OpenLayers Workshop Documentation

Release 2.8

The Open Planning Project

September 29, 2009

CONTENTS

Welcome to the **FOSS4G 2009 OpenLayers Workshop**. This workshop is designed to give you a comprehensive overview of OpenLayers as a web mapping solution.

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.

CHAPTER

ONE

FUNDAMENTAL CONCEPTS

1.1 OpenLayers Basics

OpenLayers a rapidly-developing library for building mapping applications in a browser. 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.

1.1.1 What this module covers

In this module you will:

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*.

Working Example

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

```
<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",
                "/geoserver/wms",
                {layers: "topp: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/ol_workshop/map.html

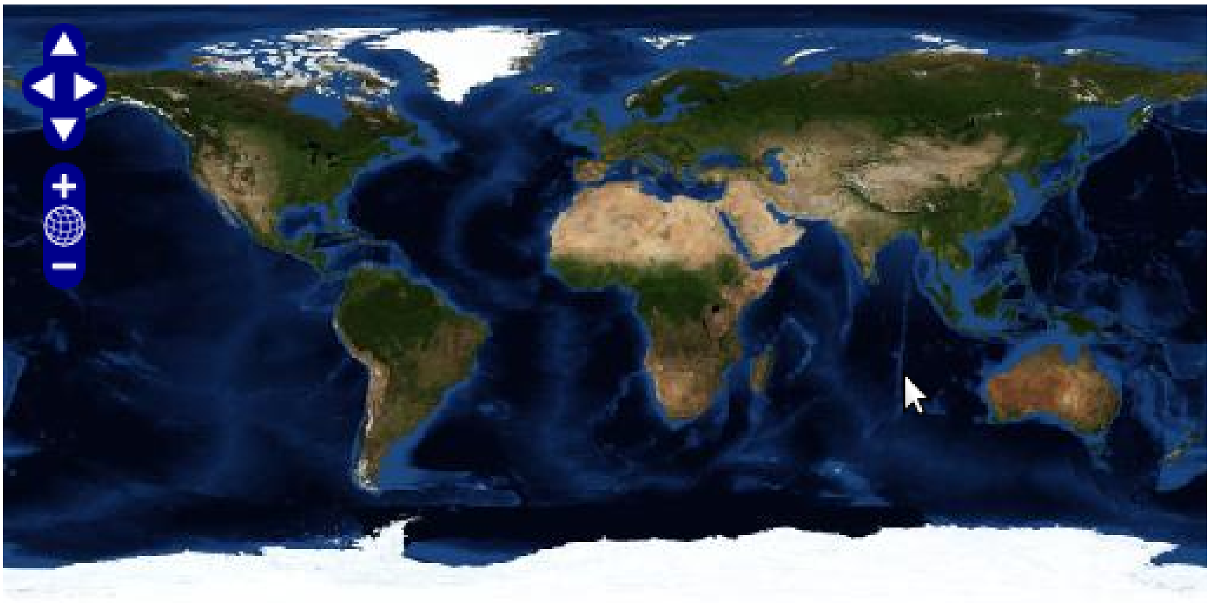


Figure 1.1: A working map of displaying imagery of the world.

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

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.

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.

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.

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",
        "/geoserver/wms",
        {layers: "topp: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 bee 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",
    "/geoserver/wms",
    {layers: "topp: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.

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.

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/

Browse the Documentation

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

- http://docs.openlayers.org/

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

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://openlayers.org/mailman/listinfo/users

- Developers list http://openlayers.org/mailman/listinfo/dev

1.2 Working With Layers

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

1.2.1 What this module covers

This module covers the basics of working with map layers. In this module you will:

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.

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:

```
<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",
                "/geoserver/wms",
                {layers: "topp: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/ol_workshop/map.html

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",
    "/geoserver/wms",
    {layers: "topp: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, `"/geoserver/wms"`, is the string URL for a Web Map Service. In our case, we have a WMS running locally so we're using a relative URL.

The third argument, `{layers: "topp: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 `"topp:bluemarble"`.

Tasks

1. This same WMS offers a layer named `"world"`. Change the value of the `layers` param from `"topp:bluemarble"` to `"world"`.
2. 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/gif"`).

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

   ```
   var imagery = new OpenLayers.Layer.WMS(
       "Global Imagery",
       "/geoserver/wms",
       {layers: "world", format: "image/gif"}
   );
   ```

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

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

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.

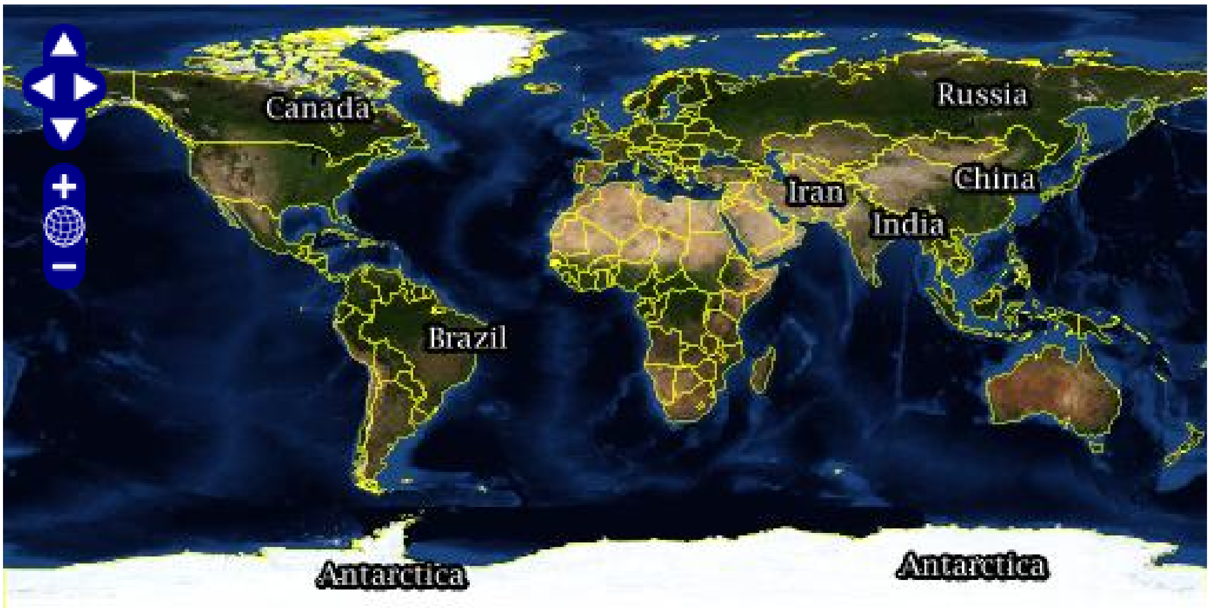


Figure 1.2: A map displaying the `"world"` layer as `"image/gif"`.

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.

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 geographic = new OpenLayers.Projection("EPSG:4326");
    var mercator = new OpenLayers.Projection("EPSG:900913");

    var world = new OpenLayers.Bounds(-180, -89, 180, 89).transform(
        geographic, mercator
    );
    var sydney = new OpenLayers.LonLat(151, -33.9).transform(
        geographic, mercator
    );

    var options = {
        projection: mercator,
        units: "m",
        maxExtent: world
    };
    var map = new OpenLayers.Map("map-id", options);

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

    map.setCenter(sydney, 9);
</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;
        right: 5px;
    }
</style>
```

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

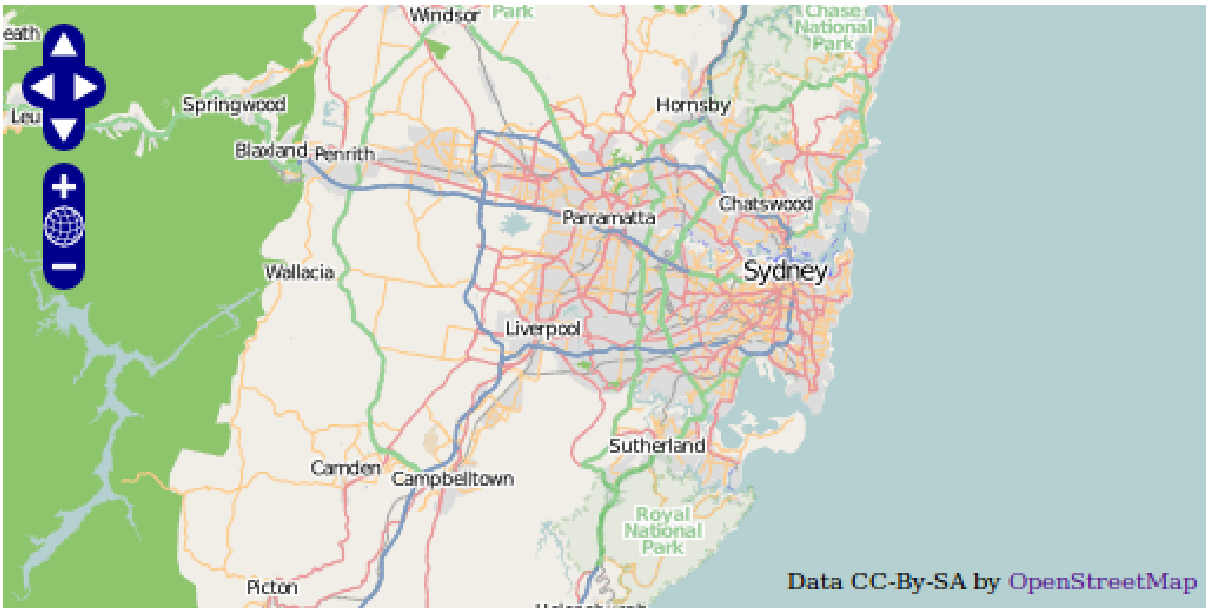


Figure 1.3: A map with an OpenStreetMap view of Sydney.

A Closer Look

Projections Review the first 2 lines of the initialization script:

```
var geographic = new OpenLayers.Projection("EPSG:4326");
var mercator = new OpenLayers.Projection("EPSG:900913");
```

Geospatial data can come in any number of coordinate reference systems. One data set might use geographic coordinates (latitude and longitude) 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. We represent this with an `OpenLayers.Projection` object. You create a new projection object by using a string that represents the coordinate reference system (`"EPSG:4326"` and `"EPSG:900913"` above).

Locations Transformed

```
var world = new OpenLayers.Bounds(-180, -89, 180, 89).transform(
    geographic, mercator
);
var sydney = new OpenLayers.LonLat(151, -33.9).transform(
    geographic, mercator
);
```

The OpenStreetMap tiles that we will be using are in a Mercator projection. Because of this, we need to set things like the maximum extent and initial center using Mercator coordinates. Since it is relatively easy to find out the coordinates for a place like Sydney, Australia in geographic coordinates, we use the `transform` method to turn geographic coordinates into Mercator coordinates.

Note: You may notice that the world bounds created above do not *really* cover the entire world. For now, you'll have to accept that this particular transform doesn't behave well at the poles so we adjust the bounds to compensate.

Custom Map Options

```
var options = {
    projection: mercator,
    units: "m",
    maxExtent: world
};
var map = new OpenLayers.Map("map-id", options);
```

In the *previous example* we used the default options for our map. In this example, we set some custom map options. First, we set the map `projection` property to the Mercator projection we created above. Next we set the map `units` to `"m"` for meters, and finally we set the `maxExtent` to the world (again in Mercator coordinates).

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(sydney, 9);
```

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

Style

```
.olControlAttribution {
    font-size: 10px;
    bottom: 5px;
    right: 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 “CC-By-SA” line at the bottom right of the map).

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

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 Virtual Earth, and we’ll toss in a layer switcher so you can decide which layers you want visible.

Bing!

Let’s add a Virutal Earth (a.k.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.VirtualEarth("VE Streets", {
       sphericalMercator: true
   });
   map.addLayer(bing);
   ```

2. Since we can’t access these tiles directly (and instead need to go through the JavaScript API that Virtual Earth provides), we need to include an additional script tag in our page. Somewhere in the `<head>` of your document, add the following:

   ```
   <script src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.2&mkt=en-us"></sc
   ```

3. 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());
   ```

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

5. Open the Layer Switcher at the upper right-hand corner of the map view, and select “VE Streets”.

Complete Working Example Your revised `map.html` file should look something like this:

```
<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;
```

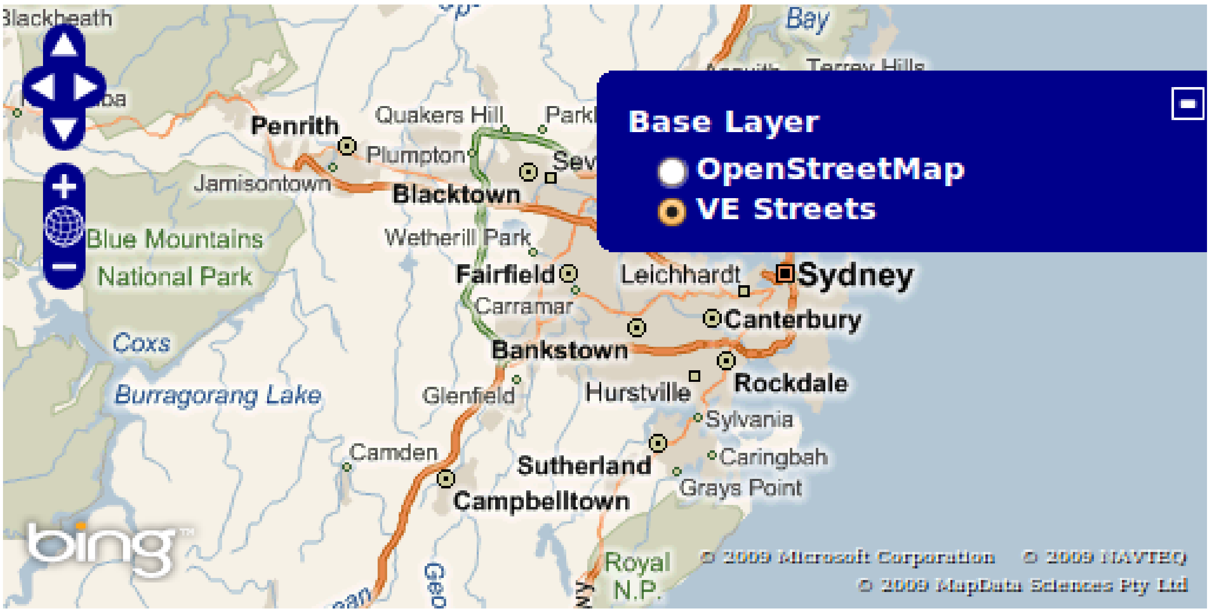


Figure 1.4: A map with a bing layer and OpenStreetMap tiles.

```
        }
        .olControlAttribution {
            font-size: 10px;
            bottom: 5px;
            right: 5px;
        }
    </style>
    <script src="http://ecn.dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.2&mkt=en-us">
    <script src="openlayers/lib/OpenLayers.js"></script>
</head>
<body>
    <h1>My Map</h1>
    <div id="map-id"></div>
    <script>
        var geographic = new OpenLayers.Projection("EPSG:4326");
        var mercator = new OpenLayers.Projection("EPSG:900913");

        var world = new OpenLayers.Bounds(-180, -89, 180, 89).transform(
            geographic, mercator
        );
        var sydney = new OpenLayers.LonLat(151, -33.9).transform(
            geographic, mercator
        );

        var options = {
            projection: mercator,
            units: "m",
            maxExtent: world
        };
        var map = new OpenLayers.Map("map-id", options);

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

        var bing = new OpenLayers.Layer.VirtualEarth("VE Streets", {
            sphericalMercator: true
        });
        map.addLayer(bing);
```

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

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

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

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).

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 `maps.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/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 GeoJSON:

   ```
   var presenters = new OpenLayers.Layer.Vector("Presenters", {
       strategies: [new OpenLayers.Strategy.Fixed()],
       protocol: new OpenLayers.Protocol.HTTP({
           url: "data/layers/f4g_presenters.json",
           format: new OpenLayers.Format.GeoJSON()
       })
   });
   map.addLayer(presenters);
   ```

   Note: GeoJSON is a format for encoding a variety of geographic data structures. A GeoJSON object may represent a geometry, a feature, or a collection of features. GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.

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

```
var presenters = new OpenLayers.Layer.Vector("Presenters", {
    strategies: [new OpenLayers.Strategy.Fixed()],
    protocol: new OpenLayers.Protocol.HTTP({
        url: "data/layers/f4g_presenters.json",
```

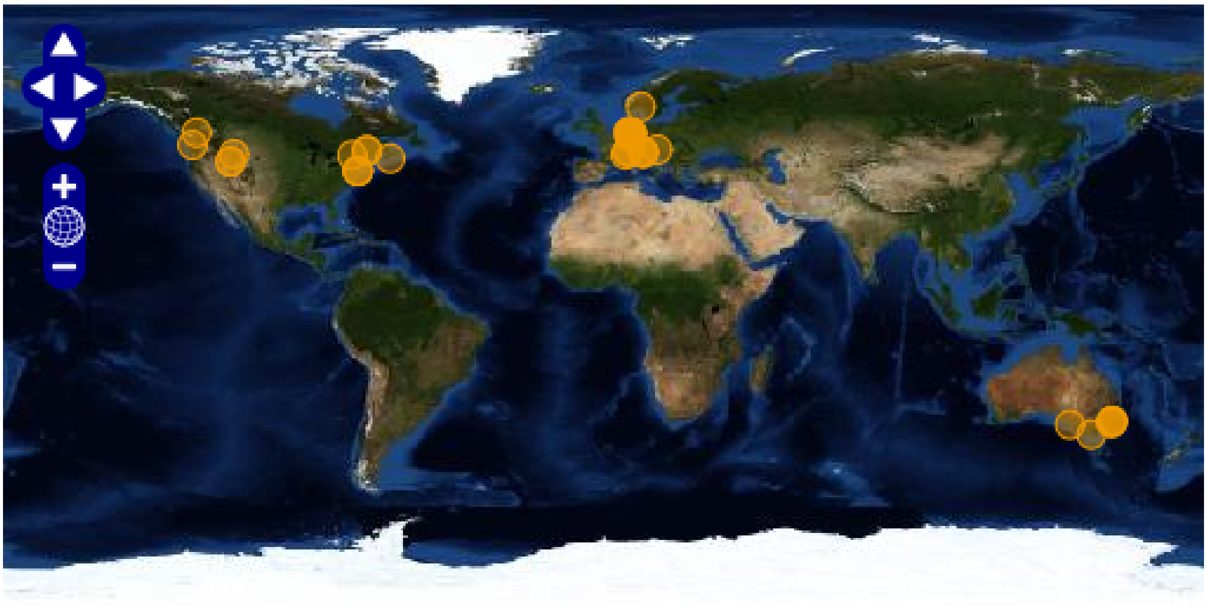


Figure 1.5: World map with orange circles representing conference presenters.

```
        format: new OpenLayers.Format.GeoJSON()
    })
});
```

The layer is given the name `"Presenters"` 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 GeoJSON.

Bonus Tasks

1. 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 `name`, `description`, and `location` properties. Add an `OpenLayers.Control.SelectFeature` control to your map, listen for the `featureselected` event on the vector layer, and display presenter information below the map viewport.
2. Continuing on with the above task, use one of the `OpenLayers.Popup` classes to display feature information in a popup on the map. The popup should open on feature selection and close when a feature is unselected.

1.3 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)

1.3.1 What this module covers

This module covers the basics of using controls in OpenLayers. In this module you will:

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:

```
<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 medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(base);

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

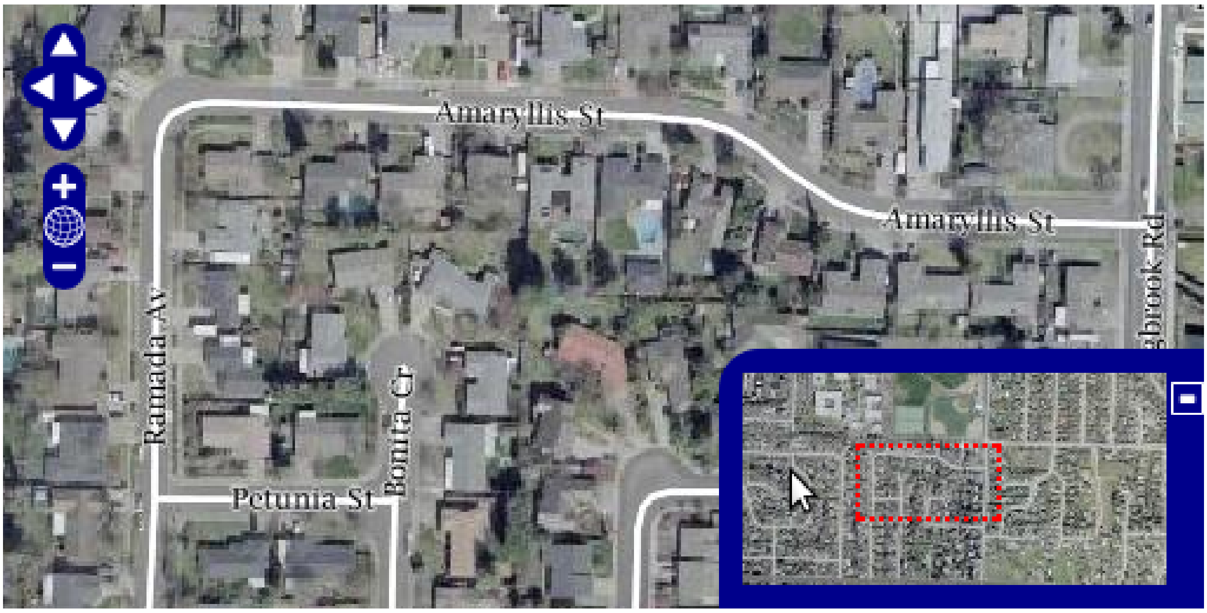
2. Open the working map in your web browser to confirm things look good: http://localhost/ol_workshop/map.html.

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:2270"),
        units: "ft",
        maxExtent: medford,
        restrictedExtent: medford,
        maxResolution: 22,
        numZoomLevels: 5
    }
});
map.addControl(overview);
```

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

Discussion

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

```
var medford = new OpenLayers.Bounds(
    4284890, 254385,
    4288865, 258380
);
var map = new OpenLayers.Map("map-id", {
    projection: new OpenLayers.Projection("EPSG:2270"),
    units: "ft",
    maxExtent: medford,
    restrictedExtent: medford,
    maxResolution: 2.5,
    numZoomLevels: 5
});
```

First, we construct the map in a custom projection. The OpenLayers default is to contruct maps in a geographic projection, with units in decimal degrees. Because of the nature of the Medford dataset, we define a `projection` more appropriate for this area (i.e., Oregon State Plane South). This change in projection also requires setting the map `units` to feet ("ft").

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 `medford` 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 2.5 means 2.5 feet per pixel (since we set map units to feet). When users are zoomed all the way out, they will be seeing 2.5 feet per pixel. 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:2270"),
        units: "ft",
        maxExtent: medford,
        restrictedExtent: medford,
        maxResolution: 22,
        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`, `units`, `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).

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

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.

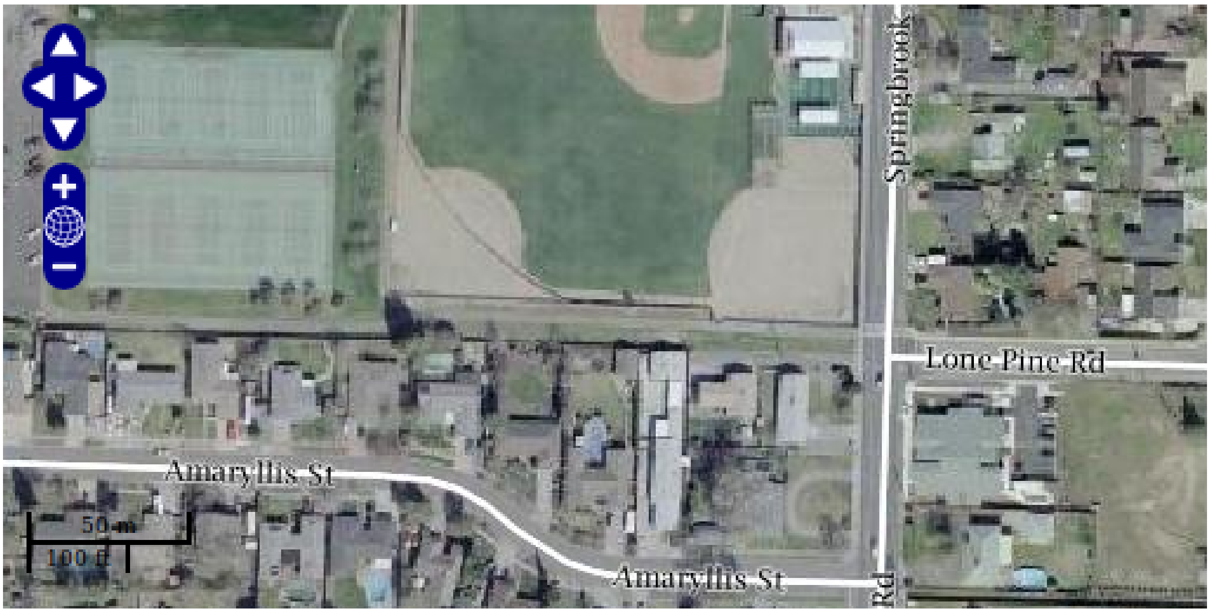
Creating a ScaleLine Control

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/ol_workshop/map.html



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

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 and padding to the scale bar with something like the following:

```
.olControlScaleLine {
    background: white;
    padding: 10px;
}
```

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/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/ol_workshop/map.html

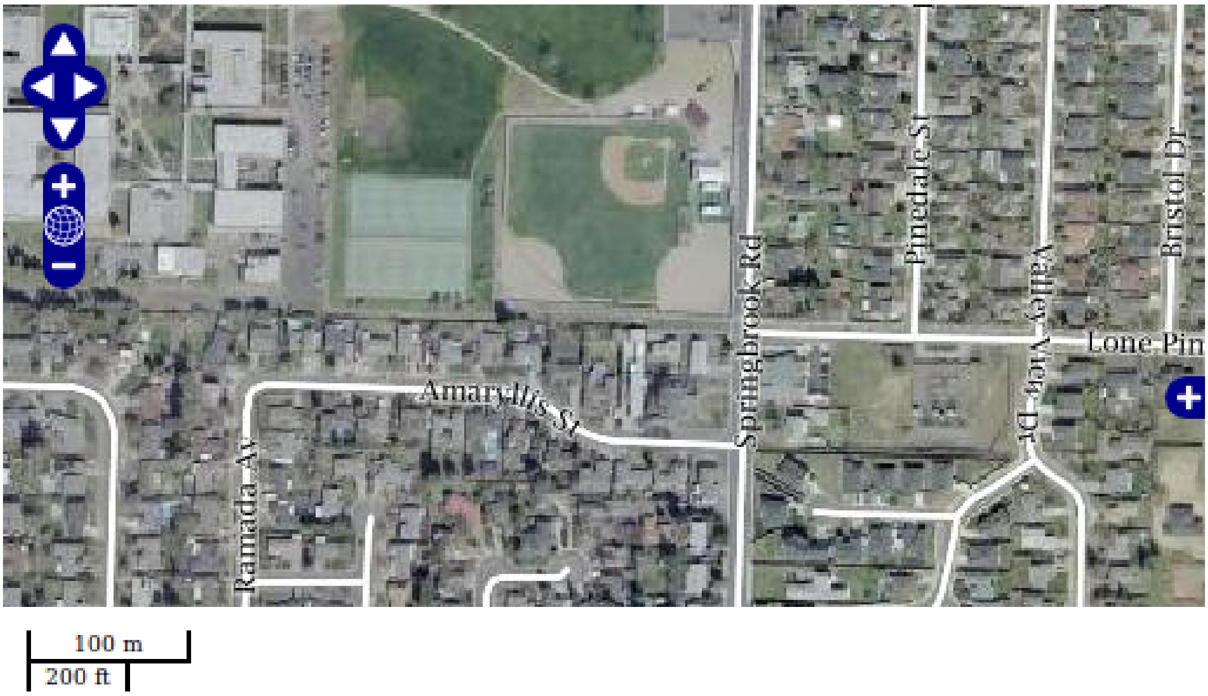


Figure 1.6: A custom styled scale bar outside the map viewport.

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.

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:

```
<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 medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(base);

            var overview = new OpenLayers.Control.OverviewMap({
                mapOptions: {
                    projection: new OpenLayers.Projection("EPSG:2270"),
                    units: "ft",
                    maxExtent: medford,
                    restrictedExtent: medford,
                    maxResolution: 22,
                    numZoomLevels: 5
                }
            });
            map.addControl(overview);

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

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

2. Next add a vector layer that requests the building footprints for Medford, Oregon. Because this data will be rendered client-side (i.e., by GeoServer), 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 buildings = new OpenLayers.Layer.Vector("Buildings", {
    strategies: [new OpenLayers.Strategy.BBOX()],
    protocol: new OpenLayers.Protocol.WFS({
        url: "/geoserver/wfs",
        featureType: "medford_buildings",
        featureNS: "http://medford"
    })
});
map.addLayer(buildings);
```

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

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

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

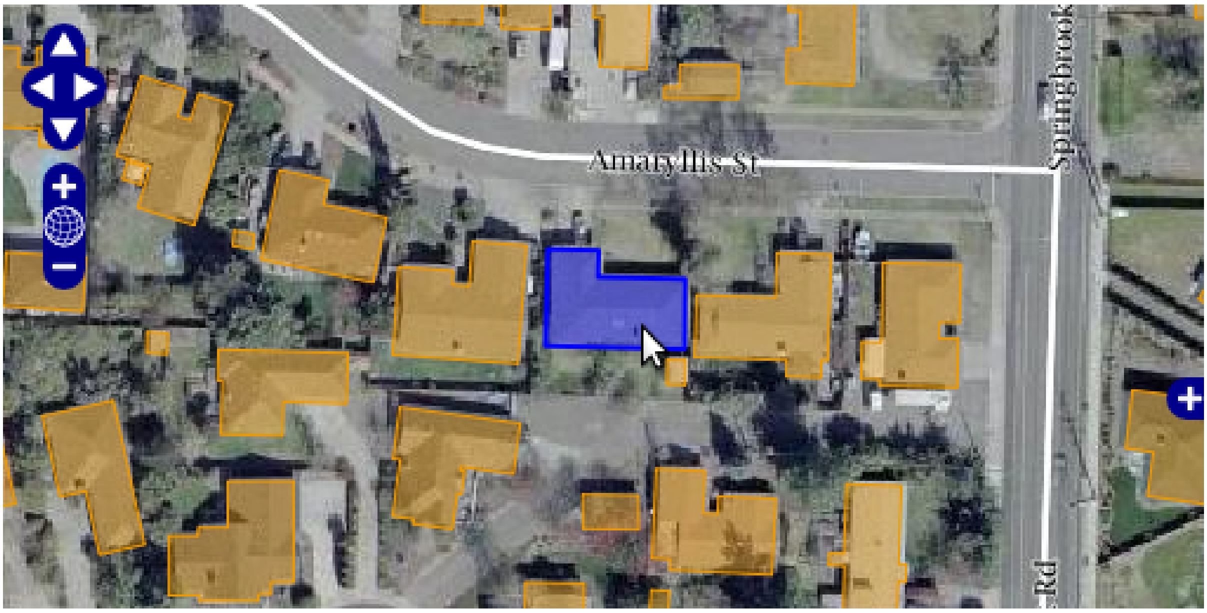


Figure 1.7: Using a control to select features from a vector layer.

Displaying Building Information on Feature Selection

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. (Placing this `output-id` element right after the map viewport element `<div id="map-id"></div>` makes sense.)

   ```
   <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 250px;
   }
   ```

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 `buildings` layer:

   ```
   buildings.events.on({
       featureselected: function(event) {
           var feature = event.feature;
           var area = feature.geometry.getArea();
           var id = feature.attributes.key;
           var output = "Building: " + id + " Area: " + area.toFixed(2);
           document.getElementById("output-id").innerHTML = output;
       }
   });
   ```

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

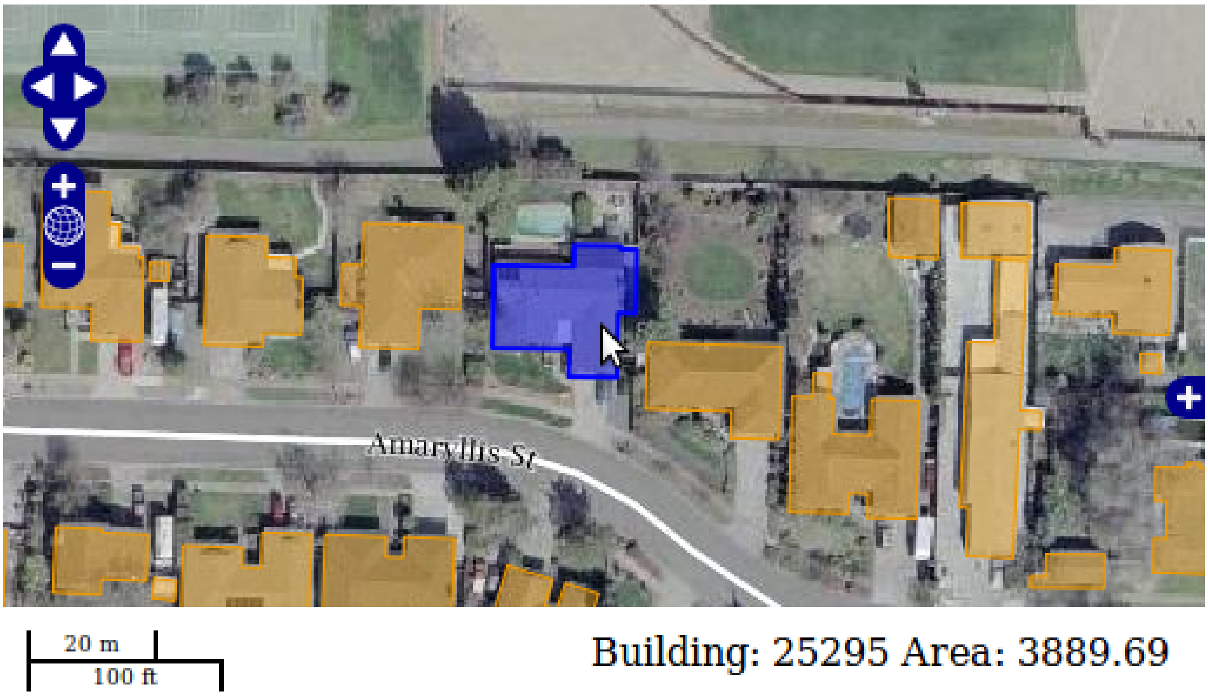


Figure 1.8: Displaying building information on feature selection.

CHAPTER

TWO

ADVANCED TOPICS

2.1 Vector Layers

2.1.1 What this module covers

This module covers vector layers in detail. In this module you will:

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.

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 Sydney, Australia). The first is serialized as GeoJSON (using the `OpenLayers.Format.GeoJSON`). The second is serialized as GML (OGC Geography Markup Language) (using the `OpenLayers.Format.GML.v3`).

GeoJSON Example

```
{
    "type": "Feature",
    "id": "OpenLayers.Feature.Vector_107",
    "properties": {},
    "geometry": {
        "type": "Point",
        "coordinates": [151.205091, -33.82597]
    }
}
```

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>151.205091 -33.82597</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.

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.

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*.

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:

```
<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 medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var imagery = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(imagery);

            var buildings = new OpenLayers.Layer.Vector("Buildings", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/wfs",
                    featureType: "medford_buildings",
                    featureNS: "http://medford"
                })
            });
            map.addLayer(buildings);

            map.zoomToMaxExtent();

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

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

```
var draw = new OpenLayers.Control.DrawFeature(
    buildings, 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/ol_workshop/map.html

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`.

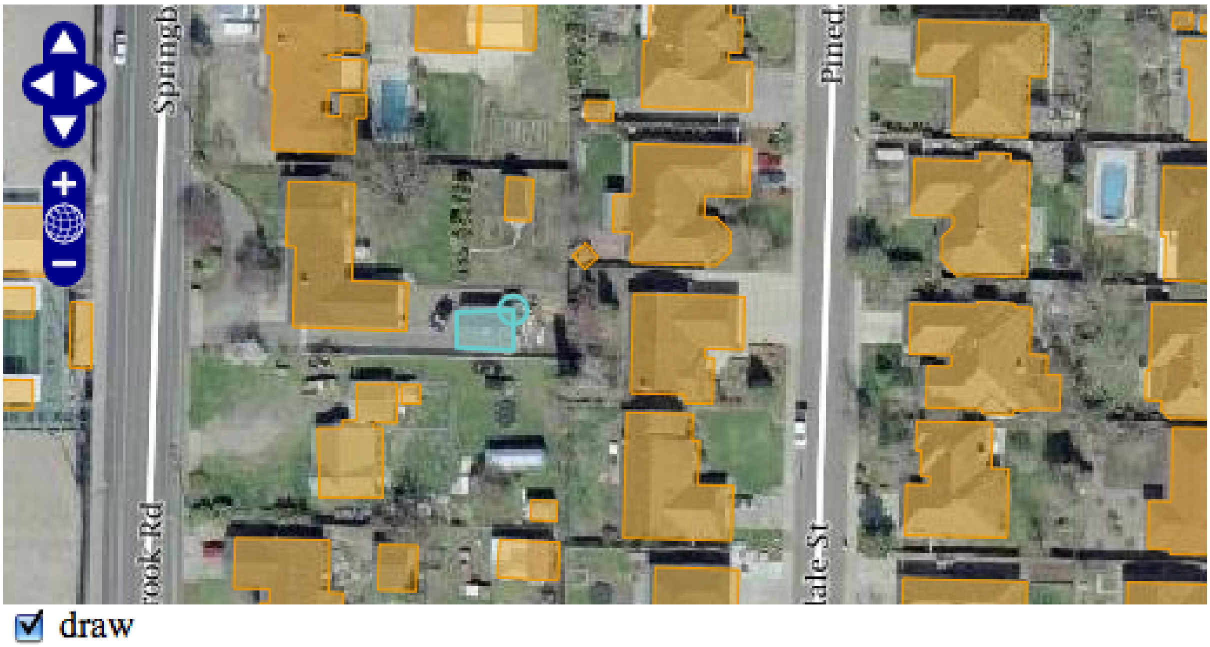


Figure 2.1: A control for adding features to a vector layer.

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

```
var buildings = new OpenLayers.Layer.Vector("Buildings", {
    strategies: [
        new OpenLayers.Strategy.BBOX(),
        new OpenLayers.Strategy.Save()
    ],
    protocol: new OpenLayers.Protocol.WFS({
        url: "/geoserver/wfs",
        featureType: "medford_buildings",
        featureNS: "http://medford"
    })
});
```

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

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

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

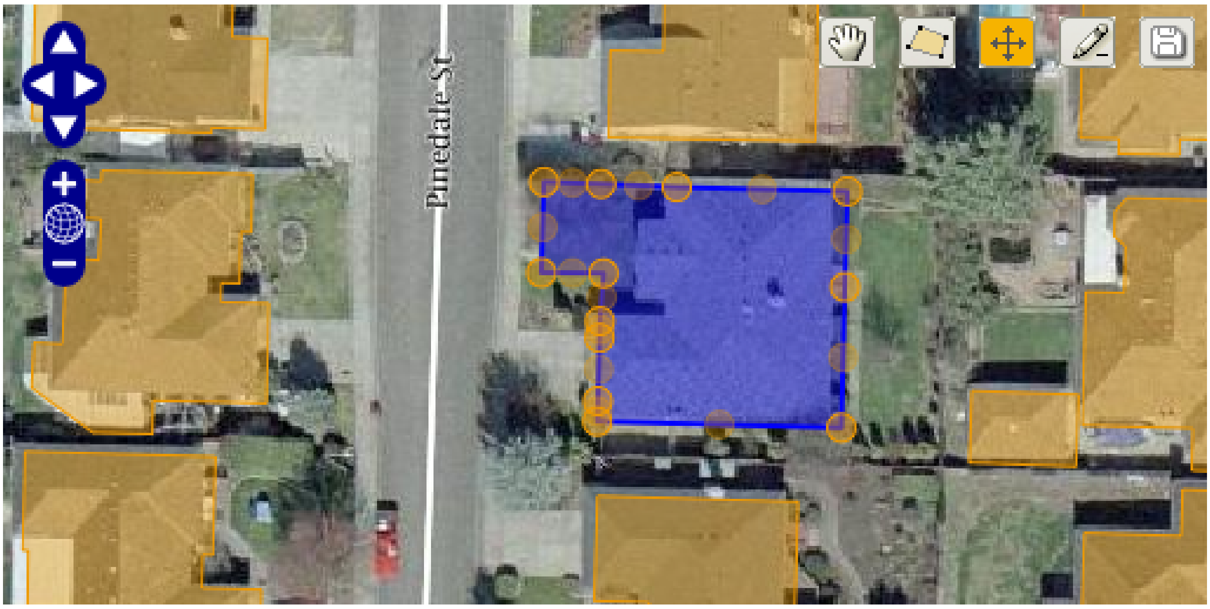


Figure 2.2: Modifying a building footprint.

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.

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"
})
```

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"
}
```

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"
    }
})
```

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.

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*.

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:

```
<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 medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var imagery = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(imagery);

            var buildings = new OpenLayers.Layer.Vector("Buildings", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/wfs",
                    featureType: "medford_buildings",
                    featureNS: "http://medford"
                })
            });
            map.addLayer(buildings);

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

2. Open this `map.html` file in your browser to see orange buildings over the base layer: http://localhost/ol_workshop/map.html

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

```
var buildings = new OpenLayers.Layer.Vector("Buildings", {
    strategies: [new OpenLayers.Strategy.BBOX()],
    protocol: new OpenLayers.Protocol.WFS({
        url: "/geoserver/wfs",
        featureType: "medford_buildings",
        featureNS: "http://medford"
    }),
    styleMap: new OpenLayers.StyleMap({
        "default": new OpenLayers.Style({
            strokeColor: "white",
            strokeWidth: 1
        }, {
            rules: [
                new OpenLayers.Rule({
                    filter: new OpenLayers.Filter.Comparison({
                        type: OpenLayers.Filter.Comparison.LESS_THAN,
                        property: "shape_area",
                        value: 3000
                    }),
                    symbolizer: {
                        fillColor: "olive"
                    }
                }),
                new OpenLayers.Rule({
                    elseFilter: true,
                    symbolizer: {
                        fillColor: "navy"
                    }
                })
            ]
        })
    })
});
```

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

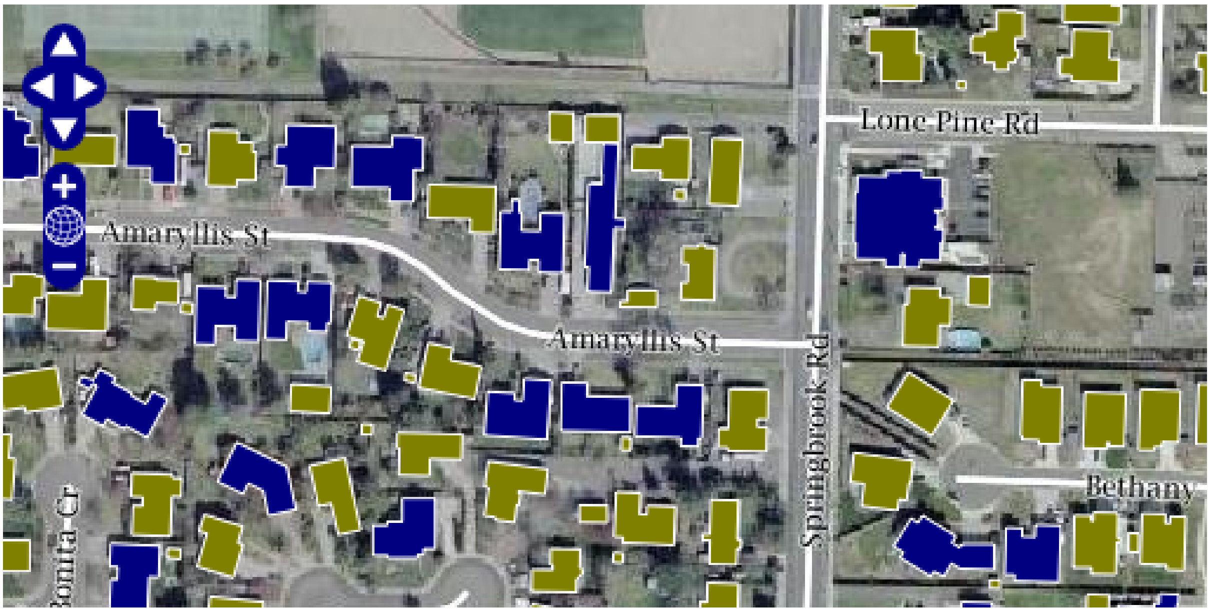


Figure 2.3: Buildings colored by footprint area.

2.2 Integration with Other Frameworks

OpenLayers provides the mapping foundation for your web application. To build an application with rich user interface components, OpenLayers is best paired with a UI framework. There are a number of full-featured UI frameworks available; Ext JS, Dojo (via Dijit), and YUI are popular frameworks with good layout controls and widgets. This module provides examples of OpenLayers integration with Ext JS as well as with the less established, but rapidly developing, jQuery UI libraries.

2.2.1 What this module covers

In this module you will:

Exploring jQuery UI

The jQuery library is focused on providing efficient shortcuts for accessing DOM elements, adding behavior to those elements, and simplifying typical AJAX coding patterns. The jQuery UI library builds on the jQuery core by providing themeable widgets. Both the UI library and the core are designed around, and dedicated to, unobtrusive JavaScript coding, by providing a concise workflow.

jQuery links:

- jQuery http://jquery.com/
- jQuery UI http://ui.jquery.com/

- jQuery UI Docs http://docs.jquery.com/UI

In addition to the links above, you'll find jQuery UI examples in the jquery-ui directory included with this workshop. Spend a bit of time reading the material on jQuery, including the demos and tutorials.

Given a basic understanding of jQuery UI, you're ready to move on to *creating an opacity slider*.

The jQuery UI Slider

The jQuery UI slider widget creates a draggable handle that can be configured to return a value based on the handle position. Raster layers in OpenLayers provide a `setOpacity` method that controls the image opacity and accepts values between 0 (totally transparent) and 1 (totally opaque). A jQuery UI slider widget is a user-friendly way to set layer opacity in an OpenLayers map.

A jQuery UI slider can be created with something like the following markup.

```
<div id="slider-id">
    <div class="ui-slider-handle"></div>
</div>
```

To give these elements the slider behavior, you would run the following code.

```
jQuery("#slider-id").slider();
```

The `jQuery` function is also exported under the alias `$`. In the examples below, you'll see the use of the `$` function. This is entirely equivalent to using the `jQuery` function.

Using a Slider to Control Layer Opacity

We'll start with a working example that displays one WMS (OGC Web Map Service) layer and one vector layer with features from a WFS (OGC Web Feature Service).

Tasks

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

    ```
    <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 medford = new OpenLayers.Bounds(
                    4284890, 254385,
                    4288865, 258380
                );
                var map = new OpenLayers.Map("map-id", {
                    projection: new OpenLayers.Projection("EPSG:2270"),
    ```

```
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(base);

            var buildings = new OpenLayers.Layer.Vector("Buildings", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/wfs",
                    featureType: "medford_buildings",
                    featureNS: "http://medford"
                })
            });
            map.addLayer(buildings);

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

2. Next we need to pull in the jQuery resources that our widgets will require. Add the following markup to the `<head>` of your `map.html` document:

   ```
   <link rel="stylesheet" href="jquery-ui/css/smoothness/jquery-ui-1.7.2.custom.css" type="text/
   <script src="jquery-ui/js/jquery-1.3.2.min.js"></script>
   <script src="jquery-ui/js/jquery-ui-1.7.2.custom.min.js"></script>
   ```

3. The slider widget needs some markup to start with. Insert the following in the `<body>` of your `map.html` page, just after the map viewport, in order to create a container for the slider:

   ```
   <div id="slider-id"><div class="ui-slider-handle"></div></div>
   ```

4. One bit of preparation before finalizing the code is to style the slider container. In this case, we'll make the slider as wide as the map and give it some margin. Insert the following style declarations into the `<style>` element within the `<head>` of your document:

   ```
   #slider-id {
       width: 492px;
       margin: 10px;
   }
   ```

5. Having pulled in the required jQuery resources, created some markup for the widget, and given it some style, we're ready to add the code that creates the slider widget. In the `<script>` element that contains your map initialization code, insert the following to create the slider widget and set up a listener to change your layer opacity as the slider value changes:

   ```
   $("#slider-id").slider({
       value: 100,
       slide: function(e, ui) {
           base.setOpacity(ui.value / 100);
       }
   });
   ```

6. Save your changes to `map.html` and open the page in your browser: http://localhost/ol_workshop/map.html

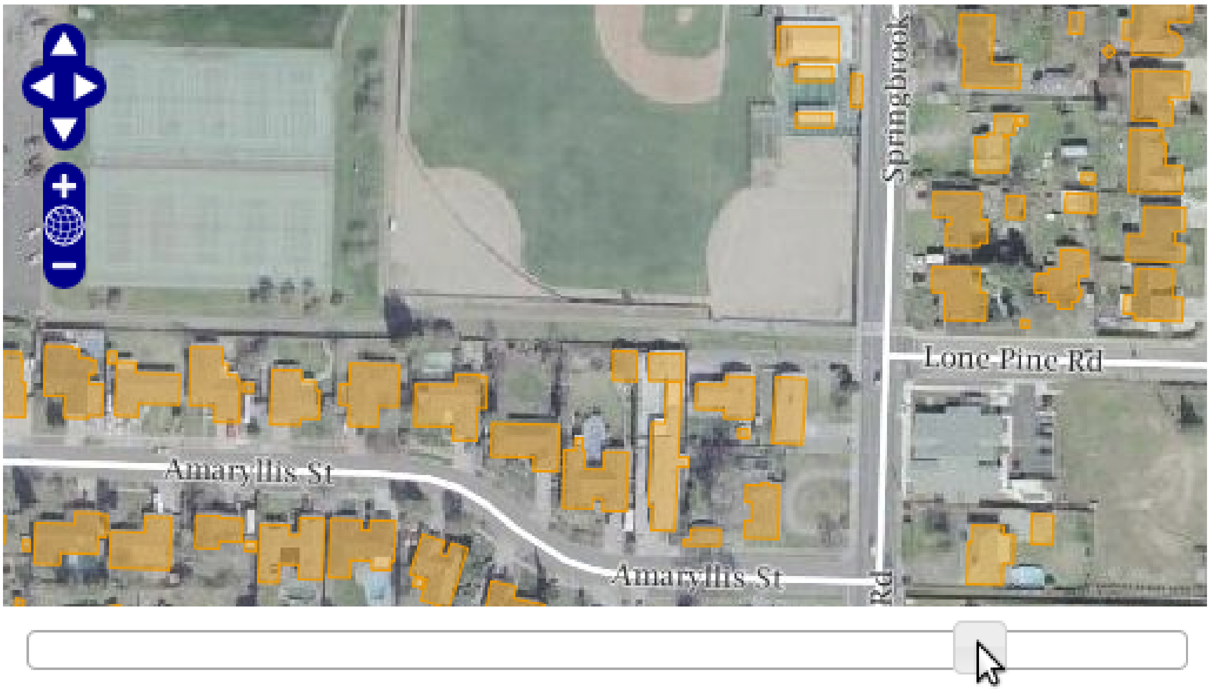


Figure 2.4: A map with a slider widget to control layer opacity.

Bonus Task

1. In the jQuery documentation, find the options for the slider function that allow you to specify a number of incremental steps within the slider range. Experiment with adding discrete intervals to the slider range. Modify the end values of the range to restrict opacity settings.

Having mastered the jQuery UI slider, you're ready to start working with *dialogs*.

The jQuery UI Dialog

If you are adding a map to a larger website, and you are already using jQuery UI for interface components, it makes good sense to create "popups" for your map that will be integrated with the style of the rest of your site. The jQuery UI `dialog` function provides a flexible way to produce themeable dialogs that serve a variety of purposes.

The *previous example* started with existing markup and used the `jQuery` function to select and modify DOM elements. The `jQuery` function can also be used to create elements given a string of HTML.

The code below creates a `<div>` element and turns it into a modeless dialog:

```
jQuery("<div>Hello!</div>").dialog();
```

This technique is used in the tasks below to create dialogs populated with information from a feature's attribute values.

Displaying Feature Information in a Dialog

Tasks

1. At the end of the *previous section*, you should have something like the code below in your `map.html` file. Open this file in your text editor and confirm the contents are similar to the following:

```
<html>
    <head>
        <title>My Map</title>
        <link rel="stylesheet" href="openlayers/theme/default/style.css" type="text/css">
        <link rel="stylesheet" href="jquery-ui/css/smoothness/jquery-ui-1.7.2.custom.css" typ
        <script src="jquery-ui/js/jquery-1.3.2.min.js"></script>
        <script src="jquery-ui/js/jquery-ui-1.7.2.custom.min.js"></script>
        <style>
            #map-id {
                width: 512px;
                height: 256px;
            }
            #slider-id {
                width: 482px;
                margin: 10px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <div id="slider-id"><div class="ui-slider-handle"></div></div>
        <script>
            var medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(base);

            var buildings = new OpenLayers.Layer.Vector("Buildings", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/wfs",
                    featureType: "medford_buildings",
                    featureNS: "http://medford"
                })
            });
            map.addLayer(buildings);
```

```
                map.zoomToMaxExtent();

                $("#slider-id").slider({
                    value: 100,
                    slide: function(e, ui) {
                        base.setOpacity(ui.value / 100);
                    }
                });

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

2. To this example, we'll be adding an `OpenLayers.Control.SelectFeature` control so that the user can select a feature. In your map initialization code, add the following *after* the creation of your `buildings` layer:

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

3. Next we need to create a listener for the `featureselected` event on our `buildings` layer. We'll create a dialog that populates with feature information, when the user selects a feature by clicking on it with the mouse. In addition, we want to remove the dialog when a feature is unselected. We can do this by listening for the `featureunselected` event. Insert the following in your map initialization code somewhere *after* the creation of the `buildings` layer:

```
var dialog;
buildings.events.on({
    featureselected: function(event) {
        var feature = event.feature;
        var area = feature.geometry.getArea();
        var id = feature.attributes.key;
        var output = "Building: " + id + " Area: " + area.toFixed(2);
        dialog = $("<div title='Feature Info'>" + output + "</div>").dialog();
    },
    featureunselected: function() {
        dialog.dialog("destroy").remove();
    }
});
```

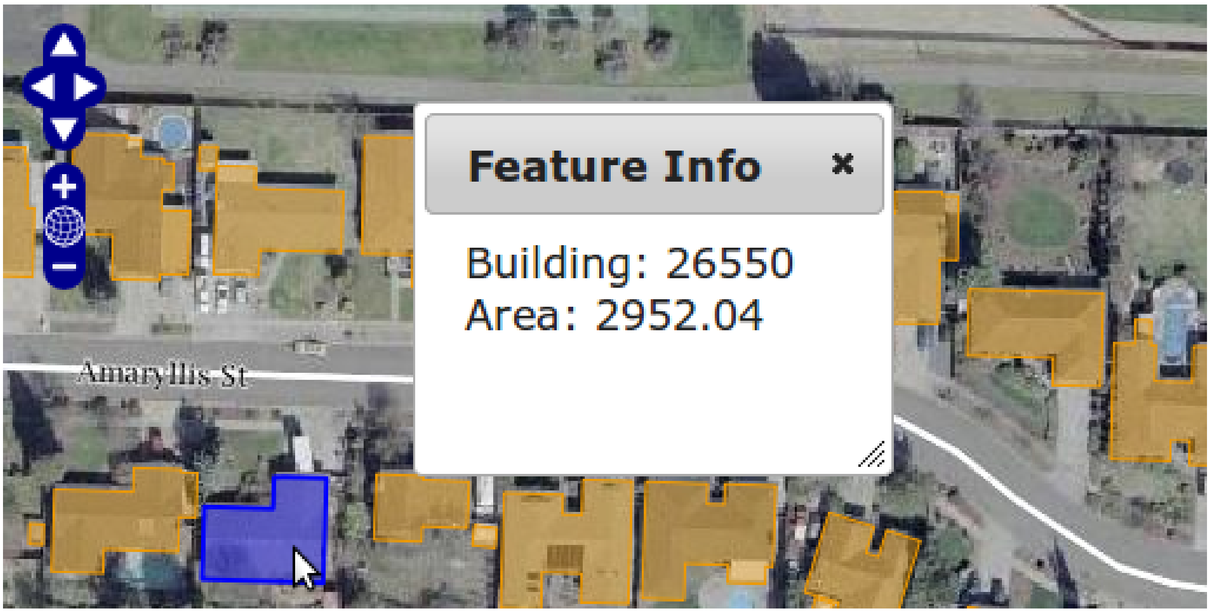
4. Save your changes to `map.html` and open the page in your browser: http://localhost/ol_workshop/map.html

Bonus Tasks

1. Find the appropriate documentation to determine how to make the feature dialog with modal behavior. Create a modal dialog for displaying feature information so a user will need to close it before interacting with anything else in the application.
2. Experiment with editing the style declarations in the head of the page in order to change the look of the displayed information. You can use the jQuery `addClass` function to add a class name to an element before calling `dialog()`.

Exploring Ext JS

Ext JS is an extremely full-featured library for complex layouts and dynamic widgets. Ext JS is especially useful for mimicking desktop applications in a browser. While the library is a heavier dependency and in turn a larger download than jQuery UI, it does provide a more robust tool set.



Feature Info

Building: 26550
Area: 2952.04

Figure 2.5: A map that displays feature information in a dialog.

Ext JS links:

- Ext JS overview http://extjs.com/products/extjs/
- Ext JS samples http://extjs.com/deploy/dev/examples/
- Ext JS documentation http://extjs.com/deploy/dev/docs/

Note that the entire Ext JS example set and documentation is available in the `ext` directory packaged with this workshop. Spend a bit of time exploring the Ext examples and figuring out how to use the documentation.

Given a basic understanding of Ext JS, you're ready to move on to *creating an opacity slider*.

The Ext JS Slider

As with the jQuery sliders, an Ext slider provides a widget for collecting a user supplied value within some range. This exercise will duplicate the functionality put together in the *jQuery UI slider* section above.

The configuration for Ext widgets is extremely flexible. One way to create a slider widget is to start with a DOM element that will serve as the slider container.

```
<div id="slider"></div>
```

Given the above, the following code creates a functioning Ext slider.

```
var slider = new Ext.Slider({renderTo: "slider"});
```

We'll use the above technique to create a slider for controlling layer opacity.

Using a Slider to Control Layer Opacity

We'll start with a working example that displays one WMS (OGC WEB MAP SERVICE) layer and one vector layer with features from a WFS (OGC WEB FEATURE SERVICE).

Tasks

1. Open your text editor and paste in the code used at the *start* of the *previous slider example*. Save this as `map.html` in the root of your workshop directory.
2. Next we need to pull in the Ext resources that our widget will require. Add the following markup to the `<head>` of your `map.html` document:

```
<link rel="stylesheet" href="ext/resources/css/ext-all.css" type="text/css">
<script src="ext/adapter/ext/ext-base.js"></script>
<script src="ext/ext-all.js"></script>
```

3. Now we'll create some markup that will create a container for the slider widget. In the `<body>` of your `map.html` file, just after the map viewport, insert the following:

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

4. One bit of preparation before finalizing the code is to style the slider container. In this case, we'll make it as wide as the map and give it some margin. Insert the following style declarations into the `<style>` element within the `<head>` of your document:

```
#slider-id {
    width: 492px;
    margin: 10px;
}
```

5. Somewhere in your map initialization code, add the following to create a slider in the container element and set up the slider listener to change layer opacity:

```
var slider = new Ext.Slider({
    renderTo: "slider-id",
    value: 100,
    listeners: {
        change: function(el, val) {
            base.setOpacity(val / 100);
        }
    }
});
```

6. Save your changes to `map.html` and open the page in your browser: http://localhost/ol_workshop/map.html

Bonus Task

1. Find the Slider widget in the Ext JS documentation. Locate the configuration option that allows you to specify a set of intervals for setting the slider value. Experiment with adding a set of intervals to the slider. Configure the slider to restrict the range of opacity values that can be set.

With a functioning layer opacity slider in your application, you're ready to move on to *working with windows*.

The Ext JS Window

Ext JS provides windows with behavior and looks familiar to desktop application developers. The theme for an Ext based application is moderately configurable, though developing custom themes can be labor-intensive. Using Ext windows to display map related information gives your application a well integrated feel if you are using Ext widgets for non-map related parts of your application.

An Ext window can be created with no existing markup. The following code creates a modeless window and opens it.

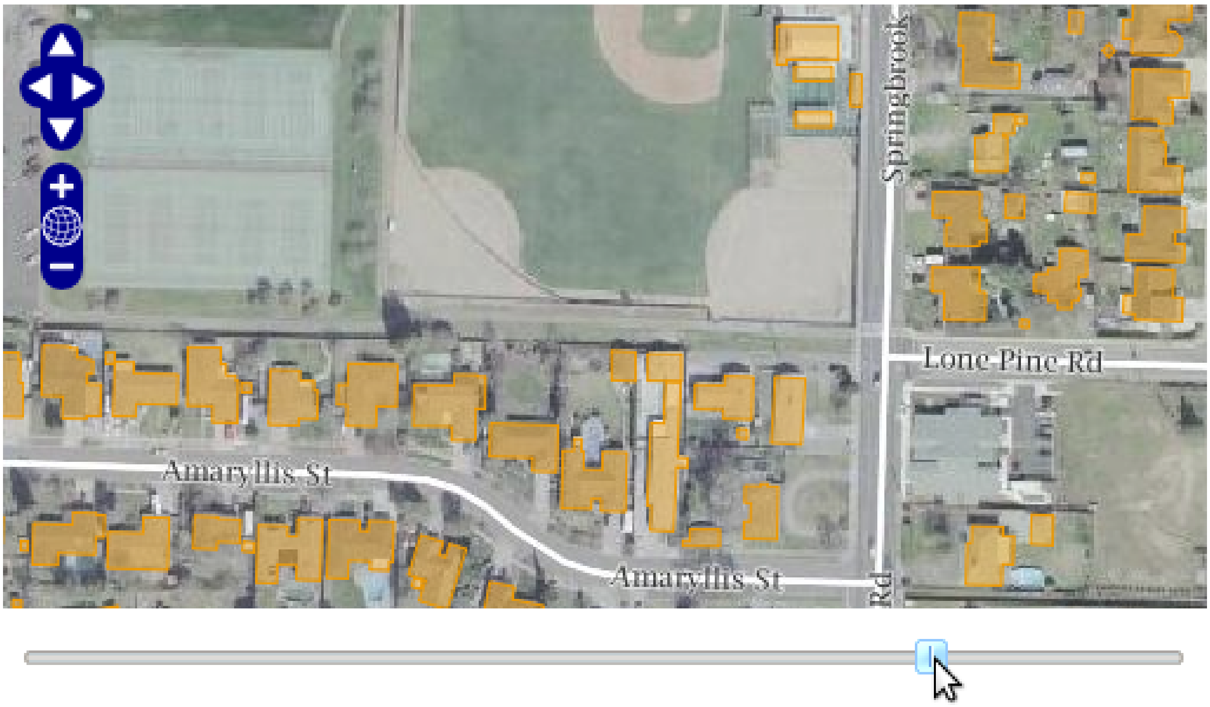


Figure 2.6: A map with a slider widget to control layer opacity.

```
var win = new Ext.Window({
    title: "Window",
    items: [
        {html: "Hello!"}
    ]
});
win.show();
```

Ext builds complex elements based on containers with one or more items. The window above contains a panel element that was created from the string `"Hello!"` Using the above technique, windows can be created to display information about features on your map.

Displaying Feature Information in a Window

Tasks

1. At the end of the *previous section*, you should have something like the code below in your `map.html` file. Open this file in your text editor and confirm the contents are similar to the following:

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

```
            #slider-id {
                width: 482px;
                margin: 10px;
            }
        </style>
        <script src="openlayers/lib/OpenLayers.js"></script>
    </head>
    <body>
        <h1>My Map</h1>
        <div id="map-id"></div>
        <div id="slider-id"></div>
        <script>
            var medford = new OpenLayers.Bounds(
                4284890, 254385,
                4288865, 258380
            );
            var map = new OpenLayers.Map("map-id", {
                projection: new OpenLayers.Projection("EPSG:2270"),
                units: "ft",
                maxExtent: medford,
                restrictedExtent: medford,
                maxResolution: 2.5,
                numZoomLevels: 5
            });

            var base = new OpenLayers.Layer.WMS(
                "Medford Streets & Imagery",
                "/geoserver/wms",
                {layers: "medford"}
            );
            map.addLayer(base);

            var buildings = new OpenLayers.Layer.Vector("Buildings", {
                strategies: [new OpenLayers.Strategy.BBOX()],
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/wfs",
                    featureType: "medford_buildings",
                    featureNS: "http://medford"
                })
            });
            map.addLayer(buildings);

            map.zoomToMaxExtent();

            var slider = new Ext.Slider({
                renderTo: "slider-id",
                value: 100,
                listeners: {
                    change: function(el, val) {
                        base.setOpacity(val / 100);
                    }
                }
            });
        </script>
    </body>
</html>
```

2. To this example, we'll be adding an `OpenLayers.Control.SelectFeature` control so that the user can select a feature. In your map initialization code, add the following *after* the creation of your `buildings` layer:

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

3. Next we need to create a listener for the `featureselected` event on our `buildings` layer. We'll create a window populated with feature information when the user selects a feature (by clicking on it with the mouse). In addition, we want to destroy the window when a feature is unselected. We can do this by listening for the `featureunselected` event. Insert the following in your map initialization code somehere *after* the creation of the `buildings` layer:

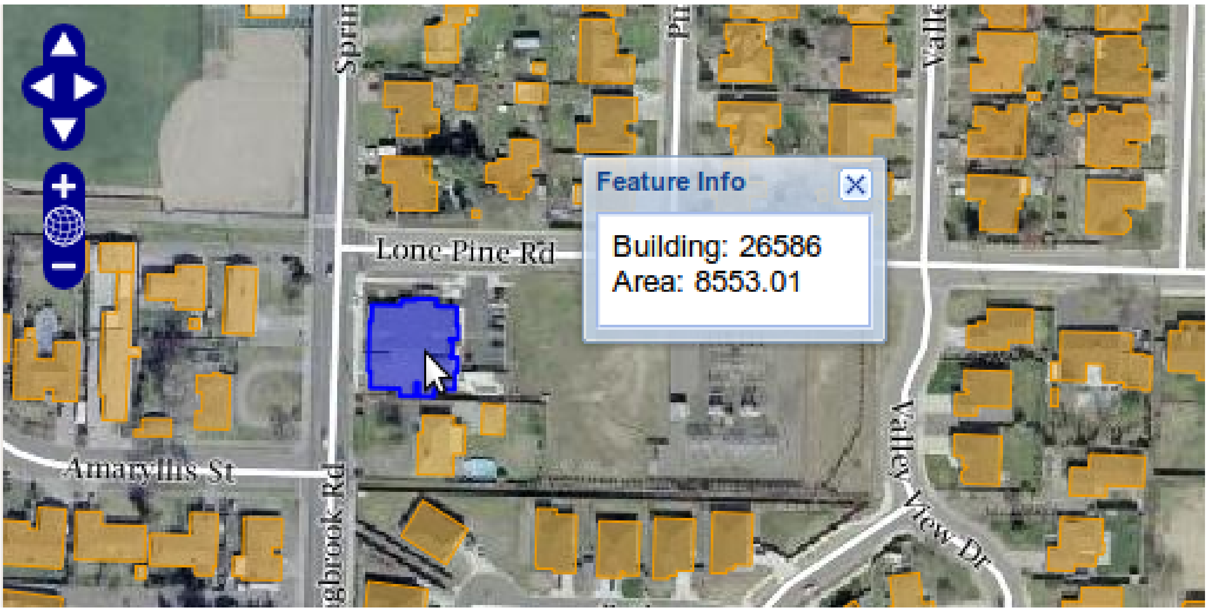
```
var dialog;
buildings.events.on({
    featureselected: function(event) {
        var feature = event.feature;
        var area = feature.geometry.getArea();
        var id = feature.attributes.key;
        var output = "Building: " + id + " Area: " + area.toFixed(2);
        dialog = new Ext.Window({
            title: "Feature Info",
            layout: "fit",
            height: 80, width: 130,
            plain: true,
            items: [{
                border: false,
                bodyStyle: {
                    padding: 5, fontSize: 13
                },
                html: output
            }]
        });
        dialog.show();
    },
    featureunselected: function() {
        dialog.destroy();
    }
});
```

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

Bonus Tasks

1. Generally speaking, setting element styles in code is bad. Ext allows for easy style declarations in the component configuration with the `style` and `bodyStyle` properties. In addition, Ext makes it easy to set CSS class names for components via the `cls` property. Experiment with adding a class name to the window contents and using declarations in your CSS instead of the `bodyStyle` configuration option above.

2. Find the `Ext.Window` constructor in the Ext documentation. Change the configuration of the windows in your `map.html` example to make them modal windows. Your application will not receive browser events until the window is closed.



Building: 26586
Area: 8553.01

Figure 2.7: A map that displays feature information in a window.

MODULE INDEX

INDEX