PI Vision 2017 Custom Extension Creation

Community Technical Preview

OSIsoft, LLC

1600 Alvarado St.
San Leandro, CA 94577
USA Tel: (01) 510-297-5800
Fax: (01) 510-357-8136
Web: http://www.osisoft.com

PI Vision 2017 (CTP) Custom Extension Creation

Version: 3.2.0.11

Published: 12 May 2017

Contents

Symbol Extension

Layers of a PI Vision Symbol

PI Vision symbols are broken up into three major layers:

- Implementation
- Presentation
- Configuration

The implementation layer is a JavaScript file that handles all of the implementation logic of the symbol. The presentation and configuration layers are the HTML that is responsible for the symbol appearance and symbol configuration, respectively.

File layout

All files for a symbol should be saved in the same directory, the "ext" folder, under:

INSTALLATION_FOLDER\Scripts\app\editor\symbols\.

Before You Begin

Before beginning development, it is recommended that you place PI Vision into debug mode. This can be done by editing the `web.config` file in your PI Vision installation folder and changing the compilation tag, under `system.web`, from:

```
<compilation debug="false" targetFramework="4.6"/>
```

to

```
<compilation debug="true" targetFramework="4.6"/>
```

This allows the PI Vision bundling and minification system to be disabled, which will make debugging your application easier.

Implementation Layer

Definition and Registration

The JavaScript implementation file can be broken down into three parts: definition, registration, and initialization.

Based on best practices, all PI Vision symbols should be wrapped in an immediately-invoked function expression (IIFE). An IIFE is simply a JavaScript function that is executed as soon as it is defined. The IIFE will take in the global PI Visualization object, passed in as a parameter.

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

The first step is to create our visualization object which will be built on later. In this step, you are creating a function as a container for your symbol. The function will be extended via PI Vision helper functions to add some default behaviors.

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

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

})(window.PIVisualization);
```

The next step is to add the symbol registration. In this step, you are registering your symbol with the PI Vision symbol catalog.

The next step is to augment the registration with an actual symbol definition. The definition object is a JSON object (key value pairs) that sets defaults for the symbol. Possible settings in the object include:

Parameter	Value	Notes
typeName	String. Internal unique name of the symbol.	Required.
displayName	String. Name that will be shown in the symbol picker menu.	Optional. `typeName` will be used if left blank.
datasourceBehavior	Number. Mapping to the number of `datasources` the symbol accepts.	Optional. Can be None, Single, or Multiple. If not specified None will be used.
iconUrl	String. Path to the icon to be used on the symbol selector.	Optional. If not specified, a default image is displayed on the symbol selector menu. This can be the path to any image file type that can be added to an HTML <img> tag.
getDefaultConfig	Function. Function returning the default configuration to save in the database	Optional. A function used to specify the collection of parameters that should be serialized to the backend database. By convention, all properties should begin with an uppercase.
loadConfig	Function. Returns true if the saved configuration should be merged into the default configuration.	Optional. This function is used to upgrade a previous version of a symbol's configuration.
templateUrl	String. Path to the presentation HTML file.	Optional. If omitted, it will look in the current directory for a file named "sym-<typeName>-template.html"

configTemplateUrl	String. Path to the configuration HTML file.	Optional. If omitted it will look in the current directory for a file named "sym-<typeName>-config.html"
configTitle	String. Title for configuration.	Optional. Used in the context menu when right-clicking the symbol and in the title of the configuration pane.
configOptions	Function. Function controlling what configuration options are available for this symbol. It takes in the symbol and returns an array of objects controlling configuration.	Optional. The objects returned can contain: `action`: Callback function to execute immediately. `title`: this is the context menu text mode: name of the configuration, so it can be shared with similar symbol configurations. `enabled`: Boolean when the menu item should be enabled.
configure	Object. Collection of key/ value pairs to be used on the configuration pane.	Optional. This is mainly useful for holding static based configuration options, such as localization, and for callbacks that can be executed from the configuration pane markup referenced in configTemplateUrl.
configInit	Function. Called when the configuration pane of a symbol is activated.	Optional.
StateVariables	Array of Strings. Properties that will return multistate information if configured.	Optional. Setting this variable allows a symbol to be multistated. The variable listed will be added to the symbol's scope and available for data binding in HTML.
resizerMode	String. The type of resizes the symbol should support.	Optional. String used for determining how a symbols should resize. Options include: '' - Empty string. This allows a symbol to be resized in any direction. (Default)

inject	Array of Strings. A list of services that should be dependency injected into the `init` function.	Optional. Default is empty array.
visObjectType	Function. Object holding symbol specific functionality.	Required. Function that was extended from deriveVisualizationFromBase.
formatMap	Object. Collection of key / value pairs used to map pre-PI Vision 2017 configuration options to the current names.	Optional. See Symbol Formats.
noExpandSelector	String. CSS class name for determines if a popup trend does not show.	Optional.
supportsCollections	Boolean. Indicates whether the symbol can be included as part of collection symbols.	Optional. Default is false. In order for the symbol to be driven by the collection data sources, the symbol must use one of the built-in data shapes. See Data Shapes.
supportsDynamicSearchCriteria	Boolean. True if the symbol supports a dynamic search criteria for assets.	Optional. Default is false.

The `getDefaultConfig` function is used to specify the collection of parameters that should be serialized to the backend database. These are the parameters that your symbol will need to use in order to render properly. The resulting object returned by `getDefaultConfig` is placed on the symbols scope property as `config`, i.e., `scope.config`. Please note, by convention, all of these parameters should start with an upper-case letter.

The main parameter from `getDefaultConfig` that is used by the PI Vision system is `DataShape`. This parameter is used to tell the application server the information that this symbol needs to represent

the data. See Data Shapes.

The `datasourceBehavior` property is determined by the following object, found in *`\Scripts\app\common\PIVisualization.enumerations.js`*:

```
// Determines if a symbol can have 0, 1, or n number of datasources added to it.
// This does not affect adding datasources for multistating a symbol.
// This is redundant to the "symbol model" derived objects which also define this
behavior;
// however, those classes are slated to be removed so that all symbols share the
same model
// (then this setting becomes more important).
Enums.DatasourceBehaviors = Object.freeze({
    None: 0,
    Single: 1,
    Multiple: 2
});
```

The `datasourceBehavior` property is used to determine the types of data sources that can be used from the PI Vision search pane. Symbols with a behavior of `None` are considered static symbols and will not be added to the symbol selector. Symbols with a data source of `Single` allow a single tag or attribute to be drag and dropped on the display to create that symbol. Symbols with data source of `Multiple` allow multiple tags, attributes or element to be drag and dropped on the display to create that symbol.

Below is a sample definition object from the native PI Vision value symbol:

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

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

    var def = {
        typeName: 'value',
        displayName: PV.ResourceStrings.ValueSymbol,
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        iconUrl: 'Images/chrome.value.svg',
        getDefaultConfig: function () {
            var config = PV.SymValueLabelOptions.getDefaultConfig({
                DataShape: 'Value',
                Height: 60,
                Fill: 'rgba(255,255,255,0)',
                Stroke: 'rgba(119,136,153,1)',
                ValueStroke: 'rgba(255,255,255,1)',
                ShowTime: true,
                IndicatorFillUp: 'white',
                IndicatorFillDown: 'white',
                IndicatorFillNeutral: 'gray',
                ShowDifferential: true,
                DifferentialType: 'percent',
                ShowIndicator: false,
                ShowValue: true,
                ShowTarget: true
            });
            return config;
        },
        loadConfig: loadConfig,
        templateUrl: 'scripts/app/editor/symbols/sym-value-template.html',
        resizerMode: 'AutoWidth',
        StateVariables: ['Fill', 'Blink'],
        inject: ['symValueLabelOptions'],
        visObjectType: symbolVis,
        configTemplateUrl: 'scripts/app/editor/symbols/sym-value-config.html',
        configTitle: PV.ResourceStrings.FormatValueOption,
        formatMap: {
            BackgroundColor: 'Fill',
            TextColor: 'Stroke',
            ValueColor: 'ValueStroke'
        },
        fontMetrics: {
            charHeight: 10,
            charMidHeight: 4,
            charWidth: 6.3
        }
    };
    PV.symbolCatalog.register(def);

    })(window.PIVisualization);
```

Initialization

The final part of the symbol implementation is the `init` function. The `init` function is defined on the prototype of the symbol container object created in `deriveVisualizationFromBase`.

symbolVis.prototype.init = function (scope, element) {

The `init` function takes in two parameters, scope and element, and optionally sets callback functions on the symbol container object to drive the symbol, such as data updates and resize events.

Set inside the `init` function:

`this.OnDataUpdate`: This function is called by the PI Vision infrastructure any time a data update occurs. It takes in a data object that contains the Value, Time, Path, Label, Units, Description, etc. The properties on the object returned are determined by the `DataShape` specified in the `getDefaultConfig`ction.

`this.OnResize`: This 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.

`this.OnConfigChange`: This function is called by the PI Vision infrastructure anytime the configuration of a symbol is updated. It takes in the new configuration and the old configuration.

`this.OnDestroy`: This function is called by the PI Vision infrastructure when the symbol is destroyed.

Here is a sample `init` function definition:

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

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

    symbolVis.prototype.init = function (scope, element) {
        this.onDataUpdate = dataUpdate;
        this.onConfigChange = configChanged;
        this.onResize = resize;

        function dataUpdate(data) {
            // ...
        }

        function configChanged(newConfig, oldConfig) {
            // ...
        }

        function resize(width, height) {
            // ...
        }
    };

    var def = {
        // ...
    };
    PV.symbolCatalog.register(def);
})(window.PIVisualization);
```

Data Shapes

The `getDefaultConfig` function in the symbol definition can include a `DataShape` field, which defines how data should be retrieved by PI Vision.

Value

A single data source shape that is used by PI Vision's value symbol. It is a single value at a specific time.

Gauge

A single data source shape that is used by PI Vision's gauge and bar symbols. This includes the ratio of a value between a minimum and a maximum. These options are available if set as fields on the symbol's `config` object:

- `Start`: Numeric value for zero on the scale, defaults to 0 if setting not present. No default.
- `ValueScale`: Return ValueScaleLabels and `ValueScalePositions` in the data update. Default = true
- `ValueScaleSetting`: An `object` with these fields:
  - `MinType`: 0 = `Autorange`, 1 = Use data item definition (default), 2 = Absolute
  - `MinValue`: If `MinType` is 2, the numeric value of the bottom of the scale
  - `MaxType`: 0 = `Autorange`, 1 = use Data item definition (default), 2 = Absolute
  - `MaxValue`: If `MaxType` is 2, the numeric value of the top of the scale

Trend

A multiple data source shape that is used by PI Vision's trend symbol. These options are available on the configuration object:

- `Markers`: If true, request recorded values instead of plot values if time range is short enough.
- `MultipleScales`: If true, each trace is scaled independently; otherwise, all traces share one scale.
- `TimeScaleType`: Controls labels on the time scale. 0 = Start, End and Duration; 1 = Timestamps; 2 = Relative to end; 3 = Relative to start.
- `ValueScaleSetting`: See the Gauge symbol configuration object above. Defaults are 0, Autorange.

The `FormatType` can be set independently for each trace by including them in a `TraceSettings` array.

Table

A multiple data source shape that is used by PI Vision's table symbol. When using the Table shape, you can specify these options on the configuration object:

- `Columns`: Array of strings. Can include 'Value', 'Trend', 'Average', 'Minimum', 'Maximum', 'StdDev', 'Range', or 'pStdDev'.
- `SortColumn`: Column on which to sort results.
- `SortDescending`: True to reverse sort order.

TimeSeries

A multiple data source shape that returns raw data values. These options are available on the configuration object and apply to each returned data source being returned:

`DataQueryMode`: This specifies the type of query to perform. All valid values can be found under the object `DataQueryMode` (reference scripts/common/PIVisualization.enumerations.js). The default is `ModeEvents`. Here are a few of the common ones:

`ModeEvents`: Returns archived values.

`ModeSingleton`: Returns the snapshot value.

`ModePlotValues`: Returns data suitable for plotting over a specified number of intervals. Intervals typically represent the pixels of the screen width.

`ModeMarkers`: Returns archived values, up to the limit set on the server (typically 400 values).

Automatically falls back to `PlotValues` if the threshold is exceeded.

`Intervals`: Used in connection with a request for `PlotValues`, typically represents the pixels of the screen width.

Configuring Number and Date formats

The configuration settings returned by the `getDefaultConfig` function can include the 'FormatType' field to control the format of numbers and dates displayed in a symbol.

If this setting is not present or not null, numbers and dates are formatted using the thousands separator, decimal separator, and date format for the primary language of the browser or the client operating system. Dates are adjusted to the time zone of the browser, unless overridden in a URL parameter or by a global server setting.

If set to "Database", the `DisplayDigits` setting in the PI Data Archive point definition is used to control precision. If set to "Scientific", numbers are shown in exponential notation.

Any other standard or custom string supported by Microsoft C# can also be used to control precision and leading or trailing zeroes, with special formats for currency, percentages and negative numbers. (See Microsoft MSDN article C# Numeric Format Strings.)

If this setting is set to null, numbers are returned in invariant format without the thousands separator, using the period as the decimal separator. Dates are returned in the ISO 8601 format 'YYYY-MM-DDThh:mm:ss.fffZ'.

Data Updates

Based on the symbol's configuration and its `datasources`, PI Vision requests data and calls the `dataUpdate` method that is defined when the symbol is initialized. The object passed to this function depends on the symbol's `DataShape`.

Metadata

Some properties of a data item change infrequently, such as the data item name or its unit of measure. To reduce the response size and improve performance, these metadata fields are returned on the first request and only periodically afterward. The symbol update code should only process updates for the following fields if they actually exist in the response:

- Path
- Label
- Units
- DataType (Included if configuration object has DataType set to true)
- Description (Included if configuration object has Description set to true)

Error fields

If data cannot be retrieved for a data item, the `IsGood` field is added to the response set to false, and the `ErrorCode` and `ErrorDescription` fields include specifics about the error.

DataShape	dataUpdate Parameter Properties, plus Metadata and Error fields
`Value`	Value, Time

Gauge	Value, Time `Indicator`: Current value as a percentage of Max - Min, between 0 and 100 `StartIndicator`: The value of Start as a percentage of Max - Min, 0 to 100 `ValueScaleLabels`: Array of scale labels `ValueScalePositions`: Array, position of labels between Min and Max, 0 to 100
Trend	`StartTime`, `EndTime`, `Duration` `TimeScaleLabels`: Array of scale labels `TimeScalePositions`: Array, position of gridlines between Min and Max, 0 to 100 `ValueScaleLabels`: Array of scale labels `ValueScalePositions`: Array, position of labels between Min and Max, 0 to 100 `Traces`: Array of objects with these fields: Metadata and error fields Value `LineSegments`: Array of trace points in 100x100 coordinate space, origin lower left `ErrorMarkers`: Coordinates of data errors or where traces go out of bounds `Markers`: True if points are for recorded values `ScaleMin`, `ScaleMax`: Scale labels if multiple scales are requested `Stepped`: If true, data item is stepped
Table	`Rows`: Array of objects with these fields: Metadata, error fields `Trend`: Array of trace points in 100x100 coordinate space, origin lower left `Summary`: Array of requested statistical columns
TimeSeries	`Data`: Array containing data objects for each individual data source associated with the symbol. Each item can contain the fields: Metadata, error fields Values array • Time • Value

Presentation Layer

The presentation layer for a symbol is basic HTML, with AngularJS for data and configuration binding. The presentation layer is defined by the symbol's `templateUrl` property in the definition.

For the linear gauge, this is defined in a file called *`\Scripts\app\editor\symbols\ext\sym-lineargauge-template.html`*

Here is the HTML code for the linear gauge symbol:

```
<div id="outer"
     style="'position': 'relative'; width:100%; height:100%; border:1px solid
white;">
    <div id="inner"
         ng-style="{'background':config.Fill, 'width':innerWidth,
'height':innerHeight, bottom: '-1px', left: '1px', 'position': 'absolute'}">
    </div>
</div>
```

The gauge symbol is made up two div elements: the outer div for the border and the inner div to show the value. The majority of the work is handled by AngularJS in the inner div. This div has an ng-style attribute, which is AngularJS's way of setting styles.

In the ng-style, we are setting the background color to be whatever is configured for the symbol's fill, which was originally defined in the `getDefaultConfig` function. The height and width are also set based on variables defined on the symbol's scope in the `init` or the `dataUpdate` function.

Custom Styles

Custom CSS files can be added to provide styling for symbols. These files should be placed into the same directory as the symbol, *\Scripts\app\editor\symbols\ext*. Note that custom CSS files placed in this directory are subject to overrides by the application styles. That is, if a custom style selector has the same target and specificity as another style in the application, the custom style may not be applied. CSS styles added to this directory should not be used for application theming.

When writing styles for custom symbols, it is a best practice to choose unique selectors; however, avoid using "id" attributes as they are not meant to be duplicated.

The most convenient way to signify a specific style target is through the use of unique class selectors:

```
<div class="my-custom-symbol">
    <span>Symbol Content</span>
</div>
```

Styles can then target this symbol without interfering with other parts of the application:

```
.my-custom-symbol {
  color: blue;
}
```

Configuration Layer

The configuration layer, much like the presentation layer, is basic HTML, with AngularJS for data binding. The configuration layer is defined by the symbol's `configTemplateUrl` property in the definition.

The configuration options are shown on the symbol's context menu, via right-click or long press on touch.

For the linear gauge, this is defined in a file called `\Scripts\app\editor\symbols\ext\sym-lineargauge-config.html`. Here is the HTML code for the linear gauge symbol:

```
<div class="c-side-pane t-toolbar">
    <span style="color:#fff; margin-
left:15px">{{::def.configure.orientationKeyword}}</span>
</div>

<div class="c-config-content">
    {{::def.configure.orientationKeyword}}
    <select ng-model="config.Orientation">
        <option value="Horizontal">{{::def.configure.horizontalKeyword}}</option>
        <option value="Vertical">{{::def.configure.verticalKeyword}}</option>
    </select>
</div>

<div class="c-side-pane t-toolbar">
    <span style="color:#fff; margin-
left:15px">{{::def.configure.fillKeyword}}</span>
</div>
<format-color-picker id="fill" property="Fill" config="config"></format-color-
picker>
```

The title of the configuration pane will be the value set in the `configOptions` for title. This is also what is shown on the context menu when launching the configuration pane.

The first div element sets the section title block in the configuration pane. Here we are using AngularJS's binding syntax to set the text based on a string set in the symbol's configure object.

The second div contains the selection menu for choosing the orientation of the gauge symbol. This is just a basic HTML select with options. The option text is based on a strings set in the symbol's configure object and bound using AngularJS syntax.

The important part of this section is the ng-model attribute. This is used to bind the value set in the configuration pane back to the symbol itself.

The next div is another section header for the fill color of the gauge symbol.

The last element, `format-color-picker`, is a predefined configuration control. This adds a color picker to the configuration pane. The property attribute tells the control what property should be bound to. In this example, it is the Fill. The `config` attribute tells the control where to find that property.

Configuration Options

A symbol can define the entries in a context menu that is shown when the symbol is right-clicked or after a long press with touch. The options are defined in the symbol definitions `configOptions` function. This function is called when the menu is opened, so the list of options can be dynamically populated based on the state of the symbol or the element that was clicked.

Here is an example of how to program the context menu:

```
configOptions: function (context, clickedElement) {
    var options = [{
        title: 'Configure My Symbol',
        mode: 'configureMySymbol'
    }, {
        'separator'
    }, {
        title: 'Hide',
        action: function (context) {
            context.def.configure.hide(context.symbol);
        }
    }];

    return options;
}
```

The first parameter passed into this function is the context object which has fields that describe the current symbol:

- symbol: symbol,
- config: symbol.Configuration,
- runtimeData: runtimeData,
- def: runtimeData.def

The second parameter is the DOM element that was clicked or touched to open the context menu.

The first option opens the configuration pane using the symbol's configuration template. The mode property causes the configuration pane to stay open if another symbol on the display is selected that supports the same mode.

The second option draws a separator line in the context menu.

The third option defines an immediate action that invokes a function defined on the symbol's 'configure' object in the definition.

Symbol Formats

This section describes standardized format names, conventions and their usage for PI Vision symbols. The purpose of a standardized symbol format is to:

- Enable a symbol to share formats with other symbols. For example, when a symbol type is switched to another symbol type or a format paint brush feature, etc.
- Support forward compatibility, i.e., ability to open/edit displays from prior PI Vision versions with formats that they were saved with.

PI Vision 2017 supports switching a symbol from one type to another supported type (e.g., Value to `LinearGauge`). Standard format options and options common to a symbol family will be preserved when changing types. The addition of a new object called FormatOptions to the symbol configuration object allows developers a place to define anything format related that they want to participate in any format copying that PI Vision has now or will provide in the future.

Common Format Names

Here is a list of standard format names used to share formats between symbols.

Format Name	Description

TitleColor	Color of title text
TitleSize	Size of title text
TitleFont	Font of title text
TitleBackgroundColor	Background color of title text
TitleAlignment	Alignment for title text
TextColor	Color of text
TextSize	Size of text
TextFont	Font of text
TextBackgroundColor	Background color of text
TextAlignment	Alignment of text
BackgroundColor	Background color of symbol
LineColor	Color of line/border
LineWidth	Width of line/border
LineDashType	Style of line/border
ValueColor	Color of value shown in the symbol

FormatOptions object

To make it easy to share formats between symbols, a new object called `FormatOptions`, which is a collection of formats, is created as one of the collection properties returned by `getDefaultConfig`. The `FormatOptions` object can contain as properties either standard format names or custom names. Any format that is part of this object is automatically shared when a symbol is switched to another allowed type or for future format sharing features.

Note:
Starting with PI Coresight 2016 R2, all newly created symbols should use the `FormatOptions` object exclusively for storing format properties; maps are only used for existing symbols with already defined format properties.

Sample code:

```
var def = {
    getDefaultConfig: function () {
        return {
            DataShape: 'Gauge',
            Height: 200,
            Width: 200,
            FormatOptions: {
                TextColor: 'rbg(0,123,127)',
                LineWidth: 12
            }
        };
    }
};
```

Forward Compatibility

Symbol formats of displays created/saved in versions prior to PI Vision 2016 R2 use different format names for common formats that need to be shared between symbols. For the sake of forward compatibility, these names are kept intact and a `formatMap` object is used to map them to common format names or symbol specific names that are shared between particular symbols. This `formatMap` object is defined as a property of the symbol definition object.

Sample Code:

```
var def = {
    getDefaultConfig: function () {
        return PV.SymValueLabelOptions.getDefaultConfig({
            DataShape: 'Gauge',
            Height: 200,
            Width: 200,

            FaceAngle: 270,
            IndicatorType: 'arc',
            IndicatorWeight: 2,
            BorderWidth: 3,

            IndicatorColor: 'rgb(0, 162, 232)',
            FaceColor: 'rgba(0, 0, 0, 0)',
            BorderColor: '#fff',
            ScaleColor: '#fff',
            ValueColor: '#fff',

            ScaleLabels: 'all',
            LabelLocation: 'bottom',
            FormatOptions: {
                TitleColor: 'rbg(0,123,127)',
                TitleSize: 12
            }
        });
    },
    formatMap: {
        GaugeBackgroundColor: 'FaceColor',
        LineColor: 'BorderColor',
        LineWidth: 'BorderWidth',
        TextColor: 'ScaleColor'
    }
};
```

Symbol type switching

Once created, PI Vision symbols can be switched into other supported types. For example, a Trend symbol can be switched into a Table symbol type and vice versa, and a Value symbol can be switched into any gauge symbol type and vice versa. When a symbol switch happens, all matching formats from the defined `formatMap` and all formats in the `FormatOptions` object are copied from the source type to the destination type. For example, when a Value symbol is switched to a gauge symbol type, the `ValueColor` format is copied to the gauge symbol. As of this writing, Value symbol type had no `FormatOptions` object defined.

SymbolFamily property

A symbol definition can define a property called `symbolFamily`. If the source and destination types of switched symbols belong to the same `symbolFamily` (e.g., `VerticalGauge` and `HorizontalGauge` both belong to the same `symbolFamily` called "gauge"), then from the source type all formats defined in the `formatMap` along with all formats in the `FormatOptions` are copied to the destination type.

Upgrading Existing Symbols

PI Coresight 2016 to PI Vision 2017

The major change in 2016 R2, and preserved in 2017, was the addition of the helper functions for

deriving symbols from a base symbol definition and the use of prototypical inheritance to set the `init` function. In addition, the `window.Coresight` namespace has been renamed to `window.PIVisualization`. To upgrade a symbol from PI Coresight 2016 to PI Vision 2017, perform the following:

1. Use the following convention for the outer IIFE function. See PI Coresight 2016 R2 to PI Vision 2017 for more details.

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

2. Create a function object to hold the symbol object.

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

3. Add an `init` onto the prototype of the function created above which can point to your original `init` function.

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


```

4. Rather than returning anything from your `init` function, you now set the update, resize, etc, event on the `this` pointer in your `init` function. These functions can point to your existing handler functions:

```
this.onDataUpdate = dataUpdate;
this.onConfigChange = configChanged;
this.onResize = resize;
```

5. Remove the `init` section from the symbol definition object.
6. Update the `datasourceBehavior` in the `init` section to point to the new location of the enumeration, `PV.Extensibility.Enums.DatasourceBehaviors.`
7. Update init section to add `visObjectType` and point it to the function object created in step 1.

A number of HTML helper directives used in configuration panes were also updated. To upgrade these directives in your configuration panes, simple change the 'cs' prefix to a 'pv'. For example, `cs-color-picker` becomes `pv-color-picker`.

PI Coresight 2016 R2 to PI Vision 2017

The major change was the renaming of files and variables to a more generic convention. In previous versions, global methods and properties were added to the `window.Coresight` namespace. In PI Vision 2017, this has been renamed to `window.PIVisualization`.

In PI Coresight 2016 and 2016 R2, the following convention was used.

```
(function (CS) {
    'use strict';
})(window.Coresight);
```

To upgrade a symbol to PI Vision 2017, change the argument to `window.PIVisualization`.

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

For simplicity, you may keep the parameter name CS as an alias for the `window.PIVisualization` argument so that existing code will continue to work with this name.

A number of HTML helper directives used in configuration panes were also updated. To upgrade these directives in your configuration panes, simple change the 'cs' prefix to a 'pv'. For example, `cs-color-picker` becomes `pv-color-picker`.

Tool Pane Extension

Layers of a PI Vision Tool Pane

PI Vision tool panes are broken up into two major layers:

- Implementation
- Presentation

The implementation layer is a JavaScript file that handles all of the implementation logic of the symbol. The presentation layer is the HTML responsible for the pane's appearance. Configuration persistence is not yet implemented.

File layout

All files for a tool pane should be saved in the same directory, the "ext" folder, under

INSTALLATION_FOLDER\Scripts\app\editor\tools\.

If the "ext" folder is not present, it should be created.

Implementation Layer

The JavaScript implementation file can be broken down into three parts: definition, initialization, and registration.

Tool pane creation proceeds much like symbol creation, but is part of a different catalog.

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

    var def = {};
    PV.toolCatalog.register(def);

})(window.PIVisualization);
```

The following options are available in the tool definition:

Parameter	Value	Notes
typeName	String. Internal unique name of the tool.	Required
displayName	String. Name that will be shown in the tool tab's tooltip.	Required
iconUrl	String. Path to the icon to be used on the tool tab.	Required

templateUrl	String. Path to the presentation HTML file	Optional. If omitted it will look in the current directory for tool- <typeName>-template.html
inject	Array of Strings. A list of services that should be dependency injected into the init function.	Optional. Default is empty array.
init	Function. Function that will be called when the symbol is being added to a display.	Required. Takes in the scope of the current symbol and the element on which it is in the

Tools are singular instances appearing in the left pane of the PI Vision application, as such they are useful for functionality that you want to have loaded at all times as a user switches displays.

They will share the same space as the built in Search and Events tool panes.

Badging

All tool extensions automatically have a property called `Badge` set on their scope. This can be used to display text in a badge on the tool tab's icon. This is typically used to show a count of new items available for viewing on an inactive tab; clicking the tab will erase the badge until the next time it is set. To set the badge, call the raise method on badge with the text you want to display. (Badge is only capable of showing 1-3 characters due to space constraints).

Example:

```
scope.Badge.raise("10");
```