocoder.geocode()`** before calling **`getDirections()`**. In fact, it's very likely that **`getDirections()`** will be called before the **`origin`** or **`destination`** variables are assigned values, and therefore, it will fail.

What we need is some way of ensuring that both addresses are successfully geocoded before we attempt to calculate the route. Thankfully, recent versions of JavaScript include *promises*, and this construct is perfect for scenarios like this.

The best (and simplest) definition of a promise that I have seen is this:

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved.

When called, the **`Promise`** object attempts some sort of operation and, if it is successful, calls **`resolve()`** to pass back the results. If it fails, it calls **`reject()`**, usually passing an error back to the caller.

Code Listing 43 shows a very simple implementation of a promise. This one checks the user's credit rating, and if it's good, grants the loan. If not, it returns an error telling you that the loan application failed. Code Listing 44 shows how we might consume this promise.

Code Listing 43: Example Promise implementation

```
var goodCreditRating = false;

// Promise
var applyForALoan = new Promise(
    function (resolve, reject) {
        if (goodCreditRating) {
            var loan = {
                amount: 10000,
```


www.dbooks.org

```
                term: 60
            };
            resolve(loan); // fulfilled
        } else {
            var reason = new Error('Credit Rating Poor');
            reject(reason); // reject
        }

    }
);
```

Code Listing 44: Consuming the Promise

```
var begTheBank = function () {
    applyForALoan
        .then(function (fulfilled) {
            // loan application successful, here are the details.
            console.log(fulfilled);
            // outputs { amount: 10000, term: 60 }
        })
        .catch(function (error) {
            // loan application failed, for this reason.
            console.log(error.message);
            // outputs 'Credit Rating Poor'.
        });
};

begTheBank();
```

Because this is a promise, we can call **`.then()`** on the function that implements the **`Promise`** object to execute some code if the **`Promise`** succeeds (resolves), and **`.catch()`** if it fails (rejects). That's a very brief overview of JavaScript promises, and is just what we need to make our application work. However, there is a lot more to promises than this, and I suggest you visit the Mozilla Developer Network to find out more about them.

Let's create a function called **`geocodeAddresses()`** that implements a promise. This function geocodes the supplied start and ending addresses and creates a marker for each. You can see the code for this in Code Listing 45. Note that the **`Promise`** object does not call **`resolve()`** until the second address has been successfully geocoded.

Code Listing 45: The geocodeAddresses() function

```
function geocodeAddresses(origin, destination) {
    return new Promise(function (resolve, reject) {

        var geocodedOrigin, geocodedDestination;
```

```
        geocoder.geocode({ 'address': origin },
            function (results, status) {
                if (status == google.maps.GeocoderStatus.OK) {
                    geocodedOrigin = results[0];
                    originMarker = new google.maps.Marker({
                        position: geocodedOrigin.geometry.location,
                        label: "O",
                        map: map
                    })
                } else {
                    reject(Error("Could not geocode origin"));
                }
            });
        geocoder.geocode({ 'address': destination },
            function (results, status) {
                if (status == google.maps.GeocoderStatus.OK) {
                    geocodedDestination = results[0];
                    destinationMarker = new google.maps.Marker({
                        position: geocodedDestination.geometry.location,
                        label: "D",
                        map: map
                    })
                    resolve({
                        origin: geocodedOrigin,
                        destination: geocodedDestination,
                    });
                } else {
                    reject(Error("Could not geocode destination"));
                }
            });
    });
}
```

Now we need an event handler for our Submit button that consumes this **`Promise`** and proceeds to calculate the route only when the **`Promise`** resolves—that is, when both origin and destination addresses have been successfully geocoded. See Code Listing 46.

Code Listing 46: Calling geocodeAddresses()

```
var map, geocoder;
var origin, destination;
var originMarker, destinationMarker;

function initMap() {
    var mapOptions = {
        center: new google.maps.LatLng(51.503, -0.135),
```


www.dbooks.org

```
        zoom: 12
    };
    map = new google.maps.Map(document.getElementById('map'), mapOptions);

    geocoder = new google.maps.Geocoder();


    origin = document.getElementById("txtOrigin").value;
    destination = document.getElementById("txtDestination").value


    google.maps.event.addDomListener(
        document.getElementById("btnSubmit"), "click", function () {
            geocodeAddresses(origin, destination).then(function (result) {
                findDirections(result);
            }, function (err) {
                alert(err);
            });
        });
}

function findDirections(journey) {
    // calculate the route.
}

function geocodeAddresses(origin, destination) {
    // ...
}
```

Open the page in a browser, and if everything is working properly, you should be able to enter values for the origin (try **Big Ben**) and destination (try **47 Fleet Street, London, EC4Y 1BJ**), and see markers appear on the map, as shown in Figure 47.

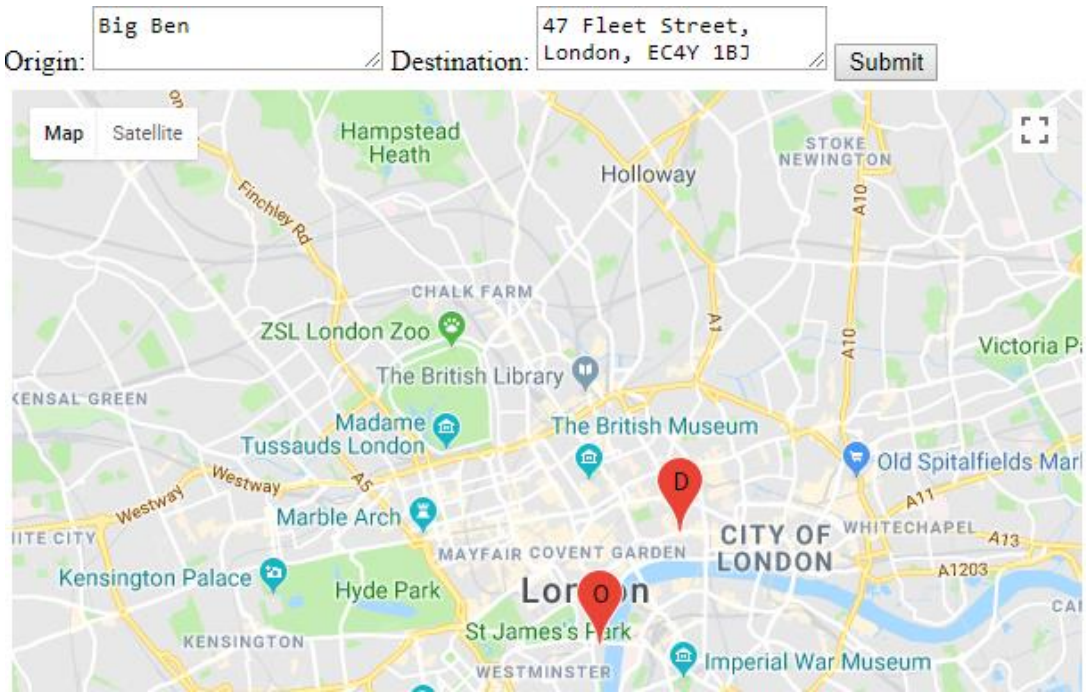


Figure 49: Successful geocoding of origin and destination addresses.

Now that you know where you're coming from and where you're headed, you're ready to start using the Directions API.

Calculating the route

The route calculations are performed by the **`google.maps.DirectionsService`** class. There is another class called **`google.maps.DirectionsRenderer`** that makes it really easy to display the resulting route. Create global variables for each and instantiate objects of both classes in the **`initMap()`** function, as shown in Code Listing 47. Also, remove the code that creates the markers in the **`geocodeAddresses()`** function, because the **`DirectionsRenderer`** will supply them for you, as well as providing the polyline that represents the route.

Code Listing 47: Creating the DirectionsService and DirectionsRenderer classes

```
var map, geocoder;
var origin, destination;
var originMarker, destinationMarker;
var directionsService, directionsRenderer;

function initMap() {
    var mapOptions = {
        center: new google.maps.LatLng(51.503, -0.135),
        zoom: 12
    };
    map = new google.maps.Map(document.getElementById('map'), mapOptions);
```


www.dbooks.org

```
    geocoder = new google.maps.Geocoder();

    directionsService = new google.maps.DirectionsService;
    directionsRenderer = new google.maps.DirectionsRenderer;
    directionsRenderer.setMap(map);


    origin = document.getElementById("txtOrigin").value;
    destination = document.getElementById("txtDestination").value


    google.maps.event.addDomListener(
        document.getElementById("btnSubmit"), "click", function () {
            geocodeAddresses(origin, destination).then(function (result) {
                findDirections(result);
            }, function (err) {
                alert(err);
            });
        });
}
```

To calculate the route, you call the **`route()`** method on the **`google.maps.DirectionsService`**, passing in a **`google.maps.DirectionsRequest`** object and a callback function.

The **`DirectionsRequest`** object supports many different properties to allow you to configure the route calculation, including driving options; the ability to avoid ferries, highways, and toll roads; and to route via waypoints. We're going to keep things simple here, though, and just specify the **`origin`**, **`destination`**, and **`travelMode`** properties. Complete the **`findDirections()`** function as shown in Code Listing 48.

Code Listing 48: The findDirections() function

```
function findDirections(journey) {
    directionsService.route({
        origin: journey.origin.geometry.location,
        destination: journey.destination.geometry.location,
        travelMode: 'DRIVING'
    }, function (response, status) {
        if (status === 'OK') {
            directionsRenderer.setDirections(response);
        } else {
            window.alert('Directions request failed: ' + status);
        }
    });
};
```

Open the page in a browser and enter **Big Ben** as the origin, and **47 Fleet Street, London, EC4Y** 1BJ as the destination. Verify that the route is plotted as shown in Figure 48.

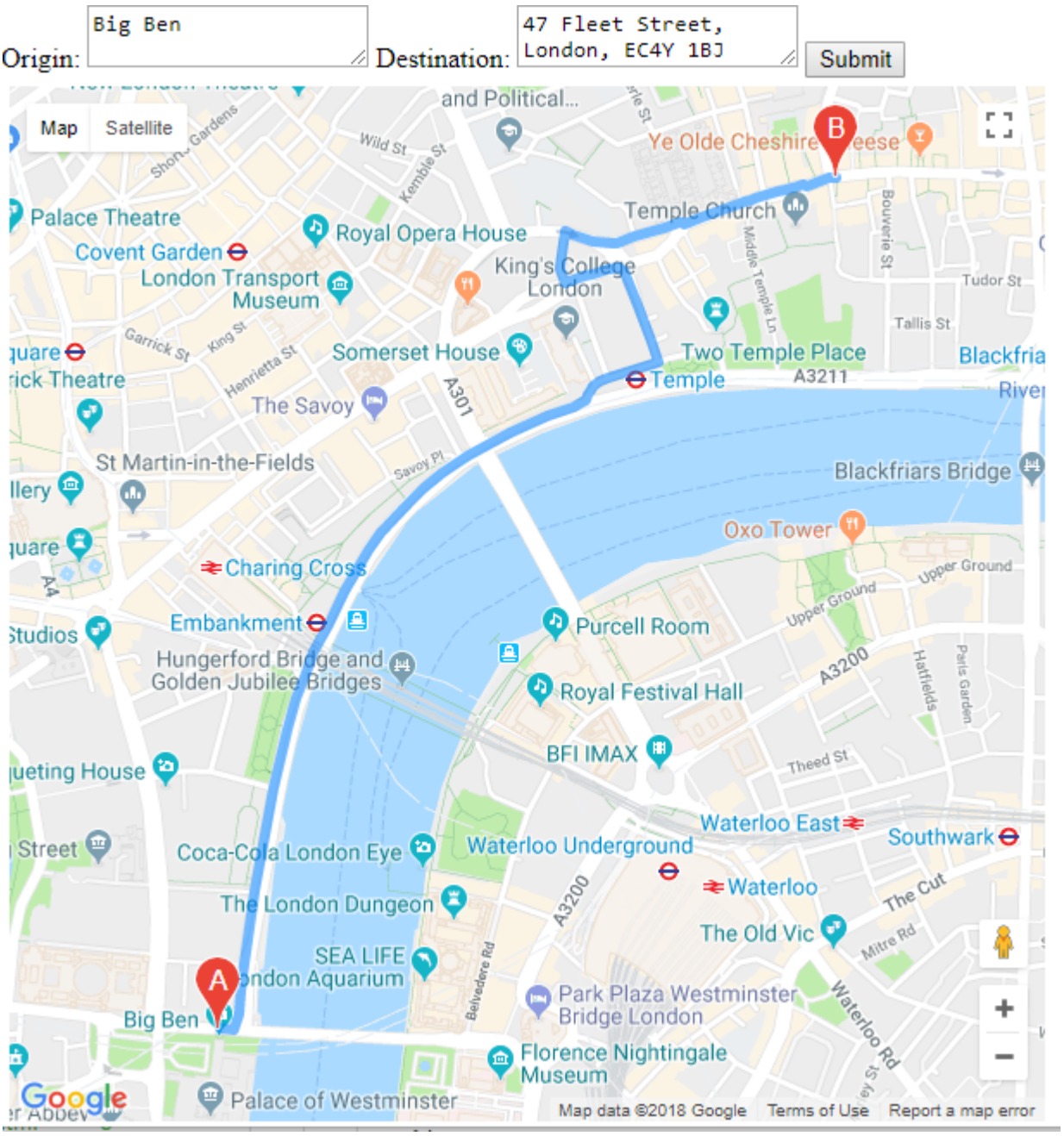


Figure 50: The calculated route.

So far, so good—but half of our page hasn't been put to use yet. Let's use the **`DirectionsRenderer`** to display turn-by-turn directions in the **`<div>`** with an **`id`** of **`directions`** that we created specifically for that purpose.

www.dbooks.org

In the **initMap()** function, call **setPanel()** on the **DirectionsRenderer** object and pass in the ** ** where you want the directions to display. The call to **DirectionsRenderer.setDirections()** in the **findDirections()** function will then also display driving directions. See Code Listing 49.

Code Listing 49: Specifying the for the turn-by-turn directions

```
function initMap() {
    var mapOptions = {
        center: new google.maps.LatLng(51.503, -0.135),
        zoom: 12
    };
    map = new google.maps.Map(document.getElementById('map'), mapOptions);

    geocoder = new google.maps.Geocoder();

    directionsService = new google.maps.DirectionsService;
    directionsRenderer = new google.maps.DirectionsRenderer;
    directionsRenderer.setMap(map);
    directionsRenderer.setPanel(document.getElementById("directions"));

    origin = document.getElementById("txtOrigin").value;
    destination = document.getElementById("txtDestination").value


    google.maps.event.addDomListener(
        document.getElementById("btnSubmit"), "click", function () {
            geocodeAddresses(origin, destination).then(function (result) {
                findDirections(result);
            }, function (err) {
                alert(err);
            });
        });
}
```

Run the application again with **Big Ben** as the origin and **47 Fleet Street, London, EC4Y 1BJ** as the destination. This time, the directions should appear on the right-hand side of the page, as shown in Figure 49.

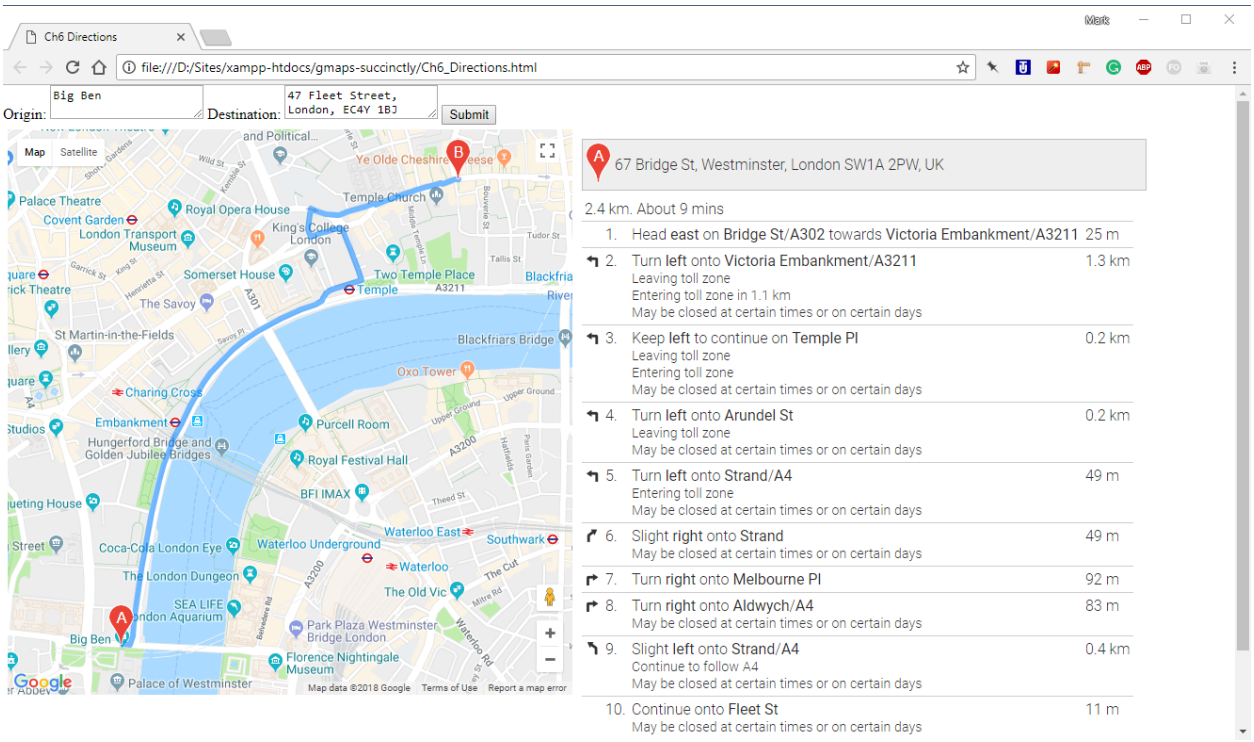


Figure 51: Route with Driving Directions.

The **`DirectionsRenderer`** object took care of all this for you. One other benefit of using **`DirectionsRenderer`** is that the route is now interactive. Click on a stage in the right-hand pane, and the corresponding part of the route is annotated on the map, as shown in Figure 50.

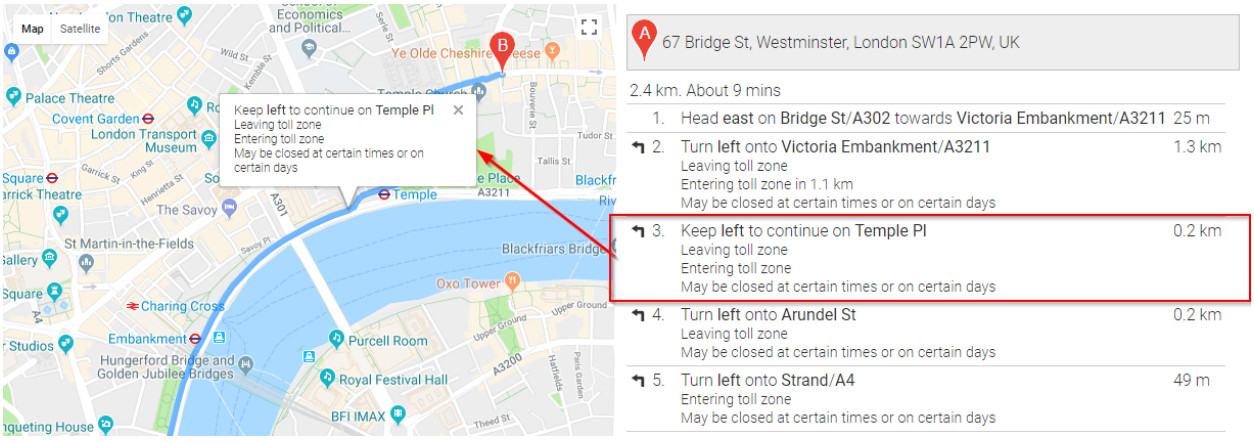


Figure 52: Displaying route segments.

www.dbooks.org

Chapter 8 Conclusion

Thank you for reading (and hopefully coding) along. If you've gotten this far, you now have a pretty good idea of what the Google Maps API can do, and how to do it.

However, this is a fairly large API, and the Google ecosystem is absolutely huge, so there's a lot more you could learn if you want to take your mapping applications to the next level. Thankfully, it's all well documented by Google, and plenty of other people have written tutorials, blog posts, and other articles on developing against Google Maps and other APIs.

I hope you are inspired to create something truly wonderful with the Google Maps API and, if you do, I'd love to hear about it! You can email me at mark@appswithmaps.net or connect with me on Twitter @gisapps.