Community Experience Distilled

Learning Bing Maps API

Obtain geographical data from Bing Maps and display them on the map

Artan Sinani

[PACKT] PUBLISHING

Learning Bing Maps API

Obtain geographical data from Bing Maps and display them on the map

Artan Sinani

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Learning Bing Maps API

First published: November 2013

Production Reference: 1151113

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

ISBN 978-1-78355-037-1

`www.packtpub.com`

Cover Image by Aniket Sawant (`aniket_sawant_photography@hotmail.com`)

Credits

Author
Artan Sinani

Reviewers
Thomas M. Anderson
Adrian Cox
Seth Richards

Acquisition Editors
Akram Hussain
Neha Nagwekar

Commissioning Editor
Sruthi Kutty

Technical Editors
Shruti Rawool
Anand Singh

Project Coordinator
Aboli Ambardekar

Proofreader
Clyde Jenkins

Indexer
Rekha Nair

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Artan Sinani is a Web Developer living between London and a small town in the north-west of Spain.

Currently, he works as a Senior Developer at Local Data Company, where he uses his passion for good design and clean programming to build beautiful interfaces, including geospatial data visualizations.

He hacks open source code at `github.com/artisinani` and blogs at `lugolabs.com`.

This book would not have been written without the help and understanding of my wife, Sonia, and son, Mattias, who put up patiently with me writing during family time. I would like to thank my parents Vasil and Elsa, sister Sara, and brother David, who have supported and inspired me throughout my life.

I would like to thank my colleagues—first and foremost—Adrian, who is a also a technical reviewer of this book, for all his help and understanding; Ivana Mogin, Barnaby Oswald, Alex Oswald, and all the great people at LDC, who gave me the opportunity to work with great technologies.

About the Reviewers

Thomas M. Anderson is a Software Engineer by trade, autodidact by leisure, and a self-proclaimed tinkerer. He studied Computer Science and Jazz percussion at North Central College in Naperville, Illinois, and web designing and development for another two years at Full Sail University in Orlando, Florida. When not behind a computer screen, he can be found making music, reading a good book, or pondering over countless questions that need answers.

He has been with his current employer and second family, Punchkick Interactive, for over three years. Punchkick Interactive is a full-service mobile marketing agency, and a leader within the mobile development industry.

I would like to thank my family and friends.

Adrian Cox is a Software Development Manager with over 15 years of commercial experience. He is still very much hands on with development and system design, and stays up-to-date with the latest technologies.

Seth Richards has over 10 years of professional software development experience. He got his start by programming embedded devices for the bar and nightclub industry and transitioned to web application development six years ago. In the past, he has worked on web-based enterprise-grade geographic information system applications. Seth is currently a Senior Software Engineer for a Massachusetts-based company that provides process automation, work automation, and document management solutions. You can follow him on `twitter@shrichards`, or view his blog at `http://blog.shrichards.com`

Table of Contents

Preface

When I first started working with the Bing Maps APIs, I was amazed at how easy it was to drop a map on a web page with just a few lines of code. Then I started changing the way the map looked; then I added some pushpins to show points of interest, and inevitably changed the way the pins looked.

This book describes many of these processes and much more, building on real applications as we explore new venues of the vast universe of Bing Maps.

What this book covers

Chapter 1, *Introduction to Bing Maps AJAX Control Version 7*, explains how to embed a map on a web page. It shows the way to create a custom theme by changing its interactive controls with a simple dashboard.

Chapter 2, *Diving into Bing Maps AJAX Control Version 7*, explains how to add interactivity to the map, by drawing custom polygons and pushpins and attaching different events to them.

Chapter 3, *Introduction to Bing Maps REST Services*, introduces the Locations API and reverse geocoding addresses from coordinates of any point of the map.

Chapter 4, *Diving into Bing Maps REST Services*, covers the Routes API and ways of fetching and displaying on the map route information between multiple places. It takes a look at different data formats that are exposed by this interface.

Chapter 5, *Spatial Data Services*, introduces geocoding addresses by setting up automated jobs on the Bing Maps servers. It examines techniques of monitoring the jobs, and on their completion, parsing the response.

Chapter 6, *Diving into Spatial Data Services*, focuses on an experimental part of Bing Maps, the Geodata API. It presents ways of obtaining geographical information about various types of administrative areas, from postcode to region or country.

Chapter 7, Enriching Bing Maps with Overlaying User Data, wraps up the book with a web application that turns the map into an interactive chart by means of user data visualizations.

What you need for this book

You need some experience with JavaScript, HTML, and CSS. For some chapters, knowledge of C#, ASP.NET MVC, and Visual Studio is also required.

Who this book is for

This book is for developers that wish to build applications with Bing Maps, but have no previous knowledge of the APIs. More experienced developers can find useful chapters about areas of Bing Maps with which they are not familiar.

Conventions

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

Code words in text are shown as follows: We can include other contexts through the use of the `include` directive.

A block of code is set as follows:

```
var map = new Microsoft.Maps.Map(container, {
  credentials: '[YOUR BING MAPS KEY]',
  center: new Microsoft.Maps.Location(39.554883, -99.052734),
  mapTypeId: Microsoft.Maps.MapTypeId.road,
  zoom: 4
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
8"/>
<title>LBM | Chapter 1: Introduction to Bing Maps</title>
<link href="app.css" rel="stylesheet" />
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: Before that, we deactivate the **active** button (by removing the `active` class) if it exists.

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.You can download this file from: `http://www.packtpub.com/sites/default/files/downloads/0371OT_ColoredImages.pdf`

Errata

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

Piracy

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

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

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

Questions

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

1
Introduction to Bing Maps AJAX Control Version 7

Bing Maps by Microsoft provide a collection of Application Programming Interfaces (APIs) that adds mapping capabilities to location-aware applications. The APIs, open to the public since their first beta release in 2005, can query for location or business by address or coordinates. They enable searching for routes, including traffic information; searching for geometrical shapes of geographical entities such as countries, regions, and other smaller administrative divisions. Consumers of Bing Maps APIs can geocode address, or reverse geocode coordinates via automated jobs.

All the APIs can be used with JavaScript, but server-side languages such as .Net, PHP, Ruby, and so on can also be used. The supported browsers include Internet Explorer Version 7 onwards, Firefox Version 3.6 onwards, Safari Version 5 onwards, Opera, Google Chrome, iPhone, Android, and Blackberry mobile browsers are also supported.

To access the APIs, a Bing Maps key is needed; it can be obtained at the Accounts Center available at `https://www.bingmapsportal.com`. Bing Maps offer a few access options, including a basic key for non-profit, educational purposes, with limited usage. For production scenarios, Bing Maps offer an enterprise key.

Bing Maps AJAX Control Version 7

The AJAX Control is a JavaScript library that adds a fully functional map to a web page. The current version, V7, released in 2010, was written with a modular approach in mind, which reduced the core part of the library to around 35 KB. Additional modules, such as the directions module, or the ones we will write in this chapter, can be loaded dynamically on request.

In this chapter, we are going to place a map on our webpage, and customize its look. First, let's create a simple HTML5 `index.html` file, as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
  8"/>
  <title>LBM | Chapter 1: Introduction to Bing Maps</title>
</head>
<body>
</body>
</html>
```

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

Microsoft recommends using `UTF-8` encoding on the page. To load the control, we add the `script` tag to the `header`, as shown in the following code snippet:

```
<script src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.
ashx?v=7.0"></script>
```

Now, let's add the `div` element that will contain the map in the `body`, as shown in the following code snippet:

```
<div id="lbm-map"></div>
```

We need to style our div with height and width, so let's add an external stylesheet to our page, which now should look as shown in the following code:

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
  8"/>
  <title>LBM | Chapter 1: Introduction to Bing Maps</title>
  <link href="app.css" rel="stylesheet" />
  <script src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.
ashx?v=7.0"></script>
</head>
<body>
```

```
    <div id="lbm-map"></div>
  </body>
  </html>
```

The width and height of the map container will be used by the AJAX Control to resize the map. The container must also be styled with relative or absolute position. So, let's code them in our stylesheet file, as follows:

```
#lbm-map {
  position: relative;
  width: 100%;
  margin: 20px auto;
  border: 1px solid #888;
}
```

We have used **Learning Bing Maps** (**LBM**) to prefix our styles.

Now, let's write the function that instantiates the map on page load in a `app.js` JavaScript file as shown in the following code:

```
window.onload = function() {
  var container = document.getElementById('lbm-map'),
    resize = function () { container.style.height = win
    dow.innerHeight + 'px'; };
  resize();

  var map = new Microsoft.Maps.Map(container, {
    credentials: '[YOUR BING MAPS KEY]',
    center: new Microsoft.Maps.Location(39.554883, -99.052734),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 4
  });

  window.onresize = function() {
    resize()
  };
};
```

First, we get a reference to the map container and store it in a `container` variable for later use. Since we want our map to cover the whole browser window, we give our container the same height as the window (we already specified the width value as 100 percent).

The main function (we will also refer to them as **classes**, even though this term is not supported as such in JavaScript) of the AJAX Control is named `Microsoft.Maps.Map`, and accepts two arguments—the first argument is a reference to the DOM element of the map container and the second argument is a `Microsoft.Maps.MapOptions` object—used to customize the map.

One of the important properties of the `MapOptions` object is `credentials`, and it holds the Bing Maps key. Other properties control the view of the map, such as the zoom level, the background color, the default controls; or the map's behavior, such as panning, user clicks or touches, keyboard input, and so on (the full reference of `MapOptions` can be found at `http://msdn.microsoft.com/en-us/library/gg427603.aspx`). Please note that `Microsoft.Maps` is the namespace that groups all the JavaScript classes of the control.

Going back to our code, at the end of it, we make sure the map is resized to fill the whole window, even when the user resizes the browser window. Note how the map follows the dimensions of its container.

We need to add a reference to this file on our `index.html` page, and we place the reference just before the closing `</body>` tag, so that the file is loaded after all the other assets of the page have been downloaded by the browser.

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

If we open the `index.html` file on the browser, we should get the map as shown in the following figure:

UNITED STATES
Washington
Montana
North Dakota
Minnesota
Lake Superior
Oregon
Idaho
Wyoming
South Dakota
Wisconsin
Michigan
Lake Huron
Lake Michigan
Iowa
Nebraska
Lake Erie
New York
Illinois
Indiana
Ohio
Pennsylvania
Nevada
Utah
Colorado
Kansas
Missouri
West Virginia
Kentucky
Virginia
California
Oklahoma
Tennessee
North Carolina
Arizona
New Mexico
Arkansas
South Carolina
Texas
Mississippi
Alabama
Georgia
Louisiana

The map is centered somewhere in Kansas, United States. To do this, we set the `center` option to a `Microsoft.Maps.Location` instance, which accepts two parameters: latitude and longitude.

```
new Microsoft.Maps.Location(39.554883, -99.052734)
```

The `zoom` option can take any value between 1 and 20, with 1 zooming out far and 20 zooming in close. The `mapTypeId` property is a value of `Microsoft.Maps.MapTypeId`, which includes *aerial*, *birdseye*, and so on. (The list can be found at `http://msdn.microsoft.com/en-us/library/gg427625.aspx`).

Now, let's remove the map's dashboard, the controls at the top-right corner of the map, and replace them with our own. To achieve the former, we add two more options to our map instantiation function, as shown in the following code snippet:

```
var map = new Microsoft.Maps.Map(document.getElementById('lbm-map'), {
    credentials: '[YOUR BING MAPS KEY]',
    center: new Microsoft.Maps.Location(39.554883, -99.052734),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 4,
    showBreadcrumb: false,
    showDashboard: false
});
```

Custom modules

We'll add our own controls to the map; they will change the zoom level and the map type. For this, we will take advantage of the module structure of Bing Maps. The modules are just JavaScript functions that are loaded dynamically by Bing Maps AJAX Control. Apart from better code organization, the modules provide on-demand downloads, reducing the initial script load.

On the flip side, this means that the JavaScript files will be loaded in multiple HTTP requests. If you prefer your scripts merged into a single HTTP request, you can use tools that merge and minimize website assets, such as Asset Bundler in ASP.NET, Sprockets in Ruby, gzipit in PHP, and so forth. In this case, make sure that the custom modules are instantiated after the map is loaded (more information about the Bing Maps modules can be found at `http://msdn.microsoft.com/en-us/library/hh125836.aspx`).

We'll call our module `LearningTheme`, and place it under a `learningTheme` folder. Let's write our module within a self-calling, immediately invoked function in the `learningTheme.js` file:

```
(function () {
  var lbm = window.lbm = {};
  lbm.LearningTheme = function(map) {
    this._map = map;
  };
  Microsoft.Maps.moduleLoaded('lbm.LearningTheme');
})(this);
```

We have placed our `LearningTheme` module under the namespace `lbm`, just as we did with the CSS styles. This way, we expose only one global variable, `lbm`.

The module is just a normal JavaScript function with one argument, that is, `map`. This is a reference to the Bing Maps map we instantiated earlier, which we'll store into a private variable named as `_map`.

We'll use the convention of starting the variable names with an underscore in this book, to indicate that they should be considered as *private* variables, and not be called outside our module. There are ways to achieve better access restriction for objects in JavaScript, but they are outside the scope of this book.

After the module body, we add the following line, which turns our function into a Bing Maps module:

```
Microsoft.Maps.moduleLoaded('lbm.LearningTheme');
```

We then load the module in our `window.onload` function by first registering it:

```
Microsoft.Maps.registerModule('lbm.LearningTheme', 'learningTheme/
learningTheme.js');
Microsoft.Maps.loadModule('lbm.LearningTheme', {
  callback: function() {
    var learningTheme = new lbm.LearningTheme(map);
  }
});
```

The registration function accepts two arguments: the name of the module, and its relative or absolute (or remote) location. We then call `Microsoft.Maps.loadModule`, which again needs the module name, and an options object. With the latter argument, we can specify what should happen when the module is loaded. In this case, we instantiate it by passing a reference to the map we created earlier.

Let's add the two buttons that will control the zoom level, one with a minus (-) sign that will decrease the zoom level, and one with a plus (+) sign that will increase it. We'll also add a third button that will toggle the map type between road and aerial.

A function within the module's prototype (a private instance method we can call later with the `this` keyword) achieves the following:

```
lbm.LearningTheme.prototype = {
  _addHeader: function() {
    var header = document.createElement('div');
    header.className = 'lbm-map-header';
    header.innerHTML = [
      '<button class="lbm-btn" id="lbm-map-btn-zoomin">-
      </button>',
      '<button class="lbm-btn" id="lbm-map-btn-
      zoomout">+</button>',
      '<button class="lbm-btn" id="lbm-map-btn-
      maptype">Aerial</button>'
    ].join('');

    this._map.getRootElement().appendChild(header);
  }
};
```

First, we create a DOM `div` element and give it a class name OF `lbm-map-header`, so that we can style it later. The markup for the buttons is simple, shown as follows:

```
<button class="lbm-btn" id="lbm-map-btn-zoomin">-</button>
```

Again we specify a class (`lbm-btn`) for styling and an ID (`lbm-map-btn-zoomin`) for controlling its behavior. We place the buttons in an array that we later join with an empty string. This is a common technique for building HTML strings in a JavaScript code, BECAUSE it improves the readability of the code.

At the end, we append the markup to the map's root element. Bing Maps provide a nice utility method to get the latter, `getRootElement`. (The list of methods, properties, and events of the map can be found at `http://msdn.microsoft.com/en-us/library/gg427609.aspx`).

We want the header to be added to the map when the module is loaded; therefore, we call the `_addHandler` method in the module's constructor, which now looks like this:

```
lbm.LearningTheme = function(map) {
  this._map = map;
  this._addHeader();
};
```

If you view the `index.html` file on the browser, you will not see the default Bing Maps dashboard, but you won't see our controls either. This is because we have not added our styles yet. Let's place them in a `learningTheme.css` file inside the `learningTheme` folder, as shown in the following code:

```
.lbm-map-header {
  position: absolute;
  top: 1em;
  left: 1em;
  width: 100%;
  color: #fff;
  z-index: 100;
}

.lbm-map-header .lbm-btn {
  font-size: 1em;
  border: 0;
  background: #F04C40;
  color: #fff;
  display: inline-block;
  text-decoration: none;
  width: 2.5em;
  height: 2em;
  line-height: 2em;
  padding: 0;
  margin: 0 1px 0 0;
  outline: 0;
  text-align: center;
  cursor: pointer;

  opacity: .8;

  border-radius: .2em;
  -webkit-border-radius: .2em;
  -moz-border-radius: .2em;
}

.lbm-map-header #lbm-map-btn-zoomin {
  border-top-right-radius: 0;
  -webkit-border-top-right-radius: 0;
  -moz-border-top-right-radius: 0;
  border-bottom-right-radius: 0;
  -webkit-border-bottom-right-radius: 0;
  -moz-border-bottom-right-radius: 0;
}

.lbm-map-header #lbm-map-btn-zoomout {
  border-top-left-radius: 0;
  -webkit-border-top-left-radius: 0;
  -moz-border-top-left-radius: 0;
  border-bottom-left-radius: 0;
  -webkit-border-bottom-left-radius: 0;
  -moz-border-bottom-left-radius: 0;
}

.lbm-map-header .lbm-btn:hover {
  background: #cf343e;
}

.lbm-map-header .lbm-btn:active {
  background: #cc1d28;
}

.lbm-map-header #lbm-map-btn-maptype {
  margin-left: 2em;
  width: auto;
  padding: 0 .8em;
}
```

We make sure that the buttons' container has an absolute `position` and a high `z-index` value, so that it appears above the map.

Now, we need to add a link to the CSS file to our HTML document, and we'll do this when the module is loaded. This way, if the module is never loaded, then the stylesheet is not loaded either. Again, loading our CSS file on demand means another HTTP request, and if you prefer to have the styles merged into a single file, then you can use the same technique as for the JavaScript files, and skip the following step:

We'll add a link to our CSS file to the head of the page as shown in the following code snippet:

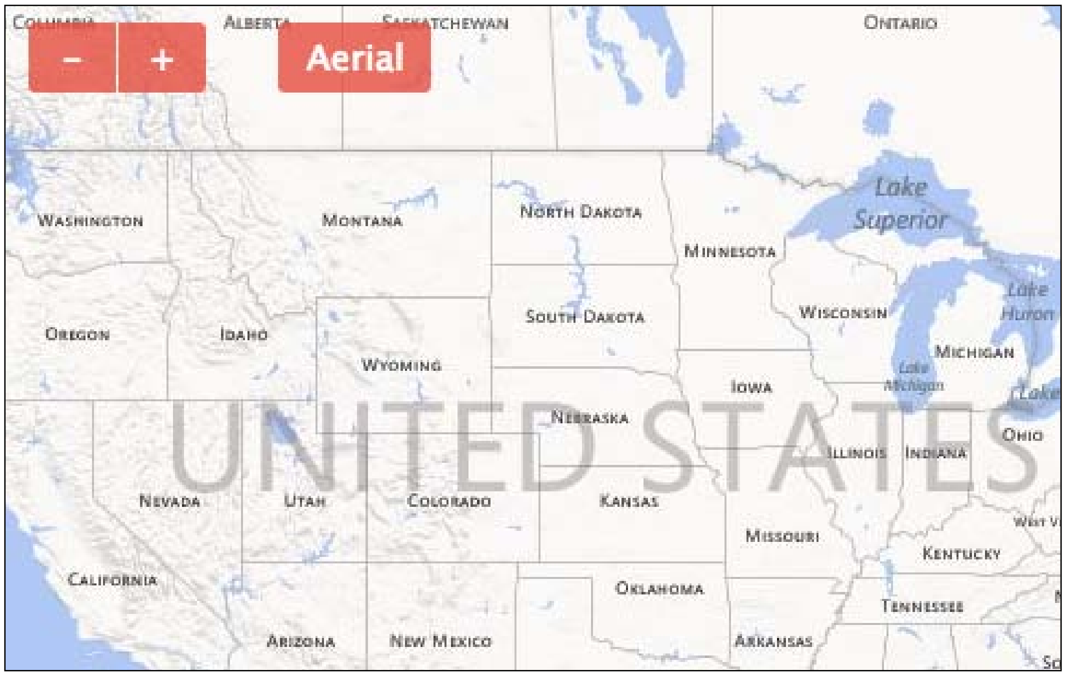
```
_addCss: function() {
  var link = document.createElement('link');
  link.setAttribute('rel', 'stylesheet');
  link.setAttribute('href', 'learningTheme /learningTheme.css');
  var head = document.getElementsByTagName('head')[0];
  head.appendChild(link);
}
```

First, the preceding function creates a `link` element, with a `href` attribute pointing at our `learningTheme.css` file. Then, it gets a reference to the `head` of the document, and it appends to it the newly created element.

As with the buttons markup, we need the stylesheet to be loaded with the module, so we'll call this function in the module's constructor, as shown in the following code:

```
lbm.LearningTheme = function(map) {
  this._map = map;
  this._addCss();
  this._addHeader();
};
```

If we open the `index.html` file on a browser now, we can finally see our controls, as shown in the following screenshot:



Map events

We need our buttons to respond to mouse clicks. Since we added them to the map's root element, we can use the Bing Maps events to add this functionality. Bing Maps AJAX Control provides a rich event system, that can be attached to the map or its entities. Some of the events include `click`, `dblclick`, `keydown`, `keypress`, `imagerychanged`, and so on.

Let's add our eventing functionality to a private method, which accepts a context argument, as shown in the following code:

```
_bind: function(self) {
  Microsoft.Maps.Events.addHandler(this._map, 'click', function(e) {
    if (e.originalEvent && e.originalEvent.target) {
     var btn = e.originalEvent.target;
      if (btn.tagName === 'BUTTON' && btn.className.match(/lbm-btn/) )
{
        if (btn.id === 'lbm-map-btn-maptype') {
          self._changeMapType(btn);
        } else {
          self._zoom(btn);
        }
        e.handled = true;
      }
    }
  });
}
```

The `self` argument passed to the `_bind` function will call any functions declared within it in the module's context. This is necessary, because JavaScript changes the meaning of `this` inside the handling function from our module to the DOM element firing the event.

We then add an event handler to the map by calling the `Microsoft.Maps.Events.addHandler` method, which accepts three parameters—the `map` (or its entity, such as a pin or shape), the name of the event, and the `handler` function. To the latter is then passed a `MouseEventsArgs` object, which exposes a few properties and methods (see the full description available at `http://msdn.microsoft.com/en-us/library/gg406731.aspx`).

One of these properties is `originalEvent`. It is tied to the DOM event of the clicked element, which can be found by calling `e.originalEvent.target`. We then make sure that the elements clicked are, in fact, our buttons, and we call the respective handler depending on their ID, and passing it the button clicked as an argument.

We set the `handled` property to the `Events` object to `true`, telling the map to not call any original handlers attached to the same event.

In order for the events handlers to work, we need to call our binding function in the module's constructor, passing it `this` as the context, because `this` in the constructor references the module itself. Now the constructor looks as shown in the following code snippet:

```
lbm.LearningTheme = function(map) {
  this._map = map;
  this._addCss();
  this._addHeader();
  this._bind(this);
};
```

To implement the handlers, we start with the `_changeMapType`:

```
_changeMapType: function(btn) {
  var options = this._map.getOptions();
  if (btn.innerHTML.match(/aerial/i)) {
    btn.innerHTML = 'Road';
    options.mapTypeId = Microsoft.Maps.MapTypeId.aerial;
  } else {
    btn.innerHTML = 'Aerial';
    options.mapTypeId = Microsoft.Maps.MapTypeId.road;
  }
  this._map.setView(options);
}
```

First, we cache the map's current options by calling the `getOptions` method. This allows us to change the options object and pass it back to the map later, without having to set all the options.

The handler then checks the text of the button, and it changes the `mapTypeId` option to the corresponding `Microsoft.Maps.MapTypeId` enum value. At the end it passes the options back to the map's `setView` method, which in return changes the map type.

If we open the `index.html` file on the browser, and click on the **Aerial** button, we see how the map changes from the road view to the aerial view. The button's text also changes to Road. Clicking on it again, will change its text back to Aerial and the map view to road. The AJAX Control adds a nice animation when changing the map view; it can be disabled by specifying `animation: false` in the options object.

The `_zoom` handler looks like this:

```
  _zoom: function(btn) {
    var zoom = this._map.getZoom();
    if (btn.id === 'lbm-map-btn-zoomout') {
      zoom++
    } else if (btn.id === 'lbm-map-btn-zoomin') {
      zoom--
    }
    var options = this._map.getOptions();
    options.zoom = zoom;
    this._map.setView(options);
  }
```

First, we get the map's current zoom value, by calling the `getZoom` method, and store it on a `zoom` variable. We then increase or decrease the value of this variable based on which button was clicked. The IDs we specified earlier for the buttons come in handy here, BECAUSE we can easily identify the event target by its DOM ID. Please note that the Microsoft.Maps API will ignore any zoom levels outside the established range between 1 and 20.

We then get the existing view options of the map and extend them with the updated zoom level. As in the previous handler, we pass them to the `setView` method of the map. After refreshing the `index.html` file on the browser, we see how our zoom buttons change the map's zoom level as we click on them.

Summary

The Bing Maps AJAX Control is a JavaScript library that adds mapping functionality to a web page. In this chapter, we've learned how to load a map on a page, and how easy it is to customize its appearance and functionality. We've looked at the map methods for getting and setting options, zoom, map type, and so on. We've also introduced the map events, and have added handlers to a custom map dashboard. In the next chapter, we will extend these events to draw polygons on the map.

2
Diving into Bing Maps AJAX Control Version 7

With the AJAX Control, we can add pins, lines, and polygons to the map. We can even add complex shapes, such as donut structures, if we load the `Microsoft.Maps.AdvancedShapes` module. However, the library does not have built-in tools that allow the user to draw shapes manually, so let's create one.

We will build the code we wrote in the previous chapter. This means that we have an `app.js` file, where we instantiate the map, and load the `LearningTheme` module, which customizes the look and the behavior of the map.

We will create our tool as a module, and name it `ShapeDrawing`. Just as we did in *Chapter 1, Introduction to BING Maps AJAX Control Version 7*, let's create a folder named `shapeDrawing`, and a JavaScript file inside it with the same name.
The module skeleton is simple, as shown in the following code:

```
(function() {
  var lbm = window.lbm || {};
  lbm.ShapeDrawing = function(map) {
    this._map = map;
  };
  Microsoft.Maps.moduleLoaded('lbm.ShapeDrawing');
})(this);
```

We declare the module under the namespace `lbm`, so that it does not conflict with other library objects with the same name. The `ShapeDrawing` function accepts only the `map` argument, which is an instance of the `Microsoft.Maps.Map` class, stored in the `_map` instance variable.

At the end we make a call to `Microsoft.Maps.moduleLoaded`, which turns the function into a loadable module by the AJAX Control.

To load the module on the page, we need to register it in the `app.js` file we created in the previous chapter, which now looks as follows:

```
window.onload = function() {
  var container = document.getElementById('lbm-map'),
    resize = function () { container.style.height = window.innerHeight
+ 'px'; };
  resize();

  var map = new Microsoft.Maps.Map(container, {
    credentials: '[YOUR BING MAPS KEY]',
    center: new Microsoft.Maps.Location(39.554883, -99.052734),
      mapTypeId: Microsoft.Maps.MapTypeId.road,
      zoom: 4,
      showBreadcrumb: false,
    showDashboard: false
  });

  Microsoft.Maps.registerModule('lbm.LearningTheme', 'learningTheme/
learningTheme.js');
  Microsoft.Maps.loadModule('lbm.LearningTheme', {
    callback: function() {
      var learningTheme = new lbm.LearningTheme(map);
    }
  });

  Microsoft.Maps.registerModule('lbm.ShapeDrawing', 'shapeDrawing/
shapeDrawing.js');
  Microsoft.Maps.loadModule('lbm.ShapeDrawing', {
    callback: function() {
      var shapeDrawing = new lbm.ShapeDrawing(map);
    }
  });

  window.onresize = function() {
    resize()
  };
};
```

When the AJAX Control loads the module, we create an instance of `ShapeDrawing`, passing it a reference to the map.

Custom events

Our tool will enable the users to draw custom polylines on the map, but we don't want this activity to interfere with other tasks performed on the map, such as dragging, clicking, and so on. Therefore, we need to add a button so that, when clicked, can allow the users to draw a shape on the map.

Respecting our modular system, we will give the power to add buttons to our `LearningTheme` module, which generates the header holding the buttons. And we will use custom events to do that.

The AJAX Control provides a simple, but a powerful `Events` object, which among others exposes two static methods: `addHandler` and `invoke`. The former method attaches a function to the map (and as we will see later in this book, also to polygons, pins, and other map entities) when an event, specified by an event name, is triggered. The invoke function triggers the event specified by the event name, with some optional data objects. The event names are usually `click`, `dblclick`, `keypress`, and so on, but in fact they can be any string; hence, the custom events are made possible.

Going back to our old `LearningTheme` class, we add some new code to the `_bind` method:

```
_bind: function(self) {
  ...
  Microsoft.Maps.Events.addHandler(this._map, 'header:button',
function(e) {
    self._addButton(e);
  });
}
```

Our custom event is named `header:button`, and `LearningTheme` adds a button whenever it's triggered. This is done in a new method, as shown in the following code snippet:

```
_addButton: function(event) {
  var header = document.getElementById('lbm-map-header'),
    btn = document.createElement('button'),
options = event.data;

  btn.className = 'lbm-btn text';
  btn.id = 'lbm-map-btn-' + options.id;
  btn.innerHTML = options.text;

  header.appendChild(btn);
  e.handled = true;
}
```

First we get a reference to the header we created earlier, and create a new `button` element. This element needs to have the same style as the other text buttons on the theme, so we'll give it the same class name. We also give the button an ID, so that we can control its behavior.

Next, we store the `data` property of the `event` argument into an `options` object (`event.data` is passed by the `Events` object, when triggered). This is a great way to pass data around different modules without tightly coupling them. In our code, we get the ID and text of the button. In this way, other modules can create buttons on the header and control their text and behavior.

The `ShapeDrawing` module does it through its own `_addButton` method, as shown in the following code snippet:

```
lbm.ShapeDrawing.prototype = {
  _addButton: function() {
    Microsoft.Maps.Events.invoke(this._map, 'header:button', {
      data: {
        id: 'draw',
        text: 'Draw'
      }
    });
  }
};
```

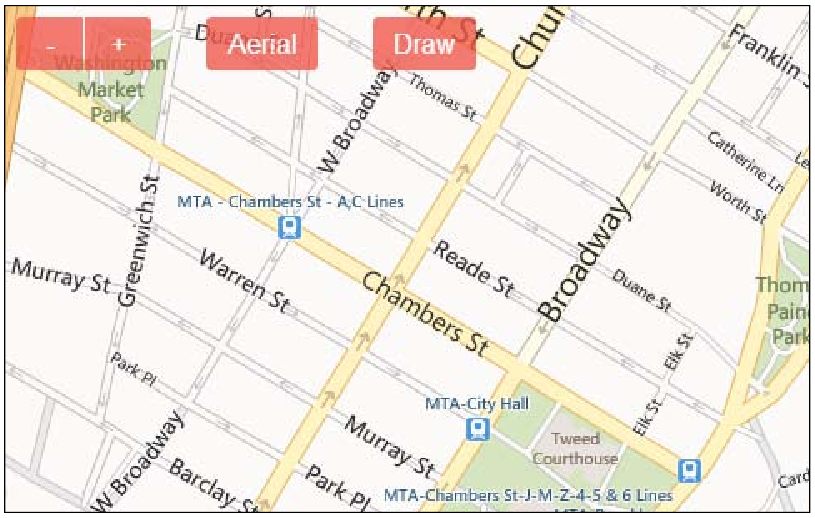
The `invoke` method of the `Events` object accepts the following three arguments:

- `target`: This argument specifies a reference to the map or its entities
- `eventName`: This argument specifies a string
- `args`: This argument specifies an object which will extend the `EventsArgs` object sent to the handler

We need the button to show when the module is first loaded. So, we'll add it to the module's constructor, as shown in the following code snippet:

```
lbm.ShapeDrawing = function(map) {
  this._map = map;
  this._addButton();
};
```

Now when we open the `index.html` file on the browser, we should see the **Draw** button, as shown in the following screenshot:



When the user clicks on the button, the map should enter into a drawing mode. Let's do this next:

```
_bind: function(self) {
  Microsoft.Maps.Events.addHandler(this._map, 'click', function(e) {
    var btn = e.originalEvent.target;
    if (btn.tagName === 'BUTTON' && btn.className.match(/lbm-btn/) ) {
      if (btn.id === 'lbm-map-btn-draw') {
        self._start();
      }
    }
  });
}
```

As on the `LearningTheme` module, we add a `_bind` function to the module's prototype that accepts a `self` argument that points the context of the function to the module itself. This method responds to the map's `click` event, but only when our button with the `lbm-map-btn-draw` ID is clicked.

We need to bind the event when the module is loaded, therefore, we call it on its constructor, as shown in the following code snippet:

```
lbm.ShapeDrawing = function(map) {
  this._map = map;
  this._mapRootElement = this._map.getRootElement();
this._bind(this);
};
```

The `_start` function sets up a drawing mode for the map, starting by changing the cursor to a type that hints to the user that he/she can draw, as shown in the following code:

```
_start: function() {
  this._changeMouseCursor();
}
```

And a method that changes the cursor type, as shown in the following code snippet:

```
_changeMouseCursor: function() {
  this._mapRootElement.style.cursor = 'crosshair';
}
```

This changes the style of the map's root element, which we store on a private instance variable for future reference. Let's declare that variable on the module's constructor, as shown in the following code snippet:

```
lbm.ShapeDrawing = function(map) {
  this._map = map;
  this._mapRootElement = this._map.getRootElement();
  this._addButton();
};
```

Pushpins

Let's stop for a minute and rethink our workflow. When the user clicks on the map, we need to show a pin, which will be a corner of the polygon. When the user double-clicks on the map, we will close the polygon. We also need to provide some visual clues as the user drags the mouse, trying to find another point.

We'll start by writing the method that adds the handlers, as shown in the following code snippet:

```
_addHandlers: function(self) {
  this._handlers = {};
  this._handlers.click =
  Microsoft.Maps.Events.addHandler(this._map, 'click', function(e)
  {
    self._onmapclick(e);
  });
  this._handlers.dblclick =
  Microsoft.Maps.Events.addHandler(this._map, 'dblclick',
  function(e) {
    self._onmapdblclick(e);
  });
}
```

We store the `click` and `dblclick` handlers on a `_handlers` object (instance variable), because we need to remove them when the map exits the drawing mode. First, we must add them when the drawing mode starts, as shown in the following code snippet:

```
  _start: function() {
    this._changeMouseCursor();
    this._addHandlers(this);
  }
```

When the user clicks on the map now, a pin is created, as shown in the following code snippet:

```
  _onmapclick: function(e) {
    var location = this._getLocation(e);
    if (location === null) return;
    this._addPin(location);
    e.handled = true;
  }
```

First, we get the location of the point where the user clicks on the map by calling the `_getLocation` method:

```
  _getLocation: function(e) {
    var point = new Microsoft.Maps.Point(e.getX(), e.getY());
    return this._map.tryPixelToLocation(point);
  }
```

This is the standard way of getting an instance of `Microsoft.Maps.Location` from a click event. We use the `getX` and `getY` methods of the `Events` object, which specify the left and top position of the clicked point relative to the map. With these, we instantiate a `Microsoft.Maps.Point` instance, which when passed to the map's `tryPixelToLocation` method, returns the `Location` object. However, this object can be null, therefore, we need to check before using it.

The Location constructor requires two arguments: latitude and longitude. For example, new Microsoft.Maps.Location(48.856930, 2.341200) produces the Paris location. It's best to limit the coordinate values to six decimal digits, which correspond to approximately 30 cm on the map. (The full reference of the Location class is available at http://msdn.microsoft.com/en-us/library/gg427612.aspx)

Having obtained a location, we now use the _addPin method in the following code snippet:

```
_addPin: function(location) {
  var pin = new Microsoft.Maps.Pushpin(location, {
    icon:     '',
    width:    10,
    height:   10,
    anchor:   new Microsoft.Maps.Point(5, 5),
    typeName: 'lbm-shape-drawing-pin'
  });
  this._pointsLayer.push(pin);
  return pin;
}
```

The Microsoft.Maps.Pushpin (more information about this class can be found at http://msdn.microsoft.com/en-us/library/gg427615.aspx) constructor accepts two arguments: a Location instance, and a PushpinOptions object. We have used some of the properties of the latter argument (see the entire list available at http://msdn.microsoft.com/en-us/library/gg427629.aspx) which can be explained as follows:

- icon: This property usually specifies the path to an image. We leave it empty, so that the AJAX Control doesn't add an image to the DOM div container of the pin, and we will style the pin with CSS.
- width: This property specifies the width (in pixels) of the pushpin icon and its container (only of the latter in our case). The default value is 25.
- height: This property specifies the height (in pixels) of the pushpin icon and its container (only of the latter in our case). The default value is 39.
- anchor: This property specifies the point of the pushpin anchored to its location and defaults to Point (0,0). We will style our pin as a square; therefore, we set an anchor at the center of it.
- typeName: This property specifies the CSS class name that will be added to the DOM element of the pushpin.

In order to display the pushpin on the map, we need to add it to the map's entities, or to a layer attached to them. The map's `entities` property is an instance of the `EntityCollection` class, and stores all the shapes and layers shown on the map. It serves as their root container. We can add our pin to it, or we can add it to a new `EntityCollection` instance—an instance variable named `_pointsLayer`. We then add this to the map's entities, so that the latter is kept tidy (more information about `EntityCollection` can be found at `http://msdn.microsoft.com/en-us/library/gg427616.aspx`).

Let's declare our layer on the module's constructor:

```
lbm.ShapeDrawing = function(map) {
  …
  this._pointsLayer = new Microsoft.Maps.EntityCollection({zIndex:
30});
  this._map.entities.push(this._pointsLayer);
  …
};
```

The `EntityCollection` class accepts an `EntityCollectionOptions` object, through which we specify a `zIndex` value (more information about the other options is available at `http://msdn.microsoft.com/en-us/library/gg427614.aspx`). This will be added to the style of the DOM element representing the layer.

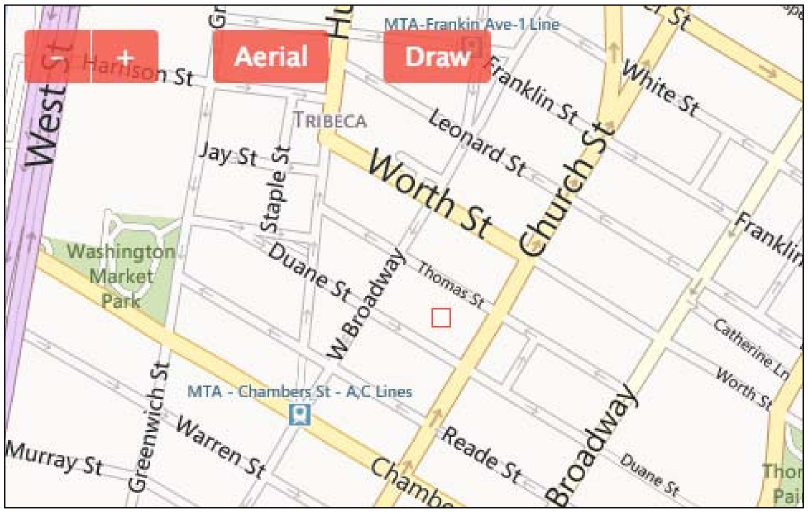
Now it's time to style the pin, and as in the previous chapter, we will create a stylesheet file named `shapeDrawing.css`, inside the `shapeDrawing` folder.

```
_addCss: function() {
  var link = document.createElement('link');
  link.setAttribute('rel', 'stylesheet');
  link.setAttribute('href', 'shapeDrawing/shapeDrawing.css');
  var head = document.getElementsByTagName('head')[0];
  head.appendChild(link);
}
```

We want our styles to be added when our module is loaded, thus we add the following line to the module's constructor:

```
this._addCss();
```

If we open the `index.html` file on a browser now, and click on the **Draw** button and then click on the map, we should see our square pin (this time the map is centered on New York City, USA, latitude 40.714550, longitude -74.007118, zoomed at level 16), as shown in the following screenshot:



Polylines

Now, let's create a line that will connect the pushpins, by adding the following code to our `_start` function:

```
this._line = new Microsoft.Maps.Polyline([new Microsoft.Maps.
Location(0, 0)], {
  strokeColor:     new Microsoft.Maps.Color(200, 240, 76, 64),
  strokeThickness: 2
});
this._counter = 0;
```

The `Polyline` class' constructor accepts two arguments: an array of locations and a `PolylineOptions` object. We initialize our polyline with an array containing a dummy location, and a couple of options (the full reference of `PolylineOptions` can be found at `http://msdn.microsoft.com/en-us/library/gg427595.aspx`):

- `strokeColor`: This option specifies the color of the polyline, which in our case is pink
- `strokeThickness`: This option specifies the thickness of the polyline in pixels

We also maintain a count of the user clicks, which also stores the number of locations that make up the polyline. We make sure to reset the `_counter` instance variable when the user starts drawing, so that the clicks are not double counted.

When the user clicks on the map, we need to add the clicked location to our polyline. The best place to do this is on the `_onmapclick` method, which appears as follows:

```
_onmapclick: function(e) {
  var location = this._getLocation(e);
  if (location === null) return;
  this._addPin(location);
  this._updateLine(location);
  if (this._counter === 0) {
    this._layer.push(this._line);
    this._addMouseMoveHandler(this);
  }
  this._changeMouseCursor();
  this._counter++;
  e.handled = true;
}
```

We saw earlier how to construct a `Location` instance from the user click, and how to build a pushpin from it. Now, we also update the polyline with the current location by calling the `_updateLine` function, as shown in the following code snippet:

```
_updateLine: function(location) {
  var locations = this._line.getLocations();
  locations[this._counter] = location;
  this._line.setLocations(locations);
}
```

In this listing, we first fetch the existing locations of the polyline, by sending it the `getLocations` message (see the entire list of methods, properties, and events of the Polyline class available at `http://msdn.microsoft.com/en-us/library/gg427597.aspx`). We then update the locations array at the same index as the `_counter` variable with the current location, and pass the whole array to the polyline's `setLocations` method. Please note here, that JavaScript will append an item to an array, if it doesn't exist at a given index.

Going back to the `_onmapclick` handler, we need to add the line to another layer when the user clicks for the first time on the map in the drawing mode. We do this by passing the polyline to the `push` method of the `EntityCollection` class. We declare our new layer by adding the following lines to the module's constructor.

```
this._layer = new Microsoft.Maps.EntityCollection({zIndex: 30});
this._map.entities.push(this._layer);
```

At the same moment we bind a mouse move handler that will allow us to provide a clue to the user as he/she draws the line on the map.

```
_addMouseMoveHandler: function(self) {
  this._handlers.mousemove = Microsoft.Maps.Events.addHandler
(this._map, 'mousemove', function(e) {
    self._onmapmove(e);
  });
}
```

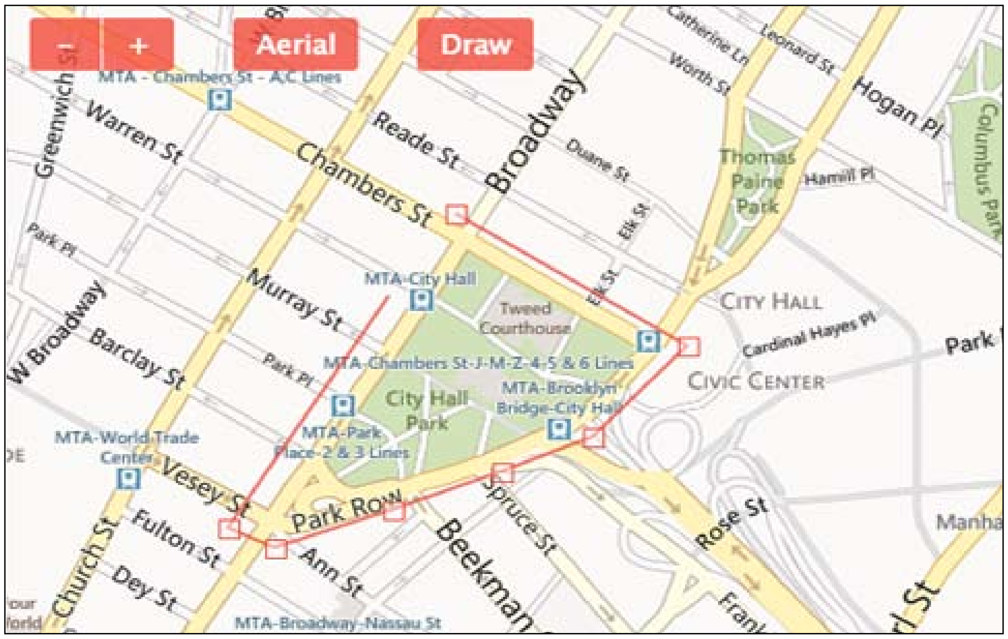
The handler is simple and similar to the `click` handlers we declared previously. It is added to our `_handlers` variable, so that it can be removed when the user finishes drawing, together with the other event handlers.

When the user moves the cursor on the map, the `_onmapmove` function is executed, as shown in the following code snippet:

```
_onmapmove: function(e) {
  var location = this._getLocation(e);
  this._updateLine(location);
  e.handled = true;
}
```

Here, we perform the same task as we did to handle the mouse clicks—update the line with the current location.

If we refresh the `index.html` file now, we can draw a line by clicking and dragging the mouse on the map, as shown in the following screenshot:



At this moment we cannot close the line. We'll do that when we double-click on the map. Let's write that event handler now, as shown in the following code snippet:

```
_onmapdblclick: function(e) {
  this._closeLine();
  this._removeHandlers();
  this._resetMouseCursor();
  this._removePins();
  e.handled = true;
}
```

First, we close the polyline by pushing its first location at the end of its locations array:

```
_closeLine: function() {
  var locations = this._line.getLocations();
  locations.push(locations[0]);
  this._line.setLocations(locations);
}
```

Next, we remove all the handlers we attached earlier, as shown in the following code snippet:

```
_removeHandlers: function() {
  for (var name in this._handlers) {
  Microsoft.Maps.Events.removeHandler(this._handlers[name]);
    delete this._handlers[name];
  }
}
```

The `Events` object provides the `removeHandler` method to remove any previously attached event handlers. It accepts a `handlerId`, which was returned by the `addHandler` method, and which we conveniently stored in the `_handlers` object. As we enumerate through the properties of the latter, we remove the handler as well as delete its reference from the `_handlers` variable.

Back at the `_onmapdblclick` function, we also reset the mouse cursor, by calling the following function:

```
_resetMouseCursor: function() {
  this._mapRootElement.style.cursor = this._initialCursor;
}
```

We can set the cursor style to `default`, but it might happen, and it usually does that Bing Maps use an image for the cursor. Therefore, we get the initial cursor style on module load, and store it in the `_initialCursor` variable. The following line should be added to the module's constructor, just under the line that declares the `_mapRootElement` variable:

```
this._initialCursor = this._mapRootElement.style.cursor || 'default';
```

Now, let's write the last method we call on the double-click handler:

```
_removePins: function() {
  this._pointsLayer.clear();
}
```

The easiest way to remove entities from a collection is by calling its `clear` method.

Other methods of the `EntityCollection` class include `getLength`, which returns the number of all entities added to the collection. We can obtain a reference to an entity inside the layer by calling its `get` method. This method accepts an integer that represents the index of the entity in the collection, and returns `Polyline`, `Pushpin`, `EntityCollection`, as well as any of the following types:

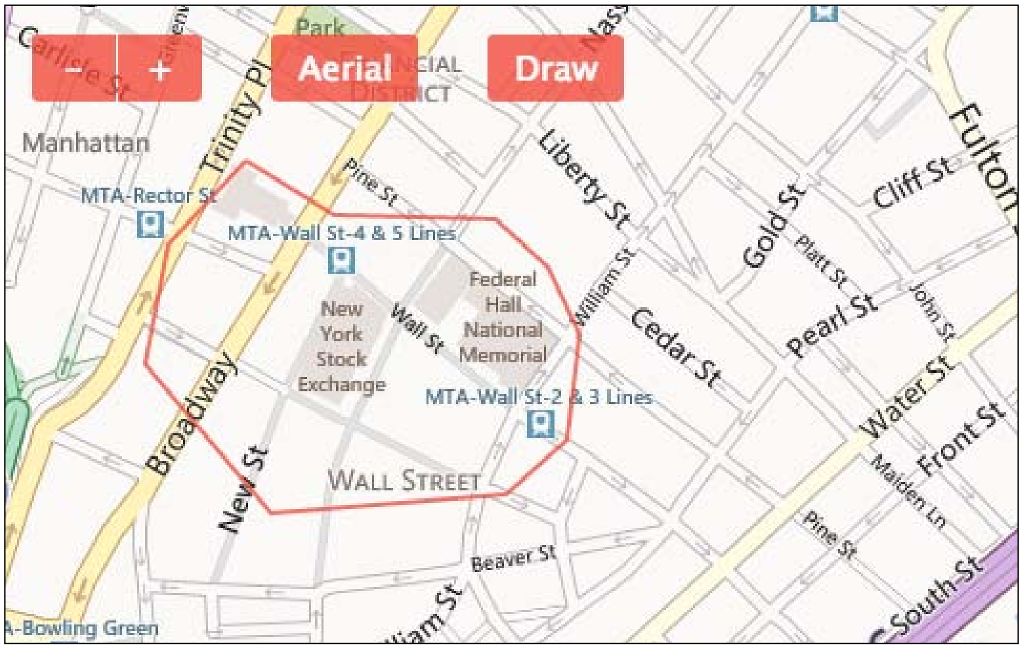
- `Infobox`: This adds an info box on the map, usually attached to a pushpin. We'll see an example of this class in the next chapter.
- `Polygon`: This is used to add polygon shapes to the map; similar to Polyline, but filled.
- `TileLayer`: This creates a custom tile layer on the map, useful for mixing Bing Maps' imagery with our own images.

Many of these classes expose a `getTypeName` method, which in the case of pushpins, returns the CSS class name we provided when we instantiated them. Not all of these classes expose the `getTypeName` method though, so we need to check for its availability on the object before calling it.

The last step is to remove all the entities from the layer, when the user starts a fresh drawing. Let's insert the following code snippet as the first line of the `_start` function:

```
this._layer.clear();
```

Now, when we refresh our `index.html` file on the browser, we can see our completed tool, as shown in the following screenshot:



When we click on the **Draw** button, the mouse cursor changes to `crosshair`, hinting that we are now able to draw. We start moving the cursor around and clicking on the map, seeing how our pink line is following the mouse. When we double-click on the map, the line closes, and the corners disappear.

We can extend our tool further, for example, instead of removing the corners, it turns them into draggable handles that change the size and form of the shape.

Summary

Bing Maps AJAX Control provides a powerful `Events` library, which can be used to add even custom events. In this chapter, we displayed a Pushpin on the map and styling it entirely via CSS. In this chapter we learned how to get the geographical location from a point on the map, generated by a mouse click. From these locations, we created polylines styled beautifully with the `PolylineOptions` object.

In the next chapter, we will learn how to get even more information, such as address data from a point on the map.

3
Introduction to Bing Maps REST Services

Apart from the AJAX Control toolkit we saw in the previous chapters, Bing Maps offer their data through a **Representational State Transfer** (**REST**) interface too. By constructing API specific URLs, this interface allows operations including the creation of static maps with pushpins, geocoding addresses, tracing routes, and retrieving imagery metadata.

The amount of data consumption is subject to your Bing Maps account (more information about billable and non-billable transactions can be found by looking at the Usage Reports, which are explained at `http://msdn.microsoft.com/en-us/library/ff859477.aspx`).

In this chapter we will introduce one of these services, the Locations API. This interface provides the means for finding a location by address, by query, or by a point on the map. The point is a pair of floating point values representing latitude and longitude in decimal format, and it is passed to a URL template. See an example using a pair of lat/long coordinates:

```
http://dev.virtualearth.net/REST/v1/Locations/40.714550/-
  74.007118?includeEntityTypes=entityTypes&includeNeighborhood
  =1&key=[YOUR BING MAPS KEY]
```

When building the template, we start with the root of the URL `http://dev.virtualearth.net/REST/v1/Locations`, followed by the latitude and the longitude of the point. A set of optional parameters can be also added:

- `includeEntityTypes`: This parameter specifies the entity types to be returned on the response, such as Address, Neighborhood, Post code 1, and so on
- `includeNeighborhood`: This parameter (or the short version `inclnb`) indicates a true value, and 0 the opposite

The application

Now, let's build a website that will make use of the Locations API. When the user clicks on the map, our app will fetch address information about the clicked location.

This time, we will consume the API on the server side, using .NET and C#. We'll start by creating an ASP.NET MVC blank website in Visual Studio (we're using Visual Studio 2012, and ASP.NET MVC 4, but you can use any .NET solutions that can communicate via HTTP). We'll name the project: `LBM.Locator`.

Next, let's add `HomeController` with one action, `Index`:

```
public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Title = "LBM | Locator";
            return View();
        }
}
```

In this listing we specify a title for the web page, and then return `View`, which looks like:

```
<div id="lbm-map"></div>
```

The listing consists of a container for our map. As specified in the `_ViewStart.chtml` file (inside the `Views` folder), our `Index` view is included within a layout, named `_Layout.chtml` (in the `Views/Shared` folder):

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
</head>
<body>
    @RenderBody()
    <script src="http://ecn.dev.virtualearth.net/mapcontrol/
mapcontrol.ashx?v=
  7.0"></script>
     <script src="/scripts/jquery-2.0.3.min.js"></script>
    @Scripts.Render("~/bundles/application")
</body>
</html>
```

The listing above is a standard ASP.NET MVC layout, with the `Content/css` indicating the stylesheet, and `bundles/applications`, grouping the JavaScript files that will be included with the page. We also add a call to the Bing Maps AJAX Control toolkit, which will generate the map.

We use `BundleConfig` to include the stylesheet and the scripts:

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new
          ScriptBundle("~/bundles/application").
          Include("~/Scripts/app.js"));
        bundles.Add(new
          StyleBundle("~/Content/css").Include
          ("~/Content/app.css"));
    }
}
```

Let's create a `app.js` file and save it in the `Scripts` folder:

```
(function () {

  var lbm = window.lbm = {};

  lbm.App = function () {
    this._map = new
      Microsoft.Maps.Map(document.getElementById('lbm-map'), {
        credentials: '[YOUR BING MAPS KEY]',
        center: new Microsoft.Maps.Location(48.856930, 2.341200),
          // Paris
        zoom: 12,
        showBreadcrumb: false,
        showDashboard: false
      });
  };

  var app = new lbm.App();

})(this);
```

Here, we create a JavaScript object to hold our map initialization, under the `lbm` namespace. The map is then set up as in previous chapters, but now is centered in Paris.

At the end we create an instance of our `App` object, and since the code is placed inside a self-calling function, it will be called as soon as it loads on the page.

We copy the `learningTheme.js` file created in the previous chapter, into the `Scripts` folder and the `learningTheme.css` into the `Content` folder. We make sure we change the reference to the stylesheet within the `learningTheme.js` file (inside `_addCss` method) to point at `Content/learningTheme.css`.

To register the `LearingTheme` module, we write a generic method into our `App`'s prototype:

```
lbm.App.prototype = {
  _addModule: function (name, path, callback) {
    Microsoft.Maps.registerModule(name, path);
    Microsoft.Maps.loadModule(name, {
      callback: callback
    });
  }
};
```

The code groups the `registerModule` and `loadModule` messages of the `Microsoft.Maps` object into a single method, to keep our code DRY (Don't Repeat Yourself – a popular principle in software development). Using this method, we add the following lines to our `App` constructor:

```
var self = this;
this._addModule('lbm.LearningTheme',
  'Scripts/learningTheme.js', function () {
  var learningTheme = new lbm.LearningTheme(self._map);
});
```

If we run our application on debug now (by pressing *F5*), we see the following screenshot:

-
+
Aerial
Clichy
Courbevoie
Levallois-Perret
Pantin
Puteaux
Neuilly-sur-Seine
Le Pré-St.-Gervais
Allée de Longchamp
Bois de Boulogne
Paris
Boulogne-Billancourt

Bing Maps REST resources

RESTful services provide their data in form of **resources** accessible via a URL. The Bing Maps Location API responds with location resources, whereas the other API responses, in JSON, or XML format, contain imagery, route, or traffic information.

The fields returned by the API are quite large, and need to be serialized in other programming languages, such as JavaScript, C# or Ruby. In .NET we can use a collection of `DataContracts` that correspond to the JSON format of the REST APIs (the latest version of them can be found at `http://msdn.microsoft.com/en-us/library/jj870778.aspx`). We will also use them in our project, so let's copy them to a `JsonDataContracts.cs` file inside the Models folder.

Now, it's time to write our proxy class that will communicate with Bing Maps; we'll place it inside the Models folder:

```
public class LocatorProxy
    {
        private const string MAP_KEY = "[YOUR BING MAPS KEY]";
        private const string REST_URL =
          "http://dev.virtualearth.net/REST/v1/Locations/";

        public static void QueryByPoint
```

```
          (double latitude, double longitude, Action<Response>
          callback)
        {
            var uri = new
              Uri(string.Format("{0}/{1},{2}?inclnb=1&key={3}",
              REST_URL, latitude, longitude, MAP_KEY));
            var wc = new WebClient();
            using (var stream = wc.OpenRead(uri))
            {
                if (stream != null)
                {
                    var serializer = new
                      DataContractJsonSerializer
                      (typeof(Response));
                    var response = serializer.ReadObject(stream)
                      as Response;
                    callback(response);
                }
            }
        }
    }
```

Our class has one asynchronous method, `QueryByPoint`, which accepts coordinates as arguments, together with a callback function. First, it builds the URL that the Location API understands, with the coordinates and the Bing Maps Key. It also specifies that the results should include neighborhood data by setting `inclb` as `1`.

Next, we use the new .NET 4.5 `WebClient` class, which makes it easy to read a URL via HTTP. We pass the stream opened by the client to the `DataContractJsonSerializer` class, which converts the JSON string into a `Response` object of a `DataContract` type.

> `DataContractJsonSerializer` is one of the three most common ways for serializing JSON in .NET. Another one is Json.NET, accepted to be faster and more resourceful then the first (you can install Json.NET via NuGet, more information can be found at `http://james.newtonking.com/projects/json-net.aspx`). The third method is via `JavaScriptSerializer`, a .NET library used for example on ASP.NET MVC to generate `JsonResult`.

In order to use `DataContractJsonSerializer` in our project, we need to reference two libraries:

- System.Runtime.Serialization
- System.ServiceModel.Web

Now let's write the controller action that will call our proxy method.

```
public JsonResult QueryByPoint(double latitude, double longitude)
{
    var result = new List<string>();
    LocatorProxy.QueryByPoint(latitude, longitude, x =>
    {
        foreach (var resourceSet in x.ResourceSets)
        {
            foreach (var resource in resourceSet.Resources)
            {
                var description = GetDescription(resource as
                  Location);
                if (description.Length > 1)
                  result.Add(description);
            }
        }
    });
    if (result.Count == 0) result.Add("No results found.");
    return Json(new { description = String.Join("<br/>", result)
      }, JsonRequestBehavior.AllowGet);
}
```

When the proxy returns the response from the Bing Maps servers, we loop through the `ResourceSets` and their `Resources`; the first resource found is the location resource we need, so we extract some information from it through the following code:

```
private static string GetDescription(Location location)
{
    var description = "";
    if (location != null && location.Address != null)
    {
        if
          (!String.IsNullOrWhiteSpace
          (location.Address.FormattedAddress))
        {
            description = String.Format("- {0}",
              location.Address.FormattedAddress);
        }
        if (!String.IsNullOrWhiteSpace(location.Address.Landmark))
        {
            description = String.Format("{0} - {1}", description,
              location.Address.Landmark);
        }
    }
    return description;
}
```

Here, we navigate through the object graph of `JsonDataContracts` to fetch some information about the address and the landmark of the place located at the coordinates provided to the controller action.

This action will be triggered when the user clicks on the map, so let's build the AJAX Control module that will handle it.

```
(function () {
  var lbm = window.lbm || {};
  lbm.Locator = function (map, $) {
    this._map = map;
    this.$ = $;
    this._bind(this);
  };
  lbm.Locator.prototype = {   };
  Microsoft.Maps.moduleLoaded('lbm.Locator');
})();
```

By now we must be accustomed to our module structure above, residing in the `lbm` namespace. Note how we also pass a `$` object, which will handle the AJAX requests. We initialize our module inside the `app.js`:

```
this._addModule('lbm.Locator', 'Scripts/locator.js', function () {
        var locator = new lbm.Locator(self._map, jQuery);
      });
```

It is seen as good practice to limit the dependencies on external libraries in our modules, with jQuery being a rare exception. jQuery is a popular JavaScript library that normalizes HTML document traversal and manipulation, AJAX and events handling, and so on across browsers.

The binding function looks like the following code:

```
_bind: function (self) {
  Microsoft.Maps.Events.addHandler(this._map, 'click', function
    (e) {
    var target = e.originalEvent.target;
    if (target.tagName === 'IMG') {
      self._onmapclick(e);
    }
}
```

In this listing, we attach the click event only to the image tiles of the map, avoiding the buttons we created with the LearningTheme module. We then add the handler to our module's prototype:

```
_onmapclick: function (e) {
  var location = this._getLocation(e),
      self = this,
      options = {
      url: '/home/querybypoint',
      data: { latitude: location.latitude, longitude:
        location.longitude },
      success: function (result) {
        self._showPlaces(result, location);
      }
    };
  this._showLoading(location);
  this._getRemote(options);
  e.handled = true;
}
```

First, we fetch the location from the clicked point of the map, as in the previous chapter, sending the `tryPixelLocation` message to the map.

```
_getLocation: function (e) {
  var point = new Microsoft.Maps.Point(e.getX(), e.getY());
  return this._map.tryPixelToLocation(point);
}
```

Then, we prepare an options object, which will be passed to the `jQuery.ajax` method. The URL option points at our `HomeController` action we wrote earlier, and the data is formed of the location coordinates.

We also add a loading message when the app fetches the data from the server. We do this inside an `Infobox` class of the `Microsoft.Maps` API:

```
_showLoading: function (location) {
  this._pushpin.setLocation(location);
  this._pushpin.setOptions({
    visible: true
  });

  this._infobox.setLocation(location);
  this._infobox.setOptions({
    title: this._round(location.latitude) + ',' +
      this._round(location.longitude),
    description: 'Querying locator ...',
    visible: true
  });
}
```

This method uses a `_pushpin` instance variable, declared inside the `_createInfo` function, which we call in the module's constructor:

```
    _createInfo: function () {
      this._addCss();
      this._pushpin = new Microsoft.Maps.Pushpin(new
        Microsoft.Maps.Location(0, 0), {
        icon: '',
        typeName: 'lbm-locator-pin',
        width: 10,
        height: 10
      });
      this._infobox = new Microsoft.Maps.Infobox(new
        Microsoft.Maps.Location(0, 0), {
        title: 'Locator',
        visible: true
      });
      this._map.entities.push(this._pushpin);
      this._map.entities.push(this._infobox);
    }
```

In this method, we first, add the stylesheet link to the header of the document, by calling the following function:

```
    _addCss: function () {
    var link = document.createElement('link');
      link.setAttribute('rel', 'stylesheet');
      link.setAttribute('href', '/Content/locator.css');         var
      head = document.getElementsByTagName('head')[0];
      head.appendChild(link);
    }
```

Then we instantiate the small pushpin with the same class name, as the styles require, `lbm-locator-pin`.

We also declare an instance of `Microsoft.Infobox` and assign it to the module's `_infobox` property, based initially on a 0,0 location (more information about the `Infobox` class can be found at `http://msdn.microsoft.com/en-us/library/gg675208.aspx`).

At the end, the pushpin and the infobox are added to the map's entities, ready to be displayed. We do that on the `_showLoading` function, when passing a location to the `setLocation` method of the pushpin and infobox. We also make the entities visible, and change the title and description of the infobox indicating that data is being downloaded from the server.

When the server comes back to us with the result we prepared from the Locations API, we display it on the infobox:

```
_showPlaces: function (result) {
  this._infobox.setOptions({
    description: result.description
  });
}
```

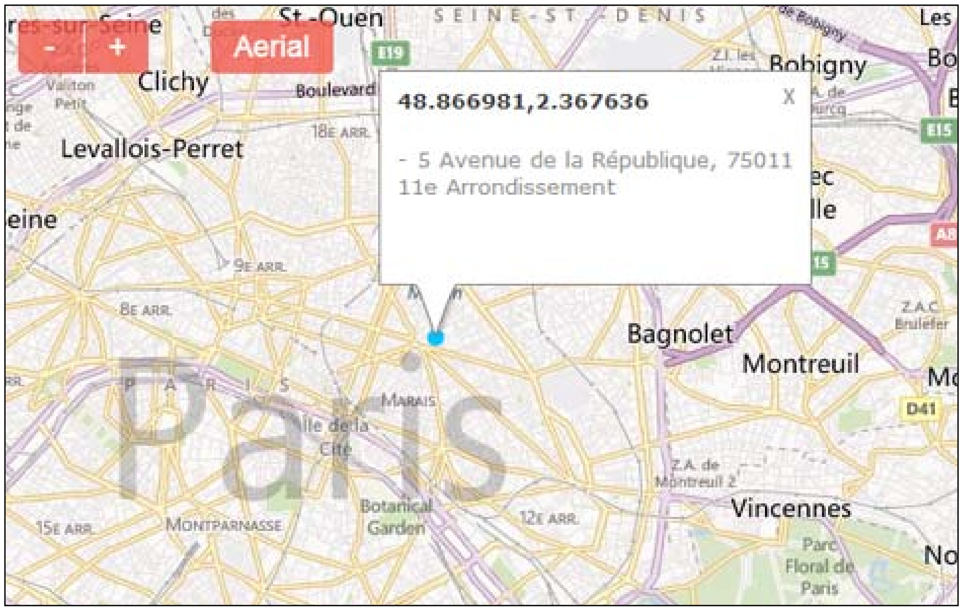
This is another example of how Microsoft Maps API uses the JavaScript object literals to update their objects.

The `locator.css` stylesheet is saved inside the `Contents` folder and turns the pushpins into blue circles:

```
.lbm-locator-pin {
    background: #00bfff;
    border-radius: 50%;
}

#lbm-map > .MicrosoftMap {
    cursor: pointer !important;
}
```

Let's refresh our browser now, and click on the map. Our app will query Bing Maps and come back to us with the address of the point we clicked on. The results will look like as shown in the following screenshot:



48.866981,2.367636

- 5 Avenue de la République, 75011
11e Arrondissement

Summary

In this chapter, we learned that REST services make Bing Maps information accessible through a list of APIs:

- **Locations**: This is used to geocode and reverse-geocode location data.
- **Elevations**: This provides elevation information about locations, polylines, or areas on the earth.
- **Imagery**: This enables creation of static maps, imagery tiles, and imagery metadata.
- **Traffic**: This supplies data regarding traffic issues.
- **Routes**: This gives directions and route information for driving, walking, or using transit.

In the next chapter, we will use the Routes API to build a simple route scheduler for a delivery company in Spain.

4
Diving into Bing Maps REST Services

In the previous chapter, we used the Locations API to build a simple web app. In this chapter, we will dive into another interface of REST Services: the Routes API. We will do that as we help a fictional delivery company in Spain with their scheduling system.

The app consists of an ASP.NET MVC 4 website: the user will enter the names of a few towns, and the app will display the route and the route's information between those towns if it finds one.

The application

Let's create a blank ASP.NET MVC 4 project as we did in the previous chapter, but this time named `LBM.Locator`. Next, we add a `HomeController` class with a single action:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Title = "LBM | Router";
        return View();
    }
}
```

The action sets the title of the website and returns the view. The `Index.chtml` view inside the `Views/Home` folder, contains the same markup as shown in the previous chapter:

```
<div id="lbm-map"></div>
```

This is where we are going to place our map. For this we need to add a reference to the Bing Maps AJAX Control, and we'll do this on the `_Layout.chtml` file, inside the `Views/Shared` folder:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
</head>
<body>
    @RenderBody()
    <script src="/scripts/jquery-2.0.3.min.js"></script>
    <script
      src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.
      ashx?v=7.0"></script>
    @Scripts.Render("~/bundles/application")
</body>
</html>
```

We also linked to the CSS and JavaScript files via the bundle file, `BundleConfig.cs`, inside the `App_Start` folder.

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new
      ScriptBundle("~/bundles/application").Include
      ("~/Scripts/app.js"));
    bundles.Add(new
      StyleBundle("~/Content/css").Include
      ("~/Content/app.css"));
}
```

We copy the `app.cs`, `learningTheme.css`, `app.js`, and `learningTheme.js` files from the previous chapter; we then paste the CSS files into the `Contents` folder, and the JavaScript files into the `Scripts` folder. Let's make the `app.js` file look like the listing below:

```
(function () {
  var lbm = window.lbm = {};

  lbm.App = function () {
    this._map = new
      Microsoft.Maps.Map(document.getElementById('lbm-map'), {
        credentials: '[YOUR BING MAPS KEY],
        showBreadcrumb: false,
        showDashboard: false,
        center: new Microsoft.Maps.Location(40.420289, -3.70567),
          /* Madrid, Spain */
        zoom: 6
    });

    var self = this;
    this._addModule('lbm.LearningTheme',
      'Scripts/learningTheme.js', function () {
      var learningTheme = new lbm.LearningTheme(self._map);
    });
  };

  lbm.App.prototype = {
    _addModule: function (name, path, callback) {
      Microsoft.Maps.registerModule(name, path);
      Microsoft.Maps.loadModule(name, {
        callback: callback
      });
    }
  };

  var app = new lbm.App();
})();
```

The router module

Our app needs to allow users to search by a number of landmarks. We will keep the functionality simple, and let the user enter the town names, separated by a comma, into a textbox. A JavaScript file, `router.js`, which we will save inside the `Scripts` folder, will handle this.

```
(function () {
  var lbm = window.lbm || {};
  lbm.Router = function (map, $) {
    this._map = map;
    this.$ = $;
  };
  Microsoft.Maps.moduleLoaded('lbm.Router');
})();
```

You will recognize by now that this is an AJAX Control module. Like the previous modules, it accepts a `Microsoft.Maps.Map` class instance, which it stores in a private `_map` instance variable.

To incorporate the textbox on the map, we write the following method:

```
_addTextbox: function(self) {
  var html = [
    '<div class="lbm-router-text-wrp">',
      '<input type="text" id="lbm-router-text" placeholder="Enter
        places and press ENTER" value="barcelona,madrid" />',
    '</div>'
  ];
  this._mapContainer.append(html.join(''));
  this._text = this.$('#lbm-router-text').on('keyup', function(e)
    {
    if (e.keyCode === 13) self._fetch();
  }).focus();
}
```

The textbox is wrapped inside a div element, which is then added to the map container. We could not use the header area of the map where the `LearningTheme` module has placed the buttons, as we did in the previous chapters, because that header is inside the map DOM element, and Microsoft.Maps API has disabled all the key related events.

We bind the `keyup` event of the textbox to the `_fetch` handler, only if the *ENTER* key (code 13) is pressed. We also make sure the textbox has the focus initially, so the user can start his/her search immediately.

[The root element of the map is a normal DIV element, which allows us to append to it other DOM elements, styled as per our needs.]

Note how we also added CSS class names to the input wrapper, which we can use to style the textbox:

```
_addCss: function () {
  var link = document.createElement('link');
  link.setAttribute('rel', 'stylesheet');
  link.setAttribute('href', '/Content/router.css');
  var head = document.getElementsByTagName('head')[0];
  head.appendChild(link);
}
```

In the listing above we, append a link to the `router.css` stylesheet, which contains the styles needed for the router. Let's create this file inside the `Content` folder with the following content:

```
.lbm-router-text-wrp {
    position: absolute;
    top: 1em;
    left: 1.9em;
    width: 24em;
}

.lbm-router-text-wrp input[type=text] {
    width: 100%;
    font-size: 1.1em;
    padding: .1em;
    border: 2px solid #F04C40;
}
```

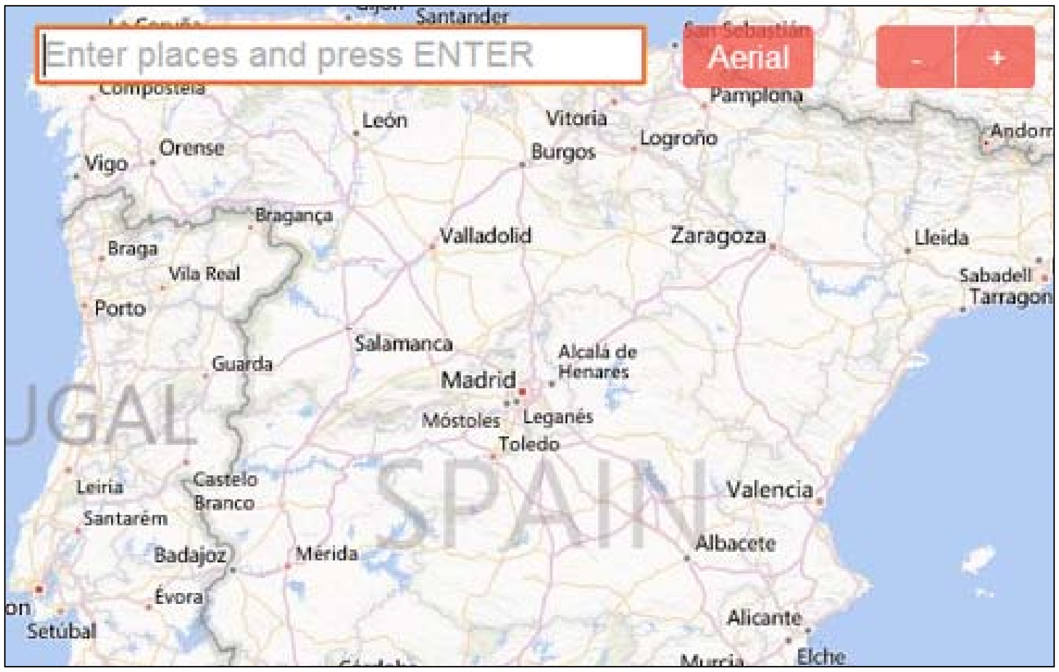
To maintain the same style of the header, we give the textbox a pinkish border. We also position the textbox to the left of the screen, so we move the `LearningTheme` header to the right by making this change to the `learningTheme.css` file:

```
.lbm-map-header {
    position: absolute;
    top: 1em;
    right: 1em;
    color: #fff;
    z-index: 100;
}
```

In order to apply the styled content when the Router module loads, we add the following lines to the module's constructor:

```
this._mapContainer = this.$('#lbm-map');
this._addCss();
this._addTextbox(this);
```

When we run our website in debug mode, our page looks like the following screenshot on the browser:



When you type in the text field now, nothing will happen, because we have yet to handle the keystrokes. Let's write the code that accomplishes that:

```
_fetch: function () {
  this._addContainer();
  this._showLoading();
  var self = this,
    options = {
      url: '/routes',
      data: { places: this._text.val() },
      success: function (result) {
        self._showRoute(result);
      }
    };
  this._getRemote(options);
}
```

First, we append a DOM element that will display the route information to the map's container:

```
  _addContainer: function () {
    if (!this._container) {
      this._container = this.$('<div class="lbm-router"></div>')
        .appendTo(this._mapContainer);
    }
  }
```

Then, we show a loading message as we are waiting for the data from the server:

```
  _showLoading: function() {
    this._container.html('Loading ...');
  }
```

The `options` variable is a just an object accepted by the `$.ajax` function (which we hide by means of our `_getRemote` method). It indicates that the server responds to the `/routes` URL, which we will implement now.

The controller

Let's add `RoutesController` in the `Controllers` folder with a single action:

```
public class RoutesController : Controller
{
    public ActionResult Index(string places)
         {
             RouteInfo route = null;
             var placesArray = places.Split(',');
             if (placesArray.Length > 1)
             {
                 LocatorProxy.QueryRoute(placesArray, x =>
                      {
                          route = x;
                      });
             }
             return View(route);
         }}
```

The model

The `RouteInfo` model holds the route information, including the coordinates of the locations that make up that route.

```
public class RouteInfo
{
    public Route Route { get; private set; }

    public string RoutePath
    {
        get
        {
            if (Route != null)
            {
                return String.Join(",", Route.RoutePath.Line.
Coordinates.Select(c => String.Join(" ", c)));
            }
            return "";
        }
    }
    public string Pins { get; private set; }

    public RouteInfo(Resource resource)
    {
        Route = resource as Route;
        GeneratePins();
    }
}
```

As specified in `JsonDataContracts`, we introduced in the previous chapter, we extract the route's locations from the `Route.RoutePath.Line.Coordinates` object graph.

We also need to show pushpins at the beginning of each route leg, and at the destination:

```
private void GeneratePins()
{
    var pins = new List<string>();
    for (var i = 0; i < Route.RouteLegs.Length; i++)
    {
        var routeLeg = Route.RouteLegs[i];
        if (i == 0) pins.Add(String.Join(" ", routeLeg.ActualStart.
Coordinates));
        pins.Add(String.Join(" ", routeLeg.ActualEnd.Coordinates));
    }
```

```
        Pins = String.Join(",", pins);
    }
```

For this, we use the `Coordinates` pair of `ActualStart` and `ActualEnd` of each `RouteLeg` object.

The proxy

The `LocatorProxy` class is similar to the class with the same name we wrote in the previous chapter; however, this time it generates a URL that points to the Routes API:

```
public class LocatorProxy
{
    private const string MAP_KEY = "[YOUR BING MAPS KEY]";
    private const string REST_URL =
      "http://dev.virtualearth.net/REST/V1/Routes/
      Driving?rpo=Points";
}
```

The Routes API URL template expects a set of parameters, including (see the full list at `http://msdn.microsoft.com/en-us/library/ff701717.aspx`):

- `waypoint.n (or wp.n)`: required, is a list of sequential locations that define the route, such as addresses, landmarks, or geographical points
- `viaWaypoint.n (or vwp.n)`: optional, specifies intermediate locations between two waypoints
- `avoid`: optional, indicates the road types (highways, tolls) to minimize or avoid when a driving route is constructed

Now, let's write the method that communicates with the API:

```
public static void QueryRoute(string[] places, Action<RouteInfo>
callback)
{
    try
    {
        var placesPart = String.Join("&", places.Select((p, i) =>
String.Format("wp.{0}={1}", i, p)));
        var uri = new Uri(string.Format("{0}&key={1}&{2}", REST_URL,
MAP_KEY, placesPart));
        Query(uri, callback);
    }
    catch (Exception)
    {
        callback(null);
    }
}
```

First, we join the places by attaching the `wp.n` parameter key. Next, we add the resulting string to the URL template, together with the Bing Maps Key.

The query is created in the following method:

```
public static void Query(Uri uri, Action<RouteInfo> callback)
{
    RouteInfo routeInfo;
    var wc = new WebClient();
    using (var stream = wc.OpenRead(uri))
    {
        if (stream != null)
        {
            var serializer = new DataContractJsonSerializer(typeof(Re
sponse));
            var response = serializer.ReadObject(stream) as Response;
            foreach (var resourceSet in response.ResourceSets)
            {
                foreach (var resource in resourceSet.Resources)
                {
                    routeInfo = new RouteInfo(resource);
                    callback(routeInfo);
                }
            }
        }
    }
}
```

The listing above is similar to the one we wrote in the previous chapter. It reads the URI directed at API into a `Stream` instance, which is then serialized into a `Response DataContract` object. Looping through its resources, we build the `RouteInfo` model, and pass it to the callback function. `RoutesController`, which specified the callback function, sends the model to the `Index.chtml` view (inside the `Views/Routes` folder).

The view

The view displays the route information, including the pins' and route's coordinates:

```
@if (Model == null)
{
    <h3>Cannot find route</h3>
}
else
{
    <div>
```

```
        <div class="info">
            Directions
            <span>@Model.Route.TravelDistance.ToKm(), @Model.Route.
TravelDuration.ToHoursMin()</span>
        </div>
        @foreach (var routeLeg in Model.Route.RouteLegs)
        {
            <div class="route-leg">
                <h3>@routeLeg.StartLocation.Name - @routeLeg.
EndLocation.Name</h3>
                <ol class="itinerary-items">
                @foreach (var itineraryItem in routeLeg.
ItineraryItems)
                {
                    <li class="itinerary-item">
                        @itineraryItem.Instruction.Text
                        <p>@itineraryItem.TravelDistance.ToMeters(), @
itineraryItem.TravelDuration.ToHoursMin()</p>
                    </li>
                }
                </ol>
            </div>
        }
    </div>

    <div id="lbm-router-pins" style="display: none;">@Model.Pins</div>
    <div id="lbm-router-route-path" style="display: none;">@Model.
RoutePath</div>

}
```

We show the travel distance and duration for the whole route, and each one of its legs, as well as itinerary instructions. The former have been transformed into a human format by using helpers defined in `RoutesHelper.cs` file in a `Helpers` folder we add to the project.

Displaying the route

When the server returns the view to the browser, our Router module shows it on the page:

```
_showRoute: function (result) {
  this._container.html(result);
  this._map.entities.clear();
  this._showRouteLine();
```

```
      this._showPins();
    }
```

First, we replace the container inner HTML with the View, and then we show the route line and pins:

```
    _showRouteLine: function () {
      var locations = this._parseLocations('lbm-router-route-path'),
          options = {};
      if (locations.length) {
        var polyline = new Microsoft.Maps.Polyline(locations);
        this._map.entities.push(polyline);
        options = {
          bounds: Microsoft.Maps.LocationRect.fromLocations(locations)
        };
      } else {
        options = {
          center: this._initialCenter,
          zoom: this._initialZoom
        };
      }
      this._map.setView(options);
    }
```

For the sake of simplicity, we transfer the coordinates of the locations inside the view, when normally we would use other data exchange methods, such as JSON or XML.

```
    _parseLocations: function(dataId) {
      var line = this.$('#' + dataId).text();
      if (!line.length) return [];
      var parts = line.split(','),
        locations = [],
        length = parts.length,
        i = 0;
      for (; i < length; i++) {
        var coords = parts[i].split(' ');
        locations.push(new Microsoft.Maps.Location(coords[0], coords[1]));
      }
      return locations;
    }
```

The coordinates are stored as a comma-delimited string of pairs of latitude and longitude, where the latter two are separated by a space.

We use the coordinates extracted from the `lbm-router-route-path` DIV to build the route line (as an array of `Microsoft.Maps.Location` instances), and add it to the map's entities. We also set the map's bounds within the route line's bounding box by sending the `fromLocations` method to the `LocationRect` object. If our app cannot find a route, it centers the map to its original center and zoom level. The latter are saved in the module's constructor, using methods of the map instance:

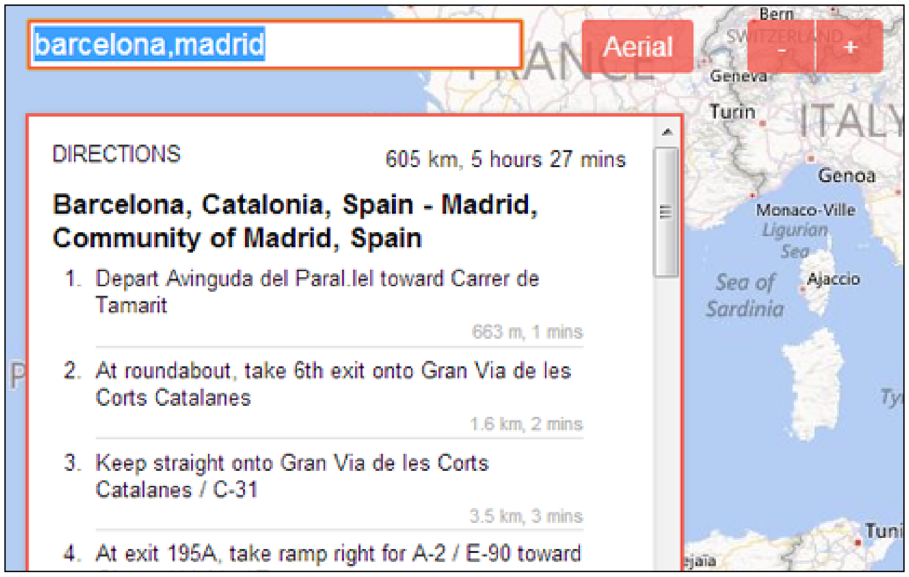
```
this._initialCenter = this._map.getCenter();
this._initialZoom = this._map.getZoom();
```

Now, let's write the code that displays the pins:

```
_showPins: function () {
  var locations = this._parseLocations('lbm-router-pins');
  if (!locations.length) return;
  for (var i = 0, length = locations.length; i < length; i++) {
    this._map.entities.push(new Microsoft.Maps.Pushpin(locations[i], {
      text: this._letters[i]
    }));
  }
}
```

The `_letters` variable is an array of letters: `['A', 'B', 'C', 'D', 'E']`, that will be used as the texts of the pushpins.

If we build our project and refresh the browser now, we see our complete page (we use Spanish landmarks for testing, such as Barcelona, Madrid, and Valencia):



DIRECTIONS 605 km, 5 hours 27 mins

Barcelona, Catalonia, Spain - Madrid, Community of Madrid, Spain

1. Depart Avinguda del Paral.lel toward Carrer de Tamarit
   663 m, 1 mins
2. At roundabout, take 6th exit onto Gran Via de les Corts Catalanes
   1.6 km, 2 mins
3. Keep straight onto Gran Via de les Corts Catalanes / C-31
   3.5 km, 3 mins
4. At exit 195A, take ramp right for A-2 / E-90 toward

Summary

The REST Services provide a rich set of interfaces to the Bing Maps data by means of simple URL templates. They can be consumed using front-end technologies like JavaScript or server languages, such as, .NET, Ruby, PHP, and Python. In this chapter, we explored the Route API, and showed on a map route line and information between towns.

In the next chapter, we will start a new topic, the Spatial Data Services, which we'll use to geocode places of interest automatically.

5
Spatial Data Services

So far we have worked with relatively small sets of data; for larger collections, Bing Maps provide Spatial Data Services. They offer the Data Source Management API to load large datasets into Bing Maps servers, and the Query API to query the data.

In this chapter, we will use the Geocode Dataflow API, which provides geocoding and reverse geocoding of large datasets. Geocoding is the process of finding geographic coordinates from other geographic data, such as, street addresses or postal codes. Reverse geocoding is the opposite process, where the coordinates are used to find their associated textual locations, such as, addresses and postal codes. Bing Maps implement these processes by creating jobs on Bing Maps servers, and querying them later. All the process can be automated, which is ideal for huge amounts of data.

Please note that strict rules of data usage apply to the Spatial Data Services (please refer `http://msdn.microsoft.com/en-us/library/gg585136.aspx` for full details). At the moment of writing, a user with a basic account can set up to 5 jobs in a 24-hour period.

Our task in this chapter is to geocode the addresses of ten of the biggest technology companies in the world, such as Microsoft, Google, Apple, Facebook, and so on, and then display them on the map. The first step is to prepare the file with the companies' addresses.

Geocoding dataflow input data

The input and output data can be supplied in the following formats:

- XML (content type application/xml)
- Comma separated values (text/plain)
- Tab-delimited values (text/plain)
- Pipe-delimited values (text/plain)
- Binary (application/octet-stream) used with Blob Service REST API

We will use the XML format, for its clearer declarative structure.

Now, let's open Visual Studio and create a new C# Console project named LBM. Geocoder. We then add a `Data` folder, which will contain all the data files and samples with which we'll work in this chapter, starting with the `data.xml` file we need to upload to the Spatial Data servers to be geocoded.

```
<?xml version="1.0" encoding="utf-8"?>
<GeocodeFeed
  xmlns="http://schemas.microsoft.com/search/local/2010/5/geocode"
  Version="2.0">
  <GeocodeEntity Id="001"
    xmlns="http://schemas.microsoft.com/search/local/
    2010/5/geocode">
    <GeocodeRequest Culture="en-US" IncludeNeighborhood="1">
      <Address AddressLine="1 Infinite Loop"
        AdminDistrict="CA" Locality="Cupertino" PostalCode="95014" />
    </GeocodeRequest>
  </GeocodeEntity>
  <GeocodeEntity Id="002"
    xmlns="http://schemas.microsoft.com/search/local/2010/5/
    geocode">
    <GeocodeRequest Culture="en-US" IncludeNeighborhood="1">
      <Address AddressLine="185 Berry St"
        AdminDistrict="NY" Locality="New York" PostalCode="10038"
        />
    </GeocodeRequest>
  </GeocodeEntity>
```

The listing above is a fragment of that file with the addresses of the companies' headquarters. Please note that the more addressing information we provide to the API, the better quality geocoding we receive. In production, this file would be created programmatically, probably based on an Addresses database. The ID of `GeocodeEntity`, could also be stored, so that the data is matched easier once fetched from the servers. (You can find the Geocode Dataflow Data Schema, Version 2, at `http://msdn.microsoft.com/en-us/library/jj735477.aspx`.)

The job

Let's add a `Job` class to our project:

```
public class Job
{
    private readonly string dataFilePath;
    public Job(string dataFilePath)
    {
        this.dataFilePath = dataFilePath;
    }
}
```

The `dataFilePath` argument is the path to the `data.xml` file we created earlier.

Creating the job is as easy as calling a REST URL:

```
public void Create()
        {
            var uri =
              String.Format("{0}?input=xml&output=xml&key={1}",
                Settings.DataflowUri, Settings.Key);

            var data = File.ReadAllBytes(dataFilePath);
            try
            {
                var wc = new WebClient();
                wc.Headers.Add("Content-Type", "application/xml");
                var receivedBytes = wc.UploadData(uri, "POST",
                  data);
                ParseJobResponse(receivedBytes);
            }
            catch (WebException e)
            {
                var response = (HttpWebResponse)e.Response;
                var status = response.StatusCode;
            }
        }
```

We place all the API URLs and other settings in the `Settings` class:

```
public class Settings
{
    public static string Key = "[YOUR BING MAPS KEY];
    public static string DataflowUri =
      "https://spatial.virtualearth.net/REST/v1/dataflows/
        geocode";
    public static XNamespace XNamespace =
      "http://schemas.microsoft.com/search/local/ws/rest/v1";
    public static XNamespace GeocodeFeedXNamespace =
      "http://schemas.microsoft.com/search/local/2010/5/geocode";
}
```

To create the job, we need to build the Dataflow URL template with a Bing Maps Key, and parameters such as input and output formats. We specify the latter to be XML.

Next, we use a `WebClient` instance to load the data with a POST protocol. Then, we parse the server response:

```
private void ParseJobResponse(byte[] response)
    {
        using (var stream = new MemoryStream(response))
        {
            var xDoc = XDocument.Load(stream);
            var job = xDoc.Descendants(Settings.XNamespace +
              "DataflowJob").FirstOrDefault();
            var linkEl = job.Element(Settings.XNamespace +
              "Link");
            if (linkEl != null) Link = linkEl.Value;
        }
    }
```

Here, we pass the stream created with the bytes received from the server to the `XDocument.load` method. This produces an `XDocument` instance, which we will use to extract the data we need. We will apply a similar process throughout the chapter to parse XML content. Note that the appropriate `XNamespace` needs to be supplied in order to navigate through the document nodes.

You can find a sample of the response inside the `Data` folder (`jobSetupResponse.xml`), which shows that the link to the job created is found under a `Link` element within the `DataflowJob` node.

Getting job status

Once we have set up a job, we can store the link on a data store, such as a database, and check for its status later. The data will be available on the Microsoft servers up to 14 days after creation.

Let's see how we can query the job status:

```
public static JobStatus CheckStatus(string jobUrl)
{
    var result = new JobStatus();

    var uri = String.Format("{0}?output=xml&key={1}", jobUrl,
      Settings.Key);
    var xDoc = XDocument.Load(uri);
    var job = xDoc.Descendants(Settings.XNamespace +
      "DataflowJob").FirstOrDefault();
    if (job != null)
    {
        var linkEls = job.Elements(Settings.XNamespace +
          "Link").ToList();
        foreach (var linkEl in linkEls)
        {
            var nameAttr = linkEl.Attribute("name");
            if (nameAttr != null)
            {
                if (nameAttr.Value == "succeeded") result.
SucceededLink = linkEl.Value;
                if (nameAttr.Value == "failed") result.FailedLink
                  = linkEl.Value;
            }
        }

        var statusEl = job.Elements(Settings.XNamespace +
          "Status").FirstOrDefault();
        if (statusEl != null) result.Status = statusEl.Value;
    }

    return result;
}
```

Now, we know that to query a Data API we need to first build the URL template. We do this by attaching the Bing Maps Key and an output parameter to the job link.

The response we get from the server, stores the job status within a `Link` element of a `DataflowJob` node (the `jobResponse.xml` file inside the `Data` folder contains an example). The link we need has a `name` attribute with the value `succeeded`.

Getting job results

Once the job is complete, it's time to fetch its results. How are we going to do this? You guessed it right: we need to make another call to the Dataflow API, using the link we extracted earlier.

```
public static void GetResult(string dataFlowLink, string
responseFilePath)
{
    var settings = new XmlWriterSettings();
    settings.Indent = true;

    using (var writer = XmlWriter.Create(responseFilePath, settings))
    {
        var uri = String.Format("{0}?output=xml&key={1}",
dataFlowLink, Settings.Key);
        var xdoc = XDocument.Load(uri);
        xdoc.WriteTo(writer);
    }
}
```

As before, we add our Bing Maps Key to the URL template, and specify XML as output. We use `XDocument` again to fetch the data, which we then store on a file inside the `Data` folder. We can then store the data in a database, or some other data store, and then match the requested addresses with their corresponding responses.

The job's result is as follows (a sample file, `dataResponse.xml`, can be found inside the `Data` folder):

```
<?xml version="1.0" encoding="utf-8"?>
<GeocodeFeed Version="2.0" xmlns="http://schemas.microsoft.com/search/
local/2010/5/geocode">
  <GeocodeEntity Id="001" xmlns="http://schemas.microsoft.com/search/
local/2010/5/geocode">
    <GeocodeRequest Culture="en-US" IncludeNeighborhood="true">
      <Address AddressLine="1 Infinite Loop" AdminDistrict="CA"
Locality="Cupertino" PostalCode="95014" />
    </GeocodeRequest>
    <GeocodeResponse Name="1 Infinite Loop, Cupertino, CA 95014"
EntityType="Address" Confidence="High" MatchCodes="Good">
      <Address AddressLine="1 Infinite Loop" AdminDistrict="CA"
CountryRegion="United States" AdminDistrict2="Santa Clara
Co." FormattedAddress="1 Infinite Loop, Cupertino, CA 95014"
Locality="Cupertino" PostalCode="95014" />
      <GeocodePoint CalculationMethod="InterpolationOffset"
Latitude="37.3318473994732" Longitude="-122.030806615949" Type="Point"
UsageTypes="Display" />
```

```
      <GeocodePoint CalculationMethod="Interpolation" 
Latitude="37.3318473994732" Longitude="-122.030750289559" Type="Point" 
UsageTypes="Route" />
      <BoundingBox SouthLatitude="37.3279846819025" 
WestLongitude="-122.03728352294" NorthLatitude="37.3357101170439" 
EastLongitude="-122.024329708958" />
      <Point Latitude="37.3318473994732" Longitude="-122.030806615949" 
/>
    </GeocodeResponse>
    <StatusCode>Success</StatusCode>
    <TraceId>d9a530c28edd4491bd642a1967c64f5d|SINM000009|02.00.161.180
0|SINMSNVM000145, SINIPEVM000011</TraceId>
  </GeocodeEntity>
 </GeocodeFeed>
```

Note how for each entity, we also get the request object we sent to the server upon job creation. Now, let's parse this file.

Parsing the geocode response

We'll start by adding another class to our project, named `SpatialParser`.

```
public class SpatialParser
{
    public SpatialParser(string dataflowUri)
    {
        this.dataflowUri = dataflowUri;
    }
}
```

This class will parse the geocoded data and will update the companies' records. In real life the latter would be stored in a database or other data store, but here we'll save it in memory. Thus, we create `Company`:

```
public class Company
{
    public string Id { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
    public GeocodePoint Point { get; set; }

    public Company(string id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

And a `Companies` class that will operate as the data store:

```
public class Companies
{
    private static IDictionary<string, Company> companies;
    public static IDictionary<string, Company> All
    {
        get
        {
            if (companies == null)
            {
                companies = new Dictionary<string, Company>();
                companies.Add("001", new Company("001", "Apple"));
                companies.Add("002", new Company("002", "Stack
                  Exchange"));
                companies.Add("003", new Company("003",
                  "Github"));
                companies.Add("004", new Company("004",
                  "Microsoft"));
                companies.Add("005", new Company("005",
                  "Paypal"));
                companies.Add("006", new Company("006",
                  "Mozilla"));
                companies.Add("007", new Company("007",
                  "Google"));
                companies.Add("008", new Company("008",
                  "Facebook"));
                companies.Add("009", new Company("009",
                  "LikedIn"));
                companies.Add("010", new Company("010", "Twitter,
                  Inc."));
            }
            return companies;
        }
    }
}
```

Now, let's parse the XML document we saved earlier, and update our companies with the coordinates of their headquarters:

```
public void Parse()
{
    var xDoc = XDocument.Load(dataflowUri);
    var entities = xDoc.Descendants(Settings.GeocodeFeedXNamespace +
"GeocodeEntity");
    foreach (var entity in entities)
```

```
        {
            if (!IsSuccessful(entity)) continue;

            var company = GetCompany(entity);
            if (company == null) continue;

            var geocodeResponseEl = Element(entity, "GeocodeResponse");
            if (geocodeResponseEl == null) continue;

            var address = GetAddress(geocodeResponseEl);
            if (address == null) continue;

            var geocodePoints = GetPoints(geocodeResponseEl);
            if (geocodePoints.Count == 0) continue;

            company.Address = address;
            company.Point = geocodePoints[0];

        }
    }
```

We loop through the entities, and extract information from entities using `StatusCode` with value `Success`:

```
    private static bool IsSuccessful(XContainer element)
            {
                var statusEl = Element(element, "StatusCode");
                return statusEl != null && statusEl.Value ==
                  "Success";
            }
```

In production we would have a complete checking system in place that would log different scenarios of failed attempts. Such systems can also try geocoding the data again in case of failure, flag certain addresses as not existent, and so on.

Websites

Now, let's add a basic `ASP.NET MVC` project to our VS solution named `LBM.Geocoder.Web`. The website is very similar to the one we built in the previous chapter, so we can copy `HomeController.cs`, `_Layout.cshtml`, the `Index.chtml` views, `app.js`, `app.css`, `learningTheme`, and `BundleConfig.cs` from it.

Next, we create a new JavaScript module, `geocoder.js`, and save it inside the `Scripts` folder:

```
lbm.Geocoder = function(map, $) {
    this._map = map;
    this.$ = $;
    this._addCss();
};
lbm.Geocoder.prototype = {
    _addCss: function () {
        var link = document.createElement('link');
        link.setAttribute('rel', 'stylesheet');
        link.setAttribute('href', '/Content/geocoder.css');
        var head = document.getElementsByTagName('head')[0];
        head.appendChild(link);
    }
};
Microsoft.Maps.moduleLoaded('lbm.Geocoder');
```

We also pass a `$` argument to the module, in the shape of an object that will perform AJAX requests (in our case `jQuery`), using the good practice of dependency injection. Furthermore, we add a link to a stylesheet file inside the `Content` folder.

The module will load the data from a URL:

```
load: function(url) {
    var self = this,
        options = {
            url: url,
            success: function(result) {
                self._show(result);
            }
        };
    this.$.ajax(options);
}
```

And show pushpins on a map:

```
_show: function (result) {
    var locations = [],
        self = this;
    $.map(result, function (place, i) {
        var location = new Microsoft.Maps.Location(place.Latitude,
place.Longitude),
            pin = new Microsoft.Maps.Pushpin(location, {
                icon: '',
                typeName: 'lbm-geocoder-pin icon-' + place.Id,
                text: place.Name
            });
        locations.push(location);
        self._map.entities.push(pin);
    });
    this._map.setView({
        bounds: Microsoft.Maps.LocationRect.fromLocations(locations)
    });
}
```

Note how we have removed the default image of the pushpin, and added to it an `icon-` prefixed class with the company ID. We use that to display the company logos on the map, through icon fonts (taken from the excellent IcoMoon App at `http://icomoon.io/app`).

We register and call the module from `app.js`:

```
this._addModule('lbm.Geocoder', 'Scripts/geocoder.js', function ()
  {
    var geocoder = new lbm.Geocoder(self._map, jQuery);
    geocoder.load('/home/places');
});
```

When the module is loaded, it requests data from the `Places` action of `HomeController`.

```
public JsonResult Places()
        {
var dataFilePath = Server.MapPath("/App_Data/dataResponse.xml");
            return Json(Companies.ToJson(dataFilePath),
              JsonRequestBehavior.AllowGet);
        }
```

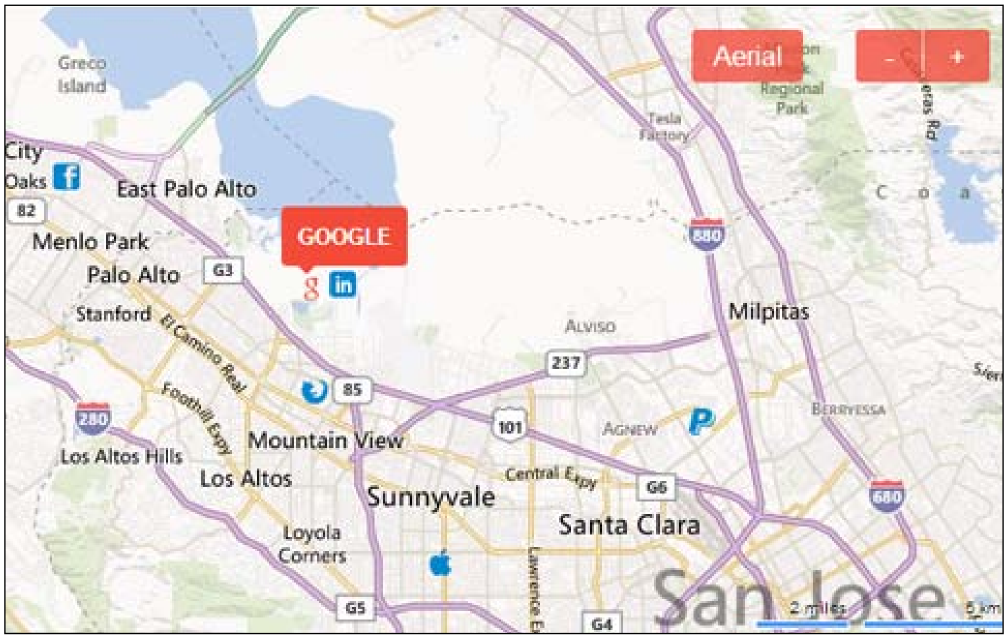
It takes the data from the `Companies` class.

```
public static IList<object> ToJson(string dataFilePath)
{
    var parser = new SpatialParser(dataFilePath);
    parser.Parse();

    var result = new List<object>();
    foreach (var company in All.Values)
    {
        if (company.Address == null || company.Point == null)
          continue;
        result.Add(new
            {
                company.Id,
                company.Name,
company.Point.Latitude,
                company.Point.Longitude
            });
    }
```

The listing above uses `SpatialParser` class to read the XML document and prepare a list of objects that can be easily serialized into a JSON string.

And we're done; let's click on *F5,* and run the website as the start project on a browser. We should see the following screenshot:



When we hover over a company icon, the company name appears on a popup.

Summary

When it comes to large amounts of data, the Spatial Data Services offer a number of interfaces to store, and query user data; geocode addresses or reverse geocode geographical coordinates. The services perform these tasks by means of background jobs, which can be set up and queried through REST URLs.

In the next chapter, we will use an experimental interface of Spatial Data Services, the Geodata API, which we'll use to geocode postcodes.

6
Diving into Spatial Data Services

We have a pretty good idea now of how the APIs to the Bing Maps data work. They are normally URL templates we can query for a single point of data, like a coordinate, or entire sets, as we did with the spatial geocoding. In *Chapter 2*, *Diving into Bing Maps AJAX Control* Version 7, we went to Paris, and when we clicked on the map, the Bing Maps gave us the address of that location.

Now we will go back to the United States and have a look at the Geodata API; it is at the moment in preview mode, which makes it interesting in giving us an insight to the direction of where the Bing Maps are headed.

The Geodata API produces a set of polygons that constitute the boundaries of a geographic area, such as different levels of postcodes, province, or state.

In this chapter, we will build a small web app that shows the postcode of the point where the user clicks on a map. The main structure of the application is the same as in the previous chapters, starting with an ASP.NET MVC website.

The project

Let's create a basic ASP.NET MVC project in Visual Studio named LBM.Geodata. We then copy the `HomeController` from the previous chapter with the `Index` action.

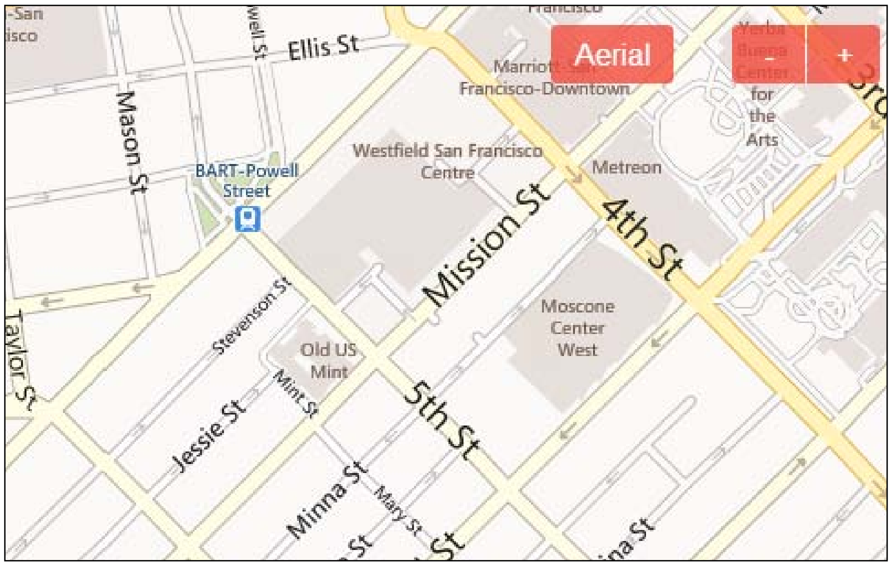
```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Title = "LBM | Geodata";
        return View();
    }
}
```

This returns a simple view, `Index.chtml`, which will contain a single DIV element to hold our map.

```
<div id="lbm-map"></div>
```

We also copy `jquery.js`, `app.js`, `learningTheme.js` and paste them into the `Scripts` folder, as well as `app.css` and `learningTheme.css` in the `Content` folder. Then, we do the same with the `_Layout.cshtml` and `BundleConfig.cs` files, which we paste into `Views/Shared` and `App_Start` folders respectively.

When we run the project in debug mode, we should be presented with the following screenshot on the browser:



Querying for postcodes

First, we create a proxy model to the Geodata API:

```
public class GeodataProxy
{
  private const string API_URL_TEMPLATE =
  "https://platform.bing.com/geo/spatial/v1/public/
    geodata?spatialFilter=GetBoundary({0},{1},1,
    'PostCode1',1,1,'en','us')&key=[YOUR BING MAPS
    KEY]&$format=json";
}
```

Note how the URL here points at a different host to the other APIs, a `platform.bing.com`.

Apart from the Bing Maps Key (this service is also subject to data usage), the following parameters are used to build the URL template (you can get the full list of parameters at `http://msdn.microsoft.com/en-us/library/dn306801.aspx`):

- `latitude`, `longitude` (required): This is a pair of double values between -90 and +90 for the latitude, and between -180 and +180 for the longitude.
- `address` (the coordinates or address are required): This is a string representing an address, as in the previous chapter. This task is performed in two steps; first the address is geocoded, and then the resulting coordinates are sent to the API.
- `levelOfDetail` (required): This is an integer ranging between 0 and 3, where 0 represents the coarsest level of boundary detail, and 3 the best. Our proxy has this value as 1. This and the following parameters are passed to the `GetBoundary` function inside the URL.
- `entityType` (required): This is a string specifying different geographical areas, such as `CountryRegion`, `AdminDivision1`, and `AdminDivision2`, `Postcode1` to `Postcode4`. We stipulate `Postcode1` as the lowest level of postcode in our app.
- `getAllPolygons` (required): This can be 0 or 1; in the first instance only the outline is returned, and in the latter the API returns all polygons. We want all polygons; therefore we put 1 on our URL.
- `getEntityMetadata` (required): This is same as above; specifies whether to return metadata or not.
- `$format` (optional): This is the default format in which data returned is `atom`, but it can also be `json`, and we specify the latter as our preferred format.

Now, let's write the code that will fetch the geodata from Bing Maps.

```
public static string Fetch(double latitude, double longitude)
{
    var result = "";
    var url = string.Format(API_URL_TEMPLATE, latitude,
      longitude);
    var wc = new WebClient();
    var data = wc.DownloadData(url);

    var jObject = JObject.Parse(Encoding.ASCII.GetString(data));
    var shape =
      (string)jObject["d"]["results"][0]["Primitives"][0]
      ["Shape"];
```

```
        var coordinates = new List<Coordinate>();
        if (TryParseEncodedValue(shape.Substring(2), out coordinates))
        {
            result = string.Join(",", coordinates.Select(c =>
              c.ToJson()));
        }

        return result;
    }
```

The method above accepts two arguments of double type, latitude and longitude, as required by the API. We pass them to the URL template, which is then used to download the data by means of a `WebClient` instance.

Because we requested the data to be returned in JSON format, we use `Json.NET` library to parse the response (`Json.NET` is a popular `.NET` library written by *James Newton-King* with a great set of features for parsing and creating JSON. More information can be found at `http://james.newtonking.com/json`.) We use its generic `JObject` to navigate through the JSON graph returned, until we get to the `Shape` node we need. You can get the full description of what each object of the response does at `http://msdn.microsoft.com/en-us/library/dn306801.aspx`).

The `Shape` object is made of two parts, separated by comma: the first is the version number of the polygon, and the second is a compressed list of polygons. Microsoft has provided the Decompression algorithm (found at `http://msdn.microsoft.com/en-us/library/dn306801.aspx`), which we need to copy and paste into our proxy class as a method named `TryParseEncodedValue`.

We pass the second part of the Shape string to this method, retrieving a list of `Coordinate` instances. Let's create the `Coordinate` class inside the `Models` folder:

```
    public class Coordinate
        {
            public double Latitude { get; set; }
            public double Longitude { get; set; }

            public string ToJson()
            {
                return string.Format("{0} {1}", Math.Round(Latitude,
                  4), Math.Round(Longitude, 4));
            }
        }
```

The model contains two properties, `Latitude` and `Longitude`, which make up the coordinates of the `Shape` string in the form of polygon rings. The `ToJson` method rounds the coordinates to four decimal places, shortening the string that will be sent to the client.

> A polygon ring, or a closed path, is a sequence of coordinates (pairs of latitude and longitude) where the first pair is the same as the last one. This is necessary for displaying polygons and polylines in Bing Maps.

The locator

When we show the user the postcode shape of the point he/she clicks on, it is a good idea to also show the name of the postcode. We will use the REST Services to do that; therefore, let's copy the `LocatorProxy.cs` and `JsonDataContracts.cs` files from the project we built in *chapter 3*, *Introduction to Bing Maps REST Services*, and paste them into the `Models` folder.

We also copy the `QueryByPoint` and `GetDescription` methods from `HomController` of that project and change them to look like this:

```
private string QueryByPoint(double latitude, double longitude)
{
    var result = "No results found.";
    LocatorProxy.QueryByPoint(latitude, longitude, x =>
    {
        foreach (var resourceSet in x.ResourceSets)
        {
            foreach (var resource in resourceSet.Resources)
            {
                var postcode = GetPostcode(resource as Location);
                if (postcode.Length > 1)
                {
                    result = postcode;
                    return;
                }
            }
        }

    });
    return result;
}
```

```
private static string GetPostcode(Location location)
{
    var postcode = "";
    if (location != null && location.Address != null)
    {
        postcode = string.Format("{0} {1}", location.Address.
AdminDistrict ?? "", location.Address.PostalCode ?? "");
    }
    return postcode;
}
```

We are only interested on the administrative area and the postcode; therefore, we extract only those parts from the location query.

The controller

We need to write a controller action that fetches the data and returns it in JSON format.

```
public JsonResult Postcode(double latitude, double longitude)
{
    var data = new
        {
            Shape = GeodataProxy.Fetch(latitude, longitude),
            Info = QueryByPoint(latitude, longitude)
        };
    return Json(data, JsonRequestBehavior.AllowGet);
}
```

The data object contains two properties, the `Shape` string of coordinates, and `Info`, with the postcode information. Now, it's time to write the client side.

The client side

As we have done previously, we will organize our client side inside a Microsoft Maps module, which we will name `geodata.js`.

```
(function () {
    var lbm = window.lbm || {};

    lbm.Geodata = function (map, $) {
        this._map = map;
        this.$ = $;
    };

    Microsoft.Maps.moduleLoaded('lbm.Geodata');
})();
```

Next, we create a stylesheet file, `geodata.css`, inside the `Content` folder, and add a link to it in our module, as we did in the previous chapter.

```
    _addCss: function () {
        var link = document.createElement('link');
        link.setAttribute('rel', 'stylesheet');
        link.setAttribute('href', '/Content/geodata.css');
        var head = document.getElementsByTagName('head')[0];
        head.appendChild(link);
    }
```

We would like our app to respond to user clicks, so we add a handler for that event:

```
    _bind: function (self) {
        Microsoft.Maps.Events.addHandler(this._map, 'click', function
          (e) {
            if (e.originalEvent.target.tagName !== 'BUTTON') {
                self._onclick(e);
            }
        });
    }
```

Note how we only handle the clicks on targets other than buttons, to avoid responding to clicks on our header buttons.

We need a way to show the postcode information, and we will use a pushpin for that.

```
    _createPin: function () {
        this._pushpin = new Microsoft.Maps.Pushpin(new Microsoft.Maps.
    Location(0, 0), {
            typeName: 'lbm-geodata-pin',
            visible: false,
            width: 200,
            height: 50,
            anchor: new Microsoft.Maps.Point(110, 50)
        });
        this._map.entities.push(this._pushpin);
    }
```

We could have used an infobox as well, but this way we can explore another option of the `Pushpin` class, `htmlContent`. When this option is set, either upon instantiation of the pushpin, or as an argument to its `setOptions` method, it replaces the HTML generated by Microsoft.Maps API. Our pushpin turns thus into an infobox.

We want the methods we have coded so far to run when the module is loaded; therefore, we add the following lines to the module's constructor:

```
this._addCss();
this._createPin();
this._bind(this);
```

Next, let's code the handler:

```
_onclick: function (e) {
    var self = this,
        location = this._getLocation(e);
    this._showLoadingPin(location);
    this.$.ajax({
        url: '/home/postcode',
        data: { latitude: location.latitude, longitude: location.
longitude },
        success: function (result) {
            self._showResult(result);
        }
    });
}
```

First, we get the location of point clicked by the user:

```
_getLocation: function (e) {
    var point = new Microsoft.Maps.Point(e.getX(), e.getY());
    return this._map.tryPixelToLocation(point);
}
```

Then we show the pushpin with a loading message, while the data is fetched from the server.

```
_showLoadingPin: function (location) {
    this._pushpin.setLocation(location);
    this._pushpin.setOptions({
        visible: true,
        htmlContent: this._pushpinContent('Loading ...')
    });
}
```

The listing above positions the pushpin where the user clicked, and sets its content to **Loading ...**.

```
_pushpinContent: function(text) {
    return '<div class="content">'+ text + '</div>';
}
```

We add a class name, `content`, to our HTML element, which is useful, when we style it.

The stylesheet

We style our pushpin with generous padding, a drop shadow, and an arrow at the bottom:

```
.lbm-geodata-pin .content {
    color: #F04C40;
    position: relative;
    width: 100%;
    background: white;
    padding: 15px;
    border: 1px solid #bbb;
    z-index: 30;
    -webkit-box-shadow: 0 3px 10px #999;
    -moz-box-shadow: 0 3px 10px #999;
    box-shadow: 0 3px 10px #999;
}

.lbm-geodata-pin .content:before {
    position: absolute;
    bottom: -9px;
    left: 110px;
    display: inline-block;
    border-right: 9px solid transparent;
    border-top: 9px solid #fff;
    border-left: 9px solid transparent;
    content: '';
    z-index: 31;
}

.lbm-geodata-pin .content:after {
    position: absolute;
    bottom: -10px;
    left: 109px;
```

```
        display: inline-block;
        border-right: 10px solid transparent;
        border-top: 10px solid #999;
        border-left: 10px solid transparent;
        content: '';
        z-index: 30;
    }
```

Showing the data

Going back to our click event handler, after we show a loading message, we fetch the data through AJAX. Upon success, we show the results on the map.

```
    _showResult: function (result) {
        this._showPin(result.Info);
        this._showShape(result.Shape);
    }
```

In the previous listing, we first show the information on the pushpin:

```
    _showPin: function (text) {
        this._pushpin.setOptions({
            htmlContent: this._pushpinContent(text)
        });
    }
```

Then, we show the shape:

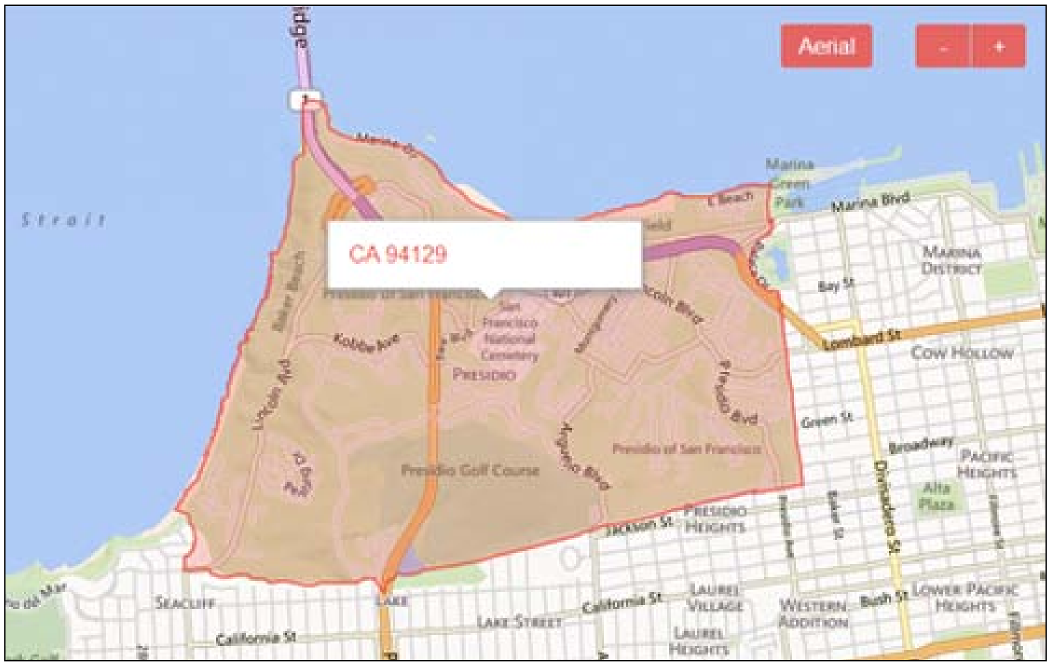
```
    _showShape: function (shapeString) {
        this._shapesLayer.clear();
        var parts = shapeString.split(','),
            length = parts.length,
            i = 0,
            locations = [];
        for (; i < length; i++) {
            var coordinates = parts[i].split(' ');
            locations.push(new Microsoft.Maps.Location(coordinates[0],
              coordinates[1]));
        }
        var shape = new Microsoft.Maps.Polygon(locations, {
            strokeColor: new Microsoft.Maps.Color(200, 240, 76, 64),
            strokeThickness: 2,
            fillColor: new Microsoft.Maps.Color(60, 240, 76, 64),
            emptyFillColor: new Microsoft.Maps.Color(0, 0, 0, 0)
        });
        this._shapesLayer.push(shape);
        this._map.setView({
            bounds:
              Microsoft.Maps.LocationRect.fromLocations(locations)
        });
    }
```

We place the shape in its own layer, so that we can clear the latter without clearing the pushpin. Let's do this in the module's constructor, by adding the following lines:

```
    this._shapesLayer = new Microsoft.Maps.EntityCollection;
    this._map.entities.push(this._shapesLayer);
```

Back to the `_showShape` method, we split the shape string by comma, and then create a `Location` instance from each part. The locations thus created, make up the shape, which we style in similar pinkish style as the rest of the web page. At the end, we add the shape to its layer and set the bounds of the map around the list of locations.

When we run the project in debug mode, it opens a browser window, and when we click on the map, we get something like the following screenshot:



The postcode contour is nicely highlighted, and its name, together with the administrative area, appears on the pop-up box.

Summary

In this chapter we tried an experimental feature of Spatial Data Services, the Geodata API. It provides geographical information of various administrative areas, from the postcode level to an entire region or country. This feature is still in preview mode at the time of this writing, which means that it may not be fully supported on every point of the globe.

In the next chapter we will place our own data on Bing Maps, building a small app on the way, which will take us to London.

7
Enriching Bing Maps with Overlaying User Data

In the previous chapters, we laid on the map data supplied by Bing Maps APIs, but Bing Maps can be a useful tool for displaying custom spatial data too. Examples include residential properties on an estate agency website, best locations in the world to visit, electoral results, custom map tiles, floor plans of venues, and many more.

Our task in this chapter is to build a simple mashup, which consists of a website that shows the last four elections for the London Assembly. This will help us to prepare the data that Microsoft.Maps API understands, and create the shapes based on this data.

The data

The Great Britain's Ordnance Survey provides a product named **Open Data** that offers a wide range of digital map products, which you can freely view or download for use in both personal and commercial applications (as specified in their website available at `http://www.ordnancesurvey.co.uk/business-and-government/products/opendata-products.html`). We need the shapes for the different areas of the London Assembly, and their boundary-line product provides the necessary files.

The latter are in the ESRI Shapefile format, which we need to convert into sequences of coordinates that Microsoft.Maps API understands. The conversion is outside the scope of this book, but if you are interested to see how it was done, you can check the `shape_reader.rb` file—a Ruby script in the code bundle.

The electoral results are supplied by the London Elects website available at `http://www.londonelects.org.uk/im-voter/results-and-past-elections/results-2012`.

At the end of the conversion, our data file looks as follows:

```
var data = [
  {
    "name":"Havering and Redbridge",
    "votes": [0,0,0,0],
    "wkt": [[51.6283,0.0233],[51.6282,0.0235],[51.6278,0
  },
  {
    "name":"Croydon and Sutton",
    "votes": [1,1,1,1],
    "wkt":[[51.321,-0.1551],[51.3235,-0.1572],[51.323
}
  ....
]
```

The objects' specifications are displayed as follows:

- `name`: This specifies the London Assembly seat name
- `votes`: This specifies the ID of the winning party over the years from 2000 to 2012, where 0 indicates the **Conservatives** party, and 1 indicates the **Labour** party.
- `wkt`: This specifies an array representing a closed polygon of the area

The application

Let's start by creating an application folder, named `poll`; then, adding an HTML file named `index.html`, which can be written as follows:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>LBM | London Assembly Elections</title>
    <link href="app.css" rel="stylesheet" />
    <script src="http://ecn.dev.virtualearth.net/mapcontrol/
mapcontrol.ashx?v=7.0"></script>
</head>
<body>
    <div id="lbm-map"></div>

    <script src="jquery-2.0.3.min.js"></script>
    <script src="data.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

As in the previous chapters, we add links to the `app.css` and `app.js` files, the link to the AJAX Control, and the link to the jQuery file. The `DIV` element will serve as the map container. We then copy the `app.css` and `app.js` files from the previous chapters and paste them inside the `poll` folder. We also copy and paste the `learningTheme` folder with our custom map controls.

In the `app.js` file, we change the map options so that the map is now centered in London, as shown in the following code snippet:

```
this._map = new Microsoft.Maps.Map(document.getElementById('lbm-map'),
{
        credentials: '[YOUR BING MAPS KEY]',
        showBreadcrumb: false,
        showDashboard: false,
        center: new Microsoft.Maps.Location(51.537237, -0.080857), /*
London */
        zoom: 10
    });
```

The poll

The approach we have taken throughout this book is to extend the Microsoft.Maps API by means of modules. This project is not an exception. So far, our project folder holds the `index.html` file and the `learningTheme` folder. Let's add a `poll` directory, inside which we create the `poll.js` and `poll.css` files.

Let's start with our familiar module skeleton inside `poll.js,` as shown in the following code snippet:

```
(function() {
  var lbm = window.lbm || {};

  lbm.Poll = function(map, $) {
    this._map = map;
    this.$ = $;
  };

  Microsoft.Maps.moduleLoaded('lbm.Poll');
})();
```

The `Poll` module is attached to the `lbm` namespace so that the global scope does not get cluttered. Its constructor accepts an instance of the map and a `jQuery` object as its arguments.

Now it's time to pause for a minute and think how we want to develop the user interface. The requirements state that we need to show the results for four elections; so probably a list with four buttons labeled with each elections year could be appropriate. This way, when the user clicks on a year button, the map devises the results for the ballot vote of that year.

First, we store the years on an array variable, declared just under the namespace declaration, and outside the module's code, so that it is accessed by all the functions in that file, which can be written as follows:

```
var YEARS = [2000, 2004, 2008, 2012];
```

Then, we write the markup of the navigation bar, as shown in the following code snippet:

```
_addMarkup: function() {
  this._container = this.$('<div class="lbm-poll-container"/>');
  var html = [];
  html.push('<p>');
  for (var i = 0; i < YEARS.length; i++) {
    html.push('<button data-year="' + i + '">' + YEARS[i] + '</
button>');
  }
  html.push('</p>');
  this._container
    .append(html.join(''))
    .appendTo(this._map.getRootElement().parentElement);

  this._position();
}
```

It is a good idea to hold a reference of any elements added to the `DOM` element, so that the latter is not queried again for it. We do that with the `_container` instance variable, which contains a `DIV` element with the `lbm-poll-container` class. Next, for each year in the `YEARS` variable we add a button labeled with the respective value. We also store the index into a `data-` attribute, a conventional way for storing small pieces of data within the DOM nodes. This is better than extracting the year from the text node of the button, because it keeps their concerns separated.

The markup is then added to the container, which in turn is appended to the map's container.

We position the container so that it sits in the middle of the browser window through the `app.css` stylesheet, as shown in the following code:

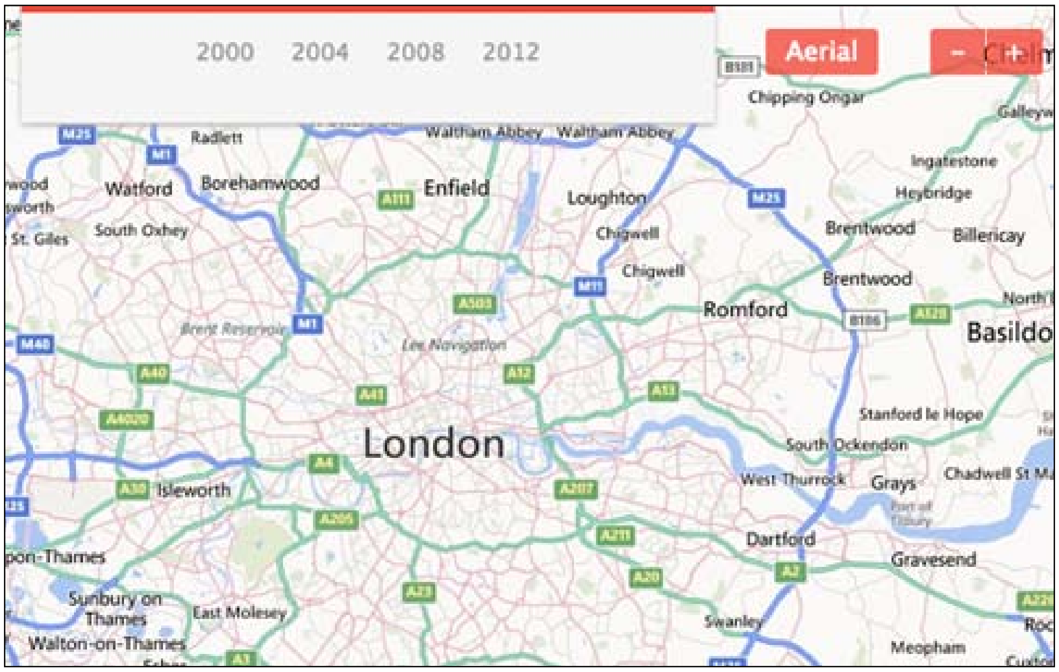
```
.lbm-poll-container {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  width: 30em;
  z-index: 30;
  background: #F5F5F5;
  padding: .5em 0;
  margin: auto;
  text-align: center;
  text-transform: uppercase;
  border-top: 5px solid #F04C40;
  box-shadow: 0 2px 5px #bbb;
}

.lbm-poll-container button {
  background: transparent;
  border: none;
  padding: .6em;
  margin: 0 .2em;
  font-size: 1em;
  color: #9C9D9E;
}

.lbm-poll-container button:hover {
  box-shadow: inset 0 1px 2px rgba(0, 0, 0, 0.125);
}

.lbm-poll-container button:active,
.lbm-poll-container button.active,
.lbm-poll-container button.active:hover {
  color: #F04C40;
  box-shadow: inset 0 3px 4px rgba(0, 0, 0, 0.125);
}
```

Now, when we open the `index.html` file on the browser, we should see our navigation bar centered at the top of the map, as shown in the following screenshot:



Let's now load the data by adding the following method to our module's prototype:

```
load: function(data) {
  this._data = data;
  for (var i = 0; i < this._data.length; i++) {
    this._showShape(this._data[i]);
  }
this._activate(this._container.find('button').eq(this._currentYear));
}
```

The function accepts a `data` argument, which we store on the `_data` instance variable. We loop through the latter, sending each of its parts to the `_showShape` method.

At the end, we activate the button found under the same index as the `_currentYear` instance variable. The latter is used to control the data of the year that will be shown on the map. We start by showing the latest election year, 2012; therefore, we set the `_currentYear` variable to its index in the `YEARS` array as `3`.

```
this._currentYear = 3;
```

We add the preceding line at the end of the module's constructor, and then write the `_activate` method, as shown in the following code snippet:

```
_activate: function(button) {
  this._activeButton = button.addClass('active');
}
```

The preceding code sets one of the buttons as active, so that by keeping track of the active button, we make it easier to deactivate it later. To tell it apart from the other buttons, we style the button with the `active` class.

Now, let's write the code for the method that shows the shape on the map, as shown in the following code snippet:

```
_showShape: function(data) {
  var shape = new Shape(this._map, data, this._currentYear);
  shape.show();
}
```

We love object oriented programming, and we take full advantage of JavaScript capabilities in the matter, by delegating the shape creation to a `Shape` class.

The shape

As we saw earlier, this class stores a reference to the map, data, and the initial year. The election results of the latter will be reflected on the shape first, as shown in the following code snippet:

```
var Shape = function (map, data, currentYear) {
  this._map = map;
  this._data = data;
  this._updateParty(currentYear);
}
```

A shape will be filled with the color of the political party that has won the elections on the London Assembly area represented by that shape. For this, we create an object instance variable (`_party`) with the `id` extracted from the `votes` array.

```
_updateParty: function(year) {
  this._party = {
    id: this._data.votes[year]
  };
  this._party.name = PARTIES[this._party.id];
}
```

The party's name is fetched from a constant array we define at the top of the file, below the line specifying the `YEARS` array.

```
var PARTIES = ['Conservative', 'Labour'];
```

These are the only two parties that have controlled the London Assembly in the past four elections, so we add only them to the list.

It's time to write the `show` method as follows:

```
Shape.prototype = {
  show: function() {
    this._getLocations();
    this._addShape();
  }
}
```

This method does two jobs: it prepares the locations and adds the shape to the map. Let's write the code for the first process, as follows:

```
_getLocations: function() {
  this._locations = [];
  for (var i = 0, length = this._data.wkt.length; i < length; i++) {
    var coords = this._data.wkt[i];
    this._locations.push(new Microsoft.Maps.Location(coords[0],
    coords[1]));
}
}
```

Here, we loop through the array of coordinate pairs with which we build the `Location` instances to make up the shape.

```
_addShape: function() {
  this._shape = new Microsoft.Maps.Polygon(this._locations, {
    strokeThickness:   2,
    strokeColor:       this._getColor({stroke: true}),

    fillColor:         this._getColor()
  });
  this._bind(this);
  this._map.entities.push(this._shape);
}
```

The preceding code generates a polygon with the fill and border color depending on the party it pertains. The method that sets the colors looks like this:

```
_getColor: function(options) {
  options = options || {};
  var opacity = options.stroke ? 200 : (options.onMouseOver ? 120 :
60);
  if (this._party.id === 1) {
    return new Microsoft.Maps.Color(opacity, 0, 134, 219);
  }
  return new Microsoft.Maps.Color(opacity, 238, 50, 36);
}
```

A reddish color is specified for the Labor party and a bluish color for the Conservatives. The opacity of the polygon's background increases on mouse over, taking advantage of the first argument of the `Color` class, which defines opacity. The other three arguments represent the red, green, and blue components of the color.

To make the shape change its color when the user moves the mouse over it, we need a couple of event handlers, as shown in the following code:

```
_bind: function(self) {
  Microsoft.Maps.Events.addHandler(this._shape, 'mouseover',
function() {
    self._onmouseover();
  });

  Microsoft.Maps.Events.addHandler(this._shape, 'mouseout', function()
{
    self._onmouseout();
  });
}
```

The `Polygon` class responds to the `mouseover` and `mouseout` events, as does the `Map` class. The former event is bound to the following handler:

```
_onmouseover: function() {
  this._shape.setOptions({
    fillColor: this._getColor({onMouseOver: true}),
  });
  this._log(this._data.name + ' <b>' + this._party.name + '</b>');
}
```

First, we send an option with the darker color to the `setOptions` method of the `Polygon` class. Then, we show the name of the area and its controlling party on the navigation bar. We do this through a `_log` method, as shown in the following code snippet:

```
_log: function(message) {
  Microsoft.Maps.Events.invoke(this._map, 'poll:log', {
    message: message
  });
}
```

This function delegates displaying of the message to a custom map event, named `poll:log`. Any object that listens to this event will be triggered at this moment. As stated earlier, we want to show the message on the navigation bar. Therefore, we need the `Poll` class to handle the `poll:log` event, as shown in the following code:

```
_bind: function(self) {
  Microsoft.Maps.Events.addHandler(this._map, 'poll:log', function(e)
{
    self._log(e.message);
  });
}
```

The preceding function is added to the prototype of the `Poll` class. It binds the `_log` handler to the event, which is described as follows:

```
_log: function (message) {
  this._logger = this._logger || this._$('#lbm-poll-log');
  this._logger.html(message);
}
```

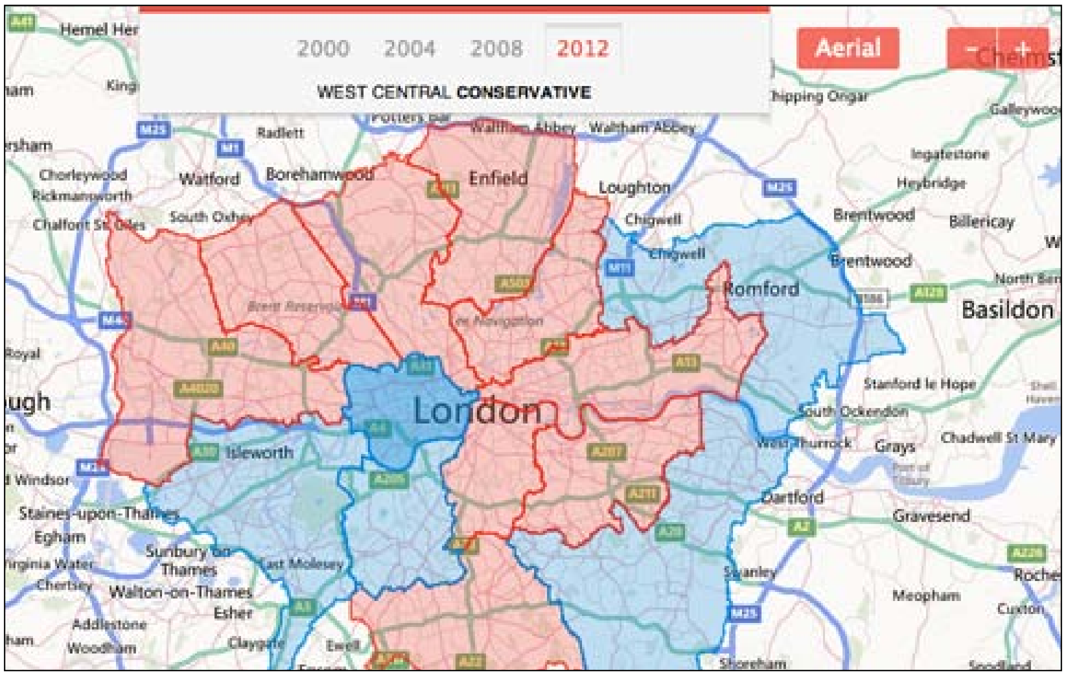
The handler needs an element with the `lbm-poll-log` ID to be available on the `DOM` element, so let's do that by adding the following lines to the `_addMarkup` function, just before we close the `DIV` node:

```
html.push('</p>');
html.push('<p id="lbm-poll-log"></p>');
html.push('</div>');
```

To make the element look nice on the page, we add the following style to the `poll.css` stylesheet:

```
.lbm-poll-container #lbm-poll-log {
  margin-top: .6em;
  font-size: .8em;
  height: 1.2em;
}
```

If we refresh the `index` page on the browser now, we should see our shapes painted with different colors; the latter changes when we move the mouse over the polygons.



We can note how the text on the navigation bar also changes according to the shape over which has the mouse.

Rolling back the years

The last requirement we need to implement, states that the application must change the colors of the polygons as the user clicks on the year buttons of the navigation bar. As before, we will achieve this through the `year:change` custom event we'll add to the map that will be triggered by the navigation buttons, and handled by the shapes.

```
Microsoft.Maps.Events.addHandler(this._map, 'year:change', function(e)
{
  self._changeYear(e.currentYear);
});
```

The preceding lines, which we add to the _bind method of the Shape class, bind the year:change event to the _changeYear function. Note that this function expects that the eventArgs object passed should contain information about the year selected in the form of its index in the YEARS array. It's time to write the handler as follows:

```
_changeYear: function(currentYear) {
  this._updateParty(currentYear);
  this._shape.setOptions({
    strokeColor: this._getColor({stroke: true}),
    fillColor:   this._getColor()
  });
}
```

This method updates the party based on its data, and sets the fill and border colors accordingly. We can note how calling the setOptions method of the polygon changes its appearance immediately.

Now, let's trigger the year:change event from the Poll class, by adding the following lines to its _bind method.

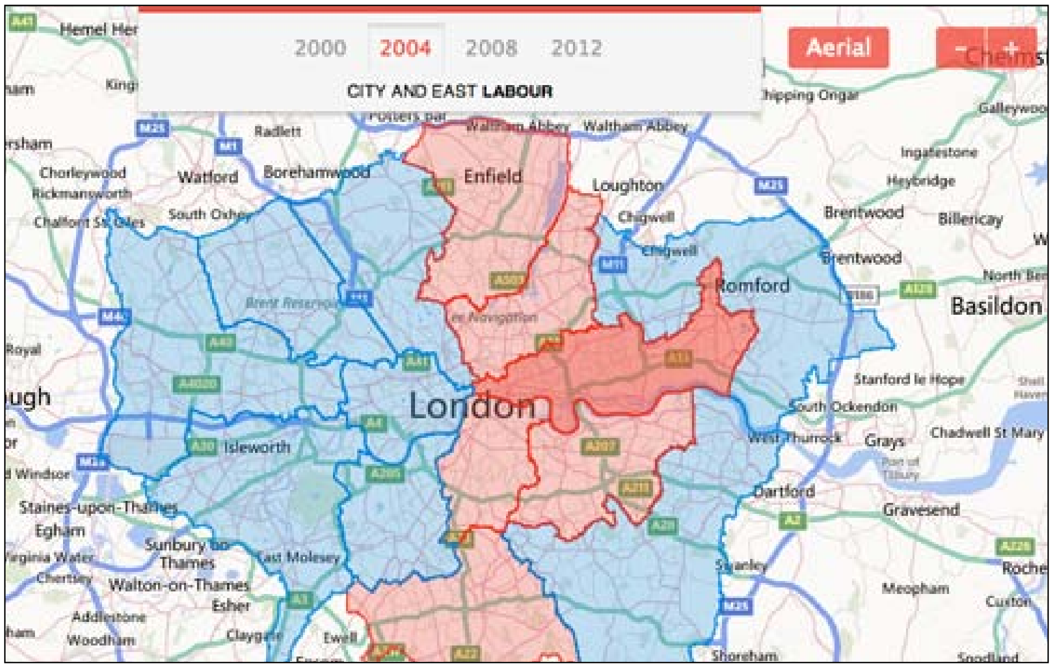
```
this._container.on('click', 'button', function() {
  self._changeYear(this);
});
```

When the navigation buttons are clicked, the app calls another _changeYear function, but this time a method of the Poll module. The function can be written as shown in the following code:

```
_changeYear: function(button) {
  if (this._activeButton) this._activeButton.removeClass('active');
  this._activate($(button));
  this._currentYear = this._activeButton.data('year');
  Microsoft.Maps.Events.invoke(this._map, 'year:change', {
    currentYear: this._currentYear
  });
}
```

We make the clicked button active by passing it to the _active routine we wrote earlier. Before that, we deactivate the **active** button (by removing the active class) if it exists. Then, we extract the current year from the data-year attribute and store it in the _currentYear variable. This is used to create the eventArgs object that is passed to the invoke method of the Events object, together with a map reference and the year:change event name.

Time to test our full application on the browser now. Let's refresh the `index` page, and click on the buttons:



When we click on different years, the polygons change their colors to reflect the political party that controlled the London area they represent after the elections of that year. Some of the shapes don't seem to change their colors, indicating that their areas continued with the same party. This is a great way of showing the political movements of different areas over the years.

Summary

Bing Maps can be a great canvas for us to draw our own picture on it. We can lay on the map any kind of geodata to illustrate an idea, such as the available schools in a geographical area, or even where check-in desks are situated on an airport based on their flight information.

In this chapter, we drew the last four electoral results for the London Assembly, painting its constituent areas with the colors of the wining party.

Index