Exercise: Working with a DojoX DataGrid

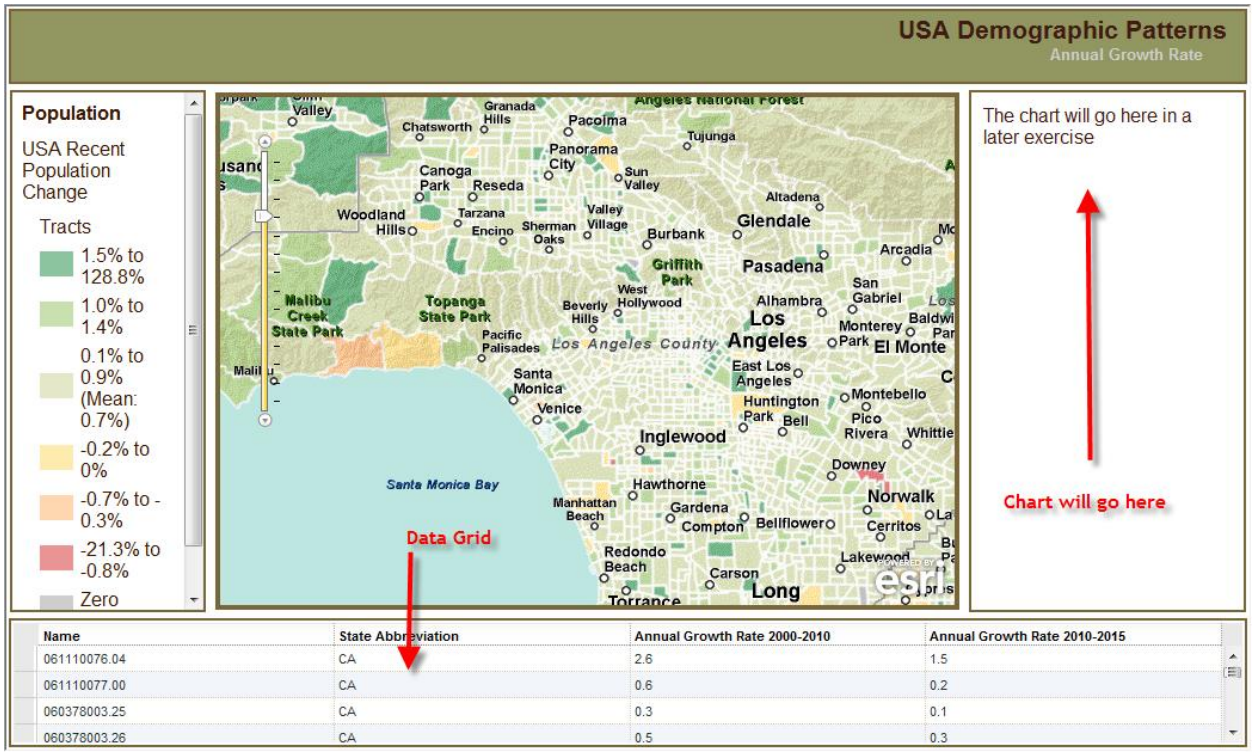
The DojoX Datagrid is a state-of-the art grid or spreadsheet widget with many features. Datagrid is tightly integrated with dojo.data and supports any data store that you supply so it's a fairly intuitive process to connect a data store to a Datagrid. This makes supplying data to your grid extremely easy and efficient. In addition, Datagrid has many features including editing capabilities, sorting, sub-rows, row highlighting, themes, and more.

In this exercise you will learn how to populate a DojoX DataGrid with data obtained from a QueryTask performed against an ArcGIS Server service that contains annual population growth rates in the U.S. by block ground, census tract, county, and state. The DataGrid will hold records obtained through a spatial query of all features from the currently active layer that are within the view extent. As you zoom or pan across the map, the contents of the DataGrid will change to reflect the features in the map extent.

In a later exercise on charting we will use this same information to create a chart showing the features with the highest population growth rates in the current map extent.

Below you see a figure displaying the finished product for this exercise. The DataGrid will be displayed in the bottom region of our application and the chart will be presented in the right region. Charting will be completed in a later exercise.

Much of the code for this exercise has already been written so that you can focus specifically on creating and populating the DataGrid.



Step 1: Examine the files for this exercise

- Navigate to the C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\lab\advanced_dojo\exercises folder. There are three files that are used in this application: grid_charting.htm, createGrid.js, and layout.css. You will only be working with the first two files in this exercise. The layout.css file controls various aspects of the display and won't be used.

Step 2: Add resources to grid_charting.htm

- Navigate to the C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\lab\advanced_dojo\exercises folder and open grid_charting.htm in your favorite HTML or text editor. This file primarily controls the layout of the page as well as the initial map creation. Most of the code has already been written for you so that you can focus on the elements related to the DataGrid.

- Add links to the DataGrid style sheets. The style sheets control the visual characteristics of the DataGrid. Add the two lines that you see highlighted below. We're working with the Nihilo Dojo theme in this exercise, but Dojo has other themes including Nihilo and Claro.

```
<link rel="stylesheet" type="text/css" href="http://serverapi.arcgisonline.com/jsapi/arcgis/2.1/js/dojo/dijit/themes/Nihilo/Nihilo.css">

<link rel="stylesheet" type="text/css" href="http://serverapi.arcgisonline.com/jsapi/arcgis/2.1/js/dojo/dojox/grid/resources/Grid.css">
<link rel="stylesheet" type="text/css" href="http://serverapi.arcgisonline.com/jsapi/arcgis/2.1/js/dojo/dojox/grid/resources/nihiloGrid.css">
<link rel="stylesheet" type="text/css" href="layout.css">
```

- Add the Dojo resources that you will need to perform the query, store the data, and populate the DataGrid. The DataGrid resource should be obvious and you'll probably remember the ItemFileReadStore resource from an exercise earlier in this section of the course. ItemFileReadStore is an object used to store JSON format data. In this exercise, ItemFileReadStore will hold the results of our query. Finally, we add the esri.tasks.query resource which is needed to perform the query for features in the current map extent.

```
<script type="text/javascript">
  dojo.require("dijit.dijit"); // optimize: load dijit layer
  dojo.require("dijit.layout.BorderContainer");
  dojo.require("dijit.layout.ContentPane");
  dojo.require("dojox.grid.DataGrid");
  dojo.require("dojo.data.ItemFileReadStore");
  dojo.require("esri.dijit.Legend");
  dojo.require("esri.tasks.query");
  dojo.require("esri.map");
```

Step 3: Connect the Map.onExtentChange Event to a Handler

Each time the map extent changes your application will need to query the map service for features that fall within the new extent. The Map object contains an 'onExtentChange' event that can be hooked into a JavaScript function that will handle this event.

- Add the code you see below .

```
dojo.connect(map, 'onLoad', function(map) {
  dojo.connect(dijit.byId('map'), 'resize', resizeMap);
  dojo.connect(map, "onExtentChange", gridHandler);
  dojo.connect(map, "onZoomStart", showLoading);
  dojo.connect(map, "onPanStart", showLoading);
});
```

Each time the map extent changes a JavaScript function called 'gridHandler' will be executed. Now, this function does not yet exist. The 'gridHandler' function is located in the createGrid.js file. Part of this function has been written, but in a later step in this exercise you will complete the code for this function.

Step 4: Create DataGrid Structure

A DojoX DataGrid can be declaratively or programmatically created. In this exercise you will declaratively create the grid. In this case, a DataGrid will populate the contents of an HTML table with special Dojo tags added.

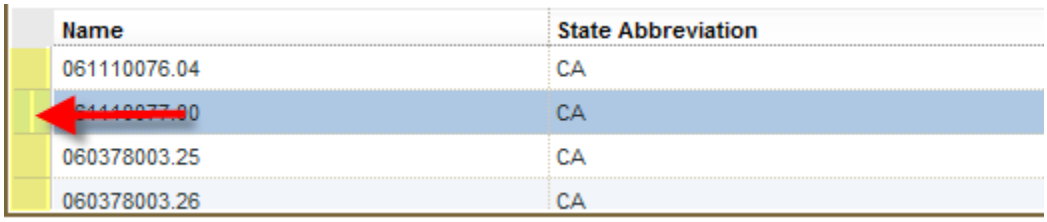
- Scroll to the bottom of grid_charting.htm until you find the following:

```
<body class="nihilo">
  <div id="mainWindow" dojotype="dijit.layout.BorderContainer" design="headline" gutters="false" style="width:100%; height:100%;">
    <div id="header" dojotype="dijit.layout.ContentPane" region="top" >
      USA Demographic Patterns
      <div id="subheader">Annual Growth Rate</div>
    </div>
    <div dojotype="dijit.layout.ContentPane" region="left" id="leftPane" >
      <div id="legendDiv"></div>
    </div>
    <div id="map" class="shadow" dojotype="dijit.layout.ContentPane" region="center" >
       <img id="loadingImg" src="loading.gif" style="position:absolute; right:512px; top:256px; z-index:100;" />
    </div>
    <div dojotype="dijit.layout.ContentPane" region="right" id="rightPane" >
      The chart will go here in a later exercise
    </div>
    <div id="footer" dojotype="dijit.layout.ContentPane" region="bottom" >


    </div>
  </div>
 </body>
</html>
```

In this step you will be creating an HTML table inside the tag for the bottom region. Add the following code and then we'll discuss.

```
<div id="footer" dojotype="dijit.layout.ContentPane" region="bottom" >

  <table dojoType="dojox.grid.DataGrid" jsid="grid" id="grid" clientSort="true" style="width: 600px; height: 600px;" rowSelector="20px" sortInfo="-4">
    <thead>
      <tr>
        <th width="auto" field="NAME">Name</th>
        <th width="auto" field="ST_ABBREV">State Abbreviation</th>
        <th width="auto" field="POPGRW00CY">Annual Growth Rate 2000-2010</th>
        <th width="auto" field="POPGRWCYFY">Annual Growth Rate 2010-2015</th>
      </tr>
    </thead>
  </table>
</div>
```

This code block defines the structure for the DataGrid. There are several things you'll want to keep in mind when creating a DataGrid. First, make sure that you create an attribute called dojoType and assign a value of dojox.grid.DataGrid. Also make sure that you give your DataGrid an ID. In this case we've simply named it 'grid'. This ID will be important in a later step when we need to supply a reference for where the data returned from a query will be placed. The 'clientSort' attribute determine whether or not the grid will be able to perform ascending and descending sorts on the columns. The 'style' attribute defines the height and width parameters, and 'rowSelector' defines a small box on the left hand side of the grid that can be used to select a row. This just defines the size of that box. You'll see a visual depiction of the 'rowSelector' in the figure below. Finally, the 'sortInfo' attribute is set to -4 indicating that we'd like to automatically perform a descending sort of the fourth column.



That is all the changes that we'll make to the grid_charting.htm file. For the remainder of the exercise we'll concentrate on finishing out the code that handles our query and displays the results in the DataGrid.

Step 5: Create Input Parameters for the Query

- Navigate to the C:\Program Files\Apache Software Foundation\Apache2.2\htdocs\lab\advanced_dojo\exercises folder and open createGrid.js in your favorite HTML or text editor. This file contains functions that will perform the query for features within the current map extent and processes the results which are loaded into the DataGrid. Again, part of the code has already been pre-written so that you can focus on performing the query and loading the results into the DataGrid.

- Create the variables you'll need to perform the query and hold the results.

```
var query, queryTask;
var grid, store;

function gridHandler(ext, delta, levelChange, lod) {

   var layerID;

   if (lod.scale <= 250000) {
      //query Blocks
      layerID = "1";
   }
   else if (lod.scale > 250000 && lod.scale <= 1000000)
   {
      //query tracts
      layerID = "2";
   }
   else if (lod.scale > 1000000 && lod.scale <= 25000000) {
      //query counties
      layerID = "3";
   }
   else {
      //query states
      layerID = "4";
   }
```

The 'gridHandler' function already contains an if/else decision tree. The map service that we are using contains four layers including census block groups, census tracts, counties, and states. Population growth data is provided on each of these layers. As you zoom in and out in the application only one of these map layers is visible at the current map scale. When zoomed in beyond a scale of 250,000, census blocks are displayed. Census tracts are displayed at scales between 250,000 and 1 million. Counties are displayed at scales between 1 million and 2.5 million, and states are displayed for scales beyond 2.5 million. This if/else decision structure determines the current map scale and assigns a value corresponding to the layer number to a variable called 'layerID'. You'll see how this is used later.

I should also point out that the 'gridHandler' function accepts four parameters including the extent, delta, level change, and lod. When the Map.onExtentChange event is fired these parameters are passed into the handling function (gridHandler). The level of detail (lod) enables us to get at the map scale which I've already described. The delta and level change parameters are not of interest in this case. The extent (ext) is the current map extent which will be used as an input parameter in our spatial query.

- Inside 'gridHandler' create a new Query object and assign the parameters. You should be familiar with the Query object from a previous lecture and exercise. The 'spatialRelationship' and 'geometry' properties are used to define the parameters of a spatial query. The geometry to use in the spatial query will be the current map extent, and

we want to find all features that intersect this extent. In addition, we define a 'where clause' that acts as an attribute query. The attribute query will filter out only those features with a population growth percentage of 2.5% or greater. The three lines of code highlighted below in green illustrate the parameters that define the spatial and attribute query. We also provide the 'outFields' parameter for the fields that we want returned and we have indicated that we want geometry to be returned. We won't use the geometry in this exercise, but it would be nice to highlight the feature when a record is clicked in the DataGrid. You may want to pursue doing as an extra step after completing the exercise.

```
function gridHandler(ext, delta, levelChange, lod) {

    query = new esri.tasks.Query();
    query.returnGeometry = true;
    query.outFields = ["NAME", "ST_ABBREV", "POPGRW00CY","POPGRWCYFY"];
    query.spatialRelationship = esri.tasks.Query.SPATIAL_REL_INTERSECTS;
    query.geometry = ext;
    query.where = "POPGRWCYFY >= 2.5";

    var layerID;

    if (lod.scale <= 250000) {
        //query Blocks
        layerID = "1";
    }
    else if (lod.scale > 250000 && lod.scale <= 1000000)
    {
        //query tracts
        layerID = "2";
    }
    else if (lod.scale > 1000000 && lod.scale <= 25000000) {
        //query counties
        layerID = "3";
    }
    else {
        //query states
        layerID = "4";
    }
```

Step 6: Create QueryTask and Execute

Here is where the 'layerID' from the if/else block will be used.

- Add the following lines of code and then we'll discuss.

```
function gridHandler(ext, delta, levelChange, lod) {

   query = new esri.tasks.Query();
   query.returnGeometry = true;
   query.outFields = ["NAME", "ST_ABBREV", "POPGRWOOCY","POPGRWCYFY"];
   query.spatialRelationship = esri.tasks.Query.SPATIAL_REL_INTERSECTS;
   query.geometry = ext;
   query.where = "POPGRWCYFY >= 2.5";

   var layerID;

   if (lod.scale <= 250000) {
      //query Blocks
      layerID = "1";
   }
   else if (lod.scale > 250000 && lod.scale <= 1000000)
   {
      //query tracts
      layerID = "2";
   }
   else if (lod.scale > 1000000 && lod.scale <= 25000000) {
      //query counties
      layerID = "3";
   }
   else {
      //query states
      layerID = "4";
   }
   queryTask = new esri.tasks.QueryTask("http://server.arcgisonline.com/ArcGIS/rest/services/Demographics/USA_Recent_Population_Change/MapServer/" + layerID);
   queryTask.execute(query, processResults, errCallback);
}
```

Notice that at the end of the URL string for initiating the QueryTask that we have supplied the layered variable. In the figure below I have displayed a portion of the metadata page for the map service we are using in this exercise. Notice the numbers just to the right of each layer name. These numbers correspond to the value being assigned to 'layerID' in our if/else block.

Demographics/USA_Recent_Population_Change (MapServer)

View In: ArcMap ArcGIS Explorer ArcGIS JavaScript Google Earth ArcGIS.com Map Bing Maps (terms of use) Google Maps

View Footprint In: Google Earth

Service Description: This thematic map indicates the annual compound rate of total population change in the United States fr
Total Population is the total number of residents in an area. Residence refers to the "usual place" where a person lives. Total Po
from the U.S. Census 2000. Total Population for 2010 is estimated with ESRI's unique methodology blending GIS and Statistical t
unique combination of data sources (ESRI Demographic Update Methodology: 2010/2015). The geography depicts states at grea
counties at 1m to 25m scale, Census Tracts at 250k to 1m scale, and Census Block Groups at less than 250k scale. The map ha
displayed with semi-transparency of about 50% for overlay on other base maps, which is reflected in the legend for the map. Fo
this map, visit us online.

Map Name: Layers

Legend

All Layers and Tables

Layers:

- USA Recent Population Change (0)
  - Block Groups (1)
  - Tracts (2)
  - Counties (3)
  - States (4)

In the QueryTask.execute method we have also specified that the callback function to

process the results of the query should be handled by the 'processResults' function and any error should be handled by the 'errBack' function.

Step 7: Handle Query Results

The 'processResults' function has been stubbed out as seen below.

```
function processResults(results) {


}
```

- Obtain the results in the form of a FeatureSet and then loop through each of the features returned in the FeatureSet, pull out the attributes, and add them to the item array. Remember that the only features returned are those that are within the current map extent AND that have a population growth percentage of 2.5% or greater for the period 2010-2015. Add the code below to accomplish this.

```
function processResults(results) {

  var featureSet = results.features;

  //Create items array to be added to store's data
  var items = []; //all items to be stored in data store
  for (var i=0, il=featureSet.length; i<il; i++) {
     items.push(featureSet[i].attributes);
  }


}
```

- We also want to highlight the features that have been returned. To do this we will create a new symbol object, set the symbol property for each feature graphic and then add the graphic to the

GraphicsLayer. Add the code you see below to accomplish this.

```
function processResults(results) {
   map.graphics.clear();
   var symbol = new esri.symbol.SimpleFillSymbol
        (esri.symbol.SimpleFillSymbol.STYLE_NULL,
               new esri.symbol.SimpleLineSymbol(esri.symbol.SimpleFillSymbol.STYLE_SOLID,
               new dojo.Color([100,100,100]), 4), new dojo.Color([0,0,255,0.20]));

   var featureSet = results.features;

   //Create items array to be added to store's data
   var items = []; //all items to be stored in data store
   for (var i=0, il=featureSet.length; i<il; i++) {
      var graphic = featureSet[i];
      graphic.setSymbol(symbol);
      map.graphics.add(graphic);
      items.push(featureSet[i].attributes);
   }


}
```

- Next we'll create the JSON object that will be used to create an instance of ItemFileReadStore. Add the following code. As you learned in a previous exercise the JSON object needs to have a very specific structure containing an identifier, an optional label, and finally the items which are simply name value pairs. In this case, we assign the 'items' variable to the 'items' property. We created the 'items' variable as an array of attributes pulled from the features that matched the query.

```
function processResults(results) {
   map.graphics.clear();
   var symbol = new esri.symbol.SimpleFillSymbol(
           esri.symbol.SimpleFillSymbol.STYLE_NULL,
           new esri.symbol.SimpleLineSymbol(esri.symbol.SimpleFillSymbol.STYLE_SOLID,
                   new dojo.Color([100,100,100]), 4), new dojo.Color([0,0,255,0.20]));

   var featureSet = results.features;

   //Create items array to be added to store's data
   var items = []; //all items to be stored in data store
   for (var i=0, il=featureSet.length; i<il; i++) {
      var graphic = featureSet[i];
      graphic.setSymbol(symbol);
      map.graphics.add(graphic);
      items.push(featureSet[i].attributes);
   }

   //Create data object to be used in store
   var data = {
      identifier: "NAME",  //This field needs to have unique values
      label: "NAME", //Name field for display. Not pertinent to a grid but may be used elsewhere.
      items: items
   };

}
```

- Finally, create an instance of ItemFileReadStore using the newly created JSON ‘data’ object and assign it to the DataGrid. Add the code below and then we’ll discuss.

```
function processResults(results) {
   map.graphics.clear();
   var symbol = new esri.symbol.SimpleFillSymbol(
           esri.symbol.SimpleFillSymbol.STYLE_NULL,
           new esri.symbol.SimpleLineSymbol(esri.symbol.SimpleFillSymbol.STYLE_SOLID,
                   new dojo.Color([100,100,100]), 4), new dojo.Color([0,0,255,0.20]));

   var featureSet = results.features;

   //Create items array to be added to store's data
   var items = []; //all items to be stored in data store
   for (var i=0, il=featureSet.length; i<il; i++) {
      var graphic = featureSet[i];
      graphic.setSymbol(symbol);
      map.graphics.add(graphic);
      items.push(featureSet[i].attributes);
   }

   //Create data object to be used in store
   var data = {
      identifier: "NAME",  //This field needs to have unique values
      label: "NAME", //Name field for display. Not pertinent to a grid but may be used elsewhere.
      items: items
   };

   //Create data store and bind to grid.
   store = new dojo.data.ItemFileReadStore({ data:data });
   grid.setStore(store);
   hideLoading();
}
```

We created a new variable called ‘store’ and assigned a new ItemFileReadStore using the JSON ‘data’ object. Next, we assigned the new instance of ItemFileReadStore (‘store’) to our DataGrid using the DataGrid.setStore() method. Remember back in step 4 when we created the structure for our DataGrid (see figure below)? I said then that it was very important to use the ‘id’ attribute to give the DataGrid a unique identifer. This is where that comes into play. We assigned an id of ‘grid’ to our newly created DataGrid. Notice in the code block that I just had you create that we called grid.setStore(store). The ‘grid’ refers to the uniquely identified DataGrid that we created in step 4 which has the same id.

The final line of code that I had you insert simply hides the loading.gif file that is displayed until

our processing is complete.

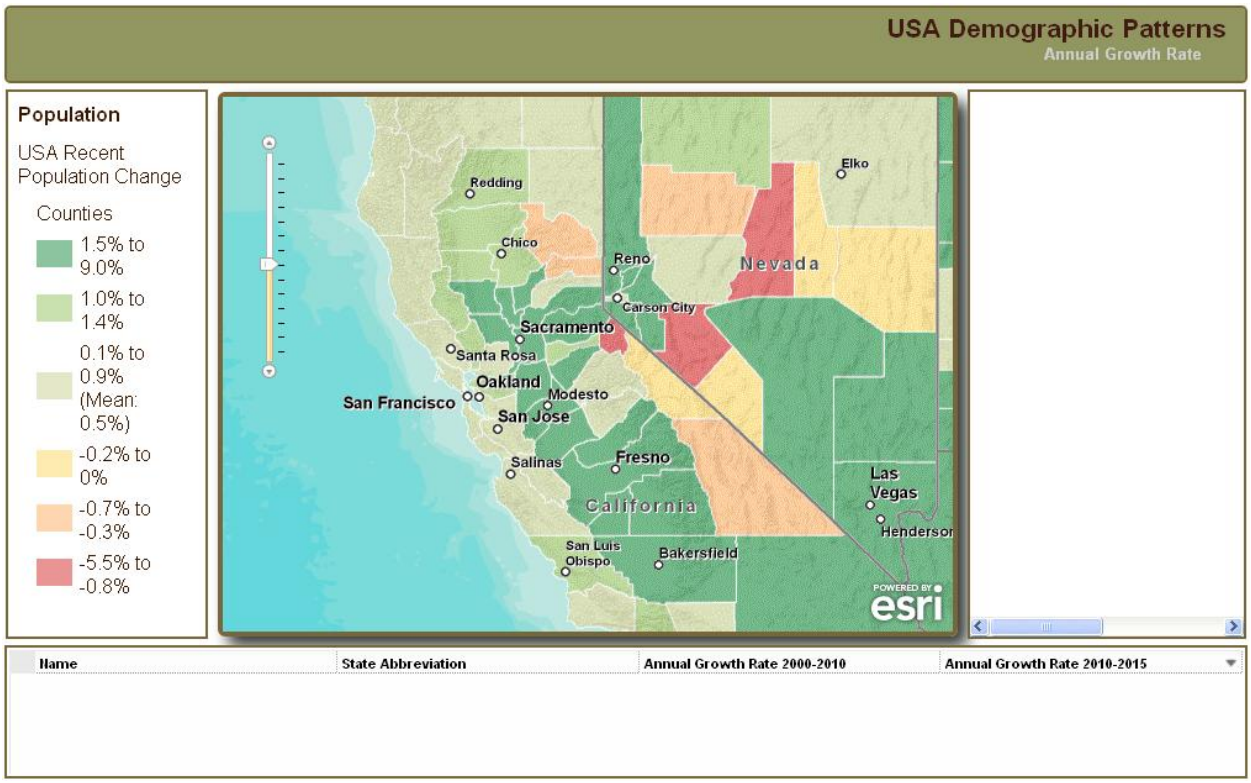
```
<div id="footer" dojotype="dijit.layout.ContentPane" region="bottom" >

<table dojoType="dojox.grid.DataGrid" jsid="grid" id="grid" clientSort="true" style="width: 600px; height: 600px;" rowSelector="20px" sortInfo="-4">
   <thead>
        <tr>
           <th width="auto" field="NAME">Name</th>
           <th width="auto" field="ST_ABBREV">State Abbreviation</th>
           <th width="auto" field="POPGRW00CY">Annual Growth Rate 2000-2010</th>
           <th width="auto" field="POPGRWCYFY">Annual Growth Rate 2010-2015</th>
        </tr>
   </thead>
</table>
</div>
```

Step 8: View the File

- Save your file.

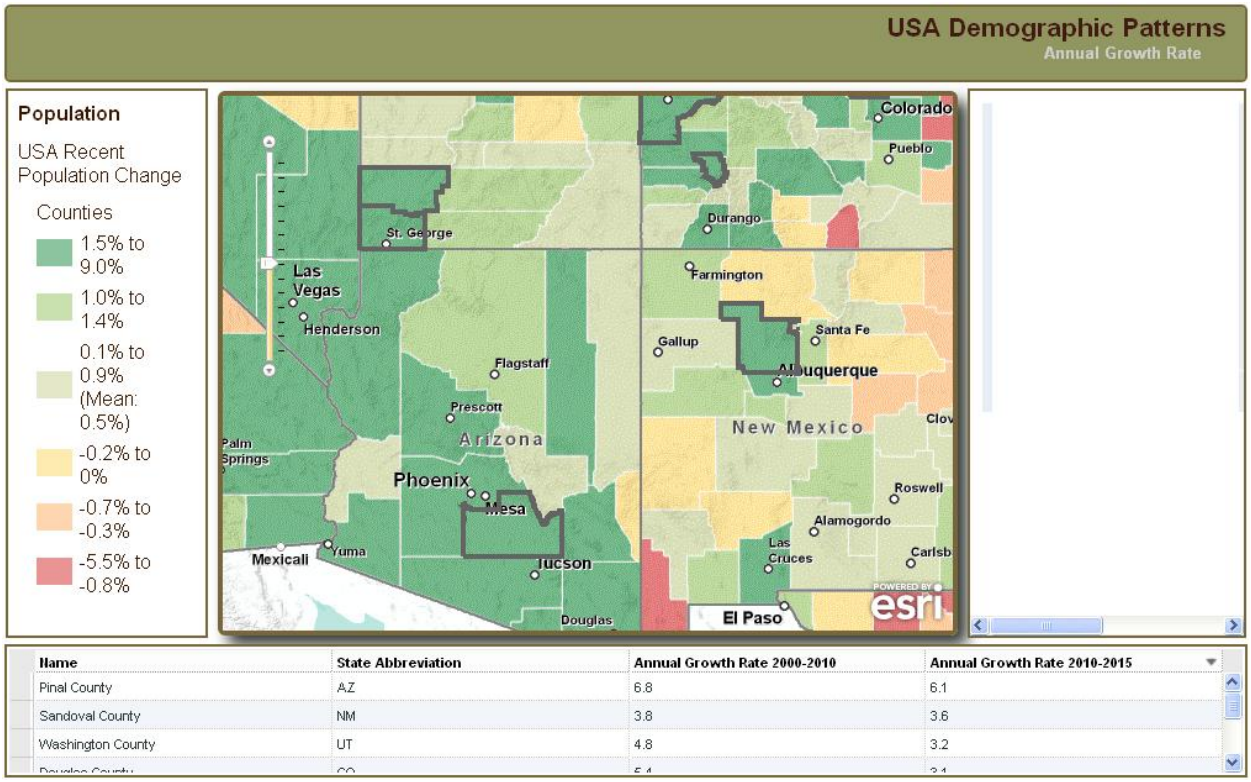
- Open a web browser and point to: http://localhost/lab/advanced_dojo/exercises/grid_charting.htm

- If you've done everything correctly you should see something like the figure below. We didn't populate the DataGrid based on the initial extent that was displayed, but in a real application you would probably want to do that. Pan to another extent on the map and the data will be displayed in the DataGrid. Each time you perform a zoom or pan operation this will be the case.



Notice that both the Legend and the DataGrid values change as you zoom in and out. This is due to the map scaling that we discussed earlier. Initially, counties are displayed, but if you continue to zoom in census tracts and then block groups will be displayed. Similarly as you zoom out

states will eventually be displayed. The DataGrid values change accordingly. Also notice that the features have also been highlighted on the map.



Name	State Abbreviation	Annual Growth Rate 2000-2010	Annual Growth Rate 2010-2015
Pinal County	AZ	6.8	6.1
Sandoval County	NM	3.8	3.6
Washington County	UT	4.8	3.2