Histogram Chart w/ PI Web API & Highcharts

The following example is used to create a PI Vision symbol that uses Highcharts and utilizes your PI Web API server for data. These instructions build off the Simple Value Symbol Instructions, so please review those first.

1. Create a new file called sym-histogram.js in your PI Vision installation folder, `INSTALLATION_FOLDER\Scripts\app\editor\symbols\ext`. If the `ext` folder does not exist, create it.

2. Below is the basic skeleton of a new PI Vision symbol for the histogram chart. It sets up the `typeName`, `datasourceBehavior`, and `getDefaultConfig` definition options and registers them with the PI Vision application. For the `DataShape`, we are using the `Custom` data shape, which tells PI Vision that the data for this symbol will be retrieved via PI Web API. For the 'datasourceBehavior' we are using the `Single` behavior which will tell PI Vision this symbol only expects one datasource at a time.

```
(function (PV) {
'use strict';

function symbolVis() { }
PV.deriveVisualizationFromBase(symbolVis);

//symbol definition
var defintion = {
    typeName: 'histogram',
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
    visObjectType: symbolVis,
    getDefaultConfig: function() {
        return {
            DataShape: 'Custom',
            Height: 200,
            Width: 400
        };
    }
};

PV.symbolCatalog.register(defintion);
})(window.PIVisualization);
```

3. The next step is to create the HTML template for this symbol. The chart that we will be using only needs a `div` tag to attach to. So we will create a HTML file in the same directory as our JavaScript file and name it `sym-histogram-template.html`. We are setting the height and width of this `div` to take up the full space available.

```
<div id="histogram" style="width:100%;height:100%"></div>
```

4. Now we need to initialize the histogram symbol. We will add an `init` to the symbol's prototype and define the `init` function. The `init` function will have stubs for data updates and resizing events. The resize function is called by the PI Vision infrastructure anytime the symbol is resized. The resize function is passed the new width and height of the symbol. The dataUpdate function is called by the PI Vision infrastructure anytime the data for the symbol is retrieved via the PI Vision data pump. This data pump by default retrieves data every 5 seconds, but can be configured within the PI Vision database settings. As stated before, the use of the `Custom` datashape will tell PI Vision to use the PI Web API data pump for data retrieval.

```
(function (PV) {
'use strict';

function symbolVis() { }
PV.deriveVisualizationFromBase(symbolVis);

//symbol initialization
symbolVis.prototype.init = function (scope, elem) {
    this.onDataUpdate = dataUpdate;
    this.onResize = resize;

    scope.histogram = null;

    //fires on every data pump refresh
    //this is where you want to include any logic to update the data on your symbol
    function dataUpdate(data) {
    }

    //fires whenever the symbol is resized
    //this is where you want to include any logic, to resize the elements within your symbol
    function resize(width, height) {
    }

};

//symbol definition
```

```
var defintion = {
    typeName: 'histogram',
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
    visObjectType: symbolVis,
    getDefaultConfig: function() {
        return {
            DataShape: 'Custom',
            Height: 200,
            Width: 400
        };
    }
};

PV.symbolCatalog.register(defintion);
})(window.PIVisualization);
```

5. Next we must include the HighCharts library so that our symbol can use it. This code comes from Highcharts 4.2.3. Extract the files from the zip and place highcharts.js into `INSTALLATION_FOLDER\Scripts\app\editor\symbols\ext\libraries`. This will make Highcharts available to PI Vision symbols.

6. Now we can begin to define the `getCustomQueries` method for the symbol. This method will be used by the PI Vision infrastructure to build the relavant PI Web API batch queries that will be executed on every data refresh. In this example, I construct a batch query that will use the PI Web API Attributes end-point to retrieve the relavant information for the data source attached to the symbol. This data source is assigned to the symbol when it is created, and uses the path of the attribute that was dragged onto the display during symbol creation. If you are not very familiar with PI Web API batch queries please read the documentation here Batch.

   I also add a second batch fragment to the batch query array, that will utilize the RecordedData end-point link provided from the Attributes end-point. This will be the data that is manipulated and bound to the histogram chart.

   You can also see that I am utilizing the `displayStartTime` and `displayEndTime` parameters of the `getCustomQueries` function. These parameters contain the start and end time information from the timebar on the current PI Vision display. I use these parameters to filter the time range for my RecordedData query.

   Finally, I have added the `getWebAPIUrl` method, which utilizes the `window.PIVisualization.ClientSettings.PIWebAPIUrl` global PI Vision setting to retrieve the PI Web API URL for the search service that PI Vision is utilizing. This server can be changed in the Web.Config for PI Vision, by updating the `SearchServiceURL` setting.

```
(function (PV) {
'use strict';

function symbolVis() { }
PV.deriveVisualizationFromBase(symbolVis);

//symbol initialization
symbolVis.prototype.init = function (scope, elem) {
    this.onDataUpdate = dataUpdate;
    this.onResize = resize;

    scope.histogram = null;

    //fires on every data pump refresh
    //this is where you want to include any logic to update the data on your symbol
    function dataUpdate(data) {
    }

    //fires whenever the symbol is resized
    //this is where you want to include any logic, to resize the elements within your symbol
    function resize(width, height) {
    }

};


//method that provides the PI Web API batch query to the data pump on every refresh
//retrieves the recorded data for the attribute represented in the symbol for the length of the display timebar dura
function getCustomQueries(sym, displayStartTime, displayEndTime) {
    var batchQueries = [];
    if (sym.DataSources && sym.DataSources.length) {
        var url = getWebAPIUrl('/attributes/?path=' + sym.DataSources[0].substring(3));
        batchQueries.push({
            "Attributes": {
                "Method": "GET",
                "Resource": url
            },
            "Data": {
                "Method": "GET",
                "RequestTemplate": { "Resource": "{0}?startTime=" + displayStartTime + "&endTime=" + displayEndTime
```

```
                "Parameters": ["$.Attributes.Content.Links.RecordedData"],
                "ParentIds": ["Attributes"]
            }
        });


    }
    return batchQueries;
}

//retrieves the global PIWebAPIUrl setting, which is represented by the SearchServiceURL setting in the Web.Config
function getWebAPIUrl(url) {
    return window.PIVisualization.ClientSettings.PIWebAPIUrl + url;
}

//symbol definition
var defintion = {
    typeName: 'histogram',
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
    visObjectType: symbolVis,
    getDefaultConfig: function() {
        return {
            DataShape: 'Custom',
            Height: 200,
            Width: 400
        };
    },
    getCustomQueries: getCustomQueries
};

PV.symbolCatalog.register(defintion);
})(window.PIVisualization);
```

7. Next we want take the data that is passed to the PI Vision dataUpdate function and convert this to a format that the chart is expecting.

```
 //converts the pi web api data into grouped data for the highcharts histogram
    function histogram(data, step) {
        var histo = {},
            x,
```

```
        i,
        arr = [];

    var items = data.Data.Data.Content.Items[0].Content.Items;

    // Group down
    for (i = 0; i < items.length; i++) {
        x = Math.floor(items[i].Value / step) * step;
        if (!histo[x]) {
            histo[x] = 0;
        }
        histo[x]++;
    }

    // Make the histo group into an array
    for (x in histo) {
        if (histo.hasOwnProperty((x))) {
            arr.push([parseFloat(x), histo[x]]);
        }
    }

    // Finally, sort the array
    arr.sort(function (a, b) {
        return a[0] - b[0];
    });

    return arr;
}
```

8. Now we want to use this histogram function in the dataUpdate method. We add this function to `init`. On the first update, `scope.histogram` will not be defined, so we initialize the chart with the data returned from the histogram method. On subsequent updates, we only update the data, and the title of the chart to include the appropriate attribute information.

```
//fires on every data pump refresh
//this is where you want to include any logic to update the data on your symbol
function dataUpdate(data) {
    if (data && data.Data && data.Data.Attributes) {
        var container = $(elem).find('#histogram');
```

```
            var series = [{
                name: data.Data.Attributes.Content.Name,
                type: 'column',
                data: histogram(data, 1),
                pointPadding: 0,
                groupPadding: 0,
                pointPlacement: 'between'
            }];

            var title = data.Data.Attributes.Content.Path.split('\\').pop();

            if (!scope.histogram) {
                scope.histogram = new Highcharts.Chart({
                    chart: {
                        renderTo: container[0],
                        type: 'column',
                        animation: false
                    },
                    title: {
                        text: title
                        },
                    xAxis: {
                        gridLineWidth: 1
                    },
                    yAxis: [{
                        title: {
                            text: 'Frequency'
                        }
                    }],
                    series: series
                });
            } else {
                scope.histogram.series[0].update(series);
                scope.histogram.setTitle({ text: title });
            }
        }
    }
```

9. Lastly, we want to handle resizing the chart appropriately. To do this, we need to use Highcharts setSize method.

```
    //fires whenever the symbol is resized
    //this is where you want to include any logic, to resize the elements within your symbol
    function resize(width, height) {
       if(scope.histogram) {
        scope.histogram.setSize(width, height);
       }
    }
```

10. (Optional) You can also choose to customize the theme of the histogram, by applying a custom theme object to the `Highcharts.theme` variable. In this case, I change my theme to use a darker theme, to better match the default colors of a PI Vision display.

```
//Highcharts theme to make my symbol have a dark theme (optional)
      Highcharts.theme = {
          colors: ['#2b908f', '#90ee7e', '#f45b5b', '#7798BF', '#aaeeee', '#ff0066', '#eeaaee',
             '#55BF3B', '#DF5353', '#7798BF', '#aaeeee'],
          chart: {
              backgroundColor: {
                  linearGradient: { x1: 0, y1: 0, x2: 1, y2: 1 },
                  stops: [
                     [0, '#2a2a2b'],
                     [1, '#3e3e40']
                  ]
              },
              style: {
                  fontFamily: '\'Unica One\', sans-serif'
              },
              plotBorderColor: '#606063'
          },
          title: {
              style: {
                  color: '#E0E0E3',
                  textTransform: 'uppercase',
                  fontSize: '20px'
              }
          },
          subtitle: {
              style: {
                  color: '#E0E0E3',
                  textTransform: 'uppercase'
```

```
    }
},
xAxis: {
    gridLineColor: '#707073',
    labels: {
        style: {
            color: '#E0E0E3'
        }
    },
    lineColor: '#707073',
    minorGridLineColor: '#505053',
    tickColor: '#707073',
    title: {
        style: {
            color: '#A0A0A3'

        }
    }
},
yAxis: {
    gridLineColor: '#707073',
    labels: {
        style: {
            color: '#E0E0E3'
        }
    },
    lineColor: '#707073',
    minorGridLineColor: '#505053',
    tickColor: '#707073',
    tickWidth: 1,
    title: {
        style: {
            color: '#A0A0A3'
        }
    }
},
tooltip: {
    backgroundColor: 'rgba(0, 0, 0, 0.85)',
    style: {
        color: '#F0F0F0'
```

```
    }
},
plotOptions: {
    series: {
        dataLabels: {
            color: '#B0B0B3'
        },
        marker: {
            lineColor: '#333'
        }
    },
    boxplot: {
        fillColor: '#505053'
    },
    candlestick: {
        lineColor: 'white'
    },
    errorbar: {
        color: 'white'
    }
},
legend: {
    itemStyle: {
        color: '#E0E0E3'
    },
    itemHoverStyle: {
        color: '#FFF'
    },
    itemHiddenStyle: {
        color: '#606063'
    }
},
credits: {
    style: {
        color: '#666'
    }
},
labels: {
    style: {
        color: '#707073'
```

```
    }
},

drilldown: {
    activeAxisLabelStyle: {
        color: '#F0F0F3'
    },
    activeDataLabelStyle: {
        color: '#F0F0F3'
    }
},

navigation: {
    buttonOptions: {
        symbolStroke: '#DDDDDD',
        theme: {
            fill: '#505053'
        }
    }
},

// scroll charts
rangeSelector: {
    buttonTheme: {
        fill: '#505053',
        stroke: '#000000',
        style: {
            color: '#CCC'
        },
        states: {
            hover: {
                fill: '#707073',
                stroke: '#000000',
                style: {
                    color: 'white'
                }
            },
            select: {
                fill: '#000003',
                stroke: '#000000',
```

```
                style: {
                    color: 'white'
                }
            }
        }
    },
    inputBoxBorderColor: '#505053',
    inputStyle: {
        backgroundColor: '#333',
        color: 'silver'
    },
    labelStyle: {
        color: 'silver'
    }
},

navigator: {
    handles: {
        backgroundColor: '#666',
        borderColor: '#AAA'
    },
    outlineColor: '#CCC',
    maskFill: 'rgba(255,255,255,0.1)',
    series: {
        color: '#7798BF',
        lineColor: '#A6C7ED'
    },
    xAxis: {
        gridLineColor: '#505053'
    }
},

scrollbar: {
    barBackgroundColor: '#808083',
    barBorderColor: '#808083',
    buttonArrowColor: '#CCC',
    buttonBackgroundColor: '#606063',
    buttonBorderColor: '#606063',
    rifleColor: '#FFF',
    trackBackgroundColor: '#404043',
```

```
            trackBorderColor: '#404043'
        },

        // special colors for some of the
        legendBackgroundColor: 'rgba(0, 0, 0, 0.5)',
        background2: '#505053',
        dataLabelsColor: '#B0B0B3',
        textColor: '#C0C0C0',
        contrastTextColor: '#F0F0F3',
        maskColor: 'rgba(255,255,255,0.3)'
    };

    // Apply the theme
    Highcharts.setOptions(Highcharts.theme);
```

Below is the full version of the implementation file.

```
(function (PV) {
'use strict';

function symbolVis() { }
PV.deriveVisualizationFromBase(symbolVis);

//symbol initialization
symbolVis.prototype.init = function (scope, elem) {
    this.onDataUpdate = dataUpdate;
    this.onResize = resize;

    scope.histogram = null;

    //Highcharts theme to make my symbol have a dark theme (optional)
    Highcharts.theme = {
        colors: ['#2b908f', '#90ee7e', '#f45b5b', '#7798BF', '#aaeeee', '#ff0066', '#eeaaee',
           '#55BF3B', '#DF5353', '#7798BF', '#aaeeee'],
        chart: {
            backgroundColor: {
                linearGradient: { x1: 0, y1: 0, x2: 1, y2: 1 },
                stops: [
                   [0, '#2a2a2b'],
```

```
            [1, '#3e3e40']
        ]
    },
    style: {
        fontFamily: '\'Unica One\', sans-serif'
    },
    plotBorderColor: '#606063'
},
title: {
    style: {
        color: '#E0E0E3',
        textTransform: 'uppercase',
        fontSize: '20px'
    }
},
subtitle: {
    style: {
        color: '#E0E0E3',
        textTransform: 'uppercase'
    }
},
xAxis: {
    gridLineColor: '#707073',
    labels: {
        style: {
            color: '#E0E0E3'
        }
    },
    lineColor: '#707073',
    minorGridLineColor: '#505053',
    tickColor: '#707073',
    title: {
        style: {
            color: '#A0A0A3'

        }
    }
},
yAxis: {
    gridLineColor: '#707073',
```

```
    labels: {
        style: {
            color: '#E0E0E3'
        }
    },
    lineColor: '#707073',
    minorGridLineColor: '#505053',
    tickColor: '#707073',
    tickWidth: 1,
    title: {
        style: {
            color: '#A0A0A3'
        }
    }
},
tooltip: {
    backgroundColor: 'rgba(0, 0, 0, 0.85)',
    style: {
        color: '#F0F0F0'
    }
},
plotOptions: {
    series: {
        dataLabels: {
            color: '#B0B0B3'
        },
        marker: {
            lineColor: '#333'
        }
    },
    boxplot: {
        fillColor: '#505053'
    },
    candlestick: {
        lineColor: 'white'
    },
    errorbar: {
        color: 'white'
    }
},
```

```
legend: {
    itemStyle: {
        color: '#E0E0E3'
    },
    itemHoverStyle: {
        color: '#FFF'
    },
    itemHiddenStyle: {
        color: '#606063'
    }
},
credits: {
    style: {
        color: '#666'
    }
},
labels: {
    style: {
        color: '#707073'
    }
},

drilldown: {
    activeAxisLabelStyle: {
        color: '#F0F0F3'
    },
    activeDataLabelStyle: {
        color: '#F0F0F3'
    }
},

navigation: {
    buttonOptions: {
        symbolStroke: '#DDDDDD',
        theme: {
            fill: '#505053'
        }
    }
},
```

```
// scroll charts
rangeSelector: {
    buttonTheme: {
        fill: '#505053',
        stroke: '#000000',
        style: {
            color: '#CCC'
        },
        states: {
            hover: {
                fill: '#707073',
                stroke: '#000000',
                style: {
                    color: 'white'
                }
            },
            select: {
                fill: '#000003',
                stroke: '#000000',
                style: {
                    color: 'white'
                }
            }
        }
    },
    inputBoxBorderColor: '#505053',
    inputStyle: {
        backgroundColor: '#333',
        color: 'silver'
    },
    labelStyle: {
        color: 'silver'
    }
},

navigator: {
    handles: {
        backgroundColor: '#666',
        borderColor: '#AAA'
    },
```

```
        outlineColor: '#CCC',
        maskFill: 'rgba(255,255,255,0.1)',
        series: {
            color: '#7798BF',
            lineColor: '#A6C7ED'
        },
        xAxis: {
            gridLineColor: '#505053'
        }
    },

    scrollbar: {
        barBackgroundColor: '#808083',
        barBorderColor: '#808083',
        buttonArrowColor: '#CCC',
        buttonBackgroundColor: '#606063',
        buttonBorderColor: '#606063',
        rifleColor: '#FFF',
        trackBackgroundColor: '#404043',
        trackBorderColor: '#404043'
    },

    // special colors for some of the
    legendBackgroundColor: 'rgba(0, 0, 0, 0.5)',
    background2: '#505053',
    dataLabelsColor: '#B0B0B3',
    textColor: '#C0C0C0',
    contrastTextColor: '#F0F0F3',
    maskColor: 'rgba(255,255,255,0.3)'
};

// Apply the theme
Highcharts.setOptions(Highcharts.theme);

//converts the pi web api data into grouped data for the highcharts histogram
function histogram(data, step) {
    var histo = {},
        x,
        i,
        arr = [];
```

```
    var items = data.Data.Data.Content.Items[0].Content.Items;

    // Group down
    for (i = 0; i < items.length; i++) {
        x = Math.floor(items[i].Value / step) * step;
        if (!histo[x]) {
            histo[x] = 0;
        }
        histo[x]++;
    }

    // Make the histo group into an array
    for (x in histo) {
        if (histo.hasOwnProperty((x))) {
            arr.push([parseFloat(x), histo[x]]);
        }
    }

    // Finally, sort the array
    arr.sort(function (a, b) {
        return a[0] - b[0];
    });

    return arr;
}

//fires on every data pump refresh
//this is where you want to include any logic to update the data on your symbol
function dataUpdate(data) {
    if (data && data.Data && data.Data.Attributes) {
        var container = $(elem).find('#histogram');

        var series = [{
            name: data.Data.Attributes.Content.Name,
            type: 'column',
            data: histogram(data, 1),
            pointPadding: 0,
            groupPadding: 0,
            pointPlacement: 'between'
```

```
            }];

            var title = data.Data.Attributes.Content.Path.split('\\').pop();

            if (!scope.histogram) {
                scope.histogram = new Highcharts.Chart({
                    chart: {
                        renderTo: container[0],
                        type: 'column',
                        animation: false
                    },
                    title: {
                        text: title
                        },
                    xAxis: {
                        gridLineWidth: 1
                    },
                    yAxis: [{
                        title: {
                            text: 'Frequency'
                        }
                    }],
                    series: series
                });
            } else {
                scope.histogram.series[0].update(series);
                scope.histogram.setTitle({ text: title });
            }
        }
    }

    //fires whenever the symbol is resized
    //this is where you want to include any logic, to resize the elements within your symbol
    function resize(width, height) {
        if (scope.histogram) {
            scope.histogram.setSize(width, height);
        }
    }

};
```

```
//method that provides the PI Web API batch query to the data pump on every refresh
//retrieves the recorded data for the attribute represented in the symbol for the length of the display timebar dura
function getCustomQueries(sym, displayStartTime, displayEndTime) {
    var batchQueries = [];
    if (sym.DataSources && sym.DataSources.length) {
        var url = getWebAPIUrl('/attributes/?path=' + sym.DataSources[0].substring(3));
        batchQueries.push({
            "Attributes": {
                "Method": "GET",
                "Resource": url
            },
            "Data": {
                "Method": "GET",
                "RequestTemplate": { "Resource": "{0}?startTime=" + displayStartTime + "&endTime=" + displayEndTime},
                "Parameters": ["$.Attributes.Content.Links.RecordedData"],
                "ParentIds": ["Attributes"]
            }
        });


    }
    return batchQueries;
}

//retrieves the global PIWebAPIUrl setting, which is represented by the SearchServiceURL setting in the Web.Config
function getWebAPIUrl(url) {
    return window.PIVisualization.ClientSettings.PIWebAPIUrl + url;
}

//symbol definition
var defintion = {
    typeName: 'histogram',
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
    visObjectType: symbolVis,
    getDefaultConfig: function() {
        return {
            DataShape: 'Custom',
            Height: 200,
            Width: 400
```

```
            };
        },
        getCustomQueries: getCustomQueries
    };

    PV.symbolCatalog.register(defintion);
})(window.PIVisualization);
```