OpenLayers Workshop Documentation

Release 2.13

OpenGeo and Prodevelop

July 03, 2014

Contents

1 OpenLayers Basics **3**
- 1.1 Creating a Map . . . 3
- 1.2 Dissecting Your Map . . . 4
- 1.3 OpenLayers Resources . . . 6

2 Working With Layers **7**
- 2.1 Web Map Service Layers . . . 7
- 2.2 Cached Tiles . . . 9
- 2.3 Proprietary Layers . . . 12
- 2.4 Vector Layers . . . 14

3 Working With Controls **19**
- 3.1 Creating an Overview Map . . . 19
- 3.2 Displaying a Scale Bar . . . 21
- 3.3 Selecting Features . . . 23

4 Vector Layers **29**
- 4.1 Working with Vector Layers . . . 29
- 4.2 Creating New Features . . . 31
- 4.3 Persisting Features . . . 33
- 4.4 Understanding Style . . . 34
- 4.5 Styling Vector Layers . . . 36

5 For more information **41**
- 5.1 OpenLayers Home . . . 41
- 5.2 OpenLayers Documentation . . . 41
- 5.3 Mailing list . . . 41
- 5.4 Bug tracking . . . 41
- 5.5 IRC . . . 41
- 5.6 OpenGeo . . . 41
- 5.7 Prodevelop . . . 42

Welcome to the **OpenLayers Workshop**. This workshop is designed to give you a comprehensive overview of OpenLayers as a web mapping solution. The exercises assume that you have set up a local GeoServer with the workshop data as described in the `setup` page.

The workshop has been adapted from the official Open Geo OpenLayers workshop for the Open Source Opportunities in GIS Summer School. This course is coordinated by the GIS and Remote Sensing Centre of the University of Girona in collaboration with the Nottingham Geospatial Institute and Prodevelop.

The instructors of the OpenLayers workshop are:

- Alberto Romeu
  - @alrocar
  - aromeu [at] prodevelop [dot] es
- Jorge Sanz
  - @xurxosanz
  - jsanz [at] prodevelop [dot] es

This workshop is presented as a set of modules. In each module the reader will perform a set of tasks designed to achieve a specific goal for that module. Each module builds upon lessons learned in previous modules and is designed to iteratively build up the reader's knowledge base.

The following modules will be covered in this workshop:

OpenLayers Basics Learn how to add a map to a webpage with OpenLayers.

Working With Layers Learn about raster and vector layers.

Working With Controls Learn about using map controls.

Vector Layers Explore vector layers in depth.

For more information More information about Open Layers project and resources

CHAPTER 1

OpenLayers Basics

OpenLayers is a library for building mapping applications in a browser. This workshop covers the well established 2.x series, while development of a new 3.x series with a different API has begun. The library lets developers integrate data from a variety of sources, provides a friendly API, and results in engaging and responsive mapping applications.

This module introduces fundamental OpenLayers concepts for creating a map.

What this module covers

In this module you will create a map, dissect your map to understand the parts, and get links to additional learning resources.

1.1 Creating a Map

In OpenLayers, a map is a collection of layers and various controls for dealing with user interaction. A map is generated with three basic ingredients: *markup*, *style declarations*, and *initialization code*.

1.1.1 Working Example

Let's take a look at a fully working example of an OpenLayers map.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <script>
            var map = new OpenLayers.Map("map-id");
            var imagery = new OpenLayers.Layer.WMS(
                "Global Imagery",
                "http://maps.opengeo.org/geowebcache/service/wms",
                {layers: "bluemarble"}
```

```
            );
            map.addLayer(imagery);
            map.zoomToMaxExtent();
        </script>
    </body>
</html>
```

Tasks

1. Copy the text above into a new file called `map.html`, and save it in the root of the workshop folder.
2. Open the working map in your web browser: http://localhost:8080/ol_workshop/map.html

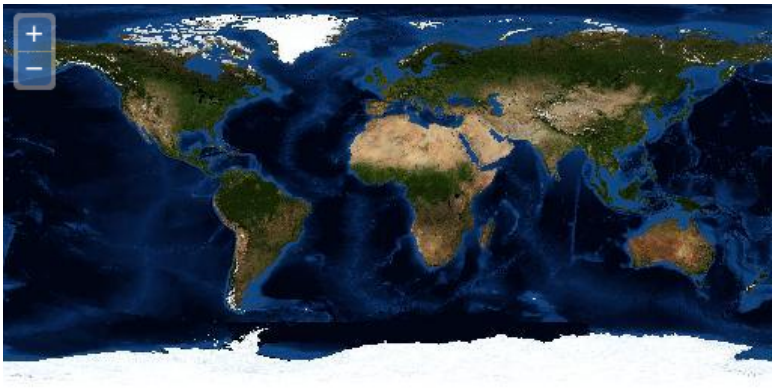


Figure 1.1: A working map of displaying imagery of the world.

Having successfully created our first map, we'll continue by looking more closely at *the parts*.

1.2 Dissecting Your Map

As demonstrated in the *previous section*, a map is generated by bringing together *markup*, *style declarations*, and *initialization code*. We'll look at each of these parts in a bit more detail.

1.2.1 Map Markup

The markup for the map in the *previous example* generates a single document element:

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

This `<div>` element will serve as the container for our map viewport. Here we use a `<div>` element, but the container for the viewport can be any block-level element.

In this case, we give the container an `id` attribute so we can reference it easily elsewhere.

1.2.2 Map Style

OpenLayers comes with a default stylesheet that specifies how map-related elements should be styled. We've explicitly included this stylesheet in the `map.html` page (`<link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">`).

OpenLayers doesn't make any guesses about the size of your map. Because of this, following the default stylesheet, we need to include at least one custom style declaration to give the map some room on the page.

```
<link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
<style>
    #map-id {
        width: 512px;
        height: 256px;
    }
</style>
```

In this case, we're using the map container's `id` value as a selector, and we specify the width (`512px`) and the height (`256px`) for the map container.

The style declarations are directly included in the `<head>` of our document. In most cases, your map related style declarations will be a part of a larger website theme linked in external stylesheets.

Note: OpenLayers enforces zero margin and padding on the element that you use for the viewport container. If you want your map to be surrounded by some margin, wrap the viewport container in another element with margin or padding.

1.2.3 Map Initialization

The next step in generating your map is to include some initialization code. In our case, we have included a `<script>` element at the bottom of our document `<body>` to do the work:

```
<script>
    var map = new OpenLayers.Map("map-id");
    var imagery = new OpenLayers.Layer.WMS(
        "Global Imagery",
        "http://maps.opengeo.org/geowebcache/service/wms",
        {layers: "bluemarble"}
    );
    map.addLayer(imagery);
    map.zoomToMaxExtent();
</script>
```

Note: The order of these steps is important. Before the our custom script can be executed, the OpenLayers library must be loaded. In our example, the OpenLayers library is loaded in the `<head>` of our document with `<script src="openlayers/lib/OpenLayers.js"></script>`.

Similarly, our custom map initialization code (above) cannot run until the document element that serves as the viewport container, in this case `<div id="map-id"></div>`, is ready. By including the initialization code at the end of the document `<body>`, we ensure that the library is loaded and the viewport container is ready before generating our map.

Let's look in more detail at what the map initialization script is doing. The first line of our script creates a new `OpenLayers.Map` object:

```
var map = new OpenLayers.Map("map-id");
```

We use the viewport container's `id` attribute value to tell the map constructor where to render the map. In this case, we pass the string value `"map-id"` to the map constructor. This syntax is a shortcut for convenience. We could be more explicit and provide a direct reference to the element (e.g. `document.getElementById("map-id")`).

The next several lines create a layer to be displayed in our map:

```
var imagery = new OpenLayers.Layer.WMS(
    "Global Imagery",
    "http://maps.opengeo.org/geowebcache/service/wms",
    {layers: "bluemarble"}
);
map.addLayer(imagery);
```

Don't worry about the syntax here if this part is new to you. Layer creation will be covered in another module. The important part to understand is that our map view is a collection of layers. In order to see a map, we need to include at least one layer.

The final step is to set the geographical limits (xmin, ymin, xmax, ymax) of the map display. This *extent* specifies the minimum bounding rectangle of a map area. There are a number of ways to specify the initial extent. In our example, we use a simple request to zoom to the maximum extent. By default, the maximum extent is the world in geographic coordinates:

```
map.zoomToMaxExtent();
```

You've successfully dissected your first map! Next let's *learn more* about developing with OpenLayers.

1.3 OpenLayers Resources

The OpenLayers library contains a wealth of functionality. Though the developers have worked hard to provide examples of that functionality and have organized the code in a way that allows other experienced developers to find their way around, may users find it a challenge to get started from scratch.

1.3.1 Learn by Example

New users will most likely find diving into the OpenLayer's example code and experimenting with the library's possible functionality the most useful way to begin.

- http://openlayers.org/dev/examples/

1.3.2 Browse the Documentation

For further information on specific topics, browse the growing collection of OpenLayers documentation.

- http://docs.openlayers.org/

1.3.3 Find the API Reference

After understanding the basic components that make-up and control a map, search the API reference documentation for details on method signatures and object properties.

- http://dev.openlayers.org/apidocs/files/OpenLayers-js.html

1.3.4 Join the Community

OpenLayers is supported and maintained by a community of developers and users like you. Whether you have questions to ask or code to contribute, you can get involved by signing up for one of the mailing lists and introducing yourself.

- Users list http://lists.osgeo.org/mailman/listinfo/openlayers-users
- Developers list http://lists.osgeo.org/mailman/listinfo/openlayers-dev

CHAPTER 2

Working With Layers

Every OpenLayers map has one or more layers. Layers display the geospatial data that the user sees on the map.

What this module covers

This module covers the basics of working with map layers. In this module you will use a standard WMS layer, work with cached tiles, use a proprietary tile service, and render data client side.

2.1 Web Map Service Layers

When you add a layer to your map, the layer is typically responsible for fetching the data to be displayed. The data requested can be either raster or vector data. You can think of raster data as information rendered as an image on the server side. Vector data is delivered as structured information from the server and may be rendered for display on the client (your browser).

There are many different types of services that provide raster map data. This section deals with providers that conform with the OGC (Open Geospatial Consortium, Inc.) Web Map Service (WMS) specification.

2.1.1 Creating a Layer

We'll start with a fully working map example and modify the layers to get an understanding of how they work.

Let's take a look at the following code:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <script>
```

```
            var map = new OpenLayers.Map("map-id");
            var imagery = new OpenLayers.Layer.WMS(
                "Global Imagery",
                "http://maps.opengeo.org/geowebcache/service/wms",
                {layers: "bluemarble"}
            );
            map.addLayer(imagery);
            map.zoomToMaxExtent();
        </script>
    </body>
</html>
```

Tasks

1. If you haven't already done so, save the text above as `map.html` in the root of your workshop directory.
2. Open the page in your browser to confirm things work: http://localhost:8080/ol_workshop/map.html

2.1.2 The OpenLayers.Layer.WMS Constructor

The `OpenLayers.Layer.WMS` constructor requires 3 arguments and an optional fourth. See the API reference for a complete description of these arguments.

```
var imagery = new OpenLayers.Layer.WMS(
    "Global Imagery",
    "http://maps.opengeo.org/geowebcache/service/wms",
    {layers: "bluemarble"}
);
```

The first argument, `"Global Imagery"`, is a string name for the layer. This is only used by components that display layer names (like a layer switcher) and can be anything of your choosing.

The second argument, `"http://maps.opengeo.org/geowebcache/service/wms"`, is the string URL for a Web Map Service.

The third argument, `{layers: "bluemarble"}` is an object literal with properties that become parameters in our WMS request. In this case, we're requesting images rendered from a single layer identified by the name `"bluemarble"`.

Tasks

1. This same WMS offers a layer named `"openstreetmap"`. Change the value of the `layers` param from `"bluemarble"` to `"openstreetmap"`.
2. If you just change the name of the layer and refresh your map you will meet a friend of any OpenLayers developer: our *loved* pink tiles. With Chrome, you can right click on any of them and go to *Open Image in New Tab* to get an idea of the problem.

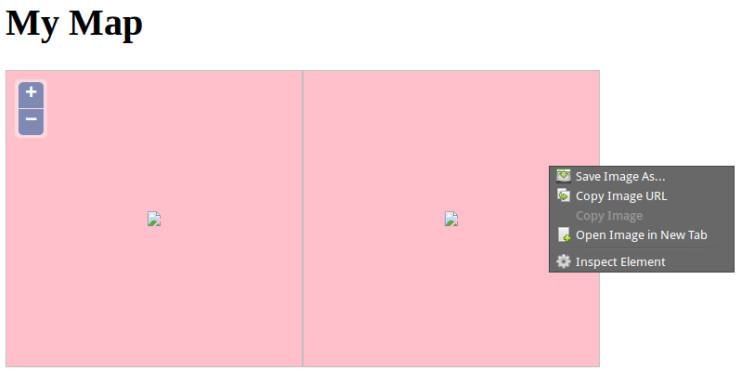


Figure 2.1: When things go weird, we see the pink tiles.

3. In addition to the `layers` parameter, a request for WMS imagery allows for you to specify the image format. The default for this layer is `"image/jpeg"`. Try adding a second property in the params object named `format`. Set the value to another image type (e.g. `"image/png"`).

   Your revised OpenLayers.Layer.WMS Constructor should look like:

```
var imagery = new OpenLayers.Layer.WMS(
    "Global Imagery",
    "http://maps.opengeo.org/geowebcache/service/wms",
    {layers: "openstreetmap", format: "image/png"}
);
```

4. Save your changes and reload the map: http://localhost:8080/ol_workshop/map.html



Figure 2.2: A map displaying the `"openstreetmap"` layer as `"image/png"`.

Having worked with dynamically rendered data from a Web Map Service, let's move on to learn about *cached tile services*.

2.2 Cached Tiles

By default, the WMS layer makes requests for 256 x 256 (pixel) images to fill your map viewport and beyond. As you pan and zoom around your map, more requests for images go out to fill the areas you haven't yet visited. While your browser will cache some requested images, a lot of processing work is typically required for the server to dynamically render images.

Since tiled layers (such as the WMS layer) make requests for images on a regular grid, it is possible for the server to cache these image requests and return the cached result next time you (or someone else) visits the same area - resulting in better performance all around.

2.2.1 OpenLayers.Layer.XYZ

The Web Map Service specification allows a lot of flexibility in terms of what a client can request. Without constraints, this makes caching difficult or impossible in practice.

At the opposite extreme, a service might offer tiles only at a fixed set of zoom levels and only for a regular grid. These can be generalized as XYZ layers - you can consider X and Y to indicate the column and row of the grid and Z to represent the zoom level.

2.2.2 OpenLayers.Layer.OSM

The OpenStreetMap (OSM) project is an effort to collect and make freely available map data for the world. OSM provides a few different renderings of their data as cached tile sets. These renderings conform to the basic *XYZ grid* arrangement and can be used in an OpenLayers map. The `OpenLayers.Layer.OSM` constructor accesses OpenStreetMap tiles.

Tasks

1. Open the `map.html` file from the *previous section* in a text editor and change the map initialization code to look like the following:

```
<script>
    var center = new OpenLayers.LonLat(2.825, 41.985).transform(
        'EPSG:4326', 'EPSG:3857'
    );

    var map = new OpenLayers.Map("map-id", {projection: 'EPSG:3857'});

    var osm = new OpenLayers.Layer.OSM();
    map.addLayer(osm);

    map.setCenter(center, 16);
</script>
```

2. In the `<head>` of the same document, add a few style declarations for the layer attribution.

```
<style>
    #map-id {
        width: 512px;
        height: 256px;
    }
    .olControlAttribution {
        font-size: 10px;
        bottom: 5px;
        left: 5px;
    }
</style>
```

3. Save your changes, and refresh the page in your browser: http://localhost:8080/ol_workshop/map.html



Figure 2.3: A map with an OpenStreetMap layer.

A Closer Look

Projections

Review the first 3 lines of the initialization script:

```
var center = new OpenLayers.LonLat(2.825, 41.985).transform(
    'EPSG:4326', 'EPSG:3857'
);
```

Geospatial data can come in any number of coordinate reference systems. One data set might use geographic coordinates (longitude and latitude) in degrees, and another might have coordinates in a local projection with units in meters. A full discussion of coordinate reference systems is beyond the scope of this module, but it is important to understand the basic concept.

OpenLayers needs to know the coordinate system for your data. Internally, this is represented with an `OpenLayers.Projection` object. The `transform` function also takes strings that represent the coordinate reference system (`"EPSG:4326"` and `"EPSG:3857"` above).

Locations Transformed

The OpenStreetMap tiles that we will be using are in a Mercator projection. Because of this, we need to set the initial center using Mercator coordinates. Since it is relatively easy to find out the coordinates for a place of interest in geographic coordinates, we use the `transform` method to turn geographic coordinates (`"EPSG:4326"`) into Mercator coordinates (`"EPSG:3857"`).

Custom Map Options

```
var map = new OpenLayers.Map("map-id", {projection: 'EPSG:3857'});
```

In the *previous example* we used the default options for our map. In this example, we set a custom map projection.

Note: The projections we used here are the only projections that OpenLayers knows about. For other projections, the map options need to contain two more properties: `maxExtent` and `units`. This information can be looked up at http://spatialreference.org/, using the EPSG code.

Layer Creation and Map Location

```
var osm = new OpenLayers.Layer.OSM();
map.addLayer(osm);
```

As before, we create a layer and add it to our map. This time, we accept all the default options for the layer.

```
map.setCenter(center, 9);
```

Finally, we give our map a center (in Mercator coordinates) and set the zoom level to `9`.

Style

```
.olControlAttribution {
    font-size: 10px;
    bottom: 5px;
    left: 5px;
}
```

A treatment of map controls is also outside the scope of this module, but these style declarations give you a sneak preview. By default, an `OpenLayers.Control.Attribution` control is added to all maps. This lets layers display attribution information in the map viewport. The declarations above alter the style of this attribution for our map (notice the small Copyright line at the bottom left of the map).

Having mastered layers with publicly available cached tile sets, let's move on to working with *proprietary layers*.

Bonus exercise

1. Review the OSM layer API documentation to how to load other OSM layers
2. Modify your layer initialization accordingly



Figure 2.4: A map with an OpenStreetMap based MapQuest layer.

Hint: You can go to the official OSM site to view the layers available, change to any of them and use the browser tools to look for the url pattern of those tiles.

2.3 Proprietary Layers

In previous sections, we displayed layers based on a standards compliant WMS (OGC Web Map Service) and a custom tile cache. Online mapping (or at least the tiled map client) was largely popularized

by the availability of proprietary map tile services. OpenLayers provides layer types that work with these proprietary services through their APIs.

In this section, we'll build on the example developed in the *previous section* by adding a layer using tiles from Bing, and we'll toss in a layer switcher so you can decide which layers you want visible.

2.3.1 Bing!

Let's add a Bing layer.

Tasks

1. In your `map.html` file, find where the OSM (OpenStreetMap) layer is added in the map initialization code. Below the `map.addLayer(osm);` line, add the following:

```
var bing = new OpenLayers.Layer.Bing({
    key: "AqTGBsziZHIJYYxgivLBf0hVdrAk9mWO5cQcb8Yux8sW5M8c8opEC2lZqKR1ZZXf",
    type: "Road",
});
map.addLayer(bing);
```

Note: The Bing tiles API requires that you register for an API key to use with your mapping application. The example here uses an API key that you should not use in production. To use the Bing layer in production, register for an API key at https://www.bingmapsportal.com/.

---

2. Now that we have more than one layer in our map, it is time to add a layer switcher that controls layer visibility. Somewhere in your map initialization code (below the part where we create the `map`), include the following to create a layer switcher and add it to the map:

```
map.addControl(new OpenLayers.Control.LayerSwitcher());
```

3. Save your changes and reload `map.html` in your browser: http://localhost:8080/ol_workshop/map.html
4. Open the Layer Switcher at the upper right-hand corner of the map view, and select "Bing Roads".



Figure 2.5: A map with a bing layer and OpenStreetMap tiles.

Complete Working Example

Your revised `map.html` (without the bonus exercise from the previous section) file should look something like this:

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

```
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
            .olControlAttribution {
                font-size: 10px;
                bottom: 5px;
                left: 5px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <script>
            var center = new OpenLayers.LonLat(-93.27, 44.98).transform(
                'EPSG:4326', 'EPSG:3857'
            );

            var map = new OpenLayers.Map("map-id", {projection: 'EPSG:3857'});

            var osm = new OpenLayers.Layer.OSM();
            map.addLayer(osm);

            var bing = new OpenLayers.Layer.Bing({
                key: "AqTGBsziZHIJYYxgivLBf0hVdrAk9mWO5cQcb8Yux8sW5M8c8opEC2lZqKR1ZZXf",
                type: "Road",
            });
            map.addLayer(bing);

            map.addControl(new OpenLayers.Control.LayerSwitcher());

            map.setCenter(center, 9);
        </script>
    </body>
</html>
```

Next we'll move on from raster layers and begin working with *vector layers*.

2.4 Vector Layers

Previous sections in this module have covered the basics of raster layers with OpenLayers. This section deals with vector layers - where the data is rendered for viewing in your browser.

OpenLayers provides facilities to read existing vector data from the server, make modifications to feature geometries, and determine how features should be styled in the map.

Though browsers are steadily improving in terms of JavaScript execution speed (which helps in parsing data), there are still serious rendering bottlenecks which limit the quantity of data you'll want to use in practice. The best advice is to try your application in all the browsers you plan to support, to limit the data rendered client side until performance is acceptable, and to consider strategies for effectively conveying information without swamping your browser with too many vector features (the technical vector rendering limits of your browser often match the very real limitations of your users to absorb information).

2.4.1 Rendering Features Client-Side

Let's go back to the *WMS example* to get a basic world map. We'll add some feature data on top of this in a vector layer.

Tasks

1. Open `map.html` in your text editor and copy in the contents of the initial *WMS example*. Save your changes and confirm that things look good in your browser: http://localhost:8080/ol_workshop/map.html
2. In your map initialization code (anywhere after the `map` construction), paste the following. This adds a new vector layer to your map that requests a set of features stored in GeoRSS:

```
var earthquakes = new OpenLayers.Layer.Vector("Earthquakes", {
    strategies: [new OpenLayers.Strategy.Fixed()],
    protocol: new OpenLayers.Protocol.HTTP({
        url: "data/layers/7day-M2.5.xml",
        format: new OpenLayers.Format.GeoRSS()
    })
});
map.addLayer(earthquakes);
```

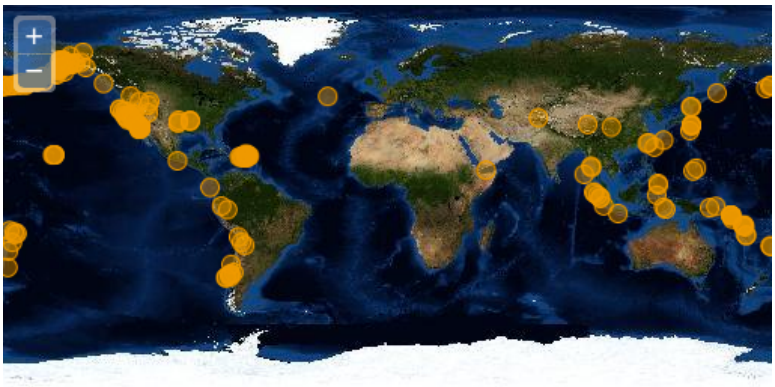


Figure 2.6: World map with orange circles representing earthquake locations.

A Closer Look

Let's examine that vector layer creation to get an idea of what is going on.

```
var earthquakes = new OpenLayers.Layer.Vector("Earthquakes", {
    strategies: [new OpenLayers.Strategy.Fixed()],
    protocol: new OpenLayers.Protocol.HTTP({
        url: "data/layers/7day-M2.5.xml",
        format: new OpenLayers.Format.GeoRSS()
    })
});
```

The layer is given the title `"Earthquakes"` and some custom options. In the options object, we've included a list of `strategies` and a `protocol`. A full discussion of strategies and protocols is beyond the scope of this module, but here is a rough sketch of what they do:

- Strategies determine when your data is requested and how to handle that data once it has been turned into features. Strategies can also be used to trigger an update of your data when something has been modified. Strategies are ordered and independent–they can work with the results of a previous strategy, but they can't rely on other strategies being there.

- Protocols refer to communication protocols between client and server for reading and writing feature data. Protocols may have a format (parser) that is responsible for serializing and deserializing feature data.

In this case, we're using a fixed strategy. The fixed strategy triggers a single request for data and doesn't ever ask for updates. We're asking for data using the HTTP protocol, we provide the URL for the data, and we expect the features to be serialized as GeoRSS.

Bonus Tasks

1. The data for the vector layer comes from an earthquake feed published by the USGS (http://earthquake.usgs.gov/earthquakes/catalogs/). See if you can find additional data with spatial information in GeoRSS or another OpenLayers supported format. If you save another feed (or other document) representing spatial data in your `data` directory, you should be able to view it in a vector layer on your map.
2. The orange circles on the map represent `OpenLayers.Feature.Vector` objects on your `OpenLayers.Layer.Vector` layer. Each of these features has attribute data with `title`, `description`, and `link` properties.
   - Create a `report` function that will receive the event with the selected feature. The minimal code for your function could just use the browser console to print some data.

     ```
     // Report function
     var report = function(e){
         console.log(e.feature.id + ": " + e.feature.data.title)
     }
     ```

   - Add an `OpenLayers.Control.SelectFeature` control to your map, listen for the `featurehighlighted` event on the vector layer. We will see OpenLayers controls on the next section, so for now just paste this code at the end of your `init` function:

     ```
     // Create a select control on the earthquakes layer and assign
     // the function to the feature highlighted event
     var control = new OpenLayers.Control.SelectFeature(earthquakes,
         { eventListeners: { featurehighlighted: report} });

     // Add the control to the map and activate it
     map.addControl(control);
     control.activate();
     ```

   - Create some mark-up below the map, use identifiers and try to use the DOM function `document.getElementById` to retrieve the DOM tree element and `someElement.innerHTML=something` setter to change the HTML inside of `someElement`. You should get something like the image below:

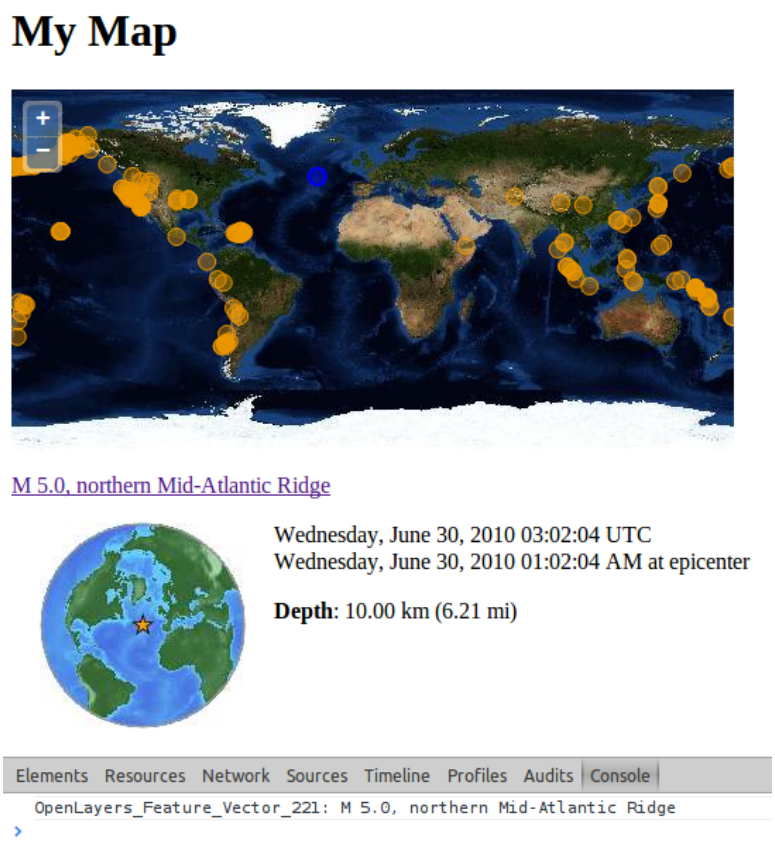


Figure 2.7: Adding a selection controls we can see additional vector information

- This exercise is quite similar to the OpenLayers select and highlight feature example.

CHAPTER 3

Working With Controls

In OpenLayers, controls provide a way for users to interact with your map. Some controls have a visual representation while others are invisible to the user. When you create a map with the default options, you are provided with a few visible default controls. These controls allow users to navigate with mouse movements (e.g., drag to pan, double-click to zoom in) and buttons to pan & zoom. In addition there are default controls that specify layer attribution and provide bookmarking of a location.

(Find a full list of available controls in the OpenLayers API documentation: http://dev.openlayers.org/apidocs/files/OpenLayers-js.html)

What this module covers

This module covers the basics of using controls in OpenLayers. In this module you will create an overview map, a scale line control, and a control to select features.

3.1 Creating an Overview Map

Online maps often contain a smaller *overview map* that displays the extent of the larger map. In OpenLayers, this is possible using the `OpenLayers.Control.OverviewMap` control.

Let's create a map with a single layer and then add an overview map control.

Tasks

1. Open a text editor and save the following page as `map.html` in the root of your workshop directory:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <script>
```

```
            var nyc = new OpenLayers.Bounds(
                -74.032, 40.685,
                -73.902, 40.876
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:4326"),
                restrictedExtent: nyc,
                maxResolution: 0.0005,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "New York City",
                "/geoserver/wms",
                {layers: "tiger-ny"}
            );
            map.addLayer(base);

            var center = new OpenLayers.LonLat(-73.96566, 40.7802);
            map.setCenter(center,4);

        </script>
    </body>
</html>
```

2. Open the working map in your web browser to confirm things look good: http://localhost:8080/ol_workshop/map.html.
3. We are now going to add an overview map with default options to confirm things are properly working. Somewhere in your map initialization code (after the creation of the `map` object), insert the following:

```
var overview = new OpenLayers.Control.OverviewMap({
    mapOptions: {
        projection: new OpenLayers.Projection("EPSG:4326"),
        restrictedExtent: nyc,
        maxResolution: 0.0015,
        numZoomLevels: 5
    }
});
map.addControl(overview);
```

4. Save your changes and refresh `map.html` in your browser: http://localhost:8080/ol_workshop/map.html
5. To see the overview map in action, open the *plus sign* at right of the map viewport.



An overview map control inside the main map viewport.

3.1.1 Discussion

The map in this example includes a few things you may not have seen before:

```
var nyc = new OpenLayers.Bounds(
    -74.032, 40.685,
    -73.902, 40.876
);
var map = new OpenLayers.Map("map-id", {
    projection: new OpenLayers.Projection("EPSG:4326"),
    restrictedExtent: nyc,
    maxResolution: 0.0005,
    numZoomLevels: 5
});
```

First, we define the map projection. The OpenLayers default is to construct maps in a geographic projection so in this case, as on the previous examples it shouldn't be needed.

The second thing to note is the use of the `restrictedExtent` property. This optional property restricts map panning to the given bounds. The imagery data used by the `tiger-nfc` layer is limited to these bounds. To keep users from panning off the edge of our base layer, we set the `restrictedExtent` to the bounds of the data.

The final set of custom options are related to map resolutions. By default, a map will be set up to view the entire world in two 256x256 tiles when all the way zoomed out. Since we want to focus on a very limited subset of the world, we set the `maxResolution` property. A value of 0.0005 means 0.0005 degrees per pixel (since our map is expressed in geographic coordinates). When users are zoomed all the way out, they will be seeing 0.0005 degrees per pixel (1.8 seconds). We also specify that we only want 5 zoom levels instead of the default 16 levels.

The overview map constructor also deserves a bit of discussion:

```
var overview = new OpenLayers.Control.OverviewMap({
    mapOptions: {
        projection: new OpenLayers.Projection("EPSG:4326"),
        restrictedExtent: nyc,
        maxResolution: 0.0015,
        numZoomLevels: 5
    }
});
map.addControl(overview);
```

Like the custom `map` above, customization to the `overview` map control must also be specified. So, for every non-default property set for the main map, we need a corresponding property for the map created by the control.

We want `projection`, `restrictedExtent` and `numZoomLevels` to be the same for the overview map as well as the main map. However, in order for the overview map to zoom "farther out" we want a different `maxResolution` property. The appropriate values for your application can be determined by trial and error or calculations based on how much data you want to show (given the map size). In this case 0.0015 degrees (5.4 seconds) is a appropriated value but experiment with other values and see the differences.

Next we'll build upon our map to include a *scale bar*.

3.2 Displaying a Scale Bar

Another typical widget to display on maps is a scale bar. OpenLayers provides an `OpenLayers.Control.ScaleLine` for just this. We'll continue building on the *previous example* by adding a scale bar.

3.2.1 Creating a ScaleLine Control

Tasks

1. Open the `map.html` produced in the *previous example* in your text editor.
2. Somewhere in the map initialization (below the `map` constructor), add the following code to create a new scale line control for your map:

```
var scaleline = new OpenLayers.Control.ScaleLine();
map.addControl(scaleline);
```

3. Save your changes and open `map.html` in your browser: http://localhost:8080/ol_workshop/map.html



A default (and very hard to see) scale bar in the bottom left-hand corner

3.2.2 Moving the ScaleLine Control

You may find the scale bar a bit hard to read over the Medford imagery. There are a few approaches to take in order to improve scale visibility. If you want to keep the control inside the map viewport, you can add some style declarations within the CSS of your document. To test this out, you can include a background color, padding and a small transparency to the scale bar with something like the following:

```
.olControlScaleLine {
    background: white;
    padding: 5px;

    /* IE 8 */
    -ms-filter: "progid:DXImageTransform.Microsoft.Alpha(Opacity=85)";

    /* Netscape */
    -moz-opacity: 0.85;

    /* Good browsers */
    opacity: 0.85;
}
```

However, for the sake of this exercise, let's say you think the map viewport is getting unbearably crowded. To avoid such over-crowding, you can display the scale in a different location. To accomplish this, we need to first create an additional element in our markup and then tell the scale control to render itself within this new element.

Tasks

1. Remove any scale style declarations, and create a new block level element in the `<body>` of your page. To make this element easy to refer to, we'll give it an `id` attribute. Insert the following markup somewhere in the `<body>` of your `map.html` page. (Placing the scale element right after the map viewport element `<div id="map-id"></div>` makes sense.):

   ```
   <div id="scaleline-id"></div>
   ```

2. Now modify the code creating the scale control so that it refers to the `scaleline-id` element:

   ```
   var scaleline = new OpenLayers.Control.ScaleLine({
       div: document.getElementById("scaleline-id")
   });
   ```

3. Save your changes and open `map.html` in your browser: http://localhost:8080/ol_workshop/map.html

4. You may decide that you want to add a bit of style to the widget. To do this, we can use the `#scaleline-id` selector in our CSS. Make the font smaller and give the widget some margin, by adding the following style declarations:

   ```
   #scaleline-id {
       margin: 10px;
       font-size: xx-small;
   }
   ```

5. Now save your changes and view `map.html` again in your browser: http://localhost:8080/ol_workshop/map.html



Figure 3.1: A custom styled scale bar outside the map viewport.

3.3 Selecting Features

So far we have been using WMS to render raster images and overlay them in OpenLayers. We can also pull features as vectors and draw them on top of a base map. One of the advantages of serving vector data is that users can interact with the data. In this example, we create a vector layer where users can select and view feature information.

3.3.1 Create a Vector Layer and a SelectFeature Control

Tasks

1. Let's start with the working example from the *previous section*. Open `map.html` in your text editor and make sure it looks something like the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
            #scaleline-id {
                margin: 10px;
                font-size: xx-small;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <div id="scaleline-id"></div>

        <script>
            var nyc = new OpenLayers.Bounds(
                -74.032, 40.685,
                -73.902, 40.876
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:4326"),
                maxExtent: nyc,
                restrictedExtent: nyc,
                maxResolution: 0.0005,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "New York City",
                "/geoserver/wms",
                {layers: "tiger-ny"}
            );
            map.addLayer(base);

            var center = new OpenLayers.LonLat( -73.987, 40.737);
            map.setCenter(center,3);

            var scaleline = new OpenLayers.Control.ScaleLine({
                div: document.getElementById("scaleline-id")
            });
            map.addControl(scaleline);
        </script>
    </body>
</html>
```

2. Next add a vector layer that requests the most important places of New York City. Because this data will be rendered client-side (i.e., by your browser), users can interact with its features. Somewhere in your map initialization (after the `map` object is created), add the following code to create

a vector layer that uses the WFS (OGC Web Feature Service) protocol:

```
var landmarks = new OpenLayers.Layer.Vector("NY Landmarks", {
    strategies: [new OpenLayers.Strategy.BBOX()],
    protocol: new OpenLayers.Protocol.WFS({
        version: "1.1.0",
        url: "/geoserver/wfs",
        featureType: "poly_landmarks",
        featureNS: "http://www.census.gov",
        srsName: "EPSG:4326"
    }),
    filter: new OpenLayers.Filter.Comparison({
                type: OpenLayers.Filter.Comparison.LIKE,
                property: "CFCC",
                value: "D*"
            })
});
map.addLayer(landmarks);
```

Note: Because we don't want all the features of the layer, we add to the layer constructor a new element, a `filter`. A filter is an object that declares some conditions over the data. OpenLayers add this condition to the requests to the server using the proper language depending on the protocol, if supported. In this case it will add to the BBOX strategy the necessary GML tags so GeoServer just answers with the features where the field `CFCC` has a value that starts with a capital D. The complete WFS request data (bounding box and filter) looks like this:

```
<wfs:GetFeature
  xmlns:wfs="http://www.opengis.net/wfs" service="WFS" version="1.1.0"
  xsi:schemaLocation="http://www.opengis.net/wfs http://schemas.opengis.net/wfs/1.1.0/wfs.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <wfs:Query typeName="feature:poly_landmarks" srsName="EPSG:4326"
    xmlns:feature="http://www.census.gov">
    <ogc:Filter xmlns:ogc="http://www.opengis.net/ogc">
      <ogc:And>
        <ogc:PropertyIsLike wildCard="*" singleChar="." escapeChar="!">
          <ogc:PropertyName>CFCC</ogc:PropertyName>
          <ogc:Literal>D*</ogc:Literal>
        </ogc:PropertyIsLike>
        <ogc:BBOX>
          <gml:Envelope xmlns:gml="http://www.opengis.net/gml" srsName="EPSG:4326">
            <gml:lowerCorner>-74.019 40.721</gml:lowerCorner>
            <gml:upperCorner>-73.955 40.753</gml:upperCorner>
          </gml:Envelope>
        </ogc:BBOX>
      </ogc:And>
    </ogc:Filter>
  </wfs:Query>
</wfs:GetFeature>
```

1. With the `landmarks` layer requesting and rendering features, we can create a control that allows users to select them. In your map initialization code, add the following *after* the creation of your `buildings` layer:

   ```
   var select = new OpenLayers.Control.SelectFeature([landmarks]);
   map.addControl(select);
   select.activate();
   ```

2. Save your changes to `map.html` and open the page in your browser: http://localhost:8080/ol_workshop/map.html. To see feature selection in action, use the mouse-click to select a building:



Figure 3.2: Using a control to select features from a vector layer.

3.3.2 Displaying Building Information on Feature Selection

Note: This section will complete the bonus exercise proposed on your first approach to OpenLayers vector data.

We can use layer events to respond to feature selection. This is useful for displaying specific feature information to the user. The `featureselected` event is triggered on a vector layer each time a feature is selected. Here we add a listener for this event that will display feature information below the map.

Tasks

1. First we need to add an element to display the feature information. Open `map.html` in your text editor and insert the following markup into the `<body>` of your page.

   ```
   <div id="output-id"></div>
   ```

2. Next we add some style declarations so that the feature information output doesn't sit on top of the scale bar. Give your output element some margin, by adding the following within the `<style>` element:

   ```
   #output-id {
       margin: 10px 220px;
   }
   ```

3. Finally, we create a listener for the `featureselected` event that will display feature information in the output element. Insert the following in your map initialization code after the creation of the `landmarks` layer:

   ```
   landmarks.events.on({
       featureselected: function(event) {
           var feature = event.feature;
           var id = feature.attributes.CFCC;
           var area = feature.geometry.getGeodesicArea();
           var output = feature.attributes.LANAME
               + " (" + id + ")" + " Area: " + area.toFixed(0) +" m2";
           document.getElementById("output-id").innerHTML = output;
       }
   });
   ```

   Note: See how we get the area from the `geometry` object. As we are dealing with geographic coordinates, OpenLayers provide a `getGeodesicArea`. When we work with projected coordinates we should use instead the `getArea` method.

4. Save your changes and refresh the `map.html` page in your browser: http://localhost:8080/ol_workshop/map.html

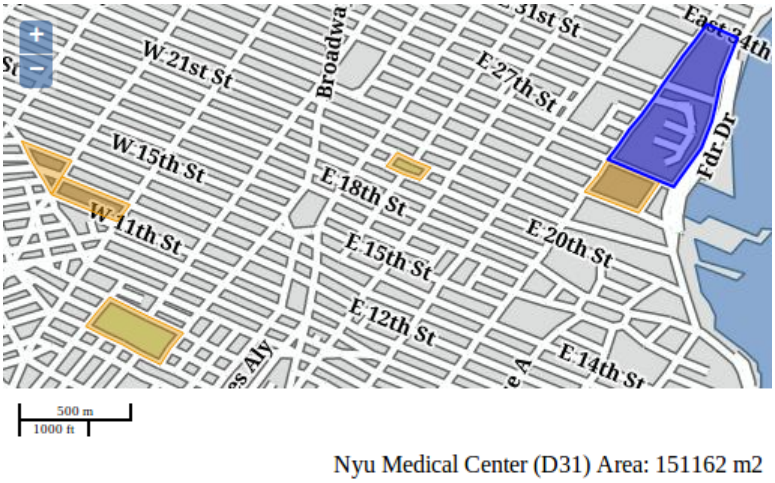


Figure 3.3: Displaying building information on feature selection.

Hint: Take some time and use the Chrome developer tools to put a breakpoint on this function and inspect the `event` object that wraps the feature selected.

CHAPTER 4

Vector Layers

Vector layers are used to render map data in the browser. In contrast with raster layers where map images are rendered on the server and requested by the client, with a vector layer raw feature data is requested from the server and rendered on the client.

What this module covers

This module covers vector layers in detail. In this module you will draw new vector features, edit existing features, persist feature modifications, and get an introduction to feature styling.

4.1 Working with Vector Layers

The base `OpenLayers.Layer.Vector` constructor provides a fairly flexible layer type. By default, when you create a new vector layer, no assumptions are made about where the features for the layer will come from. In addition, a very basic style is applied when rendering those features. Customizing the rendering style is addressed in an *upcoming section*. This section introduces the basics of vector data *formats*, the *protocols* used to read and write feature data, and various *strategies* for engaging with those protocols.

In dealing with vector layers and features, it is somewhat useful to consider a postal analogy. When writing a letter, you have to know some of the rules imposed by the postal service, such as how addresses are formatted or what an envelope can contain. You also have to know something about your recipient: primarily what language they speak. Finally, you have to make a decision about when to go to the post office to send your letter. Having this analogy in mind may help in understanding the concepts below.

4.1.1 OpenLayers.Format

The `OpenLayers.Format` classes in OpenLayers are responsible for parsing data from the server representing vector features. Following the postal analogy, the format you choose is analogous to the language in which you write your letter. The format turns raw feature data into `OpenLayers.Feature.Vector` objects. Typically, format is also responsible for reversing this operation.

Consider the two blocks of data below. Both represent the same `OpenLayers.Feature.Vector` object (a point in Barcelona, Spain). The first is serialized as GeoJSON (using the `OpenLayers.Format.GeoJSON` parser). The second is serialized as GML (OGC Geography Markup Language) (using the `OpenLayers.Format.GML.v3` parser).

GeoJSON Example

```
{
    "type": "Feature",
    "id": "OpenLayers.Feature.Vector_107",
    "properties": {},
    "geometry": {
        "type": "Point",
        "coordinates": [-104.98, 39.76]
    }
}
```

GML Example

```
<?xml version="1.0" encoding="utf-16"?>
<gml:featureMember
    xsi:schemaLocation="http://www.opengis.net/gml http://schemas.opengis.net/gml/3.1.1/profiles/
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <feature:feature fid="OpenLayers.Feature.Vector_107" xmlns:feature="http://example.com/feature
        <feature:geometry>
            <gml:Point>
                <gml:pos>-104.98, 39.76</gml:pos>
            </gml:Point>
        </feature:geometry>
    </feature:feature>
</gml:featureMember>
```

See the vector formats example for a demonstration of translation between a few OpenLayers formats.

4.1.2 OpenLayers.Protocol

The `OpenLayers.Protocol` classes refer to specific communication protocols for reading and writing vector data. A protocol instance may have a reference to a specific `OpenLayers.Format`. So, you might be working with a service that communicates via HTTP and deals in GeoJSON features. Or, you might be working with a service that implements WFS and deals in GML. In these cases you would construct an `OpenLayers.Protocol.HTTP` with a reference to an `OpenLayers.Format.GeoJSON` object or an `OpenLayers.Protocol.WFS` with a reference to an `OpenLayers.Format.GML` object.

Back on the postal analogy, the protocol is akin to the rules about how an envelope must be addressed. Ideally, a protocol doesn't specify anything about the format of the content being delivered (the post office doesn't care about the language used in a letter; ideally, they only need to read the envelope).

Neither protocols nor formats are explicitly tied to an `OpenLayers.Layer.Vector` instance. The layer provides a view of the data for the user, and the protocol shouldn't have to bother itself about that view.

4.1.3 OpenLayers.Strategy

Loosely speaking, the `OpenLayers.Strategy` classes tie together the layer and the protocol. Strategies deal with *when* to make requests for data (or *when* to send modifications). Strategies can also determine *how* to prepare features before they end up in a layer.

The `OpenLayers.Strategy.BBOX` strategy says "request new features whenever the map bounds are outside the bounds of the previously requested set of features."

The `OpenLayers.Strategy.Cluster` strategy says "before passing any new features to the layer, clump them together in clusters based on proximity to other features."

In creating a vector layer, you choose the mix: one protocol (typically) with a reference to one format, and any number of strategies. And, all of this is optional. You can very well create a vector layer *without*

protocol or strategies and manually make requests for features, parse those features, and add them to the layer.

Having dispensed with the basics of formats, protocols, and strategies, we're ready to start *creating new features*.

4.2 Creating New Features

OpenLayers provides controls for drawing and modifying vector features. The `OpenLayers.Control.DrawFeature` control can be used in conjunction with an `OpenLayers.Handler.Point`, an `OpenLayers.Handler.Path`, or an `OpenLayers.Handler.Polygon` instance to draw points, lines, polygons, and their multi-part counterparts. The `OpenLayers.Control.ModifyFeature` control can be used to allow modification of geometries for existing features.

In this section, we'll add a control to the map for drawing new polygon features. As with the other examples in this workshop, this is not supposed to be a complete working application–as it does not allow editing of attributes or saving of changes. We'll take a look at persistence in the *next section*.

Tasks

1. We'll start with a working example that displays building footprints in a vector layer over a base layer. Open your text editor and save the following as `map.html` in the root of your workshop directory:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
            #scaleline-id {
                margin: 10px;
                font-size: xx-small;
            }
            #output-id {
                margin: 10px 220px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>

        <script>
            var nyc = new OpenLayers.Bounds(
                -74.032, 40.685,
                -73.902, 40.876
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:4326"),
                maxExtent: nyc,
                restrictedExtent: nyc,
                maxResolution: 0.0005,
                numZoomLevels: 5
```

```
            });

            var base = new OpenLayers.Layer.WMS(
                "New York City",
                "/geoserver/wms",
                {layers: "tiger-ny"}
            );
            map.addLayer(base);

            var center = new OpenLayers.LonLat( -73.987, 40.737);
            map.setCenter(center,3);

            var landmarks = new OpenLayers.Layer.Vector("NY Landmarks", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    version: "1.1.0",
                    url: "/geoserver/wfs",
                    featureType: "poly_landmarks",
                    featureNS: "http://www.census.gov",
                    srsName: "EPSG:4326"
                })
            });
            map.addLayer(landmarks);

        </script>
    </body>
</html>
```

2. Open this `map.html` example in your browser to confirm that landmarks are displayed over the base layer: http://localhost:8080/ol_workshop/map.html

3. To this example, we'll be adding a control to draw features. In order that users can also navigate with the mouse, we don't want this control to be active all the time. We need to add some elements to the page that will allow for control activation and deactivation. In the `<body>` of your document, add the following markup. (Placing it right after the map viewport element `<div id="map-id"></div>` makes sense.):

```
<input id="toggle-id" type="checkbox">
<label for="toggle-id">draw</label>
```

4. Now we'll create an `OpenLayers.Control.DrawFeature` control to add features to the landmarks layer. We construct this layer with an `OpenLayers.Handler.Polygon` to allow drawing of polygons. In your map initialization code, add the following somewhere after the creation of the `landmarks` layer:

```
var draw = new OpenLayers.Control.DrawFeature(
    landmarks, OpenLayers.Handler.Polygon
);
map.addControl(draw);
```

5. Finally, we'll add behavior to the `<input>` element in order to activate and deactivate the draw control when the user clicks the checkbox. We'll also call the `toggle` function when the page loads to synchronize the checkbox and control states. Add the following to your map initialization code:

```
function toggle() {
    if (document.getElementById("toggle-id").checked) {
        draw.activate();
    } else {
        draw.deactivate();
    }
}
document.getElementById("toggle-id").onclick = toggle;
toggle();
```

6. Save your changes and reload `map.html` in your browser: http://localhost:8080/ol_workshop/map.html



Figure 4.1: A control for adding features to a vector layer.

4.3 Persisting Features

Persistence of vector feature data is the job of an `OpenLayers.Protocol`. The WFS specification defines a protocol for reading and writing feature data. In this section, we'll look at an example that uses an `OpenLayers.Protocol.WFS` instance with a vector layer.

A full-fledged editing application involves more user interaction (and GUI elements) than is practical to demonstrate in a short example. However, we can add an `OpenLayers.Control.Panel` to a map that accomplishes a few of the basic editing tasks.

Tasks

1. Open your text editor and paste in the text from the start of the *previous section*. Save this as `map.html`.

2. OpenLayers doesn't provide controls for deleting or saving features. The `extras` folder in this workshop includes code for those controls bundled together in a control panel. These controls are specific to editing a vector layer with multipolygon geometries, so they will work with our landmarks example. In the `<head>` of your `map.html` document, **after** the OpenLayers script tag, insert the following to pull in the required code and stylesheet for the controls:

```
<link rel="stylesheet" href="extras/editing-panel.css" type="text/css">
<script src="extras/DeleteFeature.js"></script>
<script src="extras/EditingPanel.js"></script>
```

3. Now we'll give the `landmarks` layer an `OpenLayers.Strategy.Save`. This strategy is designed to trigger commits on the protocol and deal with the results. The `landmarks` layer currently has a single strategy. Modify the layer creation code to include another:

```
var landmarks = new OpenLayers.Layer.Vector("NY Landmarks", {
    strategies: [
        new OpenLayers.Strategy.BBOX(),
        new OpenLayers.Strategy.Save()
        ],
    protocol: new OpenLayers.Protocol.WFS({
        version: "1.1.0",
        url: "/geoserver/wfs",
        featureType: "poly_landmarks",
        featureNS: "http://www.census.gov",
        srsName: "EPSG:4326"
```

```
    })
});
```

4. Finally, we'll create the editing panel and add it to the map. Somewhere in your map initialization code after creating the `landmarks` layer, insert the following:

```
var panel = new EditingPanel(landmarks);
map.addControl(panel);
```

5. Now save your changes and load `map.html` in your browser: http://localhost:8080/ol_workshop/map.html

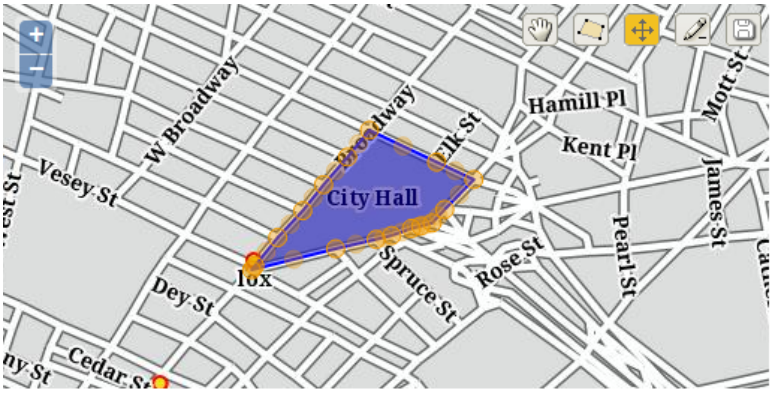


Figure 4.2: Modifying a building footprint.

4.4 Understanding Style

When styling HTML elements, you might use CSS like the following:

```
.someClass {
    background-color: blue;
    border-width: 1px;
    border-color: olive;
}
```

The `.someClass` text is a selector (in this case it selects all elements that include the class name `"someClass"`) and the block that follows is a group of named properties and values, otherwise known as style declarations.

4.4.1 OpenLayers.Filter

When styling features in OpenLayers, your selectors are `OpenLayers.Filter` objects. Assuming you want to apply a style to all features that have an attribute named `class` with a value of `"someClass"`, you would start with a filter like the following:

```
new OpenLayers.Filter.Comparison({
    type: OpenLayers.Filter.Comparison.EQUAL_TO,
    property: "class",
    value: "someClass"
})
```

4.4.2 Symbolizers

The equivalent of a declaration block in CSS is a *symbolizer* in OpenLayers (these are typically object literals). To paint features with a blue background and a 1 pixel wide olive stroke, you would use a symbolizer like the following:

```
{
    fillColor: "blue",
    strokeWidth: 1,
    strokeColor: "olive"
}
```

4.4.3 OpenLayers.Rule

To combine a filter with a symbolizer, we use an `OpenLayers.Rule` object. As such, a rule that says *"paint all features with class equal to 'someClass' using a 1px olive stroke and blue fill"* would be created as follows:

```
new OpenLayers.Rule({
    filter: new OpenLayers.Filter.Comparison({
        type: OpenLayers.Filter.Comparison.EQUAL_TO,
        property: "class",
        value: "someClass"
    }),
    symbolizer: {
        fillColor: "blue",
        strokeWidth: 1,
        strokeColor: "olive"
    }
})
```

4.4.4 OpenLayers.Style

As in CSS page, where you may have many rules– selectors and associated declaration blocks–you are likely to have more than one rule for styling the features of a given map layer. You group `OpenLayers.Rule` objects together in an `OpenLayers.Style` object. A style object is typically constructed with a base symbolizer. When a feature is rendered, the base symbolizer is extended with symbolizers from all rules that apply to the feature.

So, if you want all features to be colored red except for those that have a `class` attribute with the value of `"someClass"` (and you want those features colored blue with an 1px olive stroke), you would create a style that looked like the following:

```
var myStyle = new OpenLayers.Style({
    // this is the base symbolizer
    fillColor: "red"
}, {
    rules: [
        new OpenLayers.Rule({
            filter: new OpenLayers.Filter.Comparison({
                type: OpenLayers.Filter.Comparison.EQUAL_TO,
                property: "class",
                value: "someClass"
            }),
            symbolizer: {
                fillColor: "blue",
                strokeWidth: 1,
                strokeColor: "olive"
            }
        }),
        new OpenLayers.Rule({elseFilter: true})
    ]
});
```

Note: If you don't include any rules in a style, *all* of the features in a layer will be rendered with the base symbolizer (first argument to the `OpenLayers.Style` constructor). If you include *any* rules in your style, only features that pass at least one of the rule constraints will be rendered. The `elseFilter`

property of a rule let's you provide a rule that applies to all features that haven't met any of the constraints of your other rules.

4.4.5 OpenLayers.StyleMap

CSS allows for pseudo-classes on selectors. These basically limit the application of style declarations based on contexts such as mouse position, neighboring elements, or browser history, that are not easily represented in the selector. In OpenLayers, a somewhat similar concept is one of "render intent." Without defining the full set of render intents that you can use, the library allows for sets of rules to apply only under specific contexts.

So, the `active` pseudo-class in CSS limits the selector to the currently selected element (e.g. `a:active`). In the same way, the `"select"` render intent applies to currently selected features. The mapping of render intents to groups of rules is called an `OpenLayers.StyleMap`.

Following on with the above examples, if you wanted all features to be painted fuchsia when selected, and otherwise you wanted the `myStyle` defined above to be applied, you would create an `OpenLayers.StyleMap` like the following:

```
var styleMap = new OpenLayers.StyleMap({
    "default": myStyle,
    "select": new OpenLayers.Style({
        fillColor: "fuchsia"
    })
});
```

To determine how features in a vector layer are styled, you need to construct the layer with an `OpenLayers.StyleMap`.

With the basics of styling under your belt, it's time to move on to *styling vector layers*.

4.5 Styling Vector Layers

1. We'll start with a working example that displays building footprints in a vector layer over a base layer. Open your text editor and save the following as `map.html` in the root of your workshop directory:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>

    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>

        <script>
            var nyc = new OpenLayers.Bounds(
                -74.032, 40.685,
                -73.902, 40.876
            );
```

```
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:4326"),
                maxExtent: nyc,
                restrictedExtent: nyc,
                maxResolution: 0.0005,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "New York City",
                "/geoserver/wms",
                {layers: "tiger-ny"}
            );
            map.addLayer(base);

            var center = new OpenLayers.LonLat( -73.987, 40.737);
            map.setCenter(center,3);
            map.addControl(new OpenLayers.Control.LayerSwitcher());

            var landmarks = new OpenLayers.Layer.Vector("NY Landmarks", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    version: "1.1.0",
                    url: "/geoserver/wfs",
                    featureType: "poly_landmarks",
                    featureNS: "http://www.census.gov",
                    srsName: "EPSG:4326"
                })
            });
            map.addLayer(landmarks);

        </script>
    </body>
</html>
```

2. Open this `map.html` file in your browser to see orange landmarks over the base layer: http://localhost:8080/ol_workshop/map.html. You can use the layer switcher control we added to change the visibility of the vector layer.

3. With a basic understanding of *styling in OpenLayers*, we can create an `OpenLayers.StyleMap` that displays landmarks in different colors based on the `CFCC` code. In your map initialization code, replace the constructor for the `landmarks` layer with the following:

```
var landmarks = new OpenLayers.Layer.Vector("NY Landmarks", {
    strategies: [new OpenLayers.Strategy.BBOX()],
    protocol: new OpenLayers.Protocol.WFS({
        version: "1.1.0",
        url: "/geoserver/wfs",
        featureType: "poly_landmarks",
        featureNS: "http://www.census.gov",
        srsName: "EPSG:4326"
    }),
    styleMap: new OpenLayers.StyleMap({
        "default": new OpenLayers.Style({
            strokeColor: "white",
            strokeWidth: 1
        }, {
            rules: [
                new OpenLayers.Rule({
                     filter: new OpenLayers.Filter.Logical({
                        type: OpenLayers.Filter.Logical.OR,
                        filters: [
                            new OpenLayers.Filter.Comparison({
                                type: OpenLayers.Filter.Comparison.EQUAL_TO,
```

```
                                property: "CFCC", value: "D82"
                            }),
                            new OpenLayers.Filter.Comparison({
                                type: OpenLayers.Filter.Comparison.EQUAL_TO,
                                property: "CFCC", value: "D83"
                            }),
                            new OpenLayers.Filter.Comparison({
                                type: OpenLayers.Filter.Comparison.EQUAL_TO,
                                property: "CFCC", value: "D84"
                            }),
                            new OpenLayers.Filter.Comparison({
                                type: OpenLayers.Filter.Comparison.EQUAL_TO,
                                property: "CFCC", value: "D85"
                            })
                        ]
                    }),
                    symbolizer: {
                        fillColor: "#B4DFB4",
                        strokeColor: "#88B588",
                        strokeWidth: 2
                    }
                }),
                new OpenLayers.Rule({
                    elseFilter: true,
                    symbolizer: {
                        fillColor: "navy"
                    }
                })
            ]
        })
    })
});
```

4. See how an `OpenLayers.Filter.Logical.OR` filter groups several filters to allow a rule to match different conditions. That is, style all the features where the field `CFCC` has the values ``D80`` to `D85`.
5. Save your changes and open `map.html` in your browser: http://localhost:8080/ol_workshop/map.html

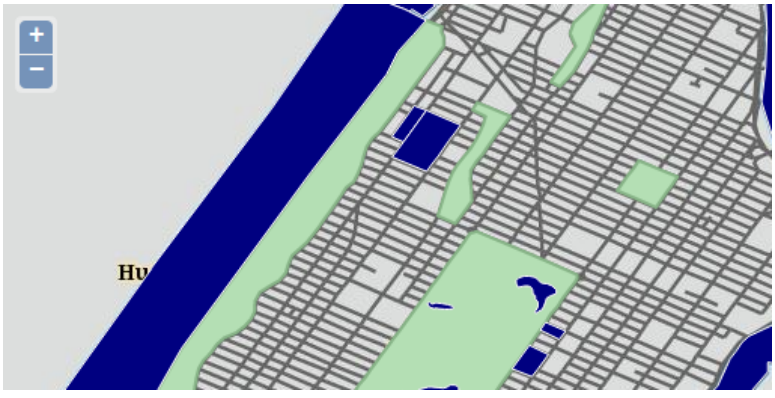


Figure 4.3: Landmarks related with green areas.

Tasks

1. Go to the GeoServer web interface and review the SLD style applied to the `poly_landmarks`. You'll see the first rule is similar to the style applied on this exercise.

2. Try to reproduce the rest of the rules of the layer, so you have a similar vector representation of this WMS layer.
3. Change the base layer to just load the `giant_polygon` WMS layer and try to render the roads as vectors using filter by scale, and loading on top the labels as a WMS layer. You will have to create a new SLD style to just render the labels. Pay attention to the image below layers to see how to obtain that effect.

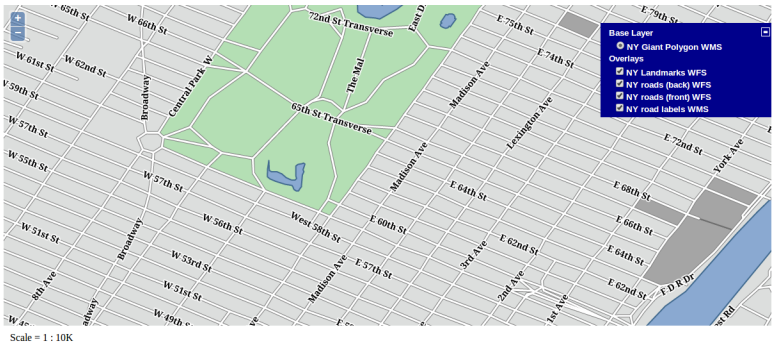


Figure 4.4: Rendering landmarks and roads as vectors

CHAPTER 5

For more information

The following is a list of external sites related to OpenLayers.

5.1 OpenLayers Home

Visit the OpenLayers home page at http://openlayers.org.

5.2 OpenLayers Documentation

Full documentation for OpenLayers is available at http://docs.openlayers.org, a great list of examples are available at http://openlayers.org/dev/examples/ and the API documentation is at http://dev.openlayers.org/apidocs/.

5.3 Mailing list

OpenLayers has an active users mailing list, which you can subscribe to at http://lists.osgeo.org/mailman/listinfo/openlayers-users. If you're a developer, you can subscribe to the developer list at http://lists.osgeo.org/mailman/listinfo/openlayers-dev.

5.4 Bug tracking

OpenLayers is maintained on a GitHub repo, issues for OpenLayers 2 branch should be reported at https://github.com/openlayers/openlayers/issues.

5.5 IRC

Join a live discussion at #openlayers, on irc.freenode.net.

5.6 OpenGeo

OpenGeo helps to develop OpenLayers and funds development through its OpenGeo Suite. Learn more at http://opengeo.org.

5.7 Prodevelop

Prodevelop does consultancy and integrates free software for geomatics like OpenLayers. It's also an official OpenGeo partner in Spain. Learn more at http://wwww.prodevelop.es/en.