Developing OGC Compliant Web Applications with GeoExt

Release 1.1

OpenGeo and Prodevelop

October 11, 2016

Welcome to the **workshop "Developing OGC Compliant Web Applications with GeoExt"**. This workshop is designed to introduce GeoExt as a web mapping frontend to OGC Web Services (OWS).

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

The instructors of the GeoExt workshop are:

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

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

The following modules will be covered in this workshop:

GeoExt Basics Learn how to create a fullscreen map with a WMS layer.

WMS and the GeoExt LayerStore Create a WMS browser using GetCapabilities, GetMap, GetFeatureInfo and GetLegendGraphic.

WFS Made Easy with GeoExt Create a WFS-T editor with a synchronized map and table view.

CHAPTER 1

GeoExt Basics

GeoExt is a young, rapidly-developing library for building rich, web-based GIS applications. The library is built upon Ext JS and OpenLayers. The former provides UI components for building web applications along with solid underlying data components, the latter is the de facto standard for dynamic web mapping.

GeoExt provides mapping related UI components. It also unifies access to information from OGC services, OpenLayers objects and arbitrary remote data sources. This allows for easy presentation of geospatial information in a wide choice of widgets, ranging from combo boxes or grids to maps and trees. It has a friendly API, reduces the number of lines of code required, and results in engaging and responsive mapping applications.

This module introduces fundamental GeoExt concepts for creating a map. You will create a map, dissect your map to understand the parts, and get links to additional learning resources.

1.1 Creating a Map Window

In GeoExt, following the conventions of the underlying Ext JS framework, a map is wrapped into an Ext.Panel. The map is an OpenLayers.Map object.

It is important to understand that Ext JS encourages a web application paradigm, as opposed to a web page paradigm. This means that we won't create markup, so the basic ingredients of our application will be:

- a *minimal html document* to include JavaScript and CSS resources,
- *JavaScript code* for application initialization,
- *JavaScript code that builds the user interface*,
- "Glue" JavaScript code that makes the pieces work together. We don't have any in this basic example, so we will be learning about it later.

1.1.1 Working Example

Let's take a look at a fully working example of a simple GeoExt application:

```
<html>
    <head>
        <title>GeoExt Workshop Application</title>
        <link rel="stylesheet" type="text/css" href="ext/resources/css/ext-all.css">
        <script type="text/javascript" src="ext/adapter/ext/ext-base.js"></script>
        <script type="text/javascript" src="ext/ext-all.js"></script>
        <script src="openlayers/lib/OpenLayers.js"></script>
        <script type="text/javascript" src="geoext/lib/GeoExt.js"></script>

        <script type="text/javascript">
        Ext.BLANK_IMAGE_URL = "ext/resources/images/default/s.gif";
        var app, items = [], controls = [];

        Ext.onReady(function() {
            app = new Ext.Viewport({
                layout: "border",
                items: items
            });
        });

        items.push({
            xtype: "gx_mappanel",
            ref: "mapPanel",
            region: "center",
            map: {
                numZoomLevels: 19,
                controls: controls
            },
            extent: OpenLayers.Bounds.fromArray([
                -122.911, 42.291,
                -122.787,42.398
            ]),
            layers: [new OpenLayers.Layer.WMS(
                "Medford",
                "/geoserver/wms",
                {layers: "medford"},
                {isBaseLayer: false}
            )]
        });
        controls.push(
            new OpenLayers.Control.Navigation(),
            new OpenLayers.Control.Attribution(),
            new OpenLayers.Control.PanPanel(),
            new OpenLayers.Control.ZoomPanel()
        );

        </script>
    </head>
    <body>
    </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 application in your web browser: http://localhost:8080/ol_workshop/map.html

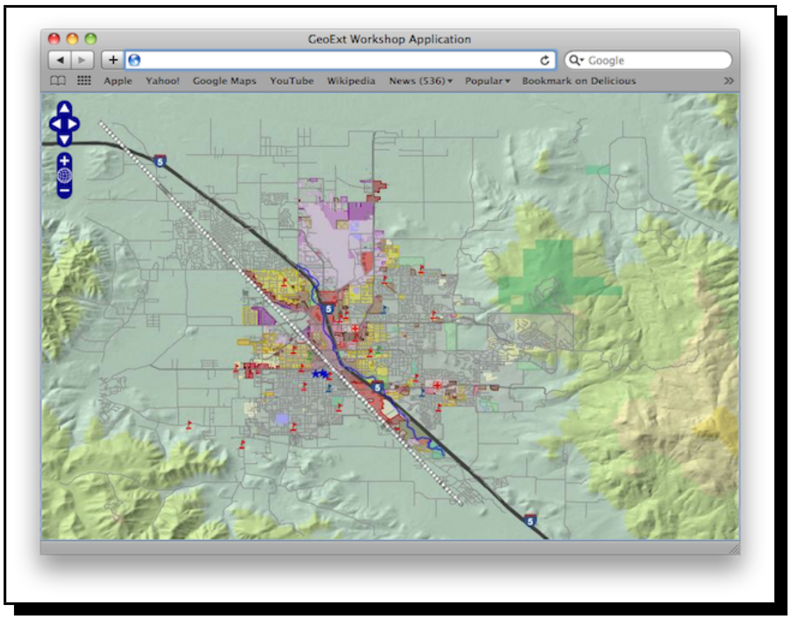


Fig. 1.1: A working map displaying the town of Medford.

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

1.2 Dissecting Your Map Application

As demonstrated in the *previous section*, a map that fills the whole browser viewport is generated by bringing together a *minimal html document*, *application initialization code*, and user interface *configuration objects*. We'll look at each of these parts in a bit more detail.

1.2.1 Minimal HTML Document

Since the mother of all web browser content is still HTML, every web application needs at least a basic HTML document as container. It does not contain human readable markup, so it has an empty body. But it makes sure that all required style and script resources are loaded. These usually go in the document's head:

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

Ext JS can be used standalone, or together with JavaScript frameworks like JQuery. Depending on this environment, an appropriate adapter has to be loaded first. We use Ext JS standalone, so we need the `ext-base.js` adapter. In the second line, we load the main library.

GeoExt not only relies on Ext JS, but also on OpenLayers. So we also have to load OpenLayers. And finally, we can load GeoExt:

```
<script src="openlayers/lib/OpenLayers.js"></script>
<script type="text/javascript" src="geoext/lib/GeoExt.js"></script>
```

Note: When using GeoExt, you also benefit from all the functionality that plain Ext JS and OpenLayers provide. You can add GeoExt to your existing Ext JS and OpenLayers applications without breaking anything.

1.2.2 Application Initialization Code

Application initialization in this context means code that is executed as early as possible.

```
Ext.BLANK_IMAGE_URL = "ext/resources/images/default/s.gif";
var app, items = [], controls = [];

items.push({
    xtype: "gx_mappanel",
    ref: "mapPanel",
    region: "center",
    map: {
        numZoomLevels: 19,
        controls: controls
    },
    extent: OpenLayers.Bounds.fromArray([
        -122.911, 42.291,
        -122.787,42.398
    ]),
    layers: [new OpenLayers.Layer.WMS(
        "Medford",
        "/geoserver/wms?SERVICE=WMS",
        {layers: "medford"},
        {isBaseLayer: false}
    )]
});
controls.push(
    new OpenLayers.Control.Navigation(),
    new OpenLayers.Control.Attribution(),
    new OpenLayers.Control.PanPanel(),
    new OpenLayers.Control.ZoomPanel()
);
```

We start with setting a local URL for the blank image that Ext JS uses frequently, and define some variables. We populate two arrays. `items` is the user interface items of our application, and `controls` is our OpenLayers map controls.

The really interesting part in the snippet above is the one with the `items` that we will add as configuration objects to the viewport. In Ext JS, we find ourselves creating configuration objects instead of writing code for most basic tasks, which usually makes application development easier and faster. The items interact through events and events listeners, the "glue" which we will talk about later.

Before we look at the items in more detail, let's find out how to *add content to our viewport*.

1.2.3 Building the User Interface

We already saw that the `body` of our HTML document is empty. Everything that we see on the web page is added by Ext JS, but for this to work we need to have the DOM of the page ready, so we can append to it. To ensure that we don't write to the DOM too early, Ext provides the `Ext.onReady()` hook.

In our example, the user interface is simple. We just create a new `Ext.Viewport` with a border layout. This allows us to fill the whole browser viewport with our application, and we don't need to add any markup to our page.

```
Ext.onReady(function() {
    app = new Ext.Viewport({
        layout: "border",
        items: items
    });
});
```

The `Ext.Viewport` here uses a "border" layout. It can have items for its `center`, `north`, `east`, `south` and `west` regions, but only the `center` region is mandatory. It takes up all the space that is not used by the other regions, which need to be configured with a `width` or `height`.

Note: To make our workshop application modular, we will be calling `Ext.onReady()` several times as we add functionality. There is no need to do this in a real life application, where all DOM dependent code usually goes into a single `Ext.onReady()` block.

1.2.4 The GeoExt.MapPanel Component

In Ext JS, all constructors of UI components take a single argument, which we will be referring to as "configuration object". Like all JavaScript objects, this configuration object is wrapped in curly braces, and contains `key: value` pairs. Let's have a look at the configuration object for our map:

```
{
    xtype: "gx_mappanel",
    ref: "mapPanel",
    region: "center",
    map: {
        numZoomLevels: 19,
        controls: controls
    },
    extent: OpenLayers.Bounds.fromArray([
        -122.911, 42.291,
        -122.787,42.398
    ]),
    layers: [new OpenLayers.Layer.WMS(
        "Medford",
        "/geoserver/wms?SERVICE=WMS",
        {layers: "medford"},
        {isBaseLayer: false}
    )]
}
```

The first three properties are not specific to GeoExt. The `xtype` tells Ext JS which constructor to send the configuration object to. `ref` defines a reference relative to the container (in this case the `Ext.Viewport` we add this item to). The `region` is the region of the viewport we want to place our map in.

Note: The following two notations are equivalent:

- `new GeoExt.MapPanel({region:  center, extent:  /* ...  */});`
- `{xtype:  "gx_mappanel", region:  center, extent:  /* ...  */});`

Ext JS keeps a registry of available components, called "xtypes". GeoExt adds its components to this registry. To make them distinguishable from others, their names start with the "gx_" prefix. In this context, the `ref` property is also important: it is used to create a reference so we can access the component later more easily.

Using xtypes is useful when loading configurations dynamically with AJAX. In that case, the configuration has to be JSON compliant, and may only contain simple types (numbers, strings and boolean values).

The other properties are specific to the `GeoExt.MapPanel`: Instead of creating an OpenLayers.Map instance, we just configure some configuration options for the map in the `map` option. `extent` sets the initial extent of the map, and `layers` the initial set of layers. For our simple map, we just want to show a single WMS layer. As in plain OpenLayers, we do this by instantiating an OpenLayers.Layer.WMS object. The only difference here is that we configure the WMS layer with the `{isBaseLayer:  false}` option. This is not strictly necessary now, but when we add a layer tree later, we want to see the tree node for this layer rendered with a checkbox, not with a radio button.

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

1.3 GeoExt Resources

The GeoExt 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, newcomers may find it a challenge to get started from scratch.

1.3.1 Learn by Example

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

- http://geoext.org/examples.html

In addition, the Ext JS and OpenLayers examples are a valuable knowledge base, especially if you are getting started with GeoExt and have not used Ext JS or OpenLayers before.

- http://dev.sencha.com/deploy/ext-3.4.0/examples/
- http://openlayers.org/dev/examples/

1.3.2 Browse the Documentation

For further information on specific topics, browse the GeoExt documentation. Especially the tutorials and the introduction to core concepts may be useful for newcomers.

- http://geoext.org/docs.html (for the latest release)
- http://dev.geoext.org/docs/docs.html (for the latest nightly build)

1.3.3 Find the API Reference

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

- http://geoext.org/lib/ (for the latest release)
- http://dev.geoext.org/docs/lib/ (for the latest nightly build)

The GeoExt API Reference links to Ext JS and OpenLayers API docs for further reference. The root of these can be found here:

- http://dev.sencha.com/deploy/ext-3.4.0/docs/
- http://dev.openlayers.org/apidocs/

1.3.4 Join the Community

GeoExt 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://www.geoext.org/cgi-bin/mailman/listinfo/users (if you are a user of the GeoExt library)

- Developers list http://www.geoext.org/cgi-bin/mailman/listinfo/dev (if you want to contribute to the development of the GeoExt library)

CHAPTER 2

WMS and the GeoExt LayerStore

In GeoExt, map layers, features of vector layers and even the zoom levels of a map are accessible like any remote Ext JS data source. Read by an Ext.data.DataReader, data records are made available to an Ext.data.Store. Depending on the Reader used, not only OpenLayers objects, but also remote OGC services can be accessed.

This module introduces GeoExt's WMSCapabilitiesStore, and shows how it can easily be used to populate grids and trees. You will create a grid view of layers from a WMS GetCapabilities request, add a tree view to manage the map panel's layers, add a legend using WMS GetLegendGraphic, and explore map features with a WMS GetFeatureInfo popup.

2.1 Creating a Grid View of WMS Capabilities

The GetCapabilities request is usually the first thing we do when we establish a connection to a WMS service. It returns a list of available layers, styles and formats, along with metadata like abstracts and attribution.

2.1.1 Configuring a Grid View

In this exercise, we will create a grid, configured to display a list of all layers from a WMS, and create a button for adding selected layers from the grid to the map. For the grid, we will be using a GeoExt.data.WMSCapabilitiesStore. The grid will be added to the "north" region of the *simple map viewer* from the previous exercise.

To understand the concept of a grid in Ext JS, let's have a look at the following code (this is not the final snippet yet):

```
items.push({
    xtype: "grid",
    ref: "capsGrid",
    title: "Available Layers",
    region: "north",
    height: 150,
    viewConfig: {forceFit: true},
    store: new Ext.data.ArrayStore({
        data: [["foo", "bar"]],
        fields: ["field1", "field2"]
    }),
    columns: [
        {header: "Field 1"}, {header: "Field 2"}
    ]
});
```

Tasks

1. If you haven't already done so, add the text above to your `map.html` file, at the end of the application's script block.
2. Open the page in your browser to confirm things work: http://localhost:8080/ol_workshop/map.html. In addition to the map, you should see a grid with two columns and a single row of dummy data.

2.1.2 Populating the Grid with Data from a GeoExt.data.WMSCapabilitiesStore

Our grid, as it is now, uses an `Ext.data.ArrayStore`, which provides data in an array along with a field definition to create records from. This is the basic principle of an Ext JS store: it provides `Ext.data.Record` instances created by its `Ext.data.Reader`. The store can be used to populate e.g. grids or combo boxes.

The GeoExt.data.WMSCapabilitiesStore uses its reader to create records from a WMS GetCapabilities response. So for most applications, the only property required in its configuration object is the `url` for the GetCapabilities request.

```
store: new GeoExt.data.WMSCapabilitiesStore({
    url: "/geoserver/wms?SERVICE=WMS&REQUEST=GetCapabilities&VERSION=1.1.1",
    autoLoad: true
}),
```

This configures the store to use a plain GeoExt.data.WMSCapabilitiesReader, which uses a HTTP GET request to fetch the data. We add the `autoLoad: true` configuration property to make sure that the request gets sent as soon as the component is ready.

The records (GeoExt.data.LayerRecord) in this store contain several fields. In the grid, we want to display the `name`, `title` and `abstract` fields of each layer. So we have to configure it with the correct column definition:

```
columns: [
    {header: "Name", dataIndex: "name", sortable: true},
    {header: "Title", dataIndex: "title", sortable: true},
    {header: "Abstract", dataIndex: "abstract"}
]
```

The `dataIndex` has to match the name of a record's field. So for a grid, we always need to configure a store that provides the records for the rows, and a column model that knows which field of each record belongs to which column.

Tasks

1. Replace the `Ext.data.ArrayStore` in the *example* with the *properly configured WMSCapabilitiesStore* from above.
2. Replace the dummy column definition with the *correct definition* of name, title and abstract for each layer.

   Your grid configuration object should now look like this:

```
    items.push({
        xtype: "grid",
        ref: "capsGrid", // makes the grid available as app.capsGrid
        title: "Available Layers",
        region: "north",
        height: 150,
        viewConfig: {forceFit: true},
        store: new GeoExt.data.WMSCapabilitiesStore({
            url: "/geoserver/wms?SERVICE=WMS&REQUEST=GetCapabilities&VERSION=1.1.1",
            autoLoad: true
        }),
        columns: [
            {header: "Name", dataIndex: "name", sortable: true},
            {header: "Title", dataIndex: "title", sortable: true},
            {header: "Abstract", dataIndex: "abstract"}
        ]
    });
```

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

2.1.3 Adding an “Add to Map” button

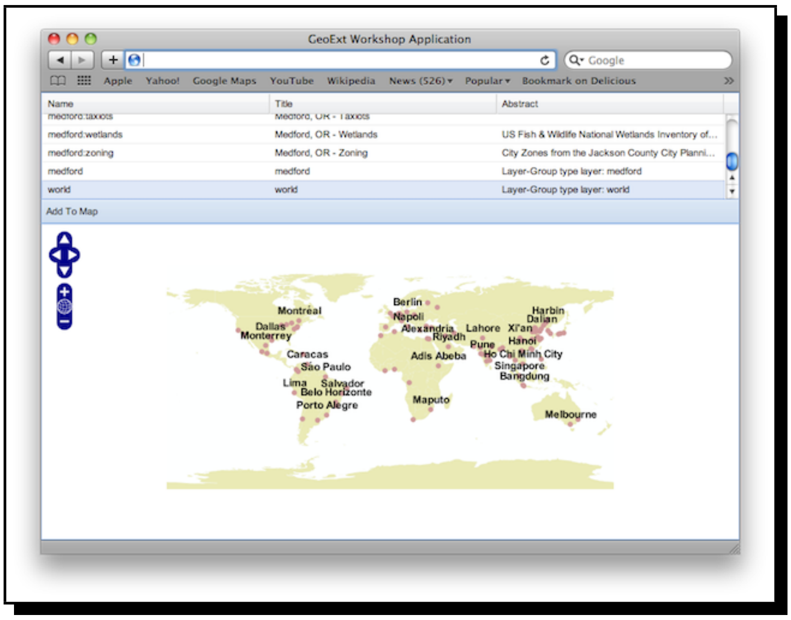
Having successfully loaded WMS Capabilities into a grid, we will now add some code so we can add layers from the grid to the map.

Tasks

1. Add a bottom toolbar (`bbar`) definition to the *grid config object*, below the columns array (don’t forget to add a comma at the end of the columns array!):

```
    bbar: [{
        text: "Add to Map",
        handler: function() {
            app.capsGrid.getSelectionModel().each(function(record) {
                var clone = record.clone();
                clone.getLayer().mergeNewParams({
                    format: "image/png",
                    transparent: true
                });
                app.mapPanel.layers.add(clone);
                app.mapPanel.map.zoomToExtent(
                    OpenLayers.Bounds.fromArray(clone.get("llbbox"))
                );
            });
        }
    }]
```

2. Reload http://localhost:8080/ol_workshop/map.html in your browser again. You should now see an “Add to Map” button on the bottom of the grid. When you select layers in the grid and hit that button, the layers should show up in the map.



Name	Title	Abstract
medford:taxlots	Medford, OR - Taxlots	
medford:wetlands	Medford, OR - Wetlands	US Fish & Wildlife National Wetlands Inventory of...
medford:zoning	Medford, OR - Zoning	City Zones from the Jackson County City Planni...
medford	medford	Layer-Group type layer: medford
world	world	Layer-Group type layer: world

Fig. 2.1: "world" layer selected in the grid and added to the map by clicking the "Add to Map" button.

A Closer Look

Let's examine the handler function of the "Add to Map" button to get an idea of what is going on when we click it:

```
handler: function() {
    app.capsGrid.getSelectionModel().each(function(record) {
        var clone = record.clone();
        clone.getLayer().mergeNewParams({
            format: "image/png",
            transparent: true
        });
        app.mapPanel.layers.add(clone);
        app.mapPanel.map.zoomToExtent(
            OpenLayers.Bounds.fromArray(clone.get("llbbox"))
        );
    });
}
```

Obviously, the grid has a selection model that we can access using `grid.getSelectionModel()`. Since we did not explicitly configure a selection model, our grid automatically instantiated an Ext.grid.RowSelectionModel. This model provides a method called `each`, which we can use to walk through the selected rows. Conveniently, this function gets called with the record of a selected row as argument.

The first thing we do inside this function is clone the record and assign the layer additional parameters.

```
var clone = record.clone();
clone.getLayer().mergeNewParams({
    format: "image/png",
    transparent: true
});
```

Why? In the layer records of the WMSCapabilitiesStore, the `OpenLayers.Layer.WMS` objects (accessed with the `getLayer()` method) are configured without an image format, without projection and without styles. This makes sense because the record also contains information about the available formats, projections and styles from the Capabilities document. For our example, we are confident that all our layers support the WGS84 (EPSG:4326) projection by default and have a neat default style, so we do not care about projection and style. We are also confident that the WMS provides the layer in png format, so we set the format without looking in the record's "formats" field. Finally, we set the `transparent: true` parameter, so we can stack layers nicely.

We have prepared everything now to finally add the layer to the map:

```
mapPanel.layers.add(clone);
mapPanel.map.zoomToExtent(
    OpenLayers.Bounds.fromArray(clone.get("llbbox"))
);
```

To make the layer appear on the map, all we need to do is add the cloned record to the map panel's layer store. Zooming to the extent of the layer is important for the first layer added (yes, you could now remove the `layers` config property from the mapPanel configuration object), because it is part of the required initialization sequence of an `OpenLayers.Map`. For subsequent layers, it is convenient to see the whole layer. The capabilities document provides the extent of the layer, and this information is stored in the record's "llbox" field.

2.1.4 Next Steps

It is nice to be able to add layer, but how do we remove them? And how do we change the order of the layers? All we need to get both is a *layer tree*.

2.2 Adding a Tree View to Manage the Map Panel's Layers

With the Ext.tree.TreePanel and its tree nodes, Ext JS provides a powerful tool to work with hierarchical information. While Ext JS trees cannot be populated from stores, GeoExt provides a tree loader that can turn information from a layer store into tree nodes. Configured with checkboxes, these can be used to turn layers on and off. In addition, thanks to drag & drop support of Ext JS trees, layers can easily be reordered.

2.2.1 Using a Tree Panel for Layer Management

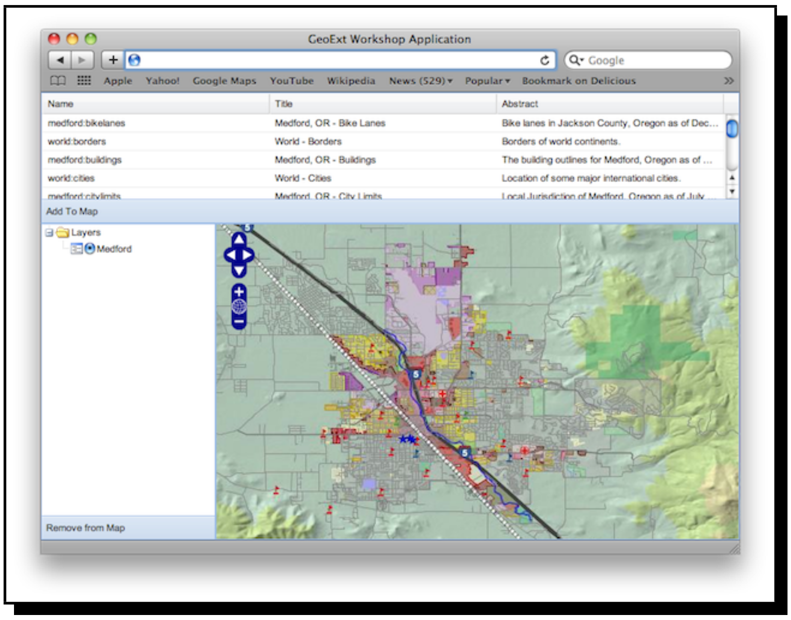
Let's add a tree to the example from the *previous* section. To do so, we create a tree panel with a GeoExt.tree.LayerContainer, and add it as new item to our application's main panel.

Tasks

1. If you don't have it open already, open `map.html` from the previous example in a text editor. Add the following tree definition at the end of our application's script block:

```
items.push({
    xtype: "treepanel",
    ref: "tree",
    region: "west",
    width: 200,
    autoScroll: true,
    enableDD: true,
    root: new GeoExt.tree.LayerContainer({
        expanded: true
    }),
    bbar: [{
        text: "Remove from Map",
        handler: function() {
            var node = app.tree.getSelectionModel().getSelectedNode();
            if (node && node.layer instanceof OpenLayers.Layer.WMS) {
                app.mapPanel.map.removeLayer(node.layer);
            }
        }
        }]
    });
```

2. Reload http://localhost:8080/ol_workshop/map.html in your browser to see the changes. On the left-hand side of the map, we have a tree now. Add some layers from the grid to the map and watch them also appear in the tree. Use the checkboxes to turn layers on and off. Drag and drop layers in the tree to change their order on the map. Select a layer by clicking on the node text, and remove it by clicking the "Remove from Map" button.



Name	Title	Abstract
medford:bikelanes	Medford, OR - Bike Lanes	Bike lanes in Jackson County, Oregon as of Dec...
world:borders	World - Borders	Borders of world continents.
medford:buildings	Medford, OR - Buildings	The building outlines for Medford, Oregon as of ...
world:cities	World - Cities	Location of some major international cities.
medford:citylimits	Medford, OR - City Limits	Local Jurisdiction of Medford, Oregon as of July ...

Fig. 2.2: A tree view of the map's layers for convenient layer management

Looking at the New Code More Closely

First, let's have a look at the *tree configuration* again to see what it consists of.

As we already saw, we can drag and drop tree nodes. This is enabled by setting `enableDD: true`. More interesting is the `root` property.

```
root: new GeoExt.tree.LayerContainer({
    expanded: true
}),
```

Every tree needs to have a root node. GeoExt provides a special layer container node type. Configured with the map panel's layer store as its `layerStore` config option, it will be populated with layer nodes for each of the map's layers. Note that we omitted the `layerStore` config option. The LayerContainer takes the `layers` property from the first MapPanel it finds in the Ext JS registry in this case.

The nodes the LayerContainer is populated with are GeoExt.tree.LayerNode instances. The container makes sure that the list of layers is always synchronized with the map, and the node's checkbox controls the layer's visibility.

Surprisingly, adding a root node that has all map layers as children requires less coding effort than the button to remove layers:

```
bbar: [{
    text: "Remove from Map",
    handler: function() {
        var node = app.tree.getSelectionModel().getSelectedNode();
        if (node && node.layer instanceof OpenLayers.Layer.WMS) {
            app.mapPanel.map.removeLayer(node.layer);
        }
    }
}]
```

We already know the concept of a bottom toolbar from a *previous exercise*. The flesh of the above snippet is the handler function that gets executed when the button is clicked. Like the grid, the tree also has a selection model. The default selection model only supports selection of one node at a time, and we can get the selected node using its `getSelectedNode()` method. All that is left to do is check if there is a selected node, and if the layer is a WMS layer (we don't want to allow removal of vector or other layers we might be adding manually), and remove the layer from the map using the `removeLayer()` method of the `OpenLayers.Map` object.

2.2.2 Next Steps

Now that we can control the content of the map using a tree, we will want a *legend* that explains the map content.

2.3 Adding a Legend Using WMS GetLegendGraphic

It looks like WMS is a good friend of ours: We already got a grid view of layers built from a WMS GetCapabilities request. Without knowing, the layers that we see on the map are images fetched using WMS GetMap, and now we are about to learn about legends created from a WMS GetLegendGraphic request.

2.3.1 A LegendPanel with WMS GetLegendGraphic Images

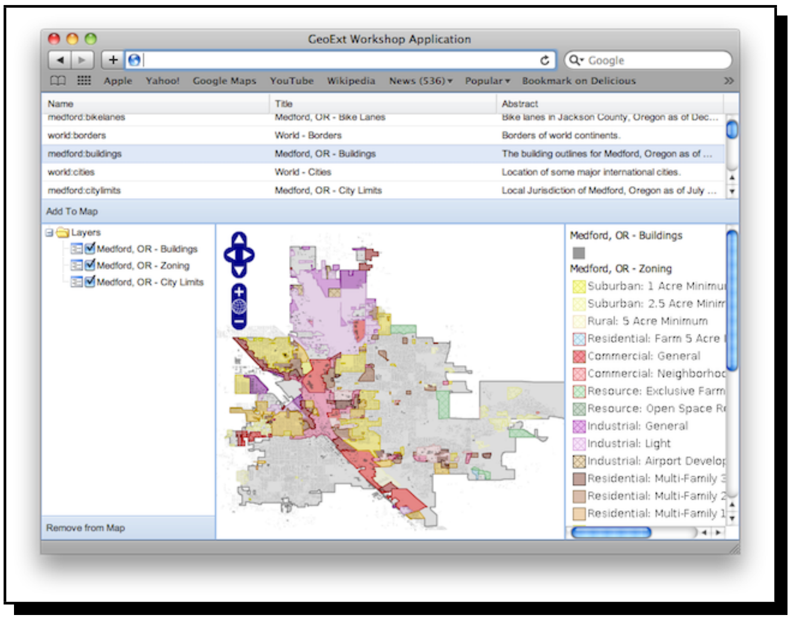
Let's add another panel to our WMS browser. For a legend view, GeoExt provides the GeoExt.LegendPanel. This panel can use a legend image configured in the record's `styles` field, or generate WMS GetLegendGraphic requests.

Tasks

1. Open `map.html` in your text editor again. Add the following legend panel definition at the bottom of the application's script block:

```
    items.push({
        xtype: "gx_legendpanel",
        region: "east",
        width: 200,
        autoScroll: true,
        padding: 5
    });
```

2. Load or refresh http://localhost:8080/ol_workshop/map.html in your browser to see the new legend panel on the right-hand side of the map. Add a layer and watch its legend image appear in the panel.



Name	Title	Abstract
medford:bikelanes	Medford, OR - Bike Lanes	Bike lanes in Jackson County, Oregon as of Dec...
world:borders	World - Borders	Borders of world continents.
medford:buildings	Medford, OR - Buildings	The building outlines for Medford, Oregon as of ...
world:cities	World - Cities	Location of some major international cities.
medford:citylimits	Medford, OR - City Limits	Local Jurisdiction of Medford, Oregon as of July ...

Medford, OR - Buildings
■
Medford, OR - Zoning
▨ Suburban: 1 Acre Minimu
▨ Suburban: 2.5 Acre Minim
▨ Rural: 5 Acre Minimum
▨ Residential: Farm 5 Acre
▨ Commercial: General
▨ Commercial: Neighborhoo
▨ Resource: Exclusive Farm
▨ Resource: Open Space R
▨ Industrial: General
▨ Industrial: Light
▨ Industrial: Airport Develop
■ Residential: Multi-Family 3
■ Residential: Multi-Family 2
■ Residential: Multi-Family 1

Fig. 2.3: WMS browser with a legend describing the map content.

A Closer Look at the New Code

What has happened? We have created a legend panel and placed it in the "east" region (i.e. on the right) of our application's main panel. The only configuration option specific to the legend panel would be the `layerStore` property, which – again – references the layer store of the map panel, and can be omitted when the application has only one map.

2.3.2 What's Next?

As you can see, adding additional components to a GeoExt application is easy – thanks to Ext JS.

In the last part of this exercise, we will see another way of adding components to an application – by using an OpenLayers.Control that creates Ext JS output in a listener function. Let's try this with a *GetFeatureInfo popup*.

2.4 Explore Map Features with a WMS GetFeatureInfo Popup

With GetFeatureInfo, WMS provides an easy way to query a map for feature info. Using OpenLayers and GeoExt makes it easy to access this information from within our application

2.4.1 The OpenLayers WMSGetFeatureInfo control and GeoExt Popups

Let's get familiar with OpenLayers.Control.WMSGetFeatureInfo control and GeoExt.Popup. Also, the Ext.grid.PropertyGrid will be useful to display the feature info in a nice grid - without the need to create another store manually.

Tasks

1. For the popup, we need to include a CSS file in our document's head, which provides the styles for the popup's anchor:

```
<link rel="stylesheet" type="text/css" href="geoext/resources/css/popup.css">
```

Note: GeoExt provides a CSS file which contains all styles that its widgets might require. So if you want to avoid having to worry about required CSS resources, you can include `geoext-all.css` (or `geoext-all-debug.css` for the developer version we are using here) instead of `popup.css`.

2. Now we can create the control. The code below should be added at the end of the application's script block:

```
controls.push(new OpenLayers.Control.WMSGetFeatureInfo({
    autoActivate: true,
    infoFormat: "application/vnd.ogc.gml",
    maxFeatures: 3,
    eventListeners: {
        "getfeatureinfo": function(e) {
            var items = [];
            Ext.each(e.features, function(feature) {
                items.push({
                    xtype: "propertygrid",
                    title: feature.fid,
                    source: feature.attributes
                });
            });
            new GeoExt.Popup({
                title: "Feature Info",
                width: 200,
                height: 200,
                layout: "accordion",
                map: app.mapPanel,
                location: e.xy,
                items: items
            }).show();
        }
    }
}));
```

Now let's examine the code we just added a bit.

Note the `eventListeners` config option for the WMSGetFeatureInfo control. We listen to the "getfeatureinfo" event, which is fired every time we get back feature information from the WMS. For each feature that we get back, we create a property grid:

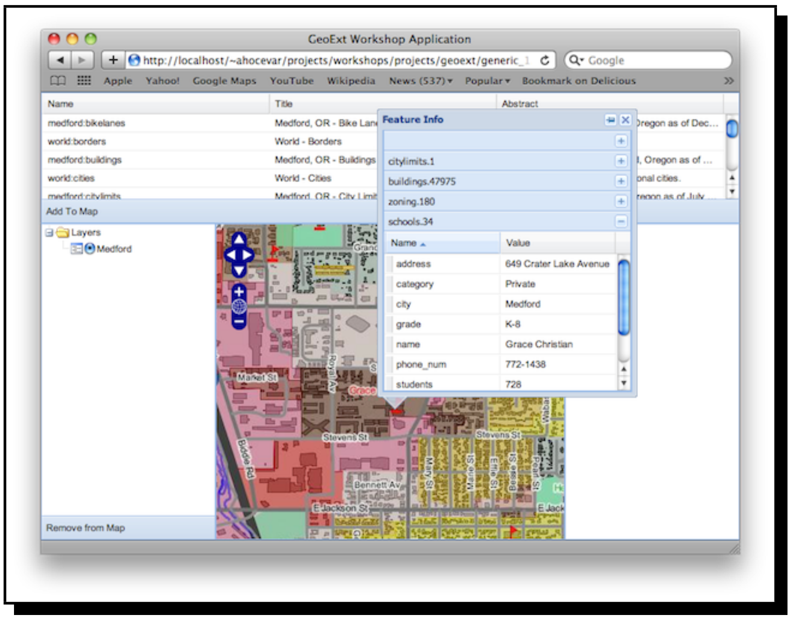
```
Ext.each(e.features, function(feature) {
    items.push({
        xtype: "propertygrid",
        title: feature.fid,
        source: feature.attributes
    });
});
```

The PropertyGrid is a very convenient component for a WMSGetFeatureInfo control configured with an `infoFormat` that returns something we can parse (i.e. not plain text or html). We do not need to configure this component with a store (like we did for the WMSCapabilities grid), we just pass it an arbitrary object (the attributes of a feature here) as `source` config option, and it will create a store internally and populate the grid with its data.

We can easily put a popup on the map and anchor it to the position we clicked on the map:

```
new GeoExt.Popup({
    title: "Feature Info",
    width: 200,
    height: 200,
        layout: "accordion",
        map: app.mapPanel,
        location: e.xy,
        items: items
    }).show();
```

With the `location` config option, we anchor the popup to the position where the click occurred (`e.xy`). We give it an "accordion" layout, and the items are the property grids we created above.



Name	Title	Abstract
medford:bikelanes	Medford, OR - Bike Lan	Oregon as of Dec...
world:borders	World - Borders	
medford:buildings	Medford, OR - Buildings	, Oregon as of ...
world:cities	World - Cities	onal cities.
medford:citylimits	Medford, OR - City Limit	regon as of July ...

Name	Value
address	649 Crater Lake Avenue
category	Private
city	Medford
grade	K-8
name	Grace Christian
phone_num	772-1438
students	728

Fig. 2.4: Map with a popup populated from WMS GetFeatureInfo

2.4.2 Conclusion

You have successfully created a WMS browser application! The whole application has about 120 lines of code. Not much when you consider how many features we were able to pack into it. And it wasn't that hard to develop, was it?

CHAPTER 3

WFS Made Easy with GeoExt

GeoExt provides access to remote WFS data via Stores and Readers, using the same mechanisms that Ext JS provides for any remote data access. Because the GeoExt.data.FeatureStore can synchronize its records with an OpenLayers vector layer, working with vector features from WFS is extremely effortless.

Users familiar with desktop based GIS applications expect to have a combined map and table (grid) view of geospatial data. GeoExt brings this feature to the web. At the end of this module, you will have built a simple WFS feature editor. The grid view comes for free because Ext JS can display data from any store in a grid, and the synchronized selection between map and table is also handled by GeoExt. Rendering the data on the map, editing and committing changes over WFS-T is provided by OpenLayers.

GeoExt's FeatureReader is not limited to WFS protocol and GML – other protocols (e.g. plain HTTP) with formats like KML, GeoRSS or GeoJSON work as well.

In this module, you will:

3.1 Creating a Synchronized Grid and Map View of WFS Features

GeoExt borrows most of its WFS support from OpenLayers. What it does provide though is the GeoExt.data.FeatureStore, so showing feature attributes in a grid is a very easy task. If we just want to display features in a grid, we can use a GeoExt.data.ProtocolProxy, so we don't even need an OpenLayers layer.

3.1.1 Vector Features on a Map and in a Table

Let's build editing functionality into the WMS browser from the *previous chapter*. But one piece at a time. Let's start with an OpenLayers.Layer.Vector for the geometries and a grid for the attributes of the WFS layer.

We'll start with some code that reads a WFS layer, displays its features on a map, shows the attributes in a grid, and synchronizes feature selection between map and grid:

Tasks

1. Open the `map.html` file from the previous exercise in a text editor. Paste the code below at the bottom of the application's script block:

```
    items.push({
        xtype: "grid",
        ref: "featureGrid",
        title: "Feature Table",
        region: "south",
        height: 150,
        sm: new GeoExt.grid.FeatureSelectionModel(),
        store: new Ext.data.Store(),
        columns: [],
        bbar: []
    });
```

2. The above does not do much. It just creates an empty grid in the "south" region of the application viewport, and prepares a selection model (`sm`) that we will use later for synchronizing map and grid selection. Now let's populate the grid with features from the medford:parks layer. To do so, we change the store and the columns in the above grid definition, and make it look like this:

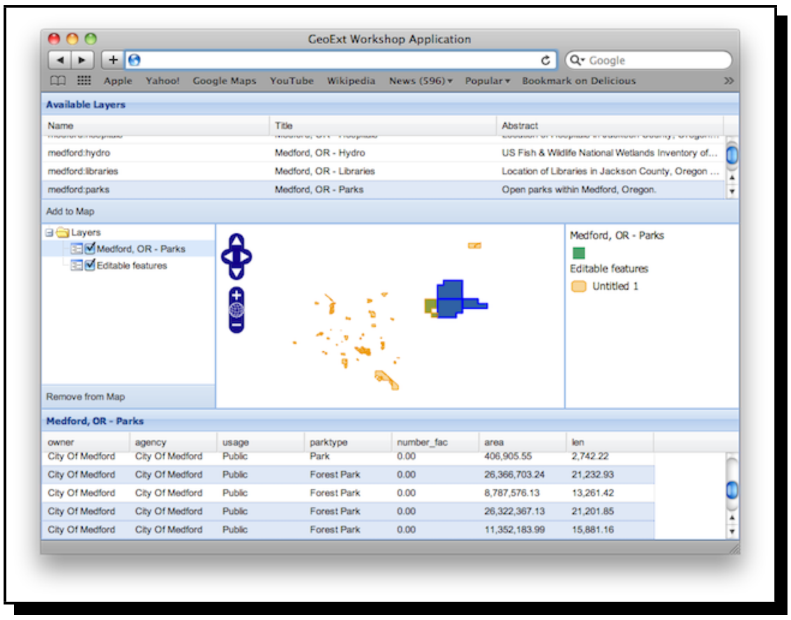
```
    items.push({
        xtype: "grid",
        ref: "featureGrid",
        title: "Feature Table",
        region: "south",
        height: 150,
        sm: new GeoExt.grid.FeatureSelectionModel(),
        store: new GeoExt.data.FeatureStore({
            fields: [
                {name: "owner", type: "string"},
                {name: "agency", type: "string"},
                {name: "name", type: "string"},
                {name: "usage", type: "string"},
                {name: "parktype", type: "string"},
                {name: "number_fac", type: "int"},
                {name: "area", type: "float"},
                {name: "len", type: "float"}
            ],
            proxy: new GeoExt.data.ProtocolProxy({
                protocol: new OpenLayers.Protocol.WFS({
                    url: "/geoserver/ows",
                    version: "1.1.0",
                    featureType: "parks",
                    featureNS: "http://medford.opengeo.org",
                    srsName: "EPSG:4326"
                })
            }),
            autoLoad: true
        }),
        columns: [
            {header: "owner", dataIndex: "owner"},
            {header: "agency", dataIndex: "agency"},
            {header: "name", dataIndex: "name"},
            {header: "usage", dataIndex: "usage"},
            {header: "parktype", dataIndex: "parktype"},
            {xtype: "numbercolumn", header: "number_fac", dataIndex: "number_fac"},
            {xtype: "numbercolumn", header: "area", dataIndex: "area"},
            {xtype: "numbercolumn", header: "len", dataIndex: "len"}
        ],
        bbar: []
    });
```

3. To make this complete, let's also display the geometries on the map, by adding a vector layer. Just append the following snippet at the bottom of the application code:

```
    var vectorLayer = new OpenLayers.Layer.Vector("Editable features");
    Ext.onReady(function() {
        app.mapPanel.map.addLayer(vectorLayer);
        app.featureGrid.store.bind(vectorLayer);
        app.featureGrid.getSelectionModel().bind(vectorLayer);
    });
```

4. After saving your changes, point your browser to http://localhost:8080/ol_workshop/map.html. You should see a new grid in the application, and geometries rendered on the map in orange.

When clicking a row in the grid, its geometry gets highlighted on the map. And when clicking a feature on a map, its attributes will be highlighted in the grid.



Available Layers

Name	Title	Abstract
medford:hydro	Medford, OR - Hydro	US Fish & Wildlife National Wetlands Inventory of...
medford:libraries	Medford, OR - Libraries	Location of Libraries in Jackson County, Oregon ...
medford:parks	Medford, OR - Parks	Open parks within Medford, Oregon.

Medford, OR - Parks

owner	agency	usage	parktype	number_fac	area	len
City Of Medford	City Of Medford	Public	Park	0.00	406,905.55	2,742.22
City Of Medford	City Of Medford	Public	Forest Park	0.00	26,366,703.24	21,232.93
City Of Medford	City Of Medford	Public	Forest Park	0.00	8,787,576.13	13,261.42
City Of Medford	City Of Medford	Public	Forest Park	0.00	26,322,367.13	21,201.85
City Of Medford	City Of Medford	Public	Forest Park	0.00	11,352,183.99	15,881.16

Fig. 3.1: A synchronized map and grid view of WFS features.

Understanding the Code

As we already know, we can configure a grid with a store to tell it where to get its data from. This time we use a GeoExt.data.FeatureStore that we configure with a GeoExt.data.ProtocolProxy to access an OpenLayers.Protocol.WFS.

The synchronization of selections in the grid and on the map is provided by the GeoExt.grid.FeatureSelectionModel, which we use instead of the default selection model:

```
sm: new GeoExt.grid.FeatureSelectionModel(),
```

When we add the vector layer to the map, we also have to bind the selection model and the store to it:

```
app.featureGrid.store.bind(vectorLayer);
app.featureGrid.getSelectionModel().bind(vectorLayer);
```

Note that this is not necessary if the FeatureStore is configured with a `layer`. But since we create the store before the layer here, we don't have access to it yet, so we bind it later.

Bonus Task

Warning: Dragons ahead! Understanding the code from this exercise will be hard for readers without solid programming background. Please skip this task unless
- you are really, really willing to learn and spend an hour on this,
- not understanding code that you paste does not make you unhappy.

The need to manually configure a WFS layer with all the fields and columns makes the application less convenient than we want it to be. Ideally, we could select a layer in the tree, and the grid would automatically display its feature attributes, and the features could be selected in the grid and the map.

Many WMS/WFS implementations, like GeoServer in our case, use the same layer (feature type) names for WMS and WFS. For other WMS implementations, there is a DescribeLayer request which returns an XML document that gives us the link from the WMS layer to an associated WFS feature type or

WCS coverage. Once we know the WFS feature type name, we can issue a WFS DescribeFeatureType request to get the fields and data types for the feature attributes. The disadvantage of omitting the DescribeLayer request is that the DescribeFeatureType request will fail for raster layers. This is not a big deal though - in our implementation it only means that the grid and layer will be cleared when we select a raster layer.

For this bonus exercise, we assume that the WMS layer name is the same as the WFS FeatureType name, so we don't issue a WMS DescribeLayer request. But we do issue a WFS DescribeFeatureType to auto-configure the feature records and the grid.

1. GeoExt has an AttributeStore which holds the field metadata of the feature attributes. It uses OpenLayers.Format.WFSDescribeFeatureType to obtain this information, but discards anything but the field data. For accessing the WFS layer, we also need the feature namespace, so we intercept the read method to obtain all of the DescribeFeatureType response's raw data. Let's add the following at the end of our script block:

```
var rawAttributeData;
var read = OpenLayers.Format.WFSDescribeFeatureType.prototype.read;
OpenLayers.Format.WFSDescribeFeatureType.prototype.read = function() {
    rawAttributeData = read.apply(this, arguments);
    return rawAttributeData;
};
```

2. The FeatureStore and the grid need to be reconfigured when we select a different layer in the tree. Append the following function to the application code:

```
function reconfigure(store, url) {
    var fields = [], columns = [], geometryName, geometryType;
    // regular expression to detect the geometry column
    var geomRegex = /gml:(Multi)?(Point|Line|Polygon|Surface|Geometry).*/;
    var types = {
        // mapping of xml schema data types to Ext JS data types
        "xsd:int": "int",
        "xsd:short": "int",
        "xsd:long": "int",
        "xsd:string": "string",
        "xsd:dateTime": "string",
        "xsd:double": "float",
        "xsd:decimal": "float",
        // mapping of geometry types
        "Line": "Path",
        "Surface": "Polygon"
    };
    store.each(function(rec) {
        var type = rec.get("type");
        var name = rec.get("name");
        var match = geomRegex.exec(type);
        if (match) {
            // we found the geometry column
            geometryName = name;
        } else {
            // we have an attribute column
            fields.push({
                name: name,
                type: types[type]
            });
            columns.push({
                xtype: types[type] == "string" ?
                    "gridcolumn" :
                    "numbercolumn",
                dataIndex: name,
                header: name
            });
        }
    });
    app.featureGrid.reconfigure(new GeoExt.data.FeatureStore({
        autoLoad: true,
        proxy: new GeoExt.data.ProtocolProxy({
            protocol: new OpenLayers.Protocol.WFS({
                url: url,
                version: "1.1.0",
                featureType: rawAttributeData.featureTypes[0].typeName,
                featureNS: rawAttributeData.targetNamespace,
                srsName: "EPSG:4326",
                geometryName: geometryName,
                maxFeatures: 250
            })
        }),
        fields: fields
    }), new Ext.grid.ColumnModel(columns));
    app.featureGrid.store.bind(vectorLayer);
    app.featureGrid.getSelectionModel().bind(vectorLayer);
}
```

Note that the way we build the `fields` and `columns` arrays results in exactly the same configuration for the medford:parks layer that we manually wrote in the previous exercise.

3. When a layer is selected in the tree by clicking on its name, we want to issue a DescribeFeatureType request. This is done by appending the following code:

```
function setLayer(model, node) {
    if(!node || node.layer instanceof OpenLayers.Layer.Vector) {
        return;
    }
    vectorLayer.removeAllFeatures();
    app.featureGrid.reconfigure(
        new Ext.data.Store(),
        new Ext.grid.ColumnModel([])
    );
    var layer = node.layer;
    var url = layer.url.split("?")[0]; // the base url without params
    var schema = new GeoExt.data.AttributeStore({
        url: url,
        // request specific params
        baseParams: {
            "SERVICE": "WFS",
            "REQUEST": "DescribeFeatureType",
            "VERSION": "1.1.0",
            "TYPENAME": layer.params.LAYERS
        },
        autoLoad: true,
        listeners: {
            "load": function(store) {
                app.featureGrid.setTitle(layer.name);
                reconfigure(store, url);
            }
        }
    });
}

Ext.onReady(function() {
    app.tree.getSelectionModel().on(
        "selectionchange", setLayer
    );
});
```

Note that this code calls the `reconfigure` function in the store's load handler. So when the user selects a layer in the tree, `setLayer` is called as `selectionchange` handler on the tree's selection model, and the DescribeFeatureType request for the selected layer is issued. Once the response is available, `reconfigure` is called.

4. After saving your changes, point your browser to http://localhost:8080/ol_workshop/map.html. When you have added a layer to the map that is available as WFS also and select it in the tree, the grid will be populated with the layer's feature attributes, and the features will be rendered on the map.

3.1.2 Next Steps

Just displaying vector features is somewhat boring. We want to edit them. The *next section* explains how to do that.

3.2 Editing Features and Their Attributes

We will now enhance our application by making the layer and its attributes editable, and using WFS-T to commit changes.

3.2.1 Making Layer and Grid Editable

Let's modify our application to allow for editing feature geometries and attributes. On the layer side this requires replacing the SelectFeature control that the FeatureSelectionModel automatically creates with a ModifyFeature control, and adding a DrawFeature control for creating new features. On the grid side, we have to replace the GridPanel with an EditorGridPanel, provide editors for the columns, and avoid multiple feature selection in the selection model (only one feature can be edited at a time).

Tasks

1. Open `map.html` in your text editor. Create controls for modifying and drawing features, by appending the following to the application's script block:

```
var modifyControl = new OpenLayers.Control.ModifyFeature(
    vectorLayer, {autoActivate: true}
);
var drawControl = new OpenLayers.Control.DrawFeature(
    vectorLayer,
    OpenLayers.Handler.Polygon,
    {handlerOptions: {multi: true}}
);
controls.push(modifyControl, drawControl);
```

2. Append code to bind the FeatureSelectionModel to the SelectFeature control that is auto-created with the ModifyFeature control, and to make sure that only one grid row is selected at a time:

```
Ext.onReady(function() {
    var sm = app.featureGrid.getSelectionModel();
    sm.unbind();
    sm.bind(modifyControl.selectControl);
    sm.on("beforerowselect", function() { sm.clearSelections(); });
});
```

3. Append code to add Create and Delete buttons to the feature grid's toolbar:

```
Ext.onReady(function() {
    var bbar = app.featureGrid.getBottomToolbar();
    bbar.add([{
        text: "Delete",
        handler: function() {
            app.featureGrid.getSelectionModel().each(function(rec) {
                var feature = rec.getFeature();
                modifyControl.unselectFeature(feature);
                vectorLayer.removeFeatures([feature]);
            });
        }
    }, new GeoExt.Action({
        control: drawControl,
        text: "Create",
        enableToggle: true
    })]);
    bbar.doLayout();
});
```

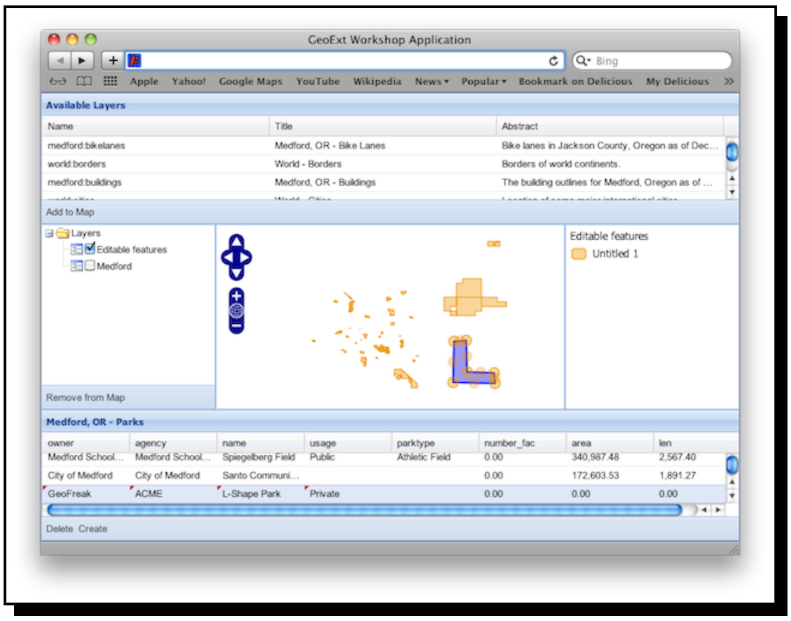
4. Find the featureGrid definition and change its xtype from "grid" to "editorgrid":

```
    xtype: "editorgrid",
    ref: "featureGrid",
```

5. Find the columns definition for the featureGrid, and add appropriate editors. The final columns definition should look like this:

```
    columns: [
        {header: "owner", dataIndex: "owner", editor: {xtype: "textfield"}},
        {header: "agency", dataIndex: "agency", editor: {xtype: "textfield"}},
        {header: "name", dataIndex: "name", editor: {xtype: "textfield"}},
        {header: "usage", dataIndex: "usage", editor: {xtype: "textfield"}},
        {header: "parktype", dataIndex: "parktype", editor: {xtype: "textfield"}},
        {xtype: "numbercolumn", header: "number_fac", dataIndex: "number_fac",
            editor: {xtype: "numberfield"}},
        {xtype: "numbercolumn", header: "area", dataIndex: "area",
            editor: {xtype: "numberfield"}},
        {xtype: "numbercolumn", header: "len", dataIndex: "len",
            editor: {xtype: "numberfield"}}
    ],
```

6. After saving your changes, point your browser to http://localhost:8080/ol_workshop/map.html. You should see the new Delete and Create buttons, and when you select a feature you can modify its vertices.



Name	Title	Abstract
medford:bikelanes	Medford, OR - Bike Lanes	Bike lanes in Jackson County, Oregon as of Dec...
world:borders	World - Borders	Borders of world continents.
medford:buildings	Medford, OR - Buildings	The building outlines for Medford, Oregon as of ...

owner	agency	name	usage	parktype	number_fac	area	len
Medford School...	Medford School...	Spiegelberg Field	Public	Athletic Field	0.00	340,987.48	2,567.40
City of Medford	City of Medford	Santo Communi...			0.00	172,603.53	1,891.27
GeoFreak	ACME	L-Shape Park	Private		0.00	0.00	0.00

Fig. 3.2: A synchronized map and grid view of WFS features.

The Changes Explained

For editing existing and creating new features, we use OpenLayers.Control.ModifyFeature and OpenLayers.Control.DrawFeature.

Both controls are configured with our `vectorLayer`. The DrawFeature control needs to know which sketch handler to use for editing. Since our editing layer is a polygon layer, we use OpenLayers.Handler.Polygon here, and configure it to create MultiPolygon geometries (as required by GeoServer) by setting the `multi` option to true:

```
var modifyControl = new OpenLayers.Control.ModifyFeature(
    vectorLayer, {autoActivate: true}
);
var drawControl = new OpenLayers.Control.DrawFeature(
    vectorLayer,
    OpenLayers.Handler.Polygon,
    {handlerOptions: {multi: true}}
);
```

The `FeatureSelectionModel`, if just used for viewing, is sufficiently configured with its built-in OpenLayers.Control.SelectFeature. For editing, we need to bind it to the SelectFeature control that is auto-created by the ModifyFeature control instead:

```
var sm = app.featureGrid.getSelectionModel();
sm.unbind();
sm.bind(modifyControl.selectControl);
```

Since it does not make sense to select multiple features for editing, we want to make sure that only one feature is selected at a time:

```
sm.on("beforerowselect", function() { sm.clearSelections(); });
```

Instead of using this handler, we also could have configured the `FeatureSelectionModel` with the `singleSelect` option set to true.

The next change is that we want a bottom toolbar on the grid, with buttons for deleting and creating features.

The "Delete" button is just a plain Ext.Button. When clicked, it performs the action defined in its handler.

```
{
    text: "Delete",
    handler: function() {
        app.featureGrid.getSelectionModel().each(function(rec) {
            var feature = rec.getFeature();
            modifyControl.unselectFeature(feature);
            vectorLayer.removeFeatures([feature]);
        });
    }
}
```

Inside the handler, we walk through the grid's current selection. Before removing a record, we use the modifyControl's `unselectFeature` method to remove the feature's editing vertices and unselect the feature, bringing the layer to a clean state.

Thanks to our FeatureStore, a feature added to the layer will automatically also show up in the grid. The "Create" button uses a GeoExt.Action to turn an OpenLayers control into a button. It is important to understand that any OpenLayers control can be added to a toolbar or menu by wrapping it into such an Action. The `enableToggle` option is inherited from Ext.Action and means that the button is a toggle, i.e. can be turned on and off:

```
new GeoExt.Action({
    control: drawControl,
    text: "Create",
    enableToggle: true
})
```

After adding the new buttons, we call `doLayout()` on the toolbar, to make sure that the buttons are rendered properly:

```
bbar.doLayout();
```

Bonus Task

Warning: This bonus exercise assumes that you have completed the *bonus exercise* from the previous chapter.

1. In the `reconfigure()` function, configure editors for the grid columns: TextField for string types, and NumberField for all others. We also need to set the correct sketch handler for the DrawFeature control, depending on the `geometryType` of the layer we are editing. This is how the whole function should look with the changes applied:

```
    function reconfigure(store, url) {
        var fields = [], columns = [], geometryName, geometryType;
        // regular expression to detect the geometry column
        var geomRegex = /gml:(Multi)?(Point|Line|Polygon|Surface|Geometry).*/;
        // mapping of xml schema data types to Ext JS data types
        var types = {
            "xsd:int": "int",
            "xsd:short": "int",
            "xsd:long": "int",
            "xsd:string": "string",
            "xsd:dateTime": "string",
            "xsd:double": "float",
            "xsd:decimal": "float",
            "Line": "Path",
            "Surface": "Polygon"
        };
        store.each(function(rec) {
            var type = rec.get("type");
            var name = rec.get("name");
            var match = geomRegex.exec(type);
            if (match) {
                // we found the geometry column
                geometryName = name;
                // Geometry type for the sketch handler:
                // match[2] is "Point", "Line", "Polygon", "Surface" or "Geometry"
                geometryType = types[match[2]] || match[2];
            } else {
                // we have an attribute column
                fields.push({
                    name: name,
                    type: types[type]
                });
                columns.push({
                    xtype: types[type] == "string" ?
                        "gridcolumn" :
                        "numbercolumn",
                    dataIndex: name,
                    header: name,
                    // textfield editor for strings, numberfield for others
                    editor: {
                        xtype: types[type] == "string" ?
                            "textfield" :
                            "numberfield"
                    }
                });
            }
        });
        app.featureGrid.reconfigure(new GeoExt.data.FeatureStore({
            autoLoad: true,
            proxy: new GeoExt.data.ProtocolProxy({
                protocol: new OpenLayers.Protocol.WFS({
                    url: url,
                    version: "1.1.0",
                    featureType: rawAttributeData.featureTypes[0].typeName,
                    featureNS: rawAttributeData.targetNamespace,
                    srsName: "EPSG:4326",
                    geometryName: geometryName,
                    maxFeatures: 250
                })
            }),
            fields: fields
        }), new Ext.grid.ColumnModel(columns));
        app.featureGrid.store.bind(vectorLayer);
        app.featureGrid.getSelectionModel().bind(vectorLayer);

        // Set the correct sketch handler according to the geometryType
        drawControl.handler = new OpenLayers.Handler[geometryType](
            drawControl, drawControl.callbacks, drawControl.handlerOptions
        );
    }
```

3.2.2 Next Steps

It is nice to be able to create, modify and delete features, but finally we will need to save our changes. The *final section* of this module will teach you how to use the WFS-T functionality of OpenLayers to commit changes to the server.

3.3 Committing Feature Modifications Over WFS-T

Until GeoExt also provides writers, we have to rely on OpenLayers for writing modifications back to the WFS. This is not a big problem though, because WFS-T support in OpenLayers is solid. But it requires us to take some extra care of feature states.

3.3.1 Managing Feature States

For keeping track of "create", "update" and "delete" operations, OpenLayers vector features have a `state` property. The WFS protocol relies on this property to determine which features to commit using an "Insert", "Update" or "Delete" transaction. So we need to make sure that the `state` property gets set properly:

- `OpenLayers.State.INSERT` for features that were just created. We do not need to do anything here, because the DrawFeature control handles this for us.
- `OpenLayers.State.UPDATE` for features with modified attributes, except for features that have `OpenLayers.State.INSERT` set already. For modified geometries, the ModifyFeature control handles this.
- `OpenLayers.State.DELETE` for features that the user wants to delete, except for features that have `OpenLayers.State.INSERT` set, which can be removed.

Tasks

1. Open `map.html` in your text editor. Find the "Delete" button's handler and change it so it properly sets the DELETE feature state and re-adds features to the layer so the server knows we want

to delete them:

```
handler: function() {
    app.featureGrid.store.featureFilter = new OpenLayers.Filter({
        evaluate: function(feature) {
            return feature.state != OpenLayers.State.DELETE;
        }
    });
    app.featureGrid.getSelectionModel().each(function(rec) {
        var feature = rec.getFeature();
        modifyControl.unselectFeature(feature);
        vectorLayer.removeFeatures([feature]);
        if (feature.state != OpenLayers.State.INSERT) {
            feature.state = OpenLayers.State.DELETE;
            vectorLayer.addFeatures([feature]);
        }
    });
}
```

Understanding the Code

By setting the `featureFilter` on the store we prevent the feature from being re-added to the store. In OpenLayers, features with DELETE state won't be rendered, but in Ext JS, if we do not want a deleted feature to show up in the grid, we have to make sure that it is not in the store.

Also note that after removing the deleted feature from the `vectorLayer`, we add it again unless it has an INSERT state (but it won't show up because of OpenLayers state handling and the `featureFilter`). This is necessary because the transaction when saving changes needs to submit information about deleted features, so they can be deleted on the server.

3.3.2 Adding a Save Button

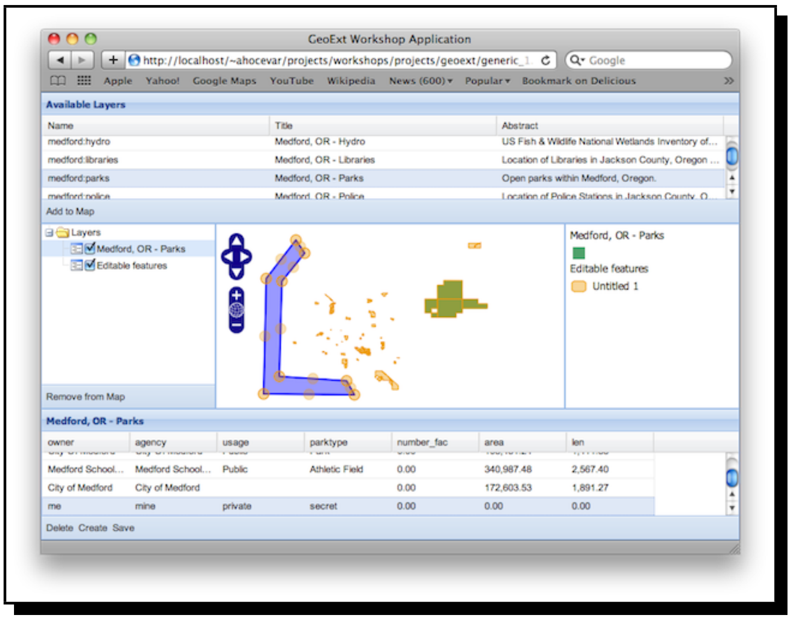
Saving feature modifications the OpenLayers way usually requires the vector layer to be configured with an OpenLayers.Strategy.Save. But since we have a GeoExt store (and not an OpenLayers layer) configured with the WFS protocol here, we cannot do that. Instead, we can call the protocol's `commit()` method directly to save changes.

Tasks

1. Find the definition of the grid toolbar's Delete and Create buttons in your `map.html` file and add the "Save" button configuration and handler. When done, the whole buttons definition should look like this:

```
bbar.add([{
    text: "Delete",
    handler: function() {
        app.featureGrid.store.featureFilter = new OpenLayers.Filter({
            evaluate: function(feature) {
                return feature.state != OpenLayers.State.DELETE;
            }
        });
        app.featureGrid.getSelectionModel().each(function(rec) {
            var feature = rec.getFeature();
            modifyControl.unselectFeature(feature);
            vectorLayer.removeFeatures([feature]);
            if (feature.state != OpenLayers.State.INSERT) {
                feature.state = OpenLayers.State.DELETE;
                vectorLayer.addFeatures([feature]);
            }
        });
    }
}, new GeoExt.Action({
    control: drawControl,
    text: "Create",
    enableToggle: true
}), {
    text: "Save",
    handler: function() {
        app.featureGrid.store.proxy.protocol.commit(
            vectorLayer.features, {
                callback: function() {
                    var layers = app.mapPanel.map.layers;
                    for (var i=layers.length-1; i>=0; --i) {
                        layers[i].redraw(true);
                    }
                    app.featureGrid.store.reload();
                }
        });
    }
}]);
```

2. Save your file and reload http://localhost:8080/ol_workshop/map.html. Make some changes and hit "Save". Reload the page to see that your changes were persisted.



Available Layers

Name	Title	Abstract
medford:hydro	Medford, OR - Hydro	US Fish & Wildlife National Wetlands Inventory of...
medford:libraries	Medford, OR - Libraries	Location of Libraries in Jackson County, Oregon ...
medford:parks	Medford, OR - Parks	Open parks within Medford, Oregon.
medford:police	Medford, OR - Police	Location of Police Stations in Jackson County, O...

Medford, OR - Parks

owner	agency	usage	parktype	number_fac	area	len
Medford School...	Medford School...	Public	Athletic Field	0.00	340,987.48	2,567.40
City of Medford	City of Medford			0.00	172,603.53	1,891.27
me	mine	private	secret	0.00	0.00	0.00

Fig. 3.3: Application with "Save" button and a persisted feature after reloading.

The Commit Callback Explained

By calling the `commit()` method with a callback option, we can perform actions when the commit operation has completed. In this case, we want to redraw all layers to reflect the changes in WMS and group layers. And we also reload the feature store, to reset all feature states and have all features with their correct feature ids.

```
callback: function() {
    for (var i=layers.length-1; i>=0; --i) {
        layers[i].redraw(true);
    }
    app.featureGrid.store.reload();
}
```

Note that reloading the store is only necessary for GeoServer layers that use a shapefile as data store, because the WFS Insert doesn't report the inserted feature ids for those.

3.3.3 Conclusion

You have successfully created a WFS based feature editor. GeoExt makes working with features easy, thanks to its FeatureStore. Although there is no write support yet for the FeatureStore in GeoExt, saving changes via WFS-T is easy because of the solid WFS-T support in OpenLayers and the interoperability between GeoExt and OpenLayers.