Learn by doing: less theory, more results

OpenLayers 3

Get started with OpenLayers 3 and enhance your web pages by creating and displaying dynamic maps

Beginner's Guide

Thomas Gratier
Erik Hazzard
Paul Spencer
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 6 “Styling Vector Layers”
- A synopsis of the book’s content
- More information on OpenLayers 3 Beginner's Guide

About the Authors

Thomas Gratier is a GIS consultant living in Nantes, France, who mainly specializes in web development. He has an MSc degree in geography and urban planning from The Institute Of Alpine Geography of The University of Grenoble. Career-wise, he decided to steer towards more technical work but kept his geospatial passion in mind. He's gained 8 years of geospatial and programming experience, working for public authorities on water and flood risk prevention and management, various private urban consultancies in urban planning and web mapping solutions, and, multinational company CapGemini's GIS Division. He's continued building a stronger knowledge in IT technologies, open source and open data, and both web and geo standards. He does his development work with JavaScript, Python, and PHP. His favorite libraries and tools for working are GDAL/OGC, PostGIS, QGIS, and OpenLayers, but he also enjoys using OpenStreetMap-related libraries, such as Mapnik or Osm2pgsql. He is an open source advocate, a Charter Member of The Open Source Geospatial Foundation (http://www.osgeo.org)(OSGeo). He gets involved in writing French translations for open source geospatial projects, such as MapServer and Zoo Project. With like-minded professionals, he contributes to weekly geospatial news updates at Geotribu (http://geotribu.net). He currently works as a freelance GIS consultant, providing development, consulting, and training services. More information can be found on his website at Web Geo Data Vore (http://webgeodatavore.com).
I'd like to thank the OpenLayers developers and contributors for developing this powerful web-mapping framework that works well for both simple and complex use cases. My thanks go to Packt Publishing's editing team for their knowledge and input to write this book. Without their invaluable help to keep me on track, this book could not have been completed. I would also like to thank Erik Hazzard, the author of the initial book, OpenLayers 2.10 Beginner's Guide, Packt Publishing. His version has been useful as a starting base. I'd like to thank Paul Spencer, my co-author for his fruitful collaboration despite time-zone constraints. It has been a real pleasure to share ideas on similar interests. His experience and insights have been invaluable to complete the task of writing my first book. My thanks go to my family and friends for their constant support, time, and kind understanding through the years.

Paul Spencer is a software engineer who has worked in the field of open source geospatial software for more than 15 years. He is a strong advocate of open source development and champions its use whenever possible. Paul has architected several successful open source projects and been actively involved in many more. Paul was involved in the early design and development of OpenLayers and continues to be involved as the project evolves. Paul joined DM Solutions Group (DMSG) in 1998, bringing with him advanced software development skills and a strong understanding of the software-development process. In his time with the company, Paul has taken on a leadership role as the CTO and primary architect for DM Solutions Group's web mapping technologies and commercial solutions. Prior to joining DMSG, Paul worked for the Canadian Military, after achieving his master's degree in software engineering from The Royal Military College of Canada.

I would like to thank my wife and son, without whose support and encouragement, I would not have been able to write this book.

Erik Hazzard is the author of OpenLayers 2.10 Beginner's Guide, Packt Publishing. He has worked as the lead developer for a GIS-based company, has done contracting work with the design studio, Stamen, and has co-founded two start-ups. Erik is passionate about mapping, game development, and data visualization. In his free time, he enjoys writing and teaching, and can be found at http://vasir.net.
OpenLayers 3 Beginner's Guide

Web mapping is the process of designing, implementing, generating, and delivering maps on the Web and its products. OpenLayers is a powerful, community-driven, open source, pure JavaScript web mapping library. With this, you can easily create your own web map mashup using a myriad of map backends. Interested in knowing more about OpenLayers? This book is going to help you learn OpenLayers from scratch. OpenLayers 3 Beginner's Guide will walk you through the OpenLayers library in the easiest and most efficient way possible. The core components of OpenLayers are covered in detail, with examples, structured so that you can easily refer back to them later.

The book starts off by showing you how to create a simple map and introduces you to some basic JavaScript programming concepts and tools. You will also find useful resources to learn more about HTML and CSS. Through the course of this book, we will review each component needed to make a map in OpenLayers 3, and you will end with a full-fledged web map application.

You will learn some context to help you understand the key role of each OpenLayers 3 component in making a map. You will also learn important mapping principles such as projections and layers. Maps require sources of data as well; so, you will see how to create your own data files and connect to backend servers for mapping. A key part of this book will also be dedicated to building a mapping application for mobile devices and its specific components.

With OpenLayers 3 Beginner's Guide, you will learn how to create your own map applications independently, without being stuck at the first stage of learning. You will acquire the information you need to develop your skills and knowledge of the OpenLayers 3 library.
What This Book Covers

Chapter 1, Getting Started with OpenLayers will introduce you to OpenLayers 3 and will help you to learn some basic mapping principles. You will see how to get ready for development in OpenLayers and create your first map.

Chapter 2, Key Concepts in OpenLayers will introduce the main components of the OpenLayers library and illustrate how they are related. We will introduce some key concepts, including events and observable properties, and learn some basic debugging techniques.

Chapter 3, Charting the Map Class will describe two of the core components, the Map and View classes, in detail. We will learn about the properties, methods, and events of both classes and apply them in practical examples.

Chapter 4, Interacting with Raster Data Source will introduce the concept of layers and focus on raster layers. We will explain the difference between tiled and untiled layers and learn how to use OpenLayers to visualize any type of image, even non-geospatial ones.

Chapter 5, Using Vector Layers will introduce vector layers and the related source, format, feature, and geometry classes. We will learn the properties, methods, and events associated with each, and how to use them to load a variety of vector data into an OpenLayers map.

Chapter 6, Styling Vector Layers will expand on our knowledge of vector layers by learning how to apply both static and dynamic styles to them. Through hands-on examples, we'll learn how to modify styles interactively in response to user interaction.

Chapter 7, Wrapping Our Heads Around Projections will cover the basic concepts behind map projections and their characteristics. We will cover projection support within OpenLayers by introducing the Proj4js library and applying it to map, vector, and raster layers.

Chapter 8, Interacting with Your Map will dive into the concept of interactions and introduce the default interactions. After covering the available interactions in detail, we will finish with an example showing how to use interactions to draw a rectangle.

Chapter 9, Taking Control of Controls will demonstrate the use of controls and introduce the default controls provided by OpenLayers. We will also review each of the controls in more detail and learn how to make a custom control.

Chapter 10, OpenLayers Goes Mobile will teach us to take advantage of mobile-specific features such as Geolocation and Device Orientation. We will also learn how to debug mobile web applications and look at some mobile-specific browser features that can be useful for geospatial applications.
Chapter 11, Creating Web Map Apps will build a complete application from scratch and learn how to use the OpenLayers build system to create a production-ready application from our code.

Appendix A, Object-oriented Programming – Introduction and Concepts covers the main concepts of Object-oriented Programming (OOP). After, we will discover how to reuse them exploring the OpenLayers API documentation with OOP in mind.

Appendix B, More details on Closure Tools and Code Optimization Techniques will cover more details on Closure Tools and code optimization techniques. This appendix introduces Closure Tools, a set of tools that OpenLayers 3 library relies on. It provides an understanding on how to use the Closure Library and Closure Compiler with a focus on compressing OpenLayers code files. We will finish with a review of styles and syntax for good coding practices.

Appendix C, Squashing Bugs with Web Debuggers provides JavaScript beginners with an in-depth review of browser developer tools. We will review Chrome Developer Tools, additional extensions and finish with debugging tools in other browsers such as IE and Firefox.
By now you should be getting pretty comfortable creating simple OpenLayers maps with a combination of raster and vector data. With raster data, there is no control over presentation, as the saying goes—what you see is what you get. Vector data, on the other hand, gives you direct control over presentation details. We’ve already alluded to vector styles, and used them in the previous chapter. Now, it is time to take full control of how we present our vector data!

In the last chapter, you saw how powerful the vector layer can be. In this chapter, we'll go a bit deeper and talk about how to customize the appearance of the features within a vector layer. We’ll explore the following:

- The basic style object
- Fill, stroke, image, and text styles
- Composing multiple styles
- Using style functions
- Applying styles interactively
What are vector styles?

So, what is a vector style? Quite simply, it is a set of instructions about how to draw graphic primitives—the points, lines, polygons, and text that make up our vector features. OpenLayers provides a basic default style that renders features in various shades of blue. While this is quite nice, it’s probably not what you’ll want to use all the time.

You’ve already seen an example of a basic vector style in the previous chapter. Let’s review it here:

```javascript
var fill = new ol.style.Fill({
  color: 'rgba(0,0,0,0.2)'
});
var stroke = new ol.style.Stroke({
  color: 'rgba(0,0,0,0.4)'
});
var circle = new ol.style.Circle({
  radius: 6,
  fill: fill,
  stroke: stroke
});
var vectorStyle = new ol.style.Style({
  fill: fill,
  stroke: stroke,
  image: circle
});
```

This code defines specific rules for the fill, stroke, and image properties of a new `ol.style.Style` object. The fill and stroke rules specify a color and opacity using the RGBA (Red, Green, Blue, and Alpha) format. The circle style is a special style that draws points as circles of a specific radius with a fill and stroke property. Together, these form a style object that OpenLayers uses to determine how to draw a feature. The fill property is used for filling polygons and circles. The stroke property is used to draw the outline of polygons, lines, and circles. The image property is used for drawing points. There is also a text property that we haven’t seen yet.

The vector layer’s style property accepts three different ways of specifying styles:

- An instance of `ol.style.Style`
- An array of `ol.style.Style` instances
- A style function
What is a style function?

We’ll look at style functions in more detail in the second half of this chapter, but briefly, a style function is one that returns an array of style objects to be used for a specific feature and resolution. In combination with feature properties, a style function allows for the implementation of advanced custom styling. Before we can run, we’d better learn how to walk.

Time for action – basic styling

We’ll start with an example that shows off most of the basic style properties. We’ll start with a new HTML page setup the same way we usually start off.

1. Make a copy of our sandbox template and add the standard setup for a map to the main <script> tag:

   ```javascript
   var center = ol.proj.transform([0, 0], 'EPSG:4326', 'EPSG:3857');
   var view = new ol.View(
       center: center,
       zoom: 1
   ));
   var map = new ol.Map({
       target: 'map',
       view: view
   }));

2. In this example, we’ll be purely using vector layers. No need for rasters here! Go ahead and create a vector layer for countries, then add it to the map:

   ```javascript
   var countries = new ol.layer.Vector({
       source: new ol.source.GeoJSON({
           projection: 'EPSG:3857',
           url: '../assets/data/countries.geojson'
       })
   }));
   var map = new ol.Map({
       target: 'map',
       layers: [countries],
       view: view
   }));
   ```
3. Load the HTML file to your browser to see the basic styling that OpenLayers provides:

4. Make it a bit more interesting by adding another vector layer, time zones. Don't forget to add it to the map's layers property:

   ```javascript
   var timezones = new ol.layer.Vector({
       source: new ol.source.KML({
           projection: 'EPSG:3857',
           url: '../assets/data/timezones.kml'
        })
   });
   ```
5. Now, the map looks like the following. The time zone KML file is drawn in different colors because KML files often contain style information for each feature.

![Map with time zone KML file drawn in different colors](image)

6. We'd like to have more control over the appearance of our map; so, let's create some rules for styling. First, create two style objects—one for each layer:

```javascript
var countryStyle = new ol.style.Style({
    fill: new ol.style.Fill({
        color: [203, 194, 185, 1]
    }),
    stroke: new ol.style.Stroke({
        color: [177, 163, 148, 0.5],
        width: 2
    })
});

var timezoneStyle = new ol.style.Style({
    stroke: new ol.style.Stroke({
        color: [64, 200, 200, 0.5],
    }),
    text: new ol.style.Text({
        font: '20px Verdana',
        text: 'TZ',
        fill: new ol.style.Fill({
            color: [64, 64, 64, 0.75]
        })
    })
});
```
7. Don't forget to add a `style` property to each of the vector layers so they know which style to use. Also, we'll need to tell the KML source not to extract the default feature styles stored in the KML document:

```javascript
var countries = new ol.layer.Vector({
  source: new ol.source.GeoJSON({
    projection: 'EPSG:3857',
    url: '../assets/data/countries.geojson'
  }),
  style: countryStyle
});
var timezones = new ol.layer.Vector({
  source: new ol.source.KML({
    projection: 'EPSG:3857',
    url: '../assets/data/timezones.kml',
    extractStyles: false
  }),
  style: timezoneStyle
});
```

8. Take a look at the result. Not arguably better, perhaps, but we now have full control!

![Map with vector layers](image)

**What just happened?**

With this example, we are illustrating some of the basic styling capabilities of OpenLayers. We combined two static vector sources—countries and time zones—and some simple styles to create our map.
Step 1 set up the same structure we've been using for all our examples. In step 2, we added the country data using a static GeoJSON source and added it to the map with the default styling. Next, we added the time zone data in step 4 and developed some basic styles for the two layers in step 6. The country data is composed of polygons, so we created a fill style and a stroke style for styling the country layer. The time zone data also contains polygons, but we want to see the countries underneath so we created a separate style with just the stroke and text properties for it.

The text property in this example is somewhat contrived to keep things simple. Showing the same text string for each time zone is not what we'd really want to do. Ideally, we'd display information specific to each time zone—perhaps, the name or number of hours from Greenwich Mean Time. We'll cover how to do this a bit later in the chapter though. The last thing is to assign our custom styles to the appropriate vector layer and turn off the automatic extraction of styles from the KML layer, which happened in step 7.

Now that we've reviewed how basic styles work, let's take a closer look at the style object and the properties we can assign to it. Along the way, we'll illustrate each with a specific example.

**The style class**

The style class, `ol.style.Style`, contains the drawing instructions to be used when rendering a feature. There are five properties we can use when creating a style property:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td><code>ol.style.Fill</code></td>
<td>This style is used when filling polygons. To draw unfilled polygons, leave this property out or set to <code>null</code>.</td>
</tr>
<tr>
<td>stroke</td>
<td><code>ol.style.Stroke</code></td>
<td>This style is used when drawing lines and drawing the outline of polygons. To draw polygons without an outline, leave this property out or set to <code>null</code>.</td>
</tr>
<tr>
<td>image</td>
<td><code>ol.style.Image</code></td>
<td>This style is used when drawing points.</td>
</tr>
<tr>
<td>text</td>
<td><code>ol.style.Text</code></td>
<td>This style is used when drawing text.</td>
</tr>
<tr>
<td>zIndex</td>
<td>number</td>
<td>The z-index determines the order in which features are drawn. To ensure that certain features are drawn on top of other features—for instance, points on top of polygons—assign a higher <code>zIndex</code> value. Note that this only affects features within the same layer.</td>
</tr>
</tbody>
</table>
Styling Vector Layers

As you can see, a style is really a composite of several specific types. Including a property turns on drawing of the relevant type and excluding it turns it off. When you specify the style property for a vector layer, this replaces all the default styles; so, you don’t need to override all the properties all the time—just specify the ones that are needed and the rest will not be drawn at all. For example, creating a style with just a stroke property will draw polygons with an outline and no fill:

```javascript
var style = new ol.style.Style({
  stroke: new ol.style.Stroke({
    color: [127,127,127,1]
  })
});
```

Now, let's look at each specific property type with some examples.

### Fill styles

The fill style—`ol.style.Fill`—is used to fill shapes with a solid color. The fill style is used by `ol.style.Style` as well as a couple of other objects we'll see shortly. It has a single property, color, of the type `ol.Color` that is used when drawing filled shapes.

Colors may be specified in three ways:

- An array of four values representing the red, green, blue, and alpha components of the color. The color components are numbers between 0 and 255, while the alpha value is between 0 (transparent) and 1 (opaque). For example, black is represented as `[0, 0, 0, 1]`, white is represented as `[255, 255, 255, 1]`, and a semitransparent blue green is `[0, 255, 0, 0.5]`.

- A CSS RGBA string expression, written as `"rgba(red,green,blue,alpha)"`, where red, green, blue, and alpha are the same as the preceding array form; for example, our semitransparent green color will be `"rgba(0,255,0,0.5)"`.

- A CSS hexadecimal, or hex, color value written as `#RRGGBB`, where RR is the red value, GG is the green value, and BB is the blue value. The values are a hexadecimal equivalent of the numeric values between 0 and 255, written as 00 to FF. The alpha value is assumed to be 1 in this case.

These different representations are equivalent (except for the missing alpha control in hex colors) and you can use whichever is more convenient.
Stroke styles

The stroke style—`ol.style.Stroke`—is used to draw lines. The line style is used by `ol.style.Style` as well as a couple of other objects we'll see shortly. A stroke style has the following options.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td><code>ol.Color</code></td>
<td>This is the color to use when drawing lines.</td>
</tr>
<tr>
<td>lineCap</td>
<td>string</td>
<td>This is the style to draw the end of lines in. This may be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- butt: These finish lines squarely right at the exact point the line ends at.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- round: These finish lines by rounding them, radius depends on width at the bottom. This is the default value if not specified.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- square: These finish lines with a square the size of width (line extends past the last point by the line width)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See the diagram following this table for an example of each lineCap style.</td>
</tr>
<tr>
<td>lineJoin</td>
<td>string</td>
<td>The line join style is used when drawing segments that are part of the same line. This may be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- bevel: This joins lines with a bevel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- round: This joins lines by rounding them.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- miter: This joins lines by mitering them (see miterLimit below).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See the diagram following this table for an example of each lineJoin style.</td>
</tr>
<tr>
<td>lineDash</td>
<td>Array.&lt;number&gt;</td>
<td>This is an array of numbers that define the on-off pattern for drawing lines with a dash pattern. The default is none (no dash pattern).</td>
</tr>
<tr>
<td>miterLimit</td>
<td>number</td>
<td>This is the limit for drawing miter joins; the default is 10.</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>This is the width, in pixels, to draw the line. This number may be a floating point number.</td>
</tr>
</tbody>
</table>
Styling Vector Layers

The following diagram illustrates the effect of the various values for `lineJoin` (top) and `lineCap` (bottom).

<table>
<thead>
<tr>
<th>Mitre Join</th>
<th>Bevel Join</th>
<th>Round Join</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Butt Cap</th>
<th>Square Cap</th>
<th>Round Cap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Have a Go Hero – fill and stroke styles**

Modify the last example and try out some fill and stroke style properties. In particular, try changing the `lineJoin` and `lineCap` properties. Use a wider stroke width to see the effect it produces. Note that the `lineCap` style won't be apparent when drawing polygons—to see it in action, you’ll need a line layer, perhaps, using the `fells_loop.gpx` file we saw in the previous chapter.

Here are a couple of examples:

```javascript
var countryStyle = new ol.style.Style({
  fill: new ol.style.Fill({
    color: [0, 255, 255, 1]
  }),
  stroke: new ol.style.Stroke({
    color: [127,127,127,1.0],
    width: 10,
    lineJoin: 'bevel',
  })
});
var timezoneStyle = new ol.style.Style({
  stroke: new ol.style.Stroke({
    color: [64, 200, 200, 0.5],
    lineJoin: 'round',
    width: 10
  })
});
```
The image style—`ol.style.Image`—is used to style point data. You won't be using it directly though. Instead, there are two subclasses that you'll be using: `ol.style.Icon` and `ol.style.Circle`. Let's look at the icon style first.

The icon style

The icon style displays an image at the location of a point. There are quite a few properties associated with the icon style that allow you to align the placement of the image relative to the precise geographic location being represented.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>Array.&lt;number&gt;</td>
<td>This property states where to position the image relative to the geographic location of the point specified as an array of two numbers. The default value is <code>[0.5, 0.5]</code> and assumes units of fraction (see <code>anchorXUnits</code> and <code>anchorYUnits</code>). This will specify aligning the center of the image to the geographic location. The position within the image is measured relative to the <code>anchorOrigin</code> property, which defaults to <code>top-left</code>.</td>
</tr>
</tbody>
</table>
| anchorOrigin| string       | This specifies where the anchor value is measured from. One of the following values can be used:  
  - top-left (the default)  
  - top-right  
  - bottom-left  
  - bottom-right |
<p>| anchorXUnits| string       | This specifies the units of the X anchor value, either pixels or fraction. The default value is fraction. When fraction is used, the value of the associated value is a floating point number between 0 and 1 as a percentage of the width or height of the image. |
| anchorYUnits| string       | This specifies the units of the Y anchor value. |
| crossOrigin | string       | The <code>crossOrigin</code> setting for the image allows you to leverage CORS (Cross Origin Resource Sharing) when loading an image. |
| img         | Image        | This is an image object to use for the icon. This may be used instead of the <code>src</code> property but the provided image object must already be loaded. |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>Array.&lt;number&gt;</td>
<td>This is the top-left corner of the image to draw the icon from. When combined with the size property, this would allow you to use an image sprite and selecting a portion of the sprite to display for a specific icon. The default is ([0, 0]).</td>
</tr>
</tbody>
</table>
| offsetOrigin | String            | This sets the origin of the offset property, one of the following:  
|              |                   | ✗ bottom-left  
|              |                   | ✗ bottom-right  
|              |                   | ✗ top-left (the default)  
|              |                   | ✗ top-right  |
| scale        | number            | This is a scale factor to use when drawing the image, the default is 1 (do not scale the image). A value of 2 will double the size of the icon and a value of 0.5 will half the size of the icon. |
| snapToPixel  | Boolean           | If true, this property will cause images to be snapped to integer pixel values and result in sharper display of images. If false, the image will be placed more accurately but may appear blurry. The default value is true. You may want to set this to false if you are animating an icon’s position, as snapping to pixels would cause noticeable jitter. |
| rotateWithView | Boolean        | If true, the icon will rotate when the map’s view is rotated. The default is false (always stay upright). |
| rotation     | number            | This is a rotation (in radians) to apply to the icon. |
| size         | ol.Size           | This is the size of the icon in pixels as an array of two values—width and height. The default value is the size of the image being used. Using a different value, when combined with origin, allows the use of image sprites. |
| src          | string            | This is the URL to load the image from. |
The following diagram illustrates the meaning of the *origin*, *size*, and *anchor* options:

![Diagram illustrating origin, size, and anchor options](image)

**Time for action – using the icon style**

As we haven’t seen the Icon style before, let’s build an example. As we’ll need some point data for this example, the `earthquake.kml` file should be perfect! We’ll use the following image sprite and pick the middle dot. This file is included with the code samples that come with the book; you can find it at `assets/img/dots.png`.

![Image sprite](image)

1. Starting from the previous example, first, we’ll create an icon style using our sprite:

   ```javascript
   var earthquakeStyle = new ol.style.Style({
     image: new ol.style.Icon({
       anchor: [0.5, 0.5],
       size: [52, 52],
       offset: [52, 0],
       opacity: 1,
       scale: 0.25,
       src: '../assets/img/dots.png'
     })
   });
   ```
2. Next, create the earthquake layer and assign it the style. Because it is a KML file, we'll need to tell OpenLayers to not extract the embedded style information:

```javascript
var earthquakes = new ol.layer.Vector({
  source: new ol.source.KML({
    projection: 'EPSG:3857',
    url: '../assets/data/earthquakes.kml',
    extractStyles: false
  }),
  style: earthquakeStyle
});
```

3. Now, add the layer to the map's layers array:

```javascript
var map = new ol.Map({
  target: 'map',
  layers: [countries, timezones, earthquakes],
  view: view
});
```

4. The result should look something like this:

![Map with earthquake layer](image)

**What just happened?**

Using the icon style properties, we displayed an image at the location of each of the points in the `earthquakes.kml` file.
First, we created an icon style pointing at our sprite image and supplied values for offset, anchor, size, and scale to ensure our red dot is used. The sprite is 156 pixels wide and 52 pixels high, and each image in the sprite is 52 by 52 pixels, so we provided a size of \([52,52]\). An offset value of \([52,0]\) moved the frame of reference 52 pixels in from the left edge. The anchor position of \([0.5,0.5]\) specified the middle of the image since the default anchor units is fraction. We can also have specified pixels for anchor units and changed our anchor position to \([26,26]\). Given the density of the earthquakes, using a 52 by 52 pixel image will overwhelm the map; so, we provided a scale value of 0.25, which effectively shrunk the resulting image to 25 percent of its size, or 13 by 13 pixels.

After creating a new vector layer that used our icon style, we added it to the map.

**The circle style**

Like the icon style, the circle style is used for point geometries. It’s a much simpler object, though with only four configurable properties:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
<td>number</td>
<td>This is the radius, in pixels, to draw the circle.</td>
</tr>
<tr>
<td>fill</td>
<td>ol.style.Fill</td>
<td>This is the style to fill the circle with.</td>
</tr>
<tr>
<td>snapToPixel</td>
<td>Boolean</td>
<td>If true, this property will cause the circle to be snapped to integer pixel values and result in sharper display of the circle. If false, the circle will be drawn more accurately but may appear blurry. The default value is true. You may want to set this to false if you are animating the circle’s position as snapping to pixels may cause noticeable jitter.</td>
</tr>
<tr>
<td>stroke</td>
<td>ol.style.Stroke</td>
<td>This is the style to draw the circumference of the circle with.</td>
</tr>
</tbody>
</table>
**Have a go hero – using the circle style**

To round out our understanding of the image styles, redo the previous example using `ol.style.Circle` instead of `ol.style.Icon` as the image property of the earthquake style. Your result should look something like this (depending on what values you choose):

![Image example]

**Text styles**

We saw the text style in action in the first example, and we also saw that it wasn't very useful to include a static text style—at least in that example. In general, text styles will be much more useful when the text that is displayed is derived from some property of the feature being displayed. We'll look at how to do that at the end of the chapter, so for now, let's finish up the basic style section with the properties available to text styles:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>font</td>
<td>string</td>
<td>This is a string containing the font size and name of the font to use for rendering the text. The size is typically given in pixels, for example: &quot;18px Verdana&quot;</td>
</tr>
<tr>
<td>offsetX</td>
<td>number</td>
<td>This is the horizontal offset, in pixels, to move the text. A positive number will move the text to the right, while a negative number will shift it left. The default value is 0.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>offsetY</td>
<td>number</td>
<td>This is the vertical offset, in pixels, to move the text. A positive number will move the text down while a negative number will move it up. The default value is 0.</td>
</tr>
<tr>
<td>scale</td>
<td>number</td>
<td>This is an amount to scale the rendered text. A value of 2 will double the size of the text while a value of 0.5 will halve it. The default is 1 (no scaling).</td>
</tr>
<tr>
<td>rotation</td>
<td>number</td>
<td>This is an angle, in radians, to rotate the text. The rotation is clockwise, and the default is 0 (no rotation).</td>
</tr>
<tr>
<td>text</td>
<td>string</td>
<td>This is the text to be rendered.</td>
</tr>
<tr>
<td>textAlign</td>
<td>string</td>
<td>This is the alignment of the text, one of start, left, center (default), right, or end.</td>
</tr>
<tr>
<td>textBaseline</td>
<td>string</td>
<td>This is the baseline of the text, one of top, hanging, middle, alphabetic (default), ideographic, or bottom.</td>
</tr>
<tr>
<td>fill</td>
<td>ol.style.Fill</td>
<td>This is the style to use to fill text characters.</td>
</tr>
<tr>
<td>stroke</td>
<td>ol.style.Stroke</td>
<td>This is the style to use to draw the outline of text characters.</td>
</tr>
</tbody>
</table>

**Multiple styles**

Remember, from the beginning of the chapter, that we said there were three different ways of specifying styles:

- An instance of `ol.style.Style`
- An array of `ol.style.Style` instances
- A style function

We've looked at the first, using `ol.style.Style`, in detail. The second, an array of styles, is really not much different from the first, that is, you are still dealing with the basic styles. The difference is that when you provide an array of styles, features are rendered once for each style. This means each feature is rendered more than once, which can provide some interesting effects. An example is probably the best way of illustrating this.
**Time for action – using multiple styles**

In this example, we'll use the country data again and draw each polygon with two styles to create a shadow effect around the continents.

1. You can start from any of the examples in this chapter. We've started from the first example and removed the time zones for clarity. First, modify the `countryStyle` to provide a somewhat darker stroke:

   ```javascript
   var countryStyle = new ol.style.Style({
   fill: new ol.style.Fill({
   color: [203, 194, 185, 1]
   }),
   stroke: new ol.style.Stroke({
   color: [101, 95, 90, 1],
   width: 1
   }),
   zIndex: 2
   });
   ```

2. Next, we'll add our second style for the shadow effect:

   ```javascript
   var shadowStyle = new ol.style.Style({
   stroke: new ol.style.Stroke({
   color: [0, 0, 127, 0.15],
   width: 8
   }),
   zIndex: 1
   });
   ```

3. Finally, use both styles for the countries layer:

   ```javascript
   var countries = new ol.layer.Vector({
   source: new ol.source.GeoJSON({
   projection: 'EPSG:3857',
   url: '../assets/data/countries.geojson'
   }),
   style: [shadowStyle, countryStyle]
   });
   ```
4. The result should look something like the following screenshot:

![Image of a map with country outlines]

**What just happened?**

By combining two styles, we achieved the desired effect. In the first step, we modified our starting style slightly to sharpen the country outline—this makes them easier to see over the shadows. In the second step, we added another style for the country layer with a wide, mostly transparent stroke and no fill. Lastly, we provided an array as the `style` property for our countries layer.

There is one thing you really need to be aware of when using arrays of styles—each feature is drawn once for each style. This can have dramatic performance implications when dealing with a lot of features. If you create a style array with two styles, you are effectively doubling the number of features being rendered. This will have no visible effect on a small number of simple features, but will start to have a noticeable effect when rendering lots (say hundreds of thousands) of features, or a smaller number of highly complex features.

In practice, it doesn’t really double the rendering time as their efficiencies are built into OpenLayers that try to avoid expensive operations as much as possible. For example, the same effect can be achieved using two vector layers. This will be more inefficient as OpenLayers has to process the geometries twice, while they only have to be processed once when using an array of styles.
### Have a go hero – understanding zIndex

If you have been paying attention, you will have noticed that we added a `zIndex` property to each style in the previous example. Try removing the `zIndex` from both the styles and observe the result. This should give you an appreciation of what the `zIndex` property does. Without it, OpenLayers renders the features one at a time with their complete style, which would mean that the shadow from one country will overlap an adjacent country rather than just forming a shadow around each continent.

### Style functions

Now that we've seen all the basic style properties and how to combine them as arrays of styles, it's time to learn how to use them in conjunction with feature properties to achieve dynamic styles. This is actually the last of our three ways of specifying the style property—the style function.

We said at the beginning of the chapter that a style function is one that returns an array of style objects to be used for a specific feature and zoom level.

What does this mean? It's really quite straightforward, but extremely powerful. A style function is nothing more than a JavaScript function that receives two parameters—the feature being styled, and the resolution of the map's view. It is required to return an array of `ol.style.Style` objects when it is called. For instance, we could have written our country style example using a style function like this:

```javascript
var countryStyleFunction = function(feature, resolution) {
  return [countryStyle]; // the basic style we already defined
};
var countries = new ol.layer.Vector({
  source: countrySource,
  style: countryStyleFunction
});
```

This example doesn’t use the feature or the resolution arguments, but does illustrate how simple the concept of a style function is. In fact, when you provide a style or an array of styles as the style property of a vector layer, OpenLayers creates a style function internally that looks exactly like this.

You might be wondering if it’s this simple, then how can we claim that it's extremely powerful? The power really comes when you use the feature and the resolution to dynamically create styles.
At the end of the previous chapter, we introduced the concept of feature properties and the methods used to retrieve them. Now, we can make practical use of this knowledge by using properties to create styles on the fly.

### Time for action – using properties to style features

For this example, we will render the country layer by styling each country based on income level by associating its country code to income level data provided by the world bank. There are quite a few brackets; so, we've simplified it to four levels: high, medium, low, and poor. We'll draw each country in a color associated with its income level based on these brackets. Let's start from the previous example.

1. At the beginning of the `<script>` tag, before anything else, we will define colors for the four brackets. Use any colours you like:

   ```javascript
   var high = [64,196,64,1];
   var mid = [108,152,64,1];
   var low = [152,108,64,1];
   var poor = [196,32,32,1];
   ```

2. The income levels for each feature are indicated by a code. We need a way to look up the color to use for each code. You don't need to include the comments, they are there to show how we are grouping the income levels:

   ```javascript
   var incomeLevels = {
     'HIC': high, // high income
     'OEC': high, // high income OECD
     'NOC': high, // high income, non-OECD
     'UMC': mid, // upper middle income
     'MIC': mid, // middle income
     'LMC': mid, // lower middle income
     'LIC': low, // low income
     'LMY': low, // low and middle income
     'HPC': poor // heavily indebted poor country
   };
   ```

3. It's good practice to have a default style to use if something goes wrong:

   ```javascript
   var defaultStyle = new ol.style.Style({
     fill: new ol.style.Fill({
       color: [250,250,250,1]
     }),
     stroke: new ol.style.Stroke({
       color: [220,220,220,1],
       width: 1
     })
   });
   ```
4. Our style function will create styles as needed and cache them. Here's the cache and the style function. We'll discuss what it does at the end of the code.

```javascript
var styleCache = {};
function styleFunction(feature, resolution) {
    var level = feature.get('incomeLevel');
    if (!level || !incomeLevels[level]) {
        return [defaultStyle];
    }
    if (!styleCache[level]) {
        styleCache[level] = new ol.style.Style(
            fill: new ol.style.Fill({
                color: incomeLevels[level]
            }),
            stroke: defaultStyle.stroke
        );
    }
    return [styleCache[level]];
}
```

5. Modify the countries layer to use the style function. While you are at it, make sure the source is specified as a separate object. We'll need this in a moment:

```javascript
var source = new ol.source.GeoJSON({
    projection: 'EPSG:3857',
    url: '../assets/data/countries.geojson'
});
var countries = new ol.layer.Vector({
    source: source,
    style: styleFunction
});
```

6. Now, we come to the part where we load the income data and associate it with our features. Add the following somewhere after the source is defined:

```javascript
var key = source.on('change', function(event) {
    if (event.target.getState() == 'ready') {
        source.unByKey(key);
        $.ajax('../assets/data/income_levels.json').done(function(data) {
            source.forEachFeature(function(feature) {
                var code = feature.get('iso_a2');
                if (data[code]) {
                    feature.set('incomeLevel', data[code]);
                }
            });
        });
    }
});
```
7. Give it a whirl, you should see something like the following screenshot:

![Map Screenshot]

**What just happened?**

Let's review what is happening in this example. We are using a standard setup for vector layers, and combining some extra data into our features dynamically. The data is used to style, or classify, the country polygons by the income level as recorded by the world bank. Some interesting things are happening in this example; so we'll look at each step and highlight what's going on.

In step 1, we created some colors to be used to fill countries that fall into one of the four income categories. The number of categories is arbitrary—you can create colors for each of the income levels, or decide to group them differently.

In step 2, we created an object that we can use to look up a color based on an income level, and in step 3, we defined a default style. It's usually good practice when dealing with data that can change to have some kind of fallback so that your code doesn't break. We'll use the default style if we can't find a income level in our lookup object.
In step 4, we created our style function. Another best practice is to reuse style objects as much as possible. Since several countries might be drawn in the same style, we created an empty object (the `styleCache`) to store previously created styles. The actual function comes next. When the style function is called, it gets a reference to the feature being styled and the current resolution of the view on which it is being rendered. We aren't using the resolution in this example. We are using the feature, however. The feature should have a property called `incomeLevel` that matches the values in our lookup tables, so we grab that value and assign it to a variable. If the income level wasn't set or if it doesn't exist in the lookup table, we'll return the default style. Otherwise, we can check to see whether the `styleCache` object already has a style for this income level. If it doesn't, we need to create a new style using the color from our lookup table. In this case, we are using the default style's stroke for everything, but you could easily change the stroke for each feature too. Because we've assigned the new style to the correct slot in the cache, we can then return it directly.

Note that in both cases, we return an array containing the style. This is required for a very important reason: performance! The Style functions are executed a lot of times when rendering vector features and if OpenLayers had to check the return type of each call to see if it was a style object or an array of style objects, this would add significant overhead to the rendering pipeline. For individual features, the difference is so tiny that it's probably not measurable even with the best of tools. For many features, this tiny difference adds up to a lot and it is measurable. The OpenLayers developers have put a lot of effort into this kind of detail and it shows!

In step 5, we modified the layer to use the style function we just created and defined a separate variable for the source. This was to make the next step a little easier.

In step 6, we loaded the income level data. The goal of the code in this step is to load the income data and associate it with the appropriate country feature. To do this, we need to make sure that the country features have been loaded. Once the countries are loaded, then, we need to load the income data. Then we can create a new property on each country feature with the appropriate income level. There are a few important things happening in this step, so we'll review each line:

- **Line 1**: This registers for the change event on the source and assigns the return value to a variable. The sources inherit from `ol.Observable` and so they provide the `on()` method for this. Recall from Chapter 2, *Key Concepts in OpenLayers*, that the `on()` method returns a key that can be used later to deregister an event handler (using the `unByKey()` method)—we'll need it in just a moment. The change event on a vector source is triggered when the source changes state.
Chapter 6

- **Line 2**: This checks the state of the source to see if the source is ready. There are three states—loading, ready, and error—for a vector source, and we are interested to know when the state changes to ready as that's when we can load our income data safely.

- **Line 3**: This unregisters the event handler by using the `unByKey()` method so that the handler doesn't get called again. This is very important. A source triggers the change event when its state changes, but also when any of its features change. This means that our event handler will get called again when we add properties to the feature. Since the state of the source will already be ready, our code will try to load the income data again and will create an infinite loop.

- **Line 4**: This line loads the income data using jQuery's `ajax()` method. The data loaded from this file is passed to the function we register using `done()`.

- **Line 5**: This line starts a loop over each of the features in the source by calling `forEachFeature()` and providing a function that will be called with each feature.

- **Line 6**: This line gets the `iso_a2` property of the feature, which is a two letter code associated with each country. The income data in our file is organized by this code so we can use this code to get the income level for each country.

- **Line 7**: This checks to see if the income data has a value for the current country and line 8 adds the income level as a property of the feature if it does. As we add the property to each feature, the map responds to the change by redrawing itself. You might think that it seems inefficient to redraw the map for each feature change. What actually happens though, is that the map schedules a redraw for the next available render cycle. This won't happen until the current JavaScript process completes, so all the features will be changed and the map will only be redrawn once at some (very short) time later.

These eight lines of code accomplish quite a bit and combine concepts from other chapters of this book, including *Chapter 2, Key Concepts in OpenLayers* (event registration and deregistration) and *Chapter 5, Using Vector Layers* (vector layers, vector sources, and feature properties).

**Interactive styles**

To round off our chapter on vector styles, let's explore combining vector styles with user interaction. In the previous chapter, we responded to the mouse moving over a country by displaying the country's name in an HTML element outside the map. We'll build on this example and take it one step further. As the mouse moves, we'll highlight the country under the mouse and draw its flag and name in the center of the country using a *feature overlay*. The feature overlays are something new; so, we'll need to learn something about them before we go ahead with our example though.
The feature overlays

The FeatureOverlay class, `ol.FeatureOverlay`, is a special type of vector layer designed to render a small number of temporary features in a specific style. It isn't a full-fledged vector layer, but it is highly optimized for the specific case of highlighting features in a temporary way. This sounds ideal for our use case!

Creating a new feature overlay is just like creating any other class in OpenLayers, just pass some options to configure it. Here are the options that can be passed to the `ol.FeatureOverlay constructor`:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>features</td>
<td>`ol.Collection</td>
<td>Array.&lt;ol.Feature&gt;</td>
</tr>
<tr>
<td>map</td>
<td>`ol.Map</td>
<td>undefined`</td>
</tr>
<tr>
<td>style</td>
<td>`ol.style.Style</td>
<td>Array.&lt;ol.style.Style&gt;</td>
</tr>
</tbody>
</table>

The FeatureOverlay class has several methods to manage its features, map, and styles as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addFeature(feature)</code></td>
<td>feature - <code>ol.Feature</code></td>
<td>This adds a single feature to the feature overlay. You may also add features by adding them to the collection returned from <code>getFeatures()</code>.</td>
</tr>
<tr>
<td><code>getFeatures()</code></td>
<td>none</td>
<td>This returns the <code>ol.Collection</code> object used to manage features in this feature overlay. Modifying this collection directly affects the features in the overlay.</td>
</tr>
<tr>
<td><code>getStyle()</code></td>
<td>none</td>
<td>This returns whatever was passed as the <code>style</code> option when constructing the feature overlay or from the last call to <code>setStyle</code>.</td>
</tr>
</tbody>
</table>

To use it, you would do something like this:

```javascript
var overlay = new ol.FeatureOverlay({
  features: [new ol.Feature({}), new ol.Feature({})],
  map: myMap,
  style: [ol.style.Fill({}, {strokeWidth: 1, stroke: 'red'})]
});
```
Method Parameters Description

getStyleFunction() none This returns a function representing the active styles for this feature overlay.

removeFeature (feature) feature - ol.Feature This removes a single feature from the feature overlay. You may also remove features by removing them from the collection returned by getFeatures().

setFeatures (collection) collection - ol.Collection This replaces the features for this feature overlay by using the provided collection of feature objects.

setMap(map) map – ol.Map This sets the map object that this feature overlay will be rendered on, maybe null to remove the feature overlay from a map.

setStyle(style) style – ol.style. Style | Array.<ol.style.Style> | function This sets the style to be used when rendering features.

Feature overlays are pretty simple and most of what you will want to do with a feature overlay once it is configured with a style and map is manage features. Let’s see how that’s done!

**Time for action – creating interactive styles**

Now we have the knowledge we need to build our final example. We will add some interactivity to our countries layer by highlighting the country under the mouse with a different style—specifically, we will:

- Draw the highlighted country with a red outline and semitransparent fill
- Draw an icon at the center of the highlighted country representing its flag
- Draw the country’s name next to the flag

1. First, we’ll need a new file. Let’s start again with the basic country vector layer:

```javascript
var countries = new ol.layer.Vector({
    source: new ol.source.GeoJSON({
        projection: 'EPSG:3857',
        url: '../assets/data/countries.geojson'
    })
});
```
var center = ol.proj.transform([0, 0], 'EPSG:4326', 'EPSG:3857');
var view = new ol.View({
    center: center,
    zoom: 1,
});
var map = new ol.Map({
    target: 'map',
    layers: [countries],
    view: view
});

2. Next, we'll set up some styles for our highlighted features. This code can go right after the map is defined. Don't worry if you don't remember what everything does—we'll review the code at the end:

```javascript
var baseTextStyle = {
    font: '12px Calibri,sans-serif',
    textAlign: 'center',
    offsetY: -15,
    fill: new ol.style.Fill({
        color: [0,0,0,1]
    }),
    stroke: new ol.style.Stroke({
        color: [255,255,255,0.5]
    }),
    zIndex: 1
};
var highlightStyle = new ol.style.Style({
    stroke: new ol.style.Stroke({
        color: [255,0,0,0.6],
        width: 2
    }),
    fill: new ol.style.Fill({
        color: [255,0,0,0.2]
    }),
    zIndex: 1
});
```

3. We'll be using a style function with our feature overlay because we need to dynamically create styles for the feature being rendered:

```javascript
function styleFunction(feature, resolution) {
    var style;
    var geom = feature.getGeometry();
    if (geom.getType() == 'Point') {
        var text = feature.get('text');
```
baseTextStyle.text = text;
var isoCode = feature.get('isoCode').toLowerCase();
style = new ol.style.Style({
  text: new ol.style.Text(baseTextStyle),
  image: new ol.style.Icon({
    src: '../assets/img/flags/' + isoCode + '.png'
  }),
  zIndex: 2
});
} else {
  style = highlightStyle;
}
return [style];
}

4. We also need to create the feature overlay itself. It's pretty straightforward as the style function is doing all the work for us:
var featureOverlay = new ol.FeatureOverlay({
  map: map,
  style: styleFunction
});

5. Finally, the interactive part. We'll add a handler for the map's pointermove event and manage the features in our feature overlay based on where the mouse is. This is a pretty big function that exercises our knowledge of geometries from the previous chapter:
map.on('pointermove', function(browserEvent) {
  featureOverlay.getFeatures().clear();
  var coordinate = browserEvent.coordinate;
  var pixel = browserEvent.pixel;
  map.forEachFeatureAtPixel(pixel, function(feature, layer) {
    if (!layer) {
      return; // ignore features on the overlay
    }
    var geometry = feature.getGeometry();
    var point;
    switch (geometry.getType()) {
      case 'MultiPolygon':
        var poly = geometry.getPolygons().reduce(function(left, right) {
          return left.getArea() > right.getArea() ? left : right;
        });
        point = poly.getInteriorPoint().getCoordinates();
        break;
Styling Vector Layers

```javascript
    case 'Polygon':
        point = geometry.getInteriorPoint().getCoordinates();
        break;
    default:
        point = geometry.getClosestPoint(coordinate);
    }
    textFeature = new ol.Feature(
        geometry: new ol.geom.Point(point),
        text: feature.get('name'),
        isoCode: feature.get('iso_a2').toLowerCase()
    ));
    featureOverlay.addFeature(textFeature);
    featureOverlay.addFeature(feature);
});
}
```

6. Load this in your browser and try it out! You should see something like the following screenshot when you move the mouse over a country:

![Screen shot of interactive vector layer](image_url)

What just happened?

As you can see, feature overlays make it very simple to create an interactive experience with vector layers. There are a few new concepts in this example, as well as some old ones, so let's review the code step by step.

Step 1 should be pretty familiar to you now—we are creating a vector layer with a GeoJSON source, and adding it to a map with a view centered on 0, 0.
In step 2, we set up some styles for our feature overlay to use. There are two styles—one for text and one for polygon highlighting. Notice that the `baseTextStyle` is an object literal, not a new instance of `ol.style.Text`. When you create an instance of `ol.style.Text`, the text to be drawn needs to be passed to that object and you can't change the text after it has been created. The style function allows us to create text styles with the text of the current feature, but we'll need to specify the other options. Since all labels will share the other text options, we can set them up once and just refer to them later in the style function. The options we've specified here are to center align the text, offset it up by 15 pixels (recall that a positive `offsetY` moves the text down), and provide a fill and stroke color. For the text, the stroke is rendered around the outside of each character so we set a wide, semitransparent stroke to make the text stand out from the map beneath it.

The `highlightStyle` is straightforward—a fill and stroke style for the highlighted polygon.

The style function needs to be defined before we can use it to create the feature overlay, so in step 3, we defined it. Recall that the style function receives the feature being rendered as the first argument. We drew two types of features, points, and polygons, with two different styles. So, the first thing we did was get the feature's geometry and check to see whether it was a point. If it is a point, we create a new text style and a new icon style specific to the current feature. We got the feature's `text` property and combined it with the `baseTextStyle` object to create the text style. Next, we got the `isoCode` property and used it to create a URL to the flag for the country (the flag icons are conveniently named using the two-letter ISO country code) for a new icon style. Then, we created a new style object for the current feature. If the feature is a polygon, it's much simpler—all we need to do is return the `highlightStyle` object. Finally, we returned an array containing the style (recall that style functions are required to return arrays for performance reasons).

Step 4, by comparison, was very short! We created a new feature overlay and configured it with the map object and style function. It's really the style function, and step 5, that do all the work.

Step 5 added a handler for the map's `pointermove` event; so, we could find the feature closest to the mouse and add it to the feature overlay. We actually wanted to highlight two features—the polygon itself and a point at the center of the polygon. It turns out that getting this center point is a bit tricky. Let's review the code carefully.

Line 1 of code in the handler clears any existing features in the feature overlay. It is much easier to retrieve the collection and clear it than to remove individual features in our case.

The `browserEvent` object provides us with the map coordinate and pixel that the event happened at. We used the pixel location with `ol.Map` classes `forEachFeatureAtPixel` function on the next line to retrieve all features at that location from our vector layer. This function invokes a callback function for every feature at the pixel location, providing both the feature and the layer that the feature was found on. The layer parameter may be null if the feature was found on a `FeatureOverlay`. 
Inside our callback function, we tested first to see if the feature was actually on a layer before proceeding. Then we needed to find both the geometry and the center point of the geometry. If all the country features were polygons, we could simply call `getInteriorPoint()` to retrieve the center and we would be done. Unfortunately, we didn't know what type of features we were dealing with—some of the features in the country data were actually `MultiPolygons`, and we needed to handle them differently. The switch statement chooses a path based on the type of the feature. Let’s look at each case separately.

The first case was for `MultiPolygon`. As the name suggests, a `MultiPolygon` class contains multiple polygons (a country and some islands perhaps) and there isn’t a convenient way to determine the center of a group of polygons. Instead, a `MultiPolygon` class has several centers, one for each of its constituent polygons. The `getInteriorPoints()` method returns the center points for us. We only really wanted a single label though. One way is to get all the interior points and use the first one. The problem with this approach is that there is no particular order to the polygons, and it looks odd to label some random island off the coast rather than the major landmass of a given country. To get around this, we wanted our label to appear at the center of the largest polygon. To get the largest polygon, we first got the array of polygons and then reduced that array to a single value with a function that compared two polygons based on their area. The reduce method is a standard method of JavaScript arrays. Once we’ve found the largest polygon, we can ask for its interior point.

The second case is for `polygon`, and was much simpler— we just needed the interior point of the polygon and we were done!

The final case was the default case. While it isn’t strictly needed, it is good practice to include a default case in switch statements. The `getClosestPoint()` method is available on all geometry types and is a safe fallback for our default case.

Now that we had a point coordinate at the best location we could determine for the feature under the mouse, we created a new Feature and provided the point geometry for its location. We also added two properties—text (that we displayed at the point) and `isoCode` (that we will use to find the flag icon for the country).

Finally, we added both features to the feature overlay so that when the map is next rendered, the country under the mouse will be highlighted in red and display the flag and country name at the center.
Summary

This concludes our chapter on styling vector layers. OpenLayers' styling capabilities are quite simple to implement, but can be incredibly powerful. The basic styles are easy to set up and give you a lot of flexibility in styling your vector features. More complex styling is readily implemented through the use of a style function by allowing you to create basic styles tailored to a specific feature and current zoom level of the map. The building blocks—basic styles and the style function—are simple, but the combination of them allows you to create highly custom cartographic representations for your vector data.

In the next chapter, we will dig into projections and discover how to use them to display both vector and raster layers in different projections.
Where to buy this book

You can buy OpenLayers 3 Beginner's Guide from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.