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

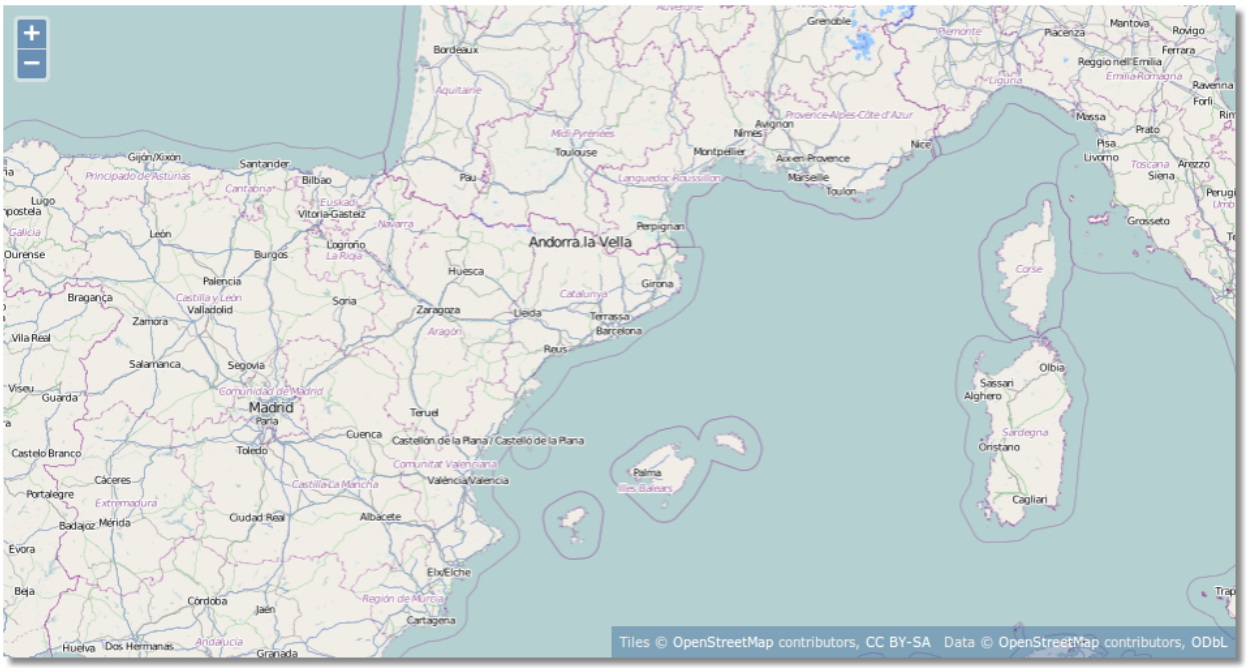
The `ol.Map` is the class that allows to handle the concept of map in our application. We can add or remove *layers*, *controls*, *overlays* and *interactions*.

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);
```

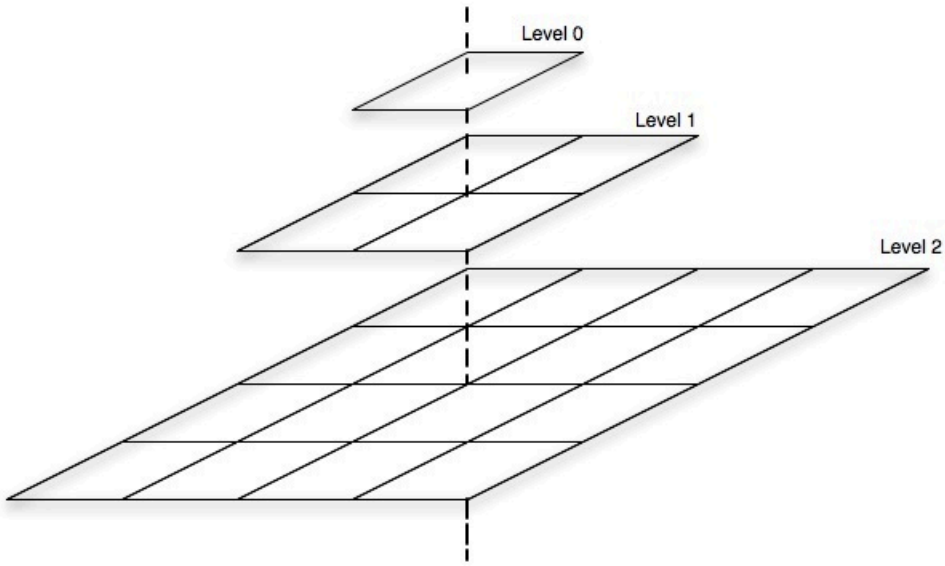
Note the rotation angle must be expressed in radians, because of this we need to transform degrees to radians.

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

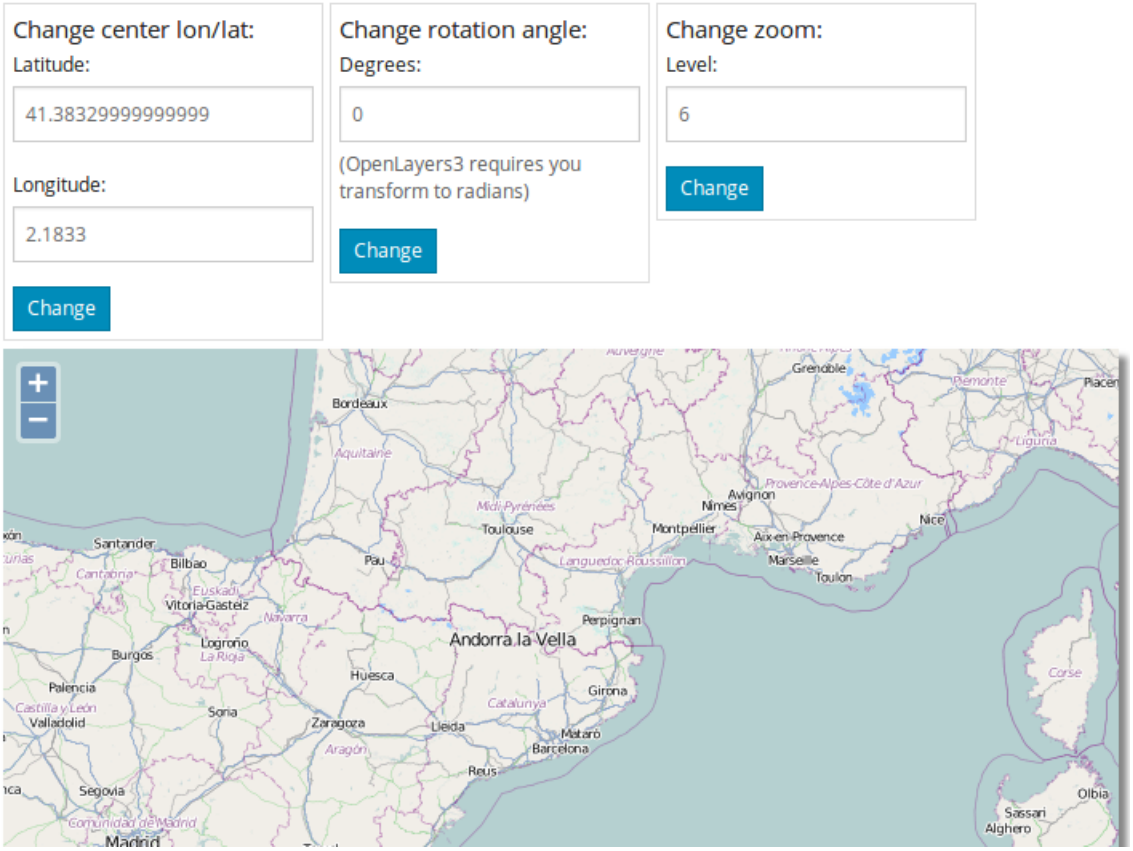
```
    view: new ol.View({
        center: ol.proj.transform([2.1833, 41.3833], 'EPSG:4326', 'EPSG:3857'),
        zoom: 6
    })
```

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:

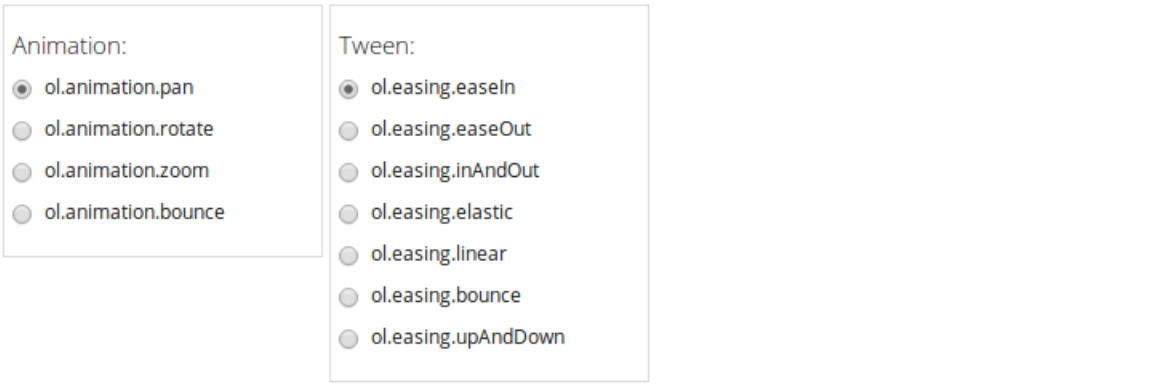
```
map.getView().setZoom($('#level').val());
```

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.

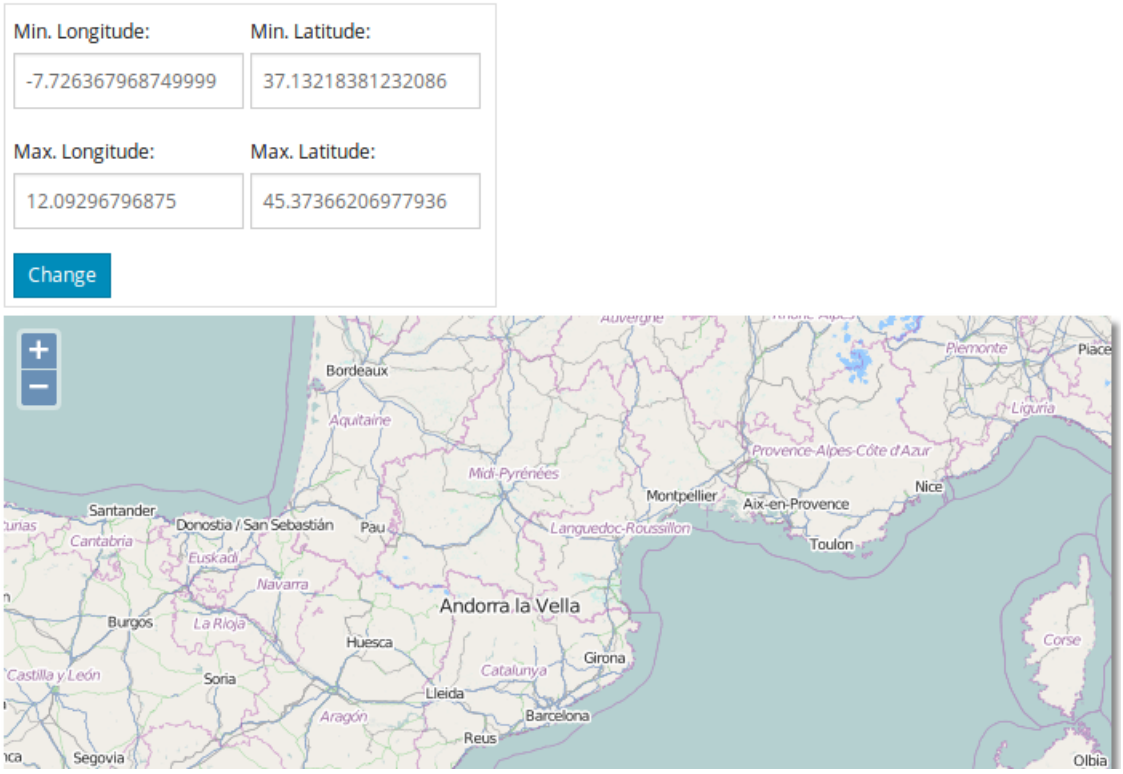
So that the animation takes effect we need to change the view property which animation modifies. Next code checks the selected animation and changes the appropriate property:

```
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());
```