PI World 2020 Lab

PI Vision Extensibility, Creating a Custom Symbol

OSIsoft, LLC
1600 Alvarado Street
San Leandro, CA 94577

Published: March 18, 2020

Table of Contents

1. Introduction

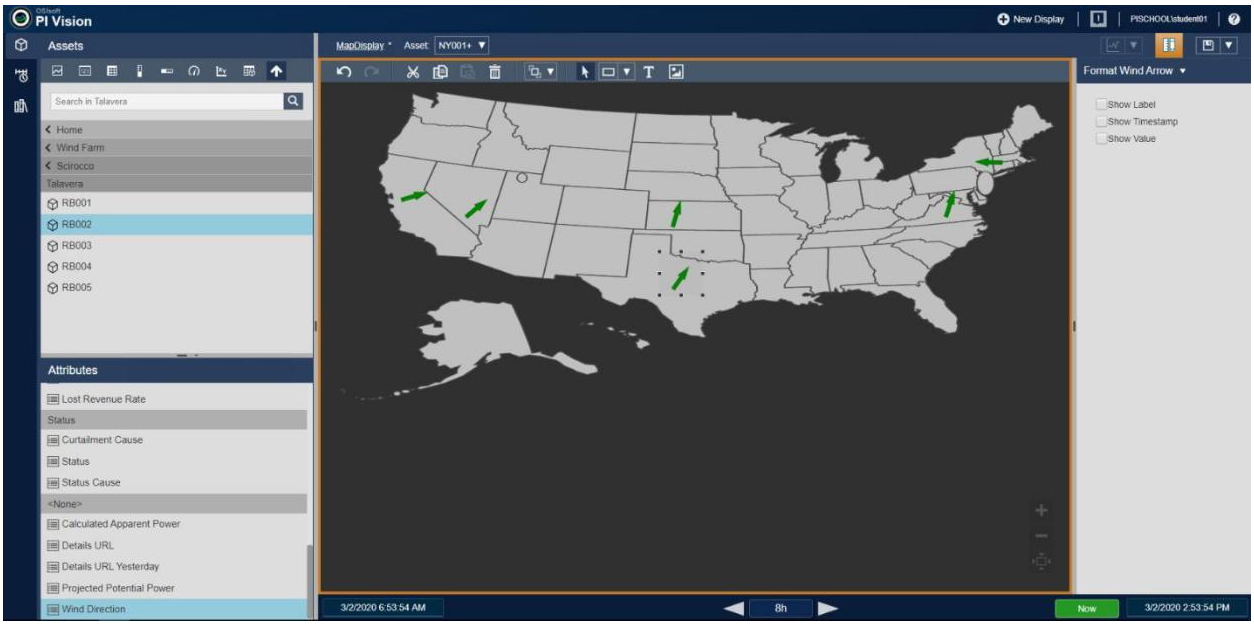
1.1. Overview of this Lab

Are the PI Vision native symbols not meeting all your users' needs? Do you have an industry specific use case for data visualization that you are not able to completely implement using the built-in PI Vision symbols? Maybe you have existing code or a web API that you would like to use within PI Vision?

You're in the right lab!

PI Vision Extensibility is a powerful model that enables you to write custom symbols and tool panes for use in PI Vision displays, including unique or industry-specific ways of visualizing PI data.

Today we will use the PI Vision extensibility framework to create a custom arrow symbol that rotates based on the value of a data item. This symbol was inspired by a customer use-case where there was a need to represent wind direction intuitively, and easily overlay that information on top of a map.



Throughout our exercises, we will learn how to register a symbol, hook into the built-in symbol events, update the symbol's view, and create a configuration pane.

While you're going through these exercises try to think about the needs of your users and how you could leverage this technology to address those needs.

1.2. Workspace

Each student's environment will consist of a PI Vision 2019 application server hosted on Microsoft IIS. The PI Vision application's SQL database will be hosted in a Microsoft SQL Express Server on the same node. Each student's PI Vision application will connect and retrieve data from the same PI Data Archive Server and PI AF Server, which are hosted on a different node.

On each student's node, is a folder, C:\Class Files\, where one can find the solutions for each exercise, a copy of the Lab PowerPoint presentation, Lab Workbook, PI Vision Extensibility Guide, and any assets to be used within the exercises, such as images, SVGs, and code snippets.

1.3. Technologies Used

Client-side PI Vision is built on ASP.NET MVC, HTML, CSS, JavaScript, AngularJS, and jQuery.

PI Vision symbols consist of JavaScript, HTML, and CSS. An understanding of AngularJS and jQuery is helpful, but not necessary.

1.4. Location of Files

The native PI Vision symbols can be found in the **%PIHOME64%\PIVision\Scripts\app\editor\symbols** folder. Native PI Vision symbols include the **Trend**, **Value**, **Table**, **Vertical Gauge**, **Horizontal Gauge**, **Radial Gauge**, **XY Plot**, and **Asset Comparison Table** symbols. All native symbols are built on our extensibility model.

Custom symbols should be placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder. The PI Vision Web Application Server will look for any JavaScript files inside of this folder and inject references, in the form of a script tag **<script src="/PIVision/Script/app/editor/symbols/ext/filname.js></script>**, to these files inside of the **index.html**, after any other native script tags. This code will get loaded and run when the user first navigates to the PI Vision Web Application in the browser. Similarly, any CSS files placed inside of this folder will also be injected into the **index.html** file in the form of a stylesheet link tag **<link href="/PIVision/Scripts/app/editor/symbols/ext/filename.css" rel="stylesheet"/>**, before any native stylesheet link tags.

To keep PI Vision from loading and executing code or CSS files on startup through the **index.html** file, you can place them inside of a subfolder within the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder. This may be useful for storing third party libraries that don't necessarily need to be loaded when the PI Vision Web Application first loads in the browser.

Additional subfolders can be created within the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder, and files within those folders will be treated as static public files that can be accessed from the browser.

1.5. Parts of a PI Vision Symbol

PI Vision symbols have three major layers, the implementation, presentation, and configuration.

Implementation

The implementation layer consists of a JavaScript file with three parts: definition, registration, and initialization. It is convention to name this file after the following pattern: **sym-<symbol name>.js**. The

code within this file is responsible for implementing a symbol definition object, a symbol initialization function, and registering the symbol with the PI Vision client/browser side global symbol catalog object.

This JavaScript file should be placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder, and not within a subfolder of this folder.

Presentation

The presentation layer consists of an HTML template file and, optionally, CSS embedded within the HTML or in a separate file. The symbol template is responsible for the symbol's appearance, and is the main view that the user will interact with the symbol through. The template has access to the symbol object's members through AngularJS directives.

It is convention for this file be named after the following pattern: **sym-<symbol name>-template.html**. If the template file is named using the standard convention and placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder, then the template URL does not need to be specified in the symbol definition, otherwise it should be specified.

Any CSS files placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder will be loaded automatically by the browser.

Configuration

The configuration layer consists of an HTML template file, and optionally, CSS embedded within the HTML or in a separate file. The configuration template is responsible for the appearance of the symbol's configuration menu, accessed through the symbol's context menu. The configuration menu is the user's main way of changing the symbol's appearance and behavior.

It is convention for this file be named after the following pattern: **sym-<symbol name>-config.html**. If the configuration template file is named using the standard convention and placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder, then the configuration template URL does not need to be specified in the symbol definition, otherwise it should be specified.

Any CSS files placed inside of the **%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext** folder will be loaded automatically by the browser.

1.6. How to use this Workbook

The following sections of this workbook are split up into exercises that should be completed in order. Each exercise builds on top of the previous one, with the goal of completing the custom arrow symbol by the last exercise.

Each exercise is made up of objectives that need to be completed in order to move on to the next exercise. You are encouraged to use the *PI Vision Extensibility Guide* to complete the objectives for each exercise on your own. However, each objective has its own subsection that presents the recommended steps to complete the objective, along with hints for when you feel stuck.

At the end of each exercise section you can find a sample solution for each of the objectives.

1.7. Before You Begin

Before you begin development, OSIsoft recommends that you place PI Vision into debug mode. To do so, edit the **web.config** file in your PI Vision installation folder to change the compilation tag, under **system.web**, from:

```
<configuration>
  <system.web>
    <compilation debug="false" targetFramework="4.6" />
  </system.web>
</configuration>
```

to

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.6" />
  </system.web>
</configuration>
```

Debug mode disables the PI Vision bundling and minification system; this makes debugging your application easier.

Note that in debug mode, PI Vision does not process minified JavaScript files.

Debug mode can have a negative impact in PI Vision's performance and should be disabled when not being used.

You should also open the PI Vision page in the browser, https://localhost/PIVision/. This may be slow the first time due to the cold start. Once the browser is open to the PI Vison page, we should open the browser developer tools (F12) and check "Disable cache" in the "Network" tab; this will ensure that we're always requesting the latest files from the server, which is useful when you're constantly making changes to those files.

1.8. Useful Resources

PI Vision Extensibility Guide

https://customers.osisoft.com/s/productcontent?id=a7R1I000000XybxUAC

PI Visualization Development Forum

https://pisquare.osisoft.com/community/developers-club/pi-visualization-development

AngularJS

https://angularjs.org/

jQuery

https://jquery.com/

2. EXERCISE 1: Create and Load Symbol

2.1. Objectives

- Create the symbol implementation file with the minimum required to be able to load the symbol in the PI Vision display editor.
- Create the symbol template file with the minimum required to be able display an arrow when an instance of the symbol is loaded in the PI Vision display editor.
- Test the newly created symbol by using it in the PI Vision display editor.

2.2. Background

The implementation file is a JavaScript file that will get executed on startup. This file is responsible for creating the base symbol function object, the symbol function object's initialization method, the symbol definition object, and for registering the symbol with the symbol catalog using the definition object. Review the *PI Vision Extensibility Guide pages 1 through 7* for examples of the implementation layer, and definition object parameters; not all parameters are required.

The symbol definition object also determines the template HTML file (presentation layer) that will be used for the symbol. The template HTML file should not include a HEAD, BODY, or FOOTER element, it should be an HTML fragment. PI Vision will inject this HTML fragment inside of a SYMBOL-HOST-DIV element.

2.3. OBJECTIVE: Create Implementation

Use the PI Vision Extensibility Guide to create the symbol's implementation file.

Steps

1. Create the JavaScript implementation file within the symbol extensibility folder.
2. Implement a JavaScript IIFE (Immediately Invoked Function Expression) that takes the *window.PIVisualization* object as an argument.
3. Instantiate the base symbol object.
4. Create the symbol definition object. You should only have to define the properties required to load and create the symbol in the editor window.
5. Define the symbol initialization function. The implementation of this function is not required for this step, it just needs to be defined.

Hints

- The *'use strict'* JavaScript directive is recommended to avoid common errors.
- Use the *provided sym-arrow.png* image for the symbol's icon in the symbol definition object.
- The *datasourceBehavior* symbol definition object property is optional, and if not specified will default to *window.PIVisualization.Extensibility.Enums.DatasourceBehaviors.None*. However, if this option is used, the symbol will be treated as a static symbol and will not be added to the symbol selector. The *window.PIVisualization.Extensibility.Enums.DatasourceBehaviors.Single* option should be used for this symbol.
- The *getDefaultConfig* symbol definition object method is optional and should return the default configuration object. However, we should define and implement this method so that the return object includes the *Height* and *Width* properties so that the symbol's presentation container div HTML element's height and width can be defined. These properties should be of Number type and correspond to pixels. If not defined the height and width will be equivalent to zero.
- The values of the properties of the object returned by the *getDefaultConfig* method should not be changed within the implementation code. These property values should only be updated by the configuration layer file using the AngularJS ngModel directive to avoid issues.

2.4. OBJECTIVE: Create Template

Use the PI Vision Extensibility Guide to create the symbol's template file.

Steps

1. Create the HTML template file within the PI Vision symbol extensibility folder.
2. Create an *SVG* element with a *PATH* element that draws an arrow in the browser.

Hints

You can use the following to create your arrow, this code snippet is also available in the class files folder:

```
<svg
  width="95%" height="95%"
  viewBox="0 0 70 70" xmlns="http://www.w3.org/2000/svg">
  <path style="fill:green" d="m27.5,
      34.94239l7.5,
      -23.9424l7.5,
      23.9424l-3.75,
      0l0,
      24.05761l-7.5,
      0l0,
      -24.05761l-3.75,
      0z" />
</svg>
```

2.5. OBJECTIVE: Load Symbol

Test the newly created symbol.

Steps

1. Navigate to the PI Vision application the browser and create a new display.
2. Make sure that the new symbol's icon is present in the symbol selector and select it.
3. Drag and drop an attribute into the editor window to create an instance of the symbol.
4. Open the browser's developer console to make sure there aren't any errors loading the symbol.

2.6. Solution

Create Implementation

1. Create the JavaScript file using the correct naming convention in the following

   location: *%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext\sym-arrow.js*

2. Define the immediately invoked function expression (IIFE) that takes the *window.PIVisualization* (PV) object as an argument. PV is created by PI Vision in the global namespace in the browser and is referenced and modified by each of the PI Vision JavaScript modules. The IIFE will be run as soon as the JavaScript file is loaded by the browser when the PI Vision page is opened.

```
(function (PV) {
  'use strict';
})(window.PIVisualization);
```

3. Define the visualization object as a function, and then use the *PV.deriveVisualizationFromBase()* method to extend the visualization object with the *SymbolBase* prototype to add some default symbol behavior to the object that is used internally by the framework.

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

4. Define the symbol definition object with the two minimum required properties: *typeName* and *visObjectType*. The *visObjectType* property value should be the symbol object we created in the previous step. We don't need to specify the *templateUrl* property because we're using the default naming convention. In addition to the minimum required properties we should add the *iconUrl* and use single for the *datasourceBehavior* since our symbol will be using a single data source. We'll also define the *getDefaultConfig* method and return an object with the *Height* and *Width* properties, otherwise the symbol will not have a height or width.

```
    var definition = {
      typeName: 'Arrow',
      datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
      iconUrl: 'scripts/app/editor/symbols/ext/icons/sym-arrow.png',
      getDefaultConfig: function () {
        return {
          Height: 80,
          Width: 80
        }
      },
      visObjectType: symbolVis
    }
```

5. Define the symbol's prototype initialization method. This method is called whenever a new symbol instance is dropped in the display editor. The implementation of this method is not required for a new symbol instance to be created, but the method does need to be defined or else an error will be thrown when PI Vision tries to call it.

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

6. The final step is to register the symbol with the *PV.symbolCatalog.register()* method. We use the symbol's definition object, which contains everything we've defined in the implementation file, to register the symbol. This should be the last line of code to be executed when this file loads and the IIFE is invoked.

```
  PV.symbolCatalog.register(definition);
```

Create Template

1. Create the HTML template file using the correct naming convention in the following location:

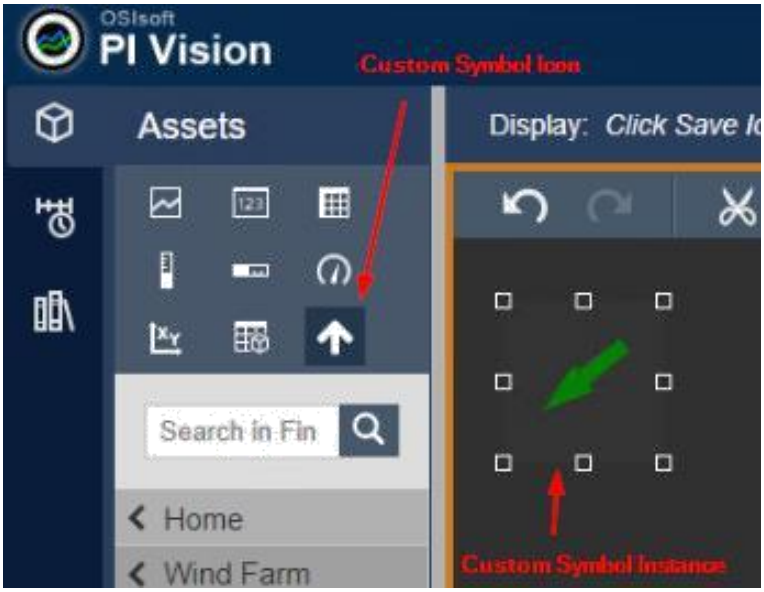
   %PIHOME64%\PIVision\Scripts\app\editor\symbols\ext\sym-arrow-template.html

2. Create the SVG and PATH element to draw the arrow on the display.

```
<svg
  width="95%" height="95%"
  viewBox="0 0 70 70" xmlns="http://www.w3.org/2000/svg">
  <path d="m27.5,
      34.9423917.5,
      -23.942417.5,
      23.9424l-3.75,
      010,
      24.057611-7.5,
      010,
      -24.057611-3.75,
      0z" />
</svg>
```

Load Symbol

The arrow symbol should be available in the symbol selector, and you should be to select it and drag and drop an attribute to the editor to create a new instance of the symbol.


OSIsoft
PI Vision
Custom Symbol Icon
Assets
Display: Click Save
Search in Fin
Home
Wind Farm
Custom Symbol Instance

3. EXERCISE 2: Update Symbol with Data

3.1. Objectives

- Specify how PI Vision should retrieve data for this symbol.
- Create a data update event listener for this symbol.
- Rotate the arrow based on the data update.

3.2. Background

The symbol initialization function is called whenever a new symbol object is created, with the context of the "this" variable being the new symbol object that was created and now being initialized. This function can be used to set up event listeners, scope properties, and DOM manipulation.

The element object passed to the symbol.prototype.init(scope, element) method is a jQuery object wrapper for the main html element (SYMBOL-HOST-DIV) that contains this instance of the symbol.

Data updates should be handled through a callback function defined through the symbol initialization method. The following are the callback functions that can be used:

- this.onDataUpdate(data)
    - data object properties determined by the data shape
    - called any time the symbol gets a data update
- this.onResize(width, height)
    - width and height are the new dimensions in pixels
    - called when the symbol is resized
- this.onConfigChange(newConfig, oldConfig)
    - newConfig and oldConfig are objects with the symbol configuration before and after the change
    - called when the symbol configuration is updated
- this.onDestroy()
    - called when the symbol instance is deleted from the editor
- this.onDataSourceChange(newSource, oldSource)
    - newSource and oldSource are arrays of the symbol data sources before and after the change
    - called when a data source is added or removed from the symbol

More information about the symbol initialization function and an example can be found on pages *6 through 7 of the PI Vision Extensibility Guide*.

Given a time range and a data source(s), PI Vision can retrieve data in a few different forms, referred to as data shapes in the documentation. *Pages 7 through 8 of the PI Vision Extensibility Guide* cover the different data shapes available for PI Vision symbols. The data shape should be specified in the *getDefaultConfig* function in the symbol definition.

Information about the data object passed to the symbol's *onDataUpdate(data)* callback can be found in *pages 9 through 10 of the PI Vision Extensibility Guide*.

3.3. OBJECTIVE: Choose How to Retrieve Data

Use the PI Vision Extensibility Guide to choose the best data shape for this symbol and add it to the symbol definition.

Steps

1. Decide which PI Vision data shape translates best to the state of the arrow's rotation or angle.
2. Modify the symbol definition object to make sure that the chosen data shape is used.

Hints

- The arrows rotation should be between 0 and 360 degrees.

3.4. OBJECTIVE: Handle Data Updates

Use the PI Vision Extensibility guide to handle the data updates to the symbol and transform the data into degrees.

Steps

1. Define the appropriate callback function within the symbol initialization function.
2. Convert the data into degrees (0 – 360) which will be used to rotate the symbol.

Hints

- Try outputting the data to the console in the browser to make sure it is what you expect.
- You should handle the case where no data is retrieved, and the argument passed is undefined.

3.5. OBJECTIVE: Rotate Arrow

Steps

1. Get a reference to the arrow HTML element.
2. Rotate the arrow based on the data update.

Hints

- The second argument passed to the symbol initialization function is a jQuery object wrapper for an HTML element.
- You can use the rotate transform function of the element to rotate it.

3.6. SOLUTION

Choose How to Retrieve Data

Since our arrow can have an angle between 0 and 360 degrees, we need to choose a data shape that will make it easy to transform the data into this value range. The data shape that makes the most sense for this application is the *Gauge* data shape. With this data shape, data object passed to the *onDataUpdate(data)* callback function will contain the following fields: *Value*, *Time*, and *Indicator*. The *Indicator* property's value is the current value as a percentage of Max – Min, between 0 and 100, which can easily be transformed to a value between 0 and 360 by the following formula:

*degrees = 360 * data.Indicator / 100*

In order let PI Vision know which data shape it should use for this symbol we need to specify it in the symbol's definition object's *getDefaultConfig* method.

Additionally, as part of the *Gauge* data shape config object properties, we won't be using the *ValueScaleLabels* and *ValueScalePositions*, so we'll set the *ValueScale* property to *false*. In our case, since we're expecting the value coming from the data source to be between 0 and 360, we can also specify this range using the ValueScaleSettings property.

```
var definition = {
  getDefaultConfig: function() {
    return {
      // <SOME CODE>
      DataShape: 'Gauge',
      ValueScale: false,
      ValueScaleSettings: {
            MinType: 2,
            MinValue: 0,
            MaxType: 2,
            MaxValue: 360
      },
    };
  }
};
```

Handle Data Updates

In order to do anything with the data retrieved with the data shape we specified we have to handle it within a callback function. This callback function should be assigned within the symbol's initialization function to the *this.onDataUpdate* field.

This function will have the data object as a parameter and will be called every time the PI Vision application gets a response to its *DiffForData* request to the PI Vision server, normally every 5 seconds.

One edge case that we should consider and handle is when the data object passed as an argument is undefined, which can occur the first time an instance of the symbol is created.

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

  this.onDataUpdate = onDataUpdate;

  function onDataUpdate(newData) {
    if (!newData) {
      return;
    }

    var degrees = 360 * newData.Indicator / 100;
  }
}
```

Rotate Arrow

Once we have the data how we need it, we then need to update the HTML to rotate the arrow. In order to easily find the arrow within the DOM we should give it a class in the HTML file.

```
<svg
  width="95%" height="95%"
  viewBox="0 0 70 70"
  xmlns="http://www.w3.org/2000/svg"> <path d="m27.5,
      34.9423917.5,
      -23.942417.5,
      23.9424l-3.75,
      010,
      24.05761l-7.5,
      010,
      -24.05761l-3.75,
      0z"
      class="svg-arrow" />
</svg>
```

We can then use the second argument passed to the symbol's initialization function when the symbol is initialized to get a reference to the SVG PATH element. This argument is a jQuery object wrapper for an HTML element reference object. Once we have the reference, we can then apply a rotation transformation to the element.

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

  this.onDataUpdate = onDataUpdate;

  function onDataUpdate(newData) {
    if (!newData) {
      return;
    }
    var degrees = 360 * newData.Indicator / 100;

    var svgArrow = elem.find('.svg-arrow')[0];
    svgArrow.setAttribute('transform', 'rotate(' + degrees + ' 35 35)');
  }
}
```

4. EXERCISE 3: Show Value, Timestamp, Label, and Units

4.1. Objectives

- Store the latest value and timestamp data updates in the symbol's scope.
- Store the latest label and units in the symbol's scope.
- Update the symbol template to display the value, timestamp, label, and units.

4.2. Background

The symbol's scope, which is the first argument passed to the symbol's initialization function can be thought of as the symbol instance's namespace. The scope object consists of several built-in properties, such as *config*, *def*, *position*, and more. However, you are also free to define your own properties on the scope object from within the initialization function and use those properties within the event handler callback functions, as well as your symbol template.

The scope object members are available inside of the symbol's template through AngularJS directives. Any scope members can be accessed through the template as if they were part of the global namespace.

It is common practice to get the value, timestamp, label, and units for a symbol from the *onDataUpdate* callback function and update the scope properties accordingly so that they can then be displayed through the symbol template.

One thing to notice is that not every update will include the label and units as part of the data object argument passed to the callback function. By default, the label and units only show up when the symbol is first created, updated, or about every 13th data update (a data update occurs about once every 5 seconds).

It is a good idea to use the *console.log()* function to output the *scope* and *data* objects to the browser console to look through all of the object properties available, and to see how the data object may change over time.

4.3. OBJECTIVE: Get Value and Timestamp

Retrieve the value and timestamp through the *onDataUpdate* function and store the results in the *scope* object.

Steps

1. Define the Value and Timestamp scope properties within the initialization function.
2. Update the Value and Timestamp scope properties with the data updates.

4.4. OBJECTIVE: Get Label and Units

Retrieve the label and units through the *onDataUpdate* function and store the results in the *scope* object.

Steps

1. Define the Label and Units scope properties within the initialization function.
2. Update the Label and Units scope properties with the data updates.

Hints

- Not every data update will contain the Label and Units properties; make sure to handle this properly.

4.5. OBJECTIVE: Display the Value, Timestamp, Label, and Units

Edit the symbol template to be able to display the Value, Timestamp, Label, and Units for the data source associated with this symbol.

Steps

1. Use HTML DIV and SPAN elements inside of the symbol template to create stubs to show the Label, Value, Units, and Timestamp.
2. Use AngularJS markup within the symbol template to bind the Label, Value, Units, and Timestamp scope properties values to the template.
3. Use the AngularJS *ng-style* directive to change the text colors to something that is easier to see within the PI Vision display editor.

Hints

- Properties within the implementation's initialization *scope* object parameter are available within the template by way of AngularJS markup and directives.
- Data binding can be achieved by using the double curly brace notation: {{ }}.
- Specify the *LabelColor* and *ValueColor* as config object properties.

4.6. SOLUTION

Get Value and Timestamp

First, we'll define the *scope.Value* and *scope.Timestamp* properties within the *symbolVis.prototype.init* function, at the start. We do this to make sure that any properties we plan on using anywhere else in the symbol are defined before they're referenced, and to help us keep track of the properties we're adding to the scope object within the initialization function.

```
symbolVis.prototype.init = function (scope, elem)
  { scope.Value = '';
  scope.Timestamp = '';

  // <SOME CODE>
}
```

Next, update the *scope.Value* and *scope.Timestamp* properties any time we get a new value and timestamp through the *onDataUpdate* callback function. The *onDataUpdate* callback is passed the *newData* object as an argument, which contains the *newData.Value* and *newData.Timestamp* properties. Assign the new values from the *newData* object to the value and timestamp properties. This is also a good place in the code to use *console.log(newData)* to see what properties are available in the *newData* object.

```
function onDataUpdate(newData) {
  // <SOME CODE>

  scope.Value = newData.Value;
  scope.Timestamp = newData.Time;

  // <SOME CODE>
}
```

Get Label and Units

Again, we'll define the *scope.Label* and *scope.Units* properties within the *symbolVis.prototype.init* function.

```
symbolVis.prototype.init = function (scope, elem)
  { scope.Value = '';
  scope.Timestamp = '';
  scope.Label = '';
  scope.Units = '';

  // <SOME CODE>
}
```

However, getting the label and units from the *newData* object passed to the *onDataUpdate* callback function will be a bit trickier. Since the *newData.Label* and *newData.Units* properties are not always defined on every data update, we should check before assigning their values to the *scope.Label* and *scope.Units* properties, otherwise, we'll cycle between our labels and units having valid values to the values being undefined.

```
function onDataUpdate(newData) {
  // <SOME CODE>

  if (newData.Label !== undefined) {
    scope.Label = newData.Label;
    if (newData.Units !== undefined) {
      scope.Units = newData.Units;
    } else {
      scope.Units = '';
    }
  }

  // <SOME CODE>
}
```

Display Value, Timestamp, Labe, and Units

First, we'll edit the HTML in the template to include stubs for Label, Value, Units, and Timestamp. We're using a non-breaking space ( ) to make sure that no matter the width of the symbol, the value and units stay in the same line, but are still separated by a space.

```
<div>
  <div>
      <span>STUB</span>
  </div>
  <div>
      <span>STUB STUB</span>
  </div>
  <div>
      <span>STUB</span>
  </div>
</div>
```

Then we can use AngularJS databinding markup, {{<scope-property-name>}}, within the HTML to bind the *scope.Label*, *scope.Value*, *scope.Units*, and *scope.Timestamp* property values to the HTML that will be presented to the user. The template has access to properties inside of the scope object of the symbol.

```
<div>
  <div>
      <span>{{Label}}</span>
  </div>
  <div>
      <span>{{Value}} {{Units}}</span>
  </div>
  <div>
      <span>{{Timestamp}}</span>
  </div>
</div>
```

The default text color for the SPAN element is black, which will be difficult to see against the PI Vision editor's default background color. We should change the text color so that they contrast more with the background. Even though we could hardcode the colors within the template file, it is better practice to specify the colors we want to use within the implementation's default config object for reusability and manageability later.

```
var definition = {
  // <SOME CODE>
  getDefaultConfig: function () {
    return {
      // <SOME CODE>
      LabelColor: 'grey',
      ValueColor: 'white',
    }
  },
  // <SOME CODE>
}
```

Now we can use these config object properties from our template along with the AngularJS *ng-style* directive to style the SPAN elements' text color with easier to read colors. We'll also specify the parent DIV element's width with the *scope.positition.width* property, which is built-in and contains the current width of the parent symbol element.

```
<div ng-style="{width: position.width}">
  <div>
      <span ng-style="{color:config.LabelColor}">{{Label}}</span>
  </div>
  <div>
      <span ng-style="{color:config.ValueColor}">{{Value}} {{Units}}</span>
  </div>
  <div>
      <span ng-
  style="{color:config.LabelColor}">{{Timestamp}}</span> </div>
</div>
```

5. EXERCISE 4: Create a Configuration Pane

5.1. Objectives

- Add user options for showing the label, value, and timestamp properties
- Conditionally show the label, value, and timestamp based on the user options
- Create the configuration pane to allow the user to toggle the options
- Expose the configuration pane through the symbol's context menu

5.2. Background

Sometimes we want to give the user options for configuring the way the symbol looks or behaves; we can do this by adding a configuration pane for the symbol. The configuration pane can be opened through the symbol's right-click context menu. The *configOptions* property in the symbol definition object can be used to create a symbol context menu shortcut to open the configuration pane. You can find additional information and examples of this symbol definition property in *pages 12 and 13* of the *PI Vision Extensibility Guide*.

We can expose the user configuration options through the configuration pane and then handle those changes accordingly. These user configuration options should have default values as part of the symbol definition. These default options will help us render the symbol correctly when it first loads. Changes to the config options will also be persisted to the backend database when the symbol is saved.

The AngularJS *ngShow* directive can make it easy to show or hide elements in our symbol template based on scope property values. And the *ngModel* AngularJS directive can make it easy to bind the value of an HTML element to a scope property value.

5.3. OBJECTIVE: Add User Options to show Label, Value and Timestamp

Add Boolean type properties, representing the user options, to the default config return object in the symbol definition for showing the symbol's label, value, and timestamp.

5.4. OBJECTIVE: Conditionally Show Label, Value, and Timestamp

In the symbol's HTML template, use AngularJS directives to show or hide the HTML elements for the label, value, and timestamp based on the user option values.

5.5. OBJECTIVE: Create the Config Pane Template

For the user to be able to change the user options, they'll need some sort of user interface. The configuration pane template provides this UI to the user and allows you to use AngularJS directives to bind the HTML element value to the user option values.

Steps

1. Create the config HTML template file within the PI Vision symbol extensibility folder.
2. Add INPUT elements to the HTML template file to toggle the Label, Timestamp, and Value.
3. Use AngularJS directives to bind the value of the INPUT elements to the value of the user options.

Hints

- Like the symbol's HTML template file, if we follow the naming convention, we do not need to specify the *configTemplateUrl* property in the symbol definition.
- You can use INPUT elements of checkbox type.
- Use the AngularJS *ngModel* directive for data binding.

5.6. OBJECTIVE: Expose the Configuration Pane through Context Menu

Use the *configOptions* symbol definition member to specify a context menu option for the symbol to open the configuration pane.

Hints

- The return object should contain a single option specifying the title, and mode, with no custom action.

5.7. SOLUTION

Add User Options to Show Label, Value, and Timestamp

We'll want to define the user options in the symbol definition's *getDefaultConfig* method by including them as properties in the return object of the function. These properties will be of Boolean type, since they'll either be set to show or hide the Label, Value, and Timestamp.

```
var definition = {
  // <SOME CODE>
  getDefaultConfig: function() {
    return {
      // <SOME CODE>
      ShowLabel: false,
      ShowValue: true,
      ShowTimestamp: false
    };
  }
  // <SOME CODE>
};
```

Conditionally Show Label, Value, and Timestamp

Next, we'll want to edit the symbol's presentation layer, i.e. the symbol template file, to either show or hide the Label, Value, and/or Timestamp, based on the values of the corresponding user options we defined in the symbol definition object.

The AngularJS *ngShow* directive lets us conditionally show an HTML element and its children based on a Boolean value. The options we defined will be available in the *scope.config* object as properties.

```
<div ng-show="config.ShowLabel">
  <span ng-
style="{color:config.LabelColor}">{{Label}}</span> </div>
<div ng-show="config.ShowValue">
  <span ng-
style="{color:config.ValueColor}">{{Value}} {{Units}}</span> </div>
<div ng-show="config.ShowTimestamp">
  <span ng-
style="{color:config.LabelColor}">{{Timestamp}}</span> </div>
```

Create the Config Pane Template File

First, we'll create the config pane template file using the naming convention, *sym-<name>-config.html*, and place the file in the following location:

%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext\sym-arrow-config.html

Then we'll add INPUT elements of checkbox type to the HTML file, one checkbox for each option that can be toggled.

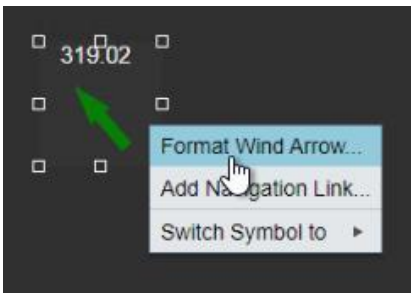
```
<div>
  <label>
      <input type="checkbox" name="ShowLabel">Show Label<br />
  </label>
  <label>
      <input type="checkbox" name="ShowTimestamp">Show Timestamp<br />
  </label>
  <label>
      <input type="checkbox" name="ShowValue">Show Value<br />
  </label>
</div>
```

Finally, we'll use the Angular *ngModel* directive to bind the value of options to the value of the checkboxes. This way the checkboxes will be checked if the option that they're modeling is true or unchecked if its false, and when we check or uncheck the checkbox it will update the corresponding property being modeled.

```
<div>
  <label>
    <input type="checkbox" name="ShowLabel"
      ng-model="config.ShowLabel"
      />Show Label<br />
  </label>
  <label>
    <input type="checkbox" name="ShowTimestamp"
      ng-model="config.ShowTimestamp"
    />Show Timestamp<br />
  </label>
  <label>
    <input type="checkbox" name="ShowValue"
      ng-model="config.ShowValue"
      />Show Value<br />
  </label>
</div>
```

Expose the Configuration Pane through a Context-Menu Option

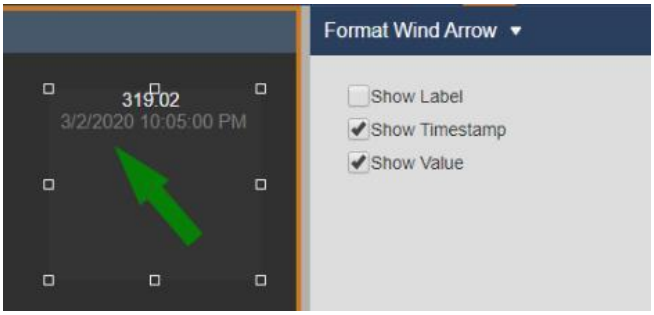
We want to be able to right click on the symbol and select the *Format Wind Arrow* option to open the configuration pane we have created.



We can use the *configOptions* property of the symbol definition object to specify the options in this context menu. The *configOptions* is a function that returns an array of objects, each object defining an option in this context menu. In our case we want a single option in the array to open the configuration pane. We should give the option object a title property, which will be the text that's shown in the context menu. We'll also define the mode, which will be the unique identifier for the configuration pane that's open, and if the user were to select another arrow symbol, then the configuration pane will stay open and display information for the other symbol. The default action of one of these options, when selected in the context menu, is to open the configuration pane, so we will not specify a custom action property.

```
var definition = {
  // <SOME CODE>
  configOptions: function() {
    return [
      {
        title: 'Format Wind Arrow',
        mode: 'formatWindArrow'
      }
    ];
  }
};
```

We should now be able to use this context menu option to open the configuration pane and modify the symbol's presentation through by toggling the checkboxes.



6. EXERCISE 5: Add Styles to Symbol and Configuration Pane

6.1. Objectives

- Add class attributes to elements
- Create the CSS file
- Configure styles through the CSS file

6.2. Background

To style our symbol we'll need to use CSS (Cascading Style Sheet). A common method to manage CSS and separation of concerns is to write the CSS in a separate file and reference that file within the HTML. Within the PI Vision extensibility framework, you're not required to reference your CSS files explicitly in your HTML template files, instead any CSS files added to the following folder will get a reference to them injected into the *index.html* file of the PI Vision Web Application:

%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext

These references to the custom stylesheets will be injected into the *index.html* file preceding any native stylesheet references. Therefore, custom CSS files with selectors that have the same specificity as the native CSS will be overridden by the native CSS. Typical CSS specificity and precedence behavior should be considered.

6.3. OBJECTIVE: Add Class Attributes to Elements

Within the symbol template and config HTML files, identify which elements to add styles to, and then add a class attribute to those elements with unique class names accordingly.

6.4. OBJECTIVE: Create the CSS File

We'll need to create the CSS file that will hold the custom styles for the symbol HTML. This file should be placed in the symbol extensibility folder.

6.5. OBJECTIVE: Configure Styles through the CSS File

Use CSS selectors to style the template and config HTML elements.

SOLUTION

Add Class Attributes to Elements

We'll examine the *sym-arrow.template.html* and *sym-arrow-config.html* files to identity which HTML elements we want to style, and then we'll add class attributes to those elements that we can then use in our CSS selectors in the CSS file.

In the sym-arrow-template.html file we'll add a class to the containing DIV for the labels, and one for the SPAN elements for each of the labels. We'll also add a class to the SVG element.

```
<div class="heading-group"
  ng-style="{width: position.width}">
  <div ng-show="config.ShowLabel">
    <span class="heading-label"
      ng-style="{color:config.LabelColor}"
      >{{Label}}</span
    >
  <!-- <SOME CODE> -->
</div>

<svg class="sample-arrow"
  width="95%" height="95%" viewBox="0 0 70 70"
  xmlns="http://www.w3.org/2000/svg">
  <!-- <SOME CODE> -->
</svg>
```

In the sym-arrow-config.html file we'll add a class for the containing DIV, and a class for the checkbox LABEL elements.

```
<div class="config-layout">
  <label class="config-checkbox">
    <!-- <SOME CODE>-->
  </label>
  <label class="config-checkbox">
    <!-- <SOME CODE>-->
  </label>
  <label class="config-checkbox">
    <!-- <SOME CODE>-->
  </label>
</div>
```

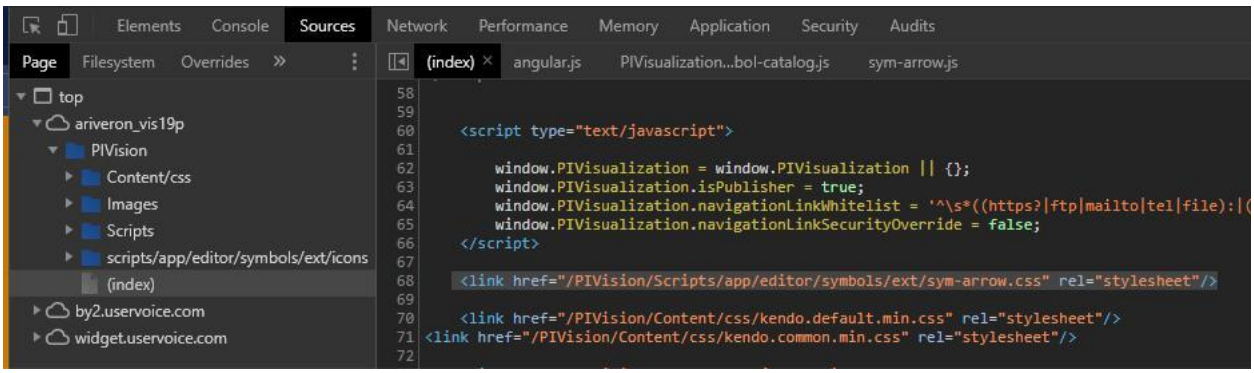
Create the CSS File

We'll create the CSS file with the following path:

%PIHOME64%\PIVision\Scripts\app\editor\symbols\ext\sym-arrow.css

The name of the CSS file does not need to follow the naming convention we've been using up to this point. All CSS files in this folder should get references to them injected into the *index.html* file. We'll be using this naming convention to easily associate the files for our custom symbol. It is possible, however, to use one master CSS file for multiple customs symbols.

Depending on the developer tools available in your browser, you can view the *index.html* file loaded by your browser, and inside of this file you should be able to find a LINK element referencing the CSS file we have created. If you're not able to see the file, even after clearing your browser cache and a hard reload, then you may have to recycle the *PIVisionServiceAppPool* application pool in IIS; you should only have to do this once, even after changing the contents of the file, as long as the file name doesn't change.



Configure Styles through the CSS File

In the CSS file, we'll use CSS selectors to modify the alignment and position of the HTML elements that make up our symbol's presentation layer and the configuration pane.

```css
.sample-arrow {
  padding-Top: 10px;
}
.heading-group {
  position: absolute;
  left: 0;
  font-size: 15px;
  text-align: center;
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
.heading-label {
  margin-right: 2px
}
.config-layout {
  padding: 15px;
  font-size: 14px
}
.config-checkbox {
  display: flex;
  align-items: center;
  margin-top: 8px;
  margin-left: 15px;
  font-size: 14px;
}
```


Have an idea how to improve our products?
OSIsoft wants to hear from you!
https://feedback.osisoft.com/

OSIsoft.