

PI World 2019 Lab

Building a Manual Data Entry Symbol in PI Vision



OSIsoft, LLC
1600 Alvarado Street
San Leandro, CA 94577 USA
Tel: (01) 510-297-5800
Web: <http://www.osisoft.com>

© 2019 by OSIsoft, LLC. All rights reserved.

OSIsoft, the OSIsoft logo and logotype, Analytics, PI ProcessBook, PI DataLink, ProcessPoint, Asset Framework (AF), IT Monitor, MCN Health Monitor, PI System, PI ActiveView, PI ACE, PI AlarmView, PI BatchView, PI Vision, PI Data Services, Event Frames, PI Manual Logger, PI ProfileView, PI WebParts, ProTRAQ, RLINK, RtAnalytics, RtBaseline, RtPortal, RtPM, RtReports and RtWebParts are all trademarks of OSIsoft, LLC. All other trademarks or trade names used herein are the property of their respective owners.

U.S. GOVERNMENT RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the OSIsoft, LLC license agreement and as provided in DFARS 227.7202, DFARS 252.227-7013, FAR 12.212, FAR 52.227, as applicable. OSIsoft, LLC.

Published: March 22, 2019

Table of Contents

Contents

Table of Contents	3
1. Introduction	4
1.1 Workspace	4
1.2 Technologies used.....	5
1.3 Location of Files	5
1.4 Parts of a PI Vision symbol.....	5
2. Loading example symbol.....	9
2.1 Directions	9
2.2 Solution	10
3. Making a test HTTP request from custom symbol to PI Web API.....	11
3.1 Directions	11
3.2 Solution	12
4. Obtaining stream WebId with GetByPath.....	13
4.1 Directions	13
4.2 Solution	14
5. Using WebId in Update Value request.....	15
5.1 Directions	15
5.2 Solution	16
6. Adding user input.....	18
6.1 Directions	18
6.2 Solution	19
7. References	20
Save the Date!.....	22

1. Introduction

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.

PI Web API is a RESTful interface to the PI System that allows client applications read and write access to PI and AF over HTTPS.

Together these two technologies make anything possible. Today we will use PI Vision extensibility framework and PI Web API to create a symbol that sends data from PI Vision display to PI.

Remember: mind security! Anyone who has write data access to AF Attribute and/or PI Point can enter data with manual entry symbols. Before deploying such symbols, make sure only allowed users and groups have write access to AF Attributes and PI Points.

1.1 Workspace

Everything you need for the lab has already been installed on your Azure VM, PISRV01:

- PI System
 - PI Data Archive
 - PI AF
 - PI Web API
- Example of a custom symbol
- Visual Studio Code

Please ask your instructor if you haven't received a copy of instructions on how to access Azure VM for this labs.

Once you are connected to your VM, run **services.msc** and make sure that the following services are running:

- SQL Server (MSSQLSERVER)
- PI AF Application Service
- PI Web API

On your VM desktop, you will find a shortcut to **Lab files** folder with this document in .pdf format as well as helper files and solutions to exercises

1.2 Technologies used

Client side PI Vision is built using a combination of ASP.NET MVC, HTML, JavaScript, specifically Angular 1.X, and CSS.

PI Vision symbols are only made up of JavaScript, HTML, and CSS. While an understanding of Angular is helpful, it is not specifically necessary.

1.3 Location of Files

For native PI Vision symbols, these files can be found

INSTALLATION_FOLDER\PIVision\Scripts\app\editor\symbols. Native PI Vision symbols include value, linear gauges, trend, and table. This location is a great place to read our code and see how we accomplished some things for native symbols. All native symbols are built on our extensibility model. The extensibility model is based on a subfolder located inside the symbols folder, called ext.

1.4 Parts of a PI Vision symbol

PI Vision symbols are broken into three distinct parts/files, the implementation, the presentation, and the optional configuration.

Implementation

The implementation of a PI Vision symbol is done through JavaScript. This file is required for all PI Vision symbols, native or custom. The implementation file for the native PI Vision value symbol can be found **INSTALLATION_FOLDER\PIVision\Scripts\app\editor\symbols\PIVisualization.sym-value.js**.

Presentation

The presentation of a PI Vision symbol is done through HTML and CSS. Like the implementation, this file is required for all symbols. The presentation file for the native PI Vision value symbol can be found **INSTALLATION_FOLDER\PIVision\Scripts\app\editor\symbols\sym-value-template.html**.

Configuration (not covered in this lab)

The configuration of a PI Vision symbol is done through HTML and CSS, much like the presentation. It differs from the presentation in that it is optional. If the symbol does not support any configuration options, this file can be omitted. The configuration file for the native PI Vision value symbol can be found **INSTALLATION_FOLDER\PIVision\Scripts\app\editor\symbols\sym-value-config.html**.

Symbol Template

```
(function (PV) {

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

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

    var definition = {
        typeName: 'example',
        datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
        visObjectType: symbolVis,
        getDefaultConfig: function() {
            return {
                DataShape: 'Value',
                Height: 150,
                Width: 150,
            };
        }
    };
    PV.symbolCatalog.register(definition);
})(window.PIVisualization);
```

The template file has been already prepared for you for this lab. You can find it under **Lab files** folder on your VM's desktop. The following is directions on how to create it yourself from zero.

1. Create a new file called sym-example.js in your PI Vision installation folder, **INSTALLATION_FOLDER\Scripts\app\editor\symbols\ext**. If the ext folder does not exist, create it.
2. Add the following code to the file, this will initialize the structure used for creating custom symbols. This creates an [IIFE](#), immediately invoked functional expression, which is a JavaScript [function](#) that runs as soon as it is defined. It takes in a PI Vision object that will be used for symbol registration.

```
(function (PV) {
})(window.PIVisualization);
```

3. The first step is to create the visualization object, which will be built on later. In this step, you create 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) {
    function symbolVis() {}
    PV.deriveVisualizationFromBase(symbolVis);
})(window.PIVisualization);
```

4. Then create the symbol definition [Object](#) that will be used to register the symbol with PI Vision. Here we use the PI Vision object passed in to gain access to the symbol catalog. The symbol catalog is the object we use for registering and holding all Vision symbols. We are creating an empty object and passing that into the register function. Since this is in an IIFE, as soon as the browser executes it, it will run the registration code.

```
(function (PV) {  
  function symbolVis() {}  
  PV.deriveVisualizationFromBase(symbolVis);  
  
  var definition = {};  
  PV.symbolCatalog.register(definition);  
})(window.PIVisualization);
```

5. Let's start by building out the required parts of the new symbol. The `typeName` is the internal name that PI Vision will use to register this symbol. It must be unique among all PI Vision symbols. `visObjectType` is the Object we created earlier in step 3. `datasourceBehavior` is used to specify the number of search results that can be used to create this symbol. The options are `None`, `Single`, `Multiple`. `None` is used for static, i.e. not data driven symbols. `Single` is used for a symbol that is based on one PI tag or attribute. `Multiple` is used for a symbol that is based on multiple PI tags, attributes or elements.

```
(function (PV) {  
  function symbolVis() {}  
  PV.deriveVisualizationFromBase(symbolVis);  
  
  var definition = {  
    typeName: 'example',  
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,  
    visObjectType: symbolVis  
  };  
  PV.symbolCatalog.register(definition);  
})(window.PIVisualization);
```

6. Next, let's begin filling in some details about the type of data will be using. Here we have added the `getDefaultConfig` property to the definition object. The `getDefaultConfig` function is used to specify the collection of parameters that should be serialized to the backend database, it returns a JavaScript object. Here we are adding the `DataShape` property to the object returned by `getDefaultConfig`. This property is used to tell the application server the information that this symbol needs to represent the data. In this case, we will be creating a value symbol.

```
(function (PV) {  
  function symbolVis() {}  
  PV.deriveVisualizationFromBase(symbolVis);  
  
  var definition = {  
    typeName: 'example',  
    datasourceBehavior: PV.Extensibility.Enums.DatasourceBehaviors.Single,
```

```

visObjectType: symbolVis,
getDefaultConfig: function() {
    return {
        DataShape: 'Value'
    };
}
};
PV.symbolCatalog.register(definition);
})(window.PIVisualization);

```

- To fix the sizing issue, update the `getDefaultConfig` function's return value to return both `Height` and `Width`. Also, we need to add an initialization function to the definition object. The `init` function is a function that will be called when the symbol is added to a display. The `init` function is defined on the prototype of the symbol container object created in `deriveVisualizationFromBase`.

```

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

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

    symbolVis.prototype.init = function () {
    }

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

```

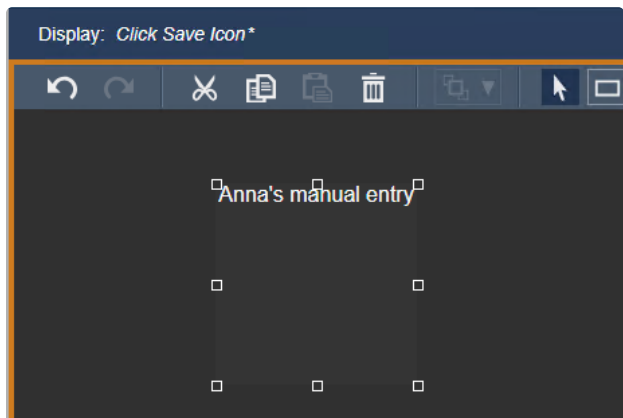

2. Loading example symbol

Let's start with testing a working example symbol to make sure everything loads fine. This example symbol files are located under the **Lab files** folder on your Desktop. We will also give our symbol more meaningful name. Remember: Extensibility model defaults to using the `typeName` property when looking for html and css files.

2.1 Directions


1. Rename example symbol to **data-entry**
2. Rename symbol files accordingly and copy them to the `ext` folder
3. In `.html` add "`<your name> manual entry`" to the existing `<div>`
4. Open PI Vision and make sure your symbol can be loaded

The result should look like this:



2.2 Solution

1. Rename **sym-example.js** to **sym-data-entry.js**. Do the same for sym-example-template.html

 **sym-data-entry.js**

 **sym-data-entry-template.html**

2. Open **sym-example.js** and replace **typeName** value to 'data-entry'

```
...  
var definition = {  
    typeName: 'data-entry',  
    ...  
};  
...  
sym-data-entry.js
```

3. In **sym-data-entry-template.html** add "<your name> manual entry" to the existing <div>

```
...  
<div class="symbol-text">  
    My manual entry  
</div>  
sym-data-entry-template.html
```

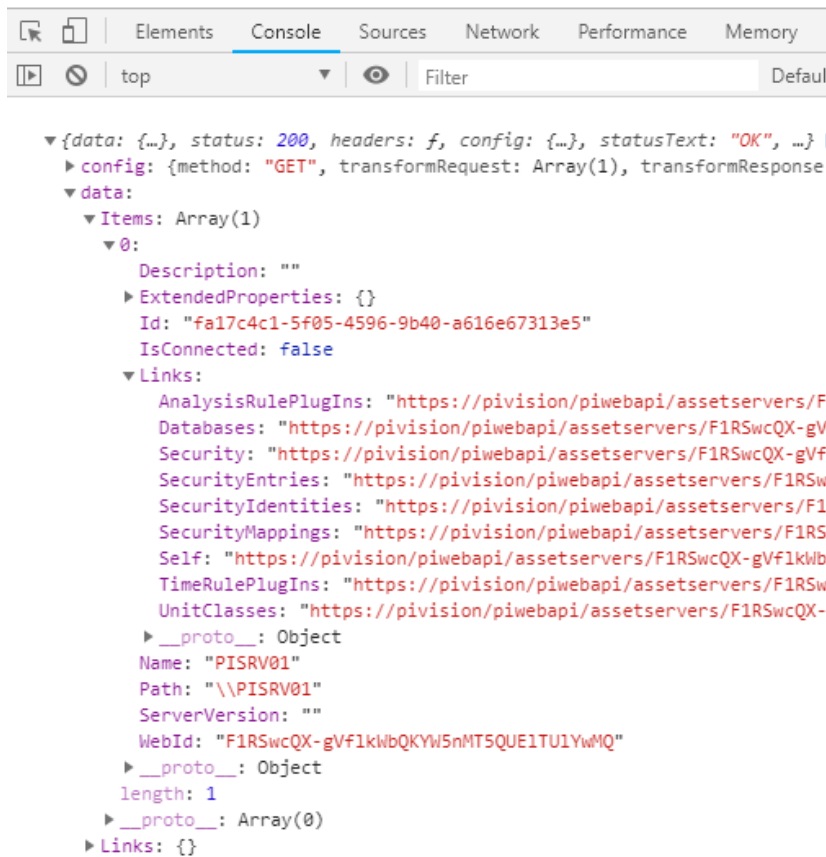
3. Making a test HTTP request from custom symbol to PI Web API

PI Web API is a [REST](#) service that allows communication with PI System in a form of HTTP requests. Requests can be of different types, in this lab we will focus on two of them: GET and POST. To make HTTP request from our custom symbol, we will use AngularJS [\\$http provider](#). To get familiar with this provider, we will now make a basic GET request to PI Web API in order to get all available Asset Servers.

3.1 Directions

1. Inject '\$http' provider string into symbol definition
2. Pass \$http provider as a parameter to the `init` function
3. Make GET HTTP request for the list of all available [asset servers](#). Output results to console

The result in developer console should look like this:



```
{data: {...}, status: 200, headers: f, config: {...}, statusText: "OK", ...} |
  ▶ config: {method: "GET", transformRequest: Array(1), transformResponse
  ▼ data:
    ▼ Items: Array(1)
      ▼ 0:
        Description: ""
        ▶ ExtendedProperties: {}
        Id: "fa17c4c1-5f05-4596-9b40-a616e67313e5"
        IsConnected: false
        ▼ Links:
          AnalysisRulePlugIns: "https://pivision/piwebapi/assetserver/F
          Databases: "https://pivision/piwebapi/assetserver/F1RSwcQX-gV
          Security: "https://pivision/piwebapi/assetserver/F1RSwcQX-gVf
          SecurityEntries: "https://pivision/piwebapi/assetserver/F1RSw
          SecurityIdentities: "https://pivision/piwebapi/assetserver/F1
          SecurityMappings: "https://pivision/piwebapi/assetserver/F1RS
          Self: "https://pivision/piwebapi/assetserver/F1RSwcQX-gVf1kwb
          TimeRulePlugIns: "https://pivision/piwebapi/assetserver/F1RSw
          UnitClasses: "https://pivision/piwebapi/assetserver/F1RSwcQX-
        ▶ __proto__: Object
        Name: "PISRVO1"
        Path: "\\PISRVO1"
        ServerVersion: ""
        WebId: "F1RSwcQX-gVf1kwbQKYW5nMT5QUE1TU1YwMQ"
        ▶ __proto__: Object
        length: 1
        ▶ __proto__: Array(0)
        ▶ Links: {}
```

3.2 Solution

1. Inject '\$http' provider string into symbol definition

```
...
var definition = {
  typeName: 'data-entry',
  inject: ['$http'],
  ...
};
...
```

sym-data-entry.js

2. Pass \$http provider as a parameter to the `init` function

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

sym-data-entry.js

3. Make GET HTTP request for the list of all available asset servers. Output results to console

```
...
var baseUrl = PV.ClientSettings.PIWebAPIUrl;
symbolVis.prototype.init = function (scope, elem, $http) {
  $http.get(baseUrl + '/asset servers').then(function (response) {
    console.log(response);
  });
};
...
```

sym-data-entry.js

4. Obtaining stream WebId with GetByPath

Before submitting data to PI Web API, we first need to know some metadata information from it (more specifically stream's [WebId](#)). In this exercise we will obtain attribute's WebId by making [GetByPath](#) request on symbol initialization. Path of your symbol attribute comes from `scope.symbol.DataSources` array. Paths in this array are stored with three character prefix ("af:" or "pi:") – this prefix has to be removed before using it for the PI Web API request.

4.1 Directions

1. Use `scope.symbol.DataSources` to get symbol's attribute path. Under `init()` add `path` variable and store there symbol attribute's path. Use [slice\(\)](#) to snip the first 3 characters of the path string
2. Then (under `init()` as well), add [\\$http.get\(\)](#) request and search for the attribute with [GetByPath](#)
3. Output returned WebId to console

4.2 Solution

1. Use `scope.symbol.DataSources` to get symbol's attribute path. Under `init()` add `path` variable and store there symbol attribute's path. Use `slice()` to snip the first 3 characters of the path string

```
...
symbolVis.prototype.init = function (scope, elem, $http) {
    var path = scope.symbol.DataSources[0].slice(3);
};
...
sym-data-entry.js
```

2. Then (under `init()` as well), add `$http.get()` request and search for the attribute using path

```
...
symbolVis.prototype.init = function (scope, elem, $http) {
    var path = scope.symbol.DataSources[0].slice(3);
    $http.get(baseUrl + '/attributes?path=' + path).then(function (response) {
    });
};
...
sym-data-entry.js
```

3. Output returned WebId to console

```
...
symbolVis.prototype.init = function (scope, elem, $http) {
    var path = scope.symbol.DataSources[0].slice(3);
    $http.get(baseUrl + '/attributes?path=' + path).then(function (response) {
        console.log(response.data.WebId);
    });
};
...
sym-data-entry.js
```

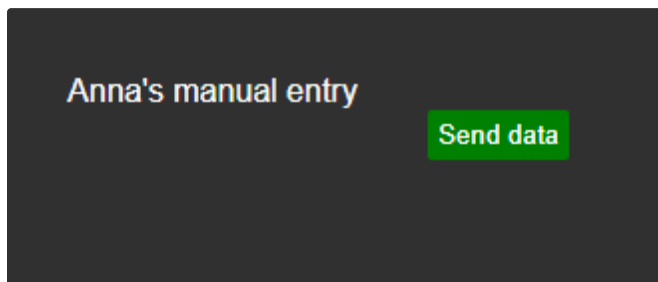
5. Using WebId in Update Value request

Now that we know WebId of our stream, we can use it to construct [UpdateValue](#) request. Notice that this is a POST request and it uses [different syntax](#) from requests we made previously in this lab. We will also need to supply a value with this request. For now we will use a mock value hardcoded in our code. To make it easier to test this, we will also add a button.

5.1 Directions

1. Add [<button>](#) to `sym-data-entry-template.html` with text "Submit". Assign it `class="send-button"` for styling.
2. Add [ng-click](#) directive and point it to `sendValue()` function
3. In `sym-data-entry.js`, add `sendValue()` function under `init()` method and put it on `scope`
4. In `sendValue()` method, make an [\\$http.post\(\)](#) request:
 - Instead of outputting WebId to console, assign WebId value to scope's `webId` property
 - Construct [UpdateValue](#) request URL using WebId we got from the previous exercise
 - Submit sample data object with `Timestamp` ("*") and `Value` (any number) properties. Remember that it needs to be [JSON.stringify](#)'ed

The result should look like this:



5.2 Solution

1. Add [<button>](#) to `sym-data-entry-template.html` with text "Submit"

```
...  
<div class="symbol-text">  
  My manual entry  
  <button class="send-button">Submit</button>  
</div>
```

sym-data-entry-template.html

2. Add [ng-click](#) directive and point it to `sendValue()` function

```
...  
<div class="symbol-text">  
  My manual entry  
  <button ng-click="sendValue()">Submit</button>  
</div>
```

sym-data-entry-template.html

3. In `sym-data-entry.js`, add `sendValue()` function under `init()` method and put it on **scope**

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

sym-data-entry.js

4. In `sendValue()` method, make an `$http.post()` request:
 - Instead of outputting `WebId` to console, assign it to scope's `webId` property
 - Construct [UpdateValue](#) request URL using `WebId` we got from the previous exercise
 - Submit sample data object with `Timestamp` ('*') and `Value` (any number) properties. Remember that it needs to be `JSON.stringify`'ed

```
...
symbolVis.prototype.init = function (scope, elem, $http) {
  ...
  $http.get(baseUrl + '/attributes?path=' + path).then(function (response) {
    scope.webId = response.data.WebId;
  });

  scope.sendValue = function () {
    var data = JSON.stringify({
      Timestamp: '*',
      Value: 0
    });
    $http.post(baseUrl + '/streams/' + scope.webId + '/value', data);
  }
};
...
```

sym-data-entry.js

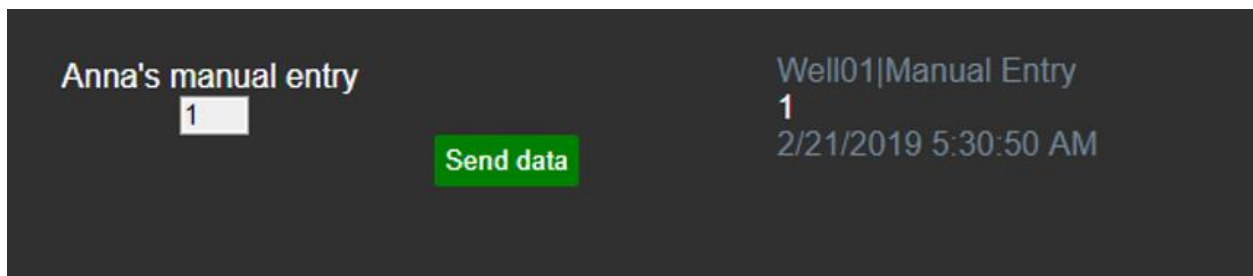
6. Adding user input

We can send data to PI, great! But only hardcoded data at the moment – not so great. In this exercise we will replace hardcoded part with actual user input.

6.1 Directions

1. In `sym-data-entry-template.html`, add `<input>` element with type number
2. Bind it to `scope.runtimeData.input` using `ng-model` directive
3. In `sym-data-entry.js`, under `sendData()` function replace the data `Value` property with actual user input

The result should look like this (value symbol on the right side is added to confirm that data gets updated):



6.2 Solution

1. In `sym-data-entry-template.html`, add `<input>` element with type number

```
...
<div class="symbol-text">
  My manual entry
  <input type="number"></input>
  <button ng-click="sendValue()" class="send-button">Submit</button>
</div>

```

`sym-data-entry-template.html`

2. Bind it to `scope.runtimeData.input` using `ng-model` directive

```
...
<div class="symbol-text">
  My manual entry
  <input type="number" ng-model="runtimeData.input"></input>
  <button ng-click="sendValue()" class="send-button">Submit</button>
</div>

```

`sym-data-entry-template.html`

3. In `sym-data-entry.js`, under `sendData()` function replace the data **Value** property with actual user input

```
...
symbolVis.prototype.init = function (scope, elem, $http) {
  ...
  scope.sendValue = function () {
    var data = JSON.stringify({
      Timestamp: '*',
      Value: scope.runtimeData.input
    });
    $http.post(scope.valueLink, data);
  }
};
...

```

`sym-data-entry.js`

7. References

1. Documentation on PI Vision Extensibility
<https://customers.osisoft.com/s/productcontent?id=a7R1I000000XybxUAC>
2. Online course on PI Vision Extensibility
<https://pisquare.osisoft.com/community/Learn-PI/customizing-pi-vision-with-extensibility>
3. Forum with discussions and questions
<https://pisquare.osisoft.com/community/developers-club/pi-visualization-development>
4. AngularJS
<https://angularjs.org/>
5. OSIsoft GitHub with symbol examples and tutorials
<https://github.com/osisoft/PI-Vision-Custom-Symbols>
6. Kerberos configuration blog
<https://pisquare.osisoft.com/groups/security/blog/2016/08/01/coresight-squared-what-s-next-kerberos-and-more>
7. Documentation on Kerberos configuration
<https://livelibrary.osisoft.com/LiveLibrary/content/en/vision-v1/GUID-0CA88969-D6C4-43CA-862F-802F42E6F102>



Have an idea how to
improve our products?

**OSIsoft wants to hear
from you!**

<https://feedback.osisoft.com/>





Save the Date!

OSIsoft PI World Users Conference in Gothenburg, Sweden. September 16-19, 2019.

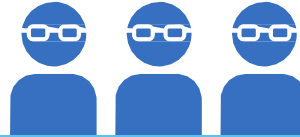
Register your interest now to receive updates and notification early bird registration opening.

https://pages.osisoft.com/UC-EMEA-Q3-19-PIWorldGBG-RegisterYourInterest_RegisterYourInterest-LP.html?_ga=2.20661553.86037572.1539782043-591736536.1533567354



OSIsoft.

PI SYSTEM LEARNING



CONTINUE YOUR PI SYSTEM LEARNING



After the conference, the **PI SYSTEM LEARNING** does not have to stop. All registered attendees for **PI World SFO 2019** will have access to all PI World Hands-on Lab cloud environments for 21 days using the discount cod below. You will receive detailed instructions via email after the conference.

Discount Code: 2019UCSF-LAB-100

Offer expires June 30, 2019

PI SYSTEM LEARNING MADE EASY

A new OSIsoft Learning experience is coming soon!



VISIT [LEARNING.OSISOFT.COM](https://learning.osisoft.com)

