OpenLayers Documentation

OpenLayers is a pure JavaScript library for displaying map data in most modern web browsers, with no server-side dependencies. OpenLayers implements a JavaScript API for building rich web-based geographic applications, similar to the Google Maps and MSN Virtual Earth APIs, with one important difference – OpenLayers is Free Software, developed for and by the Open Source software community.

Table of Contents

OpenLayers Examples

One of the guiding principles of OpenLayers development has been to maintain a set of small examples of most functionality, allowing the library to demonstrate most of what it can do by example. This means that our examples are typically our most up to date source of documentation, and provide more than 100 different code snippets for use in your applications.

These examples are available from http://openlayers.org/dev/examples/

Code Documentation

Documentation for the OpenLayers API (http://dev.openlayers.org/apidocs/)

Developer Documentation for the OpenLayers API (http://dev.openlayers.org/docs/)

Getting Started

Creating Your First Map

The OpenLayers API has two concepts which are important to understand in order to build your first map: ‘Map’, and ‘Layer’. An OpenLayers Map stores information about the default projection, extents, units, and so on of the map. Inside the map, data is displayed via ‘Layer’s. A Layer is a data source – information about how OpenLayers should request data and display it.

Crafting HTML

Building an OpenLayers viewer requires crafting HTML in which your viewer will be seen. OpenLayers supports putting a map inside of any block level element – this means that it can be used to put a map in almost any HTML element on your page.

In addition to a single block level element, it is also required to include a script tag which includes the OpenLayers library to the page.

```
<html>
<head>
  <title>OpenLayers Example</title>
    <script src="http://openlayers.org/api/OpenLayers.js"></script>
    </head>
    <body>
      <div style="width:100%; height:100%" id="map"></div>
    </body>
</html>
```

Ex. 1: Creating your first HTML Page

Creating the Map Viewer

In order to create the viewer, you must first create a map. The OpenLayers.Map constructor requires one argument: This argument must either be an HTML Element, or the ID of an HTML element. This is the element in which the map will be placed.

```
var map = new OpenLayers.Map('map');
```

Ex. 2: Map Constructor

The next step to creating a viewer is to add a layer to the Map. OpenLayers supports many different data sources, from WMS to Yahoo! Maps to WorldWind. In this example, the WMS layer is used. The WMS layer is an example provided by MetaCarta.

```
var wms = new OpenLayers.Layer.WMS(
  "OpenLayers WMS",
  "http://labs.metacarta.com/wms/vmap0",
  {'layers':'basic'} );
map.addLayer(wms);
```

Ex. 3: Layer Constructor

The first parameter in this constructor is the URL of the WMS server. The second argument is an object containing the parameterss to be appended to the WMS request.

Finally, in order to display the map, you must set a center and zoom level. In order to zoom to fit the map into the window, you can use the zoomToMaxExtent function, which will zoom as close as possible while still fitting the full extents within the window.

Putting it All Together

The following code block puts all the pieces together to create an OpenLayers viewer.

```
<html>
<head>
  <title>OpenLayers Example</title>
    <script src="http://openlayers.org/api/OpenLayers.js"></script>
    </head>
    <body>
      <div style="width:100%; height:100%" id="map"></div>
      <script defer="defer" type="text/javascript">
        var map = new OpenLayers.Map('map');
        var wms = new OpenLayers.Layer.WMS( "OpenLayers WMS",
            "http://labs.metacarta.com/wms/vmap0", {layers: 'basic'} );
        map.addLayer(wms);
        map.zoomToMaxExtent();
      </script>

</body>
</html>
```

Ex. 4: Full HTML and Javascript for simple WMS browser

Adding an Overlay WMS

WMS layers have the capability to be overlaid on top of other WMS layers in the same projection. There are several ways to mark a layer as an overlay, rather than a base layer. With WMS, the best way to do this is by setting the ‘transparent’ parameter to ‘true’. The example here uses a political borders WMS to demonstrate overlaying a transparent WMS.

```
var twms = new OpenLayers.Layer.WMS( "World Map",
    "http://world.freemap.in/cgi-bin/mapserv?",
    { map: '/www/freemap.in/world/map/factbooktrans.map',
      transparent: 'true', layers: 'factbook'}
    );
map.addLayer(twms);
```

Ex. 5: How to add a transparent WMS overlay to your map.

Using the transparent: ‘true’ parameter sets two flags automatically:

- format parameter. The format option of the WMS layer is set to image/png if the browser supports transparent PNG images. (This is all browsers except for Internet Explorer 6.) In Internet Explorer 6, this will instead be set to image/gif.
- isBaseLayer option. The isBaseLayer option controls whether the layer can be displayed at the same time as other layers. This option defaults to false for the WMS layer, but setting transparent to true changes it to true by default.

Putting this code together with our earlier example, we get the following:

```
<html>
<head>
```

```
  <title>OpenLayers Example</title>
    <script src="http://openlayers.org/api/OpenLayers.js"></script>
    </head>
    <body>
      <div style="width:100%; height:100%" id="map"></div>
      <script defer="defer" type="text/javascript">
        var map = new OpenLayers.Map('map');
        var wms = new OpenLayers.Layer.WMS( "OpenLayers WMS",
            "http://labs.metacarta.com/wms/vmap0", {layers: 'basic'} );
        var twms = new OpenLayers.Layer.WMS( "World Map",
            "http://world.freemap.in/cgi-bin/mapserv?",
            { map: '/www/freemap.in/world/map/factbooktrans.map',
              transparent: 'true', layers: 'factbook'}
            );
        map.addLayers([wms, twms]);
        map.zoomToMaxExtent();
      </script>

</body>
</html>
```

Ex. 6: How to add a transparent WMS overlay to your map.

One thing to note here is that we have used addLayers on the map object to add both layers at the same time. This allows us to save a line of code in this case, and may be useful in other cases when you need to add multiple layers to the map at the same time.

Adding a Vector Marker to the Map

To add a single marker at a latitude and longitude to the map, you can use a Vector Layer to add an overlay.

```
var vectorLayer = new OpenLayers.Layer.Vector("Overlay");
var feature = new OpenLayers.Feature.Vector(
 new OpenLayers.Geometry.Point(-71, 42),
 {some:'data'},
 {externalGraphic: 'img/marker.png', graphicHeight: 21, graphicWidth: 16});
vectorLayer.addFeatures(feature);
map.addLayer(vectorLayer);
```

This is a simple demonstration – more information is available on overlays, how to interact with them, and how to control and style them via the *Styling* and *Overlays* documentation.

Understanding OpenLayers Syntax

This section is intended to get users comfortable with the syntax used when developing an application with OpenLayers.

OpenLayers “Classes”

OpenLayers is written in a “classical” style. This means that the library provides functions intended to be used with the `new` keyword that return objects. These functions all begin with a capital letter.

```
var map = new OpenLayers.Map("map", options);
```

The code above creates a new `map` object with all the properties of the `OpenLayers.Map` function's prototype. The properties that are intended to be set or accessed are documented in the API Documentation

In general, all parameters to a constructor are required, except for the options parameter. When this is not the case, the API documentation for the constructor will usually say so.

The `options` Argument

Most OpenLayers object constructors take an `options` object as one of their arguments. In general, you can set the value of any API property in a contructor's options argument.

For example, looking at the API Docs for the Vector layer, you can see the `isBaseLayer` property. If you specify a value for `isBaseLayer` in the options argument, this will be set on the layer.

For example:

```
var layer = new OpenLayers.Layer.Vector("Layer Name", {
    isBaseLayer: true
});

layer.isBaseLayer === true;  // this is true
```

You can also see that the Vector layer inherits from `OpenLayers.Layer`. Following the link to the Layer API documentation, you'll find a reference to the layer's `projection` property. As with `isBaseLayer`, if you provide a `projection` property in the `options` argument, this value will be set on the layer.

```
var layer = new OpenLayers.Layer.Vector("Layer Name", {
    projection: new OpenLayers.Projection("EPSG:900913")
});
```

Layers

Layers are the 'datasources' in OpenLayers.

Base Layers and Non-Base Layers

OpenLayers has two types of layers when operating in your application: base layers and overlays. This difference controls several aspects of how you interact with an OpenLayers Map.

Base Layers

Base Layers are mutually exclusive layers, meaning only one can be enabled at any given time. The currently active base layer determines the available projection (coordinate system) and zoom levels available on the map.

Whether a layer is a base layer or not is determined by the isBaseLayer property on the layer. Most raster layers have the isBaseLayer property set to true by default. It can be changed in the layer options.

Base Layers always display below overlay layers.

Non Base Layers

Non base layers – sometimes called overlays – are the alternative to Base Layers. Multiple non-base layer can be enabled at a time. These layers do not control the zoom levels of the map, but can be enabled or disabled at certain scales by min/max scale/resolution parameters so that they are only enabled at a certain level.

Some types of overlays support reprojection to the base layer projection at layer load time. Most overlay layers default to non-base overlays, as does the base Layer class. Non-base Layers always display above base layers.

Raster Layers

Raster Layers are imagery layers. These layers are typically in a fixed projection which can not be changed on the client side.

Google

Layer for using Google Maps data within OpenLayers. For API information, see the Google Layer API Docs. For an example of usage, see the Spherical Mercator example.

If you are overlaying other data on a Google Maps base layer, you will want to be interacting with the Google Maps layer in projected coordinates. (This is important if you are working with imagery data especially.) You can read more about the 'Spherical Mercator' projection that Google Maps – and other commercial layers – use in the *Spherical Mercator* documentation.

The Google Layer class is designed to be used only as a base layer.

Image

For API information, see the Image Layer API Docs.

KaMap

For API information, see the KaMap Layer API Docs.

KaMapCache

For API information, see the KaMapCache Layer API Docs.

MapGuide

For API information, see the MapGuide Layer API Docs.

MapServer

This layer is not required to interact with MapServer. In general, the *WMS* Layer is preferred over the MapServer Layer. Since MapServer exposes most of its CGI functionality in WMS mode as well, the WMS layer is preferred. The MapServer layer can often lead to maps which seem to work, but don't due to projection issues or other similar misconfigurations. Unless you have a strong reason not to, you should use the Layer.WMS instead of a Layer.MapServer.

If you are using a Layer.MapServer, and your map is being repeated several times, this indicates

that you have not properly configured your map to be in a different projection. OpenLayers can not read this information from your mapfile, and it must be configured explicitly. The FAQ on setting different projection properties provides information on how to configure different projections in OpenLayers.

For API information, see the MapServer Layer API Docs.

MultiMap

For API information, see the MultiMap Layer API Docs.

TMS

For API information, see the TMS Layer API Docs.

TileCache

For API information, see the TileCache Layer API Docs.

VirtualEarth

For API information, see the VirtualEarth Layer API Docs.

WMS

Layer type for accessing data served according to the Web Mapping Service standard.

For API information, see the WMS Layer API Docs.

WorldWind

For API information, see the WorldWind Layer API Docs.

Yahoo

For API information, see the Yahoo Layer API Docs.

Overlay Layers

Overlay layers are any layers that have their source data in a format other than imagery. This includes subclasses of both *Markers* Layers and *Vector* Layers. For more information on the differences between these two base classes, see the *Overlays* documentation.

Boxes

Based on subclassing markers. In general, it is probably better to implement this functionality with a Vectors layer and polygon geometries. Maintained for backwards compatibility.

For API information, see the Boxes Layer API Docs.

GML

The GML layer is a vector layer subclass designed to read data from a file once and display it. It is

ideal for working with many formats, not just GML, and can be configured to read other formats via the ‘format’ option on the layer.

The simplest use case of the GML layer is simply to load a GML file. The GML Layer Example shows this: simply add:

```
var layer = new OpenLayers.Layer.GML("GML", "gml/polygon.xml")
map.addLayer(layer);
```

If you want to add a different type of format, you can change the format option of the layer at initialization. The KML example demonstrates this:

```
var layer = new OpenLayers.Layer.GML("KML", "kml/lines.kml", {
   format: OpenLayers.Format.KML
})
map.addLayer(layer);
```

You can also add formatOption to the layer: these options are used when creating the format class internally to the layer.

```
var layer = new OpenLayers.Layer.GML("KML", "kml/lines.kml", {
   format: OpenLayers.Format.KML,
   formatOptions: {
     'extractStyles': true
   }
});
map.addLayer(layer);
```

The format options are determined by the format class.

For API information, see the GML Layer API Docs.

GeoRSS

The GeoRSS layer uses the GeoRSS format, and displays the results as clickable markers. It is a subclass of the Markers layer, and does not support lines or polygons. It has many hardcoded behaviors, and in general, you may be better off using a GML layer with a SelectFeature Control instead of the GeoRSS layer if you want configurability of your application behavior. (For more information on how to make that transition, see *Transitioning from Text Layer or GeoRSS Layer to Vectors*.)

For API information, see the GeoRSS Layer API Docs.

Markers

The Markers base layer is simple, and allows use of the addMarkers function to add markers to the layer. It supports only points, not lines or polygons.

For API information, see the Markers Layer API Docs.

PointTrack

For API information, see the PointTrack Layer API Docs.

Text

The Text layer uses the Text format, and displays the results as clickable markers. It is a subclass of

the Markers layer, and does not support lines or polygons. It has many hardcoded behaviors, and in general, you may be better off using a GML layer with a SelectFeature Control instead of the Text layer if you want configurability of your application behavior. (For more information on how to make that transition, see *Transitioning from Text Layer or GeoRSS Layer to Vectors*.)

For API information, see the Text Layer API Docs.

Vector

The Vector Layer is the basis of the advanced geometry support in OpenLayers. Classes like GML and WFS subclass from the Vector layer. When creating features in JavaScript code, using the Vector layer directly is likely a good way to go.

As of OpenLayers 2.7, development has begun on extending the Vector Layer to have additional capabilities for loading data, to replace the large number of layer subclasses. This work on Strategy and Protocol classes is designed to make it easier to interact with data from remote datasources. For more information on Protocols and Strategies, see the OpenLayers API documentation.

For API information, see the Vector Layer API Docs.

WFS

For API information, see the WFS Layer API Docs.

Generic Subclasses

- EventPane
- FixedZoomLevels
- Grid
- HTTPRequest
- SphericalMercator

Controls

Controls are OpenLayers classes which affect the state of the map, or display additional information to the user. Controls are the primary interface for map interactions.

Default Controls

The following controls are default controls on the map:

- *ArgParser*
- Attribution
- Navigation
- PanZoom

Panels

Control panels allow collections of multiple controls together, as is common in many applications.

Styling Panels

Panels are styled by CSS. The “ItemActive” and “ItemInactive” are added to the control’s displayClass.

All controls have an overridable ‘displayClass’ property which maps to their base CSS class name. This name is calculated by changing the class name by removing all ‘.’ characters, and changing “OpenLayers” to “ol”. So, OpenLayers.Control.ZoomBox changes to olControlZoomBox.

Panel items are styled by combining the style of the Panel with the style of a control inside of it. Using the NavToolbar Panel as an example:

```
.olControlNavToolbar div {
  display:block;
  width:  28px;
  height: 28px;
  top: 300px;
  left: 6px;
  position: relative;
}
.olControlNavToolbar .olControlNavigationItemActive {
  background-image: url("img/panning-hand-on.png");
  background-repeat: no-repeat;
}
.olControlNavToolbar .olControlNavigationItemInactive {
  background-image: url("img/panning-hand-off.png");
  background-repeat: no-repeat;
}
```

Here, we say:

- Div elements displayed inside the toolbar are 28px wide and 28px high. The top of the div should be at 300px, the left side should be at 6px.
- Then, for the control, we provide two background images: one for active and one for inactive.

For toolbars to go left to right, you can also control them with CSS:

```
.olControlEditingToolbar div {
    float:right;
    right: 0px;
    height: 30px;
    width: 200px;
}
```

Simply set the ‘float: right’ parameter, and give the parent element some

In order to improve the user experience, existing panels like the *EditingToolbar* use a single background image, and control the icon to display via ‘top’ and ‘left’ parameters, offsetting and clipping the background image. This is not required, but doing this makes it so that when you select a tool, you don’t have to wait for the ‘inactive’ image to display before continuing.

Controls to be used with Panels

Panels can have controls of many ‘types’ inside of them. Each tool in a panel should have a ‘type’ attribute which is one of:

- OpenLayers.Control.TYPE_TOOL (the default)

- OpenLayers.Control.TYPE_BUTTON
- OpenLayers.Control.TYPE_TOGGLE

Customizing an Existing Panel

Several existing panels – like the *EditingToolbar* or *PanPanel* – have multiple controls combined, but it is not always desirable to use all those controls. However, it is relatively simple to createa control which mimics the behavior of these controls. For example, if you wish to create an editing toolbar control that only has the ability to draw lines, you could do so with the following code:

```
var layer = new OpenLayers.Layer.Vector();
var panelControls = [
 new OpenLayers.Control.Navigation(),
 new OpenLayers.Control.DrawFeature(layer,
     OpenLayers.Handler.Path,
     {'displayClass': 'olControlDrawFeaturePath'})
];
var toolbar = new OpenLayers.Control.Panel({
   displayClass: 'olControlEditingToolbar',
   defaultControl: panelControls[0]
});
toolbar.addControls(panelControls);
map.addControl(toolbar);
```

There are two things to note here:

- We are reusing the style of the EditingToolbar by taking its ‘displayClass’ property. This means we will pick up the default icons and so on of the CSS for that toolbar. (For more details, see *Styling Panels*.)
- We set the default control to be the Navigation control, but we could just as easily change that.

In this way, you can use any control which works in a panel – including, for example, the SelectFeature control, the ZoomToMaxExtent control, and more, simply by changing the controls which are in the list.

Map Controls

ArgParser

Takes URL arguments, and updates the map.

In order for the ArgParser control to work, you must check that ‘getCenter()’ returns null before centering your map for the first time. Most applications use a setCenter or zoomToMaxExtent call: this call should be avoided if the center is already set.

```
var map = new OpenLayers.Map('map');
var layer = new OpenLayers.Layer();
map.addLayer(layer);

// Ensure that center is not set
if (!map.getCenter()) {
    map.setCenter(new OpenLayers.LonLat(-71, 42), 4);
}
```

The ArgParser control is enabled by default.

Attribution

The attribution control will display attribution properties set on any layers in the map in the lower right corner of the map, by default. The style and location of this control can be overridden by overriding the ‘olControlAttribution’ CSS class.

Use of the attribution control is demonstrated in the Attribution example. For API information, see the Attribution API Docs.

DragFeature

DragPan

The DragPan control implements map dragging interactions.

DrawFeature

EditingToolbar

Display a *Navigation* control, along with three editing tools: Point, Path, and Polygon. If this does not fit your needs, see *Customizing an Existing Panel* above.

KeyboardDefaults

LayerSwitcher

Measure

A planar distance measuring tool.

ModifyFeature

The ModifyFeature control can be used to edit an existing vector object.

This control causes three different types of events to fire on the layer: * beforefeaturemodified - triggered when a user selects the feature to begin editing. * featuremodified - triggered when a user changes something about the feature. * afterfeaturemodified - triggered after the user unselects the feature.

To register for one of these events, register on the layer:

```
var layer = new OpenLayers.Layer.Vector("");
layer.events.on({
  'beforefeaturemodified': function(evt) {
      console.log("Selected " + evt.feature.id  + " for modification");
  },
  'afterfeaturemodified': function(evt) {
      console.log("Finished with " + evt.feature.id);
  }
});
```

There are several different modes that the ModifyFeature control can work in. These can be combined to work together.

- RESHAPE – The default. Allos changing the vertices of a feature by dragging existing vertices, creating new vertices by dragging 'virtual vertices', or deleting vertices by hovering over a vertice and pressing the delete key.
- RESIZE – Allows changing the size of a geometry.
- ROTATE – change the orientation of the geometry
- DRAG – change the position of the geometry.

When creating the control, you can use a bitwise OR to combine these:

```
var modifyFeature = new OpenLayers.Control.ModifyFeature(layer, {
  mode: OpenLayers.Control.ModifyFeature.RESIZE |
OpenLayers.Control.ModifyFeature.DRAG
});
```

For an example of using the ModifyFeature control, see the ModifyFeature example. For API information, see the ModifyFeature API Documentation.

The ModifyFeature control can only be used with a single layer at any given time. To modify multiple layers, use multiple ModifyFeature controls.

Deprecation Warning

As of OpenLayers 2.6, the onModificationStart, onModification and onModificationEnd functions on this control are no longer the recommended way to receive modification events. Instead, use the beforefeaturemodified, featuremodified, and afterfeaturemodified events to handle these cases.

MousePosition

NavToolbar

Navigation

The replacement control for the former *MouseDefaults* control. This control is a combination of:

- *DragPan*
- *ZoomBox*
- Handler.Click, for double click zooming
- Handler.Wheel, for wheel zooming

The most common request for the Navigation control is to disable wheel zooming when using the control. To do this, ensure that no other navigation controls are added to your map – for example, by an *EditingToolbar* – and call disableWheelNavigation on the Navigation control.

NavigationHistory

OverviewMap

PanPanel

A set of visual buttons for controlling the location of the map. A subclass of Control.Panel, this is easily controlled by styling via CSS. The `.olControlPanPanel` class, and its internal divs, control the styling of the PanPanel. If you wish to customize the look and feel of the controls in the upper left corner of the map, this control is the one for you.

This control is designed to work with the *ZoomPanel* control to replicate the functionality of the *PanZoom* control.

PanZoom

PanZoomBar

Permalink

The Permalink control, together with the ArgParser control, are designed to make it easy to link to an existing map view. By adding the permalink control to your map, you will make available a piece of text in the map that acts as a link for your users.

In order for the permalink to work, you must ensure that you check whether the center has already been set by the ArgParser control before you set the center of your map. To do this, simply check map.getCenter() first:

```
if (!map.getCenter()) {
  map.setCenter(new OpenLayers.LonLat(0,0),0);
}
```

Scale

ScaleLine

SelectFeature

Snapping

Provides an agent to control snapping vertices of features from one layer to nodes, vertices, and edges of features from any number of layers. The control operates as a toggle - acting as a snapping agent while active and not altering behavior of editing while not active. The control can be configured to allow node, vertex, and or edge snapping to any number of layers (given vector layers with features loaded client side). The tolerance, snapping type, and an optional filter can be configured for each target layer.

Find more detail on the *OpenLayers.Control.Snapping* page.

Split

Provides an agent for splitting linear features on a vector layer given edits to linear features on any other vector layer or a temporary sketch layer. The control operates in two modes. By default, the control allows for temporary features to be drawn on a sketch layer (managed by the control) that will be used to split eligible features on the target layer. In auto-split mode, the control listens for edits (new features or modifications to existing features) on an editable layer and splits eligible features on a target layer.

The control can be added to a map as with other controls. It has no distinct visual representation but can be connected to a button or other tool to toggle activation with a click. In addition, no GUI elements are provided for control configuration. Collecting user input to configure behavior of the control is an application specific task.

Find more detail on the *OpenLayers.Control.Split* page.

ZoomBox

ZoomPanel

A set of visual buttons for controlling the zoom of the map. A subclass of Control.Panel, this is easily controlled by styling via CSS. The `.olControlZoomPanel` class, and its internal divs, control the styling of the PanPanel. If you wish to customize the look and feel of the controls in the upper left corner of the map, this control is the one for you.

This control is designed to work with the *PanPanel* control to replicate the functionality of the *PanZoom* control.

Button Classes

These classes have no UI on their own, and are primarily designed to be used inside of a control panel.

Pan

Used inside the PanPanel; when triggered, causes the map to pan in a specific direction.

ZoomIn

Used inside the PanPanel; when triggered, causes the map to zoom in.

ZoomOut

Used inside the PanPanel; when triggered, causes the map to zoom out.

ZoomToMaxExtent

Used inside the PanPanel; when triggered, causes the map to zoomToMaxExtent.

Generic Base Classes

The following classes are used primarily for subclassing, and are not meant to be used directly.

Button

Used inside of Panel controls.

Panel

Used as a base for NavToolbar and EditingToolbar controls, as well as others. Gathers up buttons/tools to be used together.

Deprecated Controls

MouseDefaults

Replaced by the *Navigation* control.

MouseToolbar

Replaced by the *NavToolbar* control.

More documentation

- OpenLayers.Control.Snapping
- OpenLayers.Control.Split

Formats

Formats are classes that parse different sources of data into OpenLayers internal objects. Most (but not all) formats are centered around reading data from an XML DOM or a string and converting them to OpenLayers.Feature.Vector objects.

Formats are a way of transforming data from a server to objects that OpenLayers can interact with.

Built In Formats

KML

The KML format reads KML data and returns an array of OpenLayers.Feature.Vector objects.

The KML parser supports parsing local and remote styles.

The KML parser supports fetching network links.

For fetching remoteData, the maxDepth option must be greater than 0. This option tells the KML parser how far the traverse before giving up.

Note: Prior to OpenLayers 2.8, the maxDepth option was broken. No setting in the KML format would cause it to fetch network links or remote styles.

Creating Custom Formats

Creating custom formats, particularly for use with OpenLayers Protocols, is relatively easy: simply

create a subclass of Format that has a ‘read’ method which takes in a string, and returns an array of features.

```
var MyFormatClass = OpenLayers.Class(OpenLayers.Format.XML, {
    read: function(data) {
        // We're going to read XML
        if(typeof data == "string") {
            data = OpenLayers.Format.XML.prototype.read.apply(this, [data]);
        }
        var elems = data.getElementByTagName("line");
        var features = [];
        var wkt = new OpenLayers.Format.WKT();
        for (var i = 0; i < elems.length; i++) {
            var node = elems[i];
            var wktString = this.concatChildValues(node);
            features.push(wkt.read(wktString));
        }
        return features;
    }
});
```

This will read an XML document that has a series of ‘line’ XML elements with WKT embedded in each of them. It can be used with a “‘format: MyFormatClass’ line in a Layer.GML, or a ‘format: new MyFormatClass()’ in Protocols which support setting a format.

Overlays

OpenLayers allows you to lay many different types of data on top of its various data sources. Currently, there are two main ways of displaying vector feature overlays in OpenLayers, each with benefits and drawbacks. This document seeks to describe the differences, and ways in which each can be used.

Overlay Basics

There are two different types of feature rendering in OpenLayers. One type is the OpenLayers *Vector Overlays* support, which uses vector drawing capabilities in the browser (SVG, VML, or Canvas) to display data. The other type is the OpenLayers *Marker Overlays* support. This type of layer displays HTML image objects inside the DOM.

In general, the Vector layer provides more capabilities, with the ability to draw lines, polygons, and more. The Vector-based Layers are better maintained, and are the place where most new OpenLayers development is taking place. There is more support for various styling options, and more configurability over layer behavior and interactions with remote servers.

However, the Markers layer is maintained for backwards compatibility, because there are some things you can not do with vectors as they are currently implemented, and they provide a different type of interface for event registration.

Vector Overlays

Vector Layers form the core of Vector overlays. Vector overlays are powered by adding sets of OpenLayers.Feature.Vectors to the map. These can be a number of types of geometry:

- Point / MultiPoint

- Line / MultiLine
- Polygon / MultiPolygon

They are styled using the OpenLayers.Style / OpenLayers.StyleMap properties.

Examples:

- StyleMap Example:

  Use “Rules” to determine style attributes based on feature properties. This is useful for rendering based on data attributes like population.

- Context Example:

  Use a custom Javascript function to determine feature style properties. This example shows how to use which quadrant of the world a feature is in to determine its color. Similar rules can be used to do computations on a feature property to generate a style value (like size).

- Rotation Example:

  Vector features support advanced styling, like feature rotation. This can be used, for example, to display vehicle direction, wind direction, or other direction-based attributes.

- Unique Value Style Example:

  A common use case is to pick a specific style value based on a key/value mapping of a feature. This example demonstrates how to do that.

Interaction

Vector layer interaction is achieved through the SelectFeatureControl. This control allows selection of features, using DOM events to capture which feature is clicked on.

To handle feature events on a Vector Layer, you use the SelectFeature control, in combination with an event listener registered on the layer, on the ‘featureselected’ event.

```
function selected (evt) {
    alert(evt.feature.id + " selected on " + this.name);
}
var layer = new OpenLayes.Layer.Vector("VLayer");
layer.events.register("featureselected", layer, selected);
```

Once you have done this, you can add a select feature control to your map:

```
var control = new OpenLayers.Control.SelectFeature(layer);
map.addControl(control);
control.activate();
```

The activate call will move the vector layer to the forefront of the map, so that all events will occur on this layer.

As of OpenLayers 2.7, there is no support for selecting features from more than a single vector layer at a time. The layer which is currently being used for selection is the last one on which the `.activate()` method of the attached select feature control was called.

Layer Types

- *Vector* (Base Class)
- *GML* – can load many different types of data.
- *PointTrack*
- *WFS*

Marker Overlays

Markers support only point geometries. They are styled only using the OpenLayers.Icon class. They do not support lines, polygons, or other complex features. Their interaction method differs significantly from vector layers.

In general, Markers are the 'older' way to interact with geographic data in the browser. Most new code should, where possible, use vector layers in place of marker layers.

Interaction

Interaction on marker layers is achieved by registering events on the individual marker event property:

```
var marker = new OpenLayers.Marker(lonlat);
marker.id = "1";
marker.events.register("mousedown", marker, function() {
    alert(this.id);
});
```

Any number of events can be registered, and different events can be registered for each feature.

Layer Types

- *Markers* (Base Class)
- *GeoRSS*
- *Text*
- *Boxes* (Uses Special "Box" Marker)

Transitioning from Text Layer or GeoRSS Layer to Vectors

Many OpenLayers-applications make use of the *Text* Layer or *GeoRSS* Layer, which each parse a file (tab separated values) and displays markers an the provided coordinates. When clicking on one of the markers a popup opens and displays the content of the name and description from that location.

This behavior is relatively easy to achieve using vector layers, and doing so allows for more configurability of the behavior when clicking on a feature. Instead of being forced to use popups, you can instead cause the browser to go to a new URL, or change the behavior in other ways.

Loading Data

To mimic the loading behavior of a *Text* Layer or a *GeoRSS* Layer, there are two options:

- Use a *GML* Layer – covered in this document.
- Use a *Vector* Layer, with a strategy and protocol.

In either case, the way for controlling the behavior of the feature selection is the same.

Loading data with a GML Layer

The *GML* Layer is a simple “Load data from a URL once” data layer. You provide it a URL, and a format to use, and it will load the data from the URL, and parse it according to the format.

```
var layer = new OpenLayers.Layer.GML("Layer Name",
   "http://example.com/url/of/data.txt",
   { format: OpenLayers.Format.Text });
map.addLayer(layer);
map.zoomToMaxExtent();
```

This will cause your data to load, displaying your data as points on the map.

Styling Data

Some data formats do not include styling information, like GeoRSS. In order to match the default OpenLayers style to the default marker in OpenLayers, you should create a StyleMap that matches the default OpenLayers style:

```
var style = new OpenLayers.Style({
    'externalGraphic': OpenLayers.Util.getImagesLocation() + "marker.png",
    'graphicHeight': 25,
    'graphicWidth': 21,
    'graphicXOffset': -10.5,
    'graphicYOffset': -12.5
});

var styleMap = new OpenLayers.StyleMap({'default':style});

var layer = new OpenLayers.Layer.GML("Layer Name",
   "http://example.com/url/of/data.txt",
   {
     format: OpenLayers.Format.GeoRSS,
     styleMap: styleMap
   }
);
```

Using a style map like this will result in no visible difference when your feature is selected. To create a different style for selection – for example, with a different marker color – you could craft a second style object, and instead create your styleMap like:

```
var styleMap = new OpenLayers.StyleMap({
    'default': style,
    'select': selectStyle
});
```

For more information on styling your features, see the *Styling* or *StyleMap* documentation.

Displaying Popups

The *Text* Layer and the *GeoRSS* Layer open popups containing title and description text for the feature when clicked. Replicating this behavior in your application is easy.

First, define a set of functions for managing your popup.

```
function onPopupClose(evt) {
    // 'this' is the popup.
    selectControl.unselect(this.feature);
}
function onFeatureSelect(evt) {
    feature = evt.feature;
    popup = new OpenLayers.Popup.FramedCloud("featurePopup",
                            feature.geometry.getBounds().getCenterLonLat(),
                            new OpenLayers.Size(100,100),
                            "<h2>"+feature.attributes.title + "</h2>" +
                            feature.attributes.description,
                            null, true, onPopupClose);
    feature.popup = popup;
    popup.feature = feature;
    map.addPopup(popup);
}
function onFeatureUnselect(evt) {
    feature = evt.feature;
    if (feature.popup) {
        popup.feature = null;
        map.removePopup(feature.popup);
        feature.popup.destroy();
        feature.popup = null;
    }
}
```

Next, we define two event handlers on the layer to call these functions appropriately. We use the layer definition from above, and assume that the layer has been added to the map.

```
layer.events.on({
    'featureselected': onFeatureSelect,
    'featureunselected': onFeatureUnselect
});
```

Combining these two sections of code will cause the map to open a popup any time the feature is selected, and close the popup when the feature is unselected or the close button is pressed.

The HTML in the fourth argument to the FramedCloud constructor is based on the type of data you are parsing. This example is based around the Text Layer, but you can do the same with a KML layer by changing the ‘title’ to ‘name’. The GeoRSS Layer could use the `feature.attributes.link` property in addition, to create a link to the feature.

It is worth noting that this content – passed to the FramedPopup constructor – is set using innerHTML, and as such, is subject to XSS attacks if the content in question is untrusted. If you can not trust the content in your source files, you should employ some type of stripping to remove possibly malicious content before setting the popup content to protect your site from XSS attacks.

Once you’ve done this, you can customize the behavior of your layer to your heart’s content. Change the layout of your popup HTML, change the type of popup, or change the click behavior to instead open a new window – it’s all possible, and simple, with the functionality provided by the vector layers and SelectFeatureControl.

Styling

This OpenLayers Styles framework is the way to control the styling of features attached to vector

layers in OpenLayers, such as points, lines, and polygons. It provides capabilities which correspond to the features provided by standards like SLD, allowing the use of advanced feature styling with properties and rules.

Style Classes

When a feature is added to a map, its style information can come from one of three sources:

- A symbolizer hash set directly on the feature. This is unusual, but may occur when parsing remote data which embeds style information at the feature level, like some KML content.
- A symbolizer hash attached to a layer as layer.style.
- A Style or StyleMap object attached to the layer as layer.styleMap.

A symbolizer hash is a simple JavaScript object, with key/value pairs describing the values to be used.

```
{
  'strokeWidth': 5,
  'strokeColor': '#ff0000'
}
```

A StyleMap object is a more descriptive element, which allows for the use of advanced feature-based styling. It uses the OpenLayers.Style and OpenLayers.StyleMap objects. A style map has support for different ‘render intents’: ways in which a feature should be drawn. OpenLayers uses three different render intents internally:

- ‘default’: Used in most cases
- ‘select’: Used when a feature is selected
- ‘temporary’: Used while ‘sketching’ a feature.

When a feature is drawn, it is possible to pass one of these renderIntents to the Layer’s ‘drawFeature’ function, or to set the ‘renderIntent’ property of the feature to one of these three intents.

Each renderIntent in the StyleMap has an OpenLayers Style object associated with it.

OpenLayers Style objects are descriptions of the way that features should be rendered. When a feature is added to a layer, the layer combines the style property with the feature to create a ‘symbolizer’ – described above as a set of style properties that will be used when rendering the layer. (Internally, this is done via the *createSymbolizer* function.)

Attribute Replacement Syntax

The most common way of accessing attributes of the feature when creating a style is through the attribute replacement syntax. By using style values like `${varname}` in your style, OpenLayers can replace these strings with attributes of your feature. For example, if you had a GeoJSON file where each feature had a `thumbnail` property describing an image to use, you might use something like:

```
var s = new OpenLayers.Style({
  'pointRadius': 10,
  'externalGraphic': '${thumbnail}'
});
```

In this way, the style can contain rendering information which is dependant on the feature.

StyleMap

Simple OpenLayers Style objects are instantiated by passing a symbolizer hash to the constructor of the style. This Style can then be passed to a StyleMap constructor:

```
new OpenLayers.StyleMap(s);
```

As a convenience, you can also create a StyleMap by passing a symbolizer hash directly. (The StyleMap will then create the Style object for you.)

```
new OpenLayers.StyleMap({'pointRadius': 10,
                         'externalGraphic': '${thumbnail}'});
```

In almost all simple cases, this will be sufficient. By default, passing in a Style object will cause all the render intents of the StyleMap to be set to the same style. If you wish to have different Style objects for the different render intents, instead pass a hash of Style objects or symbolizer hashes:

```
var defaultStyle = new OpenLayers.Style({
  'pointRadius': 10,
  'externalGraphic': '${thumbnail}'
});

var selectStyle = new OpenLayers.Style({
  'pointRadius': 20
});

new OpenLayers.StyleMap({'default': defaultStyle,
                         'select': selectStyle});
```

The “default” intent has a special role: if the extendDefault property of the StyleMap is set to true (default), symbolizers calculated for other render intents will extend the symbolizer calcualated for the “default” intent. So if we want selected features just to have a different size or color, we only have to set a single property (in this example: pointRadius).

Using Style Objects

Once you have created a style object, it needs to be passed to a layer in order for it to be used. The StyleMap object should be passed as the layer’s ‘styleMap’ option:

```
var styleMap = new OpenLayers.StyleMap({pointRadius: 10,
                         externalGraphic: '${thumbnail}'});
var l = new OpenLayers.Layer.Vector("Vector Layer",
                                    {styleMap: styleMap});
```

Features added to the layer will then be styled according to the StyleMap.

Rule Based Styling

In addition to simple attribute based styles, OpenLayers also has support for rule-based styling – where a property on the feature can determine the other styles in use. For example, if you have an attribute, ‘size’, which is ‘large’ or ‘small’, which determines the desired size of the icon, you can use that property to control the pointRadius.

OpenLayers provides two different ways to do this. Many simple cases can be solved with the addUniqueValueRules convenience function, while more complex cases require creating your own rules.

addUniqueValueRules

In order to use addUniqueValueRules, you first create a StyleMap with the ‘shared’ properties of the style. As in the case above, we imagine that we are loading features with URLs in the ‘thumbnail’ attribute:

```
var styleMap = new OpenLayers.StyleMap({externalGraphic: '${thumbnail}'});
```

We then create a mapping between feature attribute value and symbolizer value, then add rules to the default symbolizer that check for the “size” attribute and apply the symbolizer defined in that variable:

```
var lookup = {
  "small": {pointRadius: 10},
  "large": {pointRadius: 30}
}

styleMap.addUniqueValueRules("default", "size", lookup);
```

This adds rules to the Styles in the ‘default’ renderIntent, stating that the Style should change the pointRadius based on the ‘size’ attribute of the feature.

The symbolizers inside rules do not have to be complete symbolizers, because they extend the default symbolizer passed with the constructor of OpenLayers.Style or OpenLayers.StyleMap.

The Unique Values example demonstrates the use of addUniqueValueRules.

Custom Rules

OpenLayers supports many types of Rules and Filters. The addUniqueValueRules function creats Comparison rules, with the EQUAL_TO operator. We can also create rules that allow us to apply styles based on whether a value is greater than or less than a value, or whether it matches a certain string, and more.

Here, we demonstrate how to create filters using the LESS_THAN and GREATER_THAN_OR_EQUAL_TO operators:

```
var style = new OpenLayers.Style();

var ruleLow = new OpenLayers.Rule({
  filter: new OpenLayers.Filter.Comparison({
      type: OpenLayers.Filter.Comparison.LESS_THAN,
      property: "amount",
      value: 20,
  }),
  symbolizer: {pointRadius: 10, fillColor: "green",
               fillOpacity: 0.5, strokeColor: "black"}
});

var ruleHigh = new OpenLayers.Rule({
  filter: new OpenLayers.Filter.Comparison({
      type: OpenLayers.Filter.Comparison.GREATER_THAN_OR_EQUAL_TO,
      property: "amount",
      value: 20,
  }),
  symbolizer: {pointRadius: 20, fillColor: "red",
               fillOpacity: 0.7, strokeColor: "black"}
});
```

```
style.addRules([ruleLow, ruleHigh]);
```

Each of these rules uses a Comparison filter. There are several types of filters:

- Comparison Filters: Comparison filters take an operator – one of the supported comparison filter types – and one or two values. It then evaluates whether the feature matches the comparison.
- FeatureId Filters: Takes a list of Feature IDs, and evaluates to true if the feature's ID is in the array.
- Logical Filters: Logical filters combine other types of filters together, which allows building more complex rules by concatenating them using boolean operators (AND, OR, NOT). A Logical rule (except NOT) can have child rules.

Every rule can also have a minScaleDenominator and a maxScaleDenominator property. This allows us to specify scale ranges for which the rule should apply. We might e.g. want to show small points at small scales, but image thumbnails at large scales. The result of such rules can be seen in the SLD example: Zooming in one level will turn two lakes into blue. The styles and rules from this example do not come from JavaScript-created style and rule objects, but from a SLD document read in by OpenLayers.Format.SLD.

With SLD, styles are grouped into named layers (NamedLayer), which again have a set of named user styles (UserStyle). This is the reason why the Style object also has layerName and name properties. For each named layer, there can be a default style. This is marked by setting the isDefault property of the Style object to true.

Style Properties

The properties that you can use for styling are:

- fillColor

  Default is `#ee9900`. This is the color used for filling in Polygons. It is also used in the center of marks for points: the interior color of circles or other shapes. It is not used if an externalGraphic is applied to a point.

- fillOpacity:

  Default is `0.4`. This is the opacity used for filling in Polygons. It is also used in the center of marks for points: the interior color of circles or other shapes. It is not used if an externalGraphic is applied to a point.

- strokeColor

  Default is `#ee9900`. This is color of the line on features. On polygons and point marks, it is used as an outline to the feature. On lines, this is the representation of the feature.

- strokeOpacity

  Default is `1` This is opacity of the line on features. On polygons and point marks, it is used as an outline to the feature. On lines, this is the representation of the feature.

- strokeWidth

Default is `1` This is width of the line on features. On polygons and point marks, it is used as an outline to the feature. On lines, this is the representation of the feature.

- strokeLinecap

  Default is `round`. Options are `butt`, `round`, `square`. This property is similar to the *SVG stroke-linecap* property. It determines what the end of lines should look like. See the SVG link for image examples.

- strokeDashstyle

  Default is `solid`. Options are:

  - `dot`
  - `dash`
  - `dashdot`
  - `longdash`
  - `longdashdot`
  - `solid`

- pointRadius

  Default is `6`.

- pointerEvents:

  Default is `visiblePainted`. Only used by the SVG Renderer. See SVG pointer-events definition for more.

- cursor

  Cursor used when mouse is over the feature. Default is an empty string, which inherits from parent elements.

- externalGraphic

  An external image to be used to represent a point.

- graphicWidth, graphicHeight

  These properties define the height and width of an externalGraphic. This is an alternative to the pointRadius symbolizer property to be used when your graphic has different sizes in the X and Y direction.

- graphicOpacity

  Opacity of an external graphic.

- graphicXOffset, graphicYOffset

  Where the ‘center’ of an externalGraphic should be.

- rotation

  The rotation angle in degrees clockwise for a point symbolizer.

- graphicName

  Name of a type of symbol to be used for a point mark.

- display

  Can be set to ‘none’ to hide features from rendering.

Deploying

OpenLayers comes with pre-configured examples out of the box: simply download a release of OpenLayers, and you get a full set of easy to use examples. However, these examples are designed to be used for development. When you’re ready to deploy your application, you want a highly optimized OpenLayers distribution, to limit bandwidth and loading time.

Single File Build

OpenLayers has two different types of usage: Single File, where all Javascript code is compiled into a single file, OpenLayers.js, and the development version, where Javascript files are all loaded at application start time. The single file build takes a set of OpenLayers Javascript files, orders them according to dependancies described in the files, and compresses the resulting file, using the jsmin compression library.

Building a single file version of the OpenLayers library changes the behavior of the library slightly: by default, the development version of OpenLayers expects to live in a directory called “lib”, and expects that images and CSS live in the directory above the OpenLayers.js file:

```
img/pan-hand.png
theme/default/style.css
lib/OpenLayers.js
lib/OpenLayers/Map.js
...
```

However, when deploying a single file build of OpenLayers, it is expected that the library will instead be at the same level as the theme and img directories:

```
OpenLayers.js
theme/default/style.css
img/pan-hand.png
...
```

Building the Single File Build

The single file build tools are deployed with an OpenLayers release in the ‘build’ directory. The tools require Python in order to build.

On Linux and other similar operating systems, given that OpenLayers is stored in the ‘openlayers’ directory, a single file build could be created by issuing the following commands:

```
cd openlayers/build
./build.py
```

This would create a file in the build directory called “OpenLayers.js”, which contains all the library code for your single file build of OpenLayers.

In Windows, from the Start Menu, select Run. Copy the path to build.py from the address bar of the Explorer Window into the text box and then add the name of the configuration file (or blank for the default):

C:DownloadsOpenLayers-2.6buildbuild.py lite

Custom Build Profiles

In order to optimize the end-user's experience, the OpenLayers distribution includes tools which allow you to build your own single file version of the code. This code uses a configuration file to choose which files should be included in the build: In this way, for production use, you can remove classes from your OpenLayers JavaScript library file which are not used in your application.

OpenLayers ships with two standard configurations to create a single file version:

full:

This is the full build with all files.

lite:

This file includes a small subset of OpenLayers code, designed to be integrated into another application. It includes only the Layer types neccesary to create tiled or untiled WMS, and does not include any Controls. This is the result of what was at the time called "Webmap.js" at the FOSS4G 2006 Web Mapping BOF.

Profiles are simple to create. You can start by copying library.cfg or lite.cfg to something else, e.g. myversion.cfg in the build directory.

The start of any build profile must include the same [first] section used in the lite.cfg file:

```
[first]
OpenLayers/SingleFile.js
OpenLayers.js
OpenLayers/BaseTypes.js
OpenLayers/BaseTypes/Class.js
OpenLayers/Util.js
```

These files are required for the OpenLayers build to function.

Once you have included these files, you should add more files to the '[include]' section of the file. The files listed here should be the list of files containing any class you use in your application. You can typically find these classes by looking through your code for any cases where 'new OpenLayers.ClassName()' is used.

Taking the 'lite.html' example, we see that there are two 'new' statements in the file: one for the OpenLayers.Map class, and one for the OpenLayers.Layer.WMS class. We then add the corresponding files to our include section:

```
[include]
OpenLayers/Map.js
OpenLayers/Layer/WMS.js
```

Once we have done that, we can build our profile by adding the profile name to the end of our ealier build command:

```
./build.py myversion
```

This will create a much smaller OpenLayers version, suitable for limited applications.

Almost all applications can benefit from a custom build profile. OpenLayers supports many different layer types, but most applications will only use one or two, and many applications do not need the full support of many of the features in OpenLayers. In order to limit your download time and library size, building a custom profile is highly recommended before deploying an OpenLayers application: it can help shrink the size of your library by a factor of five over using the full library.

Deploying Files

In order to deploy OpenLayers, there are several different pieces that must be deployed.

OpenLayers.js

: The library. This provides the JavaScript code for your application to use.

theme directory

: The theme directory contains CSS and image files for newer controls, whose styling and positioning is controlled entirely by CSS.

img directory

: This directory provides images to be used for some controls, like the PanZoom control, which do not use CSS for styling.

As described above, when deploying these files with a single file OpenLayers build, they should all live in the same directory: this allows OpenLayers to properly find and include them.

Requesting Remote Data

There are a number of ways to get data from the server to the client. Setting image, script, and stylesheet sources can be done at any time to request new data without refreshing an entire document. These types of requests can be made to any origin. Another way to retrieve data from a server is to update the location of a document in a frame (iframe or otherwise). These types of requests can also be made to any origin. However, the code running in the original document is restricted to reading data in documents that come from the same origin. This means if your application is served from http://example.com, your code may load a document in a frame from any other origin (say http://example.net), but your code can only access data in that document if it was served via the same protocol (http), from the same domain (example.com), and on the same port (likely 80 in this case).

The above types of request are useful because they are asynchonous. That is, the user can continue to view and interact with the original page while additional data is being requested. A more common way to request data asynchronously is to use the XMLHttpRequst object. With an XMLHttpRequest object, you can open a conection to a server and send or receive data via HTTP (or HTTPS). XMLHttpRequest objects are a bit awkward to use and are unfortunately not supported in all browsers. OpenLayers provides a cross-browser XMLHttpRequest function and wraps it in some convenient `OpenLayers.Request` methods.

In general, all communication initiated by `OpenLayers.Request` methods is restricted to the same origin policy: requests may only be issued with the same protocol, to the same domain, and through the same port as the document the code is running from. The `OpenLayers.Request` methods allow you to access data asynchronously or synchronously (synchronous requests will lock the UI while the request is pending).

Note

Though you may read about cross-domain Ajax, unless a user has specifically configured their browser's security settings, the same origin policy will apply to requests with XMLHttpRequest. The "cross-domain" functionality is typically achieved by either setting up a proxy (on the same origin) that passes on all communication to a remote server or by requesting data without the XMLHttpRequest object (in a script tag for example). One nuance of the same origin policy is that code running on a page from one domain may set the `document.domain` property to a suffix of the original domain. This means that code running on a document from sub.example.com may request data from example.com by setting `document.domain` to example.com. A document from example.com, however, cannot prefix the domain property to request data from a sub domain.

The `OpenLayers.Request` methods correspond to the common HTTP verbs: GET, POST, PUT, DELETE, HEAD, and OPTIONS. See the Request API documentation for a description of each of these methods. The short examples below demonstrate the use of these methods under a variety of conditions.

Example usage

1. Issue a GET request and deal with the response.

```
function handler(request) {
    // if the response was XML, try the parsed doc
    alert(request.responseXML);
    // otherwise, you've got the response text
    alert(request.responseText);
    // and don't forget you've got status codes
    alert(request.status);
    // and of course you can get headers
    alert(request.getAllResponseHeaders());
    // etc.
}

var request = OpenLayers.Request.GET({
    url: "http://host/path",
    callback: handler
});
```

2. Issue a GET request with a query string based on key:value pairs.

```
function handler(request) {
    // do something with the response
    alert(request.responseXML);
}

var request = OpenLayers.Request.GET({
    url: "http://host/path",
    params: {somekey: "some value & this will be encoded properly"},
    callback: handler
});
```

3. Issue a GET request where the handler is a public method on some object.

```
// assuming obj was constructed earlier
obj.handler = function(request) {
    this.doSomething(request);
}

var request = OpenLayers.Request.GET({
    url: "http://host/path",
    callback: obj.handler,
    scope: obj
});
```

4. Issue a synchronous GET request.

```
var request = OpenLayers.Request.GET({
    url: "http://host/path",
    async: false
});
// do something with the response
alert(request.responseXML);
```

5. Issue a POST request with some data.

```
// assuming you already know how to create your handler
var request = OpenLayers.Request.POST({
    url: "http://host/path",
    data: "my data to post",
    callback: handler
});
```

6. Issue a POST request with a custom content type (application/xml is default).

```
// again assuming you have a handler
var request = OpenLayers.Request.POST({
    url: "http://host/path",
    data: "this is text not xml!",
    headers: {
        "Content-Type": "text/plain"
    },
    callback: handler
});
```

7. Issue a POST request with form-encoded data.

```
var request = OpenLayers.Request.POST({
    url: "http://host/path",
    data: OpenLayers.Util.getParameterString({foo: "bar"}),
    headers: {
        "Content-Type": "application/x-www-form-urlencoded"
    },
    callback: handler
})
```

8. Issue a GET request and then abort it.

```
var request = OpenLayers.Request.GET(); // dumb, but possible
request.abort();
```

9. Deal with the many ways that a request can “fail.”

```
function handler(request) {
    // the server could report an error
    if(request.status == 500) {
        // do something to calm the user
    }
    // the server could say you sent too much stuff
    if(request.status == 413) {
        // tell the user to trim their request a bit
    }
    // the browser's parser may have failed
    if(!request.responseXML) {
        // get ready for parsing by hand
    }
    // etc.
}
// issue a request as above
```

10. Issue DELETE, PUT, HEAD, and OPTIONS requests.

```
// handlers defined elsewhere

var deleteRequest = OpenLayers.Request.DELETE({
    url: "http://host/path",
    callback: deleteHandler
});

var putRequest = OpenLayers.Request.PUT({
    url: "http://host/path",
    callback: putHandler
});

var headRequest = OpenLayers.Request.HEAD({
    url: "http://host/path",
    callback: headHandler
});

var optionsRequest = OpenLayers.Request.OPTIONS({
    url: "http://host/path",
    callback: optionsHandler
});
```

11. (Rare) Issue a GET request using a proxy other than the one specified in OpenLayers.ProxyHost (same origin policy applies).

```
// handler defined elsewhere
var request == OpenLayers.Request.GET({
    url: "http://host/path",
    params: {somekey: "some value"},
    proxy: "http://sameorigin/proxy?url=" // defaults to OpenLayers.ProxyHost
});
```

Spherical Mercator

This document describes the Spherical Mercator projection, what it is, and when you should use it. It includes some background information, demonstration of using the code with just a commercial layer, and how to add a WMS over the top of that layer, and how to reproject coordinates within OpenLayers so that you can reproject coordinates inside of OpenLayers. It is expected that readers

of this tutorial will have a basic understanding of reprojection and a basic understanding of OpenLayers.

What is Spherical Mercator?

Spherical Mercator is a de facto term used inside the OpenLayers community – and also the other existing Open Source GIS community – to describe the projection used by Google Maps, Microsoft Virtual Earth, Yahoo Maps, and other commercial API providers.

This term is used to refer to the fact that these providers use a Mercator projection which treats the earth as a sphere, rather than a projection which treats the earth as an ellipsoid. This affects calculations done based on treating the map as a flat plane, and is therefore important to be aware of when working with these map providers.

In order to properly overlay data on top of the maps provided by the commerical API providers, it is neccesary to use this projection. This applies primarily to displaying raster tiles over the commercial API layers – such as TMS, WMS, or other similar tiles.

In order to work well with the existing commercial APIs, many users who create data designed for use within Google Maps will also use this projection. One prime example is OpenStreetMap, whose raster map tiles are all projected into the ‘spherical mercator’ projection.

Projections in GIS are commonly referred to by their “EPSG” codes, identifiers managed by the European Petroleum Survey Group. One common identifier is “EPSG:4326”, which describes maps where latitude and longitude are treated as X/Y values. Spherical Mercator has an official designation of EPSG:3785. However, before this was established, a large amount of software used the identifier EPSG:900913. This is an unofficial code, but is still the commonly used code in OpenLayers. Any time you see the string “EPSG:4326”, you can assume it describes latitude/longitude coordinates. Any time you see the string “EPSG:900913”, it will be describing coordinates in meters in x/y.

First Map

The first thing to do with the Spherical Mercator projection is to create a map using the projection. This map will be based on the Microsoft Virtual Earth API. The following HTML template will be used for the map.

```
<html>
<head>
  <title>OpenLayers Example</title>
    <script src='http://dev.virtualearth.net/mapcontrol/mapcontrol.ashx?
v=6.1'></script>
    <script src="http://openlayers.org/api/OpenLayers.js"></script>
    </head>
    <body>
      <div style="width:100%; height:100%" id="map"></div>
      <script defer='defer' type='text/javascript'>
        // Code goes here
      </script>
    </body>
</html>
```

Ex. 1: HTML Template

The next step is to add the default Microsoft Virtual Earth layer as a base layer to the map.

```
var map = new OpenLayers.Map('map');
var layer = new OpenLayers.Layer.VirtualEarth("Virtual Earth",
 {
     sphericalMercator: true,
     maxExtent: new OpenLayers.Bounds(-20037508.34,-
20037508.34,20037508.34,20037508.34)
 });
map.addLayer(layer);
map.zoomToMaxExtent();
```

This creates a map. However, once you have this map, there is something very important to be aware of: the coordinates that you use in setCenter are not longitude and latitude! Instead, they are in projected units – meters, in this case. This map will let you drag around, but without understanding a bit more about spherical mercator, it will be difficult to do anything more with it.

This map has a set of assumptions about the maxResolution of the map. Specifically, most spherical mercator maps use an extent of the world from -180 to 180 longitude, and from -85.0511 to 85.0511 latitude. Because the mercator projection stretches to infinity as you approach the poles, a cutoff in the north-south direction is required, and this particular cutoff results in a perfect square of projected meters. As you can see from the maxExtent parameter sent in the constructor of the layer, the coordinates stretch from -20037508.34 to 20037508.34 in each direction.

The maxResolution of the map defaults to fitting this extent into 256 pixels, resulting in a maxResolution of 156543.0339. This is handled internally by the layer, and does not need to be set in the layer options.

If you are using a standalone WMS or TMS layer with spherical mercator, you will need to specify the maxResolution property of the layer, in addition to defining the maxExtent as demonstrated here.

Working with Projected Coordinates

Thankfully, OpenLayers now provides tools to help you reproject your data on the client side. This makes it possible to transform coordinates from Longitude/Latitude to Spherical Mercator as part of your normal operation. First, we will transform coordinates for use within the setCenter and other calls. Then we will show how to use the displayProjection option on the map to modify the display of coordinate data to take into account the projection of the base map.

Reprojecting Points, Bounds

To do this, first create a projection object for your default projection. The standard latitude/longitude projection string is “EPSG:4326” – this is latitude/longitude based on the WGS84 datum. (If your data lines up correctly on Google Maps, this is what you have.)

You will then be creating an object to hold your coordinates, and transforming it.

```
var proj = new OpenLayers.Projection("EPSG:4326");
var point = new OpenLayers.LonLat(-71, 42);
point.transform(proj, map.getProjectionObject());
```

The point is now projected into the spherical mercator projection, and you can pass it to the setCenter method on the map:

```
map.setCenter(point);
```

This can also be done directly in the setCenter call:

```
var proj = new OpenLayers.Projection("EPSG:4326");
var point = new OpenLayers.LonLat(-71, 42);
map.setCenter(point.transform(proj, map.getProjectionObject()));
```

In this way, you can use latitude/longitude coordinates to choosing a center for your map.

You can use the same technique for reprojecting OpenLayers.Bounds objects: simply call the transfrom method on your Bounds object.

```
var bounds = new OpenLayers.Bounds(-74.047185, 40.679648, -73.907005, 40.882078)
bounds.transform(proj, map.getProjectionObject());
```

Transformations take place on the existing object, so there is no need to assign a new variable.

Reprojecting Geometries

Geometry objects have the same transform method as LonLat and Bounds objects. This means that any geometry object you create in your application code must be transformed by calling the transform method on it before you add it to a layer, and any geometry objects that you take from a layer and wish to use will need to be transformed before further use.

Because all transforms are in place, once you have added a geometry to a layer, you should not call transform on the geometry directly: instead, you should transform a *clone* of the geometry:

```
var feature = vector_layer.features[0];
var geometry = feature.geometry.clone();
geometry.transform(layerProj, targetProj);
```

Reprojecting Vector Data

When creating projected maps, it is possible to reproject vector data onto a basemap. To do so, you must simply set the projection of your vector data correctly, and ensure that your map projection is correct.

```
var map = new OpenLayers.Map("map", {
  projection: new OpenLayers.Projection("EPSG:900913")
});
var myBaseLayer = new OpenLayers.Layer.Google("Google",
              {'sphericalMercator': true,
               'maxExtent': new OpenLayers.Bounds(-20037508.34,-
20037508.34,20037508.34,20037508.34)
              });
map.addLayer(myBaseLayer);
var myGML = new OpenLayers.Layer.GML("GML", "mygml.gml", {
  projection: new OpenLayers.Projection("EPSG:4326")
});
map.addLayer(myGML);
```

Note that you can also use this setup to load any format of vector data which OpenLayers supports, including WKT, GeoJSON, KML and others. Simply specify the format option of the GML layer.

```
var geojson = new OpenLayers.Layer.GML("GeoJSON", "geo.json", {
  projection: new OpenLayers.Projection("EPSG:4326"),
  format: OpenLayers.Format.GeoJSON
});
```

```
map.addLayer(geojson);
```

Note that even if you set the projection object on a layer, if you are adding features to the layer manually (via layer.addFeatures), they *must* be transformed before adding to the layer. OpenLayers will only transform the projection of geometries that are created internally to the library, to prevent duplicating projection work.

Serializing Projected Data

The way to serialize vector data in OpenLayers is to take a collection of data from a vector layer and pass it to a Format class to write out data. However, in the case of a projected map, the data that you get from this will be projected. To reproject the data when converting, you should pass the internal and external projection to the format class, then use that format to write out your data.

```
var format = new OpenLayers.Format.GeoJSON({
  'internalProjection': new OpenLayers.Projection("EPSG:900913"),
  'externalProjection': new OpenLayers.Projection("EPSG:4326")
});
var jsonstring = format.write(vector_layer.features);
```

Display Projection on Controls

Several controls display map coordinates to the user, either directly or built into their links. The MousePosition and Permalink control (and its companion control, ArgParser) both use coordinates which match the internal projection of the map – which in the case of Spherical Mercator layers is projected. To prevent user confusion, OpenLayers allows one to set a ‘display’ projection. When these controls are used, transformation is made from the map projection to the display projection.

To use this option, when creating your map, you should specify the projection and displayProjection options. Once this is done, the controls will automatically pick up this option from the map.

```
var map = new OpenLayers.Map("map", {
  projection: new OpenLayers.Projection("EPSG:900913"),
  displayProjection: new OpenLayers.Projection("EPSG:4326")
});
map.addControl(new OpenLayers.Control.Permalink());
map.addControl(new OpenLayers.Control.MousePosition());
```

You can then add your layer as normal.

Creating Spherical Mercator Raster Images

One of the reasons that the Spherical Mercator projection is so important is that it is the only projection which will allow for overlaying image data on top of commercial layers like Google Maps correctly. When using raster images, in the browser, it is not possible to reproject the images in the same way it might be in a ‘thick’ GIS client. Instead, all images must be in the same projection.

How to create Spherical Mercator projected tiles depends on the software you are using to generate your images. MapServer is covered in this document.

MapServer

MapServer uses proj.4 for its reprojection support. In order to enable reprojection to Spherical Mercator in MapServer, you must add the definition for the projection to your proj.4 data directories.

On Linux systems, edit the /usr/share/proj/epsg file. At the bottom of that file, add the line:

```
<900913> +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0
+k=1.0 +units=m +nadgrids=@null +no_defs
```

After you do this, you must add the projection to your wms_srs metdadata in your map file:

```
map
  web
    metadata
      wms_srs "EPSG:4326 EPSG:900913"
    end
  end
  # Layers go here
end
```

This will allow you to request tiles from your MapServer WMS server in the Spherical Mercator projection, which will align with commercial provider data in OpenLayers.

```
var options = {
    projection: new OpenLayers.Projection("EPSG:900913"),
    units: "m",
    maxResolution: 156543.0339,
    maxExtent: new OpenLayers.Bounds(-20037508.34, -20037508.34,
                                     20037508.34, 20037508.34)
};

map = new OpenLayers.Map('map', options);

// create Google Mercator layers
var gmap = new OpenLayers.Layer.Google(
    "Google Streets",
    {'sphericalMercator': true,
     'maxExtent': new OpenLayers.Bounds(-20037508.34,-
20037508.34,20037508.34,20037508.34)
    }
);

// create WMS layer
var wms = new OpenLayers.Layer.WMS(
    "World Map",
    "http://labs.metacarta.com/wms/vmap0",
    {'layers': 'basic', 'transparent': true}
);

map.addLayers(gmap, wms);
```

WMS layers automatically inherit the projection from the base layer of a map, so there is no need to set the projection option on the layer.

GeoServer

Current versions of GeoServer have support for EPSG:900913 built in, so there is no need to add

additional projection data. Simply add your GeoServer layer as a WMS and add it to the map.

Custom Tiles

Another common use case for spherical mercator maps is to load custom tiles. Many custom tile sets are created using the same projection as Google Maps, usually with the same z/x/y scheme for accessing tiles.

If you have tiles which are set up according to the 'Google' tile schema – that is, based on x,y,z and starting in the upper left corner of the world – you can load these tiles with the TMS layer with a slightly modified get_url function. (Note that in the past there was a 'LikeGoogle' layer in SVN – this is the appropriate replacement for that code/functionality.)

First, define a getURL function that you want to use: it should accept a bounds as an argument, and will look something like this:

```
function get_my_url (bounds) {
    var res = this.map.getResolution();
    var x = Math.round ((bounds.left - this.maxExtent.left) / (res *
this.tileSize.w));
    var y = Math.round ((this.maxExtent.top - bounds.top) / (res *
this.tileSize.h));
    var z = this.map.getZoom();

    var path = z + "/" + x + "/" + y + "." + this.type;
    var url = this.url;
    if (url instanceof Array) {
        url = this.selectUrl(path, url);
    }
    return url + path;

}
```

Then, when creating your TMS layer, you pass in an option to tell the layer what your custom tile loading function is:

```
new OpenLayers.Layer.TMS("Name",
                      "http://example.com/",
                      { 'type':'png', 'getURL':get_my_url });
```

This will cause the getURL function to be overridden by your function, thus requesting your inverted google-like tiles instead of standard TMS tiles.

When doing this, your map options should contain the maxExtent and maxResolution that are used with Google Maps:

```
new OpenLayers.Map("map", {
    maxExtent: new OpenLayers.Bounds(-20037508.34,-
20037508.34,20037508.34,20037508.34),
    numZoomLevels:18,
    maxResolution:156543.0339,
    units:'m',
    projection: "EPSG:900913",
    displayProjection: new OpenLayers.Projection("EPSG:4326")
});
```

As describe above, when using this layer, you will interact with the map in projected coordinates.