Developing and Packaging a Custom Visual

Setup Time: 60 minutes

**Lab Folder**: C:\Student\Modules\06_CustomVisuals\Lab

Overview: In this lab, you will continue to develop custom visuals using the Power BI Custom Visual CLI Tool (PBIVIZ) that you began to work with in the previous lab. Now that you know how to create new projects using the PBIVIZ utility and to integrate the D3 library, this lab will focus on moving ahead and learning the fundamentals of designing and implementing a useful custom visual for Power BI.

Unlike the previous lab where you created three different custom visual projects, this lab focuses on a single custom visual project named **barchart** that you will work on throughout all the exercises of this lab. You will learn how to define visual capabilities and how to consume data using categorical mapping. You will also learn how to add custom properties. In the final exercise, you will work through the steps to package the **barchart** visual as a PBIVIZ package file which can be imported and tested in Power BI Desktop and distributed along with a PBIX project file.

Exercise 1: Create a New Node.js Project for the Barchart Custom Visual

In this exercise, you will begin the process of developing a new custom visual by creating a new project named **barchart** with the PBIVIZ utility and integrating the D3 library.

1. Create a new visual project named **barchart**.

    a) Using the Windows Start menu, launch the **Node.js command prompt**.
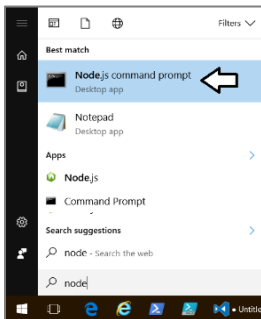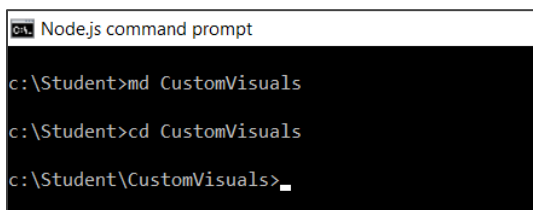

    b) You should now have an open Node.js command prompt.
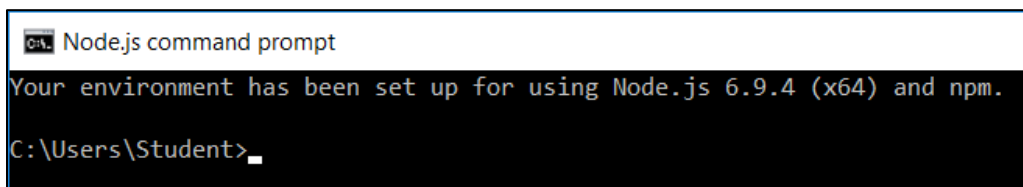
    c) Type and execute the following command to make the current directory back to **C:\Student\CustomVisuals**

    ```
    cd C:\Student\CustomVisuals
    ```
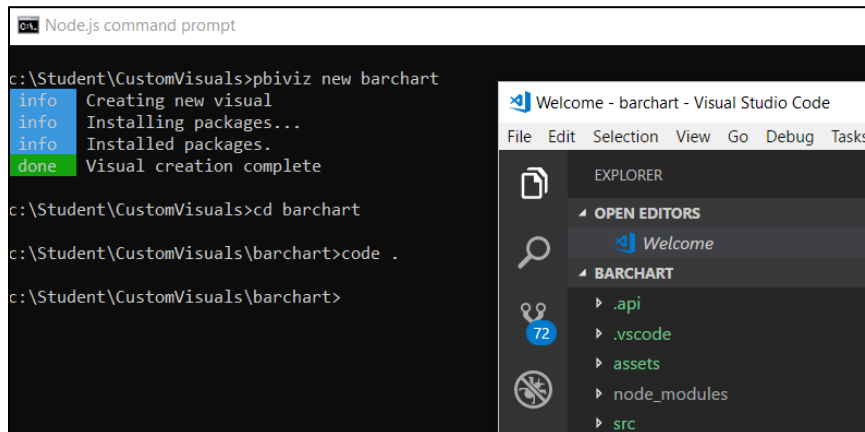

    If a directory with the path of **C:\Student\CustomVisuals** does not already exist, you should create it.
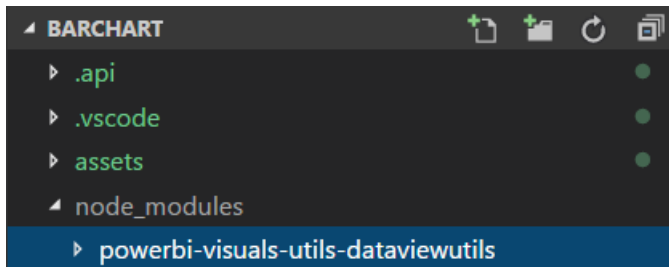
d) Type and execute the following three commands to create a new project named **barchart** and open it in Visual Studio Code.

```
pbiviz new barchart
cd barchart
code .
```

e) You should now have a new project named **barchart** that is open in Visual Studio Code.

2. Inspect the **node-modules** folder.

a) Expand the **node_modules** folder. It should contain a single folder named **powerbi-visuals-utils-dataviewutils**.

3. Add the NPM packages for the D3 library and the D3 typed definition files.

a) Select the **View > Integrated Terminal** command to display the command prompt in Visual Studio Code.

b) In the command prompt, type and execute the following **npm install** command to install the D3 JavaScript library.

```
npm install d3@3 @types/d3@3 --save-dev
```

c) If you refresh the **node-modules** folder, you will see it now contains folders for the D3 library and the D3 type definitions.
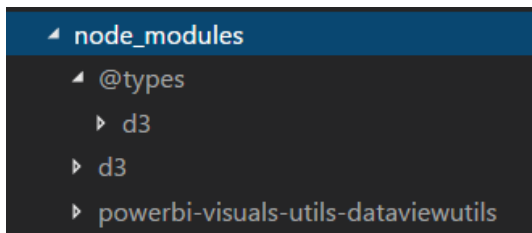

4. Add the NPM package for the Power BI visual formatting utilities.

a) Return to the Node.js command prompt.

b) Type and execute the following command to install the Power BI visual formatting utilities.

```
npm install powerbi-visuals-utils-formattingutils --save-dev
```

Install the formatting utilities will also install the Power BI visual type utilities due to dependencies between these two packages.

c) If you look inside the **node_modules** folder, you'll see two new folders for the formatting utilities and the type utilities.
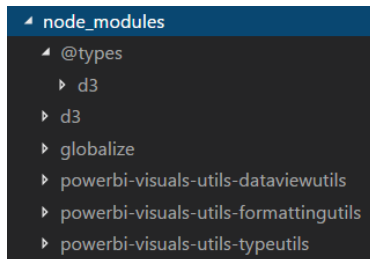
5. Modify the **externalJS** setting in the project's **pbiviz.json** file to reference **d3.js** and the Power BI visual utility libraries.

a) Return to Visual Studio Code.

b) Open the **pbiviz.json** file and locate the **externalJS** setting.

c) Update the array for the **externalJS** setting by adding the paths to the following .js files.

```
"externalJS": [
    "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.js",
    "node_modules/globalize/lib/globalize.js",
    "node_modules/globalize/lib/cultures/globalize.culture.en-US.js",
    "node_modules/powerbi-visuals-utils-typeutils/lib/index.js",
    "node_modules/powerbi-visuals-utils-formattingutils/lib/index.js",
    "node_modules/d3/d3.js"
],
```

d) Save and close **pbiviz.json**.

6. Modify the **files** setting in the project's **tsonfig.json** file to include type definition files for the Power BI utility libraries.

a) Open the **tsconfig.json** file.

b) Move to the bottom of **tsconfig.json** and locate the files property.

c) Add new entries into the array for the **files** property using the following project-relative paths.

```
"files": [
    ".api/v1.11.0/PowerBI-visuals.d.ts",
    "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.d.ts",
    "node_modules/powerbi-visuals-utils-typeutils/lib/index.d.ts",
    "node_modules/powerbi-visuals-utils-formattingutils/lib/index.d.ts",
    "src/settings.ts",
    "src/visual.ts"
]
```

Make sure the API version matches your version of the tools. This example uses **v1.11.0** but you might be using a later version.

d) Save and close **tsconfig.json**.

7. Modify the contents of **capabilities.json**.

a) Open **capabilities.json** in an editor wndow
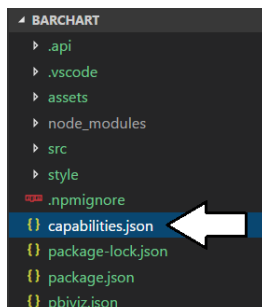
b) The **capabilities.json** contains three top-level sections which are **dataRoles**, **dataViewMappings** and **objects**.

c) Replace the contents of the **dataRoles** section with the following JSON code.

```json
"dataRoles": [
    {
        "displayName": "Bar Grouping",
        "name": "myCategory",
        "kind": "Grouping"
    },
    {
        "displayName": "Bar Measurement",
        "name": "myMeasure",
        "kind": "Measure"
    }
]
```

The **dataRoles** section defines the field wells that will appear in the **Fields** pane. You have defined one field well with a display name of **Bar Grouping** which accepts columns and a second field with a display name of **Bar Measurement** which accepts measures.

d) Replace the contents of the **dataViewMappings** section with the following JSON code.

```json
"dataViewMappings": [
    {
        "conditions": [{
            "myCategory": { "max": 1 },
            "myMeasure": { "max": 1 }
        }],
        "categorical": {
            "categories": {
                "for": { "in": "myCategory" },
                "dataReductionAlgorithm": { "top": {} }
            },
            "values": { "select": [ { "bind": { "to": "myMeasure" } } ] }
        }
    }
]
```

The **dataViewMappings** section can be used to define conditions such as only allow a field well to accept a single field. Data view mappings are also used to define how your visual will map its underlying data. This visual is using categorical mapping.

e) Replace the contents of the **objects** section with the following JSON code.

```json
"objects": {
    "barchartProperties": {
        "displayName": "Bar Chart Properties",
        "properties": {
            "sortBySize": { "displayName": "Sort by Size", "type": {"bool": true } },
            "xAxisFontSize": { "displayName": "X Axis Font Size", "type": { "integer": true } },
            "yAxisFontSize": { "displayName": "Y Axis Font Size", "type": { "integer": true } },
            "barColor": { "displayName": "Bar Color", "type": { "fill": { "solid": { "color": true } } }}
        }
    }
}
```

The **objects** section is used to define a visual's persistent properties. The custom visual you are building has defined a top-level object named **barchartProperties** that contains four child properties named **sortBySize**, **xAxisFontSize**, **yAxisFontSize** and **barColor**.

f) Save your changes and close **capabilities.json**.

8. Modify the **settings.ts** file.

   a) Open the **settings.ts** file in an editor window.
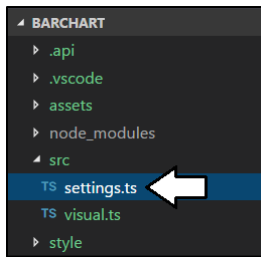
   b) Delete all the existing code inside **settings.ts** and replace it with the following code.

```
module powerbi.extensibility.visual {

  import DataViewObjectsParser = powerbi.extensibility.utils.dataview.DataViewObjectsParser;

  export class VisualSettings extends DataViewObjectsParser {
    public barchartProperties: BarchartProperties = new BarchartProperties();
  }

  export class BarchartProperties {
    sortBySize: boolean = true;;
    xAxisFontSize: number = 10;
    yAxisFontSize: number = 10;
    barColor: Fill = { "solid": { "color": "teal" } };;
  }

}
```

   c) Save your changes and close **settings.ts**.

The **VisualSettings** class extends the Power BI utility class named **DataViewObjectsParser** and is used to abstract away the low-level code required to initialize and read persistent property values. This **VisualSettings** class defines a single property named **barchartProperties** which must exactly match the object named **barchartProperties** which is defined inside **capabilities.json**.

9. Modify the code for the custom visual inside **visual.ts**.

   a) Open **visual.ts** in an editor window.

   b) Delete all the existing code inside **visual.ts** and replace it with the following starter code.

```
module powerbi.extensibility.visual {

  export class Visual implements IVisual {

    private svg: d3.Selection<SVGElement>;
    private svgGroupMain: d3.Selection<SVGElement>;
    private settings: VisualSettings;
    private dataViewInvalidMessage = "Data view not valid. Please add fields.";

    constructor(options: VisualConstructorOptions) {
        // add implementation
    }

    public update(options: VisualUpdateOptions) {
        // add implementation
    }

    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions) :
        VisualObjectInstanceEnumeration {
        // add implementation
    }

  }
}
```

c) Implement the **constructor** using the following code.

```
constructor(options: VisualConstructorOptions) {
  this.settings = VisualSettings.getDefault() as VisualSettings;
  this.svg = d3.select(options.element).append('svg');
  this.svgGroupMain = this.svg.append('g');
  this.svgGroupMain.append("g").append("text")
    .text(this.dataViewInvalidMessage)
    .attr("dominant-baseline", "hanging")
    .attr("font-size", this.settings.barchartProperties.xAxisFontSize)
    .style("fill", this.settings.barchartProperties.barColor.solid.color);

}
```

This constructor retrieves the default settings and displays a simple message. You see the font size and color are determine by the default property values of the **VisualSettings** object.

d) Implement the **update** method using the following code.

```
public update(options: VisualUpdateOptions) {
  this.svgGroupMain.selectAll("g").remove();
  this.svg.attr({
    height: options.viewport.height,
    width: options.viewport.width
  });
  // print out Hello World message
  if (options.dataViews[0]) {
    // get visual setting values
    this.settings = VisualSettings.parse(options.dataViews[0]) as VisualSettings;
    // determine value of barColor property
    var barColor: string  = "";
    if(typeof(this.settings.barchartProperties.barColor)=="object") {
      barColor = this.settings.barchartProperties.barColor.solid.color;
    }
    else {
      barColor = <string>this.settings.barchartProperties.barColor;
    }

    this.svgGroupMain.append("g").append("text")
      .text("Hello World")
      .attr("dominant-baseline", "hanging")
      .attr("font-size", this.settings.barchartProperties.xAxisFontSize)
      .style("fill", barColor);
  }
}
```

The **update** method removes any SVG graphics from earlier calls to **update** and displays a simple "Hello World" message.

e) Implement the **enumerateObjectInstances** function using the following code.

```
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions) :
      VisualObjectInstanceEnumeration {
  // register object properties
  var visualObjects: VisualObjectInstanceEnumerationObject =
   <VisualObjectInstanceEnumerationObject>VisualSettings
                                      .enumerateObjectInstances(this.settings, options);
  // configure spinners for integers properties
  visualObjects.instances[0].validValues = {
    xAxisFontSize: { numberRange: { min: 10, max: 36 } },
    yAxisFontSize: { numberRange: { min: 10, max: 36 } },
  };
  // return visual object collection
  return visualObjects;
}
```

The **enumerateObjectInstances** method initializes persistent properties in the user interface in the Format pane. Note that special treatment is given to the **xAxisFontSize** and **yAxisFontSize** properties so they appear in the **Format** pane as spinner controls.

f) Save your changes to **visual.ts**.

Note that there is a text file named **visual.ts.starter.txt** in the **C:\Student\Modules\06_CustomVisuals\Lab\StarterFiles** folder which contains all the TypeScript code that should be inside the **visual.ts** file at this point. If it helps, you can copy and paste all the TypeScript code from **visual.ts.starter.txt** into the **visual.ts** file in your project to make sure your code is correct.
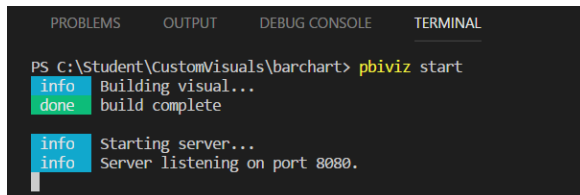
Exercise 2: Run the Project and Test Out the Persistent Properties

In this exercise, you will run the project and observe how the custom properties in this project appear and behave to those users who are using your custom visual in a Power BI report.

1. Use the **start** command of the **pbiviz** utility to start a debugging session for the **barchart** custom visual project.

   a) Make sure you've saved all the files in your project by running the **Save All** command from the Visual Studio Code **File** menu.

   b) Return to the Node.js command prompt in the Integrated Terminal and run **pbiviz start** to start a new debugging session.

   ```
   pbiviz start
   ```

   c) If you examine the console output, you can see that the web server provided by Node.js for debugging has started and is listening for incoming HTTP requests on https://localhost:8080.


2. Test the visual in the Power BI Service using the Developer visual.

   a) Log into our personal workspace at https://app.powerbi.com.

   b) Once you have logged into your personal workspace, drop down the Power BI Setting menu (*it's the one with the gear icon*) and then select the **Settings** menu command as shown in the following dialog.
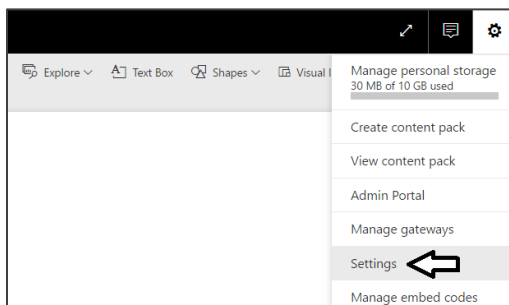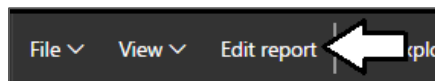

   c) Select **Developer** on the left and then select the **Enable developer visual for testing** checkbox.
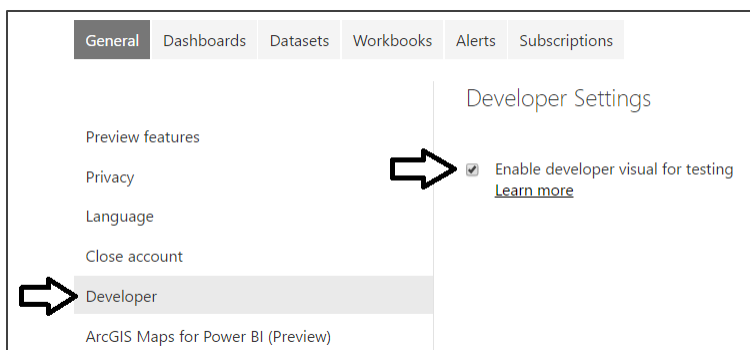
   d) Open the **Wingtip Sales Analysis** report or create a new report using the **Wingtip Sales Analysis** dataset.

e) If you open an existing report, move the report into edit mode by clicking the **Edit report** button.

f) Add a new page to the report so you have a blank report page to work with.

g) Add an instance of the **Developer Visual** to the page.
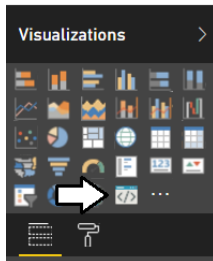
h) At this point, only the constructor has run. You can see that the SVG text element created in the constructor has a color and font size that have been read from the default values of the **VisualSettings** class.
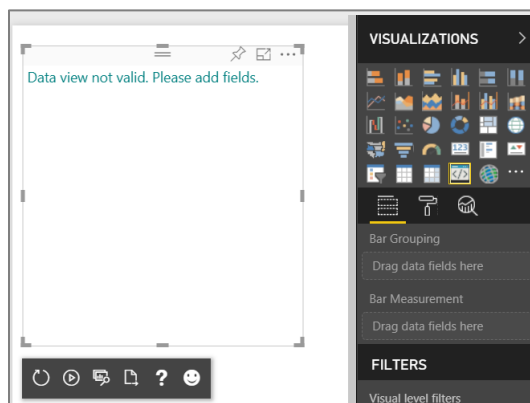

You can see that this visual defines two field wells with display names of **Bar Grouping** and **Bar measurement**. You need to add fields from the Fields pane into these wells for provide the visual with data to display.

i) Add a field into one of the field wells.

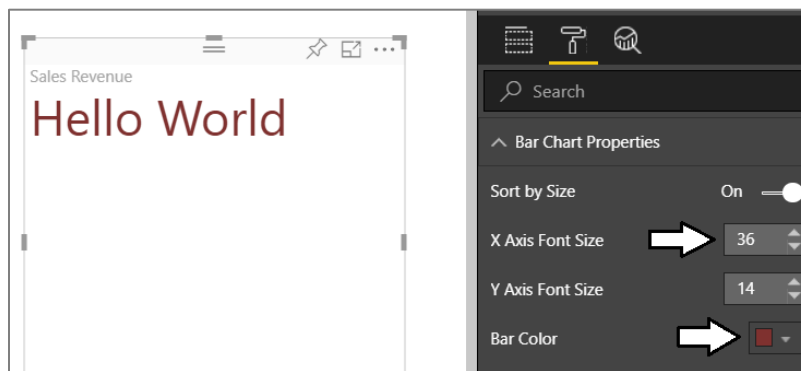j) You should see that the message displayed to the user now changes to "Hello World".
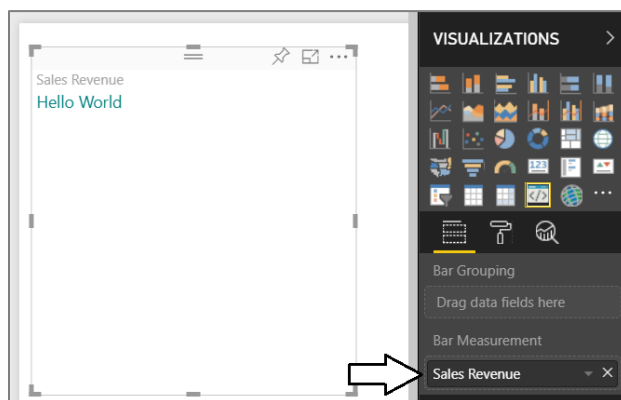
Note that the **update** method does not begin to execute until you have added at least one field into the visual's field wells.

3. Update the visual's custom property values in the **Format** pane.

    a) With the visual selected, click the paint roller icon to navigate to the **Format** pane.

    b) Expand the **Bar Chart Properties** section.

    c) You should see the properties with display names of **Sort by Size**, **X Axis Font Size**, **Y Axis Font Size** and **Bar Color**.
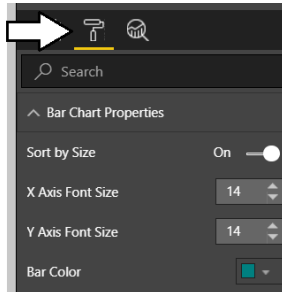
    d) Use the spinner control to increase the value of the **X Axis Font Size** property and make the font larger.

    e) Use the color picker control to modify the **Bar Color** property and change the font color.

    f) When you modify these properties, you should see the font size and color change for the "Hello World" message.

4. Stop the visual debugging session.

    a) Return to Node.js command prompt.

    b) Hold down the **Ctrl** key on the keyboard and then press **C** to interrupt the Node.js debugging session.

    c) When prompted to terminate the session, type **Y** and press **ENTER**.

Now you have created & tested a visual with custom properties. Next you will extend the custom visual project by adding a view model.

Exercise 3: Extend Your Custom Visual Project with a View Model

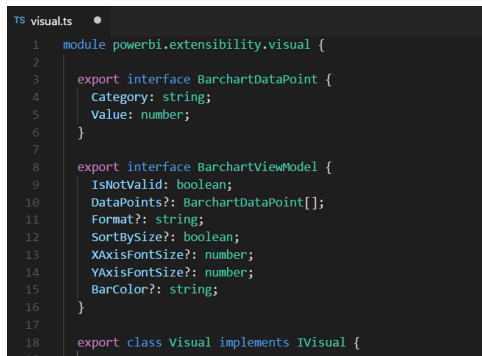In this exercise, you will enhance the design of your custom visual project by adding in a view model.

1. Add two new interfaces definitions into **visual.ts** to define a view model for your project.

   a) Open **visual.ts** in an editor window if it is not already open.

   b) Copy the following interface definition and paste it into **visual.ts** after the module definition but before the **Visual** class.

   ```
   export interface BarchartDataPoint {
     Category: string;
     Value: number;
   }
   ```

   c) Copy the following interface and paste it into **visual.ts** just under the **BarchartDataPoint** interface definition.

   ```
   export interface BarchartViewModel {
       IsNotValid: boolean;
       DataPoints?: BarchartDataPoint[];
       Format?: string;
       SortBySize?: boolean;
       XAxisFontSize?: number;
       YAxisFontSize?: number;
       BarColor?: string;
   }
   ```

   d) At this point, the top of the **visual.ts** file in your project should match the code shown in the following screenshot.
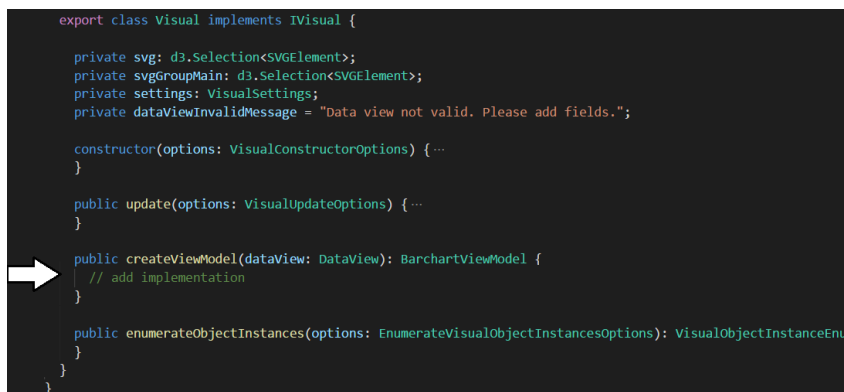

2. Create a new method in your **Visual** class named **createViewModel**.

   a) Copy this starting definition for the **createViewModel** method into the Windows clipboard.

   ```
   public createViewModel(dataView: DataView): BarchartViewModel {
       // add implementation
   }
   ```

   b) Paste the method into the **Visual** class between the **update** method and the **enumerateObjectInstances** method.

Over the next few steps you will copy and paste several code snippets to complete the implementation of **createViewModel**. If you would rather copy and paste all the code for this method at one time, the final implementation of **createViewModel** is available a text file named **createViewModel.ts.txt** inside the folder at **C:\Student\Modules\06_CustomVisuals\Lab\StarterFiles**.

c) Begin your implementation of **createViewModel** by adding the following code to determine whether the view model is valid.

```
// handle case where categorical DataView is not valid
if (typeof dataView === "undefined" ||
    typeof dataView.categorical === "undefined" ||
    typeof dataView.categorical.categories === "undefined" ||
    typeof dataView.categorical.values === "undefined") {
    return { IsNotValid: true };
}
```

d) Move down below in the **update** method and add the following code to retrieve visual settings object.

```
this.settings = VisualSettings.parse(dataView) as VisualSettings;
```

e) Next, add the following code to retrieve arrays with the category names and category values using the categorical mapping.

```
var categoricalDataView: DataViewCategorical = dataView.categorical;
var categoryColumn: DataViewCategoricalColumn = categoricalDataView.categories[0];
var categoryNames: PrimitiveValue[] = categoricalDataView.categories[0].values;
var categoryValues: PrimitiveValue[] = categoricalDataView.values[0].values;
```

f) Next, add the following code to create an array of **BarchartDataPoint** objects to track the name and value of each category.

```
var barchartDataPoints: BarchartDataPoint[] = [];

for (var i = 0; i < categoryValues.length; i++) {
    // get category name and category value
    var category: string = <string>categoryNames[i];
    var categoryValue: number = <number>categoryValues[i];
    // add new data point to barchartDataPoints collection
    barchartDataPoints.push({
        Category: category,
        Value: categoryValue
    });
}
```

The array of **BarchartDataPoint** is an essential part of the view model because it contains all the data required to create a bar chart.

g) Next, add the following code to create retrieve the format string for the measure that was added to the **Bar Measurement** well.

```
var format: string = categoricalDataView.values[0].source.format;
```

This format string will be passed in the view model giving the **update** method to ability to format the Y Axis with the correct formatting.

h) Next, add the following code to create retrieve custom property values for **sortBySize**, **xAxisFontSize** and **yAxisFontSize**.

```
// get persistent property values
var sortBySize: boolean = this.settings.barchartProperties.sortBySize;
var xAxisFontSize: number = this.settings.barchartProperties.xAxisFontSize;
var yAxisFontSize: number = this.settings.barchartProperties.yAxisFontSize;
```

i) Next, add the following code to the retrieve custom property value for **barColor**.

```
var barColor: string = "";
if (typeof (this.settings.barchartProperties.barColor) == "string") {
    barColor = <string>this.settings.barchartProperties.barColor;
}
else {
    barColor = <string>this.settings.barchartProperties.barColor.solid.color;
}
```

The code to retrieve a property value based on the **Fill** type looks a bit strange. For various reasons, once the property values are persisted and retrieved from metadata, it changes the way you must retrieve its value.

j) Complete the implementation of the **update** method by adding the following return statement at the bottom.

```
// return view model to update method
return {
  IsNotValid: false,
  DataPoints: barchartDataPoints,
  Format: format,
  SortBySize: sortBySize,
  BarColor: barColor,
  XAxisFontSize: xAxisFontSize,
  YAxisFontSize: yAxisFontSize
};
```

k) You have now finished implementing the **createViewModel** method.

l) Save your changes to **visual.ts**.

Over the next few steps you will copy and paste several code snippets to rewrite the implementation of **update**. If you would rather copy and paste all the code for this method at one time, the final implementation of **update** is available a text file named **update.ts.txt** inside the folder at **C:\Student\Modules\06_CustomVisuals\Lab\StarterFiles**.

3. Rewrite your implementation of the **update** method.

a) Remove all existing code from **update** and replace it with the following code.

```
public update(options: VisualUpdateOptions) {

    this.svgGroupMain.selectAll("g").remove();
    this.svg.attr({
        height: options.viewport.height,
        width: options.viewport.width
    });

    var viewModel: BarchartViewModel = this.createViewModel(options.dataViews[0]);
    if (viewModel.IsNotValid) {
      // handle case where categorical DataView is not valid
      this.svgGroupMain.append("g").append("text")
        .text(this.dataViewInvalidMessage)
        .attr("dominant-baseline", "hanging")
        .attr("font-size", 14)
        .style("fill", "red");
      return;
    }

    // continue if categorical DataView is valid
}
```

b) Next, add the following code to calculate an offset to the X Axis and the Y Axis.

```
var xAxisOffset: number = viewModel.XAxisFontSize * 6;
var yAxisOffset: number = viewModel.YAxisFontSize * 2;
var paddingSVG: number = 12;
```

c) Next, add the following code to create a plot in which to render the bars for the barchart.

```
// create plot variable to assist with rendering barchart into plot area
var plot = {
    xOffset: paddingSVG + xAxisOffset,
    yOffset: paddingSVG,
    width: options.viewport.width - (paddingSVG * 2) - xAxisOffset,
    height: options.viewport.height - (paddingSVG * 2) - yAxisOffset,
};
```

d) Next, add the following code to adjust the main SVG group coordinates of the plot.

```
// offset x and y coordinates for SVG group used to create bars
this.svgGroupMain.attr({
    height: plot.height,
    width: plot.width,
    transform: 'translate(' + plot.xOffset + ',' + plot.yOffset + ')'
});
```

e) Next, add the following code to create a variable based on the array of **BarchartDataPoint** objects.

```
// convert data from categorical DataView into dataset used with D3 data binding
var barchartDataPoints: BarchartDataPoint[] = viewModel.DataPoints;
```

f) Next, add the following code to create a D3 scale for the X axis.

```
// setup D3 ordinal scale object to map input category names in dataset to output range of x coordinate
var xScale = d3.scale.ordinal()
    .domain(barchartDataPoints.map(function (d) { return d.Category; }))
    .rangeRoundBands([0, plot.width], 0.1);
```

g) Next, add the following code to determine the height of the plot based on the maximum category value.

```
// determine maximum value for the bars in the barchart
var yMax = d3.max(barchartDataPoints, function (d) { return +d.Value * 1.05 });
```

h) Next, add the following code to create a D3 scale for Y axis.

```
// setup D3 linear scale object to map input data values to output range of y coordinate
var yScale = d3.scale.linear()
    .domain([0, yMax])
    .range([plot.height, 0]);
```

i) Next, add the following code to remove all existing bars and axes from earlier calls to **update**.

```
// remove existing SVG elements from previous updates
this.svg.selectAll('.axis').remove();
this.svg.selectAll('.bar').remove();
```

j) Next, add the following code to create the x axis.

```
// create the x axis
var xAxis = d3.svg.axis()
    .scale(xScale)
    .tickSize(0)
    .tickPadding(12)
    .orient('bottom')
```

k) Next, add the following code to draw the x axis.

```
// draw the x axis
this.svgGroupMain
    .append('g')
    .attr('class', 'x axis')
    .style('fill', 'black')
    .attr('transform', 'translate(0,' + (plot.height) + ')')
    .call(xAxis);
```

l) Next, add the following code to create a **valueFormatter** object using the format string that is passed in the view model.

```
// get format string for measure
var valueFormatterFactory = powerbi.extensibility.utils.formatting.valueFormatter;
var valueFormatter = valueFormatterFactory.create({
    format: viewModel.Format,
    formatSingleValues: true
});
```

The a **valueFormatter** object will be used to format the tick values displayed in the y axis.

m) Next, add the following code to create the y axis with the proper formatting.

```
var yAxis = d3.svg.axis()
    .scale(yScale)
    .orient('left')
    .ticks(5)
    .tickSize(0)
    .tickPadding(12)
    .tickFormat(function (d) { return valueFormatter.format(d) });
```

n) Next, add the following code to draw the y axis.

```
this.svgGroupMain
    .append('g')
    .attr('class', 'y axis')
    .style('fill', 'black')
    .call(yAxis);
```

o) Next, add the following D3 code which calls the **data** method to refresh the set of bar objects to be rendered in the bar chart.

```
// use dat method to refresh collection of bars
var svgGroupBars = this.svgGroupMain
    .append('g')
    .selectAll('.bar')
    .data(barchartDataPoints)
```

p) Next, add the following code to render the bars in the bar chart.

```
svgGroupBars.enter()
    .append('rect')
    .attr('class', 'bar')
    .attr('fill', viewModel.BarColor)
    .attr('stroke', 'black')
    .attr('x', function (d) {
        return xScale(d.Category);
    })
    .attr('width', xScale.rangeBand())
    .attr('y', function (d) {
        return yScale(d.Value);
    })
    .attr('height', function (d) {
        return plot.height - yScale(d.Value);
    });
```

q) Next, add the following code to remove any bars that are no longer in use.

```
svgGroupBars
    .exit()
    .remove();
```

r) Finally, add the following code to adjust the font size used to render the x axis and the y axis.

```
d3.select(".x.axis").selectAll("text").style({ "font-size": viewModel.XAxisFontSize });
d3.select(".y.axis").selectAll("text").style({ "font-size": viewModel.YAxisFontSize });
```

s) You have now completed the implementation of the update method.

t) Save your changes to **visual.ts**.

## Exercise 4: Test the Barchart Custom Visual in the Power BI Service

In this exercise, you will enhance the design of your custom visual project b adding in a view model.

1. Use the **start** command of the **pbiviz** utility to start a debugging session for the **barchart** custom visual project.

    a) Make sure you've saved all the files in your project by running the **Save All** command from the Visual Studio Code **File** menu.

    b) Return to the Node.js command prompt in the Integrated Terminal and run **pbiviz start** to start a new debugging session.

   ```
    pbiviz start
    ```

    c)  If you examine the console output, you can see that the web server provided by Node.js for debugging has started and is listening for incoming HTTP requests on https://localhost:8080.

2.  Test the **barchart** visual in the Power BI Service using the Developer visual.

    a)  Log into our personal workspace at https://app.powerbi.com.

    b)  Now navigate to the report you worked with earlier in exercise 2.

    c)  If the report is in read-only mode, move the report into edit mode by clicking the **Edit report** button.

    d)  Add a new page to the report so you have a blank report page to work with.

e) Add an instance of the **Developer Visual** to the page.
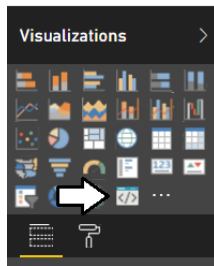
f) When you first add an instance of the **barchart** visual, you should see the display message created in the constructor.
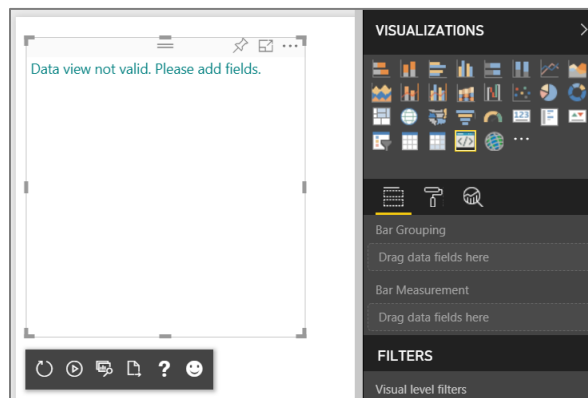
g) Add a column to the **Bar Grouping** field well and then add a measure into the **Bar Measurement** field. Once you do this, the barchart should properly render with an x axis and a y axis.
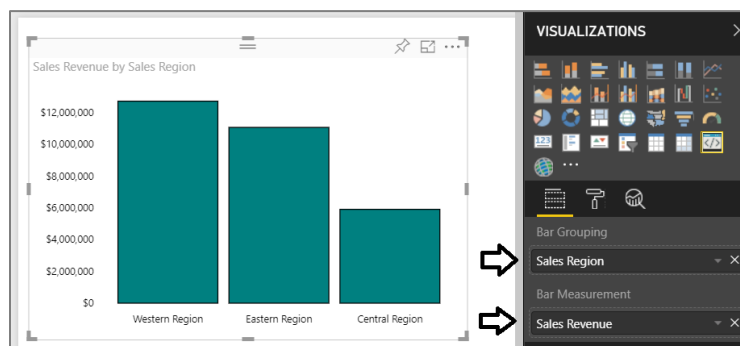
h) Navigate to the **Format** pane to see the visual's custom properties. Change the property values for the **X Axis Font Size** property, the **Y Axis Font Size** property and the **Bar Color** property and see how they affect the visual's rendering.
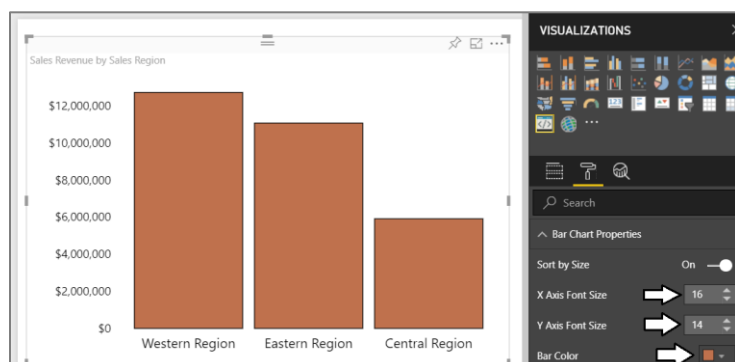
      i)   Change the property value for the **Sort by Size** property. You should see that property changes the sorting of the bars in the bar chart between being sorted by bar size versus being sorting alphabetically by category name.
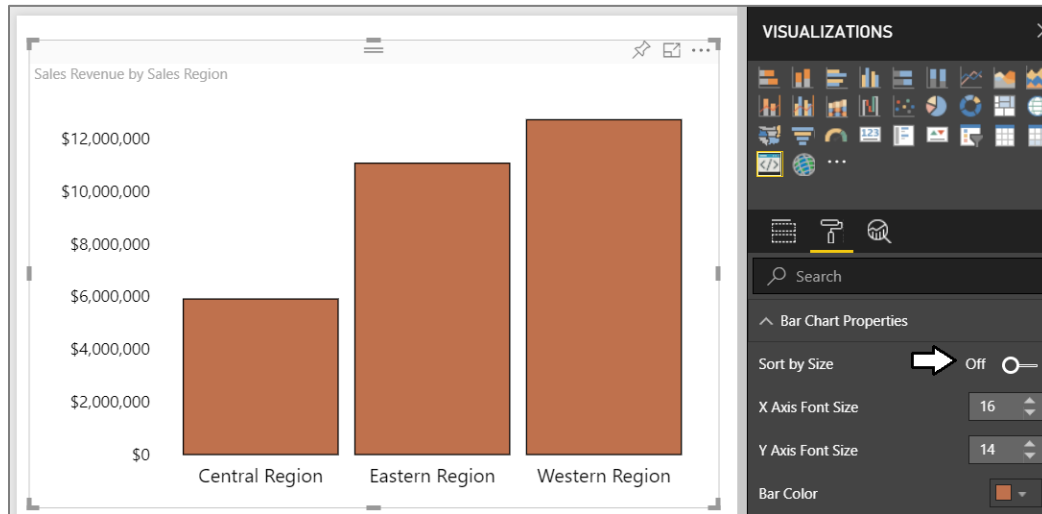


3.   Stop the visual debugging session.

    a)   Return to Node.js command prompt.

    b)   Hold down the **Ctrl** key on the keyboard and then press **C** to interrupt the Node.js debugging session.



    c)   When prompted to terminate the session, type **Y** and press **ENTER**.

Now you have created & tested your **barchart** visual. Next you will package the **barchart** visual for distribution.

## Exercise 5: Package Your Custom Visual for Distribution

In this exercise, you will learn how to package a custom visual for distribution by building a PBIVIZ file using the **pbiviz package** command. You will also move through the steps and importing and testing a packaged visual using Power BI Desktop.

1.   Add a custom icon for your barchart visual.

    a)   Using Windows Explorer, locate the file named **icon.png** at the following path.

```
D:\Git\PBD365\Modules\06_CustomVisuals\Lab\StarterFiles\icon.png
```

    b)   Copy **icon.png** into the **assets** folder of the barchart to **replace** the **icon.png** file that was added to the project by **pbiviz**.
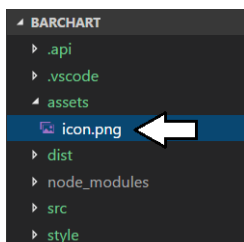
2. Run the **pbiviz package** command to build your project into a PBIVIZ file for distribution.

    a) Return to the Node.js command prompt in the Integrated Terminal.

   b) Run the **pbiviz package** to start a new debugging session.

    ```
    pbiviz package
    ```

    c) After you run the **pbiviz package** command, you should see a new file named **barchart.pbiviz** in the project's **dist** folder
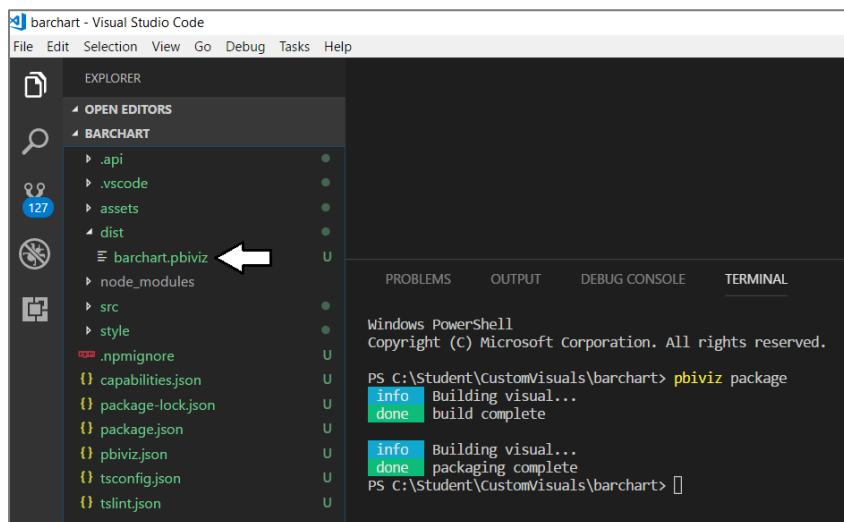

> The new package file named **barchart.pbiviz** is the file you will distribute to share your visual with the world. In the final step, you will open Power BI Desktop and import your custom visual into a report to test it out.
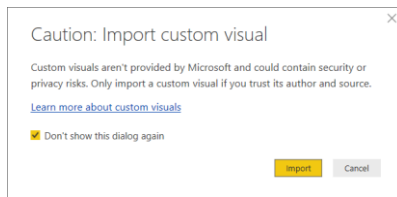
3. Launch Power BI Desktop.

    a) Open a project file such as **C:\Student\Project\Wingtip Sales Analysis.pbix**.

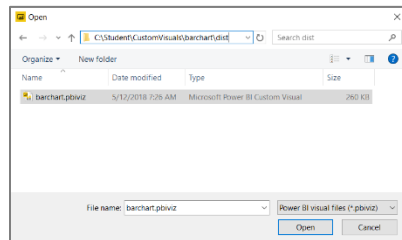   b) Click to ellipse flyout menu at the of the **VISUALIZATIONS** list.
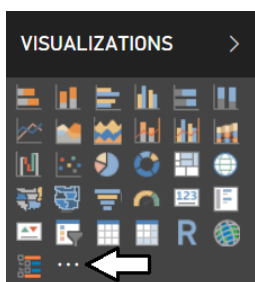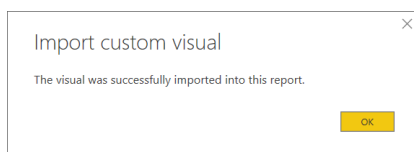

    c) Select the **Import from file** command.
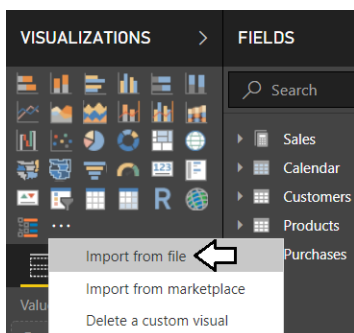
d) When you see the **Caution: Import custom** visual dialog, check **Don't show this dialog again** and click **Import**.
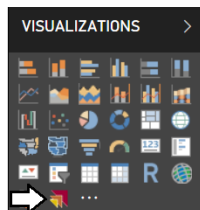

e) In the **Open** dialog, select the **barchart.pbiviz** file from the folder with the path of **C:\Student\CustomVisuals\barchart\dist**.
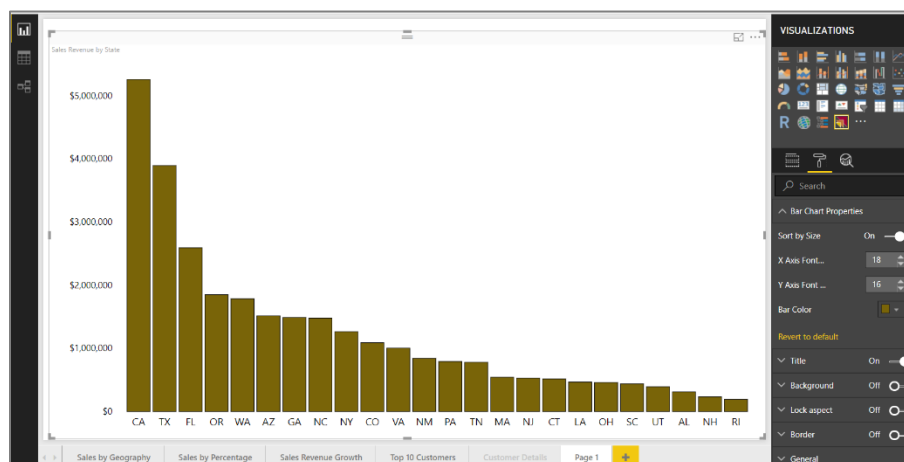

f) You should see a dialog that indicates the custom visual has been imported into the current project.

g) You should see the barchart visual in the VISUALIZATIONS list with its custom icon.

h) Add the barchart visual to a report page and test it out by changing its custom property values.

Congratulations. You have now completed the custom visuals lab.