The book of **OpenLayers 3**

Theory & Practice

Antonio Santiago

The book of OpenLayers 3

Theory & Practice

Antonio Santiago

This book is for sale at http://leanpub.com/thebookofopenlayers3

This version was published on 2015-03-04

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Antonio Santiago by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought The book of OpenLayers 3 #thebookofopenlayers3

The suggested hashtag for this book is #thebookofopenlayers3.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#thebookofopenlayers3

To my wife Pilar. To my parents. To my family.

To the OpenLayers community and to anyone who wants to learn.

Contents

About the book

Nowadays, a great degree of data is susceptible to be located and visualized in a map, from geological or climate data to marketing and sales information.

Geographic information has become one of most important and valuable kind of informations.

Professionals from many industries needs to know and be up to date with all the related GIS technologies: spatial databases, map and feature servers, desktop or web application, frameworks, libraries, etc.

Within all this network, web technologies are one that has grown more in last decade due, in part, by the browsers performance evolution.

Who is this book addressed ?

This book is for anyone interested on *Geographic Information Systems* (GIS) technologies and, concretely, on web mapping based on the new version of OpenLayers library.

OpenLayers3 is one of the most complete and powerful open source GIS solutions for web development.

Whether you are an experienced user or a new OpenLayers user, this book is a great reference to start learning the new concepts and API of the OpenLayers3. Learn to create maps, add controls and animations, add data from OGC compliant servers using standard formats, work with vector layers, style features, etc.

How is the book organized ?

No one becomes an expert reading a book. Learn anything implies two things: understand concepts (the theory) and obtain experience working in real world samples (the practice).

Because of this, I have wanted to create a book as a mix between an usual programmers book and a cookbook. All chapters has been divided in two sections, the theory where I explain the chapter related concepts, and the practice, where we can see simple but real examples.

The chapters follows the order I consider are the logical path to introduce, understand and learn how to work with OpenLayers3.

Why I wrote this book?

There are many reasons to write a book: to teach others, to earn money, for fame, ... but mine is much more selfish than any of those. I wrote this book to learn.

As developer, learn new technologies and be up to date is part of my job. My day to day is a mix between front-end and back-end developer and OpenLayers is one of the tools in my toolbox. After looking at the new OpenLayers3 API it was clear the project had been made a great evolution (new concepts and completely new API) so I started to look at it.

As someone says the best way to learn is to teach[1]. That is the real reason I started writing this book. I hope you enjoy it as I enjoy writing it.

The book cover

The book cover is the work of my friend Hugo Tobio[2] an awesome illustrator. It is based on the OpenLayers logo, its colors and shape. It expresses the ability to build great things with the components offered by the library.

[1] http://en.wikipedia.org/wiki/Frank_Oppenheimer
[2] http://hugotobio.com

Before to start

A brief history

The arrival of Google Maps in 2005 was a revolution for the world of the web mapping. It offers to the world the opportunity to explore their environment, opening the eyes to the people to understand the importance of the *location*, and gives to the developers a tool to create thousands of new map uses. A new era began.

Nowadays location is almost everywhere. It is common to see companies websites using a small map to show where they are located or companies offering location based services, for example, to store your geo-tagged vacation photos.

For years the visualization of geographic information resided on powerful desktop applications, also prepared for analysis, transformation and edition, but Google Maps started the revolution of the web mapping applications demonstrating the browsers, and JavaScript, can play an important role in the world of the Geographic Information Systems (GIS).

At the same time of the web mapping libraries born another important revolution took place in the GIS industry: the standardization. Projects like GeoServer or MapServer becomes serious competitors to proprietary GIS servers. Both open source, with powerful features and, more important, implementing many of the standards defined by the Open Geospatial Consortium[3] (OGC), like Web Map Service[4] (WMS), Web Feature Service[5] (WFS), Web Map Tile Service[6] (WMTS), Simple Feature Access[7] (SFS), Geography Markup Language[8] (GML), Styled Layer Descriptor[9] (SLD), etc.

Born of OpenLayers

OpenLayers appears in the middle of 2006 as an open source alternative to Google Maps and other proprietary API providers, but it starts gaining more attention in 2007, when the growing OpenStreetMap project adopts it for its website.

The popularity of OpenLayers has grown with the years, evolving and improving with the addition of new features. Nowadays, it is very mature project and probably the most complete and powerful open source web mapping library to work with geographic information.

[3] http://www.opengeospatial.org/
[4] http://en.wikipedia.org/wiki/Web_Map_Service
[5] http://en.wikipedia.org/wiki/Web_Feature_Service
[6] http://en.wikipedia.org/wiki/Web_Map_Tile_Service
[7] http://en.wikipedia.org/wiki/Simple_Features
[8] http://en.wikipedia.org/wiki/Geography_Markup_Language
[9] http://en.wikipedia.org/wiki/Styled_Layer_Descriptor

One of the key aspects of OpenLayers is the fact it implements many of the Open Geospatial Consortium standards so it becomes the perfect candidate to work against geographic information servers.

After years of feature additions, improvements, evolutions and bug fixes OpenLayers begun to show its age. The technology available seven years ago to build a toolkit are not the same than nowadays. On that time new projects has appeared and consolidated for front-end development: DOM manipulation or UI creation (jQuery[10], Dojo Toolkit[11] or ExtJS[12]), a new HTML version (HTML5) has been released with many awesome features implemented by many modern browsers (like Canvas[13] or WebGL[14]) and the evolution and performance improvement of JavaScript language. All these factors, among others, has determined the need to create a new version of OpenLayers from scratch.

OpenLayers3

OpenLayers is a JavaScript web mapping toolkit that offers all the required components to work with geographic information in the browser side.

The complexity of a solution that satisfies all the needed requirements goes beyond the simple idea of visualize raster and vector data in the browser. We need to request data from (or store to) many different data sources, from map servers implementing OGC standards to simple plain files. We need to read from (or write to) many data formats: GML, KML, GeoJSON, etc. We need to work with data in different projections. We need the concept of *layer* where to place different kind of data and we need to play with these layers: hiding or showing, raising or lowering on each other. We need to style the features depending on its attributes, for example we can render cities as points with different radius depending on its population.

Hopefully, OpenLayers satisfies all these and much more requirements.

As developers, our duty is to understand how the tools we use work and how they are made. Understanding how OpenLayers3 is designed is a crucial step before start working with it.

I encourage the reader to explore the OpenLayer3 source code. It is the best place to learn in depth how this great library was designed and works.

Features

OpenLayers3 is a completely rewritten version of the project, centered to make it a really up to date project and offering mobile support out of the box.

[10] http://jquery.com/
[11] http://dojotoolkit.org/
[12] http://www.sencha.com/products/extjs/
[13] http://en.wikipedia.org/wiki/Canvas_element
[14] http://en.wikipedia.org/wiki/WebGL

The API has suffer in depth changes and the programming style has changed to avoid long namespaces, avoiding those extremely long sentences required in previous versions.

The weight of the library has been reduced drastically and it is much more lightweight than in previous versions. In OpenLayers3 the production ready version is about 300kb containing all the functionalities.

The renderers has been updated to make use of WebGL, Canvas or DOM elements depending on the browser capabilities. Renderers are the piece of code responsible to draw points, lines, polygons, tiles, etc on the screen. So it is important this action has a great degree of performance to run faster in mobile and desktop applications.

OpenLayers3 is based on the Closure Library and rely on the entire Closure Tools[15] suite. The Closure Compiler *minimizes* the code, producing an extremely compact and high performance version through advanced optimizations (like variable and property renaming, unused code removal or function inlining). On the other hand, the Closure Library is a general purpose JavaScript library which allows create modular application, handle module dependencies, DOM manipulation, etc.

A great project requires a great documentation, because of this OpenLayers3 makes use of the JSDoc[16] tool. JSDoc is a JavaScript API documentation generator that offers an extensive syntax notation to put on the source code and, automatically, generate the project API documentation.

OpenLayers3 offers 3D capabilities in the browser, based on the Cesium[17] project. This means the same map could be rendered using 2D or 3D views, opening new possibilities to developers.

When a wrote this lines 3D support is not implemented yet, but there is the promise to add it.

Getting ready for programming with OpenLayer3

There are many ways to organize a JavaScript application: folders structure best practices, modules, asynchronous loaders, etc. All these concepts are beyond the scope of this book but, obviously, anyone can need them on a real application.

Use OpenLayers3 on your application requires basically two steps:

- Include the JavaScript file, typically `ol.js`, with the library implementation.
- Include the CSS file, typically `ol.css`, with the styles for some the elements of the library, like the map or the controls.

[15] https://developers.google.com/closure/
[16] http://usejsdoc.org/
[17] http://cesiumjs.org/

Depending on your needs, the files can be hosted and provided by your own server or be provided by a *Content Delivery Network* (CDN). Each solution has its pros and cons.

Serving the files from your own servers implies more data must be transfered to every client which must be token into account on sites with big traffic. On the other hand, getting the files from an external server can leverage the bandwidth issues but implies you are dependent of an external service you may can not take control, if the service is down, your application will not work.

Basic code structure

For the examples that accompany this book, we have follow some simple best practices for small web applications that can be summarized as: *put CSS at top and JavaScript at bottom.*

The next code shows the basic structure:

```
    <!DOCTYPE html>
    <html lang="en">
        <head>
            <title>Our app title</title>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width">

            <!-- OpenLayers CSS -->
            <link rel="stylesheet" href="http://ol3js.org/en/master/build/ol.css\
" type="text/css">

            <!-- Our app styles -->
            <style>
            </style>
        </head>
        <body>
            <!-- Our app HTML tags here -->

            <!-- OpenLayers JS-->
            <script src="http://ol3js.org/en/master/build/ol.js" type="text/java\
script"></script>

            <!-- Our app code -->
            <script>
            </script>

        </body>
    </html>
```

In addition, the samples uses Bootstrap[18] framework to define the layouts and better style the components, because of this you can see the use of CSS classes like `row`, `col-md-6` or `btn`. Do not worry about that. The goal of the samples is to explain the JavaScript code related to OpenLayers3 and not to become a great web designer.

How to debug an OpenLayers3 application

Often, at development time, it is needed to see what is happening at our code: check variable value, function context, loops, conditions, etc. For this purpose, most modern browsers offers tools to allow not only to debug our JavaScript code (setting breakpoints, watchers, ...) but also helping with CSS (visualizing, editing on the fly or computing the properties of an element) and HTML (allowing to modify on the fly the code).

On FireFox browsers FireBug[19] is a well known tool, although latests versions includes a built in tool called the Firefox Developer Tools[20]). Similarly, Chrome browser has the built in tool called Chrome DevTools[21].

Previous tools allows us to debug the source code of our application, but what if what we need is to analyze the OpenLayers3 source code? For me, this is the best way to learn how OpenLayers works: how it process remote data and transform into features, how map renders layers, etc.

Fortunately for us, OpenLayers3 is distributed in two different versions:

- `ol.js`, contains all the OpenLayers3 source files concatenated and minimized using the Google Closure compiler (which reduces file length renaming variables and properties, removing unused code, ...). **It is specially designed for production environments because its lightweight**.
- `ol-debug.js`, contains all the OpenLayers3 source files concatenated. **It is suitable for debug purposes** because we can navigate step by step through the OpenLayers3 internal code.

So, if you never need to study the OpenLayers3 source code, the `ol-debug.js` file is the right file to be linked in your application.

[18] http://getbootstrap.com/
[19] http://getfirebug.com/
[20] https://developer.mozilla.org/en-US/docs/Tools
[21] https://developers.google.com/chrome-developer-tools/

1. The Map and the View

Among all the elements in the OpenLayers3 puzzle, the map is probably the main piece, so it is natural to start introducing to OpenLayers3 describing how to work with it.

Contrary to previous versions, OpenLayers3 differentiates between the concept of *map*, which takes care of layers, controls, overlays, etc and the way we visualize it, which is done by the *view*. The view is like a window through which we see the map. It allows to change the location we are *looking* at, go closer or farther.

The concept of view allows us to make things like render the same map in different views (centered at different places), render the same map in a 2D view and in a 3D one, or render different maps using the same view.

The goal of OpenLayers3 is to offer two implementations of the view: 2D and 3D. Unfortunately, when I write this lines, only a 2D view implementation is available but there are plans for creating a 3D view based on the Cesium[1] project.

1.1 The Map

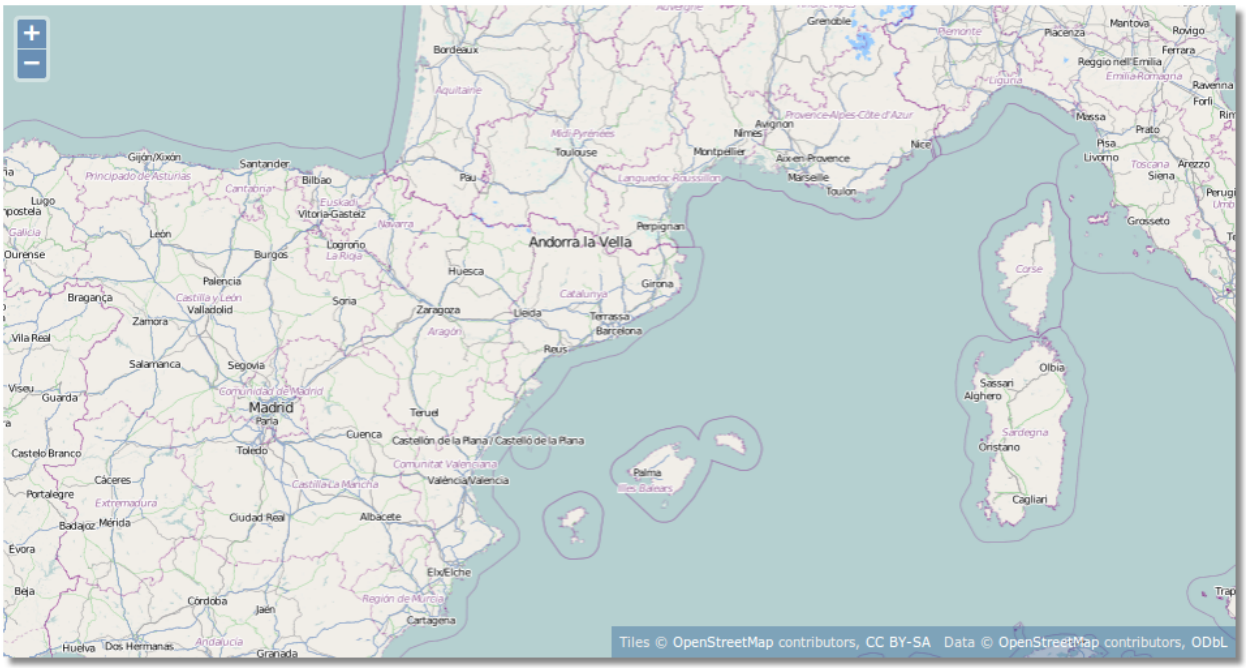
The `ol.Map` is the class that allows to handle the concept of map in our application. We can add or remove *layers*, *controls*, *overlays* and *interactions*.

On its simplest form, creating a new `ol.Map` instance requires the user specifies an object with the next properties:

- `target`, the target HTML element (a DOM[2] node), where the map will be rendered
- `layers`, one or more layer references with the data to be shown
- `view`, an `ol.View` instance responsible to manage the way to visualize the map.

[1] http://cesiumjs.org

[2] http://en.wikipedia.org/wiki/Document_Object_Model



A basic map

So, given the next DOM element, that will act as the target:

```
<div id="map" class="map"></div>
```

That uses the next style to set its dimensions:

```
.map {
    width: 600px;
    height: 400px;
}
```

A basic map can be easily created with the next code:

```
var map = new ol.Map({
    target: 'map',
    layers: [
        new ol.layer.Tile({
            source: new ol.source.OSM()
        })
    ],
    view: new ol.View({
        center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
```

```
10  '),
11                  zoom: 6
12              })
13      });
```

The `target` can be a string with the element identifier or a reference to the element itself, for example, retrieving it using `document.getElementById('map')`. The map will be created within the target element and will fill it completely. The target allows to layout or style the map within the web page.

The `layers` array must contain instances of layers, defined at the `ol.layer` namespace. In the sample, we are using a tile based layer that uses OpenStreetMap[3] project as data source. Don't worry about the code related to layers at this point, we will cover it in detail in next chapters.

Finally, the `view` instance we are passing is a 2D view initialized at zoom level `6` and centered near Barcelona city. As the `ol.source.OSM` source uses the projection `EPSG:3857`, the center of the view must be set in the same projection and, because the center is specified in `EPSG:4326`, we must transform it to be in `EPSG:3857`.

Explain map projections[4] is out of the scope of this book. We only cover how to make a few transformations to work with data in OpenLayers3.

It is worth to say that, by default, if no controls are specified when a map instance is created it is automatically initialized with the controls: attribution, logo and zoom (which we will cover in next chapters).

1.1.1 Map properties and methods

Although we usually initialize a map using the previous properties, the `ol.Map` class uses internally only four properties: the `target`, the `layergroup`, the `view` and `size`.

The `target` and the `view` were described in previous sections, while the `size` contains an array with the map size in pixels.

The `layergroup` property is a reference to a `ol.layer.Group` instance, which stores the references to the layers of the map. Thus, the initial set of layers we specify in the `layers` property at initialization time are stored in the `layergroup`.

We will discuss in depth the `ol.layer.Group` class in the Layer Groups section on Layers chapter.

[3] http://www.openstreetmap.org
[4] http://en.wikipedia.org/wiki/Map_projection

The `ol.Map` class offers methods to get and set its properties in addition to methods needed to work with layers, controls, etc. In this section we will cover the most basic ones while we will see the rest in next chapters.

All the map properties has its corresponding getter and setter methods: `getTarget`, `setTarget`, `getView`, `setView`, `getSize`, `setSize`, `getLayerGroup` and `setLayerGroup`.

```
    var size = map.getSize();   // [x,y] pixel size
    var view = map.getView();
```

In addition to the `getLayerGroup`, which returns a `ol.layer.Group` reference, the map offer the `getLayers` method which returns an array with the layers of the map:

```
    var group = map.getLayerGroup();   // ol.layer.Group instance
    var layers = map.getLayers();        // [layerA, layerB, ...]
```

Read the Controlling the layer stack section on Layers chapter to learn more about the difference of `layers` and `layergroup`.

1.1.2 What really happens when a map is created

When an `ol.Map` instance is created on a target DOM element, OpenLayers3 creates a new `div` element within the target, called the *viewport*, that is the real location where the map, the controls or the overlays are placed.

As example, next we show the code created for a DOM based map on a target `domMap` element:

```
    <div id="domMap" class="map">
        <div class="ol-viewport" style="position: relative; overflow: hidden; wi\
dth: 100%; height: 100%;">
            <div class="ol-unselectable" style="position: absolute; width: 100%;\
 height: 100%;">...</div>
        </div>
        <div class="ol-overlaycontainer"></div>
        <div class="ol-overlaycontainer-stopevent">
            <div class="ol-attribution ol-unselectable">...</div>
            <div class="ol-logo ol-unselectable">...</div>
            <div class="ol-zoom ol-unselectable">...</div>
        </div>
    </div>
```

If for any reason you will need access to this element you can obtain a reference from the map instance with the `getViewport` method.

```
var map = new ol.Map({
    target: "domMap",
    ...
});

var viewport = map.getViewport();
```

1.1.3 Different ways to render the map

OpenLayers3 comes with the ability to render the maps using three different technologies: DOM, canvas and WebGL.

Using DOM renderer, all map elements are drawn using HTML elements, for example using `img` for tiles or `svg` for features. With canvas, the maps are rendered using the HTML5 canvas element, which offers an scriptable way to render 2D shapes and bitmap images. Finally, WebGL renders the map using the WebGL technology, a subset of OpenGL standard suitable for browsers and that allows to take advantage of the GPU[5] power.

You can force the map to use a specific renderer using the `renderer` property. It accepts a single value, within the list `canvas`, `webgl` or `dom`, or an array of values that determines the order elements will try to be rendered.

```
var domMap = new ol.Map({
    ...
    renderer: 'canvas'
    ...
});
```

By default, if none of the previous properties are specified, OpenLayers3 sets the `renderer` property to the next hints `['canvas', 'dom', 'webgl']`. This means when a new map instance is created OpenLayers3 tries to use the Canvas renderer if it is supported by the browser, otherwise tries to use the DOM mechanism and, finally, if neither is supported uses the WebGL technology.

Take into account DOM mechanism has the worst performance.

1.2 The View

The concept of view is handled by the `ol.View` class and, as we have say previously, it determines how the map is visualized allowing to change the zoom level, the center location or the rotation angle.

[5] http://en.wikipedia.org/wiki/Graphics_processing_unit

When instantiating a new view we need, at least, to specify a `center` location and `zoom` level (or a `resolution` value) to have a full functional view instance:

```
var view = new ol.View({
    center: [0, 0],
    zoom: 2
});
```

1.2.1 Controlling the view

The view is controlled by three properties: `center`, `resolution` and `rotation`. For this purpose, `ol.View` class offers getter and setter methods like: `getCenter`, `setCenter`, `setResolution`, `getResolution`, `getRotation` and `setRotation`.

The view also offer the `getZoom` and `setZoom` methods that also modifies the view's zoom level. As we will see in the next section Resolutions and zoom levels, the `ol.View` works internally with the `resolution` property so modify the zoom level implicitly modifies the view's resolution.

Given the next view instance:

```
var view = new ol.View({
    projection: 'EPSG:4326'
});
```

we can set the center location:

```
view.setCenter([2.1833, 41.3833]);
```

In OpenLayers3 we can specify coordinates with a simple array with two values for *longitude* and *latitude*. Older versions requires to create an instance of the `OpenLayers.LonLat` class to work with locations.

> We usually talk about latitude and longitude while OpenLayers requires the center location be specified as `[longitude, latitude]` array. Why? This is because when translates latitude and longitude to a Cartesian plane, we express location using `[x,y]` array. The `x` element represents the horizontal displacement and the `y` element the vertical displacement. Translated to geographic location the horizontal displacement is the longitude while the vertical displacement is the latitude.

We can also change the resolution or zoom level:

```
1       view.setResolution(12000);
2       view.setZoom(7);
```

or rotate the view 5 degrees:

```
1       view.setRotation( 5 * Math.PI / 180);
```

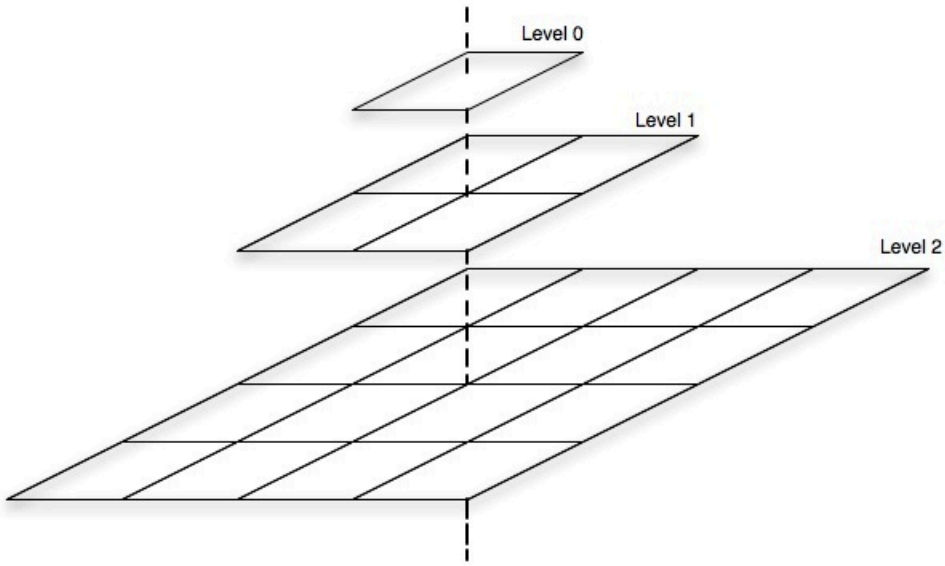
Note the rotation angle must be expressed in radians, because of this we need to transform degrees to radians.

In addition, the view has also a `projection` property, which is used to determine the coordinate system of the center. By default, the projection used is `EPSG:3857`. The properties `getProjection` and `setProjection` allows us to retrieve and set the projection value.

1.2.2 Resolutions and zoom levels

Usually, we talk about zoom levels and resolutions as if they were the same thing. It is true both are close related but it is important to understand the differences between them.

Lets take as example the tile images from the OpenStreetMap project. At level zero, one tile is used to show the whole world. At level one, four tiles are used to show the world. At level two, eight tiles. And so on. Following this rule, we can see each zoom level is formed by 2^nx2^n tiles, where `n` is the zoom level.



Zoom levels

Now, lets go to compute the resolution of each level. In `EPSG:4326` projection, the length of the equator measures `40,075,016.686 meters`. Working with a tile size of 256x256 pixels, that means, at zoom level zero the resolution of each pixel is `40,075,016.686 / 256 = 156,543.034 meter/pixel`. At level one, two tiles are required to visualize the world so each tile represents `(40,075,016.686 / 2) meters`, the resolution of each pixel is `(40,075,016.686 / 2) / 256 = 78,271.52 meters/pixel`. Next table shows the relation among zoom levels, number of tiles and resolution:

Zoom	Tiles	Resolution (meters)
0	1x1	156,543.03
1	2x2	78,271.52
2	4x4	39,135.76
3	8x8	19,567.88
…	…	…

You can find more detail at the Slippy Map[6] web page from OpenStreetMap project.

Looking at the above table, we can understand why no matter which method to use to modify the view, if `setZoom` or `setResolution`, because both modifies the `resolution` property, the first specifying the zoom level and the second the resolution directly.

Even so the `setZoom` is probably more used than `setResolution`, since it is more easy to use because does not requires to remember big numbers.

[6] http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames#Resolution_and_Scale

1.2.3 The view properties

We have seen the the `ol.View` class has three important properties through which we can control it: `center`, `resolution` and `rotation` (plus the `projection` to indicate the view's center coordinate system) but, in addition, it has other properties that can help us to configure it:

- `resolutions`, an array of valid resolutions that determines the zoom levels,
- `maxResolution`, the maximum resolution (corresponds to the resolution value at zoom level zero),
- `maxZoom`, the maximum number of zoom levels, we could change the zoom level from level zero to this value,
- `zoomFactor`, we can understant it as the step value used to increase the zoom. By default it is `2` because of this each zoom level has half of the resolution of the previous level.

Note, these properties can only be used at instantiation time, there are no methods to modify its values once the view is initialized.

We need to take into account **if the `resolutions` property is passed to the view constructor the other three will be ignored** or, said with other words, we can not pass the `resolutions` property if we want to make use of the other properties. They are exclusive.

The `resolutions` property takes precedence over the `maxResolution`, `maxZoom` and `zoomFactor`. This is true because we can compute them from the `resolutions` array.

Next code shows a view instance initialized using the `resolutions` property:

```
1     var view = new ol.View({
2         center: [0, 0],
3         zoom: 0,
4         resolutions: [78271.52 , 39135.76]
5     });
```

This means the view will have two zoom levels. The zoom level `0` that corresponds to the resolution value `78271.52` and the zoom level `1` that corresponds to the resolution `39135.76`. Because the second value is half of the first, the `zoomFactor` of the view is `2`. The `maxResolution` property corresponds to the value `78271.52` and the `maxZoom` is `1`.

Now, compare to the next example:

```
var view = new ol.View({
    center: [0,0],
    zoom: 0,
    maxResolution: 78271.52,
    maxZoom: 1,
    zoomFactor: 2
});
```

With this configuration we will have two zoom levels, because the `maxZoom=1`, level `0` with a corresponding resolution of `78271.52` and level `1` that will be automatically computed as `78271.52 / 2 = 39135.76`. If we were specified a `maxZoom=2` we have been got three zoom levels, from `0` to `2`, and the zoom level `2` would be computed as `39135.76 / 2 = 19567.88`.

1.2.4 Other useful methods

In addition to the getter and setter methods previously seen, the `ol.View` offers other methods can be useful in some situations.

- `fitExtent`, adjust the view location and zoom level to a given extent,
- `calculateExtent`, returns the current extent visualized by the view,
- `constrainResolution`, given a resolution value returns the closest valid available resolution,
- `constrainRotation`, adjusts a rotation value given a set or rules.

So, given a map instance we can get the current extent we are visualizing with:

```
var view = map.getView();
var extent = view.calculateExtent( map.getSize() );
```

Because a view can be used for more than one map instance, you need to specify the map size in the `calculateExtent`, that is, the view can not retrieve automatically the map size because it has no reference to the map.

Now, we can navigate in the map and return to the original place applying the previous extent:

```
view.fitExtent( extent, map.getSize() );
```

Supposing the next map instance that uses a view with two resolutions:

```
    var map = new ol.Map({
        ...
        view: new ol.View({
            resolutions: [78271.52 , 39135.76],
            ...
        }),
        ...
    });
```

we can change the view resolution specifying a zoom level with the `setZoom` method:

```
    view.setZoom(1);
```

or we can change the view resolution to any desired value using `setResolution` method:

```
    view.setResolution(40000);
```

But.. what if we want to restrict resolution values to one of the valid values defined? Using the `constrainResolution` method we can constrain the resolution value before setting it:

```
    var resolution = map.constrainResolution(40000);    // This will return 3913\
5.76
    view.setResolution(resolution);
```

1.3 Animations

Modifying the view using its setter methods gives us lot of flexibility but, unfortunately, the changes on its properties (zoom level or center location) can be too much sudden. Sometimes what we desire is a more user friendly movement, like a nice displacement to some location. For this purpose, OpenLayers3 offers the concept of *animation*.

1.3.1 The animation functions

Animations are functions specially designed to tween the view properties (center, rotation and resolution) producing a nice transition effects.

Next are all the available animations, which resides within the `ol.animation` namespace:

- `ol.animation.pan`, modifies the view's center,
- `ol.animation.rotate`, modifies the rotation angle,

- `ol.animation.zoom`, modifies the zoom property,
- `ol.animation.bounce`, modifies the resolution property to create a bounce effect.

Each of these functions returns a new one that does the real animation task. Because each function is appropriate for a view property (center, rotation or resolution) each function requires we need to specify at least one option. For the `ol.animation.pan` we need to specify the `source` point where to start the animation:

```
1    var pan = ol.animation.pan({
2        source: ... // Initial center location
3    });
```

For the `ol.animation.rotate` we need to specify at least the `rotation` options with the angle value.

```
1    var rotate = ol.animation.rotate({
2        rotation: ... // Initial rotation angle
3    });
```

Finally, for the `ol.animation.zoom` and `ol.animation.bounce` we need to specify the `resolution` option.

```
1    var zoom = ol.animation.zoom({
2        resolution: ... // Initial resolution
3    });
```

1.3.2 The tween functions

In addition to the animation functions, OpenLayers3 offers a set of tween[7] functions that allows control the transition of the values to be modified. This way, we have more degree of flexibility, because we can mix the animation function with the tween function it must be used.

A *tween function* allows to modify a variable in a specified duration of time from an initial to a final value. Within that duration, each time we execute the function it will return a value between the initial and final according to a mathematic formula.

The tween functions are located at the `ol.easing` namespace and they are:

- `ol.easing.easeIn`, start slow and speed up,
- `ol.easing.easeOut`, start fastest and slows to a stop,
- `ol.easing.inAndOut`, start slow, speed up, then slow down,

[7] http://en.wikipedia.org/wiki/Inbetweening

- `ol.easing.linear`, constantly in time,
- `ol.easing.bounce`, bounce effect,
- `ol.easing.elastic`, elastic effect,
- `ol.easing.upAndDown`, increases and decreases a value.

Note the `ol.easing.upAndDown` is more suitable to be used with the `ol.animation.bounce` animation.

All the animations accepts the `easing` and `duration` options. With the `easing` we can specify the tween function to be used, while with the `duration` we specify the time of the transition in milliseconds:

```
1     var pan = ol.animation.pan({
2         duration: 2000,
3         easing: ol.easing.bounce,
4         source: map.getView().getCenter()
5     });
```

1.3.3 Applying animations

The map is almost constantly refreshing. Each time a change is produced, like when you pan the map or when a layer loads new data, the map is completely rendered.

The renderer process is reasonably complicated. It needs to render all the data layers of the map, both raster and vector, the controls and the overlays. All this taking into account properties like the projection, the view center or the resolution, not to mention the features styling, and with the addition it can be made using three different mechanisms: WebGL, canvas and DOM.

You'll be asked why I'm talking about the rendering process, and the answer is because the animations are placed at the beginning of it.

The `ol.Map` class offers the `beforeRender` method that accepts a so called *preRender* function as parameter. If you look at the OpenLayers3 source code you will find they are functions with the `ol.PreRenderFunction` type definition. Hopefully for us, all the functions in the `ol.animation` namespace are suitable to be used as *preRender* functions.

After all the explanation, we can summarize that to create an animation we need to follow the next three steps:

- Create the desired animation function, optionally setting the desired tween function,
- Attach it to the map, using the `beforeRender` method,
- Make the change on the view to trigger the animation.

Remember the options passed to the animation function acts as the initial value while the value specified when modifying the view acts as the final value, so the animation will go from the initial to the final value using the specified tween function.

Next code creates a pan animation from the current view center to the coordinate origins:

```
    // Define an animation function setting the current view's center
    // as the initial position
    var pan = ol.animation.pan({
        source: map.getView().getCenter()
    });
    // Attach to the map
    map.beforeRender(pan);
    // Modify the view to a final position
    map.getView().setCenter([0, 0]);
```

Next, increases 10 degrees the view's rotation angle, animating it with a bounce effect:

```
    // Define an animation function setting the current view's rotation
    // as the initial angle value
    var pan = ol.animation.rotate({
        rotation: map.getView().getRotation(),
        easing: ol.easing.bounce
    });
    // Attach to the map
    map.beforeRender(pan);
    // Modify the view's rotation angle to a final value
    map.getView().setRotation(map.getView().getRotation() + 10 * Math.PI / 180);
```

Take into account, the animations are available only once within the rendering process. Each time we add an animation to the map using the `beforerender` method they are queued, in a list of *preRender* functions, ready to be *consumed* in the next map rendering action. So, when we change any property of the view, it triggers the rendering processes that *consumes* all the *preRender* queued functions and remove them from the queue.

1.4 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[8] repository.

1.4.1 A basic map

1.4.1.1 Goal

Introduce the `ol.Map` and `ol.View` classes and see how to create a basic map.

1.4.1.2 How to do it...

The map needs to be rendered on an HTML element, so we are going to use a `<div>` element:

```
<div id="map" class="map"></div>
```

The HTML element uses a CSS class we use to define its size. Within the `<head>` section of the HTML document add an `<style>` with the next code:

```
.map {
    width: 100%;
    height: 500px;
    box-shadow: 5px 5px 5px #888;
}
```

It makes the map 100% width and 500 pixels height and adds a nice shadow effect to emulate the map is floating over the page.

Now we can add the JavaScript code responsible to create the map. At the end of the document body add a `<script>` element with:

[8] https://github.com/acanimal

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

1.4.1.3 How it works...

A map requires a `target`, a HTML element where to render it, so we have specified the identifier of the `<div>` element previously created:

```
    target: 'map'
```

Remember the `target` can be an string with the element identifier or a reference to the element, for example, using `document.getElementById('map')`.

There has no sense a map without at least a layer. The layers of the map can be specified at initialization time passing an array of layers to the `layers` property. In the sample, we have created a layer that loads tiles from OpenStreetMap[9] project:

```
    layers: [
        // Add a new Tile layer getting tiles from OpenStreetMap source
        new ol.layer.Tile({
            source: new ol.source.OSM()
        })
    ]
```

Finally, the map requires a view which controls how it is rendered (defines the center location, the zoom level, etc). In the sample, we have created a 2D view, centered near of Barcelona city (Spain) and using a zoom level equal to 6:

[9] http://www.openstreetmap.org

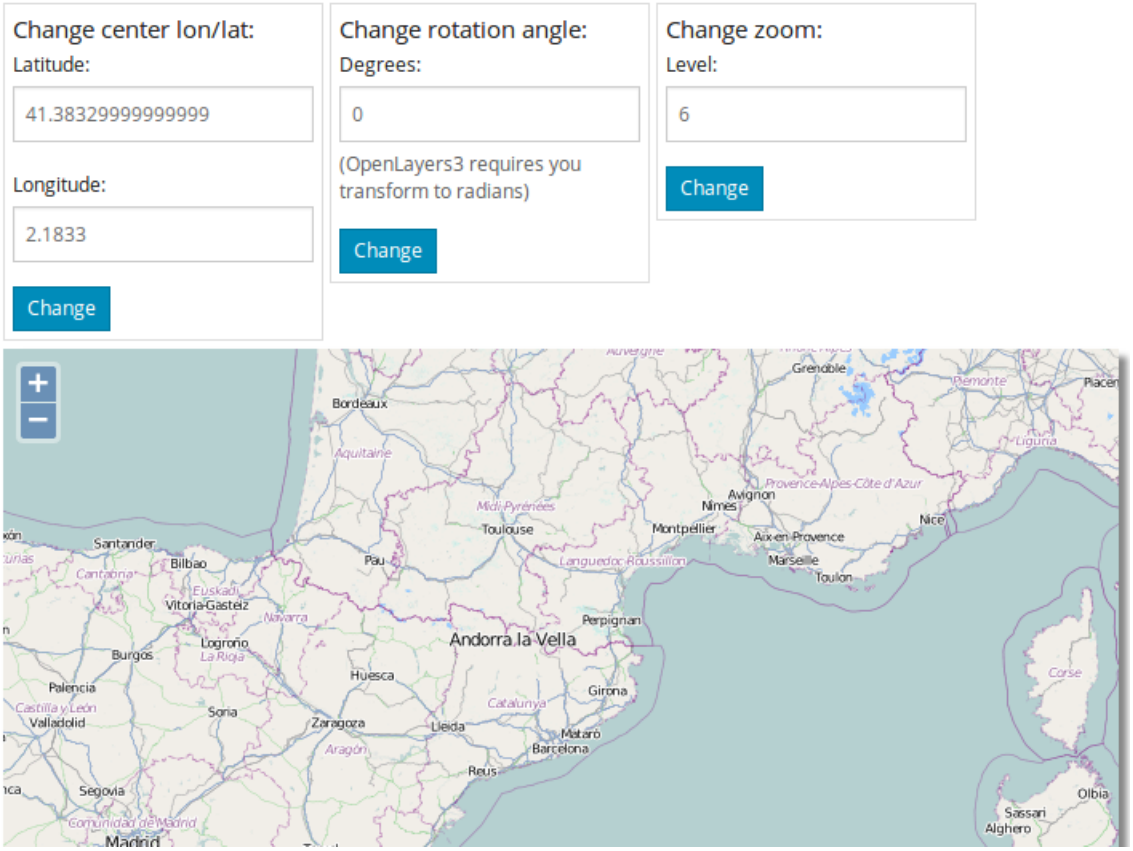
```
    view: new ol.View({
        center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        zoom: 6
    })
```

The `ol.View` has a `projection` property that determines the coordinate system of the view's `center`. By default this projection is `EPSG:3857` and because of this, if we specify the center location in a different projection we need to transform it. In the sample we have set the center to `[2.1833, 41.3833]`, which is `EPSG:4326`, so we transform it using the `ol.proj.transform` method.

1.4.2 Moving around

1.4.2.1 Goal

Demonstrate how we can change the view properties programmatically. We are going to create three forms to modify the center location, the rotation angle and the zoom level.



Moving around

1.4.2.2 How to do it...

First we need to add the HTML code for the forms and the HTML to hold the map:

```
    <div class="example">
        Change center lon/lat:
        <form role="form">
            <div class="form-group">
                <label for="lat">Latitude:</label>
                <input type="text" class="form-control" id="lat" placeholder="lo\
ngitude" value="0.0">
            </div>
            <div class="form-group">
                <label for="lon">Longitude:</label>
                <input type="text" class="form-control" id="lon" placeholder="la\
titude" value="0.0">
            </div>

            <button type="button" class="btn btn-primary btn-xs" id="changeCente\
r">Change</button>
        </form>
    </div>

    <div class="example">
        Change rotation angle:
        <form role="form">
            <div class="form-group">
                <label for="angle">Degrees:</label>
                <input type="text" class="form-control" id="angle" placeholder="\
rotation angle" value="5.0">
                <span class="help-block">(OpenLayers3 requires you transform to \
radians)</span>
            </div>

            <button type="button" class="btn btn-primary btn-xs" id="changeRotat\
ion">Change</button>
        </form>
    </div>

    <div class="example">
        Change zoom:
        <form role="form">
            <div class="form-group">
```

```
                <label for="level">Level:</label>
                <input type="text" class="form-control" id="level" placeholder="\
zoom level" value="7">
            </div>

            <button type="button" class="btn btn-primary btn-xs" id="changeZoom"\
>Change</button>
        </form>
    </div>

    <div id="map" class="map"></div>
```

We use some CSS classes to beautify the map and define the forms width. Within the `<head>` section of the HTML document add an `<style>` with the next code:

```
    .map {
        width: 100%;
        height: 500px;
        box-shadow: 5px 5px 5px #888;
    }
    .example {
        width: 200px;
        border: 1px solid #ddd;
        padding: 5px;
        display: inline-block;
        vertical-align: top;
    }
```

Finally add the JavaScript to initialize the map and modify view properties when buttons are clicked. At the end of the document body add a `<script>` element with:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
```

```
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });

    $(document).ready(function() {

        var center = ol.proj.transform(map.getView().getCenter(), 'EPSG:3857', '\
EPSG:4326');

        $('#lon').val(center[0]);
        $('#lat').val(center[1]);
        $('#angle').val(map.getView().getRotation());
        $('#level').val(map.getView().getZoom());

        $('#changeCenter').on('click', function() {
            var center = [parseInt($('#lon').val()), parseInt($('#lat').val())];
            map.getView().setCenter(ol.proj.transform(center, 'EPSG:4326', 'EPSG\
:3857'));
        });

        $('#changeRotation').on('click', function() {
            map.getView().setRotation($('#angle').val() * Math.PI / 180);
        });

        $('#changeZoom').on('click', function() {
            map.getView().setZoom($('#level').val());
        });
    });
```

We are using jQuery library, so the code responsible to handle buttons events is enclosed within the `$(document).ready()` to ensure it is executed once the document is full loaded.

1.4.2.3 How it works...

The map is initialized in the same way as in the A basic map example.

Once the document is fully loaded we initialize the form input texts with the current view properties values (the current center point, rotation and zoom level):

```
    var center = ol.proj.transform(map.getView().getCenter(), 'EPSG:3857', 'EPSG\
:4326');

    $('#lon').val(center[0]);
    $('#lat').val(center[1]);
    $('#angle').val(map.getView().getRotation());
    $('#level').val(map.getView().getZoom());
```

Thanks to jQuery, we have registered a listener function that is executed when the button is clicked. Next is the registration for the button responsible to modify the center location:

```
    $('#changeCenter').on('click', function() {
        // Actions here
    });
```

Because the view uses, by default, the `EPSG:3857` projection and we are specifying the center location using `EPSG:4326` values, when the user changes the center, the listener function gets the new longitude and latitude values and transforms them using the `ol.proj.transform()` method:

```
    var center = [parseInt($('#lon').val()), parseInt($('#lat').val())];
    map.getView().setCenter(ol.proj.transform(center, 'EPSG:4326', 'EPSG:3857'));
```

Note how the array with the new position is specified with `[lon, lat]` values (and not `[lat, lon]`).

When user changes the rotation angle, the listener function must get the new value specified in degrees and translate to radians:

```
map.getView().setRotation($('#angle').val() * Math.PI / 180);
```

Finally, the listener responsible to change the zoom level simply requires to get the new value and apply it:

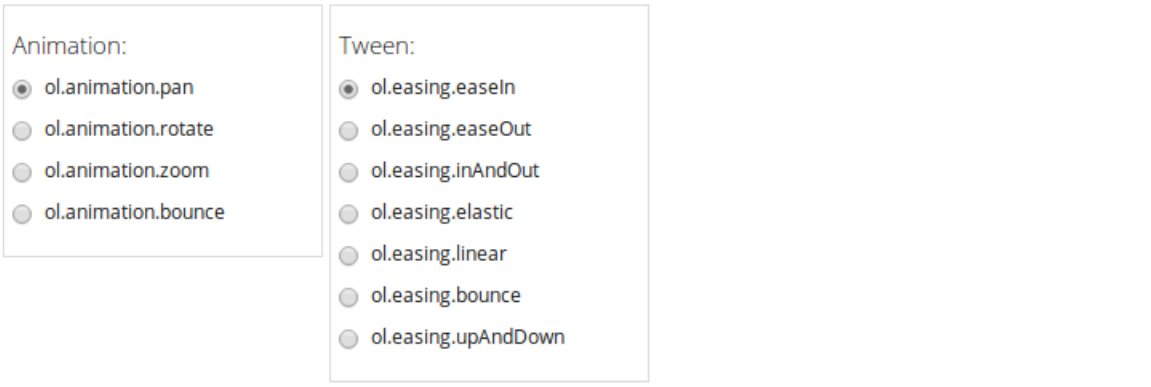
```
map.getView().setZoom($('#level').val());
```

1.4.3 Animating the view

1.4.3.1 Goal

Explain how to create animations to interact with the view.

We will create two different radio buttons list where to select the different combinations between animation and tween functions. Each time the user clicks the map the animation will be executed.



Moving around

1.4.3.2 How to do it...

First we need to add the HTML code for the radio buttons and the map element:

```
<div class="example" id="animation">
    <h5>Animation:</h5>

    <div class="radio">
        <label>
            <input type="radio" name="animationGroup" value="pan" checked>
            ol.animation.pan
        </label>
    </div>
    <div class="radio">
        <label>
            <input type="radio" name="animationGroup" value="rotate">
            ol.animation.rotate
        </label>
    </div>
```

```
16          <div class="radio">
17              <label>
18                  <input type="radio" name="animationGroup" value="zoom">
19                  ol.animation.zoom
20              </label>
21          </div>
22          <div class="radio">
23              <label>
24                  <input type="radio" name="animationGroup" value="bounce">
25                  ol.animation.bounce
26              </label>
27          </div>
28      </div>
29
30      <div class="example" id="tween">
31          <h5>Tween:</h5>
32
33          <div class="radio">
34              <label>
35                  <input type="radio" name="tweenGroup" value="easeIn" checked>
36                  ol.easing.easeIn
37              </label>
38          </div>
39          <div class="radio">
40              <label>
41                  <input type="radio" name="tweenGroup" value="easeOut">
42                  ol.easing.easeOut
43              </label>
44          </div>
45          <div class="radio">
46              <label>
47                  <input type="radio" name="tweenGroup" value="inAndOut">
48                  ol.easing.inAndOut
49              </label>
50          </div>
51          <div class="radio">
52              <label>
53                  <input type="radio" name="tweenGroup" value="elastic">
54                  ol.easing.elastic
55              </label>
56          </div>
57          <div class="radio">
```

```
            <label>
                <input type="radio" name="tweenGroup" value="linear">
                ol.easing.linear
            </label>
        </div>
        <div class="radio">
            <label>
                <input type="radio" name="tweenGroup" value="bounce">
                ol.easing.bounce
            </label>
        </div>
        <div class="radio">
            <label>
                <input type="radio" name="tweenGroup" value="upAndDown">
                ol.easing.upAndDown
            </label>
        </div>
    </div>

    <div id="map" class="map"></div>
```

Add some CSS classes and styles to beautify the map and the controls:

```
    .map {
        height: 500px;
        margin: 5px auto;
        box-shadow: 5px 5px 5px #888;
    }
    .example {
        width: 200px;
        border: 1px solid #ddd;
        padding: 5px;
        display: inline-block;
        vertical-align: top;
        font-size: 0.8em;
    }
```

Finally add the JavaScript code responsible to create the view animations:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });

    /**
     * Creates the appropriate animation given the specified animation
     * and tween functions.
     */
    function createAnimation(animationFunction, tweenFunction) {
        var params = {
            easing: eval(tweenFunction)
        };

        if (animationFunction === ol.animation.pan) {
            params.source = map.getView().getCenter();
        } else if (animationFunction === ol.animation.rotate) {
            params.rotation = map.getView().getRotation();
        } else if (animationFunction === ol.animation.bounce) {
            params.resolution = map.getView().getResolution() * 2;
        } else {
            params.resolution = map.getView().getResolution();
        }

        return animationFunction(params);
    }

    /**
     * Register a listener for a singleclick event on the map.
     */
```

```
43      map.on('singleclick', function(event) {
44          var animationFunction = ol.animation[ $("#animation input:checked").val(\
45  ) ];
46          var tweenFunction = ol.easing[ $("#tween input:checked").val() ];
47
48          var animation = createAnimation(animationFunction, tweenFunction);
49
50          // Add animation to the render pipeline
51          map.beforeRender(animation);
52
53          // Modify the view
54          if (animationFunction === ol.animation.pan) {
55              // Change center location
56              map.getView().setCenter(event.getCoordinate());
57          } else if (animationFunction === ol.animation.rotate) {
58              // Increase rotation angle 10 degrees
59              map.getView().setRotation(map.getView().getRotation() + 10 * Math.PI\
60   / 180);
61          } else if (animationFunction === ol.animation.bounce) {
62              map.getView().setCenter(map.getView().getCenter());
63          } else {
64              // Change zooom
65              map.getView().setResolution(map.getView().getResolution() / 2);
66          }
67      });
```

1.4.3.3 How it works...

As we saw at Animations section from The Map and the View chapter, *animations* are functions specially designed to modify a view property. In addition, each animation function will change the property value using a *tween* function, which determines how a value changes on time. This way we can use different tween functions for the same animation producing different effects.

Because we want to start the animation when the user clicks on the map we have registered a listener function for the `singleclick` event:

```
1       map.on('singleclick', function(event) {
2           ...
3       });
```

The listener function is responsible to:

1. Get the selected animation and tween functions,

2. Create the animation,
3. Apply to the map,
4. Modify the view to the animation takes effect.

Animations are defined as properties of the `ol.animation` object (for example `ol.animation.pan`), while the tween functions are defined at `ol.easing` (for example `ol.easing.easeOut`).

In JavaScript we can get an object property using `object.propertyName` or `object['propertyName']`. We are going to use the second form, because of this we first need to get the animation function name and get a reference to the animation function (and the same for tween function):

```
    var animationFunction = ol.animation[ $("#animation input:checked").val() ];
    var tweenFunction = ol.easing[ $("#tween input:checked").val() ];
```

Once we have a reference to the functions we want to use we create the animation with the `createAnimation` function:

```
    var animation = createAnimation(animationFunction, tweenFunction);
```

All the animation functions accepts `easing` property with a reference to the tween property to be use (and also other common properties like the `duration`).

The issue is, because each animation function is implemented to modify a specific view property, we need appropriate property. As we can see in the code, of the animation function is the `ol.animation.pan` we specify the `source` property pointing the initial center location where to start the pan animation.

When the animation function is the `ol.animation.rotate` we specify the `rotation` property with the current rotation angle value and, finally if the animation function is `ol.animation.bounce` or `ol.animation.zoom` we specify the `resolution` property.

```
    function createAnimation(animationFunction, tweenFunction) {
        var params = {
            easing: eval(tweenFunction)
        };

        if (animationFunction === ol.animation.pan) {
            params.source = map.getView().getCenter();
        } else if (animationFunction === ol.animation.rotate) {
            params.rotation = map.getView().getRotation();
        } else if (animationFunction === ol.animation.bounce) {
            params.resolution = map.getView().getResolution() * 2;
        } else {
```

```
13              params.resolution = map.getView().getResolution();
14          }
15
16          return animationFunction(params);
17      }
```

Once we have the animation created we need to apply it to the map:

```
1       map.beforeRender(animation);
```

Remember the animations are *attached* to the beginning of the rendering pipeline to be *consumed* the next time the map is rendered. Once the animation is *consumed* it is removed from the pipeline. See Applying animations section at The Map and the View chapter.

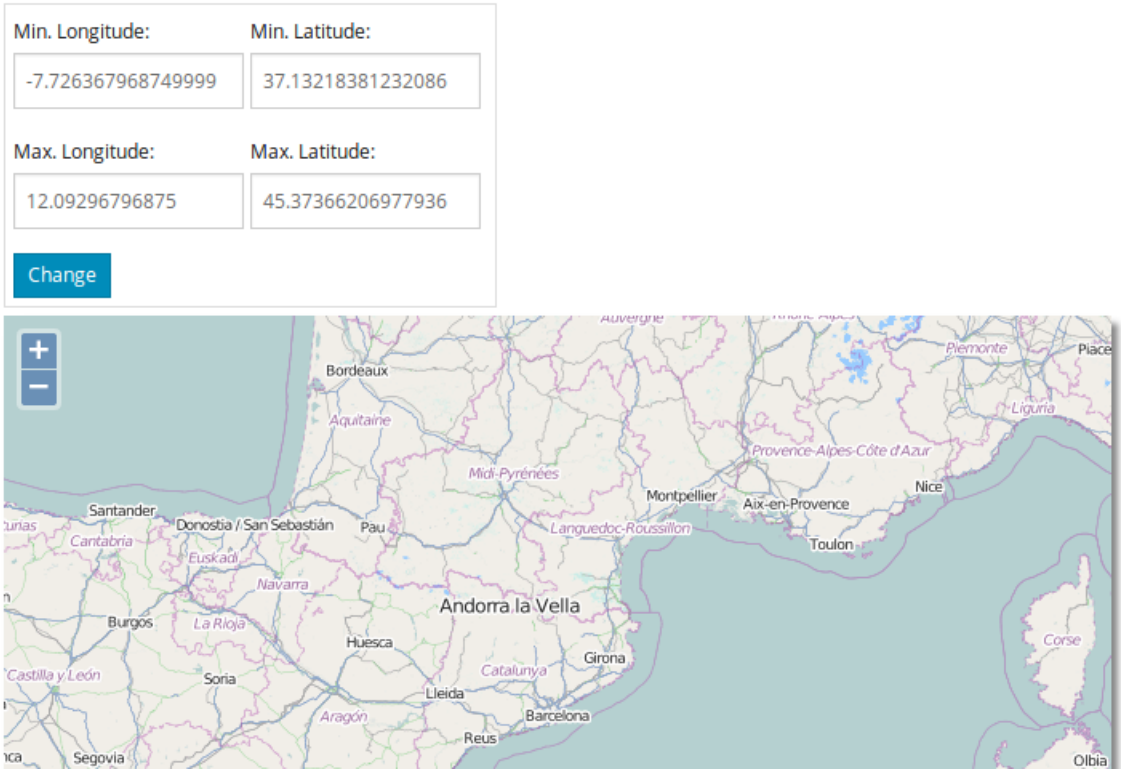
So that the animation takes effect we need to change the view property which animation modifies. Next code checks the selected animation and changes the appropriate property:

```
1       if (animationFunction === ol.animation.pan) {
2           // Change center location
3           map.getView().setCenter(event.getCoordinate());
4       } else if (animationFunction === ol.animation.rotate) {
5           // Increase rotation angle 10 degrees
6           map.getView().setRotation(map.getView().getRotation() + 10 * Math.PI / 1\
7   80);
8       } else if (animationFunction === ol.animation.bounce) {
9           map.getView().setCenter(map.getView().getCenter());
10      } else {
11          // Change zooom
12          map.getView().setResolution(map.getView().getResolution() / 2);
13      }
```

1.4.4 Fit an extent

1.4.4.1 Goal

This example demonstrate how we can use some `ol.View` methods to set the view in a given zoom level and center to fit a given extent.



Fit extent

For this purpose we will create some input controls that will show the current view extent and allows us to modify it.

1.4.4.2 How to do it...

Start adding the HTML for the form controls:

```
    <form role="form">
        <div class="form-group">
            <label for="lon">Min. Longitude:</label>
            <input type="text" class="form-control" id="minlon" placeholder="lat\
itude" value="0.0">
        </div>
        <div class="form-group">
            <label for="lat">Min. Latitude:</label>
            <input type="text" class="form-control" id="minlat" placeholder="lon\
gitude" value="0.0">
        </div>
        <br/>
        <div class="form-group">
            <label for="lon">Max. Longitude:</label>
```

```
            <input type="text" class="form-control" id="maxlon" placeholder="lat\
itude" value="0.0">
        </div>
        <div class="form-group">
            <label for="lat">Max. Latitude:</label>
            <input type="text" class="form-control" id="maxlat" placeholder="lon\
gitude" value="0.0">
        </div>
        <br/>
        <button type="button" class="btn btn-primary btn-xs" id="change">Change<\
/button>
    </form>
```

Add the `<div>` element to hold the map and some CSS code to beautify it:

```
    <div id="map" class="map"></div>
```

Read the A basic map if you want to know how to beautify a bit the map element.

Add JavaScript code to initialize the map:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

Finally add the JavaScript code responsible to get and set the view extent:

```
    $(document).ready(function() {

        // Compute the current extent of the view given the map size
        var extent = map.getView().calculateExtent(map.getSize());

        // Transform the extent from EPSG:3857 to EPSG:4326
        extent = ol.extent.applyTransform(extent, ol.proj.getTransform("EPSG:385\
7", "EPSG:4326"));

        $('#minlon').val(extent[0]);
        $('#minlat').val(extent[1]);
        $('#maxlon').val(extent[2]);
        $('#maxlat').val(extent[3]);

        $('#change').on('click', function() {

            var minlon = parseInt($('#minlon').val());
            var minlat = parseInt($('#minlat').val());
            var maxlon = parseInt($('#maxlon').val());
            var maxlat = parseInt($('#maxlat').val());

            // Trasnform extent to EPSG:3857
            var extent = [minlon, minlat, maxlon, maxlat];
            extent = ol.extent.applyTransform(extent, ol.proj.getTransform("EPSG\
:4326", "EPSG:3857"));

            map.getView().fitExtent(extent, map.getSize());
        });
    });
```

1.4.4.3 How it works...

The first action we do when the page is loaded is to get the current view extent and initialize the form inputs. To do so, we use the `calculateExtent` method that requires to pass the map size in pixels.

Remember, because a view can be used on different maps it has no reference to a specific map, so we need to pass the map size. The view can not get it automatically.

```
    var extent = map.getView().calculateExtent(map.getSize());
```

Now, we have an array with the extent the view is currently shown in the form of `[min-longitude, min-latitude, max-longitude, min-latitude]`.

There is no class to represent the extent, neither a latitude-longitude pair location. In OpenLayers3 any four element array can us used as an extent and any two element array as a latitude-longitude location.

The map is using the default view projection, that is the `EPSG:3857`, but we want to show values as `EPSG:4326` so we need to transform it.

The `ol.extent` object has some functions designed to work with extents. Among them we can find the `ol.extent.applyTransform` that accepts two parameters: the extent to work on and a transform function. Anyone can create a transform function that operates on a extent but OpenLayers3 offers us the most common ones.

In a similar way, `ol.proj` object contains functions to work with projections. Given two projection codes, the `ol.proj.getTransform` returns a function that knows how to transform between the specified projections.

So, the code to transform the view extent from `EPSG:3857` to `EPSG:4326` is:

```
1     extent = ol.extent.applyTransform(extent, ol.proj.getTransform("EPSG:3857", \
2 "EPSG:4326"));
```

Once we have the extent in our preferred projection we set the values in the inputs:

```
1     $('#minlon').val(extent[0]);
2     $('#minlat').val(extent[1]);
3     $('#maxlon').val(extent[2]);
4     $('#maxlat').val(extent[3]);
```

Now the user can change the input values and set a different extent for the view. When the user clicks the *change* button we must obtain the input values:

```
    $('#change').on('click', function() {

        var minlon = parseInt($('#minlon').val());
        var minlat = parseInt($('#minlat').val());
        var maxlon = parseInt($('#maxlon').val());
        var maxlat = parseInt($('#maxlat').val());

        // Continue code here
        ...
    });
```

create an extent and transform it from `EPSG:4326` to `EPSG:3857` projection:

```
        // Trasnform extent to EPSG:3857
        var extent = [minlon, minlat, maxlon, maxlat];
        extent = ol.extent.applyTransform(extent, ol.proj.getTransform("EPSG:432\
6", "EPSG:3857"));
```

and apply to the view using the `fitExtent` method that, similarly than `calculateExtent`, requires the map size in pixels:

```
    map.getView().fitExtent(extent, map.getSize());
```

2. Layers

The concept of layer is a way to allow classify or group information. We usually create layers with terrain imagery, containing the cities of the World, the rivers of a country, etc.

Hopefully, the layer hierarchy has been simplified in OpenLayers3 (compared to previous versions) and has been improved, among other things, with the addition of *layer groups* that allows to apply modification to a collection of layers.

Usually, a layer requires to load content from a remote source. It can be raster or vector data: imagery from a WMS server, vector data in GeoJSON format, ...

The OpenLayers3 design differentiate between the concept of *layer*, which represents a set of information, with the concept of *source*, which is the responsible to obtain the data to be used by the layer.

This chapter is focused on explaining the layer hierarchy, the kind of layers OpenLayers3 offers us and how to manage them. For an in depth explanation of *source* implementations you need to refer to the Data sources and formats.

2.1 Managing the layers on the map

We can understand the map as a container where we can *stack* layers, each one grouping a set of information.

I have emphasized the concept of *stack* because this is the real behavior the map brings to the layers. Layers at highest position will overlap layers at lower position or, said in a different way, when the map is rendered the layers are rendered one by one from bottom to top of the stack.

Each time we add a layer to the map it is added to the top of the stack.

As we have seen, we can set the layers of the map at initialization time, by using the `layers` property that accepts an array of layers:

```
var map = new ol.Map({
    ...
    layers: [], // Instantiating a map without layers
    ...
});
```

To add and remove layers the `ol.Map` class offers the methods `addLayer` and `removeLayer`.

```
var layerA = ... // A layer instance
var layerB = ... // Another layer instance

// Add layers by hand
map.addLayer(layerA);
map.addLayer(layerB);

// Remove a layer from the map
map.removeLayer(layerB);
```

In the code, the `layerA` is added to the map becoming the new layer on the top of the stack. Later, the `layerB` is added, overlapping the previous layers. Finally, we remove the `layerB` that disappear from the map and leaves again the `layerA` on the top.

2.1.1 Controlling the layer stack

OpenLayers3 comes with the helper class `ol.Collection` that is who really implements the concept of stack. The `ol.Collection` not only allows to push and pop elements but also provide methods to iterate over the elements, insert, get or remove at a specified position or retrieve the collection length.

The `ol.Map` class uses internally an `ol.Collection` to store all the map layers and we can get a reference to it with the `getLayers` method.

```
var layers = map.getLayers();
```

> Really, the map stores all the map layers in a *layer group*, implemented by the `ol.layer.Group`, which internally makes use of `ol.Collection`. So, the method `getLayers` really returns a reference to the collection store in the *layer group*. See the Layer Groups section. The `ol.Map` offers to similar methods:
>
> - `getLayerGroup`, which returns a reference to the `ol.layer.Group` and

- `getLayers`, which returns a reference to the `ol.Collection` pointed by the previous `ol.layer.Group` instance.

The methods we can found in the `ol.Collection` class are:

- `clear`, empties the collection
- `extend`, adds to the collection a set of elements specified in an array
- `forEach`, iterates over the elements of the collection calling a given function
- `getArray`, returns the elements of the collection as an array
- `item`, get the element at a specified position
- `getLength`, get the length of the collection
- `insertAt`, insert an element at a specified position (displacing any existing one)
- `pop`, retrieves an element from the end of the collection
- `push`, adds an element to the end of the collection
- `remove`, removes the first occurrence of the element
- `removeAt`, removes the element at a specified position (displacing the rest)
- `setAt`, set the element at the given position

Remember, for all the methods that requires a position the index starts at zero.

The `pop` and `push` methods allows to work with a collection as if it were a stack. When we call the `addLayer` method in the `ol.Map` class, what it is really doing is delegating the action to the `push` method of the `ol.Collection`.

Anyway the preferred method to add or remove a layer from the map is using the `ol.Map` methods `addLayer` and `removeLayer`.

So, given a map instance and a set of layers:

```
    // The map instance
    var map = new ol.Map({
        ...
        layers: [], // Instantiating a map without layers
        ...
    });

    // Create some layer instances
    var layerA = ...;
    var layerB = ...;
    var layerC = ...;
    var layerD = ...;
```

we can get a reference to the layers collection and get its size:

```
    var layers = map.getLayers();   // Get layers collection reference
    var count = layers.getLength(); // Initially the collection is empty
```

add a layer using the preferred method `addLayer`:

```
    map.addLayer(layerA);           // Add layers (the preferred way)
    count = layers.getLength();     // Count must be 1
```

or we can also add a layer using collection's `push` method:

```
    layers.push(layerB);
    count = layers.getLength();     // Count must be 2
```

change a layer by another with `setAt` or insert a new one at some specific position with `insertAt`:

```
    map.setAt(0, layerC);       // Change layerA by layerC
    map.insertAt(0, layerD);    // Add layerD at the bottom
    count = layers.getLength(); // Count must be 3
```

we can remove an element with `removeAt` or `remove`:

```
    layers.removeAt(1);         // Removes the layerC
    layers.remove(layerB);      // Removes the layerB
    count = layers.getLength(); // Count must be 1
```

or add a set of layers using the `extend` method:

```
1     layers.extend([layerA, layerB, layerC]);
2     count = layers.getLength();     // Count must be 4
```

2.2 The base class

In OpenLayers3, all the layers inherits from the base class `ol.layer.Base`, which offers the common properties and functionalities that can later be extended by each concrete subclass.

The list of these common properties are:

- `visible`, a boolean value specifying it the layer must be rendered or not,
- `opacity` , a value between 0.0 and 1.0 that determines the opacity of the layer,
- `maxResolution`, the maximum resolution at which the layer must be rendered,
- `minResolution`, the minimum resolution at which the layer must be rendered.

In addition to these properties, the `ol.layer.Base` class defines `brightness`, `contrast`, `hue` and `saturation` that brings the possibility to make some image editing at browser side.

This capability is not present when rendering with DOM mechanism, so in this mode changes on this properties will not take effect.

All these properties can be passed at instantiation time in addition to be used through their setter and getter methods.

2.2.1 Additional properties: Where is the layer name?

Once read the above section and, more probably if you have worked with previous versions of OpenLayers, you have been notices the lack of `name` property. Why?

We have said OpenLayers3 is much more flexible and powerful than its previous versions and, the way to manage properties is not an exception.

In OpenLayers3 all the main objects (like the map, view, layers and many more) inherits from the class `ol.Object`. This class allows, among other things, to attach any number of properties to an instance, like a *key-value* store. This way the user is not restricted to use some number of predefined properties but has the flexibility to attach any property interesting for his/her purposes, like the use of the `id` or `name` properties on each layer.

More information about object properties in the Working with object properties section at Events, listeners and properties chapter.

The methods that allows to work with the properties are:

- `get`, given a key returns its associated value,
- `getProperties`, returns and object with all the *key-value* of the object,
- `set`, stores a new value for a given key,
- `setValues`, allows to set any number of *key-value* at the same time.

This way, we can create a layer instance and associate a `name` or an `id`:

```
1    var layerOSM = new ol.layer.Tile({
2        source: new ol.source.OSM()
3    });
4
5    layerOSM.set('id', Math.random());
6    layerOSM.set('name', 'OpenStreetMap');
```

and later retrieve them:

```
1    var id = layerOSM.get('id');
2    var name = layerOSM.get('name');
```

Note, we can also set additional properties when we creating the object:

```
1    var layerOSM = new ol.layer.Tile({
2        id: Math.random(),
3        name: 'OpenStreetMap',
4        source: new ol.source.OSM()
5    });
```

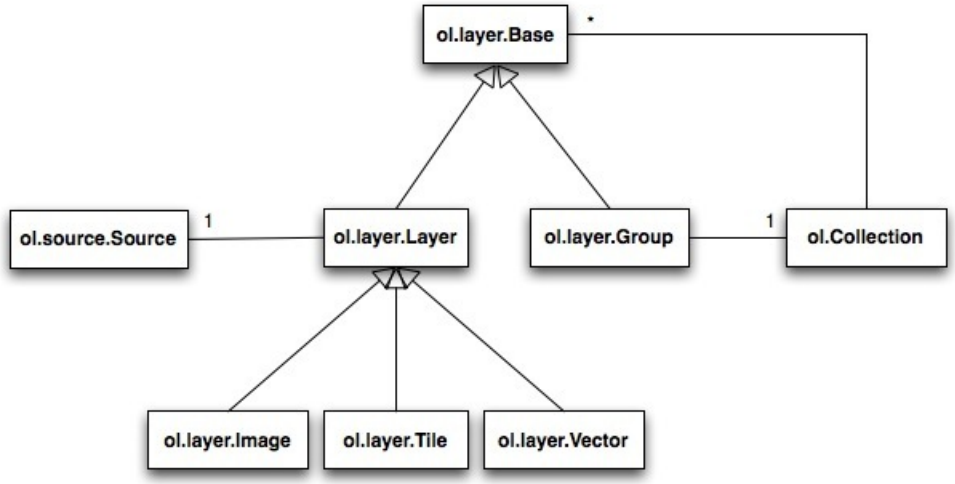
2.3 The layer hierarchy

The layer hierarchy defined in OpenLayers3 has been simplified and reduced from previous versions. All the layers inherits from the `ol.layer.Base` class. From it, two main child classes has been defined:

- `ol.layer.Layer`, which acts as a base class for all those implementations that requires to load data from a *data source*,
- `ol.layer.Group`, which is a helper class that allows to create layers that contains other layers (see the composite pattern[1]).

[1] http://en.wikipedia.org/wiki/Composite_pattern

Next figure shows the hierarchy of these classes:



Layer hierarchy

Aside the `ol.layer.Group`, the layers requires to load content from a *data source*. Because of this, the `ol.layer.Layer` offers the `source` property, which is a reference to an `ol.source.Source` instance, so each time the layer needs to load data it delegates the task to the `source` object.

In previous versions of OpenLayers there were layers for almost any kind of source: a layer to read from OpenStreetMap tile, to read from a WMS server, etc. In OpenLayers3, thanks to the separation of the concepts *layer* and *source* we can create, for example, an instance of a tiled layer loading data from OpenStreetMap and another instance loading data from a WMS server. You can find more information about sources at Data sources and formats chapter.

2.3.1 Layer Groups

One of the improvements in OpenLayers3 version has been the addition of the *layer group*, a kind of layer that can contain other layers, acting as a container or a folder.

The great of this kind of layer is that any changes applied to it *affects* all its contained layers. I highlight *affects* because the modification of a property in the layer group does not modifies the same property in the contained layers. Let's explain with an example.

Supposing the next situation:

```
var myLayer = ...;
myLayer.setOpacity(0.7);

var group = var layerStm = new ol.layer.Group({
    layers: [myLayer]
});

group.setOpacity(0.5);
```

changing the `group` opacity does not modifies the `myLayer` opacity. What OpenLayers3 does at rendering time is first reduces the `myLayer` opacity value to `0.7` and later reduces the `group` group opacity to `0.5`. The result is the `myLayer` instance is rendered with an opacity value of `0.35`, that is, `0.7 x '0.5`.

In addition, the way how modification affects the contained layers varies depending on the property. We can see the `opacity` property for the group and the contained layers is multiplied. For the `visible` property the natural operation is a logical `AND` between the layer group and the contained layer, rendering only when both are `true`, while for the `hue` property the operation is a addition.

2.3.1.1 Working with layer groups

The `ol.layer.Group` class stores all the references to the contained layers within the `layers` property, which is an instance of `ol.Collection`.

We have seen the `ol.Collection` class in the Controlling the layer stack section.

When creating an `ol.layer.Group` instance we can optionally specify the set of layers to group passing the `layers` property with an array of layers references:

```
var layerA = ...;
var layerB = ...;

var group = var layerStm = new ol.layer.Group({
    layers: [layerA, layerB]
});
```

The `ol.layer.Group` class has two methods `setLayers` and `getLayers` which sets or gets the `layers` property. In the case, we want to set the layers of the group after create the group instance we need to pass an `ol.Collection` reference:

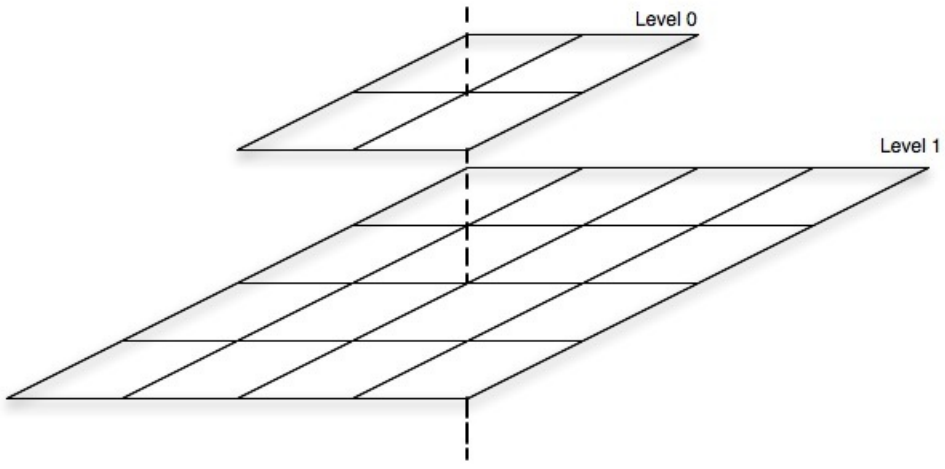
```
var group = new ol.layer.Group();   // An empty layer group

var layerA = ...;
var layerB = ...;

var collection = new ol.Collection([layerA, layerB]);
group.setLayers( collection );
```

Additionally, we can have more control over the layer group working directly with the collection of layers:

```
var layerA = ...;
var layerB = ...;

var group = var layerStm = new ol.layer.Group({
    layers: [layerA]
});

var collection = group.getLayers();
collection.push(layerB);
```

2.3.2 Tiled layers

A tile layer is a kind of layer composed by a grid of images for each zoom level it supports and conforming a pyramid structure. Within this structure the first zoom level is the named zoom level `0` and on each subsequent zoom level each tile is subdivided in four more.



Pyramid of tiles

See Resolutions and zoom levels section in the The Map and the View chapter to get more information about zoom levels and resolutions.

Tiled layers are implemented by the `ol.layer.Tile` class. To create a new instance we simply require to specify the `source` property to be used:

```
1     // Tile layer loading tiles from MapQuest service
2     var mapQuestLayer = new ol.layer.Tile({
3         source: new ol.source.MapQuest({
4             layer: 'sat'
5         })
6     });
7
8     // Tile layer loading tiles from a WMS server
9     var wmsLayer = new ol.layer.Tile({
10        source: new ol.source.TileWMS({
11            url: 'http://demo.opengeo.org/geoserver/wms',
12            params: {'LAYERS': 'topp:states', 'TILED': true},
13            extent: [-13884991, 2870341, -7455066, 6338219]
14        })
15    });
```

As we can see, the key aspect in the previous samples is to know the available sources implementations, which are the responsibles to load data.

2.3.3 Image layers

At the opposite of the tiled layers, where information is represented by a grid of tiles and we can have different zoom level resolutions, image layers are suitable to render data from a single image file.

Image layers are implemented by `ol.layer.Image` class and are used in conjunction with the `ol.source.ImageStatic` source class, which loads data from a specified `url`.

```
1     var imageLayer = new ol.layer.Image({
2         source: new ol.source.ImageStatic({
3             url: 'http://desired_image_url',
4             ...
5         })
6     });
```

We will see in detail `ol.source.ImageStatic` class at the Loading an static image section from Data sources and formats chapter.

With image layers we have no pyramid structure and we don't load different images with different resolution for each zoom level. We only have a fixed image with a fixed resolution. Because of this, each time the user changes the map's zoom level, OpenLayers must change the image dimension to *transform* the image resolution to the current map resolution.

Image layers are appropriate to work with lightweight images. For big images it is better to use a map server to serve tiles computed from the original big file.

There can be cases where we must control the resolutions at which layer must be visible. Imagine an image layer that its limits encompasses a few kilometers and we set the map zoom level to see the whole world. OpenLayers will render the image layer although is practically imperceptible. In this situations, remember to use the `maxResolution` and `minResolution` properties to control the zoom levels at which layer must be visible (and be rendered).

2.3.4 Vector layers

While previous layers covers the need to render raster information, the `ol.layer.Vector` class allow us to work with vector data.

Vector layers are probably the most complex kind of layers. The data must be read and transformed to features with the addition it can be stored in different data sources and formats. The features can be represented using different geometries, can be styled in different ways and depending on many conditions. In addition, vector layers could require create new features, edit or delete existent ones and send changes to the server side to be persisted.

Here we simply introduce the `ol.layer.Vector` class. An in depth description of how to work with vector information is made in the Vector layers chapter.

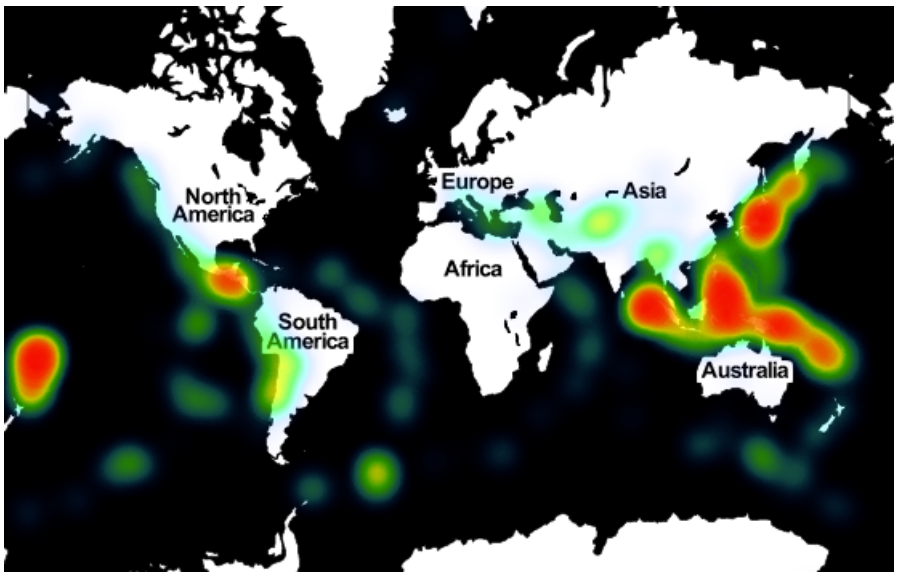
In it most basic form, we can create a new `ol.layer.Vector` instance simply specifying the `source` property responsible to load data. Next sample creates a vector layer which loads content from a GeoJSON file:

```
1    var vector = new ol.layer.Vector({
2        source: new ol.source.GeoJSON({
3            url: ... // GeoJSON file url
4        })
5    });
```

When loading remote files we can have issues with cross domain request. Older browsers follows the Same-origin policy[2] and avoid make HTTP requests[3] between different domains. Newer browser implements CORS[4] machanism that solves this issue waranting the security. More information at Working with the JavaScript XMLHttpRequest object[5].

2.3.4.1 Heatmap layer

A heat map is a graphical representation of data where the individual values are represented as colors. Heat maps are specially suitable to create density maps where, through a very colorful and visual way, we can see which zones are *more important*. As a sample, next image show the earthquakes activity (it is taken from the Earthquakes heatmap[6] OpenLayers3 examples):



Earthquakes activity

Hopefully for us, OpenLayers3 offers a special kind of vector class suitable to render vector information as a heat map, the `ol.layer.Heatmap` class.

As any other vector layer, the class requires we specify valid source instance in its `source` property to retrieve the features to be rendered.

The `ol.layer.Heatmap` extents its parent class offering the next set of properties that allows to control the rendering aspects of the heat map:

[2] http://en.wikipedia.org/wiki/Same-origin_policy
[3] http://en.wikipedia.org/wiki/XMLHttpRequest
[4] http://en.wikipedia.org/wiki/Cross-Origin_Resource_Sharing
[5] http://acuriousanimal.com/blog/2011/01/27/working-with-the-javascript-xmlhttprequest-object/
[6] http://ol3js.org/en/master/examples/heatmap-earthquakes.html

- `gradient`: an array with the set of colors in hexadecimal format that will determine the heat map. From these values a linear gradient ramp is created. Default gradient value is `['#00f', '#0ff', '#0f0', '#ff0', '#f00']`.
- `weight`: a string, with the name of the feature's property to be used as weight value, or a function that determines the weight of each feature. The values must be in the range `0..1`. By default, the code looks for a `weight` property on each feature to be used as the weight value.
- `radius`: the radius of the circles, by default 8.
- `blur`: the blur effect size in pixels to apply, by default 15.
- `shadow`: the shadow effect size in pixels to apply on each circle, by default 250.

To apply in the right way the previous properties it is important to understand the way `ol.layer.Heatmap` works, so we are going to try to summarize it in a couple of sentences.

For each feature the layer applies (or computes) a *weight* and, in addition, given the gradient of colors a linear gradient of 256x1 pixel is created.

At rendering time, each feature is rendered as a black blurred circle with the specified radius, which produces a gray scaled circle with values from 0 to 255. Final step is to obtain an image (a matrix of pixels) from the canvas element and substitute each gray scaled pixel with the corresponding color within the previous computed gradient.

Finally, as example, the next code shows how we can create a heat map from a GeoJSON source file (and supposing each feature has a weight property):

```
1     var vector = new ol.layer.Heatmap({
2       source: new ol.source.GeoJSON({
3         url: 'http://some_server/geojson_file'
4       }),
5       radius: 5
6     });
```

2.4 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[7] repository.

2.4.1 Adding and removing layers

2.4.1.1 Goal

This sample shows how we can add and remove layers to the map. We will create some controls to add and remove raster layers using tiles from MapQuest and OpenStreetMap projects. The examples also helps to understand the layers behavior within the layer stack and how they can overlap each other.

Adding and removing layers

2.4.1.2 How to do it...

Let's start adding the HTML for the form controls and the map. Independently the layout we choose we need to define four buttons, two to add/remove the OpenStreepMap layer and two more to add/remove the MapQuest layers:

[7] https://github.com/acanimal

```
    <div class="row">
        <div class="col-md-2 example">
            OpenStreetMap<br/>
            <button id="addOSM" class="btn btn-primary btn-xs">Add</button>
            <button id="removeOSM" class="btn btn-danger btn-xs">Remove</button>
        </div>
        <div class="col-md-2 example">
            MapQuest<br/>
            <button id="addMQ" class="btn btn-primary btn-xs">Add</button>
            <button id="removeMQ" class="btn btn-danger btn-xs">Remove</button>
        </div>
    </div>
    <div id="map" class="map"></div>
```

Now add the code to initially the map. Note it is an empty map (without layers):

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [],     // Initially empty map
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

Add the code that creates the two layer instances:

```
    var layerOSM = new ol.layer.Tile({
        source: new ol.source.OSM()
    });
    var layerMQ = new ol.layer.Tile({
        source: new ol.source.MapQuest({
            layer: 'osm'
        })
    });
```

Finally, add the code that reacts when buttons are pressed, responsible to add or remove the layers:

```
    $(document).ready(function() {
        $('#addOSM').on('click', function() {
            map.addLayer(layerOSM);
        });
        $('#removeOSM').on('click', function() {
            map.removeLayer(layerOSM);
        });
        $('#addMQ').on('click', function() {
            map.addLayer(layerMQ);
        });
        $('#removeMQ').on('click', function() {
            map.removeLayer(layerMQ);
        });
    });
```

2.4.1.3 How it works...

We have created a map instance specifying an empty array of layers, this way we obtain en empty map.

Next, we have created two layers instances. Both are instances of `ol.layer.Tile` class but using a different `ol.source` instance.

In the case of MapQuest, the service offers different kind of layers, so the `ol.source` instance accepts a `layer` property where we we specify the layer to be request:

```
    var layerMQ = new ol.layer.Tile({
        source: new ol.source.MapQuest({
            layer: 'osm'
        })
    });
```

Valid values for `layer` property are:

- `osm`, for OpenStreetMap data based map,
- `sat`, for satellite imagery,
- `hyb`, for tiles mixing OpenStreetMap data over satellite imagery.

At this point we have two global variables `layerOSM` and `layerMQ` that references the two raster layers we can add from or remove to the map. In addition, we have registered some jQuery listeners that are executed when buttons are clicked and, using the map `addLayer()` and `removeLayer()` methods adds or removes the layers:

```
...
$('#addOSM').on('click', function() {
    map.addLayer(layerOSM);
});
$('#removeOSM').on('click', function() {
    map.removeLayer(layerOSM);
});
...
```

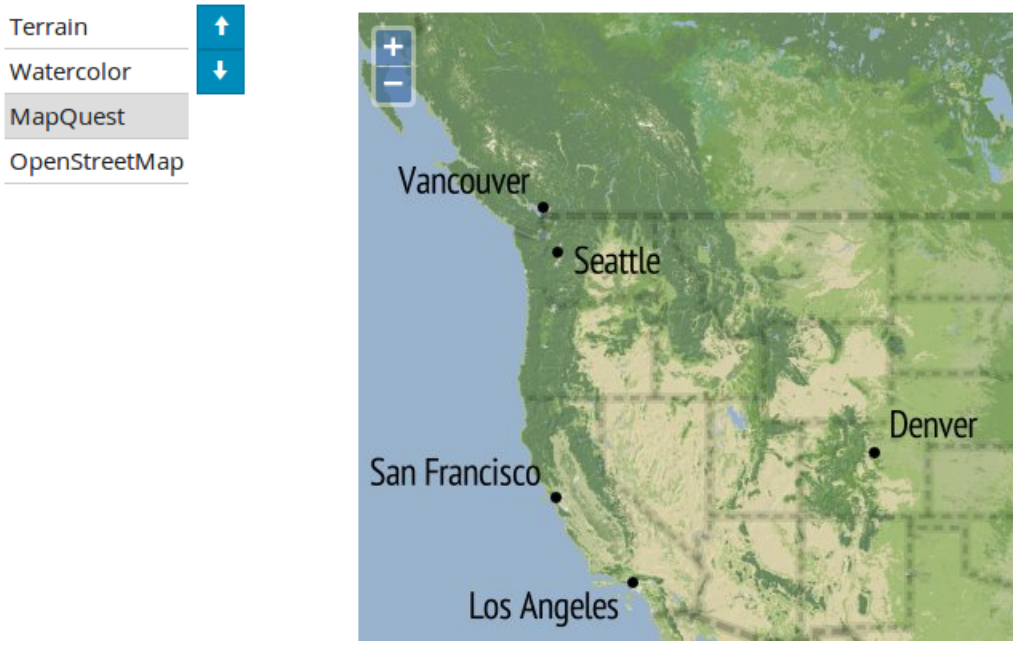
Remember `addLayer()` and `removeLayer()` requires we pass an `ol.layer.Base` reference, so we need to store these references on a variable available to the listener code. To have more degree control over layers you can work directly with the layer stack getting a reference to the layers collection with the map `getLayers()` method.

2.4.2 Raise and lower layers in the layer stack

2.4.2.1 Goal

On this example we are going to demonstrate how we can more degree of control over the layer stack working directly on the layers collection.

We are going to create a map with some layers and a control to raise or lower layers within the stack:



Layer stack

2.4.2.2 How to do it...

Add the HTML code for the application layout. We are going to create two columns. On the left we will place the elements to control the layer stack, that is, an unsigned list `<ul>` element that will contain each layer and two buttons to raise and lower layers:

```
    <div class="col-md-3">
        <ul class="layerstack controls"></ul>
        <div class="controls">
            <button id="raise" class="btn btn-primary btn-xs"><span class="glyph\
icon glyphicon-arrow-up"></span></button><br/>
            <button id="lower" class="btn btn-primary btn-xs"><span class="glyph\
icon glyphicon-arrow-down"></span></button>
        </div>
    </div>
    <div class="col-md-9">
        <div id="map" class="map"></div>
    </div>
```

Add some CSS classes to style the stack control and buttons:

```
    ul.layerstack {
        list-style: none;
    }
    ul.layerstack li {
        border-bottom: 1px solid #ccc;
        padding: 3px;
    }
    ul.layerstack li:hover {
        background-color: #eee;
    }
    ul.layerstack li.selected {
        background-color: #ddd;
    }
    .controls {
        float: left;
        margin-right: 5px;
    }
```

Now add the JavaScript code that creates the four layers and initializes the map:

```
    // Create layers instances
    var layerOSM = new ol.layer.Tile({
        source: new ol.source.OSM(),
        name: 'OpenStreetMap'
    });
    var layerMQ = new ol.layer.Tile({
        source: new ol.source.MapQuest({
            layer: 'osm'
        }),
        name: 'MapQuest'
    });
    var layerStamenWater = new ol.layer.Tile({
        source: new ol.source.Stamen({
            layer: 'watercolor'
        }),
        name: 'Watercolor'
    });
    var layerStamenTerrain = new ol.layer.Tile({
        source: new ol.source.Stamen({
            layer: 'terrain'
        }),
        name: 'Terrain'
    });

    // Create map
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [   // Add the set of layers
            layerOSM, layerMQ, layerStamenWater, layerStamenTerrain
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([-100.1833, 41.3833], 'EPSG:4326', 'EPSG:3\
857'),
            zoom: 4
        })
    });
```

Now, add the JavaScript that will be executed once the layer is loaded. Basically we initialize the stack control with the layers in the map and registers two listener function to raise and lower the selected layer when the buttons are clicked:

```
$(document).ready(function() {

    initializeStack();

    $('#raise').on('click', function() {
        var layerid = $('ul.layerstack li.selected').data('layerid');
        if (layerid) {
            var layer = findByName(layerid);
            raiseLayer(layer);
        }
    });

    $('#lower').on('click', function() {
        var layerid = $('ul.layerstack li.selected').data('layerid');
        if (layerid) {
            var layer = findByName(layerid);
            lowerLayer(layer);
        }
    });
});
```

As you can see in the code there are other methods, like `initializeStack()` `findByName()` or `raiseLayer()` but we will describe them in detail in the next section to avoid enlarge too much this one.

2.4.2.3 How it works...

We have create a set of layer instances using the extra property `name`, which we will use to show in the stack control:

```
var layerOSM = new ol.layer.Tile({
    source: new ol.source.OSM(),
    name: 'OpenStreetMap'
});
```

Once the map is initializes with the set of layers the first action we make is to initialize the stack control to show the current layers of the map. This is done with the `initializeStack()` function:

```
    function initializeStack() {
        var layers = map.getLayers();
        var length = layers.getLength(), l;
        for (var i = 0; i < length; i++) {
            l = layers.item(i);
            $('ul.layerstack').prepend('<li data-layerid="' + l.get('name') + '"\
>' + l.get('name') + '</li>');
        }

        // Change style when select a layer
        $('ul.layerstack li').on('click', function() {
            $('ul.layerstack li').removeClass('selected');
            $(this).addClass('selected');
        });
    }
```

As we can see, the code:

- Iterates over all the layers in the map, using the `map.getLayers()` and `layers.item()` methods.
- Create a `<li>` element with layer name. Like any other `ol.Object` subclass, we can extract a layer's attribute using the `layer.get()` method.
- Finally, register a `click` event over all list elements to control the element style when one is selected.

Note we have use the jQuery `prepend()` method to add the layers in reverse order to the list to emulate a stack order, that is, the layer in the last position is on top of the stack.

Once the stack control is initialized, we register a couple of function listener for the raise and lower buttons. Let's go to explain how to raise layer works:

```
    $('#raise').on('click', function() {
        var layerid = $('ul.layerstack li.selected').data('layerid');
        if (layerid) {
            var layer = findByName(layerid);
            raiseLayer(layer);
        }
    });
```

When the raise button is clicked, we get the selected layer obtaining a reference to the `<li>` element that has set the `selected` CSS class and retrieve the `data-layerid` attribute values, which contains the layer name.

Given the layer name we look for the layer reference with the `findByName()` and raise the layer using the `raiseLayer()` method.

The `findByName()` method is responsible to iterate over all the layers of the map and return a reference to that layer that has a `name` attribute equal to the specified:

```
    function findByName(name) {
        var layers = map.getLayers();
        var length = layers.getLength();
        for (var i = 0; i < length; i++) {
            if (name === layers.item(i).get('name')) {
                return layers.item(i);
            }
        }
        return null;
    }
```

On its way, the `raiseLayer()` search the index position of the specified layer and increases it in the layer collection. Additionally it moves the `<li>` element up in the stack control:

```
    function raiseLayer(layer) {
        var layers = map.getLayers();
        var index = indexOf(layers, layer);
        if (index < layers.getLength() - 1) {
            var next = layers.item(index + 1);
            layers.setAt(index + 1, layer);
            layers.setAt(index, next);

            // Moves li element up
            var elem = $('ul.layerstack li[data-layerid="' + layer.get('name') +\
 '"]');
            elem.prev().before(elem);
        }
    }
```

The index of the layer is obtained with the `indexOf()` function that is implemented as:

```
    function indexOf(layers, layer) {
        var length = layers.getLength();
        for (var i = 0; i < length; i++) {
            if (layer === layers.item(i)) {
                return i;
            }
        }
        return -1;
    }
```

Given an array of layers and a layer reference it iterates over the array and return the index position of the layer if found or -1 otherwise.

Finally, when the lower buttons is pressed the `lowerLayer()` function is executed, which almost identical to the raise one. It interchanges the selected layer with the previous one and lowers the `<li>` element in the stack control:

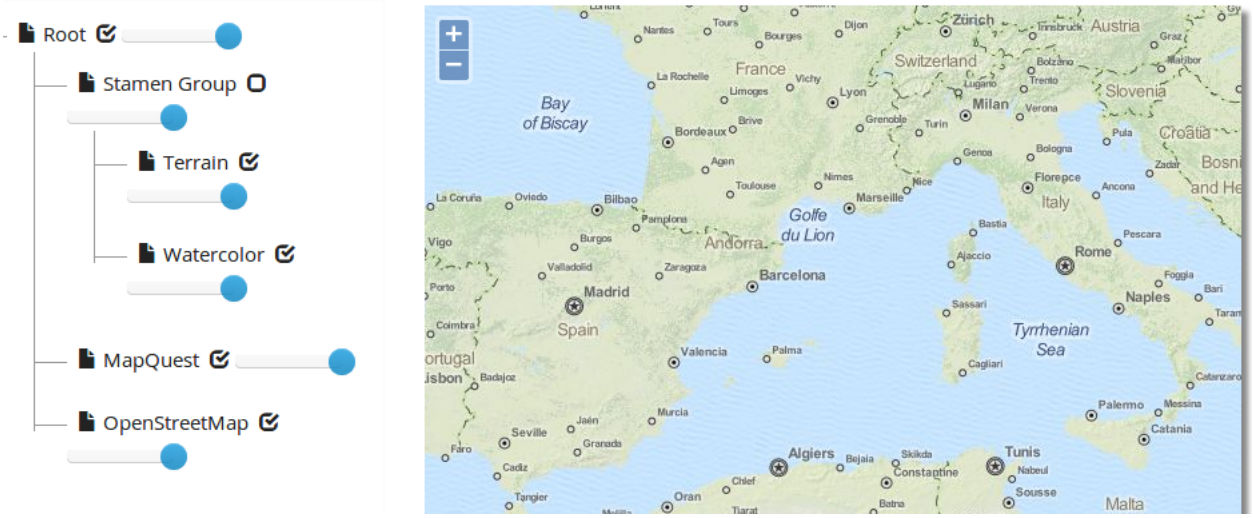
```
    function lowerLayer(layer) {
        var layers = map.getLayers();
        var index = indexOf(layers, layer);
        if (index > 0) {
            var prev = layers.item(index - 1);
            layers.setAt(index - 1, layer);
            layers.setAt(index, prev);

            // Moves li element down
            var elem = $('ul.layerstack li[data-layerid="' + layer.get('name') +\
 '"]');
            elem.next().after(elem);
        }
    }
```

2.4.3 Layer groups

2.4.3.1 Goal

This example shows how we can use the `ol.layer.Group` class to create layer groups creating a tree like structure of layer. In addition we can see in action how changes on the layer group affect all the contained layers.

We are going to create a two columns layout. On the left we will place the tree control that will show the *tree* structure that conforms the map layers and, on the right, we will place the map.



Layer group

2.4.3.2 How to do it...

Start adding the required HTML code for the layout:

```
<div class="col-md-4">
    <div id="layertree" class="tree"></div>
</div>
<div class="col-md-8">
    <div id="map" class="map"></div>
</div>
```

The previous elements has also associated some CSS classes to style them. The code is a bit extensive to be placed in the book and the most important thing here is understand OpenLayers code not to learn CSS. You can find the whole code in the examples source code.

Now, let's go to create the layer instances. We are going to create tile layer with sources from OpenStreetMap, MapQuest and Stamen services:

```
    var layerOSM = new ol.layer.Tile({
        source: new ol.source.OSM(),
        name: 'OpenStreetMap'
    });
    var layerMQ = new ol.layer.Tile({
        source: new ol.source.MapQuest({
            layer: 'osm'
        }),
        name: 'MapQuest'
    });
    var layerStamenWater = new ol.layer.Tile({
        source: new ol.source.Stamen({
            layer: 'watercolor'
        }),
        name: 'Watercolor'
    });
    var layerStamenTerrain = new ol.layer.Tile({
        source: new ol.source.Stamen({
            layer: 'terrain'
        }),
        name: 'Terrain'
    });
```

Note how depending on the source subclass we need to specify additional properties.

Create a layer group to store the two Stamen layers:

```
    var layerStm = new ol.layer.Group({
        layers: [layerStamenWater, layerStamenTerrain],
        name: 'Stamen Group'
    });
```

As you can see, we have added an extra property `name` to the layers and layer group, which we will allows as to show a nice name in the tree control.

Once we have the layers organized we can create the map instance:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [layerOSM, layerMQ, layerStm],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([-100.1833, 41.3833], 'EPSG:4326', 'EPSG:3\
857'),
            zoom: 4
        })
    });
```

The map instance contains a `ol.layer.Group` instance to store all the layer reference. It can be understood as the *root* node of the tree and we also want to add an extra `name` property to it:

```
    map.getLayerGroup().set('name', 'Root');
```

Once we have the main elements our application requires to build the tree control from the map layers, which is nothing more than an unsigned list with a `<li>` element for each layer. The control is made through the `initializeTree()` function:

```
    initializeTree();
```

The code for the `initializeTree()` function and other helper functions involved are explained in the next section to avoid make this one too extensive. You can see the complete code in the examples source code of the book.

Each element in the tree is represented with a name, an slider to change the layer opacity and a checkbox to change its visibility. So we register a listener for the `slide` event in the slider control to change the layer opacity:

```
    // Handle opacity slider control
    $('input.opacity').slider().on('slide', function(ev) {
        var layername = $(this).closest('li').data('layerid');
        var layer = findBy(map.getLayerGroup(), 'name', layername);
        layer.setOpacity(ev.value);
    });
```

The opacity slider is a simply `<input>` element which we have applied the jQuery slider[8] plugin.

and another listener for the `click` event on the checkbox to change the layer visibility:

[8] http://www.eyecon.ro/bootstrap-slider/

```
    // Handle visibility control
    $('i').on('click', function() {
        var layername = $(this).closest('li').data('layerid');
        var layer = findBy(map.getLayerGroup(), 'name', layername);

        layer.setVisible(!layer.getVisible());

        if (layer.getVisible()) {
            $(this).removeClass('glyphicon-unchecked').addClass('glyphicon-check\
');
        } else {
            $(this).removeClass('glyphicon-check').addClass('glyphicon-unchecked\
');
        }
    });
```

2.4.3.3 How it works...

Once the layers and map instances are created the first thing we must do is to initialize the tree control to show the layers of the map and allows to change its opacity and visibility. As we seen previously this is done using the `initializeTree()` function:

```
    function initializeTree() {
        var elem = buildLayerTree(map.getLayerGroup());
        $('#layertree').empty().append(elem);

        $('.tree li:has(ul)').addClass('parent_li').find(' > span').attr('title'\
, 'Collapse this branch');
        $('.tree li.parent_li > span').on('click', function(e) {
            var children = $(this).parent('li.parent_li').find(' > ul > li');
            if (children.is(":visible")) {
                children.hide('fast');
                $(this).attr('title', 'Expand this branch').find(' > i').addClas\
s('glyphicon-plus').removeClass('glyphicon-minus');
            } else {
                children.show('fast');
                $(this).attr('title', 'Collapse this branch').find(' > i').addCl\
ass('glyphicon-minus').removeClass('glyphicon-plus');
            }
            e.stopPropagation();
        });
    }
```

We create the HTML for the tree with the `buildLayerTree()` function and add to the `<div id="layertree" ...>` element. Next code is responsive expand and collapse tree nodes with a nice effect. It is not related to OpenLayers3 so we do not spent time describing it.

The `buildLayerTree()` function iterates over the map layers and creates a `<li>` element for each one. The element contains the layer name, a checkbox and slider element:

For the slider element we have used a jQuery plugin which applied on an `<input>` element transforming it on a slider.

```
    function buildLayerTree(layer) {
        var elem;
        var name = layer.get('name') ? layer.get('name') : "Group";
        var div = "<li data-layerid='" + name + "'>" +
                "<span><i class='glyphicon glyphicon-file'></i> " + layer.get('n\
ame') + "</span>" +
                "<i class='glyphicon glyphicon-check'></i> " +
                "<input style='width:80px;' class='opacity' type='text' value=''\
 data-slider-min='0' data-slider-max='1' data-slider-step='0.1' data-slider-tool\
tip='hide'>";
        if (layer.getLayers) {
            var sublayersElem = '';
            var layers = layer.getLayers().getArray(),
                    len = layers.length;
            for (var i = len - 1; i >= 0; i--) {
                sublayersElem += buildLayerTree(layers[i]);
            }
            elem = div + " <ul>" + sublayersElem + "</ul></li>";
        } else {
            elem = div + " </li>";
        }
        return elem;
    }
```

Now we have the tree control initialized we are going to see how the listener functions responsible to change layers opacity and visibility are implemented.

Each time a slider is modified the next anonymous listener function is executed receiving the `ev` parameter with event information. From the modified `<input>` element we retrieve the layer's name stored in the `data-layerid` attribute, obtain a layer object reference with the `findBy()`function and finally modify the layer opacity with the `setOpacity()` method:

```
    $('input.opacity').slider().on('slide', function(ev) {
        var layername = $(this).closest('li').data('layerid');
        var layer = findBy(map.getLayerGroup(), 'name', layername);

        layer.setOpacity(ev.value);
    });
```

The `findBy()` function is a helpful function which returns the value of the specified property:

```
    function findBy(layer, key, value) {
        if (layer.get(key) === value) {
            return layer;
        }
        // Find recursively if it is a group
        if (layer.getLayers) {
            var layers = layer.getLayers().getArray(),
                    len = layers.length, result;
            for (var i = 0; i < len; i++) {
                result = findBy(layers[i], key, value);
                if (result) {
                    return result;
                }
            }
        }
        return null;
    }
```

The code for the listener function which changes the layer visibility follows the same steps than the previous one. In addition we need to change the checkbox glyph between checked and unchecked:

```
    $('i').on('click', function() {
        var layername = $(this).closest('li').data('layerid');
        var layer = findBy(map.getLayerGroup(), 'name', layername);

        layer.setVisible(!layer.getVisible());

        if (layer.getVisible()) {
            $(this).removeClass('glyphicon-unchecked').addClass('glyphicon-check\
');
        } else {
            $(this).removeClass('glyphicon-check').addClass('glyphicon-unchecked\
');
        }
    });
```

2.4.4 Image layer

2.4.4.1 Goal

Because not always we need to load imagery from a tiled provider, in this example we are going to see how we can add a single image layer to the map using the `ol.layer.Image` class.

We are going to load an historical image from the University of Texas[9] and place on top of a tiled layers. In addition we will place a slider control to change the image opacity.

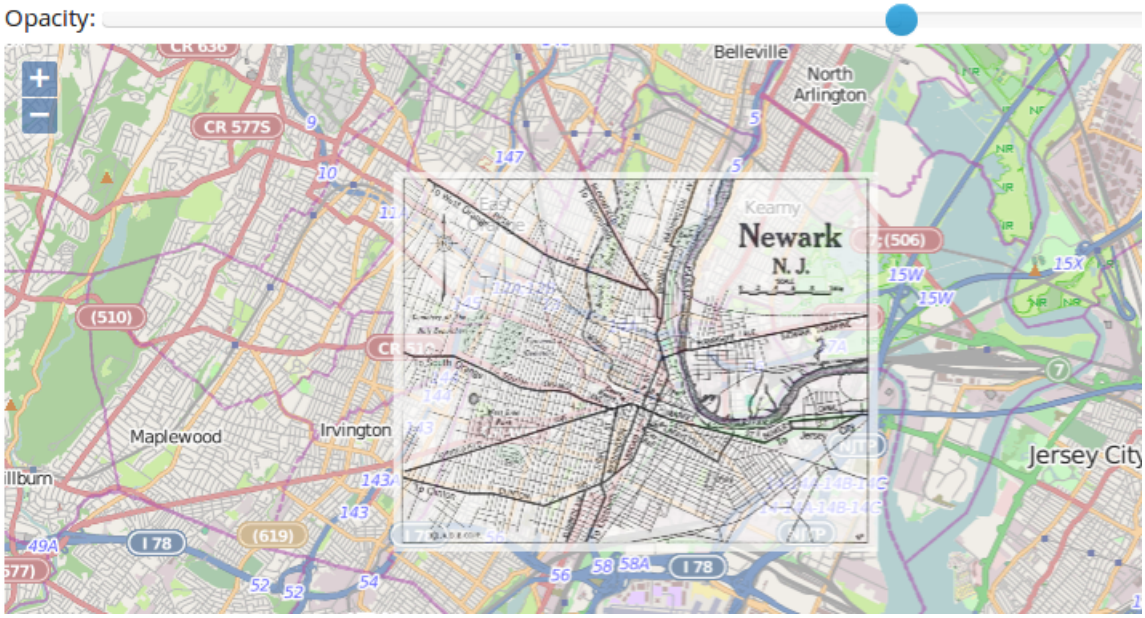


Image layer

2.4.4.2 How to do it...

Create the HTML elements for the slider and the map:

```
    Opacity: <input style='width: 80%;' class='opacity' type='text' data-slider-\
min='0' data-slider-max='1' data-slider-step='0.01' data-slider-tooltip='hide'>

    <div id="map" class="map"></div>
```

The opacity slider is a simply `<input>` element which we have applied the jQuery slider[10] plugin.

Add the code to create the map instance with a tiled layer loading tiles from OpenStreetMap source:

[9] https://www.lib.utexas.edu/maps/historical
[10] http://www.eyecon.ro/bootstrap-slider/

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([-74.15655, 40.74222], 'EPSG:4326', 'EPSG:\
3857'),
            zoom: 12
        })
    });
```

Add the code to create an `ol.layer.Image` instance and add to the map:

```
    var imageLayer = new ol.layer.Image({
        opacity: 0.75,
        source: new ol.source.ImageStatic({
            attributions: [
                new ol.Attribution({
                    html: '&copy; <a href="https://www.lib.utexas.edu/maps/histo\
rical/">University of Texas Libraries</a>'
                })
            ],
            url: 'https://www.lib.utexas.edu/maps/historical/newark_nj_1922.jpg',
            imageSize: [691, 541],
            projection: map.getView().getProjection(),
            imageExtent: ol.extent.applyTransform([-74.22655, 40.71222, -74.1254\
4, 40.77394], ol.proj.getTransform("EPSG:4326", "EPSG:3857"))
        })
    });

    map.addLayer(imageLayer);
```

Finally, add the code to apply the slider plugin to the `<input>` element and register a listener to change the image layer opacity:

```
    $('input.opacity').slider({
        value: imageLayer.getOpacity()
    }).on('slide', function(ev) {
        imageLayer.setOpacity(ev.value);
    });
```

2.4.4.3 How it works...

The code is relatively easy to understand. Because we want to use a single image as a layer we need to use the `ol.layer.Image` and, because its content comes from a single image file we load it using the `ol.source.ImageStatic` source, which allows to load a single image file from a url.

```
    var imageLayer = new ol.layer.Image({
        source: new ol.source.ImageStatic({
        ...
        })
    });
```

The layer is responsible to load the image while the source is responsible to load the data. In addition, the source must have the needed attributes to inform the layer about the data bounds, projection and resolution, so the layers knows how to place it within the map. This is done through the properties specified in the source instance: `imageSize`, `imageExtent` and `projection`.

More information about data sources can be found in the chapter Data sources and formats

Finally, to change the image layer opacity we have registered a couple of listener function which changes the layer opacity with the layer's `setOpacity()` method.

2.4.5 Visualizing layers depending on resolution

2.4.5.1 Goal

The goal of this example is to show how we can control when layers can be visible using of `minResolution` and `maxResolution` properties.

We are going to create a simple map with two layers where the previous parameters are configured so only one layer are visible depending on the view resolution (zoom level).

2.4.5.2 How to do it...

Create the HTML element to hold the map:

```
    <div id="map" class="map"></div>
```

Add the next code to initialize the map with two layers:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add to tile layers specifying its min and max resolution property
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'sat'
                }),
                minResolution: 2500,
            }),
            new ol.layer.Tile({
                source: new ol.source.OSM(),
                maxResolution: 2500
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

2.4.5.3 How it works...

As we say at the goal section, the magic of this example resides in the `minResolution` and `maxResolution` properties. These properties determines between which resolution values the layer must be visible. If we no specify a value it means there is no top or bottom resolution value to restrict the visibility.

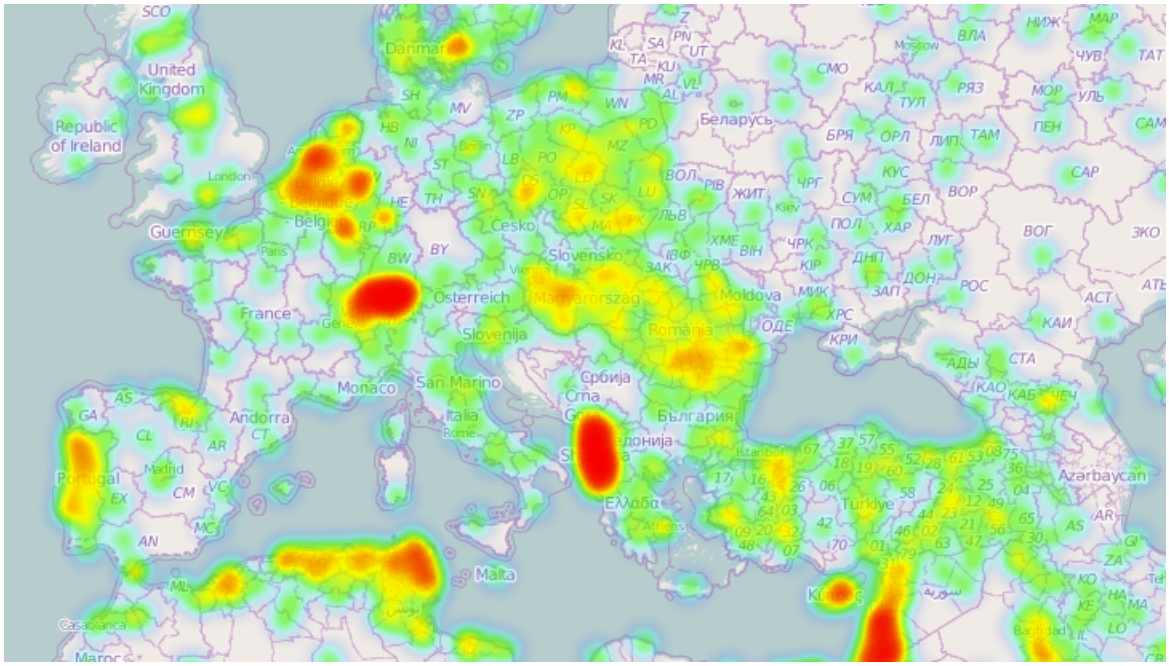
We have configured the MapQuest layer to be visible from an undefined maximum resolution, that is, there is no maximum value at which layer must not be visible, until a minimum resolution value of `2500`. On the other hand, the OpenStreetMap layer is configured to be visible from a maximum resolution of `2500` until any minimum value (because it is not specified).

This way we can make two layers complement themselves. This is useful, for example, when we have two tile sets with different resolutions and the first goes from 0 to 9 zoom level and the second with more resolution degree from 10 to 15 zoom level.

2.4.6 A heatmap with the world's cities density

2.4.6.1 Goal

We are going to create a heat map from a GeoJSON file containing the most important cities of the world so we can see which countries has a high density of cities.



Heatmap from a set of cities

2.4.6.2 How to do it...

Create the HTML element to hold the map:

```
1       <div id="map" class="map"></div>
```

Create an OpenStreetMap layer that acts as base layer and the heatmap layer from the GeoJSON source:

```
    var osmLayer = new ol.layer.Tile({
        source: new ol.source.OSM()
    });

    var heatmapLayer = new ol.layer.Heatmap({
        source: new ol.source.GeoJSON({
            url: './data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });
```

Create the map instance and add the two previous layers:

```
    var map = new ol.Map({
        renderer: 'canvas',
        target: 'map',
        layers: [osmLayer, heatmapLayer],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 4
        })
    });
```

2.4.6.3 How it works...

The code has no secret. We have created an `ol.layer.Heatmap` instance using a GeoJSON source which contains most of the cities of the world.

As we said at the beginning, we have not set any weight for the features, neither the features have a weight properties, so at rendering time all features has the same importance and the map becomes a density map of cities:

```
    var heatmapLayer = new ol.layer.Heatmap({
        source: new ol.source.GeoJSON({
            url: './data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });
```

Note, we are set the `projection` property. Because the map is using, by default, the `EPSG:3857` projection and our GeoJSON data is in `EPSG:4326`, so we need to specify to the source it must transform data between projections.

3. Data sources and formats

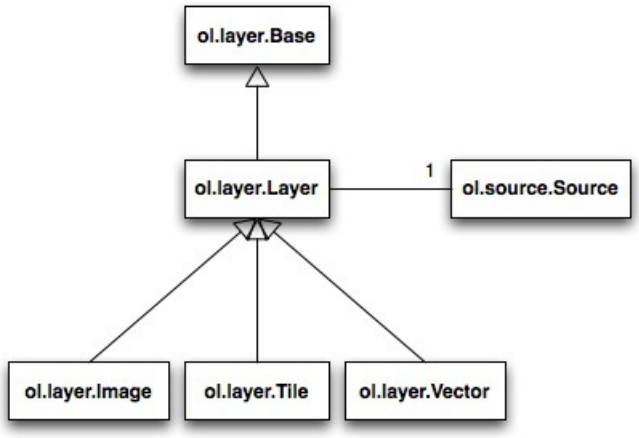
The OpenLayers3 design differentiates between the concept of *layer*, which represents a set of information, from the concept of *source*, which is the responsible to obtain the data to be used by the layer.

Thanks to this differentiation we have a great degree of flexibility and we are allowed to create instances from the same layer type but using different data sources. For example, we can create a tiled layer that loads tiles from OpenStreetMap project and another tiled layer that loads tiles from MapQuest service. There is no need to have two types of layers, but two types of sources.

A key aspect of OpenLayers3 is the wide range of *source* implementations it offers, allowing to load both raster and vector information.

3.1 The root source class

As we have saw in the Layers chapter, the layer delegates the task to load data to a *source* instance. The source classes encapsulates all the logic required to load and read data, leaving this complexity outside the layer. This relationship can be found in the next figure.



Layer and source relationship

`ol.source.Source` is the root class within the source classes hierarchy and defines the properties and methods common to all subclasses. Later, each concrete subclass follows its own rules extending the base class, with properties or methods, to allow read a concrete kind of data.

These common properties we can found at the `ol.source.Source` class are:

- `attributions`: It is a message to specify attribution information to the source. It must be an array of `ol.Attribution` instances. Each instance accepts a `html` property that must contain the HTML code that later will be placed on top of the map.
- `extent`: Specifies the extent that occupies the data loaded by the source class. It must be an array with `[minLongitude, minLatitude, maxLongitude, maxLatitude]` values.
- `logo`: A string with the URL to the logo to be shown with the source attribution information.
- `projection`: a target projection. The data loaded by the source will be transformed to this projection. It must be an instance of `ol.proj.Projection` or a string with the projection code.

Note, the class also offers the setters and getters methods necessary to get and modify these properties.

Next code, shows how to use previous properties in a hypothetical subclass named `SomeSource` source class:

```
var mysource = new ol.source.SomeSource({
    logo: 'http://some_server/my_logo_is_here',
    projection: 'EPSG:4326',
    extent: [-100, -50, 100, 50],
    attributions: [
        new ol.Attribution({
            html: 'This tiles comes from ...'
        })
    ]
});
```

The attribution control (see `ol.control.Attribution` class) collects all the attributions from all the layers sources and shows them on the map, usually at bottom place. Each time we create a new `ol.Map` instance, if no controls are specified by default, an attribution control is created and attached to the map.

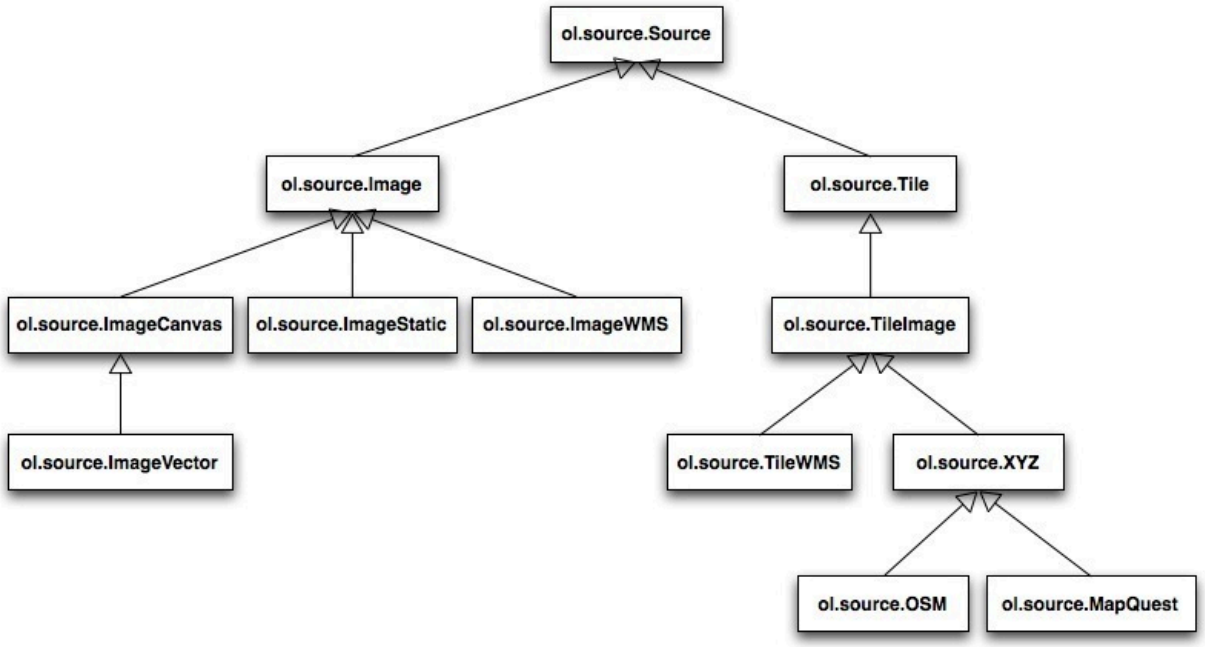
Below `ol.source.Source` class we found an extensive set of classes that conforms its own hierarchies and allows to access raster or vector data. Next sections describes each hierarchy and describes how to use each concrete source implementation.

3.2 Raster sources

3.2.1 Introducing the raster hierarchy

For raster sources we found two main subclasses: `ol.source.Image` and `ol.source.Tile`. The main difference between them is while the first is oriented for layers composed by a single image, the

second has been designed for tiled layers, that is, layers that are composed by a set of images conforming a pyramid of tiles (see the Tiled layers section at Layers chapter for better understand of tiled layers).



Raster source hierarchy

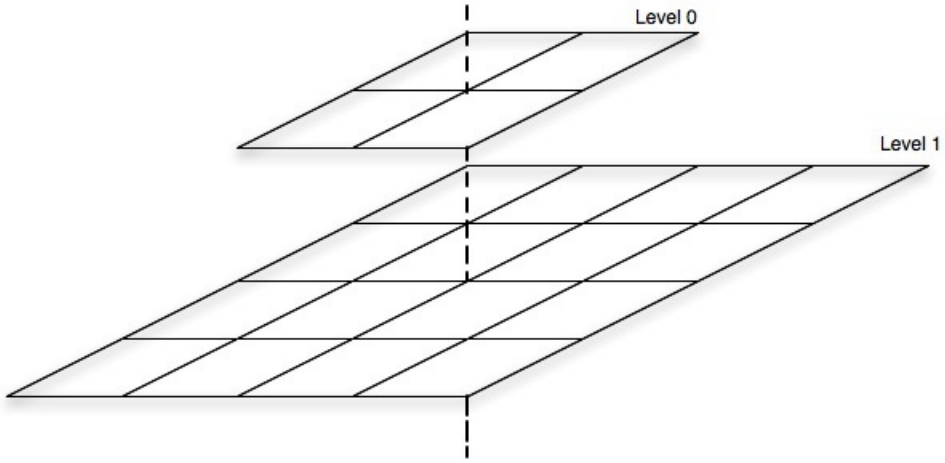
The figure shows some (not all) of the subclasses available to access raster data and how they conform two well differentiate hierarchies. We can find sources suitable to read data from: a WMS or WMTS compliant server, a tile provider like OpenStreeMap or MapQuest service, a static image, etc.

Note we can not use all the available sources on a given layer, that is, some layers restricts the kind of sources we can use on it. For example, `ol.layer.Tile` class only accepts subclasses of `ol.source.Tile` and `ol.layer.Image` only works with `ol.source.Image`. This applies for vector sources too.

3.2.2 Tile grids

In previous chapter we learned tiled layers conforms a pyramid of tiles (see the Tiled layers section at Layers chapter). At the beginning of this chapter, we have learned these tiles are obtained through a source (concretely by a subclass of `ol.source.Tile`) and because of this, the source must know the pattern followed by the tiles: what are the minimum and maximum zoom level, the resolution of each level, the tile size, etc.

All these grid information is represented in OpenLayers3 by the `ol.tilegrid.TileGrid` class.



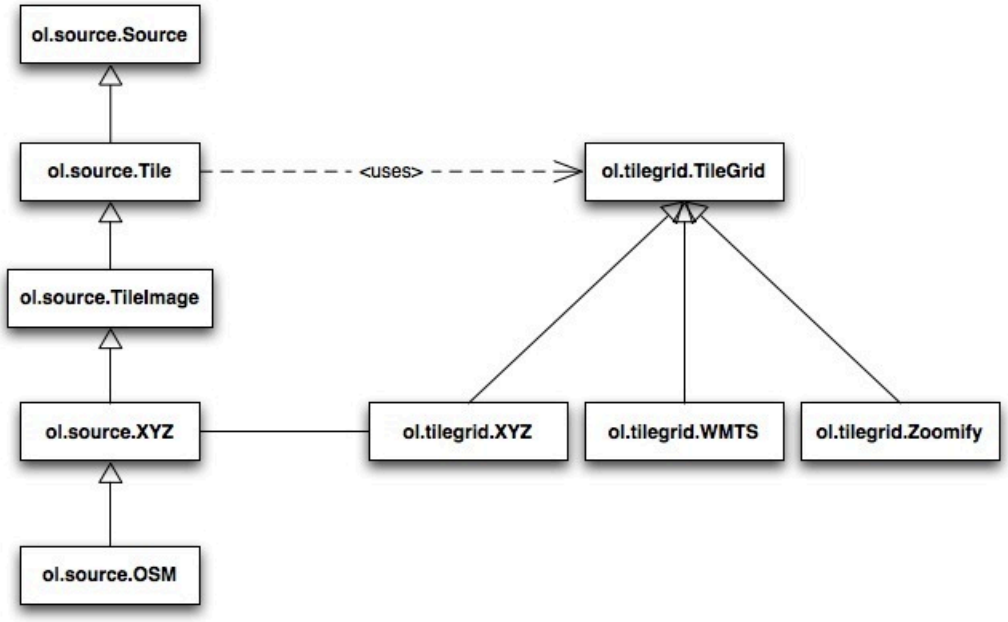
Pyramid of tiles

`ol.tilegrid.TileGrid` gives us flexibility enough to configure any possible pattern about the pyramid of tiles but, to reduce our work as developers, OpenLayers3 offers us an implementations for some common patterns. This way, we can found the classes:

- `ol.tilegrid.XYZ`, follows the common `XYZ` pattern, where `XY` specifies the tile position within grid and `Z is the zoom level. Usually, the requested URLs that uses this tilegrid follows the pattern` http://some_server/z/x/y.png‘.
- `ol.tilegrid.WMTS`, follows the WMTS standard pattern.
- `ol.tilegrid.Zoomify`, specially designed to work with the Zoomify[1] service pattern.

In the next figure, we can see the `ol.source.Tile` class has relationship with the `ol.tilegrid.TileGrid` class. In practice, this means each concrete instance of `ol.source.Tile` uses a concrete instance of `ol.tilegrid.TileGrid` to know how request tiles. For example, `ol.source.OSM` uses a `ol.tilegrid.XYZ` to know the tile grid it must use.

[1] http://www.zoomify.com



Tile source and tile grid relationship

3.2.3 Sources to access tile providers

We can define *tile provider* as any service, public or private, that serves tiles in some way. In this category we found services like Bing, MapQuest, Stamen or the open source project OpenStreetMap.

Tile providers usually pre-generates images and stores them using some file structure organization. Each provider may differ on its configuration about grids, resolution, tile dimensions, etc and because of this the parameters each source class may need can differ from each other.

3.2.3.1 OpenStreetMap

Probably, the best description about OpenStreetMap project is what we can find at the project's About[2] section:

> OpenStreetMap[3] is built by a community of mappers that contribute and maintain data about roads, trails, railway stations, and much more, all over the world.

The results of all this work are stored in a database and, periodically, the huge set of tile images are recreated to reflect all the updates and changes.

These tiles can be freely accessed through the servers the project has available for public usage.

[2] http://www.openstreetmap.org/about
[3] http://www.openstreetmap.org/

Please, read the Tile usage policy[4]. OpenStreetMap run entirely on donated resources so make a good use of them.

To use OpenStreetMap imagery in your tile layer you can simply need to create a fresh `ol.source.OSM` source instance:

```
1     var osmLayer = new ol.layer.Tile({
2         source: new ol.source.OSM()
3     });
```

3.2.3.2 MapQuest and Stamen

As we have mentioned, the data collected by the OpenStreetMap is freely available. Some companies has used this data to produce its own maps, with custom styles and colors. Two examples of these companies are MapQuest[5] and Stamen[6].

OpenLayers3 offers sources to load data from the two providers: `ol.source.Stamen` and `ol.source.MapQuest`. Both sources offers tiles using different color styles and visualization options, so we need to specify the `layer` property when creating a new source instance.

For `ol.source.Stamen`, the available `layer` values are `terrain`, `toner` and `watercolor`, while for the `ol.source.MapQuest` it can take the values `osm`, `sat` and `hyb`.

Next, are examples on how to use both sources:

```
1     var satLayer = new ol.layer.Tile({
2         source: new ol.source.MapQuest({
3             layer: 'sat'
4         })
5     });
6
7     var waterLayer = new ol.layer.Tile({
8         source: new ol.source.Stamen({
9             layer: 'watercolor'
10        })
11    });
```

3.2.3.3 Bing Maps

Bing Maps[7] is the map service offered my MicroSoft. Its data can be explored via its web page and, in addition, it can be accessed as a web service. Bing Maps not only offers imagery but location, elevation, routes and traffic services.

[4] http://wiki.openstreetmap.org/wiki/Tile_usage_policy
[5] http://wiki.openstreetmap.org/wiki/MapQuest
[6] http://maps.stamen.com
[7] http://www.bing.com/maps/

All the services from Bing Maps requires you obtain a so called *API key* to be placed on each request and that will identify your application when requesting data to Bing Maps.

See the Getting a Bing Maps key[8] for an in depth description.

OpenLayers3 offers the `ol.source.BingMaps` source which allows to get imagery from the Bing Maps service. The class allows to specify next properties: * `key`: the API key obtained from Bing Maps * `imagerySet`: the kind of imagery to be requested. Allowed values are : * `Aerial`: Aerial imagery. * `AerialWithLabels`: Aerial imagery with a road overlay. * `Road`: Roads without additional imagery. * `culture`: Determines some the result values depending on your language, like map labels, route instructions, etc (See Supported Culture Codes[9]).

You can get more `imagerySet` information at Get Imagery Metadata[10].

Next code shows a tile layer using Bing Maps imagery. It is configured to show the `AerialWithLabels` imagery set using labels in Spanish language:

```
1     var bingLayer = new ol.layer.Tile({
2         source: new ol.source.BingMaps({
3             key: 'your-API-key',
4             imagerySet: 'AerialWithLabels',
5             culture: 'es-ES'
6         })
7     });
```

3.2.4 Sources to access OGC compliant servers

In addition to proprietary tile providers, OpenLayers3 offers sources to load imagery from OGC compliant servers using both WMS and WMTS standards.

3.2.4.1 Requesting a WMS server

A well known OGC standard is the WMS (Web Map Service[11]), which allows serve georeferenced map images.

[8] http://msdn.microsoft.com/en-us/library/ff428642.aspx
[9] http://msdn.microsoft.com/en-us/library/hh441729.aspx
[10] http://msdn.microsoft.com/en-us/library/ff701716.aspx
[11] http://en.wikipedia.org/wiki/Web_Map_Service

Note, the work of a WMS server is generally very CPU expensive: extract and validate query parameters (like bounding box, layers, styles, projection, ...), load data from files or database, reproject if required, apply a palette, etc.

It is out of the scope of this book to introduce the WMS protocol so I encourage the reader to take a look at the specification and understand the related concepts like requests, allowed parameters, etc.

OpenLayers3 offers two possible source classes to request a WMS servers: `ol.source.ImageWMS` and `ol.source.TileWMS`.

3.2.4.1.1 Single image query `ol.source.ImageWMS` allows to request for a single image that conforms the map's view extent (well, really it depends on the `ratio` property. See it later). This means, each time we move the map, no matter how little the change is, a new query is made to the WMS server for the new bounding box.

```
1     var tileLayer = new ol.layer.Tile({
2         source: new ol.source.ImageWMS({
3             url: 'http://server_url',
4             params: {
5                 // Put WMS parameters here
6             }
7         })
8         ...
9     });
```

Because the CPU expensive nature of the work of the WMS server, most servers allows to cache the generated images following some cache rules. As you can suppose, caching the generated images in this scenario, where each movement implies generate a new image, is not really effective.

We can improve a bit the client performance loading an image bigger than the view's extent, so that slightly movements does not implies a new query to the server if we does not go beyond the limits of the previously queried image. This can be achieves using the `ratio` property.

The `ratio` property is directly related with the proportion of the map's view which is queried to the server. A `ratio=1` means the source queries the server a bounding box equal to the current view's extent. A `ratio=2` means the source queries the server a bounding box twice the current view extent.

If the resultant image is greater than the map view extent, more probably when user pans the view no new query to the server will be needed because the image will cover the new view extent.

By default the `ol.source.ImageWMS` source uses a `ratio=1.5`, which is a well equilibrated proportion.

3.2.4.1.2 Tiled query Instead make a single query for the whole view extent, there is another approach to get imagery from a WMS server.

`ol.source.TileWMS` request WMS images as if it were a tile provider. To make this, the source computes a tile grid for each zoom level and divides it in tiles, each one with a bounding box.

So, given a zoom level the `ol.source.TileWMS` source makes one WMS request for each tile, instead making a single query for the whole view.

The approach is similar than the used by tile providers. See the Tiled layers section at Layers chapter.

Initially, this can seem a bad strategy but in practice it has two main benefits:

- The WMS server works on smaller pieces of data and produces smaller images, meaning the server works less.
- The fact the tiles has always the same bounding box helps helps in the sense we can cache data and be sure they will be requested again in the future.

The way to use `ol.source.TileWMS` source is very similar to `ol.source.ImageWMS`. We only need to understand the way to work is different.

```
var tileLayer = new ol.layer.Tile({
    source: new ol.source.TileWMS({
        url: 'http://server_url',
        params: {
            // Put WMS parameters here
        }
    })
    ...
});
```

3.2.4.1.3 Using WMS parameters WMS is a complex but powerful protocol. It offers different operations, like *GetCapabilities*, *GetMap* or *GetFeatureInfo*, and for each of these operations we can pass different parameters.

When working with the WMS source classes, requests to the server are always made using the *GetMap* operation, which is responsible to return a georeferenced image. This operation accepts a great number of parameters (as example, see the GeoServer[12] documentation for the GetMap[13] operation).

Both source classes, `ol.source.TileWMS` and `ol.source.ImageWMS`, accepts the next properties:

[12] http://geoserver.org

[13] http://docs.geoserver.org/stable/en/user/services/wms/reference.html#getmap

- `url`: The URL to the server to be queried.
- `urls`: For the `ol.source.TileWMS` source class, we can specify an array of servers, instead use a single one using `url` property, so that OpenLayers3 will balance the requests among them.
- `params`: This is a JavaScript object where we can specify many of the WMS *GetMap* operation parameters, like `LAYERS`, `TRANSPARENT` or `STYLES`.

```
1     params: {
2         'LAYERS': 'custom_layer',
3         'TRANSPARENT': true
4     }
```

Note, not all the *GetMap* operation parameters specified in the `params` property will be used: *BBOX* is obtained from the current view extent, *CRS* and *SRS* parameters are ignored and the used values are taken from the `projection` property and, for the `ol.source.TileWMS` source class, the *WIDTH* and *HEIGHT* parameters are ignored and the values are taken from a computed `tileSize` variable.

Once created a WMS source, no matter if `ol.source.ImageWMS` or `ol.source.TileWMS`, we can change the WMS parameters with the method `updateParams()`, which accepts a `params` object as argument. `updateParams()` will request for information again and forces to redraw the layer again.

3.2.4.1.4 Reading WMS server capabilities Another interesting operation in WMS protocol is the *GetCapabilities*. This operation allows a client to query the server and know the available layers it can serve, the extent each layer covers, the available styles for the layers, the image formats in which the server can return the produced images or the available projections the server can translate the data.

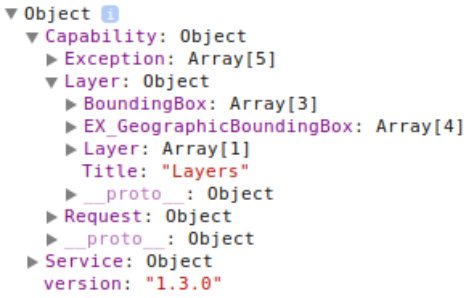
The response of the *GetCapabilities* operation is a XML file, where all the previous information is specified, and it can be tedious to navigate among the information.

Fortunately, OpenLayers3 offers the `ol.format.WMSCapabilities` class that allows to easily parse the server capabilities response to a JavaScript object, much more easy to navigate.

As we can see in the next code, we only need to create a new format instance and invoke its `read()` method to convert some XML string into a JavaScript object:

```
1     var response = ...; // Some XML document response
2     var capabilities = new ol.format.WMSCapabilities();
3     var result = capabilities.read(response);
```

The image shows the browser output for a capabilities document:



Parsed WMS capabilities

3.2.4.2 Loading tiles from a WMTS server

Demonstrated the benefit of tile providers serving pre-rendered images, the OGC (Open Geospatial Consortium) develops a new standard specially designed for this purpose. The result was the WMTS (Web Map Tile Service[14]) specification.

It is out of the scope of this book to introduce the WMTS protocol so I encourage the reader to take a look at the specification and understand the concepts of tile matrix set, dimension, tile row, etc.

OpenLayers3 offers the `ol.source.WMTS` class that allows to request tiles from a WMTS compliant server. In a similar way the `ol.source.TileWMS` class accepts a `params` property to take full control of the parameters sent on each request, the `ol.source.WMTS` class accepts a set of properties that allows us to control many of the WMTS parameters too:

- `url`, the server URL. We can also use the `urls` property to make request against more than one server.
- `layer`, the layer name to request for
- `style`, the style to be used on the server side
- `matrixSet`, the matrix set name to be used
- `tileGrid`, specifies the tile grid to be used by the source that, in this case, must be an instance of `ol.tilegrid.WMTS`.

Respect the `ol.tilegrid.WMTS` instance it must contain, at least, the properties:

- `origin`, the top left coordinate where tiles start,
- `resolutions`, an array of resolutions for each zoom level (See Resolutions and zoom levels section at The Map and the View chapter for better understand how to compute resolutions).

[14] http://en.wikipedia.org/wiki/Web_Map_Tile_Service

- `matrixIds`, an array of matrix identifiers

To put all in a sample, the next code shows a tile layer using a WMTS source (go to the practice section to see a real life example):

```
new ol.layer.Tile({
  opacity: 0.7,
  source: new ol.source.WMTS({
    url: 'http://wmts_server_url/',
    projection: 'EPSG:3857',
    // WMTS properties
    layer: 'layer_name',
    format: 'image/png',
    matrixSet: 'matrix_set_name',
    tileGrid: new ol.tilegrid.WMTS({
        origin: [-180, 85], // top-left origin coordinates
        resolutions: resolutionsArray,
        matrixIds: matrixIdsArray
    }),
    style: 'default'
  })
})
```

At the moment to write these lines and, at the opposite like with WMS servers, OpenLayers3 does not offers any class to get the WMTS capabilities of a server. So, you need to query the server on your own way and create the right tile grid to make the requests.

3.2.5 Other raster sources

3.2.5.1 Loading an static image

There can be situations where we have a set of images we want to draw on the map and they were not served as a tile set, like previous tile providers, neither using an WMS compliant server. We are talking about a set images served directly by a HTTP server.

For this purpose, OpenLayers3 offers the `ol.source.ImageStatic` source class that allows to render a georeferenced image file in the map.

Before see how to use it we need to understand how OpenLayers3 works with raster layers. We know each zoom level has a different resolution, that is, each pixel is equivalent to a distance (see Resolutions and zoom levels section in the The Map and the View chapter to get more information

about zoom levels and resolutions). For example, at level zero each pixel represents `156,543.034` meters.

For each layer, no matter if it is a tiled layer or single image layer, OpenLayers3 needs to know the size of the image, the resolution of each pixel and its bounds. Because of this, when working with `ol.source.ImageStatic` class we need to explicitly indicate the image pixel size and its extent.

Note the class automatically computes the pixel resolution of the image given its pixel size and extent as: `var imageResolution = (imageExtent[3] - imageExtent[1]) / imageSize[1];`.

As we have commented previously, we can not use all the sources on a given layer. This way, the `ol.source.ImageStatic` source class is restricted to be used on an `ol.layer.Image` layer class.

Next code, shows how we can create an `ol.layer.Image` layer loading a remote image using the `ol.source.ImageStatic` source:

```
1     var imageLayer = new ol.layer.Image({
2         source: new ol.source.ImageStatic({
3             url: 'http://server/path/to/image.png',
4             imageSize: [width, height],
5             projection: 'EPSG:4326',
6             imageExtent: [minLon, minLat, maxLon, maxLat]
7         }),
8         ...
9     });
```

3.2.5.2 Using a HTML5 canvas as source

Canvas element[15] is a powerful new feature introduced in HTML5 specification, which allows scriptable rendering of 2D shapes, and is supported by the major browsers.

As an example of canvas powerful we can highlight the Heatmap layer, which internally makes use of the canvas element to render features. See Heatmap layer section at Layers chapter.

OpenLayers3 offers us the `ol.source.ImageCanvas` class, which gives us an absolutely free degree of creativity to render anything we can need. It is a subclass of `ol.source.Image`, so it is designed to render a single image each time.

The most important property of the class is the `canvasFunction`, which allows to pass a function that will be executed each time the layer needs to be rendered and must return `<canvas>` element reference. OpenLayers3 will get its content as a raster and will render on the map at the right place.

[15] http://en.wikipedia.org/wiki/Canvas_element

The `<canvas>` element you return in the `casnvasFunction` is never added to to DOM. The content is extracted as an image and placed on the map.

The `canvasFunction` receives four arguments that give us all the necessary information to render data:

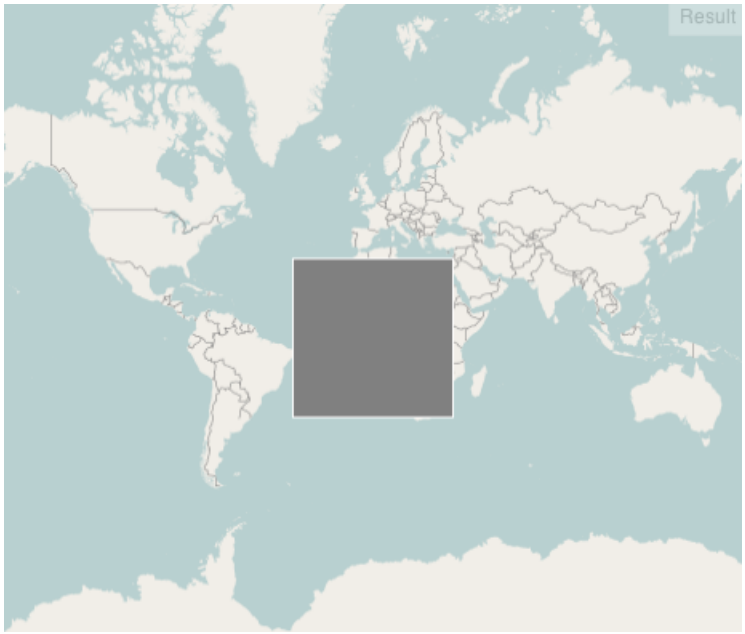
- `extent`, the canvas extent. Note, it is not the same as the current map view extent.
- `resolution`, image resolution.
- `pixelRatio`, the device pixel ratio.
- `size`, the canvas size. Note, it is not the same as the current map viewport.
- `projection`, the canvas projection.

Neither the `extent` nor `size` passed to the `canvasFunction` are the same as the current map view. For performance reasons, canvas is usually twice the map's view size so we can draw data outside the map's view. This way if we pan the map there is no need to render the layer again.

Next code shows a really simple image canvas source sample that renders a square in the center of the map:

```
1  // Create layer with an Image Canvas source rendering a square
2  var canvasLayer = new ol.layer.Image({
3      source: new ol.source.ImageCanvas({
4          canvasFunction: function(extent, resolution, pixelRatio, size, projectio\
5  n) {
6              // Create canvas element and set size
7              var canvas = document.createElement('canvas');
8              var context = canvas.getContext('2d');
9              var canvasWidth = size[0], canvasHeight = size[1];
10             canvas.setAttribute('width', canvasWidth);
11             canvas.setAttribute('height', canvasHeight);
12
13             // Draw square
14             var sqsize = 100;
15             var ox = (canvasWidth - sqsize) / 2;
16             var oy = (canvasHeight - sqsize) / 2;
17             context.fillStyle = 'grey';
18             context.strokeStyle = 'white';
19             context.fillRect(ox, oy, sqsize, sqsize);
20             context.strokeRect(ox, oy, sqsize, sqsize);
```

```
21
22              return canvas;
23          },
24          projection: 'EPSG:3857'
25      })
26  });
27
28  map.addLayer(canvasLayer);
```



Sample ImageCanvas result

3.3 Vector sources and formats

3.3.1 Introducing vector source and format hierarchies

OpenLayers3 offers an extensive set of source classes suitable to read vector information in the most common and used data formats: GeoJSON[16], TopoJSON[17], GPX[18], KML[19], OSMXML[20], etc.

This section is oriented to show how to use vector source classes. In chapter Vector layers we will see in depth how to work with vector information.

[16] http://en.wikipedia.org/wiki/GeoJSON
[17] http://en.wikipedia.org/wiki/TopoJSON
[18] http://en.wikipedia.org/wiki/GPS_eXchange_Format
[19] http://en.wikipedia.org/wiki/Keyhole_Markup_Language
[20] http://wiki.openstreetmap.org/wiki/OSM_XML

Vector data are a bit more complex than raster one, by the fact they can be represented in different data formats so, to help on the task of read and write vector information, OpenLayers3 introduces the concept of *format* classes, which knows how to read/write from/to a given data format file. While the vector source classes retrieves data, the *format* classes reads the information.

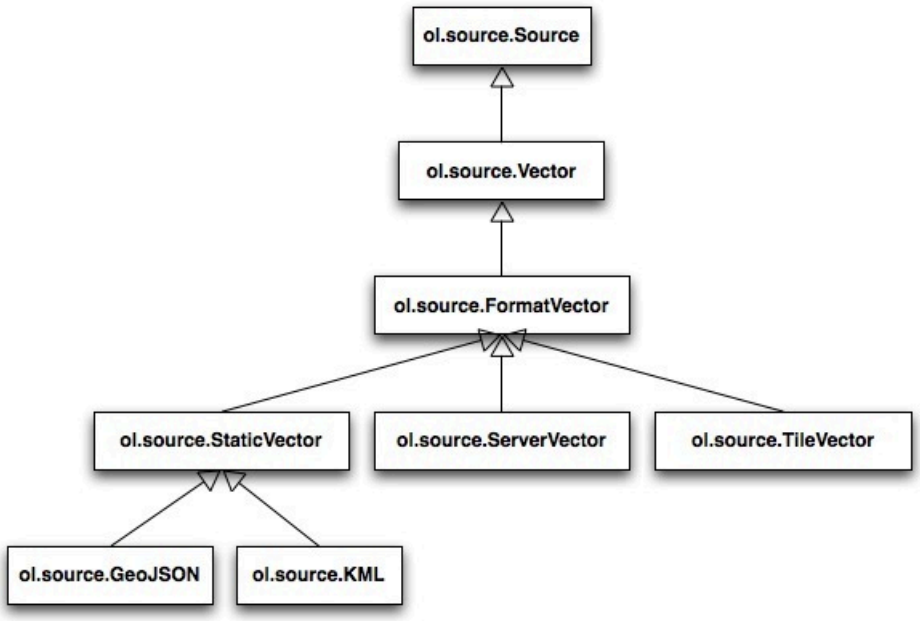
More formally, we can say the mission of a format class is to convert data from its representation format (GeoJSON, KML, ...) to a set of *features* and, vice versa, from a set of *features* to a data representation.

We will see *features* in depth in the Vector layers chapter, for the moment, think on them as a representation of a real world thing: a river, a city, a country, etc.

This way, the chain of responsibilities are as follows: the layer delegates the task of retrieving data to the source. The source communicates to a server, retrieves data and delegates the reading task to the format class. The format class reads the data and converts it to features ready to be rendered.

We can think on a **vector source as a container of features** that brings them to the layer (and the renderer) when necessary and has operations to add, get or remove features, get the extent covered by the set of features, etc.

Say all that, we introduce the next figure which shows most of the classes involved in the source vector hierarchy:



Vector source hierarchy

The main class is `ol.source.Vector` and defines most of the methods necessary to implement the previous commented actions, like: `addFeatures()`, `getFeatures()`, `removeFeature()`, `getExtent()`,

etc.

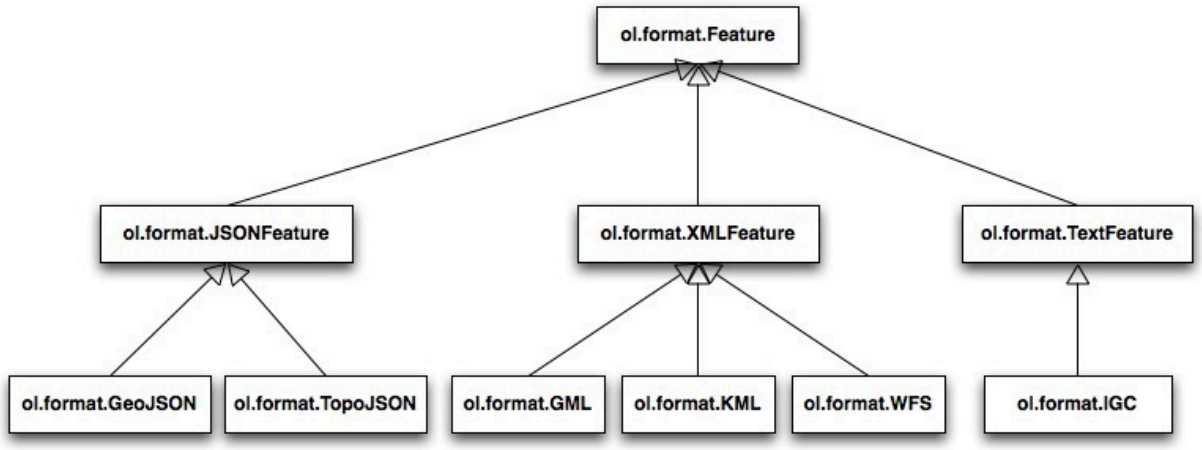
Previously, we have explained a vector source requires a format class to read a specific data format. Well, this functionality is introduced by the `ol.source.FormatVector` class, which allows to set a format instance to delegate the reading task.

Strictly speaking, it is not true a vector source class always requires a format class to read data. We can work directly with `ol.source.Vector` instances adding features programatically. See section Creating features by hand at Vector layers chapter.

The `ol.source.FormatVector` class is extended by three important subclasses: `ol.source.StaticVector`, `ol.source.ServerVector` and `ol.source.TileVector`. As we will see in the next sections, the main difference between `ol.source.StaticVector` and `ol.source.ServerVector` is number of times it allows to load data. On its way, the `ol.source.TileVector` is specially designed to work with those servers that can serve tiles in some vector data format.

On the other hand, looking at the format classes, we can see they conforms a hierarchy too. The base class `ol.format.Feature` defines a set of methods every subclass implements to allow read/write features from/to a data format. Within all the methods, probably the most important ones are `readFeatures()` and `writeFeatures()` that are responsibles to read/write features from/to the data source.

Next figure shows some of the classes involved in the format hierarchy:

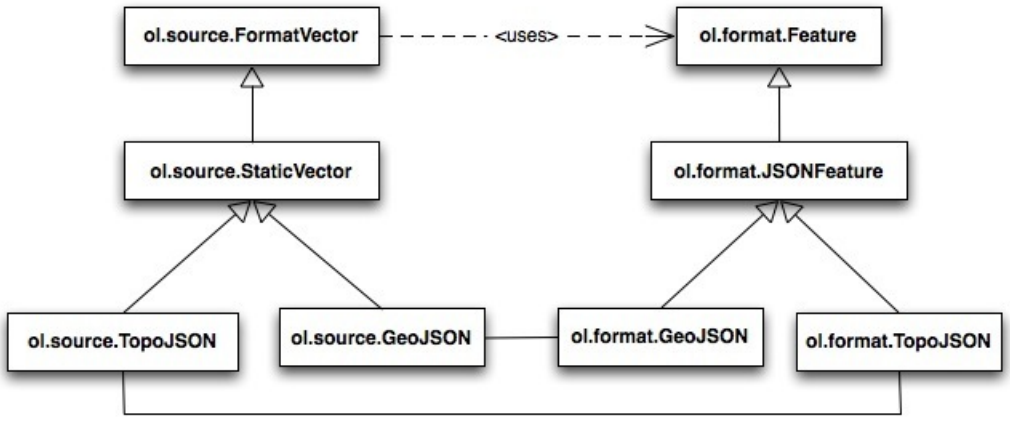


Format hierarchy

Note `ol.format.Feature` is an abstract class, which means we can not create instances of it directly but instantiating some of its subclasses.

Concrete format classes are what brings the real flexibility to OpenLayers3 to allow read a great degree of data formats.

Next figure show the relationship between `ol.source.FormatVector` and `ol.format.Feature`. What means is each concrete vector source class uses a concrete format class. For example, the `ol.source.GeoJSON` source class delegates the task to read data to a `ol.format.GeoJSON` instance.



Relation between sources and formats

3.3.2 Understanding the `StaticVector` based classes

The `ol.source.StaticVector` is designed to load data once, usually from an URL (or from a given variable). This is the typical behavior when reading, for example, a GeoJSON file, where we read the whole file and render the features on the map.

Although these source reads the whole set of features once, the renderer process is optimized to draw only the features visible in the map's view.

Usually, `ol.source.StaticVector` is not used directly, instead we use a concrete subclass for each supported data format, like `ol.source.GeoJSON`, `ol.source.TopoJSON`, `ol.source.GPX`, `ol.source.KML`, `ol.source.OSMXML`, etc.

Subclasses of `ol.source.StaticVector` supports different ways of loading data, depending on the property we set:

- `url`: an URL pointing to the file to be loaded.
- `urls`: an array of URLs pointing to the set of files to be loaded.
- `text`: a string representation of the information.
- `object`: a JavaScript object with the information.
- `doc`: a XML document reference containing data
- `node`: a XML node with the information to be loaded.

Not all properties are available in all subclasses. In classes that works with JSON data has sense to use `object` property passing a JavaScript object but no the `node` property. Similarly for XML based sources it has no sense to pass an `object` but it has to pass a `node` or `doc` property.

For example, to read a TopoJSON file it is enough with:

```
var vector = new ol.layer.Vector({
  source: new ol.source.TopoJSON({
    url: ... // URL to TopoJSON file
  })
});
```

or to read a KML file:

```
var vector = new ol.layer.Vector({
  source: new ol.source.KML({
    url: ... // URL to KML file
  })
});
```

As we commented, usually we work with concrete subclasses, but we can also achieve a similar result working directly with the `ol.source.StaticVector` class specifying the format to be used:

```
var vector = new ol.layer.Vector({
  source: new ol.source.StaticVector({
    format: new ol.format.KML(),
    url: ... // URL to KML file
  })
});
```

Remember like with any other source we can specify properties like `projection` or `extent`.

Next code shows how to create a vector layer from GeoJSON source passing data through the `text` or `object` properties:

```
    // Passing a string
    var vector = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            text: '{"type":"Feature","geometry":{"type":"Polygon","coordinates":\
[[[-5000000,-1000000],[-4000000,1000000],[-3000000,-1000000]]]}}'
        })
    });

    // Passing a JavaScript object
    var vector = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            object: {
                type: 'Feature',
                geometry: {
                    type: 'Polygon',
                    coordinates: [[[-5000000, -1000000], [-4000000, 2000000], [-\
3000000, -1000000]]]
                }
            }
        })
    });
```

3.3.3 Understanding the `ServerVector` class

At the opposite of `ol.source.StaticVector`, the `ol.source.ServerVector` class is prepared to load data multiple times if needed. These behavior is typically used when requesting a WFS server, where instead retrieving all the features from the server at a time, we query the server each time view extent changes, for example when panning, requesting only those features within the new bounding box.

The `ol.source.ServerVector` class has two main properties:

- `loader`: A function responsible to load the remote data and set the features served by the source to the layer. This function receives three arguments, `extent`, `resolution` and `projection`, to know what must be loaded.
- `strategy`: A function which determines when `loader` function is executed: once, each time bounding box changes, etc. The function receives two parameters, the current `extent` and `resolution`, and must return an array with the extents to be loaded.

Although we are allowed to implement our custom strategy function, OpenLayers3 offers the `ol.loadingstrategy` object, which contains some predefined strategies:

- `ol.loadingstrategy.all`: Load all the features (it is like make a request specifying an infinity bounding box).

- `ol.loadingstrategy.bbox`: This is the default strategy if none is specified. Load features based on the view's extent, so each time we change the view a request is made to update the features.
- `ol.loadingstrategy.createTile`: Similar to `bbox` strategy but it loads features based on a tile grid, so each time we change the zoom level or pan the view new features are loaded if new tiles require to be rendered.

Lets go to see an example to understand how to work with all the involved classes.

Suppose we want to load data from a WFS server in GeoJSON format. Next code, declares a vector source, which uses an `ol.format.GeoJSON` instance to translate data to features, uses a tile strategy to load from the WFS server and uses the `loaderFunction` (that we will see next) to make requests to the server:

```
var vectorSource = new ol.source.ServerVector({
    format: new ol.format.GeoJSON(),
    strategy: ol.loadingstrategy.createTile(new ol.tilegrid.XYZ({
        maxZoom: 19
    })),
    loader: loaderFunction,
    projection: 'EPSG:3857'
});
```

As we commented before, the `loaderFunction` receives three arguments: `extent`, `resolution` and `projection`.

In this example, the loader function creates a valid WFS request URL to get data in JSON format for the specified `extent` and using the JSONP technique so, once the data is received the `addFeatures()` function is called. Note, to load data we are using `$.ajax()` method from jQuery[21] library:

```
var loaderFunction = function(extent, resolution, projection) {
    var url = 'http://some.remote.wfs.server/?service=WFS&' +
        'version=1.1.0&request=GetFeature&typename=someName&' +
        'outputFormat=text/javascript&format_options=callback:addFeatures' +
        '&srsname=EPSG:3857&bbox=' + extent.join(',') + ',EPSG:3857';
    $.ajax({
        url: url,
        dataType: 'jsonp'
    });
};
```

Finally, the `addFeatures` function is executed when the loader function receives data and is responsible to transform data to feature instances and add them to the vector source:

[21] http://api.jquery.com/jquery.ajax/

```
    var addFeatures = function(response) {
        vectorSource.addFeatures(vectorSource.readFeatures(response));
    };
```

As we can see, and thanks to the loader function, the `ol.source.ServerVector` is a powerful class which offers us a great degree of flexibility to solve almost any situation where we need to load remote content in any format and strategy.

3.3.4 Loading vector tiles

Browsers performance has been improved considerably in recent years and, thanks to this, they are becoming a new great tool to render vector information.

Until now, many information (like roads, rivers, countries, etc) where rendered on server side as raster images and served to the clients. The tile providers, or WMS servers, does the hard work generating images at different zoom levels, ready to be used in OpenLayers3 with the `ol.source.TileWMS` class. Following the same concept, we can generate tiles at different zoom levels but in vector data format instead as of raster.

Hopefully, OpenLayers3 offers the `ol.source.TileVector` class, which has been designed to allow requesting vector tiles.

As any other source instance, we can specify the properties `url` or `urls`, to indicate where to load the data, the target `projection` to transform the data, etc. As a subclass of `ol.source.FormatVector`, we can specify a format instance in the `format` property, allowing us to read any kind of vector data.

In addition to the inherited properties, the `ol.source.TileVector` offers some more specially required to work with tiles:

- `tileGrid`, which determines the tiles the source will be query (see the Tile grids section),
- `tileUrlFunction`, a function that will be responsible to create the URLs for each tile request.

Next code, show a sample tile vector source configured to request vector tiles from OpenStreetMap project:

```
var tileVector = new ol.source.TileVector({
    format: new ol.format.TopoJSON({
        defaultProjection: 'EPSG:4326'
    }),
    projection: 'EPSG:3857',
    tileGrid: new ol.tilegrid.XYZ({
        maxZoom: 19
    }),
    url: 'http://{a-c}.tile.openstreetmap.us/vectiles-highroad/{z}/{x}/{y}.topoj\
son'
});
```

Because data is in TopoJSON format we have set a `ol.format.TopoJSON` instance, indicating the data is in `EPSG:4326` projection and the target projection to translate data is `EPSG:3857`. We have specified an XYZ tile grid with 19 zoom levels and the request URLs will follow the template specified at `url` property.

We can use the `ol.source.OSMXML` class too, which as any other `ol.source.StaticVector` subclass is designed to make a single request to the OpenStreetMap servers requesting data in OSMXML vector format.

3.3.5 Be aware with the *Same Domain Policy*

Each time a JavaScript code needs to load a remote file it creates a XMLHttpRequest[22] (XHR) instance, which makes the browser open a connection to the server, load data and make the response available to the JavaScript code.

For security reasons, these action is subject to the Same Domain Policy[23], that is, if you load a web page that contains JavaScript code from a domain A, the code can not make XHR requests to a domain B.

Supposing our application resides at `my_domain` and we load a web page from the URL `http://my_domain/my_app` that contains the next JavaScript code:

[22] http://en.wikipedia.org/wiki/XMLHttpRequest
[23] http://en.wikipedia.org/wiki/Same-origin_policy

```
    var vector = new ol.layer.Vector({
      source: new ol.source.KML({
        url: `http://another_domain/some_kml_file`
      })
    });
```

that will raise the next error (or similar) in the browser's console:

```
XMLHttpRequest cannot load http://another_domain/some_kml_file. No 'Access-Contr\
ol-Allow-Origin' header is present on the requested resource. Origin 'http://my_\
domain/my_app' is therefore not allowed access.
```

Hopefully, new browsers implements the Cross-origin resource sharing[24] (CORS) mechanism that allows to solve this limitation, letting the client (the browser) and the server interact and determine whether or not to allow cross origin requests.

You can find more information at Working with the JavaScript XMLHttpRequest object[25].

Because the *Same Domain Policy* is a limitation of older browsers and the CORS mechanism requires some server configuration, sometimes is a common solution make the requests through a proxy installed in our server side that avoids violate the *Same Domain Policy*.

3.3.6 Rendering vector data as raster

The process of rendering vector layers is complex and can be expensive in performance sense. It involves draw points, lines, polygons, images, etc, usually each time the map is modified, that is, when new information is added, when map is panned, when the zoom changes, ...

To improve the vector layers rendering experience, Openlayers3 offers the `ol.source.ImageVector` source class, which allows to render a vector layer in the same way if it were a raster layer.

This source class can be used within an `ol.layer.Image` and will serve the information as an image instead as a list of features. The user experience can be improved with this source class because image layers are rotated, scaled and translated during animations and interactions, as opposed to be re-rendered which is the normal behavior in vector layers.

The way `ol.source.ImageVector` works is wrapping a vector source, rendering data as a raster and offering it to an image layer. Next, code shows how to do it:

[24] http://en.wikipedia.org/wiki/Cross-origin_resource_sharing

[25] http://acuriousanimal.com/blog/2011/01/27/working-with-the-javascript-xmlhttprequest-object

```
    // The initial source with the features
    var vectorSource = new ol.source.Vector({
        text: ...
    });

    // The image source wraps the previous source
    var imageSource = new ol.source.ImageVector({
      source: vectorSource
    });

    // Create the image layers using the image source
    var imageLayer = new ol.layer.Image({
        source: imageSource
    });
```

3.3.7 Working with *format* classes

As we have explained previously, the *layer* delegates the task to read content to a *source* instance, which is responsible to retrieve the data from an origin and, in the case of vector sources, it delegates the task of converting the data to *feature* instances to a *format* class.

Format classes are mainly used within source classes but in some situations it can be interesting to use them directly to read or write features. So, it is important to understand how we can make use of them.

Given a string with a GeoJSON representation of a feature, next code reads it and returns a *feature* instance and, viceversa, writes the feature instance in GeoJSON string format:

```
    var geoJsonString = '{"type":"Feature","geometry":{"type":"Polygon","coordin\
ates":[[[-5000000,-1000000],[-4000000,1000000],[-3000000,-1000000]]]}}';

    // Create format instance
    var format = new ol.format.GeoJSON();

    // Obtain a feature instance from string
    var feature = format.readFeature(geoJsonString);

    // Convert feature to string representation
    var stringFeature = format.writeFeature(feature);
```

3.4 The practice

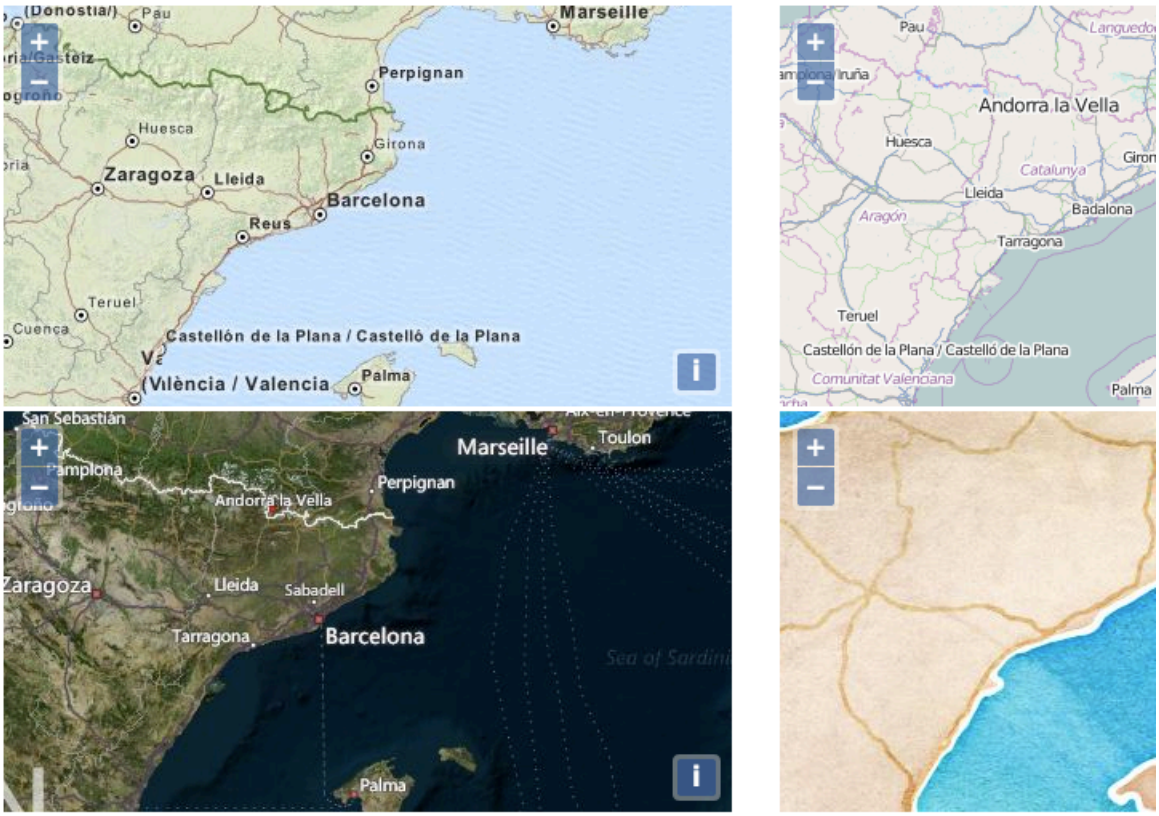
All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[26] repository.

3.4.1 Tile providers

3.4.1.1 Goal

This example demonstrate the use of various sources providing access to different tile provides as Bing, OpenStreetMap, MapQuest or Stamen.



Tile providers sample

3.4.1.2 How to do it...

We are going to create four maps, each one showing a layer from a different providers, so we can see how they looks for the same extent.

[26] https://github.com/acanimal

First step is to create the needed HTML for the four maps:

```
<div class="row">
    <div class="col-md-6"><div id="mapMQ" class="map"></div></div>
    <div class="col-md-6"><div id="mapOSM" class="map"></div></div>
</div>
<div class="row">
    <div class="col-md-6"><div id="mapBing" class="map"></div></div>
    <div class="col-md-6"><div id="mapStamen" class="map"></div></div>
</div>
```

Now we can add the JavaScript code responsible to create the layers and maps:

```
// Create a view object shared by the four maps.
var view = new ol.View({
    center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
    zoom: 8
});

var mapMQ = new ol.Map({
    target: 'mapMQ',
    renderer: 'canvas',
    layers: [
        new ol.layer.Tile({
            source: new ol.source.MapQuest({
                layer: 'osm'
            })
        })
    ],
    view: view
});

var mapOSM = new ol.Map({
    target: 'mapOSM',
    renderer: 'canvas',
    layers: [
        new ol.layer.Tile({
            source: new ol.source.OSM()
        })
    ],
    view: view
});
```

```
30
31      var mapStamen = new ol.Map({
32          target: 'mapStamen',
33          renderer: 'canvas',
34          layers: [
35              new ol.layer.Tile({
36                  source: new ol.source.Stamen({
37                      layer: 'watercolor'
38                  })
39              })
40          ],
41          view: view
42      });
43
44      var mapBing = new ol.Map({
45          target: 'mapBing',
46          renderer: 'canvas',
47          layers: [
48              new ol.layer.Tile({
49                  source: new ol.source.BingMaps({
50                      key: 'Ak-dzM4wZjSqTlzveKz5u0d4IQ4bRzVI309GxmkgSVr1ewS6iPSrOv\
51  OKhA-CJlm3',
52                      imagerySet: 'AerialWithLabels'
53                  })
54              })
55          ],
56          view: view
57      });
```

3.4.1.3 How it works...

The JavaScript code is relatively simple, it creates four maps attached to a target DOM element and using a `ol.layer.Tile` layer loading data from a different source provider.

Note, we have created a view instances used in all four maps. These produces the nice effect of changing the map location and zoom on the four maps when a change is made. Although initial strange, these behavior is completely logical. When we drag in a map the attached view instance is modified and, because this instance is shared among the four maps, all maps are modifies, that is, a change in the view produces an update in all the maps.

3.4.2 Reading WMS capabilities

3.4.2.1 Goal

Before to query a WMS server it is necessary to know its capabilities, which layers it can server, which styles, supported projections, etc.

The goal of this recipe is to demonstrate how easy we can get read the WMS server capabilities.

Capabilities are simply information provided by the server, so there is no data to be shown in a map. The returned information will be printed in the browser console in addition in a *text zone* within the web page.

3.4.2.2 How to do it...

First we are going to create a `<code>` element where to place the server response:

```
<pre><code id="result" class="json"></code></pre>
```

Now, lets place the JavaScript that loads the capabilities data and converts it to a JavaScript object:

```
$.get('data/MTN-Raster.xml')
.done(function(data) {
    var capabilities = new ol.format.WMSCapabilities();
    var result = capabilities.read(data);
    console.log(result);

    $('#result').text( JSON.stringify(result) );
});
```

3.4.2.3 How it works...

It is important to note OpenLayers3 does not offers a way to retrieve the server capabilities. It is our responsibility to retrieve it request, typically an URL like: `http://some_server?SERVICE=WMS&REQUEST=GetCapabili`

To avoid same origin policy[27] issues we are loading a XML file with some server capabilities we previously requested.

To get the server capabilities we are using the `$.get()` method, which queries the given a URL using AJAX mechanism. If request is successful the done method is executed passing the data:

[27] http://en.wikipedia.org/wiki/Same-origin_policy

```
    $.get('data/MTN-Raster.xml')
    .done(function(data) {
        ...
    });
```

It is within the `done()` method where we use the `ol.format.WMSCapabilities` to convert the returned XML document into a JavaScript object easiest to read and extract property values:

```
    var capabilities = new ol.format.WMSCapabilities();
    var result = capabilities.read(data);
    console.log(result);
    $('#result').text( JSON.stringify(result) );
```

Finally, we show the `result` object in the browser console and also convert it to a string ready to be place in the `<code>` element, previously created, to see the capabilities in the web page.

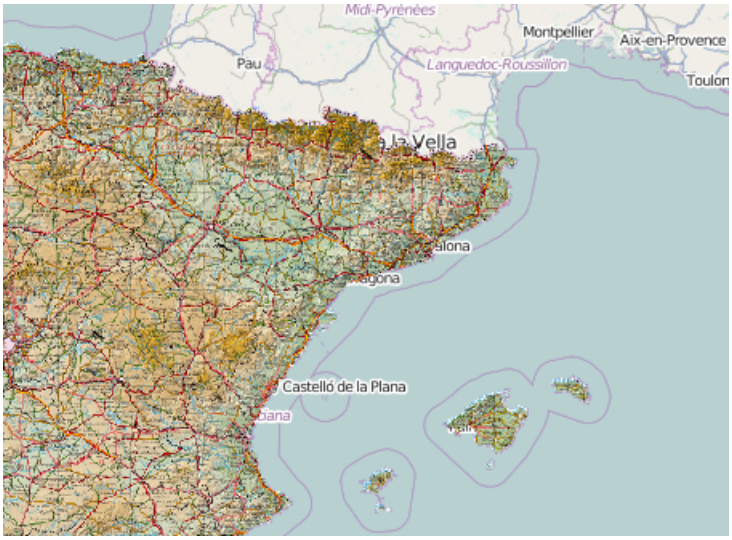
3.4.2.4 See also

- Loading data from a WMS server example at Data sources and formats chapter, to see how we can access WMS server data.
- Requesting data from a WFS server with and without JSONP example at Data sources and formats chapter, to see how you can request data via JSONP technique using jQuery.

3.4.3 Loading data from a WMS server

3.4.3.1 Goal

In this recipe we will show how we can load raster data from a WMS server. The example will show two maps. The map on the left works loading a single image for the whole view extent. The right map works in a tiled mode, requesting raster data as a pyramid of tiles.



Data from WMS server

3.4.3.2 How to do it...

First lest go to create the HTML with the element to hold the maps:

```
    <div class="row">
        <div id="mapImage" class="map col-sm-6"></div>
        <div id="mapTiles" class="map col-sm-6"></div>
    </div>
```

Now add the code for the left map, the one we request raster data as a single image:

```
    var mapImage = new ol.Map({
        target: 'mapImage',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            new ol.layer.Image({
                source: new ol.source.ImageWMS({
                    url: 'http://www.idee.es/wms/MTN-Raster/MTN-Raster',
                    ratio: 1,
                    params: {
                        'LAYERS': 'mtn_rasterizado',
                        'TRANSPARENT': 'true'
                    }
```

```
                })
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

Finally, add the code for the right map, which works in a tiled mode:

```
    var mapTiles = new ol.Map({
        target: 'mapTiles',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            new ol.layer.Tile({
                source: new ol.source.TileWMS({
                    url: 'http://www.idee.es/wms/MTN-Raster/MTN-Raster',
                    params: {
                        'LAYERS': 'mtn_rasterizado',
                        'TRANSPARENT': 'true'
                    }
                })
            })
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        })
    });
```

3.4.3.3 How it works...

The secret on the different way of working between the maps resides in the layer and source classes used on each one.

For the left map, we have used a `ol.layer.Image` layer that loads data through a `ol.source.ImageWMS` source instance. In addition to the server `url`, we pass an object with the WMS parameters `LAYERS` and `TRANSPARENT`. The first indicates the list of layers to request and the second indicates the server generate transparent images on those pixels without data.

Each time the map's view extent changes, the `ol.source.ImageWMS` source makes a new request to the WMS passing a `BBOX` parameter with the view extent modified given the `ratio` parameter. For example, for a `ratio=1` the `BBOX` passed is equal to the view's extent. For a `ratio=2` the BBOX is double of the view's extent. Requesting an image greater than the view extent can avoid making new requests when the map is panned.

```
    new ol.layer.Image({
        source: new ol.source.ImageWMS({
            url: 'http://www.idee.es/wms/MTN-Raster/MTN-Raster',
            ratio: 1,
            params: {
                'LAYERS': 'mtn_rasterizado',
                'TRANSPARENT': 'true'
            }
        })
    })
```

For the left map, we have used a `ol.layer.Tile` layer loading data through a `ol.source.TileWMS` source. This source uses internally a tile grid that conforms a pyramid of tiles used to request the server. This way, each time we change the map's view panning or zooming, the source computes the tiles visible on the map and makes a request to the WMS server passing the bounding box of each tile as the `BBOX` parameter.

```
    new ol.layer.Tile({
        source: new ol.source.TileWMS({
            url: 'http://www.idee.es/wms/MTN-Raster/MTN-Raster',
            params: {
                'LAYERS': 'mtn_rasterizado',
                'TRANSPARENT': 'true'
            }
        })
    })
```

You will probably notice the right map loads faster than the left one. The advantage of requesting WMS servers in a tiled mode is many servers caches the request. Because the tiles requests has always the same `BBOX`, they are easiest to cache than single image request (where its `BBOX` change slightly on each map change).

3.4.3.4 See also

- Reading WMS capabilities example at Data sources and formats chapter, to see how we can read the available WMS server capabilities.

3.4.4 Requesting WMTS server

3.4.4.1 Goal

This example shows how we can request tiles from a WMTS compliant server.

WMTS access

At the time of writing there is no way to read the WMTS server capabilities, so it is our responsibility to know the layer names, styles, tile matrix sets, etc.

3.4.4.2 How to do it...

Before to request a WMTS server we need to know the layer name, style, ... plus the available tile matrix sets and resolutions. Because of this the first step requires we initialize some variables with the right values:

```
    var projection = ol.proj.get('EPSG:3857');
    var projectionExtent = projection.getExtent();
    var size = ol.extent.getWidth(projectionExtent) / 256;
    // Generate and array of resolutions and matrixIds for this WMTS
    var resolutions = new Array(18);
    var matrixIds = new Array(18);
    for (var z = 0; z < 18; ++z) {
        resolutions[z] = size / Math.pow(2, z);
        matrixIds[z] = z;
    }
```

With previous values we can now to create the map with a `ol.layer.Tile` layer that loads content through a `ol.source.WMTS` source:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            // Add a WMTS layer
            new ol.layer.Tile({
                opacity: 0.5,
                extent: projectionExtent,
                source: new ol.source.WMTS({
                    url: 'http://demo-apollo.geospatial.intergraph.com/erdas-iws\
/ogc/wmts/',
                    layer: 'sampleiws_images_geodetic_worldgeodemo.ecw',
                    matrixSet: 'ogc:1.0:googlemapscompatible',
                    format: 'image/jpeg',
                    projection: projection,
                    tileGrid: new ol.tilegrid.WMTS({
                        origin: ol.extent.getTopLeft(projectionExtent),
                        resolutions: resolutions,
                        matrixIds: matrixIds
                    }),
                    extent: projectionExtent,
                    style: 'default'
                })
            })
```

```
        ],
        // The view to be used to show the map is a 2D
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 2
        })
    });
```

3.4.4.3 How it works...

The previous map contains two layers, the OpenStreetMap acts as the base layer and while the tiled layer with WMTS data is placed above it with some transparency:

```
    ...
    layers: [
        // Add a new Tile layer getting tiles from OpenStreetMap source
        new ol.layer.Tile({
            source: new ol.source.OSM()
        }),
        // Add a WMTS layer
        new ol.layer.Tile({
            opacity: 0.5,
            extent: projectionExtent,
            source: ...
        })
    ],
    ...
```

To request tiles from the WMTS server, the source requires we set the right properties. Most important are:

- `url`, with the server URL to be queries,
- `layer`, with the layer name,
- `style`, with the style of the layer
- `matrixSet`, with the identifier of the matrix set to be requested,
- `tileGrid`, with a `ol.tilegrid.WMTS` instance configured with the right matrix identifiers and resolutions,

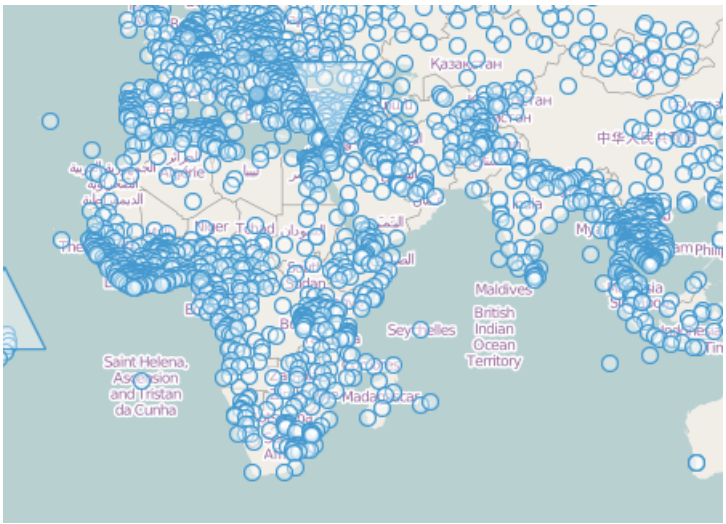
```
    ...
    source: new ol.source.WMTS({
        url: 'http://demo-apollo.geospatial.intergraph.com/erdas-iws/ogc/wmts/',
        layer: 'sampleiws_images_geodetic_worldgeodemo.ecw',
        matrixSet: 'ogc:1.0:googlemapscompatible',
        format: 'image/jpeg',
        projection: projection,
        tileGrid: new ol.tilegrid.WMTS({
            origin: ol.extent.getTopLeft(projectionExtent),
            resolutions: resolutions,
            matrixIds: matrixIds
        }),
        extent: projectionExtent,
        style: 'default'
    })
    ...
```

Thanks to these information, OpenLayers3 generates the right URL, including parameters like `TileMatrix`, `TileCol` and `TileRow`, used to request the WMTS server.

3.4.5 Different ways to load data using a vector source

3.4.5.1 Goal

The goal of this recipe is to show the flexibility of source classes, like the `ol.source.GeoJSON`, which allows to load content in different ways.



Vector source

3.4.5.2 How to do it...

As always create a `div` element to hold the map:

```
    <div id="map" class="map"></div>
```

Now add the JavaScript code necessary to create the map, create some `ol.layer.Vector` layers that will load information from a `ol.source.GeoJSON` source using different approaches:

```
    // Vector layer from a GeoJson file url
    var gjsonFile = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            url: 'data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });

    // Vector layer from a geojson object
    var gjsonObject = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            object: {
                'type': 'Feature',
                'geometry': {
                    'type': 'Polygon',
                    'coordinates': [[[3000000, 6000000], [4000000, 4000000], [50\
00000, 6000000]]]
                }
            }
        })
    });

    // Vector layer from a geojson string
    var gjsonString = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            text: '{"type":"Feature","geometry":{"type":"Polygon","coordinates":\
[[[-5000000,-1000000],[-4000000,1000000],[-3000000,-1000000]]]}}'
        })
    });

    // NOTE: This layer is made to force a cross domain error.
    // We will get an error similar to:
    //   XMLHttpRequest cannot load http://ol3js.org/en/master/examples/data/top\
```

```
ojson/world-110m.json. No 'Access-Control-Allow-Origin' header is present on the\
 requested resource. Origin 'http://localhost:9000' is therefore not allowed acc\
ess.
    var topoJson = new ol.layer.Vector({
        source: new ol.source.TopoJSON({
            projection: 'EPSG:3857',
            url: 'http://ol3js.org/en/master/examples/data/topojson/world-110m.j\
son'
        })
    });

    // Create the map
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            gjsonFile,
            gjsonString,
            gjsonObject,
            topoJson
        ],
        view: new ol.View({
            center: ol.proj.transform([0, 0], 'EPSG:4326', 'EPSG:3857'),
            zoom: 2
        })
    });
```

3.4.5.3 How it works...

We have created the map as usual, attaching a OpenStreetMap layer acting as base layer plus the set of vector layers previously initialized.

First, the `gjsonFile` layer shows, probably, the most common way to use a source class. It is a vector layer initialized to load data from a GeoJSON file specified with some url:

```
    // Vector layer from a GeoJson file url
    var gjsonFile = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            url: 'data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });
```

In addition from an URL, the `ol.source.GeoJSON` can also read data from an object or a string. This is the purpose of `gjsonObject` and `gjsonString` layers:

```
    // Vector layer from a geojson object
    var gjsonObject = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            object: {
                'type': 'Feature',
                'geometry': {
                    'type': 'Polygon',
                    'coordinates': [[[3000000, 6000000], [4000000, 4000000], [50\
00000, 6000000]]]
                }
            }
        })
    });

    // Vector layer from a geojson string
    var gjsonString = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            text: '{"type":"Feature","geometry":{"type":"Polygon","coordinates":\
[[[-5000000,-1000000],[-4000000,1000000],[-3000000,-1000000]]]}}'
        })
    });
```

3.4.5.4 There is more...

We have created a fourth layer which loads TopoJSON data from an URL which is never shown because the request fails.

This is intended to see a typical cross domain error in the browser console. Because the samples runs on a server different than the server we are requesting the data, for security reasons and following the *same domain policy*, the browser aborts the connection if the remote server does not give us permissions.

3.4.5.5 See also

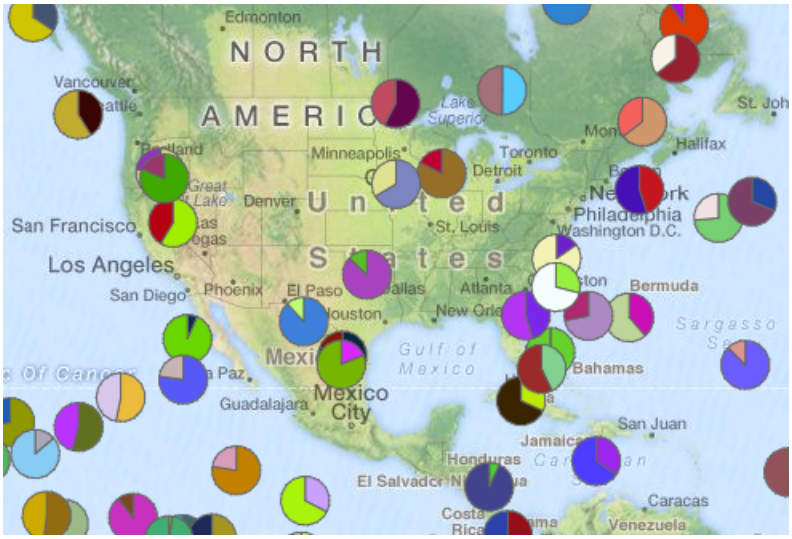
- Creating features programmatically example at Vector layers chapter, to see how we can create features programmatically instead loading them through a source instance.

3.4.6 Working with ImageCanvas

3.4.6.1 Goal

The goal of this examples is to demonstrate the flexibility of the `ol.source.ImageCanvas` source, allowing us to render almost anything, which is an easy way to extend OpenLayers3 functionalities.

For this purpose, we are going to create a bunch of random pie charts to demonstrate how we can work with the canvas and geo locate items within the map.



ImageCanvas example

3.4.6.2 How to do it...

Create a `div` element to hold the map:

```
1     <div id="map" class="map"></div>
```

Create the map, setting the view to the desires location and add a tile layer that will act as base layer:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([0, 30], 'EPSG:4326', 'EPSG:3857'),
            zoom: 3
        })
    });
```

Know add an `ol.layer.Image` layer that obtains the data from an `ol.source.ImageCanvas` source.

```
    var canvasLayer = new ol.layer.Image({
        source: new ol.source.ImageCanvas({
            canvasFunction: canvasFunction, // The function responsible to rende\
r data
            projection: 'EPSG:3857'
        })
    });

    map.addLayer(canvasLayer);
```

Finally, we need to define the `canvasFunction` function we have set at the `canvasFunction` property. To avoid extending unnecessarily this section we will explain the code in the next one.

3.4.6.3 How it works...

All the magic of this example resides in the `canvasFunction` function we pass to the `ol.source.ImageCanvas` source, which is responsible to render the desired data in the desired place and in the desired way.

Our pie charts are very simple, they are formed by two wedges with random percent values and colors. Because this, the first step before rendering the pies is to create some array of values with the coordinates of each pie, its percent values and colors:

```
    var numPieCharts = 750, coordinates=[], data=[], colors=[];
    var i, p;
    for(i=0; i< numPieCharts; i++) {
        coordinates.push([-180+360*Math.random(), -90+180*Math.random()]);
        p = 100*Math.random();
        data.push([p, 100-p]);
        colors.push([
            '#'+(0x1000000+(Math.random())*0xffffff).toString(16).substr(1,6),
            '#'+(0x1000000+(Math.random())*0xffffff).toString(16).substr(1,6)]);
    }
```

Note, the pie coordinates are EPSG:4326 while the map's view is in EPSG:3857.

Now that we have the data needed to be rendered by the `canvasFunction` lets go to define it. The function receives five parameters needed to know the map status at rendering time:

```
    var canvasFunction = function(extent, resolution, pixelRatio, size, projecti\
on) {
```

The function must return a `<canvas>` element, so the first step is to create the canvas element and set the right size:

```
    var canvas = document.createElement('canvas');
    var context = canvas.getContext('2d');
    var canvasWidth = size[0], canvasHeight = size[1];
    canvas.setAttribute('width', canvasWidth);
    canvas.setAttribute('height', canvasHeight);
```

For performance reason, the `size` passed to the canvas function is usually greater than the current size of the map. This way, we can draw elements currently outside the map and when move it slightly there is no need to render again.

Next code computes the offset between the origin (top-left) of the canvas element and the map:

```
    // Canvas extent is different than map extent, so compute delta between
    // left-top of map and canvas extent.
    var mapExtent = map.getView().calculateExtent(map.getSize())
    var canvasOrigin = map.getPixelFromCoordinate([extent[0], extent[3]]);
    var mapOrigin = map.getPixelFromCoordinate([mapExtent[0], mapExtent[3]]);
    var delta = [mapOrigin[0]-canvasOrigin[0], mapOrigin[1]-canvasOrigin[1]]
```

Later, we define two functions. First `drawWedge` which knows how to draw a pie wedge and later `drawPie` that using the previous function draws a pie charts.

```
    var radius = 15;

    // Track the accumulated arcs drawn
    var totalArc = -90*Math.PI / 180;
    var percentToRadians = 1 / 100*360 *Math.PI / 180;
    var wedgeRadians;

    function drawWedge(coordinate, percent, color) {

        var point = ol.proj.transform(coordinate, 'EPSG:4326', 'EPSG:3857');
        var pixel = map.getPixelFromCoordinate(point);
        var cX = pixel[0] + delta[0], cY = pixel[1] + delta[1];

        // Compute size of the wedge in radians
        wedgeRadians = percent * percentToRadians;

        // Draw
        context.save();
        context.beginPath();
        context.moveTo(cX, cY);
        context.arc(cX, cY, radius, totalArc, totalArc + wedgeRadians, false);
        context.closePath();
        context.fillStyle = color;
        context.fill();
        context.lineWidth = 1;
        context.strokeStyle = '#666666';
        context.stroke();
        context.restore();

        // Accumulate the size of wedges
        totalArc += wedgeRadians;
    }
```

```
33
34      var drawPie = function(coordinate, data, colors) {
35          for(var i=0;i<data.length;i++){
36              drawWedge(coordinate, data[i],colors[i]);
37          }
38      }
```

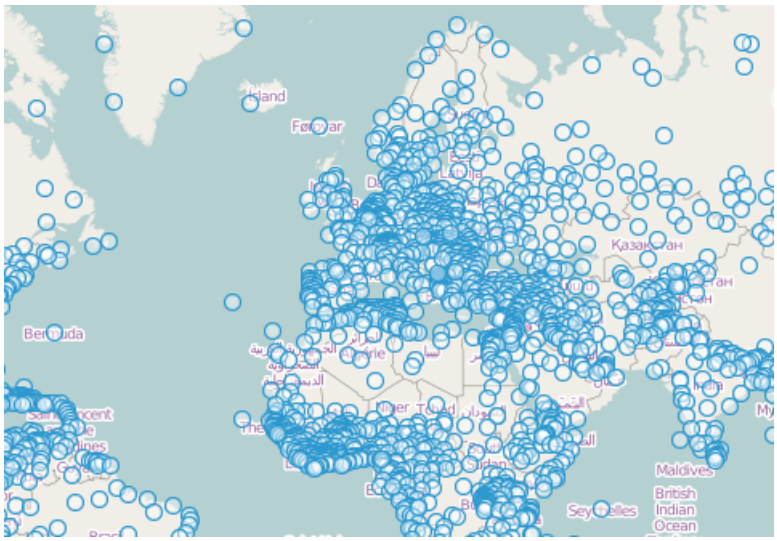
Last step is to create a loop to create a pie chart for each previously created coordinate, percent values and colors. Finally, we return a reference to the `canvas` element:

```
1       for(var i=0; i<coordinates.length; i++) {
2           drawPie(coordinates[i], data[i], colors[i]);
3       }
4
5       return canvas;
```

3.4.7 Rendering vector data as raster

3.4.7.1 Goal

In this examples we are going to see how we can render vector data as a raster image.



Rendering vector data as raster

This can be a great option for situations in which we have vector files with a big number of features, because the time needed to render features using vector primitives like lines, circles, etc is greater than the time required to render a raster.

3.4.7.2 How to do it...

We are going to create two maps. On the left we are going to add a vector layer while in the right side we are going to add an image layer, both loading data from a GeoJSON file.

The first step is to put the HTML code for the two maps:

```
    <div class="row">
        <div id="mapVector" class="map col-sm-6"></div>
        <div id="mapImage" class="map col-sm-6"></div>
    </div>
```

Both maps will use an OpenStreetMap as the base layer so we store a reference:

```
    // OSM tile layer
    var layerOSM = new ol.layer.Tile({
        source: new ol.source.OSM()
    });
```

Next create the left map using a vector layer:

```
    // Vector layer from GeoJSON source
    var layerVector = new ol.layer.Vector({
        source: new ol.source.GeoJSON({
            url: 'data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });

    // Map showing vector layer
    var map = new ol.Map({
        target: 'mapVector',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [layerOSM, layerVector],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 2
        })
    });
```

And finally the right map:

```
    // Image layer from a GeoJSON source using the ImageVector wrapper class.
    var layerImage = new ol.layer.Image({
        source: new ol.source.ImageVector({
            source: new ol.source.GeoJSON({
                url: 'data/world_cities.json',
                projection: 'EPSG:3857'
            })
        })
    });

    // Map showing raster vector layer
    var map = new ol.Map({
        target: 'mapImage',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [layerOSM, layerImage],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 2
        })
    });
```

3.4.7.3 How it works...

The `ol.source.ImageVector` source acts as wrapper for other vector sources. Internally it renders the vector data within a canvas element so it will be suitable to be added on an `ol.layer.Image` layer:

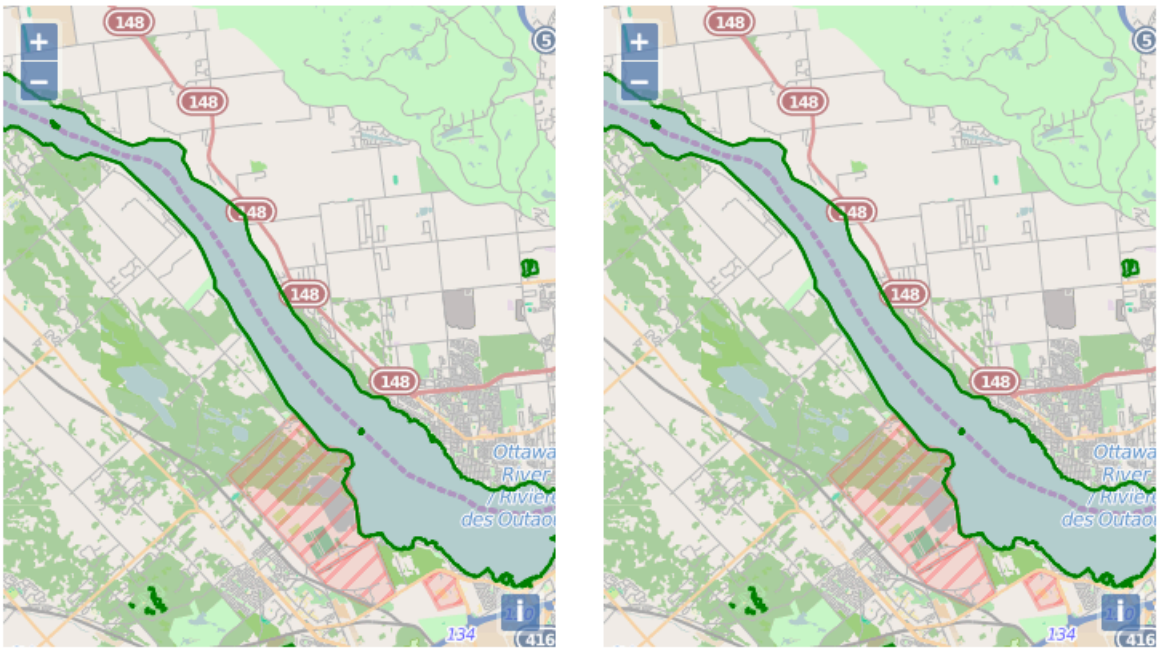
```
    var layerImage = new ol.layer.Image({
        source: new ol.source.ImageVector({
            source: new ol.source.GeoJSON({
                url: 'data/world_cities.json',
                projection: 'EPSG:3857'
            })
        })
    });
```

3.4.8 Requesting data from a WFS server with and without JSONP

3.4.8.1 Goal

In this recipe we are going to show how we can load data from a WFS server both using AJAX and JSONP techniques. For this purpose, we are going to create two maps. On the left side the map will

load vector data using normal AJAX call while the right one loads data using the JSONP technique.



Loading from WFS server

We are calling a server that accepts Cross-Origin Resource Sharing (CORS), because of this we can make AJAX requests without the same origin policy issues.

We are going to see how to use the `ol.source.ServerVector` source class, which allows to query any kind of server using different kind of strategies (see [Understanding the `ServerVector` class]) section at Data sources and formats chapter for more information about strategies).

3.4.8.2 How to do it...

Start adding the HTML needed to create the two maps side by side:

```
1     <div class="row">
2         <div id="map" class="map col-sm-6"></div>
3         <div id="mapJsonp" class="map col-sm-6"></div>
4     </div>
```

Both maps will use a tiled layer with OpenStreetMap tiles, so we create it:

```
    var osmLayer = new ol.layer.Tile({
        source: new ol.source.OSM()
    });
```

Now, we well create the map that loads vector data using AJAX call. We need to create the source, a layer and finally the map:

```
    // Source retrieving WFS data in GML format using AJAX
    var vectorSource = new ol.source.ServerVector({
        format: new ol.format.WFS({
            featureNS: 'http://openstreemap.org',
            featureType: 'water_areas'
        }),
        loader: function(extent, resolution, projection) {
            var url = 'http://demo.opengeo.org/geoserver/wfs?'+
                'service=WFS&request=GetFeature&'+
                'version=1.1.0&typename=osm:water_areas&'+
                'srsname=EPSG:3857&'+
                'bbox=' + extent.join(',');

            $.ajax({
                url: url
            })
            .done(function(response) {
                vectorSource.addFeatures(vectorSource.readFeatures(response));
            });
        },
        strategy: ol.loadingstrategy.createTile(new ol.tilegrid.XYZ({
            maxZoom: 19
        })),
        projection: 'EPSG:3857'
    });

    // Vector layer
    var vectorLayer = new ol.layer.Vector({
        source: vectorSource,
        style: new ol.style.Style({
            stroke: new ol.style.Stroke({
                color: 'green',
                width: 2
            })
        })
```

```
    });

    // Map
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        layers: [osmLayer, vectorLayer],
        view: new ol.View({
            center: ol.proj.transform([-75.923853, 45.428736], 'EPSG:4326', 'EPS\
G:3857'),
            maxZoom: 19,
            zoom: 11
        })
    });
```

Finally, create the map on the right side, the one that loads using JSONP technique:

```
    // Source retrieving WFS data in GeoJSON format using JSONP technique
    var vectorSourceJsonp = new ol.source.ServerVector({
        format: new ol.format.GeoJSON(),
        loader: function(extent, resolution, projection) {
            var url = 'http://demo.opengeo.org/geoserver/wfs?'+
                'service=WFS&request=GetFeature&'+
                'version=1.1.0&typename=osm:water_areas&'+
                'outputFormat=text/javascript&'+
                'format_options=callback:loadFeatures&' +
                'srsname=EPSG:3857&'+
                'bbox=' + extent.join(',');

            $.ajax({
                url: url,
                dataType: 'jsonp'
            });
        },
        strategy: ol.loadingstrategy.createTile(new ol.tilegrid.XYZ({
            maxZoom: 19
        })),
        projection: 'EPSG:3857'
    });

    // Executed when data is loaded by the $.ajax method.
    var loadFeatures = function(response) {
```

```
26          vectorSourceJsonp.addFeatures(vectorSourceJsonp.readFeatures(response));
27      };
28
29      // Vector layer
30      var vectorLayerJsonp = new ol.layer.Vector({
31          source: vectorSourceJsonp,
32          style: new ol.style.Style({
33              stroke: new ol.style.Stroke({
34                  color: 'green',
35                  width: 2
36              })
37          })
38      });
39
40      // Map
41      var mapJsonp = new ol.Map({
42          target: 'mapJsonp',
43          renderer: 'canvas',
44          layers: [osmLayer, vectorLayerJsonp],
45          view: new ol.View({
46              center: ol.proj.transform([-75.923853, 45.428736], 'EPSG:4326', 'EPS\
47  G:3857'),
48              maxZoom: 19,
49              zoom: 11
50          })
51      });
```

3.4.8.3 How it works...

Creating the map and layer has not much secrets at this point of the book. So we will obviate them. See previous chapters to see examples on that topics.

Lets go to describe the common code used for the sources to request and read the vector data. In both maps, the sources instance requires we set the `format`, `loader`, `strategy` and `projection` properties:

```
1       // Source retrieving WFS data in GML format using AJAX
2       var vectorSource = new ol.source.ServerVector({
3           format: ... ,
4           loader: ... ,
5           strategy: ... ,
6           projection: 'EPSG:3857'
7       });
```

As the `strategy` we have specify a function that determines when the data must be requested (through the `loader` function). Here we are using an `ol.loadingstrategy.createTile` strategy with an `ol.tilegrid.XYZ` tile grid. This means the vector data is queried like a tiled pyramid, producing a request with the bounding box of each tile:

```
    strategy: ol.loadingstrategy.createTile(new ol.tilegrid.XYZ({
        maxZoom: 19
    })),
```

Until here the common code. Now lets go to describe the code used for the `format` and `loader` properties, which differs depending on using AJAX or JSONP.

For the AJAX source and, because no output format is specified in the request sent to the WFS server (see the `loader` function), data is returned in WFS format. So, in the source's `format` property, we have set an instance of `ol.format.WFS` indicating the `featureNS` and `featureType`properties:

```
    format: new ol.format.WFS({
        featureNS: 'http://openstreemap.org',
        featureType: 'water_areas'
    }),
```

On its way, the `loader` function computes the URL to request from the received parameters. In the code we can see how the `bbox` parameter is built from the `extent` received in the function.

The `loader` function is also responsible to make the request, read the returned data and add them to the layer:

```
    loader: function(extent, resolution, projection) {
        var url = 'http://demo.opengeo.org/geoserver/wfs?'+
            'service=WFS&request=GetFeature&'+
            'version=1.1.0&typename=osm:water_areas&'+
            'srsname=EPSG:3857&'+
            'bbox=' + extent.join(',');

        $.ajax({
            url: url
        })
        .done(function(response) {
            vectorSource.addFeatures(vectorSource.readFeatures(response));
        });
    },
```

The request is made using jQuery `$.ajax()` method. This method returns a promise that runs the `done()` method, passing the requested data, if the operation goes fine.

To load data using AJAX we required the remote server implements the CORS mechanism, otherwise the browser will raise an error related message saying we are not allowed to request the server.

The `done()` method converts the returned data to features, using the source `readFeatures()` method and finally add them to the feature collection with the `addFeatures()` method.

Remember the source delegates the tasks of reading features to a format class. See Introducing vector source and format hierarchies section at Data sources and formats chapter.

For the source that uses JSONP technique, the `format` and `loader` properties differs slightly.

Before to continue, we need to understand to make requests using JSONP the server must also implement the JSONP technique, because it does not returns only data but a call to the specified JavaScript function passing the data as argument.

More information about the JSONP technique at http://en.wikipedia.org/wiki/JSONP[28].

Fortunately for us, the target server (`http://demo.opengeo.org`) is a GeoServer[29] instance which allows to return data using the JSONP technique. To do it we need to set in the URL to request the parameter `outputFormat=text/javascript` and `format_options=callback:loadFeatures`. Here, the `loadFeatures` is the function to be called, passing the requested data, once the request is finished.

Note the value `text/javascript` for the `outputFormat` property and the `format_options` property are vendor specific parameters and they are not part of the WFS standard specification.

Knowing all these, the `loader` function looks like:

[28] http://en.wikipedia.org/wiki/JSONP
[29] http://geoserver.org/

```
    loader: function(extent, resolution, projection) {
        var url = 'http://demo.opengeo.org/geoserver/wfs?'+
            'service=WFS&request=GetFeature&'+
            'version=1.1.0&typename=osm:water_areas&'+
            'outputFormat=text/javascript&'+
            'format_options=callback:loadFeatures&' +
            'srsname=EPSG:3857&'+
            'bbox=' + extent.join(',');

        $.ajax({
            url: url,
            dataType: 'jsonp'
        });
    },
```

Here we set the `dataType` to `jsonp` in the `$.ajax()` method, to indicate we want to use the JSONP technique. Because of this, there is no need to specify a `done()` method, it will be never called, because once the data were received the `loadFeatures` function will be invoked.

```
    var loadFeatures = function(response) {
        vectorSourceJsonp.addFeatures(vectorSourceJsonp.readFeatures(response));
    };
```

When data is requested using the JSONP technique, Geoserver returns the data in GeoJSON format. So we need set the source `format` property as:

```
    format: new ol.format.GeoJSON(),
```

3.4.8.4 See also

- Working with loading strategies example at Data sources and formats chapter.

3.4.9 Working with loading strategies

3.4.9.1 Goal

On this example we are going to see the different loading strategies we can apply when working with the `ol.source.ServerVector` source.

In the sample we create two maps. The left one will use a *fixed* strategy, that is, the data will be requested only once and for a given bounding box. The right one will request data each time the map's view change and with a bounding box equal to the view extent.

3.4.9.2 How to do it...

As always start adding the HTML code necessary for the maps:

```
    <div class="row">
        <div id="mapFixed" class="map col-sm-6"></div>
        <div id="mapBbox" class="map col-sm-6"></div>
    </div>
```

Both maps will use OpenStreetMaps tiles as base layers so create it:

```
    var osmLayer = new ol.layer.Tile({
        source: new ol.source.OSM()
    });
```

Now create the maps. Both follows the same steps. Create a source that using JSONP technique to load data, create a vector layer that uses the previous source and add it to the map.

The code for the left map:

```
    // Source using a fixed box strategy
    var vsStrategyFixed = new ol.source.ServerVector({
        format: new ol.format.GeoJSON(),
        loader: function(extent, resolution, projection) {
            var url = 'http://demo.opengeo.org/geoserver/wfs?'+
                'service=WFS&request=GetFeature&'+
                'version=1.1.0&typename=osm:water_areas&'+
                'outputFormat=text/javascript&'+
                'format_options=callback:loadFeaturesFixed&' +
                'srsname=EPSG:3857&bbox=' + extent.join(',');

            $.ajax({
                url: url,
                dataType: 'jsonp'
            });
        },
        strategy: function() {
            return [ [-8473015.930372493, 5673984.22207263, -8430593.37967422, 5\
704559.033386701] ];
        },
        projection: 'EPSG:3857'
    });

    // Executed when data is loaded by the $.ajax method.
    var loadFeaturesFixed = function(response) {
        vsStrategyFixed.addFeatures(vsStrategyFixed.readFeatures(response));
```

```
    };

    // Vector layer
    var vectorLayerFixed = new ol.layer.Vector({
        source: vsStrategyFixed,
        style: new ol.style.Style({
            stroke: new ol.style.Stroke({
                color: 'green',
                width: 2
            })
        })
    });

    // Map
    var mapFixed = new ol.Map({
        target: 'mapFixed',
        renderer: 'canvas',
        layers: [osmLayer, vectorLayerFixed],
        view: new ol.View({
            center: ol.proj.transform([-75.923853, 45.428736], 'EPSG:4326', 'EPS\
G:3857'),
            maxZoom: 19,
            zoom: 10
        })
    });
```

And finally, the code for the right map, which uses the bounding box strategy:

```
    // Source using a bbox strategy
    var vsStrategyBbox = new ol.source.ServerVector({
        format: new ol.format.GeoJSON(),
        loader: function(extent, resolution, projection) {
            var url = 'http://demo.opengeo.org/geoserver/wfs?'+
                'service=WFS&request=GetFeature&'+
                'version=1.1.0&typename=osm:water_areas&'+
                'outputFormat=text/javascript&'+
                'format_options=callback:loadFeaturesBbox&' +
                'srsname=EPSG:3857&bbox=' + extent.join(',');

            $.ajax({
                url: url,
                dataType: 'jsonp'
```

```
            });
        },
        strategy: ol.loadingstrategy.bbox,
        projection: 'EPSG:3857'
    });

    // Executed when data is loaded by the $.ajax method.
    var loadFeaturesBbox = function(response) {
        vsStrategyBbox.addFeatures(vsStrategyBbox.readFeatures(response));
    };

    // Vector layer
    var vectorLayerBbox = new ol.layer.Vector({
        source: vsStrategyBbox,
        style: new ol.style.Style({
            stroke: new ol.style.Stroke({
                color: 'green',
                width: 2
            })
        })
    });

    // Map
    var mapBbox = new ol.Map({
        target: 'mapBbox',
        renderer: 'canvas',
        layers: [osmLayer, vectorLayerBbox],
        view: new ol.View({
            center: ol.proj.transform([-75.923853, 45.428736], 'EPSG:4326', 'EPS\
G:3857'),
            maxZoom: 19,
            zoom: 10
        })
    });
```

3.4.9.3 How it works...

As we have learned in the theory section, the `ol.source.ServerVector` source accepts a `strategy` property which determines when the source must request data. Each time the `strategy` function *says* data must be loaded, the `loader` function is executed.

The function specified as `strategy` receives two parameters, the current `extent` and `resolution`, and must return an array with the extents to be loaded.

So, to create a *fixed* strategy, one that always load the same extent, it is enough with a function that returns an array with the same extent. For the source used in the left map, the `strategy` function looks like:

```
    strategy: function() {
        return [ [-8473015.930372493, 5673984.22207263, -8430593.37967422, 57045\
59.033386701] ];
    },
```

As explained at Understanding the `ServerVector` class section, OpenLayers3 offers three different strategy function implementations.

For the right map's source we have used the `ol.loadingstrategy.bbox` function. At time of writing, the code for the function looks like:

```
    ol.loadingstrategy.bbox = function(extent, resolution) {
      return [extent];
    };
```

A simple function that receives current map's view `extent` and `resolution` and returns an array of the extents to be requests, that is, the current view's extent.

To make use of the `ol.loadingstrategy.bbox` strategy in our sources, it is enough with:

```
    strategy: ol.loadingstrategy.bbox,
```

3.4.9.4 See also

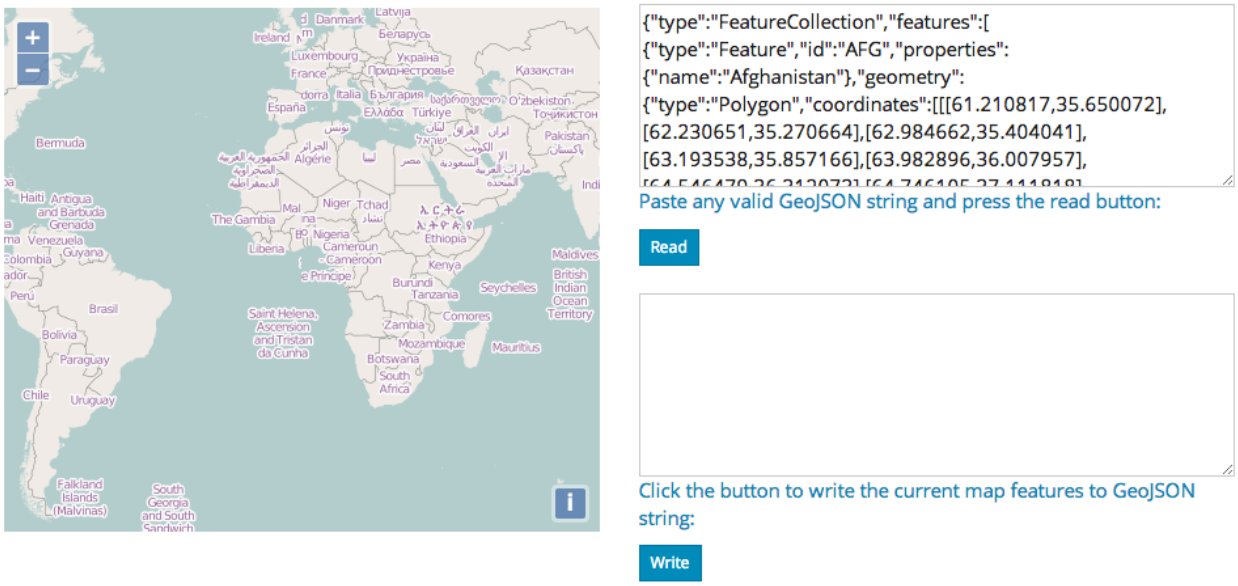
- Requesting data from a WFS server with and without JSONP example at Data sources and formats chapter, to see how to use a `ol.loadingstrategy.createTile` strategy.

3.4.10 Reading and writing features through the source class

3.4.10.1 Goal

This example show the power of format classes that allows, not only read features in a data format, but writing a set of feature instance in a given format.

We are going to create a map plus two text areas that will allow to read and write data in GeoJSON format. The read input will allow to paste a GeoJSON text, read it and put the features in the map (an empty string will erase the map). The write input will be filled with the current map features written in GeoJSON format.



Reading and writing features

3.4.10.2 How to do it...

Start adding the HTML code required for the layout that contains the map and the two textareas. The layout will be divided in two sides. On the left we will place the map and on the right we will place the textareas to read and write features:

```
    <div class="row">
       <div class="col-md-6">
           <div id="map" class="map"></div>
       </div>
       <div class="col-md-6">
           <textarea id="read">...</textarea>
           <p class="text-primary">Paste any valid GeoJSON string and press the\
 read button:</p>
           <button id="readButton" class="btn btn-primary btn-xs">Read</button>
           <br/><br/>

           <textarea id="write"></textarea>
           <p class="text-primary">Click the button to write the current map fe\
atures to GeoJSON string:</p>
           <button id="writeButton" class="btn btn-primary btn-xs">Write</butto\
n>
       </div>
    </div>
```

Note, we have obviated the content within the read textarea, that initially contains a big GeoJSON text with the world administration limits.

Now create the source and layer that will contain the features we add:

```
var format = new ol.format.GeoJSON();
var vectorLayer = new ol.layer.Vector({
    source: new ol.source.StaticVector({
        format: format,
        projection: 'EPSG:3857'
    })
});
```

Create the map instance. It will contains initially two layers. One with OpenStreetMap data, acting as base layer, and the previous empty vector layer:

```
var map = new ol.Map({
    target: 'map',  // The DOM element that will contains the map
    renderer: 'canvas', // Force the renderer to be used
    layers: [
        // Add a new Tile layer getting tiles from OpenStreetMap source
        new ol.layer.Tile({
            source: new ol.source.OSM()
        }),
        vectorLayer
    ],
    view: new ol.View({
        center: ol.proj.transform([0, 0], 'EPSG:4326', 'EPSG:3857'),
        zoom: 2
    })
});
```

Finally, register two listener function for the `click` event on the read and write buttons:

```
    $('#readButton').on('click', function() {
        var source = vectorLayer.getSource();
        var text = $('#read').val();

        if(text==='') {
            source.clear();
            return;
        }
        var json = JSON.parse(text);
        var features = source.readFeatures(json);
        source.addFeatures(features);
    });

    $('#writeButton').on('click', function() {
        var source = vectorLayer.getSource();
        var features = source.getFeatures();
        var json = format.writeFeatures(features);
        var text = JSON.stringify(json);
        $('#write').val(text);
    });
```

3.4.10.3 How it works...

As we will see in while, for this example we require to have access to the source and format classes used to read the features.

When using source classes like `ol.source.GeoJSON` or `ol.source.KML` they internally makes use of the right format class (`ol.format.GeoJSON` and `ol.format.KML` respectively) but the issue is we can get a reference to the format class. See the Understanding the `StaticVector` based classes section at Data sources and formats chapter for more information.

To avoid this issue, we have make use of the `ol.source.StaticVector` class, (which is the base class concrete types like `ol.source.GeoJSON` or `ol.source.KML`). This way we can specify the format class to be used and later can reference the format class to write features:

```
    var format = new ol.format.GeoJSON();
    var vectorLayer = new ol.layer.Vector({
        source: new ol.source.StaticVector({
            format: format,
            projection: 'EPSG:3857'
        })
    });
```

With this explained we can continue describing how the sample works. All the magic occurs when we click the read or write buttons. We have used jQuery to register an anonymous listener function on the `click` event of each button.

The steps done by the read action are summarized as:

- Get the text from the input area.
- Read it and return a set of `ol.Feature` instances. To so we use the source `readFeatures()` method, which internally delegates to the format class this task (invoking the also called readFeatures() within the format class).
- Add the features to the source. When the source contents changes it automatically emits a `change` event and OpenLayers3 automatically redraws the map.

```
1   $('#readButton').on('click', function() {
2       var source = vectorLayer.getSource();
3       var text = $('#read').val();
4
5       if(text==='') {
6           source.clear();
7           return;
8       }
9       var json = JSON.parse(text);
10      var features = source.readFeatures(json);
11      source.addFeatures(features);
12  });
```

Because the `readFeatures()` method requires a JavaScript object instance we need to parse the text with `JSON.parse()`, which are available in any modern browser.

Note, if the read textarea is empty we erase the contents of the source with the `clear()` method.

The steps for the write action are similar:

- Retrieve the features from the source.
- Convert the to a set of object instances in the right format.
- Write the previous data to the write textarea.

```
1     $('#writeButton').on('click', function() {
2         var source = vectorLayer.getSource();
3         var features = source.getFeatures();
4         var json = format.writeFeatures(features);
5         var text = JSON.stringify(json);
6         $('#write').val(text);
7     });
```

Because the source class ha no method write feature, only to read, we need to use the format class directly. With the `writeFeatures()` method we convert a set of `ol.Features` to an array of JavaScript object instances.

Using the `JSON.stringify()` method, available in all modern browser, we can convert an object to a string representation and write it to the write textarea.

Note, the return type of the `writeFeatures()` may differ among the format class we use. This way, while in the `ol.format.GeoJSON` class returns an object in the `ol.format.WKT` class it returns an string with the WKT[30](Well-Known Text) representation.

[30] http://en.wikipedia.org/wiki/Well-known_text

4. Vector layers

Working with vector information is a hard tasks when designing any GIS tool. Vector information can be stored in many different data formats and hosted on different data sources. In the Data sources and formats chapter we have seen the *sources* solves the access challenge, that is, how to read vector data.

This chapter focuses on how to work with vector information. We will explain the concept of feature, geometry and style and how to work with them. Finally we will see how we can manage features within a source.

4.1 Introducing features, geometries and styles

A *feature* represents a real world thing. It can be a city, a river, a country, etc, anything that can have interest for us.

A feature can have one or more *properties* associated that give us information about it. For example, a city can have the properties: name, area, male and female population, etc.

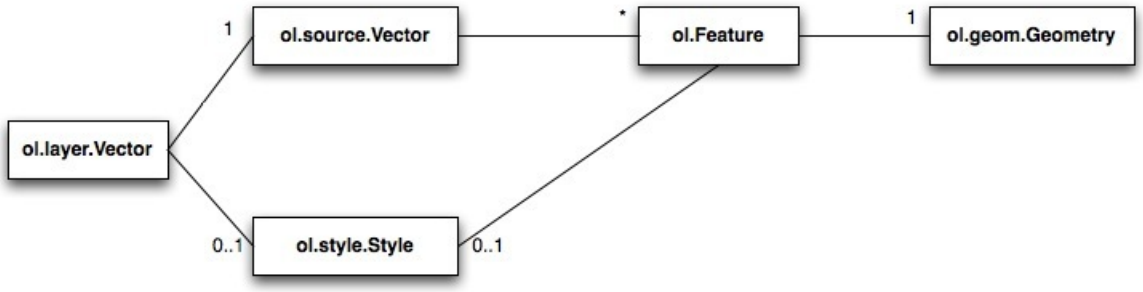
Vector information, that is, features can be represented in many different ways: as a point (for a city), a line (for a river), a polygon (for a region), etc. OpenLayers3 implements, as close as possible, the Simple Feature Access[1] standard, which determines a common model for two dimensional geographical data.

The geographical representation of the feature is so called the *geometry* and each feature has one associated with it.

Usually, although two features were represented by the same geometry we want to render them in a different way, that is, using a different *style*. This way, we could render longer rivers as lines with a wider stroke than shortest rivers.

The next figure shows the relationships among the whole set of related classes:

[1] http://en.wikipedia.org/wiki/Simple_Features



Relationship among vector layer, source, features, geometries and styles

A *vector layer* makes use of a vector *source* to access data. From vector layer point of view the source is like a container of features where the layer can get, add, remove or modify features.

All *features*, as we have explained, has a *geometry* instance associated with them that determines the *shape* of the feature.

Given a *style* instance we can set it both to a layer, so the style will be applied to all features, or to a single feature, so the style will only be applied on that concrete feature. If a feature has a style associated, no matter if the layer it belongs has an style, the feature's style takes precedence over the layer's style.

A style attached to a feature takes precedence over the layer style.

How styles are applied

OpenLayers3 contains a special kind of classes, called *renderers*, which know how to render the map, layers, features, ... using the appropriate code sentences depending on the technology to be used to render the map. This technology is determined by the map's `renderer` property and can take the values `canvas`, `webgl` or `dom`.

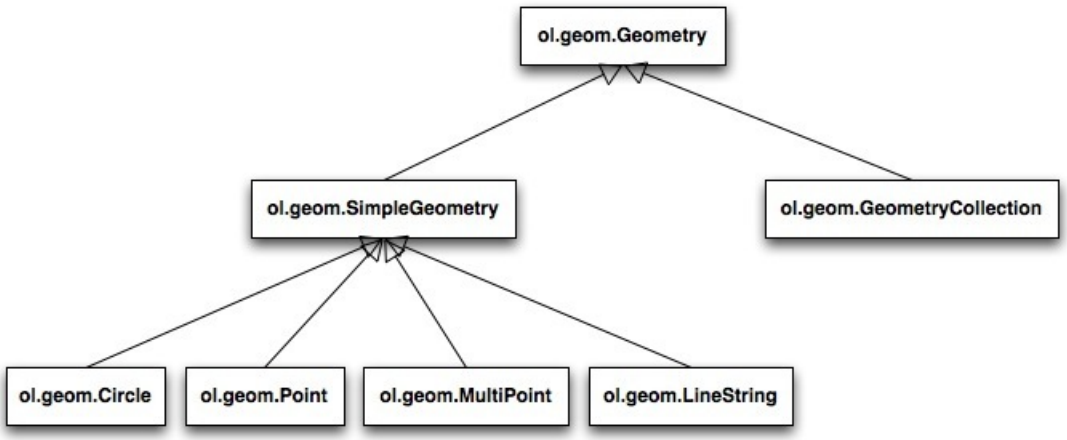
It is important to know the existence of the *renderers* because all the styling magic is created within these classes. When the renderer must render a vector layer, it gets each contained feature and extracts its geometry and style and renders its accordingly.

The renderer know how to render points, lines, polygons, etc. For the style, it uses the associated feature style (if has one), the layer style (if has one) or a default style predefined by OpenLayers3.

4.2 Playing with geometries

Before explain in depth about features and, because features makes use of geometries, lets first take a look to the geometries in OpenLayers3.

OpenLayers3 defines a hierarchy of geometry classes that covers the Simple Feature Access[2] standard.



Geometry hiearchy

The root class `ol.geom.Geometry` defines the common behavior of the hierarchy, like the action to `clone()` or get the whole extent covered by the geometry with `getExtent()`.

After the root class we found two main subclass:

- `ol.geom.GeometryCollection`, which acts as a container of other geometries and,
- `ol.geom.SimpleGeometry`, which, on its way, is the base class for the rest of available geometry classes like: `ol.geom.Point`, `ol.geom.MultiPoint`, `ol.geom.LineString`, etc.

All geometries can be cloned through the `clone()` method. In addition, each concrete subclass has its own methods. It is logically because, for example, a linear ring does not have the same properties than a point or a circle.

In all concrete geometry subclasses (except for `ol.geom.Circle`) we can get its coordinates using the `getCoordinates()` method. Note, we can get different data depending on the kind of geometry. This way, for `ol.geom.Point` we obtain a pair `[x,y]`, while for `ol.geom.LineString` we get an array of pairs `[ [x0,y0], [x1,y1], [x2,y2], ... ]`

For `ol.geom.Circle` we can get its center coordinate and radius with `getCenter()` and `getRadius()`.

[2] http://en.wikipedia.org/wiki/Simple_Features

The main difference between `old.geom.Point` and `ol.geom.Circle` is the second represents a feature that covers a fixed geographic circular area, that is, when rendered the size of the circle in the map changes depending on the zoom level. The `ol.geom.Point` are rendered as a fixed radius the size of which does not changes with map's resolution.

In addition to the common methods each subclass can have its own. For example, `ol.geom.Polygon` has the `getArea()` method that returns the area occupied by the polygon or `ol.geom.LineString` has the `getFirstCoordinate()` and `getLastCoordinate()`.

As the author I consider not relevant to make an exhaustive list of methods for each geometry. That information is perfectly covered by the API documentation[3]. As always, I encourage the reader not only to explore the API documentation but the project source code.

Because the data each geometry requires is different, the way we create geometries differs. Next code shows how we can instantiate some different geometries and some of its properties:

```
1     // Create a point
2     var point = new ol.geom.Point([10,30]);
3     console.log(
4         "point coordinates", point.getCoordinates(),
5         "extent", point.getExtent(),
6         "layout", point.getLayout());
7
8     // Create a line string
9     var lineString = new ol.geom.LineString([ [0,0], [10,0], [10,10], [0,10] ]);
10    console.log(
11        "lineString coordinates", lineString.getCoordinates(),
12        "extent", lineString.getExtent(),
13        "layout", lineString.getLayout());
14
15    // Create a circle
16    var circle = new ol.geom.Circle([4,5], 10);
17    console.log(
18        "circle center", circle.getCenter(),
19        "radius", circle.getRadius(),
20        "extent", circle.getExtent(),
21        "layout", circle.getLayout());
```

At the time of writing these lines OpenLayers3 does not offers spatial operations to work with geometries (like *contains*, *intersects*, etc).

[3] http://ol3js.org/en/master/apidoc

Once we know how to create geometries lets go to describe how we can create features.

4.3 Creating features by hand

A feature is represented as an instance of the `ol.Featue` class, which offers methods to work with its geometry, style and properties. Like many other classes in OpenLayers3, `ol.Feature` inherits from base class `ol.Object`, which is designed to allow store any kind of property and observe for changes on them.

See the Events, listeners and properties chapter for a better description and understanding on how to work with properties and event listeners.

Next is a list with a subset of the most common used methods when working with features:

- `getGeometry()/setGeometry()`, gets/sets the geometry instance associated with the feature.
- `getStyle()/setStyle()`, gets/sets the style associated with the feature.
- `get()/set()`, to get or set any kind of property. Inherited from `ol.Object`.
- `clone()`, creates a new feature equal like the current one. Its geometry, style and any other properties are also cloned.

Note, although the feature's geometry has its own getter and setter methods, the value is stored as a regular property with name `geometry`. We can see this retrieving all the feature properties with `getProperties()` method. In addition, we can change the name of the property used to store the geometry using the `setGeometryName()` method.

We have two ways to create a new feature instance:

1. Using an object literal passing the `geometry` field to specify the geometry plus any other properties:

```
var circle = new ol.geom.Circle([4,5], 10);
var circleFeature = new ol.Feature({
    geometry: circle,
    name: 'a property to hold the feature name'
});
```

1. Passing a geometry reference and later set the feature properties:

```
var circle = new ol.geom.Circle([4,5], 10);
var circleFeature = new ol.Feature(circle);
circleFeature.set('name', 'a property to hold the feature name');
```

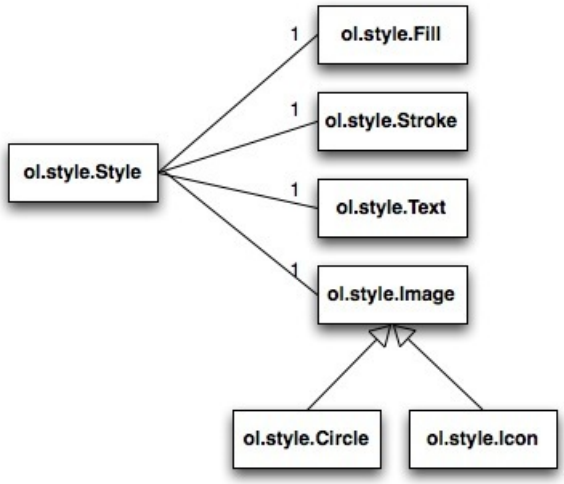
In both cases, we can not set the style at initialization time, we need to set it once we have a feature instance reference using `feature.setStyle(...)`.

4.4 Styling features

Style must be flexible enough to satisfy our needs in any situation while developing a mapping application.

Because features are represented by geometry points, lines, polygons, etc, we need a way to specify things like the fill color or the stroke width. But also, there are times we want to place icons at some places, a marker indicating a point of interest.

OpenLayers3 comes with a small but powerful set of classes that offers us a great degree of control on the way we render our features. Basically, we can say a style is a combination of fill, stroke, text and image options.



Style related classes

The main class is `ol.style.Style` and , to specify the fill, stroke, text and image it has the properties `fill`, `stroke`, `text` and `image`, which are specified using an instance of the appropriate class.

Each one of these classes has its own properties that varies depending on their needs. For example, `ol.simple.Fill` simple accepts a `color` property, because it is the only required property necessary to specify how to fill a shape, while `ol.style.Stroke` accepts `color`, `width` and some more properties that allows to play with the stroke of the shape.

See the API documentation[4] to know the available properties on each class. The goal of this chapter is not to describe the properties in depth, but introduce the reader to styling in OpenLayers3 through some theory and practice exercises.

For representing features as images, OpenLayers3 defines two classes: `ol.style.Icon` and `ol.style.Circle`, which inherits from the `ol.style.Image` class. The first is designed to render any image specified by passing an URL to the property `image`. The second is designed to render circles with great performance and, for this reason, it is used to render point features.

When using `ol.style.Circle` with the `canvas` renderer, a circle is generated in a canvas element, *extracted* as an image and reused for each point feature to be renderer.

As example, lets go to examine the code for the default style used to render vector features (that can be found at `ol.style.defaultStyleFunction` function within the `src\ol\style\style.js` file):

```
1     var fill = new ol.style.Fill({
2         color: 'rgba(255,255,255,0.4)'
3     });
4     var stroke = new ol.style.Stroke({
5         color: '#3399CC',
6         width: 1.25
7     });
8     var styles = [
9         new ol.style.Style({
10            image: new ol.style.Circle({
11                fill: fill,
12                stroke: stroke,
13                radius: 5
14            }),
15            fill: fill,
16            stroke: stroke
17        })
18    ];
```

As we can see, the default style makes use of three properties to style features. The `stroke` is used in lines and determines its color (almost blue) and width. The `fill` is used to fill closed features, like polygons, and determines its color (a translucent blue). Finally, for the `image` an instance of `ol.style.Circle` is used and it is used to render point features.

[4] http://openlayers.org/en/v3.0.0/apidoc/ol.style.html

About colors

Colors are specified as strings and can take the form of the next four formats:

- `rgb(r,g,b)` or `rgba(r,g,b,a)`, where `r`, `g` and `b` values should be integers in the range `0..255` and the `a` value must be a float in the range `0..1`.
- `#rrggbb` or `#rgb`, where `r`, `g` and `b` are values in hexadecimal format (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`). The `#rgb` format means a value like `#36a` will be interpreted as `#3366aa`.

Finally, note the `r`, `g`, `b` and `a` values stands for red, green, blue and alpha (transparency) color components.

4.4.1 Using icons to style features

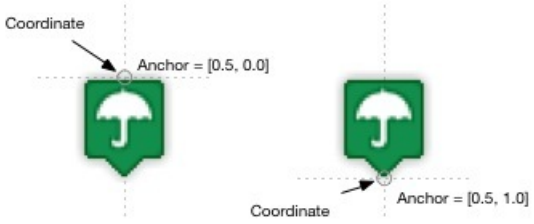
Sometimes, we desire to style a feature to be rendered as a marker, that is, using an icon to indicate it is a point of interest. We can achieve this using the `image` property of the `ol.style.Style` class.

It is important to remember, within the rendering process **only point based features (features using an `ol.geom.Point` and `ol.geom.MultiPoint` geometries) are rendered taking into account the `image` property** of the `ol.style.Style` class. For other geometries the `image` property is ignored.

We can set the `image` property passing an `ol.style.Icon` instance. The class is rich enough to control most of the aspect needed when rendering the icon. Among others, the class has the next properties:

- `src`, an URL to an image file to be used as the icon.
- `opacity`, a value between the range 0..1 indicating the transparency of the icon.
- `rotation`, a value in radians specifying the rotation to be applied to the icon.
- `scale`, a multiplication factor of the icon size.
- `rotateWithView`, if true the icon rotates when the view is rotated, otherwise they are fixed respect the initial rotation angle.
- `anchor`, a `[X, Y]` number array that determines where the icon is placed. Given a coordinate the icon can be placed centered, on top-left respect the coordinate, bottom-center, etc. The values are specified in the units indicated by `anchorXUnits` and `anchorYUnits`, that can take the value `pixel` or `fraction`.

To better understand of `anchor`, `anchorXUnits` and `anchorYUnits` properties, next image shows two examples using `fraction` units. We can see how the anchor and units determines the location the icon is places respect the feature's geometry coordinate:



Two samples of icon anchor

Using `pixel` units the `anchor` values acts as an offset to be applied to the icons.

4.4.2 Working with text

Sometimes we want to show some text when rendering our features. For example, we can render the cities of a country as circles (with different radius depending on its population) plus a text with the city name.

As we commented, the `ol.style.Style` class offers the `text` property where to we can attach a *text style* represented by an instance of `ol.style.Text`.

Text style takes effect on any feature, no matter the kind of geometry it uses, and the text is rendered centered on the geometry.

Next code shows a `text` attribute initialized to render the `hello` text message with white stroke and black fill:

```
    text: new ol.style.Text({
      text: 'hello',
      fill: new ol.style.Fill({
        color: '#000'
      }),
      stroke: new ol.style.Stroke({
        color: '#fff',
        width: 3
      })
    })
```

The `ol.style.Text` class has many features we can use to determine the way the text is rendered:

- `text`, the message to be rendered
- `fill`, determines the text fill color. Must be an instance of `ol.style.Fill`.
- `stroke`, determines the textr stroke (color, with, ...). Must be an instance of `ol.style.Stroke`.
- `font`, a string specifying the font to be used. It can contain font size, variant, weight, etc.
- `rotation`, rotates the text the given value in radians.
- `scale`, factor applied to the text size.
- `textAlign`, determines the horizontal alignment of the text (right, center, left, ...).
- `textBaseline`, determines the vertical alignment of the text (top, bottom, middle, ...).
- `offsetX` and `offsetY`, allows to set an offset respect the original text position.

When using the canvas rendered, the `font`, `textAlign` and `textBaseline` properties are passed directly to the same named properties in the `canvas` element used to render the font. For more information about allowed values, look for documentation about canvas properties in the Internet.

4.4.3 Applying styles to layers and features

We have seen both `ol.layer.Vector` and `ol.Feature` has the `style` property that allows to specify the style to be used at rendering time. In addition and, hopefully for us, OpenLayers3 offers different ways to apply styles to a target given us a great degree of flexibility. So we can set the `style` property with:

1. An instance of `ol.style.Style`. This is the most basic form. We have shown in previous sections how to create style instances specifying the fill, stroke, icons or text to be used.
2. An array of `ol.style.Style`. OpenLayers3 allows to apply more than one style to the features. This is useful for example to render the same feature with two different styles. See sample Markers: Styling features with icons at the practice section of Vector layers chapter.
3. A so called *style function* responsible to return an array of styles to be applied to the feature. This can be similar two the second options but as we will see it is the most flexible one. See sample Working with style functions at the practice section of Vector layers chapter.

There is an important exception when styling features: the `ol.source.ImageVector`. Remember by the section Rendering vector data as raster at chapter Data sources and formats, the `ol.source.ImageVector` acts as a wrapper around a vector source rendering data to a raster. In this cases we need to apply the desired style to the `style` property of the `ol.source.ImageVector` so it can render features properly when converting to raster.

4.4.3.1 Understanding the *style functions*

A *style function* is a kind of function responsible to return the array of styles to be applied when rendering a feature and it is invoked each time the feature is going to be rendered.

Depending if we apply the style to a vector layer or to an individual feature, the signature of the *style function* can vary. This way, when the *style function* is applied to an `ol.layer.Vector` layer the function must be of type `ol.style.StyleFunction`, which conforms the signature `function(ol.Feature, number): Array.<ol.style.Style>`. Next is a valid function following the signature:

```
var customStyleFunction =  function(feature, resolution) {
    var style1 = ...;
    var style2 = ...;

    return [style1, style2];
}
```

The function is invoked for each feature contained within the layer (really the layer's source) so we can style each feature given some conditions.

On the other hand, when the function is applied to an `ol.Feature` the function must be of type `ol.feature.FeatureStyleFunction`, which conforms the signature `function(this: ol.Feature, number): Array.<ol.style.Style>`. Next is a valid function following the signature:

```
var customStyleFunction =  function(resolution) {
    var feature = this;
    var style1 = ...;
    var style2 = ...;

    return [style1, style2];
}
```

Here we do not need any reference to the feature. The `this` keyword within the style function point to the feature to be rendered.

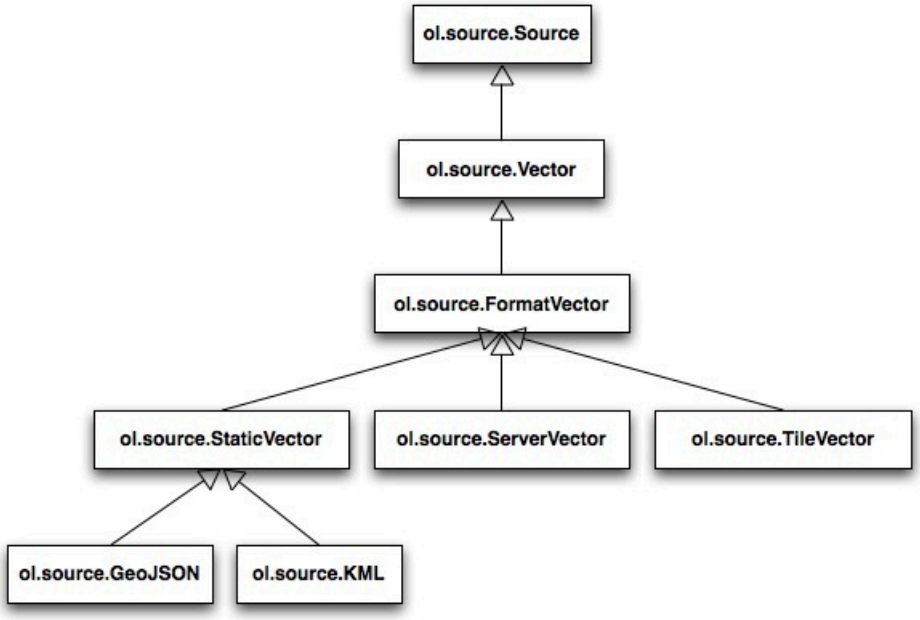
Next code, shows how we can apply a style function to a vector layer so it renders all the features using its `name` property:

```
var customStyleFunction = function(feature, resolution) {
    return [new ol.style.Style({
        text: new ol.style.Text({
            text: feature.get('name'),
            fill: new ol.style.Fill({
                color: '#000'
            }),
            stroke: new ol.style.Stroke({
                color: '#fff',
                width: 1
            })
        })
    })];
};

var vectorLayer = new ol.layer.Vector({
    style: customStyleFunction
    ...
});
```

Because the style function is called each time a feature is rendered, it is a good idea to store the generated styles within an array and reuse them. In summary, take into account apply some cache technique.

4.5 Managing features

We can think on vector sources as a *container* or array of features (see Vector sources and formats section at Data sources and formats chapter). Because of this the vector source classes offers methods to retrieve, add or remove features.



Vector source hierarchy

All the common methods required to manage features are implemented by the base class `ol.source.Vector`:

- `addFeature()/addFeatures()`, allows to add a feature or an array of features to the source,
- `getFeatures()`: returns an array with all the features within the source,
- `getFeaturesAtCoordinate()`, returns an array of features that intersects the given coordinate,
- `removeFeature()`, removes a given feature from the source,
- `clear()`, removes all the source features,
- `getExtent()`, returns the extent occupied by all the features within the source.
- `forEachFeature()/forEachFeatureInExtent()`, allows to apply a function to each feature or, for the second method, only those features within a given extent.

With this set of methods, it is clear we can create a feature programmatically and add it to a source vector instance:

```
1    var geom = new ol.geom.Point([40,2]);
2    var feature = new ol.Feature(geom);
3    var source = new ol.source.Vector();
4    source.addFeature(feature);
```

Note, it is more elegant to pass an array of features at source's initialization time, using the `features` property, and leave the `addFeature()` method for subsequent additions:

```
var source = new ol.source.Vector({
    features: [feature]
});
```

Also we can make a modification on each feature:

```
source.forEachFeature( function(feature){
    feature.set('randomNumber', feature.get('randomNumber') + Math.random());
});
```

Note it is better to use the `forEachFeature()` for this purpose instead to loop the array returned by the `getFeatures()` method.

Be aware not modify a feature's geometry within the `forEachFeature` method or you will get an error like `Error: cannot update extent while reading`. Next code would produce the previous error:

```
source.forEachFeature( function(feature){
    var geom = feature.getGeometry();
    if(geom.getType() === 'Point') {
        var coords = geom.getCoordinates();
        geom.setCoordinates([coords[0] + 10, coords[1] + 10]);
    }
});
```

Finally, we can remove a feature from the source with:

```
source.removeFeature(feature);
```

or simply completely clean the source:

```
source.clear();
```

4.5.1 A word about events

As it is logical the source and format classes requires some time to read the data content and transform into feature instances. This can lead in weird situation where, after creating a source and try reading its features, sometimes get a null reference and sometimes get a valid result. The problem is we are trying to read features without to be sure they are fully loaded. Next example shows the mentioned situation:

```
1     // Create source
2     var source = new ol.source.GeoJSON({
3             ...
4     });
5     // Get features reference. Are we sure they are loaded?
6     var features = source.getFeatures();
```

To solve this situation, the source triggers an event indicating the data is ready to be used.

Events are described in more detail at Events, listeners and properties chapter. In addition, a specific sample has been created, see Listening for changes on vector data sample, to show how to solve this issue when reading vector data.

4.6 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[5] repository.

4.6.1 Playing with geometries

4.6.1.1 Goal

The goal of this recipe is simply demonstrate how we can create geometries by hand. This will become necessary when we want to create features programmatically.

Note, this recipe only requires JavaScript code. All the results are shown through the browser console.

4.6.1.2 How to do it...

Add the next JavaScript code to the `<script>` element on your HTML page:

```
    // Create a point
    var point = new ol.geom.Point([10,30]);
    console.log(
        "point coordinates", point.getCoordinates(),
        "extent", point.getExtent(),
        "layout", point.getLayout());

    // Create a line string
    var lineString = new ol.geom.LineString([ [0,0], [10,0], [10,10], [0,10] ]);
    console.log(
        "lineString coordinates", lineString.getCoordinates(),
        "extent", lineString.getExtent(),
        "layout", lineString.getLayout());

    // Create a circle
    var circle = new ol.geom.Circle([4,5], 10);
    console.log(
        "circle center", circle.getCenter(),
        "radius", circle.getRadius(),
        "extent", circle.getExtent(),
        "layout", circle.getLayout());
```

[5] https://github.com/acanimal

4.6.1.3 How it works...

Previous code create three different kind of geometries: a point, a line string and a circle.

All three have been created specifying its coordinates in `EPSG:4326` projection. For each one we show some of its properties in the browser's console, like its coordinates or extent.

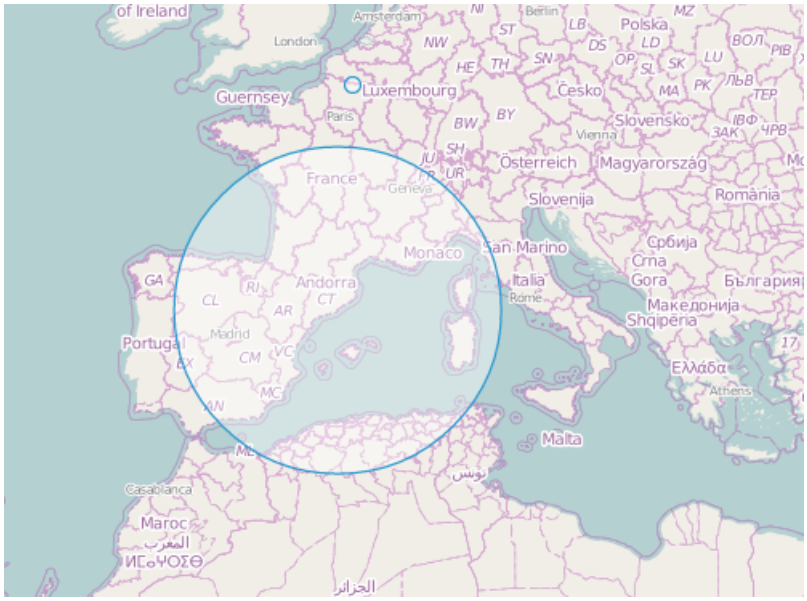
4.6.1.4 See also

- Creating features programmatically example at Vector layers chapter, to see how we can create features programmatically instead loading them through a source instance.

4.6.2 Creating features programmatically

4.6.2.1 Goal

On this recipe we are going to show how we can create features by code instead of using a source instance.



Creating features programmatically

4.6.2.2 How to do it...

As always add a HTML element to hold the maps. Next add the JavaScript code necessary to create a circle and point geometry and create their corresponding features:

```
    // Geometries
    var point = new ol.geom.Point(
        ol.proj.transform([3,50], 'EPSG:4326', 'EPSG:3857')
    );
    var circle = new ol.geom.Circle(
        ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        1000000
    );

    // Features
    var pointFeature = new ol.Feature(point);
    var circleFeature = new ol.Feature(circle);
```

Now initialize a `ol.source.Vector` instance with the previous features and add it to a vector layer:

```
    // Source and vector layer
    var vectorSource = new ol.source.Vector({
        projection: 'EPSG:4326'
    });
    vectorSource.addFeatures([pointFeature, circleFeature]);

    var vectorLayer = new ol.layer.Vector({
        source: vectorSource
    });
```

Finally create the map instance with the vector layer and a OSM raster layer that will act as background layer:

```
    // Map
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
```

```
15 '),
16             zoom: 4
17         })
18     });
```

4.6.2.3 How it works...

The map's view uses by default the `EPSG:3857` projection. This is fine because we are using an OSM layer as background but that means the geometry of our features must be specified in that project. Because of this we are using the `ol.proj.transform()` to transform coordinates specified in `EPSG:4326` to `EPSG:3857`:

```
1     var point = new ol.geom.Point(
2         ol.proj.transform([3,50], 'EPSG:4326', 'EPSG:3857')
3     );
```

After creating the geometries we have create a new `ol.Feature` instance for each one:

```
1     var pointFeature = new ol.Feature(point);
```

As we explained in the Creating features by hand section at Vector layers chapter, we specify the features in two ways, passing the geometry reference directly, like in our code, or passing an object with a `geometry` property plus any other desired property. So, if we would like to attach a `name` property to our feature we could made something like:

```
1     var pointFeature = new ol.Feature({
2         geometry: point,
3         name: 'This is the name'
4     });
```

Because we are not reading features from a data source, we do not need to use a concrete source, like `ol.source.GeoJSON`. We use the base class `ol.source.Vector`, which as any source acts as a container of features.

```
1     var vectorSource = new ol.source.Vector({
2         projection: 'EPSG:4326'
3     });
4     vectorSource.addFeatures([pointFeature, circleFeature]);
```

Note, instead using the `adFeatures()` method we could initialize the source passing an array of features to the property `features`:

```
    var vectorSource = new ol.source.Vector({
        projection: 'EPSG:4326',
        features: [pointFeature, circleFeature]
    });
```

After this, we have created a vector layer passing the previous source and added it to a map. All these is well known by the samples at previous chapters.

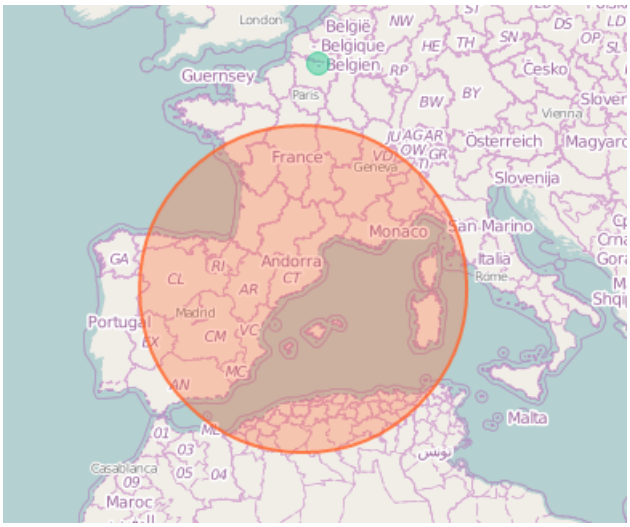
4.6.2.4 See also

- Different ways to load data using a vector source example at Data sources and formats chapter to see how load vector data using concrete source classes.
- Basic styling example at Vector layers chapter, to see how apply basic styling to features.
- Managing features example at Vector layers chapter, to see how we can handle the set of features contained within a source.

4.6.3 Basic styling

4.6.3.1 Goal

This recipe shows how to set the fill and stroke properties of the style.



Basic styling

4.6.3.2 How to do it...

As always create a DOM element where to attach the map and add the next JavaScript code.

First create the geometries, the features and a source to attach them:

```
    // Geometries
    var point = new ol.geom.Point(
        ol.proj.transform([3,50], 'EPSG:4326', 'EPSG:3857')
    );
    var circle = new ol.geom.Circle(
        ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        1000000
    );

    // Features
    var pointFeature = new ol.Feature(point);
    var circleFeature = new ol.Feature(circle);

    // Source and vector layer
    var vectorSource = new ol.source.Vector({
        projection: 'EPSG:4326',
        features: [pointFeature, circleFeature]
    });
```

Define a style and attach it to a vector layer with the previous source:

```
    var style = new ol.style.Style({
        fill: new ol.style.Fill({
            color: 'rgba(255, 100, 50, 0.3)'
        }),
        stroke: new ol.style.Stroke({
            width: 2,
            color: 'rgba(255, 100, 50, 0.8)'
        }),
        image: new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(55, 200, 150, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(55, 200, 150, 0.8)'
            }),
            radius: 7
        }),
    });

    // Vector layer
```

```
    var vectorLayer = new ol.layer.Vector({
        source: vectorSource,
        style: style
    });
```

Finally, create the maps instance with the vector layer:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 4
        })
    });
```

4.6.3.3 How it works...

Because map's view uses 'EPSG:3857' projection we need to create geometries in this projection. In our code, we specified coordinates in 'EPSG:4326' and transformed to 'EPSG:3857' with the `ol.proj.transform()` function.

For the style, we can set the properties `fill`, `stroke` and `image` which offers the necessary to render points, lines and polygons:

```
    var style = new ol.style.Style({
        fill: new ol.style.Fill({
            color: 'rgba(255, 100, 50, 0.3)'
        }),
        stroke: new ol.style.Stroke({
            width: 2,
            color: 'rgba(255, 100, 50, 0.8)'
        }),
        image: new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(55, 200, 150, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(55, 200, 150, 0.8)'
            }),
            radius: 7
        }),
    });
```

Note the `image` property is used to render point features in an efficient way. Internally a circle is drawn in a canvas element, exported to an image and reused for each point feature that exists in the layer (if it no overrides the layer style). On its way, `fill` and `stroke` are used to render lines and polygons.

We are specifying the colors using the `rgba` syntax format offered by OpenLayers3 where we can set the red, green, blue and alpha (transparency) factors for the colors.

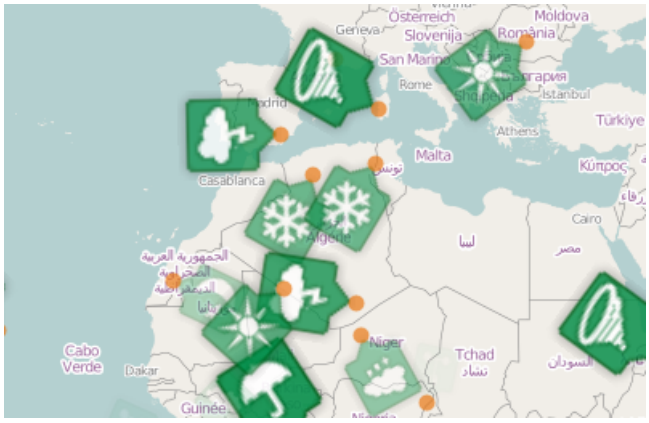
4.6.3.4 See also

- Markers: Styling features with icons example at Vector layers chapter, to see how style features with icons.
- Using text to style features example at Vector layers chapter, to know how to apply text on styles.
- Working with style functions example at Vector layers chapter, to see how determine the feature's style through using a *style function*.

4.6.4 Markers: Styling features with icons

4.6.4.1 Goal

In this recipe we are going to show how to use icons to style features. For this purpose, we are going to create a map with some random icons using random values on some of its properties, like the rotation angle, anchor position, etc.



Using icons to style features

In addition, we will see how we can apply more than a style to a feature that, in this case, allows us to render an icon plus a point to indicate where the real geometry coordinates are locates.

4.6.4.2 How to do it...

As always, add a HTML element where to attach the map instance and add the next JavaScript code.

Define an array with the set of available icons to be used:

```
    var icons = [
        "images/mapiconscollection-weather/anemometer_mono.png",
        "images/mapiconscollection-weather/cloudy.png",
        "images/mapiconscollection-weather/cloudysunny.png",
        "images/mapiconscollection-weather/moonstar.png",
        "images/mapiconscollection-weather/rainy.png",
        "images/mapiconscollection-weather/snowy-2.png",
        "images/mapiconscollection-weather/sunny.png",
        "images/mapiconscollection-weather/thunderstorm.png",
        "images/mapiconscollection-weather/tornado-2.png",
        "images/mapiconscollection-weather/umbrella-2.png",
        "images/mapiconscollection-weather/wind-2.png"
    ];
```

Next generate some random features, at random locations and using random values for its properties:

```
    var features = [], i, lat, lon, geom, feature, features = [], style, rnd;
    for(i=0; i< 200; i++) {
        lat = Math.random() * 174 - 87;
        lon = Math.random() * 360 - 180;

        geom = new ol.geom.Point(
            ol.proj.transform([lon, lat], 'EPSG:4326', 'EPSG:3857')
        );

        feature = new ol.Feature(geom);
        features.push(feature);

        rnd = Math.random();
        style = [
            new ol.style.Style({
                image: new ol.style.Icon(({
                    scale: 1 + rnd,
                    rotateWithView: (rnd < 0.9) ? true : false,
                    rotation: 360 * rnd * Math.PI / 180,
                    anchor: [0.5, 1],
                    anchorXUnits: 'fraction',
                    anchorYUnits: 'fraction',
                    opacity: rnd,
                    src: icons[ Math.floor(rnd * (icons.length-1) ) ]
                }))
            }),
            new ol.style.Style({
                image: new ol.style.Circle({
                    radius: 5,
                    fill: new ol.style.Fill({
                        color: 'rgba(230,120,30,0.7)'
                    })
                })
            })
        ];

        feature.setStyle(style);
    }
```

Now create a vector source, initialized with the previous features, and create a vector layer:

```
    var vectorSource = new ol.source.Vector({
        features: features
    });

    var vectorLayer = new ol.layer.Vector({
        source: vectorSource
    });
```

Finally, create a map instance:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 3
        }),
        interactions: ol.interaction.defaults().extend([
            new ol.interaction.DragRotateAndZoom()
        ])
    });
```

4.6.4.3 How it works...

We need to have in mind icon styling only works for point features, that is, features with using an `ol.geom.Point` or `ol.geom.MultiPoin` features. In addition, styles can be attached to a single feature (so each feature can have its own style) or to a layer (so that it applies to all layer features).

For each feature we have attached two styles, the first to render an icon and second to show the exact point where the feature's geometry is placed.

```
    rnd = Math.random();
    style = [
        new ol.style.Style({
            image: new ol.style.Icon(({
                scale: 1 + rnd,
                rotateWithView: (rnd < 0.9) ? true : false,
                rotation: 360 * rnd * Math.PI / 180,
                anchor: [0.5, 1],
                anchorXUnits: 'fraction',
                anchorYUnits: 'fraction',
                opacity: rnd,
                src: icons[ Math.floor(rnd * (icons.length-1) ) ]
            }))
        }),
        new ol.style.Style({
            image: new ol.style.Circle({
                radius: 5,
                fill: new ol.style.Fill({
                    color: 'rgba(230,120,30,0.7)'
                })
            })
        })
    ];

    feature.setStyle(style);
```

Because we want to play with the different `ol.style.Icon` class properties we have created a different style for each feature. To do so, we obtain a random value to compute each property value.

Givent the `rnd` value we take it as degree value and transform to radians to be applied at the `rotation` property. The `rotationWithView` property takes the value `true` if the `rnd` value is less than `0.9`, otherwise it is `false`.

Because the `rnd` value is between the range `0..1` we use it directly as the `opacity` property.

For the `anchor` property we have set the `anchorXUnits` and `anchorYUnits` to `fraction`. The `anchor: [0.5, 1]` means the icons is placed centered-top respect the geometry location.

Note, to see the effect of `rotateWithView` property we have attached to the map an `ol.interaction.DragRotateAnd2` interaction, which allow to rotate the map when the *shift* is pressed:

```
    ...
    interactions: ol.interaction.defaults().extend([
        new ol.interaction.DragRotateAndZoom()
    ])
    ...
```

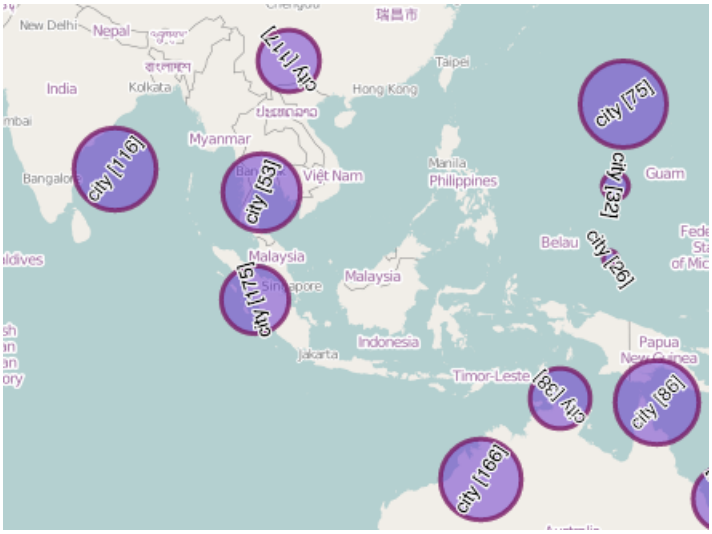
4.6.4.4 See also

- Basic styling example at Vector layers chapter, to see how to work with style properties fill, stroke and image.
- Using text to style features example at Vector layers chapter, to know how to apply text on styles.
- Working with style functions example at Vector layers chapter, to see how determine the feature's style through using a *style function*.
- Styling features under the pointer example at Events, listeners and properties chapter, to see how we can change style on features under the pointer.

4.6.5 Using text to style features

4.6.5.1 Goal

On this recipe we will show how we can use text to style our features. Concretely we are going to create some random point features with two properties `radius` and `name` that will be used within its style.



Text styling

4.6.5.2 How to do it...

First we are going to define a `computeFeatureStyle` function that given a feature it will return a `ol.style.Style` instance:

```
    function computeFeatureStyle(feature) {
        return new ol.style.Style({
            image: new ol.style.Circle({
                radius: feature.get('radius'),
                fill: new ol.style.Fill({
                    color: 'rgba(100,50,200,0.5)'
                }),
                stroke: new ol.style.Stroke({
                    color: 'rgba(120,30,100,0.8)',
                    width: 3
                })
            }),
            text: new ol.style.Text({
                font: '12px helvetica,sans-serif',
                text: feature.get('name'),
                rotation: 360 * rnd * Math.PI / 180,
                fill: new ol.style.Fill({
                    color: '#000'
                }),
                stroke: new ol.style.Stroke({
                    color: '#fff',
                    width: 2
                })
            })
        });
    }
```

Next, we are going to create a bunch of random point features that will use the previous function to compute the style to be used:

```
    var i, lat, lon, geom, feature, features = [], style, rnd;
    for(i=0; i< 200; i++) {
        lat = Math.random() * 174 - 87;
        lon = Math.random() * 360 - 180;

        geom = new ol.geom.Point(
            ol.proj.transform([lon, lat], 'EPSG:4326', 'EPSG:3857')
        );

        rnd = Math.random();
        feature = new ol.Feature({
            geometry: geom,
            radius: rnd * 30,
            name: 'feature [' + i + ']'
        });
        features.push(feature);

        style = computeFeatureStyle(feature);
        feature.setStyle(style);
    }
```

Next create a source and a vector layer to contains the previous features:

```
    var vectorSource = new ol.source.Vector({
        features: features
    });

    var vectorLayer = new ol.layer.Vector({
        source: vectorSource
    });
```

Finally, create a map instance to show the vector layer in addition to a OpenStreetMap raster layer to act as base layer:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 2
        })
    });
```

4.6.5.3 How it works...

We have created features setting two properties, `name` and `radius`, at initialization time. The first will be used for the text to be used as the message to be shown while the second determines the point radius size:

```
    ...
    feature = new ol.Feature({
        geometry: geom,
        radius: rnd * 30,
        name: 'feature [' + i + ']'
    });
    ...
```

For each feature we have created an style instance passing the feature to the `computeFeatureStyle` function. The created style sets the `image` property, which is used to render point features, to an `ol.style.Circle` with the feature's radius:

```
    ...
    image: new ol.style.Circle({
        radius: feature.get('radius'),
        fill: new ol.style.Fill({
            color: 'rgba(100,50,200,0.5)'
        }),
        stroke: new ol.style.Stroke({
            color: 'rgba(120,30,100,0.8)',
            width: 3
        })
    }),
    ...
```

For the `text` property we have set the text message based on the feature's `name` property. In addition we have set a random rotation angle (in radians):

```
    ...
    text: new ol.style.Text({
        font: '12px helvetica,sans-serif',
        text: feature.get('name'),
        rotation: 360 * rnd * Math.PI / 180,
        fill: new ol.style.Fill({
            color: '#000'
        }),
        stroke: new ol.style.Stroke({
            color: '#fff',
            width: 2
        })
    })
    ...
```

4.6.5.4 See also

- Basic styling example at Vector layers chapter, to see how to work with style properties fill, stroke and image.
- Working with style functions example at Vector layers chapter, to see how determine the feature's style through using a *style function*.
- Markers: Styling features with icons example at Vector layers chapter, to see how style features with icons.

4.6.6 Working with style functions

4.6.6.1 Goal

The goal of this recipe it to show how we can use a so called *style function* to determine the style of our features.

Style function example

In this example we are going to load most important cities from a GeoJSON file and will render them using a text style with a different font size depending on the resolution.

4.6.6.2 How to do it...

Let's start defining our custom style function. It will be applied to the vector layer so it will receive two arguments: a reference to the `feature` to be renderer and the current map's view `resolution`.

What we desire is to render the `CITY_NAME` property with small font size when map is far (that is for higher resolution values) and gradually use a bigger font when map is closer (lower resolutions).

```
1     var customStyleFunction = function(feature, resolution) {
2         var fontSize = '18';
3         if(resolution>=39134) {
4             fontSize = '10';
5         } else if(resolution>=9782) {
6             fontSize = '14';
7         } else if(resolution>=2444) {
8             fontSize = '16';
9         }
10
```

```
        return [new ol.style.Style({
            text: new ol.style.Text({
                font: fontSize + 'px sans-serif,helvetica',
                text: feature.get('CITY_NAME'),
                fill: new ol.style.Fill({
                    color: '#'+Math.floor(Math.random()*16777215).toString(16)
                }),
                stroke: new ol.style.Stroke({
                    color: '#ddd',
                    width: 1
                })
            })
        })];
    };
```

Next, we need to create the source responsible to load data from a GeoJSON file and the vector layer:

```
    var vectorSource = new ol.source.GeoJSON({
        url: './data/world_cities.json',
        projection: 'EPSG:3857'
    });

    var vectorLayer = new ol.layer.Vector({
        source: vectorSource,
        style: customStyleFunction
    });
```

And finally create the map instance. We will add an OpenStreetMap layer to act as base layer.

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
```

```
            zoom: 2
        })
    });
```

4.6.6.3 How it works...

Because we have passes a function to the vector layer `style` property the function must conform the signature specified by `ol.feature.FeatureStyleFunction`.

```
    var vectorLayer = new ol.layer.Vector({
        source: vectorSource,
        style: customStyleFunction
    });
```

The function receives the current `feature` to be rendered and the current `resolution` of the map's view. With these information the first step is to compute the right font size to be used:

```
    var customStyleFunction = function(feature, resolution) {
        var fontSize = '18';
        if(resolution>=39134) {
            fontSize = '10';
        } else if(resolution>=9782) {
            fontSize = '14';
        } else if(resolution>=2444) {
            fontSize = '16';
        }
        ...
    };
```

Once we have the font size we want to use, we need create the right `ol.style.Style` instance and setting the `text` property with the value of the `CITY_NAME` property from the feature. In addition, we set the text color as a random to color:

```
var customStyleFunction = function(feature, resolution) {
    ...
    return [new ol.style.Style({
        text: new ol.style.Text({
            font: fontSize + 'px sans-serif,helvetica',
            text: feature.get('CITY_NAME'),
            fill: new ol.style.Fill({
                color: '#'+Math.floor(Math.random()*16777215).toString(16)
            }),
            stroke: new ol.style.Stroke({
                color: '#ddd',
                width: 1
            })
        })
    })];
};
```

The function must return an array of styles to be used when rendering the feature. In our case only one style.

Note, this way to works is not very efficient. Each time a feature is going to be rendered, the `customStyleFunction` is called, the font size is computed and a new style instance is created and returned.

A good exercise would be to implement some kind of cache. For example, for each style generated using a resolution value, we could store it in an object like `cache[resolution] = style;`. This way, each time a feature is going to be rendered the code would look if there is a style pre-generated and stored in the cache or, otherwise, will create, store in the cache and return it.

4.6.6.4 See also

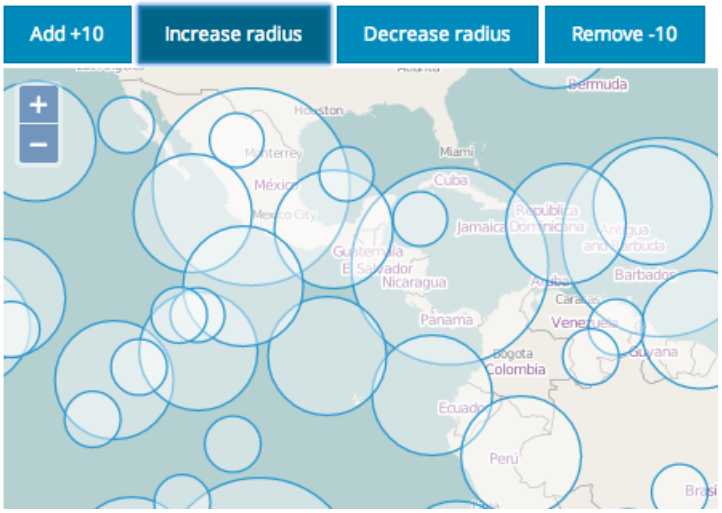
- Markers: Styling features with icons example at Vector layers chapter, to see how style features with icons.
- Using text to style features example at Vector layers chapter, to know how to apply text on styles.

4.6.7 Managing features

4.6.7.1 Goal

When working with vector data we need a way to manage the set of features that belongs to a given vector layer, adding, removing or modifying them.

This recipe demonstrate in a simple example the use of some of the available `ol.source.Vector` methods available for this purpose. We will create an initially empty map with four option buttons that will allow the user to add ten new circle features, increase the radius of the current circles, decrease the radius or remove ten features.



Managing features programmatically

4.6.7.2 How to do it...

Start creating an empty vector source, vector layer and map. Note, we are using a raster `ol.source.OSM` layer as base layer:

```
var vectorSource = new ol.source.Vector({
    projection: 'EPSG:4326'
});

var vectorLayer = new ol.layer.Vector({
    source: vectorSource
});

var map = new ol.Map({
    target: 'map',  // The DOM element that will contains the map
    renderer: 'canvas', // Force the renderer to be used
    layers: [
        // Add a new Tile layer getting tiles from OpenStreetMap source
        new ol.layer.Tile({
            source: new ol.source.OSM()
        }),
        vectorLayer
```

```
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 2
        })
    });
```

Now, for each button attach a listener that will invoke the function that will make some action:

```
    $('#add').on('click', function(){
        addFeatures();
    });
    $('#remove').on('click', function(){
        removeFeatures();
    });
    $('#increase').on('click', function(){
        increaseRadius(30000);
    });
    $('#decrease').on('click', function(){
        increaseRadius(-30000);
    });
```

Add the code for the function responsible to add new random circle features:

```
    function addFeatures() {
        var i, lat, lon, geom, feature, features = [];
        for(i=0; i< 10; i++) {
            lat = Math.random() * 174 - 87;
            lon = Math.random() * 360 - 180;

            geom = new ol.geom.Circle(
                ol.proj.transform([lon, lat], 'EPSG:4326', 'EPSG:3857'),
                100000
            );

            feature = new ol.Feature(geom);
            features.push(feature);
        }
        vectorSource.addFeatures(features);
        return features;
    }
```

The function to change the circle radius:

```
    // Increase circle feature radius. Negative values decreases.
    function increaseRadius(inc) {
        var features = vectorSource.getFeatures();
        var geom;

        // Note we use 'getFeatures()' because with forEachFeature we
        // can not modify feature's geometry or will get a
        // 'cannot update extent while reading' error.
        for(var i=0; i< features.length; i++) {
            geom = features[i].getGeometry();
            geom.setRadius( geom.getRadius() + inc);
        }
    }
```

and, finally, to remove remove features:

```
    function removeFeatures() {
        var features = vectorSource.getFeatures();
        for(var i=0; i< features.length && i<10; i++) {
            vectorSource.removeFeature(features[i]);
        }
    }
```

4.6.7.3 How it works...

We have used jQuery to register listener functions when buttons receive the `click` event. These functions are responsible to invoke the method to add, remove or increase the circle features.

Within the `addFeatures()` method we have created circle features at random positions. For simplicity we have created random latitude and longitude positions using `EPSG:4326` coordinate system:

```
    lat = Math.random() * 174 - 87;
    lon = Math.random() * 360 - 180;
```

Because the map's view (and the OSM layer) uses `EPSG:3857`, the feature's geometry must be specified in that coordinate system, so we need to transform them from `EPSG:4326` to `EPSG:3857` with the `ol.proj.transform()` method:

```
    geom = new ol.geom.Circle(
        ol.proj.transform([lon, lat], 'EPSG:4326', 'EPSG:3857'),
        100000
    );
```

All the features created are stored in an array and added to the source with the `addFeatures()` method. For performance reasons, it is better to attach a set of features at a time than adding features one by one with `addFeature()`.

Internally, the `ol.source.Vector` class triggers a `change` event each time the set of contained features changes. Adding features with the `addFeature()` method implies we are firing the `change` event for each added feature, while adding all the features in an array with `addFeatures()` means we are only firing the `change` event one time.

The `removeFeatures()` function is relatively simple. We get a references of the source's features and loop through them removing the ten first:

```
    function removeFeatures() {
        var features = vectorSource.getFeatures();
        for(var i=0; i< features.length && i<10; i++) {
            vectorSource.removeFeature(features[i]);
        }
    }
```

To increase and decrease the radius of circle features we have created the `increaseRadius()` function, which modifies the radius size given a value. If the `inc` value is negative the circle radius will decrease.

The function's code loops through the source's features, retrieves a reference to the feature's geometry, which is an `ol.geom.Circle`, and modifies its radius with the `setRadius()` method:

```
    function increaseRadius(inc) {
        var features = vectorSource.getFeatures(), geom;
        for(var i=0; i< features.length; i++) {
            geom = features[i].getGeometry();
            geom.setRadius( geom.getRadius() + inc);
        }
    }
```

While `forEachFeature()` method is preferable to make massive feature modifications, here it is not an option. OpenLayers3 do not allow to modify those properties that affect the source while we are traveling its contents. We will get an error similar to `cannot update extent while reading`. In this sample, the issue is we want to modify the feature's geometry and this can change the computed source extent.

4.6.7.4 See also

- Creating features programmatically example at Vector layers chapter, to see how we can create features programmatically instead loading them through a source instance.

5. Events, listeners and properties

As an event driven programming[1] language, the flow of JavaScript programs is determined by events. These events can be triggered by a mouse click, a key press or when a web page is completely loaded.

When developing a web mapping application we need the framework to trigger events related to the mapping components too, for example, when map is panned (so we can know the new map extent), when mouse is clicked (to get the clicked coordinate), when a layer is loaded (to inform the user data loading process is complete), etc.

In this chapter we will describe the events triggered by the most important components of OpenLayers3 and how we can work with them through the so called listener functions. In the explanation, we will see the hierarchy, on which the OpenLayers3 components are based, and how we can work with the properties of these objects.

5.1 Introducing event driven paradigm in OpenLayers3

When programming in an event driven paradigm objects must be able to do two main things:

1. Allow to register (and unregister) so called *listener* functions (or *callbacks*), so they were invoked when an event is triggered, and
2. Trigger events, that is, invoke any listener function attached to it.

By convention, listener functions always receive an *event* object with all the information of interest we can require to handle the event.

As example, in the next code we register a function on a `map` object listening for the `singleclick` event, which indicates the mouse has clicked at some location. As we can see, the listener function receives an event object that includes the coordinates clicked:

```
1    var map = new ol.Map({ ... });
2    map.on('singleclick', function(event) {
3        var c = event.coordinate;
4        ...
5    });
```

[1] http://en.wikipedia.org/wiki/Event-driven_programming

5.2 Where the events and listeners comes from?

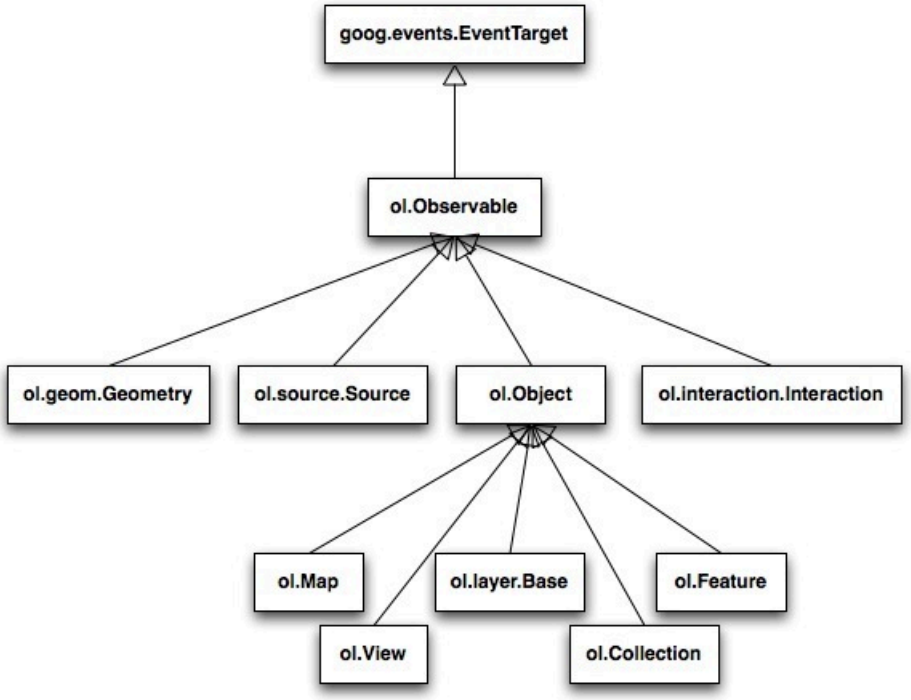
OpenLayers3 makes extensive use of the Google Closure Tools[2]. The JavaScript code is based on the Closure Library[3], which offers classes inheritance, data structures, DOM manipulation plus a large set of features, and later it is compiled through the Closure Compiler[4], which compiles from JavaScript to JavaScript minimizing the code rewriting functions, removing death code and also checking syntax, variable references and types.

The explanation of Closure Tools is out of the scope of this book. In this section we simply introduce some concepts so the reader can begin to understand how things are internally implemented in OpenLayers3.

It is not a surprise all the fundamentals related to events and listeners where based on classes from Google Closure Library, like `goog.events.Event` and `goog.events.EventTarget`. In OpenLayers3, all event objects passed to listener functions inherits from the `goog.events.Event` class and objects able to listen or trigger events are subclasses of `goog.events.EventTarget`.

OpenLayers3 defines two main classes, which conforms the root of all the components hierarchy: `ol.Observable` and `ol.Object`.

[2] https://developers.google.com/closure/
[3] https://developers.google.com/closure/library/
[4] https://developers.google.com/closure/compiler/



Classes hierarchy from `ol.Observable`

The `ol.Observable` class (which inherits from `goog.events.EventTarget`) implements the mechanism required to allow register listener functions and to dispatch events (see the Listening for changes section).

On its way, the `ol.Object` class inherits from `ol.Observable` and extents it allowing associate properties to the object and listening for changes on them (see Working with object properties section).

Based on this couple of classes, most important components within OpenLayers3 inherits from them. This way `ol.geom.Geometry`, `ol.source.Source`, `ol.interaction.Interaction` or `ol.Object` inherits from `ol.Observable` and because of this they are able to emit events and register listener functions. Other components, requires additional properties to be attached to the instances or listen for certain property changes, because of this they inherits from `ol.Object`. This is the case of `ol.Map`, `ol.View`, `ol.layer.Base` and many more classes.

5.2.1 Listening for changes in `ol.Observable` instances

We can register and unregister listener functions on any subclass of `ol.Observable`. The base class defines the next set of methods that helps us on this purpose:

- `on()`, registers a listener function for a particular event or an array of events.

- `once()`, similar to `on()` but the listener is automatically unregistered once an event is triggered, that is, the listener function is invoked only once.
- `un()`, unregisters a given listener function on a particular event.
- `unByKey()`, similar to `un()` but requires we pass a *key* reference returned by the `on()` or `once()` methods.

Next code show the previous methods in action:

```
var map = new ol.Map({ ... });

// Listener function for 'click' events
var clickHandler = function(event) {
    // Invoked each time mouse is clicked on the map
};

// Register 'click' listener
map.on('click', clickHandler);

// Register 'dblclick' anonymous function
var key = map.on('dblclick', function(event) {
    // Invoked each time mouse is double clicked on the map
});

// Register a listener only once
map.once('singleclick', function(event) {
    // Invoked only once
});

// Unregister 'click' listener
map.un('click', clickHandler);

// Unregister using the 'key'
map.unByKey(key);
```

In addition to register events one by one, we can specify an array of events to be registered invoking the same listener function:

```
map.on(['click', 'dblclick'], function(event) {
    // Invoked when 'click' or 'dblclick' events are fired.
})
```

5.3 Working with object properties

As we commented, all the subclasses of `ol.Object` can have custom properties. Ideally, `ol.Object` was designed to act as an abstract class, that is, a class that can not have direct instances, so you will probably never need to create an instance but, as the root class, all the concepts explained here are applicable to its subclasses.

To work with properties `ol.Object` defines the next list of methods:

- `get()`, returns the value associated to a given property name.
- `getKeys()`, obtains a list with the property names.
- `getProperties()`, obtains a list of property names and values.
- `set()`, sets a given property name with a given value.
- `setProperties()`, sets a bunch of properties at once.

Do not confuse JavaScipt object properties with OpenLayers3 object properties.

Usually, objects are instantiated passing a set of properties:

```
1    var person = new ol.Object({
2        name: 'antonio',
3        surename: 'santiago'
4    });
```

but we can set or get new properties after initialization, one by one with `get()/set()`:

```
1    person.set('twitter', '@acanimal');
2
3    var twitter = person.get('twitter');
```

or many at once with `getProperties()/setProperties()`:

```
    person.set({
        name: 'Chuck',
        surename: 'Norris',
        twitter: '@chucknorris'
    });

    var props = person.getProperties();
```

In addition to the previous properties, the `ol.Object` class implements the next functions, that increases the inherited methods of `ol.Observable` related with listener functions:

- `bindTo()`, bind a property on the current object (the source) to a property on a target instance.
- `unbind()`, removes a binding on a given property.
- `unbindAll()`, removes all the properties bindings.

Additionally, we can force to trigger a property with the `dispatchEvent()` method.

5.3.1 Events in the `ol.Object` properties

`ol.Object` class triggers next three kind of events we can listen for:

- `beforepropertychange`, emitted before a property is changed,
- `propertychange`, triggered after any property is changed, and
- `change:*`, triggered when the property specified by the asterisk is changed.

From here, each subclass can trigger additional events but all properties modifications are covered by these events.

This way, given the next object instance:

```
    var obj = new ol.Object({
        name: 'some value here',
        password: 123456,
        age: 32
    });
```

we can listen for changes at the `password` property the `change:*` event like:

```
obj.on('change:password', function(evt) {
    console.log('-> password changed to: ' + evt.target.get('password'))
});
```

or listen for changes at any property with the `propertychange` event:

```
obj.on('propertychange', function(evt) {
    console.log('* Property changed: ', evt.key);
});
```

Although the `beforepropertychange` and `propertychange` can be similar, we can use the first to be notified before a property value changes and, for example, backup its previous value so we can undo in certain situations.

5.3.2 Binding properties between objects

As we seen previously, the `ol.Object` offers methods to bind properties among instances. Binding is a two way listening mechanism that can be extremely useful, for example, to maintain two instances synchronized.

The method `bindTo()` requires three parameters:

- `key`, the key name in the source object (the current instance) to be binded,
- `target`, the target object to bind keys
- `opt_targetKey`, an optional key name where to bind the source key.

Remember only one binding is allowed per key.

The next code shows the bind and unbind mechanism in action. We are going to bind the age properties of two instances:

```
var john = new ol.Object({
    name: 'John Smith',
    age: 32
});
var peter = new ol.Object({
    name: 'Peter Smith',
    age: 32
});

// Bind the 'age' property between john and peter
john.bindTo('age', peter);

// Change the 'age' property. Both john and peter must change.
john.set('age', 33);

// Unbind property
john.unbind('age');
```

When we create a binding between two instances a listener on the target key and instance is registered, this way if the target changes the source is notified and can update its key.

The `bindTo()` method returns a reference to an `ol.ObjectAccessor` instance. This class simply offers the `transform(from, to)` method, which accepts two functions, where **we can take control the way in which the values are assigned between the binded property**.

Take care and not confuse with the meaning of this functions. Each time there is a change in the source key, the `from()` function determines the value that goes *from* the source to the target. On the other hand, when there is a change in the target the `to()` function determines the value that goes from the target *to* the source.

Following the same previous example, next code shows how we can made *john* always was 5 year old greater than *peter*:

```
var accessor = john.bindTo('age', peter);
accesor.transform(
    function(sourceAge) {
        // Value from john to peter
        return sourceAge - 5;
    },
    function(targetAge) {
        // Value from peter to john
        return targetAge * 5;
    }
);
```

5.4 OpenLayers3 components events

As we have see, most of components within OpenLayers3 inherits from `ol.Object` class and many of them triggers events different by those inherited by the root class. As example, we found the `ol.source.Vector` class, the root of the vector sources, that fires `addFeature` or `removeFeature` event when a feature is added to or removed from the source.

All the events and properties of OpenLayers3 are well documented in the API[5] documentation, so there is no need to make an exhaustive description of each one when there is a place well maintained and updated.

As developers, understanding the previous sections are the key to understand how to work with events, listeners and properties. Once understood we can play with any object property or event.

I encourage the reader to see the samples related with this chapter where he/she can see real day usages examples, like, synchronizing two map view locations or know when a source has finished loading data.

5.4.1 A word about `ol.source.Source` events

As we commented at the A word about events section on Vector layers chapter, the source classes requires some time to read the data contents. That means the source instances must have different states so we can know if source is working or it has finished and we can work with the data. Ideally, all sources should be able to change its state among something like:

- `loading`, to indicate the source is working loading data,
- `ready`, to indicate the data is ready to be used,
- `error`, so we can know if there was any error reading data.

For this purpose, the `ol.source.Source` class has the `getState()` method that returns the current source status.

Additionally, each time a source changes its state, it should fire a `change` event, to indicate something has changed. This way we could know when the source starts reading and when stops processing data.

Unfortunately, while I write this lines, OpenLayers3 is not able to fire the `change` event for each change on a source's state. Only subclasses of `ol.source.Vector` triggers `change` event when data is loaded. See the Listening for changes on vector data sample.

Eric Lemoine made a clarification on this issue in this post[6] of the OpenLayers3 discussion group.

[5] http://openlayers.org/en/v3.0.0/apidoc/

[6] https://groups.google.com/forum/#!topic/ol3-dev/9YqTvjnDFdw

Finally, note when a source fires a `change` event, it is catched by the vector layer it belongs to and, the layer, triggers its own `change` event indicating the content of its source has changed. So, we can register a lister function both to the layer or to the source.

5.5 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[7] repository.

5.5.1 Events, listeners and properties

5.5.1.1 Goal

This recipes shows the basics to understand and work with events, listeners and properties.

5.5.1.2 How to do it...

The goal is to create some instance of `ol.Object` and play with its methods and properties. Start creating a new `ol.Object` instance and printing some information by browser console:

```
var obj = new ol.Object({
    name: 'some value for property',
    age: 32
});
obj.set('password', 123456);

console.log('I\'m an object with custom properties: ');
console.log(' > keys: ', obj.getKeys());
console.log(' > key/value: ', obj.getProperties());
```

Next register some listeners and make some modification to see the listeners in action:

```
obj.on('beforepropertychange', function(evt) {
    console.log('* Before property change event: ', evt.key);
});
obj.on('propertychange', function(evt) {
    console.log('* Property change event: ', evt.key);
});

obj.on('change:name', function(evt) {
    console.log('-> name changed to: ' + evt.target.get('name'));
```

[7] https://github.com/acanimal

```
    });

    obj.on('change:password', function(evt) {
        console.log('-> password changed to: ' + evt.target.get('password'))
    });

    obj.once('change:age', function(evt) {
        console.log('-> age changed to: ' + evt.target.get('age') + '. This will\
 not be notified again.');
    });

    // Change some attributes
    console.log('lets go to change the name property...');
    obj.set('name', 'new name');

    console.log('lets go to change the password property...');
    obj.set('password', 000000);

    console.log('lets go to change the age property...');
    obj.set('age', 25);

    console.log('lets go to change the age property again. Nothing must happen.'\
);
    obj.set('age', 35);
```

Finally, create two new instances of `olObject` that will represent brothers, john and peter, bind its age property and make some change to see the effect:

```
    // Create two new instances
    var john = new ol.Object({
        name: 'John Smith'
    });
    var peter = new ol.Object({
        name: 'Peter Smith'
    });

    // Define a binding for the 'age' property where john is always
    // 5 years greater than peter
    var accessor = john.bindTo('age', peter);
    accessor.transform(
        function(sourceAge) {
            // Value from john to peter
```

```
            return sourceAge - 5;
        },
        function(targetAge) {
            // Value from peter to john
            return targetAge + 5;
        }
    );

    // Make some changes
    console.log('set john to 30...')
    john.set('age', 30);
    showAges();

    console.log('set john to 40...')
    john.set('age', 40);
    showAges();

    console.log('set peter to 40...')
    peter.set('age', 40);
    showAges();

    function showAges() {
        console.log('john(' + john.get('age') + ') - peter(' + peter.get('age') \
+ ')');
    }
```

5.5.1.3 How it works...

Execute the code and see the browser console. The printed outputs are self explanatory. Next, there are some things we can highlight in the code.

The attributes of an `ol.Object` instance can be defined at construction time, passing an object, or later using the `set()` or `setProperties()` methods:

```
    var obj = new ol.Object({
        name: 'some value for property',
        age: 32
    });
    obj.set('password', 123456);
```

Depending on the kind of event the `event` parameter can contain deferent information. So, given the next listeners:

```
    obj.on('propertychange', function(evt) {
        console.log('* Property change event: ', evt.key);
    });

    obj.on('change:name', function(evt) {
        console.log('-> name changed to: ' + evt.target.get('name'));
    });
```

The `propertychange` listener receives an `evt` parameter with `key` property, which contains the name of the modified property, and `target` property, with a reference to the modified instance.

On the other hand, the `change:name` listener receives an `evt` parameter with only the `target` property. There is no need for the key because we know the key we have registered the listener on.

Finally, the binding between the `john` and `peter` instances allows to have their ages updated with a difference of five years:

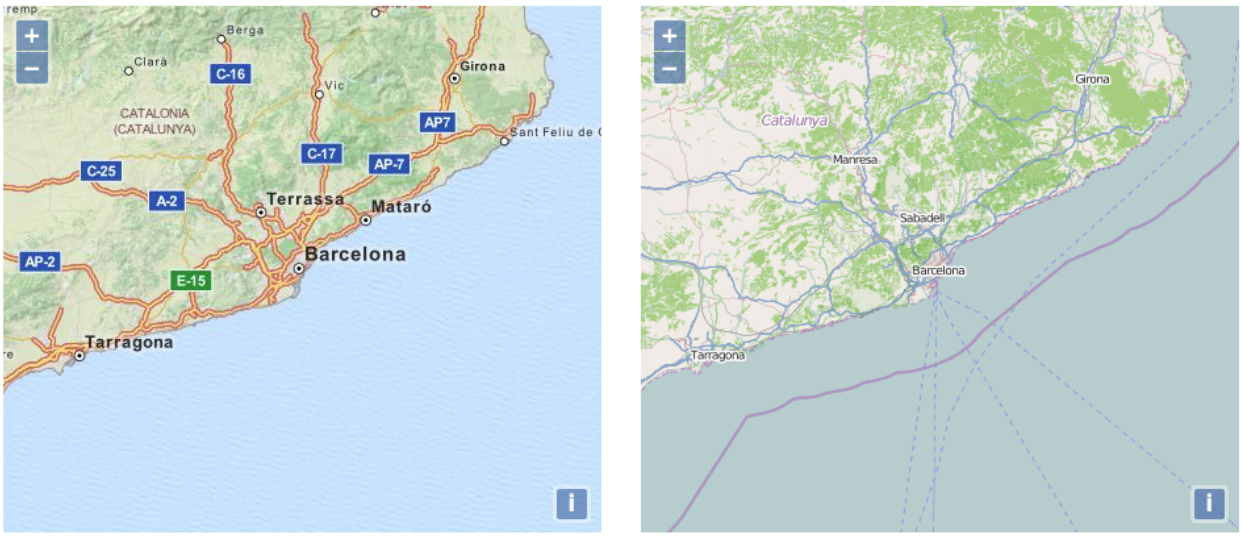
```
    var accessor = john.bindTo('age', peter);
    accessor.transform(
        function(sourceAge) {
            // Value from john to peter
            return sourceAge - 5;
        },
        function(targetAge) {
            // Value from peter to john
            return targetAge + 5;
        }
    );
```

Remember the first function (the `from` one) determines the value that goes *from* source to the target, while the second function (the `to` one) determines the value that goes from the target *to* the source.

5.5.2 Synchronize maps

5.5.2.1 Goal

This recipe demonstrate how easy we can synchronize properties among object with the binding mechanism. Concretely we are going to create two maps, side by side, with its `center` and `resolution` properties binded, so each movement in one map will be reflected in the other. In addition, a check box, will allow to chose if we want the maps make resolution twice in one map.



Synchronizing map views'

5.5.2.2 How to do it...

Start creating the necessary HTML code to layout the maps side by side (note the real work is made by the CSS classes) and the check box

```
    <div class="checkbox">
        <label>
            <input id="twice" type="checkbox"> Make resolution twice
        </label>
    </div>
    <div class="row">
        <div class="col-md-6"><div id="mapMQ" class="map"></div></div>
        <div class="col-md-6"><div id="mapOSM" class="map"></div></div>
    </div>
```

Now create the two map instances:

```
    var mapMQ = new ol.Map({
        target: 'mapMQ',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 8
        })
    });

    var mapOSM = new ol.Map({
        target: 'mapOSM',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 8
        })
    });
```

Add the code necessary to bind the `resolution` and `center` properties:

```
mapMQ.getView().bindTo('center', mapOSM.getView());
var accessor = mapMQ.getView().bindTo('resolution', mapOSM.getView());
accessor.transform(
    function(sourceResolution) {
        if( $('#twice').prop('checked') ) {
            return sourceResolution / 2;
        } else {
            return sourceResolution;
        }
    },
    function(targetResolution) {
        if( $('#twice').prop('checked') ) {
            return targetResolution * 2;
        } else {
            return targetResolution;
        }
    }
);
```

And add a listener function for the checkbox:

```
$('#twice').on('click', function() {
    mapOSM.render();
    mapMQ.render();
});
```

5.5.2.3 How it works...

Before to continue, note each map uses its own `ol.View` instance. So, two have both views synchronized we need to make both views have the same `center` and `resolution` values. This is easily achieve binding the properties:

```
mapMQ.getView().bindTo('center', mapOSM.getView());
var accessor = mapMQ.getView().bindTo('resolution', mapOSM.getView());
```

As we can see in the previous code, for the `center` property we do not make any special action but for `resolution` we get a reference to the `ol.ObjectAccessor` object so we can control how the binding process is made. The object has a `transform(from, to)` method which accepts two function we call `from` and `to`. The first determines the value that goes *from* the source to the target when the property changes in the source, while second determines the value that goes from the target *to* the source when the property is modified in the target.

Depending on the checkbox, we want both maps has the same resolution or one has double resolution:

```
    accessor.transform(
        function(sourceResolution) {
            if( $('#twice').prop('checked') ) {
                return sourceResolution / 2;
            } else {
                return sourceResolution;
            }
        },
        function(targetResolution) {
            if( $('#twice').prop('checked') ) {
                return targetResolution * 2;
            } else {
                return targetResolution;
            }
        }
    );
```

The binded properties only are notified if a change is made on this properties. This means if we check or uncheck the checkbox nothing happens in the maps, but we want the resolution of them changes. Because of this, we have registered a listener on the checkbox that forces map's refresh:

```
    $('#twice').on('click', function() {
        mapOSM.render();
        mapMQ.render();
    });
```

The `ol.Map` class has different methods suitable to refresh the map. `render()` forces to make a new rendering process. On the other hand `updateSize()` is designed to update the map if the size of the DOM element that contains the map changes.

5.5.2.4 See also

- Tile providers example at Data sources and formats chapter, to see how we can share the same `ol.View` instance among maps.

5.5.3 Showing the mouse location

5.5.3.1 Goal

This recipe shows how we can listen on the `ol.Map` for changes in the mouse position so we can get its current coordinates and print in two different projections.

Mouse position

5.5.3.2 How to do it...

Start adding the HTML code necessary to layout the application. We will place two labels, to show the current mouse position, on top of the map:

```
    <p>
        EPSG:3857 <span id="mouse3857" class="label label-info">0 / 0</span>
        EPSG:4326 <span id="mouse4326" class="label label-info">0 / 0</span>
    </p>
    <div id="map" class="map"></div>
```

Now create the map instance adding a raster layer from MapQuest provider:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
```

```
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 8
        })
    });
```

Finally add the code responsible to get and print the current mouse position:

```
    map.on('pointermove', function(event) {
        var coord3857 = event.coordinate;
        var coord4326 = ol.proj.transform(coord3857, 'EPSG:3857', 'EPSG:4326');

        $('#mouse3857').text(ol.coordinate.toStringXY(coord3857, 2));
        $('#mouse4326').text(ol.coordinate.toStringXY(coord4326, 4));
    });
```

5.5.3.3 How it works...

The code is relatively simple. After creating the map instance, we register a listener function on the `pointermove` event of the `ol.Map`.

```
    map.on('pointermove', function(event) {
        ...
    });
```

`pointermove`, in addition to other map events like `pointerdrag`, `click` or `dblclick`, invokes the listener functions passing an instance of the class `ol.MapBrowserEvent`. This instances, in addition to the `target` property contains two valuable properties we can use to handle the event: `coordinate` and `pixel`. Both contains the current mouse position, the first expressed in the current view's projection and, the second, in pixel units with origin at top-left map place.

Because we want to show the current mouse position in `EPSG:3857` and `EPSG:4326` we need to convert the current coordinate to `EPSG:4326` using the `ol.proj.transform()` method:

```
    var coord3857 = event.coordinate;
    var coord4326 = ol.proj.transform(coord3857, 'EPSG:3857', 'EPSG:4326');
```

Finally, we need to print the coordinates in the labels on top of the map. OpenLayers3 offers a set of functions within the `ol.coordinate` namespace (do not confuse with the interal class `ol.Coordinate` which represents an `x,y` coordinate) that allows to work with coordinates. Among the function we found the `ol.coordinate.toStringXY()` function that given a coordinates and a number it returns an string in the format `x, y` (that is, separated by a coma) using the specified number of decimals:

```
$('#mouse3857').text(ol.coordinate.toStringXY(coord3857, 2));
$('#mouse4326').text(ol.coordinate.toStringXY(coord4326, 4));
```

When programming with OpenLayers3 any `[x,y]` two element array can be used as a coordinate, so we could pass `event.pixel` two to the `ol.coordinate.toStringXY()` function and transform the current pixel location into a valid string.

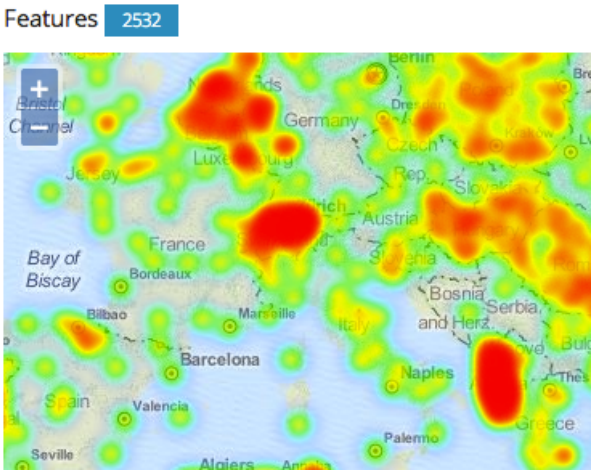
I encourage the reader to take a look to the `ol.coordinate` namespace API documentation and discover all the functions it contains. Among other you will find the powerful `ol.coordinate.format()` which allows to print a coordinate using any kind of template. Next code show an example of use:

```
var template = 'Coordinate: {x} / {y}';
var coordStr = ol.coordinate.format(coord4326, template, 2);
// coordStr value is like: 'Coordinate: 10.23 / 3.12'
```

5.5.4 Listening for changes on vector data

5.5.4.1 Goal

The goal of this recipe is to show how to ensure vector data is ready to work with them once loaded by a vector source. For this purpose, we are going to load a GeoJSON file and once loaded we will show the number of features.



Changes on vector data'

5.5.4.2 How to do it...

First we need to add the HTML code required to show the map and the label with the number of features:

```
    <p>
        Features <span id="features" class="label label-primary">0</span>
    </p>
    <div id="map" class="map"></div>
```

Now, create a vector layer that load a GeoJSON file:

```
    var layerVector = new ol.layer.Heatmap({
        source: new ol.source.GeoJSON({
            url: 'data/world_cities.json',
            projection: 'EPSG:3857'
        })
    });
```

Register a listener for the change event on the layer:

```
    layerVector.on('change', function(event) {
        $('#features').text(layerVector.getSource().getFeatures().length);
    });
```

Finally, create a map instance containing the previous layer plus a raster layer acting as base layer. In this case we create a tile layer using MapQuest as our data source:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            }),
            layerVector
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 4
        })
    });
```

5.5.4.3 How it works...

The code of this recipe is really simple. We have seen similar code in other examples so we do not spent time explaining it.

What is important from this example is to understand how sources works and the events they fire.

The `change` event is triggered by the `ol.source.Source` classes when the set of features it contains changes, for example, adding or removing them. In addition, the vector layer that uses the source catch the event and triggers its own `change` event, indicating to its listeners the source data has been changed. So, it is valid to register a listener both to the layer or to the source

In our sample we have create a vector layer loading GeoJSON data. If we want to know the number of features loaded be the source, we need to wait until the data has been read. This can be done registering a listener on the source `change` event:

```
    // Vector layer
    var layerVector = new ol.layer.Heatmap({
        source: new ol.source.GeoJSON({
            ...
        })
    });

    // Listen for changes
    layerVector.on('change', function(event) {
        $('#features').text(layerVector.getSource().getFeatures().length);
    });
```

When the `change` event is triggered it is safe to read the features, otherwise we can get an invalid value. For example, next code is incorrect:

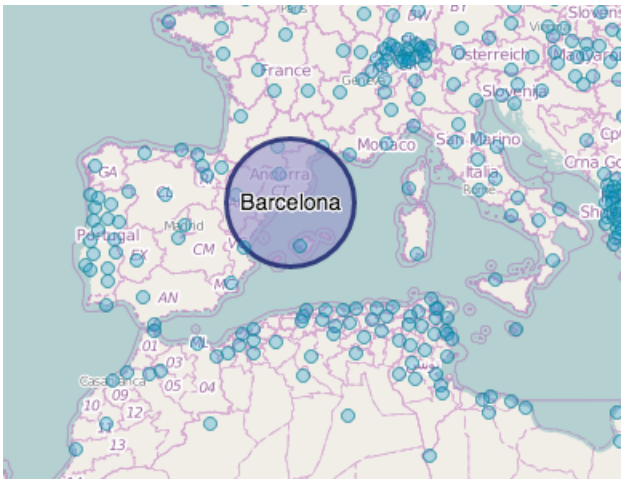
```
    // Vector layer
    var layerVector = new ol.layer.Heatmap({
        source: new ol.source.GeoJSON({
            ...
        })
    });

    // We must wait for the 'change' event to ensure features are loaded
    $('#features').text(layerVector.getSource().getFeatures().length);
```

5.5.5 Styling features under the pointer

5.5.5.1 Goal

Once we know how to deal with events, this example will merge our knowledge with vector layers and features.

This recipe shows how we can change the features under the point position, that is, initially we will show a set of features with a given style and when mouse were over it their style will be changed to some some attribute value.



Selecting features under pointer

5.5.5.2 How to do it...

Let's start defining the styles that will be applied to the features:

```
    var normalStyle = new ol.style.Style({
        image: new ol.style.Circle({
            radius: 4,
            fill: new ol.style.Fill({
                color: 'rgba(20,150,200,0.3)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(20,130,150,0.8)',
                width: 1
            })
        })
    });
    var selectedStyle = new ol.style.Style({
        image: new ol.style.Circle({
```

```
            radius: 40,
            fill: new ol.style.Fill({
                color: 'rgba(150,150,200,0.6)'
            }),
            stroke: new ol.style.Stroke({
                color: 'rgba(20,30,100,0.8)',
                width: 3
            })
        })
    });
    var selectedTextStyleFunction = function(name) {
        return new ol.style.Style({
            text: new ol.style.Text({
                font: '14px helvetica,sans-serif',
                text: name,
                fill: new ol.style.Fill({
                    color: '#000'
                }),
                stroke: new ol.style.Stroke({
                    color: '#fff',
                    width: 2
                })
            })
        });
    };
```

Now, create the source to load features and the vector layer:

```
    var geojsonSource = new ol.source.GeoJSON({
        url: 'data/world_cities.json',
        projection: 'EPSG:3857'
    });

    var vectorLayer = new ol.layer.Vector({
        source: geojsonSource,
        style: normalStyle
    });
```

Create a map instance using some raster layer as background and the previous vector layer:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            vectorLayer
        ],
        // Create a view centered on the specified location and zoom level
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 4
        })
    });
```

Finally, add the code responsible to handle when pointer is over features and changes their style:

```
    var selectedFeatures = [];

    // Unselect previous selected features
    function unselectPreviousFeatures() {
        var i;
        for(i=0; i< selectedFeatures.length; i++) {
            selectedFeatures[i].setStyle(null);
        }
        selectedFeatures = [];
    }

    // Handle pointer
    map.on('pointermove', function(event) {
        unselectPreviousFeatures();
        map.forEachFeatureAtPixel(event.pixel, function(feature) {
            feature.setStyle([
                selectedStyle,
                selectedTextStyleFunction(feature.get('CITY_NAME'))
            ]);
            selectedFeatures.push(feature);
        });
    });
```

5.5.5.3 How it works...

At the beginning we have defines three styles to be applied to our features. The `normalStyle` defines a little blue circle for each feature and is applied to the vector layer, which means all the features are rendered with this style.

The `selectedStyle` defines a bigger circle that we use when want to select the point. The `selectedTextStyleFunction` is function that returns a style showing a text. These two styles are applied on the features under the pointer, to indicate the are selected.

The data we are loading in the GeoJSON file is expressed with `EPSG:4326` projection, because of this, we are specifying the `projection` property within the `ol.source.GeoJSON` instance, so the source automatically transform between projections.

```
1     var geojsonSource = new ol.source.GeoJSON({
2         url: 'data/world_cities.json',
3         projection: 'EPSG:3857'
4     });
```

To know when the pointer is moved over the map we have registered a `pointermove` listener on the map instance. This function is responsible to change the features style to show if they are selected or not. What this function does is:

1. Unselect any previously selected features, that is, change the style to `normalStyle`.
2. For each feature under pointer, change it style and *mark* as selected so we can later unselect.

Note, To store the currently selected features we add them to the `selectedFeatures` array.

```
1     map.on('pointermove', function(event) {
2         unselectPreviousFeatures();
3         map.forEachFeatureAtPixel(event.pixel, function(feature) {
4             feature.setStyle([
5                 selectedStyle,
6                 selectedTextStyleFunction(feature.get('CITY_NAME'))
7             ]);
8             selectedFeatures.push(feature);
9         });
10    });
```

The `event` object contains two important properties: `coordinates`, which contains the pointer coordinates in view's projection, and `pixel` which contains the `[x, y]` pixels that corresponds to the current pointer position.

To know the current features under the pointer position we use the `map.forEachFeatureAtPixel()` method. This methods must receive two parameters: the pixel to query for and a callback function to be executed one time for each selected feature. This callback function receives a reference to the feature.

As we say previously, we apply the `selectedStyle` and `selectedTextStyleFunction` styles to the features under the pointer. Note, while the first is a style, the second is a function that returns a text style with the value of the `CITY_NAME` property.

5.5.5.4 See also

- Using text to style features example at Vector layers chapter, to know how to apply text on styles.
- Working with style functions example at Vector layers chapter, to see how determine the feature's style through using a *style function*.

6. Overlays

Maps are not only used to visualize information but, usually, to interact with them. As we saw in previous chapters, OpenLayers3 allows to render vector data allowing a great degree of flexibility styling the features using different shapes, text and icons.

There are times feature styling is not enough. What happens if we can draw some complex visualization (like a tooltip) in a given map position? To solve this need OpenLayers3 offers the concept of *overlay*.

In this chapter, we are going to describe what are overlays, how we can work with them and what are their most common usages.

6.1 Introducing overlays

An *overlay* allows to place any kind of HTML element in map at some location. This way, we can geolocate an HTML element at a given position and each time the map is panned the element will be panned too. Thanks to overlays we can place tooltips, complex icons using CSS3 animation or SVG.

Given an HTML element, an overlay is created using the `ol.Overlay` class. To allow control all the overlay aspects (like the element or the position within the map) the class provides the next set of properties:

- `element`, a reference to the HTML element that must be rendered within the map.
- `position`, the position of the element, the overlay, in the map.
- `offset`, and `[x, y]` offset applied to the position
- `positioning`, determines how the overlay is placed respect the `position` property. Available values are: `bottom-left`, `bottom-center`, `bottom-right`, `center-left`, `center-center`, `center-right`, `top-left`, `top-center`, and `top-right`.
- `stopEvent`, if true events on the overlay element (for example a click) are propagated to the parent (more on this in the next section).
- `insertFirst`, determines if the overlay must be places as the first element within the container of overlays (more on this in the next section).

See A basic overlay example at Overlays chapter for abetter understand of the `stopEvent` property.

So, given an HTML that shows some label like:

```
<div id="customOverlay">I'm an overlay !!!</div>
```

We can create an overlay and add it to the map with:

```
var customOverlay = new o.Overlay({
    element: document.getElementById('customOverlay'),
    position: [2, 40]
});
```

6.1.1 Adding overlays to the map

Overlays, in the same way as layers or controls, are managed through the map instance. As we know, by the What really happens when a map is created section atThe Map and the View chapter, when we initialize an `ol.Map` instance it creates HTML elements in the document to represent the information. In the same way, when we add overlays to the map a new `<div>` element is created to contain all the overlays attached to the map.

Depending on the value of the `stopEvent` property, the overlays `<div>` container has attached the CSS class `ol-overlaycontainer` (`stopEvent=false`) or `ol-overlaycontainer-stopevent` (`stopEvent=true`).

Note, by default all controls are attached to the `<div class='ol-overlaycontainer-stopevent'>` container.

To manage overlays the `ol.Map` class offers the methods:

- `addOverlay()`, attach a new `ol.Overlay` instance to the map.
- `removeOverlay()`, removes a given overlay from the map.
- `getOverlays()`, returns an `ol.Collection` with all the `ol.Overlay` instances attached to the map. See Controlling the layer stack section at Layers chapter to know more about `ol.Collection`.

Take into account, when an overlay is removed from the map the attached HTML code is removed from the document too.

Following the previous section example code, we can add our overlay with:

```
    // Create a map instance
    var map = new ol.Map({ ... });

    // Add the overlay
    map.addOverlay(customOverlay);
```

Note, we can also set the overlays to be used by the map at initialization time passing an array of overlays:

```
    var map = new ol.Map({
        overlays: [customOverlay]
    });
```

This code will produce an HTML code similar than:

```
<div id="map" class="map">
    <div class="ol-viewport" style="position: relative; overflow: hidden; width:\
 100%; height: 100%;">
        <canvas class="ol-unselectable" width="1140" height="400" style="width: \
100%; height: 100%;"></canvas>
    </div>

    <div class="ol-overlaycontainer-stopevent">
        <div class="ol-zoom ol-unselectable ol-control">...</div>
        <div class="ol-rotate ol-unselectable ol-control ol-hidden">...</div>
        <div class="ol-attribution ol-unselectable ol-control ol-collapsed">...<\
/div>

        <div style="position: absolute; right: 676px; top: 182px;">
            <div id="customOverlay" class="overlay" style="display: block;">
                // Overlay HTML code here...
            </div>
        </div>
    </div>
</div>
```

Because we are using `stopEvent=true` (default value) our overlay is stored in the same container used by the map controls: `<div class="ol-overlaycontainer-stopevent">` in the last position. This means it will be rendered over the rest of controls. To make the overlay rendered under the rest of the controls we need to set `insertFirst=true`. Because of this, usually, when we set `insertFirst=true` we also set `stopEvent=true` too.

6.2 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

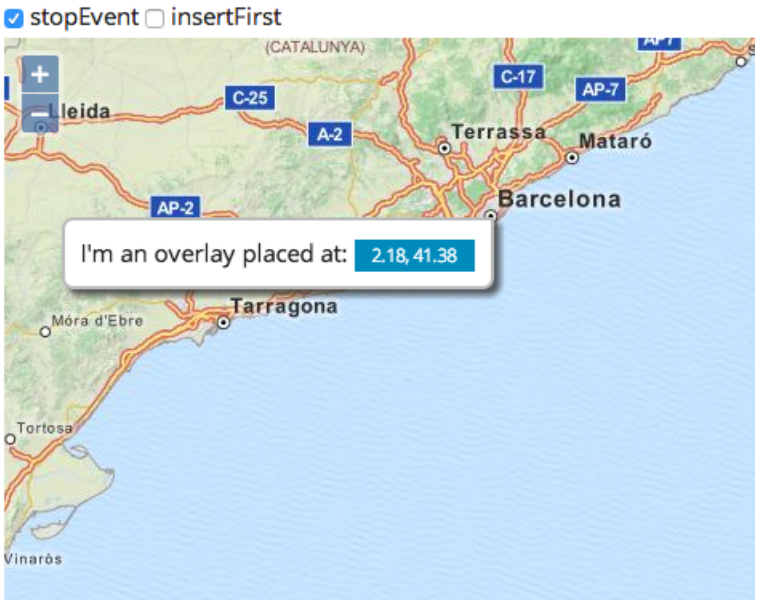
The source code for all the examples can be freely downloaded from thebookofopenlayers3[1] repository.

6.2.1 A basic overlay

6.2.1.1 Goal

In this recipe we are going to create a simple overlay shown when the user clicks on the map. The overlay will show a message with the clicked location.

The goal of the recipe is to show also the behavior of the overlays when working with `stopEvent` and `insertFirst` properties. Because of this we are going to create two checkboxes so the user can play changing their value.



Custom overlay

6.2.1.2 How to do it...

Lets start defining the HTML code of our custom overlay. The idea is to show a message with the current location:

[1] https://github.com/acanimal

```
    <div id="myOverlay" class="overlay">
        I'm a overlay placed at: <span id="coordinate" class="label label-primar\
y">0, 0</span>
    </div>
```

In addition we are going to define a CSS class to give a nice look to our overlay. It will add a background color, a border color and radius and a nice shadow effect:

```
    .overlay {
        background-color: #fff;
        border: 2px #bbb solid;
        border-radius: 7px;
        border-top-right-radius: 0px;
        -webkit-box-shadow: 4px 4px 5px 0px rgba(50, 50, 50, 0.75);
        -moz-box-shadow:    4px 4px 5px 0px rgba(50, 50, 50, 0.75);
        box-shadow:         4px 4px 5px 0px rgba(50, 50, 50, 0.75);
        padding: 10px;
        display: none;
    }
```

Next create the map instance:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 8
        })
    });
```

Add the next function, which is responsible to create an overlay with the given values for the `stopEvent` and `inserFirst` properties:

```
    function createOverlay(stopEvent, insertFirst) {
        return new ol.Overlay({
            element: document.getElementById('myOverlay'),
            positioning: 'top-right',
            stopEvent: stopEvent,
            insertFirst: insertFirst
        });
    }
```

Next, add the function responsible to set the overlay's position, the message for the current location and shows the overlay:

```
    function setCoordinateAndShow(coordinate) {
        overlay.setPosition(coordinate);
        $('#coordinate').text(ol.coordinate.toStringXY(ol.proj.transform(coordin\
ate, 'EPSG:3857', 'EPSG:4326'), 2));
        $(overlay.getElement()).show();
    }
```

Add an initial overlay to be map:

```
    var overlay = createOverlay(true, false);
    map.addOverlay(overlay);
```

Next, add the code required to show the overlay each time the user clicks the map:

```
    map.on('click', function(event) {
        var coordinate = event.coordinate;
        setCoordinateAndShow(coordinate);
    });
```

Finally, add the code to re-create the map each time the *stopEvent* or *insertFirst* checkboxes changes:

```
    $('#stopEvent, #insertFirst').on('click', function(){
        var stopEvent = $('#stopEvent').is(':checked');
        var insertFirst = $('#insertFirst').is(':checked');

        var prevPos = overlay.getPosition();
        map.removeOverlay(overlay);
        overlay = createOverlay(stopEvent, insertFirst);
        map.addOverlay(overlay);

        setCoordinateAndShow(prevPos);
    });
```

6.2.1.3 How it works...

An overlay is a way to locate any kind of HTML element within the map. The issue you can find with this, is when you attach the overlay HTML code to the document it will be shown too.

To avoid show the overlay HTML element before add we can set the `display: none;` style to the HTML element. In this example, it is done setting it in the `.overlay` CSS class.

```
    .overlay {
        ...
        display: none;
    }
```

This way, although we attach the overlay to the map:

```
    var overlay = createOverlay(true, false);
    map.addOverlay(overlay);
```

it is not shown until we force to show it. Each time the map is clicked we need to retrieve the clicked coordinates, assign them to the overlay and show it:

```
    map.on('click', function(event) {
        var coordinate = event.coordinate;
        setCoordinateAndShow(coordinate);
    });

    function setCoordinateAndShow(coordinate) {
        // Set position
        overlay.setPosition(coordinate);
        // Update overlay label
```

```
10          $('#coordinate').text(ol.coordinate.toStringXY(ol.proj.transform(coordin\
11  ate, 'EPSG:3857', 'EPSG:4326'), 2));
12          // Show overlay
13          $(overlay.getElement()).show();
14      }
```

As you can see we have register a listener function on the map's `click` event. The function receives an `event` object, that among others, contains the `coordinate` property with the clicked coordinates in the current map view's projection.

To reuse code, we made the main action within the `setCoordinateAndShow()` function. Note, we want to show the coordinates in the overlay in `EPSG:4326` so we required to transform them because the map uses by default `EPSG:3857`.

We set the new overlay position with the `setPosition()` method and show the overlay setting the `display` CSS property to a value different than `none`. In our case we are using jQuery library and we are applying the `show()` method to the overlay element.

Finally, each time the user changes the value of the checkboxes we need to change the `stopEvent` and `insertFirst` properties too. `ol.Overlay` does not offers any method to modify them, we can only set at instantiation time, so we need to create a new overlay with the new desired property values.

```
1       $('#stopEvent, #insertFirst').on('click', function(){
2           var stopEvent = $('#stopEvent').is(':checked');
3           var insertFirst = $('#insertFirst').is(':checked');
4
5           var prevPos = overlay.getPosition();
6           map.removeOverlay(overlay);
7           overlay = createOverlay(stopEvent, insertFirst);
8           map.addOverlay(overlay);
9
10          setCoordinateAndShow(prevPos);
11      });
```

The checkboxes values are retieved using jQuery `is()` method, to know if they are checked or not. Next we store the current position of the overlay, so we can create a new one in the same position with the new specified properties, in a way the user think it is the same overlay.

Because we create a new overlay we need to remove the old one from the map and attach the new one. This is achieve with the `map.removeOverlay(overlay)` sentence.

6.2.1.4 See also

- Using overlays as markers example at Overlays chapter, to know how to create overlays that uses CSS3 to create animations.

- Showing the mouse location example at Events, listeners and properties chapter, to see another example using map's `click` event in action and how to show the current mouse location.

6.2.2 Using overlays as markers

6.2.2.1 Goal

Because overlays are any kind of HTML element located within the map we can use whatever supported by our browser. In this recipe we are going to use CSS3 for our overlays and create nice animated icons to show earthquakes places.

Overlays using CSS3

6.2.2.2 How to do it...

Let's start creating the map instance:

```
var map = new ol.Map({
    target: 'map',
    renderer: 'canvas',
    layers: [
        new ol.layer.Tile({
            source: new ol.source.MapQuest({
                layer: 'osm'
            })
        })
    ],
    view: new ol.View({
```

```
            center: [0, 0],
            zoom: 2
        })
    });
```

Now, add the code necessary to read the KML file, with the earthquakes information, responsible to add features to the map as overlays:

```
    $.get('./data/2012_Earthquakes_Mag5.kml')
        .done(function(response){
            var format = new ol.format.KML();
            var features = format.readFeatures(response, {
              featureProjection: 'EPSG:3857'
            });
            var coordinates, overlay;
            for (var i = 0; i < features.length; i+=15) {
                coordinates = features[i].getGeometry().getCoordinates();
                overlay = createCircleOutOverlay(coordinates);
                map.addOverlay(overlay);
            };
        });
```

Finally add the `createCircleOutOverlay()` function that creates an overlay in a given location:

```
    function createCircleOutOverlay(position) {
        var elem = document.createElement('div');
        elem.setAttribute('class', 'circleOut');

        return new ol.Overlay({
            element: elem,
            position: position,
            positioning: 'center-center'
        });
    }
```

> Note, I have not copied the CSS code necessary to make the animation. It is not the goal of this recipe, neither the book, to teach about CSS3 animations.

6.2.2.3 How it works...

First we have instantiated the map with a MapQuest raster layer. Next action is to read the data information to create the overlays for each earthquake.

In this recipe, we do not want to create a vector layer that loads features through KML source, what we want is simply to read the file data. Hopefully, this can be easily achieved using a format class, concretely the `ol.format.KML` class.

The first step consist on reading the KML file content. This is up to us and here we have used jQuery `get()` method:

```
$.get('./data/2012_Earthquakes_Mag5.kml')
    .done(function(response){
        // 'response' contains the XML data
    });
```

Know we have the data we need to transform to a set of features using the `ol.format.KML` format class:

```
$.get('./data/2012_Earthquakes_Mag5.kml')
    .done(function(response){
        var format = new ol.format.KML();
        var features = format.readFeatures(response, {
          featureProjection: 'EPSG:3857'
        });
        ...
    });
```

Know the `features` variable contains an array with all the file features. Because the KML contains data in `EPSG:4326` and we want to get features with `EPSG:3857` project we have specified the `featureProjection` property so the format makes that transformation. Otherwise we should do it using `ol.proj.transform()` function.

In the last step we need to create an overlay for each feature located at the feature's coordinates.

Note, the file we are reading contains too much features to be rendered as HTML elements with a good performance. So, to reduce the number of created overlays, we are creating an overlay for each fifteen features.

```
$.get('./data/2012_Earthquakes_Mag5.kml')
    .done(function(response){
        ...
        var coordinates, overlay;
        for (var i = 0; i < features.length; i+=15) {
            coordinates = features[i].getGeometry().getCoordinates();
            overlay = createCircleOutOverlay(coordinates);
            map.addOverlay(overlay);
        };
    });
```

Looping through the feature's array we get each feature coordinate accessing to its geometry with `getGeometry()` and from it getting its coordinates with `getCoordinates()`. Note, the returned coordinates are in `EPSG:3857` projection because we use the `featureProjection` property when reading the KML file.

Final step consists on create the overlay and add it to the map. The responsible to create the overlays is the `createCircleOutOverlay()` function which accepts an array with the coordinates.

For each overlay we want to create a `<div>` element with the `circleOut` associated, that is, we want an element like: `<div class='circleOut'></div>`. In the sample, we need a HTML element for each feature and because we do not know the number of features we need to create them programmatically. We do it using the `document.createElement()` and `elem.setAttribute()` methods:

```
function createCircleOutOverlay(position) {
    var elem = document.createElement('div');
    elem.setAttribute('class', 'circleOut');

    return new ol.Overlay({
        element: elem,
        position: position,
        positioning: 'center-center'
    });
}
```

Given the `elem` reference we can create the `ol.Overlay` instance passing the `element` reference, the `position` and a `positioning` property indicating we want to place the element at the center of the coordinate.

6.2.2.4 See also

- Reading and writing features through the source class example at Data sources and formats chapter, to see how to read and write features using the source and format class.

7. Controls and Interactions

Usually a map is not only necessary to show geospatial data but also we need to interact with it: pan and zoom the map, edit features, show and modify feature's properties, etc. To solve this need, OpenLayers3 counts with two great tools to interact with the components and that we can classify as *controls* and *interactions*.

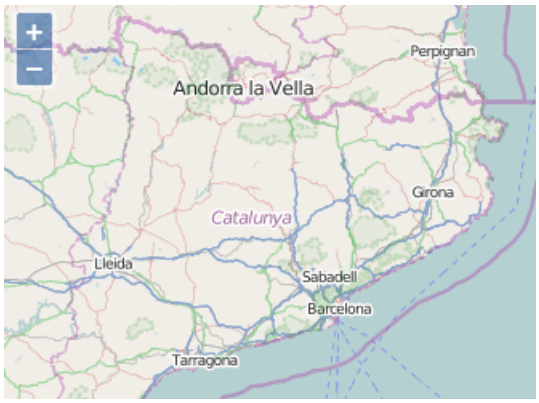
We can see *controls* and *interactions* in action in any simple map example because, by default, the `ol.Map` instances makes use of some controls and interactions to allow the users a minimal degree of control, panning and zooming the map.

This chapter presents two of the most important concepts within OpenLayers3, how we can work with controls and interactions, which kind of controls and interaction the library offers and how we can extend them.

7.1 Controls

Main difference between controls and interactions is a control is a visible widget with a DOM element in a given position of the map. Similarly to an overlay (see Overlays chapter), a control has attached a DOM element through which the user can start actions like zoom in or out.

Next images shows two controls, the zoom control, where we can zoom in or out the map's zoom:



Zoom control

and the attribution control, that shows information about each layer data provides:

Attribution control

7.1.1 The base class `ol.control.Control`

All control implementations inherits from the base class `ol.control.Control`. Basically, this class offers the properties and methods necessary to attach a DOM element with the `ol.control.Control` instance. This link between the elements is made at initialization time through the properties:

- `element`, a reference to the the DOM element that will act as the widget.
- `target`, a reference to the DOM element where the `element` will be attached. By default the control `element` is attached to a special DOM element on top of the map, but we can place the controls at any place of our page. More on this in the section Managing controls.

`ol.control.Control` inherits from `ol.Object` so we can add properties and listen for changes on them.

So, given a HTML piece of code that shows a button like:

```
1    <div id="someControl">I'm a control</div>
```

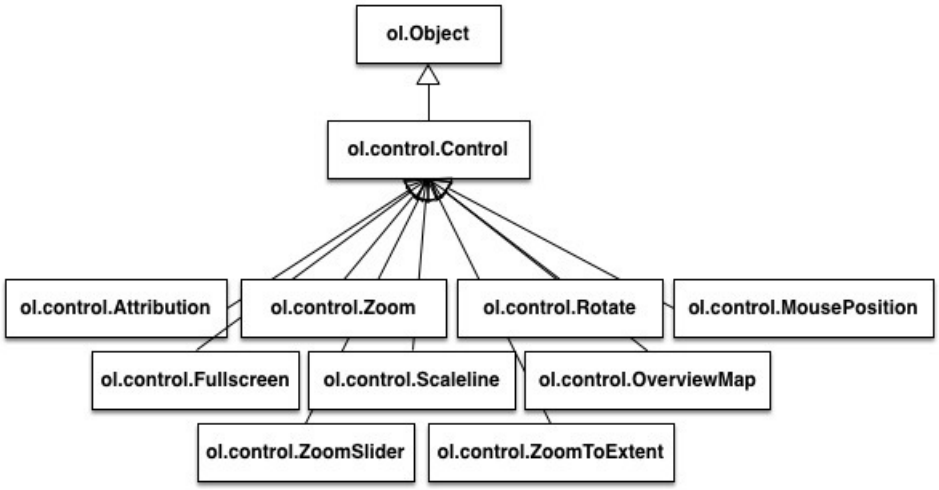
we could create a new control with:

```
1    var control = new ol.control.Control({
2        element: 'someControl'
3    });
```

The way to code a custom control is a bit different from the previous initialization code. See example Creating a custom control.

7.1.2 The controls hierarchy

Next bad-looking figure shows the control class hierarchy and the available set of controls within OpenLayers3. Note how the root class `ol.control.Control` inherits from `ol.Object`, which allows to work with properties and event listeners:



Controls hierarchy

7.1.3 Styling controls

Contrary to the previous code, where we have created the HTML code of our control, the control components implemented within OpenLayers3 creates HTML elements dynamically using JavaScript code. The JavaScript code is also responsible to register listeners, for example, to know when the user clicks on a button so we can make some action, and attaches CSS classes to the HTML element that determines the style of the components.

> The look of a control depends on the combination of the HTML code used to build the control and the CSS classes used on it.

As an example, if we look into the generated HTML code of the `ol.control.Zoom` control we found:

```
<div class="ol-zoom ol-unselectable ol-control">
    <button class="ol-zoom-in ol-has-tooltip" type="button">
        <span role="tooltip">Zoom in</span>+
    </button>
    <button class="ol-zoom-out ol-has-tooltip" type="button">
        <span role="tooltip">Zoom out</span>â^'
    </button>
</div>
```

Although the HTML is always the same, we can redefine the styles overriding the CSS classes `ol-control`, `ol-zoom`, `ol-zoom-in`, `ol-zoom-out` and `tooltip`.

- `ol-control`, is applied to every control and because this determines common properties shared by all controls.
- `ol-zoom`, it is used to place the control within the map.
- `ol-zoom-in` and `ol-zoom-out` are used to set specific properties on buttons.

Note, the CSS rules are not as simple use a CSS class. Within the OpenLayers3 CSS code we found combinations like:

```
.ol-control button {
    ....
}
```

that determines the look of all buttons within a control.

You can find the CSS code use by OpenLayers3 in the source code project file `SOURCE_DIR/css/ol.css`. Note, when we create an OpenLayers3 project we are including a reference to the style with `<link rel="stylesheet" href="http://ol3js.org/en/master/css/ol.css">`. See the section Getting ready for programming with OpenLayer3 on chapter Before to start.

So, to change the style of OpenLayers3 controls you need to know previously the HTML code used to build each control. In addition, the process of styling requires some CSS experience, patience and, in some cases, you will be not able to change as you want, simply because the HTML code has not the structure you need. In those cases, the only solution requires to create your own controls.

7.1.4 Managing controls

Controls are attached to a map instance, because of this, like layers or overlays the `ol.Map` class offers methods to add, remove and retrieve controls from the map. This way, we found the methods:

- `addControl()`, add a new `ol.control.Control` instance to the map. Once added the control is visualized.
- `removeControl()`, removes a given control instance.
- `getControls()`, returns an `ol.Collection` with all the controls attached to the map instance.

As example, next code attached a zoom control to a map instance:

```
var map = new ol.Map({ ... });

var control = new ol.control.Control({
    element: 'someControl'
});

map.addControl(control);
```

Note, the `ol.Map` class also allows to set the controls to be used at initialization time, passing an array of controls:

```
var control = new ol.control.Control({
    element: 'someControl'
});

var map = new ol.Map({
    controls: [control]
    ...
});
```

It is important to note, OpenLayers3 creates a `<div>` element within the HTML code, specially designed to contain all the HTML control's code. This element has the CSS class `ol-overlaycontainer-stopevent`. Next code shows the HTML code generated for three controls:

```
<div class="ol-overlaycontainer-stopevent">
    <div class="ol-zoom ol-unselectable ol-control">
        ...
    </div>
    <div class="ol-rotate ol-unselectable ol-control ol-hidden">
        ...
    </div>
    <div class="ol-attribution ol-unselectable ol-control ol-collapsed">
        ...
    </div>
</div>
```

The `ol-overlaycontainer-stopevent` container element can be also used by overlays depending on its `stopEvent` property. See the Adding overlays to the map section at Overlays chapter.

7.1.4.1 Default controls

When a new `ol.Map` instance is created, if no `controls` property is defined, OpenLayers3 initializes it with a default set of controls, which includes the `ol.control.Zoom` zoom control, the `ol.control.Rotation` rotation control and the `ol.control.Attribution` attribution control.

This set of default controls are created through the `ol.control.defaults()` function, which returns an `ol.Collection` with the default control instances. In addition, the function accepts the boolean `attribution`, `rotate` and `zoom` properties, indicating if we want to enable the corresponding control.

Next code, shows a map initialization using the default controls but disabling the rotation control:

```
var map = new ol.Map({
    controls: ol.control.defaults({
        rotate: false
    }),
    ...
});
```

Because `ol.control.defaults()` returns an `ol.Collection` we can easily extend extend the set of controls to be used with the `ol.Collection.extend()` method. Remember the `extend()` method accepts an array of elements to be added to the collection. See Controlling the layer stack section at Layers chapter.

Next code, shows a map initialization using the default controls plus an `ol.control.MousePosition`:

```
var map = new ol.Map({
    controls: ol.control.defaults().extend([new ol.control.MousePosition()]),
    ...
});
```

On the other way, passing an empty array to the `controls` property implies to create a map without controls.

7.1.5 OpenLayers3 controls

Here we present some of the controls available at OpenLayers3. Because of its continue evolution it makes no sense to make en exhaustive list, because, more probably in a short time there will be new controls available.

7.1.5.1 `ol.control.Attribution`

Implemented by the `ol.control.Attribution` class, this control shows all the attributions related with the current visible layers of the map. Internally, it gets the `attributions` property of each current layer's source and combines them in a control. (See The root source class section at Data sources and formats chapter to see `ol.source.Source` attributes).

Attribution control

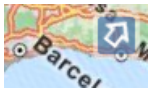
Some of the control look aspects can be controlled with the CSS class `.ol-attribution`. Also, the control accepts next properties at initialization time:

- `collapsible`, boolean value indicating if attributions can be collapsed.
- `collapsed`, boolean value that specifies the initial state of the control (collapsed or expanded). Default is `true` (that is, collapsed).
- `label`, text to show for the collapsed attributions button. Default is `i`.
- `tipLabel`, text to show when mouse is over the control. Default is `Attributions`.
- `collapseLabel`, text to show for the expanded attribution button. Default is `››`.

In addition to the previous properties, the `ol.control.Attribution` class has the methods `getCollapsed()/setColl` and `getCollapsible()/setCollapsible()` through which we can change the values of `collapsed` and `collapsible` properties at any time.

7.1.5.2 `ol.control.Rotate`

The `ol.control.Rotate` class implements the rotate control, which allows to reset the current map rotation to zero degrees. By default, when map rotation is zero, the button is hidden and it is only shown when the map is rotated.



Rotate control

The controls makes used on the CSS class `.ol-rotate` to set its look and, in addition, it accepts a set of properties at initialization time that determines some aspects of its behavior:

- `label`, text to be placed in the button. By default it is a vertical arrow.
- `tipLabel`, text to show when mouse is over the control. Default is `Reset rotation`.
- `duration`, a number with the duration of the reset animation.
- `autoHide`, determines if the button must be shown when rotation is zero. By default is `true` (hidden).

7.1.5.3 `ol.control.MousePosition`

Implemented by the `ol.control.MousePosition` class this control allows to show the mouse position.

Mouse position control

The look is handled through the CSS `.ol-mouse-position` and the control accepts the properties:

- `projection`, by default, the position is shown in the map view's projection. Specifying a projection in this property we can change it.
- `coordinateFormat`, a function that given an `ol.Coordinate` returns an string. It determines how coordinates must be rendered. See the example Playing with controls to see format utilities in action.
- `undefinedHTML`, a text to show for undefined coordinates. It is usually placed when the mouse is outside the map. By default it is an empty string.

7.1.5.4 `ol.control.Scaleline`

The `ol.control.Scaleline` control is a nice widget that shows us the current map scale. It show a bar and indicates the correspondence between pixels and map resolution.

Scale line control

The scale line control look can be modified with the `.ol-scale-line` CSS classes. In addition it has the next properties:

- `minWidth`, The minimum size in pixels to apply to the bar. Default is 64 pixels.
- `units`, by default control shows scale in metric system but we can choose among: `degrees`, `imperial`, `nautical`, `metric` and `us`.

The class has also the methods `getUnits()/setUnits()` that allow us to change the `units` property at any time.

7.1.5.5 `ol.control.Zoom`

The zoom control show a two button widget that allows to increase or decrease the view resolution, that is, change the zoom level. It is implemented by the `ol.control.Zoom` class and it look can be configured with the CSS classes `.ol-zoom-in` and `.ol-zoom-out`.

Zoom control

The `ol.control.Zoom` class allows to set the next properties at initialization time:

- `delta`, determines the increment between each zoom step.
- `duration`, specified the animation duration to applied to the zoom action.
- `zoomInLabel`, text to be shown for the zoom in button. Default is `+`.
- `zoomOutLabel`, text to be shown for the zoom out button. Default is `-`.
- `zoomInTipLabel`, text to be shown for the zoom in button on mouse hover. Default is `Zoom in`.
- `zoomOutTipLabel`, text to be shown for the zoom out button on mouse hover. Default is `Zoom out`.

7.1.5.6 `ol.control.ZoomSlider`

The zoom slider control is an slide bar that allows to change the view's resolution dragging it. It is implemented by the `ol.control.ZoomSlider` class and we can control its look working with the `.ol-zoomslider` and `.ol-zoomslider-thumb` classes.

Zoom slider control

The control allows to change the resolution between two given values we can set with the properties `maxResolution` and `minResolution`.

7.1.5.7 `ol.control.ZoomToExtent`

The *zoom to extent* control is implemented by the `ol.control.ZoomToExtent` class. It shows a button that allows to change the view to zoom a predefined extent.

Zoom to extent control

The look of the control can be changed playing with the `.ol-zoom-extent` CSS class. In addition, the `ol.control.ZoomToExtent` class accepts the next properties at construction time:

- `extent`, a four array number `[minx, miny, maxx, maxy]` representing the extent where to zoom the view.
- `tipLabel`, text to show when mouse is over the control button.

7.1.5.8 `ol.control.OverviewMap`

The overview map control is a useful control that allows to understand better the current location of view displaying a little map with a lower zoom level. For example, in the figure, we are exploring a map at city resolution, the overview map is displayed top-right with a country resolution, so we can quickly know at which country we are looking in.

Overview map control

The control is implemented by the `ol.control.OverviewMap` class and its look can be customized through the CSS classes `.ol-overviewmap` and `.ol-overviewmap-map`.

- `layers`, an `ol.Layer` or array of `ol.Layer` instances to be used by the overview map. If none is specified all the main map layers are used.
- `collapsed`, specified the initial state of the control collapsed (`true`) or expanded (`false`).
- `label`, text to be shown by the collapse button. By default `>>`.
- `collapseLabel`, the text to be shown by the expanded button. By default it is `<<`.
- `collapsible`, determines if the control can be collapsed.
- `tipLabel`, text to be shown when mouse is over button. Default is `Overview map`.

7.1.5.9 `ol.control.FullScreen`

The fullscreen control allows to render the map in fullscreen mode. It is implemented by the `ol.control.FullScreen` class and makes use of the Fullscreen API[1].

Fullscreen control button

The control shows a button that when pressed changes the rendering mode from normal to fullscreen. When rendering mode is fullscreen it shows a button to exit this mode and return to normal rendering.

Note, if your browser does not support the fullscreen API the control will not show any button.

[1] http://www.w3.org/TR/fullscreen/

7.2 Interactions

Interactions are user actions that changes the state of the map. Contrary to controls, interactions has no DOM element associated, so we can not modify the map pressing a button or any other widget, but are applied directly using mouse, keyboard or programmatically. Examples of interactions provided by OpenLayers3 are the map pan action through the mouse or keyboard, the zoom action using the mouse wheel or the map rotation (pressing `shift+alt+mouse`).

Note, although interactions has no widgets associated, some interactions can draw elements in the map, like the *draw* interaction that allows to draw features within a vector layer.

7.2.1 The base class `ol.interaction.Interaction`

All the interaction implementations are based on the root class `ol.interaction.Interaction`. It is an abstract class, which means it is not designed to be instantiated, but to create subclass. As the root class within the interaction class hierarchy it defines attributes and private methods that are used by many of the subclasses.

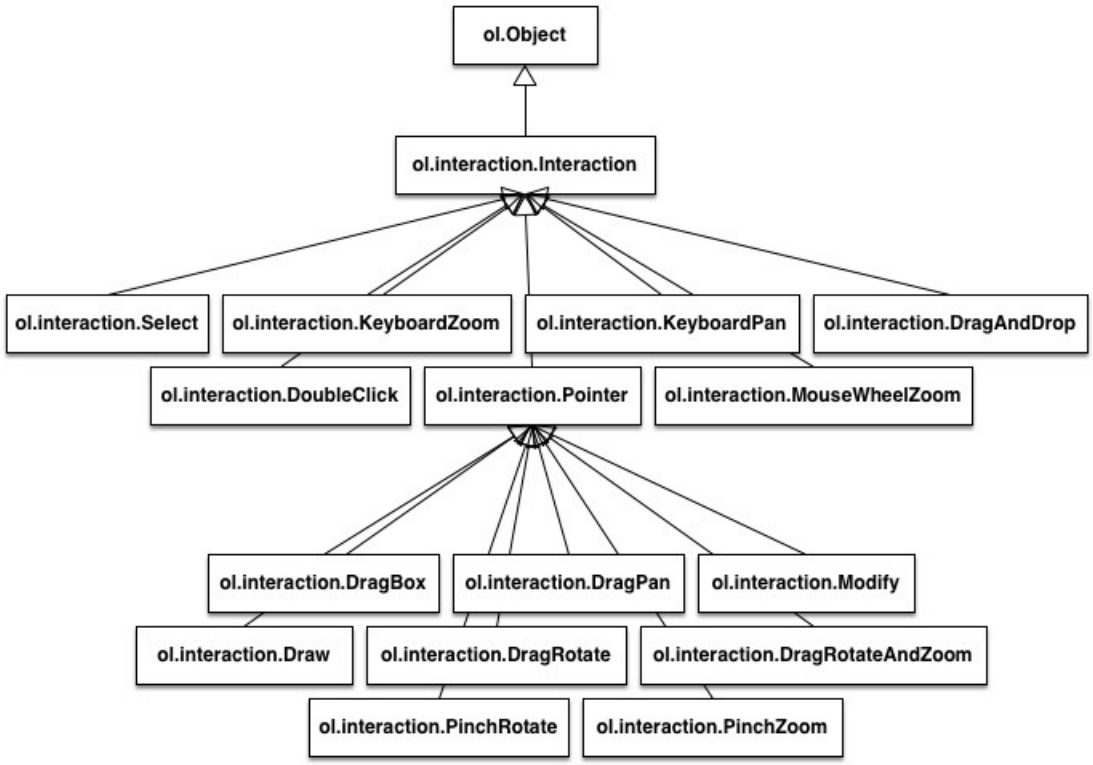
The most notable public method we can find within the `ol.interaction.Interaction` class is the abstract method `handleMapBrowserEvent()`, which is implemented by each subclass on it own way and is invoked each time an event occurs.

Other important methods we can find in the root class are `getActive()` and `setActive()`. At the opposite of controls, interactions can be attached to the map but be enabled or disabled and, this can be changed with the `setActive()` method passing a boolean value. This way, if an interaction is deactivated it has no effect on the map (see section Understanding how interactions works to know more about this).

Finally, it is important to note the `ol.interaction.Interaction` triggers the `change:active` and `propertychange` events to notify a property has changed and, logically, this events are inherited by all the subclasses.

7.2.2 The interactions hierarchy

Next figure shows the set of available interaction we can found at OpenLayers3. Note how the root class `ol.interaction.Interaction` inherits from the `ol.Object` class, which allows to work with properties and event listeners:



Interactions hierarchy

We can see, many interactions inherits from the `ol.interaction.Pointer` class. This is an abstract class that implements many features required for those controls that works through the point, like the actions to draw and modify features, or the actions necessary to drag and rotate.

7.2.3 Managing interactions

Like layers and controls, interactions are managed through the map. This way `ol.Map` class offers the methods `addInteraction()` and `getInteractions()`. The first allows to attach a new interaction to the map, while the second returns an `ol.Collection` with all the interactions currently attached to the map.

In addition, when the map is initialized we can pass an array of interactions to be used through the property `interactions`:

```
var interactionsArray = [
    new ol.interaction.DragRotate(),
    new ol.interaction.DragZoom()
];

// Initialize interactions on map construction
var map = ol.Map({
    interactions: interactionsArray,
    ...
});

// Add a new interactions
var panInteraction = new ol.interaction.DragPan();
map.addInteraction(panInteraction);

// Get an ol.Collection with all the interactions
var collection = map.getInteractions();
```

7.2.3.1 Default interactions

When a map is initialized, if no interaction are specified the `ol.Map` instance is automatically initialized with a set of so called *default* interactions. This is done through the helper function `ol.interaction.defaults()`, which returns an `ol.Collection` with the next set of interactions: `ol.interaction.DragRotate`, `ol.interaction.DoubleClickZoom`, `ol.interaction.DragPan`, `ol.interaction.PinchRotate`, `ol.interaction.PinchZoom`, `ol.interaction.KeyboardPan`, `ol.interaction.Keyb` `ol.interaction.MouseWheelZoom` and `ol.interaction.DragZoom`. (See the section OpenLayers3 interactions to know about all these interaction subclass).

The `ol.inteaction.defaults()` function accepts and `options` parameter, through which we can control what interactions must be created. This parameter accepts the properties:

- `altShiftDragRotate`, a boolean indicating if the `ol.interaction.DragRotate` must be instantiated.
- `doubleClickZoom`, idem for the `ol.interaction.DoubleClickZoom` interactions.
- `keyboard`, idem for the `ol.interaction.KeyboardPan` and `ol.interaction.KeyboardZoom` interactions, that is, it determines if user can interact using the keyboard.
- `mouseWheelZoom`, idem for the `ol.interaction.MouseWheelZoom`.
- `shiftDragZoom`, idem for `ol.interaction.DragZoom`.
- `dragPan`, idem for `ol.interaction.DragPan`.
- `pinchRotate`, idem for `ol.interaction.PinchRotate`.
- `pinchZoom`, idem for `ol.interaction.PinchZoom`.

In addition to this properties, the `options` parameter accepts two more properties, `zoomDelta` and `zoomDuration`, which are passed to the `ol.interaction.KeyboardZoom` interactions and determines its behavior.

7.2.4 Understanding how interactions works

Interactions implements actions done when events occurs in the map, for example, when the mouse wheel is rotated the zoom is change, or when the user double clicks the mouse button zoom is increased.

What happens within OpenLayers3 is all the events produced within the map (no matter if mouse, keyboard or touch event) are captured by the `ol.Map` instance, which acts as a hub of events, and notifies all the attached interactions about the event.

Internally, the `ol.Map` class has a private method `handleMapBrowserEvent()` that, among other steps, iterates over all the interactions and invokes its own `handleMapBrowserEvent()`. Note, an interaction is only invoked if it is activated (remember the `getActive()/setActive()` methods). In addition, within this sequential invocation process, an interaction can break the chain after executing it, returning `false` on its `handleMapBrowserEvent()` method.

In summary, the process of catching events and invoke interactions made by the map is:

- Loop over the interactions array that stores all interactions references.
- Check if the interaction is active, if so then execute its `handleMapBrowserEvent()` method, otherwise ignore and continue with on the next.
- Check if the interaction's `handleMapBrowserEvent()` method returns `false`, if so break the loop, otherwise continue handling the next interactions.

7.2.5 Managing feature changes through `ol.FeatureOverlay` class

Some interactions are designed to work with the features of vector layers, like `ol.interaction.Select`, `ol.interaction.Modify` and `ol.interaction.Draw` (that allow to select, modify or draw new features) and, to indicate some action is taking place, these interactions changes the affected features style.

This way, the *select* interaction changes the style of the selected features, so we can identify them visually. Similarly, the *modify* interaction renders the feature we are modifying with a different style and, finally, the *draw* interaction draws the new features we create with a different look. Next figure shows an example on how features looks when they are selected:

Selected features

The issue is managing this *temporal* style while the interaction is doing some action with the fact a layer can have features using different styles, can become a hard task. What happens if user cancels the *select* action? How does OpenLayers3 know which features must be set with its previous style before selection? Hopefully, to manage these *temporal* styles OpenLayers3 offers the concept of *feature overlay*.

7.2.5.1 The `ol.FeatureOverlay` class

The `ol.FeatureOverlay` class is a special kind of class designed to render a set of features with a given style. It acts as a collection of features and uses a different rendering mechanism than vector layers to draw the contained features overlaid on top of the map. Because of this, the previous commented interactions uses internally a *feature overlay* to render the affected features with a different style.

As example, the `ol.interaction.Draw` interaction adds the new created features to a `ol.FeatureOverlay` instance, so they were drawn with a different style and, once we confirm the edition, the features are moved to a target source vector and removed from the *feature overlay*.

The `ol.FeatureOverlay` class allows to pass a set of options at construction time, which are:

- `features`, the initial set of features to be rendered. This can be get/set with the `getFeatures()/setFeatures()` methods.
- `map`, a map reference where the feature overlay must be rendered on.
- `style`, the style to be used. In the same way as vector layers it can be an `ol.style.Style` instance, an array or a `ol.style.StyleFunction` reference (see Styling features section at Vector layers chapter).

Next code example shows how we can create a point feature and add it to a feature overlay to be rendered on a map:

```
// Some map intance
var map = olMap({ ... });

// Create a point feature
var pointGeometry = new ol.geom.Point([10,30]);
var pointFeature = new ol.Feature(pointGeometry);

// Create a feature overlay that renders the point feature
// in the previous map instance
var featureOverlay = new ol.FeatureOverlay({
    map: map,
    features: [pointFeature],
    style: new ol.style.Style({
        image: new ol.style.Circle({
            fill: new ol.style.Fill({
                color: 'rgba(55, 200, 150, 0.5)'
            }),
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(55, 200, 150, 0.8)'
            }),
            radius: 7
        })
    })
});
```

In addition, the class offers methods to add features new features at any time (`addFeature()`), get the current list of features (`getFeatures()`), remove a given feature (`removeFeature()`), set an array of features at once (`setFeatures()`) and get or set the style to be used by the features (`getStyle()`/ `setStyle()`).

Note, the way to work with `ol.FeatureOverlay` differs from layers, controls and interactions, where they are attached to a map instance. The feature overlay is not attached to a map but needs a reference to the map instance where they must be rendered on.

7.2.6 OpenLayers3 interactions

OpenLayers3 offers an extensive set of interactions that satisfies most of the needs we could find in a real project. They cover from mouse or keyboard interactions to features creation, edition and modification. Next list enumerates and describes shortly each interactions:

7.2.6.1 `ol.interaction.DoubleClickZoom`

Allows the user to zoom in double clicking on the map. We can configure the zoom in animation passing an object with properties:

- `duration`, determines the number of milliseconds the zoom in animation must take, by default `250`.
- `delta`, determines the zoom increment to apply, by default is `1`, which means one zoom level is changed.

7.2.6.2 `ol.interaction.KeyboardPan`

Allows to move the map using the keyboard arrows. It accepts two properties at construction time:

- `pixelDelta`, determines the amount of pixels to displace in the movements. By default `128`.
- `condition`, a function that returns a boolean value indicating under which conditions the action must take place. By default it is when no modifier key is pressed (`ol.events.condition.noModifierKeys`) and selected target is not editable (`ol.events.condition.targetNotEditable`).

> **A word about conditions**
>
> The `ol.events.condition` package implements a set of functions that given a map event determines if some condition is met. As example, next is the implementation of the `ol.events.condition.singleClick` that determines if a single mouse click has produced:
>
> ```
> 1    ol.events.condition.singleClick = function(mapBrowserEvent) {
> 2        return mapBrowserEvent.type == ol.MapBrowserEvent.EventType.SINGLECLICK;
> 3    };
> ```
>
> This functions are used by the interaction implementation to determine under which circumstances the events must by handled.

7.2.6.3 `ol.interaction.KeyboardZoom`

Allows to zoom in and out the map using the keyboard `+/-`. Similarly to the `ol.interaction.DoubleClickZoom` interaction we can control the behavior of the interaction passing the properties:

- `duration`, the number of milliseconds the zoom animation takes. By default `100`.
- `delta`, the zoom step applied to the zoom. By default is `1` which means one zoom level is increased.
- `condition`, a function that returns true indicating if the event must be handled. By default it is managed when the target is not editable (`ol.events.condition.targetNotEditable`).

7.2.6.4 `ol.interaction.MouseWheelZoom`

Allows to change the zoom using the mouse wheel. We can determine the duration of the animation passing the `duration` property, which determines the number of milliseconds. By default `100`.

7.2.6.5 `ol.interaction.PinchRotate`

Applicable for touch screens, this interaction allows to rotate the view by twisting with two fingers. We can pass the `threshold` property, which determines the minimum angle **in radians** to start a rotation. By default it is `0.3` (approximately `17` degrees).

7.2.6.6 `ol.interaction.PinchZoom`

Applicable for touch screens, allows to change the zoom of the map by pinching the screen. We can determine the duration of the animation passing the number of milliseconds in the `duration` property. By default it is `400`.

7.2.6.7 `ol.interaction.DragPan`

It is probably the most used interaction within the set offered by OpenLayers3. This is the interaction that allows to move the map by dragging it.

The interactions accepts a `kinetic` property at initialization time, which allows to configure the movement effect or, in other words, the inertial deceleration of the movement. The values specified at the `kinetic` property must be instances of `ol.Kinetic` class. This class accepts three properties to control the movement:

- `decay`, the decay rate (must be negative so the animation slow down until stop).
- `minVelocity`, the minimum velocity, expressed in `pixels/millisecond`.
- `delay`, the number of milliseconds to calculate the kinetic initial values.

The default `ol.interaction.DragPan` created by the `ol.interaction.defaults()` function used an `ol.Kinectic` instance like: `var kinetic = new ol.Kinetic(-0.005, 0.05, 100);`.

7.2.6.8 `ol.interaction.DragBox`

This interaction allows to draw a vector box by dragging on the map. Alone, it can seem not very important, but combined with other interactions like `ol.interaction.Select` it can be used to, for example, select those features intersection with a given box.

The interaction accepts two properties:

- `style`, determines the style of the box and must be an instance of `ol.style.Style`.

- `condition`, a function that determines when the map events must be handled. By default it is `ol.events.condition.always`.

Note, the `style` property is mandatory for this interaction.

Because the interaction draws a vector box, we can get a reference to the geometry of the box with `getGeometry()` method. In addition, the `ol.interaction.DragBox` class triggers the next events to notify changes:

- `boxstart`, triggered when the box selection starts.
- `boxend`, triggered when the box selection has finished.

7.2.6.9 `ol.interaction.DragZoom`

Allows to zoom to a specified area previously selected by dragging the mouse and selecting a box area. By default, we select the area pressing the `shift` key and dragging the mouse and it is visualizes as a blue rectangular polygon.

The interaction accepts two properties that can change this default behavior:

- `style`, determines the style used to render the selected area. Must be an `ol.Style` instance.
- `condition`, determines under which circumstances the events must be handled by the interactions. By default it occurs when the shift key is pressed (`ol.events.condition.shiftKeyOnly`).

Note, it is a subclass of `ol.interaction.DragBox` so it inherits its `boxstart` and `boxend` event in addition to its `getGeometry()` method.

7.2.6.10 `ol.interaction.DragRotate`

Allows to rotate the map by dragging the mouse while pressing a *conditional* keys. Accepts a `condition` property that determines under which circumstances the event must be processed, which by default are when pressing `shift + Alt` keys (`ol.events.confition.altShiftKeysOnly`).

7.2.6.11 `ol.interaction.DragRotateAndZoom`

This interaction is a mix between `ol.interaction.DragZoom` and `ol.interaction.DragRotate`, that is, it allows to zoom and rotate the view all at once. By default it is activated when pressing the `shift` key and depending on the direction we drag the map the interaction applies a zoom or a rotation to the map.

We can change the condition under the interaction must handle the event by setting the `condition` property at initialization time, which by default is `ol.events.condition.shiftKeyOnly`.

Note, this control is not present within the default interactions created by the `ol.interaction.defaults()` function.

7.2.6.12 `ol.interaction.DragAndDrop`

The *drag & drop* component brings some fresh feature to OpenLayers3 allowing to drag and drop a vector file (or a set of files) into the map and automatically load its content as features. Note, **the interaction does not adds the content to the map**, it simply loads the content and it is our responsibility to add them to a layer.

The interaction accepts two properties to configure it behavior:

- `formatConstructors`, this allows to limit the type of files we can drop on the map. We must specify an array of format classes, for example: `[ol.format.GeoJSON, ol.format.KML]` will limit the type of files we can drop to GeoJSON and KML.
- `projection`, the target project data will be transformed to. By default, read features are transformed to the current map's view's projection.

For each file dropped into the map, once the interaction has loaded the content and transformed to features, it triggers the `addfeatures` event passing the set of features. In concrete, it trigger an `ol.interaction.DragAndDropEvent` instance passing the loaded `features` (an array of `ol.Feature`), the `file` reference and the `projection`.

For better understanding, next code show a sample using the `ol.interaction.DragAndDrop`. First create the interaction, limiting to GeoJSON files, and the map instance:

```
    // The interaction
    var dragAndDrop = new ol.interaction.DragAndDrop({
        formatConstructor: [ol.format.GeoJSON]
    });

    // The map
    var map = olMap({
        ...
        // Add the DragAndDrop interactions
        interactions: ol.interaction.defaults().extend([dragAndDrop]),
        ...
    });
```

Now, listen for `addfeatures` event and create a new vector layer to store them:

```
    // Listen for 'addfeatures' event
    dragAndDrop.on('addfeatures', function(features, file, projection){
        var vectorLayer = new ol.layer.Vector({
            source: vectorSource = new ol.source.Vector({
                features: features
            })
        });
        map.addLayer(vectorLayer);
    });
```

7.2.6.13 `ol.interaction.Select`

As its name suggest, the select interaction allows to select features within the map. Each selected feature is rendering using a different style, specified by the `style` property, and the interaction allows to retrieve the set of selected features through the `getFeatures()` method.

The selected features are rendered using an `ol.FeatureOverlay`. See section Managing feature changes through `ol.FeatureOverlay` class.

The `ol.interaction.Select` is a complex interaction which accepts the next properties to modify its behavior:

- `style`, the selected features will be rendered using this style. It can be an `ol.Style` instance, an array of `ol.Style` instances or an `ol.style.StyleFunction` (see Styling features section at Vector layers chapter).

- `layers`, we can restrict the layer from which the features must be selected indicating them. Here we can pass an array of `ol.layer.Layer` instances allowed to be selected or a *filter* function. The filter function is called for each layer of the map and it is responsible to return `true` if the layer features are allowed to be selected.
- `condition`, a function that determines if the map event must be handled by the interactions. By default it is `ol.events.condition.singleClick`, which means the click events are managed by the select interaction. Clicking on a feature will select it and deselect any previously selected features. Clicking outside any feature will removed the current selection.
- `toggleCondition`, this is the condition to be satisfied to toggle the state of a feature. By default it is processed when `shift` key is pressed (`ol.events.condition.shiftKeyOnly`) and the previous `condition` is satisfied the clicked feature is selected, if it is not currently selected, or deselected if it is.
- `addCondition` and `removeCondition`, Allows to specify different conditions if, for some reason you desire to use different event, instead `toggleCondition`, to add or remove features.

7.2.6.14 `ol.interaction.Draw`

It is common in GIS application the need to draw vector features in a vector layer, that later can be stored sending to the server side so they can be saved on a file or WFS server. Within OpenLayers3 this functionality is implemented by the `ol.interaction.Draw` class.

The `ol.interaction.Draw` allows to draw features using only one type of geometry, specified in the mandatory `type` property, and stores all them in an array of `ol.Feature` or a given vector source, depending if we specify the `features` or `source` properties.

Remember, once the interaction is initialized you can only draw points, lines or polygons, but not all three. To create a different kind of geometry you need to create a different initialized *draw* interactions.

The `ol.interaction.Draw` is a complex and powerful interaction and, because of this, offers a great set of properties to control the interaction behavior:

- `type`, determines the type of geometry the interaction can draw. It must be one of the values determined by the `ol.geom.GeometryType` data type, that is, `Point`, `LineString`, `LinearRing`, `Polygon`, `MultiPoint`, `MultiLineString`, `MultiPolygon`, `GeometryCollection` or `Circle`.
- `features`, an `ol.Collection` of `ol.Feature` where the new created features will be stored.
- `source`, an `ol.source.Vector` instance where features will be stored
- `style`, specifies the style used to render the features. It can be an `ol.style.Style` instance, an array of `ol.style.Style` instances or an `ol.style.StyleFunction` function.
- `condition`, a function that return true when the interaction must manage a given map event. By default it handled when no modifier key is pressed `ol.events.condition.noModifierKeys`.

- `snapTolerance`, the tolerance (in pixels) for snapping the drawing finish. By default it is `12pixels`.
- `minPointsPerRing`, determines the minimum number of points to be drawn before a polygon ring can be finished. Be default it is `3`.
- `geometryName`, the property name used to store the geometry within the features. By default it is `geometry` (see Creating features by hand section at Vector layers chapter.)

Note, you can specify at the same time `features` and `source`. In that case new features will be stored in both places.

Next code creates two different draw interactions, the first allows to draw points and stores them within a collection:

```
1     var pointFeatures = new ol.Collection();
2     var drawPoints = new ol.interaction.Draw({
3         type: 'Point',
4         features: pointFeatures
5     });
```

while the second allows to draw polygons and stores them in an empty vector source:

```
1     var polygonSource = new ol.source.Vector();
2     var drawPolygons = new ol.interaction.Draw({
3         type: 'Polygon',
4         source: polygonSource
5     });
```

In addition to the previous properties, the `ol.interaction.Draw` interaction triggers events to notify some change has produced:

- `drawstart`, fired when draw starts, for example, when the first point of a polygon is created.
- `drawend`, fired when a draw ens, for example, when the last point of a polygon has drawn and it is closed.

7.2.6.15 `ol.interaction.Modify`

The *modify* interactions allows to modify features. It is implemented by `ol.interaction.Modify` class and requires we specify a set of features to work on with the mandatory `features` property. The interaction will allows to modify the feature's geometry, changing its points location.

The `ol.interaction.Modify` has the next set of properties which allows to configure the interaction:

- `features`, an `ol.Collection` with the features allowed to be modified.
- `style`, the style to be used to render features in the modification process.
- `pixelTolerance`, the number of pixels to consider the pointer is close enough to a point for editing. By default it is `10pixels`.
- `condition`, determines when map events are handled by the interaction. By default the interaction works with user clicks mouse button without pressing any modifier key (`ol.events.condition.single` and `ol.events.condition.noModifierKeys`).

7.3 The practice

All the examples follows the code structured described at section Getting ready for programming with OpenLayer3 on chapter Before to start.

The source code for all the examples can be freely downloaded from thebookofopenlayers3[2] repository.

7.3.1 A static map

7.3.1.1 Goal

Sometimes we desire to make single visualization of data within a map but, here is the difference, without any interaction, that is, no interaction, no pan, no zoom, etc. We want to present some *static map* with the results.

On this situations it is easy to create an *static* map, that is, a map without controls neither interactions that allows the user to change the map's view.

7.3.1.2 How to do it...

Create a `<div>` element to hold the map (in the same way we saw in the A basic map example at The Map and the View chapter).

Add the next JavaScript code that creates an `ol.Map` instance:

```
/**
 * Create a map using a 2D view with a tile layer retrieving tiles from
 * OpenStreetMap project
 */
var map = new ol.Map({
    // The DOM element that will contains the map
    target: 'map',
    renderer: 'canvas',
    // The initial set of layers
    layers: [
        // Add a new Tile layer getting tiles from OpenStreetMap source
        new ol.layer.Tile({
            source: new ol.source.OSM()
        })
    ],
    // The view to be used to show the map is a 2D
```

[2] https://github.com/acanimal

```
        view: new ol.View({
            center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 6
        }),
        // Initialize with empty arrays for controls and interactions
        controls: [],
        interactions: []
    });
```

7.3.1.3 How it works...

As we saw in the theory section, the `ol.Map` instance can be initialized passing an array of `controls` and `interactions` with the desired tools to make use. So, all the magic resides on passing an empty array to this properties:

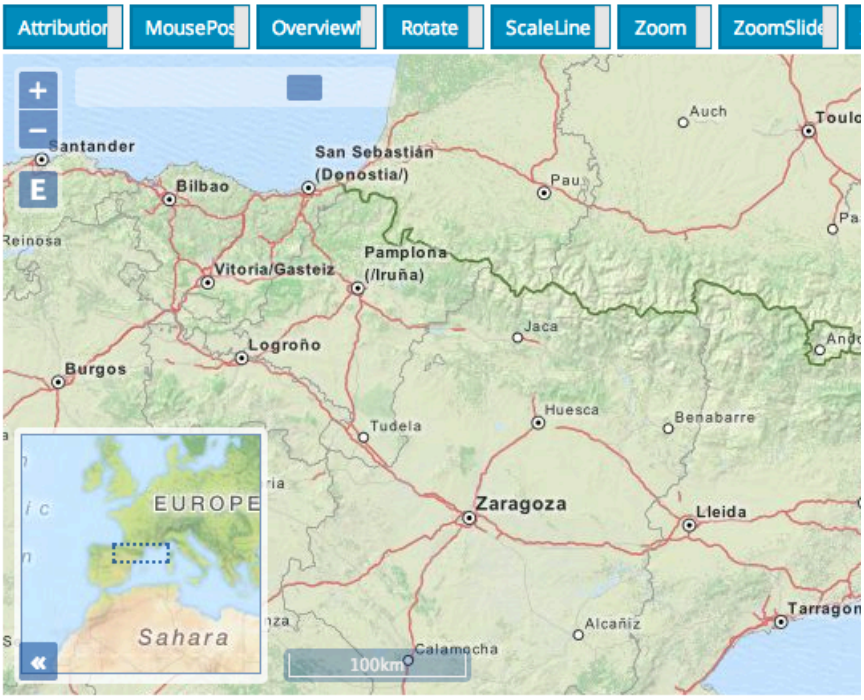
```
    ...
    // Initialize with empty arrays for controls and intereactions
    controls: [],
    interactions: []
    ...
```

This way the user have the feeling to see a static map or an image.

7.3.2 Playing with controls

7.3.2.1 Goal

In this recipe we are going to show how to use most common controls available at OpenLayers3. We are going to create a map with all the control and a set of checkboxes that will allow to activate or deactivate them.



Playing with controls

7.3.2.2 How to do it...

Let's start adding the HTML code necessary for the checkboxes and the map:

```
    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="Attribution" data-off="Attribution">
    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="MousePosition" data-off="MousePosition">
    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="OverviewMap" data-off="OverviewMap">
    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="Rotate" data-off="Rotate">

    <!-- more controls -->

    <div id="map" class="map"></div>
```

If we add all the controls to the map we will see some of them overlaps each other. To solve this, we have redefined some of the CSS control's styles, moving them slightly. Create a `<style>` element and place the next code:

```
    .ol-mouse-position {
        right: 3em;
        color: #fff;
        background-color: rgba(0,60,136,.5);
        padding: 2px 10px;
        border-radius: 5px;
    }
    .ol-scale-line {
        left: 175px;
    }
    .ol-rotate {
        top: 3em;
    }
    .ol-zoomslider {
        width: 200px;
        height: 25px;
        left: 3em;
        top: 0.5em;
    }
    .ol-zoomslider-thumb {
        height: 16px;
    }
```

Now, we are going to create an array with all the controls to be used by the map. This will help us later to add and remove controls:

```
    var controls = [
        new ol.control.Attribution(),
        new ol.control.MousePosition({
            undefinedHTML: 'outside',
            projection: 'EPSG:4326',
            coordinateFormat: function(coordinate) {
                return ol.coordinate.format(coordinate, '{x}, {y}', 4);
            }
        }),
        new ol.control.OverviewMap({
            collapsed: false
        }),
        new ol.control.Rotate({
            autoHide: false
        }),
        new ol.control.ScaleLine(),
```

```
        new ol.control.Zoom(),
        new ol.control.ZoomSlider(),
        new ol.control.ZoomToExtent(),
        new ol.control.FullScreen()
    ];
```

Next, create a map instance passing the previous array to the `controls` property:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        controls: controls,
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 42.3833], 'EPSG:4326', 'EPSG:3857\
'),
            zoom: 7
        })
    });
```

Finally, add the code responsible to listen on the `change` event in the checkboxes, that will add to or remove controls from the map:

```
    $('input').on('change', function(event) {
        var index = $('input').index(event.target);
        var checked = $(event.target).is(':checked');

        if(checked) {
            map.addControl(controls[index]);
        } else {
            map.removeControl(controls[index]);
        }
    });
```

7.3.2.3 How it works...

We are making use of the `controls` property to specify the set of controls to be used by the `ol.Map` instance.

Previously we have defined an array of controls to be used. We have customized some of them passing specific properties. In the case of `ol.control.OverviewMap` it is show collapsed by default, so we have set the property `collapsed` to `false` to show it expanded. Similarly, the `ol.control.Rotate` control is initially hidden, when rotation is zero, until we rotate the map (using `shift+alt+mouse`). To show the button we have set `autoHide` to `false`.

```
    ...
    new ol.control.OverviewMap({
        collapsed: false
    }),
    new ol.control.Rotate({
        autoHide: false
    }),
    ...
```

On the `ol.control.MousePosition` control we have applied some changes too. First, when there are no values to be shown (mainly because the mouse is outside the map) we want to show a message to the user, because of this we set the property `undefinedHTML`. In addition we want to show coordinates in `EPSG:4326` and not in `EPSG:3857`, which is the default view projection. Finally, we want to give a custom format for the coordinates shown by the control. For this purpose, we are using the `coordinateFormat` property:

```
    ...
    new ol.control.MousePosition({
        undefinedHTML: 'outside',
        projection: 'EPSG:4326',
        coordinateFormat: function(coordinate) {
            return ol.coordinate.format(coordinate, '{x}, {y}', 4);
        }
    }),
    ...
```

Note, the `coordinateFormat` property accepts a function that will receive a parameter with the coordinate values. Given this value and, using `ol.coordinate.format()` function, we return an string with format `{x}, {y}` and using four digits.

Returning to the program, we have registered (using jQuery) a listener function to get notified each time user changes the value of an `<input>` element (that is, our checkboxes):

```
    $('input').on('change', function(event) {
        ...
    });
```

Each time user changes a checkbox we need to know which checkbox has changed, that is, its index within all the checkboxes, and whether if it is checked or not. To show or hide the control we simply need to add or remove it from the map:

```
    var index = $('input').index(event.target);
    var checked = $(event.target).is(':checked');

    if(checked) {
        map.addControl(controls[index]);
    } else {
        map.removeControl(controls[index]);
    }
```

Note removing a control from the map does not deletes the control instance. In addition we need to store the control instance if we want to add it later.

7.3.2.4 See also

- See the Managing interactions sample to know how to add and remove interactions.

7.3.3 Creating a custom control

7.3.3.1 Goal

Because sometimes we can desire a kind of control not implemented by OpenLayers3, it is important to understand how we can extend the `ol.control.Control` class to create our custom controls.

In this sample we are going to create our custom zoom in/out control, which will serve as the base to explain the related concepts.

Custom control

Custom controls can be created following the steps of this recipe or, much better, in the same way the existent controls, that is, extending the OpenLayers3 source code. For this later option, your require to download the project source code, understand how to work with Google Closure library and build your own OpenLayers3 distribution. Unfortunately, this is out of the scope of this book.

7.3.3.2 How to do it...

Our control consists in a two buttons widget, so lets start creating the HTML for our control and map:

```
    <div id="customControl">
        <button type="button" class="btn btn-primary btn-xs"><span class="glyphi\
con glyphicon-plus"></span></button>
        <button type="button" class="btn btn-primary btn-xs"><span class="glyphi\
con glyphicon-minus"></span></button>
    </div>

    <div id="map" class="map"></div>
```

Now, add the JavaScript code responsible to manage our new control:

```
    var CustomControl = function(opt_options) {
        var options = opt_options || {};
        var element = document.getElementById('customControl');
        var this_ = this;

        var zoomInBtn = $(element).find('button')[0];
        $(zoomInBtn).on('click', function() {
            var view = this_.getMap().getView();
            var newResolution = view.constrainResolution(view.getResolution(), 1\
);
            view.setResolution(newResolution);
        });

        var zoomOutBtn = $(element).find('button')[1];
        $(zoomOutBtn).on('click', function() {
            var view = this_.getMap().getView();
            var newResolution = view.constrainResolution(view.getResolution(), -\
1);
            view.setResolution(newResolution);
        });

        ol.control.Control.call(this, {
            element: element,
            target: options.target
        })
    };
    ol.inherits(CustomControl, ol.control.Control);
```

Finally, create an `ol.Map` instance passing a new instance of our new control to the `controls` property:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        controls: [new CustomControl()],
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
```

```
12          view: new ol.View({
13              center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857\
14  '),
15              zoom: 8
16          })
17      });
```

7.3.3.3 How it works...

First, we have created the HTML of our code. It is a two buttons widget, so we have created a `<div>` element to contain the two buttons. Note we have identified it with the `id=customControl` attribute so we can later reference it:

```
1       <div id="customControl">
2           <button type="button" class="btn btn-primary btn-xs"><span class="glyphi\
3   con glyphicon-plus"></span></button>
4           <button type="button" class="btn btn-primary btn-xs"><span class="glyphi\
5   con glyphicon-minus"></span></button>
6       </div>
```

Next, we need to create a new control. Because OpenLayers3 is based in Google Closure, it is a best practice create the control following the same rules. So, to create a new control we need to create a new object and make it inherits from the `ol.control.Control` class:

```
1       var CustomControl = function(opt_options) {
2           ...
3       };
4       ol.inherits(CustomControl, ol.control.Control);
```

All controls accepts options at construction time, because of this the `CustomControl` has the `opt_options` parameter.

Inside the function, we found three different sections: first we create an `options` variable, equal to the `opt_options` if any passed or an empty object if no options are specified. Second, we put the code responsible to manage the control. Third, we call the superclass constructor passing all the required parent options.

```
    var options = opt_options || {};

    ...
    // Code to handle the control
    ...

    ol.control.Control.call(this, {
        element: element,
        target: options.target
    });
```

As we can see the last step implies call the `ol.control.Control` superclass passing references to the `element` and the options `target` properties.

Now, lets going take a look the control code for managing zoom in and out actions. First we get a reference to the main element of our control, the `<div>` identified as `customControl`:

```
    var element = document.getElementById('customControl');
```

Next, we store a reference to the control itself in the `this_` variable.

```
    var this_ = this;
```

Next, get a reference to the zoom in button and attach a listener function for the `click` event. Note, both for button selection and listener attachment we are using jQuery:

```
    var zoomInBtn = $(element).find('button')[0];
    $(zoomInBtn).on('click', function() {
        ...
    });
```

Each time user clicks on the zoom in button we need to get a map's view reference and change the zoom level.

As we know, by Resolutions and zoom levels section at The Map and the View chapter, the view works internally with resolutions not zoom levels, because of this we need to increase the resolution of the view, that is, if at a given level each pixels corresponds to, for example, `39135.76` meters, in the next level each pixel must represent `19567.88` meters.

In the code, this is done using the helper method `view.constrainResolution()`. This method, given a resolution and an increase value (a delta that can be negative or positive) computes the right resolution value:

```
    var view = this_.getMap().getView();
    var newResolution = view.constrainResolution(view.getResolution(), 1);
    view.setResolution(newResolution);
```

Similarly, for the zoom out button we apply the same steps. Note this time we pass a negative delta value to the `view.constrainResolution()` method.

```
    var zoomOutBtn = $(element).find('button')[1];
    $(zoomOutBtn).on('click', function() {
        var view = this_.getMap().getView();
        var newResolution = view.constrainResolution(view.getResolution(), -1);
        view.setResolution(newResolution);
    });
```

As you can observe in the running sample, this implementation of the zoom control is really simply and changes the current view's resolution without applying a nice animation like the `ol.control.Zoom` does.

7.3.3.4 There is more...

At the opposite of this example, OpenLayers3 controls implementation does not uses HTML code to create the widget associated to the control. Why?

Well, if you run the sample, you will see an ugly effect due how we implement the control. While the page is loading we can see for an instants the control on the top-left and, later, once the HTML code is associated with the CustomControl, it is paced on top of the map.

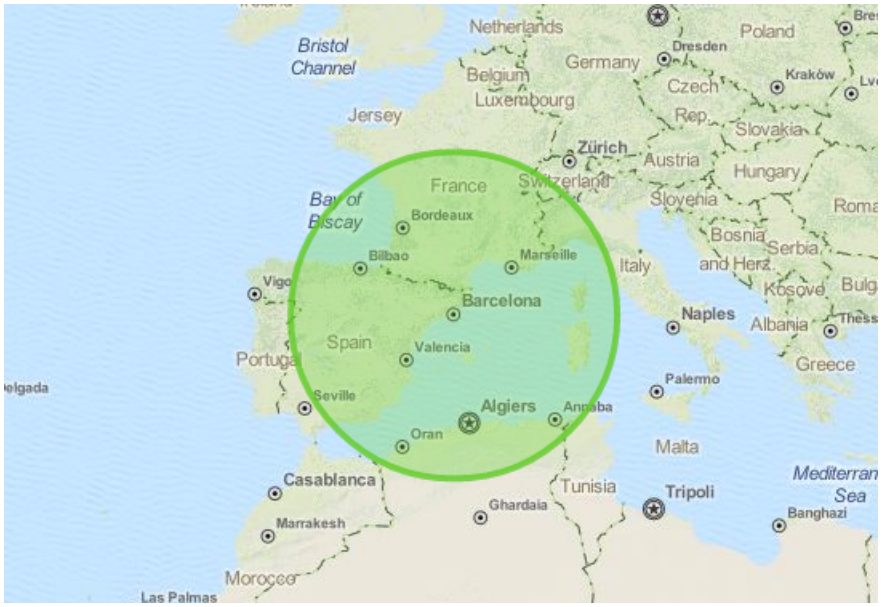
In addition, because our HTML code uses an `id` attribute to identify our control, that means we can not place more than one control at once in our map. Usually this is true, who wants two zoom controls at the same time?, but what happens if our control is a little window that allows to show the attributes of some selected feature and we require more than one?

In practice, the way to implement controls, or at least how OpenLayers3 implements them, is creating the HTML elements that conforms the widget using JavaScript.

7.3.4 Working with feature overlay

7.3.4.1 Goal

Feature overlays are used by some of the most important interactions. They allows to render selected features or features temporary created with a different style. So, this sample shows how we can work directly with the `ol.FeatureOverlay` class.



Feature overlay

The sample shows how we can create a circle feature but, instead of add it to a vector layers class, we will add to an `ol.FeatureOverlay` instance.

7.3.4.2 How to do it...

As always start creating a HTML element to hold the map instance:

```
    <div id="map" class="map"></div>
```

Now, create an `ol.Map` instance adding a raster layer using MapQuest provider:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 42.3833], 'EPSG:4326', 'EPSG:3857\
'),
```

```
            zoom: 4
        })
    });
```

Next, create a circle feature:

```
    var circle = new ol.geom.Circle(
        ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        1000000
    );
    var circleFeature = new ol.Feature(circle);
```

Finally, create an `ol.FeatureOverlay` to be rendered on the map and containing the previous feature:

```
    var featureOverlay = new ol.FeatureOverlay({
        map: map,
        features: [circleFeature],
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(100, 210, 50, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 4,
                color: 'rgba(100, 200, 50, 0.8)'
            })
        })
    });
```

7.3.4.3 How it works...

It is relatively simple to use an `ol.FeatureOverlay` we simply have in mind a feature overlay is not attached to a map but we pass an `ol.Map` reference to it.

The process used to render feature overlays is different than the process used to render a vector feature. It is rendered on top of the map in the final step of frame rendering.

In our code, we pass a map reference through the `map` property, an array of features to be renderer using the `features` property and, finally, the style to be used with the `style` property:

```
    var featureOverlay = new ol.FeatureOverlay({
        map: map,
        features: [circleFeature],
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(100, 210, 50, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 4,
                color: 'rgba(100, 200, 50, 0.8)'
            })
        })
    });
```

Note, because a circle feature only requires a fill and stroke to be rendered we only have specified the `fill` and `stroke` style properties.

One final note about projections. Because the map's view has no projection specified and also uses MapQuest provider, the default projection is `EPGS:3857`. Both specifying the map's view's center and setting the coordinates for the circle feature we have made use of `ol.proj.transform()` function, that allows us to write coordinates in a more *natural* way using `EPSG:4326` and transforming to required `EPSG:3857`:

```
    ...
    view: new ol.View({
        center: ol.proj.transform([2.1833, 42.3833], 'EPSG:4326', 'EPSG:3857'),
        zoom: 4
    })
    ...
    var circle = new ol.geom.Circle(
        ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        1000000
    );
    ...
```

7.3.5 Managing interactions

7.3.5.1 Goal

This sample shows the basis about how using interactions and how to attach to or detach them from the map. We will create some checkboxes that will allow to enable or disable interactions.

Managing interactions

7.3.5.2 How to do it...

Lest create the required HTML code to represent the set of checkboxes and the map element. Note we have make use of a plugin to beatify the chekboxes. Do not worry about this, the essence of the sample is not to create a nice application but to explain of to work with interactions.

```
    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="DoubleClickZoom" data-off="DoubleClickZoom">

    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="KeyboardPan" data-off="KeyboardPan">

    <input type="checkbox" checked data-size="mini" data-toggle="toggle" data-on\
="KeyboardZoom" data-off="KeyboardZoom">

    ...

    <div id="map" class="map"></div>
```

Create an array with all the interactions, note we create in the same order we have created the corresponding checkboxes:

```
    var duration = 400;

    // Create interactions array
    var interactions = [
        new ol.interaction.DoubleClickZoom({
            duration: duration
        }),
        new ol.interaction.KeyboardPan({
            pixelDelta: 256
        }),
        new ol.interaction.KeyboardZoom({
            duration: duration
        }),
        new ol.interaction.MouseWheelZoom({
            duration: duration
        }),
        new ol.interaction.PinchRotate(),
        new ol.interaction.PinchZoom({
            duration: duration
        }),
        new ol.interaction.DragPan({
            kinetic: new ol.Kinetic(-0.01, 0.1, 200)
        }),
        new ol.interaction.DragZoom(),
        new ol.interaction.DragRotate(),
    ];
```

Create an `ol.Map` instance setting an empty array for the `interactions` property:

```
    var map = new ol.Map({
        target: 'map',
        renderer: 'canvas',
        interactions: [interactions],
        layers: [
            new ol.layer.Tile({
                source: new ol.source.MapQuest({
                    layer: 'osm'
                })
            })
        ],
        view: new ol.View({
            center: ol.proj.transform([2.1833, 42.3833], 'EPSG:4326', 'EPSG:3857\
```

```
'),
            zoom: 7
        })
    });
```

Add the code to listening for changes on the checkboxes. This will be responsible to check if we need to add to or remove the interaction from the map:

```
    $(document).ready(function(){
        $('input').on('change', function(event) {
            var index = $('input').index(event.target);
            var checked = $(event.target).is(':checked');

            if(checked) {
                map.addInteraction(interactions[index]);
            } else {
                map.removeInteraction(interactions[index]);
            }
        });
    });
```

7.3.5.3 How it works...

It is important to note we have created the interaction within the array in the same order we have added the HTML checkboxes. This is necessary because the listener function adds or removes interactions depending on the clicked checkbox index.

In our sample, we have declared a `duration` variable and applied to all the interactions that modifies the zoom to work with the same duration animation. In addition, we have applied different values than the default ones to the `ol.interaction.KeyboardPan` (to displace a distance of a tile) and the `ol.interaction.DragPan` to use a different kinetic animation movement:

```
    var interactions = [
        new ol.interaction.DoubleClickZoom({
            duration: duration
        }),
        new ol.interaction.KeyboardPan({
            pixelDelta: 256
        }),
        ...
        new ol.interaction.PinchRotate(),
        new ol.interaction.PinchZoom({
```

```
            duration: duration
        }),
        new ol.interaction.DragPan({
            kinetic: new ol.Kinetic(-0.01, 0.1, 200)
        }),
        ...
    ];
```

The function responsible to add or remove interactions from the map is the listener function that reacts when any checkbox changes. This is done using jQuery selector (that selects all the `<input>` elements) and applying a listener for the `change` event:

```
    $(document).ready(function(){
        $('input').on('change', function(event) {
            ...
        });
    });
```

Note we set the listener function once the document is ready. This is done because when the plugin that beautifies checkboxes is applied it generates a not desired `change` event, which makes the anonymous function adds the interactions twice.

Within the anonymous function the code obtains the `index` of the selected `<input>` element (the checkbox) and a boolean `checked` value that indicates if the checkbox is selected or not.

```
    var index = $('input').index(event.target);
    var checked = $(event.target).is(':checked');
```

With that values we can determines if the interaction must be active or not. This is done adding or removing the interactions with the `map.addInteraction()` and `map.removeInteraction()` methods.

```
    if(checked) {
        map.addInteraction(interactions[index]);
    } else {
        map.removeInteraction(interactions[index]);
    }
```

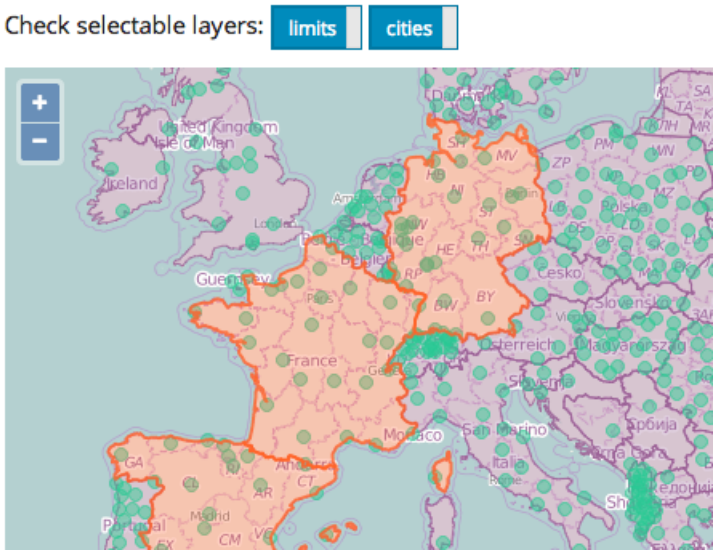
Note, we can also use the interaction's method `setActive()` that, instead of add to or remove the interaction from the map, it is simply flagged as active or inactive. Remember deactivated interaction are ignored within the events handler loop (see Understanding how interactions works section).

7.3.6 Selecting features

7.3.6.1 Goal

Selecting features is a very common feature in most GIS applications. Once selected we can remove the features from the layer, change its style or do whatever our application requires.

In this example we are going to work with the `ol.interaction.Select` to demonstrate how easy we can select features from different vector layers.



Selecting features

We are going to create a map with two vector layers that will be allowed to be selected depending on two checkboxes. In addition, the selected features will be rendered with an style different than the default one.

7.3.6.2 How to do it...

Start adding the necessary HTML code for the checkboxes and map element. Note, the checkboxes are `<input>` elements where we have applied a plugin to beautify them:

```
    <p>
        Check selectable layers:
        <input id="limits" type="checkbox" checked data-size="mini" data-toggle=\
"toggle" data-on="limits" data-off="limits">

        <input id="cities" type="checkbox" checked data-size="mini" data-toggle=\
"toggle" data-on="cities" data-off="cities">
    </p>

    <div id="map" class="map"></div>
```

Next create two vector layers. The first will show world administrative limits from a TopoJSON file:

```
    var limitsLayer = new ol.layer.Vector({
        source: new ol.source.StaticVector({
            url: 'data/world_limits.json',
            format: new ol.format.TopoJSON(),
            projection: 'EPSG:3857'
        }),
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(155, 100, 150, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(155, 100, 150, 0.8)'
            })
        })
    });
```

the second layer will show the most important cities of the World from a GeoJSON file:

```
    var citiesLayer = new ol.layer.Image({
        source: new ol.source.ImageVector({
            source: new ol.source.GeoJSON({
                url: 'data/world_cities.json',
                projection: 'EPSG:3857'
            }),
            style: new ol.style.Style({
                image: new ol.style.Circle({
                    fill: new ol.style.Fill({
                        color: 'rgba(55, 200, 150, 0.5)'
```

```
                    }),
                    stroke: new ol.style.Stroke({
                        width: 1,
                        color: 'rgba(55, 200, 150, 0.8)'
                    }),
                    radius: 4
                })
            })
        })
    });
```

Next initialize the select interaction. It uses a customized style for selected features and a custom function to determine which layers must be selectable:

```
    var selectInteraction = new ol.interaction.Select({
        layers: function(layer) {
            var limits = $('#limits').is(':checked');
            var cities = $('#cities').is(':checked');
            return (limits && layer === limitsLayer) ||
                (cities && layer === citiesLayer);
        },
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(255, 100, 50, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 2,
                color: 'rgba(255, 100, 50, 0.8)'
            }),
            image: new ol.style.Circle({
                fill: new ol.style.Fill({
                    color: 'rgba(255, 100, 50, 0.5)'
                }),
                stroke: new ol.style.Stroke({
                    width: 2,
                    color: 'rgba(255, 100, 50, 0.8)'
                }),
                radius: 7
            })
        }),
        toggleCondition: ol.events.condition.never,
        addCondition: ol.events.condition.altKeyOnly,
```

```
        removeCondition: ol.events.condition.shiftKeyOnly
    });
```

Finally, initialize the map instance using the previous layers and the select interaction:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        interactions: ol.interaction.defaults().extend([
            selectInteraction
        ]),
        layers: [
            // Add a new Tile layer getting tiles from OpenStreetMap source
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            limitsLayer,
            citiesLayer
        ],
        view: new ol.View({
            center: ol.proj.transform([2, 40], 'EPSG:4326', 'EPSG:3857'),
            zoom: 4
        })
    });
```

7.3.6.3 How it works...

We have started creating two vector layers. The first shows the World's limits using a custom style. Because layer is composed of polygons the style has only defined the `fill` and `stroke` properties.

Note also, we have used an `ol.source.StaticVector` instance indicating in the `format` property the source data is in TopoJson format. Another options would be to use directly the `ol.source.TopoJSON` source:

```
    var limitsLayer = new ol.layer.Vector({
        source: new ol.source.StaticVector({
            url: 'data/world_limits.json',
            format: new ol.format.TopoJSON(),
            projection: 'EPSG:3857'
        }),
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(155, 100, 150, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 1,
                color: 'rgba(155, 100, 150, 0.8)'
            })
        })
    });
```

The second layer shows cities around the world as points. Because the number of features we have wrap the GeoJSON source within the `ol.source.ImageVector`, which allows to render the features as raster. Because of this we have used an `ol.layer.Image` and not a vector layer.

Note also, we have applied the style to the `ol.source.ImageVector` instead to the layer. Because the cities are rendered as points and points, for performance reasons, are rendered using images (concretely circles) we have only defined the `image` property (see Styling features section at Vector layers chapter).

```
    var citiesLayer = new ol.layer.Image({
        source: new ol.source.ImageVector({
            source: new ol.source.GeoJSON({
                url: 'data/world_cities.json',
                projection: 'EPSG:3857'
            }),
            style: new ol.style.Style({
                image: new ol.style.Circle({
                    fill: new ol.style.Fill({
                        color: 'rgba(55, 200, 150, 0.5)'
                    }),
                    stroke: new ol.style.Stroke({
                        width: 1,
                        color: 'rgba(55, 200, 150, 0.8)'
                    }),
                    radius: 4
                })
```

```
            })
        })
    });
```

Once the layers have been defined we need to initialize the select interactions. We desire to visualize the selected features with a different style and, because they can be points or polygons, we have set the `fill`, `stroke` and `image` attributes.

```
    var selectInteraction = new ol.interaction.Select({
        ...
        style: new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(255, 100, 50, 0.3)'
            }),
            stroke: new ol.style.Stroke({
                width: 2,
                color: 'rgba(255, 100, 50, 0.8)'
            }),
            image: new ol.style.Circle({
                fill: new ol.style.Fill({
                    color: 'rgba(255, 100, 50, 0.5)'
                }),
                stroke: new ol.style.Stroke({
                    width: 2,
                    color: 'rgba(255, 100, 50, 0.8)'
                }),
                radius: 7
            })
        }),
        ...
    });
```

We have changes slightly the behavior of our controls. By default pressing the `shift` key features are added or removed (toggled) from the list of selected features. Here we have disabled the toggle feature and configured the `alt` key must be pressed to add features to the current selection and `shift` to remove then. This is done through the properties `toggleCondigion`, `addCondition` and `removeCondition`:

```
    var selectInteraction = new ol.interaction.Select({
        ...
        toggleCondition: ol.events.condition.never,
        addCondition: ol.events.condition.altKeyOnly,
        removeCondition: ol.events.condition.shiftKeyOnly
    });
```

Finally, the select interaction allows to specify the set of layers allowed to be selectable using the `layers` property. The simplest way is to pass an array of references, for example, `layers: [citiesLayer, limitsLayer]` but the interaction does not allows to change it later (there is no `set` method). In the sample we have used a function to specify the selectable layers, which is the most flexible way:

```
    var selectInteraction = new ol.interaction.Select({
        layers: function(layer) {
            var limits = $('#limits').is(':checked');
            var cities = $('#cities').is(':checked');

            return (limits && layer === limitsLayer) ||
                (cities && layer === citiesLayer);
        },
        ...
    });
```

The function specified in the `layers` property is invoked for each contained layer in the map and must return `true` or `false` if it can be selectable. The code simply checks which layer checkbox is activated and returns `true` accordingly.

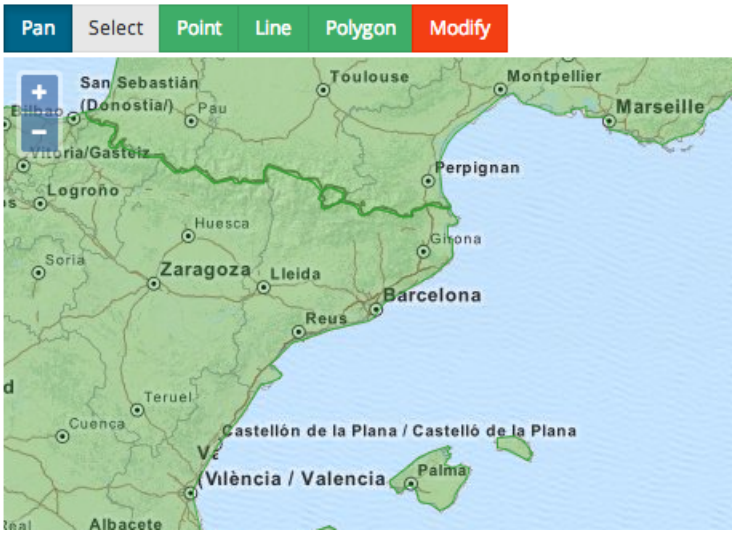
7.3.6.4 See also

- See Selecting features within a vector box to know how to select those features that intersects with a given vector box.

7.3.7 Editing features

7.3.7.1 Goal

The goal of this recipe is to create a basic vector editing application. We are going to create a toolbar were user could select pan the map, select features, draw points, lines or polygons and, finally, modify features:



Editing features

Note, it is out of the scope of this example how to send the changes to the server side.

7.3.7.2 How to do it...

As always start creating the HTML elements for the toolbar and the map:

```
    <div class="btn-group btn-group-sm" role="group" aria-label="Draw">
        <button id="pan" type="button" class="btn btn-primary">Pan</button>
        <button id="select" type="button" class="btn btn-default">Select</button>
        <button id="point" type="button" class="btn btn-success">Point</button>
        <button id="line" type="button" class="btn btn-success">Line</button>
        <button id="polygon" type="button" class="btn btn-success">Polygon</butt\
on>
        <button id="modify" type="button" class="btn btn-danger">Modify</button>
    </div>

    <div id="map" class="map"></div>
```

Next create a vector layer, so the map has an initial set of content we can modify. Here we are going to create a vector layer, with the World's administrative limits, from a TopoJSON file and using a green style:

```
var limitsLayer = new ol.layer.Vector({
    source: new ol.source.StaticVector({
        url: 'data/world_limits.json',
        format: new ol.format.TopoJSON(),
        projection: 'EPSG:3857'
    }),
    style: new ol.style.Style({
        fill: new ol.style.Fill({
            color: 'rgba(55, 155, 55, 0.3)'
        }),
        stroke: new ol.style.Stroke({
            color: 'rgba(55, 155, 55, 0.8)',
            width: 1
        }),
        image: new ol.style.Circle({
            radius: 7,
            fill: new ol.style.Fill({
                color: 'rgba(55, 155, 55, 0.5)',
            })
        })
    })
});
```

Now, initialize the map instance. We are using a MapQuest raster layer as background layer and put the previous vector layer on top:

```
var map = new ol.Map({
    target: 'map',  // The DOM element that will contains the map
    renderer: 'canvas', // Force the renderer to be used
    layers: [
        new ol.layer.Tile({
            source: new ol.source.MapQuest({
                layer: 'osm'
            })
        }),
        limitsLayer
    ],
    view: new ol.View({
        center: ol.proj.transform([2, 41], 'EPSG:4326', 'EPSG:3857'),
        zoom: 3
    })
});
```

Finally, add the code responsible to create and attach to the map the corresponding interaction depending on the selected button (note we have cut a bit the pasted code):

```
    var button = $('#pan').button('toggle');
    var interaction;
    $('div.btn-group button').on('click', function(event) {
        var id = event.target.id;

        // Toggle buttons
        button.button('toggle');
        button = $('#'+id).button('toggle');
        // Remove previous interaction
        map.removeInteraction(interaction);
        // Update active interaction
        switch(event.target.id) {
            case "select":
                interaction = new ol.interaction.Select();
                map.addInteraction(interaction);
                break;
            case "point":
                interaction = new ol.interaction.Draw({
                    type: 'Point',
                    source: limitsLayer.getSource()
                });
                map.addInteraction(interaction);
                break;
            ...
            case "modify":
                interaction = new ol.interaction.Modify({
                    features: new ol.Collection(limitsLayer.getSource().getFeatu\
res())
                });
                map.addInteraction(interaction);
                break;
            default:
                break;
        }
    });
```

7.3.7.3 How it works...

Note we have created a map instance without specifying any interactions. That means, by default, the map uses the default set of interactions (like pan, zoom, etc).

The main part of this samples is located in the function that controls which interaction is active depending on the toolbar button we click on. To help us on this work, we have make use of to global variables (not a good practice) to know what are the current selected button and interaction:

```
var button = $('#pan').button('toggle');
var interaction;
```

Note, that when application starts the selected button is the pan button and there is no one of our interactions activated.

Next, we have registered a listener function that is invoked each time a button is clicked:

```
$('div.btn-group button').on('click', function(event) {
    var id = event.target.id;
    ...
});
```

Within the function we get the `id` of the clicked button but, before handle the event, we change the selected button and deactivate the previous, if any, activated interactions (removing it from the map with `map.removeInteraction()` method):

```
// Toggle buttons
button.button('toggle');
button = $('#'+id).button('toggle');
// Remove previous interaction
map.removeInteraction(interaction);
```

Now, given the `id` variable we can handle the event accordingly. This is done using a `switch` statement (note we have cut the next code):

```
switch(event.target.id) {
    case "select":
        interaction = new ol.interaction.Select();
        map.addInteraction(interaction);
        break;
    case "point":
        interaction = new ol.interaction.Draw({
            type: 'Point',
            source: limitsLayer.getSource()
        });
        map.addInteraction(interaction);
        break;
```

```
13          ...
14          case "modify":
15              interaction = new ol.interaction.Modify({
16                  features: new ol.Collection(limitsLayer.getSource().getFeatures(\
17  ))
18              });
19              map.addInteraction(interaction);
20              break;
21          default:
22              break;
23      }
```

When the `select` button is selected, we create a new `ol.interaction.Select` instance and add it to the map with `map.addInteraction()` method.

When user selects `point`, `line` or `polygon` buttons we create an `ol.interaction.Draw` instance specifying the kind of geometry allowed to be drawn, in this example, it is limited to `Point`, `LineString` and `Polygon`. In addition, we pass to the `source` property a reference to the source of the existing `limitsLayer` vector layer, so the new features will be added to that layer.

When the `modify` button is clicked, we create a new `ol.interaction.Modify` instance. This interaction requires we set a collection containing the set of features allowed to be modified and pass in the `features` property. In our case we want to modify any feature of the `limitsLayer` layer so we have created an `ol.Collection` containing all the features of the layer's source: `new ol.Collection(limitsLayer.getSource().getFeatures())`.

Finally, if user select the `pan` button, no interaction is attached to the map and, because, previously we have removed the current interaction from the map that means the default pan interaction is working again.

In the sample, we have make no use of the `style` property of the interactions. Using it we can change the look of the features affected by the interaction, for example, when they are selected, new created or while modifying.

Note also, in addition to play with interactions adding to or removing from the map (using `map.addInteraction()` and `map.removeInteraction()`), we can also activate or deactivate interactions (using `setActive()` and `getActive()`), which results in a similar effect.

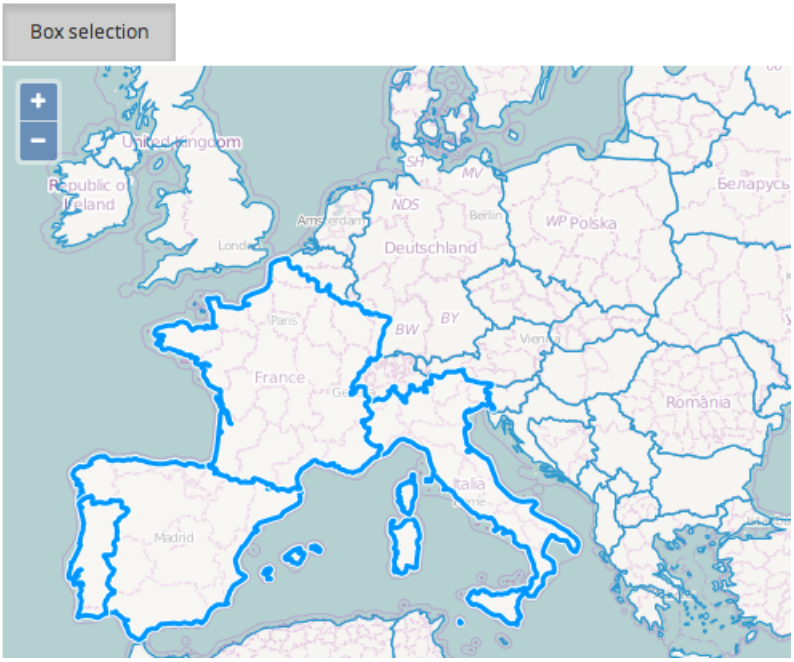
7.3.7.4 See also

- Reading and writing features through the source class example at Data sources and formats chapter, to see how to read and write features using the source and format class.

7.3.8 Selecting features within a vector box

7.3.8.1 Goal

In this example we are going to demonstrate how we can select those features that intersects with a vector box. The goal is to show the flexibility of interactions and we can combine them to create nice controls:



Box selection

The `Box selection` button will activate the interaction that will draw a vector box and select those features that intersects with it.

7.3.8.2 How to do it...

Start creating the HTML components for the button and the map:

```
    <button id="select" type="button" class="btn btn-default btn-sm" data-toggle\
="button" aria-pressed="false">Box selection</button>

    <div id="map" class="map"></div>
```

Now create a vector layer with some features to be selected:

```
    var limitsLayer = new ol.layer.Vector({
        source: new ol.source.StaticVector({
            url: 'data/world_limits.json',
            format: new ol.format.TopoJSON(),
            projection: 'EPSG:3857'
        })
    });
```

Create an initialize the map instance:

```
    var map = new ol.Map({
        target: 'map',  // The DOM element that will contains the map
        renderer: 'canvas', // Force the renderer to be used
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            }),
            limitsLayer
        ],
        view: new ol.View({
            center: ol.proj.transform([2, 40], 'EPSG:4326', 'EPSG:3857'),
            zoom: 2
        })
    });
```

Know, create the code that allow to draw a vector box and select those features intersection the box:

```
    var selectInteraction = new ol.interaction.Select({
        condition: ol.events.condition.never
    });
    var dragBoxInteraction = new ol.interaction.DragBox({
        style: new ol.style.Style({
            stroke: new ol.style.Stroke({
                color: [250, 25, 25, 1]
            })
        })
    });
    dragBoxInteraction.on('boxend', function(event) {
        var selectedFeatures = selectInteraction.getFeatures();
        selectedFeatures.clear();
        var extent = dragBoxInteraction.getGeometry().getExtent();
        limitsLayer.getSource().forEachFeatureIntersectingExtent(extent, functio\
```

```
n(feature) {
            selectedFeatures.push(feature);
        });
    });
```

Finally, add a listener to add to or remove from the map the interactions when the button is toggled:

```
    $('#select').on('click', function(event) {
        var checked = !$('#select').hasClass('active');
        if(checked) {
            map.addInteraction(selectInteraction);
            map.addInteraction(dragBoxInteraction);
        } else {
            map.removeInteraction(selectInteraction);
            map.removeInteraction(dragBoxInteraction);
        }
    });
```

7.3.8.3 How it works...

The magic of the code resides in the fact we are merging the work of two interactions at the same time: `ol.interaction.DragBox` which draws a box and `ol.interaction.Select` that allows to render a set of features using a different style. So, the idea is to use the `ol.interaction.DragBox` interaction to drag a box, compute those features that intersects with the box and add them to the `ol.interaction.Select` interaction.

First, when the user toggles the button we check if it is checked or not and, depending on the value, we add to or remove from the map both interactions.

By default, the `ol.interaction.Select` is applied when the user presses the `shift` key. In this case we do not want the user can select features by clicking on it but only through the box selection. Because of this, we have set the `condition` property to `ol.events.condition.never`, that means the `ol.interaction.Select` interaction will never handle map events:

```
    var selectInteraction = new ol.interaction.Select({
        condition: ol.events.condition.never
    });
```

Next, we have initialized the `ol.interaction.DragBox` interaction. It has not much mystery, we only need to remember we need to apply a `style` property, which is mandatory:

```
    var dragBoxInteraction = new ol.interaction.DragBox({
        style: new ol.style.Style({
            stroke: new ol.style.Stroke({
                color: [250, 25, 25, 1]
            })
        })
    });
```

Finally, we need to register listener function on the `boxend` event, triggered by the `ol.interaction.DragBox` when the selection finishes. This listener is responsible to get the box geometry and compute which features intersect with it:

```
    dragBoxInteraction.on('boxend', function(event) {
        var selectedFeatures = selectInteraction.getFeatures();
        selectedFeatures.clear();
        var extent = dragBoxInteraction.getGeometry().getExtent();
        limitsLayer.getSource().forEachFeatureIntersectingExtent(extent, functio\
n(feature) {
            selectedFeatures.push(feature);
        });
    });
```

Selected features are managed by the `ol.interaction.Select` interactions, because of this, we store a reference to the managed features on the `selectedFeatures` variable:

```
    var selectedFeatures = selectInteraction.getFeatures();
```

Using the `selectedFeatures` reference we can add, remove or clean the features managed by the interaction at any time.

To compute the selected features by the box we use the `forEachFeatureIntersectingExtent()` method on the `ol.source.Vector` source class. This method accepts an `extent` and a `callback` function, that is invoked for each feature that intersects with the extent. This way, we add each selected feature to the `selectedFeatures` variable, which mean we are adding them to the `ol.interaction.Select` interaction.