Leveraging PI Vision Extensibility

How to create custom symbols

From the OSIsoft Team:

Robert Schmitz, Product Support Engineer

Questions we want to answer

- How do we add a custom symbol to PI Vision?
- How can we get data to the custom symbol?
- How can we leverage other libraries in our custom symbols to avoid starting from scratch?
- How can we data from other sources?

Adding a symbol to PI Vision

Adding a Custom Symbol to PI Vision

1. Create a folder for our custom symbols
   - %pihome64%PIVision\Scripts\App\editor\symbols**\ext**
2. Create a file for the symbol's logic, and another one for the symbol's layout
   - Logic: Sym-YourSymbolName.js
   - Layout: Sym-YourSymbolName-template.html
   - **Must follow naming convention**

Adding a Custom Symbol to PI Vision

3. Add the initializing code for the symbol
   - Lot of code here, all of it stays the same across all custom symbols
   - Can think of this as a template

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

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

    var definition = {
        typeName: "simplevalue",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) { };

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

Quick Cheat Sheet

- Symbol Name
- Height and Width in no. pixels
- How many tags it consumes

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

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

    var definition = {
        typeName: "simplevalue",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) { };

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

Adding a Custom Symbol to PI Vision

4. Add code to our layout file
   - This can be just about any valid HTML code

```
1  <div>
2      "Hello, Robert! 6*7 is {{ 6*7 }}"
3  </div>
```

Getting PI Data to the symbol

Getting data to Layout

Using AngularJS “Magic”

1. Add some fake data to our Logic file
2. Use AngularJS’s ‘scope’ variable to make the fake data in the logic file available in our layout file
3. Reference the scope variable in the layout file

Using AngularJS “Magic”

1. Add some fake data

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

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

    var definition = {
        typeName: "simplevalue",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) {
    };

    var dataitem = {
        time: "22-Aug-2017 10:00:00",
        value: 100
    }

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

Using AngularJS “Magic”

2. Use AngularJS’s ‘scope’ to reference data in other files

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

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

    var definition = {
        typeName: "simplevalue",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) {
        scope.time = dataitem.time;
        scope.value = dataitem.value;
    };

    var dataitem = {
        time: "22-Aug-2017 10:00:00",
        value: 100
    }

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

Using AngularJS “Magic”

3. Reference the scope variable in the layout file
   - Can use {{ }} to reference anything on the scope variable

```
1  <div>
2      <h3>Value: {{ value }}</h3>
3      <h3>Time: {{ time }}</h3>
4  </div>
```

Getting PI Data to the symbol

Retrieving Data from PI

1. Add a this.onDataUpdate function to retrieve data
   - Built-in event handler
   - Retrieves data in the same way for native Vision symbols
2. Map this data to AngularJS's scope to make it available in the layout file
3. Reference the scope variable in the layout file

Retrieving Data from PI

1. Add a this.onDataUpdate function to retrieve data

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

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

    var definition = {
        typeName: "challenge3",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                DataShape: "Value",
                Height: 150,
                Width: 400
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) {
        this.onDataUpdate = dataUpdate

        function dataUpdate(data) { }
    };

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

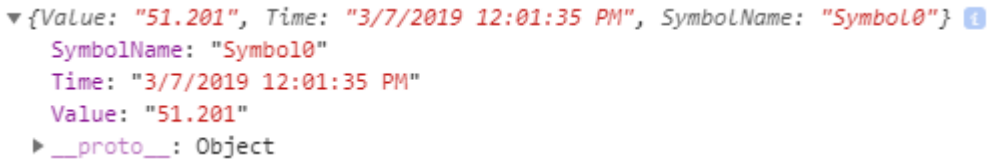
Retrieving Data from PI

2. Map this data to AngularJS's scope to make it available in the layout file
   - **What format is the data coming like?**
   - Let's log the data to console to get a better idea

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

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

    var definition = {
        typeName: "challenge3",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                DataShape: "Value",
                Height: 150,
                Width: 400
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) {
        this.onDataUpdate = dataUpdate

        function dataUpdate(data) {
            console.log(data);
        }
    };

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

Retrieving Data from PI

2. Map this data to AngularJS's scope to make it available in the layout file
   - **What format is the data coming like?**
   - Let's log the data to console to get a better idea



Retrieving Data from PI

- Periodically, you'll retrieve "Meta-data" updates, but these aren't in every update
  - Includes: Units, Symbol name, Path


```
▼{Label: "SINUSOID", Path: "pi:\\RSCHMITZ-PISRV1\SINUSOID", Value: "51.201", Time: "3/7/2019 12:01:35 PM", SymbolName: "Symbol0"}
   Label: "SINUSOID"
   Path: "pi:\\RSCHMITZ-PISRV1\SINUSOID"
   SymbolName: "Symbol0"
   Time: "3/7/2019 12:01:35 PM"
   Value: "51.201"
 ▶ __proto__: Object
```

Retrieving Data from PI

2. Map this data to AngularJS's scope to make it available in the layout file

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

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

    var definition = {
        typeName: "challenge3",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                DataShape: "Value",
                Height: 150,
                Width: 400
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) {
        this.onDataUpdate = dataUpdate

        function dataUpdate(data) {
            scope.Time = data.Time;
            scope.Value = data.Value;
        }
    };

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

Retrieving Data from PI

3. Reference the scope variable in the layout file

```
1  <div>
2      <h3>Value: {{ Value }}</h3>
3      <h3>Time: {{ Time }}</h3>
4  </div>
```

Leveraging 3rd party libraries

Leveraging 3rd Party Libraries
(varies from party to party - Amcharts presented here)

1. Download any necessary Javascript libraries and place in “ext” folder
   - For us this will be amcharts.js and serial.js
2. Create an container for Amcharts object
3. Create configuration for Amcharts object
4. Convert data into format Amcharts can consume

Leveraging 3rd Party Libraries

2. Create an container for Amcharts object

```
symbolVis.prototype.init = function(scope, elem) {
    var symbolContainerDiv = elem.find("#container")[0];
    symbolContainerDiv.id = "barChart_" + Math.random().toString(36).substr(2, 16);

    this.onDataUpdate = dataUpdate;

    function dataUpdate(data) {
        console.log(data);

    }

};
```

Leveraging 3rd Party Libraries

3. Create configuration for Amcharts object

```
74  symbolVis.prototype.init = function(scope, elem) {
75      var symbolContainerDiv = elem.find("#container")[0];
76      symbolContainerDiv.id = "barChart_" + Math.random().toString(36).substr(2, 16);
77
78      var chartConfig = getConfig();
79      var chart = AmCharts.makeChart(symbolContainerDiv.id, chartConfig);
80
81      this.onDataUpdate = dataUpdate;
82
83      function dataUpdate(data) {
84          console.log(data);
85      }
86
87  };
```

Leveraging 3rd Party Libraries

3. Create configuration for Amcharts object
   - But what's in that "getConfig()" function?
   - Can use amCharts Editor to customize the config and copy it over (varies chart to chart)

```
function getConfig() {
    return {
        "type": "serial",
        "categoryField": "attribute",
        "startDuration": 1,
        "categoryAxis": {
            "gridPosition": "start"
        },
        "trendLines": [],
        "graphs": [
            {
                "balloonText": "[[title]] of [[attribute]]:[[value]]",
                "fillAlphas": 1,
                "id": "AmGraph-1",
                "title": "graph 1",
                "type": "column",
                "valueField": "value"
            },
        ],
        "guides": [],
        "valueAxes": [
            {
                "id": "ValueAxis-1",
                "title": "Axis title"
            }
        ],
        "allLabels": [],
        "balloon": {},
        "legend": {
            "enabled": true,
            "useGraphSettings": true
        },
        "titles": [
            {
                "id": "Title-1",
                "size": 15,
                "text": "Chart Title"
            }
        ],
        "dataProvider": [
            {
                "attribute": "category 1",
                "value": 8,
            }
        ]
    }
}
```

Leveraging 3rd Party Libraries

4. Convert data into format Amcharts can consume
   - Varies chart type to chart type
   - Varies from data source type to data source type as well

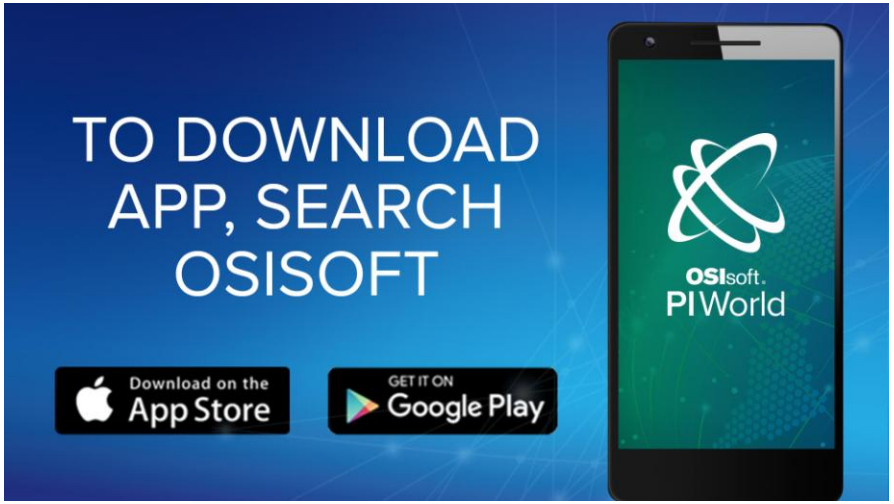
```
74  symbolVis.prototype.init = function(scope, elem) {
75      var symbolContainerDiv = elem.find("#container")[0];
76      symbolContainerDiv.id = "barChart_" + Math.random().toString(36).substr(2, 16);
77
78      var chartConfig = getConfig();
79      var chart = AmCharts.makeChart(symbolContainerDiv.id, chartConfig);
80
81      this.onDataUpdate = dataUpdate;
82
83      function dataUpdate(data) {
84          console.log(data);
85          var dataprovider = convertToChartDataFormat(data);
86
87          chart.dataProvider = dataprovider;
88          chart.validateData();
89      }
90
91      function convertToChartDataFormat(data) {
92          return data.Rows.map(function(item){
93              return {
94                  value: item.Value,
95                  attribute: item.Label
96              }
97          });
98      }
99  };
```

Questions?

Please wait for the **microphone**

State your **name & company**

Please remember



Merci

谢谢

Danke

Спасибо

Gracias

Thank You

감사합니다

ありがとう

Grazie

Obrigado

Optional: Click to add a takeaway you wish the audience to leave with.


OSIsoft.
PI World
GOTHENBURG • SEPT. 16 - 19, 2019

Built-in Angular Providers

What are AngularJS Providers

AngularJS Providers

- Resources that can be injected into multiple Angular components for use
  - Commonly a service, special value, or constant
- Allows one to expose an API for application-wide configuration

Why are Providers useful?

Why should we use providers in symbols

- Allows one to access resources outside of the symbol
  - Access to other data sources on the web
  - Access other contextual features in a display
- Recyclable functions for multiple symbols
  - Writing your own provider is outside this talk's scope

How do we use Providers?

How to Inject Providers

Steps

- Basic Building Block for a new PI Vision Symbol

Code Block

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

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

    var definition = {
        typeName: "",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) { };

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

How to Inject Providers

Steps

- The term for adding a provider to symbol/component is ‘inject’
- Added to the symbol’s definition

Code Block

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

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

    var definition = {
        typeName: "challenge1",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        inject: ['$http'],
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem) { };

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

How to Inject Providers

Steps

- Add the injected Provider/Service to any symbol functions that will leverage provider functions

Code Block

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

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

    var definition = {
        typeName: "challenge1",
        visObjectType: symbolVis,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        inject: ['$http']
        getDefaultConfig: function(){
            return {
                Height: 150,
                Width: 150
            }
        }
    }

    symbolVis.prototype.init = function(scope, elem, $http) { };

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

External Angular Providers

- **$http**
  - A core AngularJS service that facilitates communication with remote HTTP servers
- **$q**
  - A core AngularJS service that assist in handling asynchronous functions and their returns
- **$animate**
  - Exposes DOM utility methods support animation hooks
- **$interval**
  - For executing functions with a set period/interval
- Can find more built in ones in AngularJS's documentation
  - https://docs.angularjs.org/api/ng/provider

Goal for the Live Coding

- Create a symbol that can connect to the PI Web API and retrieve some information

Live Coding!

Inject the $http and $q Providers into the symbol

```
var definition = {
  typeName: “PIWebAPIData",
  visObjectType: symbolVis,
  datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
  getDefaultConfig: function(){
    return {
      Height: 150,
      Width: 150
    }
  },
  inject: ['$http','q']
}
```

Add the Providers to our initialization function to use it later

```
symbolVis.prototype.init = function(scope, elem, $http, $q) {
  this.onDataUpdate = dataUpdate;
  …
}
```

Make a Function to that can return asynchronously

```
function callPIWebAPI(tag) {
  return $q(function(resolve, reject) {
    setTimeout(function() {
      if(true){
        resolve('Hello ' + tag);
      }
      else{
        reject('Who is ' + tag);
      }, 5000);
    }
  }
}
```

Let the Data Update function resolve the function call

```
function dataUpdate(data) {
  var promise = callPIWebAPI('Robert');

  promise.then(function(result) {
    alert('Success: ' + result);
  }, function(reason) {
    alert('Failed: ' + reason);
  }
}
```

Added some alerts, just to prove the code does keep running

```
function dataUpdate(data) {
  var promise = callPIWebAPI('Robert');
  alert('Returned a promise!');

  promise.then(function(result) {
    alert('Success: ' + result);
  }, function(reason) {
    alert('Failed: ' + reason);
  }
  alert('patiently waiting');
  setTimeout(function() {
    alert('Still waiting…');
  }, 1500}
}
```

Mid-way recap

What we did

- Made a function that has an artificial delay on that the dataUpdate function calls
- Used the $q provider to unwrap the result from this artificial asynchronous function

What comes now

- Make a function that actually calls the PI Web API for information using $http provider
- Unwrap the promise returned from this actual asynchronous function

Use the Vision Client Settings to get PI Web API URL

```
function callPIWebAPI(tag) {
  if(tag) {
  // Set up useful URLs
  var _piwebapiurl  = PV.ClientSettings.PIWebAPIUrl.replace(/\/?$/, '/');
  …
}
```

Making calls with the $http Proivder

```
function callPIWebAPI(tag) {
  …
  $http.get(_piwebapiurl)
    .then(function success(response) {
      // What to do if the call completes successfully
      return response;

    } , function error(response) {
      // Log errors to console if unsuccessful
      console.log = response.statusText;

  });
}
```

Making calls with the $http Proivder

- Code

```
function callPIWebAPI(tag) {
  …
  var JSON = $http.get(_piwebapiurl)
    .then(function success(response) {
      // What to do if the call completes successfully
      return response;

    } , function error(response) {
      // Log errors to console if unsuccessful
      console.log = response.statusText;

  });
  return JSON;
}
```

Making calls with the $http Proivder

```
function callPIWebAPI(tag) {
  …
  var JSON = $http.get(_piwebapiurl, {withCredentials: true})
    .then(function success(response) {
      // What to do if the call completes successfully
      return response;

    } , function error(response) {
      // Log errors to console if unsuccessful
      console.log = response.statusText;

  });
  return JSON;
}
```

Let the Data Update function resolve the PI Web API call

```
function dataUpdate(data) {
  var promise = callPIWebAPI('Robert');

  promise.then(function(result) {
    console.log('Success: ' + result);
  }, function(reason) {
    alert('Failed: ' + reason);
  }
}
```