1. Vision . . . 3
1.1 Vision Designer Interface . . . 7
1.2 Vision Windows . . . 16
1.2.1 Window Types . . . 24
1.2.2 Popup Windows . . . 27
1.2.2.1 Parameterized Popup Windows . . . 32
1.2.3 Navigation Strategies in Vision . . . 38
1.2.3.1 Navigation - Tab Strip . . . 42
1.2.3.2 Navigation - Two Tier . . . 44
1.2.3.3 Navigation - Tree View . . . 48
1.2.3.4 Navigation - Forward and Back Buttons . . . 52
1.2.3.5 Navigation - Drill Down . . . 56
1.2.3.6 Navigation - Menubar . . . 59
1.2.3.7 Navigation - Retargeting . . . 62
1.3 Working with Vision Components . . . 64
1.3.1 Creating Vision Components . . . 76
1.3.2 Vision Component Customizers . . . 79
1.3.3 Drawing Tools . . . 90
1.3.3.1 Shape Geometry . . . 95
1.3.3.2 Fill and Stroke . . . 100
1.3.4 Images and SVGs in Vision . . . 105
1.3.5 Comparison Charts . . . 111
1.3.6 HTML in Vision . . . 125
1.3.7 Localization in Vision . . . 127
1.4 Binding Types in Vision . . . 129
1.4.1 Property Bindings in Vision . . . 133
1.4.2 Tag Bindings in Vision . . . 135
1.4.3 Indirect Tag Bindings in Vision . . . 138
1.4.4 Tag History Bindings in Vision . . . 142
1.4.5 Expression Binding in Vision . . . 149
1.4.6 Named Query Bindings . . . 151
1.4.7 DB Browse Bindings . . . 154
1.4.8 SQL Query Bindings in Vision . . . 159
1.4.9 Cell Update Bindings . . . 163
1.4.10 Function Bindings . . . 173
1.4.11 Color Animation in Vision . . . 175
1.5 Vision Templates . . . 182
1.5.1 Creating a Template . . . 188
1.5.2 Template Indirection . . . 195
1.5.3 Using the Template Repeater . . . 201
1.5.4 Using the Template Canvas . . . 209
1.6 Security in Vision . . . 219
1.6.1 Component and Window Security . . . 222
1.6.2 Security in Scripting . . . 224
1.7 Scripting in Vision . . . 228
1.7.1 Script Builders in Vision . . . 233
1.7.2 Component Events . . . 240
1.7.3 Extension Functions . . . 250
1.7.4 Custom Component Methods . . . 256
1.7.5 Focus Manipulation . . . 259
1.7.6 Client Event Scripts . . . 263
1.7.7 Read a Cell from a Table . . . 269
1.8 Historian in Vision . . . 273
1.8.1 Using the Vision Easy Chart . . . 276
1.8.1.1 Easy Chart - Axes . . . 281
1.8.1.2 Easy Chart - Subplots . . . 289
1.8.1.3 Easy Chart - Pen Names and Groups . . . 293
1.8.1.4 Easy Chart - Pen Renderer . . . 297
1.8.1.5 Easy Chart - Digital Offset . . . 300
1.8.1.6 Easy Chart - Calculated Pens . . . 303
1.8.1.7 Using the Tag Browse Tree for Charting . . . 311
1.8.1.8 Indirect Easy Chart . . . 314
1.8.1.9 Easy Chart - Database Pens . . . 319
1.8.2 Using the Classic Chart . . . 322
1.8.3 Other Vision Trending Charts . . . 330
1.9 Vision Client Tags . . . 333
1.10 Vision Project Properties . . . 338
1.10.1 Client Update Modes . . . 351
1.10.2 Setting Up Auto Login . . . 355
1.10.3 Using Touch Screen Mode . . . 357
1.11 Common Tasks in Vision . . . 362
1.11.1 Component Animation . . . 363
1.11.2 Custom Input Template . . . 368
1.11.3 Client Tags for Indirection . . . 371
1.11.4 High Performance HMI Techniques . . . 383
1.11.5 Open Dynamic Windows on Startup . . . 386
1.11.6 Tank Cutaway . . . 388
1.11.7 Dropdown List Example . . . 391
1.11.8 Multi-Monitor Clients . . . 398

1.12 Local Client Fallback 401
1.13 Vision Client Interface 403

Vision

The Vision module is a tool for creating and maintaining an interactive, accurate Human-Machine Interface (HMI) for your site. Many other modules, as well as platform level features, seamlessly integrate with the Vision module, providing a simple method of visualizing and presenting data to your users.

On this page ...

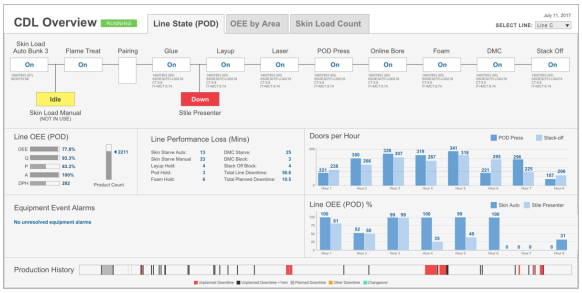
- Windows
- Navigation
- Components
- Bindings
- Graphics
  - Scalable Vector Graphics
- Templates
- Vision Client Disconnections

Windows

Vision Windows are the basic building blocks for all of your HMI screens. There are three basic window configurations that define how a window behaves:

- **Main Windows** - A main window is one that is set to start maximized to take up all available screen space (except space used by any docked windows).
- **Popup Windows** - A popup window is one that appears (pops up) when the user performs an action such as clicking the mouse, pressing a function key, or touching a button (if using a touchscreen). Popup windows usually remain on top of the current window until closed, enabling users to quickly choose options or settings before returning to the previous window.
- **Docked Windows** - A docked window is set to a static location on the screen. Docked windows are often used to hold navigation trees or status information that needs to remain on the screen at all times.

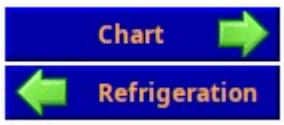
By changing a window's properties, you can transform any window into various configurations, with each behaving differently based on those settings. Passing custom parameters into windows allows you to create the window once, and then re-use your screens multiple times within the same project. You get to choose what windows are available on startup and how your navigation is configured. The following is an example of a Vision module screen displaying a docked window for navigation on the top of a main window covering the remaining available screen space.



Navigation

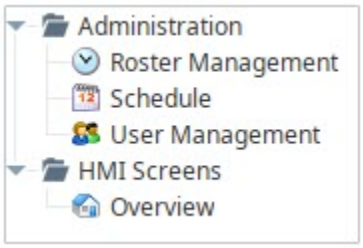
A large number of navigation options exist in the Vision module. For example, docked windows can be set up with navigation trees, tab strips, or menu bars. Components such as buttons can be used to navigate to other windows. A graphic or photograph of a map can be customized with clickable zones. Many of the common options are available as project templates that you can take advantage of when first creating your project. Here are some examples:

- **Back/forward buttons**



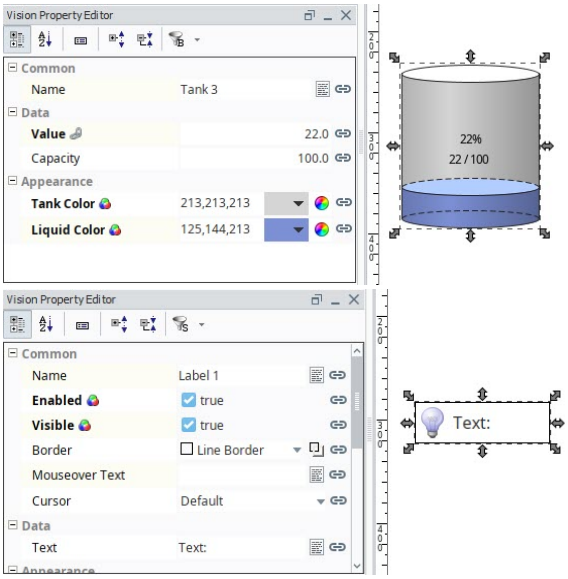
- **Navigation tab strips**

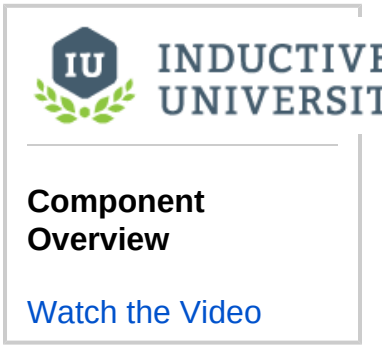
- **Multi-tier navigation**



Components

Components are building blocks for your project. The Vision module has a variety of built-in components s uch as displays, buttons, charts, and other elements that display information. Each component has multiple properties that control its appearance, behavior, and data. For example, a Tank has a level, capacity, and a liquid color, while a Label has text, font, and an image. You can enhance components with custom properties to create additional functionality.





Bindings

A binding is a mechanism that allows a property on a component to change based on a a change to a value elsewhere in Ignition. For example, with binding, the liquid level displayed in a tank graphic can be bound to the realtime liquid level in a tank. The value of a Tag could be bound to a linear scale, a meter, or a label on your window. The power of bindings comes from the variety of binding types.

Click on the following links for complete information about binding types:

- **Properties** - Property Binding, Cell Update Binding, Component Styles
- **Tags** - Tag Binding, Indirect Tag Bindings in Vision, Tag History Binding
- **Expressions** - Expression Binding
- **Databases** - DB Browse Binding, SQL Query Binding, Named Query Bindings
- **Functions** - Function Binding

Graphics

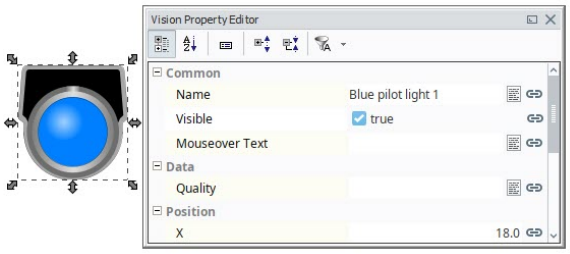
In addition to standard components, the Vision module supports the use of SVG, PNG and JPEG images on Vision Windows. You can create your own images and import them into your project or use Ignition's 2 D Drawing tools to create graphics.

You can use the built-in SVGs from Symbol Factory, which contains hundreds of ready-to-use graphics, or use the raster image library with an Image component to get a jump start on your project.

Scalable Vector Graphics

Scalable Vector Graphics (SVG) have several advantages over other graphic types. Because they are vector graphics, they can be scaled without a loss of clarity or resolution. Additionally, you can drill into an SVG to change individual parts of the image. To use an SVG in Ignition, simply drag the file directly onto the window in which you want it to appear.

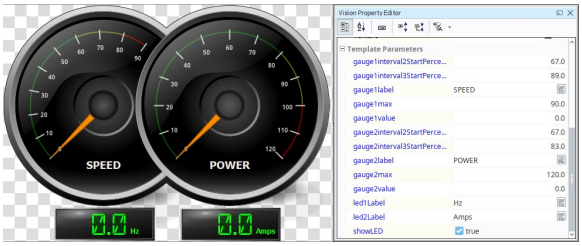
Watch the Video



Templates

Components and images can be combined to create Vision Templates. These are re-usable objects that can be configured once and used throughout your project. Templates work under a principal of inheritance. When a change is made to a template, that change is inherited by each instance of that template. Most templates use one or more custom properties (Template Parameters) to tie data from a window to the internals of the template.

The Cloud Templates Browser portal in the Designer gives you access to pre-built templates. Ignition community members can also share their own templates in the Cloud Templates.



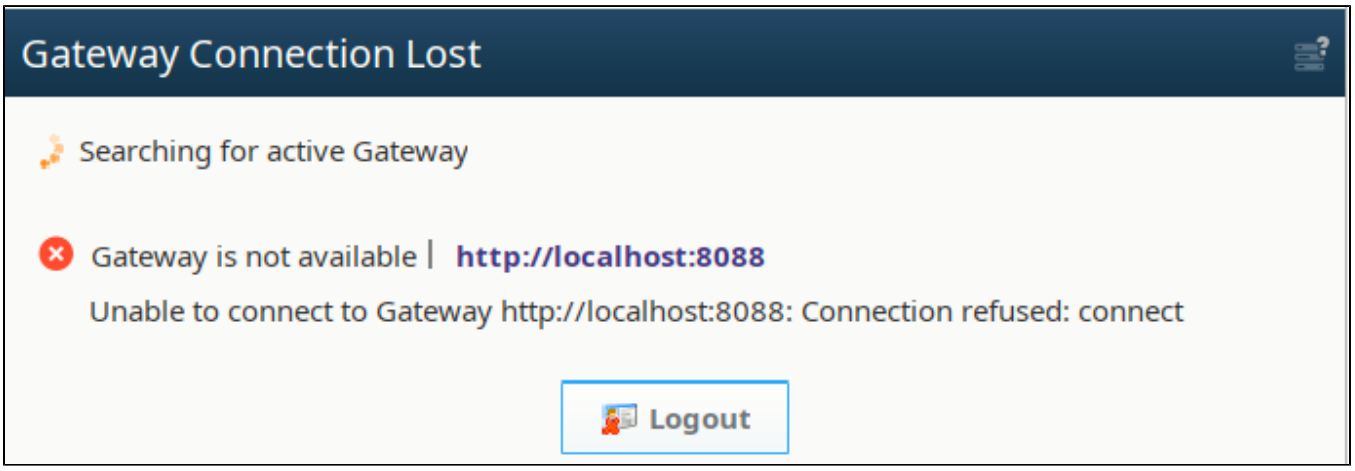
INDUCTIVE UNIVERSITY

About Templates

Watch the Video

Vision Client Disconnections

If a Vision Client loses its connection to the Ignition Gateway, a Connection Lost message will be displayed immediately on your screen:



The Vision Client will continuously attempt to reconnect to the Gateway. When the connection between the Vision Client and the Gateway is re-established, your Vision Client will resume its normal operation. If the project associated with the Vision Client is somehow deleted, the Client will fail to reconnect because its source project is missing. A project can be manually deleted or overwritten by a Gateway restore.

In This Section ...

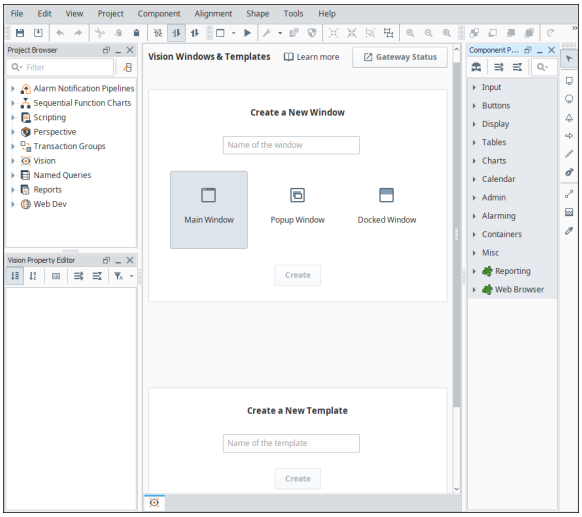
Vision Designer Interface

Vision Designer Workspace

The Vision Designer Interface is where the bulk of the designer's work is done. The Vision Designer Interface provides some built-in functionality to help you get started designing your project whether you are logging into your project for the first time or the 50th time. From the Vision Windows & Templates Welcome tab, you can easily create Main windows, Popup windows, Docked windows and Templates. It even shows you the recently modified windows, so picking up where you left off the last time you logged into your project, is right at your fingertips.

You can also check the Gateway Status from the Designer Interface and see all the Vision Clients that are running along with the client details and stats.

When looking at a Vision specific element in the Designer, such as a window or template, the Designer is organized with some panels that are specific to the Vision Designer Interface, such as the **Property Editor** and **Component Palette**. Other elements of the workspace that are shared between spaces are discussed in the General Designer Interface.



On this page ...

- Vision Designer Workspace
- Component Palette
- Vision Property Editor
  - Filters
  - Binding Icon
  - Status Indication
  - Dropdown Lists in Properties
  - Common Properties
- Vision Menubar
  - File Menu
  - Edit Menu
  - View Menu
  - Project Menu
  - Component Menu
  - Alignment Menu
  - Shape Menu
- Tools Menu

The Designer User Interface

Watch the Video

Component Palette

The Vision module comes with a host of useful components out of the box, such as buttons, text areas, dropdowns, charts, many of which are specialized for industrial controls use. The Component Palette is located on the right side of the Designer workspace. The basic workflow is to drag a component from the component palette and drop it into a container on a window. From there, you can use the mouse to drag and resize the component into the correct position. While the component is selected, you can use the Property Editor panel to alter the component's properties, which changes the component's appearance and behavior.

Vision Property Editor

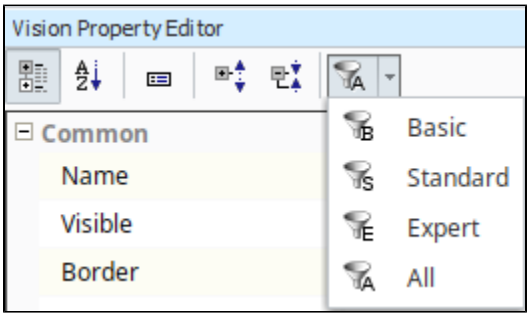
The Vision Property Editor is a dockable panel that appears in the Designer's central workspace, usually in the lower left corner. It displays the properties of the selected component. If more than one component is selected, it will show all properties that the current selection set have in common. Hovering your mouse over a property will display a tooltip that gives a description of the property, as well as its data type and scripting name. Alternately, you can click on the Show/Hide Description Area  icon to bring up the description area which displays the same information for the currently selected property.

You can also change how the properties are sorted in the property editor. By default, they are sorted with the Categorized  icon, with similar components grouped under different categorical headers. However, they can also be sorted in alphabetical order by clicking on the Alphabetic  icon.

Filters

It is common for components to have many properties, so by default the Property Editor only shows the **Basic** properties. These are the properties that you'll most commonly want to set or bind for a given component. However, the property filter can be changed to show different sets of properties. The designer will remember your selection for future sessions.

- **Basic:** The Name property and any very commonly used properties. Most components only show two to four properties in Basic.
- **Standard:** Most of the common properties that a designer would want to use. Few or none of the Expert properties are in the Standard list.
- **Expert:** The properties that are most commonly used with more advanced features of the component. Few or none of the Standard properties are in the Expert list.
- **All:** All properties



Most users find it best to set the property filter to All, so they can see all of the properties available to them at all times.

Binding Icon

To the right of most properties is the **Binding** icon. Click this icon to modify the property binding that is driving that property. You can only use this button when the window workspace is not in Preview mode. Some properties cannot be bound because their datatype is not supported by the binding system. You can still use scripting to affect these properties.

Status Indication

The name of a property in the **Property Editor** conveys important information about that property:

- A blue name indicates that the property is a custom property.
- A **bold name** with a Link icon next to the property indicates that the property is bound using a property binding.
- A **bold name** with a Color Palette icon indicates that the property is being affected by the component styles settings.
- A **red bold** name with a Warning icon indicates that the property is double-bound. This means that two things, a property binding and the styles settings are both trying to drive the property value. This will result in errors as the two systems fight each other to write to the property.

Dropdown Lists in Properties

Some of the properties you will encounter on components will have a dropdown list instead of a field to type into. The property description will say it is an integer value, and in most of these cases you can still create a binding on that property. These dropdown lists are an enumeration, meaning each element in the dropdown has an integer value. In all cases, the first value in the list is 0, the second is 1, the third is 2, and so on. You can use this knowledge to create a dropdown list on-screen for your operators that matches the list. In this case, you would just bind this property to the Selected Value of the dropdown.

Common Properties

Every component has properties arranged into categories based on what it has available (i.e., Common, Behavior, Data, Appearance, Layout, etc.). Each component has a different list of properties to effect how it behaves, but every component has the **Common** group of properties located at the top of the list. These **Common** properties will behave the same for all components. Here's a list for each Common property and when it might be used.

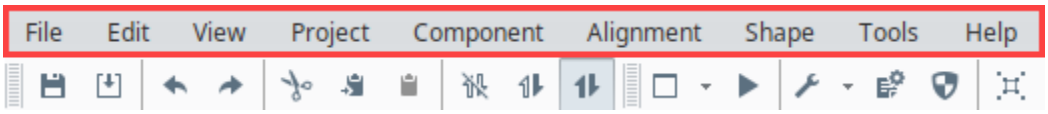
Function	Description
Name	The name of the component. This string is used to identify your components in the Project Browser. This is especially important for Bindings and Scripting. Binding is allowed on this property, but it is recommended to **never bind this property.** Binding it can break your scripts, bindings, and cause errors.
Enabled	This Boolean controls whether a component can be interacted with. Most commonly used with data entry components to allow the user to see the value, but not change it.
Visible	This Boolean controls whether the component is shown on the window. You can bind this property to show/hide the component

<table>
<tr><td></td><td>based on any logic you want (i.e., security, process step, etc.).</td></tr>
<tr><td>Border</td><td>The border that surrounds the component. There is a dropdown to select from a list of common borders, and a button to the right to manually edit a border from several different options with a second tab that shows Titled Borders. When binding this property, note that this is a complex data type. It is a Java Border data type, not a string or an enumeration. The common ways to make this property dynamic are to bind it with an Expression binding type or to set it through a script, but using the Expression binding is preferred. If you are using an Expression binding, you must use the toBorder() expression function to return the correct data type. If you are using a script, you need to make sure you use the Java Border data type. See the Java documentation for more information on setting a border through scripting.</td></tr>
<tr><td>Mouseover Text</td><td>The text that is displayed when a user moves the mouse over the component. This string is commonly used to provide your operators more information about an object (i.e., showing the PLC address of an on-screen value, or telling the operator exactly what will happen when a button is pressed). HTML is allowed in this property.</td></tr>
<tr><td>Cursor</td><td>The mouse pointer image to use when the operator moves the mouse over the component. This int property corresponds to one of the options in the list. Selecting 'default' means the operating system decides what pointer to use.

<table>
<tr><th>Value</th><th>Cursor</th></tr>
<tr><td>0</td><td>Default</td></tr>
<tr><td>1</td><td>Crosshair</td></tr>
<tr><td>2</td><td>Text</td></tr>
<tr><td>3</td><td>Wait</td></tr>
<tr><td>4</td><td>SW Resize</td></tr>
<tr><td>5</td><td>SE Resize</td></tr>
<tr><td>6</td><td>NW Resize</td></tr>
<tr><td>7</td><td>NE Resize</td></tr>
<tr><td>8</td><td>N Resize</td></tr>
<tr><td>9</td><td>S Resize</td></tr>
<tr><td>10</td><td>W Resize</td></tr>
<tr><td>11</td><td>E Resize</td></tr>
<tr><td>12</td><td>Hand</td></tr>
<tr><td>13</td><td>Move</td></tr>
</table>
</td></tr>
</table>

Vision Menubar

There is a menubar at the top of the Designer Workspace that provides functionality when working in the Vision workspace. Each menu dropdown reveals a host of functions related to the menu item. The other menus shared between Vision and Perspective are discussed in the General Designer Interface.

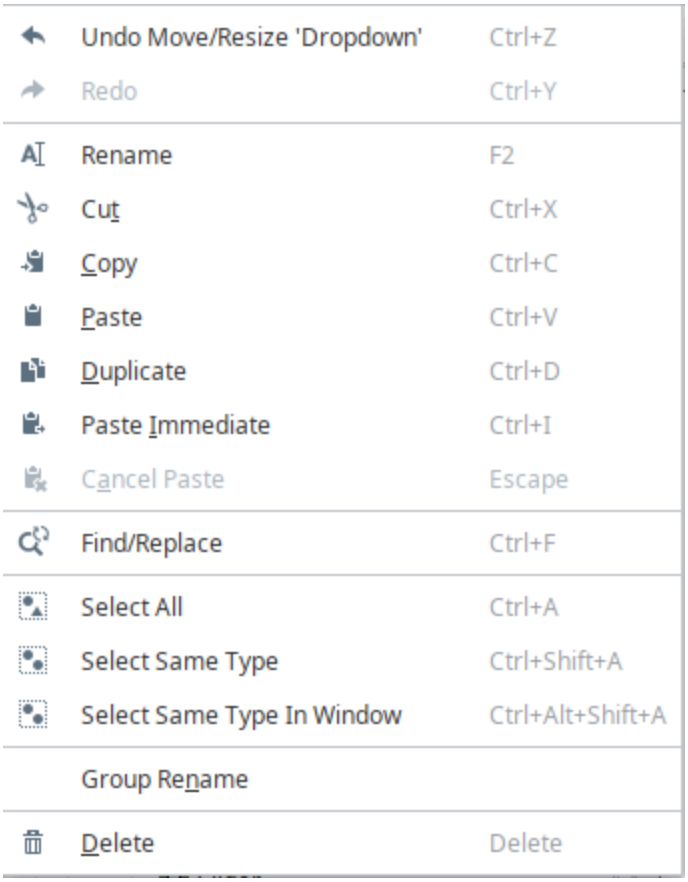


File Menu

See General Designer Interface.

Edit Menu

The **Edit Menu** is similar to other applications edit menus as it provides much of the basic copy/paste functionality. You can also access this menu by right-clicking on an item.



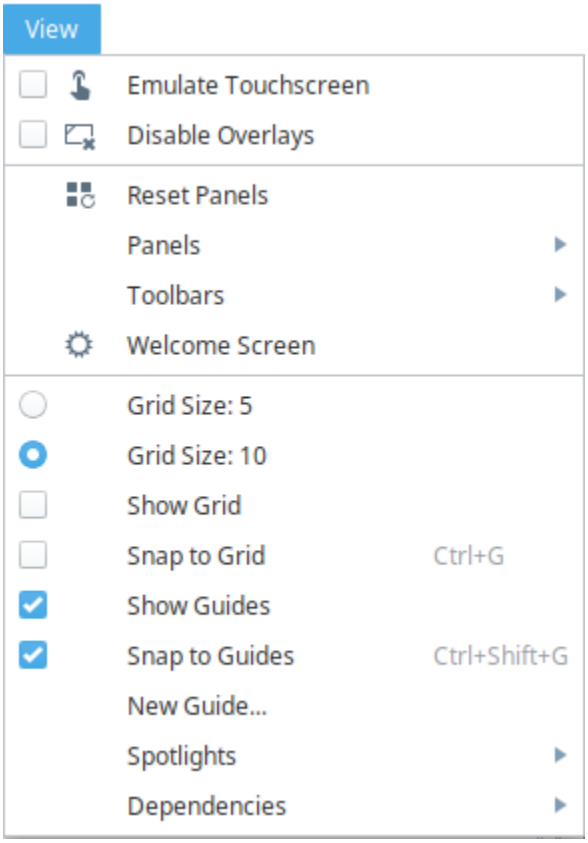
The following feature is new in Ignition version **8.1.24**
Click here to check out the other new features

Note: You can use certain Edit Menu actions for Vision Component properties. For example, you can copy a component's exposed property by selecting it in the Vision Property Editor and choosing "Copy" from the Edit Menu.

Function	Description
Undo and Redo	Can be used to revert to the previous state, essentially removing the last change, or redoing it again after having been removed. This has a large queue that can be traversed, but does not include every change (i.e., Tag edits cannot be undone).
Cut/Copy /Paste /Duplicate	These functions much the same as they do everywhere else. Most things in the Designer can be copied and pasted elsewhere, from individual components on the window to entire folders of windows. The difference is that when using Paste with an object on a window, it will instead create a paste action, and allow you to move the mouse and select where you want to paste it, clicking the mouse to confirm. Cancel Paste will cancel the paste action, while paste immediate will bypass the paste action, and instead immediately paste the object from where it was cut or copied from.
Find /Replace	Brings up the Find and Replace interface to allow you to find specific objects within the project. See also: Find and Replace
Select All	Selects All siblings in the same container as the currently selected component.
Select Same Type	Selects all components of the same type in the same container as the currently selected component.
Select Same Type in Window	Selects all components of the same type as the currently selected component, regardless of what container they are in.
Group Rename	Renames a group of components to a prefix with a number afterwards. For example, if your prefix is Button, it will rename all selected components Button (1), incrementing the number each component.
Delete	Deletes the currently selected component. This can also be done using the delete key.

View Menu

The **View Menu** allows you to manipulate how various objects look or act in the Designer.

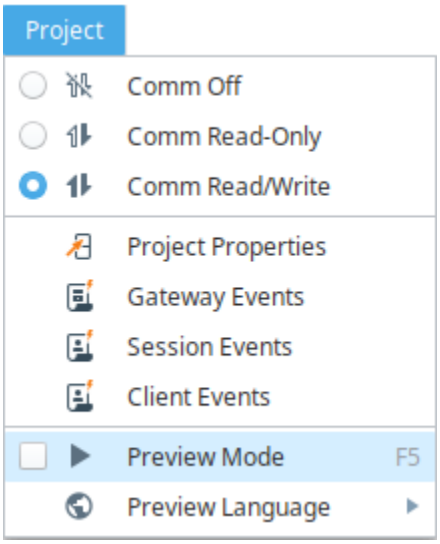


Function	Description
Emulate Touchscreen	Simulates Touchscreen mode in the Designer to be able to test it without having to open a client.
Disable Overlays	Disables the red or gray overlays on components because of a bad Tag or binding, but only in this Designer session.
Reset Panels	Resets panels (Project Browser, Tag Browser, etc.) to the default panel configuration.
Panels	Allows you to enable or disable certain panels within the Designer.
Toolbars	Allows you to enable or disable certain toolbars within the Designer.
Welcome Screen	Takes you to the welcome screen in the Designer, or reopen it if it had been closed.
Grid Size	Allows you to select a grid size of 5 or 10.
Show Grid	Toggles the grid on and off.
Snap to Grid	Changes click-and-drag behavior to snap components to grid lines. This works even when Show Grid is off.
Show Guides	Shows any guide lines.
Snap to Guides	Changes click-and-drag behavior to snap components to any created guidelines.
New Guide	Adds a guide line to the current window.

Spotlights	Puts a highlighted border around components that have the selected spotlight. Bound objects will get a green highlight, objects with scripting will get a blue highlight, and invisible objects will get a pink highlight. If a component has multiple highlights, and both are enabled, it will alternate the colors throughout the highlight.
Dependencies	Shows the binding dependencies (as arrows) based on the selected component or components. Show Supporters will show all components that the currently selected component is bound to, Show Dependents will show all components that are bound to the currently selected component, and Show All will show all of the bindings, regardless of the selected components.

Project Menu

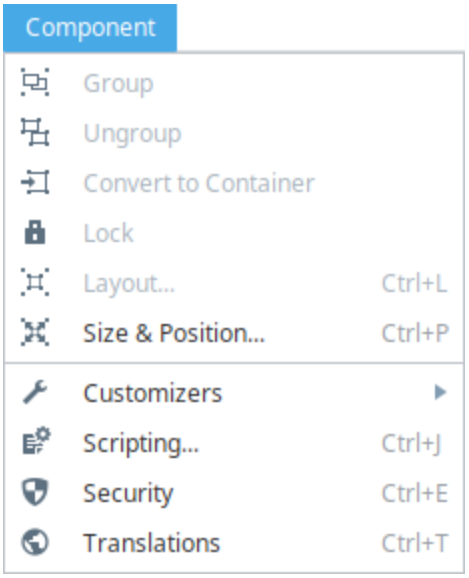
The **Project Menu** is where many project specific settings can be changed.



Function	Description
Designer Comms	The Comm settings allow you to select the level of communication the Designer can have with the Gateway. By default, this is set to Comm Read-Only, which will make any information coming from the Gateway read only, but this can be changed to Comm Off which will prevent Gateway communication, or Comm Read/Write, which will allow both read and write communications between the Gateway. The default that the Designer opens at can be changed in the Project Properties. See also: Communication Modes
Properties	Opens up the Project Properties window, allowing project settings to be changed. See also: Project Properties
Event Scripts	Opens up the appropriate event script window, either client or Gateway. These can also be accessed from the Project Browser. See also: Client Event Scripts and Gateway Event Scripts.
Preview Mode	Puts the Designer into Preview Mode, allowing you to interact with it like a client. See also: Previewing the Project
Preview Language	Determines the language that the Designer will revert to when in Preview Mode. See also Localization in Vision

Component Menu

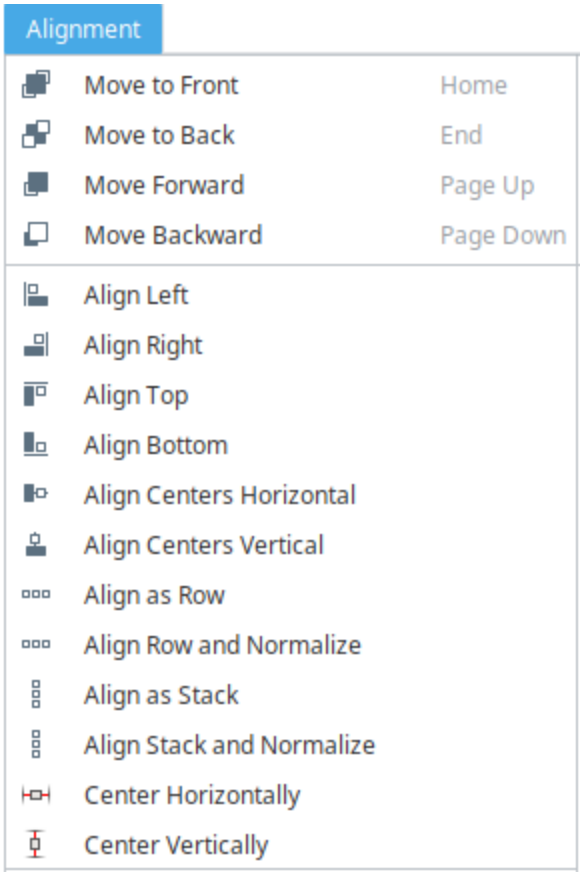
The **Component Menu** offers many of the same selections for the selected component that right clicking on that component would contain.



Function	Description
Group	Only available when multiple components are selected. Grouping will place the currently selected components into a group. Ungroup will remove the grouping. See also: Working with Components
Ungroup	Only available when a group is selected. This option removes the group (and any custom properties that are on the group) and places all items from that group into the object the group was in.
Convert to Container	Only available when a group is selected. Converts the selected group to a container. See also: Container
Lock	Locks or unlocks the selected component's size and position.
Layout	Set layout constraints for the selected component.
Size and Position	Change the size and position of the currently selected component.
Customizers	Allows you to select any of the available customizers for the currently selected component.
Scripting	Brings up the scripting window for the currently selected component.
Security	Opens up the Security Settings Panel, allowing security to be placed on the selected components.
Translations	Brings up the Translatable Terms Panel, showing any translations for the selected component.

Alignment Menu

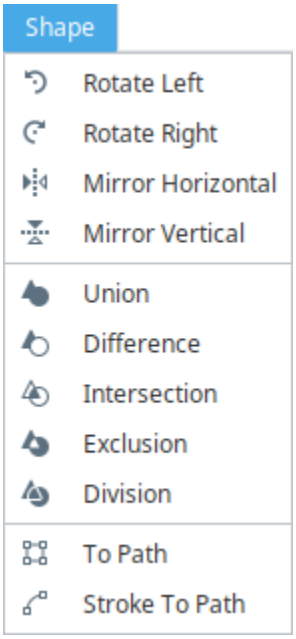
The **Alignment Menu** options allow you to adjust the alignment of components relative to other components.



Function	Description
Move to Front	Move the selected components to the front of the z-order.
Move to Back	Move the selected components to the back of the z-order.
Move Forward	Move the selected components forward in the z-order relative to any overlapping components.
Move Backward	Move the selected components backward in the z-order relative to any overlapping components.
Align Left	Align the left edges of a group of components.
Align Right	Align the right edges of a group of components.
Align Top	Align the top edges of a group of components.
Align Bottom	Align the bottom edges of a group of components.
Align Centers Horizontal	Aligns all of the selected components horizontally on their centers.
Align Centers Vertical	Aligns all of the selected components vertically on their centers.
Align Centers	Aligns all of the selected components either vertically or horizontally on their centers.
Align as Row	Aligns all of the components on their centers as a row, and will add padding between them that you can select. Normalizing them will change the size of all of the components to the first selected component.
Align as Stack	Aligns all of the components on their centers as a stack, and will add padding between them that you can select. Normalizing them will change the size of all of the components to the first selected component.
Center Horizontally	Centers the currently selected components horizontally.
Center Vertically	Centers the currently selected components vertically.

Shape Menu

The **Shape Menu** allows for manipulation of shape or path objects.



Function	Description
Rotate	Rotates the currently selected shape 90 degrees either right or left.
Mirror	Flips the component either vertically or horizontally.
Union	Alters the first shape to be the combination of all selected shapes.
Difference	Alters the first selected shapes by removing the last selected shape from them.
Intersection	Alters the first shape to become a new shape consiting of the area they share.
Exclusion	Alters the first shape to become a new shape consiting of the area they do not share.
Division	Cuts the first shape into multiple shapes along the borders of other shapes.
To Path	Converts a shape to a simple path object.
Stroke To Path	Converts the selected shape into a new shape defined by its stroke.

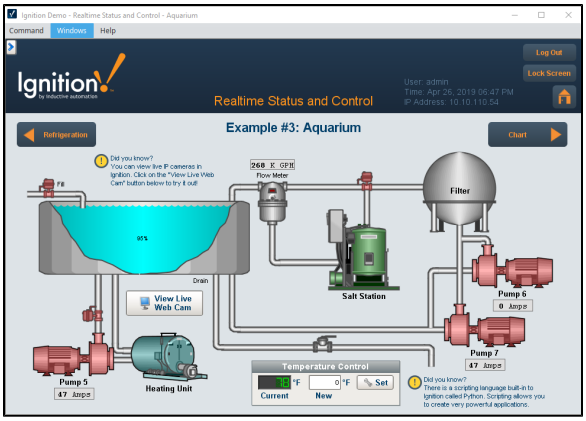
Tools Menu

The Designer comes with many tools that allow you to manage and test various resources within a project. Each of the tools have their own interface and are accessed within the Tools menu on the menu bar of the Designer. For more information, see Designer Tools.

Vision Windows

Windows are the key to your HMI/SCADA application. A window is the basic building block of any Vision project, where each window can contain any number of components that can display Tag values, run scripts, write values to the database, and accept user input. When you publish your project, these windows are loaded into the Vision Client where any number of windows can be opened at one time.

Your windows are brought to life through the property bindings and event handlers on your components. They can be designed to fit any need, from simple screens showing basic information, to complex diagrams outlining an entire plant floor with various controls. Despite their abilities, using windows is relatively simple so that even new users can get started creating windows right away. The possibilities are endless when designing windows for your project.



On this page ...

- Window Anatomy
  - Root Container
  - Window Name and Title
  - Titlebar and Border
- Creating a Window
  - Project Browser
  - Welcome Window
  - File Menu
- Organizing Windows
- Window Right-Click Menu
  - Exporting Window Example
  - Importing Window Example
- Navigation Strategy

Window Anatomy

While there is only one type of window object, windows have various properties that determine how they behave within the client. When these settings are configured in specific ways, they create certain categories or types of windows: **Main Windows** act like a typical HMI screen and take up all available space, **Popup Windows** are often opened by a component in a Main Window and appear to float on top of the Main Window, and **Docked Windows** stick to one side of the screen and are typically always open. These types of windows all provide different functionality to a project which, when combined create the basis for a Vision project that displays relevant information while remaining intuitive and user friendly.

INDUCTIVE UNIVERSITY

Anatomy of a Window

Watch the Video

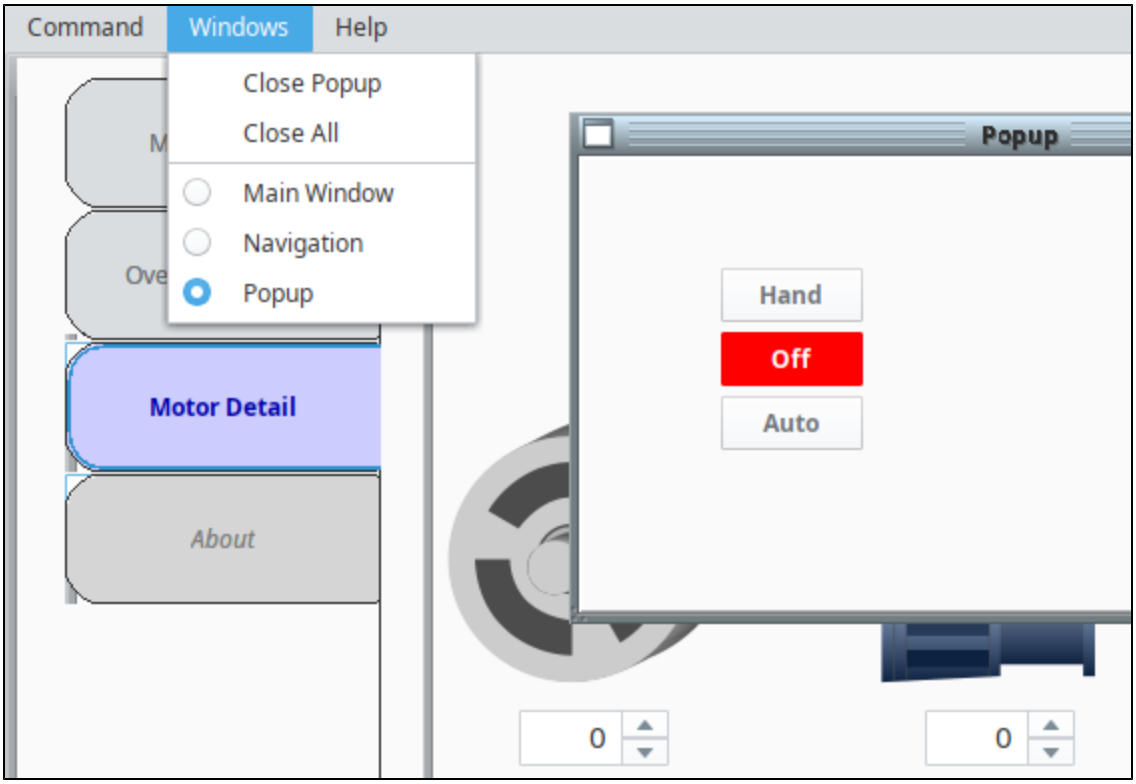
Root Container

Inside a window there is always a **Root Container**. The Root Container is where you place all components in the window. It's a normal Container component except that it cannot be deleted or resized, and is always set to fill the entire window. The root container will be the root of all components that go onto the window.

Window Name and Title

Windows have both a **Name** and a **Title**. The name is used within the Project Browser to differentiate the windows from each other and to form part of the path to the window. Windows can be renamed by right clicking on the window object and selecting rename or by pressing F2. Each window must have a unique path, so windows can have the same name as long as they are not in the same folder.

The Title property is a property within the property editor and works a little differently than the name. By default, Ignition assigns the Title property the same name as the window type that is created (i.e., Main Window, Docked Window, or Popup Window). These window titles are used for the titlebar of a window, but are also used when viewing currently opened windows. In the Client, the Windows menubar command will display a list of all currently opened windows, as well as allow you to switch between which one is in focus. The list of opened windows displays the title of the window, and not the window name or path, so it is also important to have good titles for your window.



Titlebar and Border

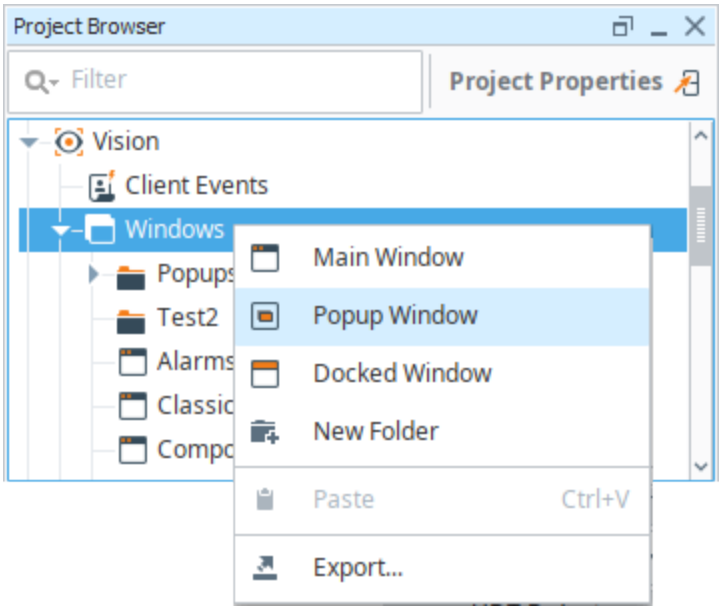
A window can display a **Titlebar** and/or a **Border**. A titlebar allows you to drag a window around the workspace, and contains the window's close, maximize and restore buttons. The border of a window also lets you resize the window when it is floating or docked. Whether or not the titlebar and border are displayed depends on the property values set for your Titlebar and Border properties. A window typically displays both a titlebar and border when it is floating, but only a titlebar when maximized. It is often desirable to remove titlebars and borders on maximized windows.

Creating a Window

Creating windows is easy. You can create a new window by right-clicking in the Project Browser, using the Welcome Window, or using the File menu.

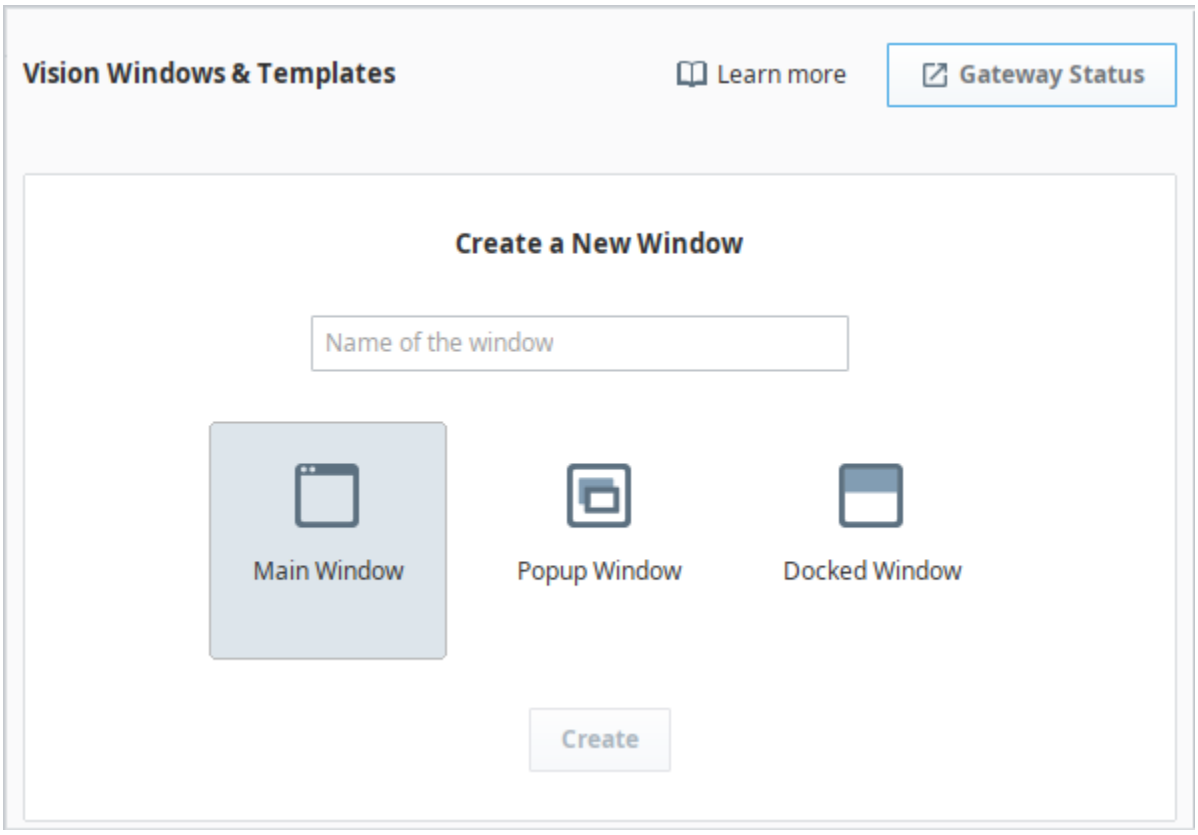
Project Browser

A common method is to right-click within the Windows section of the Project Browser and select one of the window types to create a window. While you can create each type of window, it is important to remember that the only difference is the configuration of the window properties.



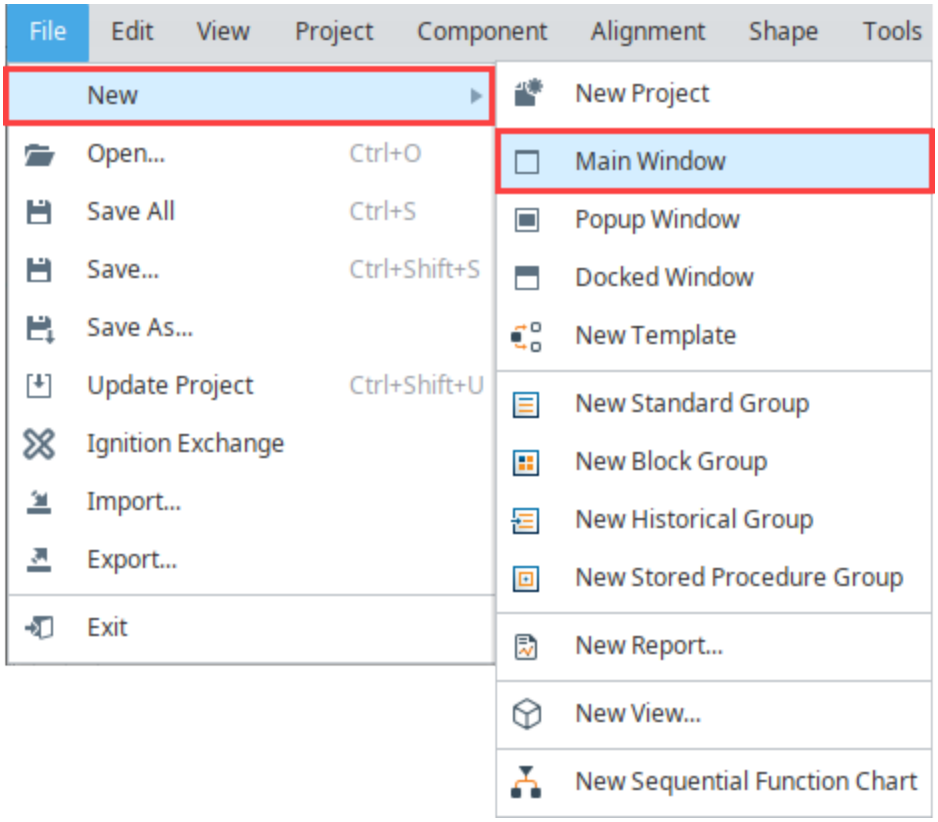
Welcome Window

The Welcome Window that is shown when the project is first opened has a few quick start options. One of these options is the ability to create a new window by selecting the desired window type you want to create.



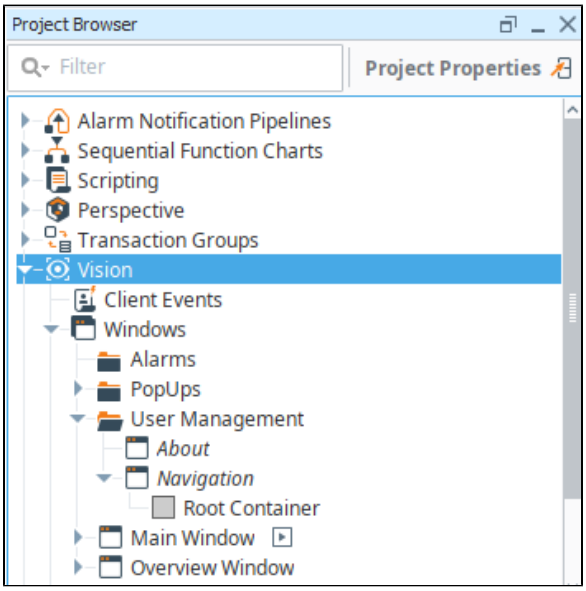
File Menu

In the menubar of the Designer, the **File** menu has a **New** option that allows you to create a new window regardless of where you are in the project.



Organizing Windows

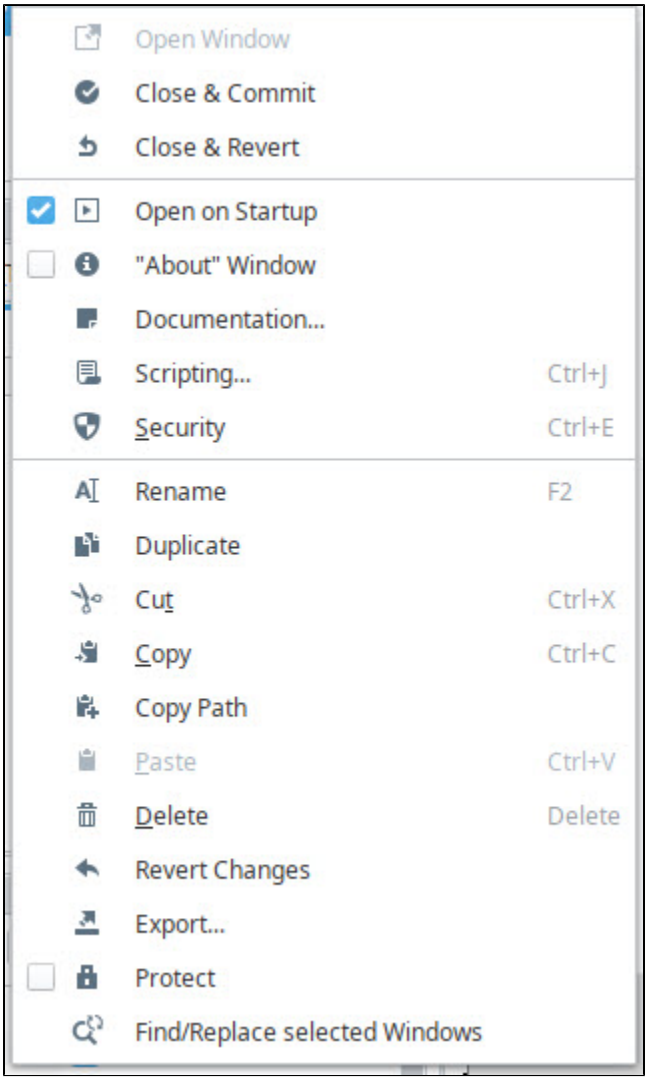
You can create folders to organize your windows. A window's name must be unique among the windows in its folder, but you can have the same window name in multiple folders. The window name and folder path are very important, they are used as references by other windows. You can create as many folders as you want and nest them as deep as you need for your project. To rearrange a window, just click and drag the window where you want to place it.



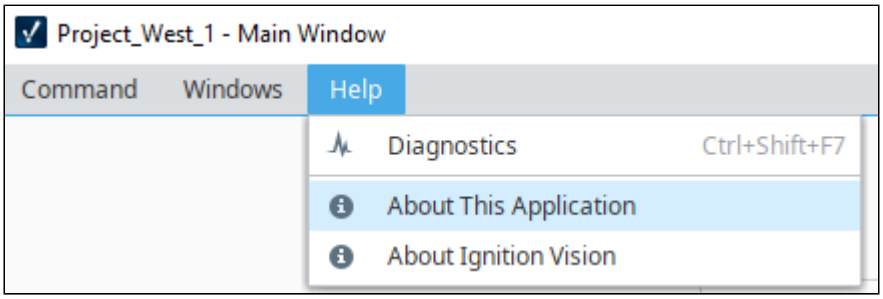
Note: If you have a security requirement to open a different startup window depending who is logged in, you can create a client startup script to open a dynamic set of windows. To learn more, refer to Open Dynamic Windows on Startup.

Window Right-Click Menu

For a full list of properties that can be set on windows, refer to Vision - The Window Object. Windows also have right-click menu of options for additional functionality.



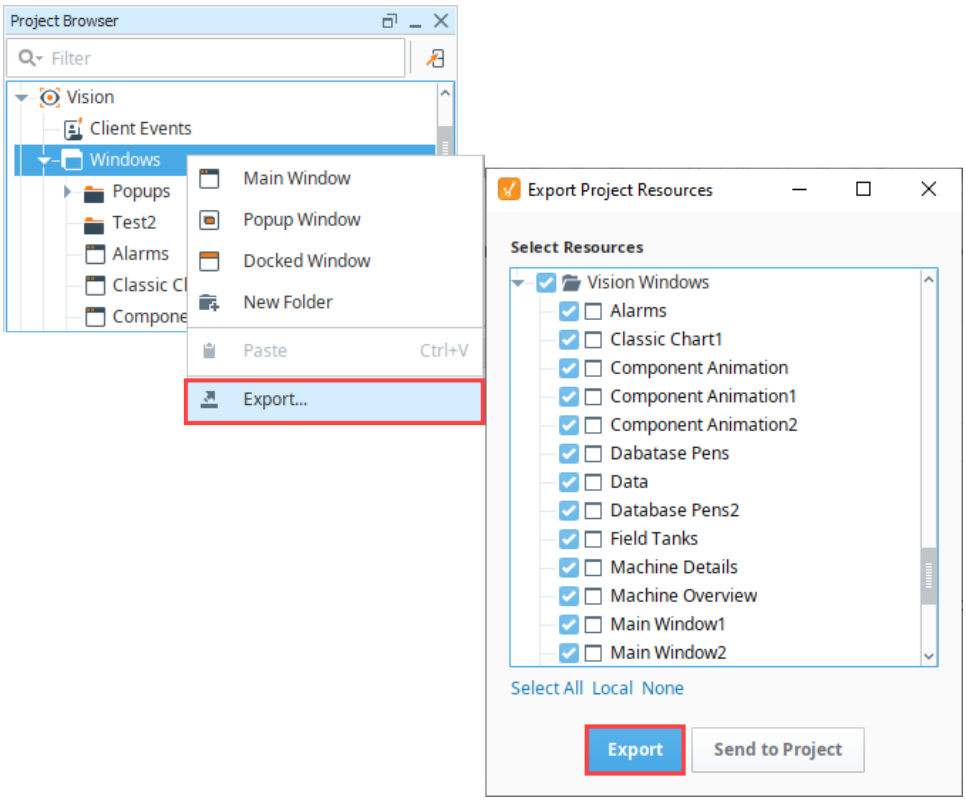
Function	Description
Open Window	Opens the selected window.
Close & Commit	Commits any changes to your workspace and closes the window.
Close & Revert	Reverts any changes that were made since the window was last opened or saved.
Open on Startup	One of the most useful properties is the **Open on Startup** property, which when enabled will automatically open the window when the client first starts up. This makes it easy to open a static set of default windows that everyone can see after logging in to the project. Multiple windows can be set to open on startup, though it is recommended that only a single main window is set to open on start, as opening multiple at once will cause them to be hidden behind one main window. All windows that **Open on Startup** have a little box with a **Right Arrow** icon next to the window name.
About Window	

	An "About" Window relays information to the user that may be important, such as instructions on how to use the project, or information about the projects creator. To specify a window as the About window, right click on the window in the Project Browser. Then click the About Window checkbox. The window will have a Information Bubble icon displayed next to its name. In the client, the window will be displayed when a user selected **Help > About This Application**. 
Documentation	Windows can also have notes attached to them. The notes provide a way for a windows designer to provide some documentation on what the window is doing and how the various components interact with one another. Any windows that have notes will have a small **Document** icon next to the window name. This feature was changed in Ignition version **8.1.19**: Prior to 8.1.19, this field was called Notes.
Scripting	The Scripting option takes you to the Component Scripting for that window. For more information, refer to Component Events and Script Builders in Vision.
Security	The Security options displays Security Settings for role-base security. For more information, refer to Security in Vision.
Rename	To rename a window, select this option then enter a new name.
Duplicate	Duplicates the selected window.
Cut	Cuts the selected window onto the clipboard.
Copy	Copies the selected window onto the clipboard.
Copy Path	Copies the path of the selected window into the clipboard.
Paste	Pastes the content in the clipboard into the selected context.
Delete	Deletes the current selection.
Protect	Locks the individual project resource from inside the Designer.
Export	Exports the window as a project resource file which can then be imported into other projects. See the following sections for examples of Export and Import.

Exporting Window Example

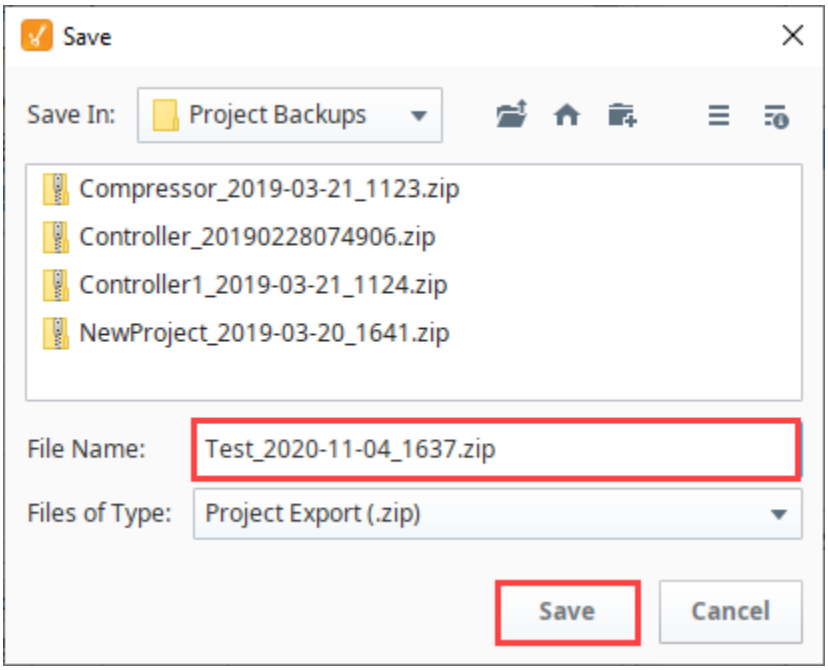
In the Designer, you can export and import windows from one project to another project using external files or sending it directly to a project on the same gateway.

1. You can export windows in two ways.
    a. To export multiple windows, right click on the folder of windows. and select either the **Export** option or **Send to Project** option. The window export works similarly to the project export, the difference being that it automatically highlights only that window to export from the list of project resources.



b. To export one window, right click on an individual window and click **Export**, then choose the **Export** or **Send to Project** option.

2. If you choose the **Export** option, the **Save** window is displayed. You can save the windows with the existing project name (not recommended if you are only exporting part of a project), or type a new name in the File Name field.

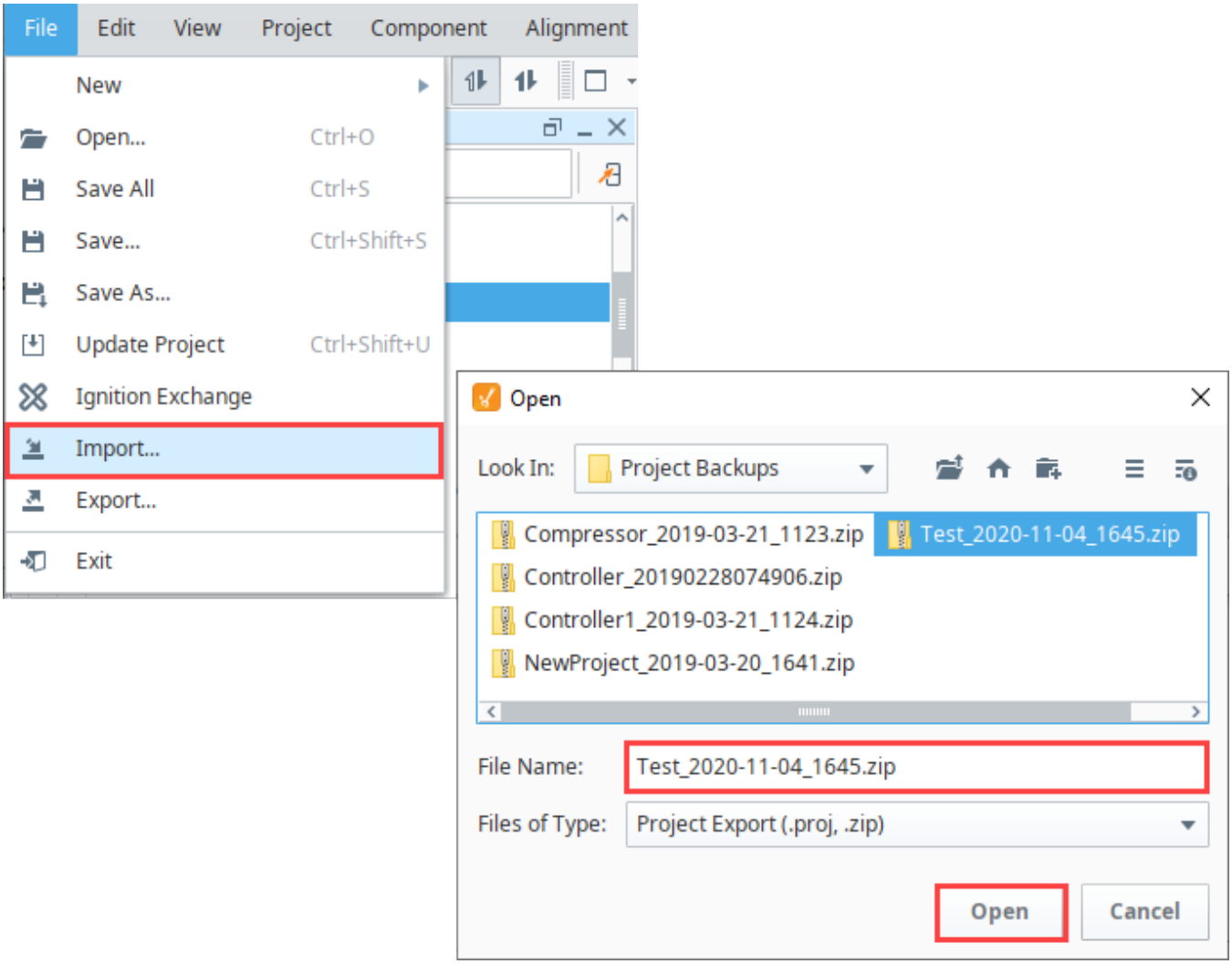


3. Click **Save** to save the windows as a project export file.

Importing Window Example

Importing the individual windows can be done by right clicking **File** from the top menubar and selecting **Import**.

Browse to the folder that contains the `.zip` file you want to import, and click **Open**.



Navigation Strategy

Setting up a navigation strategy allows you to navigate between different windows in the runtime Client. While we have a few examples of the most common navigation strategies, it is certainly not an exhaustive list as most users tend to combine multiple strategies to create a project that fits their needs.

A typical navigation strategy for a Vision project is as follows:

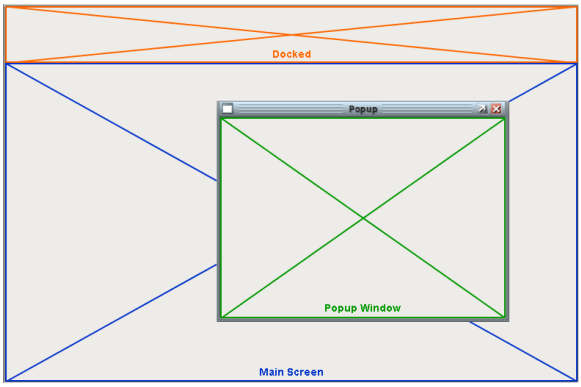
- Have a **Docked** window or two, usually docked North and/or West.
- Have a single **Main** window visible at a time.
- Use swap navigation to swap between the **Main** windows. This ensures that only one main window is open at a time.
- Use standard open navigation to open various **Popup** windows as necessary.

This style of project is so common, that the default operation of the Tab Strip component expects it. When it is in its default automatic operation, it expects that each tab represents a main window, and will automatically swap from the current screen to the desired screen. Additionally, the `[System]/Client/User/CurrentWindow` Tag is calculated based on this strategy: its value is the name of the current maximized window. This navigation strategy is used in the Ignition Online Demonstration that you can download from our website.

In This Section ...

Window Types

There are three Vision window types: **Main** windows, **Popup** windows, and **Docked** windows. You can create windows from the **File > New** menu or by right clicking on the Windows object in the Project Browser. By changing a window's properties, you can transform any window into various configurations, with each behaving differently based on those settings.



On this page ...

- Main Windows
- Popup Windows
- Docked Windows
  - Docking Settings

INDUCTIVE UNIVERSITY

Window Types

Watch the Video

It is important to understand that just because a certain type of window was created does not mean that it must always be that type of window. A windows type is determined by its settings, so changing its settings to match a different window type will change that window to a new type.

Main Windows

A **Main** window is one that is set to start maximized, and has its Border and Titlebar display policies set to 'When Not Maximized' or 'Never.' This will make the window take up all available space (minus space used by any "docked" windows). This makes the window act much like a typical "HMI screen." There can be many main windows in a project, but only one should be open at any time since they would all overlap.

Popup Windows

A popup window is a window whose Dock Position is set to Floating and is not maximized. Its Border and Titlebar display policies are typically set to 'When Not Maximized' or 'Always,' so that they can be manipulated by the end-user. These windows are often opened by components in a main window, and are meant to be on top of the screen. To this end, they should have their **Layer** property set to a number higher than zero so they don't get lost behind the main window. Popups can be set to open at a specific position on the screen using window's **Location** property. Popup windows can also be parameterized so they can be made once and used for multiple similar applications, dynamically changing the content on the screen based on a parameter that gets passed in.

Docked Windows

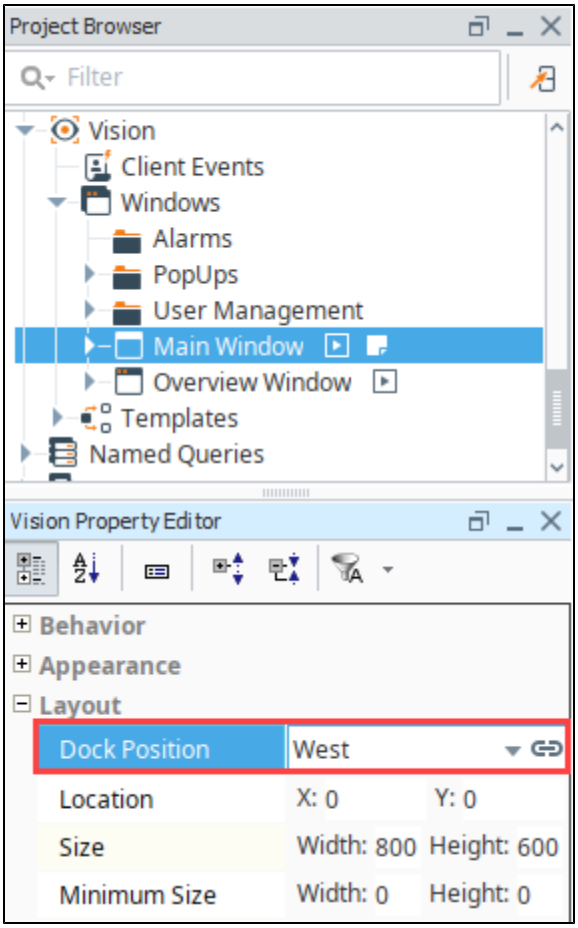
A **Docked** window is one whose Dock Position is set to anything but Floating. Docked windows are locked to the edges of the Client and fill all the space on that edge (i.e., West Docked fills the left side of the Client). It will also typically have its Border and Titlebar display policies set to Never. This makes the "docked" window appear to be joined seamlessly with the current main window. These screens are usually tall and skinny or short and wide, depending on the side they're docked to. The purpose of a docked window is to make some information always available; typically navigation controls and overall status information. Using docked windows can help eliminate repetitive design elements from being copied to each screen, making maintenance easier.

Setting which side the window is docked on is done through the window's **Dock Position** property.

INDUCTIVE UNIVERSITY

Docked Windows - Order Precedence

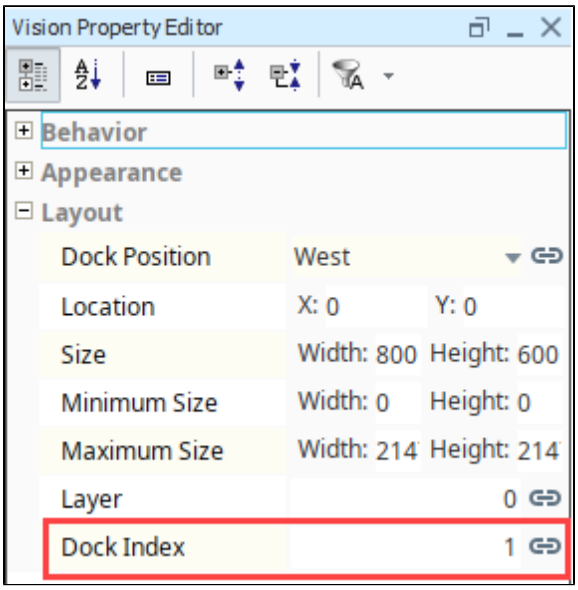
Watch the Video



Docking Settings

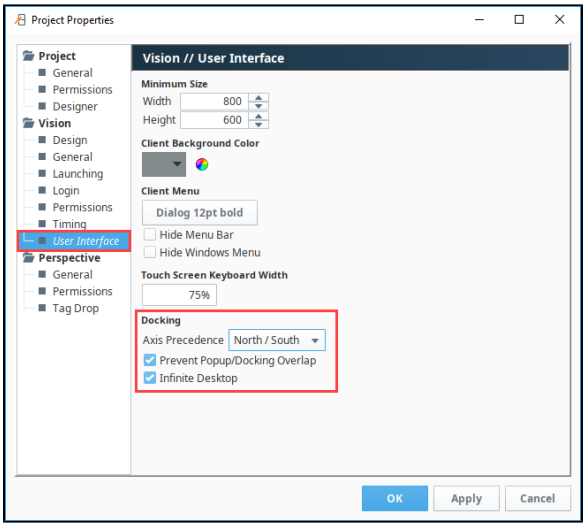
Having multiple docked windows means you need to decide how you want them to appear in relation to each other. For that, Ignition has several settings.

The Dock Index is a property on the window that determines the order of docked windows if multiple windows are docked to the same side. The window with the lowest Dock Index will appear closest to the edge on that side, whereas the highest Dock Index will appear closest to the middle of the client.

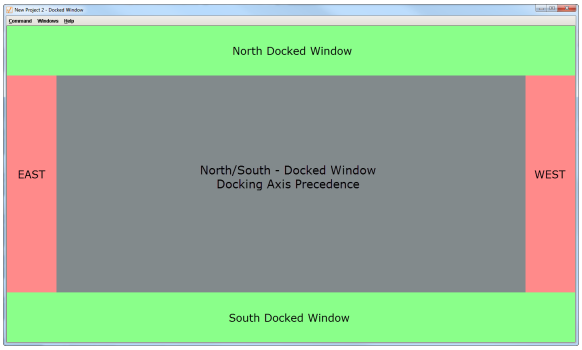


The remaining three settings are located in the Project Properties > Vision > User Interface section:

- **Axis Precedence** - Project wide property that determines which sides get to extend to the edge of the window, North and South or East and West.
- **Prevent Popup/Docking Overlap** - When set to true, then floating (popup) windows will not overlap with docked windows.
- **Infinite Desktop** - When set to true, then the desktop area will be expanded if windows are dragged out of frame.



North/South Axis Preference has the top and bottom docked windows reaching the left and right edges of the screen, while the docked windows on the left and right sides fill the remaining space.



East/West Axis Preference has the left and right docked windows reaching the top and bottom edges of the screen, while the docked windows on the top and bottom fill the remaining space.

Related Topics ...

- Vision Project Properties

Popup Windows

A popup window is typically a window that "floats" on top of the main window. It can be resized and moved around by the user, all while the main window is still open in the background. Popup windows are great for displaying additional information about a selected item on the screen, for example a details screen about one particular component. Popup windows are often opened by components in a main window and are meant to be on top of the screen. They are used to view setpoints and zoom into a specific area.

A great thing about popup windows is how they can be parameterized and be reused. One popup window design can be reused for many components as long as the proper information is passed through.

On this page ...

- Creating a Popup Window
- Opening a Popup Window
- Popup Window Properties
  - Layer
  - Location
- Parameterized Popup Windows
- Multiple Instances of a Popup Window

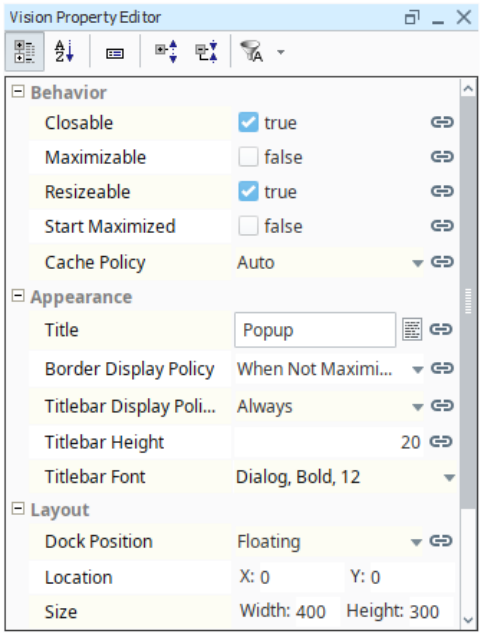
INDUCTIVE UNIVERSITY

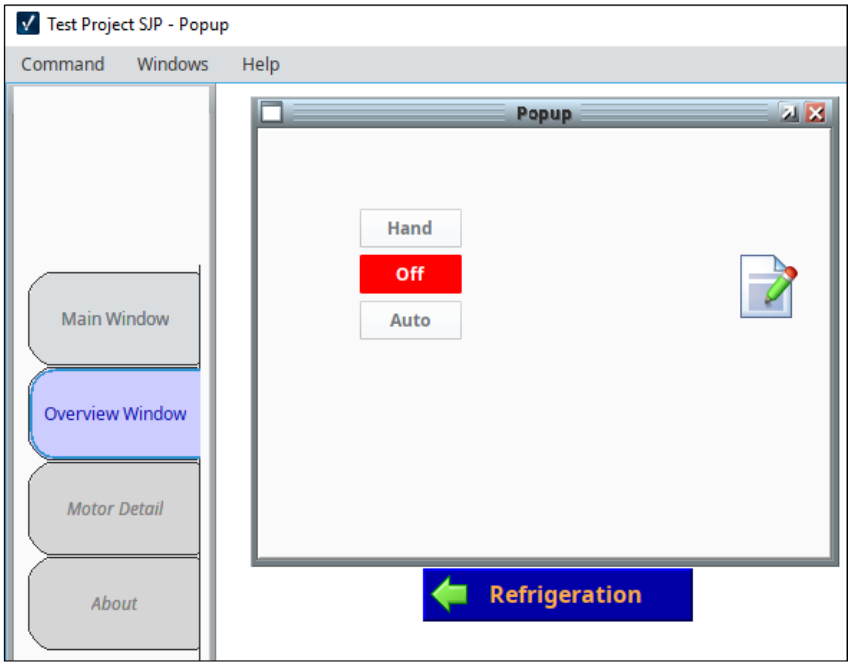
Open Popup Window

Watch the Video

Creating a Popup Window

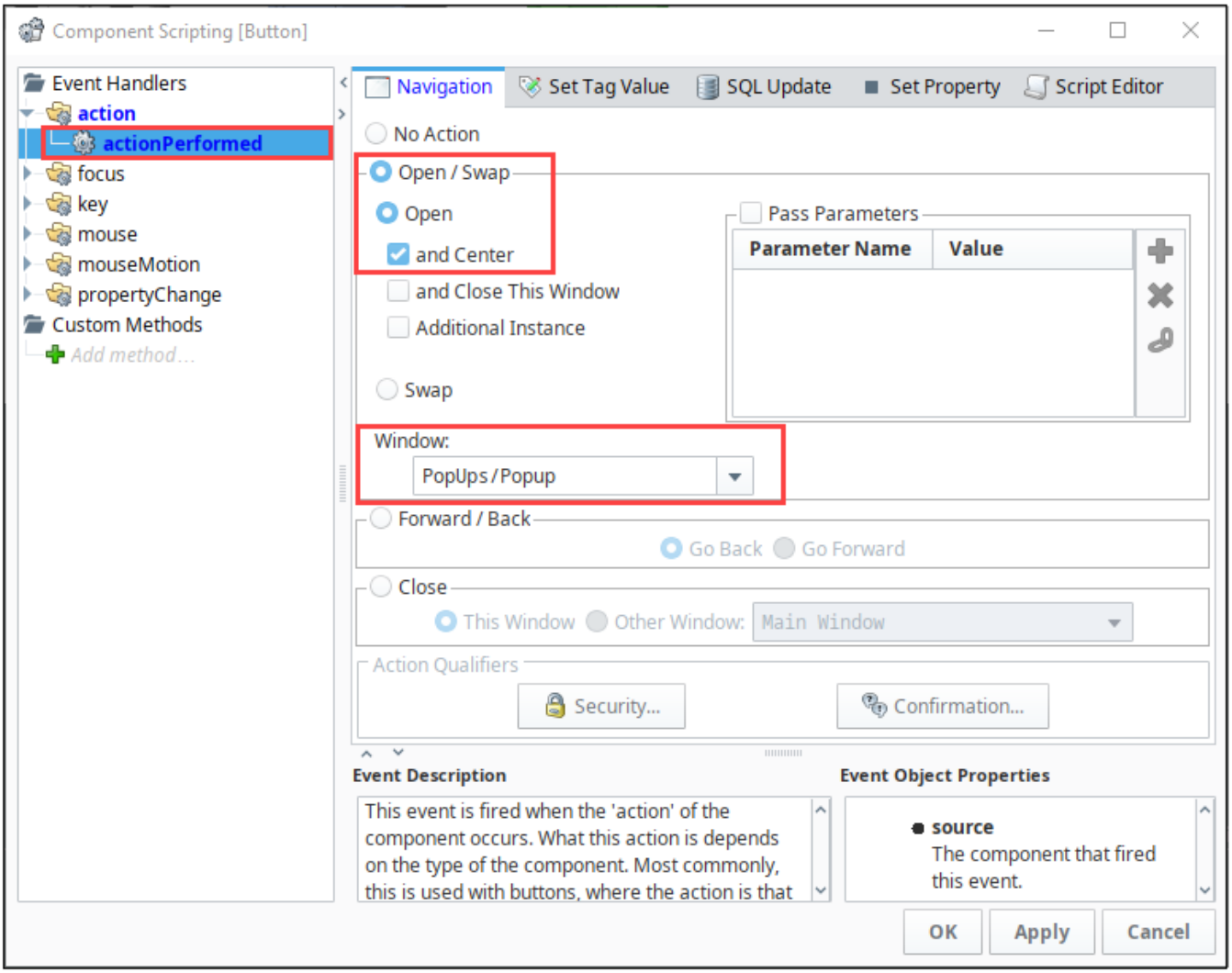
Like main windows and docked windows, popup windows are simply windows that have specific settings. In particular, popup windows are floating windows that are not set to start maximized. When adding a new window in the Designer, selecting the popup window option creates a window with these presets. Once you have your popup window created, you can make it as big or small as you want. You can also set properties in the Property Editor to make it closeable, make it resizable, give it a new title, and display the title bar and border in the window.





Opening a Popup Window

In any window, you can add a script to any component to open your popup. This is easiest to do from a component like a button on the main window using the Navigation Script Builder. Simply select the Open action and the window that you want to open. Clicking on the button will then open the popup window that was selected. Alternately, you can use one of the many scripting functions that open a window.

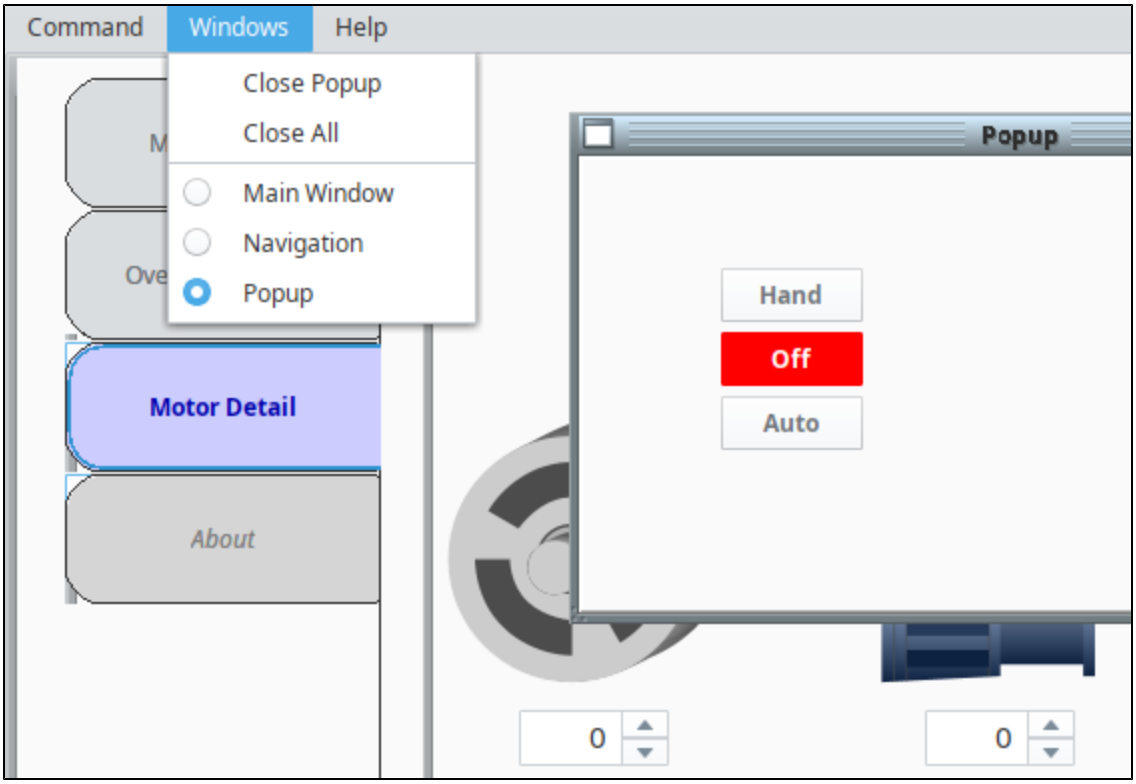


Popup Window Properties

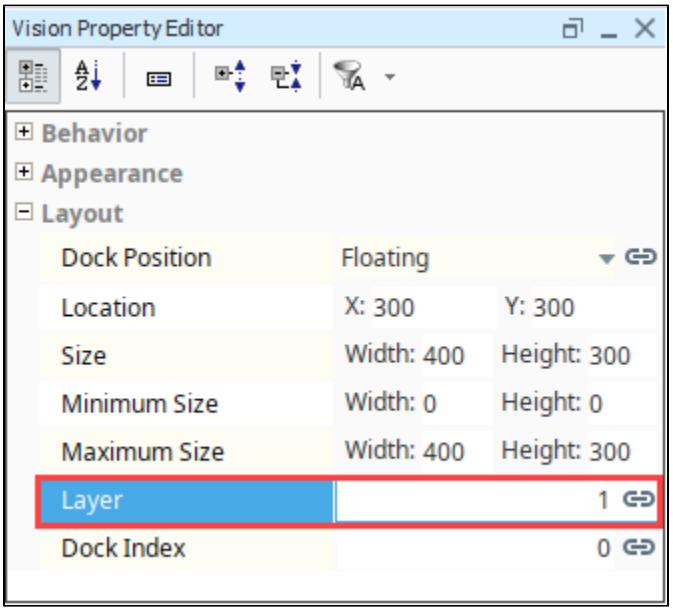
There are a few properties of the Window that are useful to popups, such as Layer and Location.

Layer

The Layer property of windows controls the z-order of the windows. Windows with a higher Layer will always be on top of windows with a lower Layer, regardless of which window is in focus. This is useful for keeping popup windows at the forefront. By default, all windows have a Layer of 0, but we can change this so that popups always remain on top. If popups have a layer that is the same as the main window, clicking on your main window makes it look like the popup window disappears, but it's actually behind your main window. The **Windows Menu** will show you all the open windows in your Client, with the popup still being open.

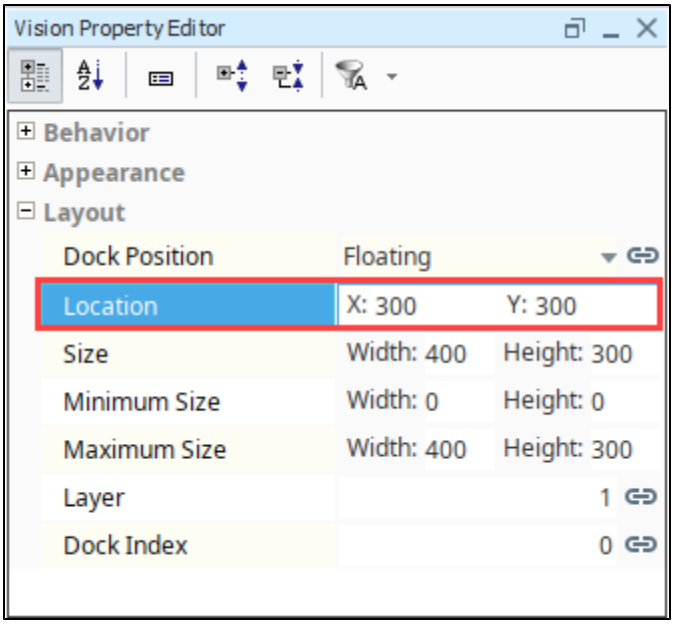


The Layer property is located on the window object itself, in the Property Editor. Simply set it to a higher value so that the popup is always on top.



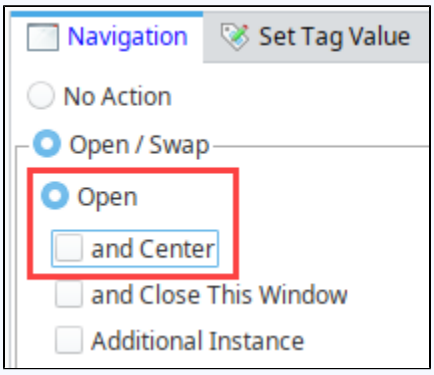
Location

Popup windows can also be given a specific location to open up at, when the not being automatically centered by the script. In the Vision Property Editor, go to **Layout > Location** and provide a specific **X** and **Y** position (in pixels).



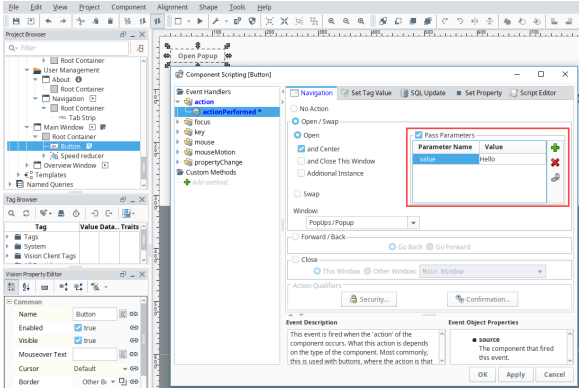
Note:

When opening a popup window to a specific location, ensure the **Open** and **Center** option is unchecked so that it doesn't override the location coordinates.



Parameterized Popup Windows

A parameterized popup window lets you pass information from one window to another window. You can make a single popup window, change what it does and what it points to from a parameter(s) that gets passed into the receiving window using Custom properties. Parameters can range from simple integers and strings, to properties on the window that is opening the popup, and even entire UDT custom properties.



INDUCTIVE UNIVERSITY

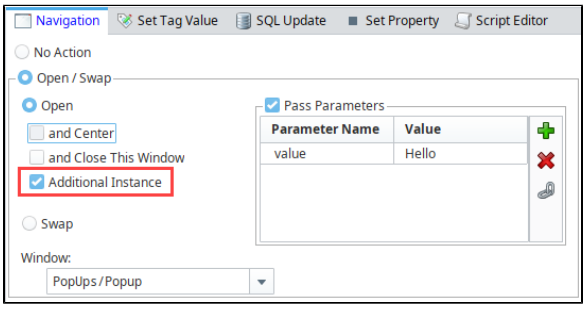
Parameterized Popup Window

Watch the Video

Multiple Instances of a Popup Window

By default, the client only opens a single instance of a popup window, but you can change this behavior. For example, suppose you have four different Tanks passing all the same parameters with the only difference being the individual Tank number. In order to see all four instances of your tanks, you need to configure component scripting to display Additional Instances. This is done by selecting the **Additional Instance** option when setting up the navigation scripting action.

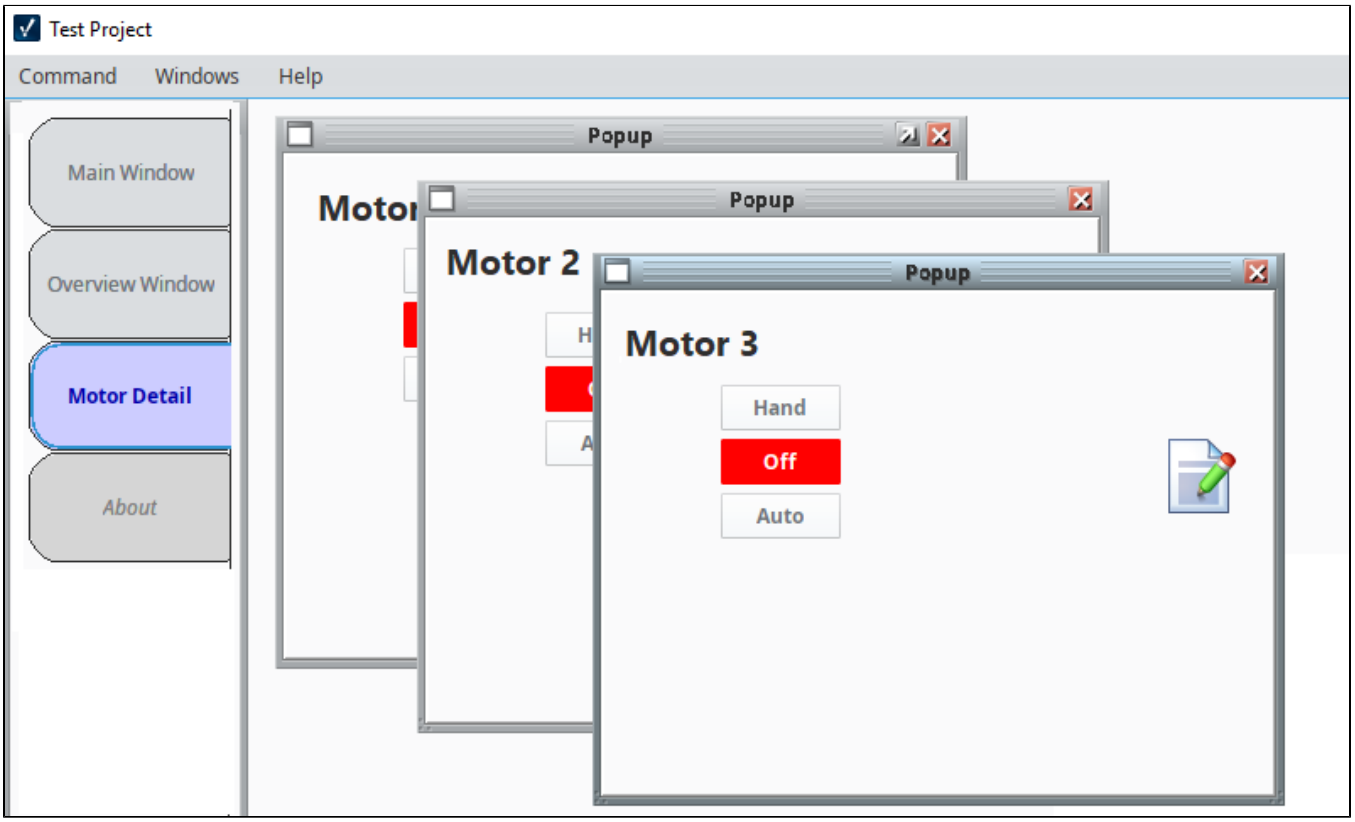
Alternately, the system.nav.openWindowInstance can be used within a more complex script instead.



INDUCTIVE UNIVERSITY

Popup Window - Multiple Instances

Watch the Video



In This Section ...

Parameterized Popup Windows

A parameterized popup window lets you pass parameters from one window into a popup window, where the receiving popup window could then use that parameter to display relevant information. This also allows you to maintain a single window that can be used to display similar information. For example, suppose you have two compressors: Compressor 1 and Compressor 2. Imagine clicking on one of the compressors on the main window and a popup displays the diagnostic information about that specific compressor. Instead of creating a popup window for each compressor, you can create a single popup and use indirection with the passed parameter to display a different compressor's information depending on which was selected.

On this page ...

- Passing Parameters to a Popup Window
  - Setting up the Popup Window
  - Setting up the Main Window
- Passing a UDT to a Popup

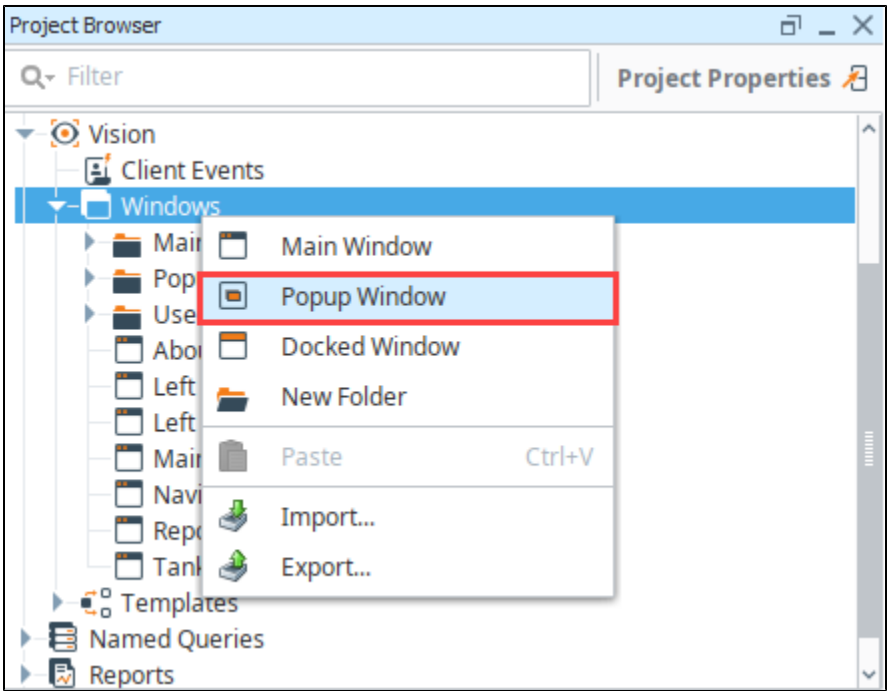
Passing Parameters to a Popup Window

To pass parameters from one window to a popup window, the receiving popup window must have custom properties that receive the passed parameters. When the event on the parent window is called, the parameters are passed to the receiving Popup Window's custom properties on its root container. The component's properties on the receiving window can use the root container's custom properties to address their bindings.

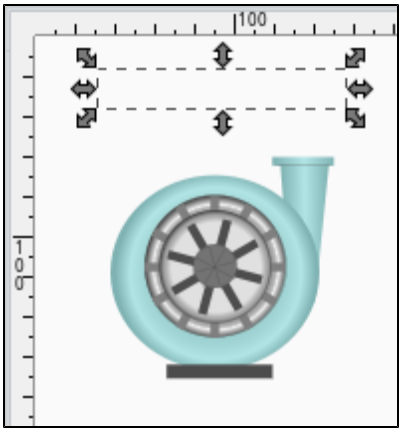
The following examples explain how to set up a popup window and main window to pass compressor numbers to the popup window in order to display relevant information about each compressor.

Setting up the Popup Window

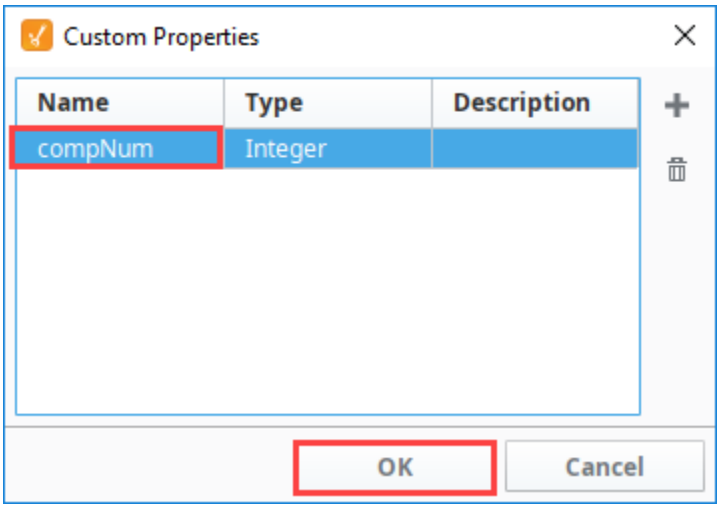
1. Right click on a folder in the Project Browser and select Popup Window to create a new popup.



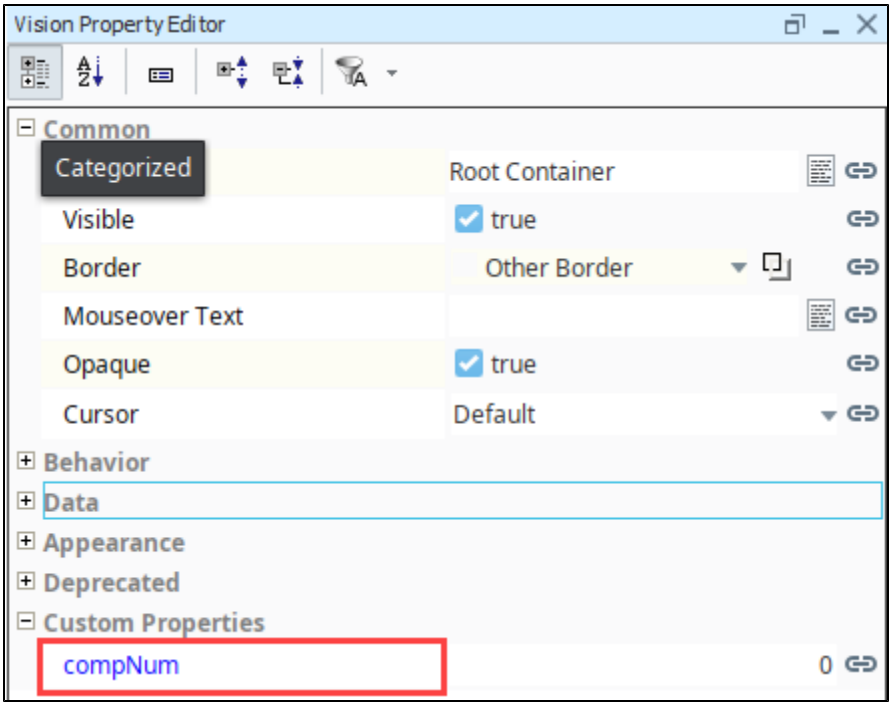
2. Drag a **Compressor** image from Symbol Factory.
3. Drag a **Label** component from the component palette to your window.



4. Create a custom property on your popup window's root container that will receive the passed parameters. Right click on your window and select **Customizers > Custom Properties**. The Custom Properties window is displayed.
5. Click the **Add** ✚ icon to add a property.
6. Specify a **Name** for the Custom Property, such as **compNum,** and click **OK**.

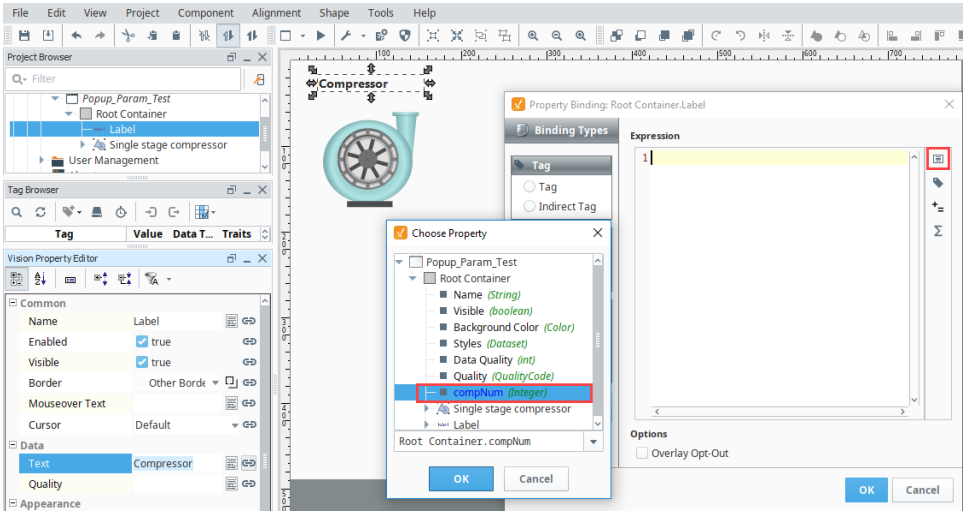


The custom property is created and displayed in blue at the bottom of the Property Editor.



Caution: Do not bind these custom properties to anything, leave them unbound so you can pass values into them without any other values to override them.

7. Let's use an expression on the Label to show what compressor number we are on.
    a. Select the **Label** and click the binding icon for the **Text** property.
    b. Select **Expression** for the binding type.
    c. Click the **Insert Property Value** icon in the Expression window and choose the **compNum** custom property on the root container for the compressor popup window as shown in the image below. Click **OK**.

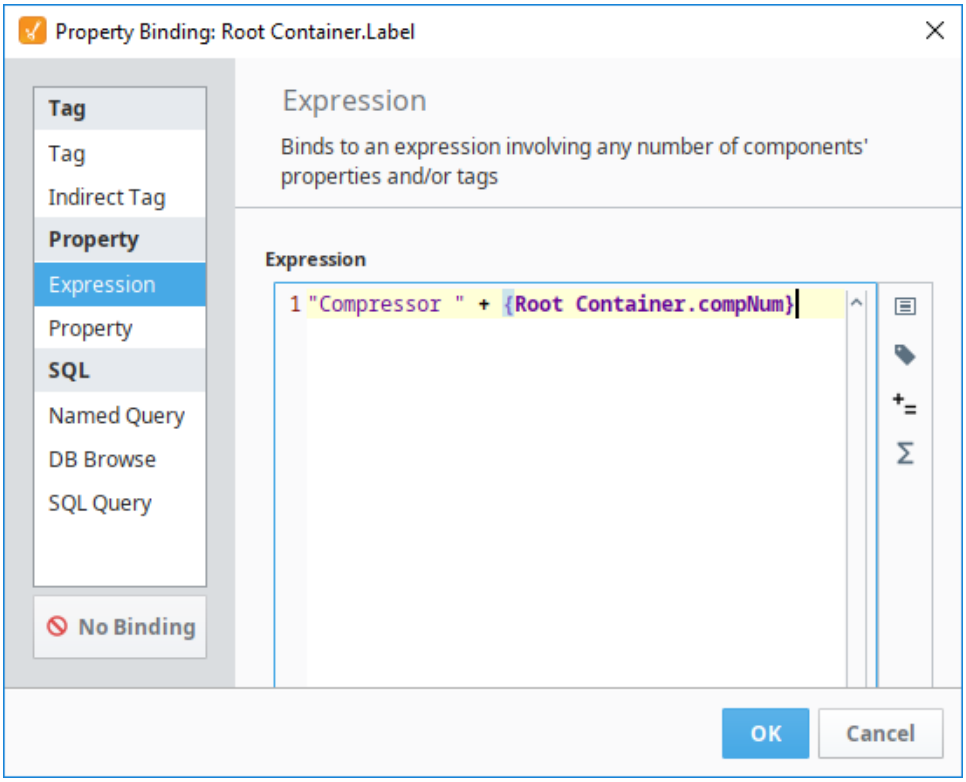


8. Now, let's update the Expression using the script below to show the word **"Compressor"** before the number in the label. Update the Expression in the Property Binding field as follows:

Script to change the compressor number

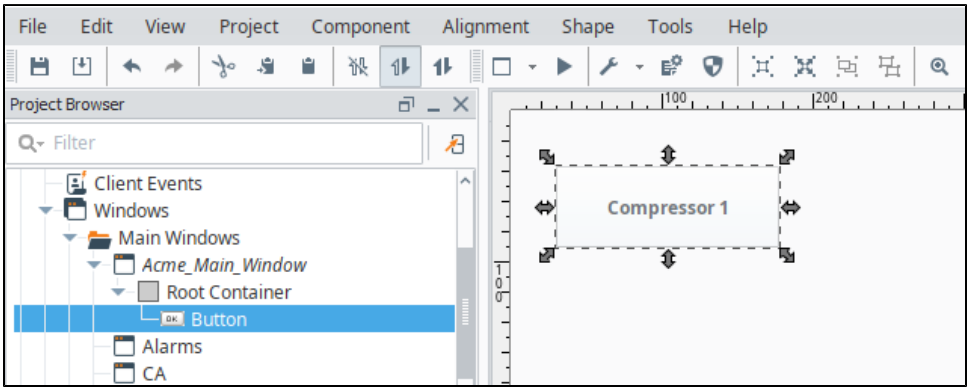
```
"Compressor " + {Root Container.compNum}
```

9. Click **OK** to save the property binding.



Setting up the Main Window

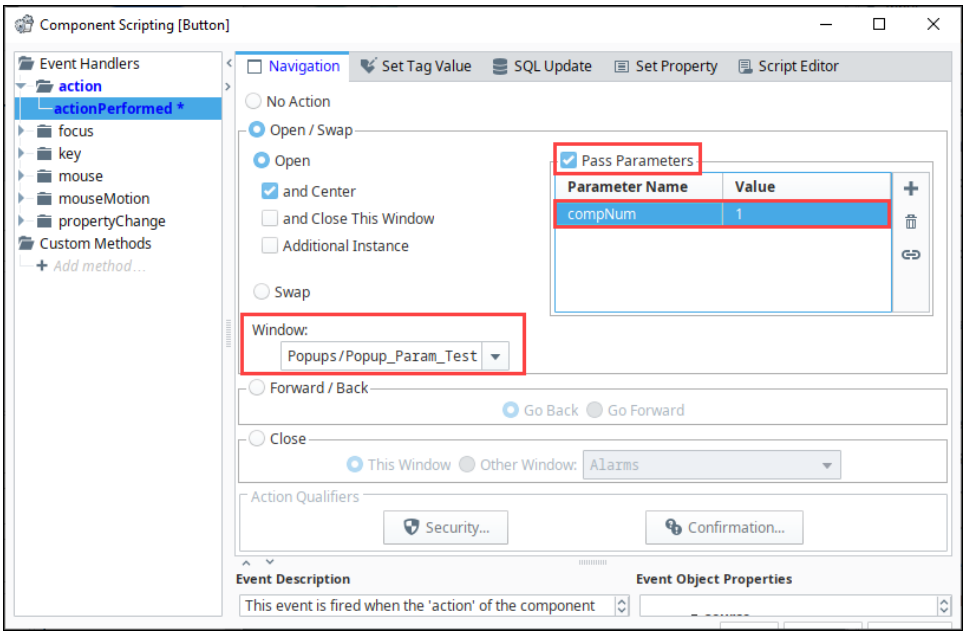
1. In a Main Window (parent window) drag a **Button** from the component palette to your window. Type **"Compressor 1"** into the text property.



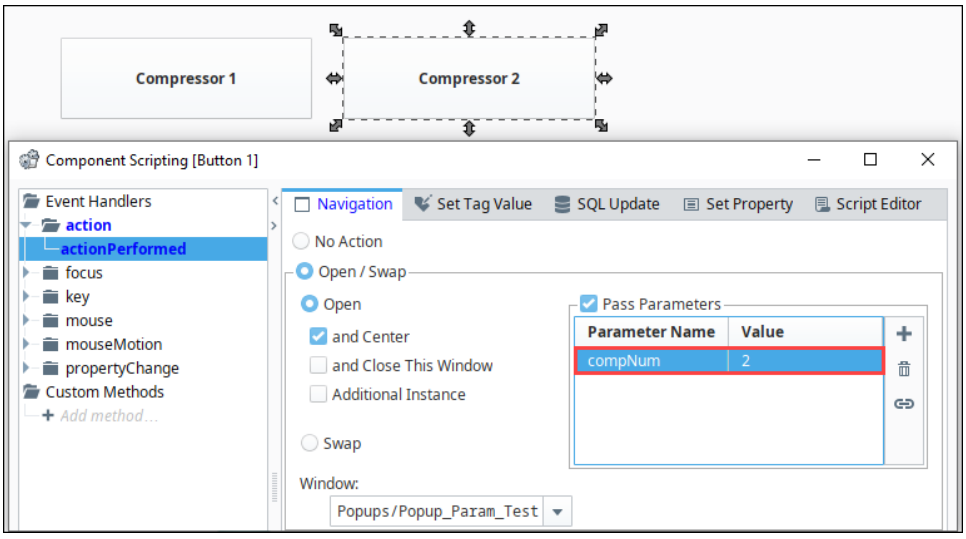
2. Let's add a script to the button which opens the popup that we created earlier. We can then pass in a value to designate that this button should be opening Compressor 1.
    a. Right-click on the **Compressor 1** button and select **Scripting**.
    b. Under Event Handlers, select **actionPerformed**.
    c. Click the **Open / Swap** radio button.
    d. Under Window, use the dropdown list to select the path to your Popup Window (i.e.,Popup_Param_Test).
    e. Check the **Pass Parameters** check box, and click the **Add** ✚ icon to add a parameter.
    f. Click the new row under **Parameter Name** and a dropdown list will appear. Select the custom property **compNum**.

> **Note:** Ignition will automatically check the Root Container of the window selected in the **Window** dropdown. If you do not see the **compNum** parameter, it may have been created on wrong component, so check the Root Container of the Compressor Popup window.

    g. Enter **"1"** in the **Value** column because the button will be for Compressor 1.
    h. Click **OK** to save the script.

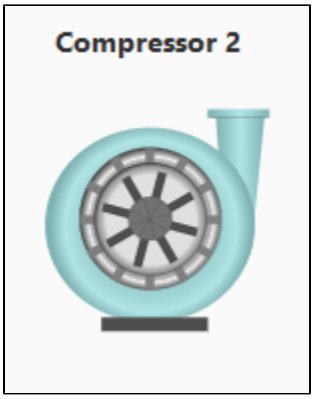


3. Now, create a second compressor button. A quick way to do this is duplicate (**Ctrl D**) the Compressor 1 button so it inherits the script. Update the Text property to **Compressor 2**.
4. Right-click on the new **Button** component and select **Scripting**. Update the parameter Value being passed in from a **1** to a **2**.
5. Click **OK** to save the script.



6. Test it out by putting the Designer in **Preview Mode**. Click one of the Compressor buttons, then navigate back and click the other Compressor button. While these buttons are opening the same popup, they display different information because they are using the parameter that we passed in for indirection. In this example, we just used a label, but the parameters can be used in things like indirect Tag

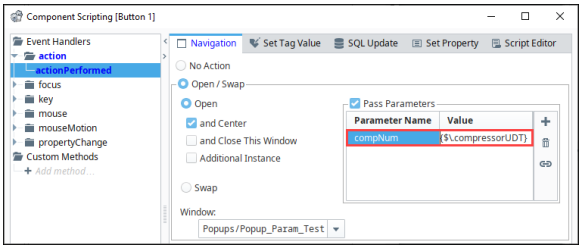
bindings or scripts to pull in various Tag bindings.

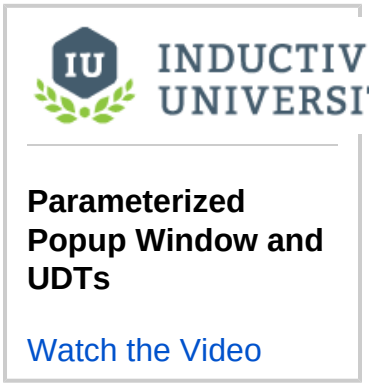


Passing a UDT to a Popup

In addition to the basic types, parameters can be a complex UDT type. This works much the same as passing in basic values, where the popup window has a custom property on the root container, and a parameter is passed in when opening the window.

The difference is that the custom property on the popup window needs to be a UDT that has been previously defined, and the value being passed in when opening the window needs to be an entire UDT instance. This gives the popup access to every Tag within the UDT, which can be useful when making popups that show all the details of a certain area which has a UDT.





Navigation Strategies in Vision

Navigation Strategy

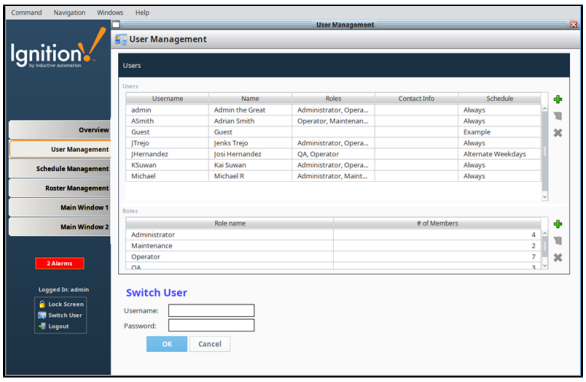
Setting up a navigation strategy allows you to navigate between different windows in the runtime Client. Ignition provides several different types of runtime navigational strategies you can choose from when designing your project including several Vision project templates to help you get started. Before selecting the proper navigation strategy or template for your project, there are several things to consider. These considerations will help you determine the best navigation strategy to use for your project and your users. Once you address these considerations, then you can choose the best navigation strategy from the types below. This will also help you decide if you want to use a Vision Project Template to quickly for your project.

- Does your project have a lot of windows?
- How complex is your project structure?
- Is your project structure organized?
- What types of things are you doing?
- Do you want to use navigation windows or fill the screen?

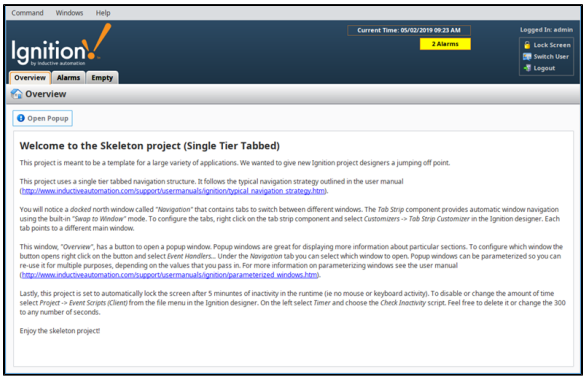
Types of Navigation Strategies

To help you select the right strategy that fits your project structure, here is a brief description of each navigation strategy that Ignition provides. Keep in mind your project structure, size, organization, and types of things you are doing while you are reviewing these strategies so you can select the best runtime strategy for your project.

- **Tab Strip Navigation** is a simple strategy used for small structures regardless of how organized your project is. It lends itself perfectly to only having a few windows and showing all of them on a navigation window. Having too many tabs does not work well with the Tab Strip because of size limitations. You want your users too see all the navigation tabs immediately on the first screen. The Tab Strip works by clicking to swap one main window for another.



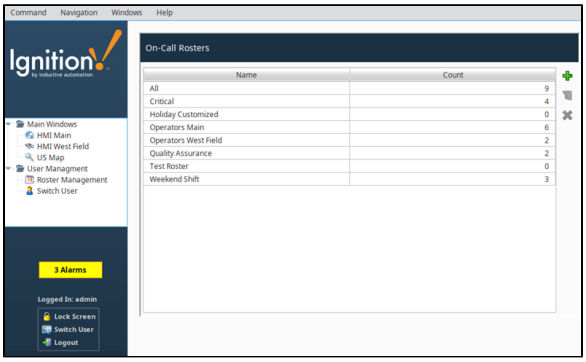
- **Two Tier Navigation** is similar to the Tab Strip, but is good for small and regular size project structures where windows are grouped. It contains a second level of tabs allowing you to navigate around various areas of your project. This strategy has a docked window that contains tabs that are always open to do navigation, and the main window which fills the rest of the space.



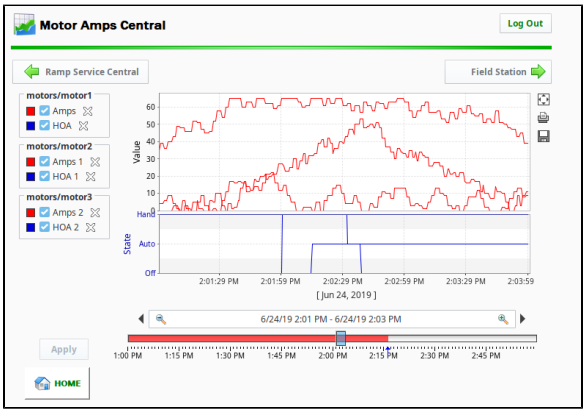
- **Tree View Navigation** is excellent for large project structures. You can view the entire project structure at a glance allowing you to navigate to any structure within the multi-tier Tree View component.

On this page ...

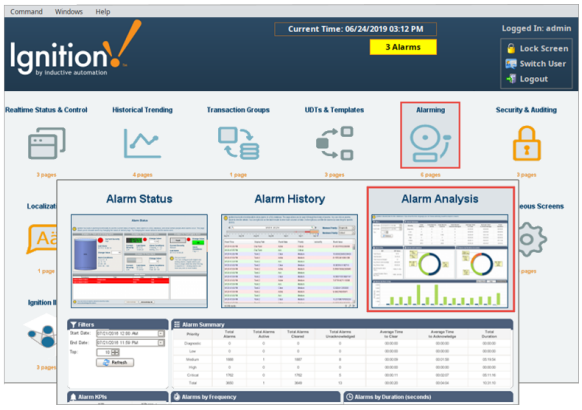
- Navigation Strategy
  - Types of Navigation Strategies
- Navigation Operations - Swapping vs. Opening
  - Opening
  - Swapping
- Common Navigation Mistakes
  - Multiple 'Main' Windows
  - Swapping a Main Window with a Docked Window



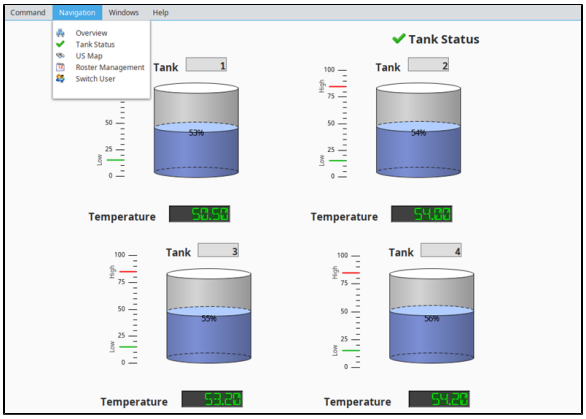
- **Back and Forward Buttons** are perfect if you have a small process with ordered steps. It is one big main window that has Back and Forward buttons to step through each process step or operation one right after the other.



- **Drill Down Navigation** is ideal if you have different geographical locations, whether it's in a local facility or facilities sprinkled around the world. The project opens with an overview that has areas that correspond to specific locations/areas in your facility. With the Drill Down strategy, you can select a specific area representing the facility, and the client swaps windows to display details pertaining to that specific area.



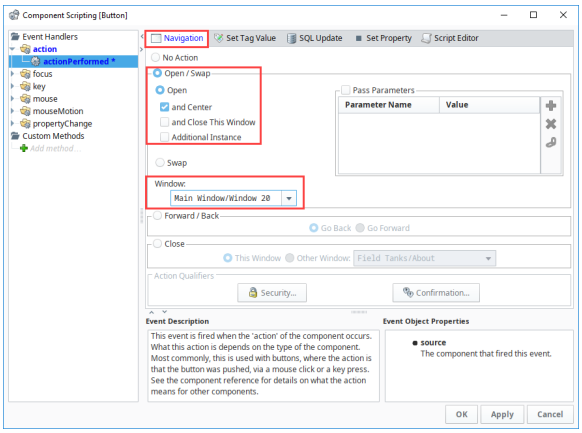
- **Menubar** is ideal for maximizing the usable screen space, while still having the ability to navigate to any window at any time by selecting from a list of windows.



- **Retargeting** enables navigation between multi-project operations: a simple script can push the user between many different projects, even on different Gateways.

Navigation Operations - Swapping vs. Opening

Any time you open a window, you have to use one of the two navigation operations: **Swapping** or **Opening**. These operations can be performed on any type of window, but are usually reserved for specific cases. Navigating between different windows typically involves some sort of scripting, but the Navigation Script Builder makes this a simple task. Otherwise, you can use the specific scripting functions to completely customize navigation.



Opening

Opening and by extension closing are the basic window navigation options. Opening a window opens the window at the same size it was in the Designer, unless the Start Maximized property is **true** or the Dock Position is *not* **Floating**. This is typically reserved for opening popup windows. They have a scripting function that can open and a function that can close.

- system.nav.openWindow
- system.nav.closeWindow

Swapping

When Ignition swaps a window, it closes the current main window and then opens another window in its place. This is typically reserved for moving between main windows, as it performs the close action automatically. There are two different scripting functions that can be used to swap windows, depending on what needs to be done.

- system.nav.swapWindow
- system.nav.swapTo

Common Navigation Mistakes

Multiple 'Main' Windows

The most common mistake that will cause windows to stay open unintentionally is to implement a swapping navigation system using the system.nav.openWindow() function on main windows instead of swapTo. When you do this, the next time the swapTo function is called, it may swap from a window that is hidden behind the current 'Main' window and look like nothing happened. It is easy to check the client's Windows menu to see what windows are currently open. If there are more windows listed there than you can currently see, there is a problem in your navigation logic that is failing to close windows properly.

Swapping a Main Window with a Docked Window

Another common mistake that will cause windows to stay open unintentionally is to implement a swapping navigation system using the system.nav.swapTo() function on windows that are docked. This will cause your docked windows to be 'swapped in' as a maximized window instead of its usual size. When you do this, the client will not see it as a main window and next time the swapTo function is called, it may not have space on screen to open the main window. Logging out and back in to the client (or restarting it) is the only solution to this. Identify the offending button or script that is swapping the docked window and change it accordingly.

Related Topics ...

- Vision Project Templates

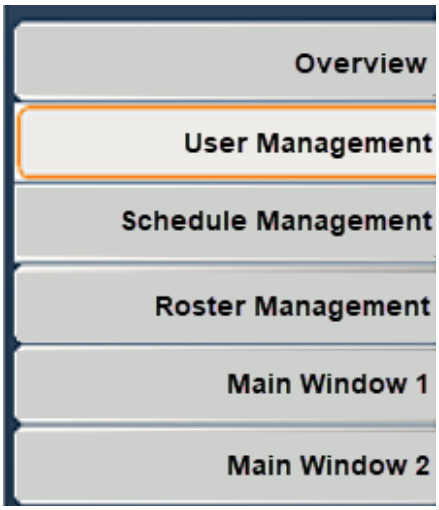
In This Section ...

Navigation - Tab Strip

The Tab Strip component provides a simple navigation strategy used for small project structures having only a few windows. It allows users to see all the navigation tabs on the screen of the client. It is most commonly used in a docked window to provide automatic window navigation. The Tab Strip allows clicking on a tab to swap one main window for another. The Tab Strip has two navigation modes:

- **Swap Windows** - the Tab Strip automatically calls system.nav.swapTo() with the name of the selected tab for easy navigation from one window to another.
- **Disabled** - the Tab Strip doesn't do anything when a tab is pressed. Users can customize tabs using property bindings or by responding to the propertyChange scripting event.

A Tab Strip is an effective primary navigation strategy, particularly when you don't have many items to choose from.



On this page ...

- Tab Strip Navigation Example

Navigation - Tab Strip

Watch the Video

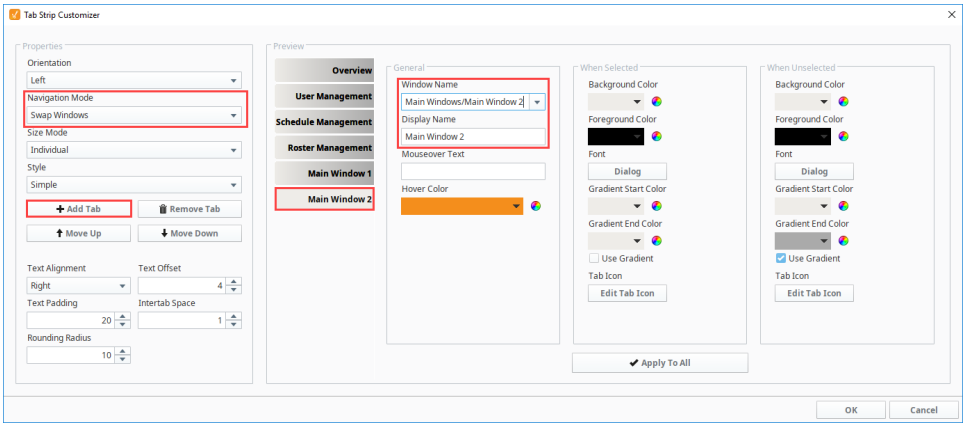
Tab Strip Navigation Example

Tab Strip navigation is simple to set up. In the following example, we'll set up a small project that has a few windows which are visible on the navigation tabs.

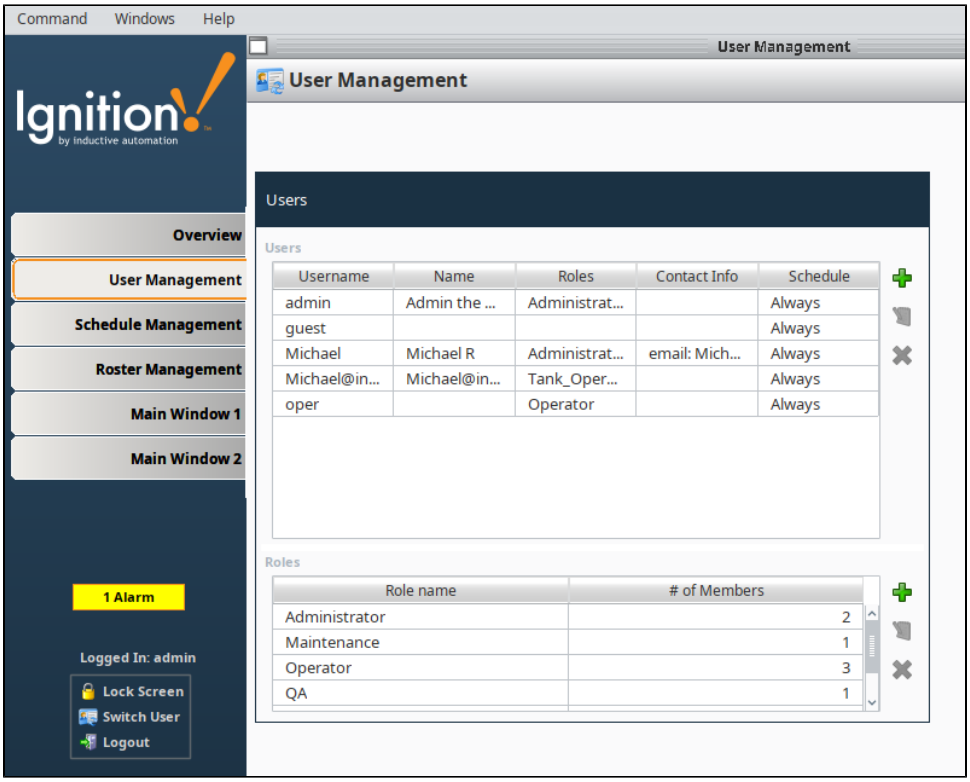
1. Add a Tab Strip component to a window, typically a docked window.
2. Right-click on the Tab Strip component, choose **Customizers > Tab Strip Customizer**.
3. In the Tab Strip Customizer you can specify which window to open with each tab. Notice the **Navigation Mode** property which is set to **Swap Windows** as shown in the screenshot below.
4. To create a new tab, click **Add Tab.** If you have a tab already selected, clicking the **Add Tab** button creates a Tab with the same colors and font as the selected tab.

> **Main Windows already created**
>
> This step assumes you already have your main windows created in your Project Browser.

   a. Under **Window Name** dropdown list, select the window you want to open. Note, it is the full path from the window and not just the name (i.e., **Main Windows/Main Window 2**).
   b. Enter the **Display Name** for your new tab (i.e., **Main Window 2**).
   c. With the **Move Up**, **Move Down**, and **Remove Tab** buttons you can move tabs up and down on the tab strip, and remove a tab if it is no longer required.
   d. You can also set the **Background** and **Foreground** colors when to show when a tab is selected and unselected.
   e. When you're finished, press **OK**.



5. **Save** your project.
6. Open your project in the **Client** and confirm each tab navigates to a different window.

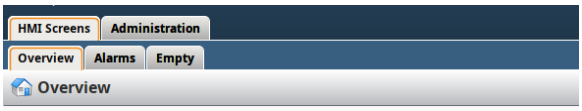


Navigation - Two Tier

Two Tier Navigation is similar to the Tab Strip navigation strategy. It's good for small and regular size project structures where windows are grouped, and lets you organize your main windows into different sections making navigation easy for users. It uses two levels of tabs to navigate around various areas of your project. Once you select a first tier tab, a different set of tabs appear in the second tier to switch between different windows.

This works a bit differently than the default Tab Strip navigation, as the first tier Tab Strip will actually not do any window swapping, which will instead be left up to the second tier of tabs. The Two Tier approach has a docked window that contains multiple sets of tabs. One set is used as a higher level of grouping of windows, and is used to conditionally swap out another set of tabs based upon user selection.

In the image below, the top tier of tabs contains the HMI Screens and Administration tabs. Clicking on **HMI Screens** makes a second tier of tabs appear (the set containing **Overview**, **Alarms**, and **Empty**). Clicking on the **Administration** tab would make a different set of sub tier tabs appear. Thus, the top tier of tabs isn't directly responsible for any sort of window navigation. Rather, it's used to make other sets of tabs appear.



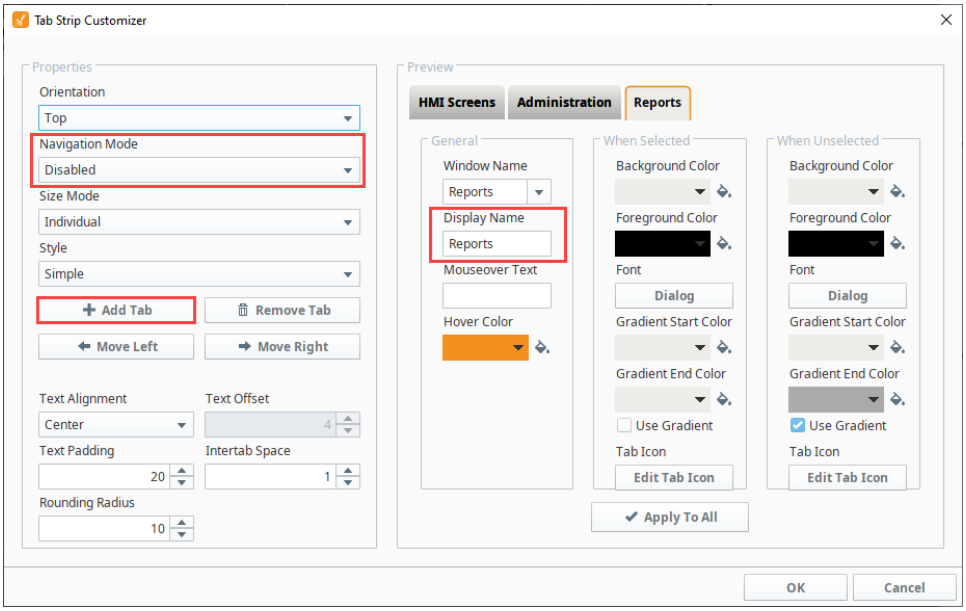
On this page ...

- Two Tier Navigation Example

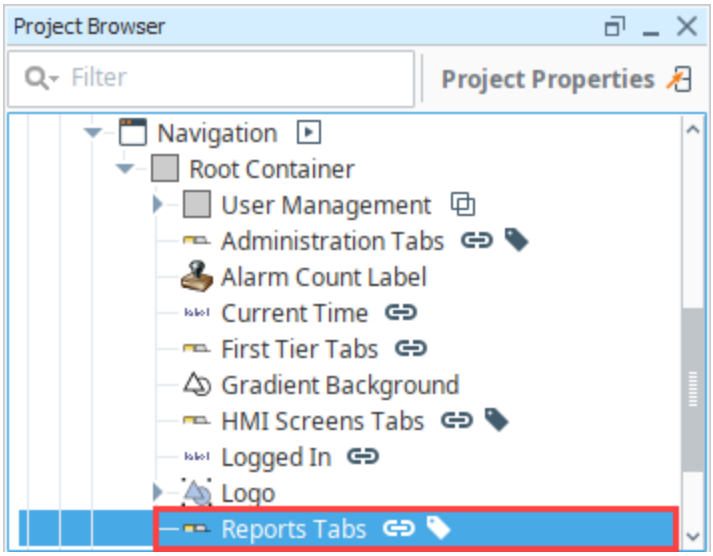
Two Tier Navigation Example

In this example, we are using the 2-Tier Tab Nav project template which is selectable upon project creation. This comes with two tiers of tab strips and several default tabs. We will add one tab on the first tier and two tabs on the second tier.

1. Right click on the **First Tier Tabs** in the Project Browser to add another tab. Select **Customizers > Tab Strip Customizer.**
2. To create a new tab, click **Add Tab**, and position it on the tab strip any where you like.
3. Enter your **Window Name** and **Display Name**. Make sure the Navigation Mode is set to **Disabled** since this first tab strip is not swapping to any windows.



4. Now let's create a second tier category of tabs for the Reports tab. The easiest way to do this, is copy the second tier of tabs from either the HMI Screens or Adminstration tabs in the root container of the Navigation folder of the Project Browser. **Paste** it in the root container of the Navigation folder giving it a unique name identifying what it is (i.e. Reports Tabs).

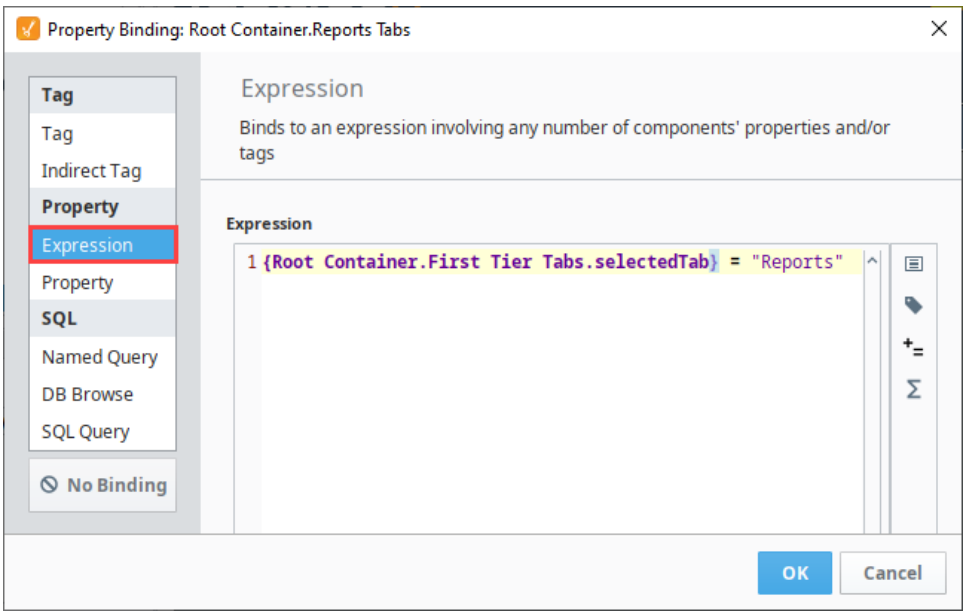


5. Right click on the **Reporting Tabs Tab Strip** of the Designer and select the **Customizers > Tab Strip Customizer.**
6. Add Tabs like you would for a normal Tab Strip, but only add tabs fitting that category of windows. In this step, we are adding two tabs: Report 1 and Report 2. Click **OK** to create the new tabs.

> **Note:** This example assumes you already have some main windows created in your Project Browser (i.e., Report 1 and Report 2).

7. The second tier tabs can be shown or hidden depending on what tab is selected in the first tier.
   a. Select a second tier Tab Strip (i.e.,Reports).
   b. Go to the **Property Editor**, select the **Visible** property and set it to true, and then select its binding icon.
   c. Select the **Expression** binding and set up an expression to be true when the appropriate first tier tab is selected, as shown in the image below. Click **OK**.

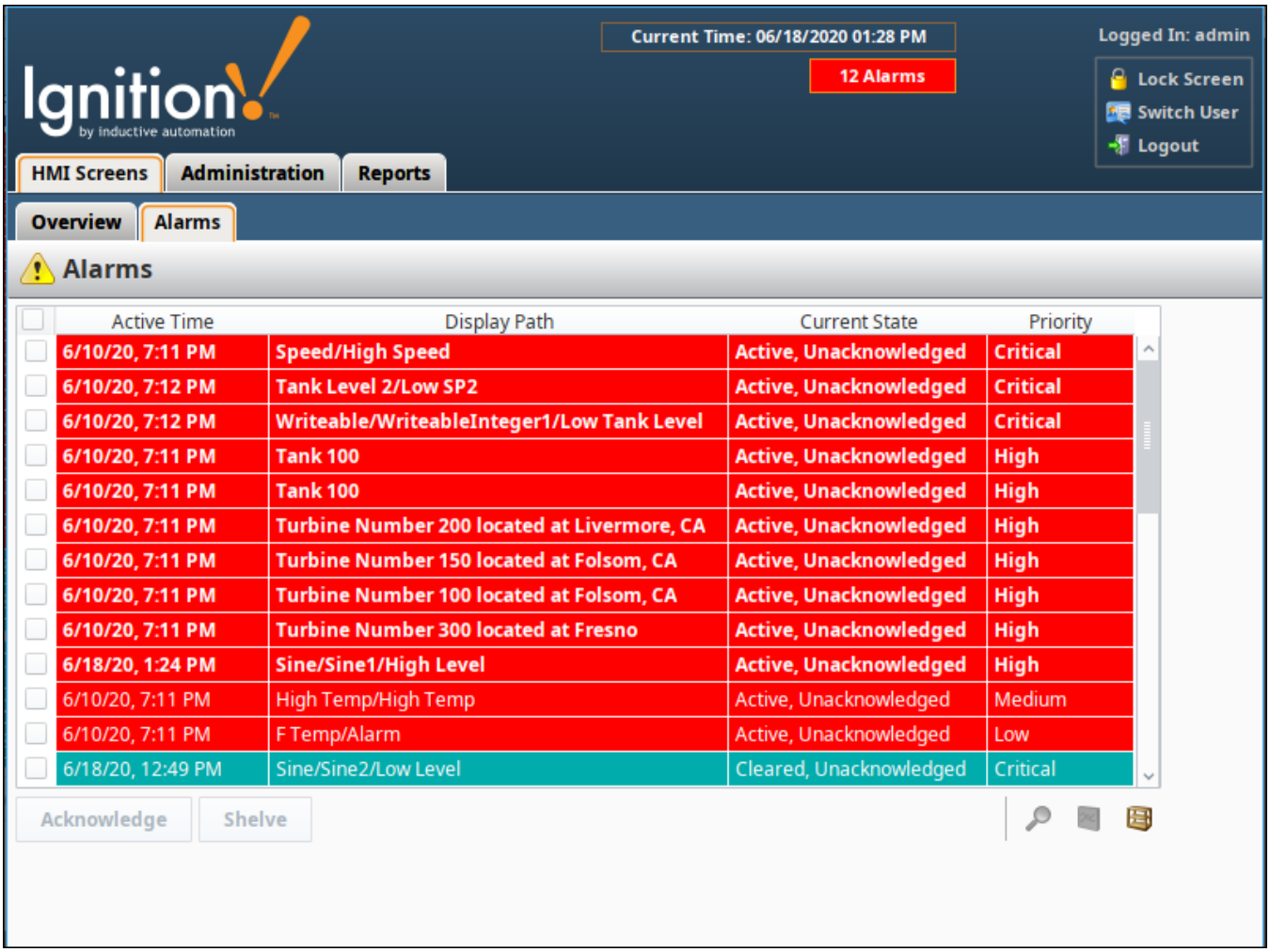
```
{Root Container.First Tier Tabs.selectedTab} = "Reports"
```



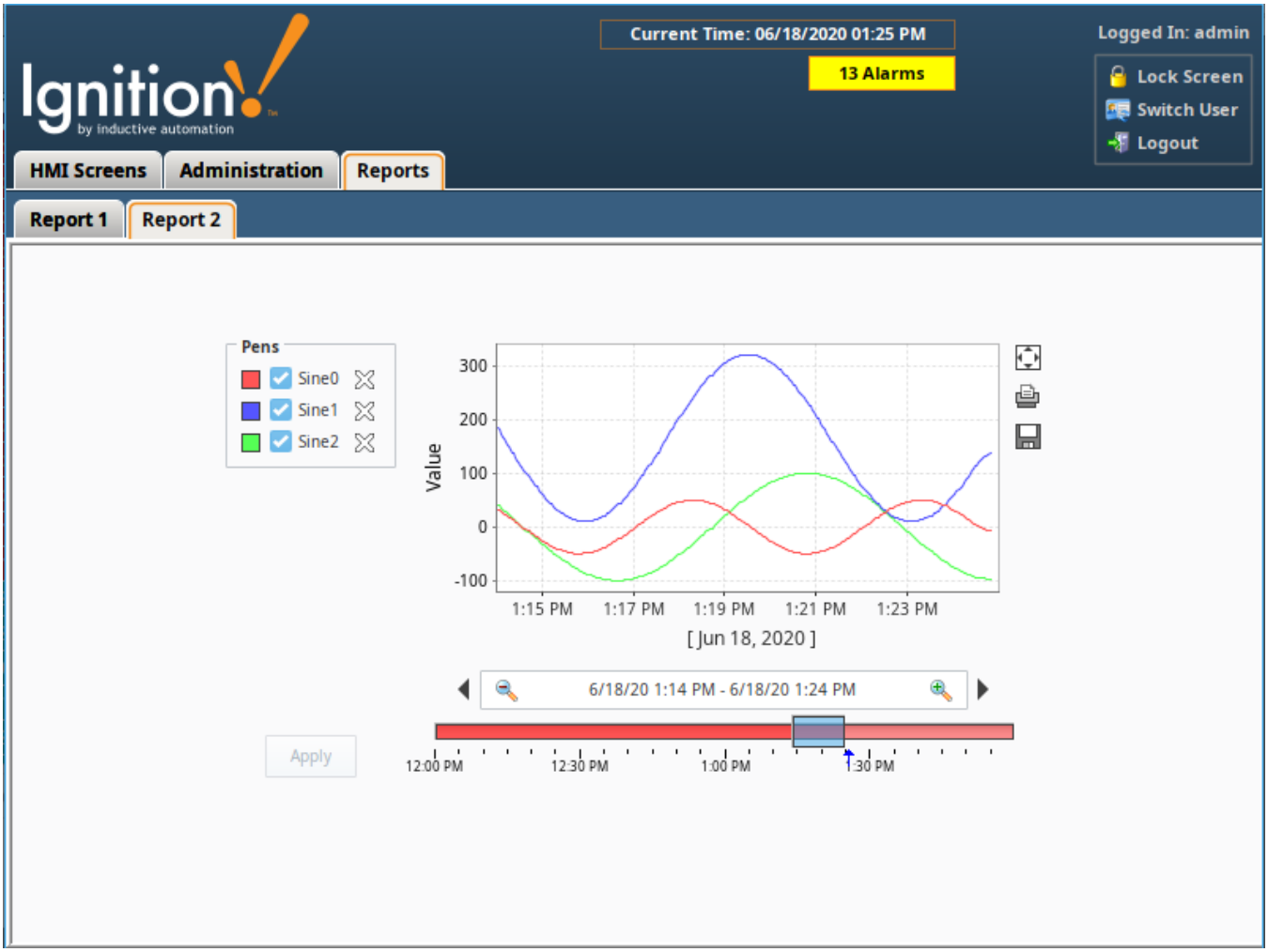
8. **Save** your project.
9. Open your project in the **Client**, click on the various tabs to see your first and second tier tabs switch between the different windows.

The following images show the first and second tier tabs for **HMI Screens** and **Reports** tabs.

The **HMI Screens** tab shows the **Overview** and **Alarms** windows on the second tier.

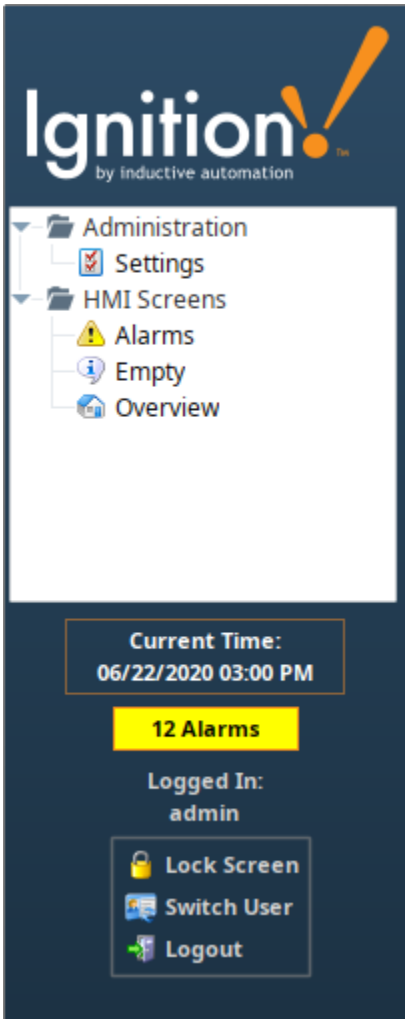


The **Reports** Tab shows the **Report 1** and **Report 2** windows on the second tier.


Ignition
by inductive automation
Current Time: 06/18/2020 01:25 PM
13 Alarms
Logged In: admin
Lock Screen
Switch User
Logout
HMI Screens
Administration
Reports
Report 1
Report 2
Pens
Sine0
Sine1
Sine2
300
200
100
0
-100
Value
1:15 PM
1:17 PM
1:19 PM
1:21 PM
1:23 PM
[Jun 18, 2020]
6/18/20 1:14 PM - 6/18/20 1:24 PM
12:00 PM
12:30 PM
1:00 PM
1:30 PM
Apply

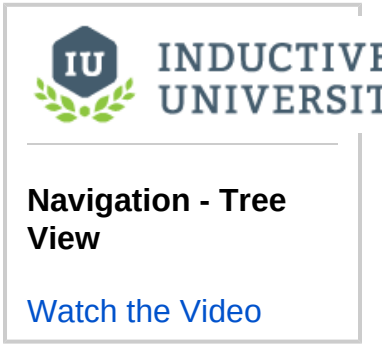
Navigation - Tree View

The **Tree View** navigation strategy is excellent for large project structures. It uses a typical navigation strategy again with a docked west window that contains a Tree View to navigate around to various areas. Users can double click on an item in the tree view and it will swap out one main window for another. The list is fairly compact, and can contain folders, helping you to group similar windows just like you would in the project browser.



On this page ...

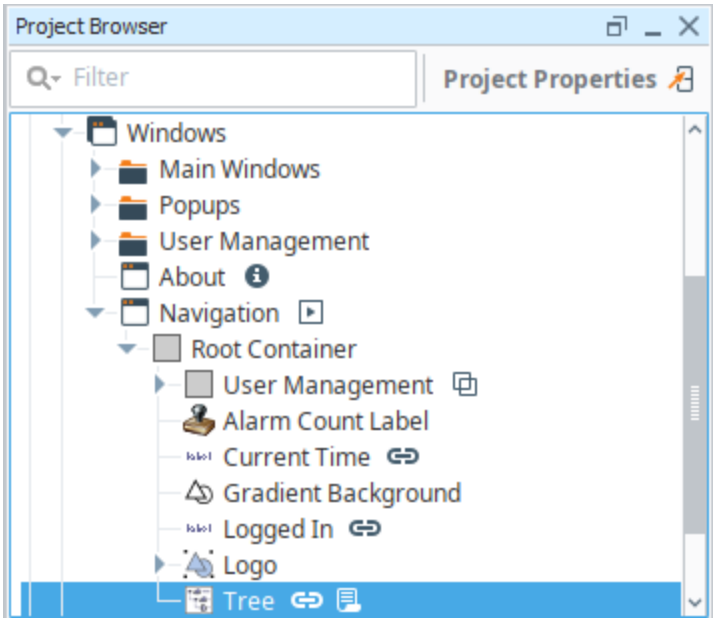
- Tree View Navigation Example



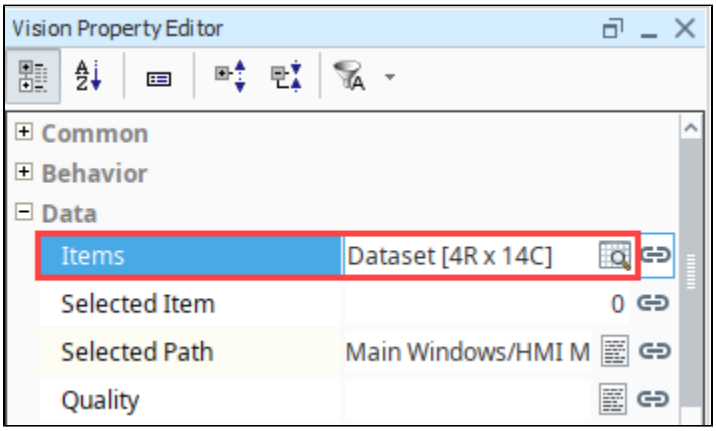
Tree View Navigation Example

In this example, we are using the Tree View Nav project template which is set up when the project is created. By default, Tree View navigation comes with several default folders to help get you started. This simple example adds one new main window.

1. Once you create your project and set the Tree View as your navigation strategy, open the **Project Browser**, and expand the **Main Windows** folder. Click the **Navigation** folder to open the skeleton project, then select the **Tree** property.



2. From **Property Editor**, find the **Items** property and click on the **Dataset Viewer** icon.

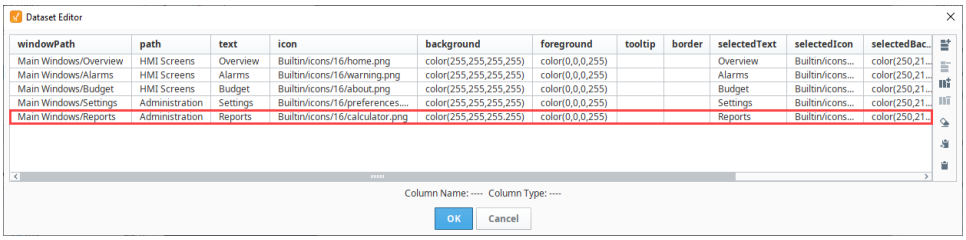


3. This brings up a **dataset editor**. You will see a number of columns that identify how each tree view item is displayed. Each row corresponds to a node in the tree view. The **windowPath** column is the window that we want to navigate to. The **path** is the folder that the window will display in the Tree View.

> **Note:** This example assumes you already have your new window created in your Project Browser.

4. Let's add a new main window under Administration. To add a new Main Window (i.e., Reports) we first need to add a row. Click the **Add Row** icon and populate the fields manually using the data from the previous row, or you can also copy the entire dataset into a notepad where you can manually manipulate the data to add a row, and then paste it back into the dataset editor. Click **OK**.

> ⚠ Do not use the Tree View Customizer to edit any of your data. Use only the Dataset Editor, otherwise it will overwrite that item's dataset.

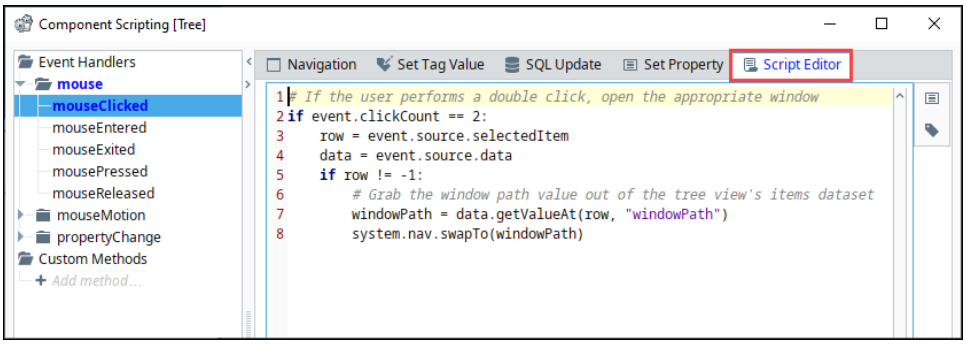


5. You can then add a script that will use the newly added windowPath to open the correct window when a user double clicks on a node. Right click on the Tree View component and select **Scripting**.

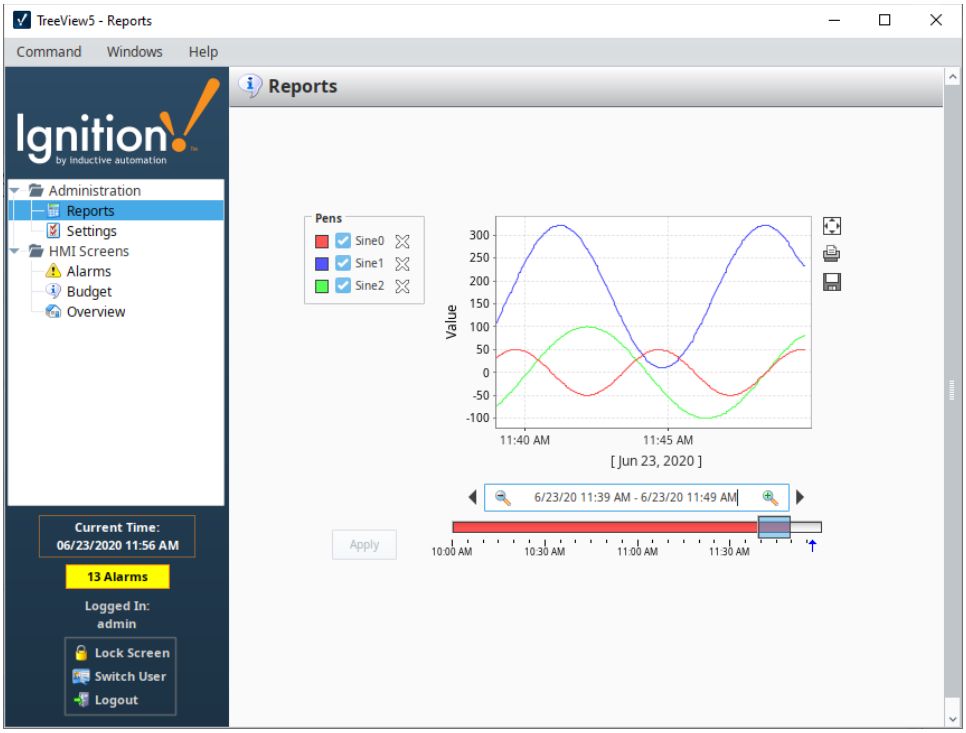
6. Select the mouseClicked event handler, and add the following script to the **Script Editor** tab.

mouseClicked code

```
# If the user performs a double click, open the appropriate window
if event.clickCount == 2:
        row = event.source.selectedItem
        data = event.source.data
        if row != -1:
                # Grab the window path value out of the tree view's items dataset
                windowPath = data.getValueAt(row, "windowPath")
                system.nav.swapTo(windowPath)
```

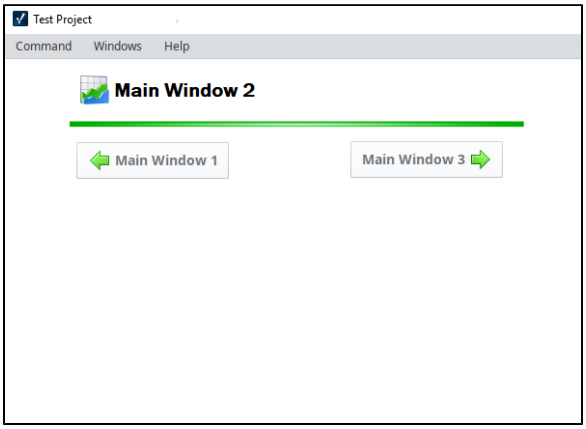


7. **Save** your project and launch the **Vision Client** and test out the Tree View navigation.


TreeView5 - Reports
Command
Windows
Help
Ignition
by inductive automation
Administration
Reports
Settings
HMI Screens
Alarms
Budget
Overview
Current Time:
06/23/2020 11:56 AM
13 Alarms
Logged In:
admin
Lock Screen
Switch User
Logout
Reports
Pens
Sine0
Sine1
Sine2
300
250
200
150
100
50
0
-50
-100
Value
11:40 AM
11:45 AM
[Jun 23, 2020]
6/23/20 11:39 AM - 6/23/20 11:49 AM
Apply
10:00 AM
10:30 AM
11:00 AM
11:30 AM

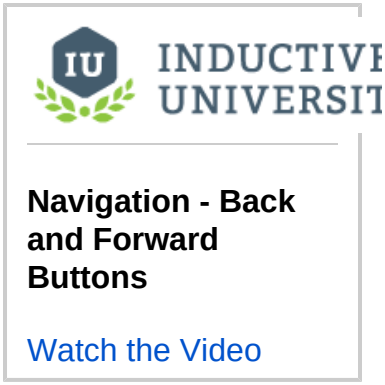
Navigation - Forward and Back Buttons

Another navigation strategy in Vision is to set up Forward and Back Buttons to navigate between different windows. This strategy is perfect for a small business process with ordered steps. It does not have a docked window, tree view, or tabs to navigate around. It is one big main window and has buttons to navigate forward and back from one Main Window to the next in the list.



On this page ...

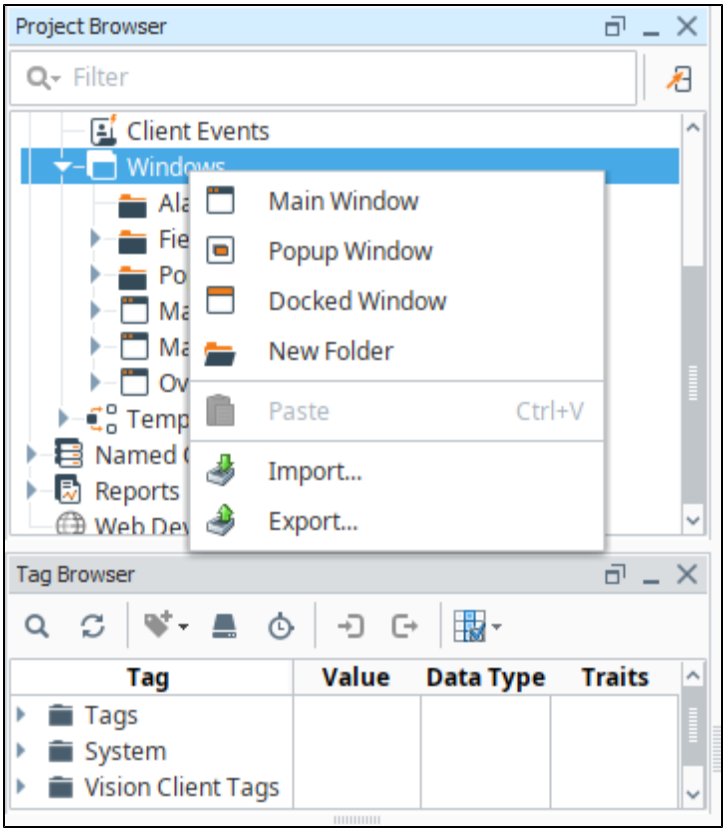
- Forward and Back Buttons Example



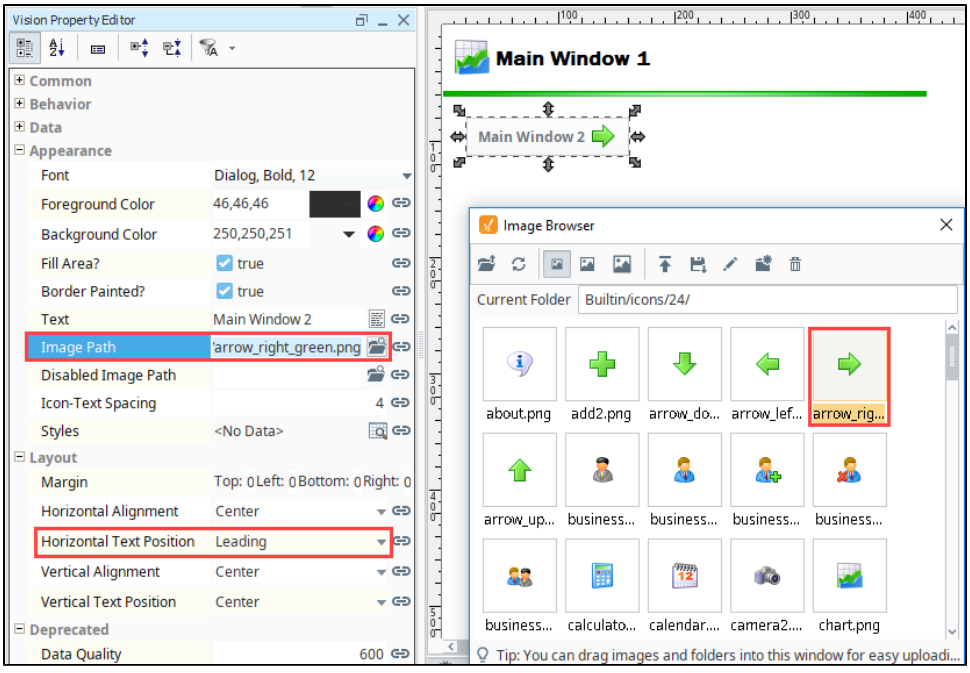
Forward and Back Buttons Example

In this example, we'll use the forward and back buttons on a main window to navigate between different windows in a project.

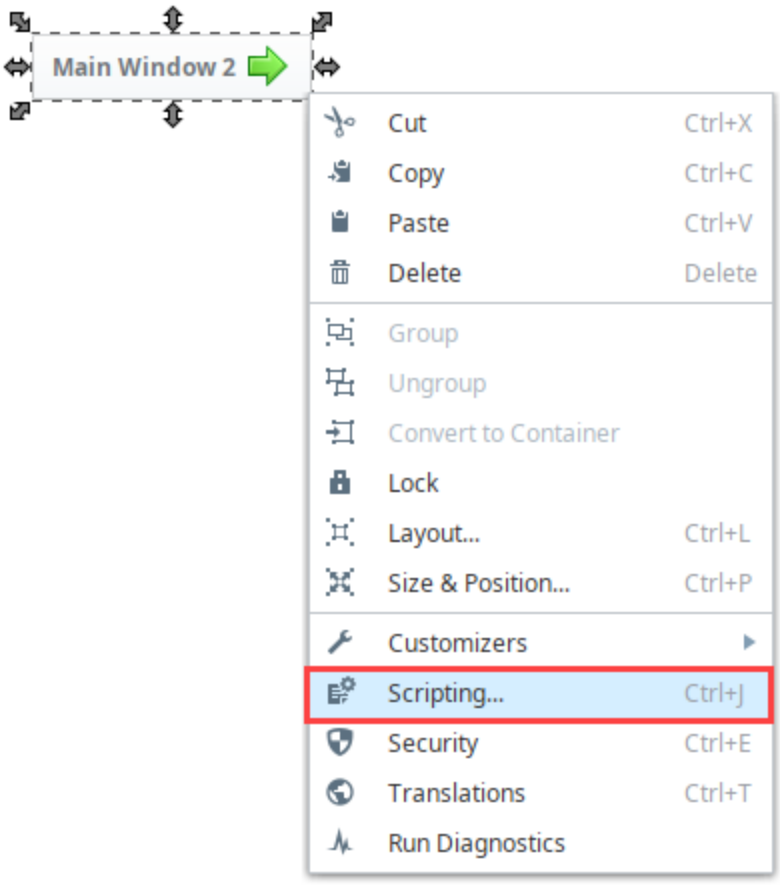
1. In the **Project Browser**, right click on the Main Windows folder and create a new **Main Window**. Enter a name for your window to whatever best describes the window (i.e., Main Window 1). Add a label to the window for clarity (i.e., Main Window 1) so when you navigate through different windows, you know precisely what window you're viewing. Repeat this step to create Main Window 2 and Main Window 3.



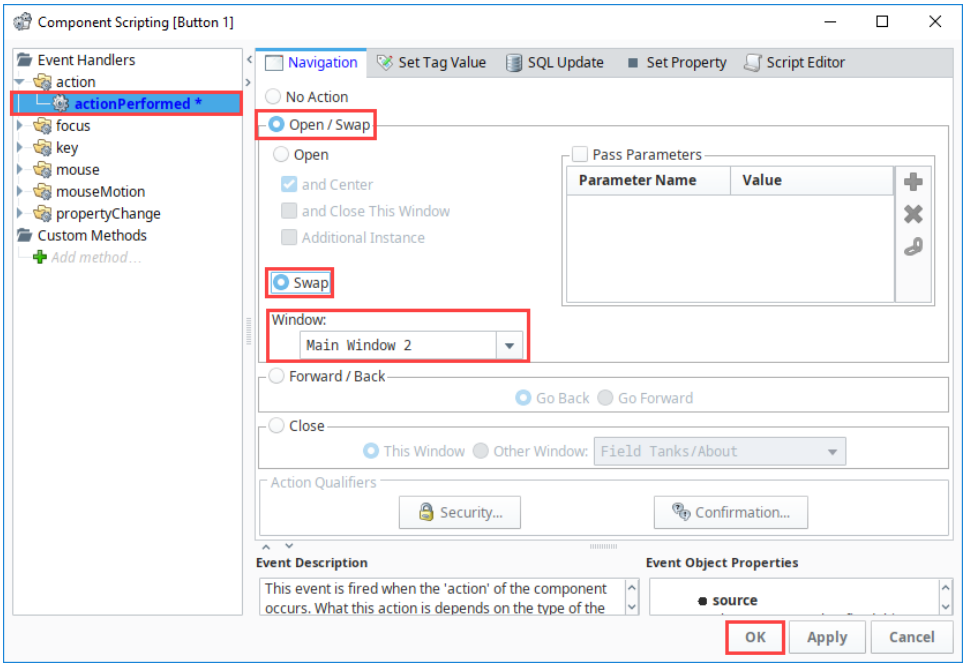
2. From the **Component Palette** in Designer, drag a **Button** component to your Main Window 1. Label the button, **Main Window 2**.
3. Next we'll add a right green arrow to the Button component. With the Button component selected, click the folder to the right of the **Image Path** property . This opens the Image Management Tool.
4. Open the Built-in/icons/24/ folder and select a right green arrow. Close the Image Management Tool.
5. Set the **Horizontal Text Position** property to **Left** in the Property Editor.



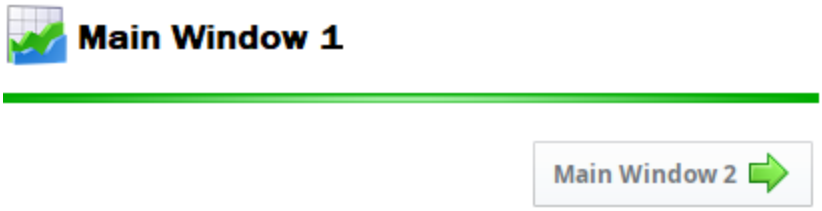
6. Now you need to tell the Main Window 2 button what to do when a user clicks on it. Right click on the **Main Window 2** button, select **Scripting** . The component scripting dialog box will open.



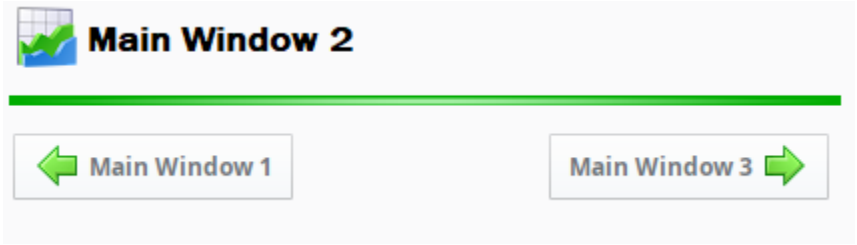
7. Under the Event Handlers, open the **action** folder and select **actionPerformed**.
    a. Click the **Open / Swap** and **Swap** radio buttons. The Swapping function builds a simple script to go back and forth between different windows. The Swap function ensures only one main window will be open at a time.
    b. From the **Window dropdown box**, select **Main Windows/Main Window 2**.
    c. Click **OK.**



8. **Save and Publish** your project.
9. Open your **Client.** Click on the **Main Window 2** button and Main Window 1 will be swapped out with Main Window 2.



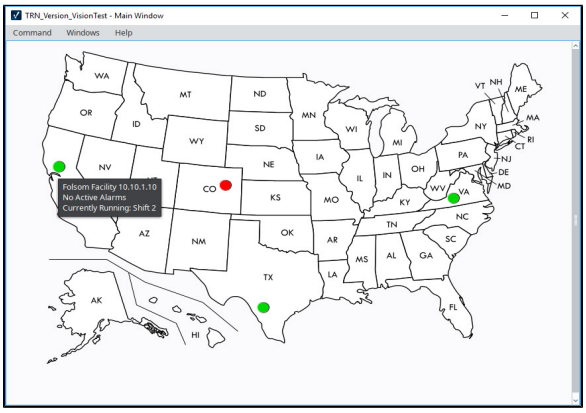
10. Repeat Steps 2 and 3 to create Main Windows 2 and 3, respectively. On Main Window 2 make sure you have a button for both Main Window 1 and Main Window 3. Set your window path for the Main Window 1 button to **Main Windows/Main Window 1** (refer to step 4). For the Main Window 3 button, set the window path to **Main Windows/Main Window 3**.
11. When Main Window 2 is open, the Main Window 1 and Main Window 3 buttons are displayed. Clicking either button opens the respective window.



Navigation - Drill Down

Another popular navigation strategy is to drill down into various areas of your project using a map. The Drill Down navigation strategy is ideal if you have different geographical locations. A good example is to have a main window that has an image representing a plant or factory. The image can have any type of drawing tool component, such as an rectangle, circle, etc., that overlays the image. When the user selects one of the overlay components, the Client swaps windows with a window that displays information that pertains to the selected area of the plant.

It is a very simple navigation strategy to set up and is popular because it lets users select different areas on an image and drill down to access specific information about that geographic location.



On this page ...

- Drill Down Navigation Example

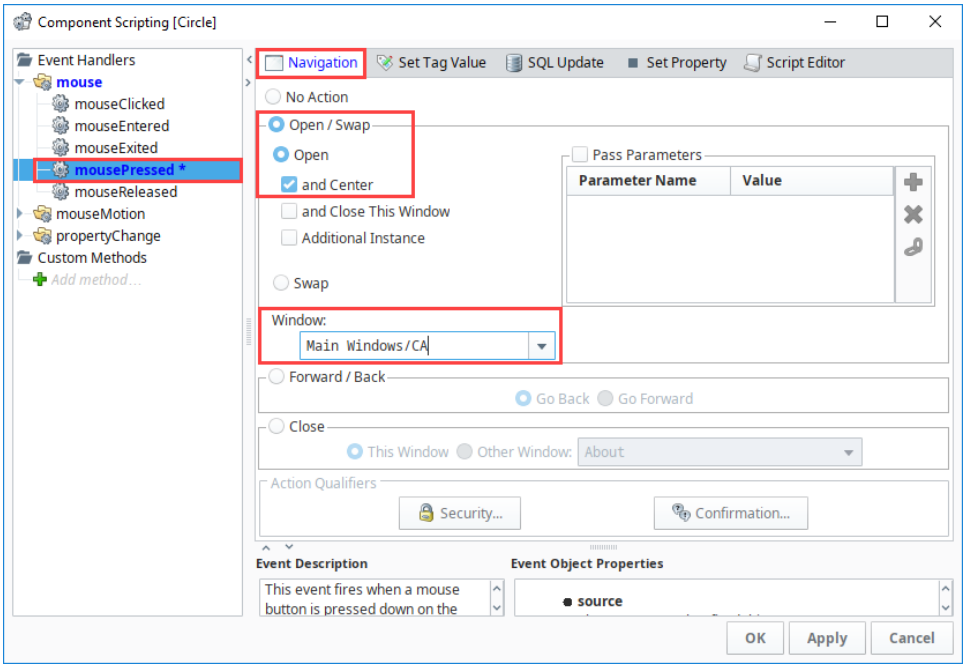
Navigation - Drill Down

Watch the Video

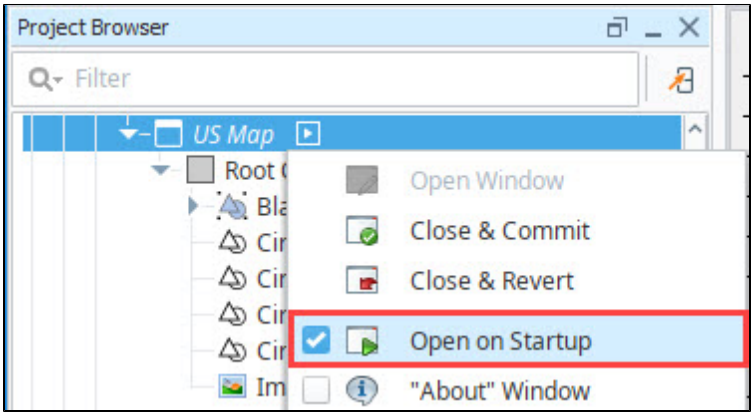
Drill Down Navigation Example

This example demonstrates how to use a US map and configure it to get information about different geographical facilities sprinkled across the US.

1. Drag an **image** on to a Main Window. It can be any type of image including a photo. This example uses an image of a US Map.
2. Add a Drawing Tool shape such as a rectangle, circle, polygon over an area on your image to identify the location. In the example, we use different color circles.
3. Right click your drawing tool and select **Scripting**.
4. Select the mousePressed event handler, and with the **Navigation tab** selected.
    a. Select the **Open/Swap** radio button.
    b. From the **Window** drop-down box, select the window that relates to the selected area on the map.
    c. Click **OK**.



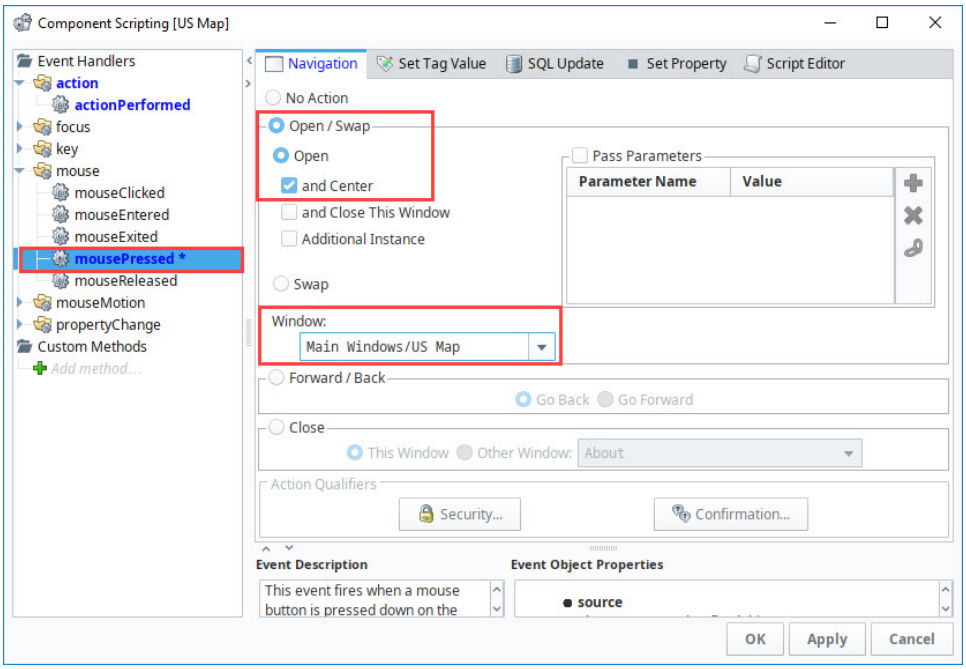
5. In the Project Browser, select your US Map window and set it to **Open on Startup**.



> **Note:** This example assumes you already have your area windows created in your Project Browser.

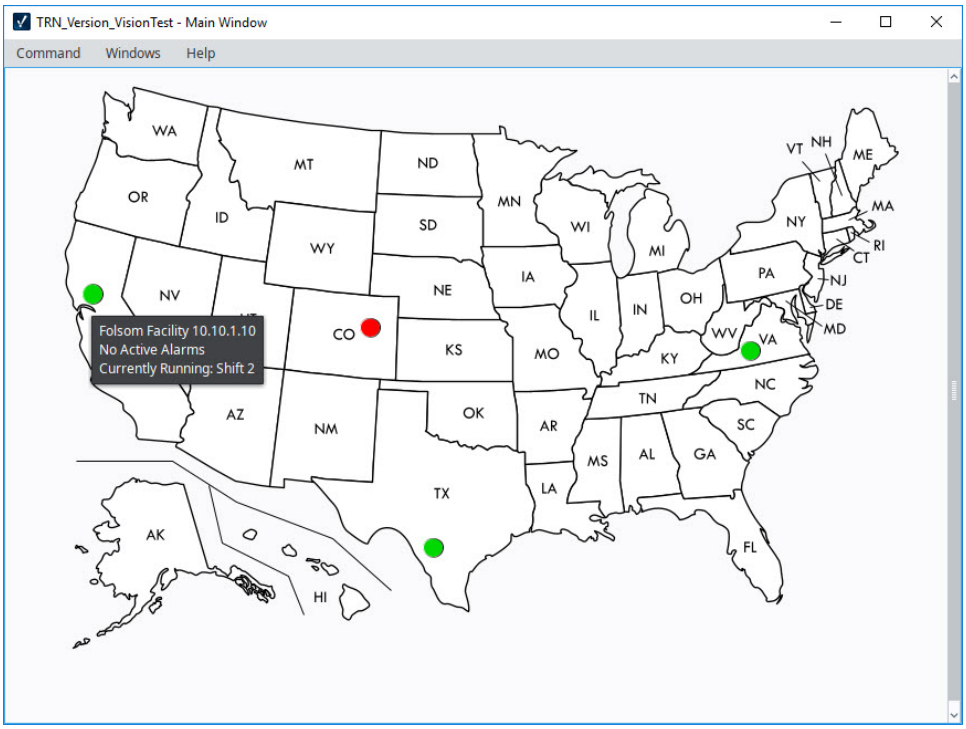
6. On your area window, add a Button component so you can navigate back to the main window containing the overview map. (If you have multiple area windows, copy and paste this button on to each window).
   a. **Open** your area window.
   b. Add a **Button** component.
   c. Right click on the **Button** component and select **Scripting**.

7. Select the mousePressed event handler, and with the **Navigation tab** selected:
   a. Select the **Open/Swap** radio button.
   b. From the **Window** drop-down box, select the window that is your overview map.
   c. Click **OK**.

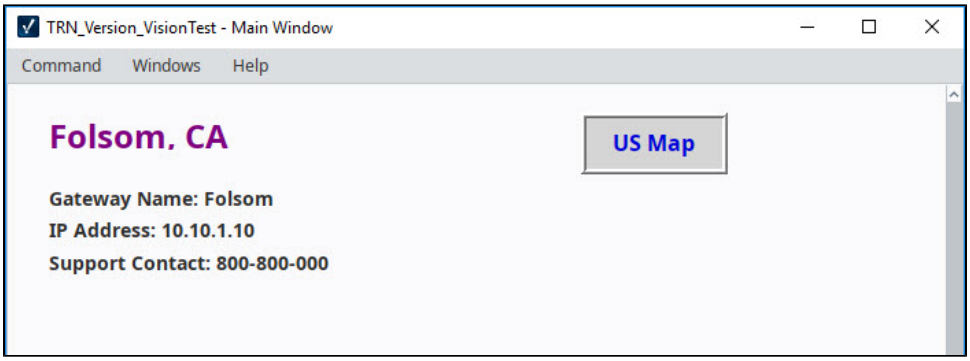


8. In **Preview Mode**, test your window navigation by switching between windows.

9. **Save** your project.
10. Now you can try it out by opening your Client and clicking on a shape to navigate to the selected area. Prior to clicking, the **Mouseover Text** displays the location information when you hover over one of the circles. Once you click on a designated area on the map, a new window will open.

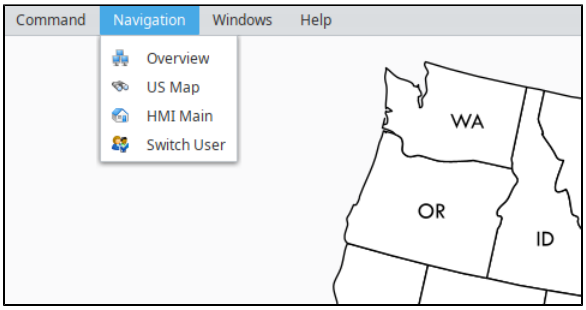


11. Click on the **US Map button** to go back to the US Map.



Navigation - Menubar

You can set up a special menu within the Menubar that allows you to navigate throughout the project using the scripting functions. They can be simple, like swapping to a window, or be more complex in how they navigate around the project. The benefit of using the Menubar for navigation is that it keeps navigation tucked away instead of using up valuable screen space.



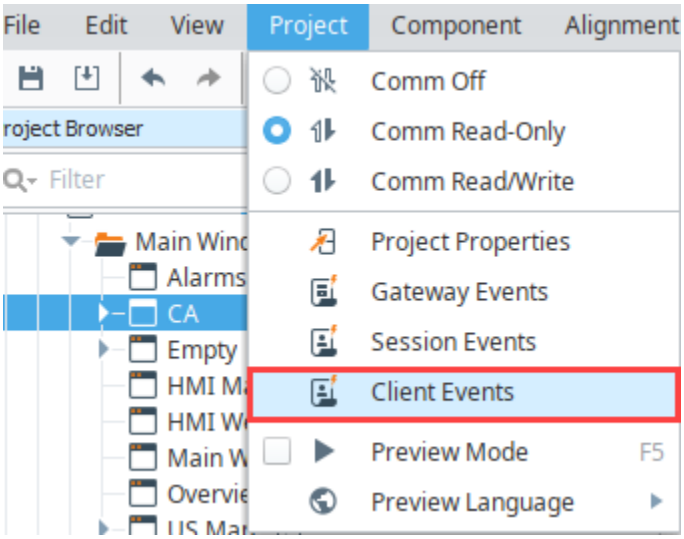
On this page ...

- Menubar Navigation Example

Menubar Navigation Example

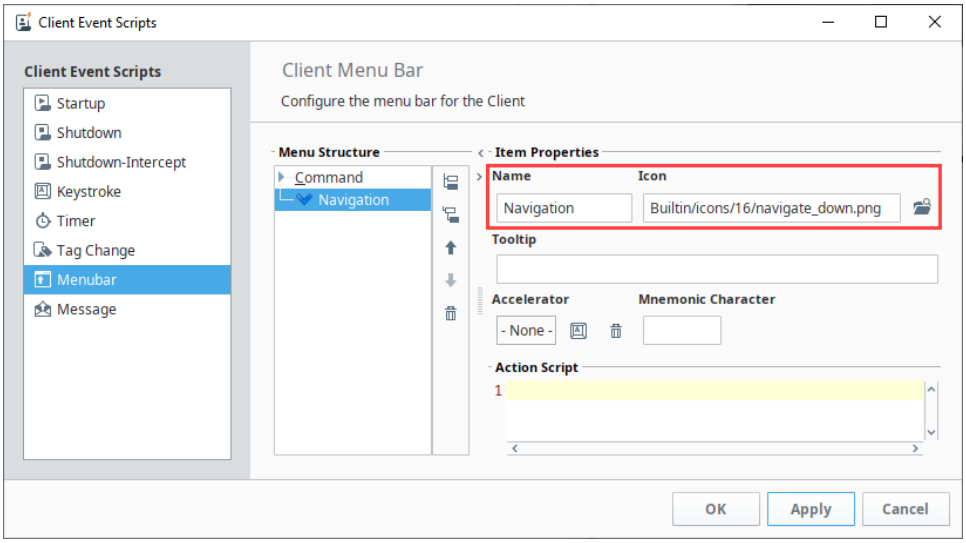
In this example, we'll set up sibling and child options on the menubar. This example assumes you have at least an Overview window created.

1. In the menubar of the Designer, click on **Project** then select **Client Events.**



2. This opens the screen below in the Client Event Scripts space. Click on the **Menubar** under Client Event Scripts.
3. Select the **Add Sibling** icon to add a Menu Item. Update the Name to a new menu option (i.e., Navigation). You can also add a path to an icon if desired.

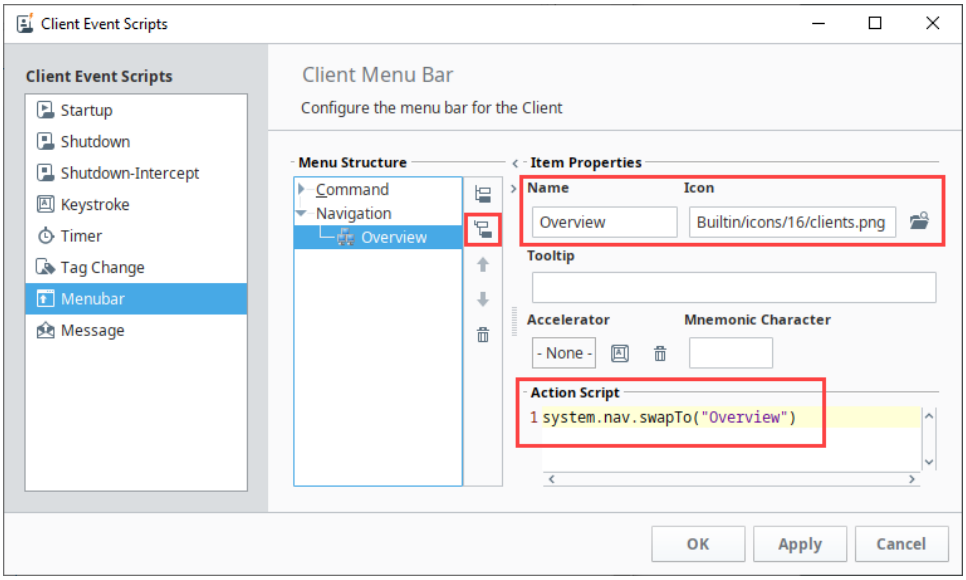
> **Note:** By default, there are three commands under the Command menu option: Logout, Lock Screen, and Exit.



4. Click the **Apply** button.
5. Click the **Add Child** icon to add a new option under the Navigation menu.
    a. Give the menu item a name that is appropriate for the window it will be navigating to (i.e., Overview).
    b. Add a script that will swap to the window.

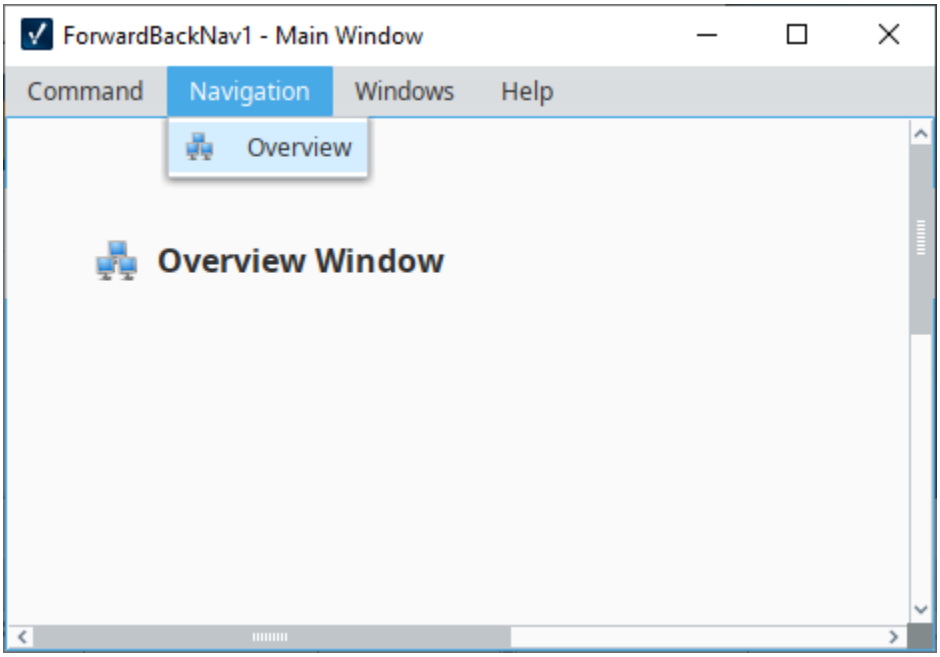
Code Snippet - Menubar navigation

```
system.nav.swapTo("Overview")
```



6. Repeat step 5 to add as many windows as needed. New groups of windows can even be nested within the parent Navigation Menu.
7. Click **OK** to save your new menu structure.
8. **Save** your project.

9. Now you can open the **Client** to navigate from one window to another using the menubar structure. Notice that the Navigation menu option is located in the menubar because it was created as a sibling, where as the window added as a child is listed under the Navigation tab.



Navigation - Retargeting

Retargeting is a special form of navigation which involves navigating to an entirely different project. Retargeting is accomplished through scripting, usually as a response to a button press or other component event. The system.util.retarget() function allows you to 'retarget' the Client to a different project. You can have it switch to another project on the same Gateway, or another Gateway entirely, even across a WAN. This feature makes the vision of a seamless, enterprise-wide SCADA application a reality.

The retarget feature will attempt to transfer the current user credentials over to the new project / Gateway. If the credentials fail on that project, the user will be prompted for a valid username and password. Once valid authentication has been achieved, the current project is shut down and the new project is loaded.

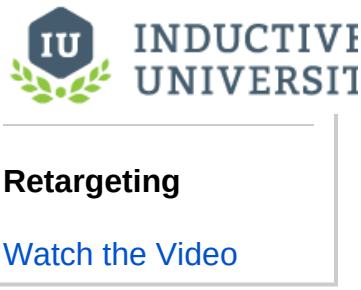
You can pass any information to the other project through the parameters dictionary. All entries in this dictionary will be set in the global scripting namespace in the other project. Even if you don't specify any parameters, the system will set the variable _RETARGET_FROM_PROJECT to the name of the current project and _RETARGET_FROM_Gateway to the address of the current Gateway.

Retargeting can be as simple as a single line of code, just make sure you are using the project name (no spaces allowed), and not the title. See the retarget function in the appendix for more information.

A typical retargeting strategy actually combines this strategy with one or more other navigation strategies. A simple landing project could be made so that all users would have access to perform basic user management functions, and then use a screen with Button components that retarget approved users out to other projects targeting a specific area of operations. The buttons that retarget to these projects can be hidden or shown based on the user, allowing you to build in an extra layer of security to your projects. Additionally, each of the other projects would utilize navigation strategies that best suit those areas.

On this page ...

- Retargeting Navigation Example



Retargeting Navigation Example

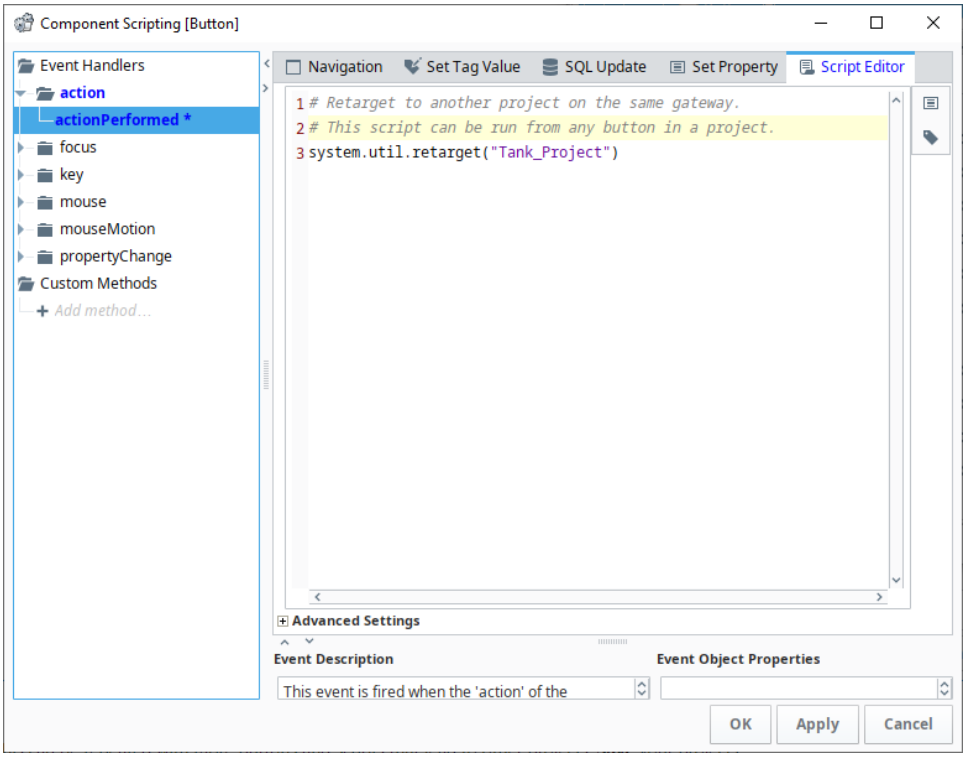
In this example, we have two projects (**Tank** and **Motor**) that we will set up to retarget to each other.

1. On a blank window, add a **Button** component, and either change the button text or add a label that informs the user that the button retargets to a new project.
2. Right-click on the **Button** and select the **Scripting** option.
3. On the **action** > **actionPerformed** event, add the following script to the script builder.

Python - Retargeting to Another Project

```
# Retarget to another project on the same gateway.
# This script can be run from any button in a project.
system.util.retarget("My_Other_Project")
```

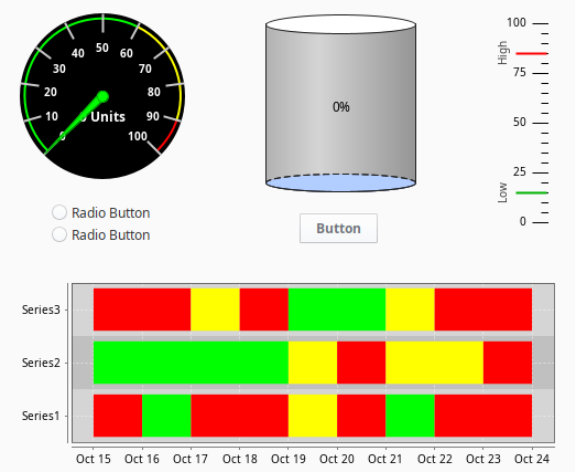
4. Replace the **My_Other_Project** text with the name of the project you want to access. In this case, the script text was changed to **Tank_Project** for the **Motor** project to navigate to the **Tank** project when clicked.



5. Click **OK**.
6. Repeat steps 1-4 in the other project, the Tank project, replacing the **Tank_Project** script with **Motor_Project**. This can be repeated with more buttons and scripts that lead to other projects.
7. **Save** your projects.
8. Open the **Client** and select the **Retarget** buttons to navigate between the **Tank** project and **Motor** project.

Working with Vision Components

The Vision Module comes with a host of built-in components that you can select from for use in your project. One thing that you'll find when working with components is there are a few different ways to manipulate and layout components on a window when working in the Designer. Here is a small sampling of the components available in Vision. See the Vision Components Appendix page for a complete list of components.



This section introduces you to how to work with components so you can learn how to quickly select, move, resize, duplicate, and group components during the design process.

On this page ...

- Selecting Components
  - Mouse Selection
  - Tree Selection
- Component Properties
  - Data Types
  - Dataset Editor
- Manipulating Components
  - Resizing
  - Moving
  - Duplicating
  - Rotating
  - Size and Position
  - Component Grouping
  - Component Layout
  - Relative Layout
  - Font Scaling
  - Anchored Layout

Selecting Components

There are a number of different ways to select components within a window, each of which has their own advantages.

Mouse Selection

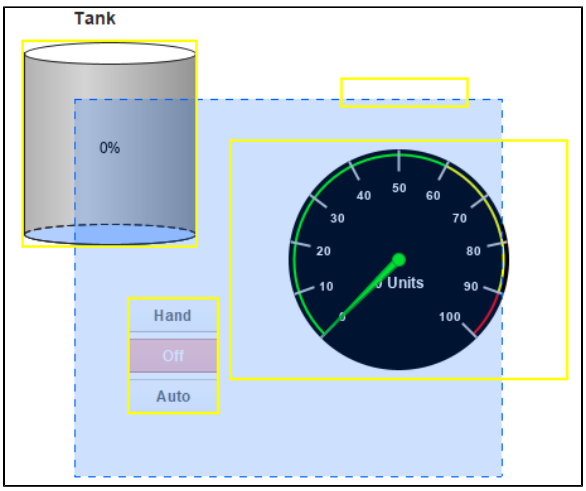
Using the mouse to directly click a component is the most common way. Make sure the Selection icon on the Drawing Tools menubar is enabled, then click on a component to select it. If the component you want to select is obscured by other components, hold down **Alt** and keep clicking, the selection will step down through the z-order.

Window-Selection

You can also select components using window-selection. Click-and-drag within a container to draw a selection rectangle. If you drag the window left-to-right, it will select all components that are completely contained within the rectangle. If you drag the window right-to-left, it will select all components that the rectangle touches.

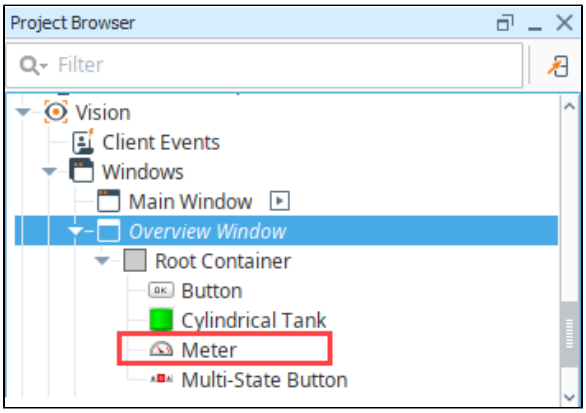
Lastly, you can start dragging a window selection and then hold down the **Alt** key to use touch selection. This will draw a line as you drag, and any components that the line touches will

become selected. As you're using these techniques, components are given a yellow highlight border.



Tree Selection

By selecting nodes in the **Project Browser** you can manipulate the current selection. This is a handy way to select the current window itself, which is hard to click on since it is behind the Root container. However, you can click to it, using **Alt-click** to step down through the z-order. It is also the only way to select components that are invisible.

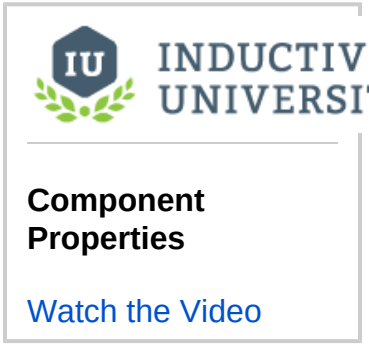


Component Properties

Each component has a unique set of properties that can be set and modified within the **Property Editor**. A property is simply a named variable with a distinct type that affects something about the component's behavior or appearance. You can also create your own custom properties on the component, which act like variables that can store any information that you want on the component.

Data Types

There are a wide variety of datatypes across all of the Vision Module's components. Each property has a distinct type, which dictate what values will be allowed. Below are the common data types.

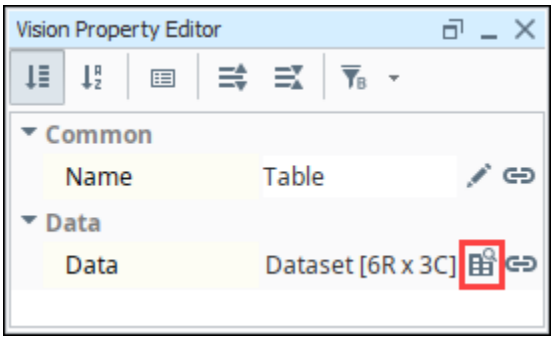


Numeric Types	
Boolean	A true/false value. Modeled as 0/1 in Python. Technically, 0 is false and anything else is true.
Short	A 16-bit signed integer. Can hold values between -32,768 to 32,767, inclusive.
Integer/int	A 32-bit signed integer. Can hold values between -2,147,483,648 to 2,147,483,647 inclusive.

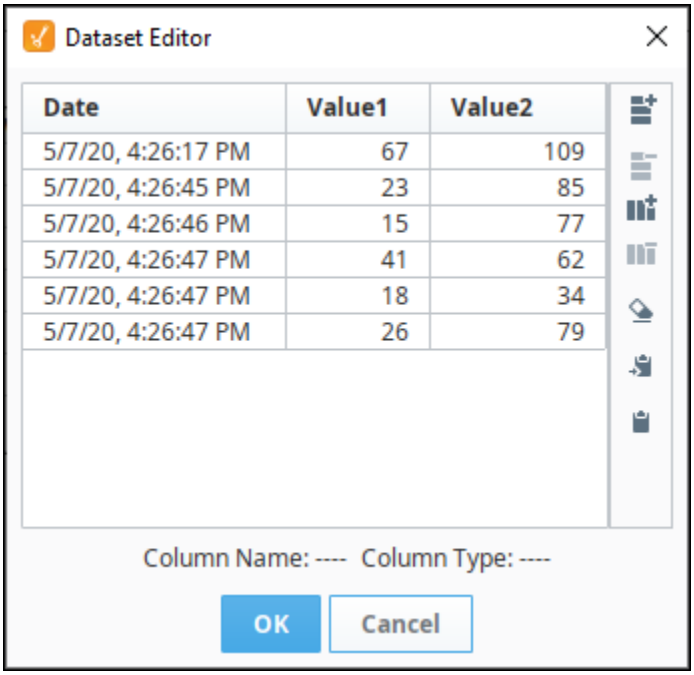
Long	A 64-bit signed integer. Can hold values between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive.
Float	A 32-bit signed floating point number in IEEE 754 format.
Double	A 64-bit signed floating point number in IEEE 754 format.
Non-Numeric Types	
String	A string of characters. Uses UTF-16 format internally to represent the characters.
Color	A color, in the RGBA color space. Colors can easily be made dynamic or animated using Property Bindings or Styles.
Date	Represents a point it time with millisecond precision. Internally stored as the number of milliseconds that have passed since the "epoch", Jan 1st 1970, 00:00:00 UTC.
Data set	A complex data structure that closely mimics the structure of a database table. A Dataset is a two-dimensional matrix (also known as a table) of data organized in columns and rows. Each column has a name and a datatype.
Font	A typeface. Each typeface has a name, size, and style.
Border	A component border is a visual decoration around the component's edges. You can make a border dynamic by using the Style Customizer, the toBorder() expression function, or scripting with the Java border object.

Dataset Editor

The Dataset Editor icon appears next to the binding icon for the Data property. Clicking the icon brings up the Dataset Editor window where you can view and make changes to the raw data. Note, any changes will be overwritten the next time your binding polls.



With the Dataset Editor, you can add and delete columns and rows, delete all rows, and copy information to or from the clipboard. When adding columns, you have multiple formats to choose from including string, date, integer, double, float, etc.



The Dataset Editor icons actions are detailed in the table below.

Icon	Action
	Add row
	Delete selected rows
	Add a column
	Delete selected column
	Delete all rows
	Add to clipboard
	Paste from clipboard

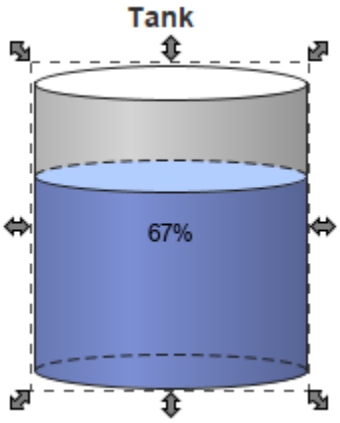
Manipulating Components

Manipulating components can be done with both the mouse and the keyboard. You can move components around, resize them, and rotate them.

Resizing

When you select the component you want to resize, they'll get eight resize-handles displayed around the edge of the selection. These handles look like double-sided arrows around the perimeter. Use the mouse to drag them to change the size of the components in the selection. To maintain the selection's aspect ratio, hold down **Ctrl** as you resize. To resize around the center of the current selection, hold down **Shift**. These can be used at the same time.

You can also resize the current selection using the keyboard. To nudge the right or bottom edge of the selection in or out, use **Shift** combined with the arrow keys, which resizes by the nudge distance, which defaults to one pixel at a time. To nudge the top or left edge of the selection, use **Ctrl-Shift** co mbined with arrow keys. To resize faster, hold the **Alt** key as well, to move the component the alt nudge distance, which defaults to ten pixels at a time.



Moving

To move the component, simply drag it anywhere within the container's bounds. You can also move whatever is currently selected by holding down **Alt** while dragging, regardless of whether or not the mouse is over the current selection. This is important because it is the primary way to move a Container component. (Normally, dragging in a container draws a selection rectangle inside that container).

While a component is selected, you may also use the keyboard's arrow keys to move a component around the nudge distance. Just like resizing with the arrow keys, to move the alt nudge distance, use the **Alt** key.

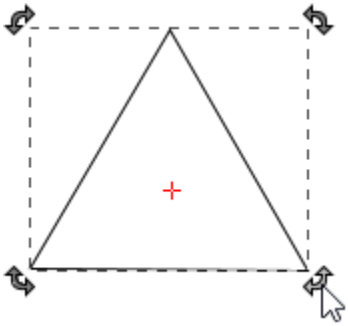
Duplicating

Components can be easily duplicated by dragging them as if you were going to move them and holding down the **Ctrl** key. This will drop a copy of the component at the desired drop location. It is often useful to also hold down **Shift** key as you do this to ensure exact alignment. You may also use the **Ctrl-D** shortcut to quickly duplicate a component in place.

Rotating

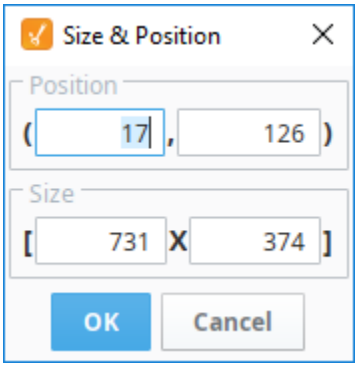
Shapes can be rotated directly using the selection tool. Other components cannot be rotated in this manner. To rotate a shape, first select it using the selection tool so that you see the resize handles around it. Then simply click on it once again and you'll see the rotation handles appear. Clicking (but not double-clicking) on selected shapes toggles back and forth between the resize handles and the rotation handles.

Once you see the rotation handles, simply start dragging one to rotate the shape or shapes. Holding down the **Ctrl** key will snap your rotation movements to 15° increments. When the rotation handles are present, there is also a small red crosshair handle that starts in the middle of the selection. This is the rotation anchor: the point that the selection will rotate around. You can drag it anywhere you'd like to rotate around a point other than the center of the shape.



Size and Position

Components can also be positioned and resized with the Size and Position window. This window allows you to type in an exact pixel size of the component as well as x/y coordinates that the component will be at (with the upper left point of the component moving to that point). To access the size and position window, right-click on the component and select Size and Position.



Component Grouping

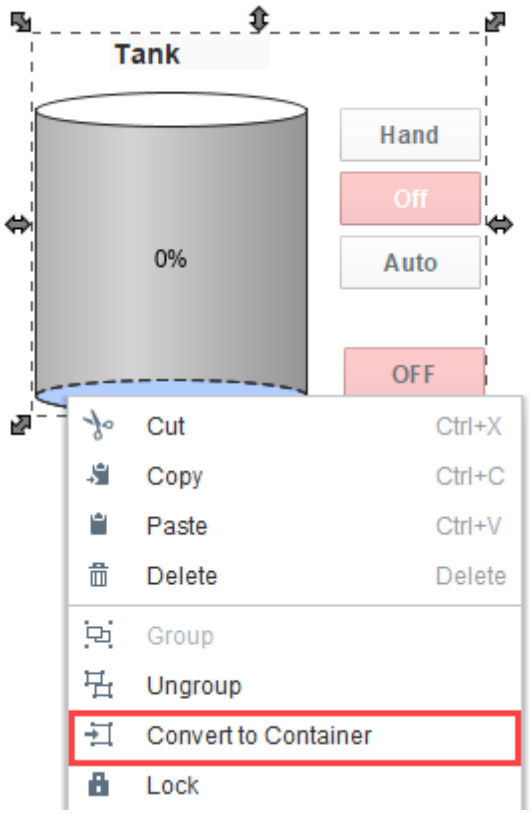
Shapes and components can be grouped together so that they act like a single component in the Designer. Grouping components is very similar to putting them in a Container. In fact, it is the same thing as cutting and pasting them into a perfectly-sized container and then putting that container into group mode, with one exception. If the group contains only shapes and no other kinds of components, it will be a special shape-group that has the ability to be rotated and has some other shape-like properties.

When components or shapes are in a group, clicking on them in the Designer will select the group instead of the shape. If you double-click on a group, it will become "super-selected", which will allow you to interact with its contents until you select something outside of that group.

Groups can contain other groups, creating a nested structure. Groups themselves are also components, meaning that you can add custom properties to groups, bind them, and so on.

Difference between a Container and a Group

It is helpful to use Groups and Containers to organize the components on your window. You can select multiple components and right-click to convert them into a group, then right-click again convert that group to a container.



Layout works differently for groups. The layout setting for components and shapes inside a group is **ig nored**. All members of a group act as if they are in relative layout with no aspect ratio restrictions. This special group-layout mode is also active when resizing a group inside of the Designer, whereas traditional (container) layout doesn't take effect in the Designer.

Component Layout

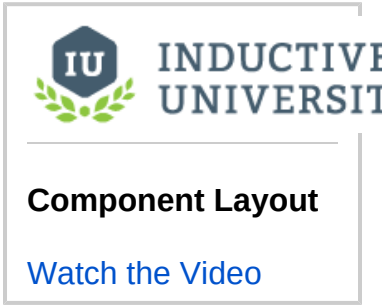
Layout is the concept that a component's size and position, relative to its parent container's size and position, can be dynamic. This allows the creation of windows that resize gracefully using either **Anc hored** or **Relative** layouts and can optionally keep the original aspect ratio.

This is a very important concept because of the web-launched deployment of Vision clients - they often end up being launched on many different monitors with many different resolutions.

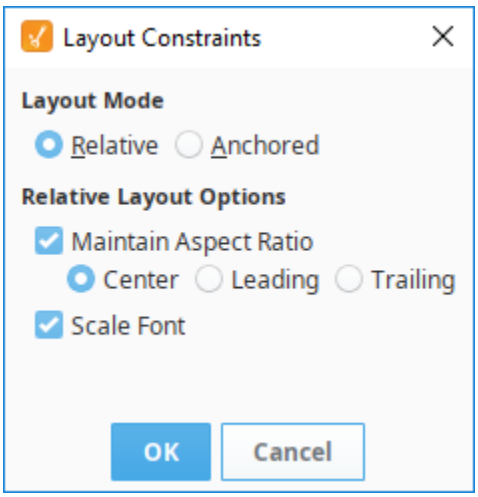
This is also important for components that have user-adjustable windows like popup windows. Imagine a popup window that is mostly displaying a large table or chart. If you're running on a large monitor, you may want to make the window bigger to see the table or chart easier. Of course, this is only useful if the table or chart actually gets larger with the window.

Changing a component's layout is as simple as right-clicking on the component and opening the Layout dialog box. You can also alter the default layout mode that gets assigned to new components. See Desig ner/Window Editing Properties.

There are two layout modes, and they are set on a per-component basis. Both affect the component's size and position relative to its parent container. The root container's size is dictated by the window size. To edit the layout of a component, right-click on the component and select **Layout** from the menu. the Layout Constraints window displays showing all the default settings. These default settings



can be altered in the Project Properties.

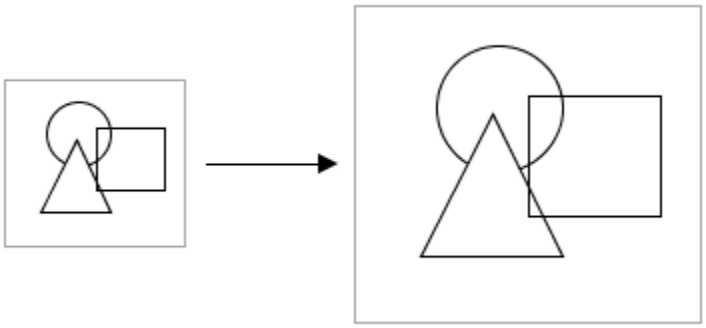


Layout Modes

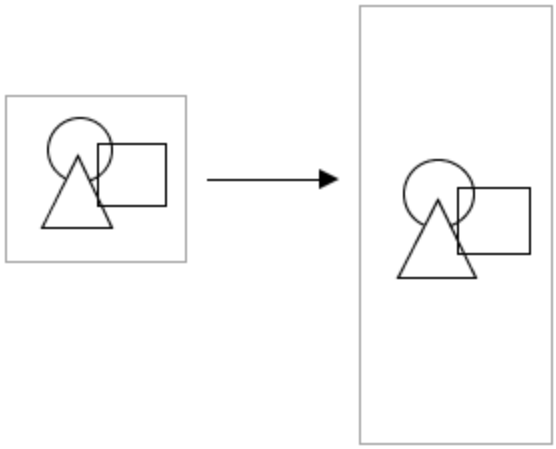
- **Relative**
  This mode makes a component's size and location relative to its parent's size and location. When the parent changes size, the component changes accordingly. This creates components that auto-scale.
- **Anchored**
  This mode makes the edge of a component's two axes (horizontal and vertical) anchored to the edge or edges of its parent.

Relative Layout

Relative Layout is the default mode. This is a simple and effective layout mode that keeps a component's size and position constant relative to its parent container, even when the parent container grows or shrinks. More precisely, it remembers the component's position and size as a percentage of its parent's bounds at the last time the window was saved. Relative Layout also has the option of scaling a component's font appropriately.



Note that Relative Layout mode respects aspect ratio. So, if the parent component is distorted, the content will not be. The extra space is distributed evenly on both sides of the content.



Relative Layout Options

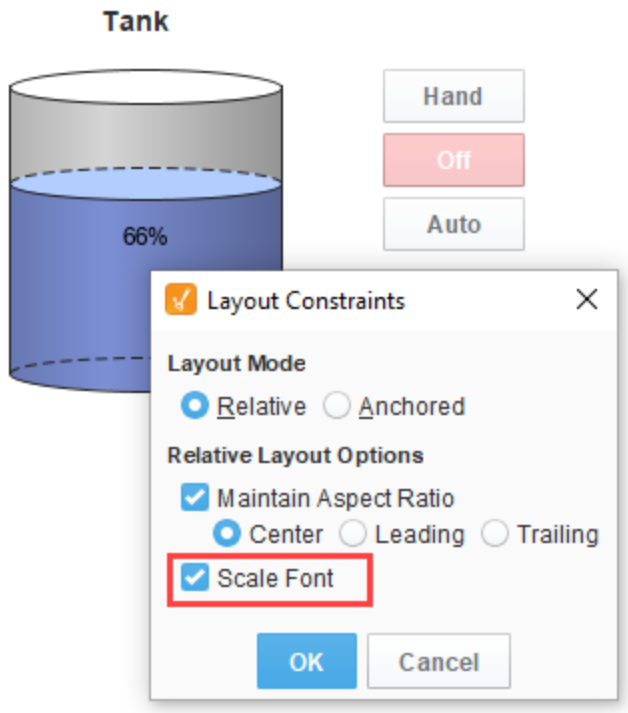
- **Maintain Aspect Ratio**
  If selected, the component's original aspect ratio is preserved. Otherwise, it can stretch tall or wide.
- **Center**
  When maintaining the aspect ratio, centers the component with respect to its parent.
- **Leading**
  When maintaining the aspect ratio, aligns the component with the parent's leading edge.
- **Trailing**
  When maintaining the aspect ratio, aligns the component with the parent's trailing edge.
- **Scale Font**
  If selected, the component's font will scale along with its size as the relative layout adjusts the component. This will override other font size settings. If this setting is applied to a Group, then all components in the group will use this setting.

Font Scaling

By default, font scaling is enabled on all components, but it can behave differently on some components so it’s good to test it out before putting it into production. You can change the default Component Layout settings in **Project Properties** under **Vision > Design > Relative Layout Options.**

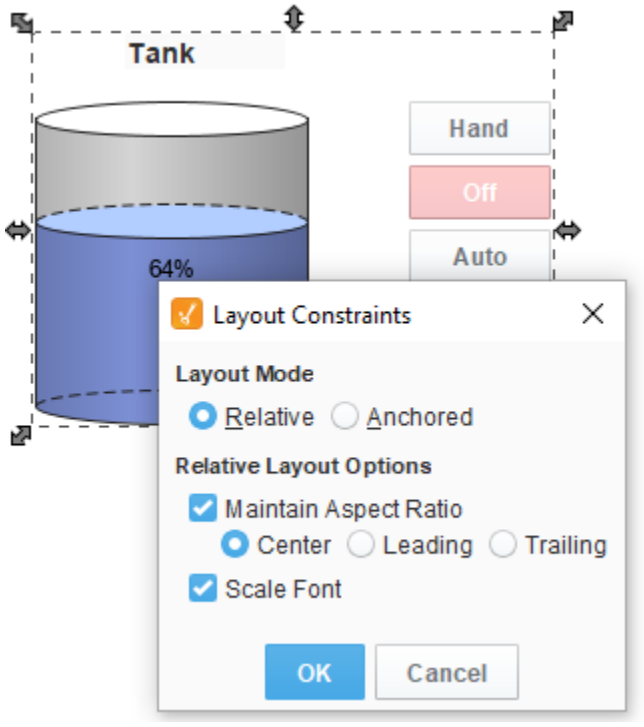
Font Scaling on Individual Components

When individual components are dragged into a window, the following default Layout settings for each component are applied. All components on the window default to font scaling.

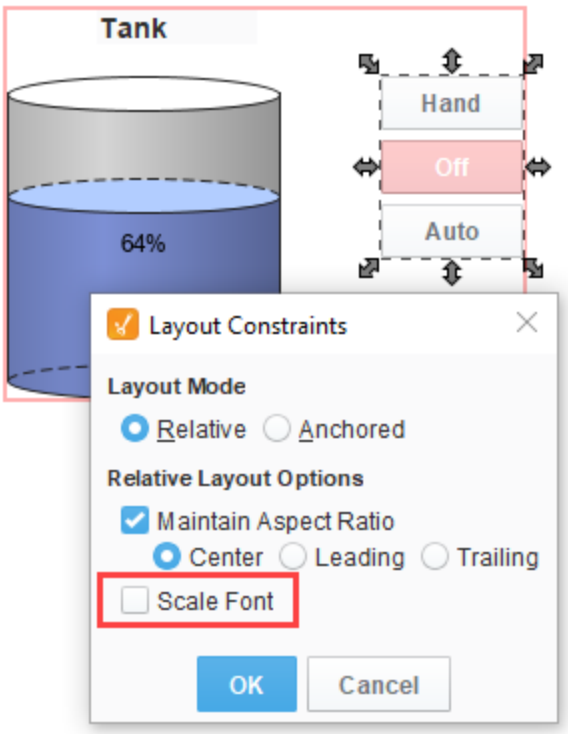


Font Scaling on Groups

When individual components are grouped together, the default Layout settings are applied to the new group component, and to each component in the group. All the components in the group default to font scaling.



Font scaling on individual components in a group can be disabled by selecting the group, and double clicking on a single component. You will get a red outline around the group, then you can select the individual component(s), and disable font scaling. **Note:** The individual component font scale setting takes precedence within a group.



When you remove the group (right-click and select **Ungroup**), all the individual components within that group will get reset to the default Layout settings including font scaling, even if font scaling was set differently.

Font Scaling on Containers

You can convert a group to a container to change the way scaling works. Ignition remembers the last Layout Settings you made to each individual component in that group prior to the conversion to a container (i.e., if any of the components had font scaling disabled, once the group is converted to a container, those same components will still have font scaling disabled).

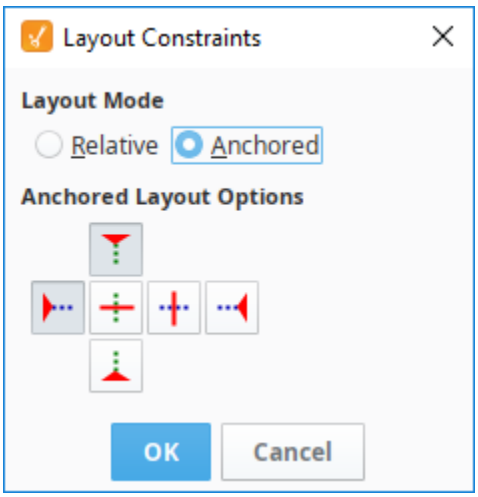
Anchored Layout

Anchored Layout lets you specify various "anchors" for the component. The anchors dictate how far each of the 4 edges of the component stay from their corresponding edges in the parent container.

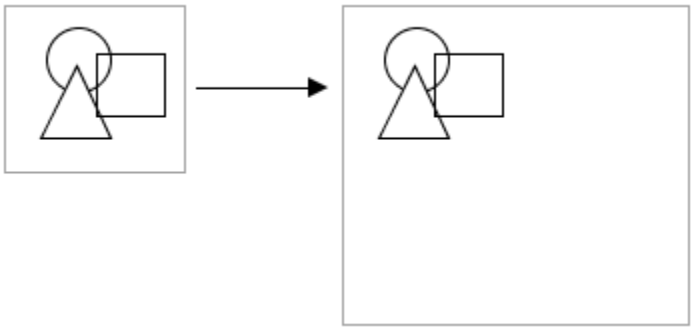
Anchored Layout Options

- **North/South**
  If one of these is selected, the distance between that edge of the component and that edge of the container is preserved. If both are selected, the component will stretch its **height** to maintain both distances.
- **West/East**
  If one of these is selected, the distance between that edge of the component and that edge of the container is preserved. If both are selected, the component will stretch its **width** to maintain both distances.
- **Center Vertically**
  When selected, both top and bottom buttons will be deselected. This option maintains the height of the component and centers it **vertically** in the container.
- **Center Horizontally**
  When selected, both left and right buttons will be deselected. This option maintains the width of the component and centers it **horizontally** in

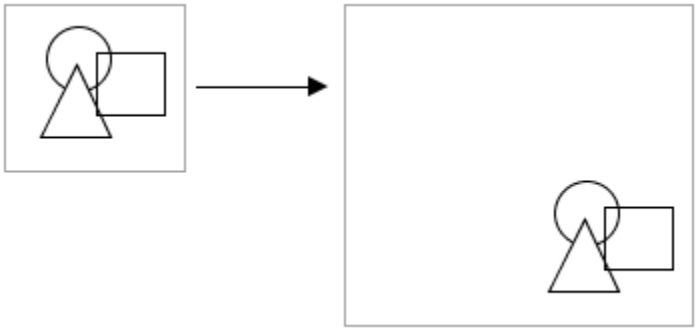
the container.



For example, if you anchor top and left, then your component will stay a constant distance from top and left edges of its parent. Since you didn't specify an anchor for the right or bottom sides, they won't be affected by the layout.

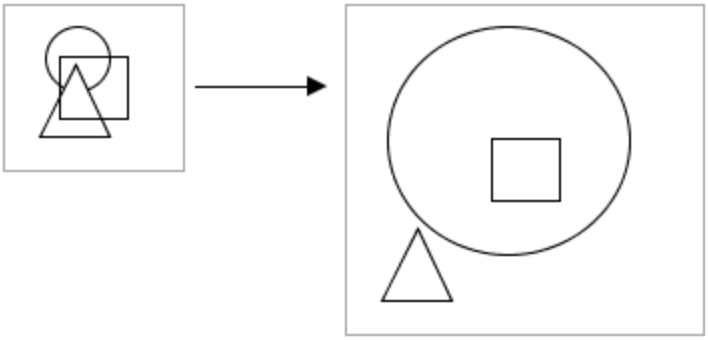


If you anchor bottom and right instead, the components will again stay the same size. Since you didn't specify an anchor for their other edges, but they will stay a constant distance from their parent's right and bottom edges.



Additionally, you can mix and match the various modes for the different components in a given container. To demonstrate, we can apply different anchors to each of the shapes at once and then change the size of the layout.

- Applying the horizontal and vertical centering anchors on the **square**, centers it in the layout and maintains the original size.
- Applying the south and west anchors on the **triangle**, holds it to the lower left area of the layout.
- Applying the north, south, west, and east anchors on the **circle**, causes it to expand as the edges of the layout expand.



In This Section ...

Creating Vision Components

Adding Components to a Window

There are four primary methods for adding Vision components to a window:

1. Select the component in the palette, and then clicking and dragging on the window.
2. Drag a component's icon from a palette onto a container.
3. Drag one or more Tags onto a window.
4. Adding shapes using the drawing tools or SVGs.

The Component Palette

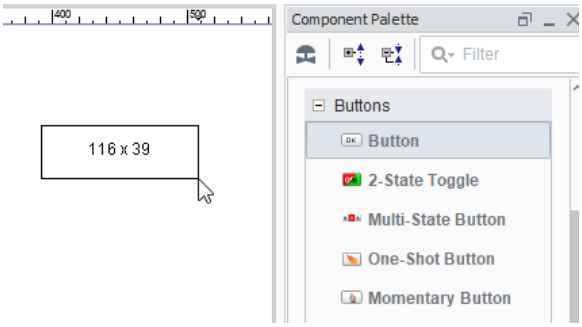
There are two styles of component palette in Ignition Vision: the tabbed palette and the collapsible palette. These palettes work in the same way, but the tabbed palette docks to the north or south edge of the workspace, and the collapsible palette docks to the east or west edge. By default, the collapsible palette is visible in the window workspace. To switch palettes, navigate to the **View > Panels** menu, and select either **Component Palette - Tabbed Palette** or **Component Palette - Collapsible Palette.**

On this page ...

- Adding Components to a Window
  - The Component Palette
- Creating Components Using Click and Drag
- Creating Components by Dragging from the Palette
- Creating Components Using Tags
- Creating Components Using Shapes
- Custom Palette

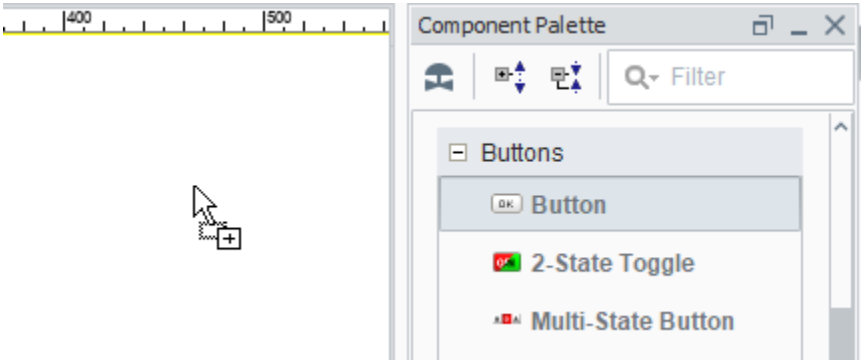
Creating Components Using Click and Drag

Components can be created on the window by first selecting them in the component palette, and then clicking and dragging on the window space. Draw a rectangle in the container to specify where the component should be placed and what size it should be.



Creating Components by Dragging from the Palette

Components can be created by dragging them from the component palette to the window. The component will be placed where they were dropped at its default size. Once on the window, the component can be resized using its resize handles.



Creating Components Using Tags

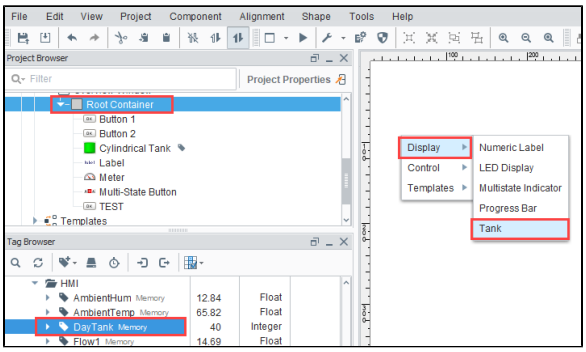
Components can also be created by simply dragging a Tag onto a container. Depending on the data type of the Tag, you will get a popup menu prompting you to select an appropriate type of component for that Tag. This technique is great for rapid application design as it does two things for you:

- The component is created at the position you dropped it.
- A variety of property bindings are created automatically.

Tags are used in windows to power property bindings on components. The easiest way to make some components that are bound to Tags is to drag and drop some Tags onto your window.

Tag Binding - Drag and Drop

Watch the Video



In the example above, we dragged the DayTank Memory Tag onto the window and were given the option of Display, Control, or Templates. Within the display components, we were given the option of displaying the tag in a Numeric Label, LED Display, Multistate Indicator, Progress Bar, or Tank component.

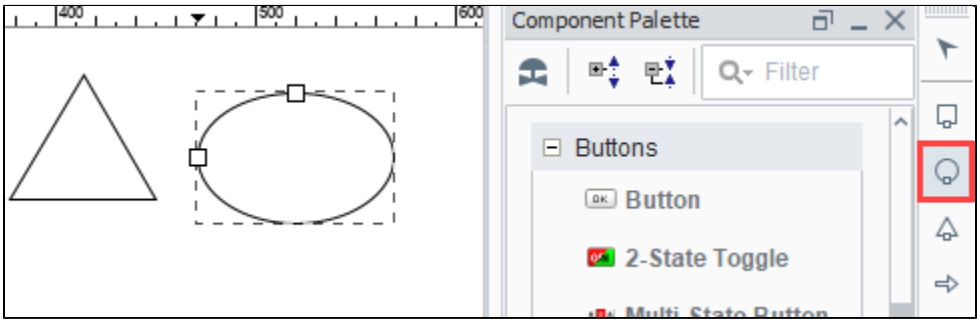
The bindings depend on what kind of Tag was dropped and what kind of component was created. For example, lets suppose you have a Float8 point that represents a setpoint, and you want to set it. Drop the Tag onto a container and choose to control it with a Numeric Text Field. The following bindings will be set up automatically:

- The text field's **doubleValue** property gets a bidirectional Tag binding to the Tag's **Value** propert y.
- The text field's **minimum** and **maximum** properties get Tag bindings to the Tag's **EngLow** and **EngHigh** properties, respectively.
- The text field's **decimalFormat** property gets a Tag binding to the Tag's **FormatString** property.
- The text field's **toolTipText** property gets a Tag binding to the Tag's **Tooltip** property.

It is important to realize that multiple property bindings are created when creating components this way. These bindings not only using the Tag's value, but much of the Tag's metadata as well. Using the Tags metadata in this way can greatly improve a project's maintainability. For example, if you decide that the setpoint needs 3 decimal places of precision, you can simply alter the Tag's **FormatString** to be `#,##0.000`, and anywhere you used that Tag will start displaying the correct precision because of the metadata bindings.

Creating Components Using Shapes

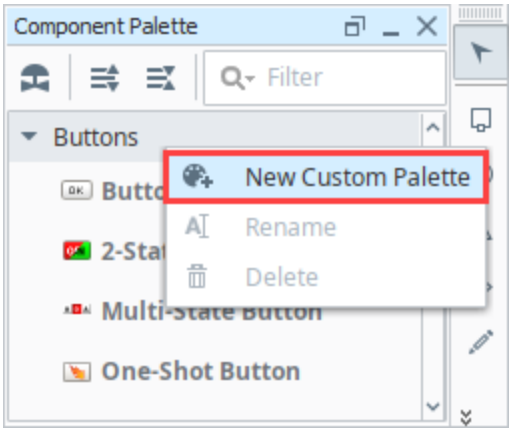
All of the shapes that you can draw using the shape tools are themselves components. As such, they have properties, event handlers, names, layout constraints, and all of the other things that you'll find on other components.



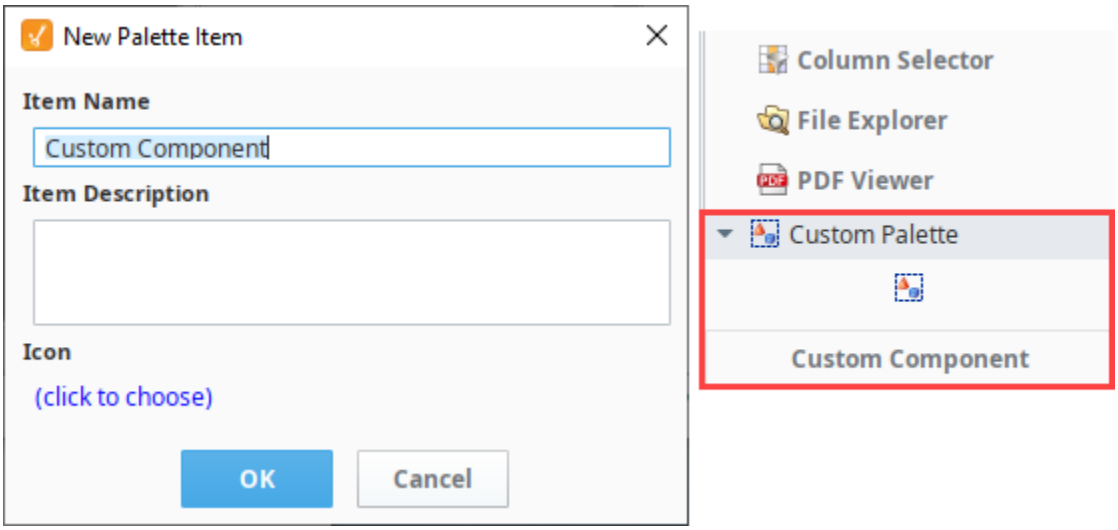
Custom Palette

Custom palettes are like expanded copy/paste clipboards. You can put customized components or groups of components into a palette for quick access.

To create a custom palette, right-click on a tab in the tabbed palette or a header in the collapsible palette, and choose New Custom Palette. Your custom palette will appear as the last palette. You can rename it by right clicking on the palette. Your custom palette has one special icon in it, the Capture  icon. To add components to your palette, select them and press the capture button. This effectively does a copy, and stores the captured components as a new item in the clipboard. You can then use that item much like a normal component, and add multiple copies of it to your windows.



You can assign your custom component a name and it will appear under the Custom Palette. Note that these are simple copies, and are not linked back to the custom palette. Re-capturing that palette item will not update all uses of that item across your windows.



Vision Component Customizers

The Vision module provides a number of customizers to configure components in ways that are more complex or detailed for basic properties.

The two main customizers are the Component Customizer and the Style Customizer. These two customizers are used repeatedly for many different components. For special purpose components like the Easy Chart, Table, Tab Strip, and Multi-State Button, they have their own special customizers for you to create your own custom properties.

On this page ...

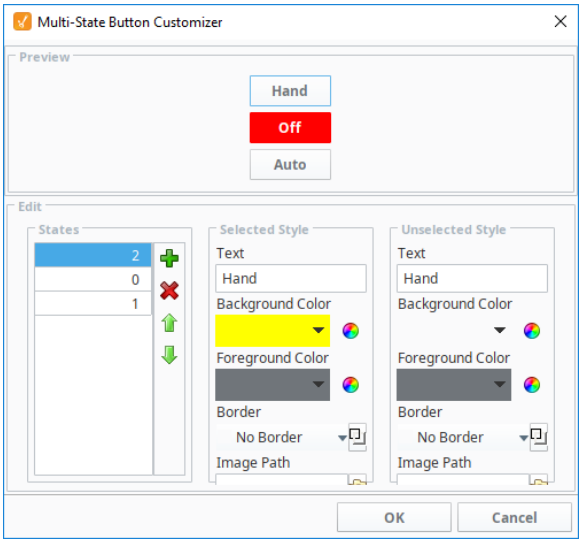
- Component Customizers
  - Custom Properties
- Style Customizer
  - Configuring the Style Customizer
  - Configuring Custom Properties
  - Value Conflict

Component Customizers

To use a customizer:

1. Right-click on the component.
2. Choose **Customizers.**
3. Select the desired Customizer.

You can also select the component and click the **Customizer** icon in the Vision Main Toolbar located on the title bar. The following image is an example of the Customizer for the Multi-State Button component.



Expert Tip

Often, a Customizer works as a user-friendly interface to one or more expert properties. For example, the Easy Chart Customizer modifies the contents of the pens, tagPens, calcPens, axes, and subplot dataset properties. This means you can also use property bindings and scripting to modify the values of these expert properties at runtime, giving you the ability to dynamically perform complex manipulations of components.

Custom Properties

In addition to the component's basic property settings, you can also create your own custom properties to enhance and add functionality to a component. You can use the custom properties like any other properties, such as with data binding, scripting, and styles. Custom properties are important for passing parameters from one window to another, especially with a popup window. Properties on the window's Root Container double as a window's parameters. For example, when you click on a Button component to open a popup window, it can pass a set of values into the window. These values are then set to the custom property on the Root Container for use on that window.

To configure a custom property:

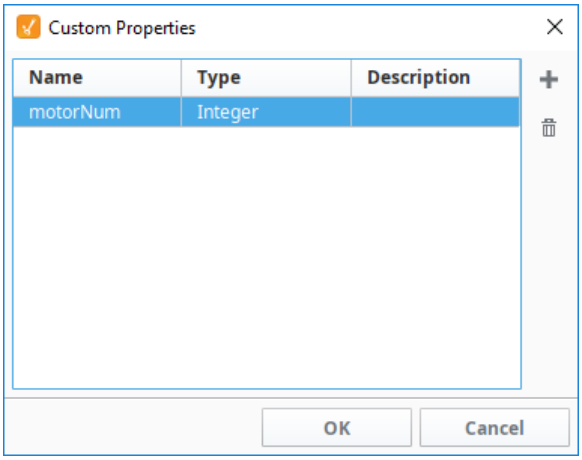
1. Right-click on the component.
2. Select **Customizers > Custom Properties**.
3. Click the plus icon to add a row.
4. Enter the **Name** (i.e., motorNum) of the custom property and data **Type**. Click **OK.**

INDUCTIVE UNIVERSITY

Custom Properties

Watch the Video

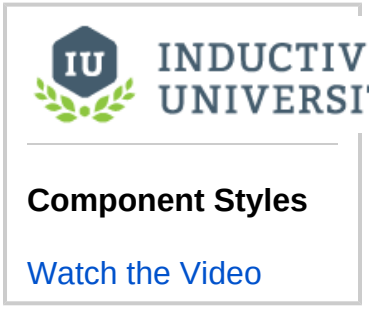
5. In the **Property Editor**, scroll to the bottom of the panel to see your custom property in blue.



Custom properties can be any of the basic property types, but can also be a UDT Definition. When specified, the UDT option will create a property shape that matches the shape of the UDT, allowing each member in to UDT to be represented as a separate property in the resulting custom property.

Style Customizer

Many components support the Style Customizer, which allows you to define a set of visual styles that change based on a single property. Typically, you'll have a property (often a custom property) on your component that you want to use as a driving property. The Style Customizer enables you to define many visual relationships at once, and allows you to preview them before implementing. Without the customizer, this would have to be done to each property individually.

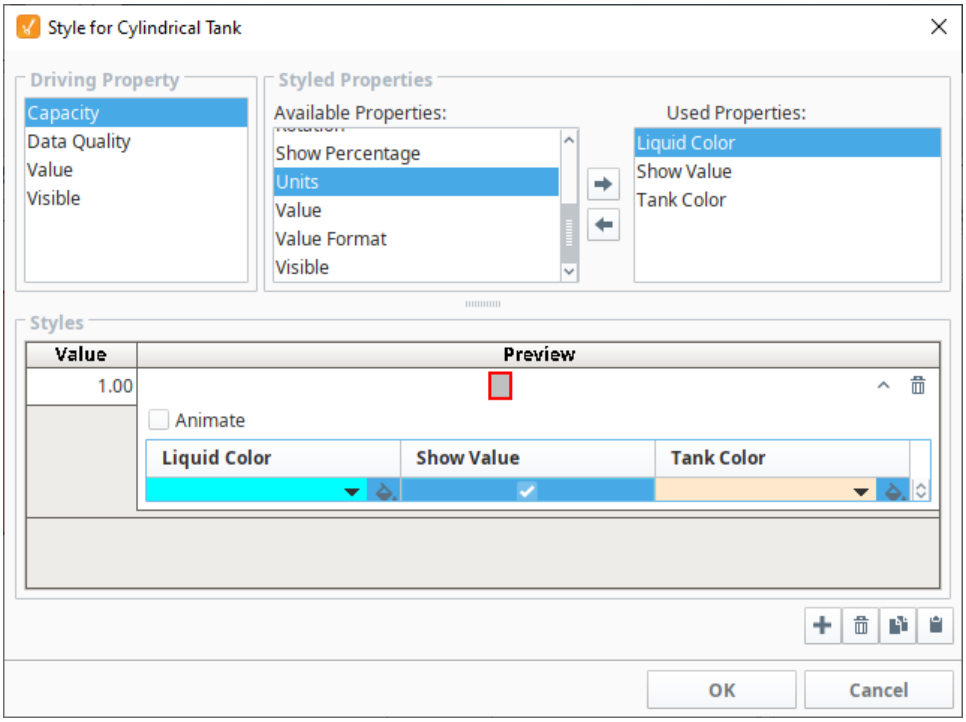


Configuring the Style Customizer

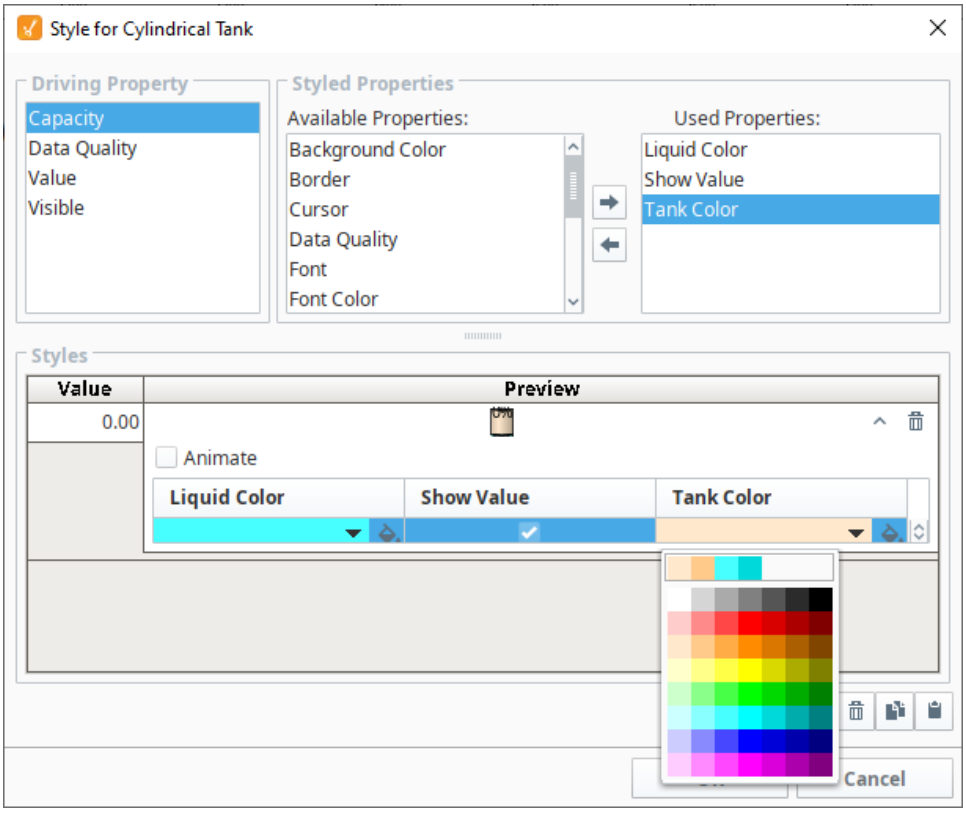
Some components have styles already set up, while others do not. The default styles help users get started, but these properties can be further modified if desired. The following example configures a Cylindrical Tank component that already has a style defined. There are four driving properties on the component where styles can be configured: Capacity, Data Quality, Value, and Visible.

1. Drag in a Cylindrical Tank from the component palette to the window.
2. Right-click on the Cylindrical Tank and select the **Style Customizer**.
3. Click on **Capacity** > **Liquid Color**.
4. Click the **Add Property** icon.

5. Repeat this step for the **Show Value** and **Tank Color** Properties.



6. Click the **Add** icon under Styles.
7. Click the **Expand** icon to see the color palette for the **Liquid Color**.

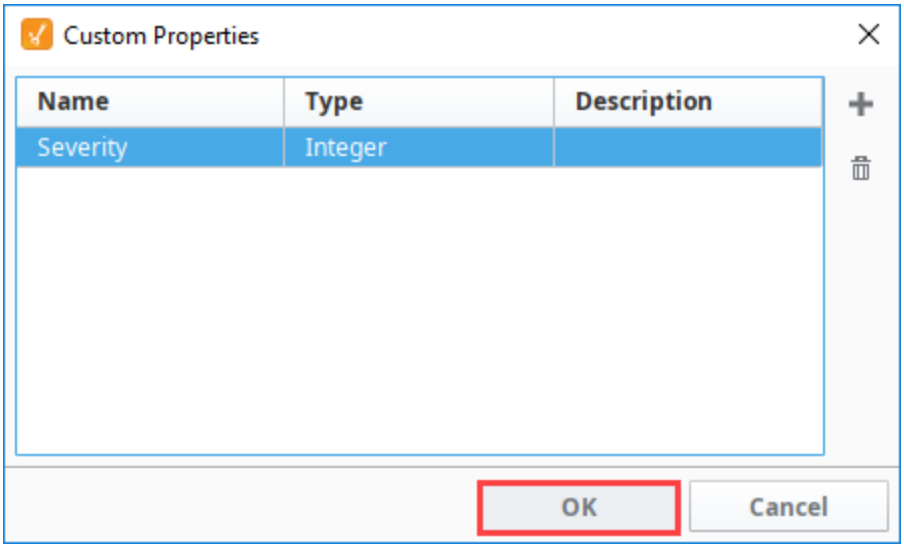


8. Choose a color from the palette.
9. Repeat this step for the **Show Value** and **Tank Color** Properties.
10. Click **OK** to save your updates.

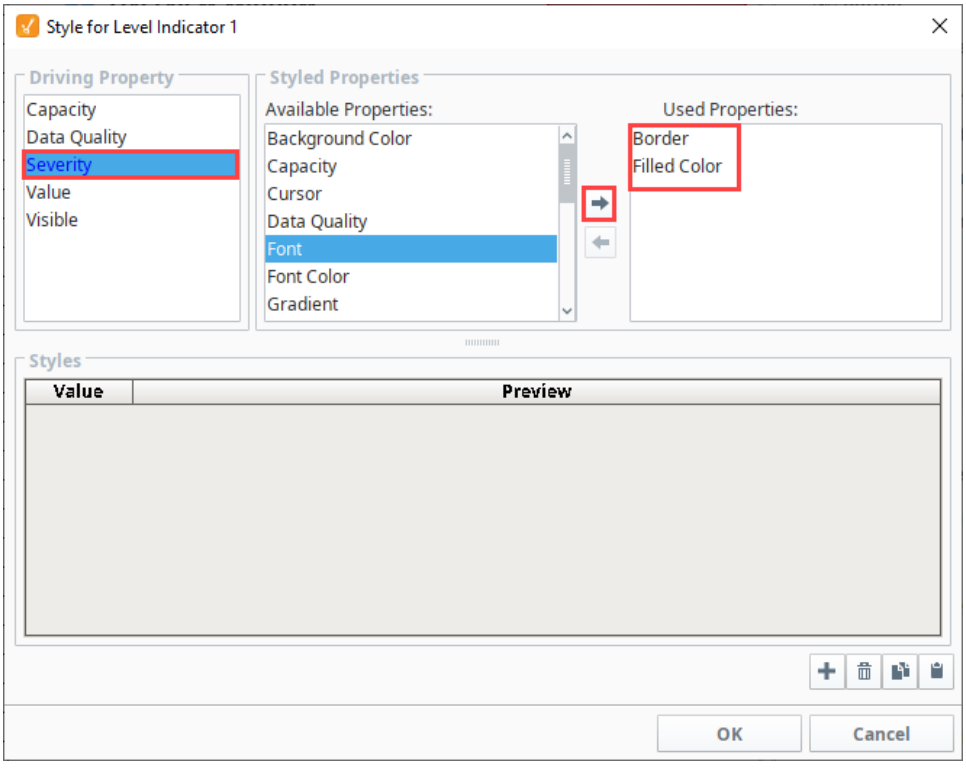
Configuring Custom Properties

Below is an example of how to use a custom property to configure the appearance of a Level Indicator component by changing the fill color based on the alarm state of the tank's temperature.

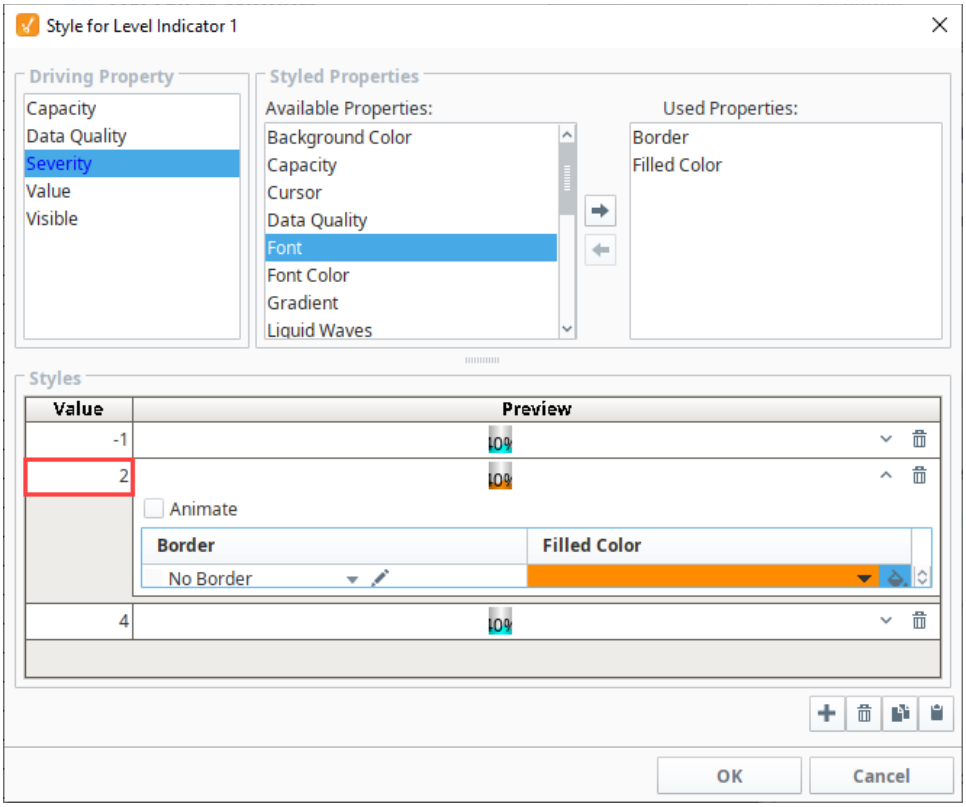
1. Add a Level Indicator and open **Custom Properties**.
2. Click the Add ✚ icon.
3. Name the new property **Severity** and set it to an Integer type. Click **OK**.



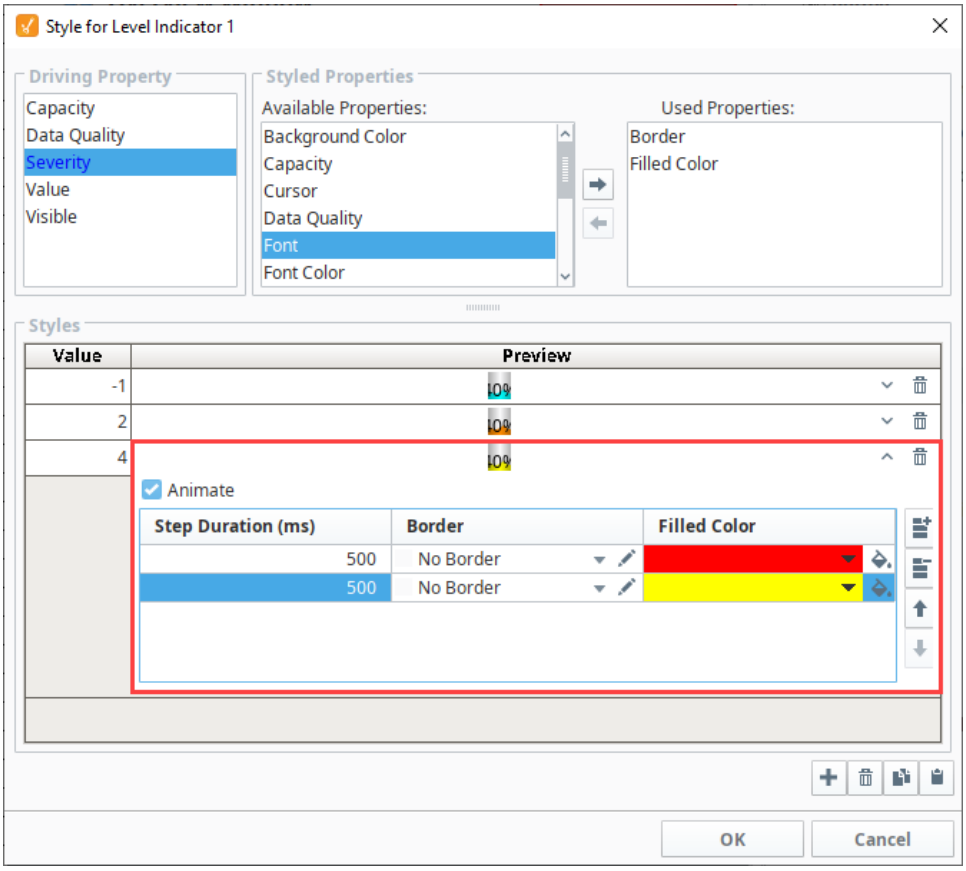
4. Right-click on the component and choose **Style Customizer**.
5. Choose your **Severity** property as the driving property, and the **Border** and **Filled Color** properties as the styled properties.



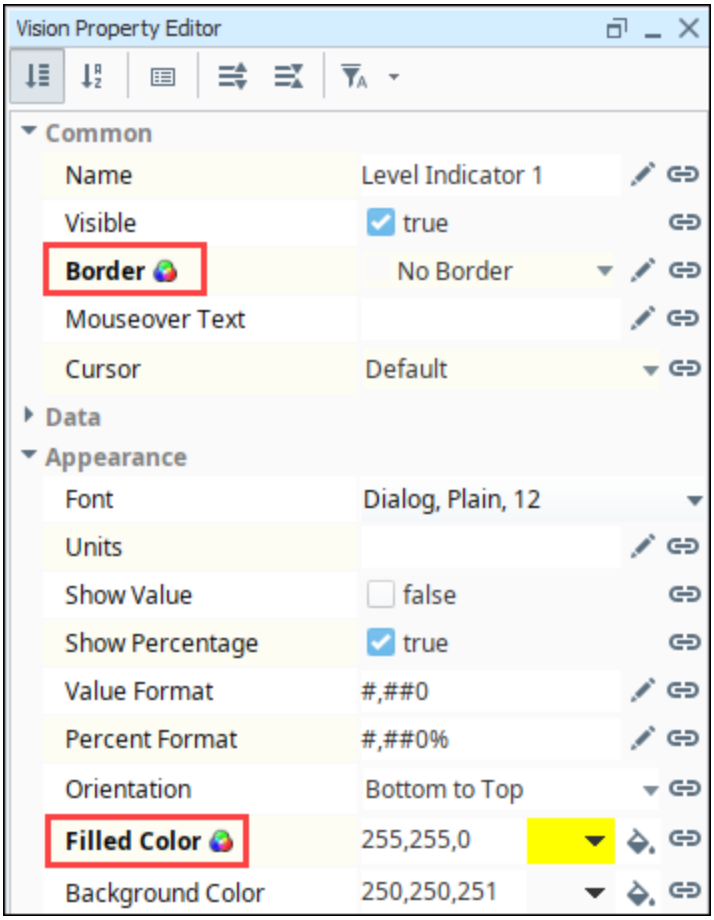
6. Under Styles, click the **Add** ✚ icon three times
7. Create three styles for the three alarm states you want to show.
8. For the first style, enter a value of -1 (not an alarm) and don't change anything else.
9. For the second, enter a value of 2 (medium alarm). Set the filled color to orange.



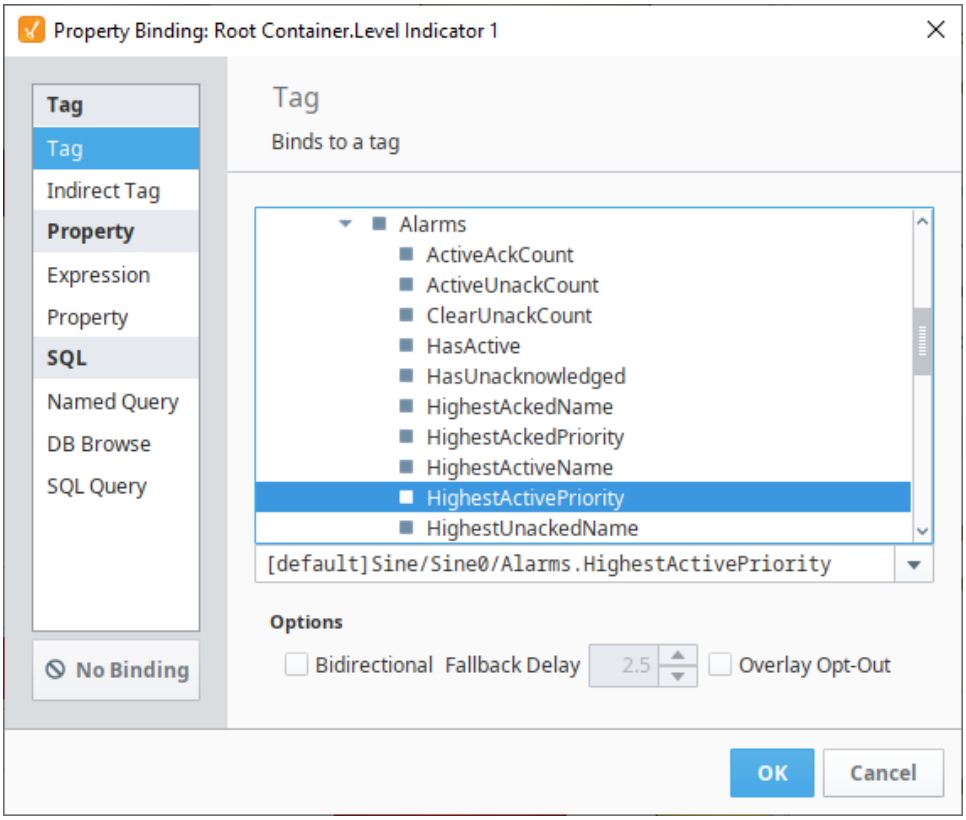
10. For the third style, enter a **Value** of 4 (high alarm).
    a. Click the **Expand** icon.
    b. Select the **Animate** checkbox.
    c. Click the **Add** icon.
    d. Set the **StepDuration** to 500 for both frames.
    e. For the first frame set the **FilledColor** to red.
    f. For the second frame, set the **FilledColor** to yellow.



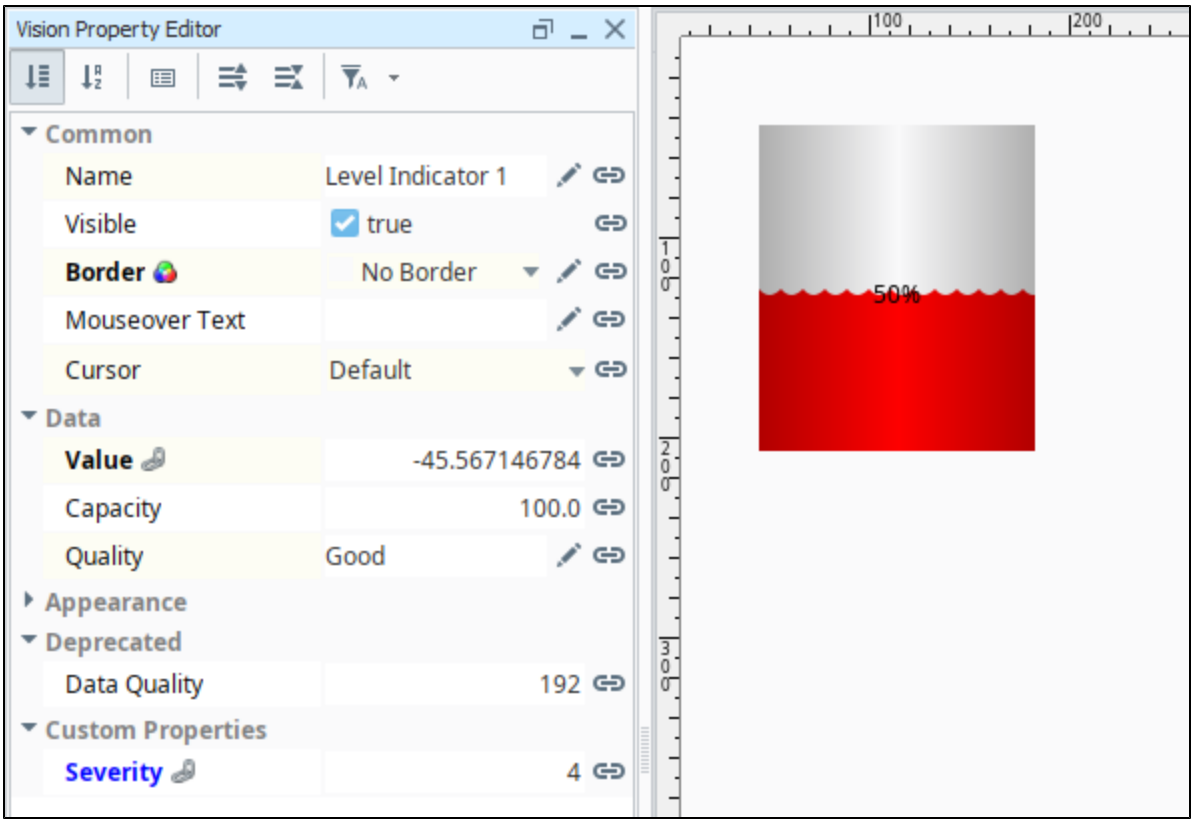
11. Click **OK.** Notice that the styled properties you chose are now bold and have the **Styles** icon next to them. This is to help remind you that those properties are being driven, so if you change their values directly, your changes will be overwritten.



12. In the Property Editor, click on the **Binding** icon for the **Severity** custom property.
13. Bind it to the tank temperature tag's **Alarms.HighestActivePriority** property.

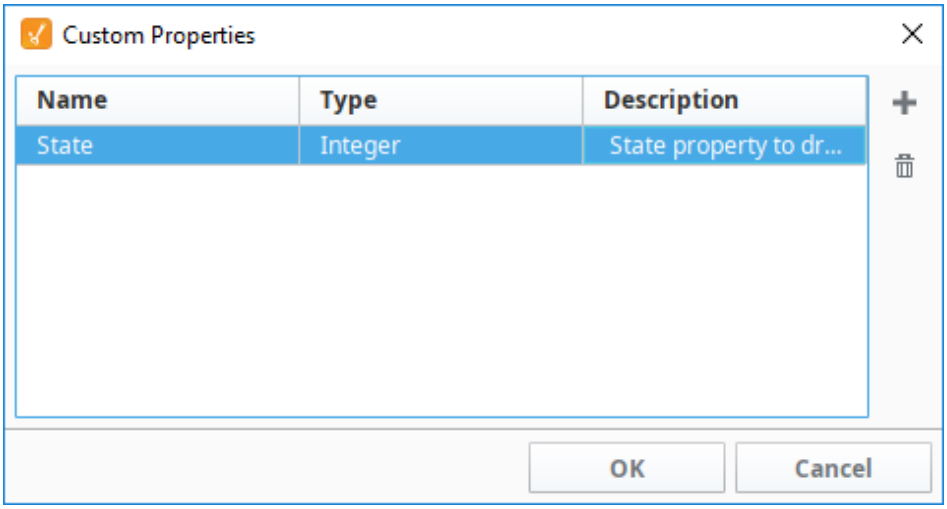


Now, when the alarm state for the tank's temperature changes, the color of the indicator will change based on the settings in the Style Customizer. For instance, the indicator will flash red and yellow if the high alarm is triggered.



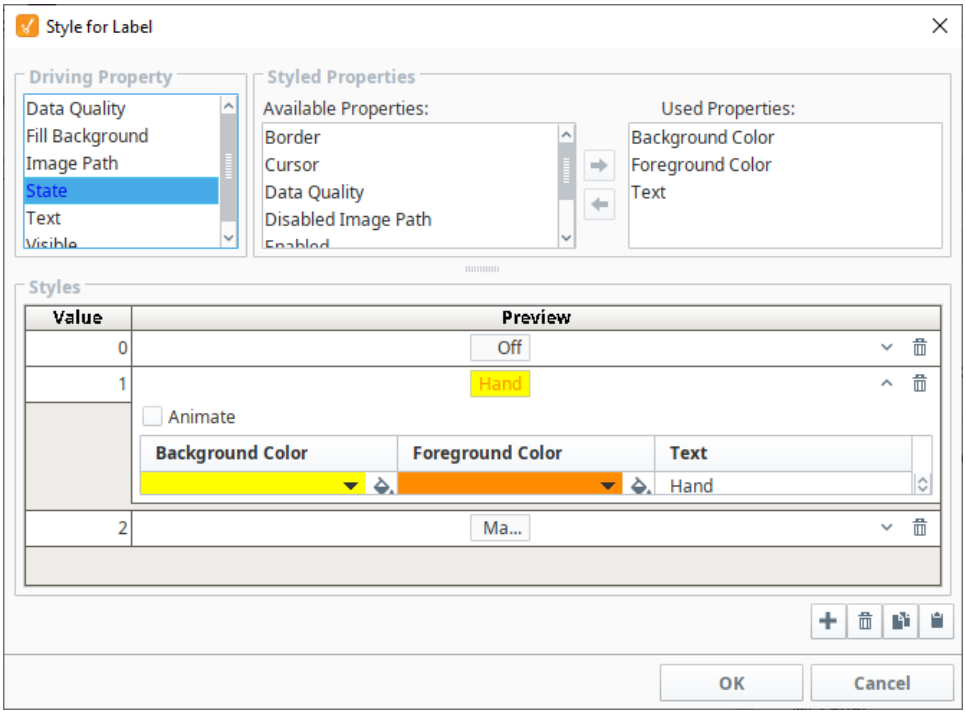
Below is another example to demonstrate the possibilities of customization by using Custom Properties and the Styles feature together to transform the Label component to appear more like the Multi-State Indicator. The Label component initially displays just a string, however you change the foreground color, background color, and border to make it even more functional.

1. Right-click on the Label component and choose **Custom Properties**.
2. Click the Add ✚ icon.
3. Name the new property **State** and set it to an **Integer** type. Click **OK**.

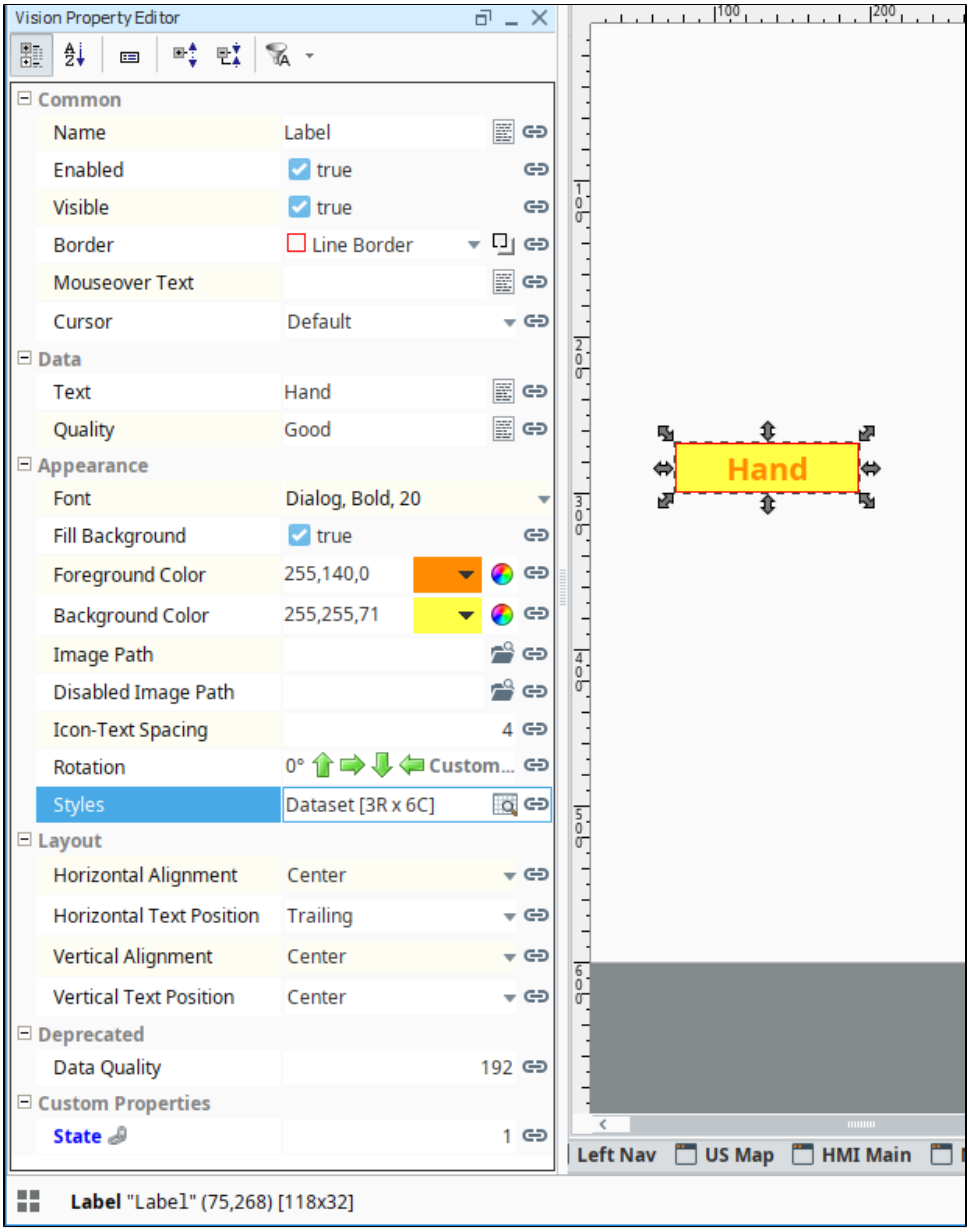


4. Bind that property to a discrete state Tag coming out of a PLC.
5. Next use the **State** property to drive its Styles configuration to make the component look different and display different text based on the value being 0, 1, or 2, like a Multi-State Indicator does. The Used Properties for this configuration include **Background Color**, **Foreground**

Color, and **Text**.

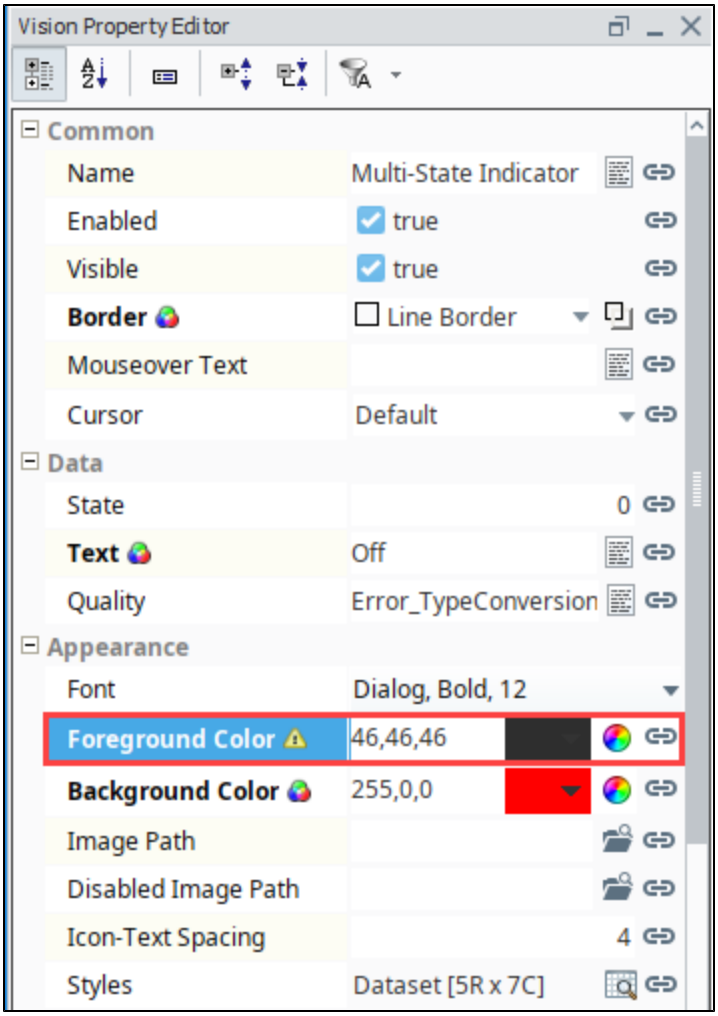


6. Click **OK** when all style selections are complete to apply them.



Value Conflict

You can bind a property that is already being used by a style, but a warning icon will appear on the property to indicate there is a conflict between the binding on the property and the style on the component. As a general practice, only the style or binding should write to the property, not both.


Vision Property Editor
Common
Name
Multi-State Indicator
Enabled
true
Visible
true
Border
Line Border
Mouseover Text
Cursor
Default
Data
State
0
Text
Off
Quality
Error_TypeConversion
Appearance
Font
Dialog, Bold, 12
Foreground Color
46,46,46
Background Color
255,0,0
Image Path
Disabled Image Path
Icon-Text Spacing
4
Styles
Dataset [5R x 7C]

Drawing Tools

Drawing Tools Overview

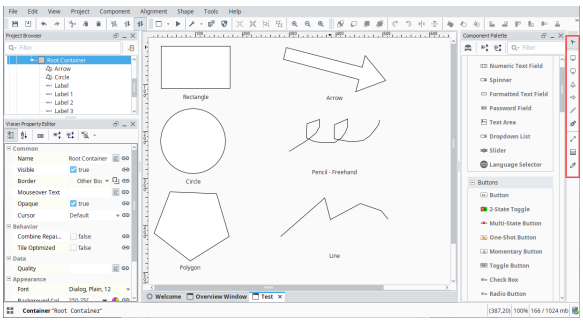
Vision comes with its own set of drawing tools so you can draw your own vector graphics on a window. Using the drawing tools you can create your own shapes such as lines, rectangles, and circles. These shapes are components with their own set of properties. Shapes or graphics such as lines, rectangles, and circles can be created using the 2D drawing tools in Vision. All SVG (Scalable Vector Graphics) images are importable in Vision and are made up of these basic shapes.

Using Drawing Tools

By default, the drawing toolbar is always located on the right side, but you can drag it to anywhere on your window that you prefer. At the very top of the toolbar is a **Selection** tool that allows you to select various components on a window. You can use the Selection tool to change the component's size and position as well as to configure the component. Below the Selection tool are all the tools that draw graphics. Click on the tool's icon to make it the active tool, then click in the Designer and drag to place the tool in your workspace. Once you draw a graphic, and want to drag a different graphic tool on to your window, click on the Selection tool. When a drawing tool is active, a toolbar will appear in the top menubar that has specific settings and actions for that tool.

Types of Drawing Tools

There are multiple drawing tools that each fulfill a different purpose. Some, like the selection tool simply allow you to select different components, while others like the rectangle tool allow you to create shapes. When you create a shape it has a default Fill Paint color of white. After a shape is created, you can change its Fill Paint color, Stroke color, and Stroke Style properties. All shapes can be treated as paths and be used with composite geometry functions to alter or create other shapes.

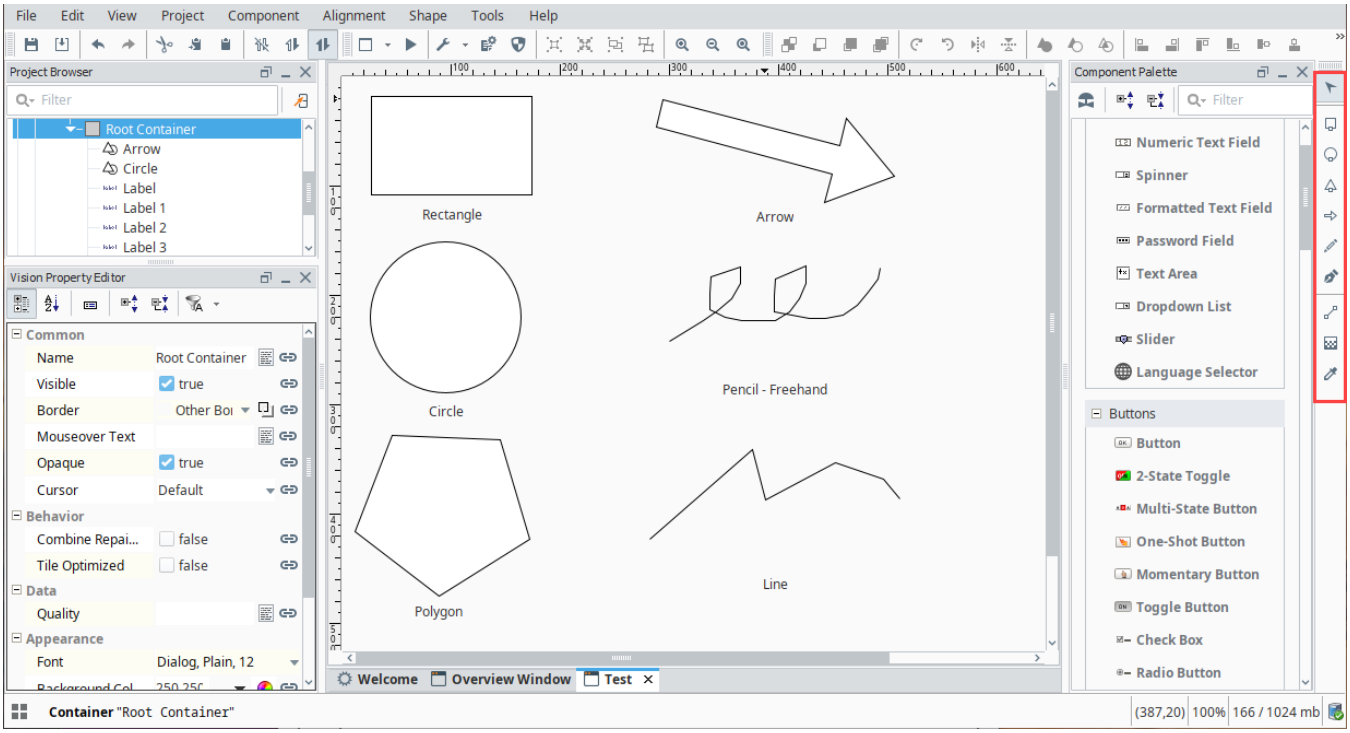


Selection Tool

The **Selection** tool is active by default. When this tool is active, you can select shapes and components. Selected components can be moved, resized, and rotated. For more on using the Selection tool to manipulate components and shapes, see Manipulating Components.

On this page ...

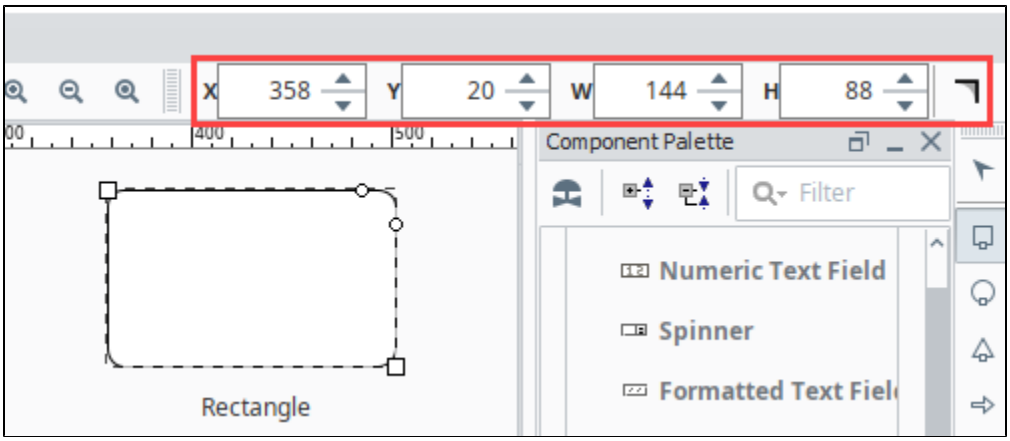
- Drawing Tools Overview
  - Using Drawing Tools
- Types of Drawing Tools
  - Selection Tool
  - Rectangle Tool
  - Circle Tool
  - Polygon Tool
  - Arrow Tool
  - Pencil Tool
  - Line Tool
  - Path Tool
  - Gradient Tool
  - Eyedropper Tool
- Shape Size, Position, and Angle



Rectangle Tool

The **Rectangle** tool creates and edits rectangle shapes. To create a rectangle, select the tool and then click and drag inside a window to create a new rectangle. Hold down **Ctrl** to make it a perfect square, and the **Shift** key to make it grow from the center point. Once a rectangle is created, you can use the square handles to change the rectangle's height and width. This is important because it is the only way to resize a rotated rectangle and let it remain a rectangle. If you resize a non-orthogonally rotated rectangle using the Selection tool, it will skew and become a parallelogram. If you double-click on the rectangle so the tool is active, you can change the rectangle's width and height using the tool-specific handles. From the toolbar, you can also change the rectangle's location (in pixels) on the window using the X and Y axes.

There are also small circle handles on the rectangle that allow you to alter the rectangle's corner rounding radius. Simply drag the circle down the side of the selected rectangle to make it a rounded rectangle. Hold down **Ctrl** to drag each rounding handle independently if you want non-symmetric corner rounding. You can use the **Make Straight** button in the rectangle's toolbar to return a rounded rectangle back to a standard, straight-corner rectangle.

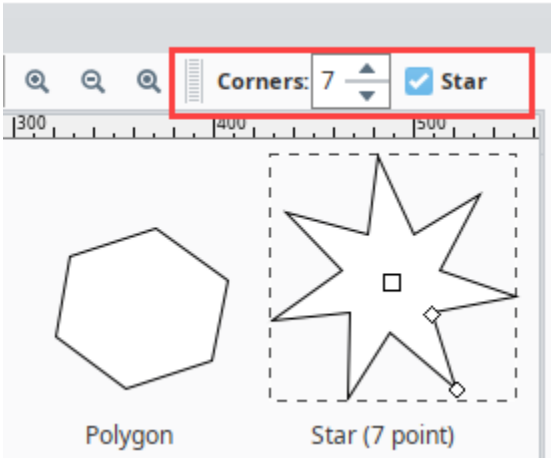


Circle Tool

The **Circle** tool creates and edits circles and ellipses. It is used in much the same way as the rectangle tool. While it is the active tool, you can click and drag inside a window to create a new ellipse. Hold down **Ctrl** to make it a perfect circle, and the **Shift** key to make it grow from the center point. When an ellipse is selected, use the width and height handles to alter the shape. You can also use the ellipse toolbar at the top of the Designer to change the width and the height as well as the X and Y axes.

Polygon Tool

The **Polygon** tool is used to create polygons and stars. Use the polygon toolbar at the top that becomes visible when this tool is active to alter the settings of the shape that is created when you drag to create a polygon. This tool can be used to make any polygon with three corners (a triangle) or more. On the Polygon menu you can specify the number of corners for the polygon. Once created, you can use the center square handle to move the polygon around, and the diamond handles to alter the size and angle of the polygon. Hold down **Ctrl** to keep the polygon's rotation an even multiple of 15°. For a star shape, specify the number of corners (points) and select the Star check box. A second handle that is between each corner will appear on the polygon allowing you to make a star shape.



Arrow Tool

The **Arrow** tool is used to create single or double-sided arrow shapes. When it is active, simply drag to create a new arrow. Use the checkbox on the arrow toolbar to choose a single or double-sided arrow. To alter the arrow, use the diamond handles to change the two ends of the arrow, and the circle handles to change the size of the shaft and the arrow head. When changing the arrow's direction, you may hold down **Ctrl** to snap the arrow to 15° increments.

Pencil Tool

The **Pencil** tool is used to draw freehand lines and shapes. When this tool is active, you can draw directly on a window by holding down the mouse button. Release the mouse button to end the path. If you stop drawing inside the small square that is placed at the shape's origin, then you will create a closed path, otherwise, you'll create an open path (line).

On the pencil toolbar, there are options for simplification and smoothing, as well as a toggle between creating straight line segments or curved line segments. The simplification parameter is a size in pixels that will be used to decrease the number of points used when creating the line. Points will be in general as far apart as this setting. If you find the line isn't accurate enough, decrease this setting. If you choose to create curved segments, then the segments between points will be Bézier curves instead of straight lines. The smoothing function controls how curvy these segments are allowed to get.

Line Tool

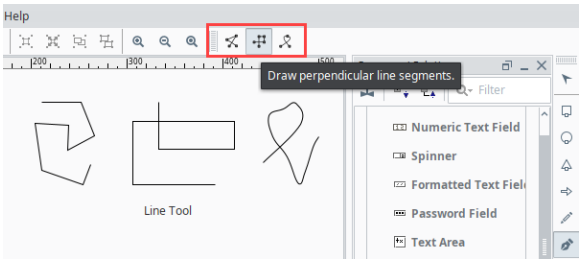
The **Line** tool can be used to draw lines, arbitrary polygons, or curved paths. Unlike all of the other tools, you don't drag to create new paths with the line tool. Instead, you click for each vertex you'd like to add to your path.

To draw a straight line, simply click once where you want the line to start, and double-click where you want the line to end. To make a multi-vertex path, click for each vertex and then double-click, press enter, or make a vertex inside the origin box to end the path.

As you draw the line, "locked-in" sections are drawn in green and the next segment is drawn in red. Hold down **Ctrl** at any time to snap the next segment to 15° increments.

On the line toolbar, you can choose between three different type of line settings:

- straight-line segments
- perpendicular-line segments
- curve-line segments

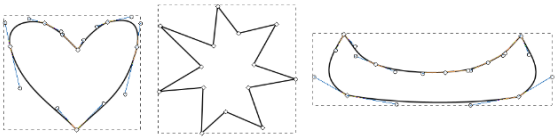


Perpendicular-line segment is just like a straight-line segment except that each segment is restricted to either horizontal or vertical.

The curve-line segment will create a Bézier curve path by attempting to draw a smooth curve between the previous two vertices and the new vertex.

Path Tool

All shapes and paths can be edited directly by using the **Path** tool. This tool lets you directly modify the nodes in the path, adding new nodes, removing nodes, and toggling segments between straight or curved. For more information, see Shape Geometry.



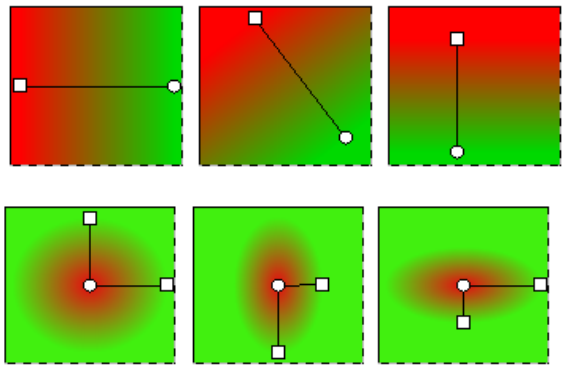
INDUCTIVE UNIVERSITY

Editing Shape Paths

Watch the Video

Gradient Tool

The **Gradient** tool is used to affect the orientation and length of any gradient paints. They work hand-in-hand with the Fill Paint property. Gradients smoothly blend any number of colors that can be positioned along a straight line or form an ellipse across the shape. A **Linear** gradient uses a horizontal line drawn across the width of the shape by default. By switching to the gradient tool, the horizontal line can be changed to move in any direction by dragging the handles. The **Radial** gradient uses a 45° angle drawn over the shape which starts at the center and moves out. Just like the Linear gradient, the Radial gradient can also be changed by dragging the handles around.



INDUCTIVE UNIVERSITY

Gradients

Watch the Video

Eyedropper Tool

With the **Eyedropper** tool you can set a selected shape(s) and/or component(s) foreground /background or stroke/fill colors by pulling the colors from somewhere else in the window. Select the component you want to change, and then activate the eyedropper tool. When this tool is active, left-click to set the selection's fill or background color, and right-click to set the selection's stroke or foreground color.

Remember to turn off the Eyedropper tool when you're finished by clicking the Selection tool, otherwise, you will continue to change colors on your component each time you do a mouse click. This tool works on most components as well as shapes. For example, right clicking will set the font color on a Button component, or left-clicking will set the background color.

INDUCTIVE UNIVERSITY

Eyedropper Tool

Watch the Video

Shape Size, Position, and Angle

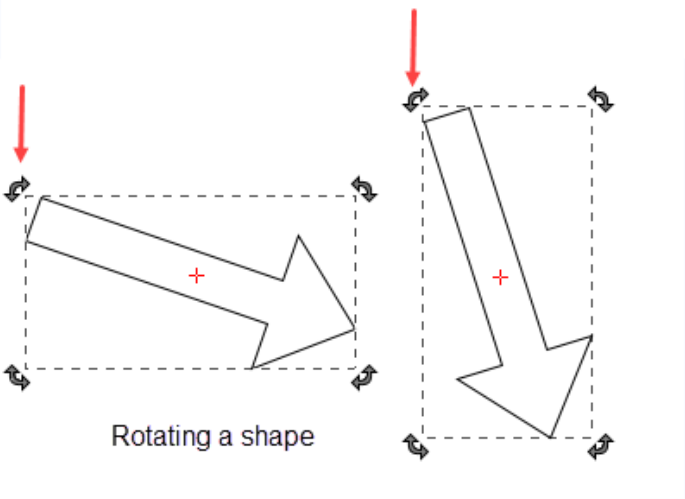
Shapes are different from other components in that they have properties that determine their size and position that can easily be bound. These properties are called X, Y, Height, and Width. The values of these properties are always relative to the shape's parent container's width and height, even in a running Client where that container may be a wildly different size due to the layout mechanism.

For example, let's say that you have a shape that is located at x=100, y=100, and was 125 by 125 inside a container that is 500 by 500. If you want to animate that shape so that it moves back and forth across the screen, you'd set up a binding so that X changed from 0 to 375. (You want X to max out at 375 so that the right-edge of the 125px wide shape aligns with the right edge of the 500px container). Now, at runtime, that container might be 1000 by 1000 on a user's large monitor. By binding X to go between 0 and 375, the true X value of your shape (whose width will now be 250px due to the relative layout system), will correctly move between 0 and 1750, giving you the same effect that you planned for in the Designer.

Another ability unique to shapes is the ability to be rotated. Simply click on a selected shape and the resize controls become rotate controls. There's even an Angle property that can be edited directly or bound to something dynamic like a Tag.

Note:

Use caution when binding the rotation. When you change a shape's rotation, its position also changes. The position of any shape is the top-leftmost corner of the rectangle that completely encloses the shape. Because of this effect, if you wish to both dynamically rotate and move a component, special care must be taken since rotation alters the position.



If you want to both dynamically rotate and move a component, special care must be taken since rotation alters the position. You don't want your position binding and the rotation binding fighting over the position of the component. The way to both rotate and move a shape is as follows:

1. Bind the rotation on your shape as you wish.
2. Create a shape (for example, a rectangle) that completely encloses (in other words, it's bigger than) your shape at any rotation angle.
3. Set that rectangle's visible property to false.
4. Select your shape and the rectangle and group them.
5. Bind the position on the resulting group.

Now the rotation and position of a shape are animated.

In This Section ...

Shape Geometry

Shape Paths

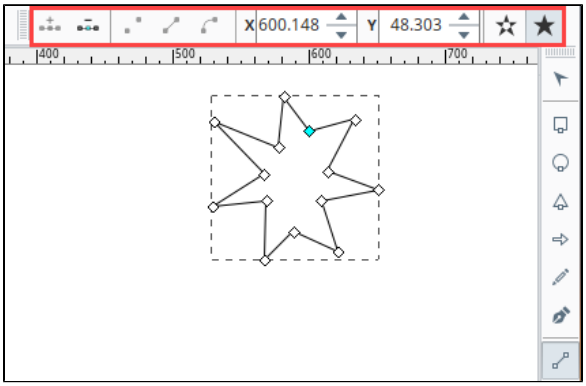
Once you draw shapes using the drawing tools in Vision, it's possible to alter and edit shapes after they've been created. By default, all shapes have a white fill color. Editing the paths of your vector graphic shapes is done by using the **Path** tool. Simply select any shape or line while the Path tool is active to start editing. If the shape is already a path, you can switch to the Path tool by double-clicking on the shape.

You can convert any shape into a general path by selecting the **To Path** function under the **Shape** menu. Shapes will also implicitly turn into paths if they are altered in a way not supported by the underlying shape. For example, if you stretch a rotated rectangle, thereby skewing it into a parallelogram, it will become a path automatically.

Editing a Shape Path

Each point on the path is represented by a diamond-shaped handle when the path editor is active. These handles can be dragged to move them around. They can also be selected by clicking on them or dragging a selection rectangle to select multiple points. This allows groups of points to be altered simultaneously.

To change a line segment between open, straight, and curved, select the **Path** tool and use the toolbar functions that become visible. Points can also be added and removed using the functions on the Path Editor toolbar.



Filling a Shape

Filled shapes have two fill settings that control whether or not holes in the shape should be filled. To remove the fill entirely, simply set the **Fill Paint** property in the Property Editor to **No Paint**.

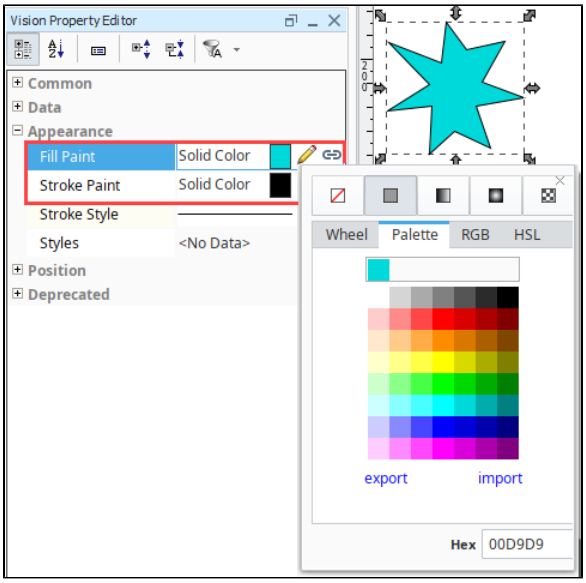
On this page ...

- Shape Paths
  - Editing a Shape Path
  - Filling a Shape
- Bézier Curve
  - Making Bézier Curves
- Creating and Editing Shapes Using Constructive Area Geometry
  - Union
  - Difference
  - Intersection
  - Exclusion
  - Division

INDUCTIVE UNIVERSITY

Shape Geometry

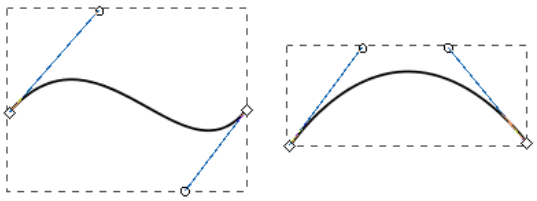
Watch the Video



When editing paths directly, it is often useful to zoom in on the path. Don't forget that you can zoom in on a location by holding down **Ctrl** an d using your mouse wheel to zoom in on a particular area without having to zoom in and then scroll. Also, if you press your mouse wheel in, you can pan around your window.

Bézier Curve

A **Bézier curve**, also sometimes called a quadratic curve, is a type of curved line used in vector graphics that connects two points, allowing you to create smooth vector graphic shapes. A Bézier curve is configured using four points: the two end-points and two control points. The curve starts along the line between the an endpoint and the first control point, and then curves to smoothly meet the line between the second control point and the next endpoint.



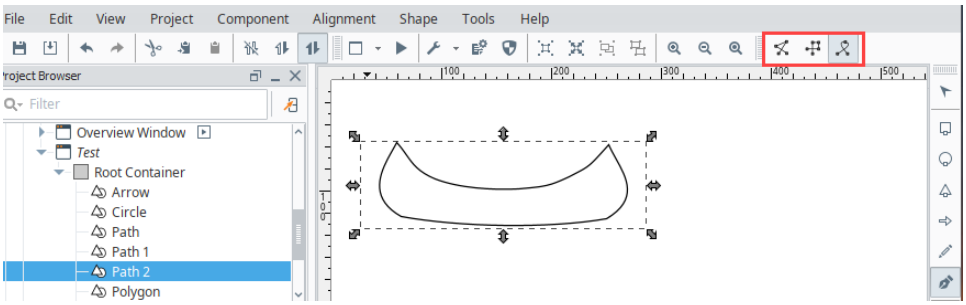
Bezier Curves

Watch the Video

Making Bézier Curves

Curves are made using the **Line** tool.

1. In the Designer, select the **Line** tool icon on the toolbar on the right side of your window. When the Line tool is active, a toolbar will appear in the top menubar. There are three different type of line settings: straight-line segments, perpendicular-line segments, and curved-line segments.
2. Select the curve-line segment on the menubar. Click on the window to begin drawing your image. Each time you want to make a curve, click on the screen. The curve-line segment will draw a smooth curve between points creating a smooth vector graphic shape.

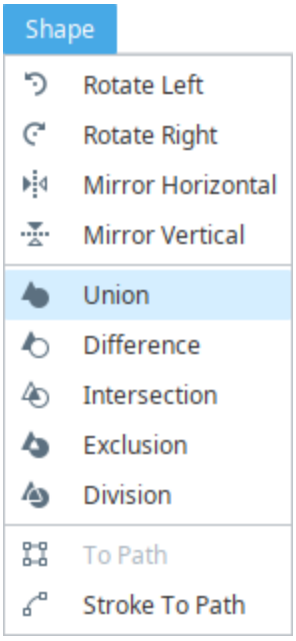


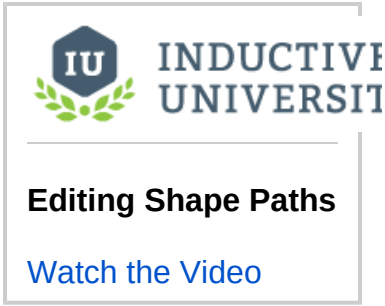
3. The line tool can make lines as well as shapes. To complete the line, simply click a second time on your final location. To make the line into a shape, click on the starting point as your final location.

4. Using the Path tool in the drawing toolbar, you can see a vector graphic shape showing where the points meet and the smooth curves between each point. If you want to edit or alter the shape after you create it, use the Path tool and drag the circles or diamonds to change the shape.

Creating and Editing Shapes Using Constructive Area Geometry

Editing paths directly can be a bit awkward. Using Constructive Area Geometry is usually an easier and more intuitive way to get the shape that you want. These functions are accessed from the **Shape** menu and operate when two (or more) shapes are selected.



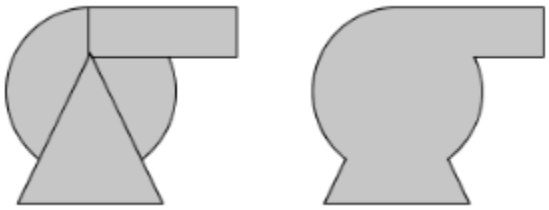


Selection Order Matters

The order that you select the shapes is important for many of these functions. Typically, the first shape you select is the shape you want to retain, and the second shape is the shape that you want to use as an "operator" on that first shape.

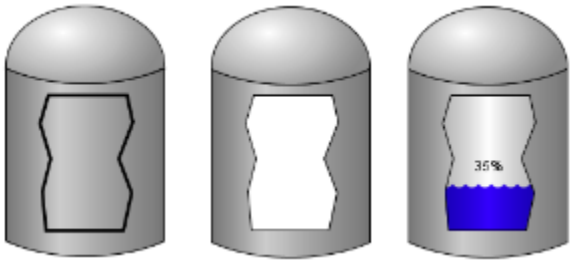
Union

The **Union** function combines two or more paths into one. The resulting shape will cover the area that any of the shapes covered initially. The example shows how the union of a circle, rectangle, and triangle can be "unioned" together to create a basic pump symbol. Creating the symbol using this method took a few seconds, whereas attempting to draw this shape by hand using paths would be quite frustrating.



Difference

The **Difference** function can be thought of as using one shape as a "hole-punch" to remove a section of another shape. The example shows how a zigzag shape drawn with the line tool can be used to punch a cutaway out of a basic tank shape. The level indicator is added behind the resulting shape to show how the area where the zigzag shape was is no longer part of the tank shape.

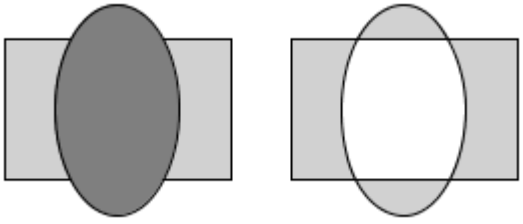


Intersection

The result of an **Intersection** function will be the area only where where two shapes overlap. The example shows how the "top" of the tank in the difference example was easily made using two ellipses.

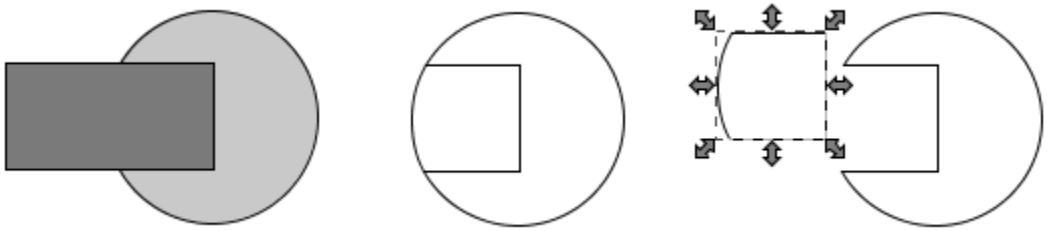
Exclusion

The **Exclusion** function, sometimes called X-OR, creates a shape that occupies the area covered by exactly one of the source shapes, but not both.



Division

The **Division** function divides or cuts one shape up along the outline of another shape. This works the same as a difference for the first shape selected, and an intersection for the second.



Fill and Stroke

All shapes have three properties that affect how they look: **Fill Paint**, **Stroke Paint**, and **Stroke Style**.

- **Fill Paint:** Determines the interior color of the shape.
- **Stroke Paint**: Represents the color of the shape's outline.
- **Stroke Style**: Determines the thickness, corners, and dash properties of the shape's outline.

On this page ...

- Fill Paint
  - Editing Paints
  - Paint Types
- Gradients
  - Gradient Paint Bounds
  - Gradient Cycles
  - Setting a Gradient
- Stroke Style

Fill Paint

Editing Paints

Both the Fill and Stroke paints can be a variety of different types of paints. To edit a shape's fill or stroke paint, you can either use the paint dropdown in the Property Editor table by clicking on the **Edit**  icon or open up the dedicated **Fill and Stroke** panel from the **View** menu.

Paint Types

The top of the paint editor is a selection area that allows you to choose between the five different types of paints.

1. **No Paint**  when used as a fill paint, then the interior of the shape will be transparent. If used as the stroke paint, then the paint's outline will not be drawn.
2. **Solid Color Paint**  is equivalent to the Color type used elsewhere throughout the component library. A solid color is any color, including an alpha (transparency) level.
3. **Linear gradient**  smoothly blends any number of colors along a straight line across the shape. Each color is called a Stop. Each stop is represented as a drag-able control on a horizontal preview of the gradient in the gradient editor. You can click on a stop to select it and change its color or drag it to reposition it. You can right-click on it to remove it. You can right-click on the preview strip to add new stops and change the gradient's cycle mode.
4. **Radial gradient**  are similar to linear paints except that the colors emanate from a point creating an ellipse of each hue. Radial paints are configured in the same way as linear paints.
5. **Pattern paint**  uses a repeating pixel-pattern with two different colors. You can pick a pattern from the dropdown or create your own using the built-in pattern editor.

Fill and Stroke

Watch the Video

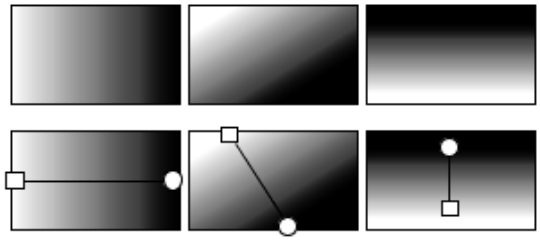
Gradients

Gradient Paint Bounds

The two gradient paints, **Linear** and **Radial**, are more than a list of colored stops, they also need to be placed relative to the shape. The same gradient may look wildly different depending on how it is placed against the shape. By default, a Linear gradient will run horizontally across the width of the entire shape, but this is readily changed. By switching to the **Gradient**  Tool located on the drawing tools toolbar, you can drag handles around to change the orientation of the gradient. You can even make the gradient larger or smaller depending on how big you want it to be.

Gradients

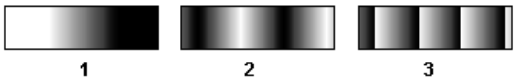
Watch the Video



Gradient Cycles

The two gradient paints (Linear and Radial) both have a cycle mode that you can change by right-clicking within the preview strip.

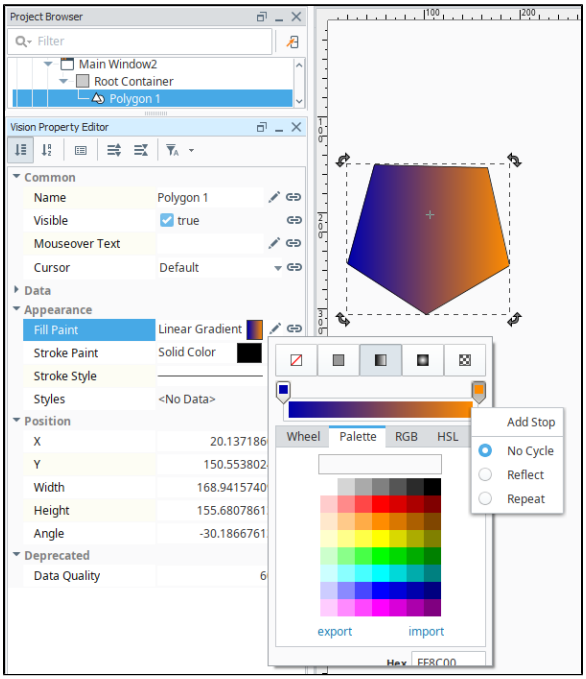
The cycle modes are illustrated below: **No Cycle, Reflect, and Repeat.**



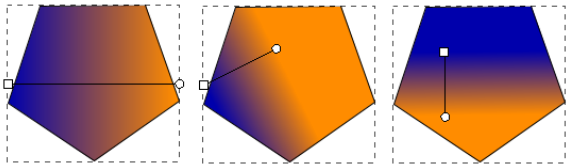
1. **No Cycle -** The first and last stops are repeated forever after the edge of the gradient bounds.
2. **Reflect -** Beyond the bounds of the gradient, it will be reflected and drawn in reverse, and then reflected again, creating a smooth repetition.
3. **Repeat -** Beyond the bounds of the gradient, it will be repeated forever.

Setting a Gradient

1. In the **Designer**, select your component.
2. In the **Property Editor** under **Appearance**, click the **Edit** icon.
3. Select either the **Linear** or **Radial** gradient.
4. You will see two stops: white and black. Click on each **Stop** and choose a different color.
5. If you want to add an additional stop, right click on the color bar and select Add stop. You can also add / remove **Stops**, and select your desired cycle: **No Cycle**, **Reflect** or **Repeat**.

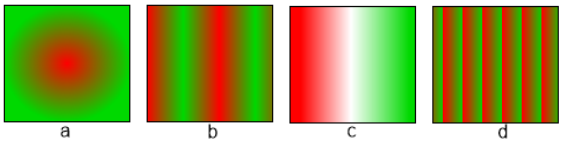


6. Close the Color Selector by clicking the X in the upper right corner.
7. With your component still selected on your window, and click the **Gradient** tool in the toolbar. You'll notice a line on your component. Now, you can drag the line's handles to change the orientation and lengthen or shorten the gradient.



8. Here are a few examples of what you can do with gradients:

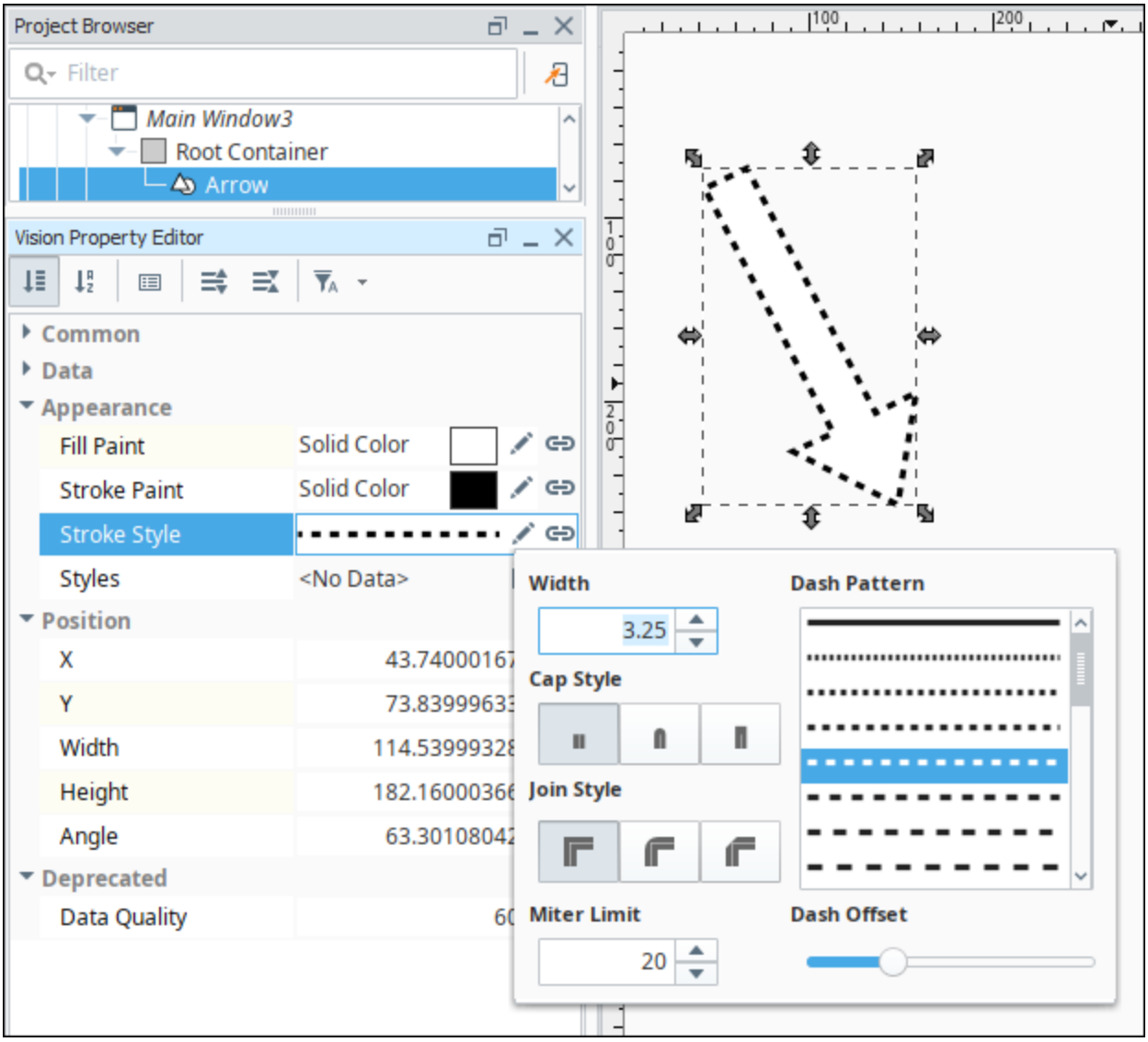
    a. **Radial** - No cycle
    b. **Linear** - Reflective
    c. **Linear** - with a 3rd Stop
    d. **Linear** - Repeat



Stroke Style

A shape's stroke paint is only half the story. The **Stroke Style** is also an important component of how an outline is drawn. Primarily the style controls the thickness of the line drawn, but it also can be used to create a dashed line. The setting for thickness is specified in pixels, and creating a dashed line is as easy as picking the style from the list. The effect of the thickness and dash pattern settings is fairly self-explanatory, but the other stroke settings are a bit more subtle. You can notice their effect more readily on thick lines.

You can access the **Stroke Style** in the Property Editor under **Appearance**.



Cap style is a setting that controls what happens at the end of a line segment. You can either have the line simply be terminated with no decoration (#1), round-off the end with a semi-circle (#2), or cap the end with a square (#3).

Join style is a setting that affects how a line is drawn where two segments meet (a corner). The default setting is called a miter join (#1), where the stroke is extended into a point to make 90-degree angle. The other options are rounded corners (#2) or beveled edge corners (#3).

Miter Limit style joins can become a problem for very sharp angles. With a sufficiently sharp angle, the miter decoration can become extremely long. To control this, there is a miter length setting to limit the length of a miter decoration. The illustration below shows the same miter join with two different miter length settings. The first drawing illustrates the length of the miter join.

Images and SVGs in Vision

Using SVGs

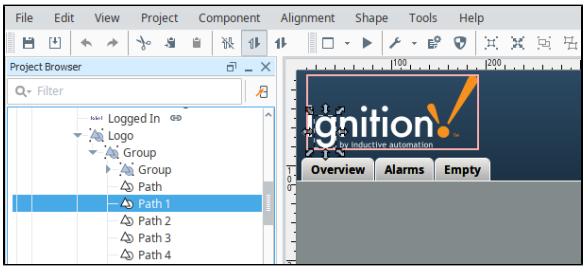
Ignition can import SVGs (Scalable Vector Graphic) into a Vision window. Once imported, SVGs can be modified and styled. To use an SVG in your project, simply drag the SVG file directly onto the window you want the SVG to appear on. The SVG becomes a new polygon component on the window.

Sometimes the way the SVG imports may result in the SVG appearing very small, in which case you can manually expand the SVG to your desired size.

Note: Some elements, attributes, and properties in an SVG are not supported. The Vision module uses the Apache Batik library to handle SVGs, so a list of supported elements, attributes, and properties can be found on Apache Batik's website.

SVGs as Grouped Components

All SVG images are made up of a group or groups of several (and often many!) paths. These paths are Ignition's Drawing Tools, and are the basic building blocks of all SVGs in Ignition. You can select each path individually from the Project Browser, or by double-clicking on an SVG then single clicking on an object inside it.



Coloring an SVG

With SVGs, one useful HMI technique is to color the SVG (Scalable Vector Graphic) to show the state of whatever the SVG represents. Whether you are bringing in a vector graphic from the Symbol Factory or importing an SVG from your computer, you can easily color the SVG to suit your needs. There are two ways of coloring an SVG: coloring an individual piece of the SVG, or placing a tint over the whole SVG.

Coloring SVG Parts

Individual pieces of the SVG can be pulled out and colored, by finding the path that corresponds with the part of the SVG that you want colored and applying a color to it. This can be done for a single piece of the SVG, or multiple different pieces. These colors can even be made dynamic by setting up a binding on the Fill Color. Since the property is expecting a value with a data type of **Color**, you can either set up an expression binding which uses the color() function to create a color object, or use Ignition's built in Number to Color Translator, which will automatically be made available to you when selecting a binding that will typically return a value, such as a Tag or Property binding.

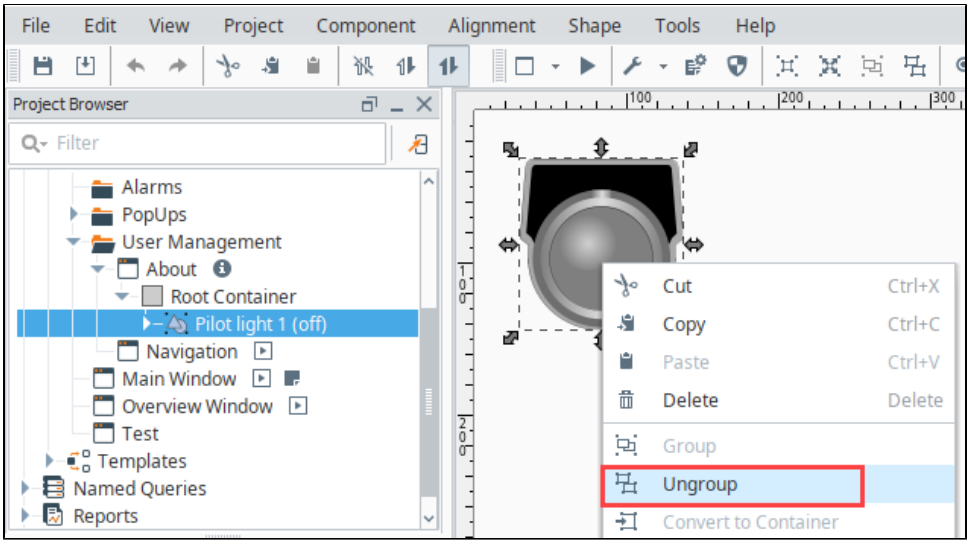
Coloring SVG Example

In this example, we selected an individual piece in a push button symbol and added a color to that area.

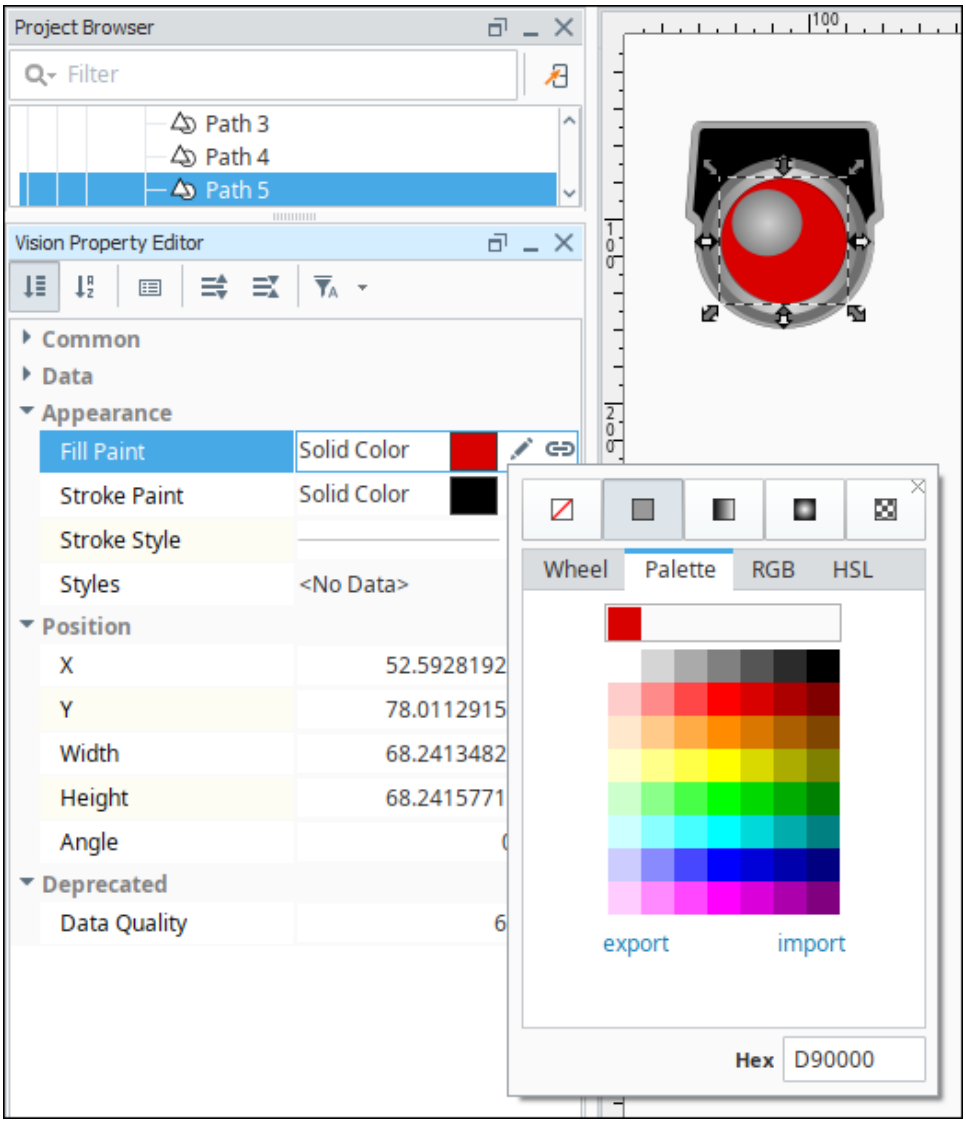
1. Place a push button image or any image from Symbol Factory onto the window.
2. Right-click on the image and select **Ungroup**.

On this page ...

- Using SVGs
  - SVGs as Grouped Components
- Coloring an SVG
  - Coloring SVG Parts
  - Coloring SVG Example
- SVG Tinting
  - Tinting Example
- Using Images
  - Using the Image Management Tool

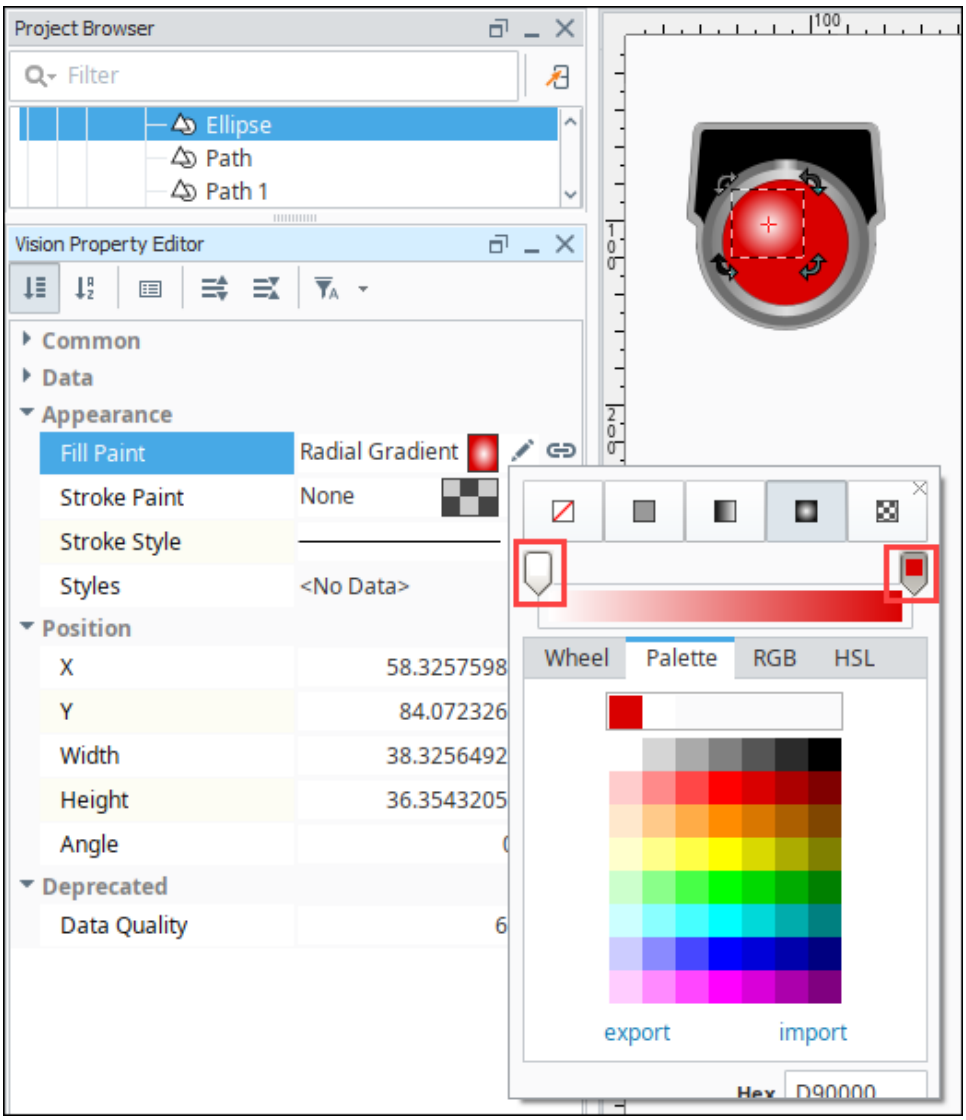


3. Right-click in the center of the image and select **Ungroup** again.
4. Click on the larger inner circle in the image, then select the Fill Paint color you want. We chose red #D90000.

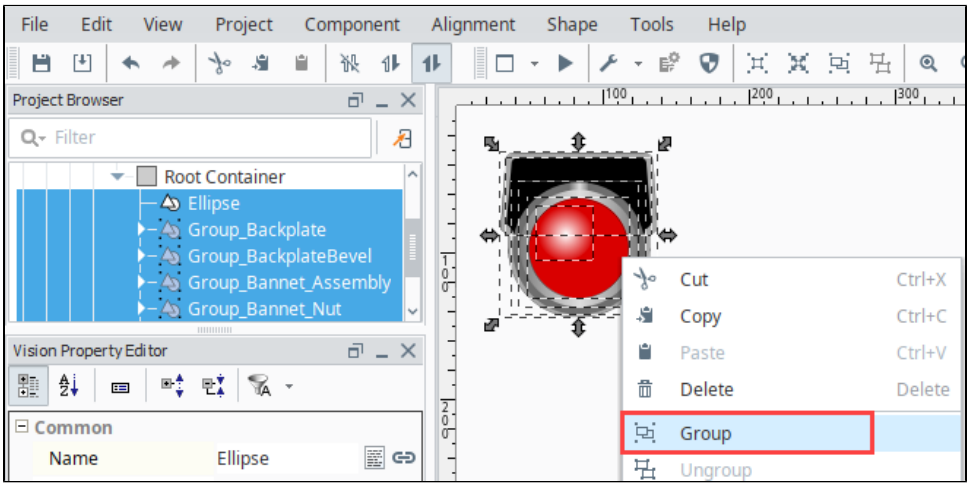


5. To get color in the "highlight" section of the graphic, click on the highlight. The Fill Paint for this section is a Radial Gradient.
6. Click on the **Edit** icon to change the colors in the gradient.

7. Click on the left side stop and select white. Click on the right side stop and select red.



8. To regroup all the parts of the SVG, drag a box around them with your mouse. Then right-click and select Group.



Alternately, you can create a style using the Style Customizer to change the color of an SVG path based on a driving property on that path. This is typically done by setting up a custom property to use as the driving property, and then binding the custom property to the property or Tag that will drive the color change.

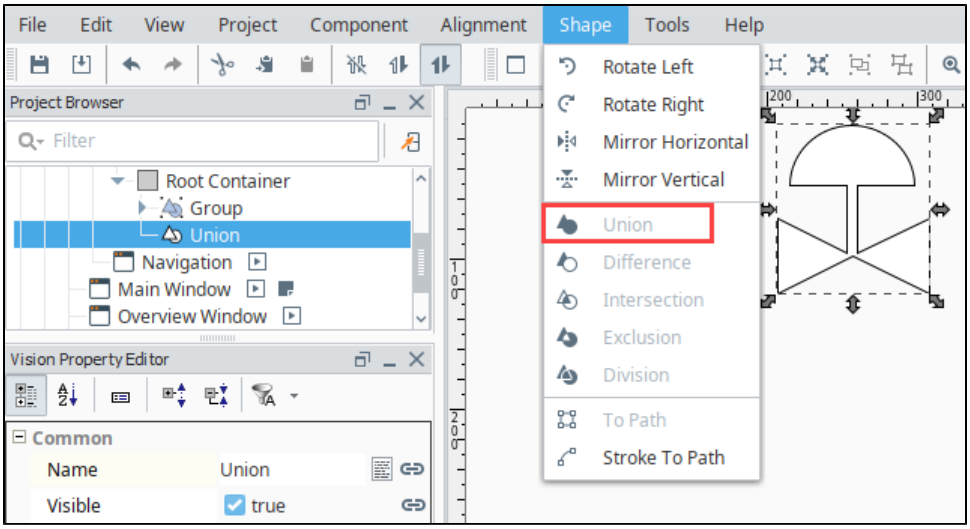
SVG Tinting

Because an SVG is typically composed of many smaller shapes, it is difficult to color the entire object, as it would require changing the color on every shape within the SVG. So instead of changing the color of the entire SVG, we can make a new shape that is the same shape and size as the SVG, and make it opaque, so that its color acts like a tint on top of the SVG. A clever way to do this is to duplicate the SVG, union together the new SVG, so that all shapes combine into one shape, and then change the color.

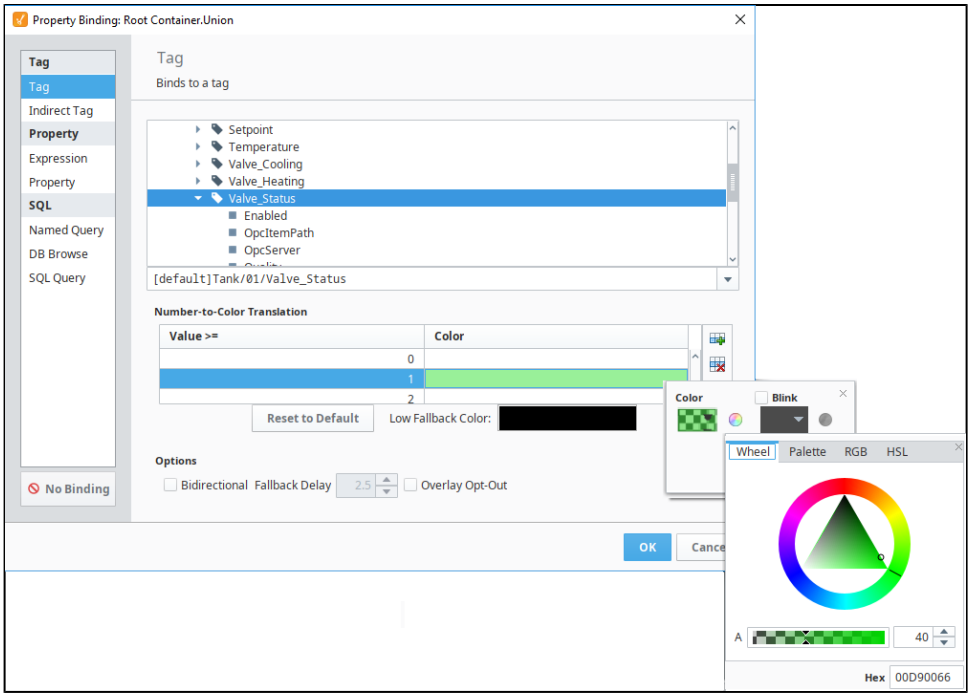
Tinting Example

Let's say you chose one of the many grayscale symbols, such as the **3-D Valve** symbol from the **Valves** category, and you want to tint the valve green when the valve is open, red when the valve has a fault, and keep it gray when the valve is closed. For this example, you'll also need to have a Tag called **ValveStatus**, that is 0 for closed, 1 for open, and 2 for faulted.

1. Drag the symbol onto the screen.
2. Duplicate the symbol by selecting it and choosing **Duplicate** from the **Edit** menu, or pressing **CTRL-D**.
3. Now, select the duplicate symbol, which will be above the original.
4. Click the **Union** icon in the toolbar or find the **Union** item under the **Shape** menu. This will combine the duplicate SVG into a single shape.



5. Remove the outline by setting the **Stroke Paint** property to **No Paint**.
6. Click on the **Binding** icon next to the Fill Paint property.
7. Select the **Tag** binding type.
8. Navigate to the Tag you want to use. For this example, we used a **Valve_Status** Tag.
9. In the Number-to-Color Translation, double click the color next to the Value 0 and select white.
10. Click the Add New Translation icon. Set the Value 1 at 40% opaque green.
11. Click the Add New Translation icon. Set the Value 2 at 40% opaque red.



12. Click **OK** to save the binding.
13. On the window, place another copy of the original symbol.
14. Select the colored symbol and select **Alignment > Move to Front**.
15. Next place the colored symbol on top of the original.
16. Select them both, then select **Component > Group**.

In summary, we made a flat shape that had the exact same outline as the symbol, and used semi-transparent fills to achieve a tint effect for the underlying symbol.

Using Images

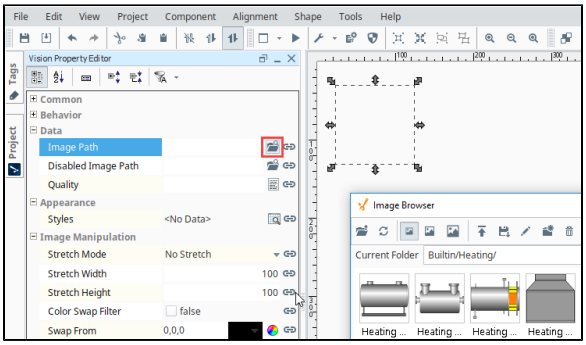
Images can be very useful for displaying important information, such as giving visual representations of real world objects. There are a few ways that images can be brought into a Vision project. The first is by pulling images from The Image Management Tool, where the images are stored in the Gateway. The other way is to grab images using a filepath.

Using the Image Management Tool

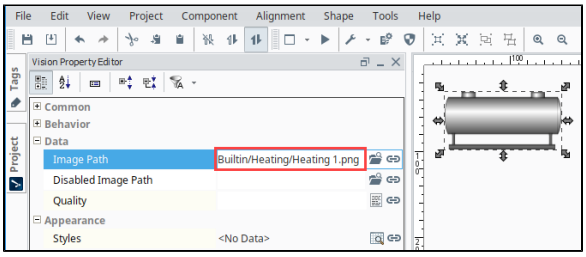
Bringing in images using the Image Management Tool is easy.

1. Place an Image component on the Window.
2. In the Vision Property Editor, scroll down to Data and click on the **Folder Search** icon next to the Image Path property. This will bring up the Image Management Tool.

3. In the Image Management Tool, find the image you want, and double click to select it.



The path to the image is now displayed in the components **Image Path** property. Images can be displayed in the image component, but they can also be used in components like labels and buttons.



Note: You can add any images you want to the Image Management Tool, which are stored on the Gateway.

Instead of using images in the Image Management Tool in Image Path properties of components, you can use the file path to a local image. This is done by prefixing the file path with `file:///`. An example Image Path would look like this:Using Local Images

```
file:///C:/Users/Public/Pictures/Sample Pictures/Chrysanthemum.jpg
```

It is important to understand that this will only work if the image is accessible from where the client is running. So if you access an image from the Designer on the local machine, clients that launch elsewhere may not have the image stored in the same location. For this reason, we recommend storing the images in a location that everyone can reach, such as a shared drive.

Note: When working with images found online, make sure to follow all applicable copyright laws.

Related Topics ...

- Symbol Factory
- Image Management Tool

Comparison Charts

Overview

This page provides an overview of the various comparison charts in Ignition, or charts that allow you to compare difference sets of data. Comparison charts differ from trending charts in that they tend to utilize a timestamp to visualize records over a period of time. Several types of comparison charts and how they are used are described on this page.

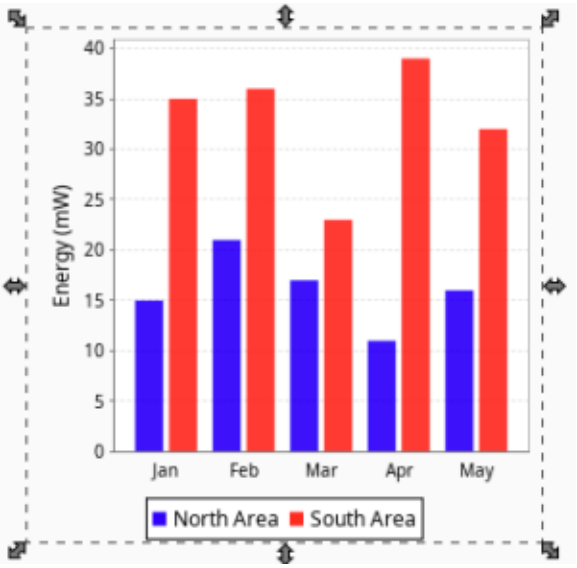
Bar Chart

The Bar Chart is an easy-to-use chart that is driven by the **Data** property, and expects a specific format. The first column in the Data property defines the names of the categories, and each additional column defines the groupings for each item in the series (depending on the Extract Order).

Note that additional datasets may not be added to the Bar Chart, so all values must be aggregated into the Data property via SQL Binding, or scripting. If multiple datasets are desired, then the Classic Chart configured with a bar renderer should be used instead.

Using the Bar Chart

Typically, data is pulled into the chart from a database using either a **Named Query** or **SQL Query** binding on the **Data** property. This data is typically category based, which typically means there is no timestamp. Generally, if values are split up by time, it is into large chunks of time, such as showing energy usage by month.

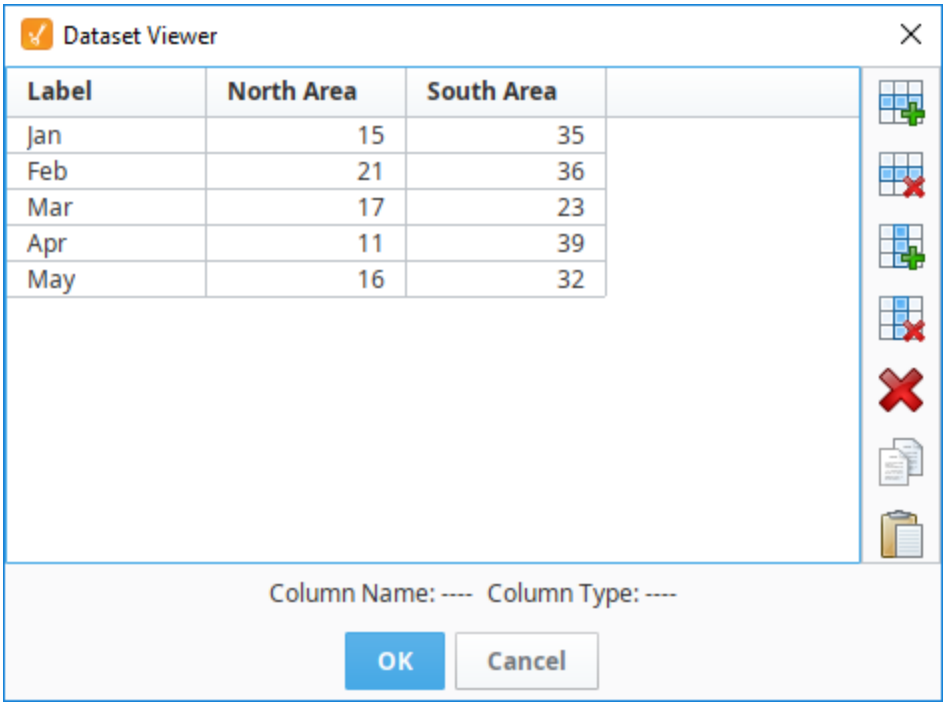


Initial Dataset

When a Bar Chart is first created, the component will contain a dataset that looks like the following:

On this page ...

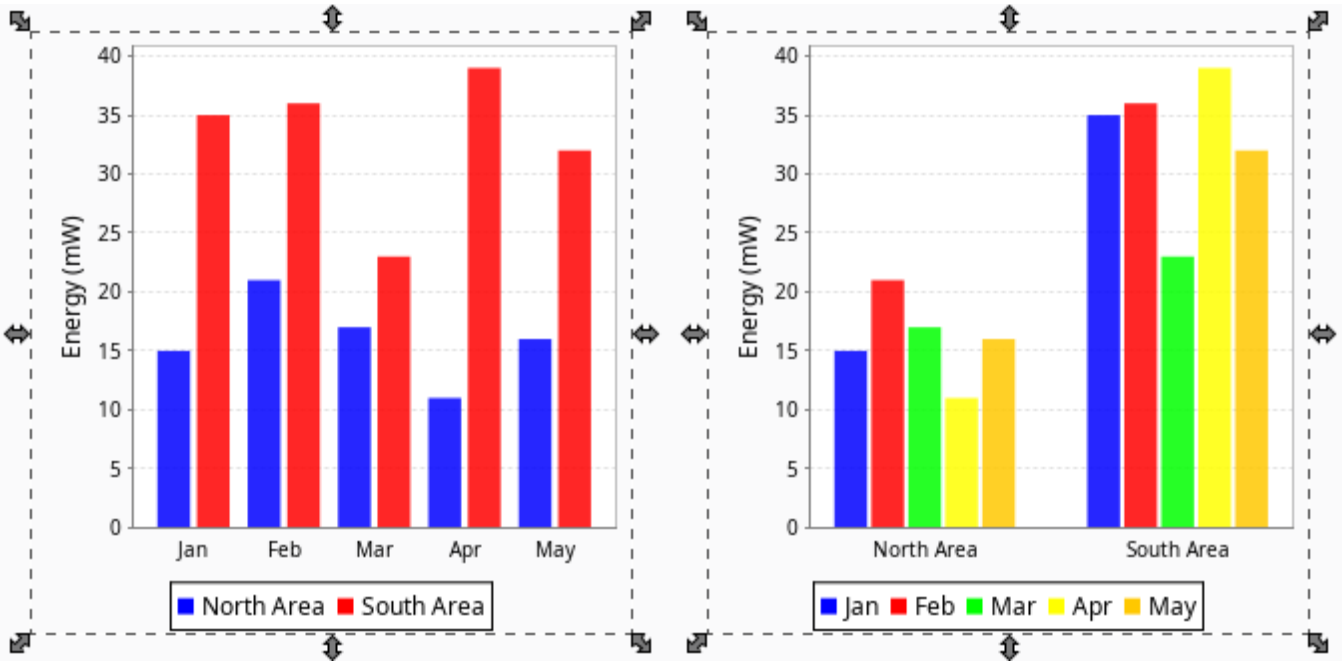
- Overview
- Bar Chart
  - Using the Bar Chart
  - Initial Dataset
  - Extract Order
- Chart
  - Using the Chart
  - Initial Dataset
- Radar Chart
  - Using the Radar Chart
  - Initial Dataset
  - Min and Max
- Pie Chart
  - Using the Pie Chart
  - Initial Dataset
  - Extract Order
- Box and Whisker Chart
  - Box Anatomy
  - Using the Box and Whisker Chart
  - Initial Dataset




```
"#NAMES"
"Label","North Area","South Area"
"#TYPES"
"str","I","I"
"#ROWS","5"
"Jan","15","35"
"Feb","21","36"
"Mar","17","23"
"Apr","11","39"
"May","16","32"
```

Extract Order

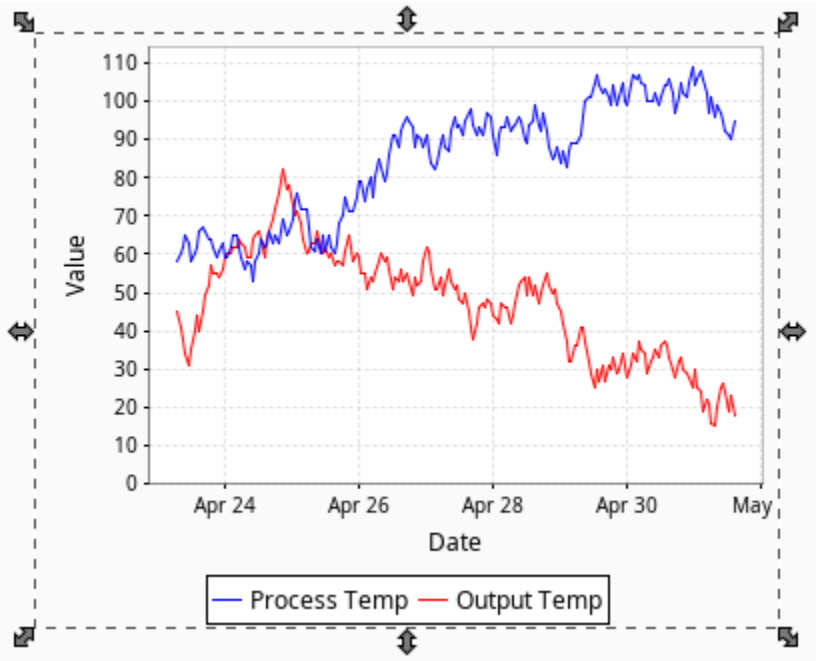
The **Extract Order** property on the chart determines how data series are defined. By default, the property is set to **By Row**, which means each row is a **series**, and each column (except the first) is a **category**. Based on the initial dataset, datapoints are grouped by area (column) and then grouped by each month (row). If we set the **Extract Order** property to **By Column**, then we see that each data point is grouped by the month (row), and then grouped by each area (column). Note that the underlying data has not changed, but rather how it is rendered.



Chart

The Chart component, also known as the Classic Chart can be used to create many different types of charts by rendering the data in different ways. This means that depending on the type of data you have, you may use the Chart component in very different ways.

By default the chart is set up to be used as a time series chart, with default data that shows this behavior. However with the right data, the chart can also display an XY coordinate plot or a categorical plot.



Using the Chart

The default settings on the Chart allow it to be used as a time series chart. Simply alter the dataset in the Data property with new time series data to display it in the chart. You can also alter the renderer in the Dataset Properties to any of the XY renderers to change the way the data is displayed.

To use the chart as an XY coordinate plot, data should be loaded with a two column dataset, where one column is the Y coordinate, and another is the corresponding X coordinate. The chart will also need to be set up with a new X axis, since the default axis is a date axis.

To use the Chart as a categorical plot, a few things need to change from the defaults. The Data property will need to be loaded with some categorical data. Categorical data will have one column of the dataset be categories of information in the form of a string. The chart will also need to be set up

with a new categorical X axis, as well as a category renderer in the Chart Customizer. Lastly, the Chart Type will need to be a Category Chart. When the chart is a Category Chart type, the Extract Order property can be changed to alter how the data is pulled out and displayed in the chart. It works very similarly to the Bar Charts Extract Order seen above.

Initial Dataset

Each new Chart randomly generates a new dataset. This Data property will use the default timeseries behavior of the chart, with a t_stamp column for the domain, and two other columns (Process Temp and Output Temp) as values at the specified times.

```
"#NAMES"
"t_stamp","Process Temp","Output Temp"
"#TYPES"
"date","I","I"
"#ROWS","200"
"2018-04-30 00:07:15","64","35"
"2018-04-30 01:07:15","60","35"
"2018-04-30 02:07:15","56","36"
"2018-04-30 03:07:15","52","31"
"2018-04-30 04:07:15","53","26"
"2018-04-30 05:07:15","57","28"
"2018-04-30 06:07:15","60","27"
"2018-04-30 07:07:15","57","26"
"2018-04-30 08:07:15","59","31"
"2018-04-30 09:07:15","57","36"
"2018-04-30 10:07:15","55","39"
"2018-04-30 11:07:15","52","41"
"2018-04-30 12:07:15","56","40"
"2018-04-30 13:07:15","51","41"
"2018-04-30 14:07:15","52","36"
"2018-04-30 15:07:15","53","32"
"2018-04-30 16:07:15","57","30"
"2018-04-30 17:07:15","52","32"
"2018-04-30 18:07:15","57","32"
"2018-04-30 19:07:15","55","29"
"2018-04-30 20:07:15","53","30"
"2018-04-30 21:07:15","54","31"
"2018-04-30 22:07:15","50","29"
"2018-04-30 23:07:15","54","25"
"2018-05-01 00:07:15","49","21"
"2018-05-01 01:07:15","53","21"
"2018-05-01 02:07:15","50","16"
"2018-05-01 03:07:15","51","19"
"2018-05-01 04:07:15","49","23"
"2018-05-01 05:07:15","48","25"
"2018-05-01 06:07:15","51","20"
"2018-05-01 07:07:15","55","18"
"2018-05-01 08:07:15","50","22"
"2018-05-01 09:07:15","49","26"
"2018-05-01 10:07:15","53","22"
"2018-05-01 11:07:15","50","27"
"2018-05-01 12:07:15","46","26"
"2018-05-01 13:07:15","48","27"
"2018-05-01 14:07:15","52","26"
"2018-05-01 15:07:15","51","24"
"2018-05-01 16:07:15","55","24"
"2018-05-01 17:07:15","58","23"
"2018-05-01 18:07:15","61","22"
"2018-05-01 19:07:15","60","27"
"2018-05-01 20:07:15","59","32"
"2018-05-01 21:07:15","60","33"
"2018-05-01 22:07:15","56","38"
"2018-05-01 23:07:15","51","35"
"2018-05-02 00:07:15","47","34"
"2018-05-02 01:07:15","45","32"
"2018-05-02 02:07:15","46","27"
"2018-05-02 03:07:15","43","31"
"2018-05-02 04:07:15","39","33"
"2018-05-02 05:07:15","39","30"
"2018-05-02 06:07:15","40","26"
"2018-05-02 07:07:15","41","27"
```

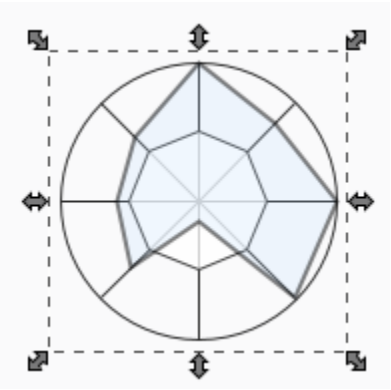
"2018-05-02 08:07:15","46","29"
"2018-05-02 09:07:15","43","25"
"2018-05-02 10:07:15","47","24"
"2018-05-02 11:07:15","49","19"
"2018-05-02 12:07:15","45","19"
"2018-05-02 13:07:15","42","20"
"2018-05-02 14:07:15","43","20"
"2018-05-02 15:07:15","43","23"
"2018-05-02 16:07:15","39","25"
"2018-05-02 17:07:15","37","22"
"2018-05-02 18:07:15","33","22"
"2018-05-02 19:07:15","31","21"
"2018-05-02 20:07:15","35","22"
"2018-05-02 21:07:15","34","21"
"2018-05-02 22:07:15","30","22"
"2018-05-02 23:07:15","34","27"
"2018-05-03 00:07:15","35","27"
"2018-05-03 01:07:15","36","32"
"2018-05-03 02:07:15","39","32"
"2018-05-03 03:07:15","39","33"
"2018-05-03 04:07:15","41","38"
"2018-05-03 05:07:15","40","35"
"2018-05-03 06:07:15","36","39"
"2018-05-03 07:07:15","38","41"
"2018-05-03 08:07:15","33","41"
"2018-05-03 09:07:15","30","38"
"2018-05-03 10:07:15","30","42"
"2018-05-03 11:07:15","33","38"
"2018-05-03 12:07:15","37","42"
"2018-05-03 13:07:15","33","37"
"2018-05-03 14:07:15","29","38"
"2018-05-03 15:07:15","24","37"
"2018-05-03 16:07:15","24","42"
"2018-05-03 17:07:15","21","45"
"2018-05-03 18:07:15","24","44"
"2018-05-03 19:07:15","28","49"
"2018-05-03 20:07:15","24","45"
"2018-05-03 21:07:15","24","49"
"2018-05-03 22:07:15","19","51"
"2018-05-03 23:07:15","24","48"
"2018-05-04 00:07:15","19","45"
"2018-05-04 01:07:15","16","44"
"2018-05-04 02:07:15","20","40"
"2018-05-04 03:07:15","25","38"
"2018-05-04 04:07:15","29","40"
"2018-05-04 05:07:15","27","36"
"2018-05-04 06:07:15","24","36"
"2018-05-04 07:07:15","29","41"
"2018-05-04 08:07:15","34","45"
"2018-05-04 09:07:15","37","47"
"2018-05-04 10:07:15","40","48"
"2018-05-04 11:07:15","42","52"
"2018-05-04 12:07:15","45","57"
"2018-05-04 13:07:15","46","58"
"2018-05-04 14:07:15","51","59"
"2018-05-04 15:07:15","46","56"
"2018-05-04 16:07:15","46","59"
"2018-05-04 17:07:15","47","56"
"2018-05-04 18:07:15","43","56"
"2018-05-04 19:07:15","46","53"
"2018-05-04 20:07:15","49","55"
"2018-05-04 21:07:15","51","51"
"2018-05-04 22:07:15","46","51"
"2018-05-04 23:07:15","50","50"
"2018-05-05 00:07:15","52","50"
"2018-05-05 01:07:15","51","51"
"2018-05-05 02:07:15","49","51"
"2018-05-05 03:07:15","46","55"
"2018-05-05 04:07:15","51","54"
"2018-05-05 05:07:15","56","52"
"2018-05-05 06:07:15","61","54"

"2018-05-05 07:07:15","62","52"
"2018-05-05 08:07:15","57","47"
"2018-05-05 09:07:15","54","47"
"2018-05-05 10:07:15","59","46"
"2018-05-05 11:07:15","56","44"
"2018-05-05 12:07:15","58","46"
"2018-05-05 13:07:15","62","44"
"2018-05-05 14:07:15","64","41"
"2018-05-05 15:07:15","62","45"
"2018-05-05 16:07:15","66","42"
"2018-05-05 17:07:15","61","37"
"2018-05-05 18:07:15","63","38"
"2018-05-05 19:07:15","61","38"
"2018-05-05 20:07:15","64","40"
"2018-05-05 21:07:15","64","44"
"2018-05-05 22:07:15","60","40"
"2018-05-05 23:07:15","64","44"
"2018-05-06 00:07:15","63","45"
"2018-05-06 01:07:15","61","47"
"2018-05-06 02:07:15","61","52"
"2018-05-06 03:07:15","61","48"
"2018-05-06 04:07:15","61","47"
"2018-05-06 05:07:15","60","46"
"2018-05-06 06:07:15","58","44"
"2018-05-06 07:07:15","56","43"
"2018-05-06 08:07:15","61","45"
"2018-05-06 09:07:15","66","49"
"2018-05-06 10:07:15","68","51"
"2018-05-06 11:07:15","63","54"
"2018-05-06 12:07:15","66","58"
"2018-05-06 13:07:15","69","63"
"2018-05-06 14:07:15","69","63"
"2018-05-06 15:07:15","67","58"
"2018-05-06 16:07:15","71","54"
"2018-05-06 17:07:15","74","50"
"2018-05-06 18:07:15","79","49"
"2018-05-06 19:07:15","75","51"
"2018-05-06 20:07:15","80","49"
"2018-05-06 21:07:15","79","50"
"2018-05-06 22:07:15","82","50"
"2018-05-06 23:07:15","80","53"
"2018-05-07 00:07:15","85","54"
"2018-05-07 01:07:15","87","49"
"2018-05-07 02:07:15","87","51"
"2018-05-07 03:07:15","84","56"
"2018-05-07 04:07:15","82","60"
"2018-05-07 05:07:15","81","57"
"2018-05-07 06:07:15","83","55"
"2018-05-07 07:07:15","83","55"
"2018-05-07 08:07:15","81","52"
"2018-05-07 09:07:15","77","49"
"2018-05-07 10:07:15","75","46"
"2018-05-07 11:07:15","79","45"
"2018-05-07 12:07:15","82","47"
"2018-05-07 13:07:15","81","48"
"2018-05-07 14:07:15","82","53"
"2018-05-07 15:07:15","81","48"
"2018-05-07 16:07:15","81","43"
"2018-05-07 17:07:15","85","40"
"2018-05-07 18:07:15","90","37"
"2018-05-07 19:07:15","94","34"
"2018-05-07 20:07:15","90","38"
"2018-05-07 21:07:15","89","40"
"2018-05-07 22:07:15","85","45"
"2018-05-07 23:07:15","81","48"
"2018-05-08 00:07:15","83","43"
"2018-05-08 01:07:15","78","42"
"2018-05-08 02:07:15","73","40"
"2018-05-08 03:07:15","72","44"
"2018-05-08 04:07:15","71","42"
"2018-05-08 05:07:15","75","44"

```
"2018-05-08 06:07:15","71","46"
"2018-05-08 07:07:15","67","50"
```

Radar Chart

Radar Charts, also known as web charts, spider charts, and spider plots, are useful for displaying values that are out of spec, and several of them at once. Each value is plotted on a separate axis with the middle of the axis representing the ideal value. The chart draws a line between the different values, which create a shape that changes as those values change. Inside the chart, there is a polygon that represents what the chart would look like if all of its values were in their ideal range. A good use of radar charts is to display realtime information in such a way that outliers can be quickly identified. This can be an efficient way to convey if a process is running on-spec or off-spec at a glance. So the Radar Chart lets you quickly see where the values are in comparison to where they should ideally be.

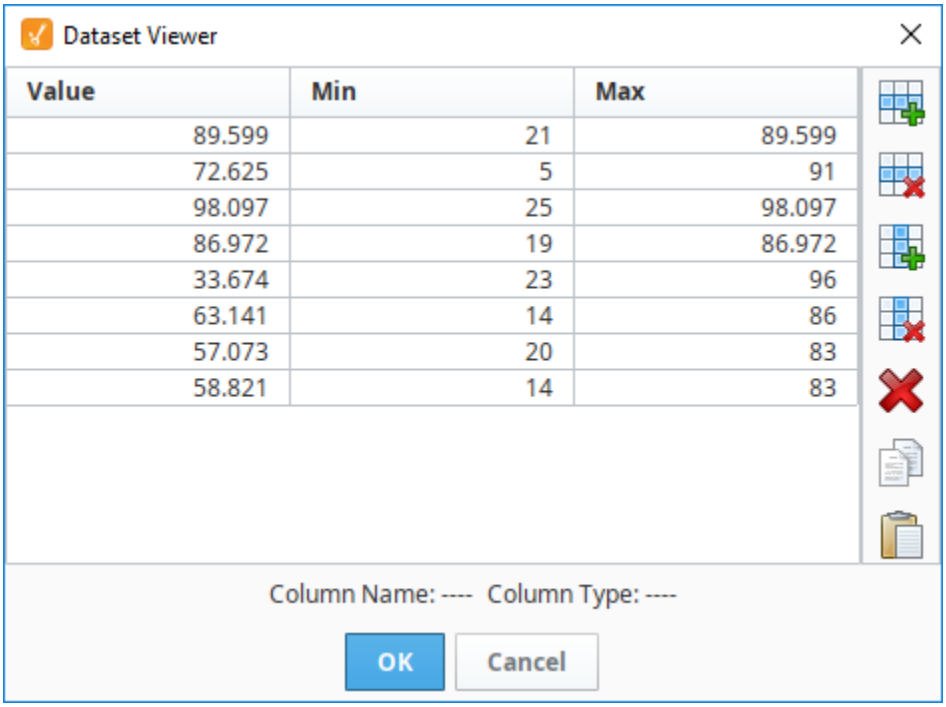


Using the Radar Chart

The Radar Chart can be used to show realtime values by dragging and dropping Tags from the **Tag Browser** on to the chart. Doing so will create a **Cell Update** binding on the Data property that is tied to the Value, EngLow, and EngHigh properties on the Tag. Adding additional Tags will add additional spokes to the chart. Alternatively, a **Named Query** or a **SQL Query** binding on the data property can be used to display historical values, or aggregate previous historical values.

Initial Dataset

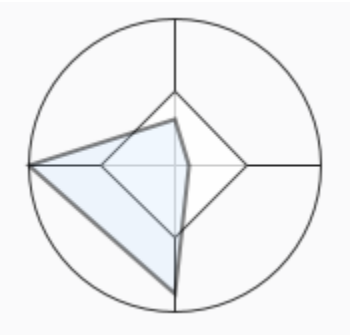
Each new Radar Chart randomly generates a new dataset. The **Data** property on the Radar Chart must have at least a **Value**, **Min**, and **Max** column. Any additional columns are ignored.
To render properly, the dataset must have at least three rows. Datasets with only one or two rows will be drawn as a vertical line.



Min and Max

The **Min** and **Max** column, aside from determining the limits on the the chart, are also used to determine the desired value. The **Desired** value is drawn as the midpoint between the **Min** and **Max** for a single row in the Dataset. Each row of the dataset has a **Min** and **Max** column. The values in these columns are used to determine the scale of the spoke for that variable with the midpoint representing the desired value.

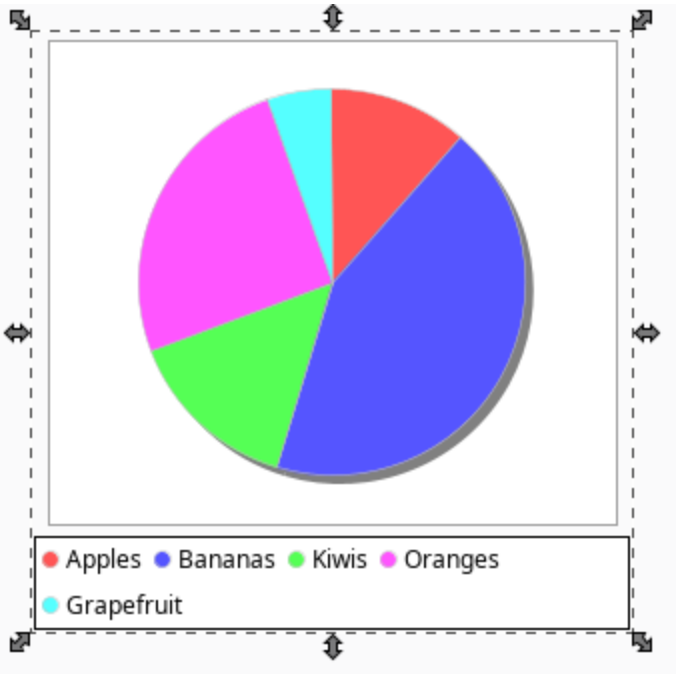
Below we see the white polygon in the center of the chart. This represents the midpoint between **Min** and **Max**



```
"#NAMES"
"Value","Min","Max"
"#TYPES"
"D","D","D"
"#ROWS","8"
"41.51196715968135","18.0","86.0"
"72.21343683086239","2.0","88.0"
"98.91484924220774","16.0","98.91484924220774"
"23.189112936965692","1.0","78.0"
"33.45468212322838","23.0","82.0"
"77.17126241429432","7.0","100.0"
"53.529302336166836","25.0","79.0"
"62.058120439146435","6.0","94.0"
```

Pie Chart

A Pie Chart displays values from several categories, each category is a separate "wedge" of the chart. The total is the sum of all wedges. The key to the Pie Chart component is the Data property, which contains the items that will be displayed as pie wedges.

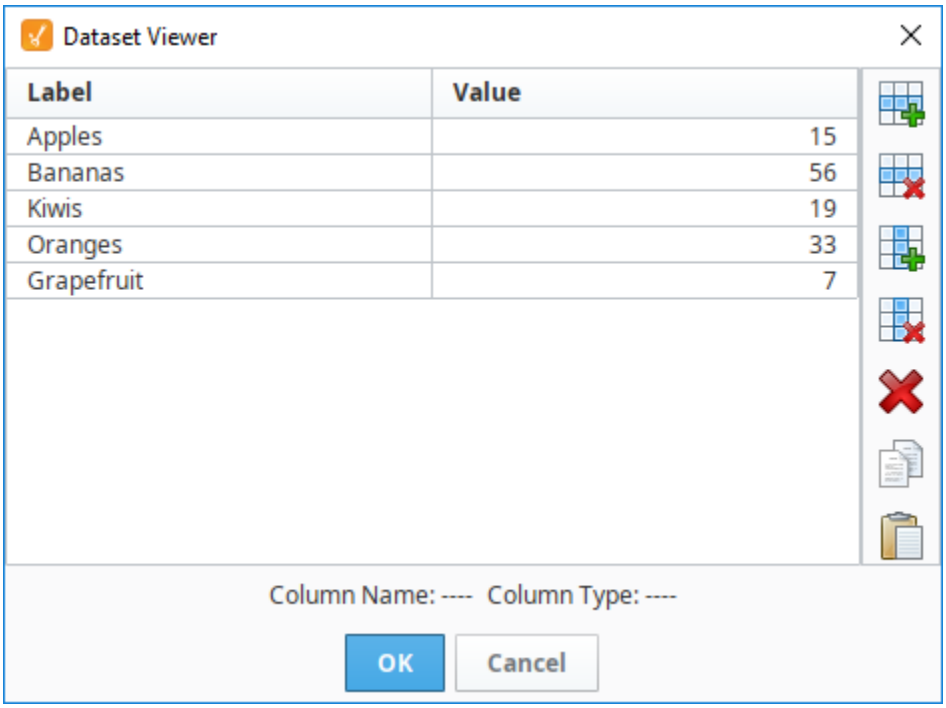


Using the Pie Chart

Typically, data is pulled into the chart from a database using either a **Named Query** or **SQL Query** binding on the **Data** property. The data typically consists of a list of name-value pairs of things that are related.

Initial Dataset

The Pie Chart component contains an initial dataset with two columns, **Label** and **Value**. As the name implies, the **Label** column determines the text associated with each wedge of the pie, while the value is the weight of the wedge.




```
"#NAMES"
"Label","Value"
"#TYPES"
"str","I"
```

```
"#ROWS","5"
"Apples","15"
"Bananas","56"
"Kiwis","19"
"Oranges","33"
"Grapefruit","7"
```

Extract Order

When **Extract Order** is set to **By Row**, then the data must be formatted differently. This order expects each column to be a wedge. Note that only the first row is utilized when extracting by row: subsequent rows are ignored.

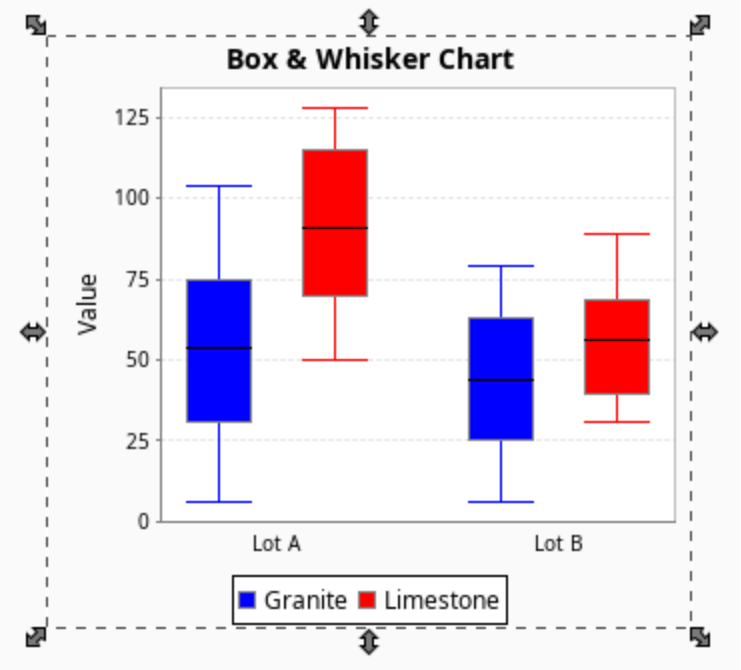
```
"#NAMES"
"Grapefruit","Apples","Bananas","Kiwis","Oranges"
"#TYPES"
"I","I","I","I","I"
"#ROWS","1"
"7","15","56","19","33"
```

Box and Whisker Chart

A Box and Whisker Chart displays pertinent statistical information about sets of data. Each 'Box and Whisker' item on the chart should represent a large amount of data: The high, low, median, and where the middle 50% of the data falls. The dataset that is required for this chart type will be all of your raw data, and it will calculate the box and whiskers for you.

Box Anatomy

The upper and lower bounds of each box (the colored in parts) represent the 1st and 3rd quartiles (quarters of a dataset range). This means the space filled in by the box is 50% of your raw data.
The horizontal line inside of the box represents the median (middle) value.
The lines that stick out above and below the box (whiskers), represent the minimum and maximum values from the raw data.



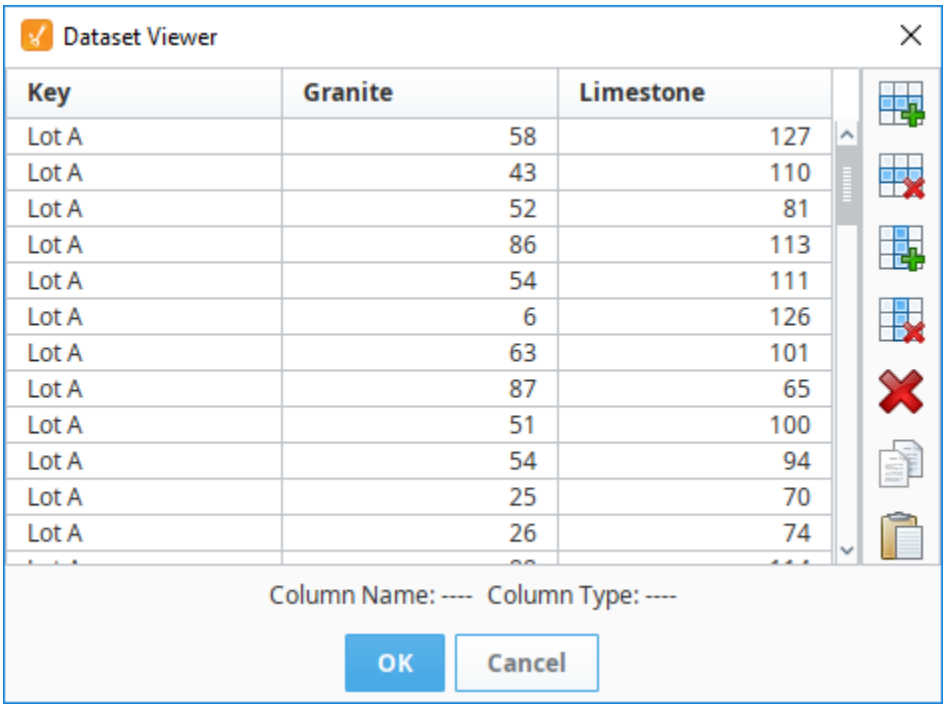
Using the Box and Whisker Chart

Typically, data is pulled into the chart from a database using either a **Named Query** or **SQL Query** binding on the **Data** property. The data typically comes in categories separated by an optional key column, with each category containing multiple values.

Initial Dataset

The first column in the Box and Whisker Chart's dataset is the **Key** column. The Key column determines which series the data pertains to (domain labels). Values in the **Key** column are case sensitive.

The second and additional columns denote categories (legend labels). The initial dataset contains two categories: **Granite** and **Limestone**. Additional columns in the dataset would add additional boxes to the chart.




```
"#NAMES"
"Key","Granite","Limestone"
"#TYPES"
"str","I","I"
"#ROWS","200"
"Lot A","28","108"
"Lot A","46","81"
"Lot A","103","57"
"Lot A","16","93"
"Lot A","41","91"
"Lot A","55","68"
"Lot A","23","93"
"Lot A","49","97"
"Lot A","36","69"
"Lot A","47","106"
"Lot A","75","86"
"Lot A","14","115"
"Lot A","42","70"
"Lot A","100","129"
"Lot A","16","118"
"Lot A","62","125"
"Lot A","14","51"
"Lot A","73","64"
"Lot A","35","55"
"Lot A","96","113"
"Lot A","50","93"
"Lot A","97","72"
"Lot A","7","80"
"Lot A","86","62"
"Lot A","87","78"
"Lot A","80","51"
"Lot A","100","94"
"Lot A","79","124"
```

```
"Lot A","39","107"
"Lot A","16","119"
"Lot A","20","60"
"Lot A","50","124"
"Lot A","37","50"
"Lot A","36","98"
"Lot A","46","77"
"Lot A","33","106"
"Lot A","49","75"
"Lot A","84","60"
"Lot A","17","94"
"Lot A","44","93"
"Lot A","72","105"
"Lot A","35","106"
"Lot A","20","119"
"Lot A","90","51"
"Lot A","37","88"
"Lot A","75","103"
"Lot A","13","104"
"Lot A","47","55"
"Lot A","65","126"
"Lot A","32","90"
"Lot A","85","126"
"Lot A","95","77"
"Lot A","74","123"
"Lot A","104","68"
"Lot A","90","109"
"Lot A","63","66"
"Lot A","60","90"
"Lot A","28","65"
"Lot A","64","69"
"Lot A","55","62"
"Lot A","98","64"
"Lot A","69","100"
"Lot A","35","110"
"Lot A","31","115"
"Lot A","51","106"
"Lot A","16","76"
"Lot A","91","93"
"Lot A","90","77"
"Lot A","93","64"
"Lot A","98","84"
"Lot A","61","95"
"Lot A","65","97"
"Lot A","67","54"
"Lot A","80","92"
"Lot A","104","123"
"Lot A","104","112"
"Lot A","20","71"
"Lot A","95","99"
"Lot A","37","98"
"Lot A","91","51"
"Lot A","101","106"
"Lot A","68","94"
"Lot A","9","96"
"Lot A","14","77"
"Lot A","46","95"
"Lot A","45","95"
"Lot A","79","90"
"Lot A","92","110"
"Lot A","29","80"
"Lot A","42","80"
"Lot A","15","126"
"Lot A","68","77"
"Lot A","69","98"
"Lot A","52","119"
"Lot A","11","72"
"Lot A","14","122"
"Lot A","36","115"
"Lot A","41","66"
"Lot A","98","73"
```

```
"Lot A","46","116"
"Lot B","49","75"
"Lot B","33","46"
"Lot B","53","32"
"Lot B","51","58"
"Lot B","34","81"
"Lot B","44","73"
"Lot B","71","43"
"Lot B","64","37"
"Lot B","58","77"
"Lot B","35","37"
"Lot B","76","88"
"Lot B","11","42"
"Lot B","11","64"
"Lot B","28","85"
"Lot B","26","58"
"Lot B","78","43"
"Lot B","43","69"
"Lot B","66","32"
"Lot B","7","42"
"Lot B","17","71"
"Lot B","59","68"
"Lot B","7","31"
"Lot B","53","48"
"Lot B","20","52"
"Lot B","71","58"
"Lot B","57","85"
"Lot B","14","61"
"Lot B","34","47"
"Lot B","59","74"
"Lot B","78","58"
"Lot B","64","81"
"Lot B","19","31"
"Lot B","43","48"
"Lot B","58","38"
"Lot B","22","48"
"Lot B","20","83"
"Lot B","36","61"
"Lot B","40","69"
"Lot B","64","50"
"Lot B","67","70"
"Lot B","46","36"
"Lot B","9","51"
"Lot B","10","41"
"Lot B","66","35"
"Lot B","46","44"
"Lot B","10","62"
"Lot B","13","35"
"Lot B","74","49"
"Lot B","69","64"
"Lot B","15","68"
"Lot B","56","38"
"Lot B","35","69"
"Lot B","61","37"
"Lot B","25","80"
"Lot B","38","89"
"Lot B","79","56"
"Lot B","6","64"
"Lot B","49","58"
"Lot B","5","54"
"Lot B","6","35"
"Lot B","38","75"
"Lot B","6","77"
"Lot B","39","36"
"Lot B","27","63"
"Lot B","72","78"
"Lot B","55","38"
"Lot B","9","36"
"Lot B","40","65"
"Lot B","57","76"
"Lot B","65","55"
```

```
"Lot B","74","81"
"Lot B","47","85"
"Lot B","66","84"
"Lot B","10","38"
"Lot B","23","53"
"Lot B","79","80"
"Lot B","27","58"
"Lot B","71","58"
"Lot B","27","32"
"Lot B","73","43"
"Lot B","24","57"
"Lot B","27","59"
"Lot B","56","30"
"Lot B","32","55"
"Lot B","7","40"
"Lot B","20","63"
"Lot B","68","74"
"Lot B","64","57"
"Lot B","57","31"
"Lot B","54","61"
"Lot B","33","35"
"Lot B","61","73"
"Lot B","36","61"
"Lot B","26","34"
"Lot B","9","59"
"Lot B","47","60"
"Lot B","61","86"
"Lot B","45","88"
"Lot B","5","87"
"Lot B","6","36"
```

Related Topics ...

- Bar Chart
- Classic Chart
- Pie Chart
- Radar Chart
- Box and Whisker Chart

HTML in Vision

HTML stands for HyperText Markup Language. It is commonly used to style text within web pages. The features that HTML brings to style web pages can be applied to many components within Ignition to style the text within components.

On this page ...

- Using HTML in Components
- Common HTML Elements
- Applying HTML to Components

Using HTML in Components

Many Vision components display a text string. By default, a component's text is displayed in a single font and color and will not wrap when its content exceeds the space the component has made available to the text. However, you can use HTML if you want to mix fonts or colors within the text or if you want formatting such as multiple lines. HTML formatting can be used in Vision components such as buttons, labels, and tables. It can be used in common properties such as the mouse over text property.

To specify that a component's text has HTML formatting, just put the `<html>` element at the beginning of the text, then use any valid HTML element in the remainder.

Closing the HTML element is optional. In other words, there is no need to place a **</html>** at the end of your stylized text. Also, the HTML elements are not case sensitive.

Common HTML Elements

HTML Tags are the special characters that instruct text to become stylized differently than other text within the same text. The following table describes the most common HTML elements that you can use within Ignition.

HTML Element	Name	Description
<html>...</html>	HTML	Initiates an html formatting. In most cases closing the html with </html> is optional.
...	Bold	Applies a bold style to the contents of these elements.
<u>...</u>	Underline	Underlines the text contained within the elements.
<s>...</s>	Strikethrough	Draws a line through the text contained within the elements.
 	Break	Applies a line break at this specific location.
...	Ordered List	Places the text into an ordered list. Text inside list items are ordered by number.
...	Unordered List	Places the text into an unordered list. Text inside list items are ordered by bullets.
...	List Item	Used to represent a list item. Should be contained in an order list ( ) or unordered list ( ).
	Center	Centers the contents of the text. Used directly after the HTML Tag (that is, <html> ...)
...	Font	Colors the contents red. Works with standard color names, hex numbers, or RGB numbers.

Applying HTML to Components

In Vision, you can add HTML to the text property of any component such as, a label, button, or table. These examples aren't unique to their specific components, but can be used on any component that has a Text or Mouseover Text property.

A good rule of thumb for what can be html formatted is text on components that is used for display, not for input. So while the Label components have a Text property that accepts html formatting, the Text Field component's Text property does not accept html formatting, as a user may type into the component.

For example, individual words or phrases within the text can be made bold:

```
<html>This is a <b>bold</b> word
```

You can also create a list, such as instructions in the Mouseover Text on a component:

HTML in mouseover property

```
<html>
These are the instructions:
<ol>
<li>Stop the process.</li>
<li>Check on this.</li>
<li>Remove that.</li>
</ol>
```

Watch the Video

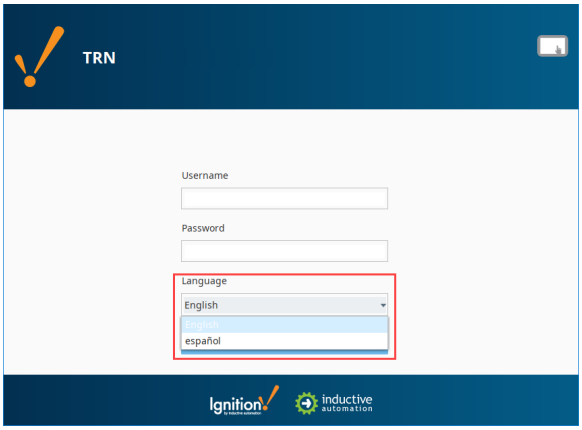
Localization in Vision

Localization in the Vision module utilizes the terms in the platform's Translation system. Once terms have been defined, translations can be enabled by either component or scripting.

Selecting a Language on Client Startup

Once you create a second language, the Client Login Screen will automatically display a Language Selector where you can select your preferred language. There is Project Property setting that allows you to Show or Hide the Language Selector at login. By default, it is set to Automatic so you will see the Language Selector at login when two or more languages are created unless you choose to hide it.

If a user that has a preferred language selected in their user profile settings, Ignition will login to the Client with their preferred language automatically.

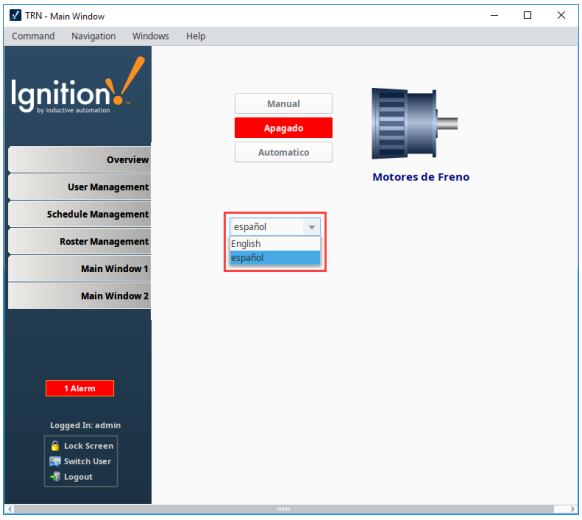


Using the Language Selector Component

A single Language Selector on a window has the potential to trigger translations on all windows. There is no binding involved in selecting a language because they are compared directly against the Translation Manager database, so the component only needs to be placed onto a window after a second language has been defined.

The component also offers an easy way to switch languages without forcing the user to log out first. This way a single Language Selector component can exist on a navigation window and provide language translations for all components on all windows.

More information on the Language Selector Component can be found in the Appendix.



On this page ...

- Selecting a Language on Client Startup
- Using the Language Selector Component

Related Topics ...

- Localization and Languages

Binding Types in Vision

Binding is perhaps the most important concept to understand when designing a project using the Vision module. It is primarily through property bindings that you bring windows to life and have them display useful things. A binding simply links one component's property to something else.

When you initially place a component on a screen, it doesn't really do anything. Changing its properties in the Designer will make it look or act different, but it has no connection to the real world and this is what bindings adds.

Binding, as its name suggests, lets you bind a property to something else, such as

- A Tag
- The results of a SQL query executed against a remote database
- Another component's property
- An expression involving any of these things

For example, bind the Value property of an LED Display to an OPC Tag, and voilà - the value property will always be the value of that Tag - creating a dynamic display. Bindings can also work the other way, using a bidirectional binding. Bind the value of a numeric text box to a Tag, and that Tag will be written to when someone edits the value in the text box.

The power of bindings comes from the variety of different binding types that exist, and the fact that you can bind nearly any property of a component to anything else. Want its foreground to turn red when an alarm is above a certain severity? Bind its **LED Lit** property color to a Tag's **Alarms. HighestActivePriority** property. Want it to only appear if a supervisor is on shift? Bind its visible property to the result of a SQL query that joins a personnel table with a shift table. The possibilities are nearly endless.

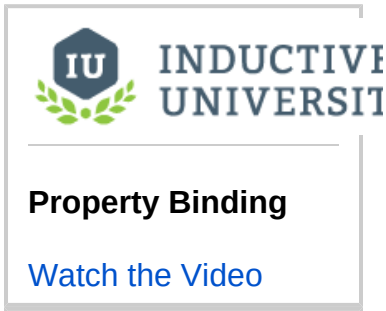
On this page ...

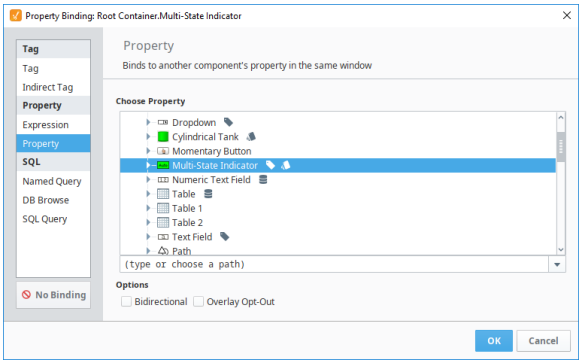
- Property Binding Types
- Setting Up Bindings
- Event-Based Bindings vs. Polling Bindings
  - Polling Options
- Copying Bindings

Property Binding Types

A property can have one of many different types of bindings. Instead of setting a label statically, the text might change based on a PLC value or on-screen selection. There are many ways to bind your components to show values from PLCs, databases, other components, or user input. You can even bind some or all of the properties on each component. You can bind component values using:

- **Property** simply binds one property to another. When that property changes, the new value is pushed into the property that the binding is set up on.
- **Tag** binds a property directly to a Tag property (typically the value) which sets up a Tag subscription for that Tag, and every time the chosen Tag property changes, the binding is evaluated, and pushes the new value into the bound property.
- **Indirect Tag** is similar to the standard Tag binding except that you can introduce any number of indirect parameters to build a tag path dynamically in the runtime.
- **Tag History** is used for Dataset type properties. It runs a query against the Tag Historian.
- **Expression** uses the simple expression language to calculate a value which can involve lots of dynamic data.
- **Named Query** executes a Named Query that had been previously created.
- **SQL Query** is a polling binding type that runs a SQL Query against any of the database connections configured in the your Gateway.
- **Database Browse** is equivalent to the SQL Query binding except that it helps write the queries for you.
- **Cell Update** enables you to easily make one or more cells inside a dataset dynamic. This is useful for components that store configuration information inside datasets like the Easy Chart.
- **Function** is a generic binding type that lets you bind a dataset property to the results of a function. It allows any of the function's parameters to be calculated dynamically via Tag and property bindings.
- **Style Customizer** is not one of the standard bindings, but changes properties to create cohesive styles based on different states.

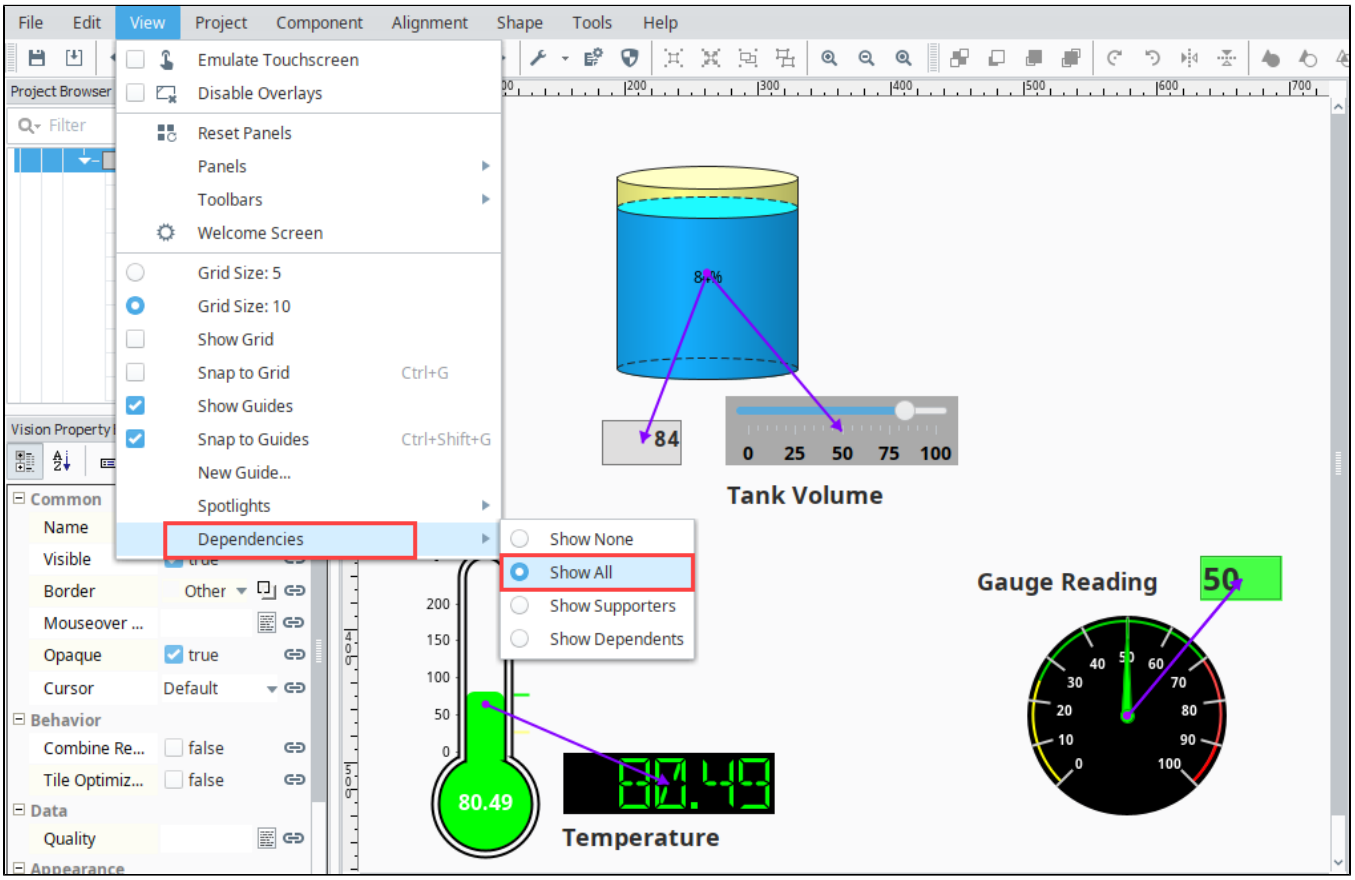




Setting Up Bindings

Every component that you put on the screen has various properties that change the component's appearance and behavior. To make components do something useful, like display dynamic information or control a device register, you configure bindings on the component. It's the bindings that brings your components to life and have them do useful things. Components can be configured to do just about anything using bindings. To set up a binding on a property, simply click the Binding icon to the right of the property in the Property Editor.

In this image, bindings were set to make these random components do something. You can quickly view dependencies to determine what is linked to what by going to **View > Dependencies > Show All**. As shown below, a line is drawn from the Tank to the Slider letting you know the Tank is bound to the Slider.



Event-Based Bindings vs. Polling Bindings

While there are quite a few different binding types, they fall into two broad categories: event-based and polling. Some complex bindings can span both categories.

Event-based bindings are evaluated when the object they are bound to changes. For example, when you bind a property to a Tag, that binding listens to the Tag, and every time the Tag changes, it assigns the Tag's new value into the property that it is on. If you bind the value of a Cylindrical Tank to the value of a Slider, every time the slider changes, it fires a propertyChangeEvent. The binding is listening for this event, and when it is fired, the binding updates the tank's value. The following bindings are event-based:

- Tag and Indirect Tag bindings
- Property bindings
- Some Expression bindings
- Cell Update bindings

Polling bindings are evaluated when a window first opens, on a timer, or when they change. For example, if you bind the data property of a Table to the results of a SQL query, that query will run on a timer, updating the Table every time it executes. The following bindings are based on polling:

- Bindings that query a database, including Named Query bindings, DB Browse bindings, SQL Query bindings, and Tag History bindings
- Some Expression bindings, like runScript() or now()
- Function bindings

Many bindings can combine elements of a polling binding and event-based binding. An expression binding may combine lots of other bindings to calculate a final result. A query binding will often itself be dynamic, altering the query based on other bindings.

For example, you might have a dropdown on a window that lets the operator choose a type of product that is produced. Then you can use a query binding like the following to calculate the defect rate for the given product:

SQL - Using a Component Property Reference

```
SELECT
   SUM(defective) / COUNT(*) AS DefectRate
FROM
   production_table
WHERE
   productCode = '{Root Container.ProductPicker.selectedValue}'
```

The blue code is a property binding inside of the query binding. Every time this (event-based) binding fires, the query will run again, but will also run on a set timer based on its polling schedule. Using bindings like this, you can create highly dynamic and interactive screens with no scripting whatsoever.

Polling Options

The following are the options you can choose from for bindings that poll:

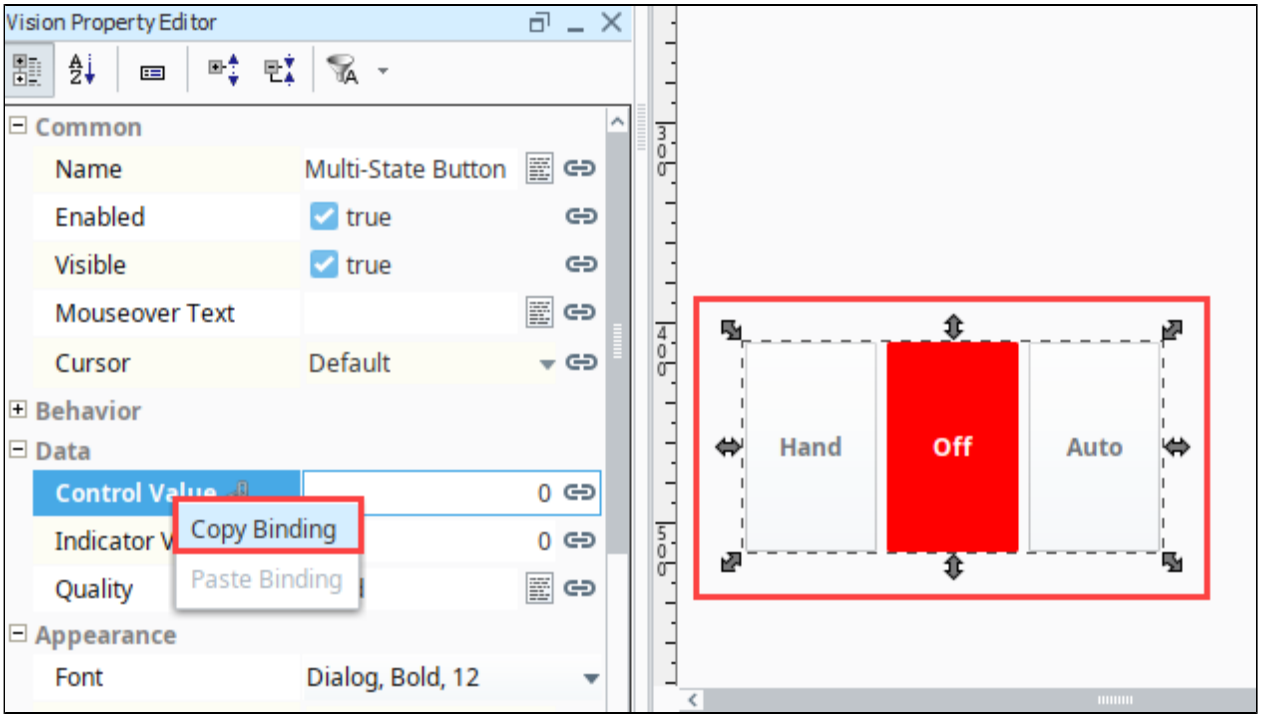
- **Polling Off**
  The query will run once when the window is opened, and again whenever a reference inside of the the binding changes.
- **Relative Rate**
  The binding will poll at the project's Base Polling Rate, which is 5 seconds by default, plus or minus the given Polling Rate.
- **Absolute Rate**
  Using this option, you can specify an absolute rate for the binding to execute at, instead of one that is based off the relative rate.

Regardless of which option is selected, polling bindings always fire when the window the component is on opens. This allows the component an opportunity to fetch an initial value.

Additionally, all three types will **always** update if the binding contains a reference to something else, such as a Tag or property value (noted with the brace-notation "{}"), and the value of that reference changes. Typically this is seen in SQL query bindings: polling can be turned off, and the query can reference a component value in a WHERE clause. When the referenced property value changes, the query will execute and retrieve new results.

Copying Bindings

When you copy a component, all bindings, scripts, etc. are copied along with it, but you can also copy a property binding from one property to another. Bindings can be copied from one property to another by right clicking on a property with a binding on it and selecting **Copy Binding**. Then, on another property, right click and select **Paste Binding** to paste the binding onto the property. This can be on the same component or a completely different component. The only prerequisite is that both property bindings must use a compatible property type. (For example, a binding that resolves to a string will not work on an integer property.)



In This Section ...

Property Bindings in Vision

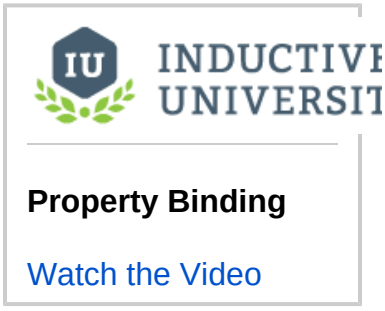
Property to Property Binding

A property binding is a simple type of binding. It binds one component's property to another. When that property changes, the new value is pushed into the property that the binding is set up on.

> **Why aren't all properties listed?**
>
> You may notice that the list of properties available to bind to is smaller than the list of all properties. While nearly all properties can be bound, only some properties can be bound to. Only properties that fire a **propertyChangeEvent** may be bound to.

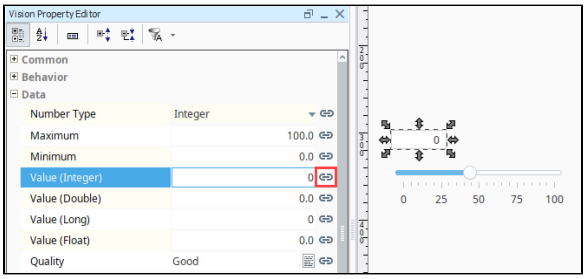
On this page ...

- Property to Property Binding
- Bidirectional Property Bindings

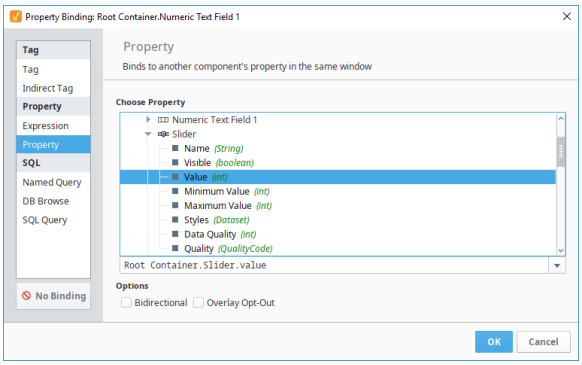


In the following example, we'll bind the value of a Numeric Text Field to a Slider component.

1. Place a Numeric Text Field component and a Slider component on window.
2. Select the **Numeric Text Field**.
3. Click the Binding icon next to the Value (Integer) property.

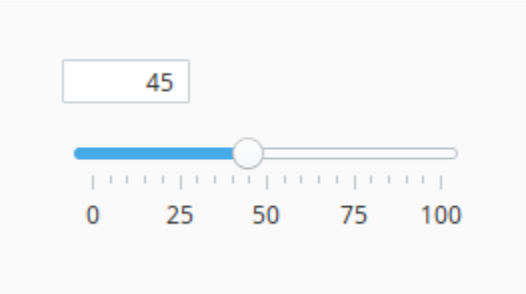


4. Select the **Property** Binding Type. Choose the **Slider's Value** property.



5. Click **OK**. Put the Designer in **Preview Mode** .

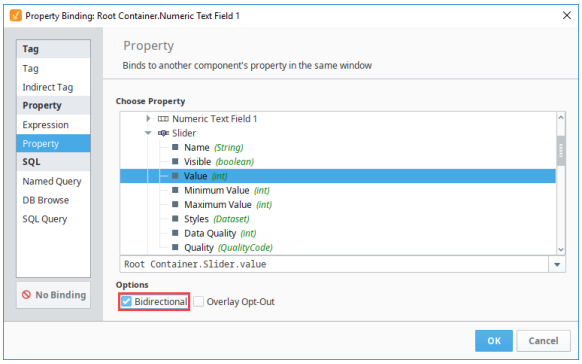
6. Move the slider. You'll see that the value from the Slider component appears in the Numeric Text field.



This can be useful to provide visual feedback to what a user is doing. The operator would input something, and they would see another component adjust to match the setting they just changed. Notice though, how if I were to change the value of the Numeric Text Field, the Slider will not update. Bindings are one direction only by default.

Bidirectional Property Bindings

Property Bindings have the ability to become Bidirectional, meaning instead of having the binding go one way only, it will work both ways, even with just the one binding. Take the previous example with the Numeric Text Field and Slider again. When changing the value of the Slider, the Numeric Text Field would update, but updating the value of the Numeric Text Field would not update the Slider. If we reopen the binding on the Value (Integer) property of the Numeric Text Field, we can see in the bottom left corner a checkbox for **Bidirectional**.



Check the **Bidirectional** option and then save the binding. It will now be a bidirectional binding.

Tag Bindings in Vision

Binding Properties to Tags

A Tag binding is a very straight-forward binding type. It simply binds a property directly to a Tag. This sets up a subscription for that Tag, and every time the chosen Tag changes, the binding is evaluated, pushing the new value into the bound property. If you choose a Tag in the tree, and not a specific property of that Tag, the Value property is assumed.

Drag and Drop

There are several ways to drag and drop Tags onto a component allowing you to create screens very quickly. You can quickly bind a Tag by dragging it from the Tag Browser into a component or into the component property.

Binding to a Component

Ignition automatically creates the Tag bindings to several of the component properties when you choose to bind a Tag to a component by dragging and dropping. This is true for both creating a component by dragging and dropping a Tag onto empty space on a window at the same time Ignition prompts you for what type of component you want to create, and by dragging and dropping a Tag directly onto a component that already exists on a window. In both cases, Ignition automatically creates the Tag bindings on the component.

In addition, some of the bindings will be bidirectional or they may be expression bindings. How Ignition handles the binding depends on the disparity between the data types of the Tag and the target component. For example, binding a numerical Tag to a label component will result in an expression binding that formats the number to a string.

Binding to a Component Property

Ignition automatically creates a Tag binding to the property that you dropped the Tag into in the Property Editor, resulting in the property binding to the value of the Tag. In addition, it is possible to bind the Tag attributes to the component's property. For example, the Tag's Engineering High Limit attribute could be bound to the capacity property of a cylindrical tank component.

On this page ...

- Binding Properties to Tags
- Drag and Drop
  - Binding to a Component
  - Binding to a Component Property
- Bidirectional Tag Bindings
- Bindings to Tag Properties

Tag Binding

Watch the Video

Tag Binding – Drag and Drop

Watch the Video

Bidirectional Tag Bindings

Tag bindings can be made bidirectional simply by checking the **Bidirectional** checkbox at the bottom of the **Property Binding** window. A Tag can be set as a bidirectional binding, if it has a read/write permission and if the user has the security permission to write to the Tag. The Fallback Delay is the amount of time that the value will remain at the written value, waiting for a Tag change to come in. If no Tag change comes in within the allotted time (specified in seconds), the property will fall-back to the value as it was before the write. This is needed, because sometimes even if a write succeeds, another write or ladder logic in a PLC might have written something different, even the old value, in which case no Tag change event will be generated. As a rule of thumb, the fallback delay should be twice the Tag's scan class rate.

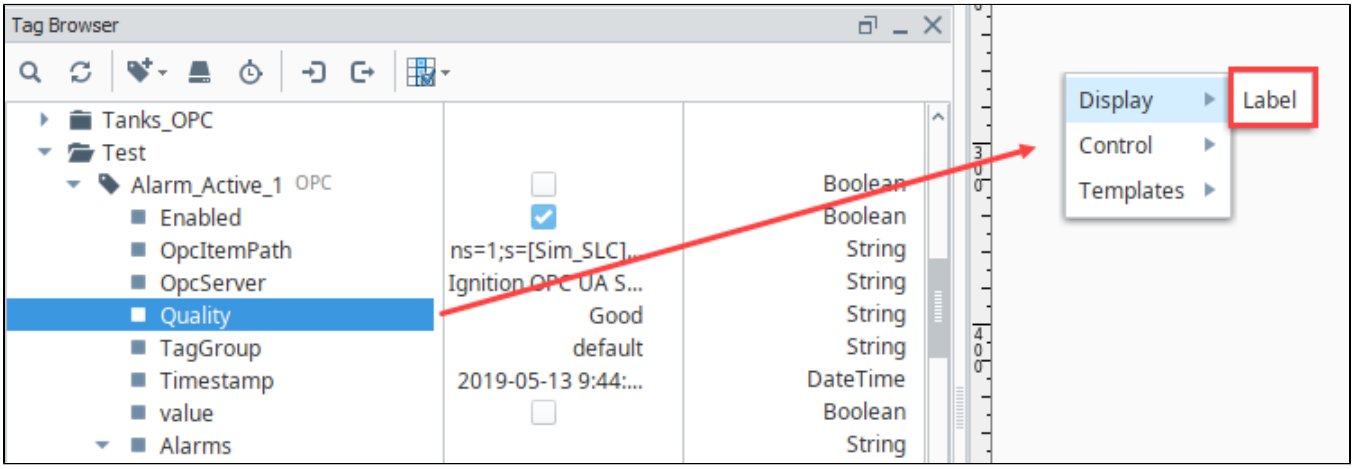
Tag Binding - Bidirectional

Watch the Video

Bindings to Tag Properties

Aside from binding a property to a Tag's value, you can also bind to properties on a Tag, such as **Tooltip, Quality,** or **AlarmActiveAckCount**. This is useful when you don't need the value of the Tag, but rather the state, or some other configuration on the Tag. Here we see a boolean Memory Tag. It has a property indicating the Tag's quality. We can easily display that quality property on a component.

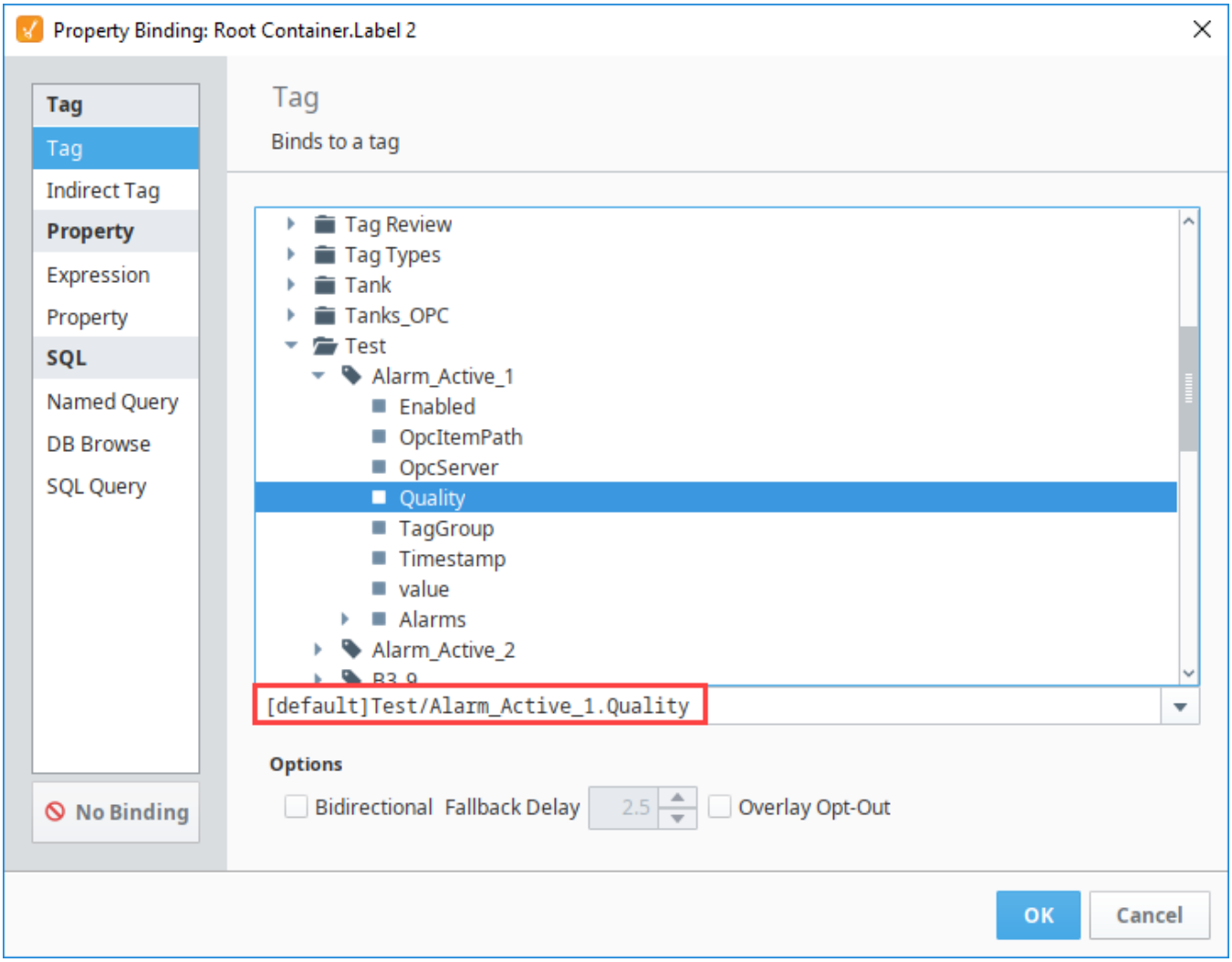
The simplest approach involves a **Tag Binding**. This can be achieved by dragging-and-dropping the Tag Property onto a Window, component, or component property. For example, you can drag a Tag's **Quality** property to a **Label** component.



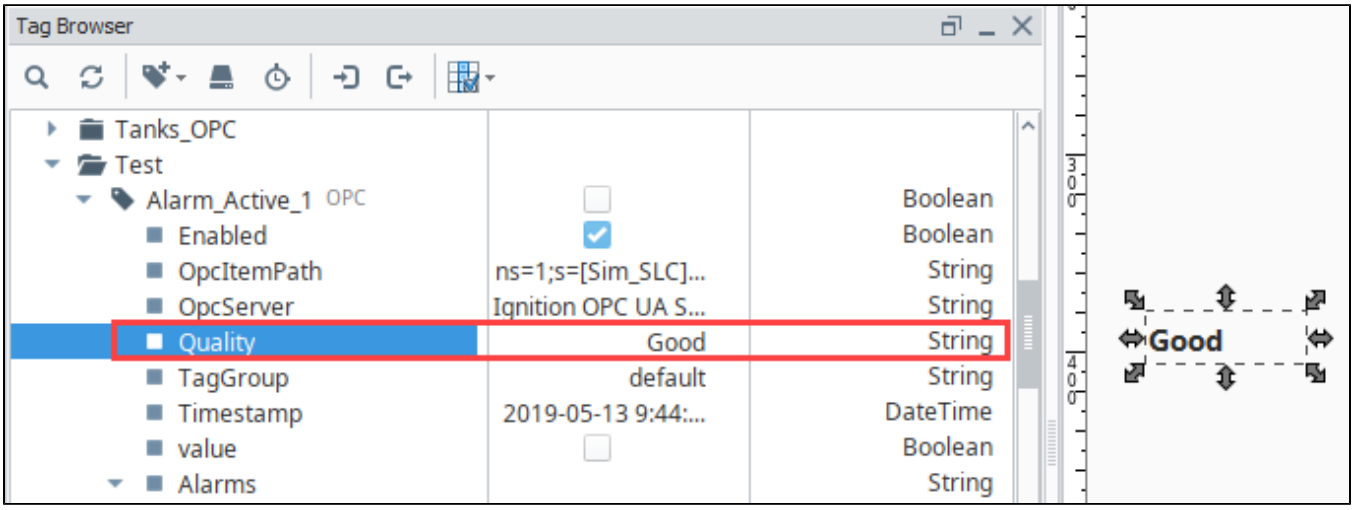
This is similar to creating a standard Tag Binding, except we're using a property on the Tag instead of the Tag's value. The resulting Tag Binding would be:

```
[default]Test/Alarm_Active_1.Quality
```

Note that the property name has been appended to the path.



The **Label** component now displays the current value of **Good** for the Tag's Quality property.



Indirect Tag Bindings in Vision

Binding Properties to a Dynamic Set of Tags

An Indirect Tag binding is very much like a standard Tag binding, except that you may introduce any number of indirection parameters to build a Tag path dynamically in the runtime. These parameters are numbered starting at one, and denoted by braces, for example, {1}. The binding will be linked to the Tag represented by the Tag path after the indirection parameters have been replaced by the literal values they are bound to. An indirection parameter may represent a property on any component in the same window.

For example, instead of binding straight to a Tag's path, like

```
[TagProvider]MyPlant/EastArea/Valves/Valve4/FlowRate
```

or

```
[TagProvider]MyPlant/WestArea/Valves/Valve2/FlowRate
```

You can use other properties to make that path indirect. Suppose the "area" and "valve" number that we were looking at was passed into our window via parameter passing. Then we might use those parameters in the Tag path, like this:

```
[TagProvider]MyPlant/{1}/Valves/Valve{2}/FlowRate
{1}=Root Container.AreaName
{2}=Root Container.ValveNumber
```

Now our binding will change which Tag it is pointing to based on the values of those Root Container properties.

Creating an Indirect Tag Binding

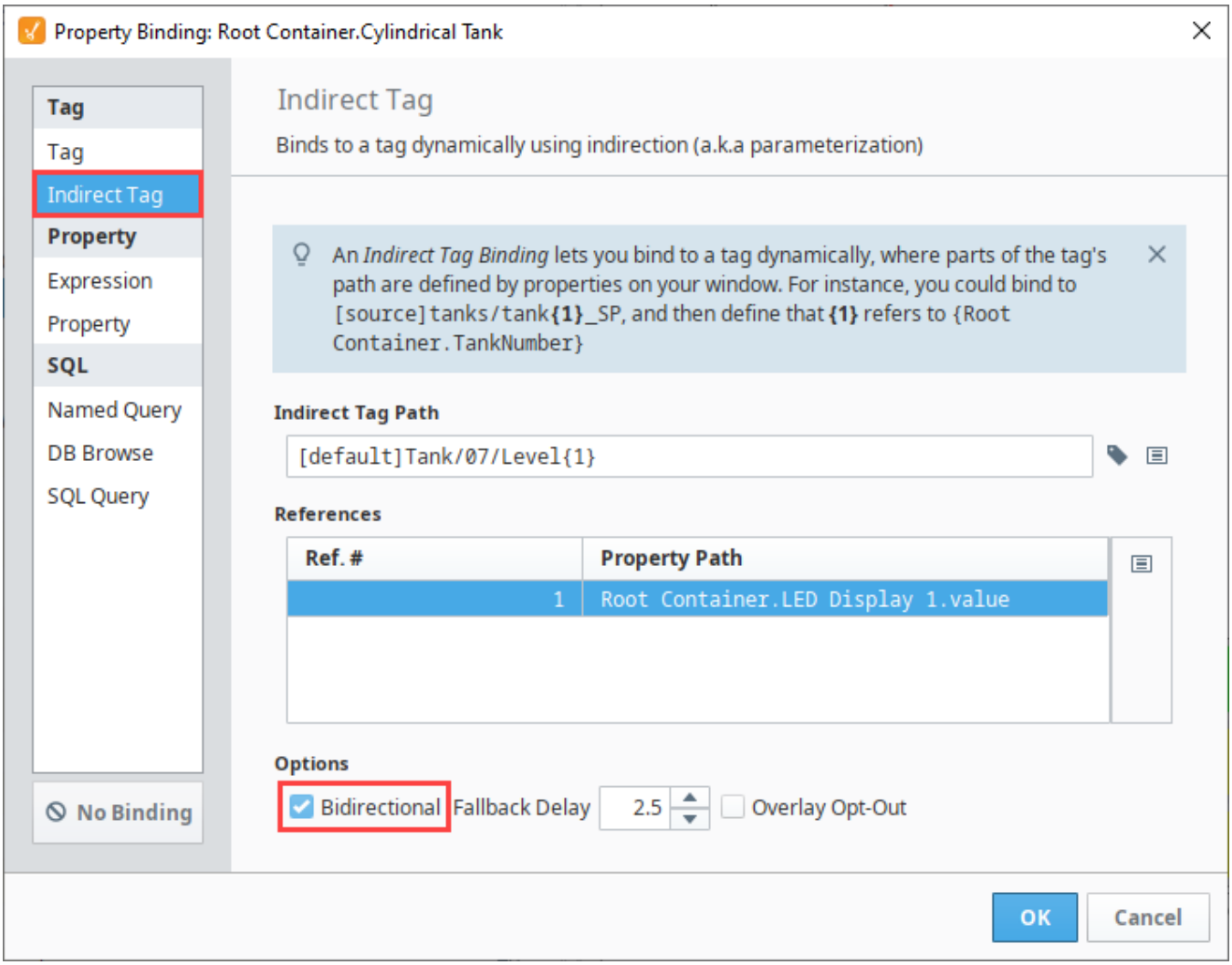
When setting up an Indirect Tag Binding, there are a few tools in the binding window that help make it easier.

First there is the Indirect Tag Path. This field is where the Tag Path with parameters needs to be entered. Indirect Tag Bindings use numbered parameters at places in the Tag Path where indirection is going to occur. To the right of the Indirect Tag Path field are the Tag and Property reference helper buttons. The Tag button will enter the full Tag Path of the selected Tag into the Indirect Tag Path, while the Property button will add a new parameter reference to the Indirect Tag Path, and bind it to the selected property. The last area is the list of references, where each row in the list corresponds to a {1} parameter reference, and each row can be bound to property on the window. To bind a parameter reference to a property, simply select its corresponding row, and use the property selector to the right of the References list to select a property from the window.

Putting some thought into your Tag structure will make using the Tags indirectly much easier!

Bidirectional Indirect Tag Binding

Indirect Tag Bindings can also be made Bidirectional by clicking the **Bidirectional** checkbox at the bottom of the binding window. This will allow any input from a user on that property to be written back to the Tag. To work properly, the Tag needs to have the proper security to accept writes.



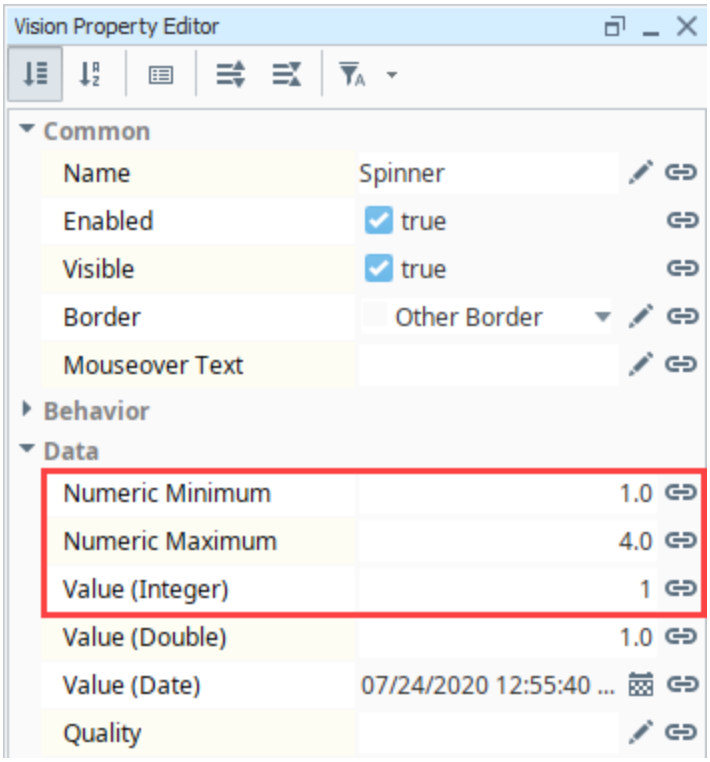
Indirect Tag Binding Example

In this example, we have some different motors, where each motor is a folder of Tags. Each motor has an amps Tag that is within the folder, so that our Tag paths look like the following:

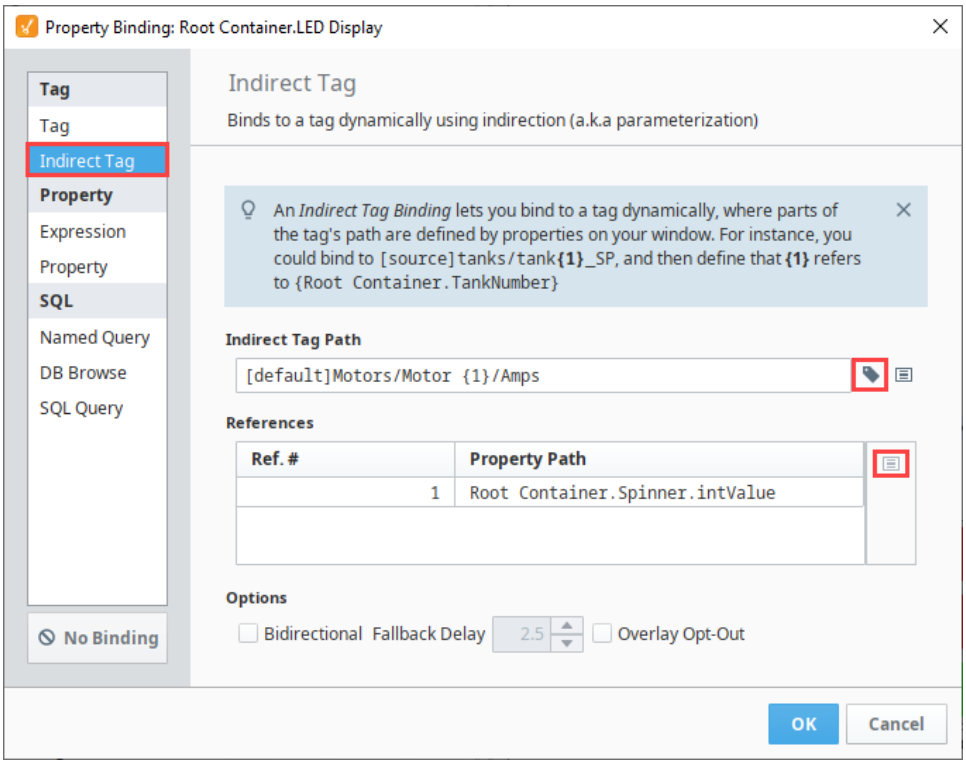
```
Motor 1/Amps
Motor 2/Amps
Motor 3/Amps
Motor 4/Amps
```

Instead of creating four different displays for these four different Tags, we can create a single display and make it indirect. We need two things for this example: a component to display the value in, and a component which allows the user to select which motor they are looking at.

1. Drag an **LED Display** component onto the window.
2. Then drag a **Spinner** component onto the window. This we will use to enable the user to select which motor they are looking at.
3. There are four motor Tags, so change the **Numeric Maximum** property of the Spinner to 4, and the **Numeric Minimum** property to 1. You may also need to change the **Value (Integer)** to 1.

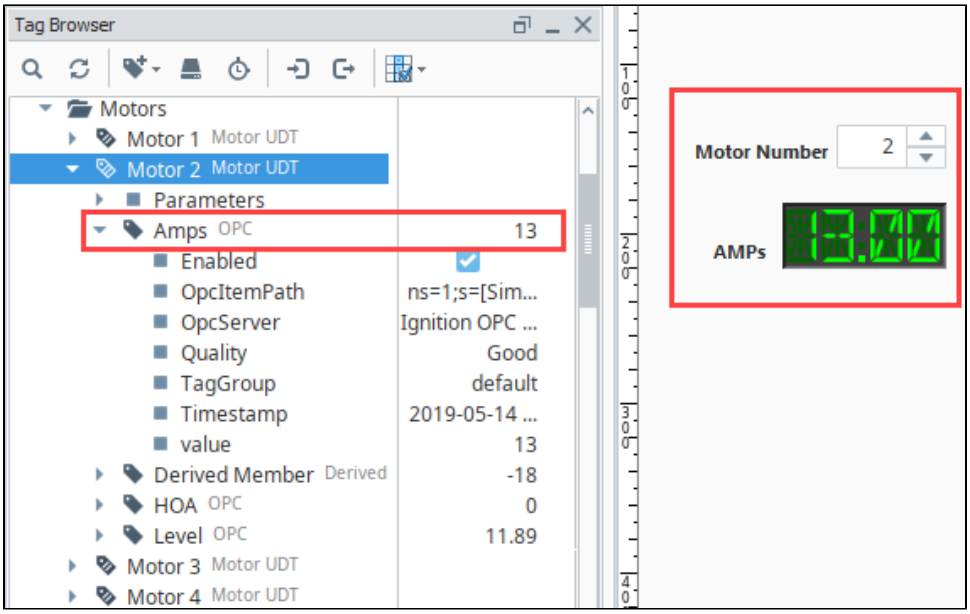


4. Select the **LED Display** component. Click on the Binding icon next to the **Value** property of the LED Display.
5. Select the **Indirect Tag** type.
    a. Click the Tag icon and select the **Motor 1/Amps** Tag.
    b. Delete the '1' in the Tag Path, and replace it with **{1}**.
    c. In the References section, select the row, and click the Insert Property Value icon. Select the **Value** (Integer) property of the Spinner.
    d. Click **OK** to save the binding.



6. To test it out, put the Designer into **Preview mode**. Notice how the value represented in the LED Display depends on what value is in the Spinner. Because the Spinner has the maximum value set to 4, users won't be able to set a motor number that does not exist. Additionally,

adding new motors simply means adjusting the maximum value on the Spinner.



Tag History Bindings in Vision

Binding Properties to the Tag Historian

The Tag Historian binding type, which is only available for Dataset type properties, runs a query against the Tag Historian.

Selected Historical Tags

For this type of query, you must select at least one Tag path from the **Available Historical Tags** to query. The Dataset returned by the query will have a timestamp column, and then a column for each path that you select here.

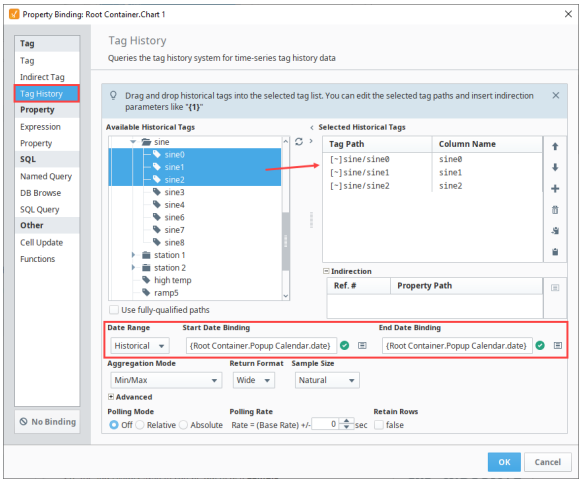
Date Range

Choose either a Historical or Realtime query. Historical queries use a date range that must be bound in from other components on the screen, typically a Date Range or a pair of Popup Calendars. Realtime queries always pull up a range that ends with the current time, so all they need is a length.

> Please note that intervals returned by Historical queries are inclusive of the End Date, including when the End Date is set to now(). This means you may see one additional interval than expected that only contains future dates, which get interpolated to 0 and can cause trending issues.
>
> For example, if you want data from 10am - 11am in 1 minute windows, you'll need to set your query from 10am-10:59am. Querying to 11am would create an interval to contain it and that window will often return 0 since there is typically no future value. Additionally, if you went on to add the results of two queries of adjoining times, such as 10am - 11am and 11am - 12pm, the first window of the second period would have duplicate data to the last window of the first period.

This example uses a Historical query and two Popup Calendars for the start and end dates. The history is presented in the Table below.

1. In the Designer, drag two **Popup Calendar** components and a **Table** component from the Component Palette into your workspace.
2. Select the **Table** and right click on the Binding icon for the **Data** property.
3. Drag a sine0, sine1, and sine2 under the Tag Path column under the **Selected Historical Tags** area.
4. Under Date Range, select **Historical**.
5. Under **Start Date Binding**, click on the Property icon and under one of the Popup Calenders, select **Date**.
6. Under the End Date Binding, click on the Property icon and under the second Popup Calender, select **Date**.
7. Click **OK**.

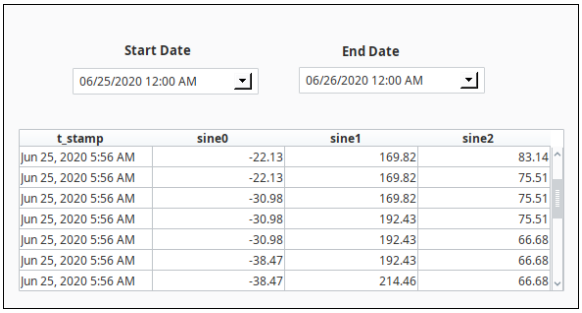


8. Now you can see the history of the three Sine tags along with a timestamp. You can scroll through the information in the table to see the history that was logged. To change the date range, click on dropdown buttons to bring up the popup calendars to change the date range.

On this page ...

- Binding Properties to the Tag Historian
  - Selected Historical Tags
  - Date Range
- Sample Size and Aggregation Mode
  - Aggregation Mode
  - Sample Size
  - Return Format
  - Advanced Options
- Indirect Tag History Binding

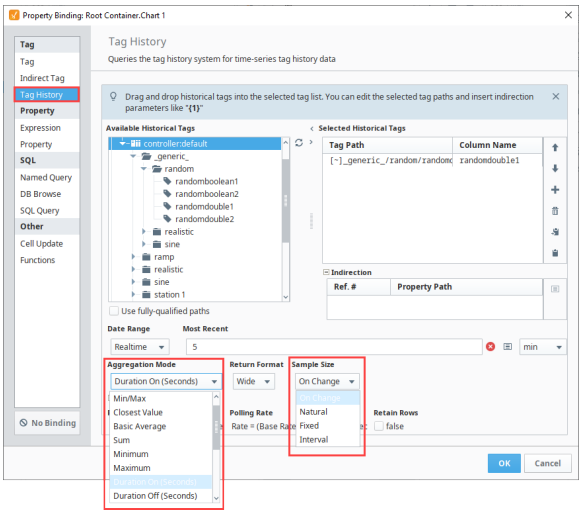
The tag history binding type allows you to bring back this history.



t_stamp	sine0	sine1	sine2
Jun 25, 2020 5:56 AM	-22.13	169.82	83.14
Jun 25, 2020 5:56 AM	-22.13	169.82	75.51
Jun 25, 2020 5:56 AM	-30.98	169.82	75.51
Jun 25, 2020 5:56 AM	-30.98	192.43	75.51
Jun 25, 2020 5:56 AM	-30.98	192.43	66.68
Jun 25, 2020 5:56 AM	-38.47	192.43	66.68
Jun 25, 2020 5:56 AM	-38.47	214.46	66.68

Sample Size and Aggregation Mode

In places where the Tag History system can be queried, a **Sample Size** and **Aggregation Mode** can be selected that will determine how the results will be queried out and how the raw values will be aggregated.



Aggregation Mode

The **Aggregation Mode** dictates what happens when multiple raw values are encountered for a given **time slice**. The number of values within a time slice is determined by the **Sample Size** properties. For more information on how time slices are populated, see the How the Tag Historian System Works page.

Aggregation Mode	Description
Time-weighted Average	The values are averaged together, weighted for the amount of time they cover in the interval. **Note:** In scripting, Time-weighted Average is instead labeled Average. These are the same.
Min/Max	The minimum and maximum values will be returned for the window. In other words, two rows will be returned. If only one value is seen in the time slice, only one row will be returned.
Closest Value	The value closest to the ending time of the interval will be returned. **Note:** In scripting, Closest Value is instead labeled Last Value. These are the same.

Basic Average	The values are summed together and divided by the number of values. **Note:** In scripting, Basic Average is instead labeled Simple Average. These are the same.
Sum	The values in the time slice are summed together.
Maximum	The maximum value in the time slice.
Minimum	The minimum value in the time slice.
Duration On	Returns the number of seconds that the value was recorded as non-zero.
Duration Off	Returns the number of seconds that the value recorded as zero.
Count On	Returns the number of times the Tag's value went from a zero value to non-zero.
Count Off	Returns the number of times the Tag's value changed from a non-zero value to zero.
Count	Returns the number of times a value was recorded
Percent Good	Time-weighted percentage of good values over the date range.
Percent Bad	Time-weighted percentage of bad values over the date range.
Range	Returns the range between the highest and lowest value for the period. This feature was changed in Ignition version **8.1.17**: Range mode will return "0" if the historical tag value remains static over the given Time Range.
Standard Deviation	Standard Deviation - Returns the standard deviation of values, or how much spread is present in the data; low standard deviation shows the values are close to the mean, and high standard deviation shows that the data points are spread out over a large range of values. Only good quality values are used when calculating
Variance	Returns the variance of values. Similar in concept to standard deviation. Only good quality values are used when calculating.

Sample Size

The sample size determines how many data points will be returned from the query.

On Change

An On Change query will return points as they were logged, and can be thought of as a "raw" query mode. This means that the results may not be evenly spaced. Also, it is important to note that every changed value will result in a row, and therefore if you are querying multiple tags and once, you may end up with more rows than you anticipated. For example, if Tag A and Tag B both change, you would end up with [[A_0, B_0],[A_1, B_0], [A_1, B_1]].

If you want to essentially retrieve raw values, while coalescing them down into fewer rows, try using the Interval sample mode, with an interval set to your largest acceptable time between rows, and select "prevent interpolation" from the advanced settings.

Natural

A Natural query will look up the logging rate for the queried tags (when possible), and return results spaced apart at that rate. This means that the return size will vary with the date range.

Fixed

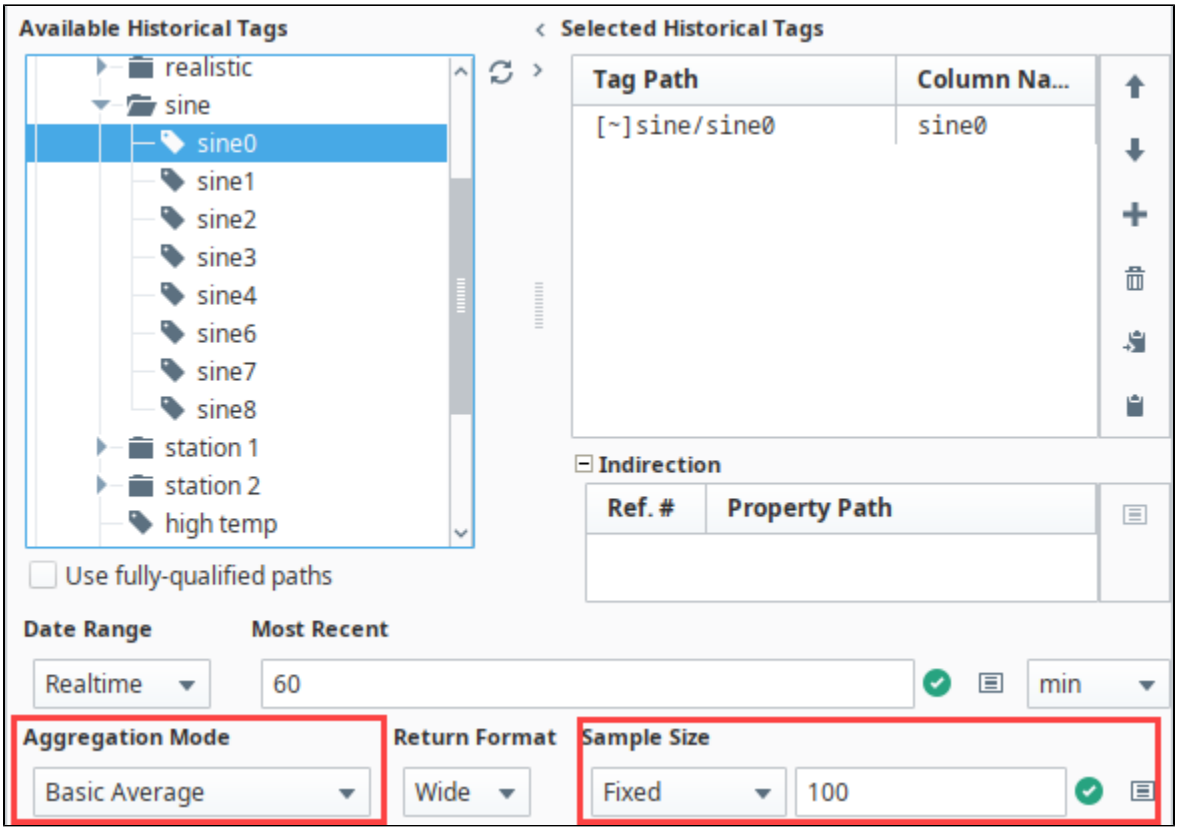
You can use the **Sample Size** and **Aggregation Mode** on the **Tag History** binding type to fix the number of records that are retrieved. The Fixed sample size will cause the binding to retrieve all records from the date range, and aggregate them evenly between a fixed number of points. This will ensure that the number of rows will remain the same without regard to the size of the dataset. In windows where users are able to select a large range of data, Fixed is recommended as it will prevent the property from loading an excessive number of records.

In cases where the number of points can not evenly represent the data from the date range, an extra point will be added, making the final size of the da taset the fixed value + 1.

Selecting the **Min/Max** aggregation mode returns two rows of data for every row requested. Each pair represents a minimum and a maximum result from the underlying data. Therefore, a table with a fixed length, would return double the requested amount with Min/Max aggregation mode selected. With Min/Max aggregation mode selected, and with a fixed row length of one, the data set returns the oldest tag value of the time range

The following image shows a Tag History Binding pulling data from the last one day. The **Sample Size** is configured to **Fixed** with a value of **100**, and the **Aggregation Mode** is set to **Basic Average**. This means that the binding will query for data from the last one day, regardless of how many records there are, and create 100 time-slices that are evenly dispersed between the start and end periods of that range. Then, a basic average of the

tag values are calculated for each time-slice. The resulting values are then returned to the property.



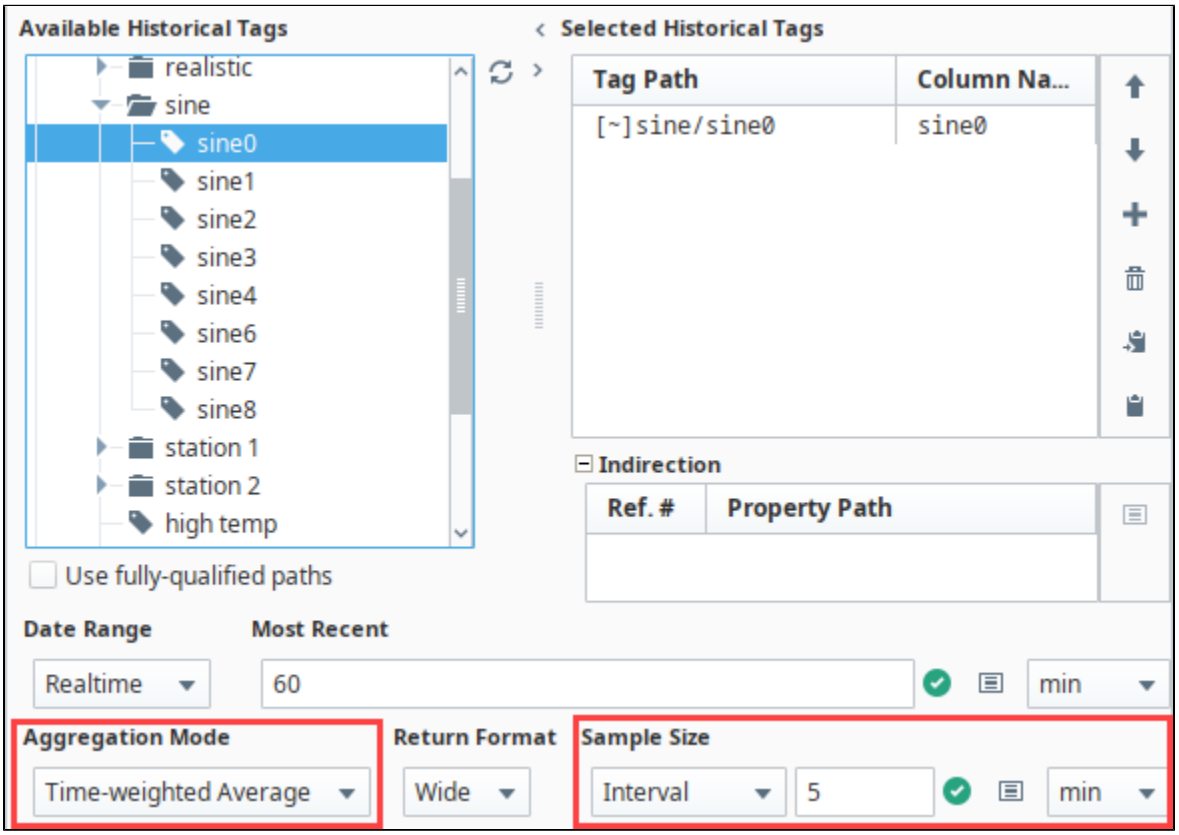
Note the **Insert Property** icon next to **Sample Size**. This allows a property binding to determine the number of data points, so you could change the size to increase or reduce the amount data points on the chart from the client.

Interval

Where as the **Fixed** sample size will calculate time slices based on the date range, the **Interval** sample size allows you to determine the size of the time slices. This sample size will divide the date range by the interval size to determine the size of each slice. Because of this, it is recommended to use an interval that is evenly divisible by the date range. However, in the event that the date range is dynamic or user driven, interpolation will handle any partially built slices. Even though the binding may attempt to evenly distribute the slices, there may be an extra row that represents the current values as they are building an interval.

The image below shows a **Realtime** range of **60 minutes**. The **Aggregation Mode** is set to **Time-weighted Average**, and the **Sample Size** is set to **Interval** for **5 minutes**. This means that the binding will query for data ranging from 60 minutes ago to now (or whenever the binding last executed, in the case that polling has been turned off). That 60 minute window will be divided as evenly as possible into 5 minute time-slices, so there should around 12 time-slices. Each time slice will aggregate its value based on the time-weighted average of all values within that slice.

The example uses a Realtime range, but a Historical range could easily be used instead.



Note the **Insert Property** icon next to **Sample Size**. This allows a property binding to determine the number of data points, so you could change the size to increase or reduce the amount data points on the chart from the client.

Return Format

Return format dictates how the requested data will be returned. The options are "wide" (default), in which each Tag has its own column, and "tall", in which the Tags are returned vertically in a "path, value, quality, timestamp" schema.

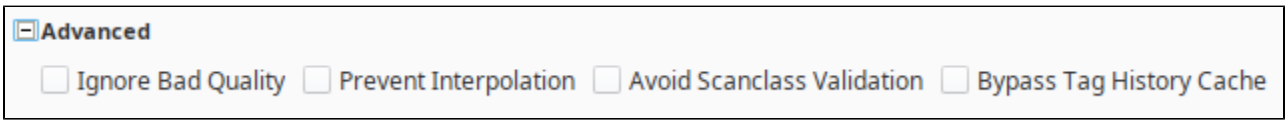
Advanced Options

These options affect the query results in more subtle ways.

- **Ignore Bad Quality** - Only data with "good" quality will be loaded from the data source.
- **Prevent Interpolation** - Requests that values not be interpolated, if the row would normally require it. Also instructs the system to not write result rows that would only contain interpolated values. In other words, if the raw data does not provide any new values for a certain window, that window will not be included in the result dataset.
- **Avoid Scan Class Validation** - "Scan class validation" is the mechanism by which the system determines when the Gateway was not running, and returns bad quality data for these periods of time. By enabling this option, the scan class records will not be consulted, which can improve performance, and will not write bad quality rows as a result of this check.

> The following feature is new in Ignition version **8.1.5**
> Click here to check out the other new features

- **Bypass Tag History Cache** - A "Bypass Tag History Cache" checkbox was added on Vision Tag History bindings which allows the option for the Tag History cache to be ignored on a per-binding basis. When the Tag History cache is enabled, query start/end dates are aligned to an existing subcache for performance which may not contain the most recent values. This can lead to a scenario where the last datapoint returned isn't reflective of realtime values if the Tag History binding date range includes the current time. Enabling the "Bypass Tag History Cache" checkbox prevents this scenario.

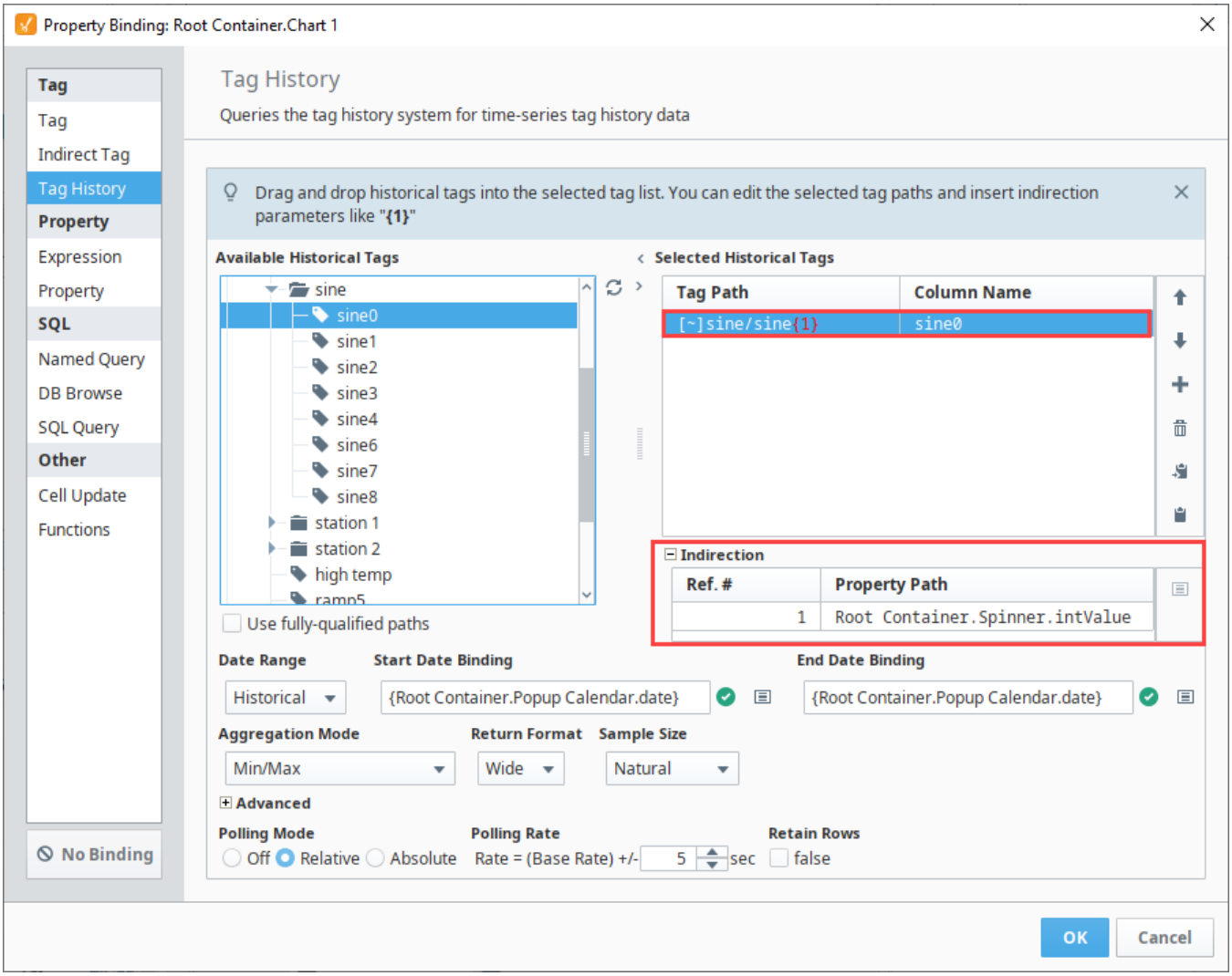


Tags Historian information is often easiest to work with in the Easy Chart component, which handles all of these options automatically.

Indirect Tag History Binding

The Tag History Binding can be made indirect by using Indirection parameters in the Tag Paths of the Selected Historical Tags. This works similarly to the Indirect Tag Binding, which uses Indirection References within the Tag Paths to substitute something into the path. Valid Indirection parameters consist of a reference number within curly braces. Simply type the Indirection parameters into a Tag path, selected by double-clicking. In this case, we will enter {1} as our Indirection parameter.

All valid parameters will appear in the Indirection table. For this example, the Tag Path points to the Spinner component for the indirection parameters.


Property Binding: Root Container.Chart 1
Tag
Tag
Indirect Tag
Tag History
Property
Expression
Property
SQL
Named Query
DB Browse
SQL Query
Other
Cell Update
Functions
No Binding
Tag History
Queries the tag history system for time-series tag history data
Drag and drop historical tags into the selected tag list. You can edit the selected tag paths and insert indirection parameters like "{1}"
Available Historical Tags
sine
sine0
sine1
sine2
sine3
sine4
sine6
sine7
sine8
station 1
station 2
high temp
ramp5
Use fully-qualified paths
Selected Historical Tags
Tag Path
Column Name
[~]sine/sine{1}
sine0
Indirection
Ref. #
Property Path
1
Root Container.Spinner.intValue
Date Range
Start Date Binding
End Date Binding
Historical
{Root Container.Popup Calendar.date}
{Root Container.Popup Calendar.date}
Aggregation Mode
Return Format
Sample Size
Min/Max
Wide
Natural
Advanced
Polling Mode
Polling Rate
Retain Rows
Off
Relative
Absolute
Rate = (Base Rate) +/-
5
sec
false
OK
Cancel

Expression Binding in Vision

Binding Properties to the Outcome of an Expression

An expression binding is one of the most powerful kinds of property bindings. It uses a simple expression language to calculate a value. This expression can involve lots of dynamic data, such as other properties, Tag values, results of Python scripts, queries, and so on. Any time information needs to be massaged, manipulated, extracted, combined, split, and so on, expressions can get the job done.

Event Based and Polling

Expression bindings fall into the unique category of having the possibility of using both Events and Polling to update. How an expression updates depends on what is being done in the expression. Expression bindings will always update immediately when the window they are in is opened. When they update again depends on if they are driven by events or polling. Typically, expressions are driven by events. If the expression was adding multiple values together, then when one of those values changed the expression would update, regardless of whether those values came from other properties or Tags. However, the expression function has some unique functions that can update at a set rate such as the now() function. When these functions are used within the expression, the expression binding will update based on the specified polling rate.

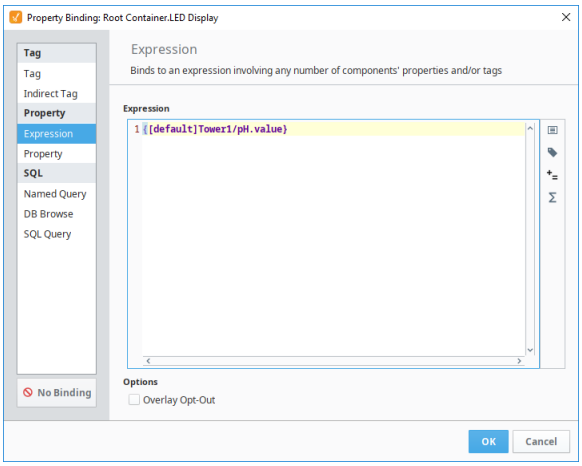
Using Expression Bindings

The expression language has lots of tools available that help calculate a specific value such as built-in expression functions, multiple operators, and the ability to reference Tags. While all of these can be manually typed into the expression, the expression binding window makes it easy to reference these options.

Helper Icons

To the right of the expression binding window, there are four icons that can be used to reference specific objects or functions easily.

- **Properties** - Places a property reference into the expression at the cursor, pulling in that property's value into the expression at the time of evaluation.
- **Tags** - Places a Tag reference into the expression at the cursor, pulling in that Tag's value into the expression at the time of evaluation.
- **Operators** - Places the operator into the expression at the cursor. Mostly used as a reference to what operators are available for use.
- **Functions** - Places the function into the expression at the cursor. Can be used as a reference for what functions are available, as well as the parameters the function is expecting.



Expression Binding Examples

Example 1

On this page ...

- Binding Properties to the Outcome of an Expression
  - Event Based and Polling
- Using Expression Bindings
  - Helper Icons
  - Expression Binding Examples

Expression Binding

Watch the Video

You have a button that starts a batch, but you only want to let it be pressed after the operator entered a scale weight. An expression binding can be set up on the enabled property of the button:

```
{Root Container.EntryArea.WeightBox.doubleValue} > 0.0
```

Example 2

You want to display a process's current state, translating a code from the PLC to a human-readable string. The examples below will yield the same results, but in different ways.

This first example uses nested "if" statements to produce the string. Notice that the false return for the first "if" statement is another "if" function, and the same with the second "if" function. Since the "if" function can only do simple if/than/else logic, this method allows us to do an if/than/else if/else.

```
if ({CurrentProcessState} = 0, "Not Running",
if ({CurrentProcessState} = 1, "Warmup phase - please wait",
if ({CurrentProcessState} = 2, "Running", "UNKNOWN STATE")))
```

This example will yield the same result as the previous example, but works differently. Instead of using multiple functions, this example uses a single switch function to decide which string to use.

```
switch ({CurrentProcessState},
   0,1,2,
   "Not Running",
   "Warmup phase - please wait",
   "Running",
   "UNKNOWN STATE")
```

For more examples, see Expression Overview and Syntax.

Named Query Bindings

Binding Properties to a Named Query

The Named Query binding is where you can configure a property to call a Named Query that you had previously created in the project. Using Named Query binding instead of a SQL Query or DB Browse binding helps to make your project more secure due to the built-in Security Zone and User Role restrictions.

Polling Mode

Each Named Query binding type will use polling to determine when to update the results and run the query again. The Polling Mode dictates how often the query will execute, and works in a similar fashion to polling on other bindings.

Creating Named Queries

In order to use a Named Query, first one has to be created. You might already have some that another developer created but if not, you will have to make one by going to the Named Query section in the Designer or converting a SQL query.

On this page ...

- Binding Properties to a Named Query
- Polling Mode
- Creating Named Queries
- Using Named Queries on Dataset Properties
- Using Named Queries on Scalar Properties

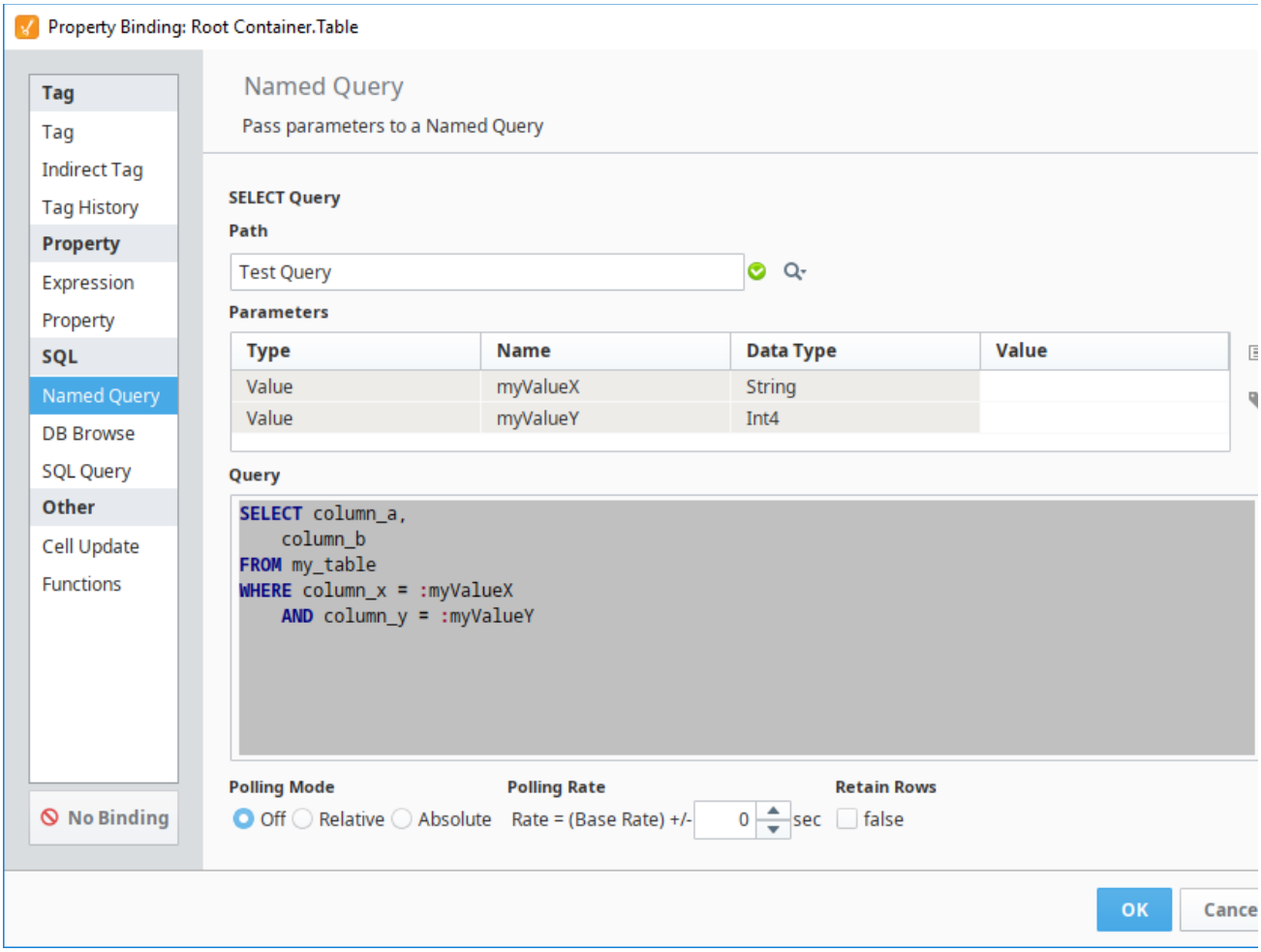
Named Query Binding

Watch the Video

Using Named Queries on Dataset Properties

The majority of your Named Query bindings will most likely be on a dataset type property. When placed on a Dataset type property, only a single Named Query needs to be specified. An explanation of the various fields on the binding are detailed below:

- **Path**: Here you can enter in the path to the Named Query. Click on the Search icon to view a list of available Named Queries.
- **Parameters**: Here you can see a table of all defined Named Query parameters. You can pass in property or Tag values to the parameters by first highlighting the parameter and then selecting either the **Insert Property** icon or the **Tag** icon
- **Query**: The Query section shows what Named Query looks like. Note that you can't modify the query on this page.
- **Polling Mode**: Here you can set the Polling Mode of the Named Query binding based on the Polling Rate.
- **Retain Rows**: If true, any rows that you have returned within the Designer will be saved along with the window. This may slow window load times.



Using Named Queries on Scalar Properties

When placed on a non-dataset type property (such as a String or Integer), then the Named Query binding allows for a second Named Query to be specified in the case that the user can update the value on the property. This provides an opportunity to both return and update values in the database from the same component.

The configuration is very similar to a Named Query binding on a dataset property. You need to specify a Named Query path, set up your Parameters, and choose a Polling Mode. You can finish setup at this point, leaving the update query disabled so that the property will simply pull the value from the database.

However, if you want the binding to be bidirectional, you need to specify an UPDATE query. This works similar to a SELECT query, in that you need to select the path to the Named Query and set up any Parameters. However, it is important to make sure that the Named Query chosen for the UPDATE query is in fact set up as an UPDATE query by setting the Query Type property on the Authoring section of the Named Query.

> The following feature is new in Ignition version **8.1.27**
> Click here to check out the other new features

Now when you configure an UPDATE query, clicking the **This** button will insert `{this}` as a Value for the selected parameter.

Related Topics ...

- Cell Update Bindings
- Named Queries
- Named Query Parameters

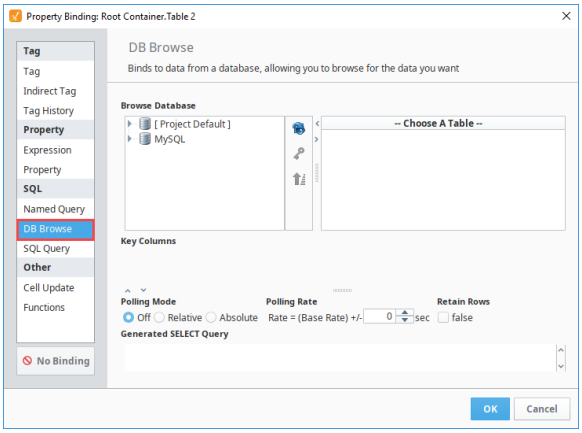
DB Browse Bindings

Binding Properties to Database Tables

The DB Browse binding is technically equivalent to the SQL Query binding, except that it helps write the queries for you. Using the Database Browse binding type, you can pick the table from a list of tables in each database that you want to pull content from. If you have a fixed range of data you need to return, simply select it in the table, and watch the query get generated.

In the Browse Database tree, you can choose which columns in your table should act as your keys (these columns get put in the WHERE clause based on your selection) and which columns should be used to sort the data (these columns are put in the ORDER BY clause).

> This binding type also serves as a convenient jumping-off point for the more flexible SQL Query binding. Construct the basic outline of your query in the DB Browse section, and then select the SQL Query radio button to convert to the new binding type. Your query will be retained and can then be modified manually.



On this page ...

- Binding Properties to Database Tables
- Configuring the Binding
  - Key Column
  - Sort Order
- Dynamic Filters
- Scalar Query Update

DB Browse Binding

Watch the Video

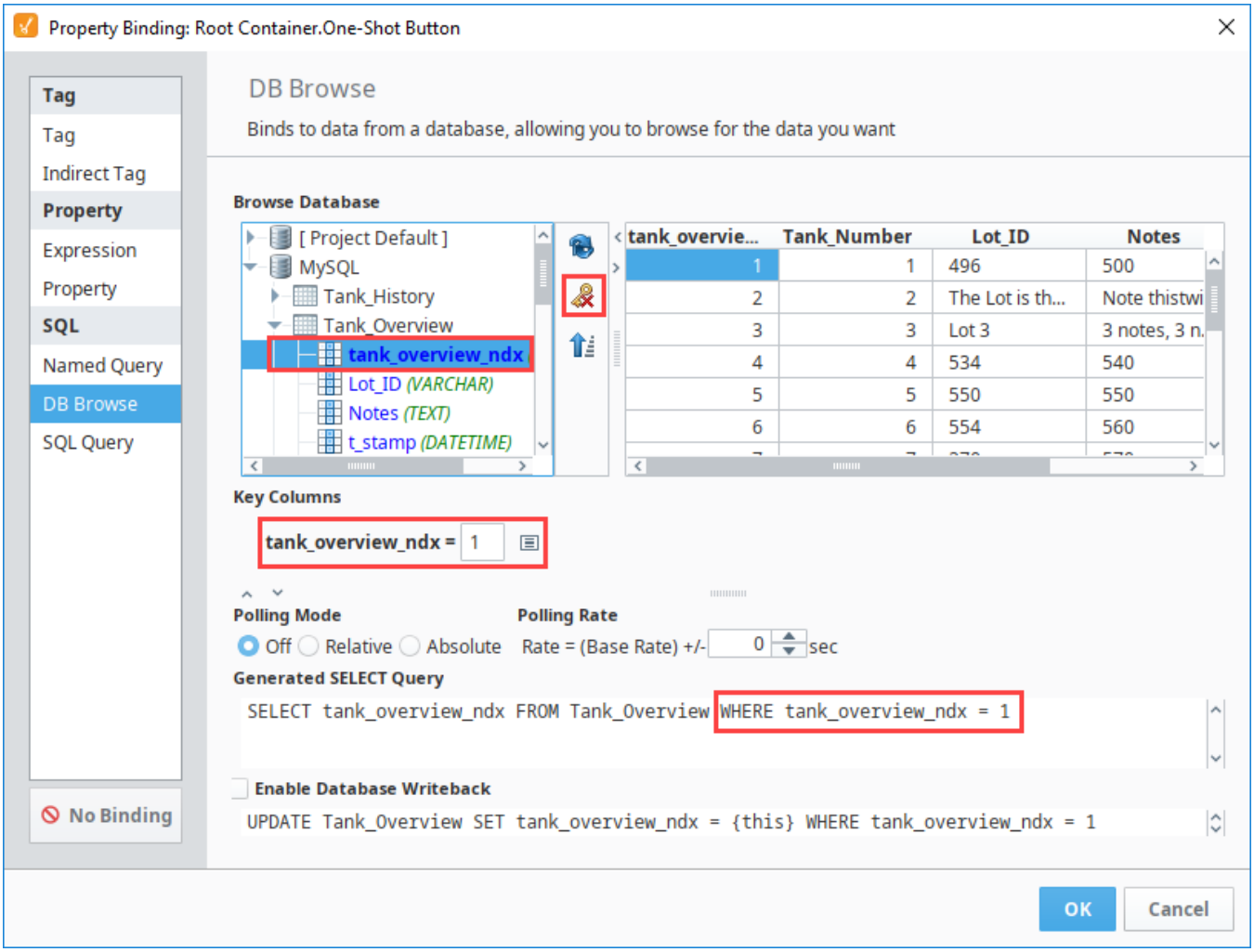
Configuring the Binding

After selecting a table in the Browse Database tree, you can customize which columns the query is selecting by selecting one or more columns under the table to select just the highlighted columns, or selecting the table to use the * symbol to select all columns.

Key Column

The DB Browse binding has the ability to designate key columns within the query. A key column is used within the select query's where clause, and can be given a value. A column is denoted as a key column when it has a key symbol next to it.

Clicking the Key icon to the right of the Browse Database tree will designate a column as a key column. Alternately, if the highlighted column is already a key column, then clicking the Key icon will remove that column as a key column.



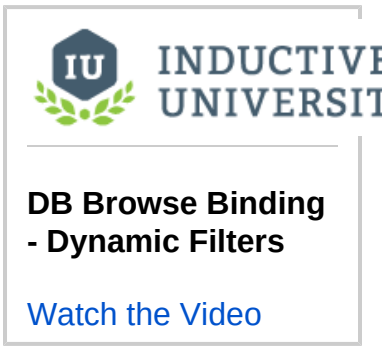
Sort Order

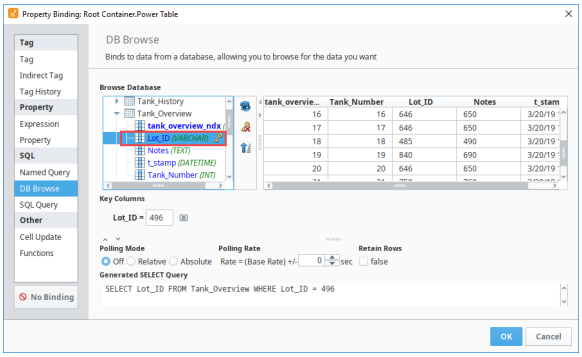
In the DB Browse binding, you can also sort data in ascending or descending order. Select the column that you want to sort by and click the Sort icon. Multiple columns can be used for sorting.

Dynamic Filters

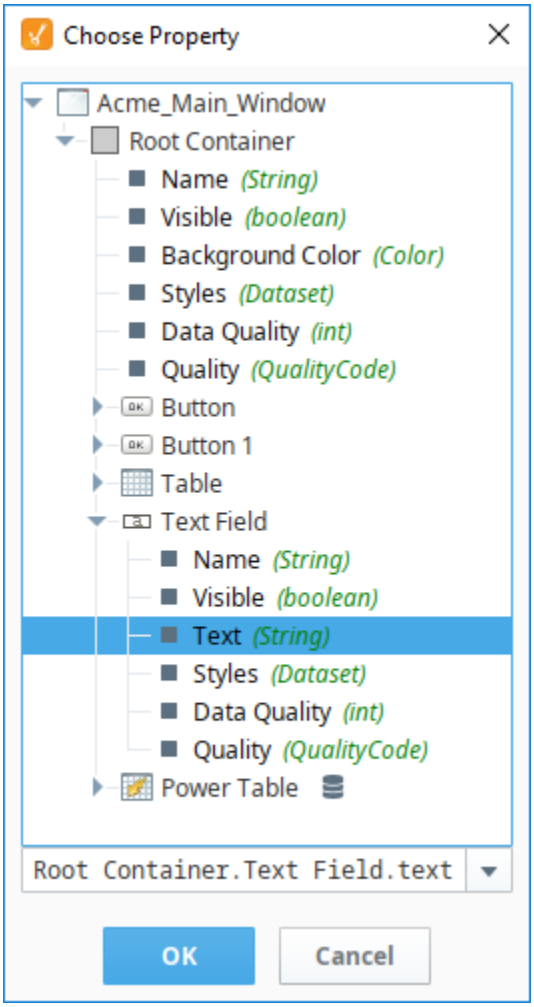
DB Browse bindings also give the ability to bind a property to a key column to allow for dynamic filtering of the returned data. Simply click the binding icon next to the key column field. This allows you to give the operators some control over the data they are seeing.

1. In the Designer, drag a Table component and a Text Field component on a window.
2. With the Table component selected, click the Binding icon next to the **Data** property.
3. Chose DB Browse under Binding Types > Database.
4. Let's pull all the data from this Table except for the id, and filter on state. Remove the Key fro m the **id** column and place it on the **state** column.

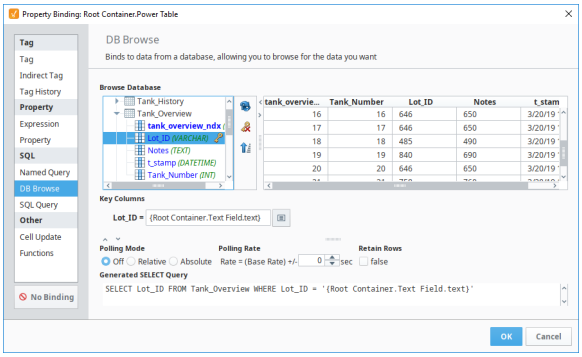




5. Select the **Tank_Number**, **Lot_ID**, **Notes**, and **t_stamp** columns. You can do this with Control+Click, or by clicking and dragging in the results table in the upper right.
6. Instead of statically typing in a value like we in did in the above example, let's make it dynamic using the Text Field. Click the Insert Property Value  icon next to the value in the **Key Columns** section, and select the Text property of the Text Field.



7. Notice there is now a property reference in the **Key Column** as well as the **Generated SELECT Query**.



8. Click **OK** to confirm the binding.
9. Put the Designer into Preview mode.
10. Enter the Lot_ID that you want to view and the Table will update to display just the data for that Lot ID.

296

Tank_Number	Lot_ID	Notes	t_stamp
23	296	300	2019-03-20 12:36:00
24	296	300	2019-03-20 12:36:00

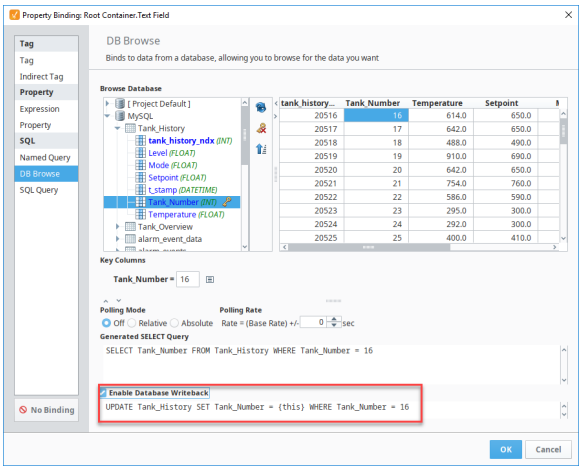
646

Tank_Number	Lot_ID	Notes	t_stamp
16	646	650	2019-03-20 12:36:00
17	646	650	2019-03-20 12:36:00
20	646	650	2019-03-20 12:36:00

Scalar Query Update

Similar to the SQL Query Binding, the DB Browse Binding has the ability to become bidirectional by doing a database write back when the property being bound is a non-dataset type. In this case, the select query should be configured to only return a single row from a single column.

For example, this option can provide a single value to the Text property of a Text Field component. If you check the **Enable Database Writeback** checkbox, then any user input will write back to the database. This will automatically generate an update query that will push the input value into the database from the location where the original value was retrieved.



SQL Query Binding - Scalar Query and Update

Watch the Video

SQL Query Bindings in Vision

Binding Properties to a SQL Query

The SQL Query binding is a polling binding type that will run a SQL Query against any of the database connections configured in the Gateway. It is very similar to the DB Browse binding type in that both query a database to return data. The difference is the SQL Query Binding can manually be modified. This is useful for complex queries where you will use the more advanced functions of the SQL language that can not be accomplished with the DB Browse binding.

> **Pro Tip!**
>
> The query that gets generated by the DB Browse will transfer over to the SQL Query binding when you switch the binding type. It may be useful to build the basic query structure with DB Browse first, then switch to SQL Query binding to modify the query to fit your needs.

Dataset Binding

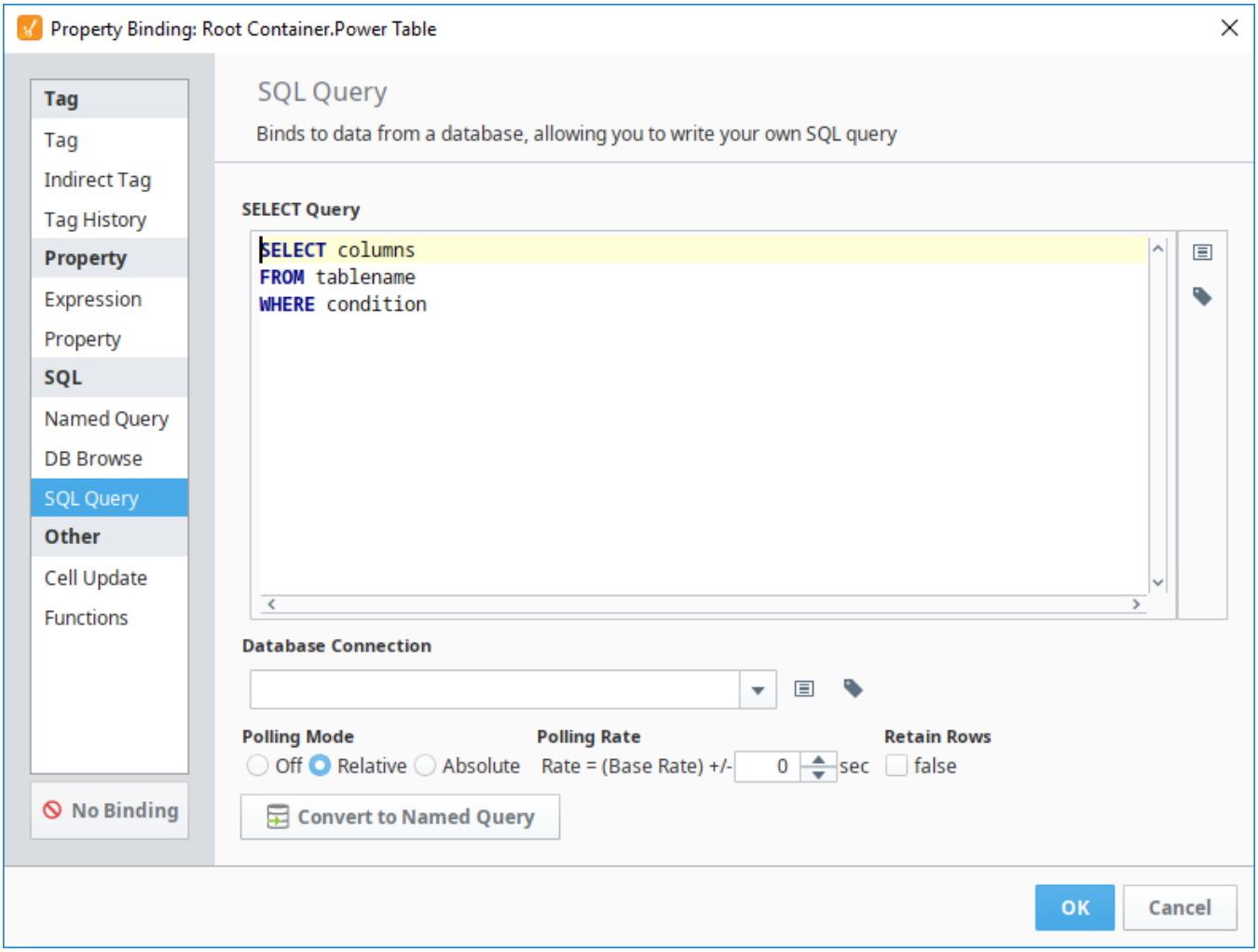
The majority of SQL Query bindings will return a dataset. These will return many rows with multiple columns. For example, showing all customer details from a certain account, or all downtime events in the facility. This type of SQL binding is used on properties of type **dataset** like the Data property on a Table component.

On this page ...

- Binding Properties to a SQL Query
- Dataset Binding
- Dynamic Filters
  - Example
- Scalar Query Update
- Scalar Query Fallback
- Stored Procedures
- Named Query Conversions

SQL Query Binding

Watch the Video



Dynamic Filters

Using the curly brace {} notation, you can include the values of component properties (within the same window) and Tag values inside your query. This is a very common technique to make your query dynamic. The values of the property or Tag represented are simply substituted into the query where the braces are.

Because the substitution is direct, you'll often need to add quotes to literal strings and dates to make your query valid. If you're getting errors running your query complaining about syntax, it is important to realize that these errors are coming from the database, not from Ignition. Try copying and pasting your query into the Query Browser and replacing the braces with literal values.

Example

A common requirement is to have a query filter its results for a date range. You can use the Date Range component or a pair of Popup Calendar comp onents to let the user choose a range of dates. Then you can use these dates in your query like this:

SQL - SQL Query Binding with Parameter References

```
SELECT
   t_stamp, flow_rate, amps
FROM
   valve_history
```

```
WHERE
   t_stamp >= '{Root Container.DateRange.startDate}' AND
   t_stamp <= '{Root Container.DateRange.endDate}'
```

Notice the single quotes around the braces. This is because when the query is run, the dates will be replaced with their literal evaluations. For example, the actual query sent to the database might look like this:

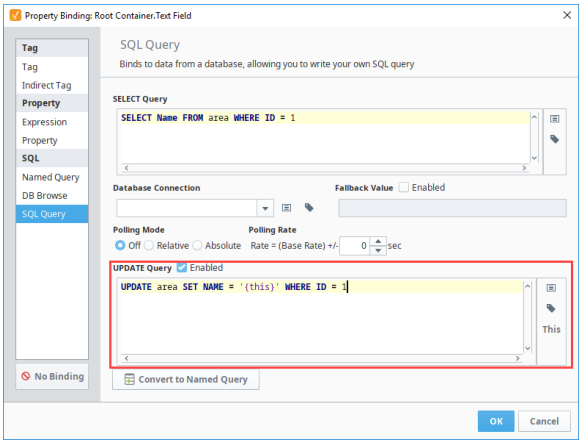
SQL - SQL Query Binding with the Values Replaced

```
SELECT
   t_stamp, flow_rate, amps
FROM
   valve_history
WHERE
   t_stamp >= '2010-03-20 08:00:00' AND
   t_stamp <= '2010-03-20 13:00:00'
```


It is important to use single quotes and not double quotes (t_stamp = "2010-03-20 08:00:00") because these mean something different in certain databases like Microsoft SQL Server.

Scalar Query Update

You can bind a non dataset type property to a SQL query to allow a singular value to be returned from the database with a scalar query. Now instead of returning multiple rows and columns, the query returns a single value from the first row of the first column. These types of SQL Query bindings can also be used to update the database on input components like a Text Field. Essentially, we mimic the bidirectionality of Tag and property bindings by adding in an update query to run whenever a value gets entered into the property with the binding. In our update query, we use the special parameter **{this}** to denote the new value from the bound property. If **{this}** is a string, it needs single quotes around it.



Take a Text Field with a simple query on it.

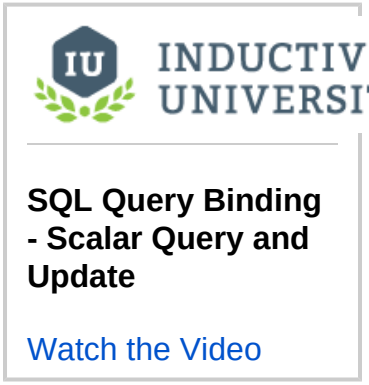
SQL - Simple Select Query

```
SELECT Name FROM area WHERE ID = 1
```

This will return a single value that can populate our text field. We then enable the Update Query at the bottom of the Property Binding window, and add in the update query.

SQL - Using Ignition's 'this' Keyword

```
UPDATE area SET Name = '{this}' WHERE ID = 1
```

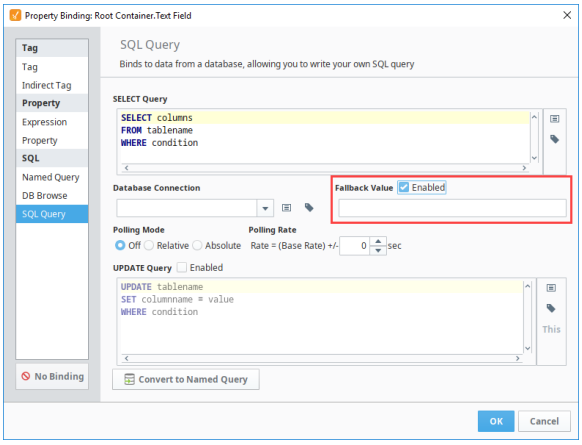


After confirming the binding, we can see that our text field contains the value from the database and will update the database cell if we enter in a new value into the text field. This is a good way to alter very specific cells in a database table.

Scalar Query Fallback

If the property that is being bound is a scalar datatype (that is, not a Dataset), the value in the first column in the first row of the query results is used. If no rows were returned, the binding will cause an error unless the Use Fallback Value option is selected. The value entered in the fallback value text box will be used when the query returns no rows.

When binding a Dataset to a SQL Query, no fallback value is needed, because a Dataset will contain zero rows.



Stored Procedures

While queries can manually be written on a SQL Query binding, SQL Stored Procedures may also be called from a SQL Query Binding.

Note: The exact syntax is highly dependent on the type of database you are using.

For example, calling a Stored Procedure from MySQL would involve using the CALL command, while SQL Server utilizes the EXEC command.

SQL - MySQL Stored Procedure Call

```
CALL retrieve_daily_total
```

SQL - SQL Server Stored Procedure Call

```
EXEC retrieve_daily_total
```

Named Query Conversions

You can convert the SQL Query created here to a Named Query. For more information, see Named Query Conversions.

Cell Update Bindings

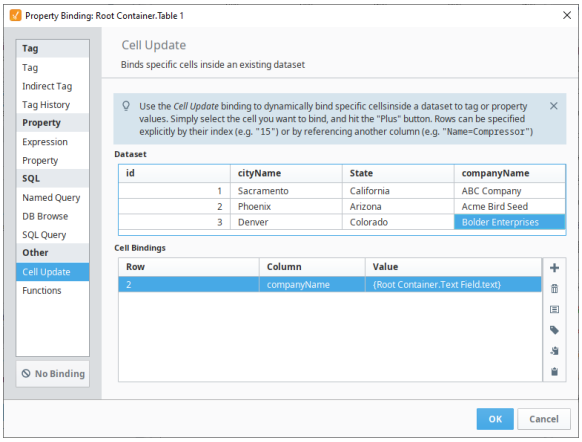
Binding a Dataset Property to Realtime Values

The Cell Update binding enables you to easily make one or more cells inside a dataset dynamic. This is particularly useful for components such as the Table to easily display realtime Tag information, or in the Linear Scale which stores configuration information in a dataset. The Cell Update binding allows you to bind a cell inside of a dataset to a Tag or to a property.

Cell Update Bindings work really well with the Easy Chart component, allowing you to indirectly show Tag history on the same Easy Chart. See Indirect Easy Chart for more details.

Note: The Cell Update binding type will only appear when setting up a binding on a **dataset** property, otherwise, it won't be present.

The cell update binding has a few tools that help you make certain cells dynamic.



The top table called **Dataset** displays the original dataset, while the bottom **Cell Bindings** table displays each of the cell updates that will be occurring. Each Cell Update binding can have multiple cell updates happening on a single dataset. To the right of the Cell Bindings table are the buttons that are used to create the Cell Bindings:

- **Add Row** - Adds a new empty row to the Cell Bindings table, which can be customized with a new Cell Binding. If a cell is selected in the Dataset table, it will instead add a row with the Row and Column values already filled in with the appropriate values.
- **Delete Row** - Removes one of the rows in the Cell Bindings table.
- **Insert Property** - Inserts a property reference into the Value cell of the selected row.
- **Insert Tag** - Inserts a Tag reference into the Value cell of the selected row.
- **Copy** the current selection to the clipboard.
- **Pastes** the contents of the clipboard into the current context.

Adding Values to the Cell Bindings Rows

Each row in the Cell Bindings table needs three values to work properly: a **Row identifier** which is used to figure out which row the cell binding is on, a **Column identifier** to determine the column of the cell, and a **Value** which will replace the original value of the cell. These three values can pinpoint a specific cell, and replace its value with the dynamic Tag or property value specified. The examples in this section use the dataset:

id	cityName	State	companyName
1	Sacramento	California	ABC Company
2	Phoenix	Arizona	Acme Bird Seed
3	Denver	Colorado	Bolder Enterprises

On this page ...

- Binding a Dataset Property to Realtime Values
- Adding Values to the Cell Bindings Rows
  - Row Column
  - Column Column
  - Value Column
- Realtime Tag Values in a Table Example
- Adding a Realtime Indicator to the Linear Scale Example

Caution: No two rows should specify the same cell, as this will throw an error.

Row Column

The Cell Bindings **Row** column can be filled one of two ways.

Row Index

The easiest way is by specifying the row index of the cell that you want to target for an update. It is important to remember that the row index is zero based, so the index of the first row is always 0. There can be multiple rows, each with the same row index, as long as the Column is different. The order of the rows in this case does not matter. As the image below show three rows, specified with a row index of 0, then 5, then 0 again.

0	Boolean Column	
5	String Column	
0	Int Column	

Column Value

The other way to identify the row that the intended cell belongs to, is to use the value of a different column. This is done using the syntax:

```
columnName=value
```

Where the columnName is the name of the column and the value is the value that needs to match. The columnName and value are both case sensitive, so you will need to use care when filling in these values.

0	Boolean Column	
5	String Column	
String Column = Test1	Boolean Column	

There are three things that make using the Column Value unique. The first is that there is the possibility for duplicates to happen. Take the image above, with both a Row index of 0 and a Column Value of 'String Column=Test 1'. Looking at the original dataset, both of those point to the same row, and with both of them pointing to the Boolean Column, they both refer to the same cell in the dataset. This instance of duplicates will not throw an error, but instead will work. In this case, the updates happen from top to bottom, so the 'String Column=Test 1' would be what writes to the cell last and what ultimately gets displayed.

The second is that there is the potential for multiple possible matches to that evaluation. For example, if I had this in the Row value:

```
Boolean Column=True
```

This could potentially be true for multiple rows of my dataset, in which case, the binding will apply to all of them. This allows you to change multiple cell values that should all be the same.

The final thing that makes the Column Value unique is that it itself is ultimately dynamic. The 'Boolean Column=True' Column Value matches the rows at index 1, 3, 4, 7, and 9 in the original dataset. However, if any of those were to change to False through another cell update, then that row would no longer be updated as part of this cell update. Conversely, if any of the currently False values were to change to a True, then those rows would fall under this cell update and the appropriate cell will be updated.

Column Column

The Cell Bindings column named **Column**, or **Column column**, expects the name of a column that will match in the "Dataset" table above. These values are case sensitive, so care should be taken when entering them manually.

Value Column

The **Value column** is what will get pushed into the cell that is being updated. There are three possible types of values that can be placed in here.

Static Value

A static value can be written in here. This will overwrite the existing value of the cell with whatever static value is in the cell update. This does defeat the purpose of using the cell update though, since the static value could just be placed directly in the original dataset.

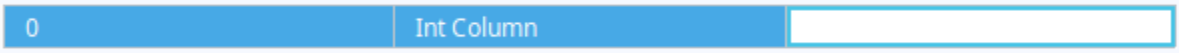
Tag or Property Reference

A Tag or Property reference can be used for the cell updates value, updating the value of the cell whenever the value of the Tag or Property changes. The Tag or Property Selectors to the right of the Cell Bindings table can be used to add in the reference.

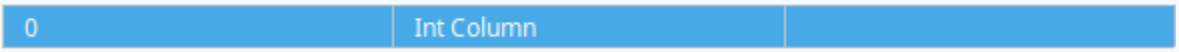
Note:

When adding a Tag or Property reference, be sure that the cell is only selected and that the cursor is not placed in the Value cell. The cursor in the cell is used for typing in a static value, and trying to enter in a Tag or Property reference will not work.

This is used to enter in static text:



This accepts Tag and Property references:



Both Static Values and a Reference

The Value column can also accept a combination of a reference and static values. This allows you to build unique strings or numeric values within the Value cell. The syntax used is:

```
numbers1234{tagOrPropertyReference}orcharacters
```

The reference value will be concatenated into the Value at the location specified. For example, if my Value was

```
{MemoryTags/IntegerTag}000
```

and the IntegerTag had a value of 5, the updated cell value would be 5000. This method of combining both a reference and a static value is great for updating Tag paths, such as in an Easy Chart's Tag Pens dataset.

```
[~]Motors/Motor {Root Container.MotorNumber}/Amps
```

Here, the MotorNumber property on the root container is replacing the motor number inside of the Tag path. See Indirect Easy Chart for more details.

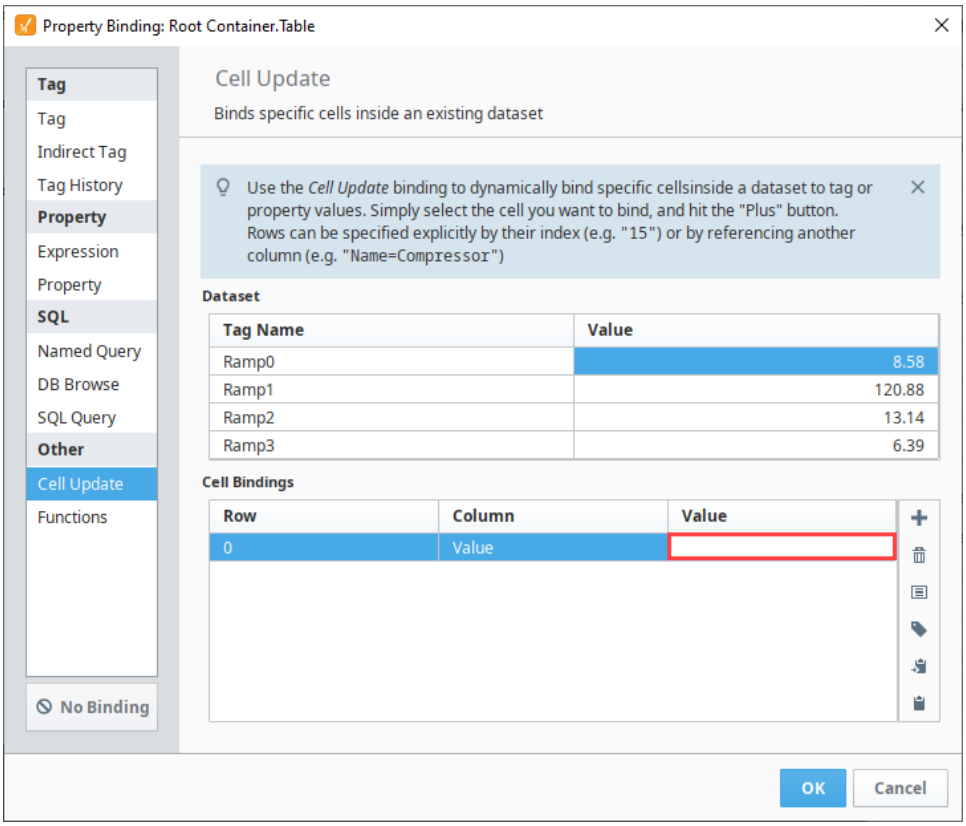
Realtime Tag Values in a Table Example

The Cell Update binding allows you to place the value of a Tag into a dataset easily. With a dataset property like the one on the Table component, getting updating values into it requires either a SQL query, or some constantly running script. With the Cell Update binding, that isn't necessary. The simple static table contains four rows for four different tags, where each has a value. As an alternative, you could use four numeric text fields and labels, but the Table component looks cleaner.

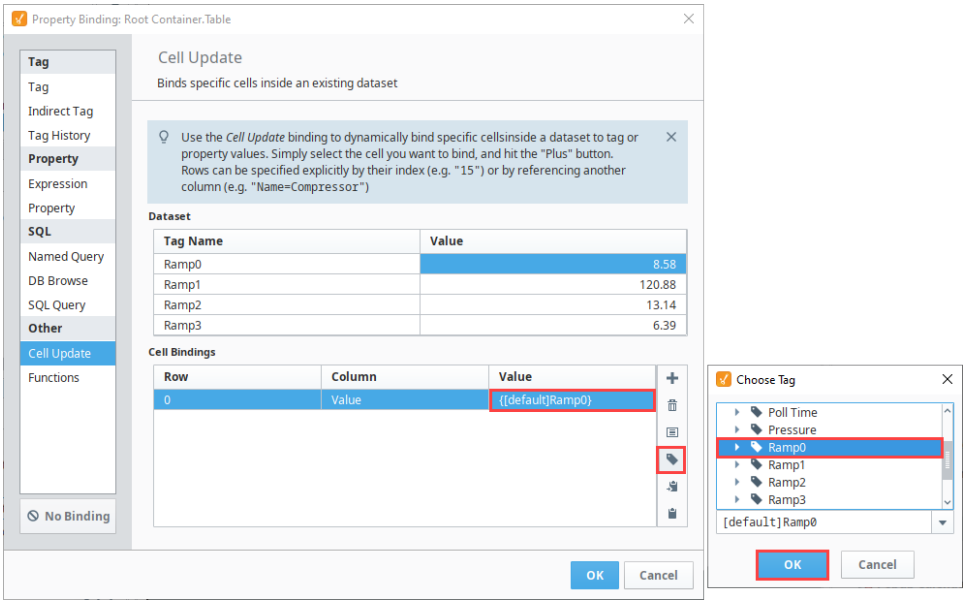
Tag Name	Value
Ramp0	8.58
Ramp1	120.88
Ramp2	13.14
Ramp3	6.39

Let's use the table above table to create a Cell Update binding to get the pressure readings into the table.

1. Open the property **binding** icon on the data property.
2. Click the **Cell Update** binding type, and you'll see your dataset along with a **Cell Bindings** area.
3. Select the first value, and click the **Add** icon in the Cell Bindings area to add a row. You'll notice it added a row number, column name, and blank value cell because we selected a cell from above. What you place in the blank **Value** cell determines what the value will be in the table.



4. Select the first **Row** and first **Value** cell in the Cell Binding area. Use the **Tag** icon to the right to select the Tag reference (i.e., Ramp0). You can also manually type in your Tag reference, or place in a property reference instead.
5. Click **OK**.



6. Now, you have the value of that Tag to the Value column in row 0. Repeat Step 3 for the other three Tags so that the Cell Binding area looks like the image below, then click **OK**.

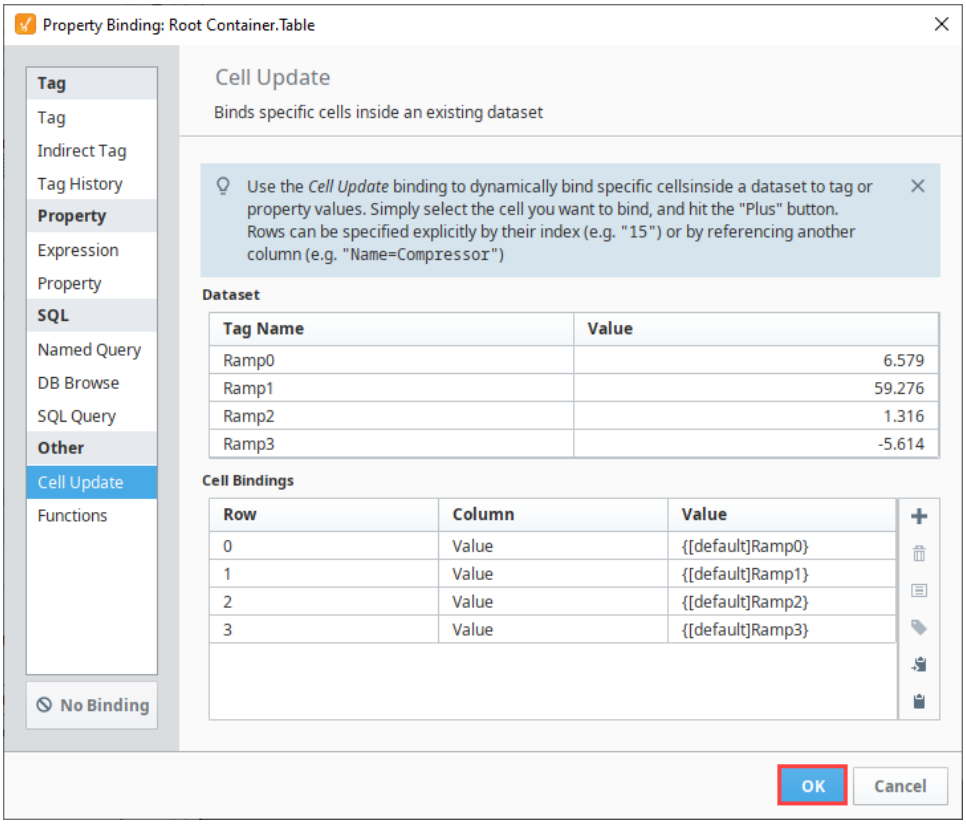


Table values are now updating with the value of their respective Tag.

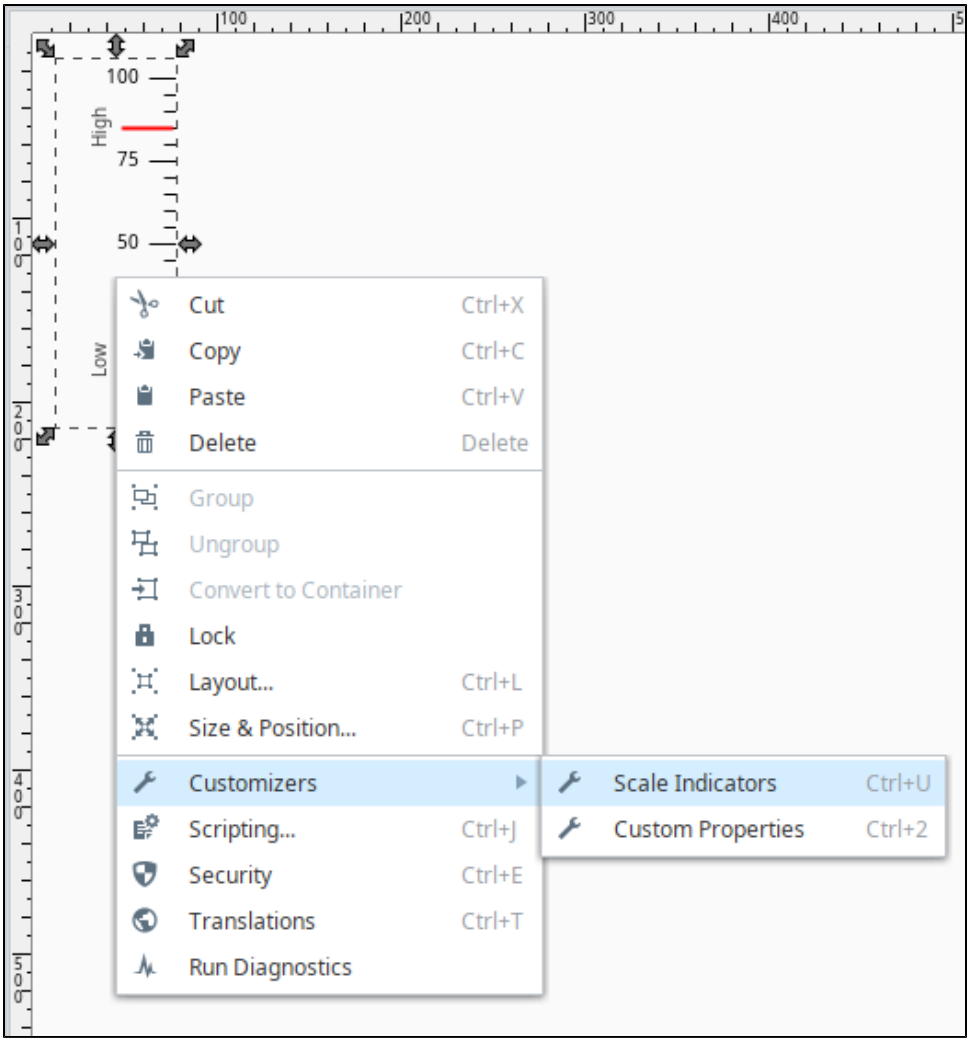
Tag Name	Value
Ramp0	2.59
Ramp1	23.44
Ramp2	1.52
Ramp3	5.06

Adding a Realtime Indicator to the Linear Scale Example

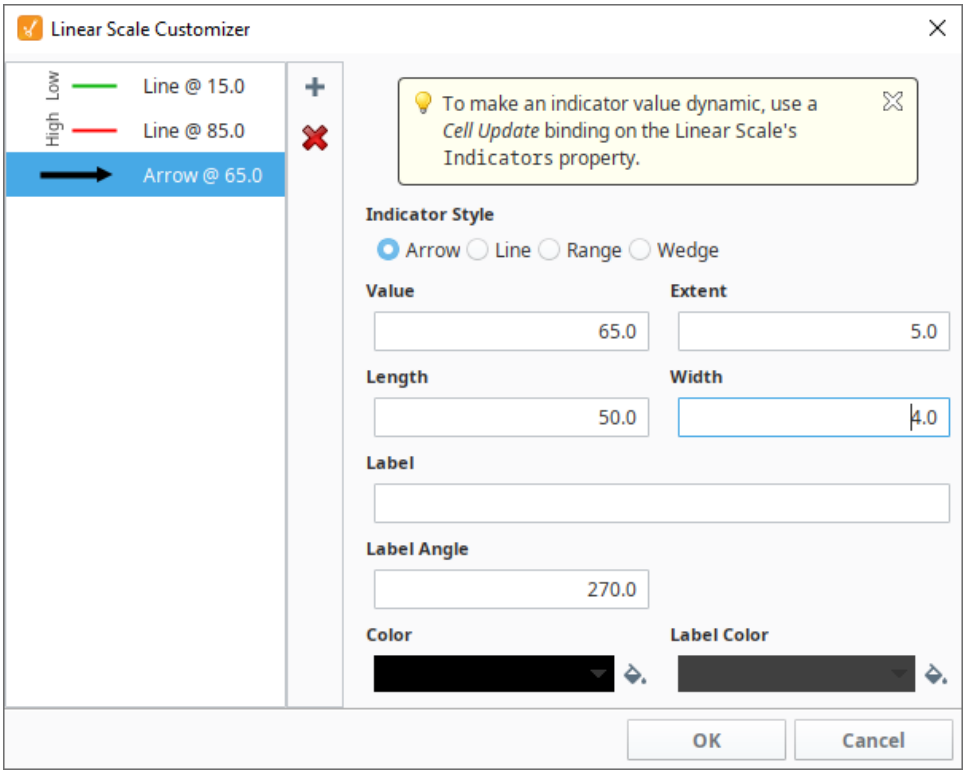
The Linear Scale is a component that has a special Scale Indicators customizer that allows you to configure setpoints. You can set up the value, color, style, and more for each of the indicators. During runtime, these values are normally static, as there is no way to set up a binding in the customizer. However, the customizer merely configures a dataset that the component uses to create the indicators. If you look at the Indicators dataset property of the Linear Scale, it has all of the properties that are configurable in the customizer. Changing them from the customizer will alter the dataset, and changing the dataset values will alter what you see in the customizer. Knowing this, we can set up a Cell Update binding to manipulate the values during runtime.

Let's add another indicator to the Linear Scale component and configure it.

1. Select the Linear Scale and right click on the **Customizers > Scale Indicators**.



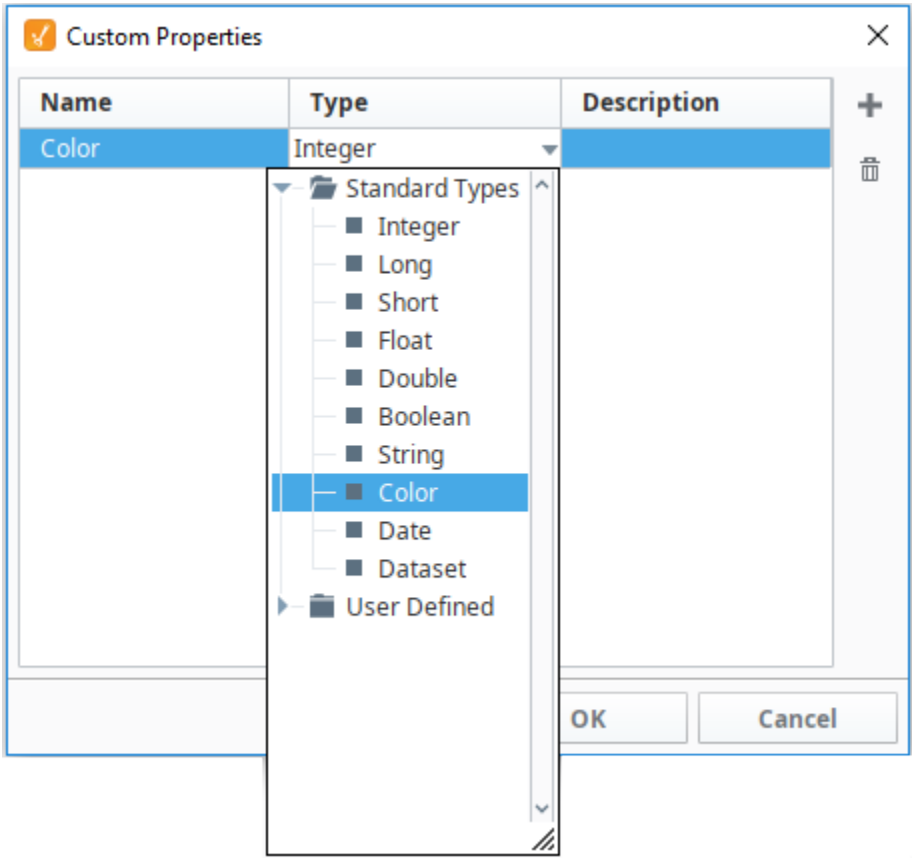
2. Click on the **Add** icon to add a new indicator.
3. Click the **Arrow** radio button and set the following properties for the Arrow.
    a. **Value:** 65.0
    b. **Length:** 50.0
    c. **Width:** 4.0



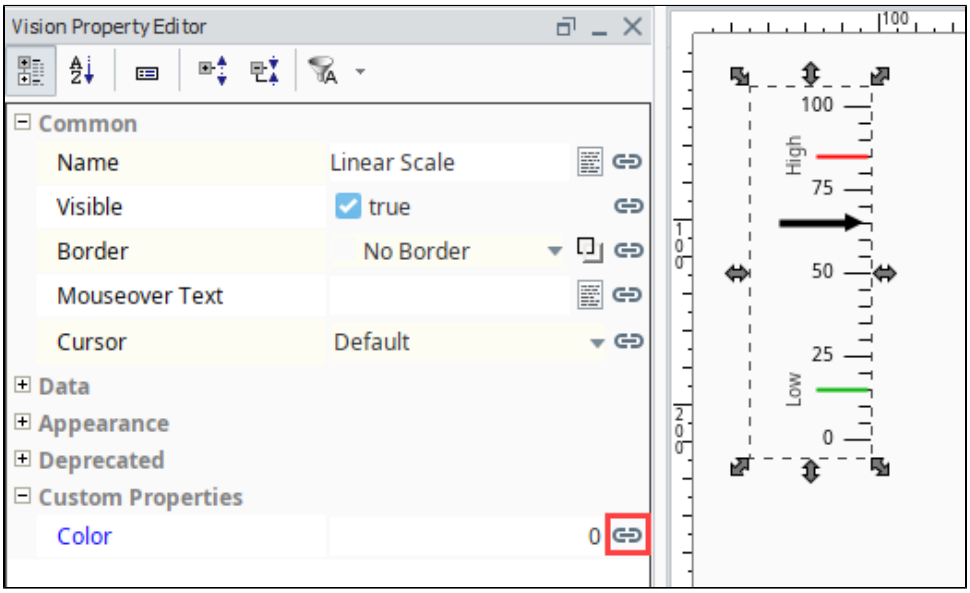
4. Click **OK**. This new indicator will be whatever the current value is.

5. In addition to changing the value of the realtime Indicator, you can also change its color based on its value. If the value is above the high indicator (i.e., 85), the Arrow will change to red, and if the value is below the green indicator (i.e., 15), it will change to green. To accomplish this, set up a custom property on the Linear Scale of type color:
    a. With the Linear Scale selected, right click on the **Customizers > Custom Properties**.
    b. Click the **Add** icon to add a property.
    c. Enter **"Color"** as the Name of the property.
    d. In the Type column, choose Color from the dropdown list.

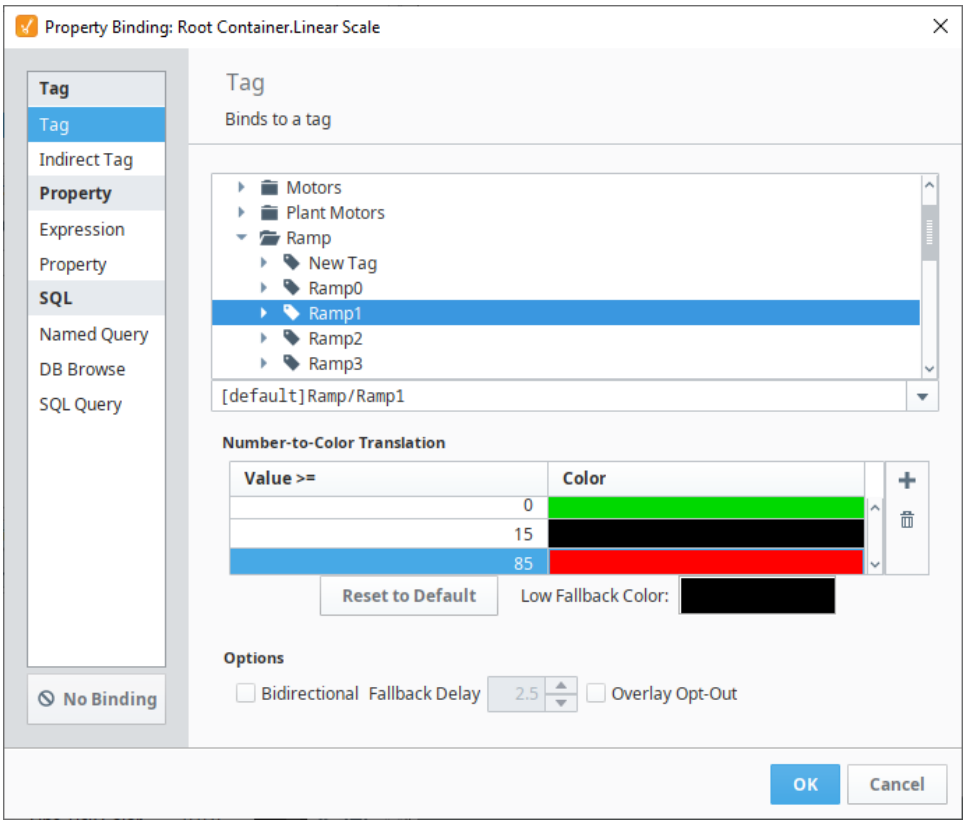
e. Click **OK** to save the custom property.



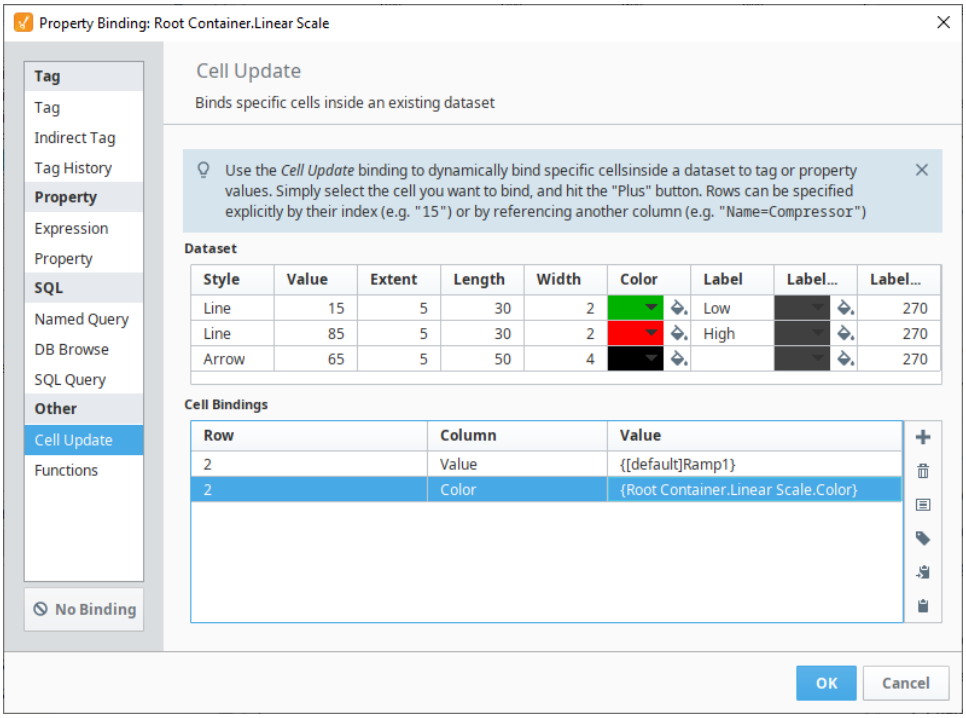
6. Next, we'll set up a Number-to-Color Tag binding on the custom **Color** property. Select the Linear Scale component, then click the **binding** icon next to the **Color** property.



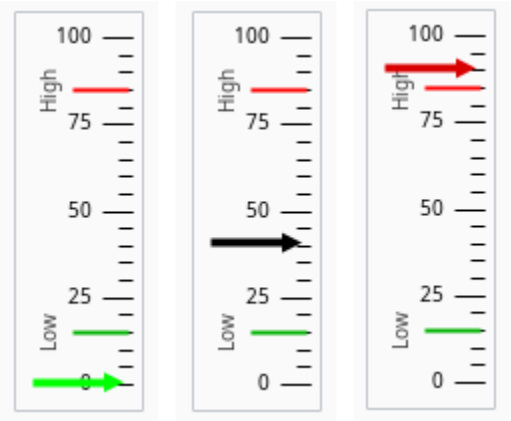
7. The values shown in the image below are based on the static setpoints that were set in the Linear Scale customizer.



8. Finally, set up a Cell Update binding on the **Indicators** dataset property of the Linear Scale. Notice below that both of the cell bindings are for the same row, just different columns. This is fine, as long as there aren't duplicate bindings for the same cell.
9. Bind the Value cell to the same Tag used in the custom property Color binding to get the value:
    a. Click the **Add** icon.
    b. Click the **Tag** icon .
    c. Select the **Ramp1** Tag,
    d. Click **OK.**
10. Bind the Color cell to the Color custom property we just created.
    a. Click the **Add** icon.
    b. Click the **Property** icon and select the **Color** property.
    c. Click **OK.**
11. Click **OK.**



Now, the Linear Scale has an Indicator that moves to show a realtime value and changes color whenever it goes outside the setpoints. The image below shows the indicator below the setpoint, within range, and then exceeding the setpoint.



Related Topics ...

- Function Bindings
- Property Bindings in Vision
- Tag Bindings in Vision
- Indirect Tag Bindings in Vision

Function Bindings

Binding Properties to Prebuilt Functions

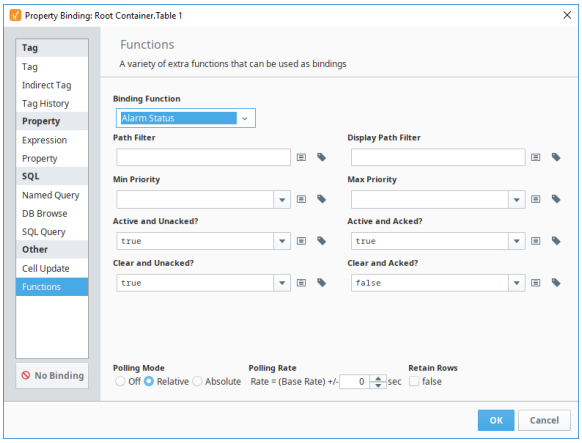
The Function Binding is a generic binding type that lets you bind a dataset property to the results of a function. It allows any of the function's parameters to be calculated dynamically via Tag and property bindings. The function that you choose determines the parameters that are available. The most common functions are the Alarm Status, Alarm Journal, and Audit Log functions, but there may be more depending on the modules you have installed.

Using Function Bindings to Customize

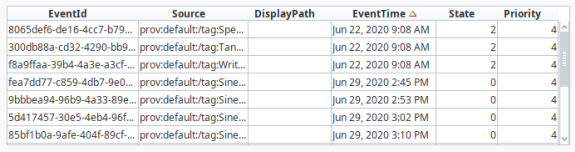
While there may already be an Alarm Status Table component, it may not be able to customize to exactly fit your needs, but the Table component still offers customization options. When you use a Function Binding on the alarm status, you can pull that same data into a Table component, and then customize the table exactly how you need.

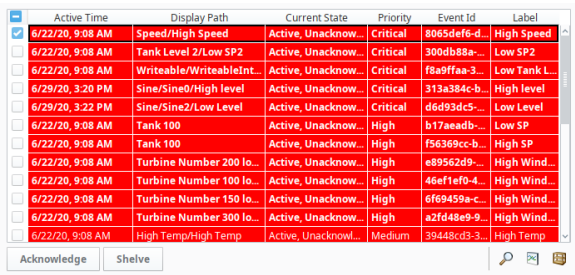
The following example shows the default settings for the Function binding. Notice how each field has Tag and property binding buttons next to them. This allows you to make this entire function dynamic by binding either Tag or property values to the different values of the function.

1. In the Designer, drag a **Table** component into your workspace.
2. Select the binding icon for the **Data** property.
3. This opens the property binding window. Select the **Functions** binding.
4. Select a **Binding Function** from the dropdown. In this example, we used the **Alarm Status** bindi ng function and the default settings for alarm state.



5. Now our Table component is pulling in similar information as the Alarm Status Table, but we have the freedom to customize the table exactly how we need using any of its available scripting or extension functions, or its customizer. We can also add our own buttons to acknowledge or shelve alarms using our own scripting functions. This may be more work in setting it up then the generic Alarm Status Table component, but it allows for full control over what is being displayed and what can be done with that information.





On this page ...

- Binding Properties to Prebuilt Functions
  - Using Function Bindings to Customize

Color Animation in Vision

Using Color on Components

Using color on components is an important part of creating effective HMIs. While static colors can help identify specific features on the screen, dynamic colors can help draw the users attention to certain areas. Making color type properties such as Fill Paint dynamic works a little bit differently than other properties with simple types. There typically aren't Tags of type color, so the way we set up bindings on these types of properties works a little bit differently, and we have a few options available to us.

On this page ...

- Using Color on Components
- Using Expression Bindings
- The Number to Color Translator
- Style Customizer
  - Style Customizer Window
  - Value Conflict
  - Style Customizer Example

Using Expression Bindings

You can use the expression language to calculate a color using the color() function. If you have a color that depends on multiple properties, then using an express is recommended to evaluate correctly. This first example returns a static color using the Fill Color property.

Expression

```
// binding on the Fill Color property
color(255,0,0) // static red color
```

This example takes a Tag value and translates it to a color that ranges from white to blue as the Tag value increases.

Expression

```
// binding on the Fill Color property
color(255,255,255-({tag value}/100*255)) // fades from white to blue when Tag value goes from 0 to 100 %
```

If you have multiple properties or Tags, you can use the logic Expression Functions to select between a few colors.

Custom Properties

```
if({Tag1}>50,
        if({Tag2},
                3, // if tag1>50 and tag2 is true
                1), // if tag1>50 and tag2 is false
        if({Tag3},
                2, // if tag1<=50 and tag3 is true
                0)) // if tag1<=50 and tag3 is false
```

This example takes one integer value and selects from several options.

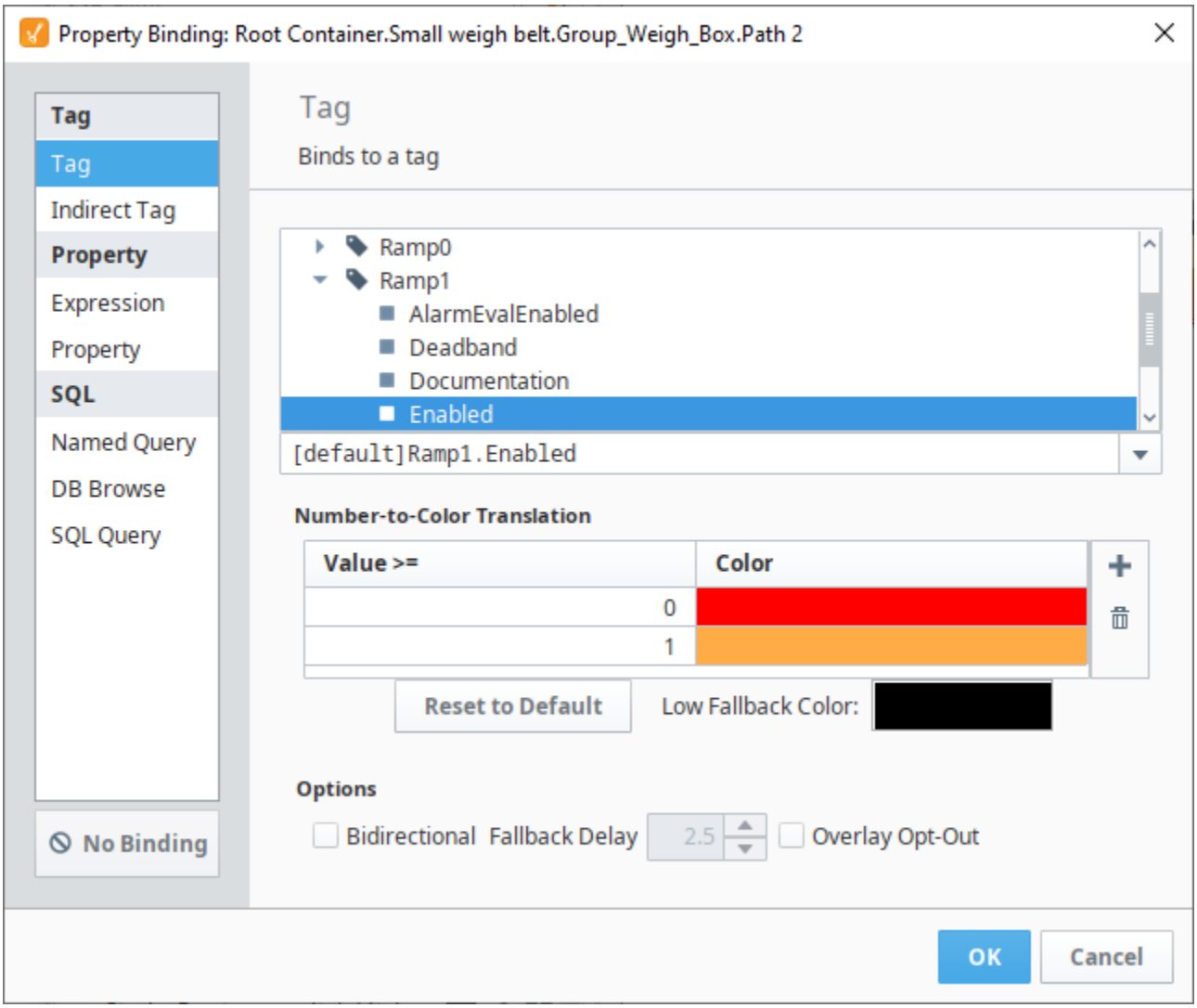
Expression Referencing a Tag

```
 // binding on the fill color property
switch({HOA tag},
        0,1,2, // off, on, hand
        color(255,0,0), color(0,255,0), color(255,255,0), // red, green, yellow
        color(0,0,0)) // black (fallback color)
```

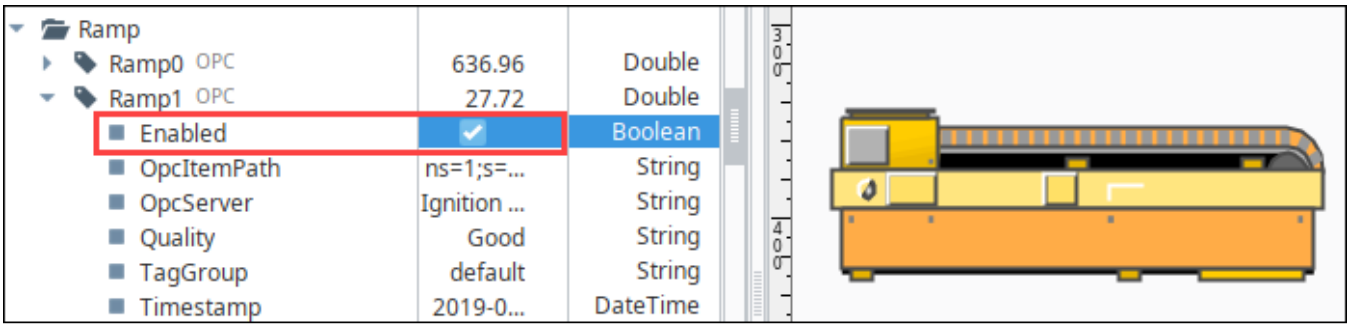
The Number to Color Translator

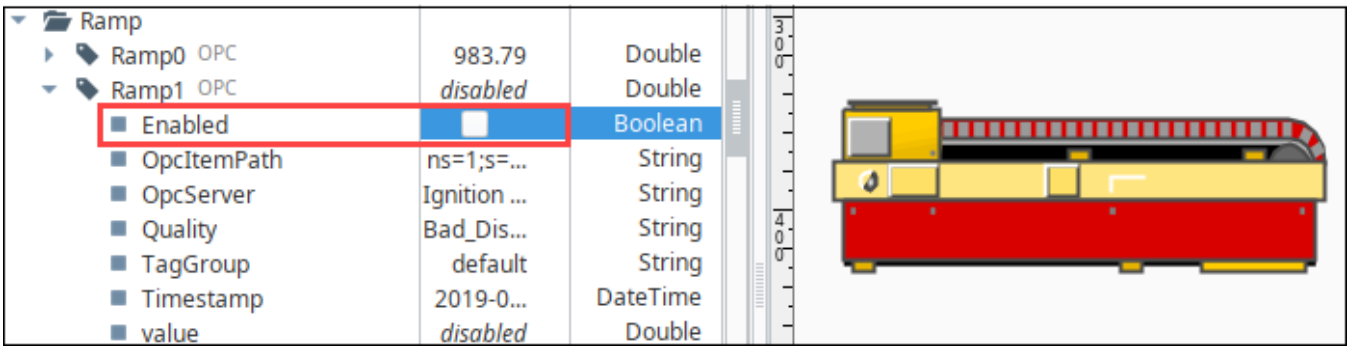
The Number-to-Color Translation, commonly known as Color Mapping is where you map a value to a color within a binding. When selecting a binding type where producing a color won't be possible, the Number-to-Color Translator will appear at the bottom of the binding window. This includes Property Bindings, Tag Bindings, and Indirect Tag Bindings. The way the Number-to-Color Translator works is that for every number range there is a set color. The binding then translates the numeric value into a color based on the mapping table. You can choose a different color for each value, and even make it blink between two different colors. If you need to add or remove values, use the **Add New Translation**  icon or **Delete Selected**

Translation icon on the right side of the Number-to-Color Translation table. There is a **Low Fallback Color** option so when a value falls below your lowest value, a default color can be set**.**



In this example, the fill color two parts of the conveyor symbol has been bound to the Ramp1 tag, enabled value. When the Ramp is enabled (value = 1), the symbol displays parts in the normal, yellow color. When the Ramp is disabled (value = 0), the fill color on those two parts is red, indicating the conveyor belt is not running. For more information on how this was applied to a symbol, see Images and SVGs in Vision.





Style Customizer

Many Vision components support the **Style Customizer**, which lets you define a set of visual styles that change based on a single **Driving Property**. Typically, you'll have a property on your component that you want to use as a driving property (like a discrete state), which then drives multiple visual properties, like the font, border, and foreground color, to change to a specific style that was set up per state beforehand. Style Customizer lets you define these relationships all at once, and lets you preview them too! Without styles, you would have to go to every property and bind them all individually.

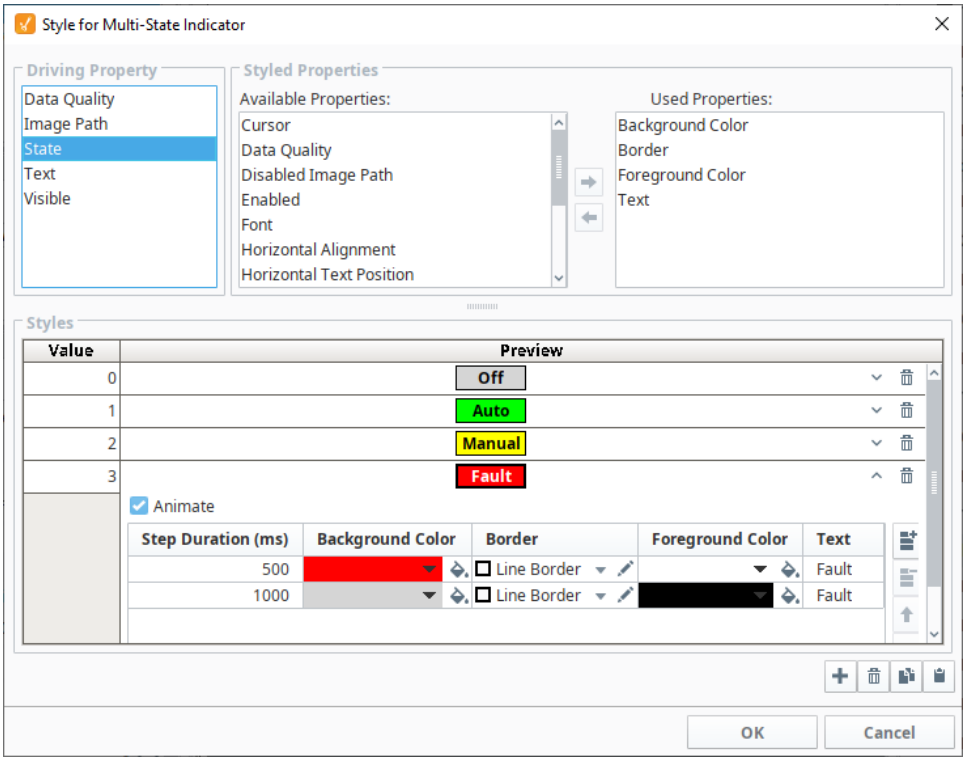
Style Customizer Window

The Style Customizer window has multiple parts to it.

- **Driving Property** - The value of the selected property will be used to determine the style used. Only certain properties on the component can be used as driving properties, but the most common are discrete state properties. Custom Properties can also be used here.
- **Styled Properties** - Here you can select which properties will be used in the styles. Any properties that are in the left panel are available to be used in the styles, while properties in the right panel are already being used in the style. Properties can be moved between the panels by selecting it and clicking the appropriate arrow button.
- **Styles** - The list of styles that will be available for this component. Each style has a Value property on the left. When the value of the Driving Property is greater than or equal to the value of a style, that style will be applied to the component. Each style gives a preview of what it looks like, and can be expanded to by clicking the expand icon to edit the properties within that style.

  You will notice in the image below that the properties being used in the Styled Properties are the Background Color, Border, Foreground Color, and Text, which corresponds to the properties we have available within each style in the Styles area. Each style can also be animated by clicking the animation checkbox. This allows you to add different steps to the style, where each step of the style can have its own unique style. Each step also gains a Step Duration (ms) property that is used to determine how long the step is active for, as shown in the fourth

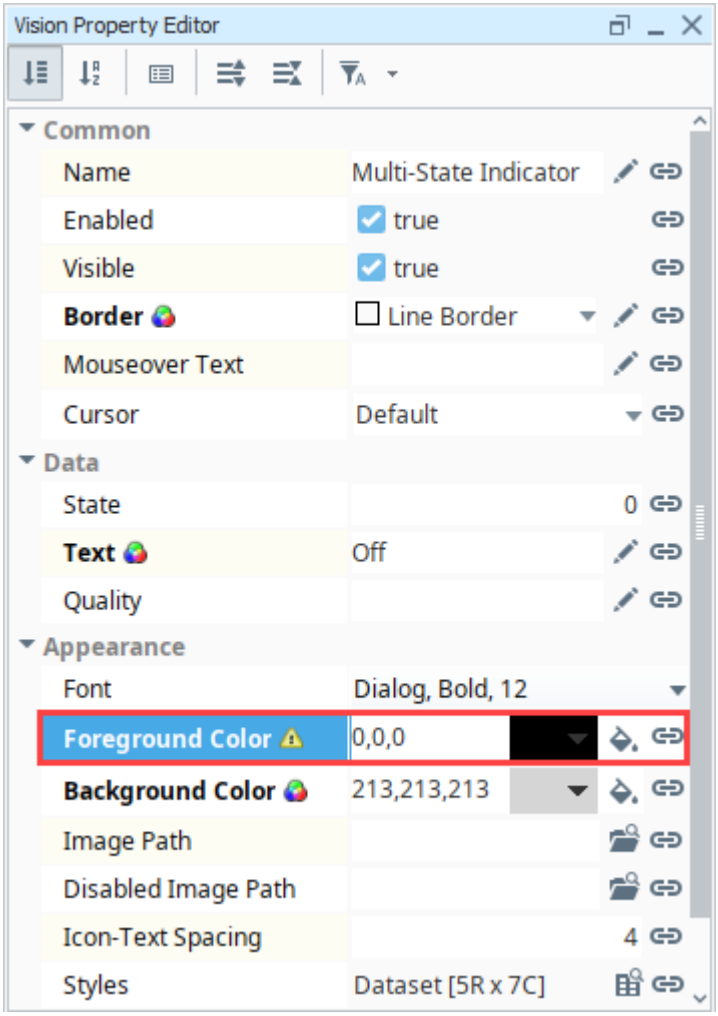
row. This is typically used to create a flashing effect, where the component will flash between two different colors such as red and gray.



Value Conflict

You can bind a property that is already being used by a style, but a warning icon will appear on the property in the Property Editor. This means there is a conflict between the binding on the property, and the style on the component. As a general practice, only the style or binding should write to the

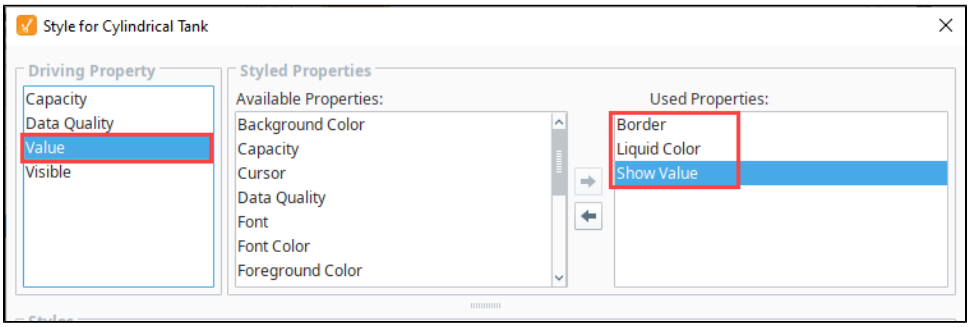
property, not both.



Style Customizer Example

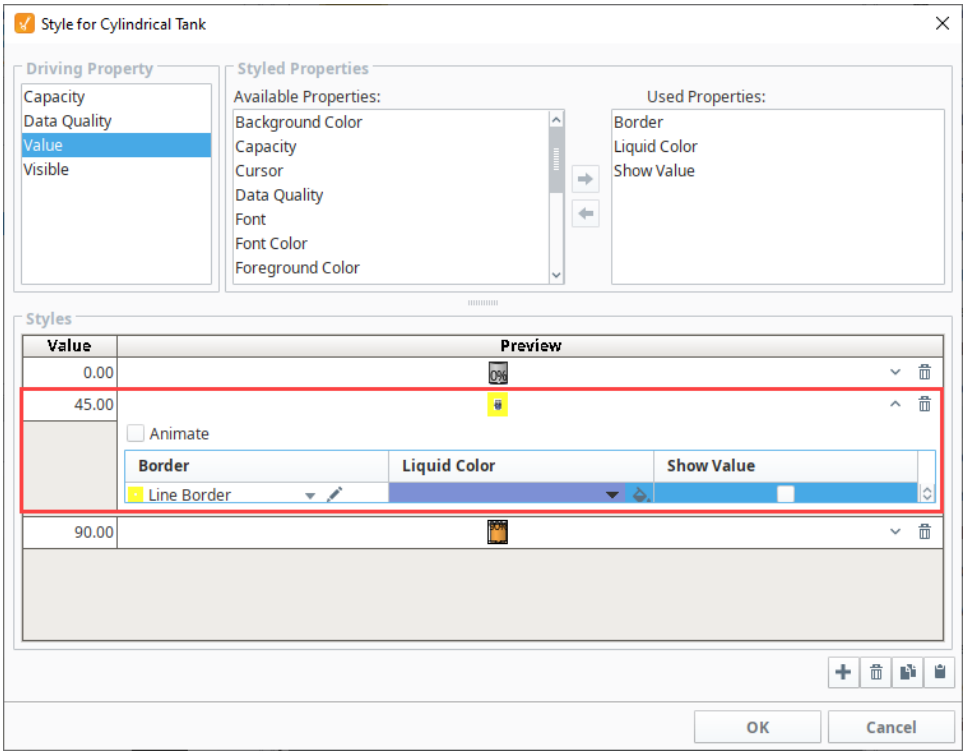
The best example of the Style Customizer in action is the Multi-State Indicator, as this component uses the style customizer to work properly and switch between different states, so it can be used as an example already built in. However, the many other components can use the Style Customizer, so this example sets up styles for a Cylindrical Tank.

1. Add a Cylindrical Tank component to the window, and add a Tag to the **Value** property. (This example uses a Rampt1 tag).
2. Right click on the Cylibdrical Tank and go to **Customizers > Style Customizer**.
3. Select a **Driving Property**. Here, the Value is a good choice as we can change the tank to flash when the contents get too high.
4. In the **Styled Properties select** the Border, Liquid Color, and Show Value.



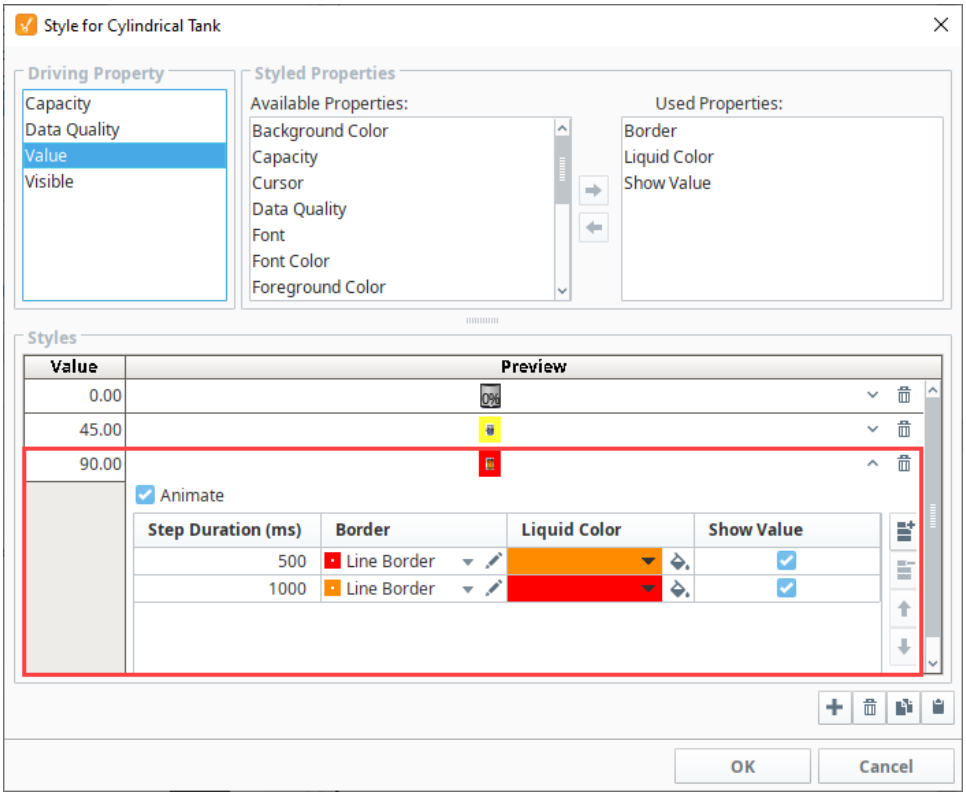
5. Next, we need to set up the different styles. Click the **Add** icon three times to add three styles.
6. Leave the first style set to Value 0.00, and don't change any of the other settings.
7. Set the second style to Value 45.

a. Click the **Expand** icon.
b. In the Border Chooser, select the **Line Border** style, set the line width to 5px, and the line color to yellow. This way, it is obvious the tank is filling up.
c. Set the **Liquid Color** to blue or keep the default color.

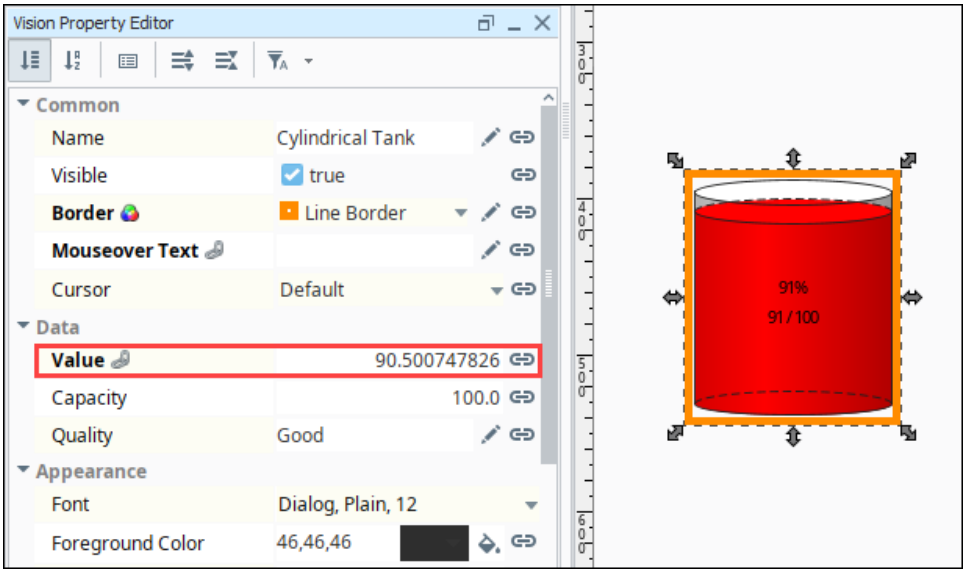


8. For the third style, we're going to animate it and and create two steps to alert the user that the tank is almost full. Set the third style to **Value** 9 0, then click the **Expand** icon. Set the following options for the first step:
    - **Animate checkbox** - checked
    - **Step Duration** 500 ms.
    - **Border** - Line Border Orange and set the Line Width to 5px, and the line color to Orange.
    - **Liquid Color** - Orange
    - **Show Value** checkbox - checked

9. Click the **Add** icon to add another step. Set the following options for the second step:
    - **Step Duration** 1000
    - **Border** - Line Border Red and set the Line Width to 5px, and the line color to Red.
    - **Liquid Color** - Red
    - **Show Value** checkbox - checked



10. Click **OK** to save the style.
11. In the Vision Property Editor, when the **Value** changes to >/= 90, the Tank component and border will change to reflect the style settings. The tank colors and border will flash between red and orange.

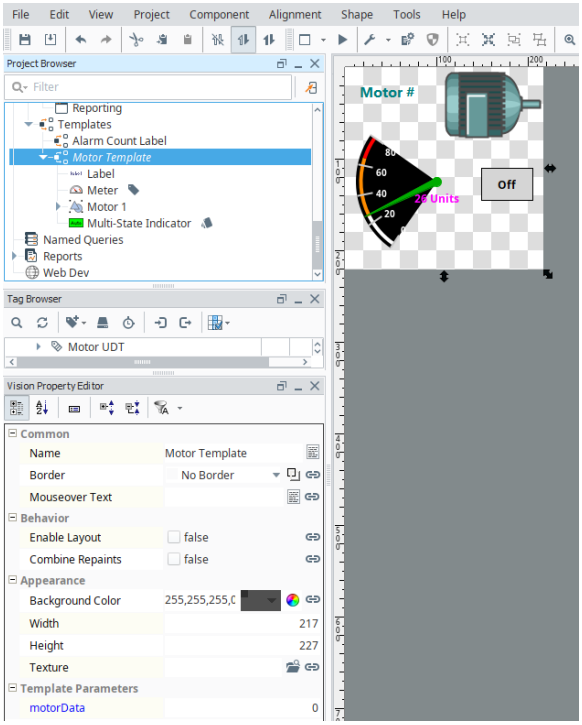


Vision Templates

Templates are a simple but a very powerful feature in Ignition that you can use with the Vision windows. The power comes from the ability to modify only the template in one location while affecting all of the instances used throughout the project. HMI and SCADA systems typically have a lot of repetitions in their screens. You might use a group of the same components over and over within your project. The data driving each set of graphics is different, but the graphics themselves are copies of each other. You can make a single template and use instances of the template over and over again.

When using templates, you define the graphical display in one place. This place is called the master template, but can also be called the template definition. You then use this master template many times in your project on multiple windows, thus making a number of template instances. Any changes made to the master template are then reflected in all of the template instances. Using templates early in your project development for any repeating displays can save a significant amount of time later on.

Without templates, the only way to do this is to copy-and-paste the components then re-bind them each time you want another. This is simple, and it works, but it can cause major headaches and time consuming corrections later on because if you ever want to make a change to how they are represented, you're stuck making the change to each copy of the group.



On this page ...

- Template Properties
  - Template Parameters and Internal Properties
- Indirection and UDT Tags
  - Standard Indirection
  - UDT Parameter
- Changing Template Path
- The Drop Target Parameter
- Resizing Templates
- Nested Templates
- Accessing Components Inside a Template Instance

INDUCTIVE UNIVERSITY

Template Overview

Watch the Video

Template Properties

Template Properties (called Template Parameters) allow each template instance to reference different data. Because the primary use of templates are the ease of maintaining repeated user interface elements, correct use of Template Parameters is very important. This is very similar to the concept of Parameterized Popup Windows. In that case, any **Custom** property on the **Root Container** of the window is used as a parameter, and is passed into the window when it is opened. With Templates, you have a property in the root of the master template that is exposed when you drop a Template Instance on a window.

Template Parameters and Internal Properties

When you open the **Custom Properties** window (right-click the checkered-box of the template and select **Customizers > Custom Properties**), you'll notice it is different than the Custom Properties of all other components. There are two kinds of custom properties here, as follows:

- **Template Parameters**
  These parameters appear on each **template instance**, allowing each instance to be configured differently. Commonly, this is some sort of indirection. For example, if you have a template representing motors, you might have MotorNumber as a parameter property. Then you can use that property as an indirection variable in other bindings within the template. Parameter properties are not bindable from **within** the template master design. When you use the template to create a template instance, the property becomes bindable. This ensures that the property only has a single binding configured for it.

- **Internal Properties**
  These properties cannot be used as parameters in your instances. They show up when designing the template master, but it does not show

up on the template instances. Internal properties are bindable from within the template master design. These properties are intended to be used for the internal workings of the template.

Indirection and UDT Tags

There are two primary ways to achieve indirection when using templates. Let's continue to use the example of a motor. Your system has many motors in it, and your template is used to display the status of the motors and control the motor's running mode. The goal is to be able to drop instances of the template onto your windows, and configure them in a single step to point to the correct motor's tags.

Standard Indirection

If the tags representing the datapoints of each motor are arranged in an orderly way in folders or with a consistent naming convention, you can use standard indirection to configure your template. You can add a parameter such as MotorNum to the template. Then you configure the contents of the template using indirect tag binding, where the value of MotorNum is used for the indirection.

UDT Parameter

If your motors are represented by a User Defined Type (UDT), you can save some effort and use a property of that type directly. Make your indirection property the same type as your custom data type. Then inside your template, you can use simple property bindings to create a link to the members of the UDT. When you create a template instance, you can simply bind that property directly to the correct Motor tag, and all of the sub-tags of motor are correctly mapped through the property bindings.

Changing Template Path

An instance of a template on a window has a property called **Template Path**. You can change this property on a window dynamically, and it can be bound to anything that produces a valid template path. For example, if there are two tank templates in a folder called Tanks, one template is called Tank A and the other is called Tank B. Each tank has a different look, but they have the same Custom properties. Their respective template paths are Tanks/Tank A and Tanks/Tank B. The template rendered on a window can swap between Tank A and Tank B by binding the instance's Template Path property to any string reference that says Tanks/Tank A or Tanks/Tank B.

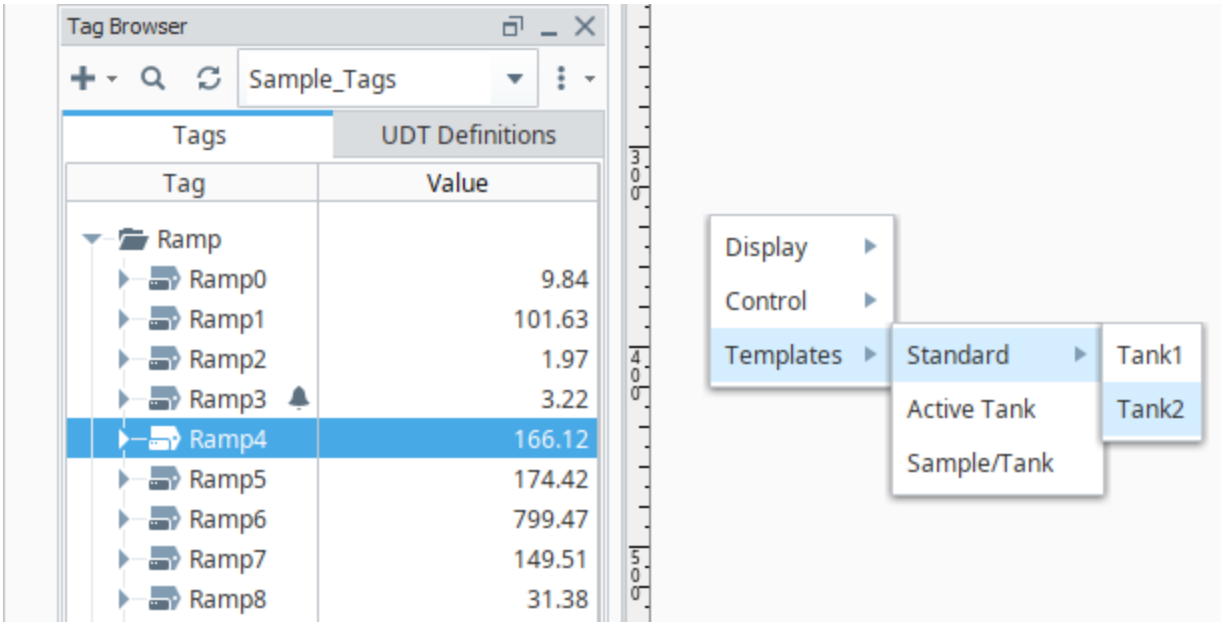
The Drop Target Parameter

When you specify parameters in the Custom Properties window (right-click the checkered-box of the template and select **Customizers > Custom Properties**), you can set one of the parameters as the Drop Target. This allows you to drop a tag of that type onto your template instances or onto a window to facilitate even quicker binding. For example, let's say that you have a parameter that is an integer and you've made it the drop target. If you drop an integer tag onto a window, your template appears in the menu dropdown list of components which is displayed. Choosing your template creates a template instance and binds that parameter to the tag.

This also works for UDT tags. Let's say you have a custom data type called Motor and a template with a Motor-typed parameter set as the drop target. If you drop a motor tag onto a window, it creates an instance of your template automatically. If you have more than one template configured with Motor drop targets, you have to choose which template to use.

> The following feature is new in Ignition version **8.1.34**
> Click here to check out the other new features

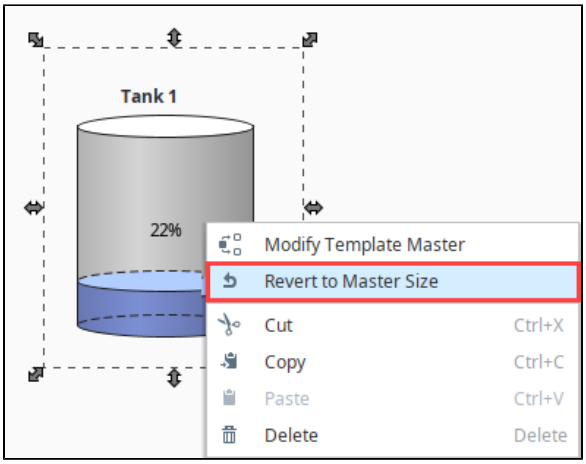
For easier management with large projects, the dropped template menu now builds nested menus for all possible template options, instead of displaying all folder paths in a single list.



Resizing Templates

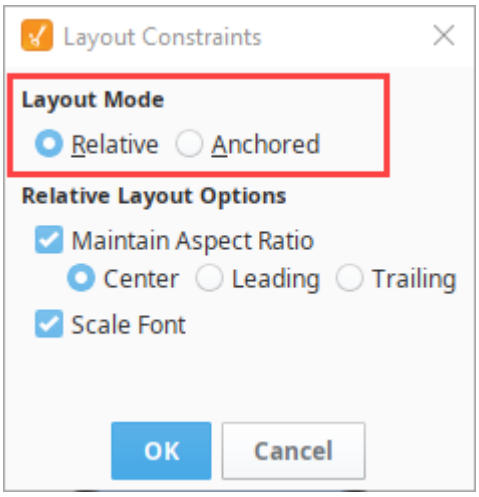
You can configure the layout of each template so it resizes properly.

By default, when you drag a template into the window from the Project Browser, the size of the instance is exactly the same size as the master template. You can make the size larger or smaller. To go back to the same size as the master template, right-click on the instance and choose **Revert to Master size**.

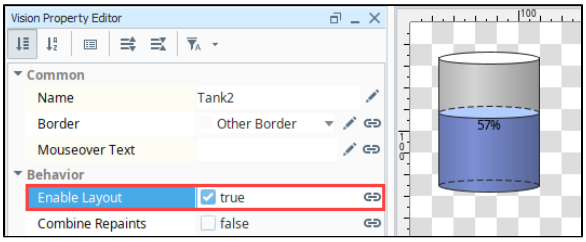


For every component or template instance you add to the window, you can pick a Layout option as to how it is going to resize in the Client. Right-click on the instance, select **Layout** from the menu, the Layout Constraints window displays showing all the default settings.

To learn more about layout, **see the Component Layout page**.



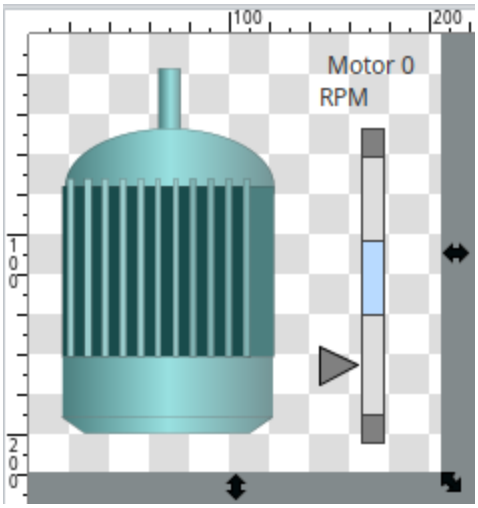
If your template instance resizes in the client, then it will stretch all of the components inside it in the same way a Component Group works. That is: it will ignore any layout settings and stretch without maintaining aspect ratio. If you want the template instances to respect your layout settings, set the **Enable Layout** property to true in the template definition.

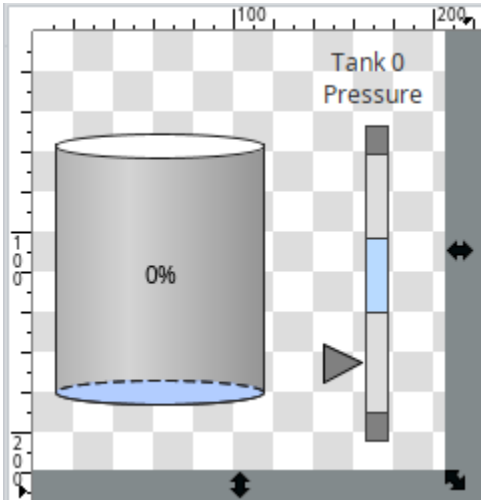


Nested Templates

You can embed templates inside of other templates. The nested template behaves like a component. This can be useful if the project can be broken down into many similar, small parts. Instead of building a template for a tank with a gauge, a motor with a gauge, and a compressor with a gauge, it might instead be better to first build a simple gauge template that can then be added to each of three templates so that it already is set up correctly.

Simply drag the already made template into the new template, just like you would onto a window. You can easily use an indicator template made to display values from a motor template, or values from a tank template, as long as you set up the indicator template with the proper indirection. This way, you only have to set up the indicator once and write in a few parameters, instead of having to customize the indicator the same way for every template that it gets added to.



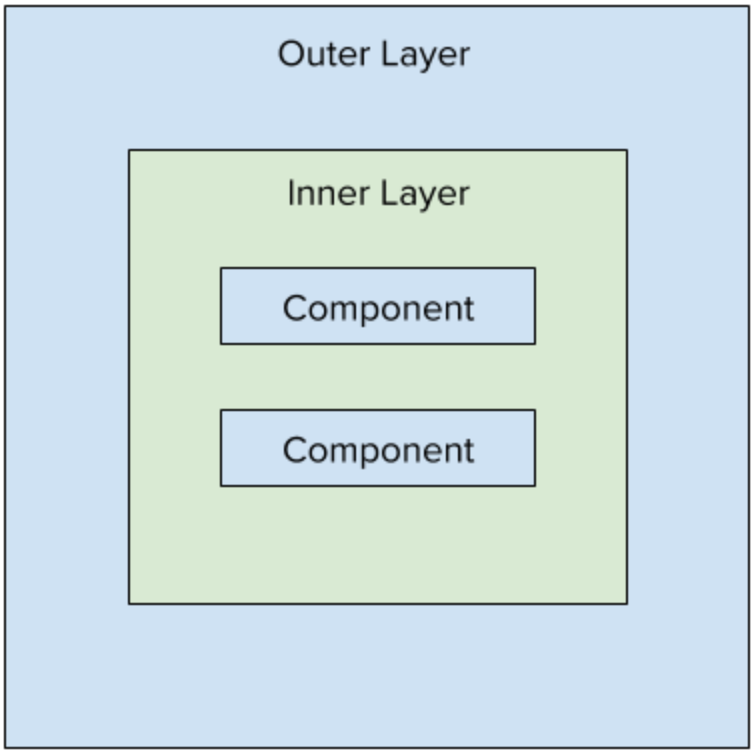


The following feature is new in Ignition version **8.1.22**
Click here to check out the other new features

Clicking on an open template in the Project Browser will open the template in the workspace. If the template you want to use is already open, you can instead alt-click to drag and drop the template where you want to use it.

Accessing Components Inside a Template Instance

When working with a Template Instance, the components inside that instance or normally hidden, and otherwise inaccessible. However, you can access these components via scripting. It helps to think of Template Instances as multi-layer containers. In most cases, users interact with only the Outer Layer, which contains the Template parameters, and the other default Template Instance properties. A Python script can access the Inner Layer, which then provides access to the components within.



A script can traverse to the Inner Layer from the Outer Layer with a `getComponent` call.

Pseudocode - Acessing the Inner Layer

```
myTemplate = event.source.parent.getComponent('MyTemplate')

# The '0' in the first getComponent call effectively refers to an index value of a component, which happens
to be the Inner Layer.
myTemplate.getComponent(0)
```

From the Inner Layer, a script can then call `getComponent` again to access any components within. Assuming a Template with a Label component named "Label", we could access the Text property with the following:

Pseudocode - Accessing a Component From the Outer Layer

```
myTemplate = event.source.parent.getComponent('MyTemplate')

print myTemplate.getComponent(0).getComponent('Label').text
```

In This Section ...

Creating a Template

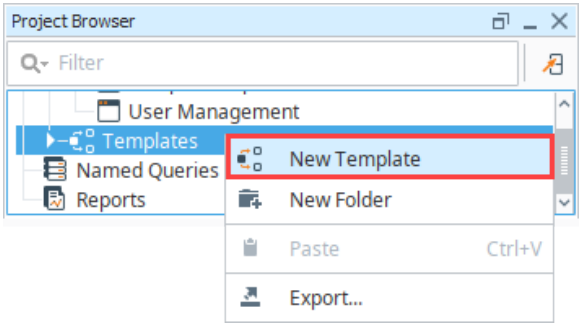
Templates are a simple but very powerful feature that you can use with your Vision windows. Templates are a built once, in one place, which is called the master template. You then create template instances throughout your project. Each of these instances will have the same components and properties as the master template, and will automatically update as changes are made to the master template.

Creating a template is easy. To start, right click on the Templates section of the Project Browser and select New Template.

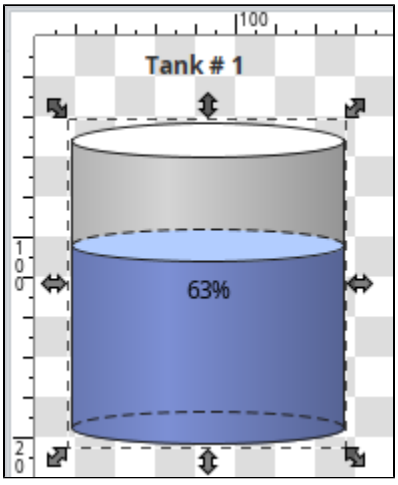
Basic Template

Here we create a basic template. After the template is configured, you can create multiple instances of it throughout your project.

1. In the Project Browser, right-click **Templates** and select **New Template**.



   A checkered box is displayed in the design space where you design your template. The checkered box means that the template is completely transparent. You can set a background if you want.
2. Right-click on **New Template** and click **Rename** to change its name to something else, for example "Tank".
3. Drag a Cylindrical Tank and a Label component onto the screen. Resize the components to fill the area of the template.
4. Bind the Value of the tank to a Tag, and set up the label to be a name for the tank.



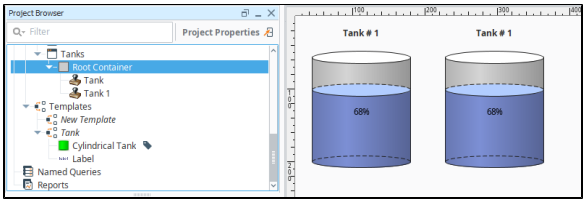
5. Now that we made a template, use that template on a window. Navigate to a window, and then click and drag our tank component onto the window. The template can be dragged onto the window multiple times to create multiple instances of the template, or even added to other windows.

On this page ...

- Basic Template
- Dynamic Templates
- Edit a Template
  - Example - Send a Template to a Different Project
- Template Custom Properties
- Creating the Template Instances

Creating a Vision Template

Watch the Video



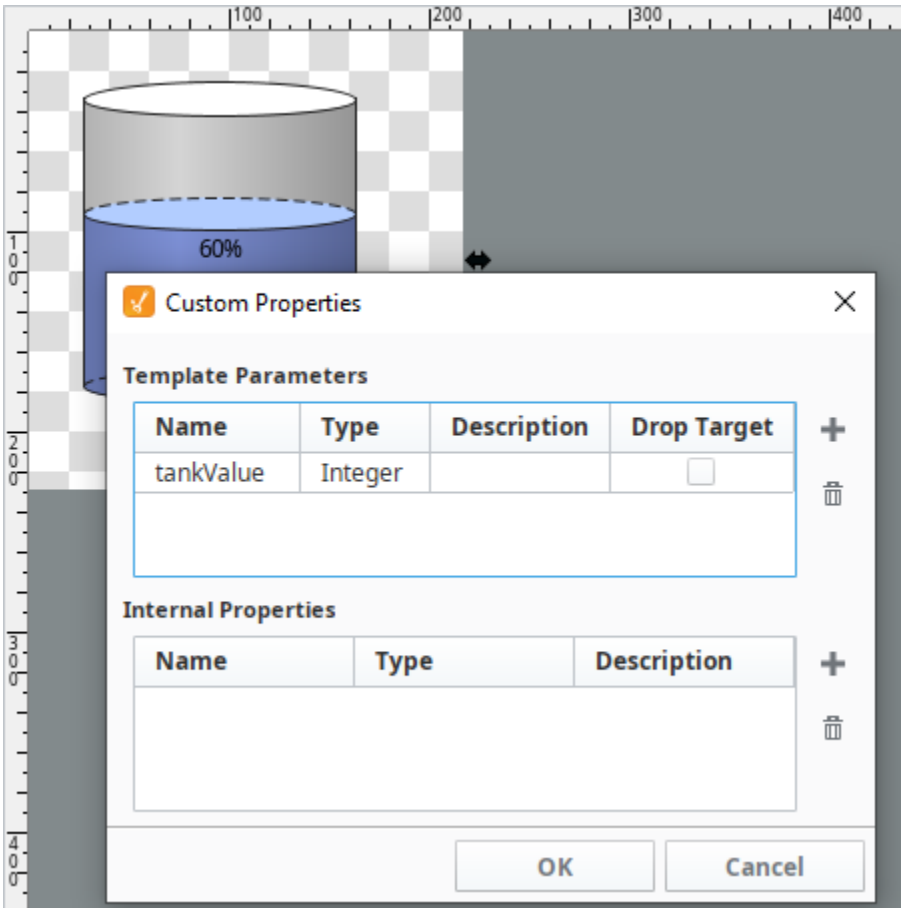
The following feature is new in Ignition version **8.1.22**
Click here to check out the other new features

Clicking on an open template in the Project Browser will open the template in the workspace. If the template you want to use is already open, you can instead alt-click to drag and drop the template where you want to use it.

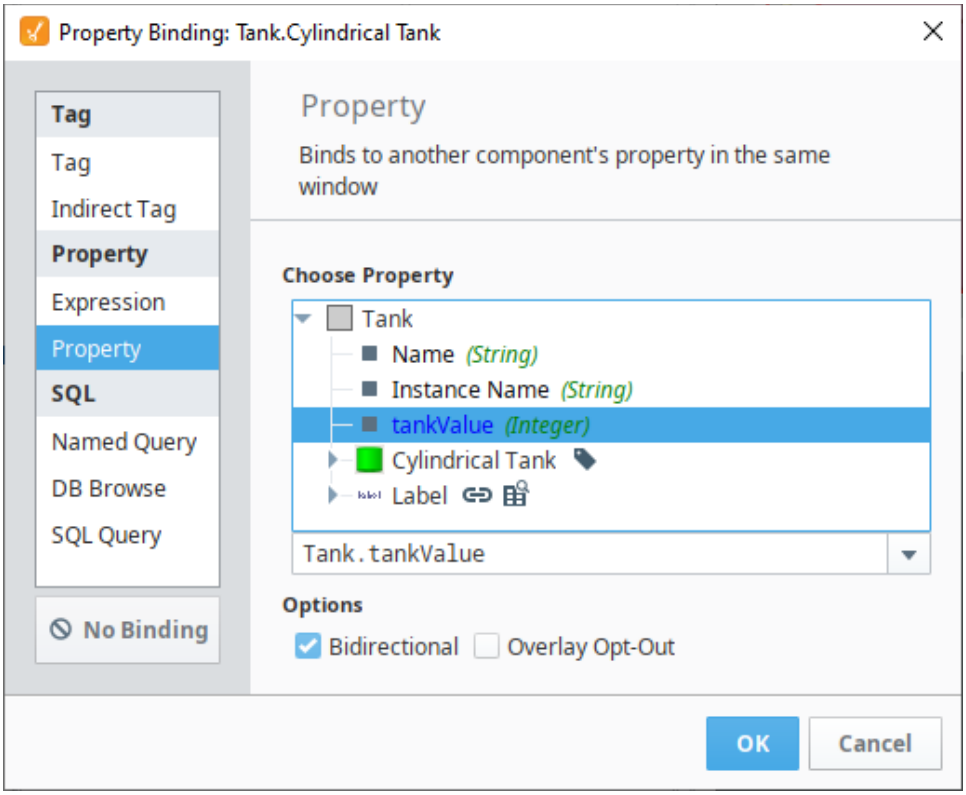
Dynamic Templates

While the basic template is a good example of what a template is, the real power of the template is its ability to be dynamic. The template can create parameters that can accept values from the instance, and use them throughout the template. This way, instead of always displaying Tank 1 values, we can pass in a tank value for the cylindrical tank component to use.

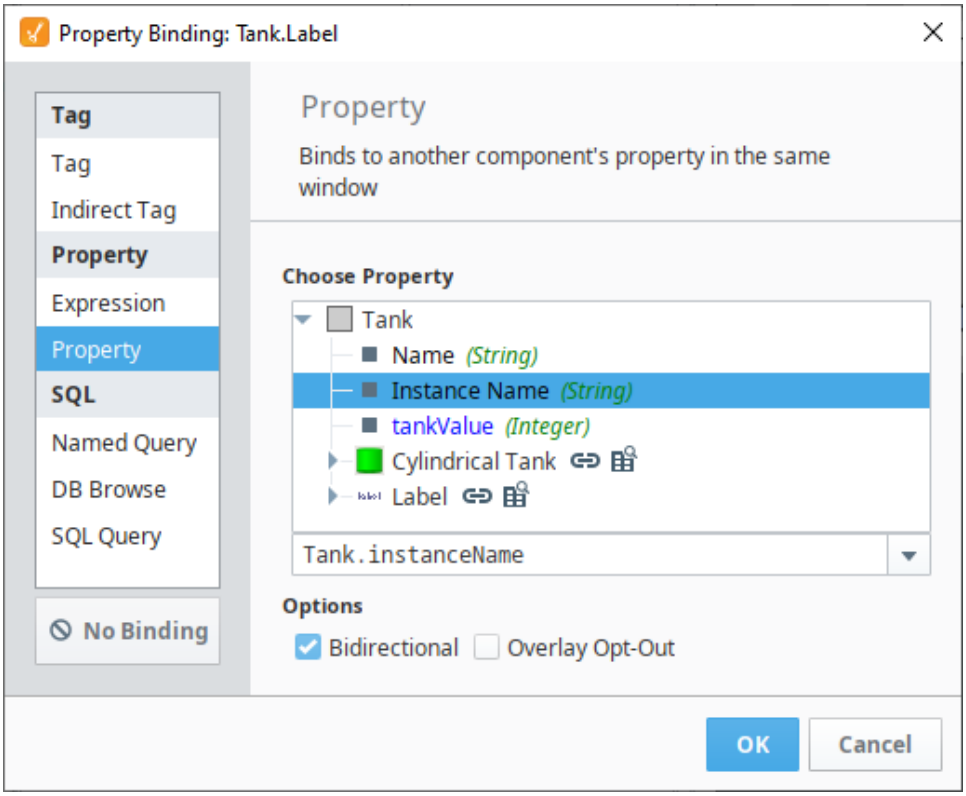
1. In the Tank template, right click on the gray background and go to **Customizers > Custom Properties**.
2. Click the plus button to add a Template Parameter. Give it a name like **tankValue**, and click **OK**. There should now be a new property on the Tank Template object called **tankValue**.



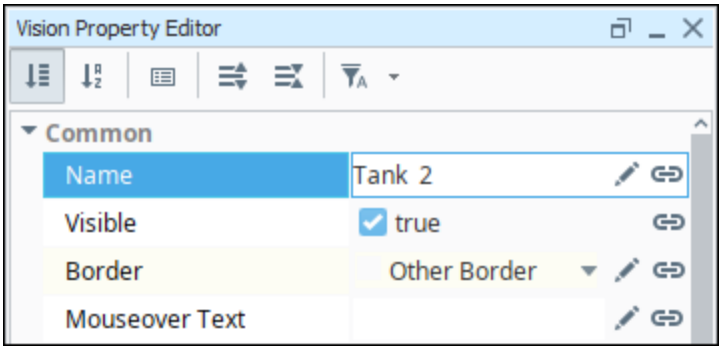
3. Now bind the **Value** property of the cylindrical tank to the **tankValue** property that we just made. Note that it will be 0 right now, but we will pass in a value later.



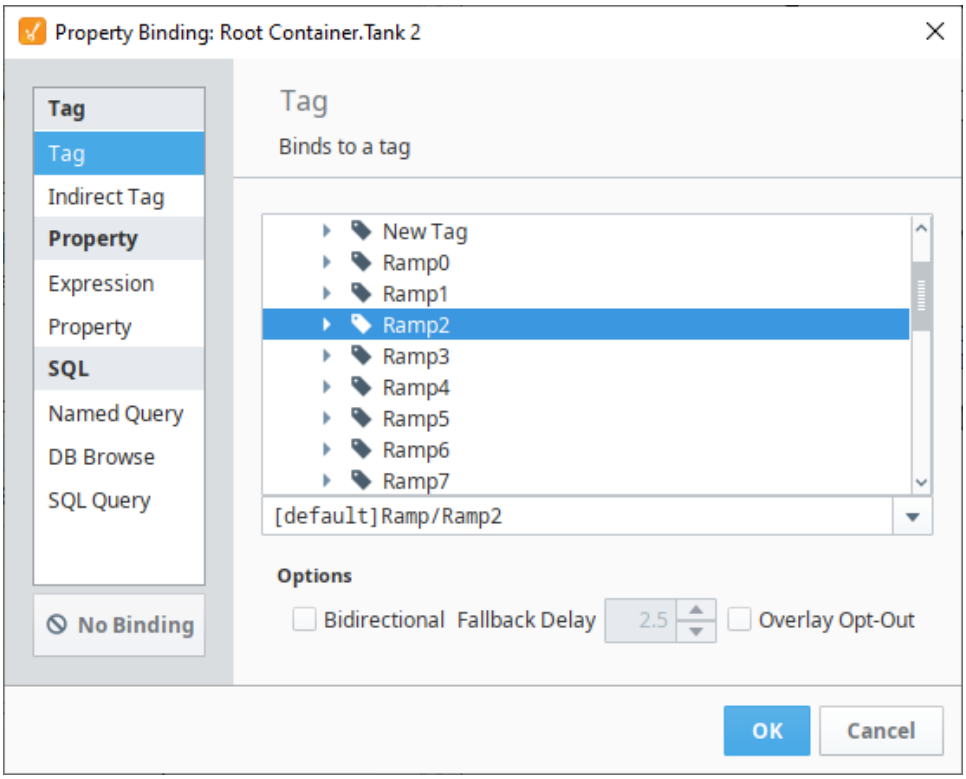
4. We also want to change the label, since this template won't always be pointing at Tank 1. So we can bind the **Text** property of the label to **Tank Template's Instance Name**. This way, whatever we name the template instance is what the label will show.



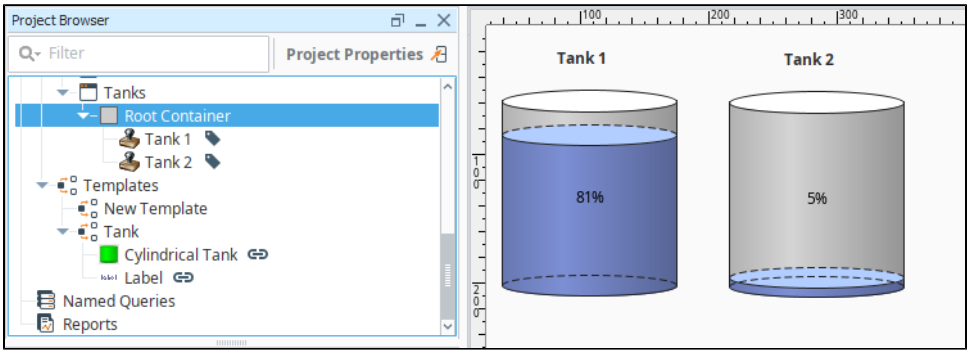
5. Back on our window, we want to change our two instances so that the first one is called Tank 1, and the second is called Tank 2. Notice how the labels change when we change the name of the template. Select your second Tank, and change the **Name** property to Tank 2.



6. We also want to bind the **tankValue** property of each instance to separate Tags. In this example, you can do this simply by selecting Tank 2 and clicking on the binding icon for the **tankValue** property and selecting another Tag.



7. We now have two instances of the same template, but they are displaying different information because we are passing different values into each.



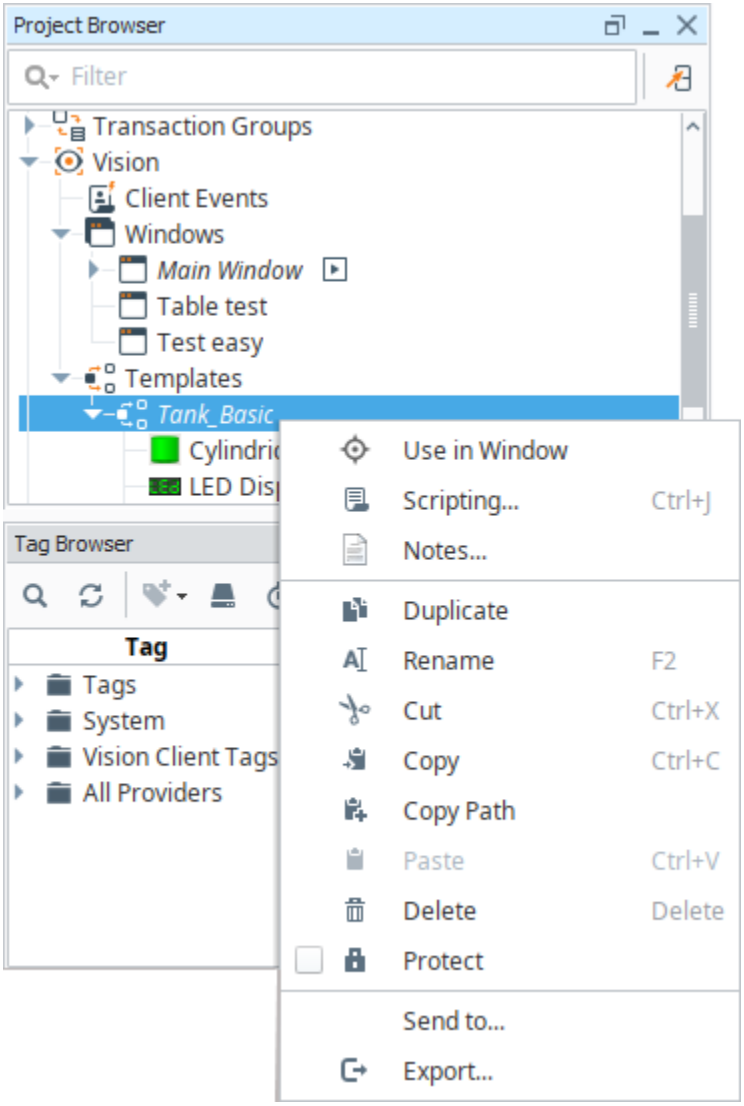
Edit a Template

You can open a template for editing by double-clicking on it in the Project Browser, or by double-clicking any instance of that template within a window. You design your template the same way you design windows: by adding components to it, and configuring those components using property bindings and scripting.

There are a few differences between templates and windows from an editing perspective. Templates, unlike windows, have a transparent background by default. This can be changed simply by editing the background color of the template. Templates also do not have the concept of the "Root Container" - the template itself acts like a container.

Once you change the master template, all the instances of that template are updated.

Templates are a project resource. As such they can be copied, duplicated, protected and more. There are several actions available with a right click menu.



Action	Description
Use in Window	Places the selected template on the current window.
Scripting	Opens the Component Scripting Window where you can set scripting on this template. For more information, see Script Builders in Vision.
Notes	Opens a popup window where you can make notes about the template.
Duplicate	Duplicates the template in the Templates folder.
Rename	Renames the template.
Cut	Cuts the template, but leaves it on the clipboard.

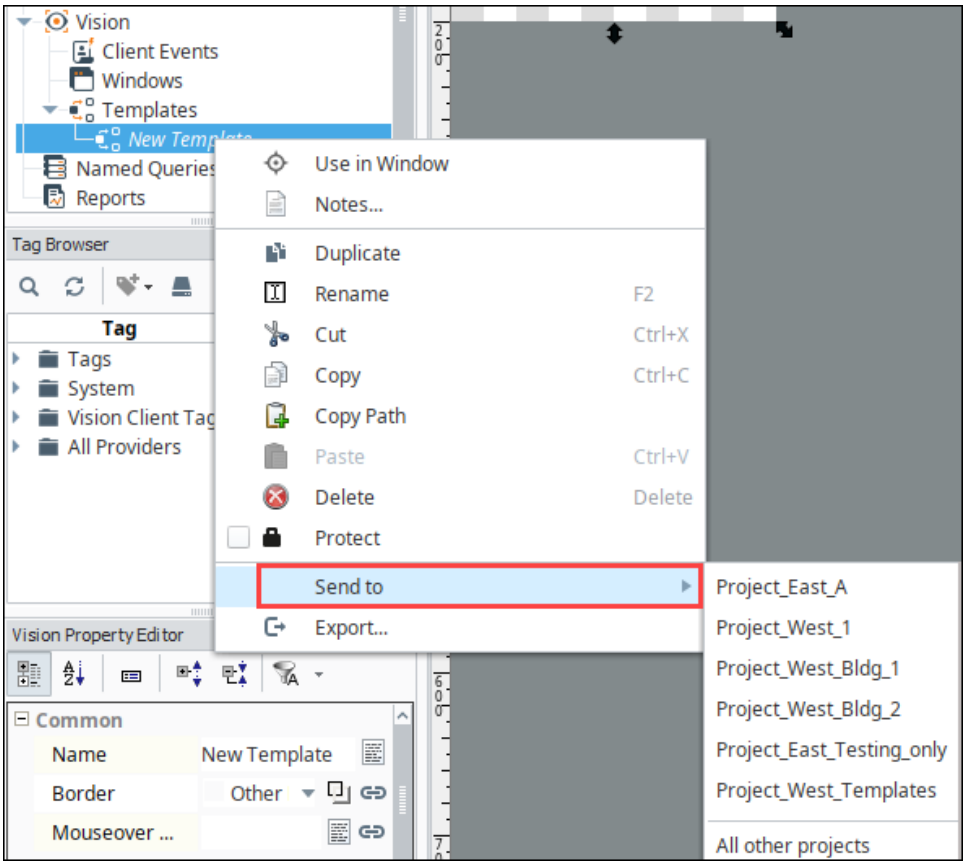
Copy	Makes a copy of the template on the clipboard.
Copy Path	Copies the path to the template and places it on the clipboard.
Paste	Pastes a template that's currently on the clipboard.
Delete	Deletes the selected template.
Protect	Once a project resource protected, it cannot be changed except by someone that has the permission to unprotect it, and modify it. For more information, see Project Security in the Designer.
Send To	Sends this template to another project on this Gateway.
Export	Opens export Project Resources window. where you can export this template and other resources. For more information, see Project Export and Import.

Example - Send a Template to a Different Project

You can share Templates with other projects.

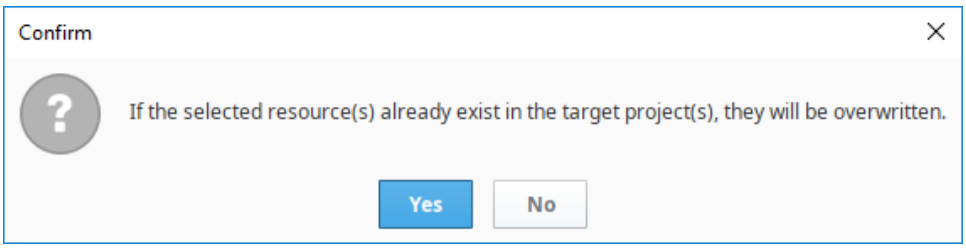
Caution: If a Template by the same name already exists in the target project, it will be overwritten.

1. To send a Template to a different project, right click on the template name and choose **Send to**.



2. A list of existing projects is displayed. Choose a specific project, or select **All other projects** if you want to send the template to all projects.

3. A confirmation message is displayed. Click Yes to confirm.



Template Custom Properties

Templates have the capability to incorporate custom properties. In this way, they are the same as any other Ignition components. The main difference between the custom properties for an Ignition component and a template is that the template has internal properties and template parameters.

- **Internal Properties** - Internal properties help facilitate the bindings within a template in the same way that a window's root container will help facilitate bindings between components that make up the template. When a template is deployed onto a window the internal custom properties are not exposed to the world outside the template.
- **Template Parameters** - The template parameters are the template's custom properties that are exposed to the outside world. In other words, when a template is deployed onto the root container of a window, the template parameters are available for binding with the objects on that window or to Tags.

Creating the Template Instances

Once you've made your template, you can use it on any of the windows in your project by doing any one of the following steps:

- You can drag the template from the Project Browser into an open window just like you can drag components into the window for display.
- You can right-click on the template in the Project Browser and choose **Use In Window**, which will let you place the template inside a window with another click.
- You can drag a Tag from the Tag Browser to a window and from the pop-up menu, which is displayed, you can choose a template. This only works if the template has a configured Template Parameter that been enabled as a drop target.

The following feature is new in Ignition version **8.1.22**
Click here to check out the other new features

- You can alt-click on an open template in the Project Browser and drag it into an open window just like you can drag components into the window for display.

The template instance can then be moved and resized like any other component.

Related Topics ... User Defined Types - UDTs

Template Indirection

Indirection in Templates

You can create templates that point indirectly to a set of Tags based on a simple parameter. This is very helpful when you have a large number of UDTs with the same type of Tags that only differ in one parameter. For example, lets say you have 100 Tank UDTs that all have the same kind and number of Tags. The only thing that is different is the Tank number.

If each Tag inside the UDT has a Tag path that looks like this:

```
Tanks/Tank 1/Volume
```

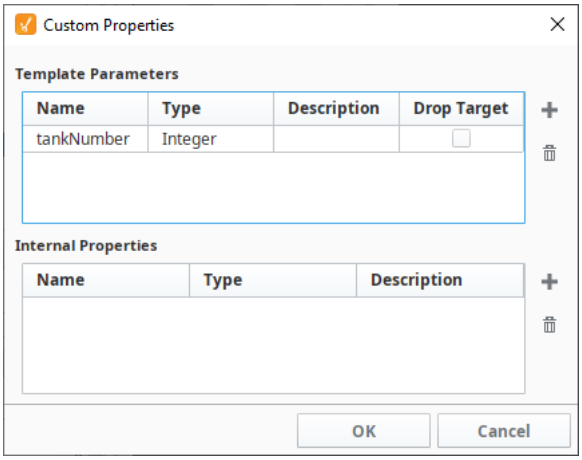
You can create a Template Parameter on the Custom Properties that can help make the template indirect. There are two main ways of doing this. The first method is to pass in an indirection value so that Indirect Tag bindings can be set up within the template. In this case, we would want to pass in a tank number, which can be substituted in for the tank number in the Tag Path. The second method is to pass in a reference to the entire UDT, essentially turning that UDT reference into a property that we can use within the template.

Indirect Binding

This example demonstrates how to set up a parameter to be used in an Indirect Tag binding.

Create a Template Parameter

1. Create a new template, and call it Tank.
2. Right click on the background of the template and select **Customizers > Custom Properties.**
3. Add a Template Parameter called **tankNumber** of type Integer.



Bind Values Indirectly

Next, add some components to our template, using the Template Parameter **tankNumber** that we created earlier to bind them to certain values indirectly. In this example, we will be binding the Tank's Volume property to a simulated Ramp tag.

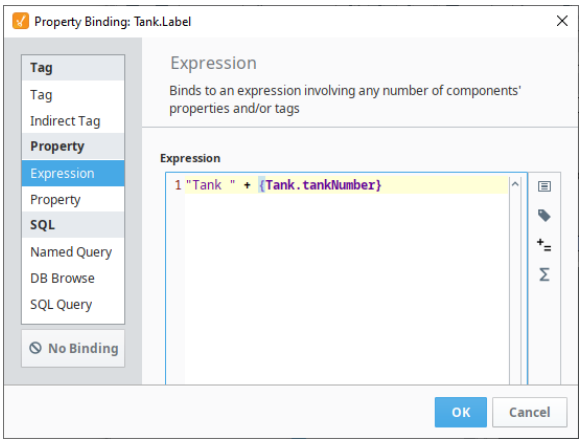
1. Add a **Label** component at the top of the template with a simple expression binding to display which tank is being shown in the template using the **tankNumber** property:

Expression - Indirect Label

```
"Tank " + {Tank.tankNumber}
```

On this page ...

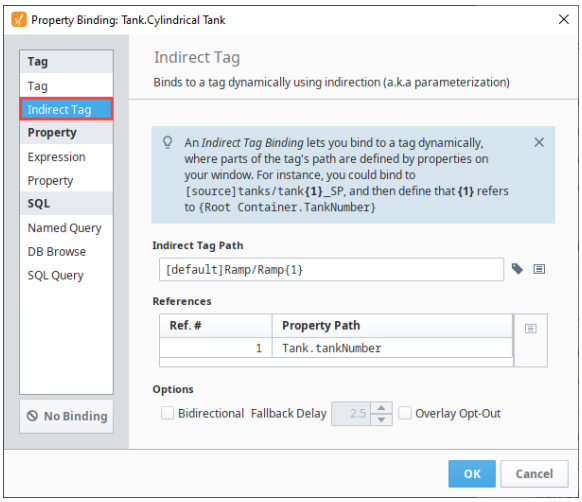
- Indirection in Templates
- Indirect Binding
- UDTs in Templates



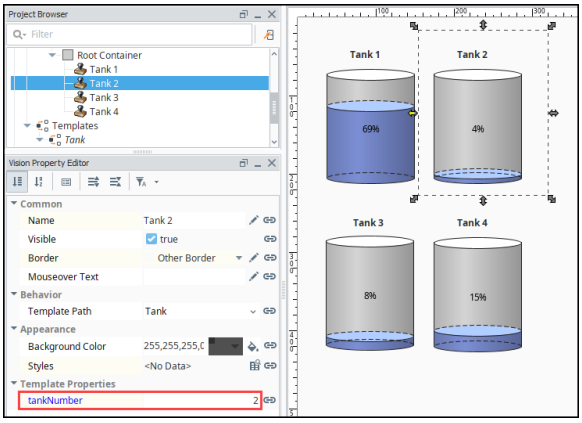
2. Add a **Cylindrical Tank** component that has an Indirect Tag binding on the Volume property, using the **tankNumber** property for indirection.
    a. Click the Tag icon to insert a Tag Path under Indirect Tag Path. In this example, we used `[default]Ramp/Ramp1`
    b. Replace the number at the end of your Tag Path with a parameter reference:

Indirect Tag Path
`[default]Ramp/Ramp{1}`

    c. Define the reference, so that **{1}** is replaced in each instance's Tag Path with the Template's **tankNumber** property.



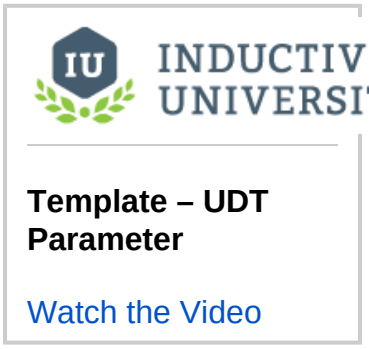
3. Drag a few instances of the Template onto a window.
4. Enter a different value into our **tankNumber** for each template. All of the templates will now display different Tag values from one another. In the below screenshot, Tank 1 is displaying the Tag value of Ramp 1, Tank 2 is displaying the value of Ramp 2, and so on.



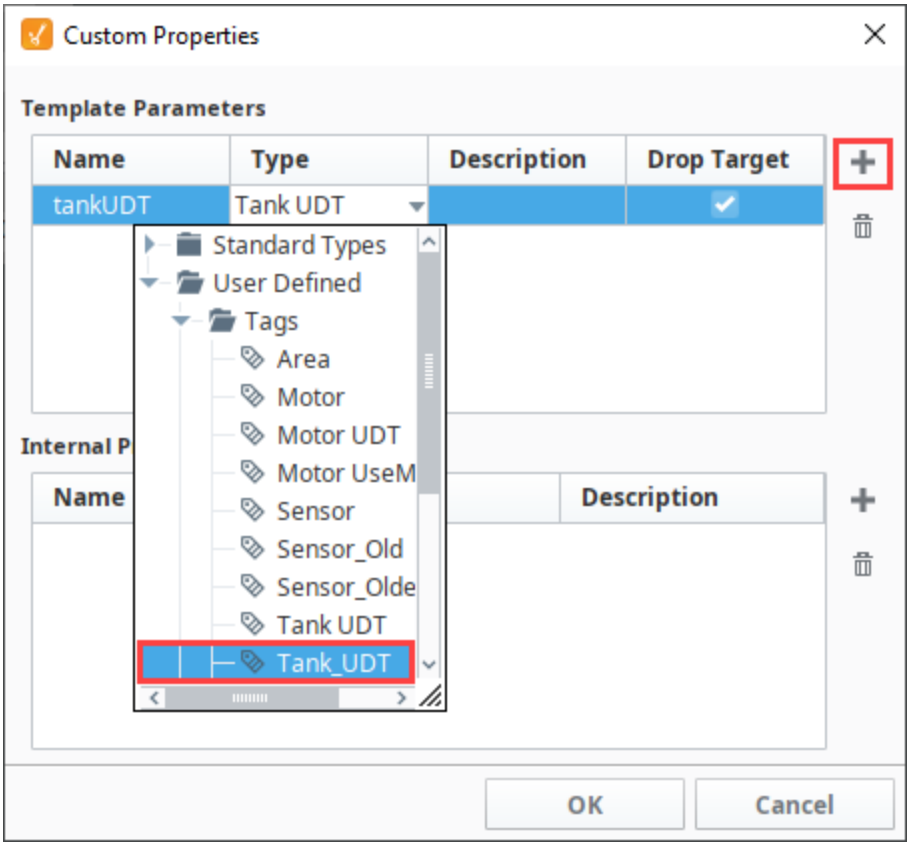
UDTs in Templates

When adding custom parameters to a template definition, the type of the property can be set to a User Defined Type (UDT). This creates a complex property with several child properties, where each sub property represents a tag in the UDT. These properties can be bound to within the template. An instance of a UDT can be passed into a Template Instance just like any other Tag would.

When using a UDT as a parameter in a template, be mindful that the Template Canvas and Template Repeaters components can not make use of UDT parameter types on embedded templates: when using either component, Indirect Binding on standard data types is the preferred approach.



1. Create a new template, and call it **Tank2**.
2. Right-click on the background of the template, and select **Customizers > Custom Property.**
3. Add a Template Parameter called **tankUDT**, but this time, we want the type to be a UDT (named "Tank_UDT") that we have already created.

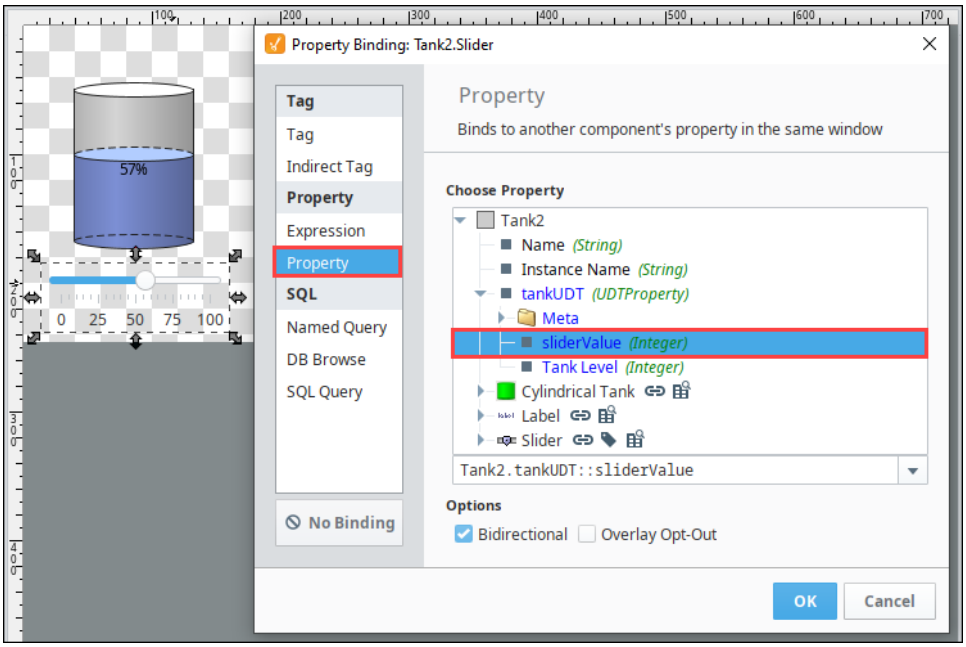


4. Next, add some components to our template: Label, Cylindrical Tank, and Slider. Be sure to utilize the Template Parameter **tankUDT**.

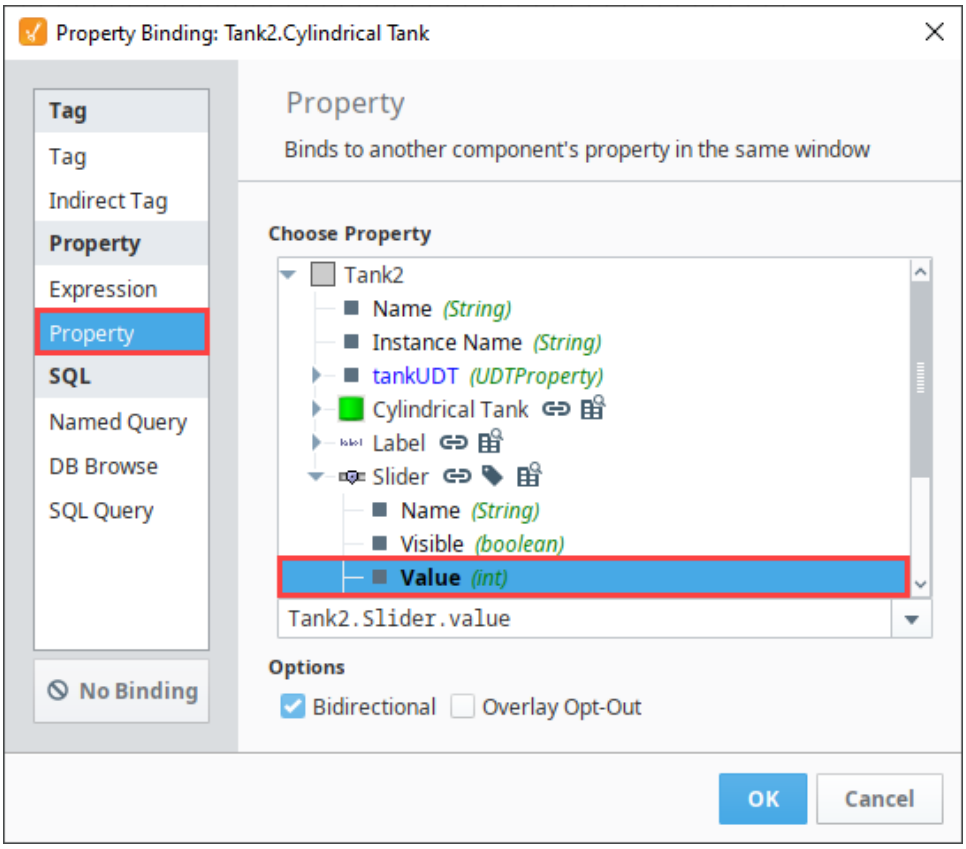
    a. Add a **Label** component at the top of the template to show the name of the tank. Using a property binding, bind the label to the Tag **Name** property of the UDT property **tankUDT** in the Meta folder. This will pull the name of the UDT instance in as the title of the template. Don't worry if it makes your label blank, as there is nothing in that custom property yet.

Expression - Indirect Label
`{Tank.tankUDT::Meta.TagName}`

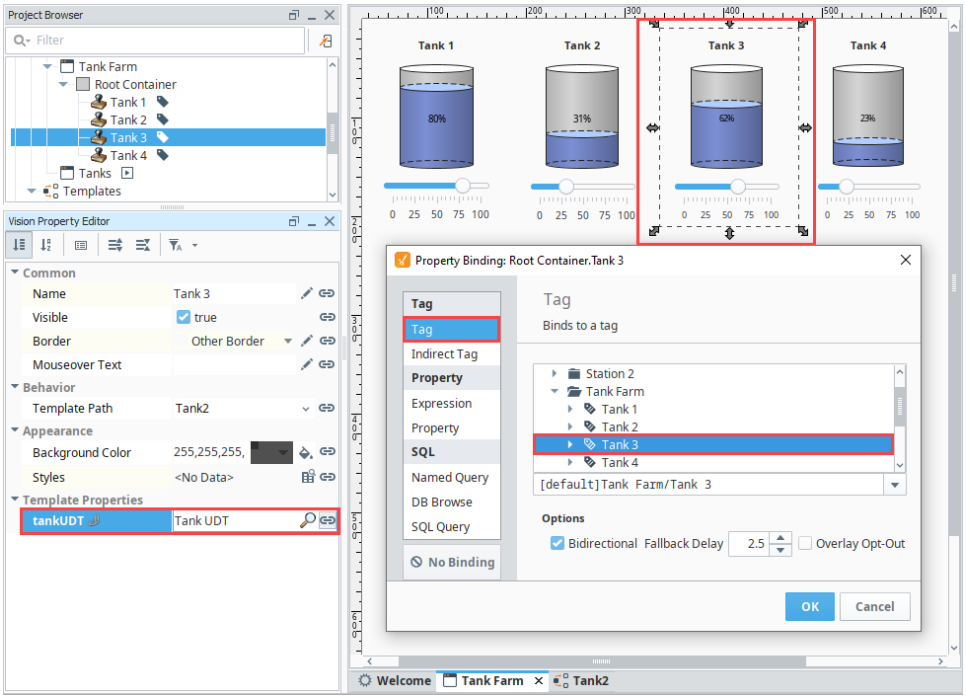
    b. Add a **Slider** component and bind it to the **sliderValue** property within the **tankUDT** property.



c. Add a **Cylindrical Tank** component and bind it to the Slider component's **Value** property.



5. Next, add an instance of the **Tank2** template onto a window. In this example, we added three instances.
6. Bind the **tankUDT** property to each of the UDT instances. In the image below, we selected the **Tank 3 instance** and bound the **tankUDT** tem plate property to the **Tank 3** Tag. The Tag bindings to the UDTs can even be made indirect to allow the passed UDT to be changed at runtime.



Related Topics ...

- Using the Template Repeater
- Using the Template Canvas

Using the Template Repeater

The Template Repeater component lets you easily create multiple instances of a master template for display on the HMI. Each instance shown in the Template Repeater has the same look, feel, and functionality of the master template. The instances can be arranged vertically, horizontally, or in a "flow" layout, which can either be top-to-bottom or left-to-right. If there are too many instances to fit, a scrollbar is added to the display. The Template Repeater also gives you the ability to pass parameters to each instance of the template, making the templates dynamic.

The Template Repeater can create multiple template instances in two different ways, which will also affect how it passes parameters to those instances. The first is Count mode, which will allow you to specify how many times the Template Repeater will repeat a template. It will then use the index number of each template as a parameter that it will pass into the template. The second method is Dataset mode, where each row of a dataset will be a new template instance and each column will be a parameter that will be passed into the template. This is useful if you have multiple parameters that need to be passed into a template. See the Vision - Template Repeater page for a more detailed description of this property and how it works. We will go over both methods of using the repeater below.

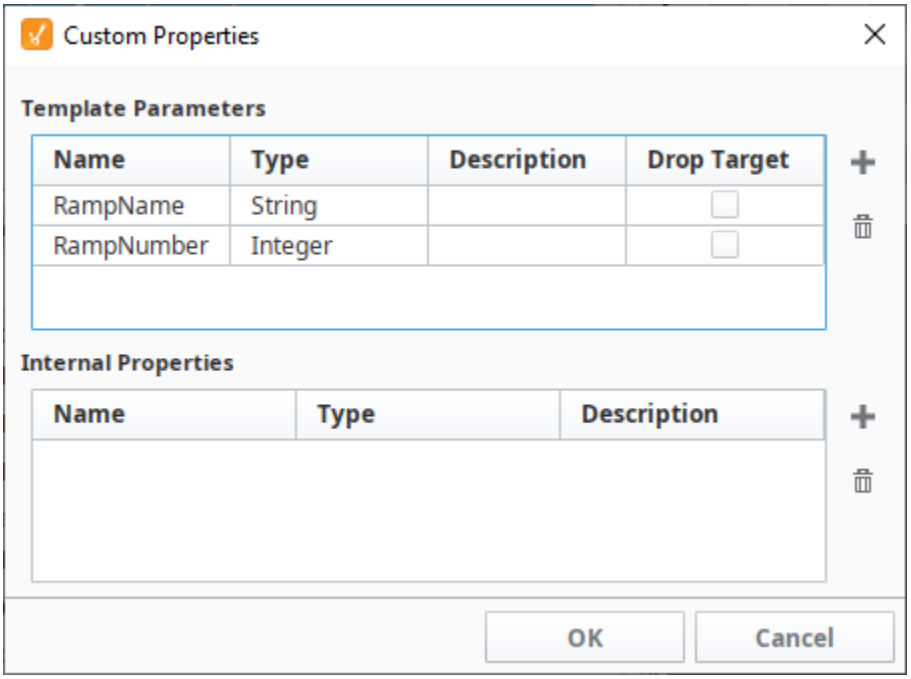
On this page ...

- Creating a Template
- Using the Template Repeater with Count Mode
- Using the Template Repeater with Dataset Mode

Creating a Template

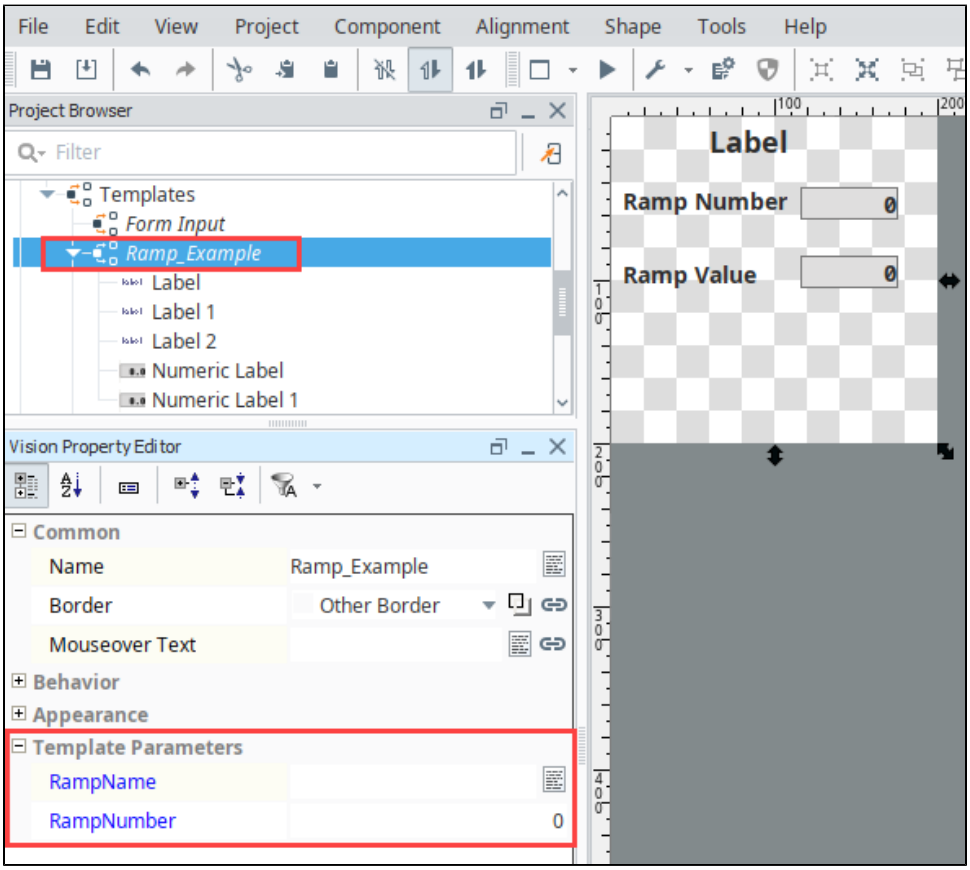
Before we use the Template Repeater, we need to first have a template that we need multiple copies of. We used Ramp Tags in the Generic Simulator device built into Ignition. The template will have a label at the top with the ramp name, and it will display the ramp number value. To do this, our template needs to have two parameters. One called RampNumber, which will be used to display the ramp number and also used for in an indirect Tag binding. The second parameter called RampName, which is passed in a string name that is the name of the ramp that I made up. The steps for making this template are listed below, or you can skip ahead to the next section if you are familiar enough with making templates.

1. In the Templates section of our Project Browser, create a new template. I named mine, Ramp_Example. See the Templates section for a more detailed description on what templates are and how to use them.
2. We are going to want to pass in a value to the template, so we need to create a Template Parameter.
    a. Right click on the Root Container, select **Customizers > Custom Properties** and add a Template Parameter called **RampNumber**, and make it an **Integer** type.
    b. Add a second Template Parameter called **RampName**, and make it a **String** type.

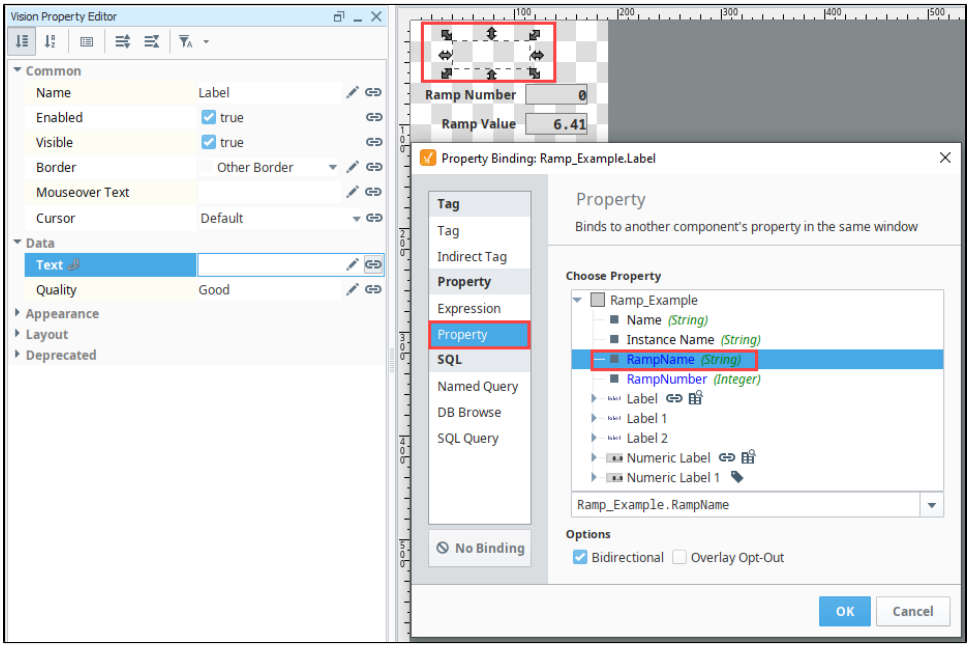


3. Next, add some components to our Template.
    a. Add two Numeric Labels and one Label. I also added two extra label components, and manually typed in some static text into their **Text** property. In one label, we entered "Ramp Number" and in the other, "Ramp Value".

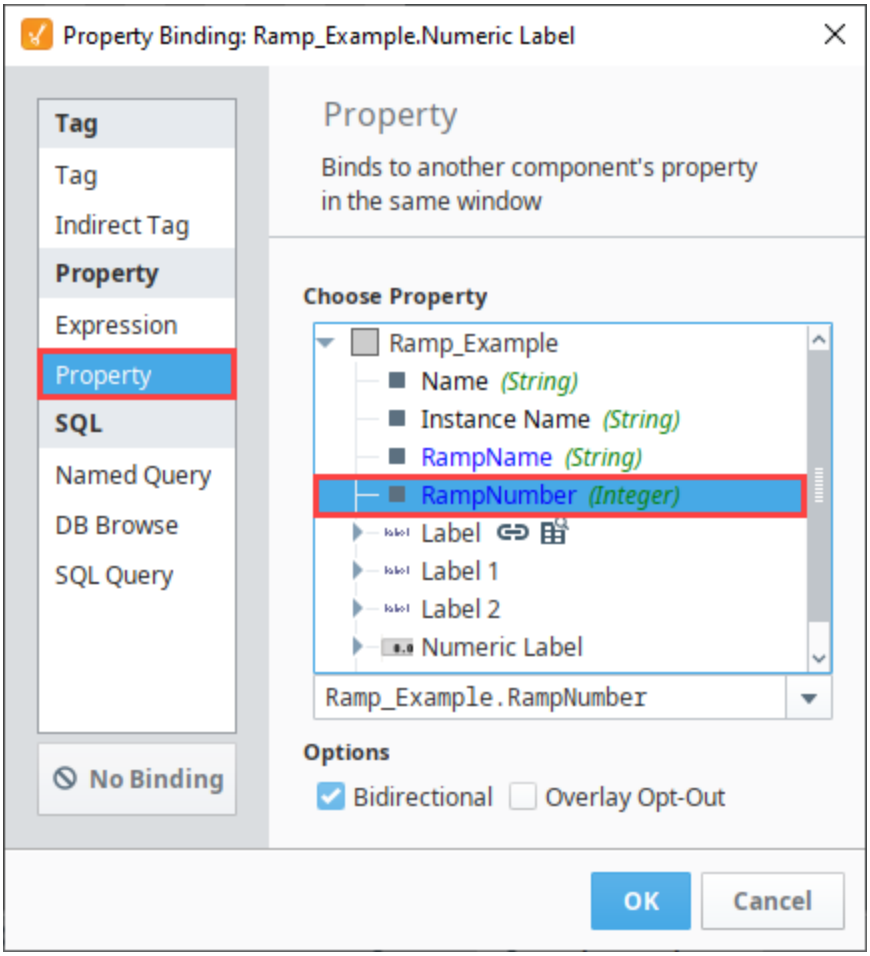
b. Resize the components so that they are easy enough to read. Place the blank Label component at the top of the Template, and place each Label that we wrote text into next to a Numeric Label.



4. Next, set up some bindings on our Template.
   a. Using a property binding, bind the **Text** property of the blank Label component at the top of the Template to the **RampName** Template Parameter.

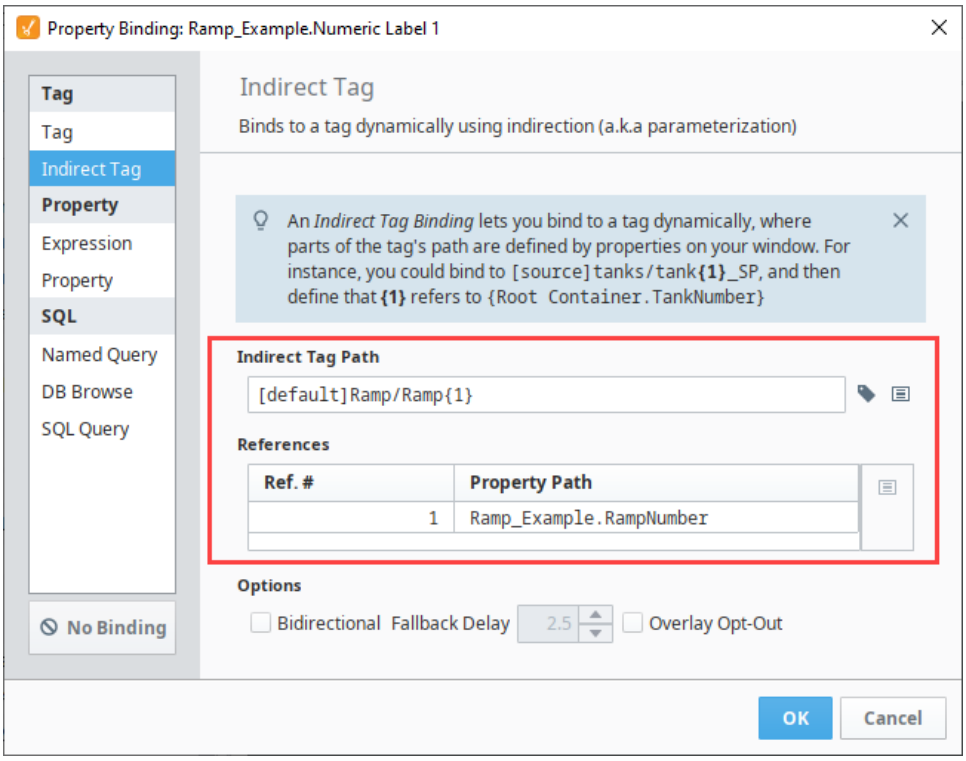


   b. Using a property binding, bind the **Value** property of the Numeric Label next to the "Ramp Number" label to the **RampNumber** Template Parameter.

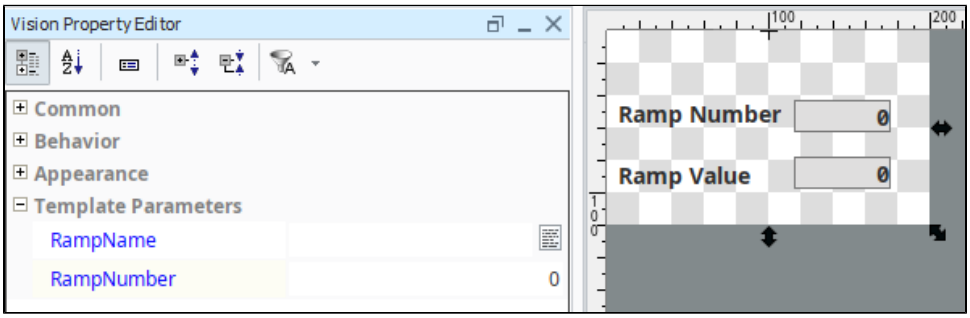


c. Using an indirect Tag binding, bind the **Value** property of the Numeric Label next to the "Ramp Value" label. We bound it to a Ramp Tag that is in our Tag Browser, using the **RampNumber** Template Parameter as the indirect reference for the number in the name of the Tag. It should look similar to the image below.

Note: Remember to remove the number portion of the tag name and replace it with {1}. For example, [default]Ramp/Ramp0 becomes [default]Ramp/Ramp{1}.



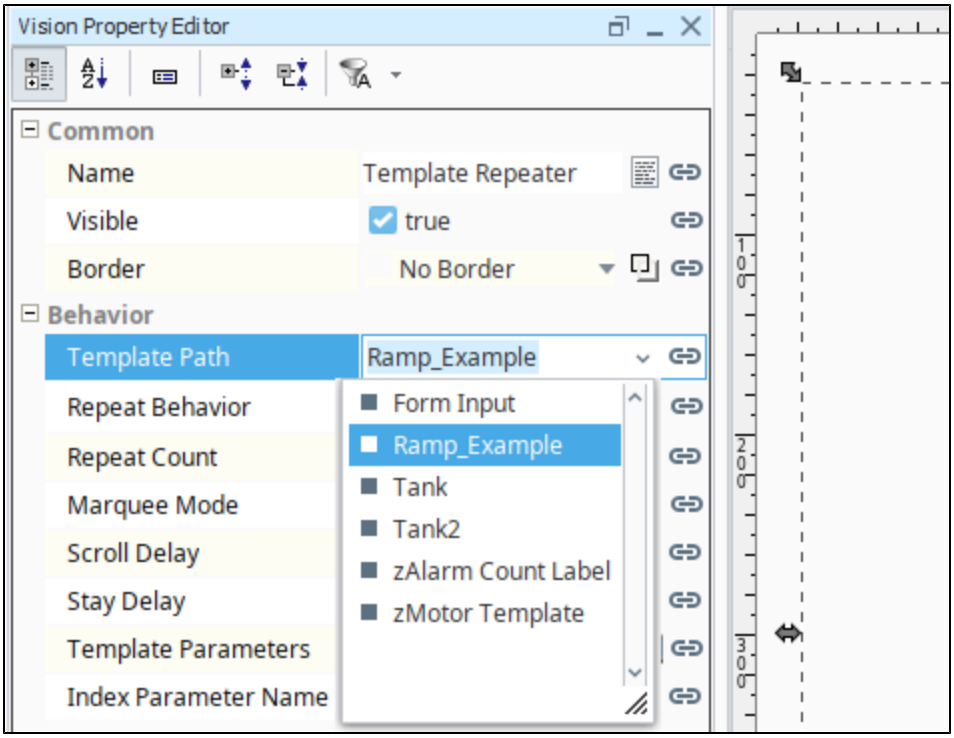
5. The template is now finished and ready to use. See the image below for an idea of what mine looks like. Yours should look similar. Notice the two Template Parameters, as well as the placement of the components. The label at the top that can't be seen because there is no text in it.



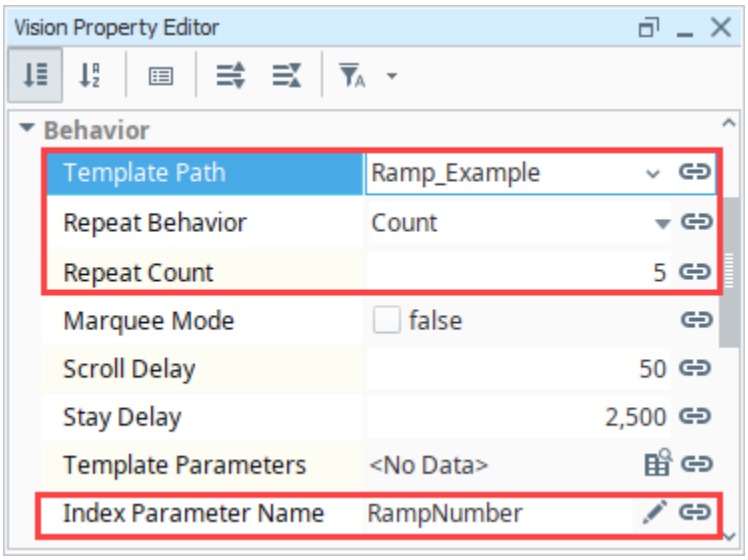
Using the Template Repeater with Count Mode

Now let's take a look at how to use the Template we just made in a Template Repeater.

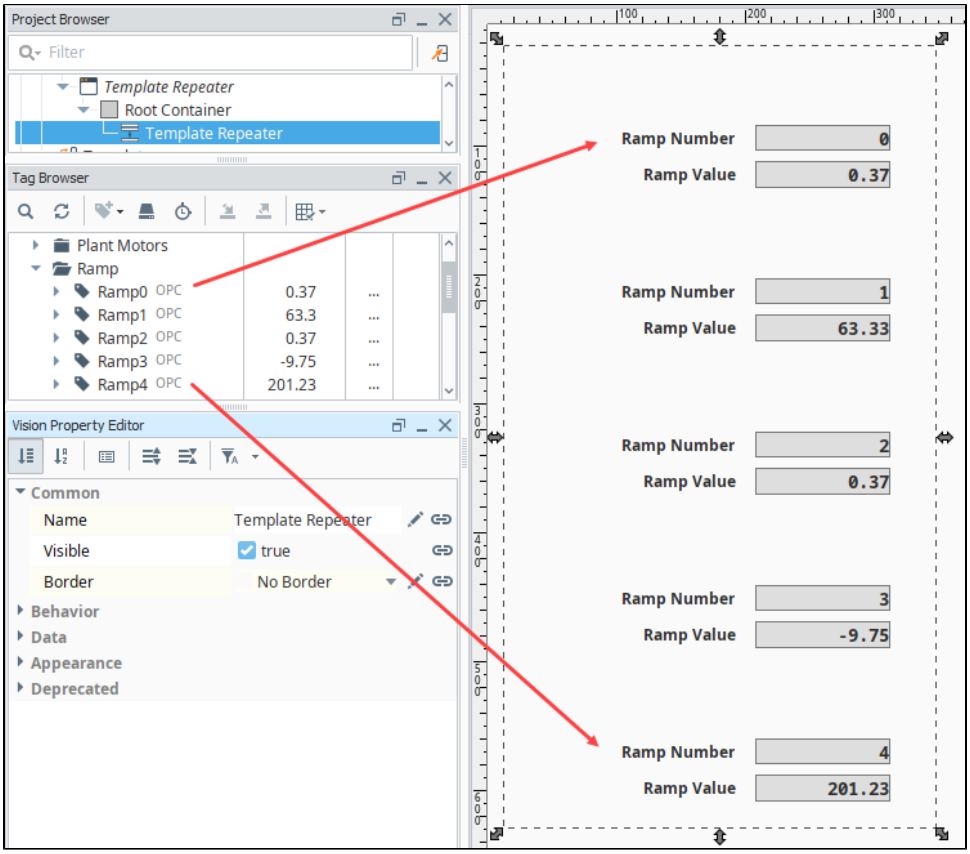
1. On a new window, drag a **Template Repeater** component. Resize it so that it is taller than it is long.
2. With the Template Repeater selected, find the **Template Path** property, and find the name of the template that we just created in the drop down menu.



3. With the Template Repeater still selected, find the **Index Parameter Name** property. Enter in '**RampNumber**', which was the name of one of our Template Parameters from our Template.
4. Ensure that the **Repeat Behavior** property of the Template Repeater is set to Count.
5. In the **Repeat Count** property of the Template Repeater, set the value to '5'.



6. You should see your Template repeated five times within the Template Repeater. Our Ramp Number value in the Template corresponds to the index value of the Template, 0 through 4 because it is 0 based. You will see that the value in our Ramp Value component corresponds to the value of one of our Ramp Tags.

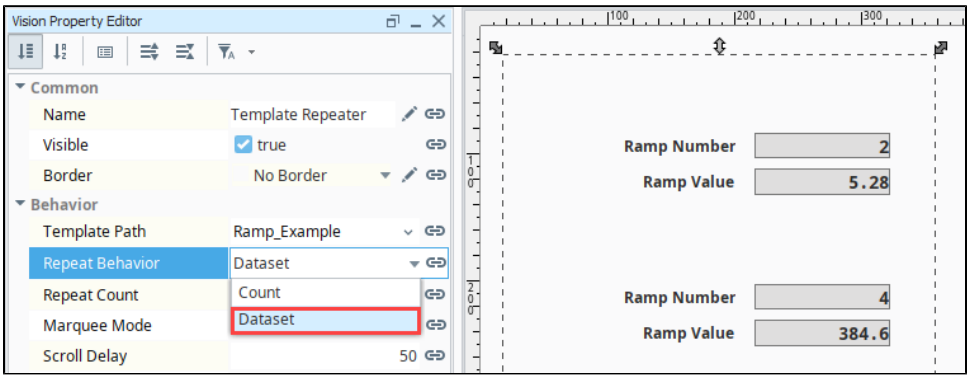


Your Repeater is complete. You can modify the Repeat Count to change how many times the Template gets repeated.

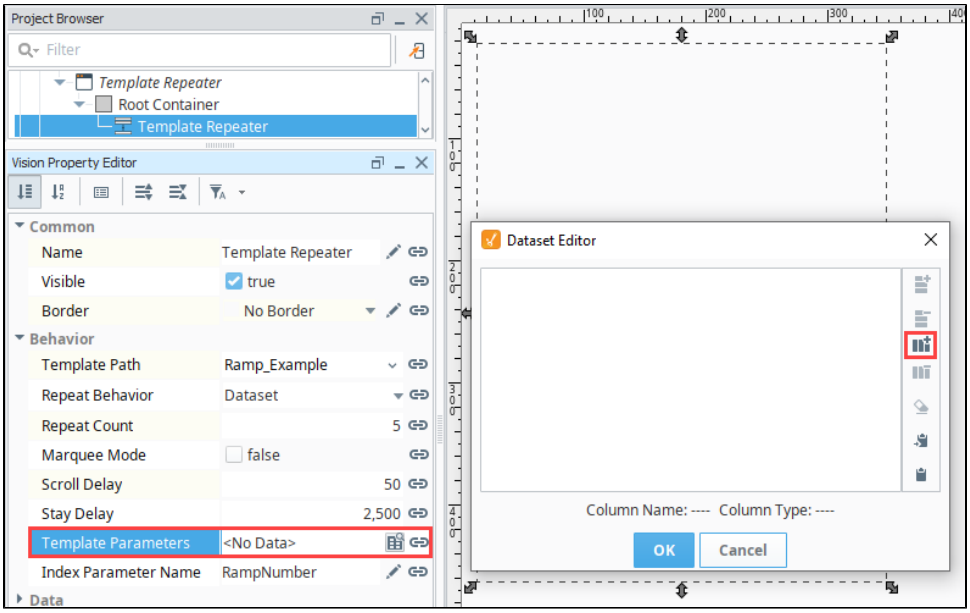
Using the Template Repeater with Dataset Mode

In our previous example, although we were able to successfully pass in the index parameter into our Template, we were not passing anything into the **RampName** Template Parameter and because of this, the blank label at the top of our Template remained blank. This is because **Count** mode only allows a single index parameter to be passed in. However, since we want to pass in two parameters, we can instead use **Dataset** mode. In addition, because we are going to be defining a dataset to pass in parameters, we don't have to use a 0 indexed parameter, and can instead use whatever values we want. We will use the same Template and Template Repeater from before.

1. On the Template Repeater, set the Repeat Behavior to **Dataset**. All of the Templates that were previously in will temporarily disappear.

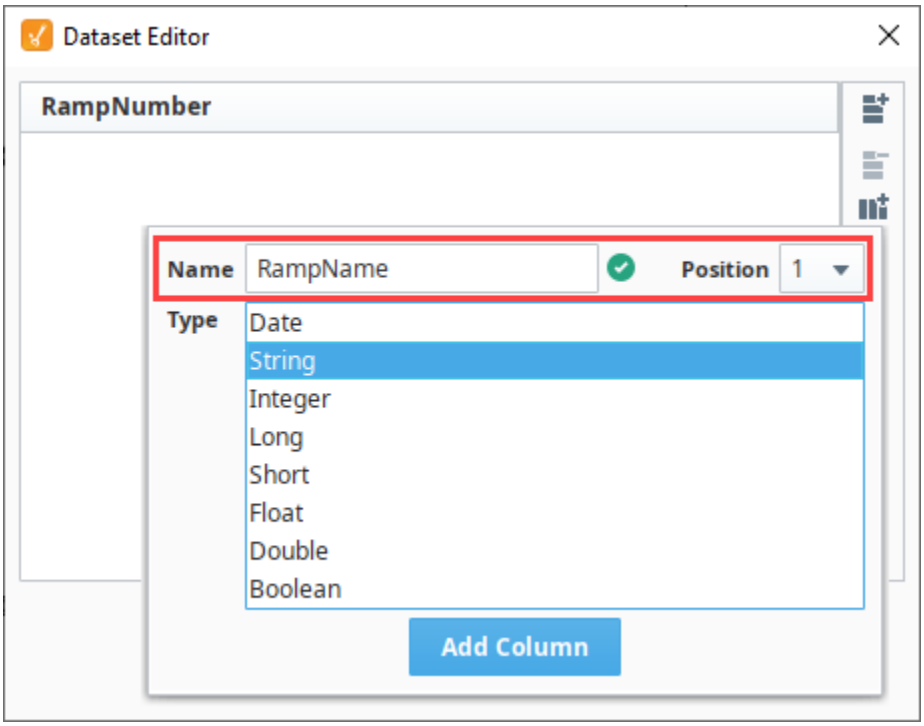


2. Click the **Dataset Viewer** icon on the **Template Parameters** property of the Template Repeater. You should see a new popup that allows you to build a dataset.



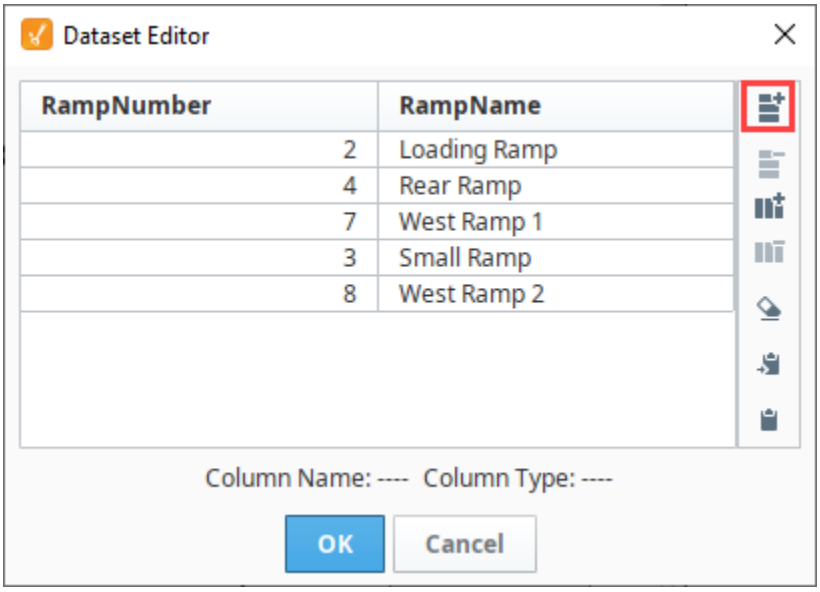
3. First, we'll add two columns to our dataset.
    a. Click the Add Column icon. Set the Name as '**RampNumber**' in **Position '0'**, and enter **'Integer'** as the **Type**. Click **Add Column**.
    b. Click the Add Column icon. Set the Name as '**RampName'** in **Position '1'**, and enter **'String'** as the **Type**. Click **Add Colum**.

> **Note:** Make sure the name of each column exactly matches the names of the Template Parameters (RampNumber and RampName). They are case sensitive.

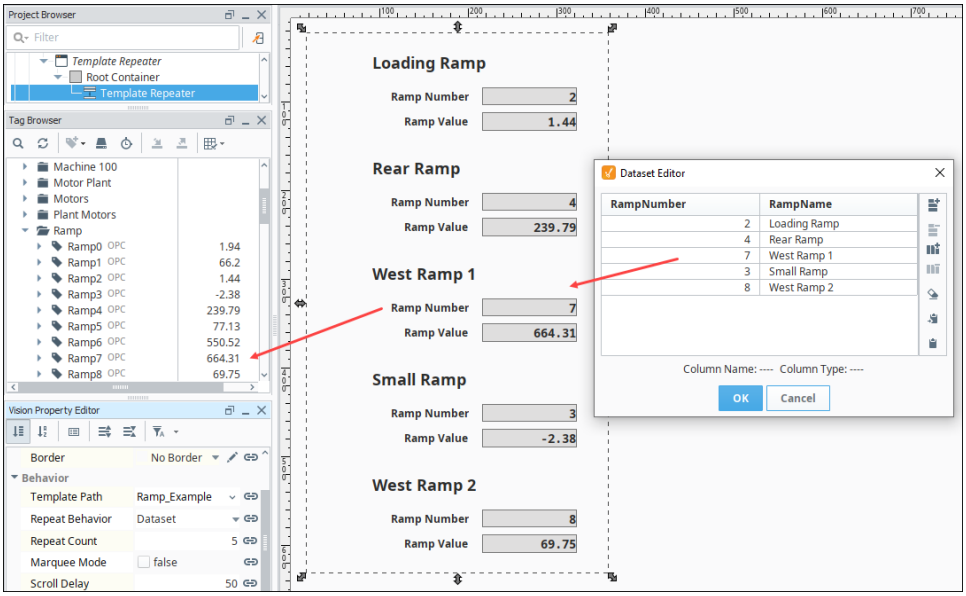


4. Next let's add some rows by clicking the **Add Row** icon. Click it five times to **add five rows**.
5. Add in some values for the RampNumber column. Since we are defining all of the values, we don't have to start with a **RampNumber** of '0'. In fact, we don't even need to have them be in sequence! We do need to make sure that the numbers each correspond to one of our Ramps, so enter in a value **'0'** through **'9'** for the **RampNumber** column for each row. In this example, we used 2, 5, 7, 3, 8 in that order.

6. Next, add some values for the **RampName**. These can be whatever you want, since we are going to be giving unique names to each of our Ramps. When finished, the dataset should look something like the image below. Click the **OK** button to save your dataset.



7. Notice how we now have our five templates back, and they are using the parameters that we are passing in from the dataset we just created. The Templates are receiving the parameters in the order that you made them in the dataset. We can also see that each Template instance corresponds to a specific ramp.



Dataset mode is great when you need to pass in multiple parameters, or if you have a single parameter that is not zero based. In addition to manually specifying values like we did in this example, you can instead set up a binding on the **Template Parameters** property to something like a database table. This allows you to easily modify the templates that are being displayed in the window simply by modifying the database table.

Related Topics ...

- Using the Template Canvas

Using the Template Canvas

The Template Canvas component works much like the Template Repeater in that it can easily create multiple copies of a master template. What makes the Template Canvas unique is that it can display instances of multiple master templates, and set their layout in any way you want. The Template Canvas has a customizer that can help put the templates together within it, but the customizer is just driven by a Templates dataset property on the Template Canvas. The Template Canvas can be made dynamic by setting up a binding on the Templates property, such as a query that pulls in an entirely new dataset of information, or even a cell update binding, which updates individual cells of the dataset. With that, you can load new templates into the canvas at runtime, or even move the templates around.

The Template Canvas has two layout systems:

- **Absolute Positioning**, where each template instance has an absolute position within the Template Canvas
- **Layout Positioning**, which uses the MiG Layout system to place the instances within a grid like system.

On this page ...

- Creating a Template
- Absolute Positioning
- Layout Positioning (MiG Layout)
- Read User Input

INDUCTIVE UNIVERSITY

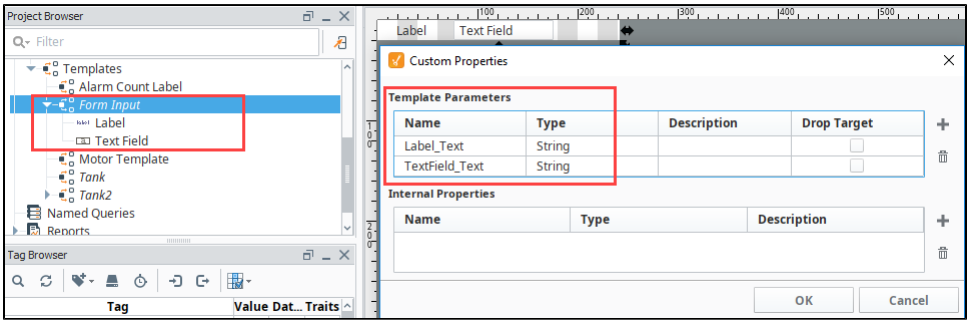
Template Canvas

Watch the Video

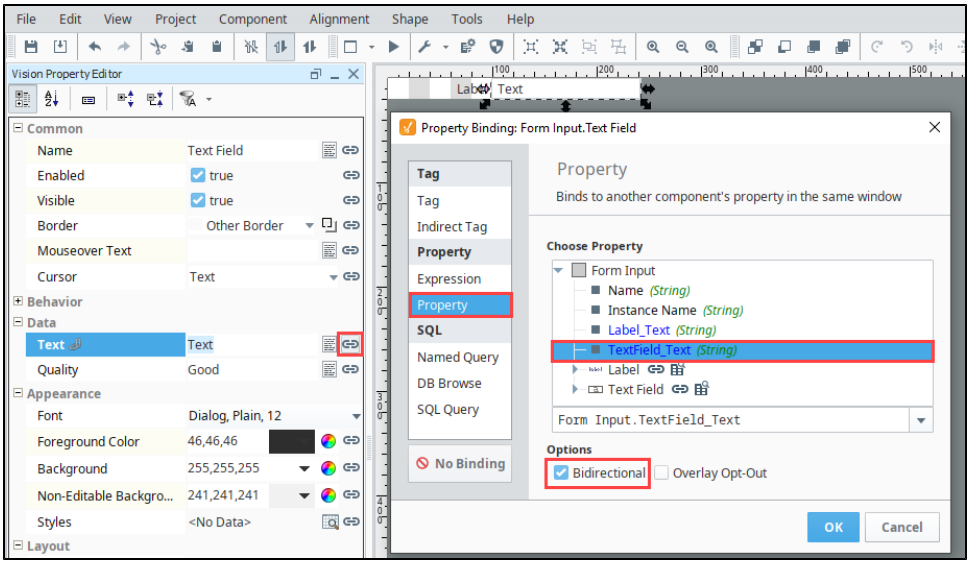
Creating a Template

Before we get started with the Template Canvas, we first need to have a template. Here we are going to make a simple Form Input template, that consists of a Label and a Text Field. The template will need two parameters. A **Label_Text** parameter which is what will get displayed in the label, and a **TextField_Text** which we can use to pass user input outside of the template by making it bidirectional. This template is a great way of quickly making user input forms, by using a template for each piece of info that needs to be collected. The steps for making the template are listed below, or you can skip ahead to using the Template Repeater in the next section.

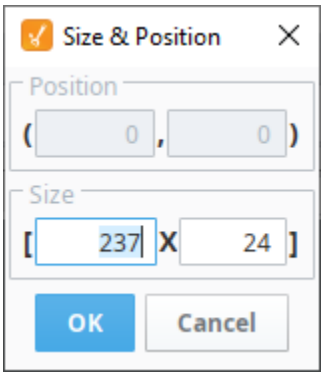
1. In the Templates section of our Project Browser, create a new template called **Form Input**. We are going to want to pass in values to and from the template, so we need to create Template Parameters.
2. Right-click in the Form Template workspace, and select **Customizers > Custom Properties**. Add a Template Parameter called **Label_Text**, and make it a **String** type.
3. Add a second Template Parameter called **TextField_Text**, and make it a **String** type, and click **OK** to save your template parameters.
4. Next, drag a **Text Field** component and a **Label** component onto the Template.
5. On the **Label** component, set the **Horizontal Alignment** property to **Trailing**.



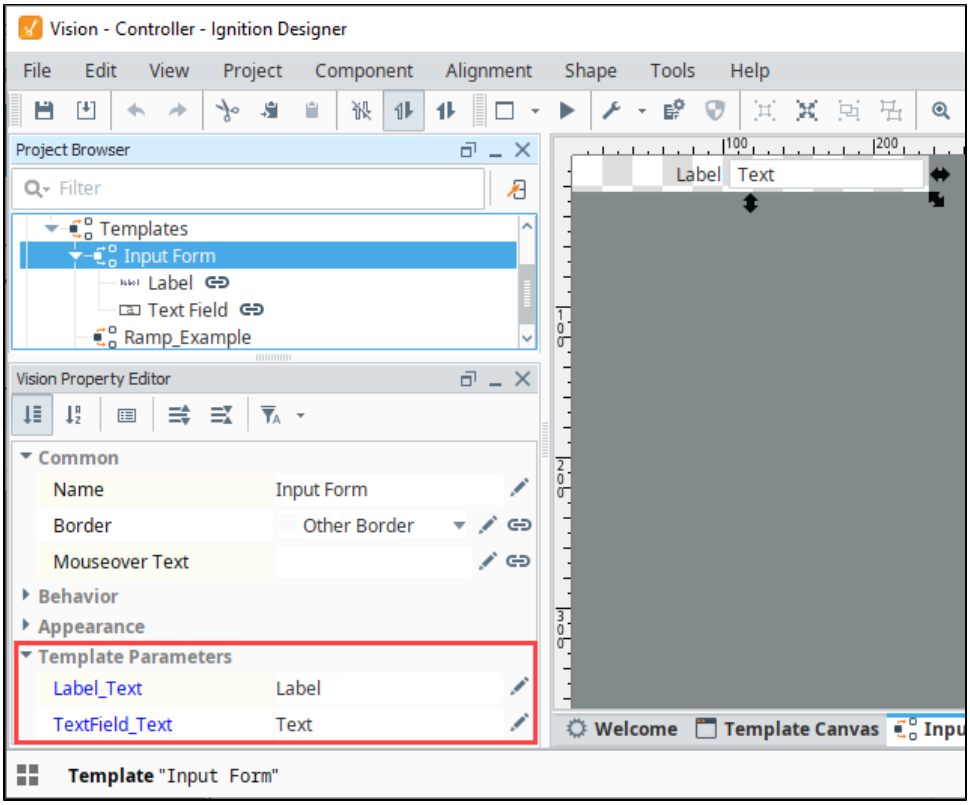
6. Finally, we want to set up some bindings on our Template.
    a. Using a property binding, bind the **Text** property of the **Label** component to the **Label_Text** Template Parameter.
    b. Using a property binding, bind the **Text** property of the **Text Field** component to the **TextField_Text** Template Parameter, and make it **Bidirectional**.



7. Set the Size and Position for the components by right-clicking on each component. Set the **Label** component to **Position: (68, 1)** and **Size: [32x20]**. Set the **Text Field** component to **Position: (104, 2)** and **Size: [130x20]**. Next, set the **Form Input** template to **Size: [237, 24]**.



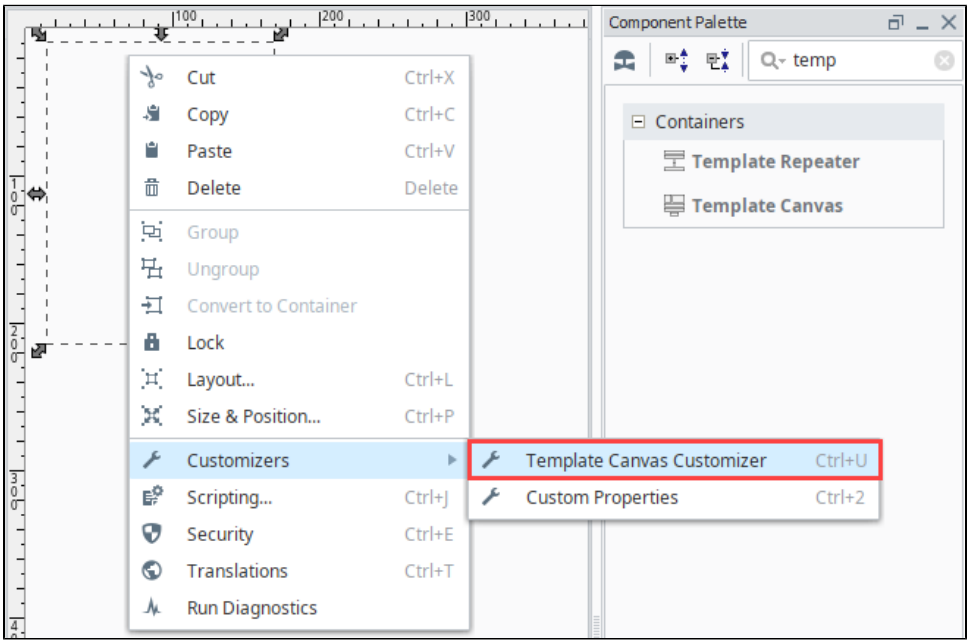
8. Once you have the properties added and components bound, type a value into each Template Parameter to confirm the bindings are working.
   a. Label_Text: **Label**
   b. TextField_Text: **Text**



Absolute Positioning

First, we can set up our Template Canvas using Absolute Positioning, which is a little simpler to understand as each template has a width and height as well as an x and y position.

1. Drag a **Template Canvas** component to a window and open the **Template Canvas Customizer**. The customizer provides an easy interface to setting up our templates.

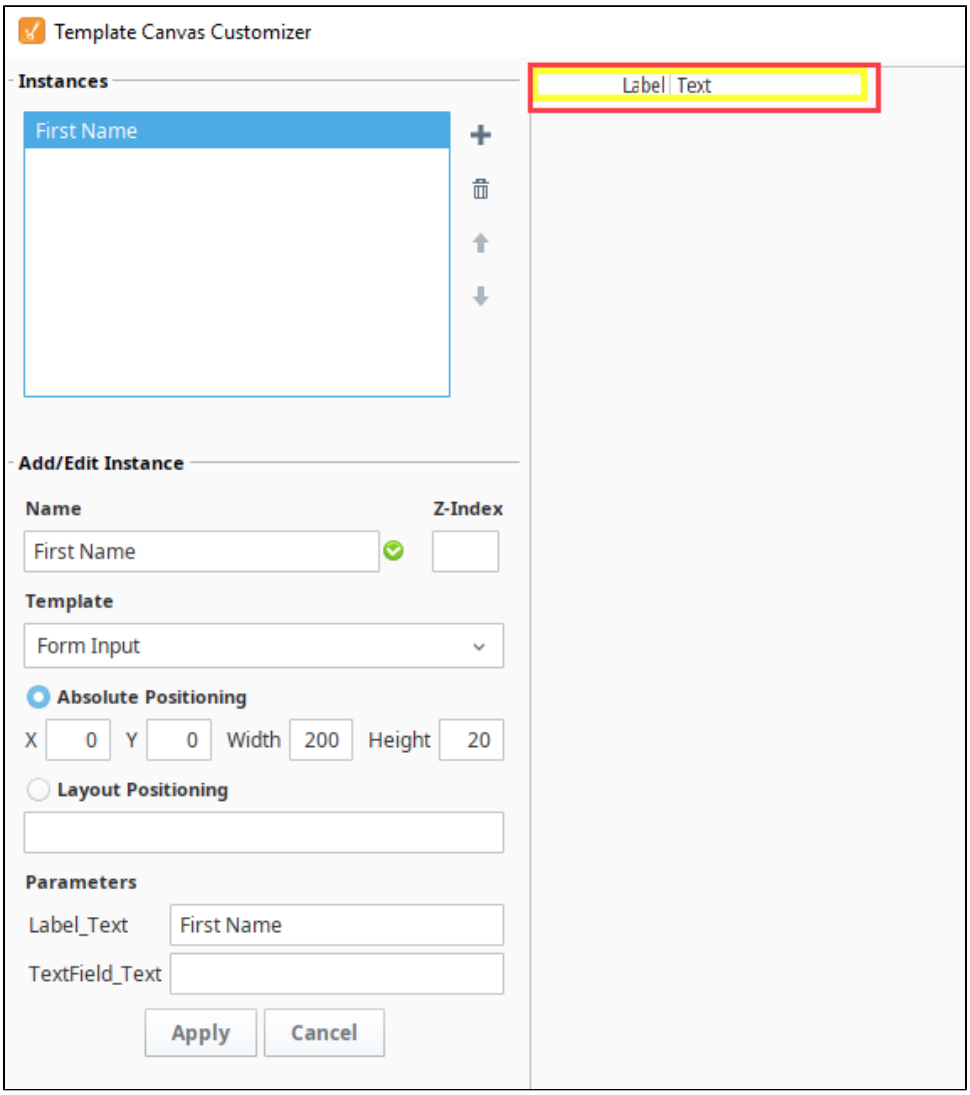


2. Enter in the following values for the first instance:
    a. **Name:** First Name
    b. **Template:** Form Input

c. **Absolute Positioning:** 0, 0, 200, 20
d. **Parameters:** Label_Text = First Name (leave TextField_Text blank)

3. Click the **Add** button to add the instance. The instance will then be visible in the preview section of the window. Values for the components are still using the default values for the Label_Text and TextField_Text. This is intentional. The new values will appear once you hit **OK** and close the canvas customizer.

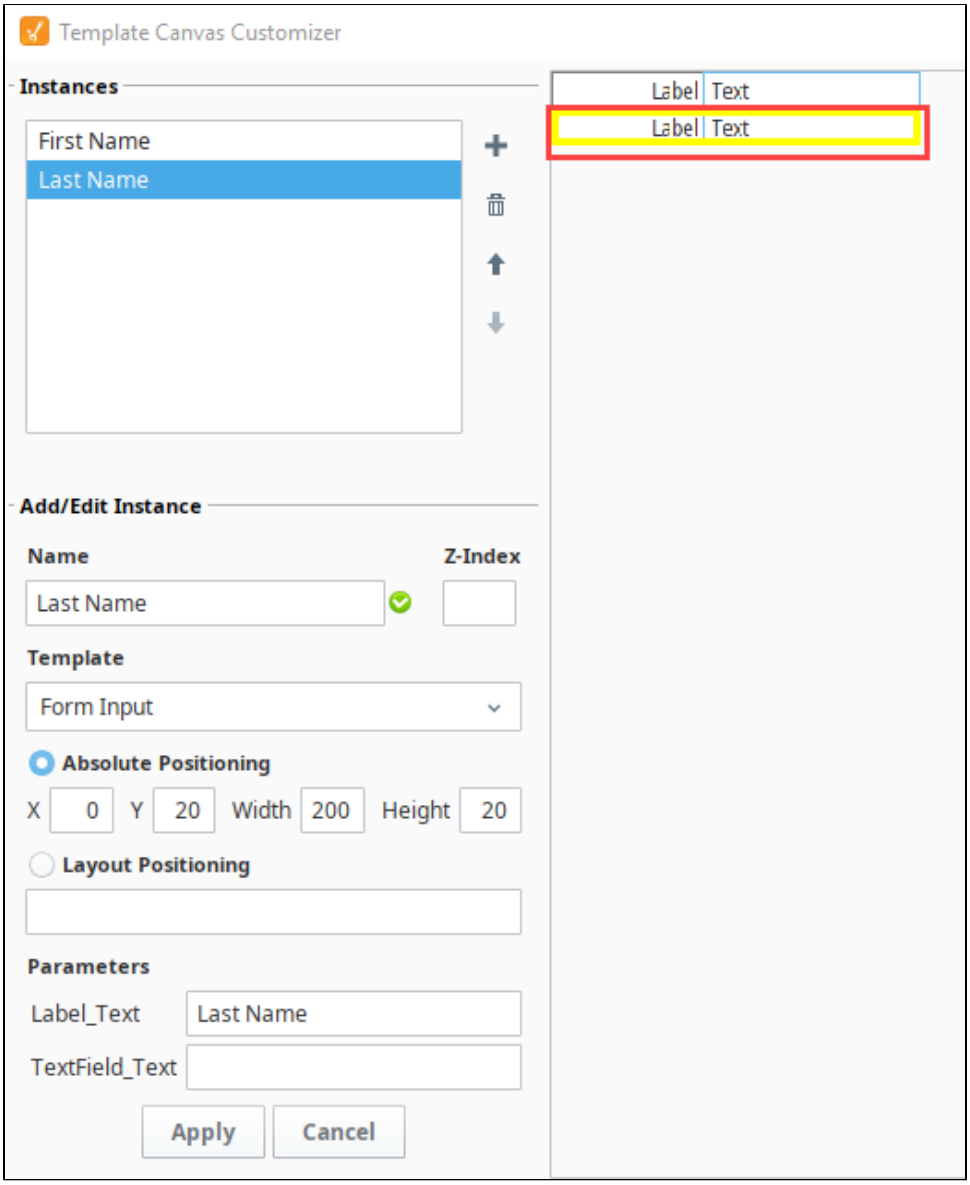
> **Note:** Labels have Absolute Positioning properties: X, Y, width, and height labels. A Z-index field indicates where a given template instance should appear in the z-order within the template canvas (top or bottom).



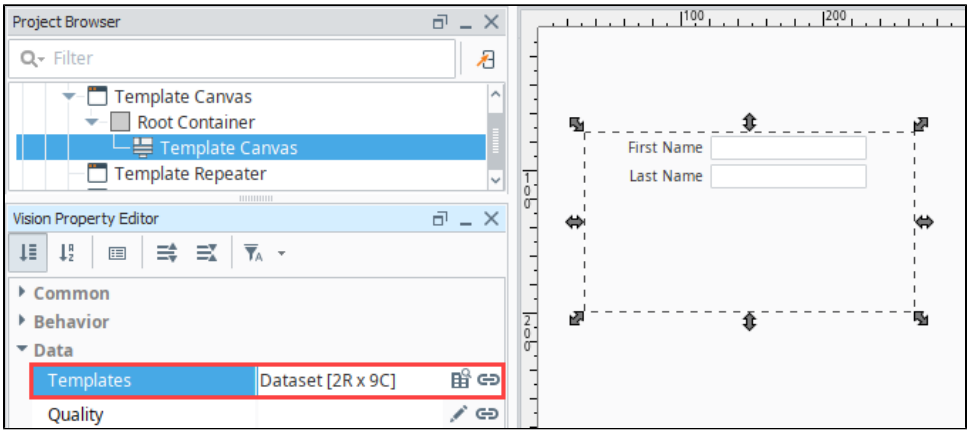
> **Note:** Take notice of of the yellow outline around the instance and how First Name is highlighted at the top of the customizer. This means that the instance is selected, and the customizer is in edit mode. This allows you to make changes to the selected instance. To exit edit mode and add a new instance, click on the **Cancel** button in the lower **left** of the window. Clicking on the **Cancel** button in the lower **right** will cancel out of the customizer.

4. Let's add another instance. Click on the **Add** + icon in the **Instance area** to clear the Add/Edit Instance fields from the prior entry, and enter the following:
   a. **Name:** Last Name
   b. **Template:** Form Input
   c. **Absolute Positioning:** 0, 20, 200, 20
   d. **Parameters:** Label_Text = Last Name (leave TextField_Text blank)

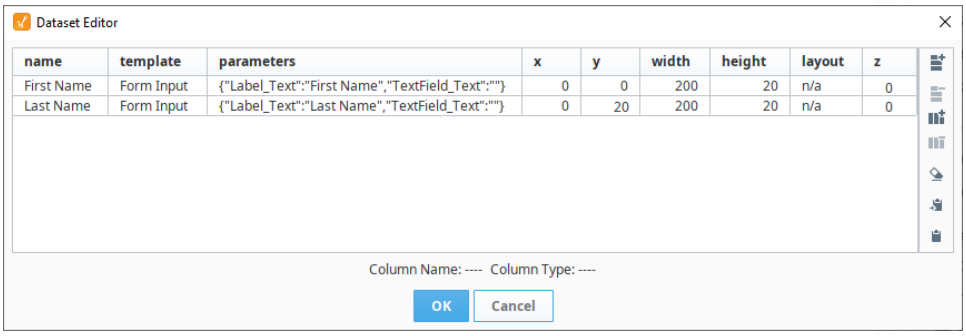
5. Once entered, click the **Add** icon again. You'll see that you have two instances visible in the preview section of the window. Once both instances are configured, click the **OK** button.



6. Now, both instances appear in the Template Canvas.



7. If a new instance needs to be added, it can be added through the Template Canvas Customizer. However, the Template Canvas also has a **'Templates'** property. This property stores all of the data that was entered into the customizer into a dataset, so new instances can be configured directly on the Templates property. View the dataset by clicking the **Dataset Viewer** button next to the **Templates** property. Furthermore, template instance configurations could be stored in a database table, and the Template Canvas could fetch the data with a SQL Query binding on the Templates property.



Layout Positioning (MiG Layout)

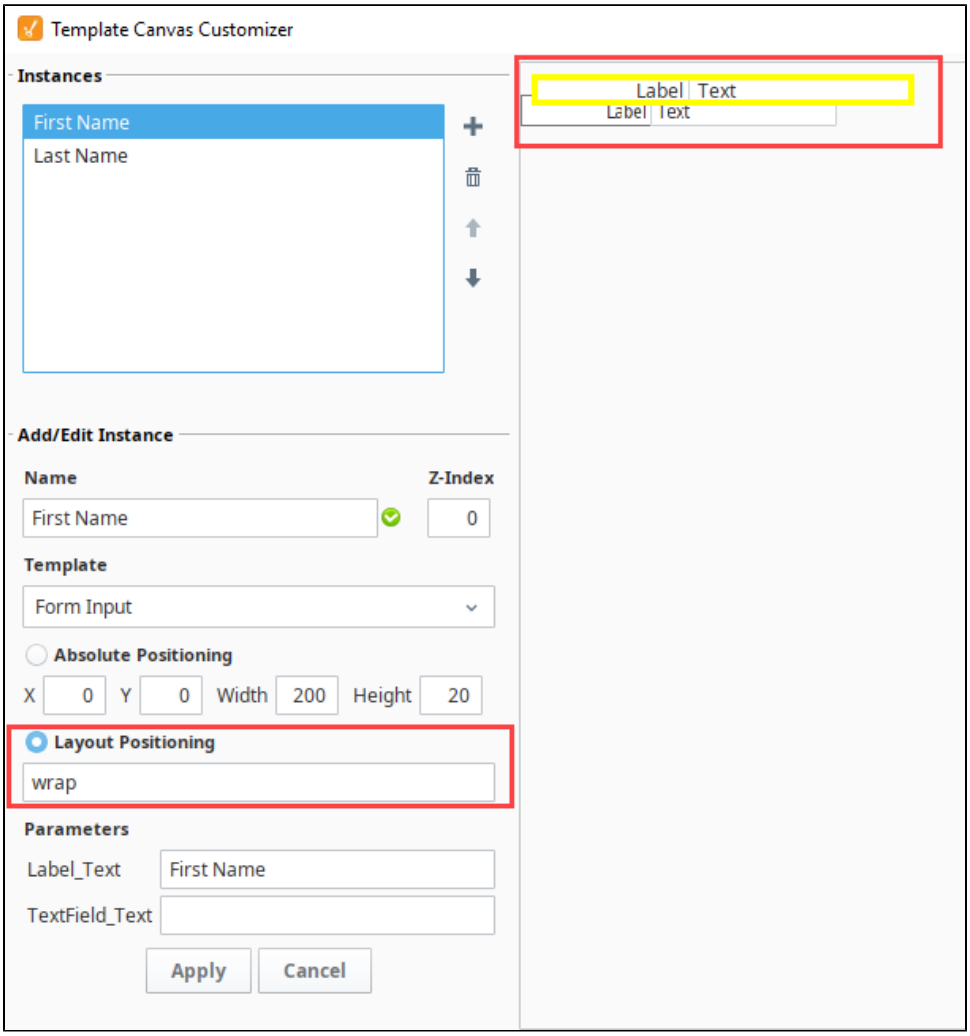
Instead of having to manually enter a size and position for each instance, we can make use of Layout Positioning to have the Template Canvas determine the best position for each instance, while also making suggestions as to where each instance is placed in relation to another. The layout positioning uses a grid-methodology to instance placement. Each instance, unless otherwise specified, is considered a single "cell" in the grid.

Caution: Don't Mix Absolute and Layout Positioning
We do not recommend using both Absolute and Layout Positioning for instances on the same Template Canvas. Select either Absolute or Layout Positioning for your instances. Layout Positioning will determine the best position for each instance in your canvas, where Absolute Positioning allows you to manually specify the width, height, x and y positions for each instance.

Continuing from the example above, the First Name and Last Name instances are using Absolute Positioning. Let's tell the First Name instance to use Layout Positioning and enter the **'wrap'** parameter. This means the next cell in the grid should be placed on the next row.

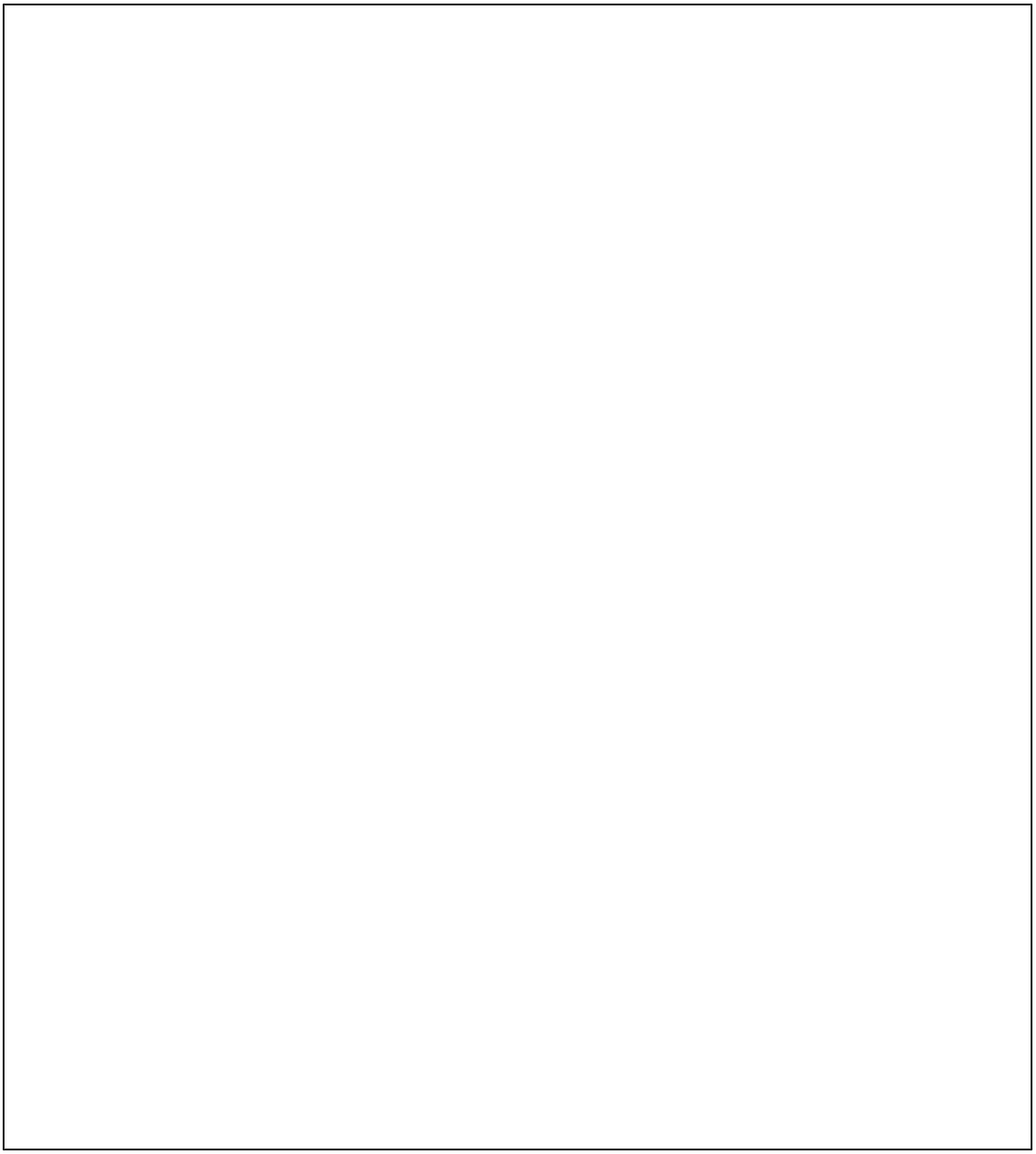
1. With the Template Canvas component selected, open the the **Template Canvas Customizer** again, and make the following modification to the **First Name** instance.
    a. Click the radio button for the **Layout Positioning** property and enter **'wrap'** the field.

2. Click **Apply**, and the First Name instance will appear to overlap with the Last Name instance. This is because the grid only accounts for instances using the Layout Positioning.

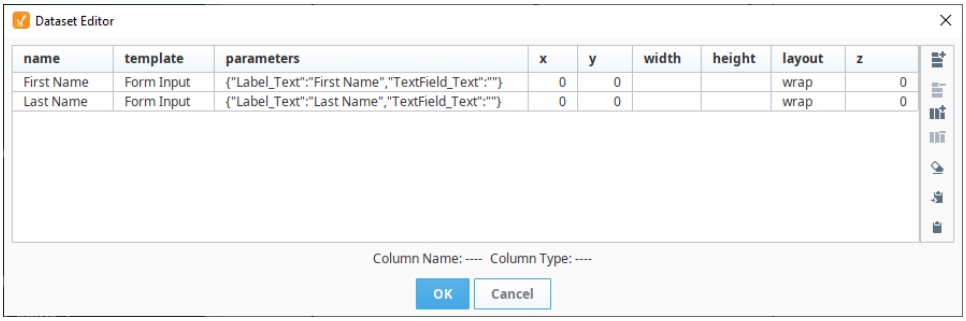


3. Next, we can configure the Last Name with Layout Positioning as well. Make the following changes to the Last Name.
    a. Click **Layout Positioning** radio button and enter **'wrap'** in the field below.

4. Once the changes have been applied, click **OK**. You'll notice in the preview section of the window that both instances are now in line.

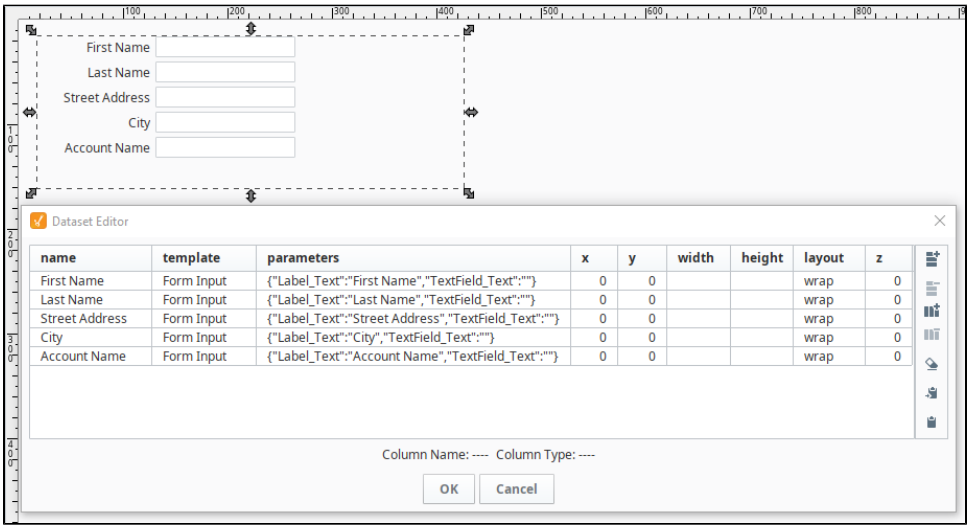


5. Open the **Templates** property by checking the **Dataset View** button. Notice that the x, y, width, and height columns are no longer used, but the **layout** columns for First Name and Last Name now have a value of **'wrap'.**



6. Like the previous example, new rows can be added directly to this dataset. Furthermore, the **'wrap'** value means the next template instance will begin on a new line. Add three new instances for Street Address, City and Account Name. Use either the Template Canvas Customizer or simply add new rows in the dataset viewer with the values:

name	template	parameters	x	y	width	height	layout	z
Street Address	Form Input	{"Label_Text":"Street Address","TextField_Text":""}	0	0			wrap	0
City	Form Input	{"Label_Text":"City","TextField_Text":""}	0	0			wrap	0
Account Name	Form Input	{"Label_Text":"Account Name","TextField_Text":""}	0	0			wrap	0



Read User Input

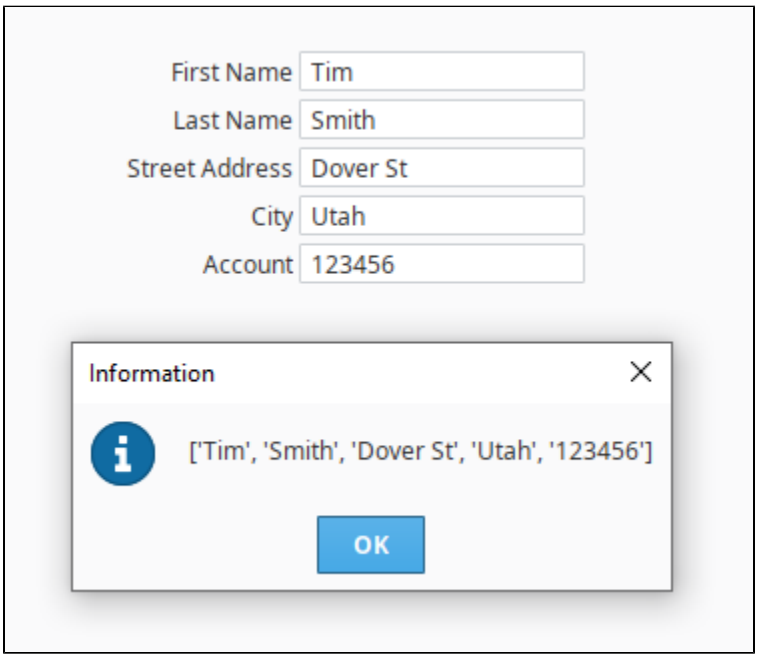
The last step is to read the user input. Put the Designer into **Preview Mode** and add some values for each text field component. Once finished, switch the Designer back to **Design Mode**, and add a Button component to the window (not the template canvas)

Add a script to the Button component using the Code Snippet below - **Read User Input Example.** Place the code on the **actionPerformed** event of your component by double clicking on the Button component and opening the **Script Editor** tab.

```
# Reference the template canvas component, and call the getAllTemplates() method.
# This will return a list of every instance in the canvas
templateList = event.source.parent.getComponent('Template Canvas').getAllTemplates()

# Initialize a list. User input from each text field will be stored in this variable
userInput = []

# Iterate through each template instance inside the canvas
for template in templateList:

    # add the user inputted value to the userInput list. The values are originally returned in Unicode.
    # the Python str() function is casting the Unicode values as string values.
    userInput.append(str(template.TextField_Text))

# Show the values in a messageBox. This could be replaced with an INSERT query, or some other action.
# str() is used again to case the list as a string. This only required to work with the messageBox function
# since the function requires a string argument be passed in
system.gui.messageBox(str(userInput))
```

When running the script, each value should appear in the message box. If you're not getting a value in the message box, make sure the Text Field property is bound to the TextField_Text template parameter as mentioned in the Creating a Template section, Step 6b.

This example can easily be expanded to do something more meaningful with the input, like store to a database table.



Security in Vision

Security in Vision is managed through one of two authentication strategies, either the Classic Authentication strategy or Identity Providers (IdP) strategy. Classic Authentication Strategy involves a concept known as a User Source, which is a configuration that contains multiple roles and users. IdPs allow users to authenticate against a trusted third party. Refer to Security for more information on these authentication strategies.

On this page ...

- Selecting an Authentication Strategy
- Client Authentication Strategies
  - Classic Strategy
  - Identity Provider Strategy
- Vision Client Permissions
- Role-Driven Client Security
- Incorporating Scripting into Security

Selecting an Authentication Strategy

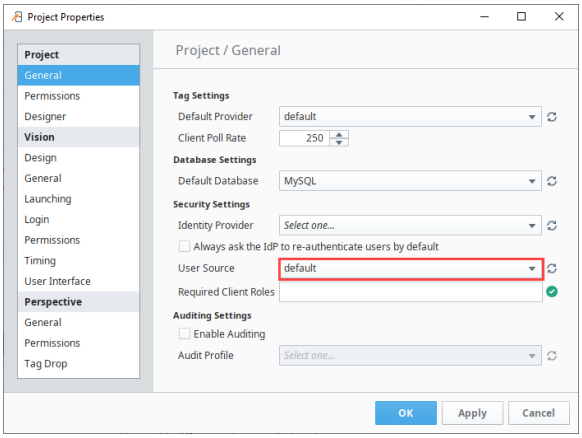
The strategy used by any given project is determined by the Authentication Strategy, found under Project Properties. See Vision Project Properties for more information.

Client Authentication Strategies

Users are given access based off of the Authentication Strategy the project is set to.

Classic Strategy

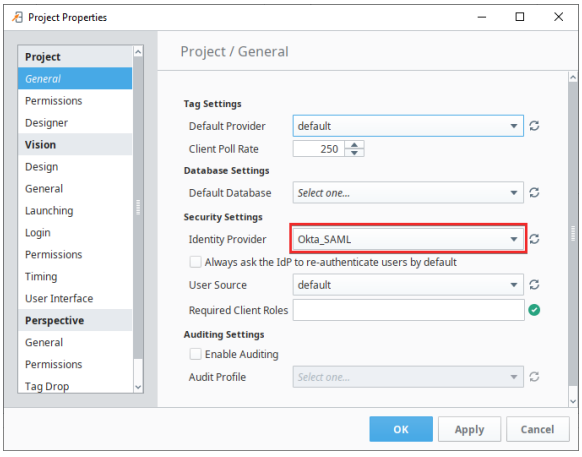
When using the classic strategy, a **User Source** needs to be assigned to the project. Users and roles are taken from the assigned User Source.



When utilizing this strategy, the **Required Client Roles** field can be used to limit access to the entire Vision Client based on a roles requirement.

Identity Provider Strategy

When using the identity provider strategy, an **Identity Provider** needs to be assigned to the project. Users are taken from the assigned IdP.



Vision's access control model is based on roles and security zones. Thus, the `Authenticated /Roles...` and `SecurityZones/...` security levels in the IdP are converted into roles and zones,

respectively. As a result, IdPs need to have user attribute mapping configured for a given IdP before Vision can utilize role based access.

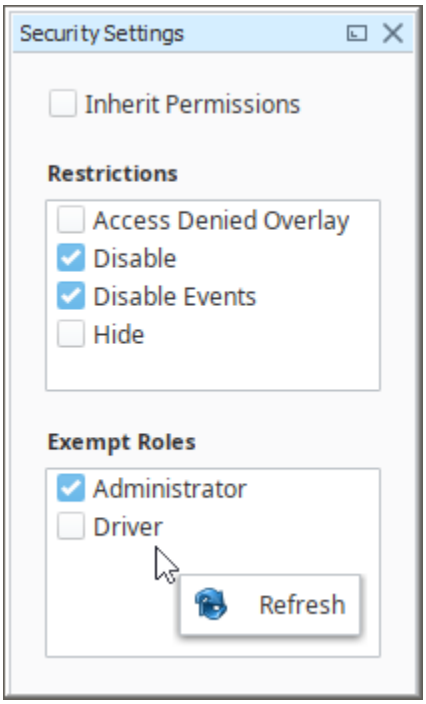
Vision Client Permissions

Every project's clients are governed by a set of permissions to control what is allowed to originate from the client. For example: access to construct queries against the database, or the ability to edit Users and Roles in your authentication profile. To maintain a secure system, these are all set to disabled by default, but you can enable them for everyone, for specific users, or even for specific users that are logged into certain zones. See descriptions of these categories and how to change them on the Project Permissions page.

Role-Driven Client Security

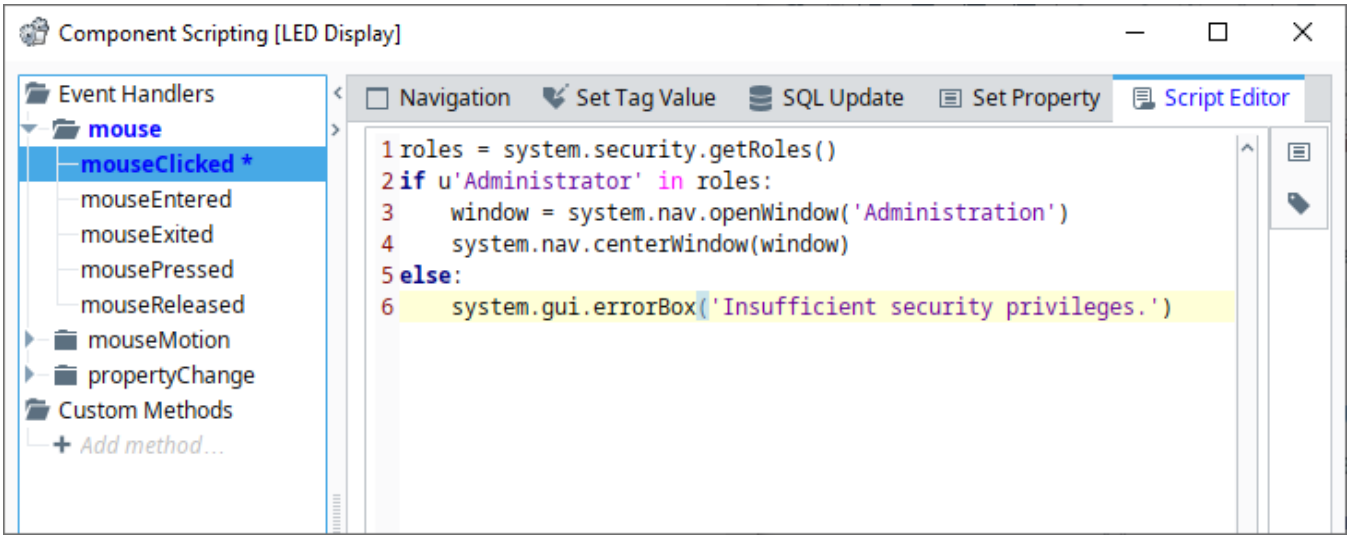
On the simplest level, security settings can be applied to individual windows or components. Users with different roles can all view the same project from the client, but the functionality and readability can change based on the roles assigned to each user. Generally, higher level access provides full functionality to all contents of a project, and lower level access is restricted to generalized read-only privileges.

Below, we see the Security Settings panel in action. This panel is the interface that applies Ignition's built-in security settings. Security settings can be applied to a single component, multiple components simultaneously, or even a whole window. Users who should be allowed full access can be selected, and restrictions can be applied for users that should not have full access.



Incorporating Scripting into Security

The component-based security settings are fairly simplistic: the user either has the required roles, or a restriction is applied. In situations where consideration for access should go beyond a simple role check, security-based scripting can provide a larger degree of granularity. Information about the logged-in user, such as user-name or roles, can be detected by scripting, allowing for the creation of a robust security system.



In This Section ...

Component and Window Security

Role-based security inside of Vision works on multiple levels: component, group, container, and window levels. Each of these levels also have special categories of security that help with tuning security to various design considerations.

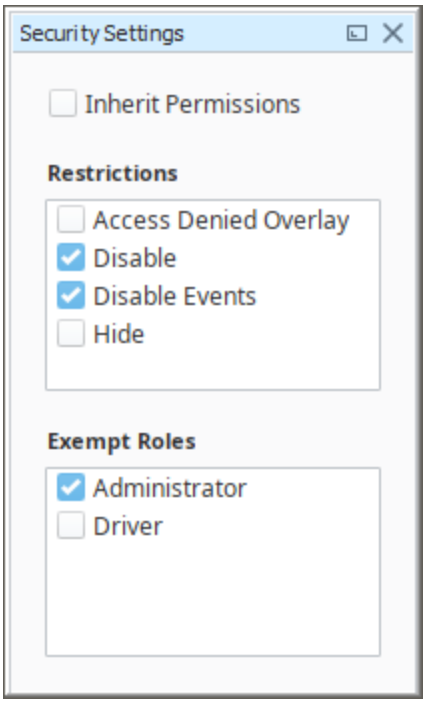
Each window and component can define its own security settings. These settings determine who can see and/or use the component. It's good business practice to have a well thought out security policy for your project.

On this page ...

- Changing Security Settings on a Component
- Exempt Roles

Changing Security Settings on a Component

By default, each component inherits the security that is on its parent which initially gives anyone with access to the project the ability to use the components. This can be changed on a per component basis by right clicking on the component and selecting **Security**. This brings up the Security Settings panel.



The **Inherit Permissions** checkbox signifies that the component is inheriting the security settings of the parent.

In the **Restrictions** section on the right, choose the restrictions that will be placed on the component if the user does not have the selected role. Multiple restrictions can be selected to combine their effects.

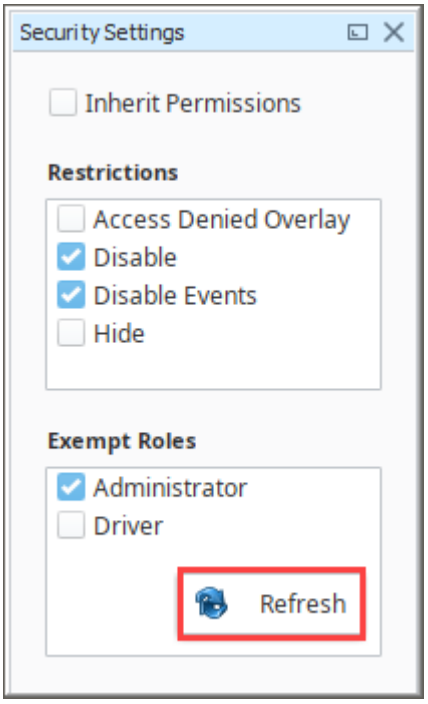
Restriction	Used On	Description	Image
Access Denied Overlay	Components	Shows an overlay on top of the component when the user doesn't have security clearance.	Access Denied
Disable	Components	Sets the Enabled property to false on the component when the window opens up. **Caution:** If you choose to disable a component, make sure that it is a component that actually does something different when it is disabled. For example, buttons and input boxes can't be used when they are disabled, but disabling a Label component has no effect.	Disabled 
Disable Events	Components, Root Container	Prevents event scripts from running when the user doesn't have security clearance.	N/A
Hide	Components, Root Container	Sets the Visible property to false on the component when the window opens up.	

			Hidden
Do Not Open	The Window Object	Only used on the window object itself, will prevent the window from opening if the user doesn't have one of the specified roles.	N/A

Exempt Roles

Unchecking the **Inherit Permissions** checkbox enables the role checkboxes under Exempt Roles. Each role that is selected will have access to the component. So if the Administrator role was checked, then all users with the Administrator role will be able to use the component, while users without the role would have Restrictions to the component. A user only needs to have one of the selected roles to be able to use the component, not all of them.

Note: If the roles you created do not appear, it is probably because the Designer was open before those roles were created. To update the list of roles, right-click in the **Exempt Roles** section and select **Refresh**.



Security in Scripting

While the Vision system has many options for securing individual components and clients, it is possible to have requirements that go further than the options that are available. With scripting, you can create any type of security that you may need. This is mainly done through the use of the system.security.getRoles function, as well as the other system.security functions. The getRoles function gets a list of the users roles, which you can check for specific roles within your script. You can then write code for what would happen if the user has the role and if they don't have the role.

Additionally, while the typical security set up only requires the user to have one of the required roles, in scripting you can ensure that the user has any combination of roles.

On this page ...

- Securing Event Handlers
  - Script Builder Security Example
- Security in Client Event Scripts
  - Setting Client to Read Only

Securing Event Handlers

Security can be added to any of the event handlers in Ignition. This works on both components as well as on event handlers within the Scripting window. While the typical security settings for a window only give you the option of not opening it if the user does not meet the required roles, you can instead do something else like opening a different window.

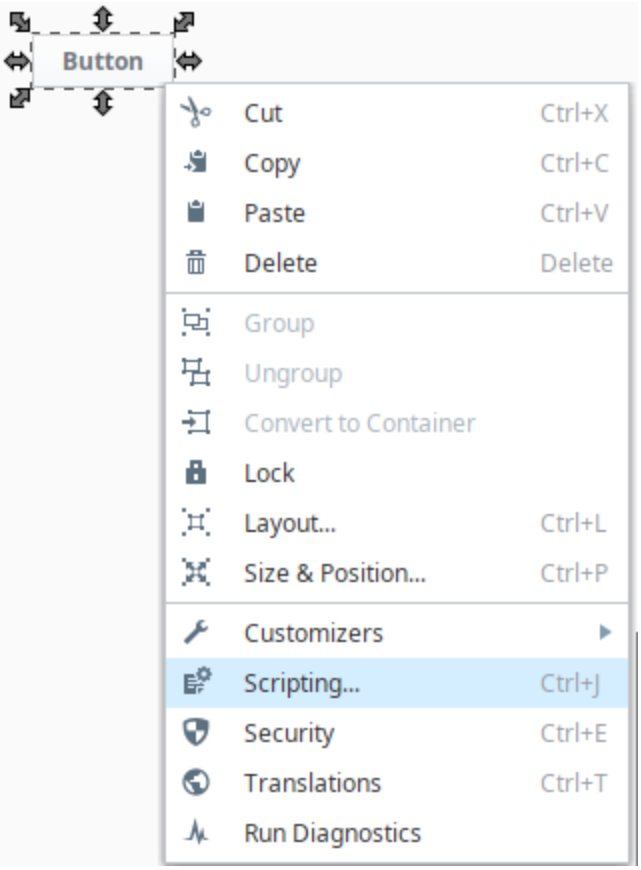
Securing Event Handlers

Watch the Video

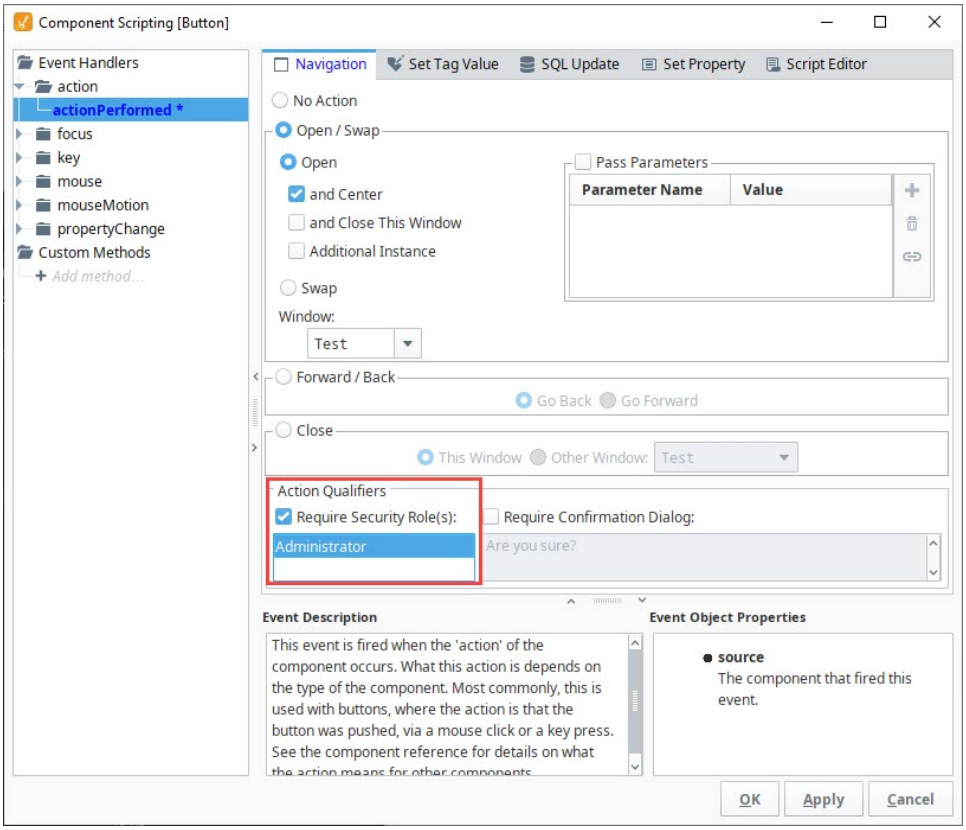
Script Builder Security Example

Each one of the script builders also has the ability to add security to them. The following is an example of setting up security on a Button component.

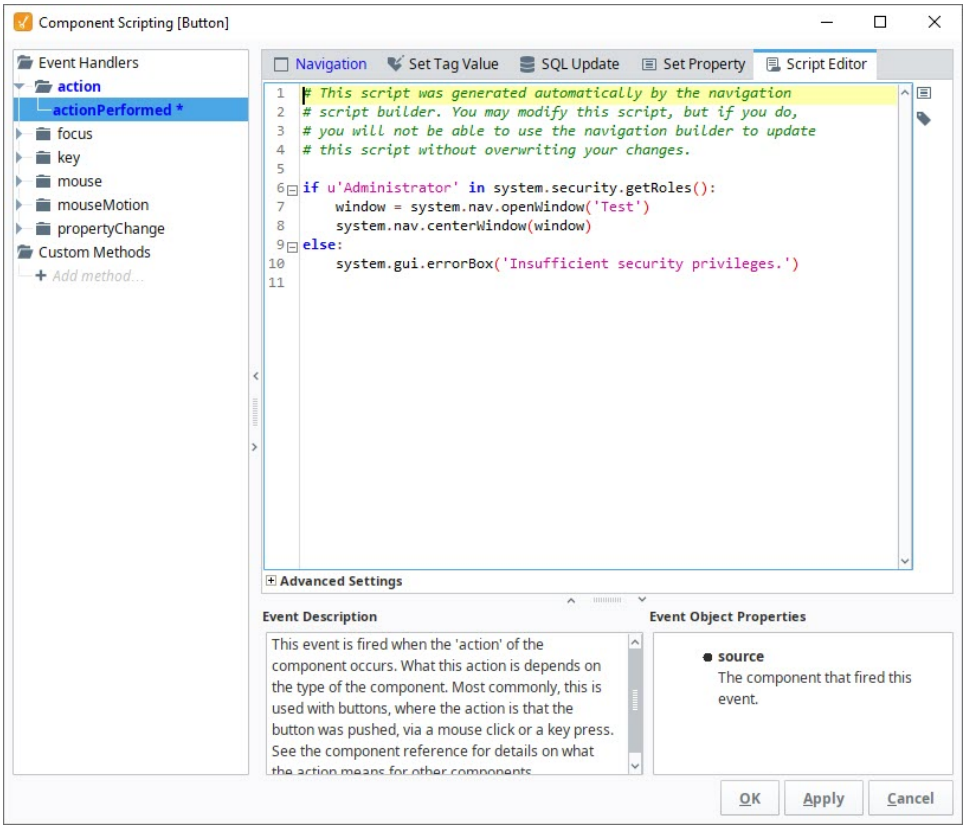
1. Drag a **Button** component onto a window.
2. Right click on the component, and choose **Scripting**.



3. Under the Navigation tab, click the **Open/Swap** radio button.
4. From the Window dropdown list, choose a window for this navigation.
5. Click the **Security** button in the **Action Qualifiers** section. Click the **Required Roles** checkbox.
6. Select the roles that are required for this navigation. Click **Close**.



7. Next, click on the **Script Editor** tab. You'll see the script that is generated by the options you chose in the Navigation tab.



8. Click **OK** to save the scripting. Now you added security to the Button component.

Security in Client Event Scripts

The Client Event Scripts can also be used to set up security within the project. While any of the Client Events can be used for different security purposes, the most common is the **Startup Script**. This allows you to create a customized, secure environment right as the Client is started. A Script here can do certain things based on the roles of the user such as open certain windows, write to client Tags to enable or disable certain things within a project, or even retarget to an entirely new project.

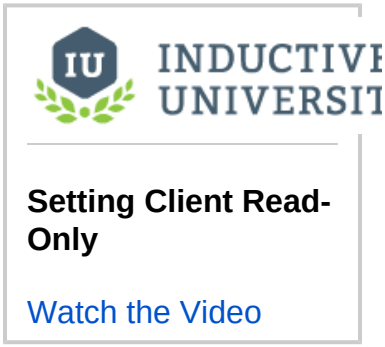
Setting Client to Read Only

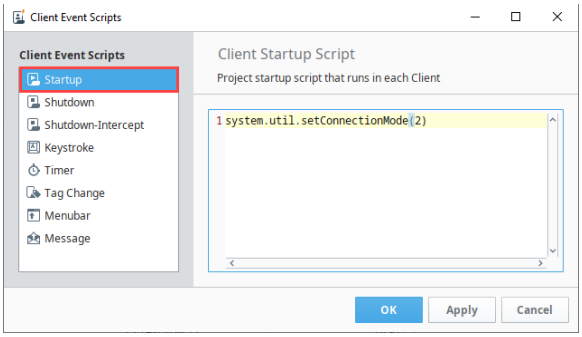
There are times when it is best to open a Client in a Read-Only mode to eliminate the possibility that a Client will affect a device or database. The Client event startup script that sets the Client mode to Read-Only is an easy way to accomplish this. Similar to the buttons in the Designer, this function can be used to set Disconnected, Read-Only, and Read/Write modes in any script in Ignition that runs in a Client. This function can be called in any Client scoped script, but is most commonly used in the Startup script.

This example creates a Client event script that sets the Connection Mode to Read-Only.

1. From the Designer, go to **Project Browser > Client Event Scripts**.
   The Client Event Scripts window is displayed.

2. In the **Startup** script area, enter the following: `system.util.setConnectionMode(2)` where 2 means Read-Only.





3. Click **OK**. The startup script will run the next time a user logs into the Client, resulting in the Client being Read-Only.

Scripting in Vision

A lot of the scripting that happens in Vision is either located on components and windows, or it manipulates component and window values. Many times, you can easily add the path to another component or property from your script. It is important to understand how the component hierarchy of a window works as well as how to properly access components and properties in a script anywhere in a project

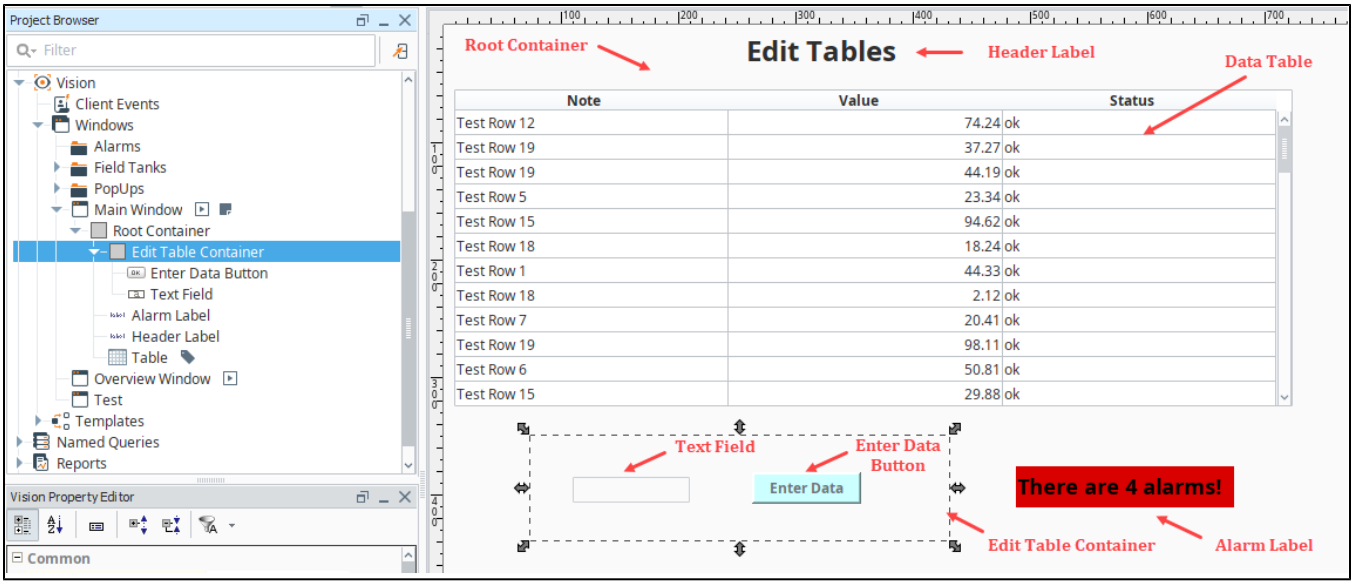
On this page ...

- Component Hierarchy
  - Accessing Component Properties
- Accessing a Component
  - From an Event Handler
  - From an Extension Function
  - From a Client Event Script
  - From a Project Script
- Accessing Components on Other Windows
- Complex Property Types

Component Hierarchy

Every window in a project has a hierarchy to it, with components and containers arranged in a tree structure, with a single parent up at the top, and many children down at the bottom. While small windows with only a few components can have a simple hierarchy, windows with many containers can get more complex. Since the components are arranged like a tree, this means that we can only move up and down through the tree structure and not sideways. To get from one component to a sibling component, we must first navigate up towards the common parent, and then back down to the desired child. Let's take a look at an example of how this works. Here we have a simple window, with just a few components:

- Root Container
  - Edit Table Container
    - Enter Data Button
    - Text Field
  - Alarm Label
  - Data Table
  - Header Label



Flipping the tree around can give a better understanding of why we can only move up and down through the tree structure, and how getting to a component can be very different depending on where you start. Let's say we want to go to the Text Field so that we can grab its value.

Start From	Path to Text Field
Edit Table Container	1. Down to Text Field
Enter Data Button	1. Up to Edit Table Container 2. Down to Text Field
Data Table	1. Up to Root Container 2. Down to Edit Table Container 3. Down to Text Field

To move up or down within the hierarchy, there are two special commands we can use on component objects: **parent** and **getComponent()**. The **parent** property allows us to grab a reference to whatever is directly above the component in the hierarchy, which in most cases is the root container. We could then access any component on the root container by using **getComponent("Component Name")** and then placing the name of the component we want to access within the parenthesis.

Pseudocode - Component Hierarchy

```
# This pseudo code shows how to grab the parent of a component
component.parent

# This pseudo code shows how to grab a child of a component
component.getComponent("Text Field")
```

Both **parent** and **getComponent()** can be used as many times as necessary to reach the desired component, drilling up or down through layers of containers or or grouped components. Once you have a component reference, you can then access any one of that component's attributes by using the name of the property, just like when accessing a property on the source component.

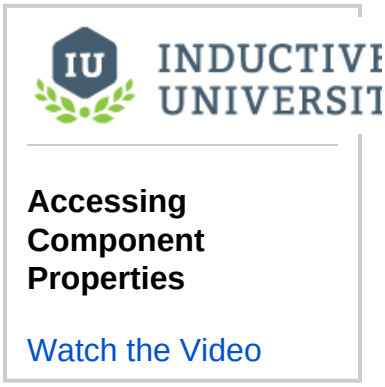
Exception to the pattern of using .parent
There is one exception to the pattern of using `.parent` to go up the hierarchy and using `.getComponent(name)` to go down. The parent of a root container is not the window, and a reference to the window does not have a `.getComponent(name)` function. To get a reference to a window, simply use system.gui.getParentWindow with any component's `event` object as the parameter. Once you have a reference to a window, you can use its `.rootContainer` property to get to the root of the component hierarchy, and from here you can follow the rules laid out above.

Accessing Component Properties

To access a property within the component, we simply need to use the scripting name of the property. The scripting name can be found in the description of each property, either by enabling the description field or hovering over the property until the mouseover text appears. The scripting names for every property on every component can also be found in the appendix. For a text field, the scripting name of the text property is just text, so we would need to call that on the text field which has the text property we want to access.

Pseudocode - Component Properties

```
# This pseudo code will access the text property of a component and assign
it to value.
value = component.text
```



Accessing a Component

Now that we have an understanding of how the component hierarchy works, we can apply that knowledge to accessing a component from anywhere within the project. While moving up and down within the hierarchy remains the same, accessing our initial component can differ depending on where we start our script from.

From an Event Handler

Event Handlers get a special **event** object that has special properties depending on the type of event. Regardless of the event, all event objects have a source property, which gives the component that fired the event. When accessing a component from an event handler, we can first use `event.source` to get a component on the window to start at. From there, we can use parent or getComponent() as needed to get to the component we need to access.

Python - Accessing a Component from an Event Handler

```
# This would access the text property of a Text Field component.
print event.source.parent.getComponent('Text Field').text
```

When accessing a component from an event on the window, there will be a different path to the component than normal. If you already have a window object, you can use the function getComponentForPath(). This allows you to enter in the path to the component as a string (similar to expression bindings), and will end up looking something like this:

Python - Accessing a Component from an Event on a Window

```
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text Field').text
```

You can also get the Root Container directly using the getRootContainer() function. This link of code works the same as the one above:

Python - Accessing a Component from an Event on a Window

```
system.gui.getParentWindow(event).getRootContainer().getComponent('Text Field').text
```

From an Extension Function

Extension Functions get a special **self** object which is actually a direct reference to the component that the extension function is on. This provides a direct reference point from which to access other components within the component hierarchy.

Python - Accessing a Property of a Component from an Extension Function

```
# This would access the text property of the component running the extension function script.
print self.text

# This would access the text property of the component named 'Text Field' if it is in the same container.
print self.parent.getComponent('Text Field').text
```

From a Client Event Script

Client Event Scripts are special because they don't start with a direct reference to anything on a particular window. So, we have to use another means of finding a starting point on the window. The system.gui.getWindow() function allows us to get a reference to a window which we can use to navigate to the root container with `.getRootContainer(),` and then to any component on that window. However, this will only work if the window is currently opened. If the window is closed, it will throw an error, which can be handled with normal exception handling.

Python - Accessing a Component from a Client Event Script

```
# Start the try block in case the window is not open.
try:
	# Grab the window reference and assign it to the variable window.
	window = system.gui.getWindow("Other Window")

	# Use the window reference to get the text property off of a text field.
	print window.getRootContainer().getComponent("Text Field").text

# Handle the exception by opening an informative error.
except ValueError:
	system.gui.errorBox("The window is not open!", "Error")
```

From a Project Script

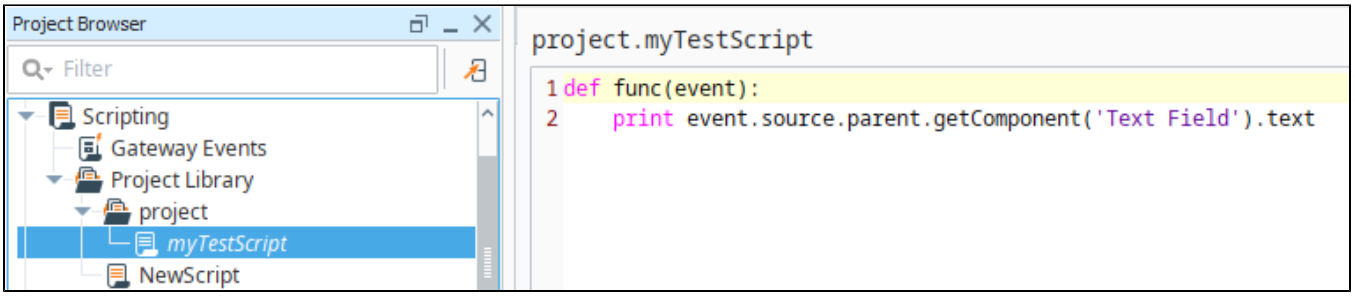
Project Library are unique in that how they access components can vary depending on where the Script Module is being called from and what is being passed to it. If the script module is being called from an event handler or an extension function, it is possible to pass in the event or self objects and use them within the script module.

Python - Accessing a Component from a Project Script

```
# This code would go in a project script. We are defining our function that takes an event object
# and uses it to find the value of the text property on the text field in the same container.
def func(event):
	print event.source.parent.getComponent('Text Field').text
```

Python - Calling a Function from the actionPerformed Button

```
# On the action performed of a button on our window, we could then use this to call our function.
myTestScript.func(event)
```



However, this may not always be the case. In these instances, it is possible to instead use the same method that Client Event Scripts use and grab the window object instead.

Accessing Components on Other Windows

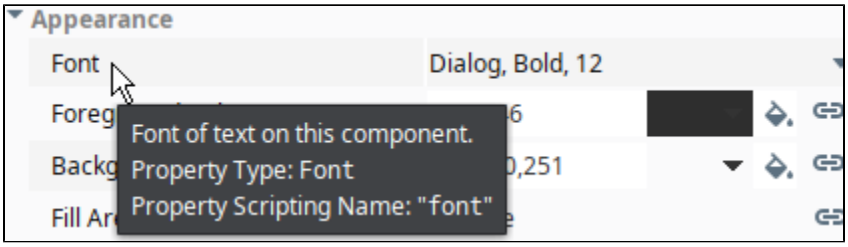
You can also grab properties from components on other open windows from anywhere in the project using the same method used in Client Event Scripts. This allows you to grab properties on a main window from an event handler on a popup window.

Note:

Remember, you can only grab a property from another window if the other window is open.

Complex Property Types

Some component properties have more complex property values. For example, Font properties typically have a type of "Font".



In most cases these property types are simply using some built-in Java AWT types. These can be manipulated from scripting in Ignition by importing the appropriate library.

```
from java.awt import Font

event.source.parent.getComponent('Text Field').font = Font('Dialog', Font.BOLD, 50)
```

See the AWT javadocs for more information.

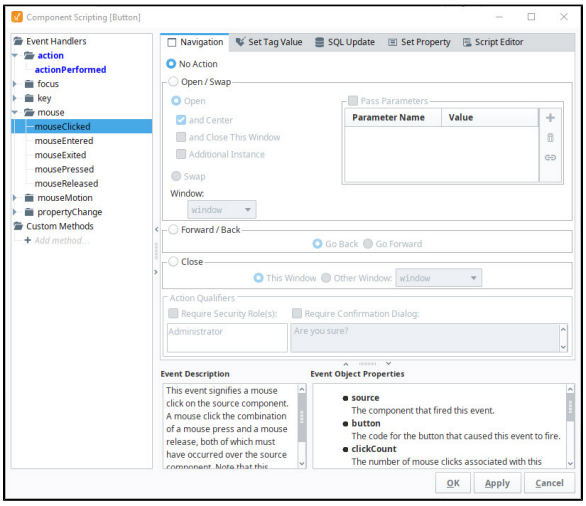
Related Topics ...

- Project Library

In This Section ...

Script Builders in Vision

When creating an Event Handler on a component, you can use one of the handy **Script Builders** instead of writing your own script. In the Component Scripting window, the Script Builders are accessible as tabs along the top. Each tab represents a different kind of action that users can associate with an event. The last tab, **Script Editor**, lets you write your own event handler. You can also use the **Script Editor** tab to view a script that was generated by one of the builders, which is a good way to get started learning how to write your own event handlers. Each script builder example on this page shows the actual script in the Script Editor, just simply click the link under each example.



On this page ...

- Navigation Script Builder
  - Open / Swap
  - Forward / Back
  - Closing Windows
- Set Tag Value Script Builder
- SQL Update Script Builder
- Set Property Script Builder
- Script Editor
  - Advanced Settings
- Action Qualifiers
  - Security Qualifier
  - Confirmation Qualifier

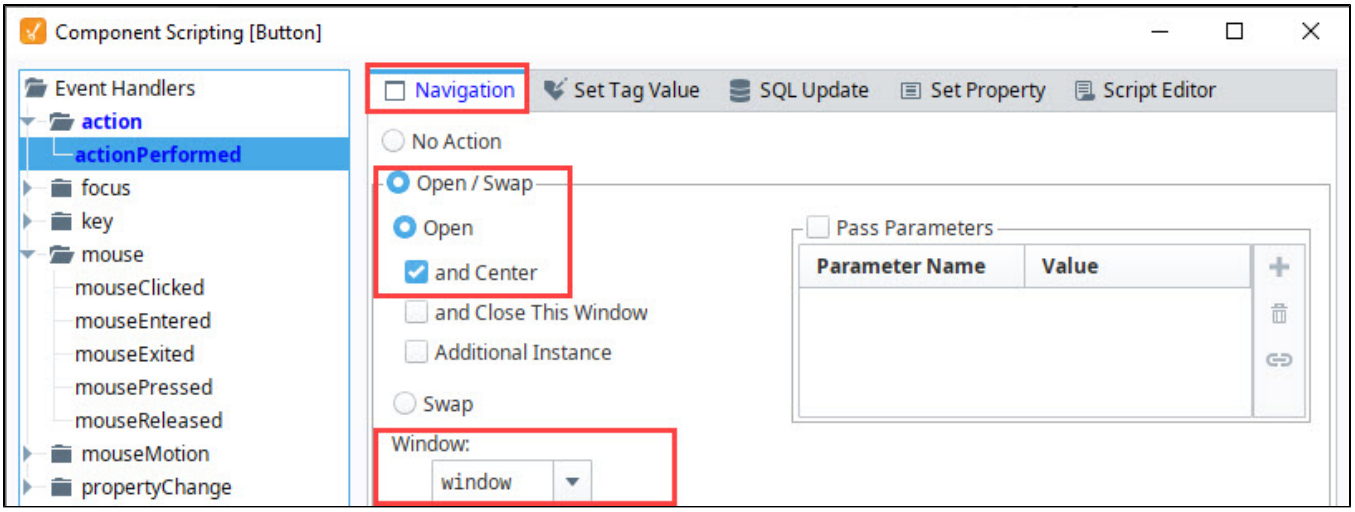
Note: Only one Script Builder can be used at a time. If you previously picked another action using a different Script Builder, it will get overwritten by enabling another Script Builder. If you need to do more than one action at a time, use the Script Editor to combine scripts, or create your own script.

Navigation Script Builder

The **Navigation** Script Builder has various functions that deal with opening and closing windows.

Open / Swap

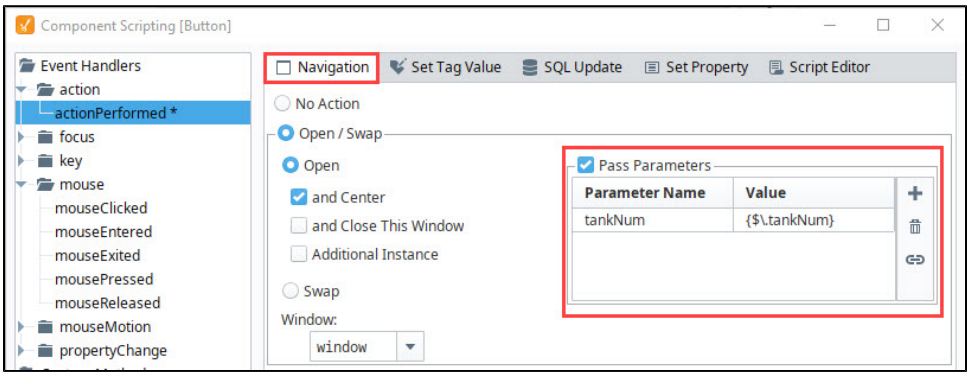
Opening is a very straight-forward operation, it simply opens the specified window at the same size it was in the Designer. Simply click on the **Open / Swap** button, and select a Window from the dropdown list that you want to open. There are options to center that window within the Client, and to close the window that the event was fired from. The opened window can also be opened as an additional instance, meaning there can be multiple copies of the same window. This is useful when opening dynamic popups, so that a couple of popups can be opened, each with different values.



Swapping is the practice of opening another window in the same size, location, and state as the current window, and closing the current window. This gives the appearance of one window simply swapping into another, seamlessly. The Navigation Builder uses the **swapWindow** version of swapping, but most "by hand" script authors will use the **swapTo** version. This last version relies on the fact that the windows being swapped are both maximized windows. See the navigation strategy section for more information.

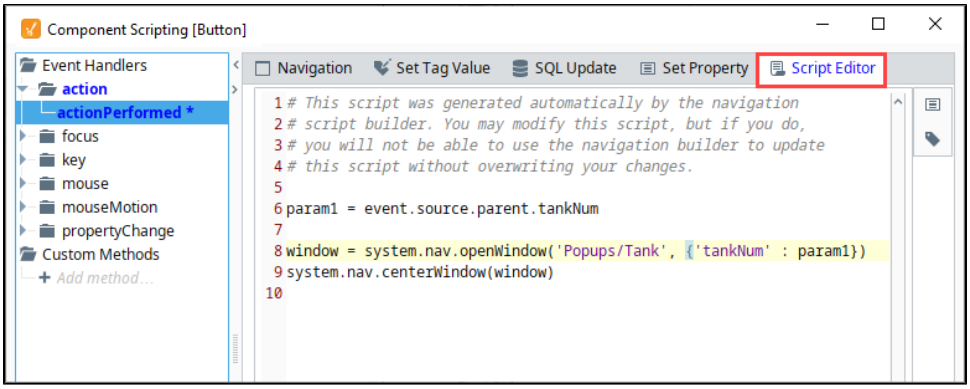
You can also **pass parameters** to the opened or swapped-to window. Check the **Pass Parameters** box, and click the **Add** icon to add a row, where each row is another passed parameter. The names of these parameters must match names of custom properties on the root container of the target window. The values can either be literals or values of other properties from the source window. Use the **Binding icon** to navigate to the properties that you want to pass. It will construct the path to the property on the window. To pass parameters in a navigation window, follow the steps below.

1. Check the **Pass Parameters** box, and click the **Add** icon to add a row.
2. Choose a custom property from the "Parameter Name" dropdown list. This will be filled with custom properties on the root container of the selected window. You can also type a name in directly.
3. Highlight the empty cell in the **Value** column of the parameter table, click the **Binding** icon , select the component property you want to enter the value and press **OK**.
4. Press **OK** to commit the change.



To learn more about passing parameters, refer to the parameterized windows section for more information.

The following Navigation builder script was generated by the Script Editor. If you compare the settings in the Navigation tab with the documented code in the Script Editor, you'll notice the Tank popup window will be opened and centered in the window, and the value of the "tankNum" parameter passed.




```
param1 = event.source.parent.tankNum


window = system.nav.openWindow('Popups/Tank', {'tankNum' : param1})
system.nav.centerWindow(window)
```

Forward / Back

The **Forward / Back** actions give you a simple way of implementing *browser*-style forward/back buttons in your client. You must be swapping between windows for this to work, because these functions rely on calls to **`system.nav.swapTo`** in order to keep track of what the sequence of recent windows has been.

Closing Windows

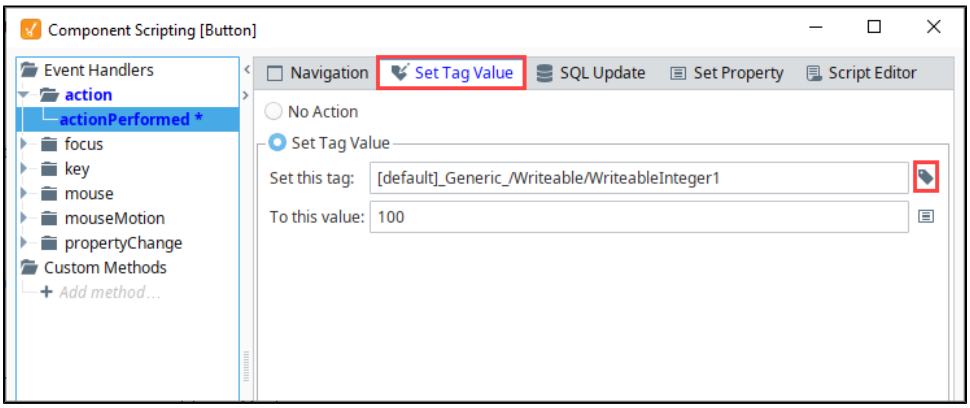
The Closing windows action allow for an easy way to have an event handler close the window that it is a part of, or any other window.

Set Tag Value Script Builder

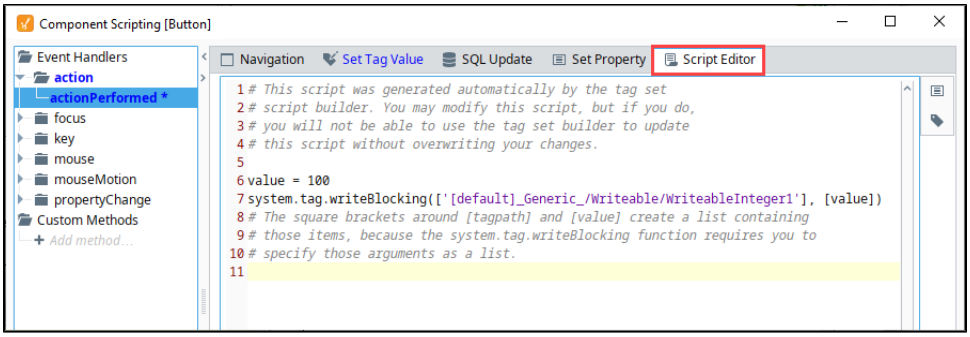
The Set Tag Value Script Builder responds to an event by setting the value of a Tag. You can set the Tag to either a literal value directly typed in, but we recommend using the chain link **Tag** icon to have the event handler use the value of another property from the same window.

Use the steps below to create a Set Tag Value.

1. Under the **Set Tag Value** tab, click the **Set Tag Value** radio button.
2. Click the **Tag** icon and choose a **Tag** from the Tag browser list to write to (i.e., WriteableInteger1).
3. In the **To this Value** field, enter a number (i.e., 100) or click the **Property** icon to to browse for a component property to use as the set-to value.
4. Press **OK** to commit the change.



The **Set Tag Value** script shown below was generated by the Script Editor. Compare the settings in the Set Tag Value tab with the documented code in the Script Editor and you'll see the Tag is set to "WriteableInteger1" and the Tag value is set to "100."

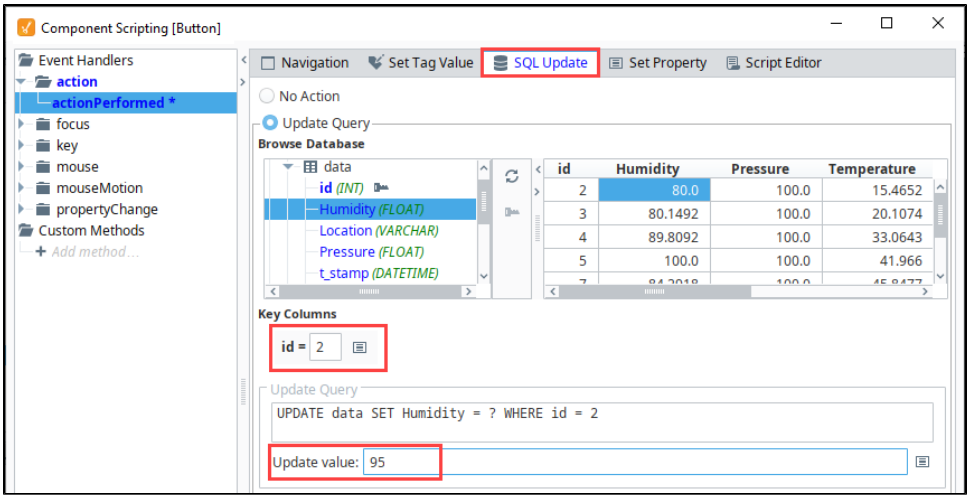



```
value = 100
system.tag.writeBlocking(['[default]_Generic_/Writeable/WriteableInteger1'], [value])

# The square brackets around [tagpath] and [value] create a list containing
# those items, because the system.tag.writeBlocking function requires you to
# specify those arguments as a list.
```

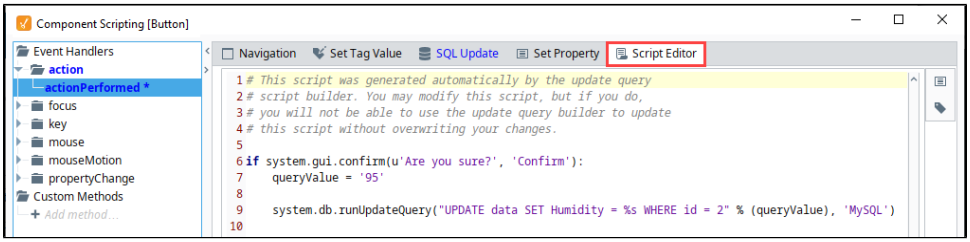
SQL Update Script Builder

The SQL Update Script Builder helps you build an update query using a database browsing interface. Choose a database and table in your target database, and the update query will be built for you. The key query will help identify a specific row to update, and can be made dynamic using Update Query and Update Value text boxes.

1. Under the SQL Update tab, click the **Update Query** radio button.
2. Select a database and choose a table in your database. Select an event value in the table on the right.
3. This example has the **id** as the Key Column, so we entered **'id =2.'**
4. We selected the **Humidity** for 'id 2'.
5. To change the value in the database, we entered the new value in the **Updated Value** field to replace the previous value when the action is executed.
6. Press **OK** to commit the change.



The following SQL Update script was generated by the Script Editor. If you compare the settings in the SQL Update tab with the documented code in the Script Editor, you'll see your database, table, and column name, including the row **id** of the searched value. You'll also see the new update value that will replace the existing value when the action is executed.

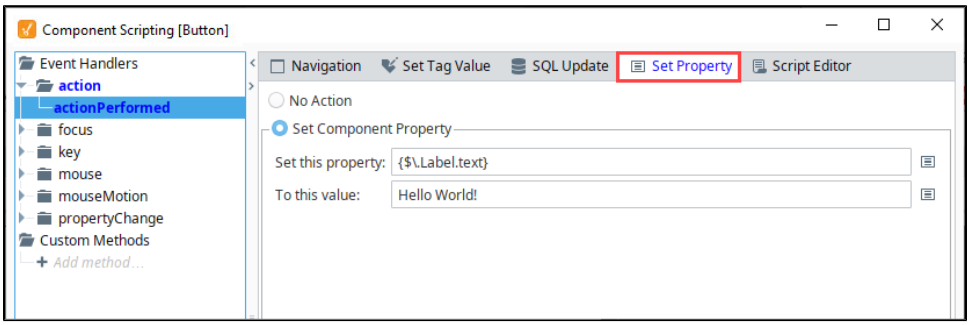


```
if system.gui.confirm(u'Are you sure?', 'Confirm'):
        queryValue = '95'

        system.db.runUpdateQuery("UPDATE data SET Humidity = %s WHERE id = 2" % (queryValue), 'MySQL')
```

Set Property Script Builder

The Set Property Script Builder will respond to an event by altering a property in the window. You must choose the property to alter and choose the value that you wish to assign to it. The new value can be a literal value or the value of any other property on the window.

1. Drag a Label component to your window to set the property to.
2. Select the **Button** component and open the Scripting window, and select **actionPerformed**.
3. In the Set Property tab click the **Set Component Property** radio button.
4. On the **Set this property** click on the **Property** icon to browse for the Label component's **Text** property, and click **OK**.
5. Type something into the **To this value:** field (i.e., Hello World!), or click the **Property** icon to browse for a component property to use as the set-to value.
6. Press **OK** to commit the change.



This Set Property script was generated by the Script Editor. Compare the settings in the Set Property tab with the documented code in the Script Editor and you'll see the value string "Hello World!" will be written to the Label component when the action executes.


```
value = u'Hello World!'
event.source.parent.getComponent('Label').text = value
```

Script Editor

The Script Editor allows you to add more complexity to existing scripts, combine scripts, and even write your own custom scripts. For example, if you need to perform two or more actions at once, (i.e., set a Tag and navigate to another window), you can update the script by combining the two actions in the Script Editor. The Script Editor even gives you the flexibility to create your own code for any action you want to perform on an event handler.

As seen elsewhere on this page, the other builders ultimately generate a script that can be modified from the Script Editor Builder.

Advanced Settings

- **Scoping Dropdown** - Allows you to specify the scoping of the script. The scoping of event handlers in older versions of Ignition work differently that modern versions. This settings was added as a way to provide backwards compatibility when upgrading. All new scripts should ideally leave the scope set to Standard Scoping, as there is no reason for new scripts to use the Legacy Scoping option.

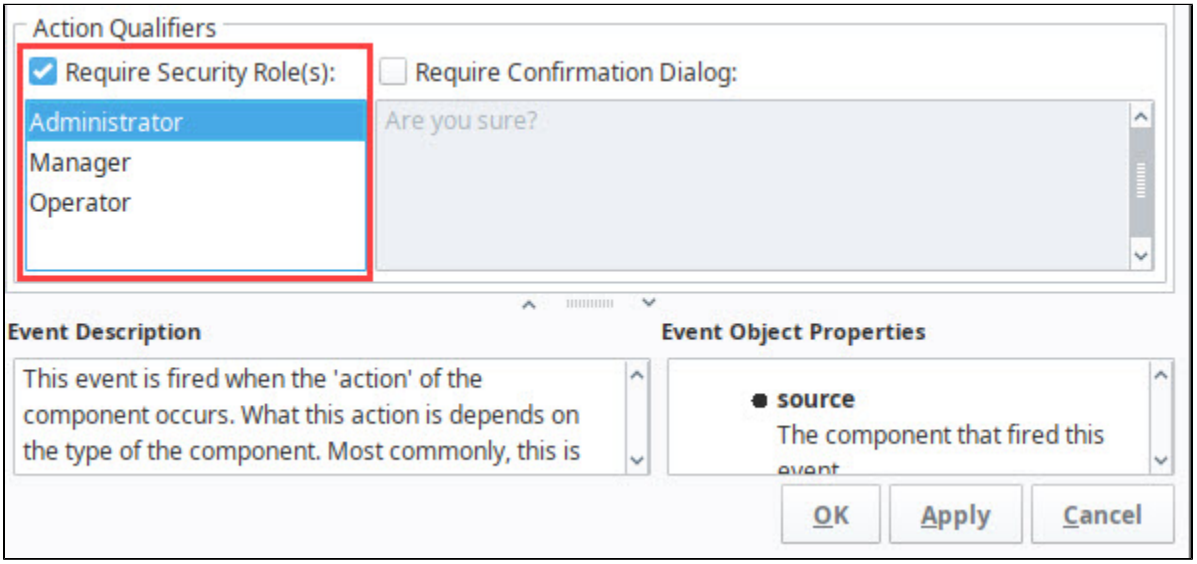
- **Invoke Later** - Provides an opportunity to allow the script to run after other events have finished processing. Most scripts will leave this setting disabled, however if can be useful in some scenarios:
    - Controlling the focus order in a window, since requesting focus from a component in the middle of an event being processed can cause undesirable results.
    - When writing a script on a window's visionWindowOpened event, you may wish to have your script run after processing.

Action Qualifiers

All of the Script Builders allow you to put Security and/or Confirmation qualifiers onto an event handler. These Action Qualifiers are optional.

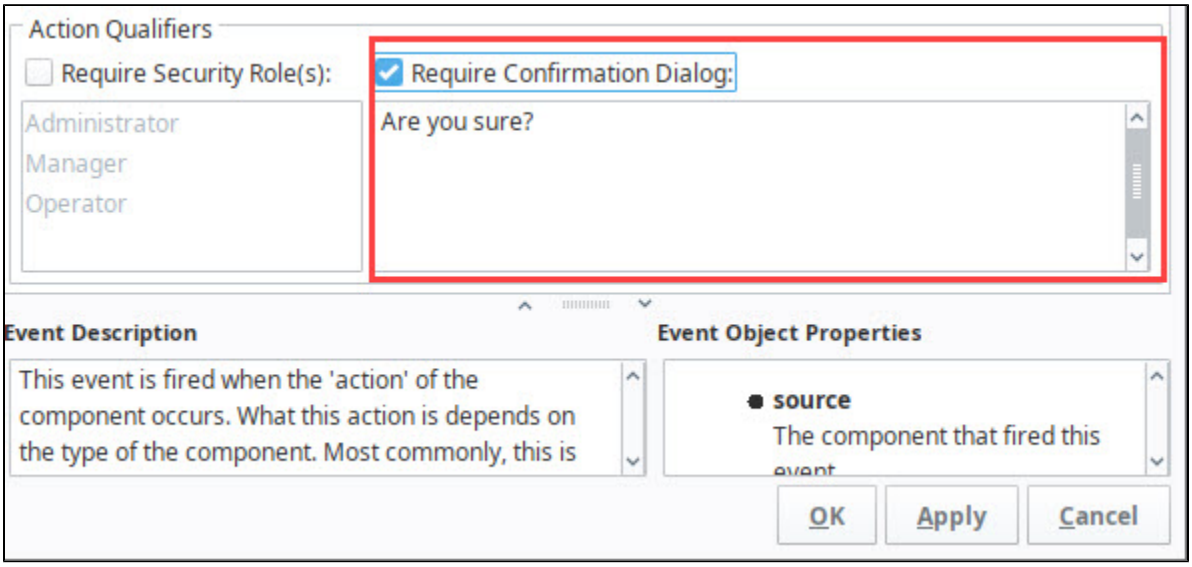
Security Qualifier

The Security Qualifier lets you restrict the event handler from running if the current user does not have one of the required roles highlighted in the Security qualifier dialog box. The roles listed will be all of the roles within the project's default user source. To set up the required roles for an event handler, select one or use **CTRL+click** to select multiple roles. Once the roles are selected, they will be highlighted. To deselect roles, use **CRTL+click** again. The action will only be executed if the **Required Roles** checkbox is enabled. Click **Close**.



Confirmation Qualifier

The Confirmation Qualifier prompts the user with a popup dialog box confirming you want to perform the action. The action will only be executed if the **Require Confirmation** checkbox is enabled. There is a default message, and if you prefer, you can delete it and enter your own message. Click **Close**.



Related Topics ...

- Parameterized Popup Windows
- DB Browse Bindings

Component Events

Event Handlers

When running a script on a component, we typically don't want it to be constantly running, but instead want the script to trigger when the user does something on screen such as clicks with the mouse. That something the user does is called an *Event* and can range from a mouse click or a keypress to a window opening or a component property change. When certain events happen, they trigger event handlers, which use a script to *handle* what happens when the event occurs.

This page lists out all of the event handlers that are on Ignition's Vision module. Any third party modules may add new components which may potentially have new event handlers.

Event Object

Every event handler contains an **event** object, which allows you to interact with the component and the entire window hierarchy within your script. While each **event** object has different properties depending on what event handler it resides in, each **event** object contains a **source** property, which is a reference to the component that fired the event. Using `event.source` not only gives us access to all of the properties available on that particular component, such as the text property of a text field,

Pseudocode - Event Handler Source Component Properties

```
# Here we start with the event object, then use source to go to the
component that fired the event,
# and then use the name of the property to access its value. In this case,
we accessed the text property.
text = event.source.text
```

but it also provides us with a way to navigate to other components within the hierarchy. For example, here we have a script on a button that references a text field.

Pseudocode - Event Handler Other Components

```
# Here again we start with event.source to get to the component that fired
the event, but now we use
# parent to go up to the root container, and then getComponent to navigate
back down to a different component.
text = event.source.parent.getComponent("Text Field")
```

On this page ...

- Event Handlers
- Action Event Handlers
- Property Event Handlers
- Mouse Event Handlers
- MouseMotion Event Handlers
- Key Event Handlers
- Focus Event Handlers
- VisionWindow Event Handlers
- InternalFrame Event Handlers
- Cell Event Handlers
- Item Event Handlers
- Paint Event Handlers

Note: Even when components are disabled, most scripting events can still occur. For example, a mouse click can still happen on a disabled component, which is why it is recommended to use the action performed event when placing a script on a button.

The **Action** category of event handlers pertains to components being "used" from the client, such as a button being pressed or a checkbox component being selected. You can access event handlers through the Scripting option.

Action Event Handlers

Events

Events	Description
actionPerformed	This event is fired when the 'action' of the component occurs. What this action is depends on the type of the component. Most commonly, this is used with buttons, where the action is that the button was pushed, via a mouse click or a key press. See the component reference for details on what the action means for other components. It is recommended to use this event over mouseClicked whenever possible.

Event Object Properties

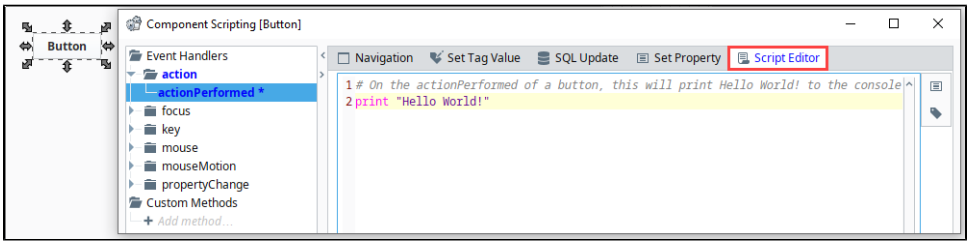
Properties	Description
source	The component that fired this event.

In this example, let's use a Button component to print "Hello World!' on the the console each time the Button is pressed.

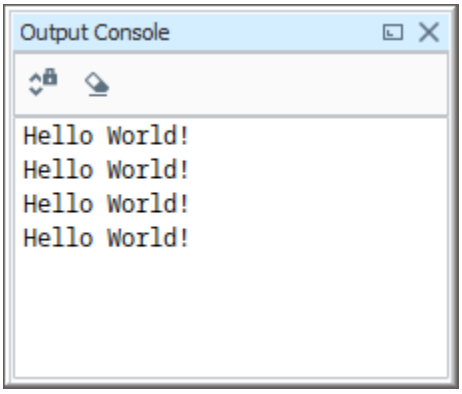
1. Drag a **Button** component to the your Designer workspace.
2. Select your Button and right-click on **Scripting**.
3. Copy the code in the code block and paste it in the Script Editor tab. Press **OK**.

Python - Button Action Performed

```
print "Hello World!"
```



4. **Save** your project, and put the Designer in **Preview Mode**.
5. From the top menubar click on **Tools > Console**. Press the button and you'll see "Hello World" printed each time the button is pressed.



Property Event Handlers

Property event handlers typically trigger based on the property of a component.

Events

Events	Description
propertyChange	Fires whenever a bindable property of the source component changes. This works for standard and custom (dynamic) properties.

Event Object Properties

Properties	Description
source	The component that fired this event.
newValue	The new value that this property changed to.
oldValue	The value that this property was before it changed. Note that not all components include an accurate oldValue in their events.
propertyName	The name of the property that changed.

	Note: Remember to always filter out these events for the property that you are looking for! Components often have many properties that change,

Python - Printing the Changing Property

```
# On the propertyChange of a component, this script will print out the name of the property that is changing.
print event.propertyName
```

Python - Looking for a Specific Property

```
# It is common to use propertyName to look for specific properties to change. This is a great way to
restrict how often your scripts execute
if event.propertyName == 'text':
  print 'The Text property changed'
```

Mouse Event Handlers

The mouse events all correspond to the clicking and movement of the mouse. They are triggered in the client by an operator interacting with a mouse. Touchscreen monitors will trigger these events when a user touches the screen, but not all touchscreens will fire the mouseEntered and mouseExited events.

Events

Events	Description
mouseClicked	This event signifies a mouse click on the source component. A mouse click is the combination of a mouse press and a mouse release, both of which must have occurred over the source component. **Note:** This event fires after the pressed and released events have fired.
mouseEntered	This event fires when the mouse enters the space over the source component.
mouseExited	This event fires when the mouse leaves the space over the source component.
mousePressed	This event fires when the mouse presses down on the source component.
mouseReleased	This event fires when a mouse button is released, if that mouse button's press happened over this component.

Event Object Properties

Properties	Description
source	The component that fired this event.
button	The code for the mouse button that caused this event to fire. The following is a list of constants that can be used in the event to determine which mouse button was pressed. • `event.BUTTON1` (typically the primary, or left, mouse button) • `event.BUTTON2` (typically the middle mouse button) • `event.BUTTON3` (typically the secondary, or right, mouse button).
clickCount	The number of mouse clicks associated with this event.
x	The x-coordinate (with respect to the source component) of this mouse event.
y	The y-coordinate (with respect to the source component) of this mouse event.

popupTrigger	Returns True (1) if this mouse event is a popup trigger. What constitutes a popup trigger is operating system dependent, which is why this abstraction exists.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise.

Python - Printing on a Mouse Enter

```
# On the mouseEntered event of a component, this script will only fire if the mouse enters the bounds of the
component.
print "The mouse is inside the component space!"
```

MouseMotion Event Handlers

The mouseMotion events deal with the motion of the mouse over a component. Not all touchscreen monitors will fire these events.

Caution:

mouseMotion events will not trigger when the project is viewed from a mobile project as these gestures are used by the browser/device to zoom or pan.

Events

Events	Description
mouseDragged	Fires when the mouse moves over a component while a mouse button is being held.
mouseMoved	Fires when the mouse moves over a component, but no buttons are being held.

Event Object Properties

Properties	Descriptions
source	The component that fired this event.
button	The code for the mouse button that caused this event to fire. The following is a list of constants that can be used in the event to determine which mouse button was pressed. • `event.BUTTON1` (typically the primary, or left, mouse button) • `event.BUTTON2` (typically the middle mouse button) • `event.BUTTON3` (typically the secondary, or right, mouse button).
clickCount	The number of mouse clicks associated with this event.
x	The x-coordinate (with respect to the source component) of this mouse event.
y	The y-coordinate (with respect to the source component) of this mouse event.
popupTrigger	Returns True (1) if this mouse event is a popup trigger. What constitutes a popup trigger is operating system dependent, which is why this abstraction exists.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise

Python - Printing on a Mouse Movement

```
# From the mouseMotion event on a component, this will print each time the mouse moves when it is over the
component.
print "The mouse is moving over the component!"
```

Key Event Handlers

The key events all have to do with the user pressing a key on the keyboard.

Events

Events	Description
keyPressed	Fires when a key is pressed and the source component has the input focus. Works for all characters, including non-printable ones, such as SHIFT and F3.
keyReleased	Fires when a key is released and the source component has the input focus. Works for all characters, including non-printable ones, such as SHIFT and F3.
keyTyped	Fires when a key is pressed and then released when source component has the input focus. Only works for characters that can be printed on the screen.

Event Object Properties

Properties	Description
source	The component that fired this event.
keyCode	The key code for this event. Used with the keyPressed and keyReleased events. Uses the standard Java key codes, see below for more information.
keyChar	The character that was typed. Used with the keyTyped event.
keyLocation	Returns the location of the key that originated this key event. Some keys occur more than once on a keyboard, e.g. the left and right shift keys. Additionally, some keys occur on the numeric keypad. This provides a way of distinguishing such keys. The keyTyped event always has a location of KEY_LOCATION_UNKNOWN. Uses the standard Java key locations, see below for more information.
altDown	True (1) if the Alt key was held down during this event, false (0) otherwise.
controlDown	True (1) if the Control key was held down during this event, false (0) otherwise.
shiftDown	True (1) if the Shift key was held down during this event, false (0) otherwise.

Python - Printing the Key Released

```
# On the keyReleased event of a component, this will print out the key code of the key that was hit on the keyboard,
# but only on release of the key, and only when the component has focus.
print event.keyCode
```

Java Keys

The key event handlers use the Java KeyEvent class, which has unique identifiers for both keys and locations on the keyboard to help differentiate which key is actually being pressed on the keyboard. The numeric codes for each unique location and character can be called from the event object using a constant. For example, the letter "a" has the constant name VK_A. This can then be used to compare against the keyCode value like this:

Python - Checking Specific Key Codes

```
if event.keyCode == event.VK_A:
        print "The key press was a"
```

We listed the locations and some common codes below, but the full list of codes can be accessed by going to https://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyEvent.html.

Note: Some Operating Systems reserve certain keys for certain function, and will capture the key press or release before it gets sent to the Client. For example, many Operating Systems use the TAB key to shift focus to the next field

Key Code Constants				
VK_0 - VK_9	VK_END	VK_PAGE_UP	VK_DOWN	VK_CONTROL
VK_A - VK_Z	VK_ENTER	VK_RIGHT	VK_PAGE_DOWN	VK_LEFT
VK_F1 - VK_F24	VK_HOME	VK_SHIFT	VK_UP	VK_TAB
VK_ALT	VK_INSERT	VK_SPACE	VK_ESCAPE	

Key Location Constants		
KEY_LOCATION_LEFT	KEY_LOCATION_RIGHT	KEY_LOCATION_NUMPAD
KEY_LOCATION_STANDARD	KEY_LOCATION_UNKNOWN	

Focus Event Handlers

Focus events deal with focus moving between different components on a window. Opening windows, using tab to move around the screen, or clicking on components will trigger these events. Note that not all components can hold focus.

Events

Events	Description
focusGained	This event occurs when a component that can receive input, such as a text box, receives the input focus. This usually occurs when a user clicks on the component or tabs over to it.
focusLost	This event occurs when a component that had the input focus lost it to another component.

Event Object Properties

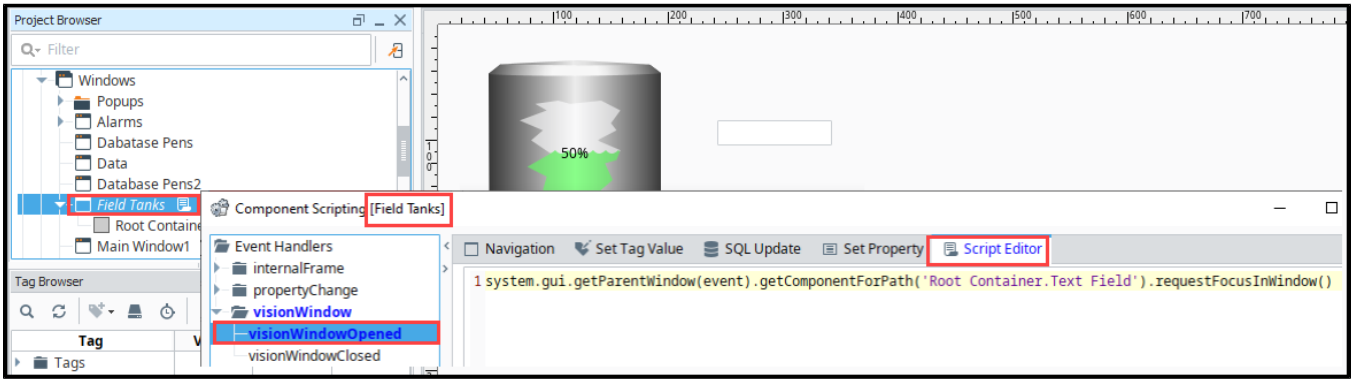
Properties	Description
source	The component that fired this event.
oppositeComponent	The other component involved in this focus change. That is, the component that lost focus in order for this one to gain it, or vise versa.

Python - Printing on Focus Gained

```
# On the focusGained event of a few different components, this script can print out when the component has gained focus.
print "The component name now has focus!"
```

VisionWindow Event Handlers

The visionWindow events are specific to windows and not available elsewhere. Right click on the window name in the Project Browser and select Scripting to get access to these events. These events are triggered by a window opening or closing.



Events

Events	Description
visionWindowOpened	This event is fired each time the window is opened and before any bindings are evaluated.
visionWindowClosed	This event is fired each time the window is closed.

Event Object Properties

Properties	Description
source	The vision window that fired this event.

Python - Grabbing Focus on Window Opened in Two Different Ways

```
# From a visionWindowOpened event on a window, you can request the focus of components in the window, to
start the focus on a component other than the upper left most component.

# Here we grab the reference to the component using the property selector on the upper right side of the
script editor.
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text Field').requestFocusInWindow()

# Here we can manually enter in the path to the component using our knowledge of the component hierarchy and
# the getRootContainer function. Both of these functions work in the same way.
system.gui.getParentWindow(event).getRootContainer().getComponent("Text Field").requestFocusInWindow()
```

InternalFrame Event Handlers

The internalFrame events are fired by windows: windows are known as "internal frames" in the underlying Java windowing system that the Vision component uses. Note that the source of these events is the window itself, just like the visionWindow events above.

Events

Events	Descriptions
internalFrameActivated	Fires whenever the window is shown or focused. If you want a script to fire every time a window is opened, use this event.
internalFrameClosed	Fires when a window is closed.
internalFrameClosing	Fires right before a window is closed.
internalFrameDeactivated	Fires when a window loses focus.
internalFrameOpened	Fires the first time a window is opened. Note that when windows are closed and cached, next time they are opened this event will not be fired. Use internalFrameActivated instead.

Event Object Properties

Properties	Description
source	The window that fired this event. Use source.rootContainer to get the root container.

Python - Printing on Frame Activation

```
# From the internalFrameActivated event on a window, this will fire each time the window is focused, so
clicking between two different windows will trigger it.
print "This window is now in focus!"
```

Cell Event Handlers

The cell event is unique in that it only appears on the Table component. It will trigger when something within a cell changes, and once for each cell changed.

Events

Events	Description
cellEdited	This event is fired when one of the cells in a table component has been modified.

Event Object Properties

Properties	Description
source	The table component that fired this event.
oldValue	The old value in the cell that changed.
newValue	The new value in the cell that changed.
row	The row of the dataset this cell represents.
column	The column of the dataset this cell represents.

Pseudocode - Updating a Database Table

```
# From the cellEdited event of a table component, this script can update our database table with any new
data that is entered in the table

# Get the id of the row we edited and the headers
id = event.source.data.getValueAt(event.row, 'id')
headers = system.dataset.getColumnHeaders(event.source.data)

# Build our Update query.
query = "UPDATE User SET %s = ? WHERE id = ?" % (headers[event.column])
args = [event.newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)
```

Item Event Handlers

The item event is unique in that it only appears on components that can be "on" or "off", such as with Radio Buttons, Check Boxes, and Toggle Buttons.

Events

Events	Description
itemStateChanged	This event fires when the state of the component changed. This event is the best way to respond to the state of that component changing.

Event Object Properties

Properties	Description
source	The component that fired this event.
stateChange	An integer that indicates whether the state was changed to "Selected" (on) or "Deselected" (off). Compare this to the event object's constants to determine what the new state is.
SELECTED	The constant that the stateChange property will be equal to if this event represents a selection.
DESELECTED	The constant that the stateChange property will be equal to if this event represents a de-selection.

Python - Printing on a Radio Button Selection

```
# On the itemStateChanged event of a radio button, this will print when this specific radio button is
selected.
if event.stateChange == event.SELECTED:
        print "This radio button is selected!"
```

Paint Event Handlers

The paint event is only found on the Paintable Canvas component, and is used to customize how the component gets painted. This event requires a heavy knowledge of programming using the Java 2D drawing tools, but there is code for a pump shape each time you add a new Paintable Canvas to a window.

Events

Events	Description
repaint	This event will fire whenever the component needs to repaint itself. It will repaint when any of its custom properties change, or when `.repaint()` is called on it. When a Paintable Canvas is first dragged onto the screen, the repaint event handler will be filled with an example that draws out a pump.

Event Object Properties

Properties	Description
source	The Paintable Canvas component that fired this event.
graphics	An instance of java.awt.Graphics2D that can be used to paint this component. The point (0,0) is located at the upper left of the component.
width	The width of the paintable area of the component. This takes into account the component's border.
height	The height of the paintable area of the component. This takes into account the component's border.

Python - Painting a Circle

```
# On the repaint event of a paintable canvas component, this will create a circle with a gradient background
color of orange and white.

from java.awt import Color
from java.awt import GradientPaint
from java.awt.geom import Ellipse2D

g = event.graphics

# Body
innerBody = Ellipse2D.Float(8,8,72,72)

#### Scale graphics to actual component size
dX = (event.width-1)/100.0
dY = (event.height-1)/100.0
g.scale(dX,dY)

# Paint body
g.setPaint(GradientPaint(0,40,Color.WHITE, 0,100, Color.ORANGE,1))
g.fill(innerBody)
g.setColor(Color.ORANGE)
g.draw(innerBody)
```

Extension Functions

What Are Extension Functions

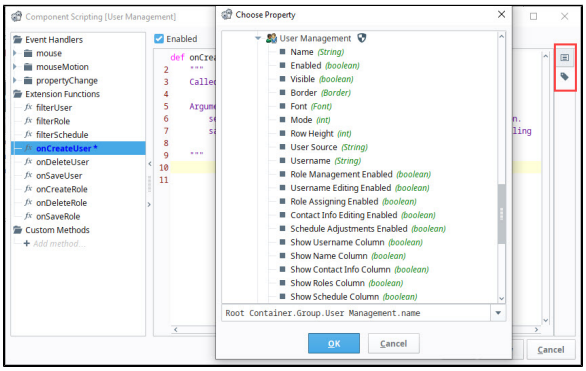
Extension Functions are found on the component scripting window of certain components, and they allow for more advanced customization of the component using scripting. These functions are generally more advanced and require a better understanding of Python. Unlike Event Handlers, Extension Functions are not driven by a specific event, but are instead called by the component itself for a specific purpose when appropriate. This may be when the component first loads in the window, or whenever the function receives new input.

From an object-oriented point of view, Extension Functions create a custom "subclass" of the base component type. Your subclass can then override and implement parts of the functionality of the component itself, in Python. Following Python object-oriented methodology, each extension function's first argument is called **`self`** . That is because these are methods that belong to the component's class itself, instance methods. The value of **`self`** will always be the component itself. Notice that this is different than Event Handler scripts where you are given an **`event`** object in your scope and the component that fired the event is under **`event.source`** . When you write an Extension Function, there is no **`event`** object so the component is given to you as the **`self`** object instead.

Each component Extension Function comes with its own documentation built-into the function's default implementation using a standard Python "doc-string". You will find that you are unable to edit the function's signature or docstring. Changing the method's signature (arguments or function name) would prevent the component from calling it correctly. Changing the docstring could be misleading or confusing as you'd lose the documentation for how your implementation of the function should work.

The following feature is new in Ignition version **8.1.13**
Click here to check out the other new features

The Extension Function Script Builder now includes **Insert Tag** and **Insert Property Reference** helper buttons that allow you to easily insert correctly formatted references within your scripts.



On this page ...

- What Are Extension Functions
- Using Extension Functions
  - Example - User Management Component
  - Example - Table Component
  - Example - Power Table Component
  - Example - Ad Hoc Charting

Component Extension Functions

Watch the Video

Using Extension Functions

Using an Extension Function works much like using an Event Handler. First select and **Enable** the Extension Function within the component scripting window, and then add in a script. The script will then automatically run when called.

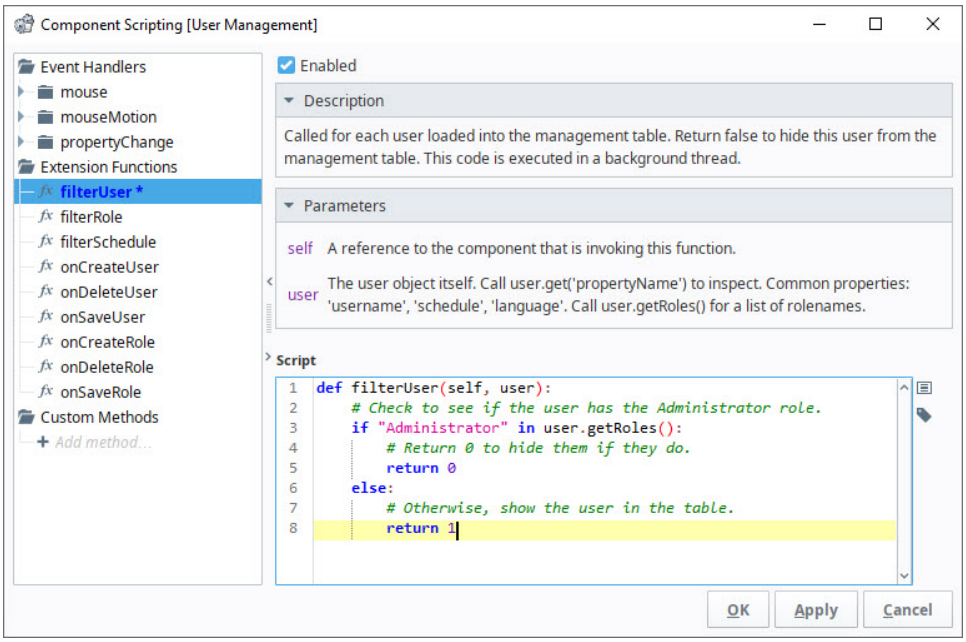
Note: When indenting in an Extension Function, you **must** use tabs for indentation. Extension functions are "pre-written" using tab indentation, so any lines added must also use tab indentation.

Example - User Management Component

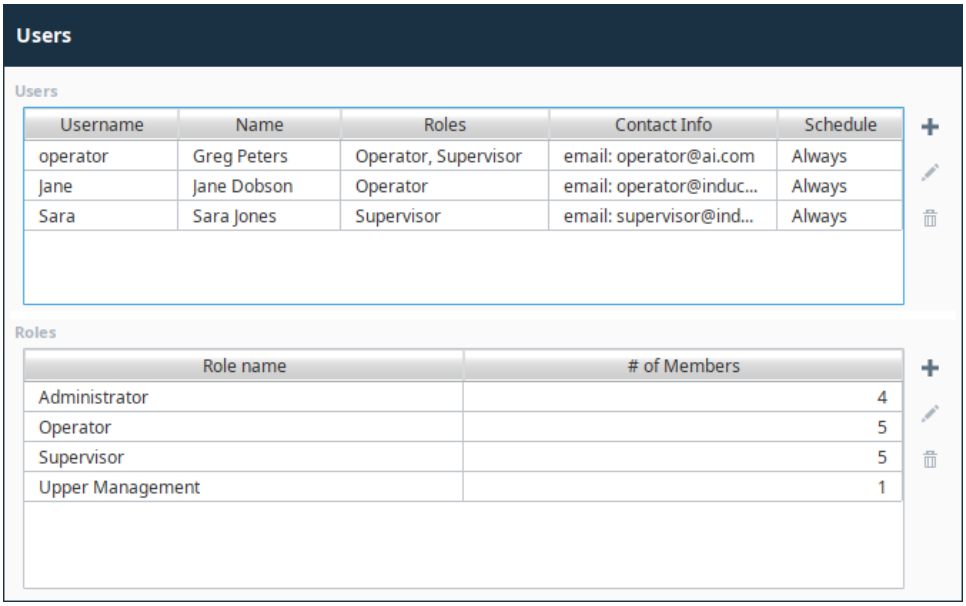
The User Management component has many Extension Functions that provide a way to customize how the component works. The **`filterUser()`** e xtension function is useful for filtering out users you don't want to see in the user source, preventing users from editing that user in the client. We can add a simple script to the **`filterUser()`** Extension Function that will hide the user from the list if they have the Administrator role.

1. Drag a **User Management** component in to your Designer workspace, and right-click on **Scripting**.
2. Under the Extension Functions folder select **filterUser** and click **Enabled.**
3. Copy the code from the code block below and add it to bottom of the script and click **OK**.

```
# Check to see if the user has the Administrator role.
if "Administrator" in user.getRoles():
        # Return 0 to hide them if they do.
        return 0
else:
        # Otherwise, show the user in the table.
        return 1
```



4. By enabling this script, we now only see the users without the Administrator role in the list of users.



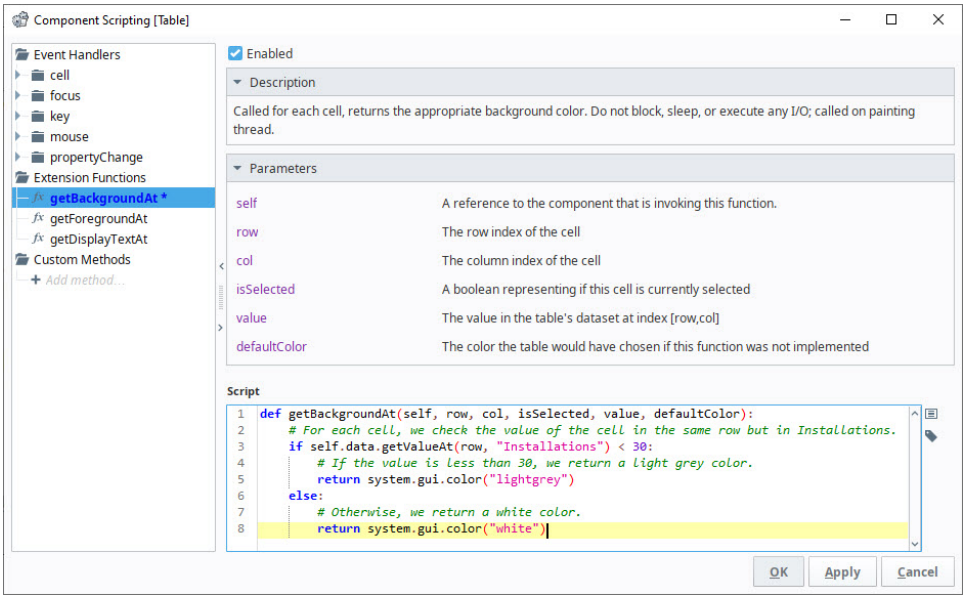
Example - Table Component

The Table component exposes an Extension Function called **`getBackgroundAt()`** . By implementing this function, you can control the background color of each cell of the table component using scripting. Starting with a Table component with some test data, then then add the following script to the **`getBackgroundAt()`** Extension Function.

Here is our sample data. We want to have a clearer indication of what companies have fewer than 30 installations. So, we'll write a script to make those rows have a light grey background color.

Installations	id	cityName	State	companyName
34	1	Sacramento	California	ABC Company
62	2	Phoenix	Arizona	Acme Bird Seed
14	3	Denver	Colorado	High Rocks Enterprises
87	4	Omaha	Nebraska	Corners Inc.
13	5	San Francisco	California	Haight Jewelry
99	6	San Antonio	Texas	Ten Gallon Foods
15	7	Chico	California	XYZ Brewing
74	8	Portland	Oregon	City Book
23	9	Iowa City	Iowa	Best Syrups Inc
15	10	San Rafael	California	Retro Redwoods

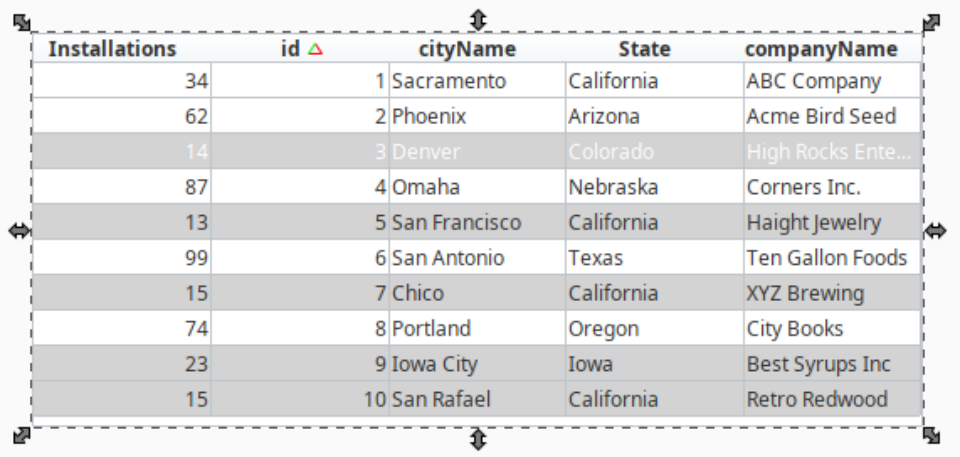
1. Right-click on the **Table** component and choose **Scripting**.
2. Choose the **`getBackgroundAt*`** extension function.
3. Select the **Enabled** check box.
4. Copy the code from the code block below and add it to bottom of the script and click **OK**.



Python - Table Row Color

```
# For each cell, we check the value of the cell in the same row but in Installations.
if self.data.getValueAt(row, "Installations") < 30:
	# If the value is less than 30, we return a light grey color.
	return system.gui.color("lightgrey")
else:
	# Otherwise, we return a white color.
	return system.gui.color("white")
```

5. We should see the script run automatically, and the background color of the table will change.

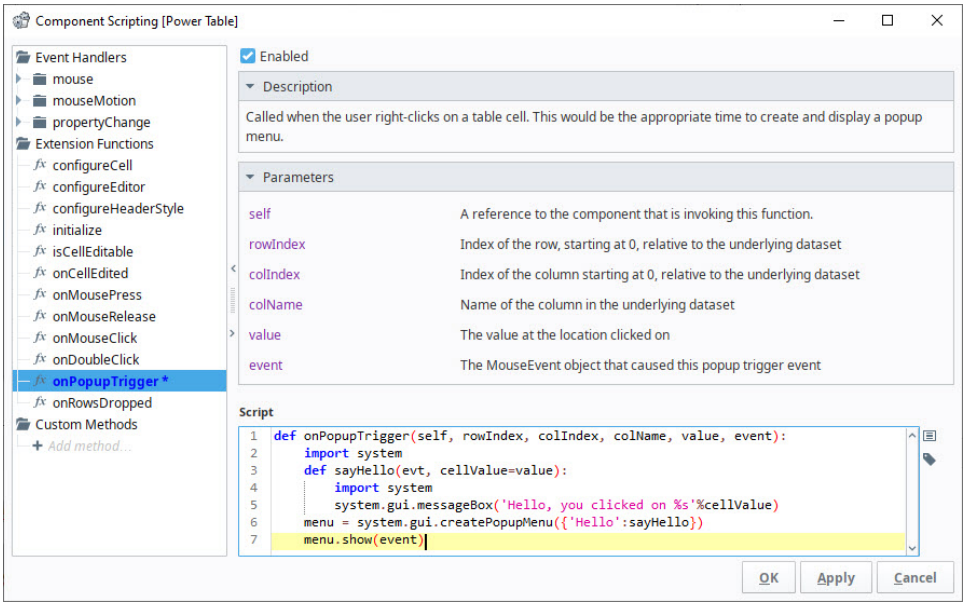


Installations	id	cityName	State	companyName
34	1	Sacramento	California	ABC Company
62	2	Phoenix	Arizona	Acme Bird Seed
14	3	Denver	Colorado	High Rocks Ente...
87	4	Omaha	Nebraska	Corners Inc.
13	5	San Francisco	California	Haight Jewelry
99	6	San Antonio	Texas	Ten Gallon Foods
15	7	Chico	California	XYZ Brewing
74	8	Portland	Oregon	City Books
23	9	Iowa City	Iowa	Best Syrups Inc
15	10	San Rafael	California	Retro Redwood

Example - Power Table Component

The Power Table component has several extension functions on it that change the way the table looks or behaves. One in particular, called **`onPopupTrigger()`**, makes it easy to implement a right-click popup menu as it is called each time a user right-clicks on a cell of the table. It can be used in conjunction with the **`system.gui.createPopupMenu`** function to create your own custom popup menu, as shown in the following example.

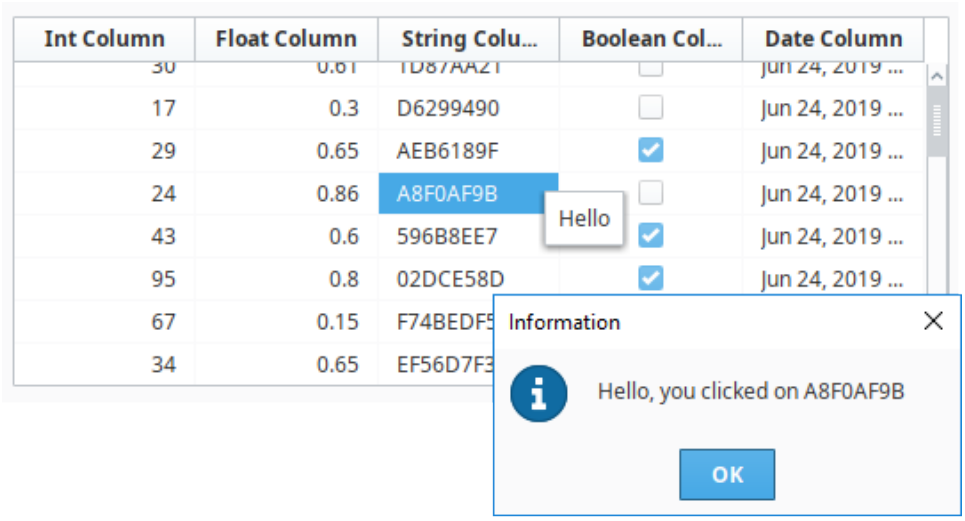
1. Drag a Power Table component on to your Designer workspace and set the **Test Data** property to **'true'** so you have some data to test on.
2. With the **Power Table** selected, click **Scripting**, and then click on the **onPopupTrigger** extension function.
3. Select the **Enabled** checkbox.
4. The **onPopupTrigger** extension function will have a pre-built example commented out in the extension function. Uncomment the lines of code to see it in action.



Python - Power Table Popup Trigger

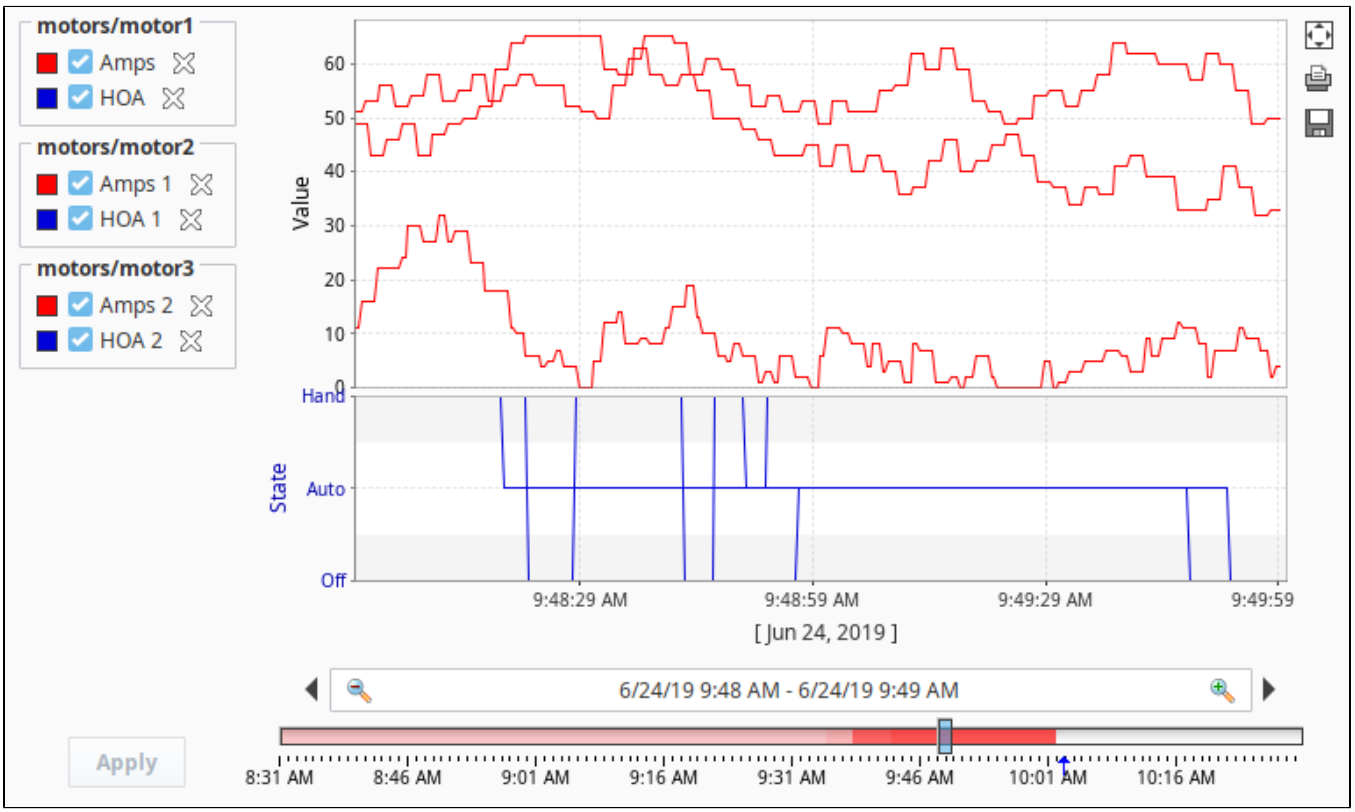
```
import system
def sayHello(evt, cellValue=value):
        import system
        system.gui.messageBox('Hello, you clicked on %s'%cellValue)
menu = system.gui.createPopupMenu({'Hello':sayHello})
menu.show(event)
```

5. Right-click on a cell in the table and a **"Hello"** menu option will have a message box appear that has the value of the cell that was right-clicked.



Example - Ad Hoc Charting

The Easy Chart component has an Extension Function to allow scripting when a Tag is dropped onto it (see Using the Tag Browse Tree for Charting). There is a lot of customization possible in the Designer, but any client side changes to the Tag Pens dataset must be done here. Generally, people want to change what axis and subplot a pen goes into based on some other information. Below is a simple example that uses the Tag's name to determine this. For this example to work, you need to have two subplots and a second axis named "HOA".



Python - Drop Tags on Easy Chart

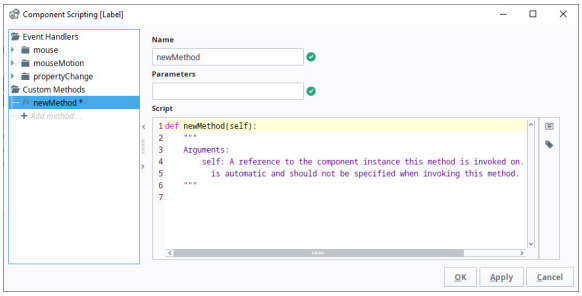
```
# Alter chart configuration when dropping pens
# sample data for the Tag Pens property:
#"NAME","TAG_PATH","AGGREGATION_MODE","AXIS","SUBPLOT","ENABLED","COLOR","DASH_PATTERN","RENDER_STYLE","
```

```
LINE_WEIGHT","SHAPE","FILL_SHAPE","LABELS","GROUP_NAME","DIGITAL","OVERRIDE_AUTOCOLOR","HIDDEN","
USER_SELECTABLE","SORT_ORDER","USER_REMOVABLE"
#"HOA","[~]Motors/Motor 1/HOA","MinMax","HOA","2","true","color(85,255,85,255)","","1","1.0","0","true","
false","","false","false","false","true",,"true"

# get old pen data and append new info
oldData = system.dataset.toPyDataSet(self.tagPens)
# get new info
for fullTagPath in paths:
        # get names for everything in the tag path
        lastSlashIndex = fullTagPath.rfind("/")
        closeBracketIndex = fullTagPath.find("]")
        tagName = fullTagPath[lastSlashIndex+1:]
        tagPath = fullTagPath[closeBracketIndex+1:]
        groupName = fullTagPath[closeBracketIndex+1:lastSlashIndex]

        # find which tags are named "hoa" and put them in the HOA subplot.
        if tagName.lower() == "hoa":
                axis = "HOA"
                subplot = 2
                color = "color(255,85,85,255)" #red
                digital = "true"
        else:
                axis = "Default Axis"
                subplot = 1
                color = "color(85,85,255,255)" #blue
                digital = "false"

        # append to the old pen data
        newData = system.dataset.addRow(oldData, [tagName,tagPath,"MinMax",axis,subplot,"true",color,"","1","
1.0","0","true","false",groupName,digital,"false","false","true","","true"])

# push new pens back to the tagPens property
self.tagPens = newData
```

Custom Component Methods

Custom Methods

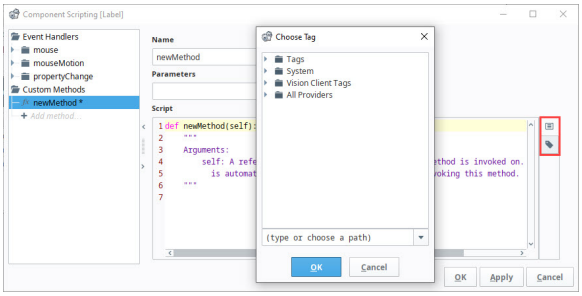
Custom Methods function much like Project Library in that you write a script and call it from somewhere else. However, Custom Methods are written on a component instead of a separate scripting section, and are also automatically passed the value of `self`, just like an Extension Function. The `self` object provides the script with easy access to everything within the window. In addition to `self`, Parameters can be added that you can use to pass in other objects into your Custom Method.



Note: When indenting in a Custom Component Method, you **must** use tabs for indentation. Custom Methods are "pre-written" using tab indentation, so any lines added must also use tab indentation.

The following feature is new in Ignition version **8.1.13**
Click here to check out the other new features

The Custom Method Script Builder now includes **Insert Tag** and **Insert Property Reference** helper buttons that allow you to easily insert correctly formatted references within your scripts.



Custom Methods can then be called from the same component or from other components. Custom Methods are called just like any other method on a component: `.methodName()`. So if I had a custom method on a text field, and I wanted to call it from the actionPerformed event on a button in the same container, I would use:

Python - Custom Method

```
# The name of the custom method in this instance is myCustomMethod.
event.source.parent.getComponent("Text Field").myCustomMethod()
```

Templates are another good use for custom methods. By adding a custom method directly to a template, all the components that make up the template can call the custom method from the template itself. Another advantage of templates is, in the event you want to share your template with another project, your custom methods (scripts) would not have to be exported separately like they would as if they were in a Script Library. When you export the template, the custom methods are included in the export automatically.

Sample Custom Methods

A great use for Custom Methods is checking for valid input on a form with a lot of text fields. Instead of checking every text field within a script on a button press, we can instead build a value check script on each text field that is unique to that Text Field's specific type of input. This keeps each script organized on the appropriate Text Field component. Take this sample code that can go into a custom method on a text field which checks for a valid email address using :

On this page ...

- Custom Methods
  - Sample Custom Methods
  - Custom Methods with Parameters

Python - Text Field Input Validation

```
# First, we need to import the Regular Expression library
import re

# We need to grab the value of the text field.
text = self.text

# Here, we put together our regular expression for email addresses.
validAddress = re.compile(r"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$)")

# Check the text against the regular expression.
# If the text does not fit in with the regular expression, it returns a value of None, which we use to show
an error box.
if validAddress.match(text) is None:
        system.gui.errorBox("Please enter a vaild email address!", "Email Not Valid")
```

A similar script can be repeated for each Text Field component, but modified to fit the expected input. Then, the another script can be on a button that simply calls each of the Text Field's Custom Methods.

Python - Validate Input on Button Press

```
event.source.parent.getComponent("Email Field").validInputCheck()
```

Custom Methods with Parameters

It is also possible to add parameters to Custom Methods. For example, we can configure a Custom Method like this one on a Button component that accepts two parameters (`left` and `right`). If a value for `right` is not provided, it will assume a default value of 7:

Parameter with a Default Value

```
def greaterThan(self, left, right=7):
        self.getSibling("ToggleSwitch").props.selected = left > right
```

We can then configure a script on the same Button that invokes the `greaterThan` Custom Method and performs an action. In the following examples, the `left` and `right` parameters are defined by the values of two Numeric Entry Field components on the same View as our Button.

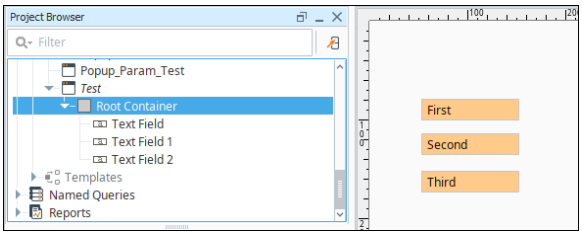
Examples with Multiple Parameters

```
def runAction(self, event):
        # The following usage omits the `right` kwarg, so `right` will equal the default value of 7
        # If the value of the `left` Numeric Entry field is greater than 7, the script will perform an action
        self.greaterThan(left=self.getSibling("NumericEntryField").props.value)

        # The following usage supplies both kwargs
        self.greaterThan(left=self.getSibling("NumericEntryField").props.value, right=self.getSibling
("NumericEntryField_0").props.value)

        # The following usage supplies positional args
        self.greaterThan(self.getSibling("NumericEntryField").props.value, self.getSibling
("NumericEntryField_0").props.value)

        # The following usage supplies only one positional arg, which will be used for `left`
        # Since only one positional arg was provided, `right` will equal the default value of 7
        self.greaterThan(self.getSibling("NumericEntryField").props.value)
```

Focus Manipulation

Focus Order

How components are laid out on a window determines the focus order. When tabbing through components on a window, the focus moves from left to right, then top to bottom. Focus will then cycle back to the first component. When determining order, the top left corner of the component is used. To see for yourself, drag several components into your window. Go to **Preview Mode**, and tab through the components.

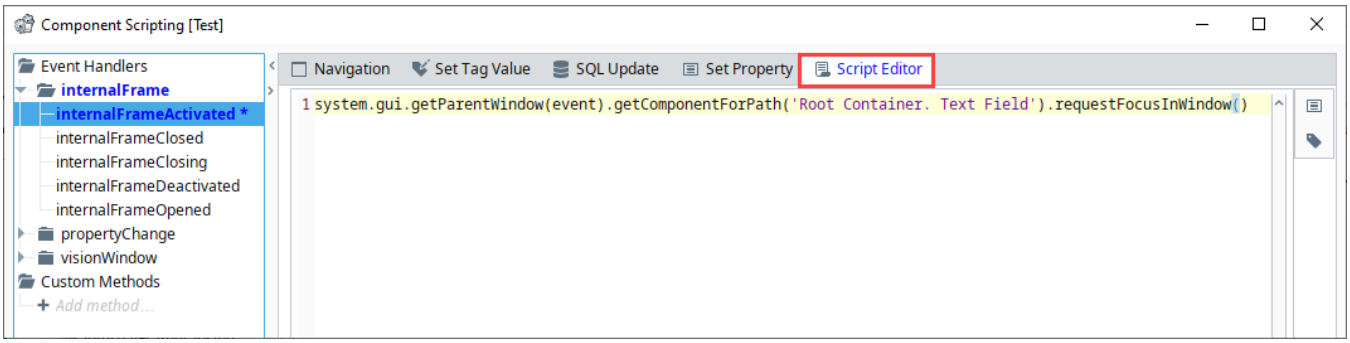


On this page ...

- Focus Order
- Requesting Focus in the Window
  - The getComponetForPath Function
- Controlling Focus Between Components in a Window
  - About Invoke Later
- Text Areas and Focus Requests

Requesting Focus in the Window

You can programmatically request that focus be given to a component by calling the function `requestFocusInWindow()` on that function. This function is called on a component and will pull the focus to that component so it is selected and ready to use. It is best used with an input component, such as a Text Field, so the user can immediately begin typing into the component. You can use it on the `internalFrameActivated` event to bring focus to the component right when the window opens.



The getComponetForPath Function

The example above references the function **getComponentForPath**. This function can be called from a window object, and allows you to specify the full path to a component inside of the window as a single string, using the following format:

Pseudocode - Get Component For Path

```
getComponentForPath('Root Container.ComponentName')
```

When referencing a component from a window event handler, such as internalFrameActivated, clicking the **Property Reference** icon will use the getComponentForPath function. While this function is useful, you never have to use the getComponentForPath function. Instead you can use the component paths that are seen from event handlers on other components. Below is a comparison of using both getComponentForPath, as well as the more traditional

Pseudocode - Component Hierarchy from a Window

```
# Both lines below will return a reference to the root container in the window that this script originates
from.

print system.gui.getParentWindow(event).getComponentForPath('Root Container')
print system.gui.getParentWindow(event).getRootContainer()
```

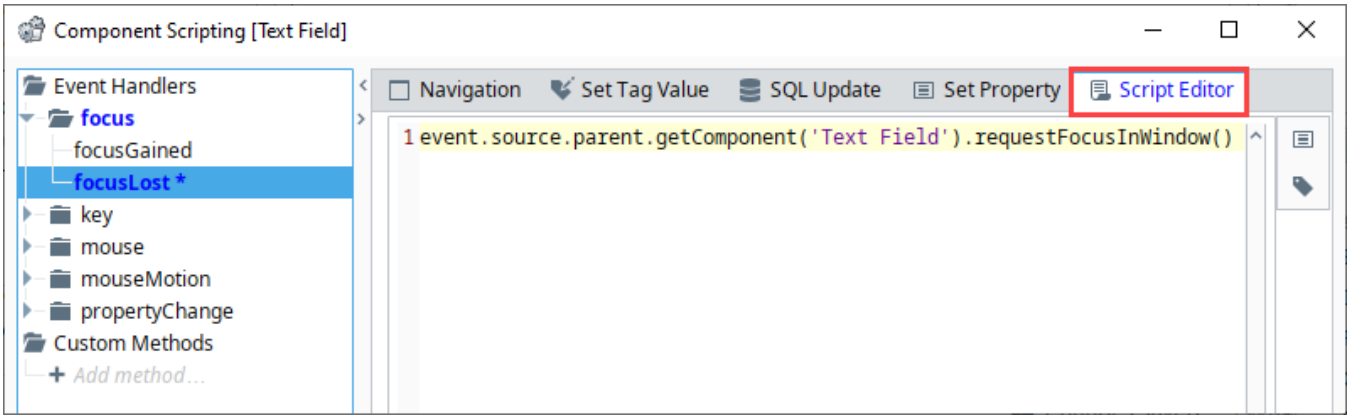
```
# These lines will both reference the text property on a Label that is nested in a Group on the window.

print system.gui.getParentWindow(event).getComponentForPath('Root Container.Group.Label').text
print system.gui.getParentWindow(event).getRootContainer().getComponent('Group').getComponent('Label').text
```

Python - Requesting Focus on Frame Activation

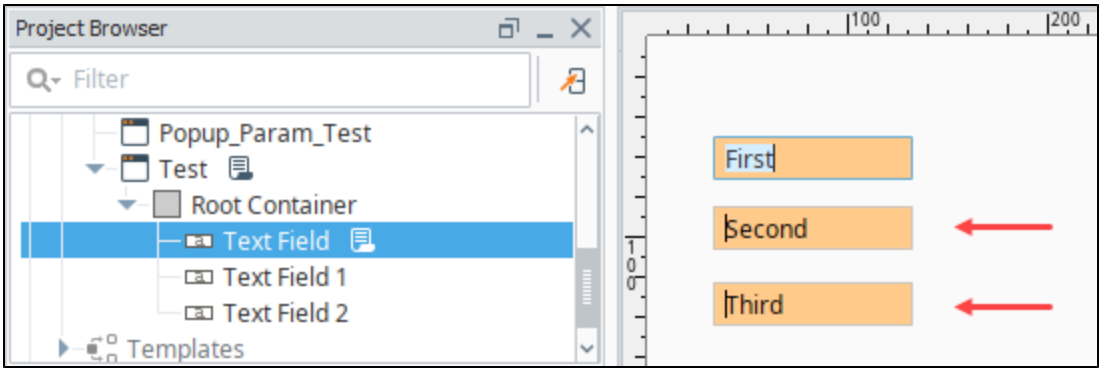
```
system.gui.getParentWindow(event).getComponentForPath('Root Container.Text Field').requestFocusInWindow()
```

Controlling Focus Between Components in a Window

In some cases, you may wish to control which component gains focus when the user clicks on a different component, or tabs away, instead of using the default focus order. You can call `requestFocusInWindow()` from the **focusLost** event to control which component should gain focus next.



However, calling `requestFocusInWindow()` may cause some irregular behavior as shown below. Notice below how the "second" and "third" Text Fields both have a text cursor in the component, but the "first" text field has focus. This is because `requestFocusInWindow()` is being called on the focusLost event, which runs when one of our components loses focus. This means that while focus is being pulled to one component (the "second" Text Field), our script changes focus again to a different component.



The solution to the problem above is to have the `requestFocusInWindow()` call occur as the last part of the event trigger. This can be accomplished in one of two ways: using **Invoke Later** under Advanced Settings, or the **invokeLater()** system function.

About Invoke Later

The concept of invoking some code later leads to a broader discussion on event handling and timing, which deviates from the purpose of this page: focus manipulation. The concept of "Invoke Later" simply means to wait for the current event to finish processing before running our focusLost script. In the scenario above, clicking from one component to another (or tabbing to a different component) natively calls a focus change event. A script on the focusLost event handler that uses requestFocusInWindow will also cause a focus change event, except it does so mid-execution of the native focus change event.

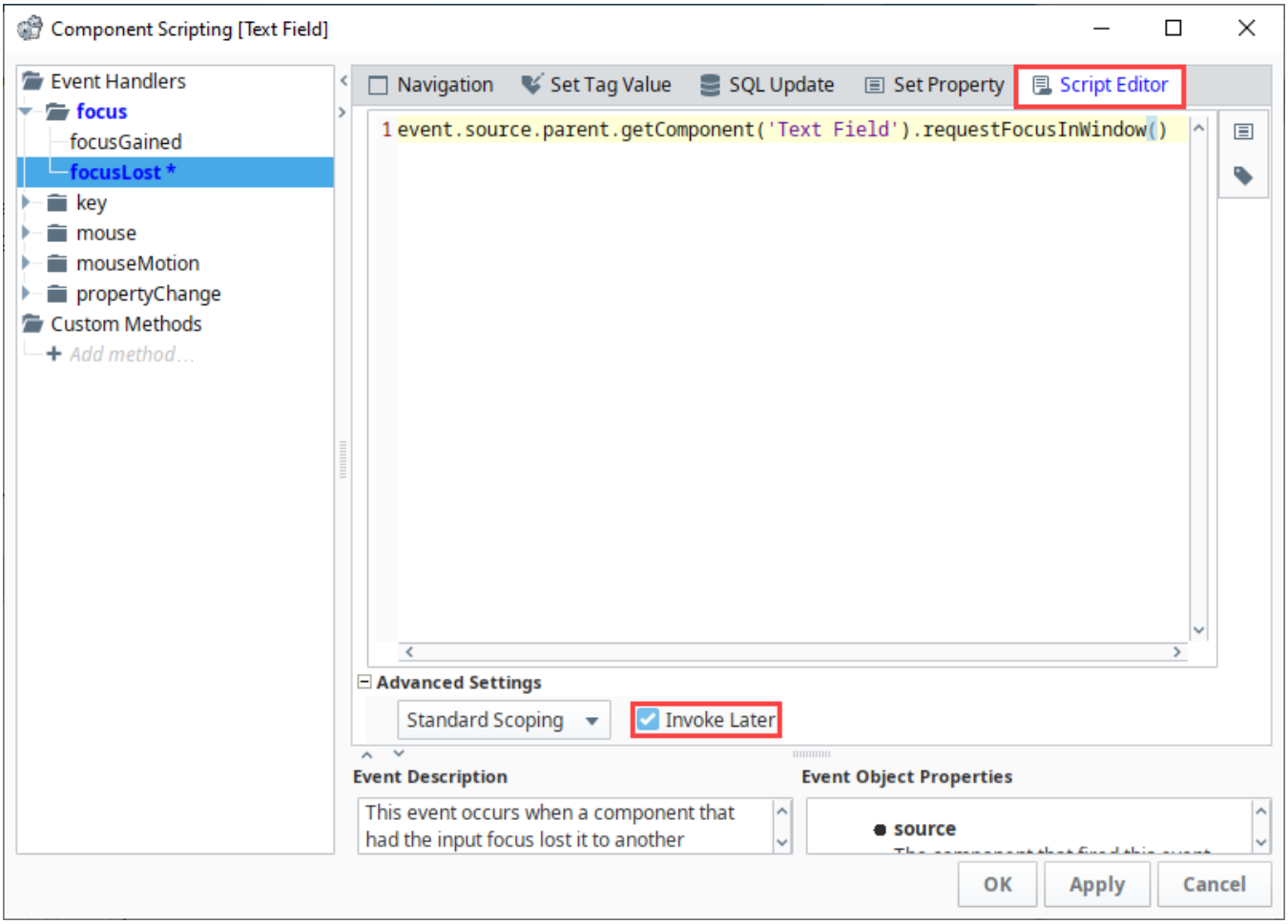
The main issue then is that focus is being moved to two components simultaneously from within the same event stack. The solution then is to have them occur in sequential order instead, with our custom focusLost script occurring last, hence using Invoke Later.

Running a script with one of the invoke later approaches mentioned here is not required on most scripts in Ignition, but it is common when the script is attempting to set or interact with focus on a window. On a related matter, invoking a script later is not the same as adding a delay mid execution.

To resolve the focus manipulation issue mentioned above, we can take one of two approaches:

Script Editor Advanced Settings

The simplest approach would be to use the **Invoke Later** option under Advanced Settings in the Script Editor.



Using the invokeLater System Function

Alternatively, we could address this by using system.util.invokeLater to request focus at the end of the event.

Python - Requesting Focus Later

```
# Create a function for invokeLater that requests focus.
def focus():
	event.source.parent.getComponent('Text Field').requestFocusInWindow()

# Call the function once the event has executed.
system.util.invokeLater(focus)
```

Text Areas and Focus Requests

Text Area components are slightly more complex when it comes to focus requests. Calling `requestFocusInWindow()` directly on the component will not allow the user to immediately start typing into the component. However, we can accomplish this by calling `getViewport()`, and then calling `getView()` first as shown in the code below.

Python - Requesting Focus from a Text Area

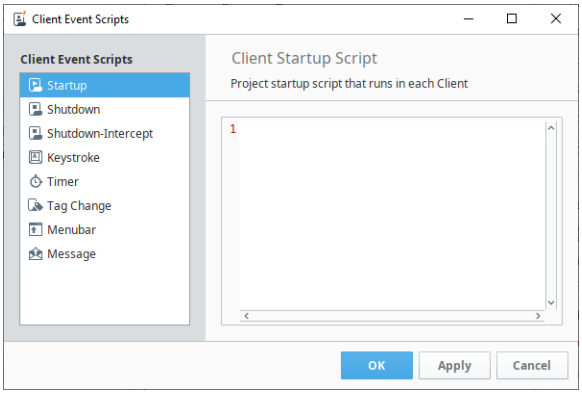
```
# Create a reference to the Text Area. This step could be skipped and combined with the line below, but is segregated in this example for clarity.
textArea = event.source.parent.getComponent('Text Area')

# This line demonstrates how to request focus on a Text Area
textArea.getViewport().getView().requestFocusInWindow()
```

Client Event Scripts

Client Event Scripts Overview

Client Event Scripts run on the computer running the Client. They allow you to execute Jython code in a running instance of a Vision Client, as opposed to running them in the Gateway.

Note: Client Event Scripts are not intended to run while in the Designer's preview mode. It is recommended that you test your Client Event Scripts in a Vision Client instead.



On this page ...

- Client Event Scripts Overview
- Startup Script
- Shutdown Script
- Shutdown-Intercept Script
  - Preventing Client Shutdown
- Keystroke Scripts
  - Client Keystroke Script Interface
  - Choose Keystroke Window
- Timer Scripts
- Tag Change Scripts
- Menubar Scripts
  - Menubar Script Interface
- Message Scripts
  - Client Message Handler Settings
- Troubleshooting Client Scripts

Startup Script

These trigger when the user logs into the client. These scripts trigger before any windows in the project are opened, so they are ideal to use when you need to dynamically open certain windows based on which user logged on.

Configurations for Client Event Startup Scripts are similar to Gateway Event Startup scripts. See the Gateway Event Scripts page for more information

Shutdown Script

These trigger when the user "shuts down" the client. The following interactions count as a shutdown:

- Logging out of the client
- The trial expiring
- The project being deleted while a client is running
- The client being terminated from the Gateway's web interface
- Closing the client, from the client

Configurations for Client Event Shutdown Scripts are similar to Gateway Event Shutdown scripts. See the Gateway Event Scripts page for more information

Shutdown-Intercept Script

The Shutdown-Intercept Script is unique in that it runs when a user attempts to close a client, but before the actual closing occurs. The main reason to use a Shutdown-Intercept script is to prevent the client from closing.

Even though this event has a similar name to the Shutdown Script, Shutdown-Intercept will only trigger when the client is requested to close from the client. Other interactions that trigger a Shutdown Script, such as logging out, will not trigger Shutdown-Intercept.

Preventing Client Shutdown

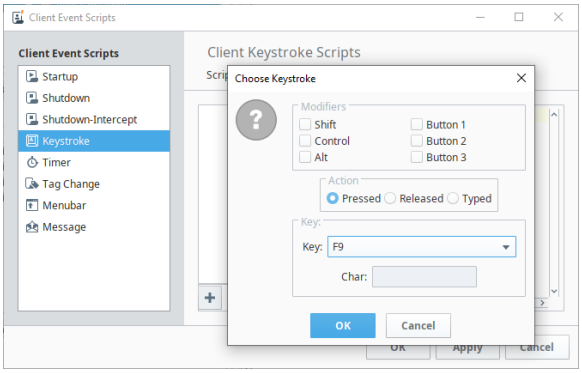
There is a special property on the event object inside the Shutdown-Intercept script that can be used to prevent the client from closing: simply type "event.cancel = 1." somewhere in your code. Doing this will cancel the shutdown event and leave the client open. This allows you to set special restrictions in regard to when the client is actually allowed to shut down, such as having a certain role, as seen in the example below:

Python - Cancel Application Exit

```
# Check to see if the user has a certain role.
if "SuperUser" not in system.security.getRoles():
        # If the role is not present, it will warn the user and cancel the shutdown process.
        system.gui.warningBox("Only administrators are allowed to shutdown the client.")
        event.cancel = 1
```

Keystroke Scripts

The Keystroke Scripts let you create different events that will activate on certain key combinations, allowing you to add keyboard shortcuts to your projects.



Client Keystroke Script Interface

- **Add Script** - Adds a new keystroke script.
- **Delete Script** - Deletes the currently selected keystroke script.
- **Script Settings** - Opens the **Choose Keystroke** window, allowing you to modify the keystroke that will trigger the script.

Choose Keystroke Window

The following areas are available on the Choose Keystroke window:

- **Modifiers** - Additional keys or mouse buttons that must be held to trigger your script. Modifiers are inclusive, so multiple modifiers must all be held down when the key is typed to trigger the script.
- **Action** - Similar to the Key Event Handlers, determines what action must occur to the key to trigger the script:
    - **Pressed** means the key was pressed,
    - **Released** means the key was released. When used in conjunction with a modifier, this action provides the user a means to prevent the script from happening after the key has already been pressed: if the user releases the modifier before releasing the key, then the script will not trigger.
    - **Typed** means the user typed a specific character. Selecting this action enabled the **Ch ar** field under the key section. This provides an easier way to trigger the script based on non-standard ascii characters.
- **Key**- Which key will trigger the script. A dropdown is available when the Action is set to **Pressed** or **Released**. A Text Field is available if the Action is set to **Typed**.

Special keys like the Function keys (F1) or ESC key are only available in the pressed and released actions.

> ⚠ Some operating systems reserve certain keys for certain functions, and will capture the key press or release before it gets sent to the Client. For example, many operating systems use the TAB key to shift focus to the next field.

Timer Scripts

Run on a timer in the same fashion as their Gateway counterpart, except each instance of the project (i.e., client looking at the project containing the timer script) has a separate instance of the timer script running. Timer scripts that insert records into a database, or write to a Tag, are better suited as Gateway Event Scripts, since there will only ever be one running. If there are not any open clients, there will not be an instances of this script running.

Configurations for Client Event Timer Scripts are similar to Gateway Event Timer scripts. See the Gateway Event Scripts page for more information

Tag Change Scripts

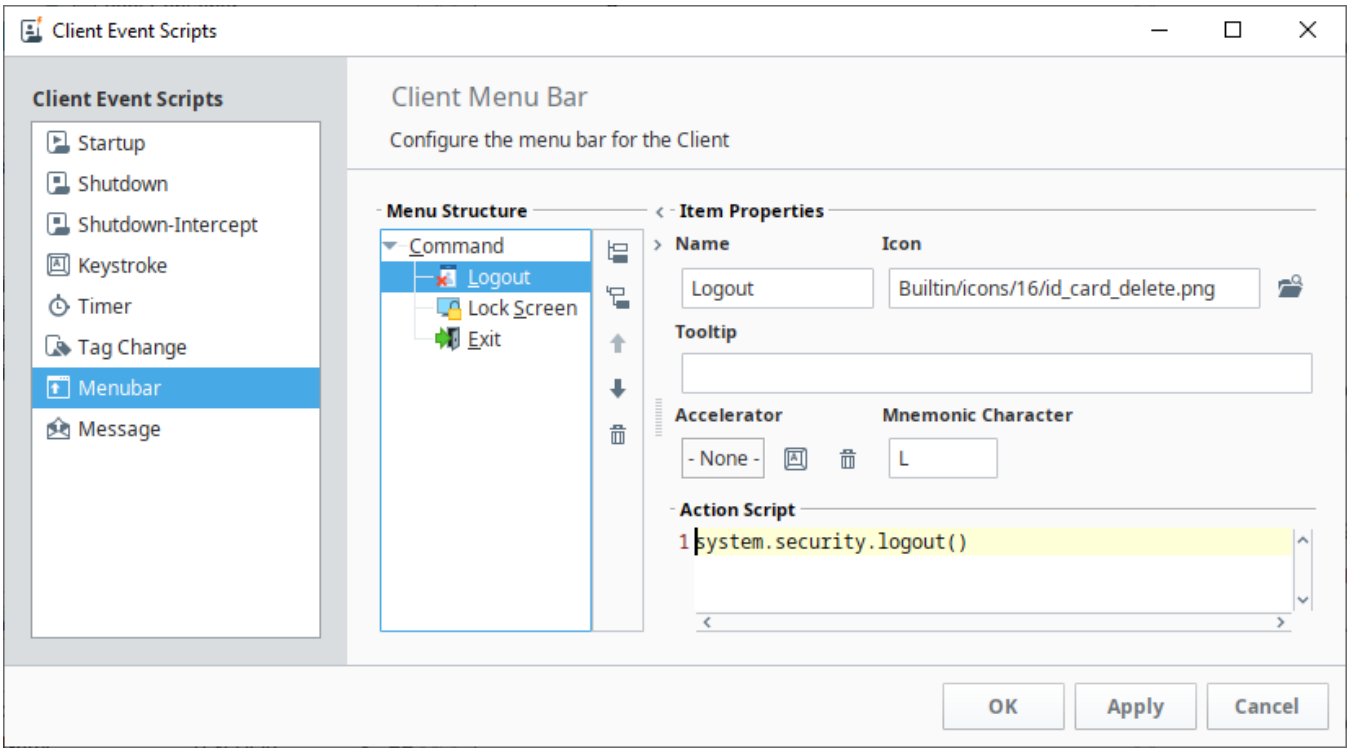
Monitor one or more Tags, and trigger a script in each instance of the client on Tag change. Unlike the Gateway Tag Change Script, Client Tag Change Scripts can monitor a Client Tag. Much like Timer Scripts, they only run in each instance of the client, so if there aren't any open clients, then the script will never execute.

Configurations for Client Event Tag Change Scripts are similar to Gateway Event Tag Change scripts. See the Gateway Event Scripts page for more information

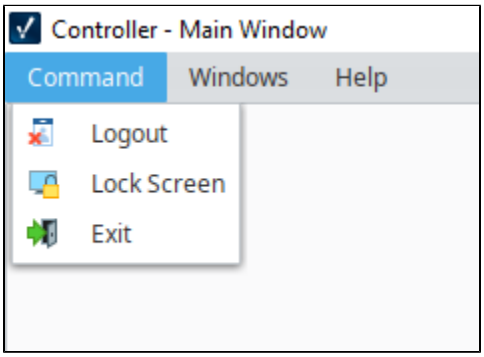
Menubar Scripts

The Client Menubar Scripts create and control the options available in the menubar of the Client. As such, these scripts are only available in the Client Scope. By default, a Client will have three menus: Command, Windows, and Help. The Windows and Help Menu are separate, and controlled through the project properties, but the Command menu is actually created in the Client Menubar Scripts.

Selecting **Menubar** displays the **Menu Structure** list.



The structure is then mimicked in the client when the menu name is selected.



INDUCTIVE UNIVERSITY

Menu Bar

Watch the Video

Menubar Script Interface

The user interface on the Menubar script is divided into two sections.

Menu Structure

A tree representing the layout of the menu bar. Items at the root of the tree will appear on the menu bar in the client, and nested items will appear as subitems in the client. In the image of the Client Event Script above, notice that the **Logout**, **Lock Screen**, and **Exit** items are children of the **Command** item. When looking at the image of the menubar in the Client, these three items appear under the Command Item.

Because the Menu Structure ultimately impacts the order, the following buttons are available to help sort each item.

- **Add Sibling** - Adds a new sibling, or peer item, to the selected item.
- **Add Child** - Adds a new child to the selected item.
- **Move Up** and **Move Down** - Moves the selected item up or down in the list. The order in the list determines the order that items appear in the menu, so these buttons can be used to group meaningful items together.
- **Delete** - Deletes the selected item, removing it from the menu.

Item Properties

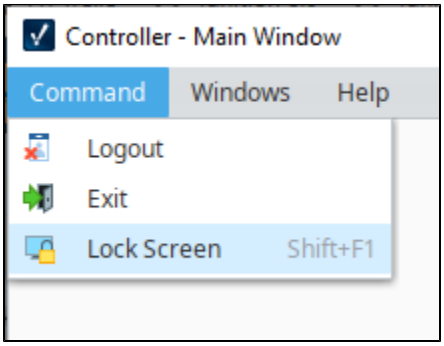
- **Name** - The text on the item in the menu.
- **Icon** - What image should appear next to the item, if any.
- **Tooltip** - Optional property allowing you to specify a tooltip when the user hovers the mouse cursor on top of the item.
- **Accelerator** - Allows you to define a keyboard shortcut that will quickly select the item. Please see the Accelerator section below for more details.
- **Mnemonic Character** - Allows you to define a character key that will trigger the option when the menu is open. Please see the Mnemonics section below for more details.
- **Action Script** - The script that will run when the user selects the item. Every item, even those at the root and branches may have a script defined.

> **Note:** It is uncommon to have a script defined on a branch, as they usually act as a means to list other items.

Accelerators

An accelerator is a key or key combination that can be pressed at any time in the client to initiate that menu item's event. If an accelerator has been configured for an item, then it will be listed on the menu in the client. Below we see our initial menu bar has been modified with the accelerator **Shift+F1**

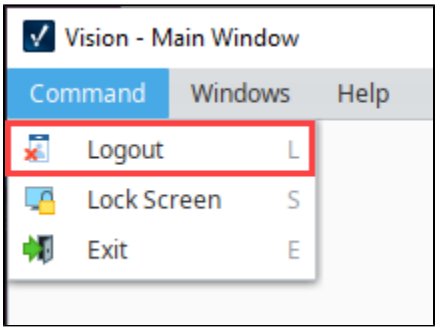
. Now the Lock Screen item may be called anywhere in the client by holding **Shift** and pressing the **F1** key.



Mnemonics

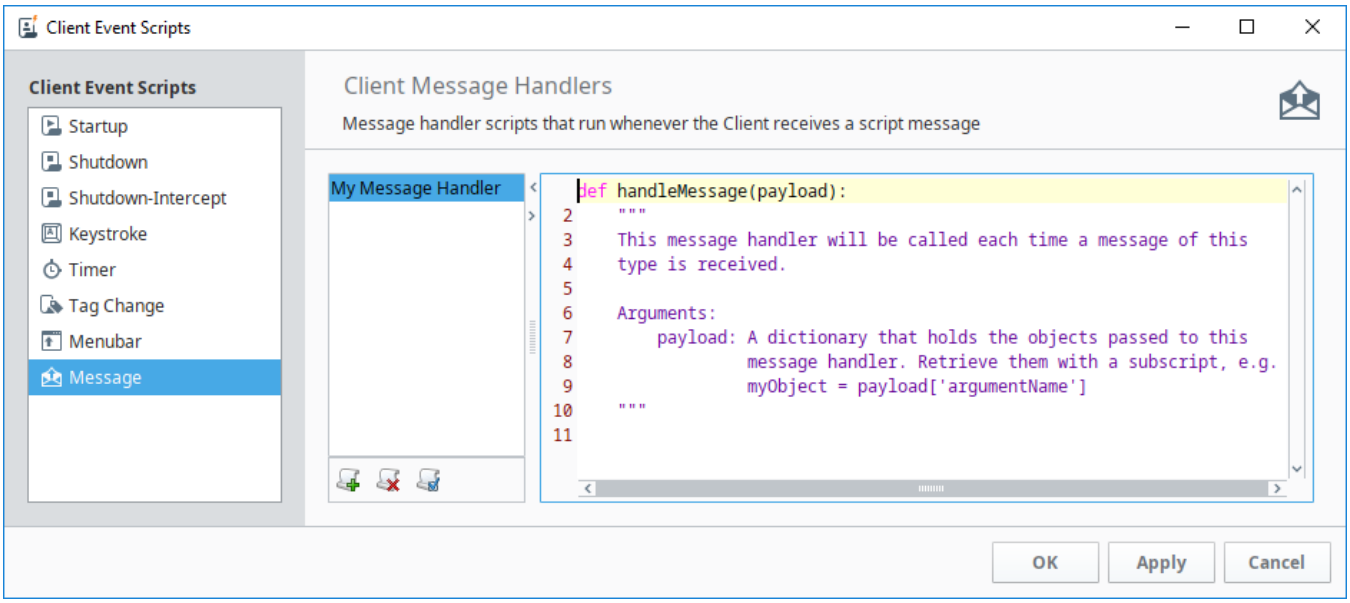
The mnemonic character is a key that can be pressed when the menu is opened. This is functionally similar to an Accelerator, in that it allows the user to select an item in the menu without clicking on it. However, mnemonics differ in that they only call an item when the menu is open, and the item is visible on the screen.

Users can identify mnemonics by a character to the right of the command in the menu. In the image below, we see that the **Logout** item has an "L" character. This means the user can now press the "L" key to select the Logout item. However, this will not work unless the menu is open, so if the user accidentally presses the L key while the menu is closed, the script will not trigger. Notice, how each command also has a mnemonic character defined for each command.



Message Scripts

Client Message Handlers are created and called using the same mechanisms as Gateway Event Scripts. There are two main differences that make Client Message Handlers stand out from Gateway Message Handlers: they run in the Client, and they have different settings.

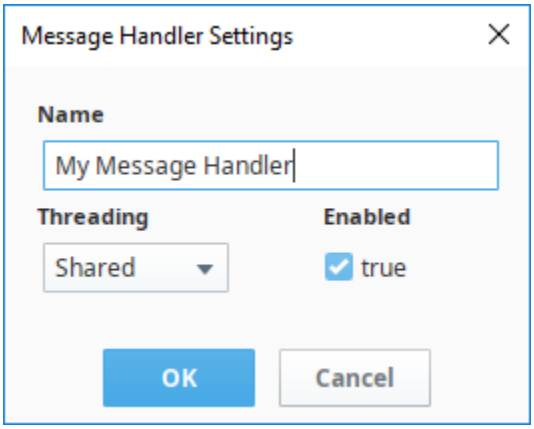


INDUCTIVE UNIVERSITY

Script Messaging

Watch the Video

Client Message Handler Settings



Client Message Handlers have the following settings:

- **Name** - The name of the message handler. Each message handler must have a unique name per project.
- **Threading** - Determines the threading for the message handler. Contains the following options:
  - **Shared** - The default way of running a message handler. Will execute the handler on a shared pool of threads in the order that they are invoked. If too many message handlers are called all at once and they take long periods of time to execute, there may be delays before each message handler gets to execute.
  - **Dedicated** - The message handler will run on its own dedicated thread. This is useful when a message handler will take a long time to execute, so that it does not hinder the execution of other message handlers. Threads have a bit of overhead, so this option uses more of the Gateway's resources, but is desirable if you want the message handler to not be impeded by the execution of other message handlers.
  - **EDT** - This will run the message handler on the **Event Dispatch Thread** (EDT) which also updates the GUI. If a message handler were to take a long time to execute, it would block the GUI from running which may lock up your client. This is helpful when your message handler will be interacting with the GUI in some way, as the GUI will not be able to update until the message handler finishes.

For more information on Message Handlers, such as working with the Payload argument, or calling them, please see the Gateway Event Scripts page.

Troubleshooting Client Scripts

The Console is very a important tool in Ignition for troubleshooting Client scripts. You can check to see if your script is working directly from the Client window, or the Designer while in Preview Mode. Any client scripting errors along with printouts go to the Console. The Console will identify the script name, error message, what line the script error is in, and a description of the problem.

To access the Console from a Client, go to the menubar and select **Help > Diagnostics > Console**. To access the Console from Preview Mode in the Designer, go to the menubar **Tools > Console**.

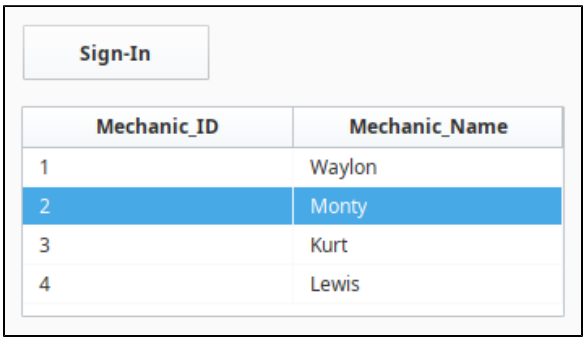
Related Topics ...

- Gateway Event Scripts

Read a Cell from a Table

After data has been populated in a Table or Power Table component, it may be useful to read or extract a particular cell from the Table, especially if users can select rows in the Table. On this page, we'll take a look at how to retrieve information from a particular cell on a Table.

The example on this page utilizes a simple Power Table and Button component. Users select a row from the table to extract one of the cells from the highlighted row in the Power Table and press the Sign-In button.



On this page ...

- Power Table Example
  - Retrieve a single cell requires that we reference the Power Table a couple of times in the same script. Because of this, we can create a variable that references the table, and use the variable later on. Be aware that a Button and Power Table can be in the same container or separate containers. If your Button and Power Table are in separate containers your path may differ.Selecting a Single Cell
  - Selecting Multiple Cells - Same Column
  - Test Your Script

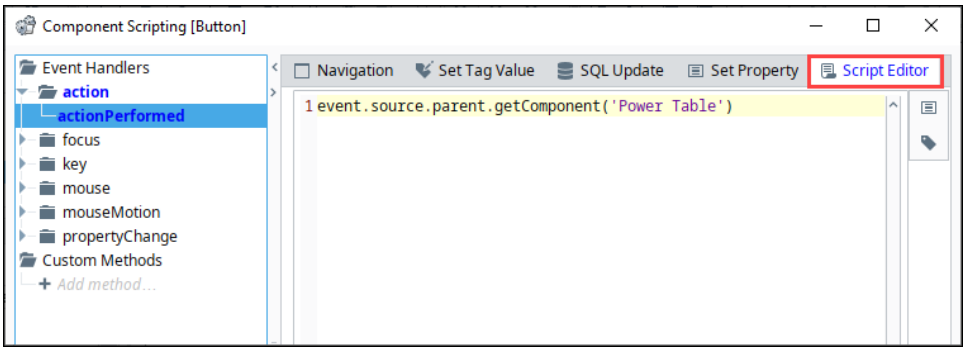
Power Table Example

This example provides the code that you can use for selecting a single cell and multiple cells in a Table.

Note: Before we get started, it is important to understand that a cell in a Table is actually a cell in a dataset. Data in a Table is stored in a property on the component (the **Data** property), and the script needs to interact with that property.

Retrieve a single cell requires that we reference the Power Table a couple of times in the same script. Because of this, we can create a variable that references the table, and use the variable later on. Be aware that a Button and Power Table can be in the same container or separate containers. If your Button and Power Table are in separate containers your path may differ.**Selecting a Single Cell**

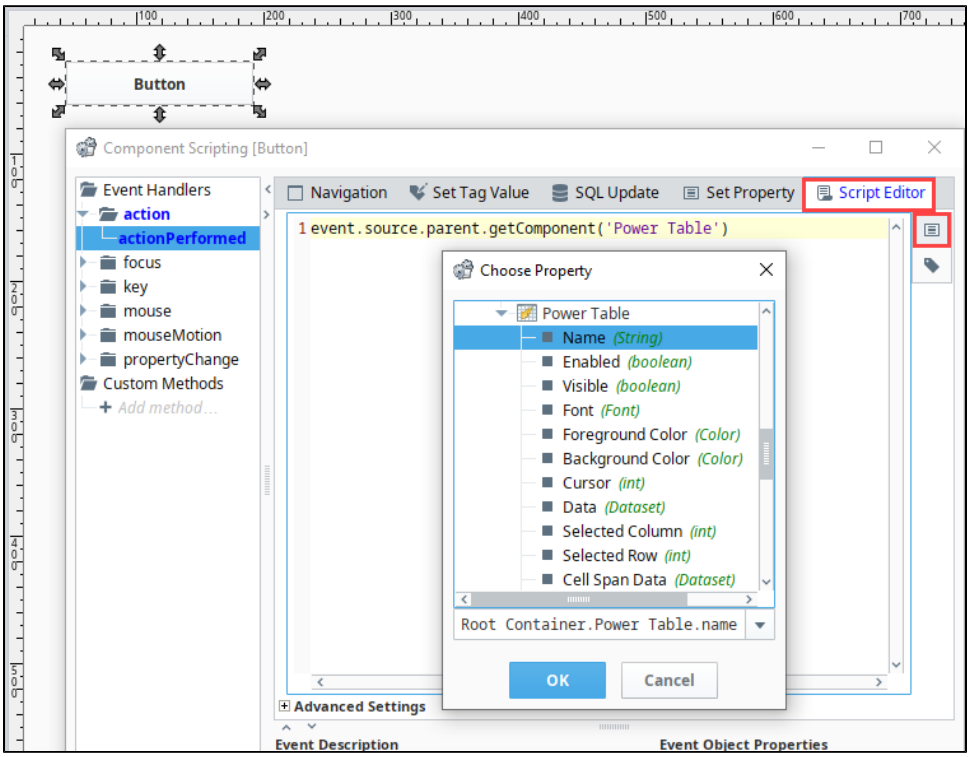
1. Drag a **Button** and **Power Table** components to your Designer workspace. Assuming the Power Table has not been renamed, we can reference the Power Table component with the line of code below as shown in the image below.



Python - Reference the Power Table

```
# Grab a reference to the Power Table.
table = event.source.parent.getComponent('Power Table')
```

2. As mentioned above, if the **Button** and **Power Table** are in separate containers, the path will be different. Double click the **Button** component to open the **Scripting** window, select the **action > actionPerformed** event handler, and click the **Property Reference** icon to generate the path to a property, but not the component.
3. Next, select a property from the **Power Table** in the **Choose Property** window and click **OK**.
4. Remove the **".propertyName"** portion at the end of the script in the Script Editor, and click **OK**.



5. Next , we need to figure out which row contains the cell we want to read from. In our example, the user will select the row for us, so we simply need to know which row is selected when the **Button** is pressed. Fortunately, the **Power Table** contains a **Selected Row** property that can be used to determine the row that is selected. Furthermore, we can use the getValueAt() function that is built into datasets.

Python - Reference the Name of the Column

```
# Here the "Mechanic_Name" argument references the name of the column.
table.data.getValue(table.selectedRow, "Mechanic_Name")
```

6. Alternatively, we can use an integer as the last argument to specify the index of the column our cell is located in.

Python - Reference the Index of the Column

```
# Here the '1' references the index of the column in the Power Table's raw dataset (Data property).
# Remember, indexes are zero-based, so this would retrieve the second column from the left.
mechanicName = table.data.getValueAt(table.selectedRow, 1)
```

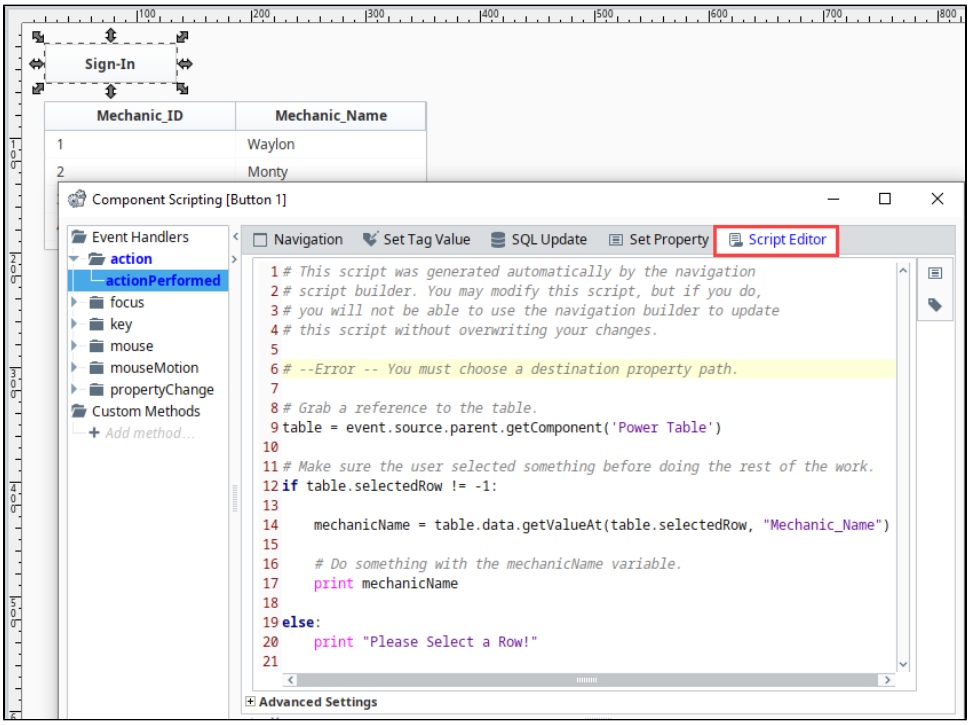
7. Lastly, we will need to account for scenarios where the user did not selected a row, otherwise, this will throw an exception. An if-statement can be used here to check for a -1 value on the Power Table's **Selected Row** property, and an else-clause can be used to notify the user that a row needs to be selected. There is a property on the Power Table named **Selection Mode** that allows users to select multiple rows. By default it is set to only allow a single row to be selected. Change it to test your button.

   Here is all the code that you'll need to read a single cell from a Power Table in the same container.

Python - Putting it all Together

```
# Grab a reference to the table.
table = event.source.parent.getComponent('Power Table')

# Make sure the user selected something before doing the rest of the work.
```

```
if table.selectedRow != -1:

        mechanicName = table.data.getValueAt(table.selectedRow, "Mechanic_Name")

        # Do something with the mechanicName variable.
        print mechanicName

else:
        print "Please Select a Row!"
```



8. To test your script, follow the steps in the Test Your Script section.

Selecting Multiple Cells - Same Column

By default, the Power Table allows for multiple rows to be selected. However, the Selected Row property only shows the row index for the first row selected. Fortunately, the Power Table also has a built-in getSelectedRows() function that can be used to return all of the indices. We simply need to iterate over each index.

We can still use the Power Table's **Selected Row** property to test if any rows are selected, but we could instead check the length of the object returned by getSelectedRows():

Here is all the code you'll need to read multiple cells from a Table in the same container. Copy the code in the code block below and replace the code in previous example to allow for multiple rows to be selected.

Python - Reading Multiple Cells

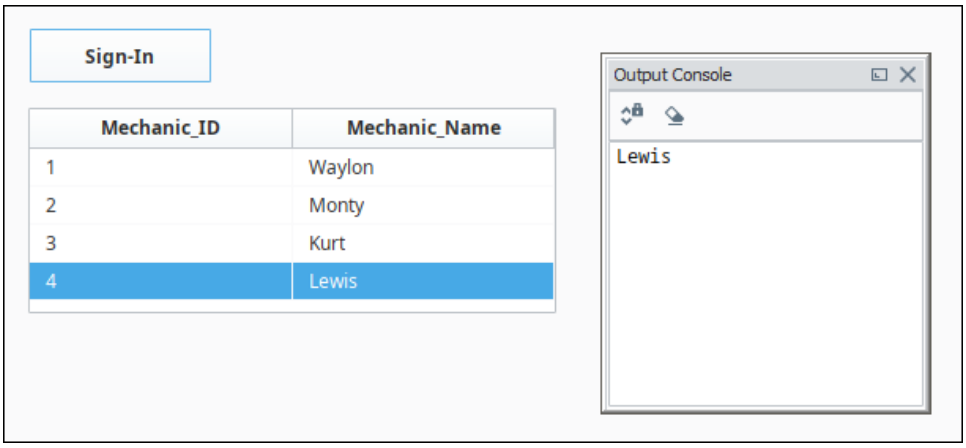
```
# Grab a reference to the table.
table = event.source.parent.getComponent('Power Table')

selectedIndices = table.getSelectedRows()

# If the length of selectedIndices is greater than 0, then at least one row is selected.
if len(selectedIndices) > 0:

        for index in selectedIndices:

                mechanicName = table.data.getValueAt(index, "Mechanic_Name")
```

```
                # We can do something with the value as we iterate, or append to a list and
                # do something with the entire list after the for-loop completes.
                print mechanicName

else:
        print "Please Select a Row!"
```

Test Your Script

Now you're ready to test your script whether you are selecting one row or multiple rows.

1. Open the Output Console and select **Tools > Console** in the menubar.
2. In **Preview Mode**, select a row in the **Power Table**, and click the **Sign-In** button. You will see the selected Mechanic_Name displayed in the console.
3. To test selecting multiple rows, shift click a couple of rows and press the **Sign-In** button, and you'll see the selected Mechanic_Names displayed in the console.



Related Topics ...

- Vision - Table
- Vision - Power Table

Historian in Vision

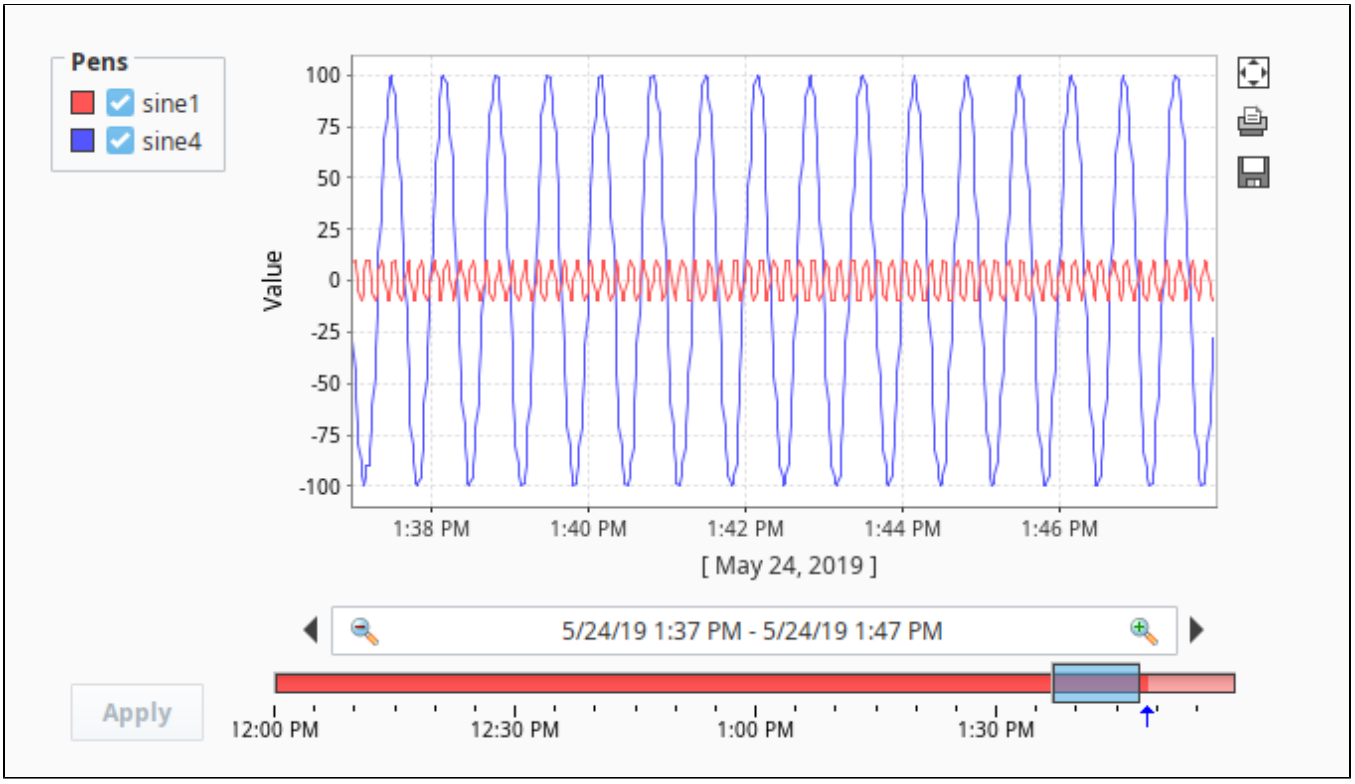
The Tag Historian is a powerful system that can easily be set up to store Tag data to a database to be accessed at a later time. The Vision system has many components that are designed to easily pull the information out of the database and display it, most commonly in chart format.

On this page ...

- The Easy Chart
- The Classic Chart
- The Sparkline Chart
- The Status Chart

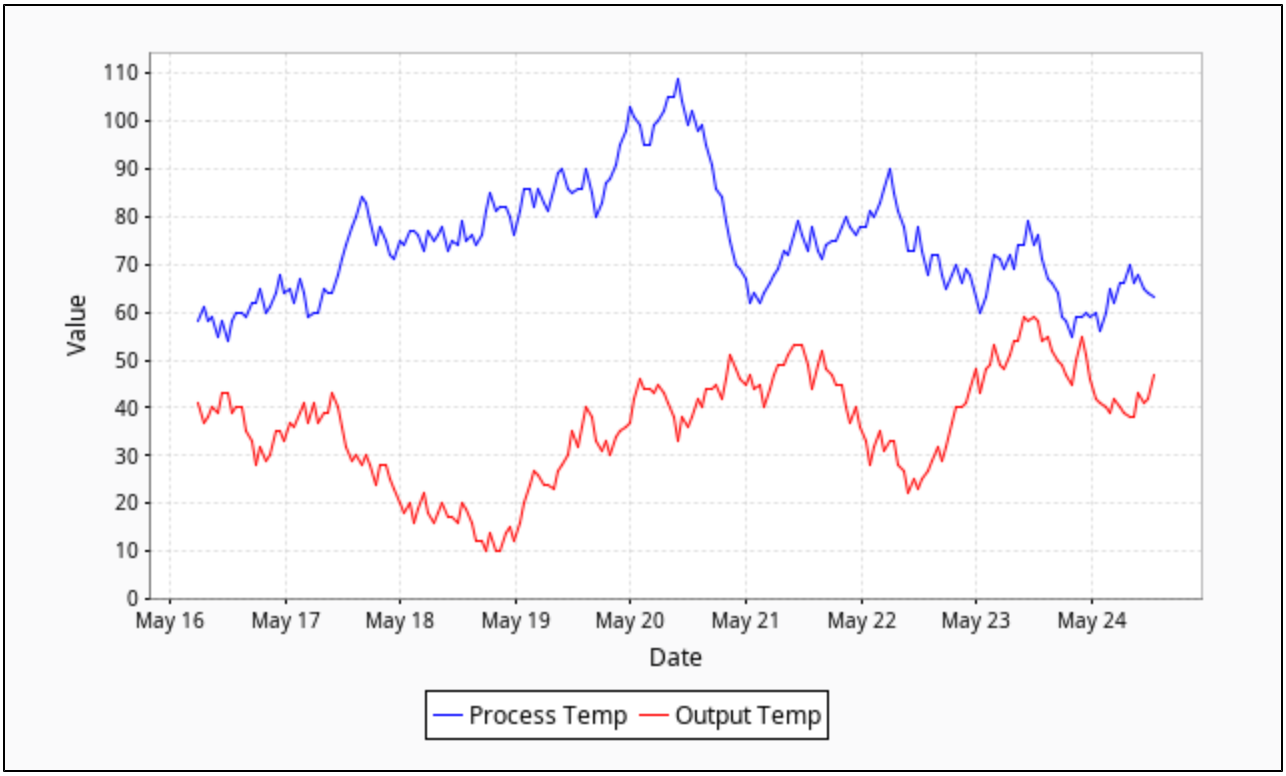
The Easy Chart

The most popular Vision component that trends historical data would be the Easy Chart. This component is simple to initially configure and contains many ways to customize the look and behavior of the chart. The Easy Chart also features a customizer that enables you to change the many different settings of the component.



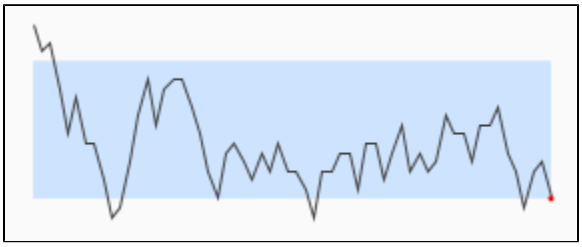
The Classic Chart

The Classic Chart pulls in data from the Tag History system and can display it in a variety of ways. While not as simple to get started with as the Easy Chart, the Classic Chart provides the unique ability to trend data that isn't based on a timestamp, but instead something like a category.



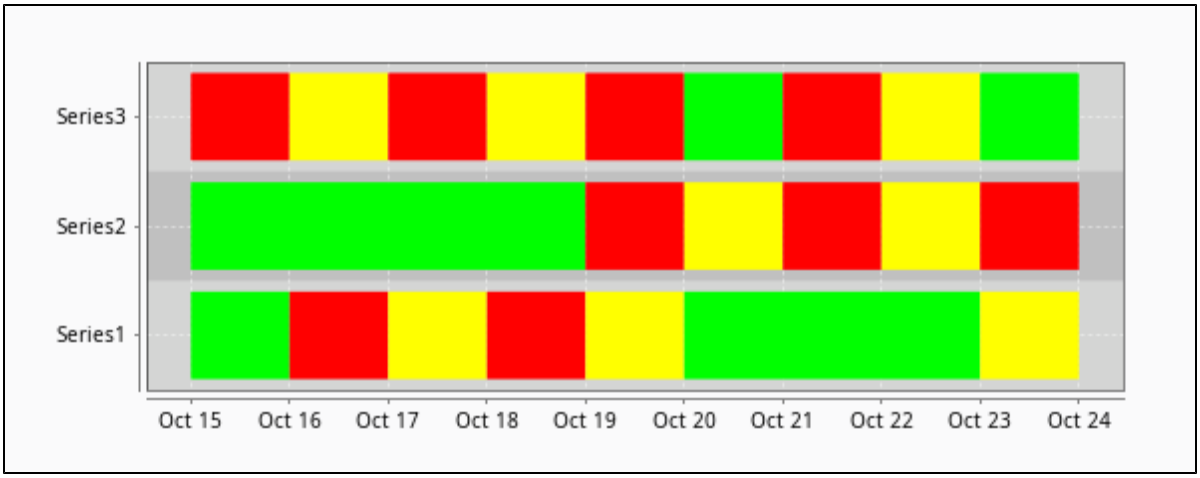
The Sparkline Chart

The Sparkline Chart is a simple chart that strives for minimalism rather than many fancy settings. Even with its simplicity, it remains a powerful tool. For example, the Sparkline Chart works great when used in High Performance HMI environments, where muted colors and lines are used to help draw the users attention to the places that matter.



The Status Chart

The Status Chart displays discrete data over a period of time. This provides a great way of displaying status information for various machines. It can also accept data in both wide and tall format, making it easy to use with any type of stored historical data.



In This Section ...

Using the Vision Easy Chart

Trends Made Easy

The Easy Chart was developed with the Tag Historian system in mind. Once an Easy Chart is created, you can drag and drop historical Tags onto the chart. The chart will immediately retrieve the results and trend the history. Data that is not set up with Tag Historian can also be displayed on the chart, as long as the data has timestamps associated with the values. For this type of data, database pens are created and displayed.

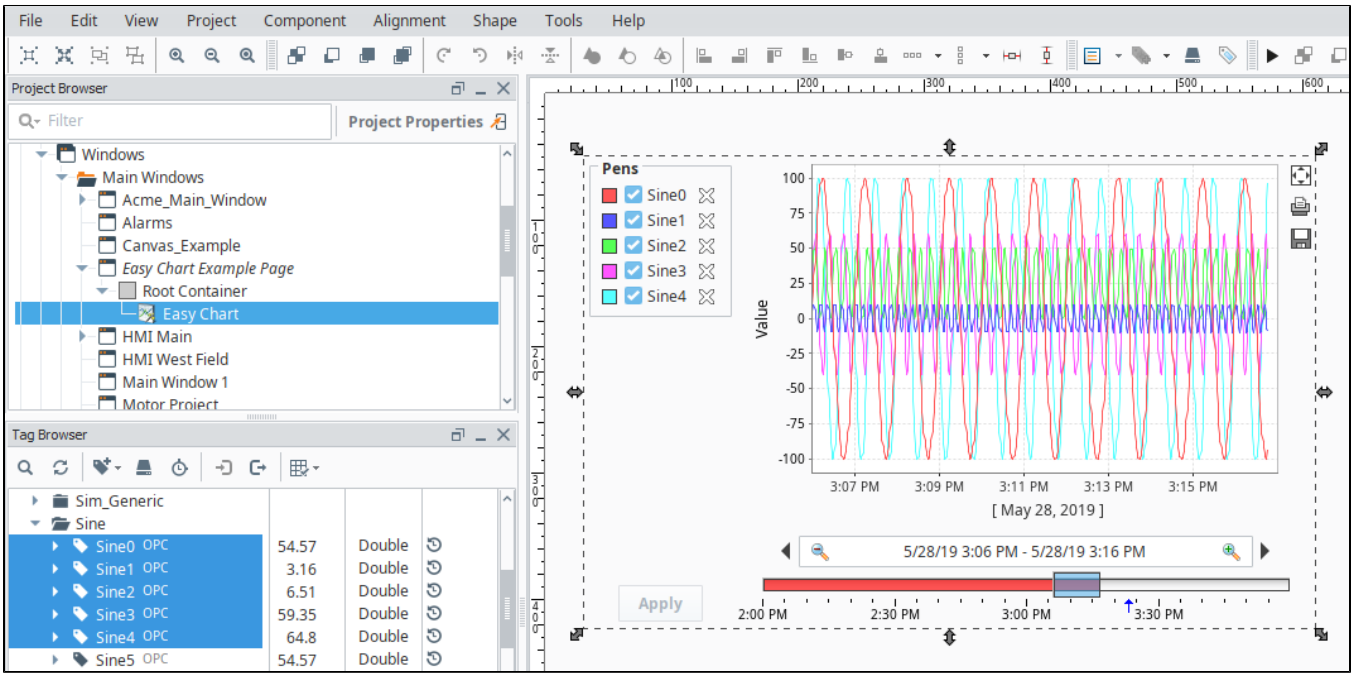
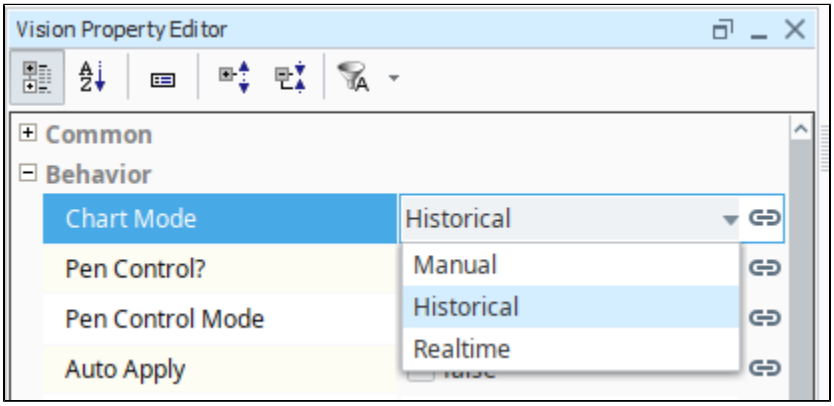


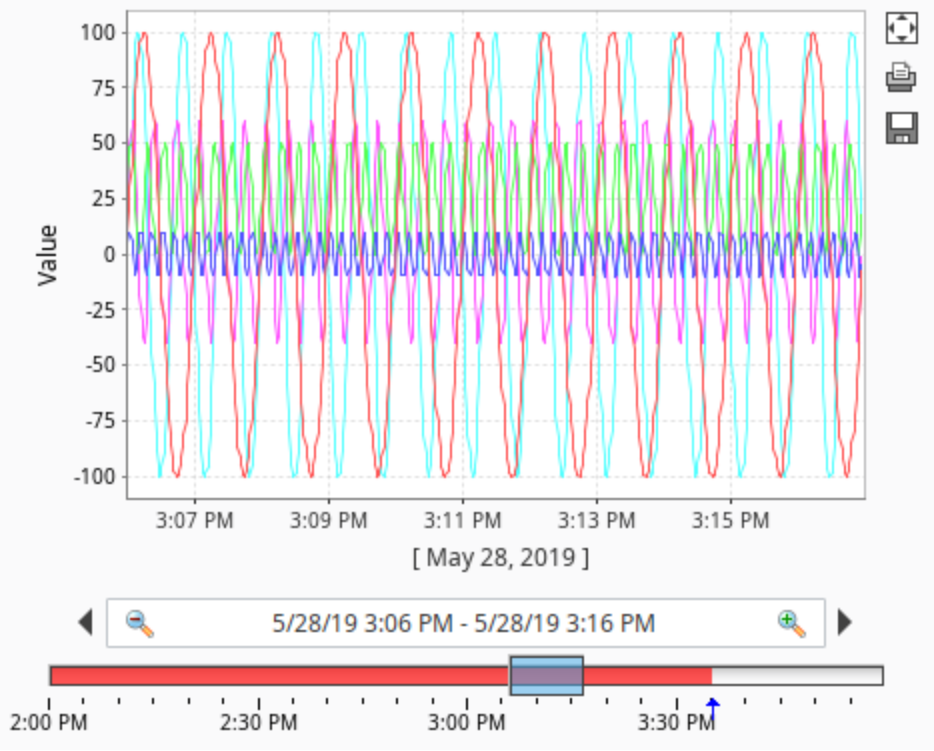
Chart Modes

The Easy Chart has a Chart Modes property that changes the behavior of the chart in several ways. The three chart modes are Historical, Realtime, and Manual. The mode is set in the Vision Property Editor in the Chart Mode Property.



Historical

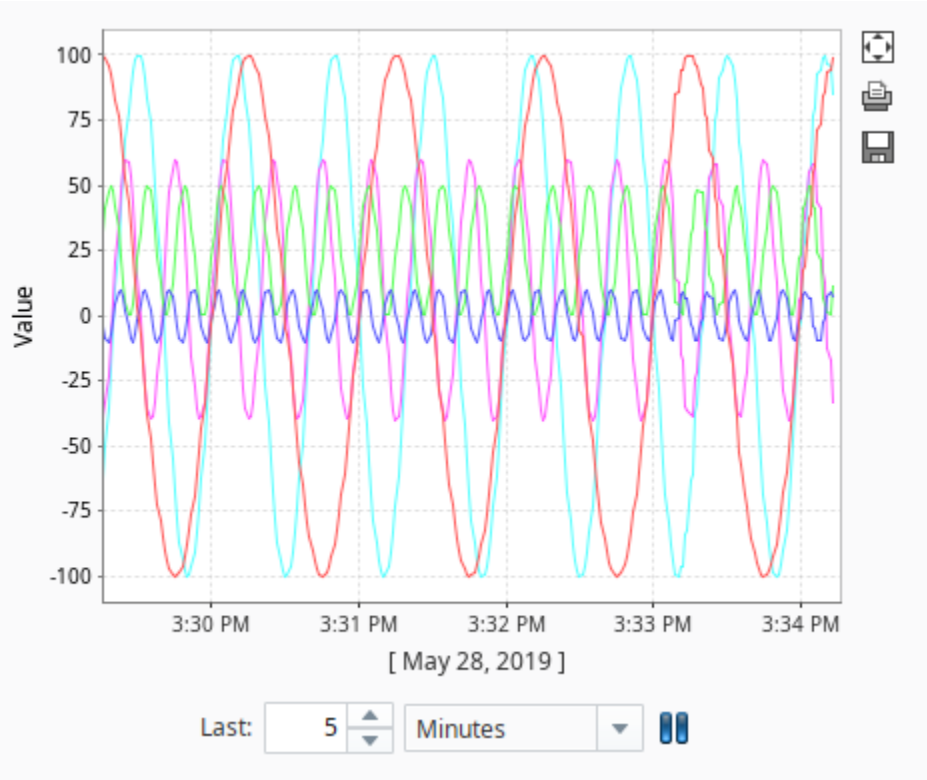
This mode places a Date Range Selector component at the bottom of the chart. This mode allows users to select a start and end date for the trends. The data density is shown at the bottom of the chart: the more vibrant the color, the higher the density. This the default mode, and commonly used in situations where users need to look at specific date ranges. It is important to remember that the chart does not poll in this mode. New values are only added to the chart when the selection box is moved or re-sized.



Realtime

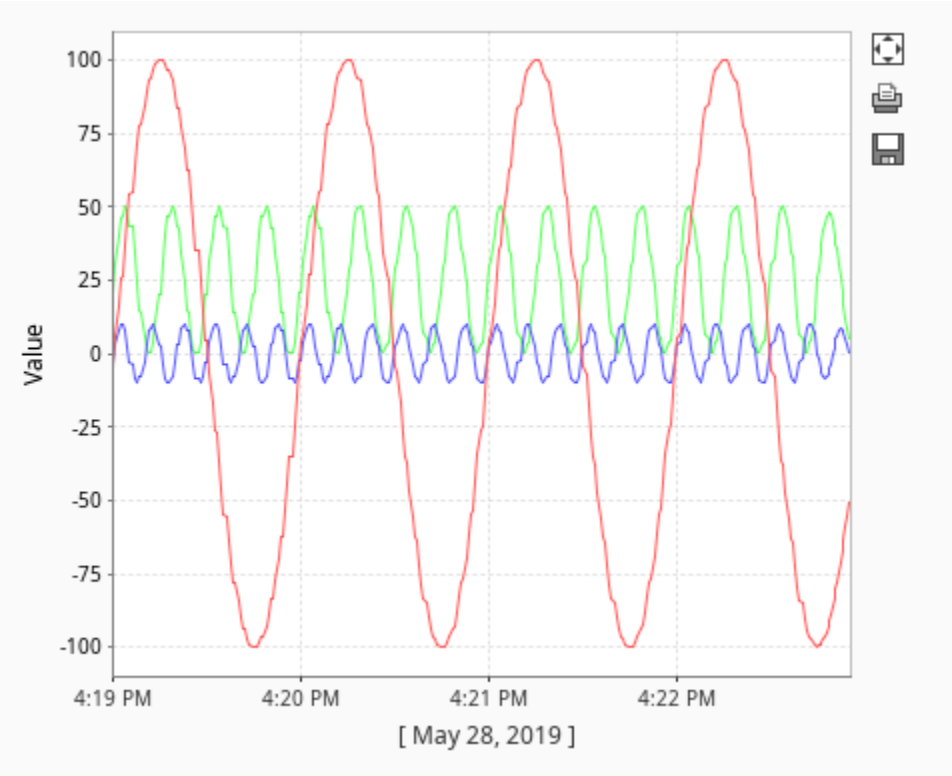
Displays the most recent data for each pen. Users are able determine how far back in time the trend should display with the Spinner and Dropdown components at the bottom of the chart. In this mode, the chart polls for data at the rate specified by the Poll Rate property.

In some cases, you may notice that the most recent values on tag pens tend to flat-line, and then 'snap' to a different value. This is generally due to how often the chart polls versus how often history is being generated. If the chart polls at a 1,000ms rate, but history is only recorded at a 10,000ms rate, then the chart will extrapolate the last recorded value for 9,000ms. After a new value is recorded, the next poll will return the latest value, and the flat-line will change position.



Manual

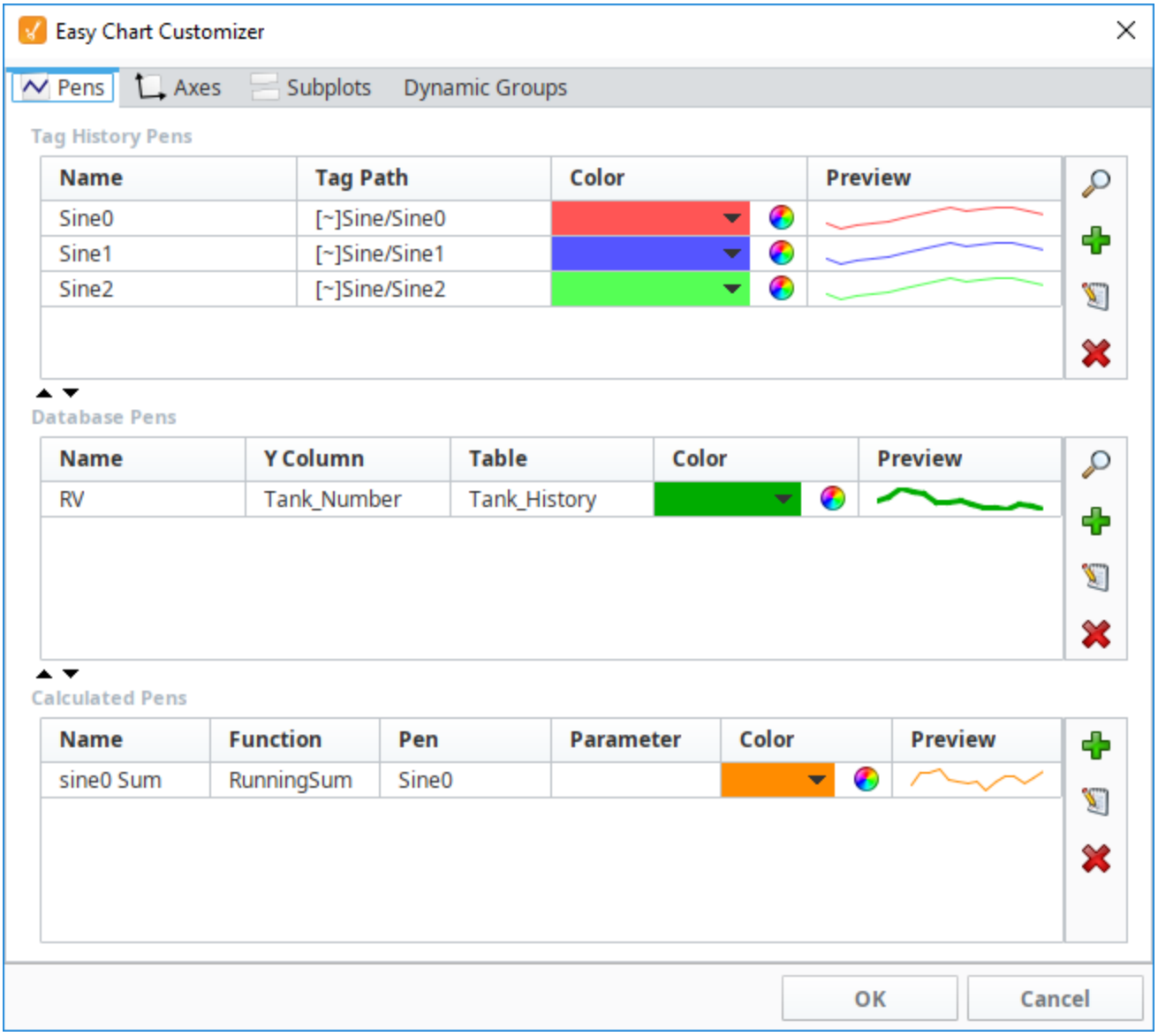
Similar to the Historical mode in that trends range from two points in time. However, there is not a built-in method for users to change the data range. Instead, some sort of binding can be applied to the chart's Start Date and End Date property. This mode is generally used in situations where only certain date ranges should be shown, such as values recorded during the previous day, or shift.



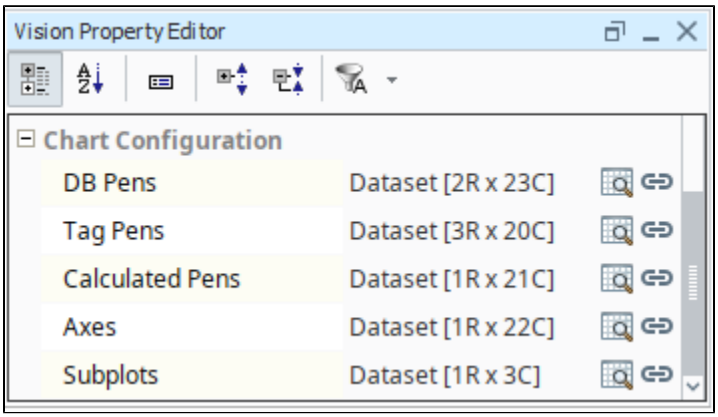
Pens

Pens on the chart, or each series of data points on the chart, can be customized to take on a number of different styles and colors. There are three types of pens, and each pen functions in a similar manner. What makes them different is how their data is collected.

- **Tag Pens** - These pens are driven by the Tag history system. Data from any historical provider can be used, and Tag history from different databases can be shown on the same chart. These are the type of pens that are created when Tags are dragged onto the chart. Since the Tag History system is being used, an Aggregation Mode must be selected, and the Tag Path needs to be specified for each pen.
- **Database Pens** - These pens are driven by a SQL query, so they are ideal to use when trending Transaction Group data. However, they can query for data in any connected SQL database, so it is possible to show historical data recorded by other systems on the Easy Chart.
- **Calculated Pens** - Pens that derive their data from calculations performed on other pens. Data for calculated pens is not stored directly into a database, but rather calculated in the runtime based on data from another pen. These type of pens are great for display running totals, control limits, or specification limits.



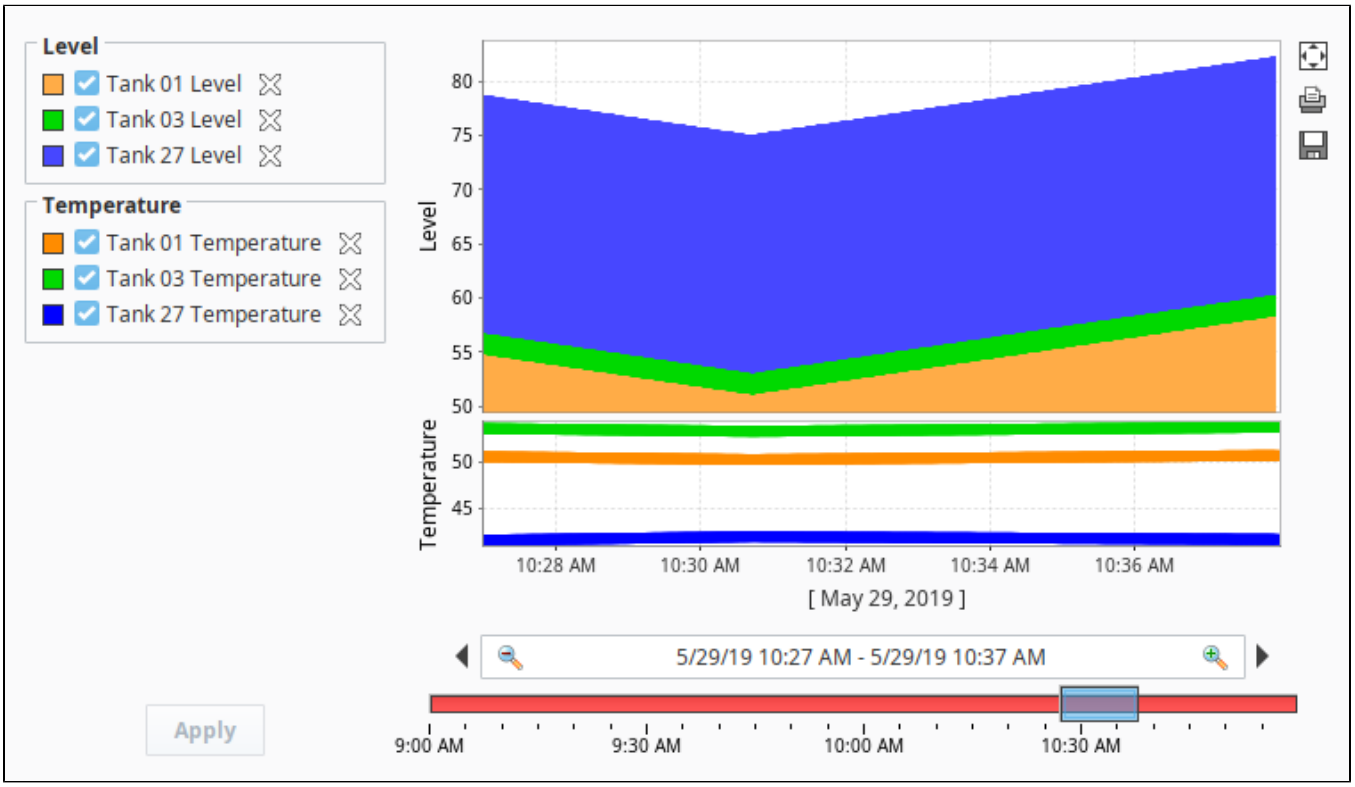
Pens can be added manually to the chart with the Easy Chart Customizer or added dynamically by modifying the various pens properties listed under Chart Configuration in the Vision Property Editor. These properties contain the configuration of each type of pen and can make use of the binding system. Because of this, pen preferences can be saved to a database table and then queried in the runtime with a SQL binding. Additional adjustments can be made with Cell Update bindings or scripting to create a dynamic-yet-robust chart.



Note: When you add pens to the Easy Chart, all pens will show up with a white X next to their name. This exists for creating an ad hoc chart used with the Tag Browse Tree component. To remove the ability to delete pens, you must edit the Tag Pens dataset property. Click on the dataset viewer icon, then in the last column "User Removable", deselect the checkbox and click OK.

Easy Chart Customizer

Aside from the properties on the component, the Easy Chart Customizer allows modifications to be made to the chart. Along with configuring pens, the customizer can be used to create subplots, new axes, and dynamic groups. Once created, each pen can be assigned to any available axes or subplots. This way different values can be shown on different plots with an axis that is specific to data at hand.



Related Topics ...

- Easy Chart Customizer

In This Section ...

Easy Chart - Axes

Configuring Multiple Axes on an Easy Chart

The Easy Chart supports the use of multiple axes for displaying data from the Tag Historian.

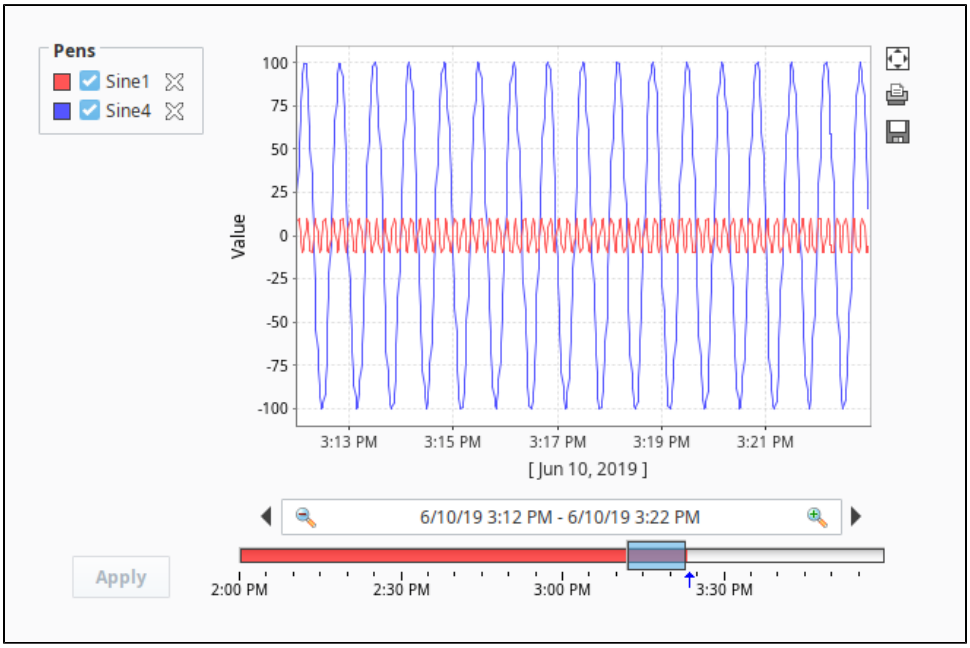
> **This section assumes that Tags and Tag History have been configured**
>
> To learn more, go to the Tag and Configuring Tag History pages.
> The examples below use OPC Tags from the Programmable Device Simulator driver, but Memory Tags can be used instead.

On this page ...

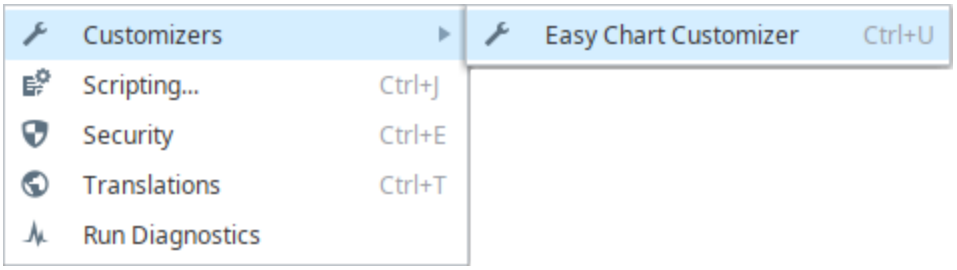
- Configuring Multiple Axes on an Easy Chart
- Hiding Pens
- Configuring an Easy Chart using the Symbol Axis

Now, let's configure multiple axes on an Easy Chart component.

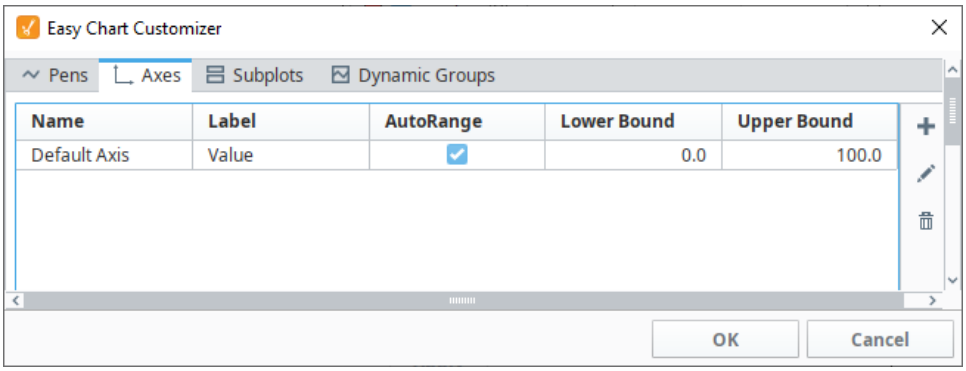
1. From the Component palette, drag an **Easy Chart** component to your workspace.
2. Next, drag your Tags over from the Tag Browser onto your Easy Chart. For the example, we used Sine 1 and Sine 4. Sine 1 and Sine 4 have completely different value ranges. Sine 1 values range between -10 and 10. Sine 4 values range between -100 and 100. Since both sines are on the same axis, it is hard to see the details of Sine 1 values because Sine 4 is throwing off the axis due to its wide range of values.



3. You have the option of putting Tags into different axes. You can do that in the Customizer of the Easy Chart component. Right click on the Easy Chart component and choose **Customizers > Easy Chart Customizer.**



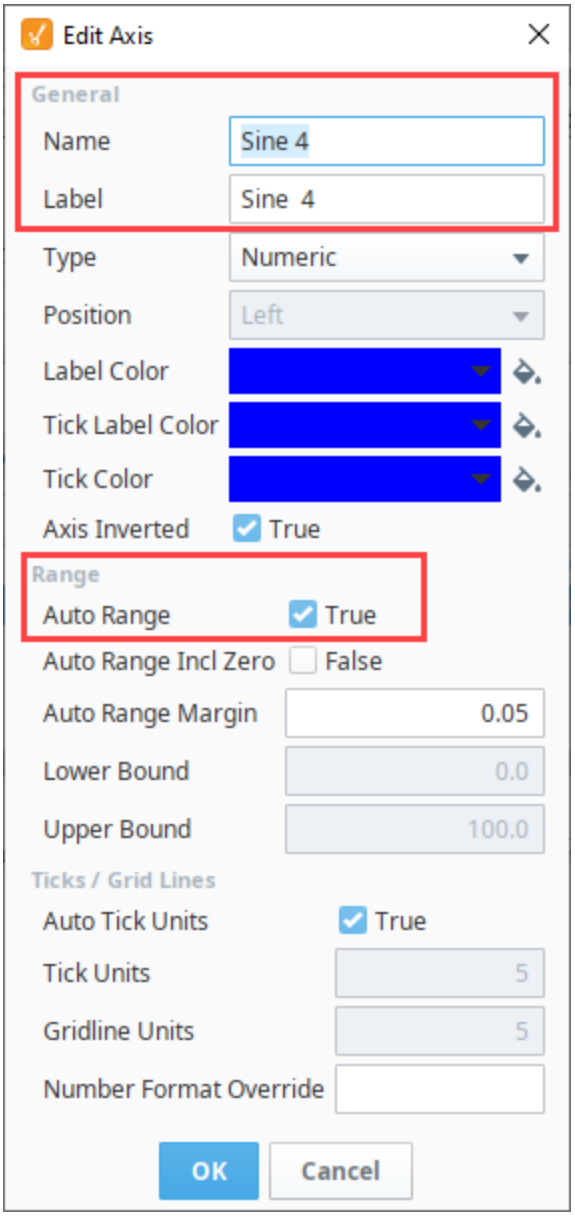
4. Click on the **Axes** tab. You'll notice that there is already one axis showing called Default Axis, which both Tags are sharing.



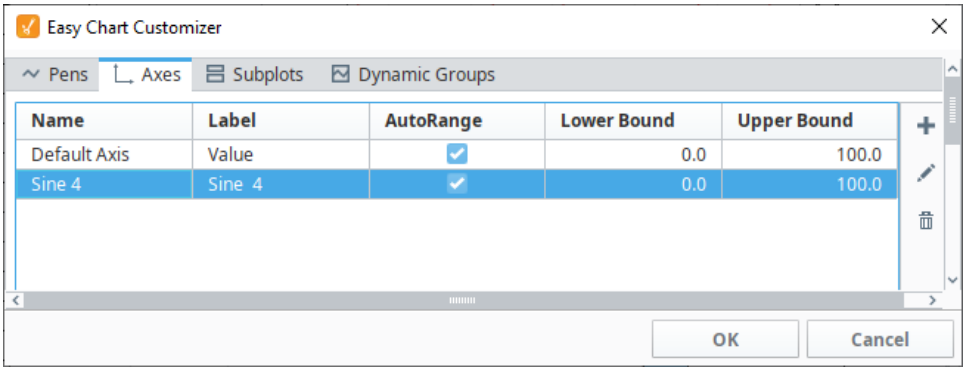
5. To add an axis, click on the **Add** icon.
6. The Edit Axis window is displayed. Assign the **Axis** a name. In this example, it's the name of the Tag that is being used.
7. Enter a **Label** name, which is a name that you want users to see on your chart.
8. Select the **Type** of axis from the dropdown: Numeric, Logarithmic or Symbol. This example uses the default, **Numeric**.
9. If desired, select the **Label**, **Tick Label** and **Tick Color** that you want to set for your axis.

The **Position** property determines which side of the chart the Axis should be drawn on. By default, this property is disabled because the Easy Chart automatically attempts to position each Axis. To manually determine the position of an Axis, locate the **Auto Axis Positioning** pro perty in the Property Editor of the Easy Chart component, and set it to **'False'**.

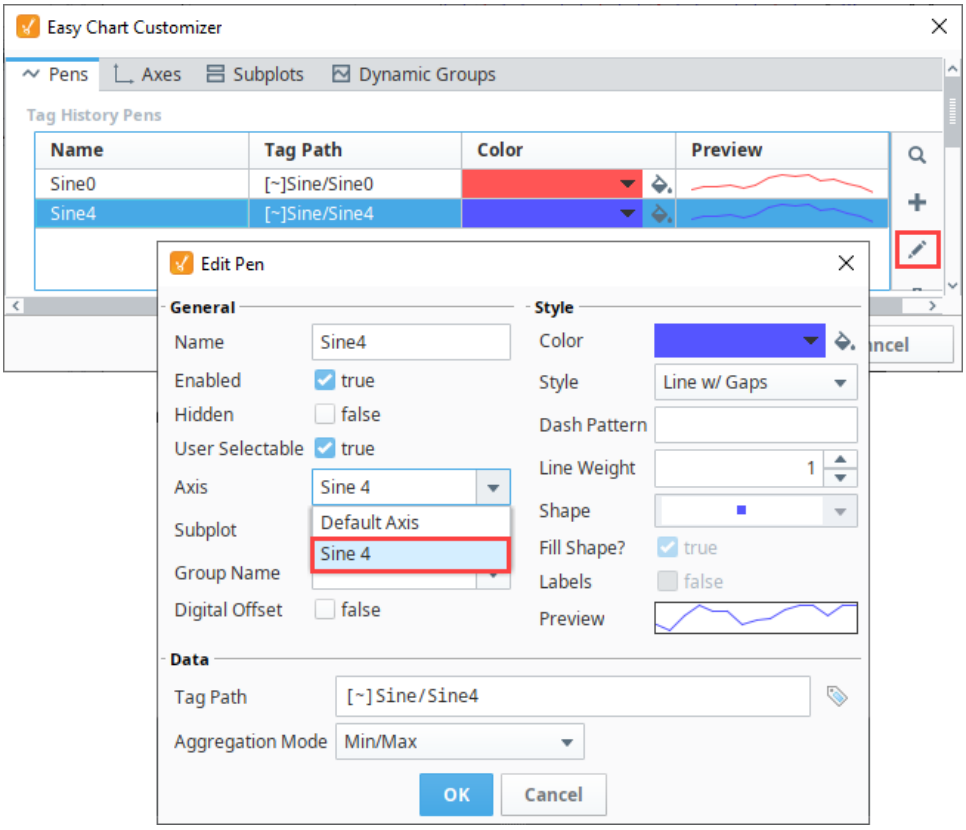
By default, the **Auto Range** is set to **'true'** and will apply padding so the pens do not draw at the top and bottom of the axis. Instead of having the Easy Chart automatically determine the range, Auto Range could be set to 'false', in which case the **Lower Bound** and **Upper Bound** properties will determine the full range of the axis.



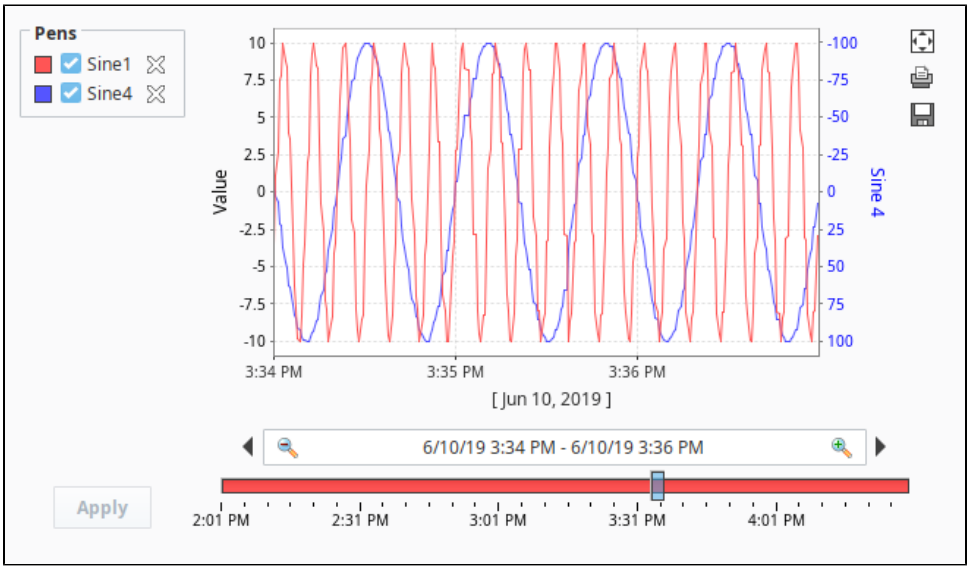
10. Now, you have two axes: Default Axis and Sine 4 axis.



11. Once a new axis has been created, you need to assign a pen to the axis. Select the **Pens** tab, select the pen row you want to change, and click the **Edit** icon . This example uses the **Sine 4** pen. In the **Axis** field, select the **Sine 4** axis from the dropdown menu, and click **OK** to save the pen.
12. Click **OK** again to close the Easy Chart Customizer.



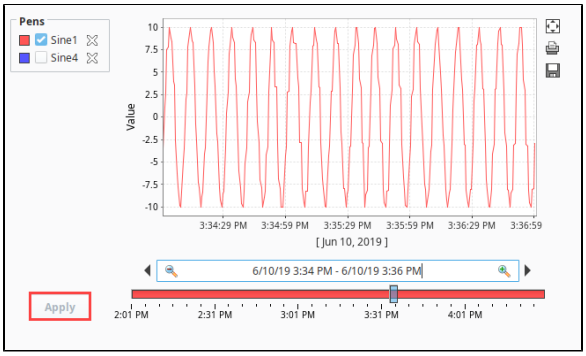
13. You'll notice that each pen is now in a different axis: Sine 1 is in the Default Axis, and Sine 4 is in the Sine 4 axis. On the right side, you can see Sine 4 axis and its values. On the left side, you can see the Default Axis (Sine 1) axis and its values. Now, you are using a different axis for each pen, and one pen is not going to throw off the values for the other pen.

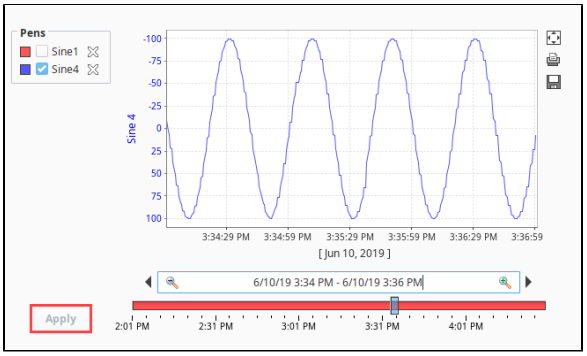


Hiding Pens

You can also hide or turn off pens so they are not displayed on the Easy Chart. To only see values for the Sine 1 axis, go to **Preview Mode**, uncheck t he **Sine 4** pen**,** and click **Apply.** To see the values for only the Sine 4 axis, check the **Sine 4** pen, uncheck the **Sine 1** pen**,** and **click Apply**. Also important to note, auto positioning on the Easy Chart will automatically move the axis should pens assigned to an axis be removed.

Sine 1 Pen **Sine 4 Pen**

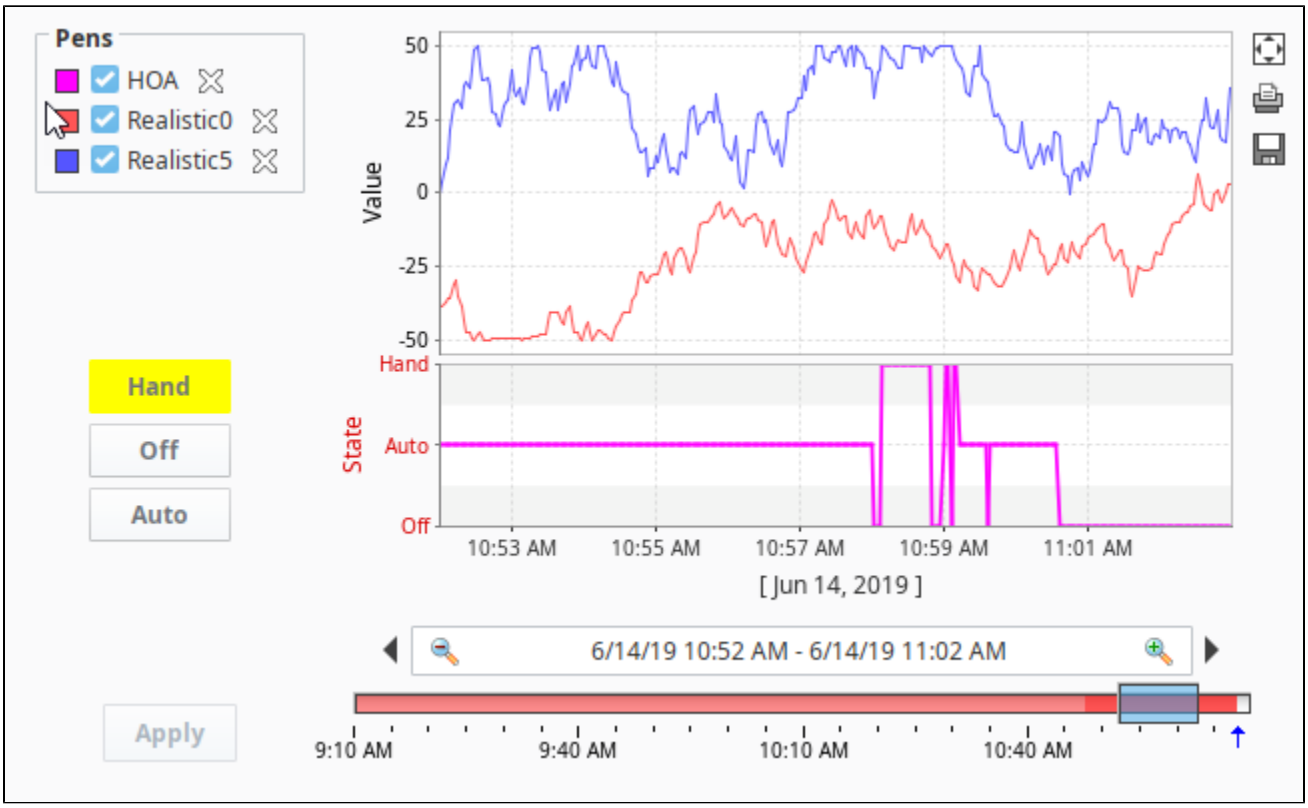




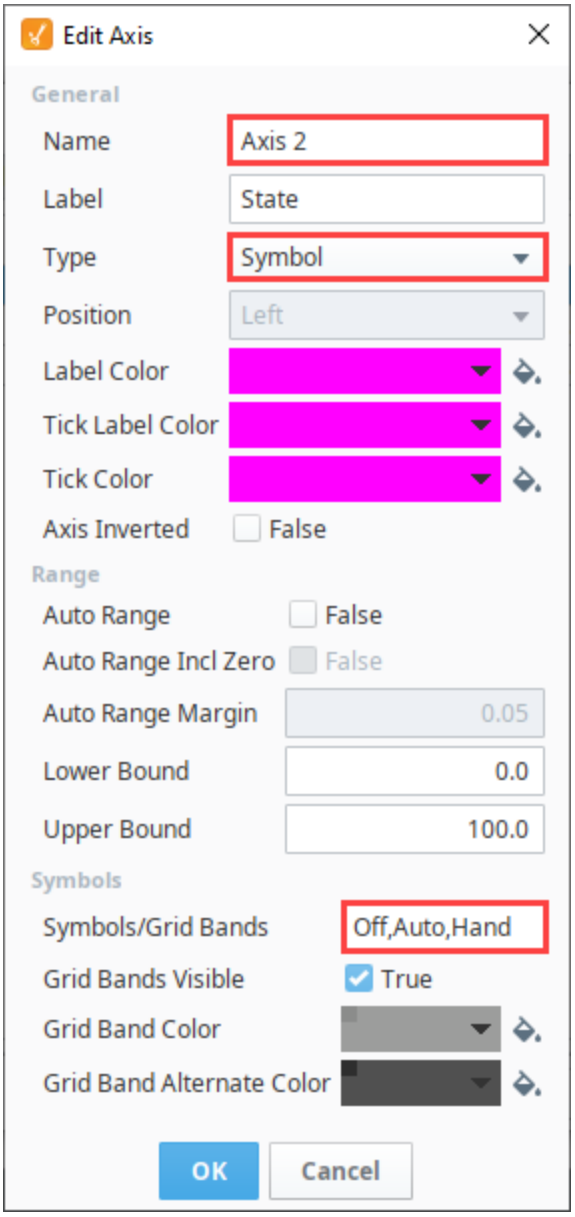
Configuring an Easy Chart using the Symbol Axis

Another feature of the Easy Chart is the use of the **Symbol Axis** type**.** Instead of showing numerical values on the axis, the Symbol Axis type can show plain text on the axis.

The second subplot in this Easy Chart uses a Multi-State Button component to demonstrate the use of the **Symbol Axis** type. The Multi-State Button component is bound to an OPC Tag, and the value of the Tag is stored in the Tag History system. Instead of showing the numerical values **'0,' '1,'** and **'2,'** you can use plain text such as **'Hand,' 'Off,'** and **'Auto.'** This is helpful to an operator who immediately knows the state of the equipment instead of having to learn what the numeric values mean.

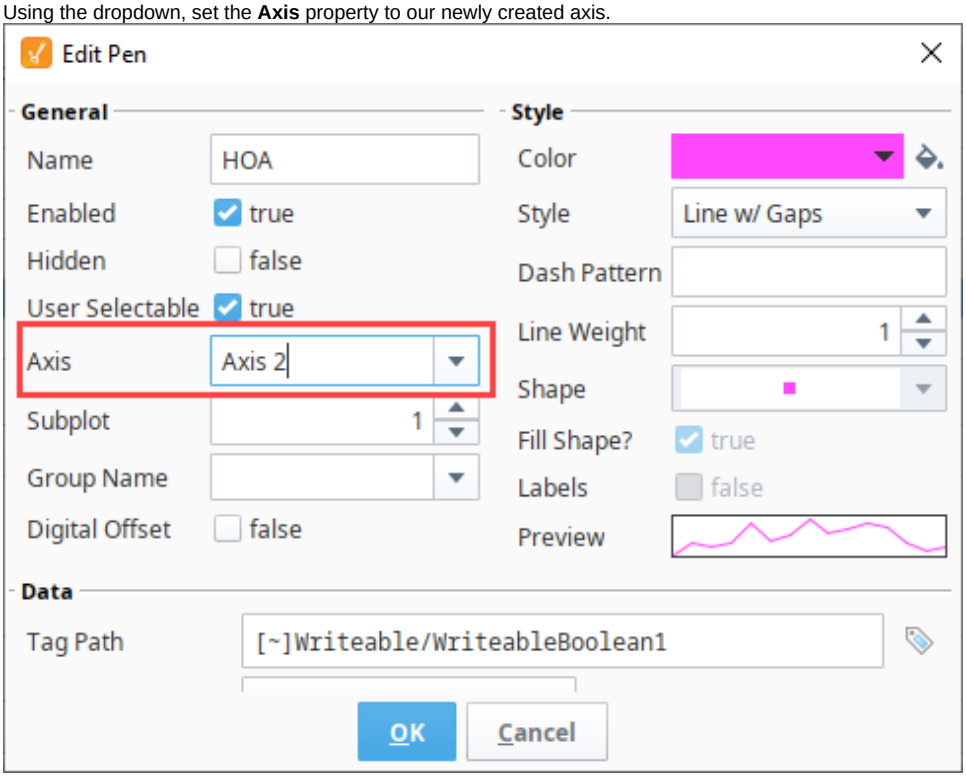


1. Click on your Easy Chart component, and select **Customizers > Easy Chart Customizer.**
2. Click on the **Axes** tab, then click the **Add** icon to add an axis.
3. Enter a **Name**, we chose 'Axis 2'. Enter a **Label**, we called 'State'.
4. In the **Type** field, select **'Symbol'** from the dropdown.
5. In the **Symbols/Grid Bands** field enter **'Auto,' 'Off,'** and **'Hand'** separated by commas, and no spaces. The order of the symbols when you type them in, will be ascending order on the axis.

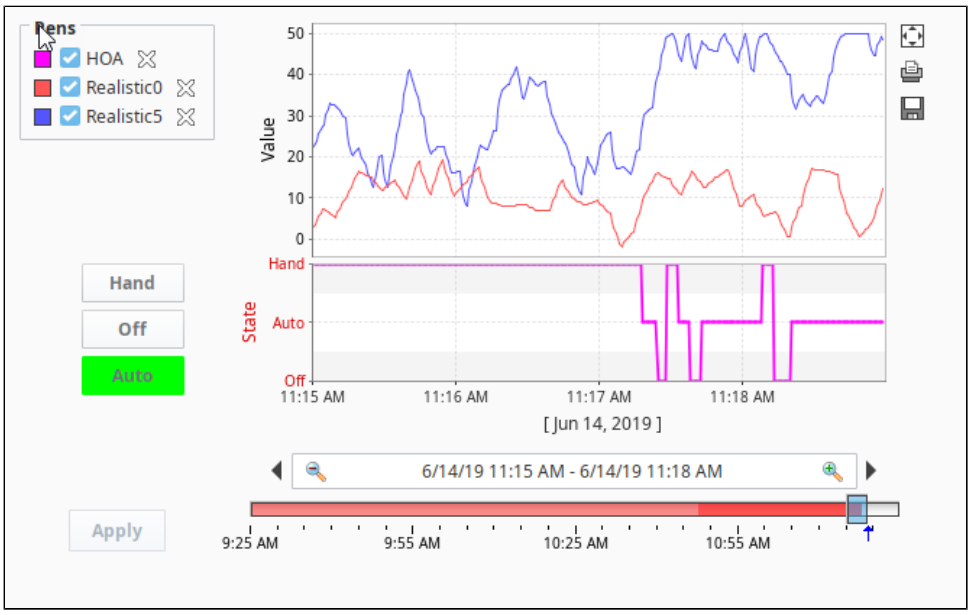


6. Click **OK** once to close the **Edit Axis** window.
7. Next we need to assign the new axis to a pen. In the Easy Chart Customizer, click on the **Pens** tab.
8. Select the Pen that should use this new axis and click **Edit** .

9. Using the dropdown, set the **Axis** property to our newly created axis.



10. Press **OK** to close the Edit Pen window, and press **OK** again to close the Tag History Customizer.
11. Put the Designer in **Preview Mode** ▶.
12. Toggle the Multi-State buttons to begin logging data to your Easy Chart. Data for the Multi-State Button component is collected in the second subplot which is using the Symbol Axis type. The first subplot is data for another Tag using the Numeric Axis type.



This Easy Chart example above shows the subplot feature of the Easy Chart component and how easy it is to break up the chart plot area into multiple distinct subplots sharing the same X axis. It is a good way to display lots of data from different Tags in one Easy Chart.

Related Topics ...

- Easy Chart Customizer
- Easy Chart - Subplots

Easy Chart - Subplots

This section assumes that Tags and Tag History have been configured

To learn more, go to the Tag and Configuring Tag History pages.
The examples below use OPC PC Tags from the Programmable Device Simulator driver, but Memory Tags can be used instead.

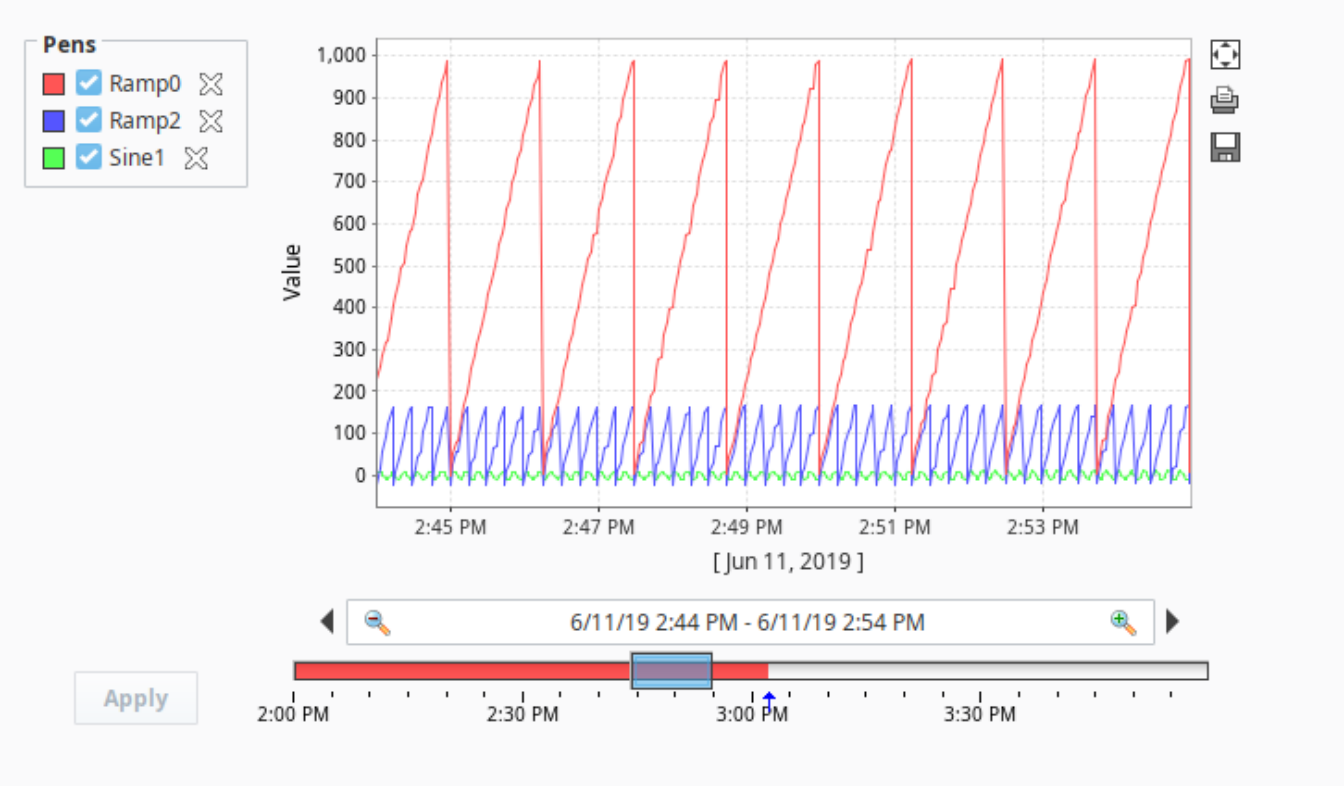
Easy Chart - Subplots

Watch the Video

Subplot Overview

The subplot feature of the Easy Chart component allows you to break up the chart plot area into multiple distinct subplots sharing the 'X' axis, but they each have their own 'Y' axis. It is a good way to display lots of data from different Tags in one Easy Chart.

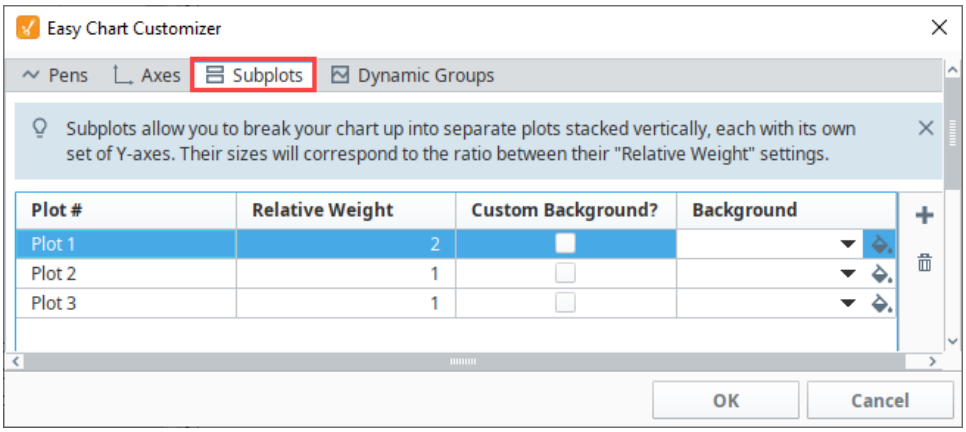
By default, the Easy Chart has one subplot which is the main white area. In this example, there are three Tags inside the chart, yet it's difficult to see the details of the data. It's possible to break up your Tags into multiple subplots which is often useful for discrete data.



Configuring Easy Chart Subplots

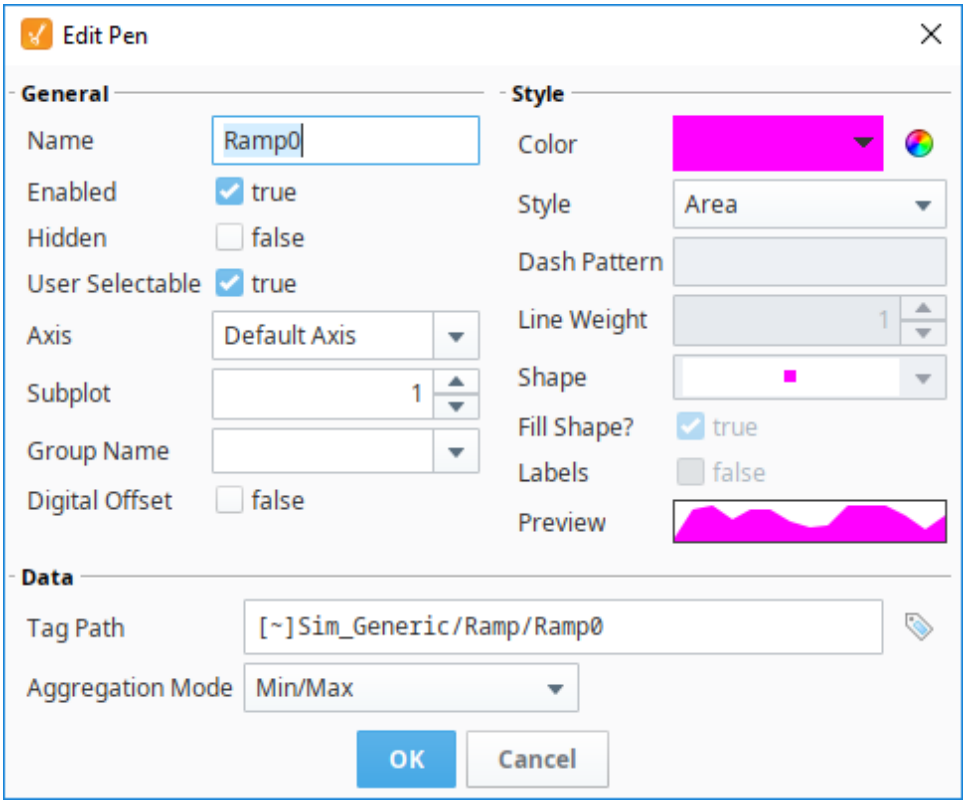
For each Tag in the Easy Chart example above, let's create its own subplot so the data is easier to view and analyze.

1. Drag an Easy Chart component onto your window.
2. Drag three Tags onto the chart. We used Ramp0, Ramp2, and Sine1 from Programmable Device Simulator.
3. Right click on the Easy Chart component and choose **Customizers > Easy Chart Customizer.** The Easy Chart Customizer window opens displaying four tabs.
4. Click on the **Subplots** tab.
5. The Subplots tab lets you add one or more subplots to the Easy Chart. Create two more subplots by clicking the **Add** **icon** two more times.
6. The size of each subplot corresponds to the ratio between their "Relative Weight" settings. By default, each subplot is assigned a weight of 1, meaning each subplot will share an equal percentage of space on the chart. In this example, we set **Subplot 1** has a weight of **'2'**, and **Subplots 2** and **3** have a weight of **'1'**. Subplot 1 is going to be 2 times larger than Subplots 2 and 3.



7. Now that we have subplots, we'll put each of the different pens into a different subplot. Click on the **Pens** tab, select the row for the Ramp0 Pen**,** and click the **Edit** icon. For this example, we chose the following settings:

    a. Set the **Subplot** to 1.
    b. Set the **Color** to pink.
    c. Set the **Style** to Area.
    d. Click **OK**.

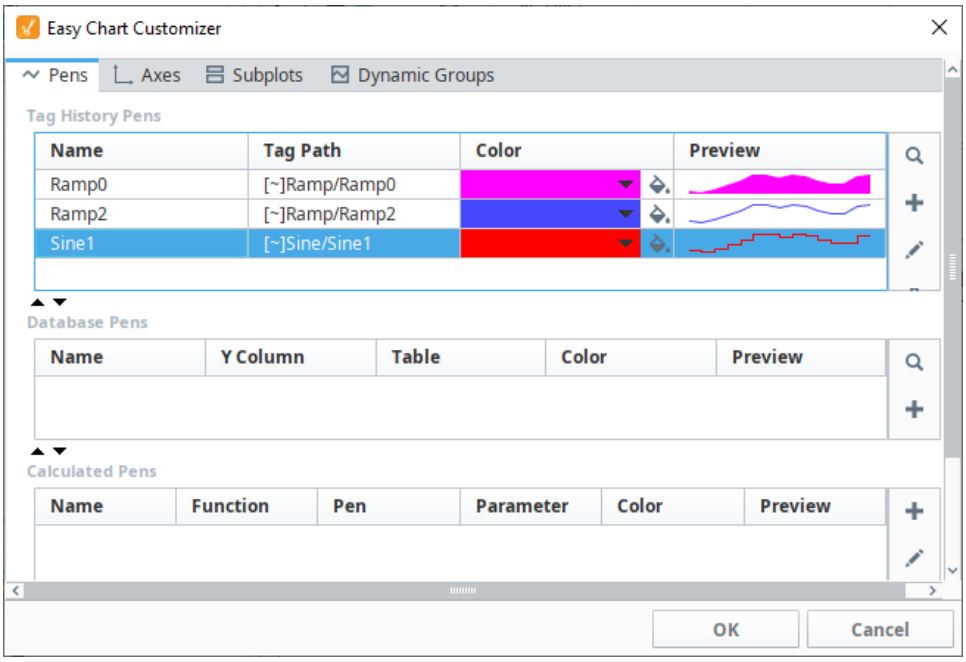


8. For the Ramp2 pen, we selected the following:

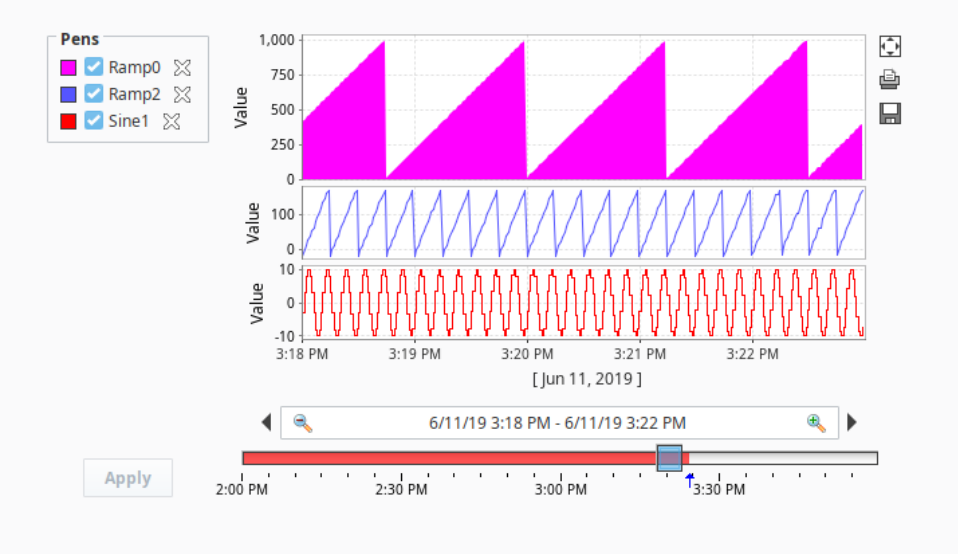
    - **Subplot:** 2
    - **Color:** Blue
    - **Style:** Line w/Gaps

9. For the Sine1 pen, we selected the following:

    - **Subplot:** 3
    - **Color:** Red
    - **Style:** Digital

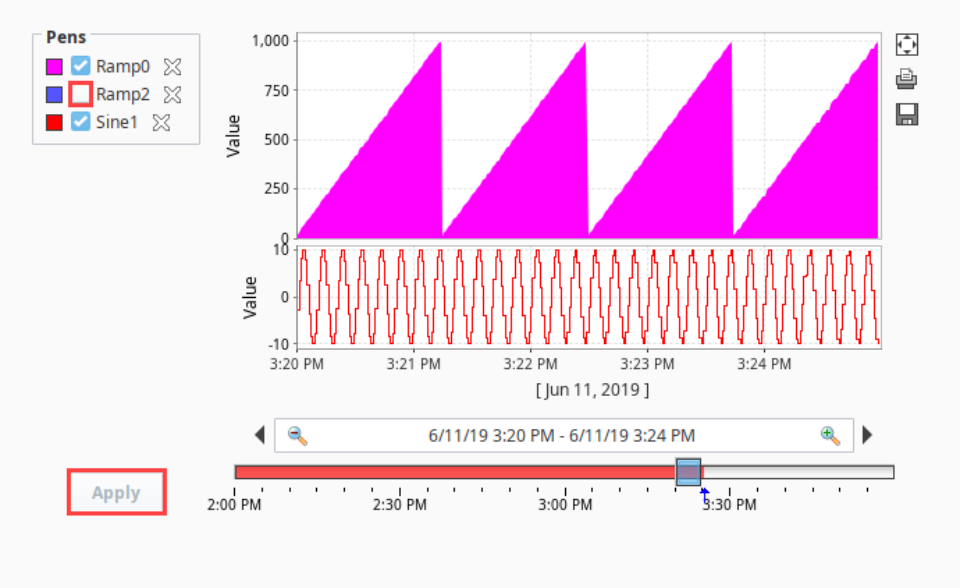


10. Click **OK**. Now, you have three distinct subplots on one Easy Chart. You are not limited to the number of subplots on one Easy Chart.



11. You can be selective about what subplots you want to view. Go to **Preview Mode,** uncheck the Pens you don't want to see, and click **Apply**.

12. Notice how the Ramp2 pen is unchecked and is no longer displayed on the Easy Chart. To add the Ramp2 pen back to the Easy Chart, check the Ramp2 pen, and click **Apply**. The Easy Chart only displays subplots that have active Tags.



Related Topics ...

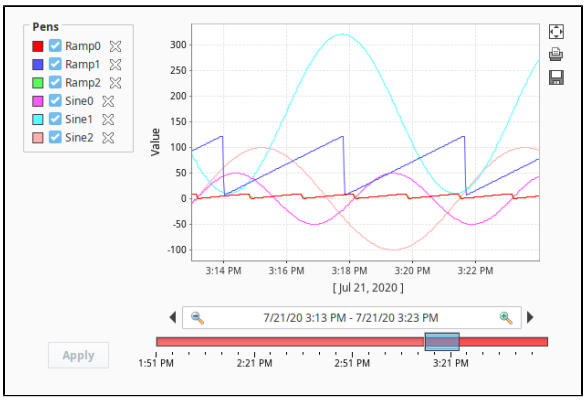
- Easy Chart Customizer
- Easy Chart - Pen Names and Groups
- Easy Chart - Pen Renderer

Easy Chart - Pen Names and Groups

> **This section assumes that Tags and Tag History have been configured**
>
> To learn more, go to the Tag and Configuring Tag History pages.
> The examples below use OPC Tags from the Programmable Device Simulator driver, but Memory Tags can be used instead.

Pen Names and Group Overview

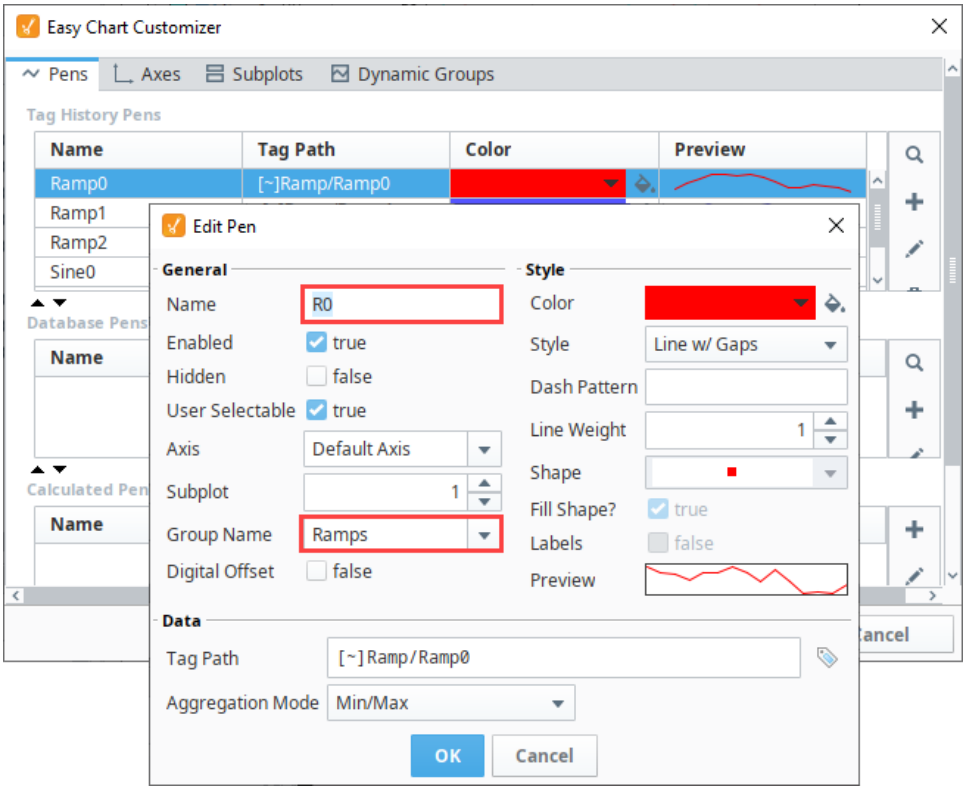
You can organize pens on the Easy Chart by creating custom names and groups for each pen. By default, when you drag Tags from the Tag Browser on to the Easy Chart component, the pen name is the same as the Tag name and organized into a single group called **'Pens.'** One of the great things about pens is you can change pen names and organize pens into different groups making it easier for the operator to quickly analyze the data.



Configuring Pen Names and Groups

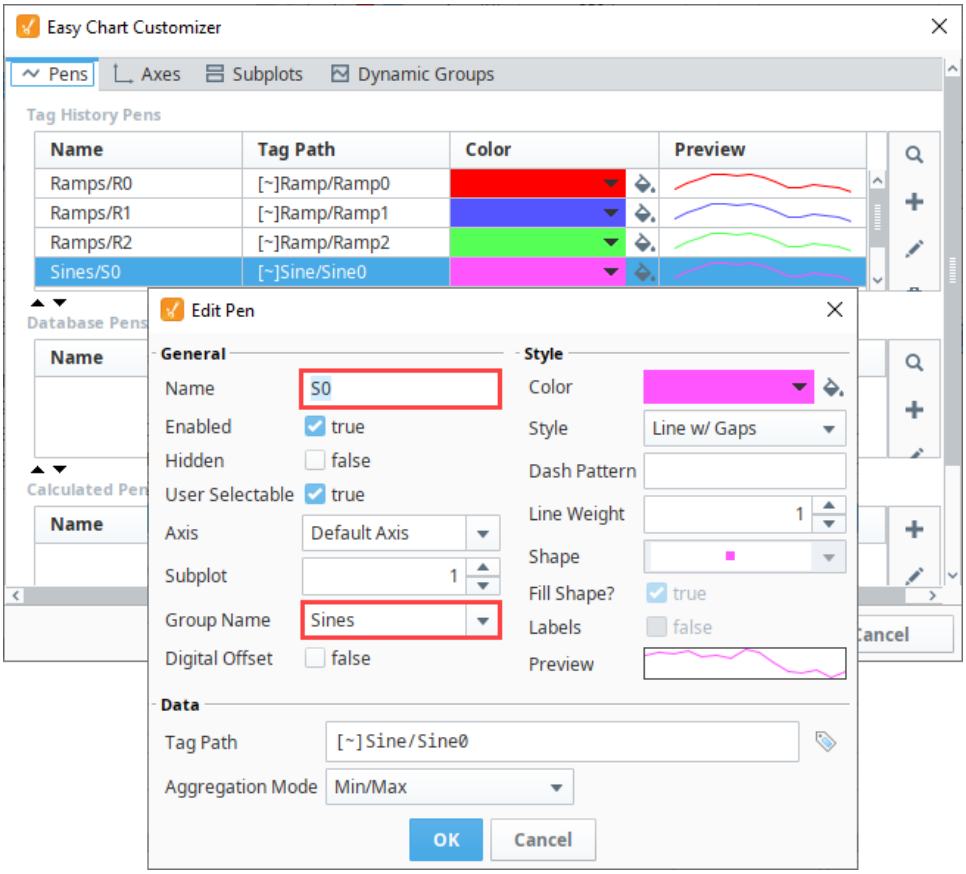
For each Tag on the Easy Chart, let's create unique pen names and organize each pen into a group using the Easy Chart Customizer.

1. Right click on the Easy Chart component, and choose **Customizers > Easy Chart Customizer.**
2. On the Pens tab, we'll edit each of the individual pens and give them a different name as well as put them inside a different group.
    a. Select the first pen row, click the **Edit**  **icon,** and rename the first Tag from **'Ramp0'** to **'R0'** as shown in the following example.
    b. At the bottom of the Edit Pen window, next to **Group Name**, create a new group called **'Ramps.'**
    c. If other groups exist, you can select one from the dropdown list or enter your own. In this example, this is the first group to be created, so simply type **'Ramps,'** and **press OK**.
3. Repeat this step for each Ramp pen assigning each pen a new name.

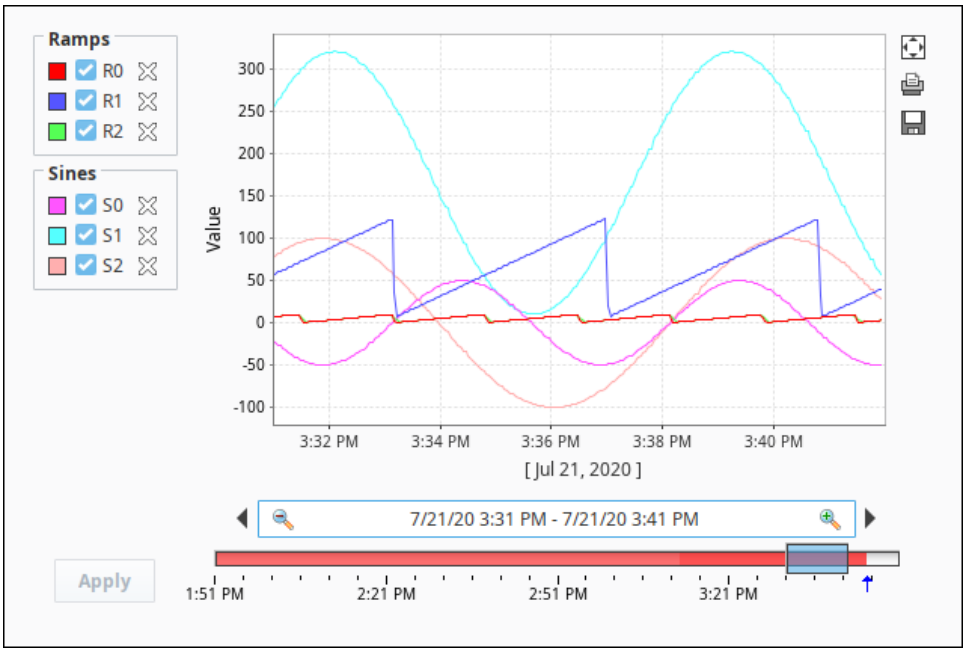


4. Next, let's keep the Sine pen names the same as the Tag name, but add them to a group.

    a. Select the **Sine0** pen row, and click the .
    b. Enter a new group name called **'Sines**'.
    c. Click **OK.**
    d. Repeat this steps a and c. to add each Sine pen to the Sines group.



5. Once you have all your pens configured, click **OK**. Operators will see the Pen and Group names organized into two legends on the Easy Chart. You can update the pen and group names on this view by double clicking on any of the fields. It's a faster way to edit this information then having to go to the Edit icon for each pen.



Tip for Viewing Specific Pen Values

To see specific Pen values on the Easy Chart, uncheck the Pen and click the Apply button. To see all Pen values, make sure all Pens are checked, and hit the Apply button.

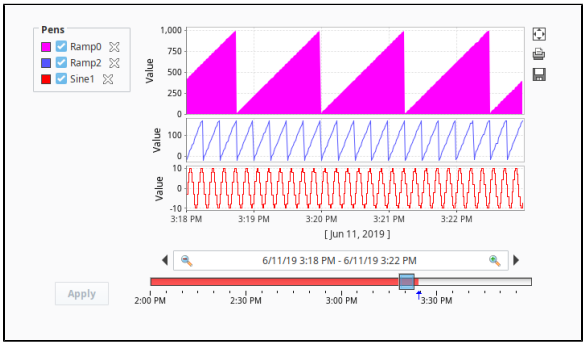
Related Topics ...

- Easy Chart Customizer
- Easy Chart - Pen Renderer

Easy Chart - Pen Renderer

Customizing the Pen Renderer

You can customize the renderer of each pen on the Easy Chart to change its style, shape, weight and color. When adding Tags to an Easy Chart component, the default renderer or style of each pen is a simple line, but it can be customized for each pen.



Easy Chart - Pen Renderer

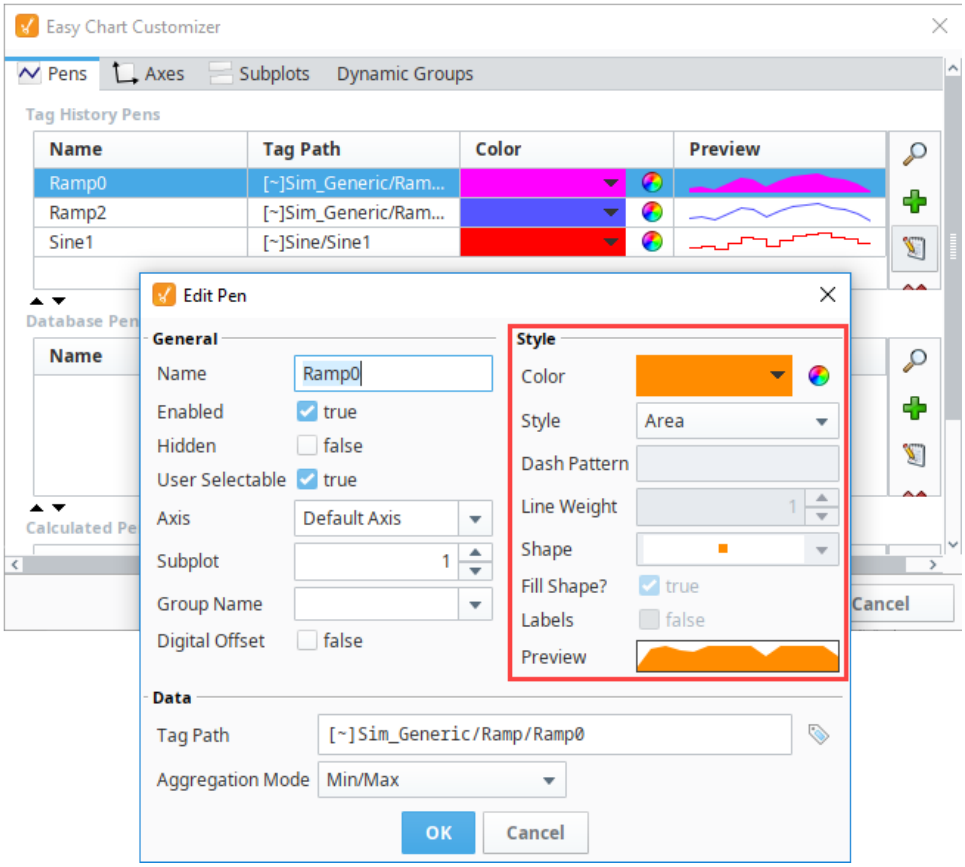
Watch the Video

In this example, we will use the Pen Renderer to customize several pens. What's nice is the Easy Chart Customizer allows you to previewthe style, shape, weight and color. If you don't like it, you can easily change it.

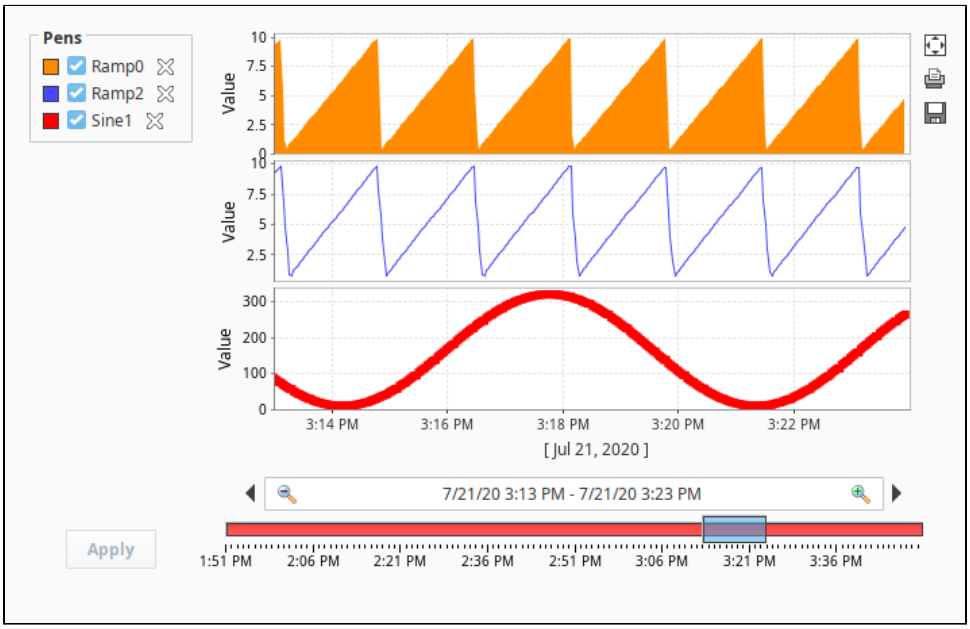
1. Click on the Easy Chart component, and scroll down to **Customizers >** **Easy Chart Customizer**.

2. From the **Pens** tab, select the pen row, and click the **Edit** **icon**. On the right side of the Edit Pen window under Style, you can change the line color, style, weight, and shape. The line style, by default, is **'Line with Gaps,'** which means that if you lose communication to the PLC or don't have data for a particular time period, you will see a gap. You can change the Style from 'Line w/ Gaps' to any other style type listed in the dropdown that fits your needs and preferences.

3. Edit each style property. If you don't like the result in the Preview window, select another style. You can edit it as many times as you like. When you're finished, press **OK.**



4. Repeat steps 2 and 3 to customize the style for all the pens you want to change. When you're finished, **press OK.**
5. The pens on the Easy Chart now reflect your style changes.



Related Topics ...

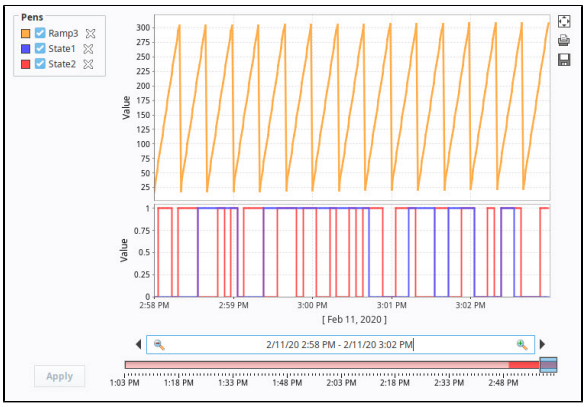
- Easy Chart Customizer

- Easy Chart - Digital Offset

Easy Chart - Digital Offset

Digital pens often share the same subplot on the **Easy Chart** component. When you have multiple digital pens on the same Easy Chart subplot, it's hard to see what the values are of each pen because they may overlap each other. There is a digital offset pen setting that can be set which prevents the values from overlapping and enables them to be seen better in the subplot.

The following example shows two digital pens on the same subplot: State1 and State2. The values are a little difficult to see because they are on top of each other.



On this page ...

- Adding a Digital Offset

INDUCTIVE UNIVERSITY

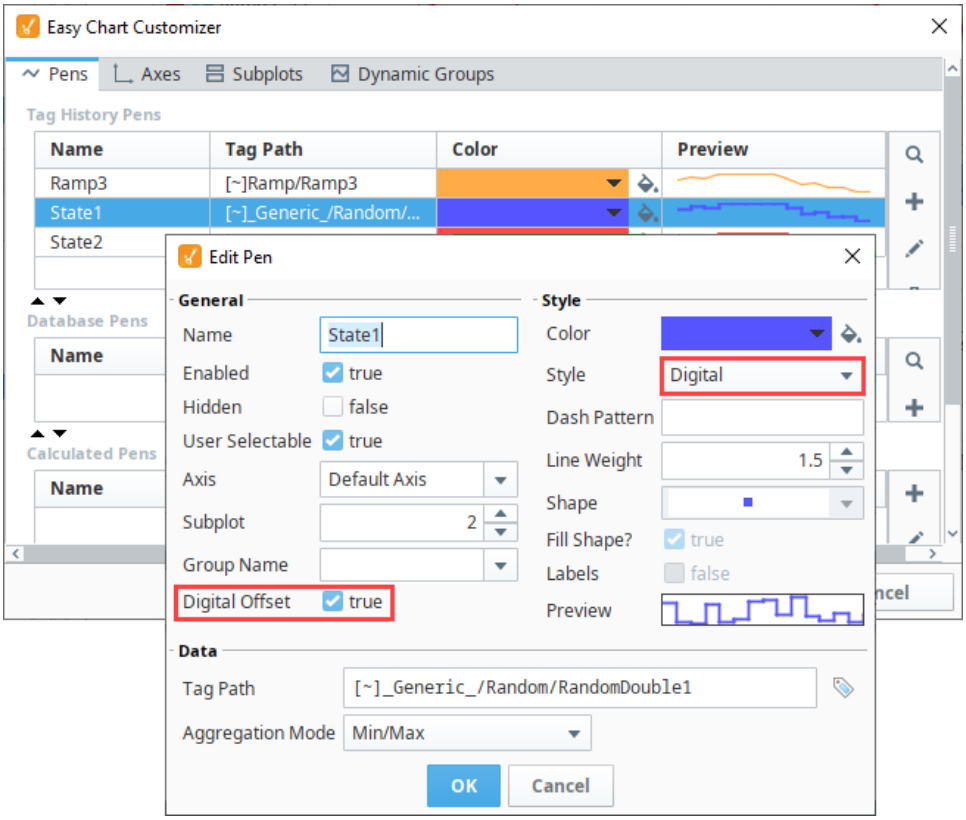
Easy Chart - Digital Offset

Watch the Video

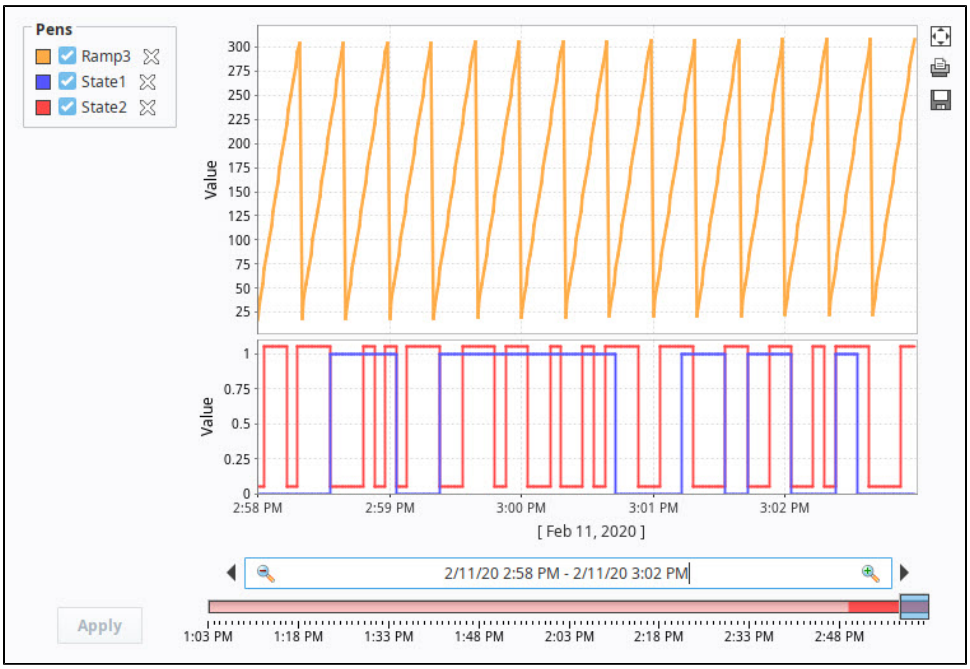
Adding a Digital Offset

In this example, we will apply a digital offset on the pen renderer so you can see the values better.

1. Right click on the Easy Chart component and choose **Customizers >** **Easy Chart Customizer**.
2. On the Pens tab, select the **State1** row pen, and click the **Edit**  icon.
3. Set the **Style** is set to **'Digital'**.
4. Set the **Digital Offset** to **'true'.** Click **OK** to save the changes to the pen.



5. Repeat steps 2-4 for the **State2** pen.
6. When you're finished editing your pens, click **OK** to return to the Easy Chart. You will see a little offset in the values between the State pens in the 2nd subplot so they don't overlap each other, making it a lot easier for the operator to read the digital values.



Related Topics ...

- Easy Chart Customizer
- Easy Chart - Calculated Pens

Easy Chart - Calculated Pens

Calculated Pens

This section assumes that Tags and Tag History have been configured
To learn more, go to the Tag and Configuring Tag History pages.
The examples below use OPC Tags, but Memory Tags could be used instead.

Calculated pens display a value that is dynamically calculated based on another pen. This can be used to calculate certain values for a pen and graph them alongside the original pen values, allowing you to gain valuable insight into your data. There are many unique calculations that can be used, with some of them containing unique customization. Almost all of the Calculated pens require a driving pen, which is a tag or database pen that you have already set up.

Note: You cannot bind the Calculated Pen values inside the Easy Chart Customizer. To bind the function values, use the Cell Update Binding

On this page ...

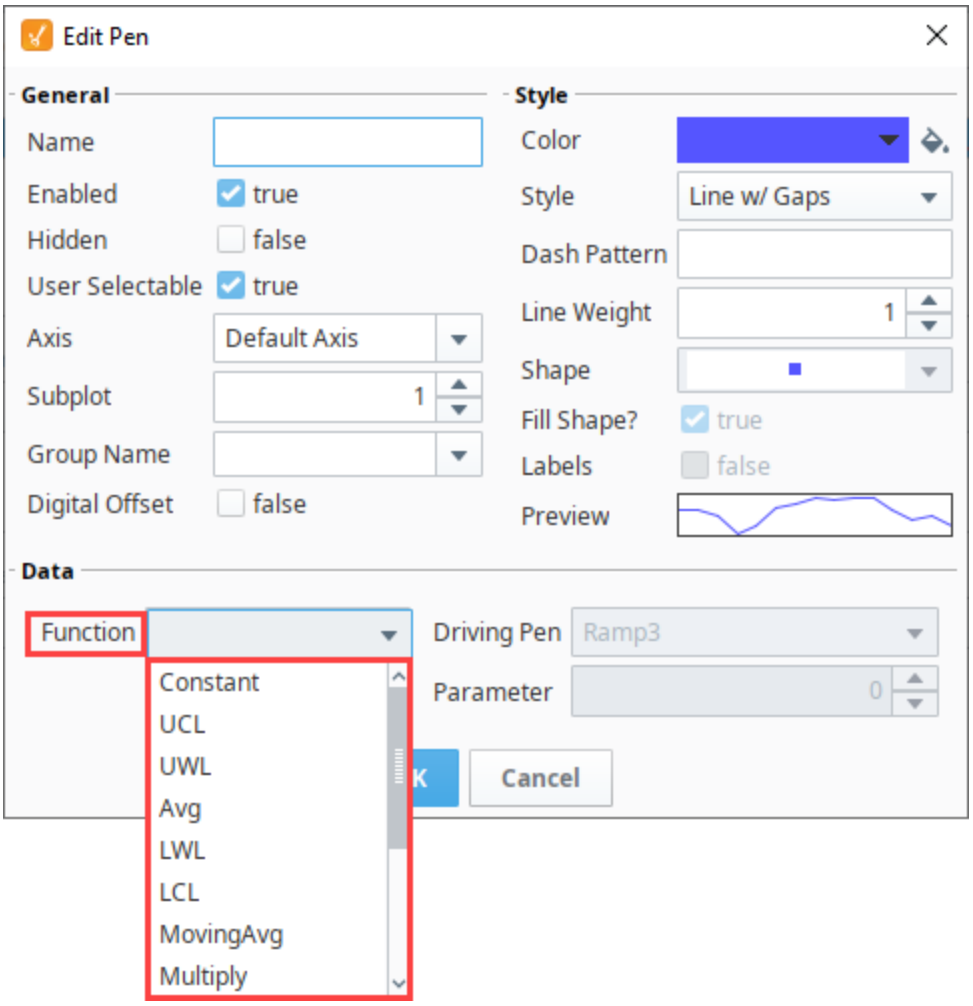
- Calculated Pens
- Calculated Pen Functions
- Configuring Calculated Pens
  - Hide Driving Pens

Easy Chart - Calculated Pens

Watch the Video

Calculated Pen Functions

There are a variety of functions that can be used to calculate the pen value. The pen functions are located on the Edit Pen screen in the Function dropdown list. The table below defines the available functions.



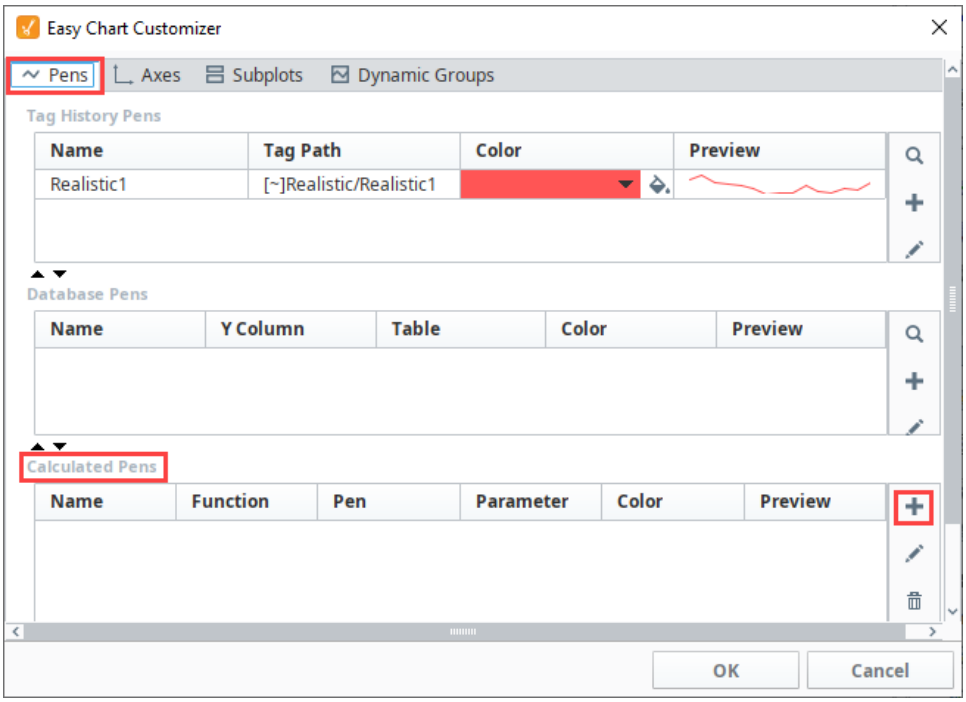
Function	Display Name	Description	Extra Properties
Constant	Constant	A constant value on the chart.	Constant Value - The constant value of the pen
Upper Control Limit	UCL	The upper control limit of the driving pen, which is three standard deviations above the mean of the Driving Pen.	
Upper Warning Limit	UWL	The upper warning limit of the driving pen, which is two standard deviations above the mean of the Driving Pen.	
Average	Avg	The average of the driving pen.	
Lower Warning Limit	LWL	The lower warning limit of the driving pen, which is two standard deviations below the mean of the Driving Pen.	
Lower Control Limit	LCL	The lower control limit of the driving pen, which is three standard deviations below the mean of the Driving Pen.	
Moving Average	MovingAvg	A series of averages based on subsets of the driving pen. The subsets are determined by the window size.	Window Size - The size of the moving average window, specified as a multiplier of the chart's date range.
Multiply Pen	Multiply	Multiply each data point of the driving pen by a factor.	Factor - The factor that each data point of the driving pen is multiplied by.
Minimum Value	Min	The minimum value of the driving pen.	
Maximum	Max	The maximum value of the driving pen.	

Value			
Running Sum	RunningSum	A running sum or running total of the driving pen.	
Sum	Sum	The sum of two different driving pens.	Secondary Pen - The second driving pen.
Difference	Difference	The difference of two separate driving pens.	Secondary Pen - The second driving pen.
Linear Regression	LinearRegression	Will create a linear regression line for the driving pen.	

Configuring Calculated Pens

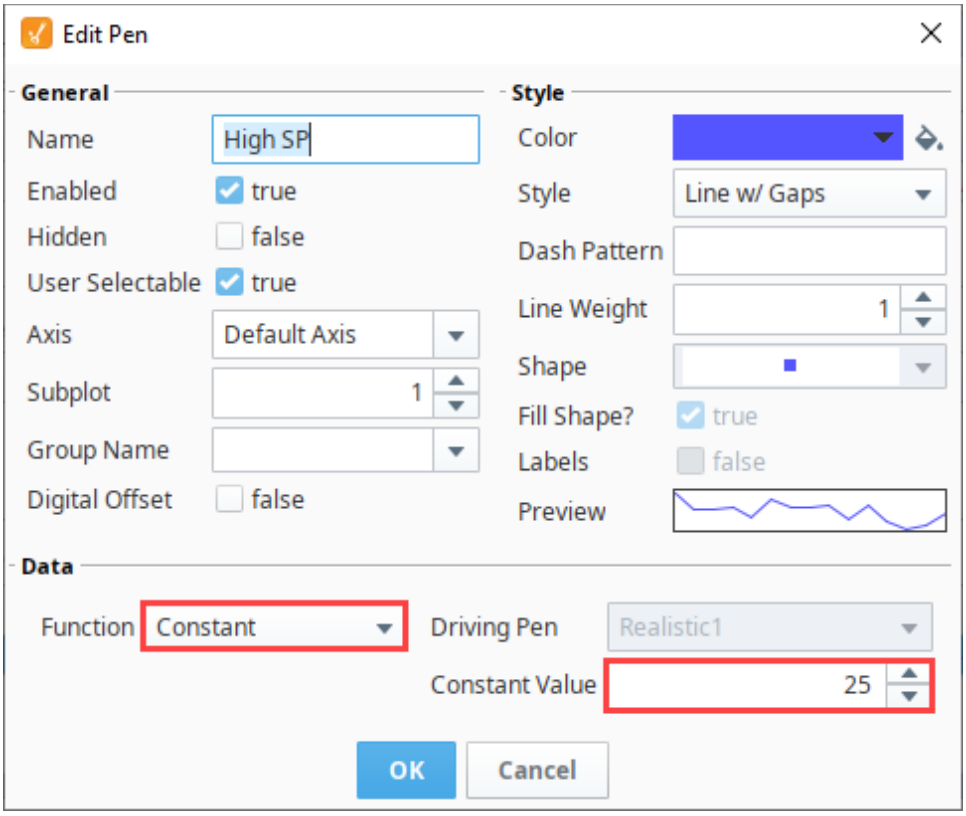
In this example, we will configure the following calculated pens on our Easy Chart: Constant, Moving Average, Upper Control Limit (UCL), and Lower Control Limit (LCL).

1. Drag an Easy Chart component onto your Designer window.
2. Drag a Tag onto the chart. In this example, we are using a **'Realistic1'** tag from Programmable Device Simulator.
3. Right click on the Easy Chart component and choose **Customizers >** Easy Chart Customizer.
4. The Pens tab will open and you'll notice a Calculated Pens Table at the bottom of the screen. Click the **Add** icon.

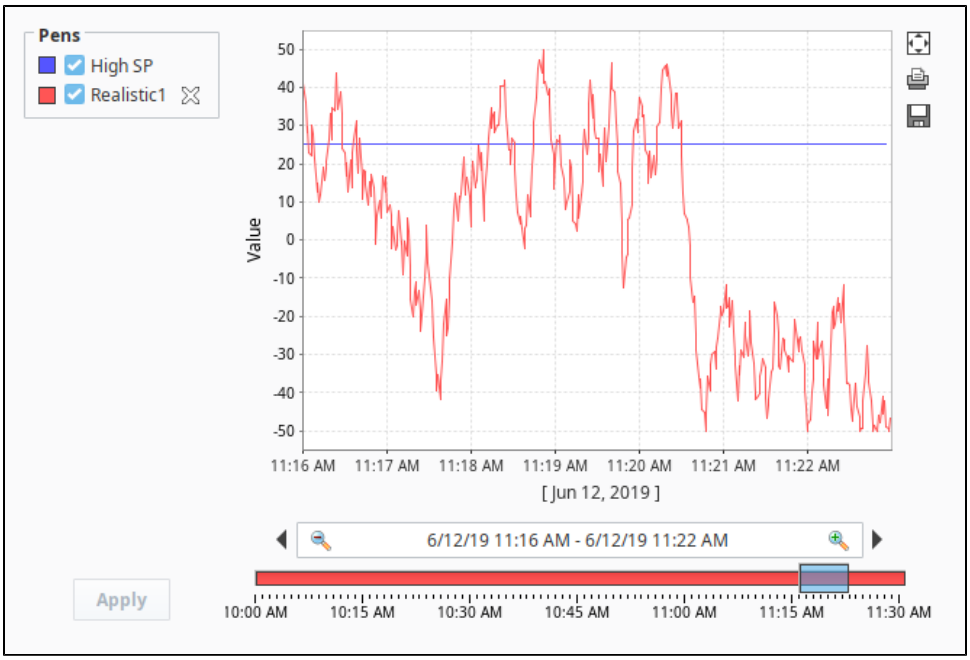


Note: Calculated pens are just like other pens, so you can specify the style, color, axis and subplot in the Edit Pen window.

5. This will open the Edit Pen window. Set the calculated pen options as follows:
   - **Name:** High SP
   - **Driving Pen:** Realistic1 (Since we only have one tag on our Easy Chart, the Realistic1 tag is the default).
   - **Function field:** Constant
   - **Constant Value:** 25
6. Click **OK**.

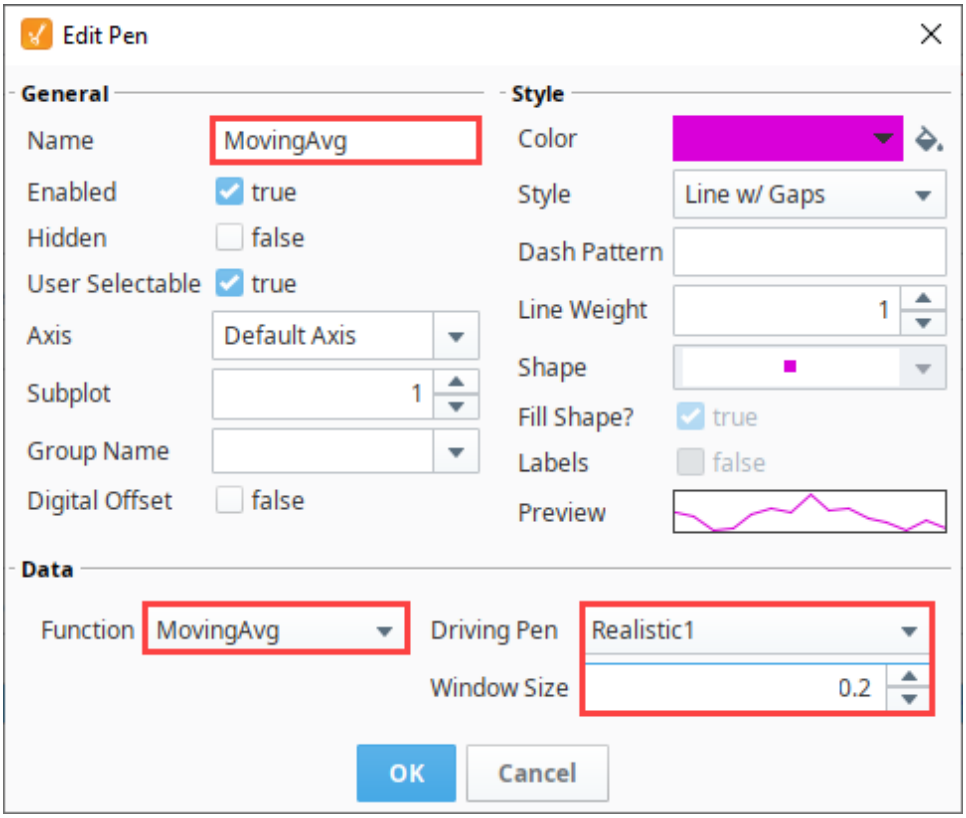


7. Click **OK** again to save your pen options. Now, view your **High SP** pen on your Easy Chart. The High SP value of 25 is represented on your Easy Chart by a blue horizontal line.

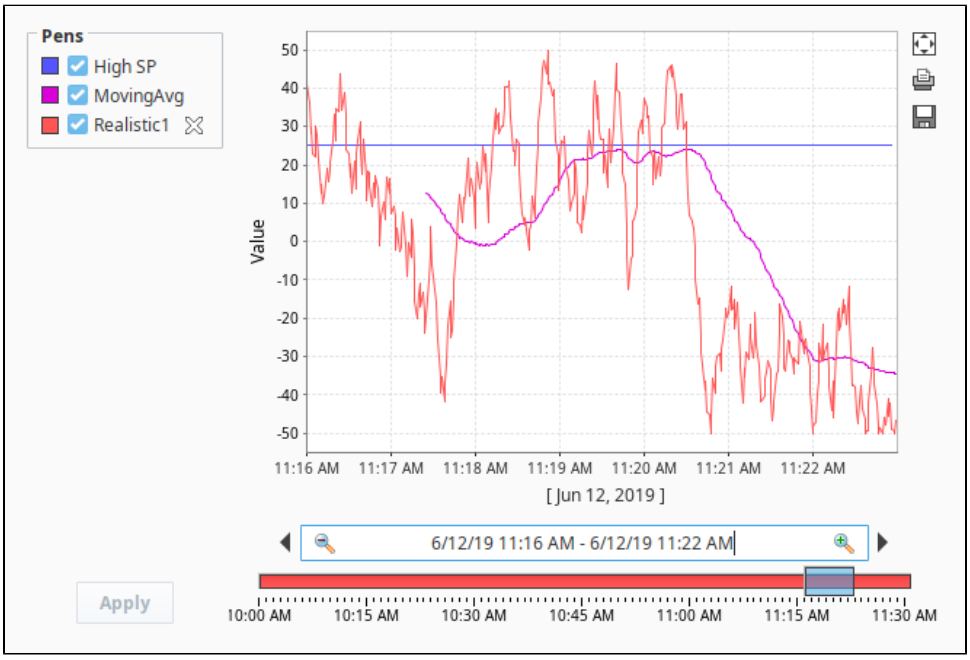


8. Next, let's add a second calculated pen. Double click on the Easy Chart to open the Style Customizer. On the **Calculated Pens Table** at the bottom of the screen. Click the **Add** icon.
9. On the Edit Pen window, set the calculated pen options as follows:
   - **Name**: MovingAvg
   - **Function field:** MovingAvg
   - **Driving Pen:** Realistic1
   - **Window Size:** .2

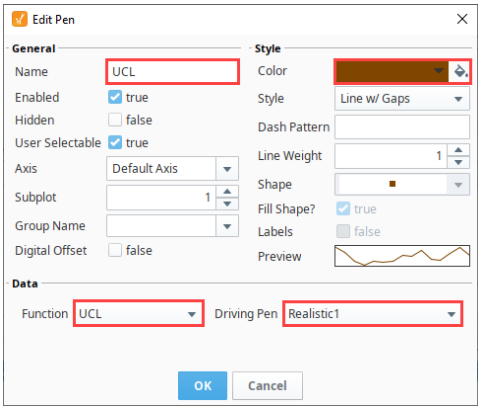
> **Note:** If you have more Tags that you dragged on to your Easy Chart from the Tag Browser, you'll have more pens to choose from in the Driving Pen dropdown list.

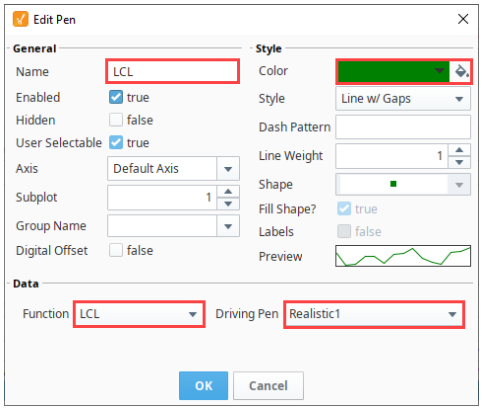


10. Click **OK**. The chart now displays a running average.

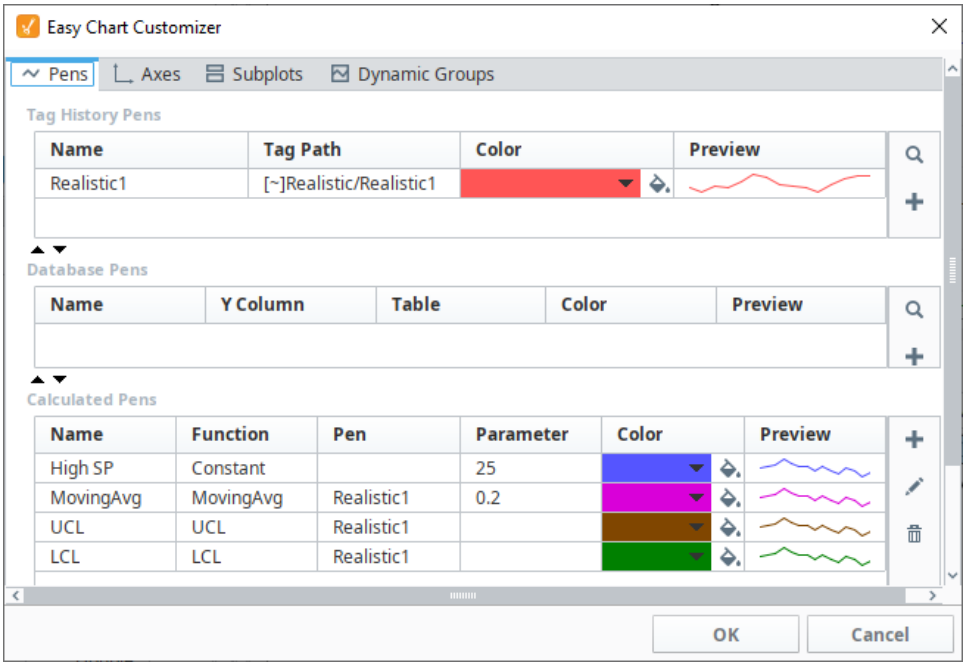


11. Lets add two more calculated pens: one for Upper Control Limit (**UCL**) and another for Lower Control Limit (**LCL**), and set the Driving Property to **'Realistic1.'** Select a pen color to change the default color.

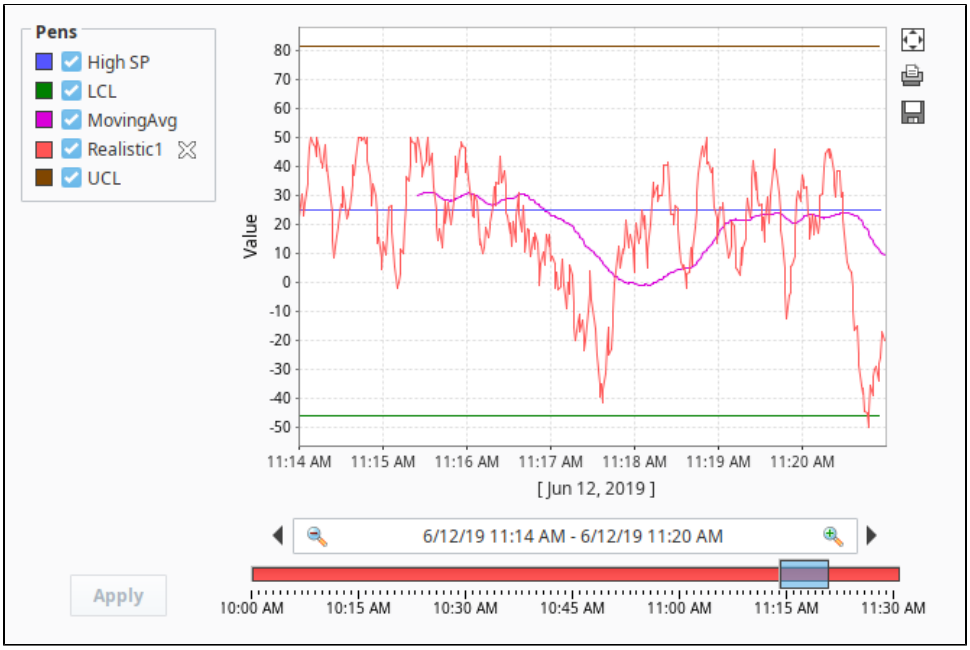




12. Once you add all your calculated pens, you'll see all your pens in the Calculated Pens section at bottom of the window. Press **OK.**

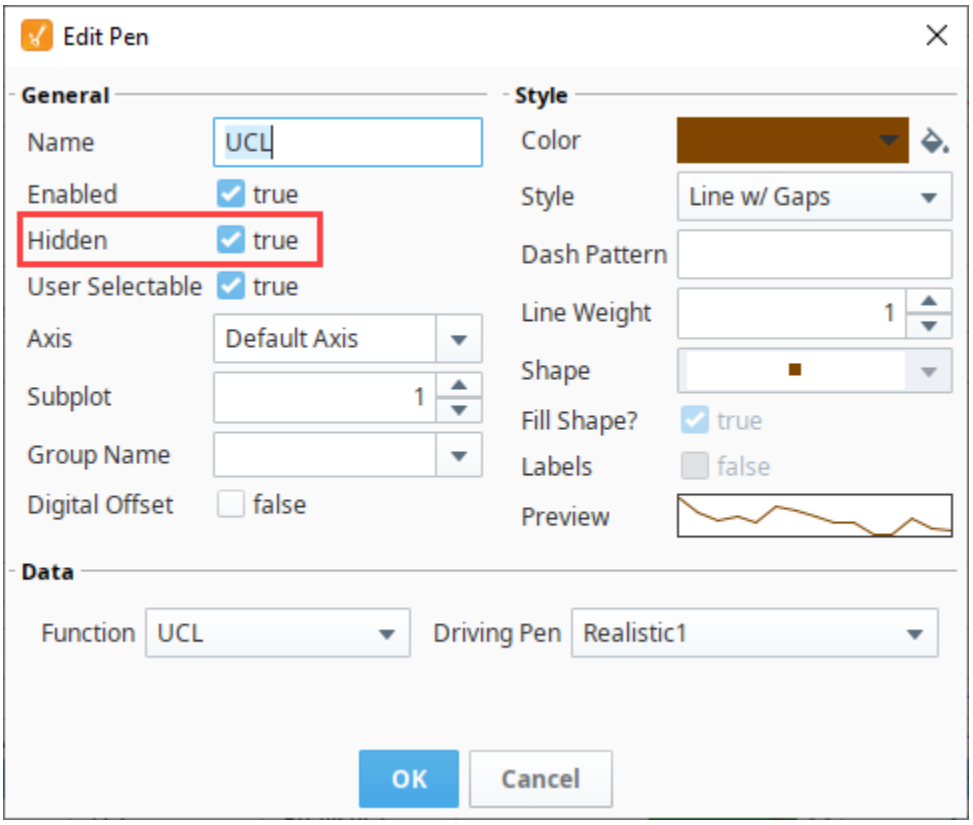


13. Now, all your calculations are displayed on your Easy Chart. The blue pen is the Constant, the pink pen is the Moving Average, and UCL and LCL are brown and green respectively.



Hide Driving Pens

Once you have your Calculated Pens created, you'll notice that they disappear if you disable the pen driving them. If you want to remove the Driving Pen but leave the Calculated Pens, set the **Hidden** property of the Driving Pen to **'true'**.



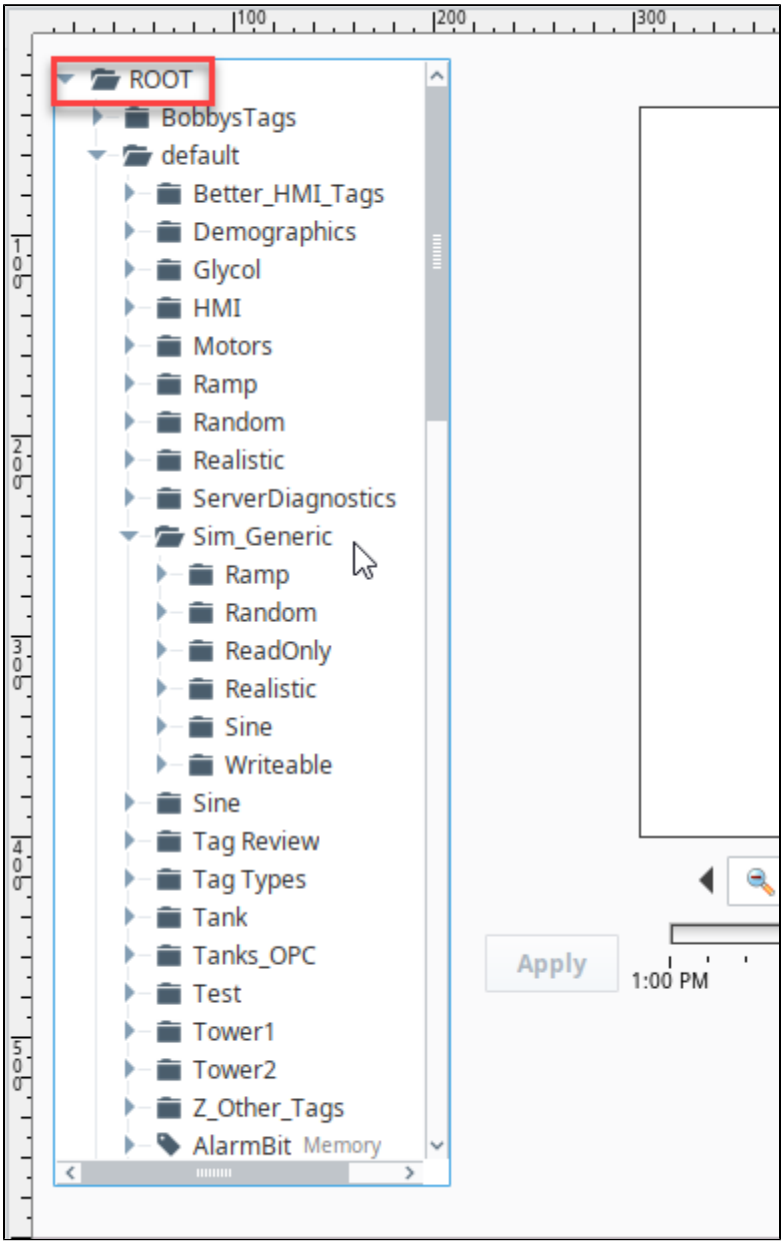
Related Topics ...

- Easy Chart Customizer
- Using the Tag Browse Tree for Charting

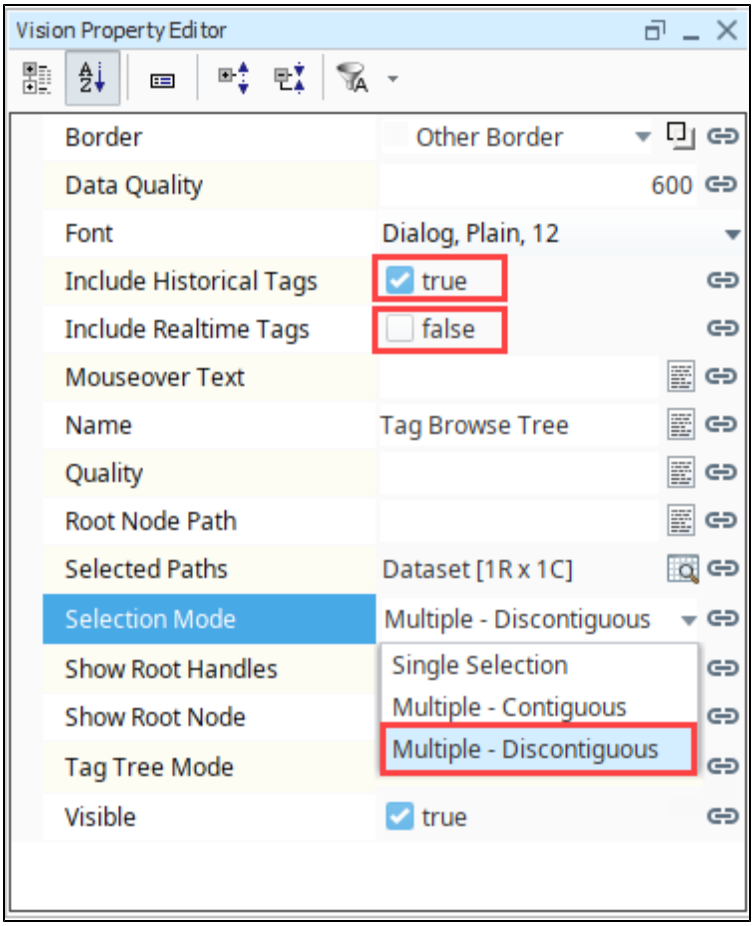
Using the Tag Browse Tree for Charting

In the Designer, you can use a special built-in component called the Tag Browse Tree to create ad hoc charts where you can pick and choose which pens you want to put on an Easy Chart. This same functionality is also available for an operator in the Runtime.

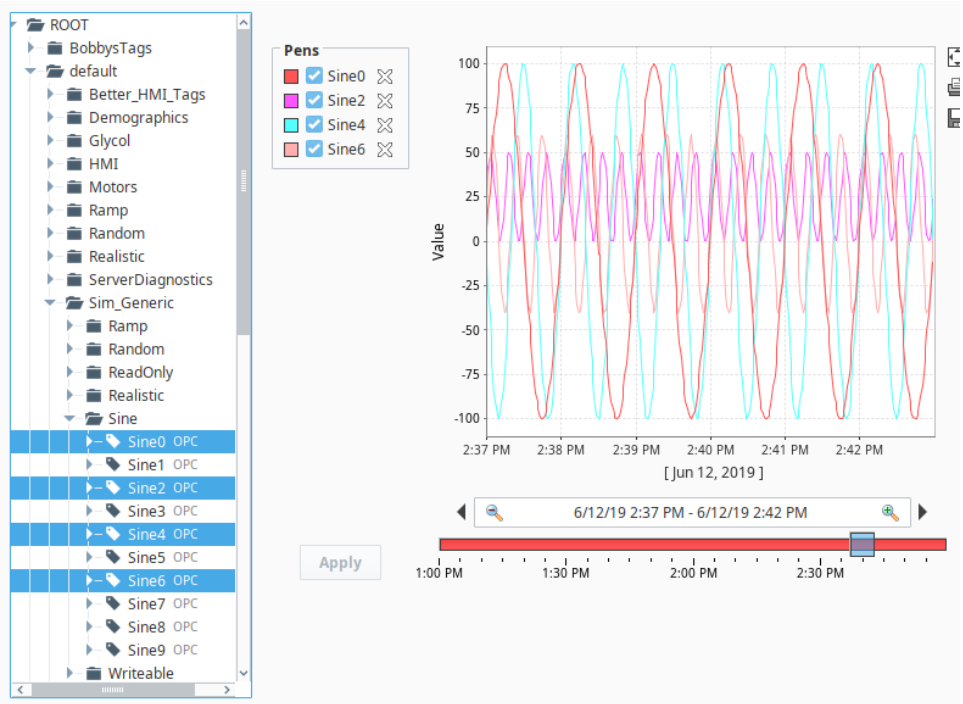
1. In Designer, drag a **Tag Browse Tree** component and **Easy Chart** component from the component palette to your workspace.
2. Put the Designer in **Preview Mode** ▶.
3. Expand the Tag folders to see the Tags in your system. By default, the Tag Browser Tree component shows you all Tags even those Tags that are not logged in Historian.



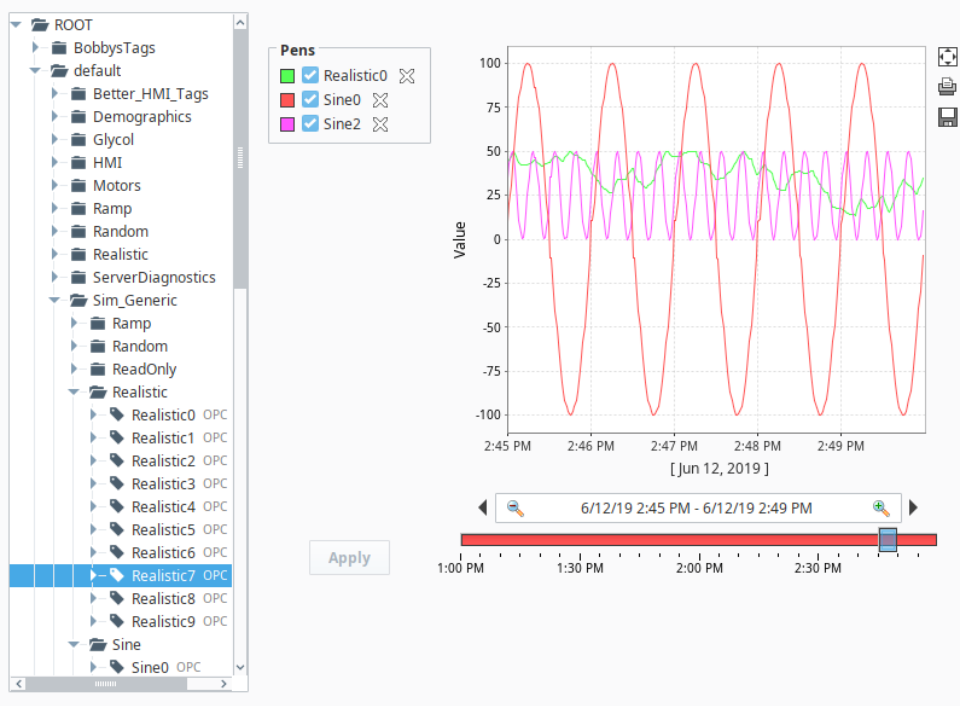
4. Put the Designer back in **Design mode** ■.
5. Select the Tag Browse Tree component.
6. In the **Property Editor**, set the following properties:
    a. Set **Include Realtime Tags** to false.
    b. Set **Include Historical Tags** to true.
    c. Set the **Selection Mode** to Multiple - Discontiguous.



7. **Save** your project.
8. Launch a Vision Project or put the Designer in **Preview Mode** ▶.
9. Drag some tags over onto the chart. We chose Sine0, Sine2, Sine4, and Sine6. Note that because we previously set selection mode to Multiple - Discontiguous, we can choose several Tags using **Shift-Click**. The Tags don't have to be contiguous within the Tag browser.



10. You can click the **Delete** icon to the right of any of the pens to remove a Tag from the chart. You can also remove all pens and go back to an empty chart, and pick and choose which of the Tags you want to drag on to the chart.



Notes: Things to keep in mind when working with Ad Hoc Charting

- When working in the Designer, whatever pens you have on your Easy Chart when you saved your project, the same pens will also be displayed on the chart when the client is opened.
- You may have multiple axes set up for your Easy Chart, but when dragging Tags from the Tag Browse Tree component to an Easy Chart, there is no way for the user to set which axis to use. Because of this limitation, any Tag that is added in this way will attempt to match their Engineering Units property to an axis on the chart. If no match is found, the default axis will be used.

Related Topics ...

- Indirect Easy Chart

Indirect Easy Chart

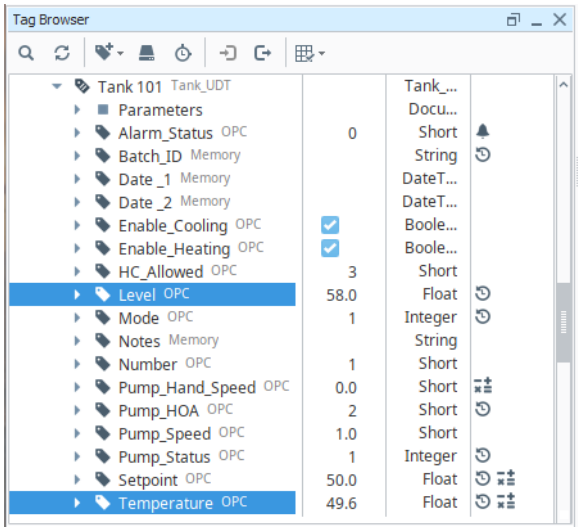
Configuring an Indirect Easy Chart

> **This section assumes that Tags and Tag History have been configured**
>
> To learn more, go to the Tag and Configuring Tag History pages.
> The examples below use OPC Tags, but Memory Tags can be used instead.

On this page ...

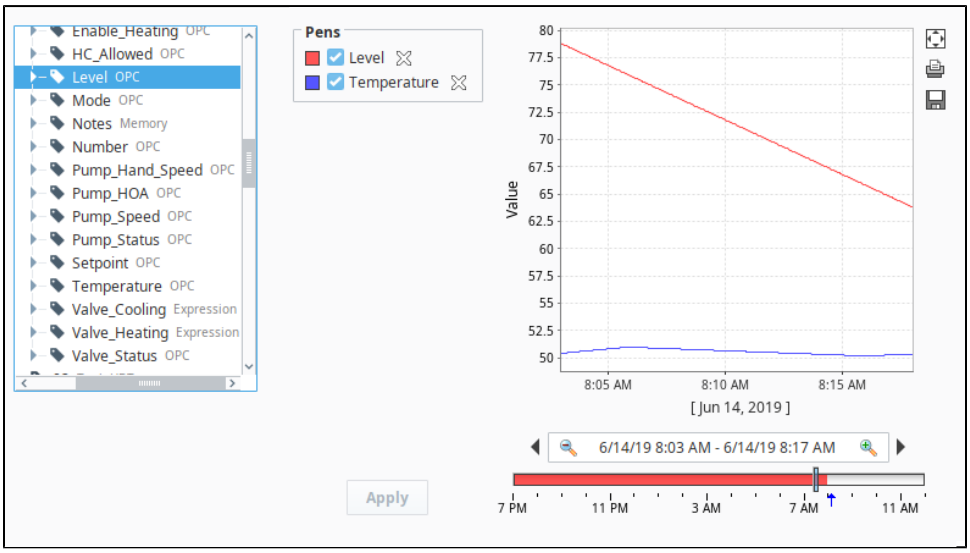
- Configuring an Indirect Easy Chart

It is possible for the Easy Chart component to be indirect and point to a set of Historical Tags based on any parameter using the Cell Update Binding type. In this example, suppose you have a small tank farm consisting of Tank 101 through Tank 109. Every tank is identical and uses the same Tags. Each tank has a **Level** Tag and **Temperature** Tag which are set to log to the Historian.



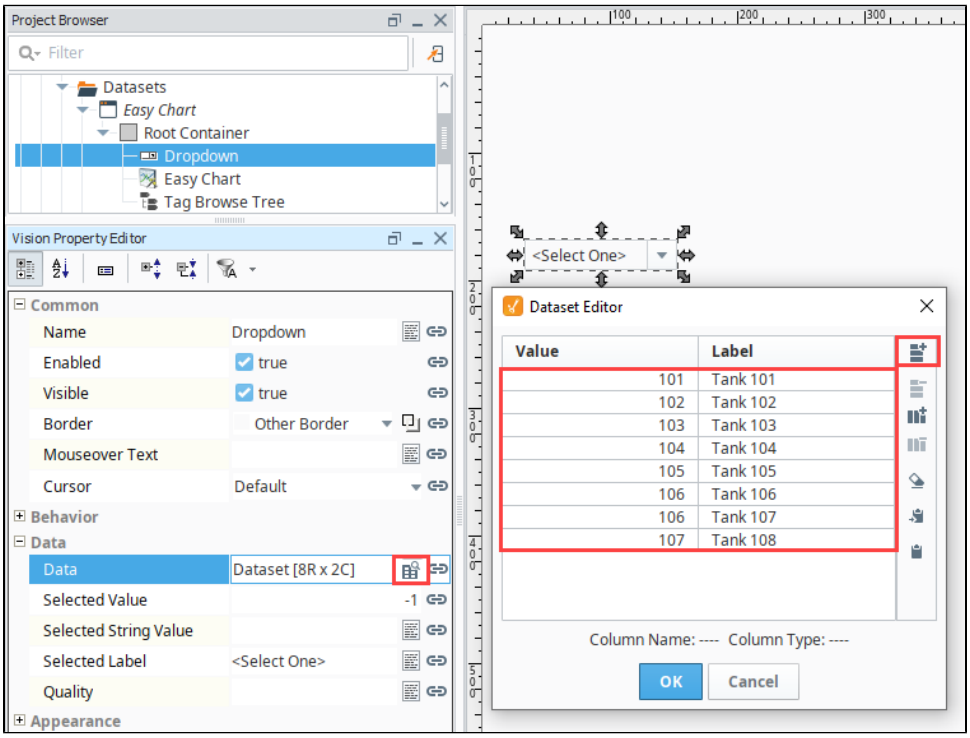
Using an Indirect Easy Chart is a good way to see the history of these Tags for each tank in the tank farm. An convenient way to set this up is using a Dropdown List component where an operator can select various tanks to see a Tank's Tag history. This example will show you how to configure a cell update binding on an Easy Chart to be indirect and point to a set of Historical Tags.

1. In the Designer, drag a **Tag Browser Tree** and an **Easy Chart** component into your workspace.
2. Put the Designer in **Preview Mode** ▶.
3. Navigate to the Tank Tags in the Tag Browser Tree. Under Tank 101, drag the **'Level'** and **'Temperature'** Tags from the Tag Browse tree onto the Easy Chart.

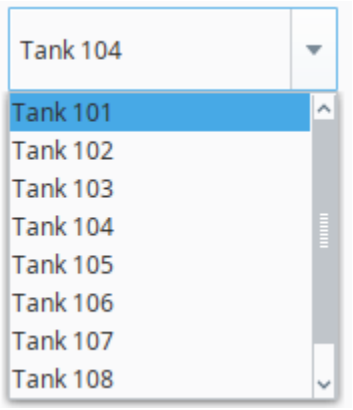


4. Put the Designer back in **Design Mode** ■.

5. Drag a **Dropdown** component to your workspace. Using a Dropdown list will make it convenient for the operator to select a Tank and see the history of the **Level** and **Temperature** Tags for that tank.
6. With the Dropdown component selected, click on the **Dataset Viewer** icon to the right of the **Data** property in the Property Editor.
7. Click the **Add Row** icon for as many Tanks that you have in your Tank Farm. Then enter the **Value** and **Label** for each of the Tanks.
8. Click **OK**.

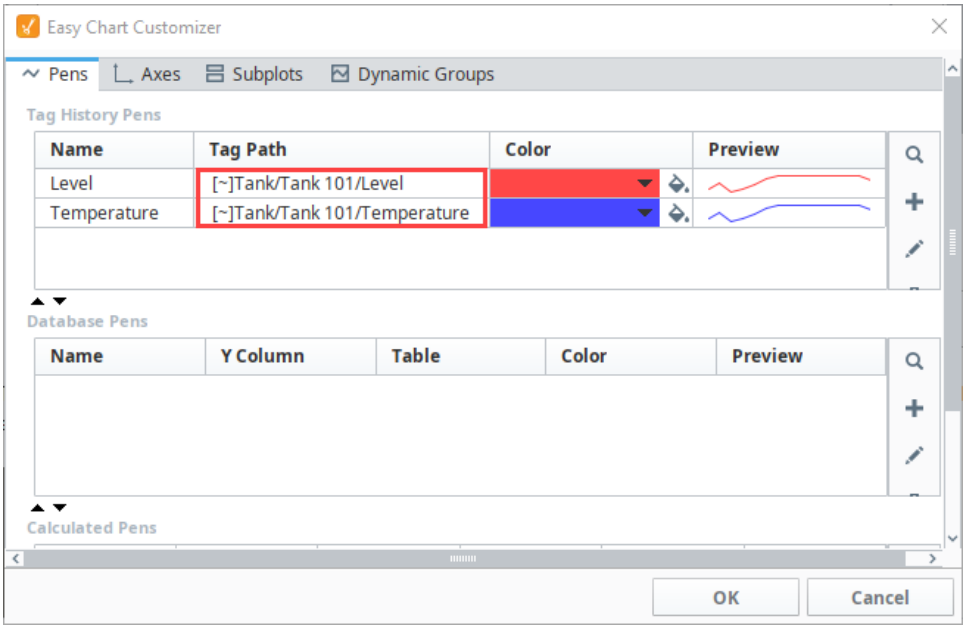


9. Put the Designer in **Preview Mode** ▶.
10. Click the **Dropdown** list to see the complete list of Tanks in your Tank Farm that you just entered.

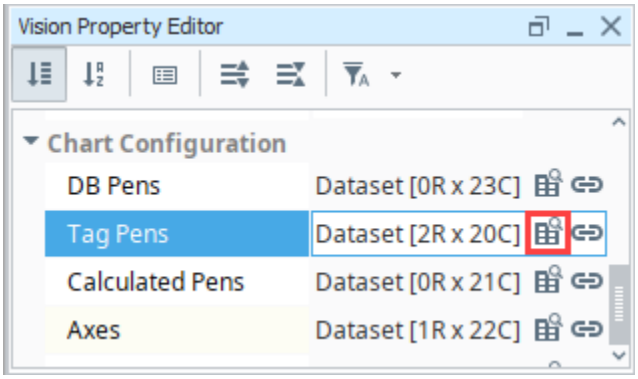


11. Put the Designer back in **Design Mode** ■.
12. Right click on your Easy Chart and choose to **Customizers >** **Easy Chart Customizer**.

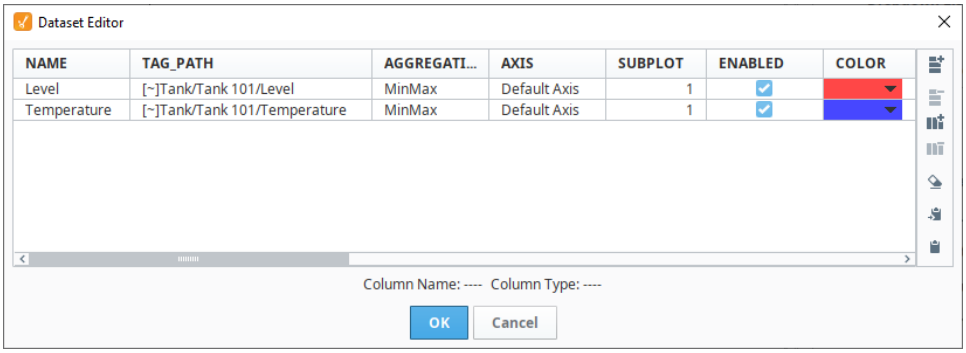
You'll notice that the Tag Paths are pointing directly to Tank 101. The only difference between Tank 101 and all the Tanks in the Tank Farm is the Tank number (i.e., 101, 102, 103, etc.). You can manually point to a different Tag Path by replacing '101' in the Tag Path with a different Tank number such as '102,' but we'd rather have the Tag Paths change dynamically when the user selects a Tank. Press **OK**.



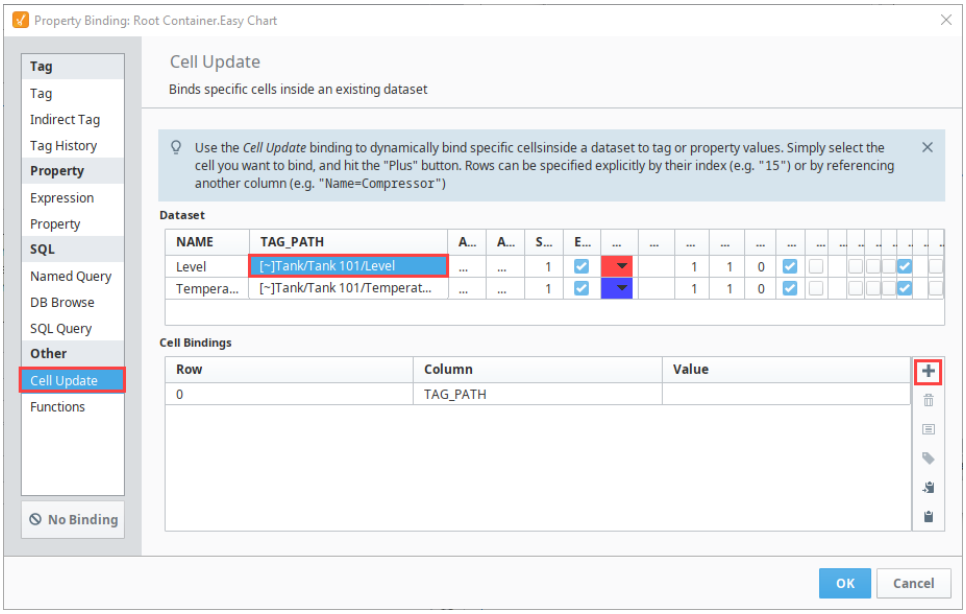
13. The Easy Chart component has a property called **Tag Pens** which stores all the configuration information that you have configured in the Easy Chart about your pens. With the Easy Chart selected, click on the **Dataset** icon in the Property Editor to view all the information for your **Tag Pens**.



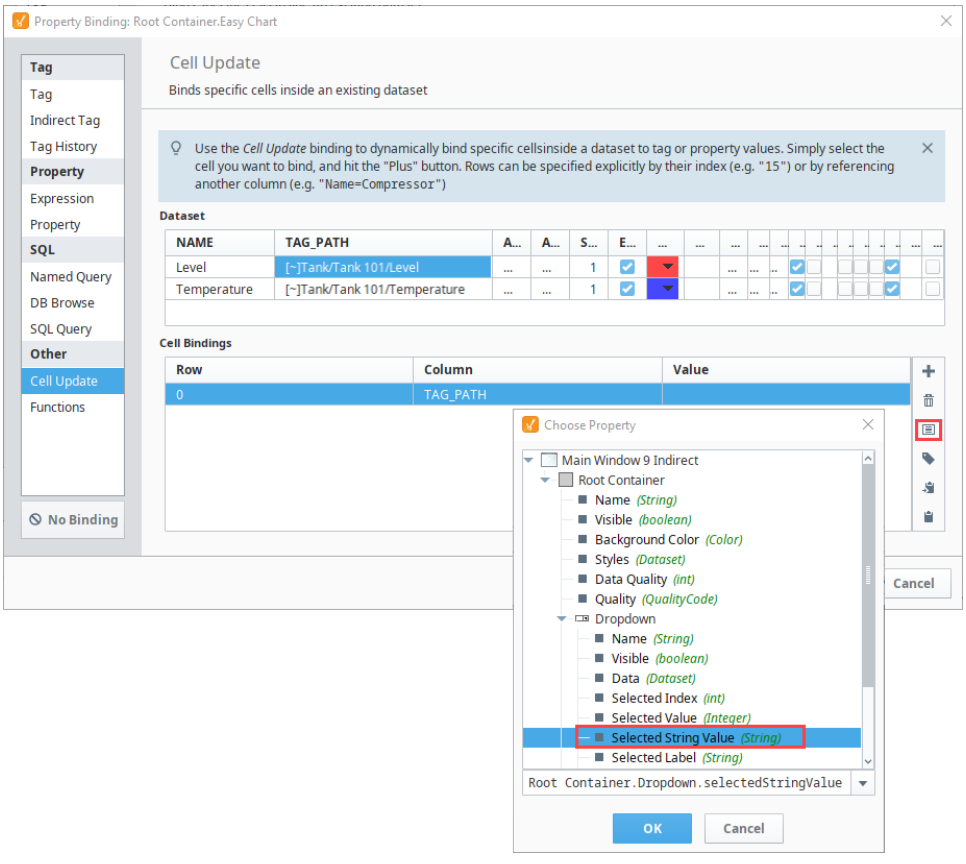
14. The dataset is displayed. It contains one row for each Tag. The second column is the Tag Path that you will want to change dynamically. Press **OK** to close the Dataset Viewer.



15. Next we'll use a Cell Update Binding in order to change an individual cell of a dataset. With a Cell Update Binding, you can select one or more cells and dynamically bind them to a property or to a Tag that you have in your system. Click on the **Binding** icon next to **Tag Pens** property to open the Property Binding window.
16. Select the **Cell Update** binding type.
17. Select Tag Path cell for **Level**, and click the **Add** icon under the **Cell Bindings** table.



18. Select the first row you just added in the Cell Bindings Table, and click the **Insert Property Value** icon.
19. From the Property Window under the **Dropdown** folder, click the **Selected String Value**. This will grab the Tank number the operator selected from the dropdown list.

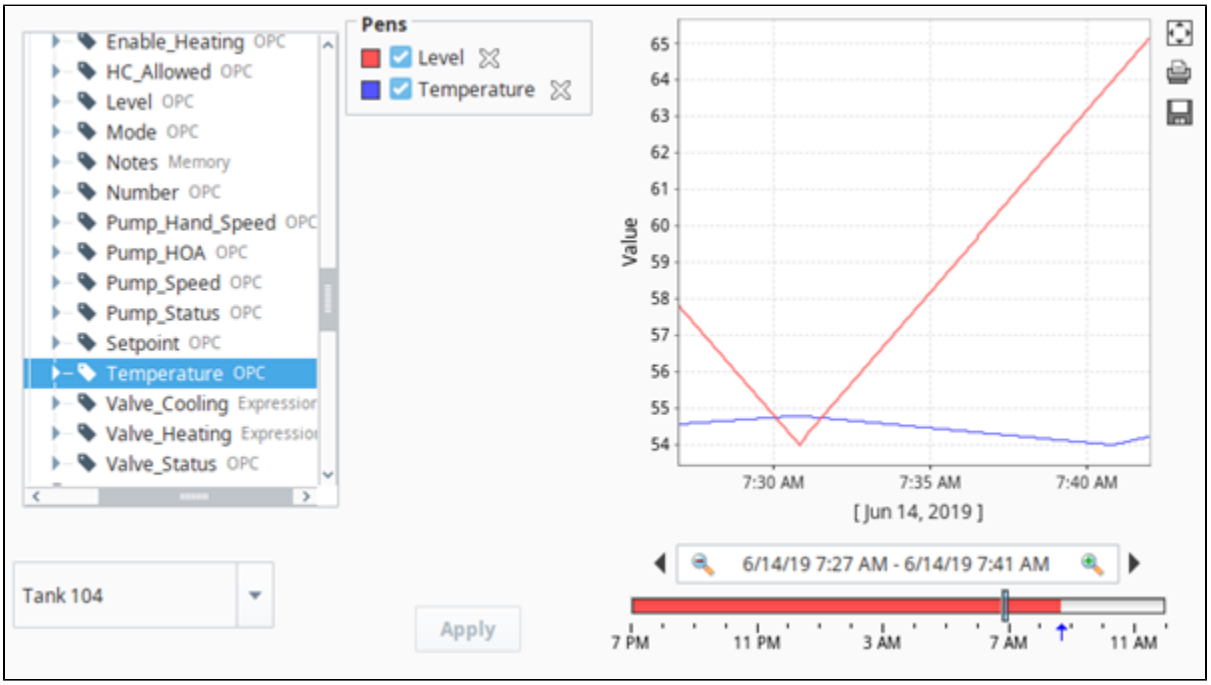


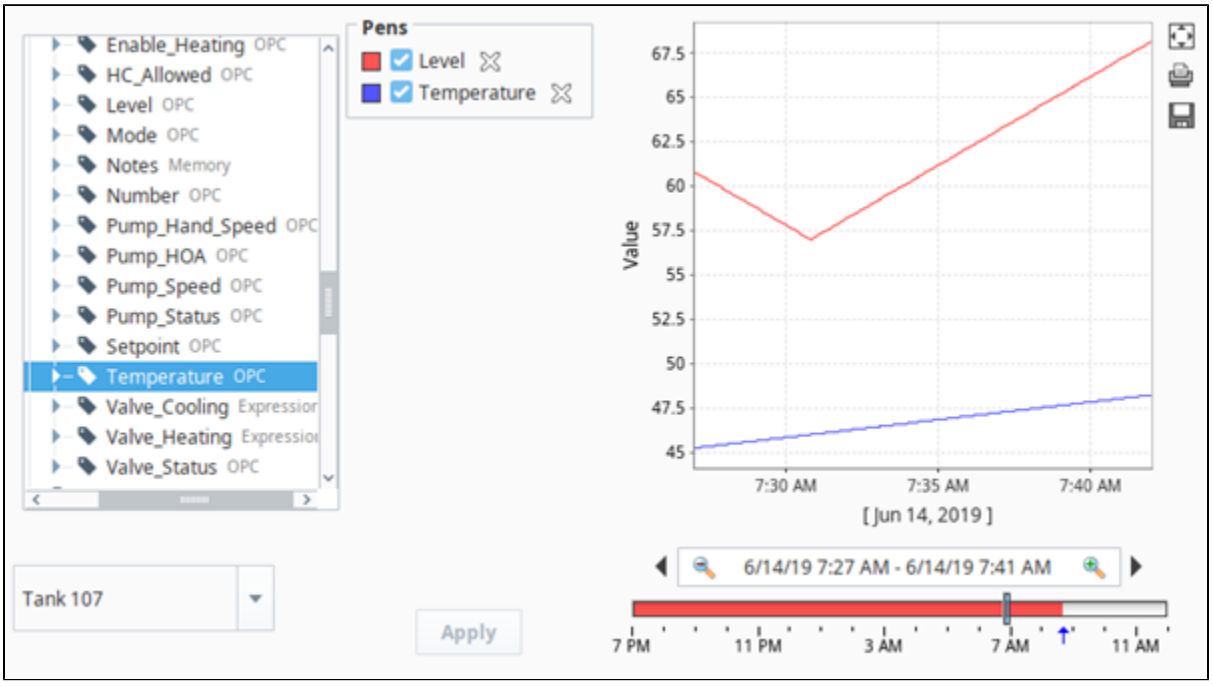
20. Click **OK**. Ignition fills in the Cell Binding value.
21. Repeat steps 17 through 20 for the Temperature Tag Path.
22. Next, expand the **Value** fields to make the Tag Path dynamic. Update the fields as follows.

Cell Binding to make Tag Path Dynamic

```
[~]Tank/{Root Container.Dropdown.selectedStringValue}/Temperature
[~]Tank/{Root Container.Dropdown.selectedStringValue}/Level
```

23. Click **OK** to save the bindings.
24. Now you have an Indirect Easy Chart. To test it, put the Designer in **Preview Mode** ▶.
25. Select a Tank from the Dropdown List. The Level and Temperature values will change in the chart. Next select a different Tank from your Dropdown List to see how the history changes on the Easy Chart.





Related Topics ...

- Easy Chart Customizer

Easy Chart - Database Pens

Database Pens

Database Pens are driven by a SQL query, so they are ideal to use when trending Transaction Group data. However, they can query for data in any connected SQL database, so it is possible to show historical data recorded by other systems on the Easy Chart.

> **This section assumes that Tags and Tag History have been configured**
>
> To learn more, go to the Tag and Configuring Tag History pages.
> The examples below use OPC Tags from the Programmable Device Simulator driver, but Memory Tags can be used instead.

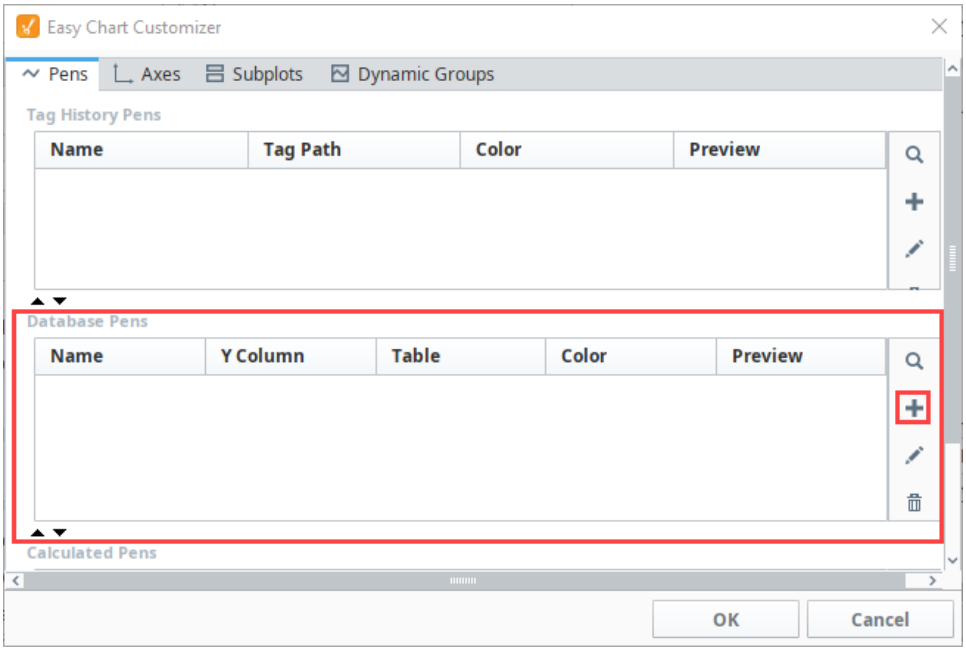
On this page ...

- Database Pens
- Configuring Database Pens

Configuring Database Pens

In this example, we'll configure a database pen driven by a SQL query to show historical data and display it on the Easy Chart.

1. Drag an Easy Chart component onto your Vision window.
2. Right click on the Easy Chart component and choose **Customizers > Easy Chart Customizer.**
3. In the Database Pens section, we'll create a database pen to trend data stored in our database. Click the **Add** icon to create the Database Pen.

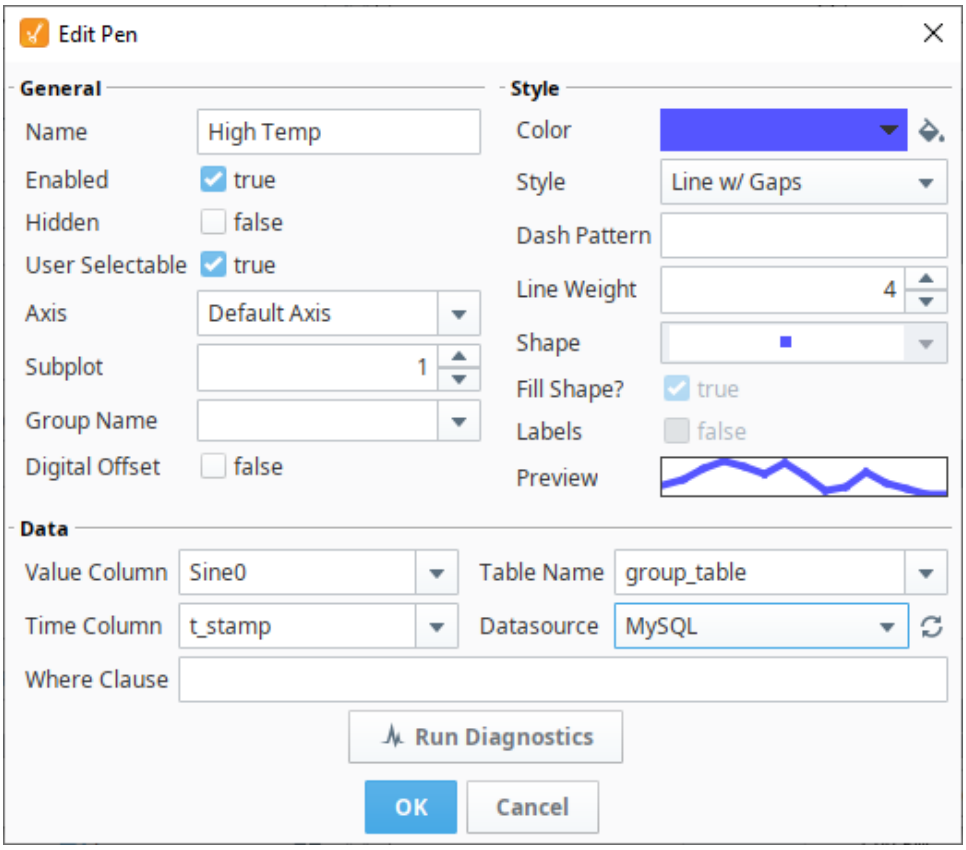


4. The Edit Pen window will open. Here is where you enter your database pen data. In the very least, we'll need values for the following properties:

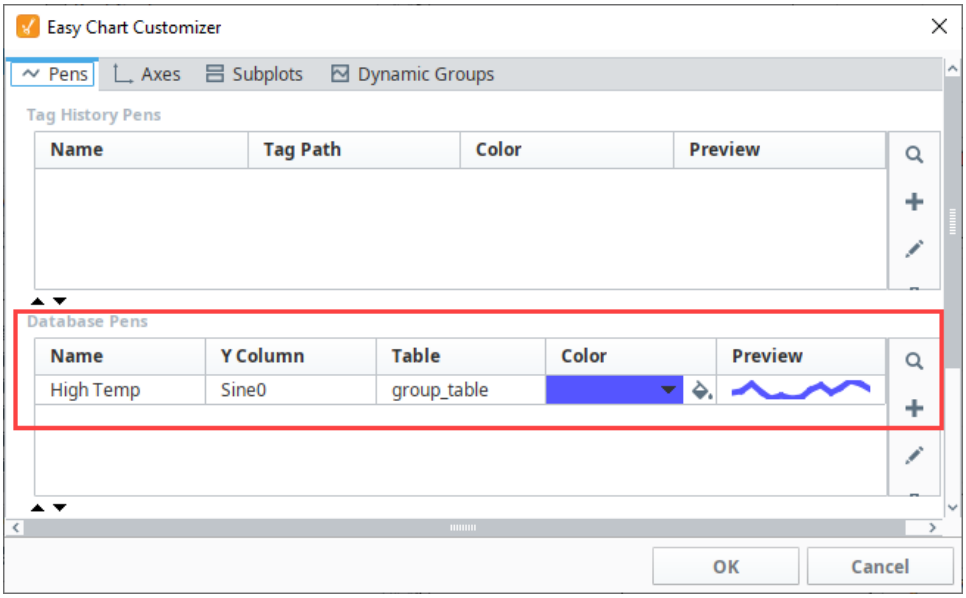
   We entered the following settings:

   a. **Name** - The name of the pen on the chart. Works the same as the name on any other pen.
   b. **Datasource** - The name of the database connection (as configured on the gateway) that contains the table we want to extract values from.
   c. **Table Name** - The name of the database table that contains the values that we want to represent on the chart.This dropdown will automatically populate with available table names once the Datasource property has a valid database selected.
   d. **Value Column** - The name of the column in the database table that contains the values we want to show, representing the value of the datapoint at a particular time. This column is ultimately responsible for the determining where each point lies on the chart's Y-axis. This dropdown will automatically populate with column names once a value for the Table Name property has been set.
   e. **Time Column** - The name of the column in the database table that contains a timestamp, representing the time of the datapoint. This column is ultimately responsible for determining where each point lies on the chart's X-axis. This dropdown will automatically populate with column names once a value for the Table Name property has been set.
   f. **Axis** - All pens require an axis. We can use the default here.

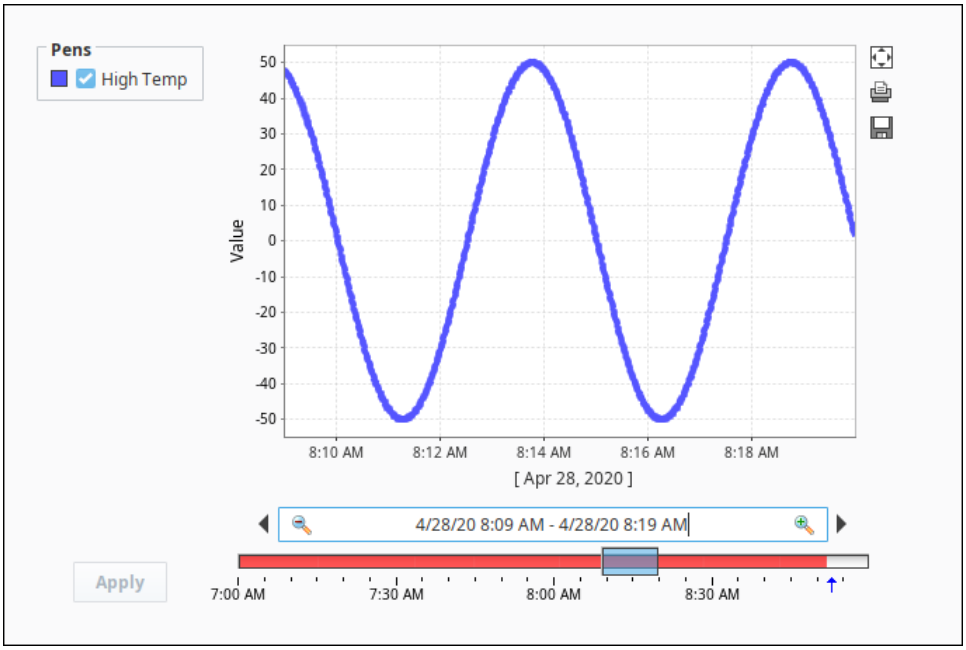
   Click **OK** to save your database pen settings.



5. In the Database Pens area, you will see your database pen that you created. Click **OK** to exit the customizer and view your Easy Chart.



6. The High Temp database pen is now trending data for the High Temp values we want to show at a selected time.



Related Topics ...

- Easy Chart Customizer

Using the Classic Chart

Chart

The Chart, sometimes referred to as the Classic Chart, is capable of rendering time series, XY, and Bar charts.

On this page ...

- Chart
  - Configuration
  - Datasets
  - Binding Techniques
  - Chart Type: XY vs Category
- Populating a Chart with Data
  - Example - Tag History
  - Customizing a Chart
  - Adding a Second Dataset to a Chart

Configuration

The basic idea behind configuring the classic chart is simple: add datasets, and fill them in with data in a format that the chart understands. You add datasets to the chart using the chart's customizer. You then use a bindings to populate each dataset. Commonly you'll use a SQL Query Binding. Since these datasets are just normal dynamic properties, you can also access them via scripting.

The Customizer also lets you add additional X and Y axes. There are various types of axes and each works slightly differently. Lastly, you can configure additional properties for each dataset, such as which axis it maps to, its visual style, subplot, etc.

Datasets

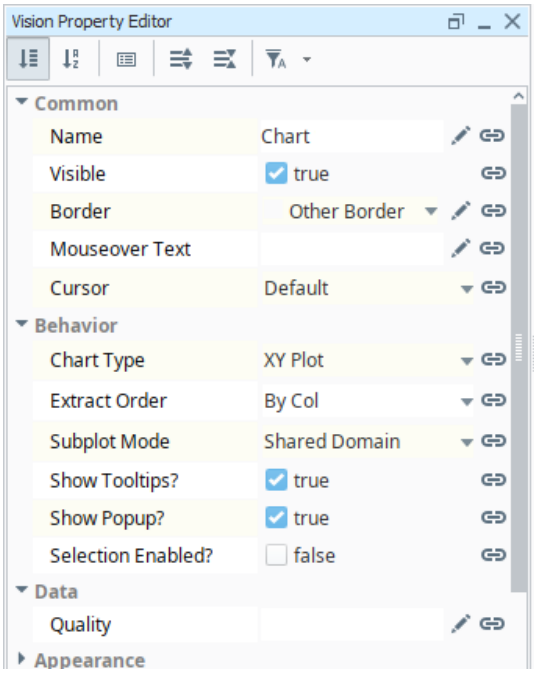
Each dataset should define one or more "series" (a.k.a "pens"). The expected anatomy of a dataset for the Classic Chart depends on several settings: namely the **Extract Order** property on the chart, as well as the type of renderer specified in the Chart Customizer's **Dataset Properties** tab.

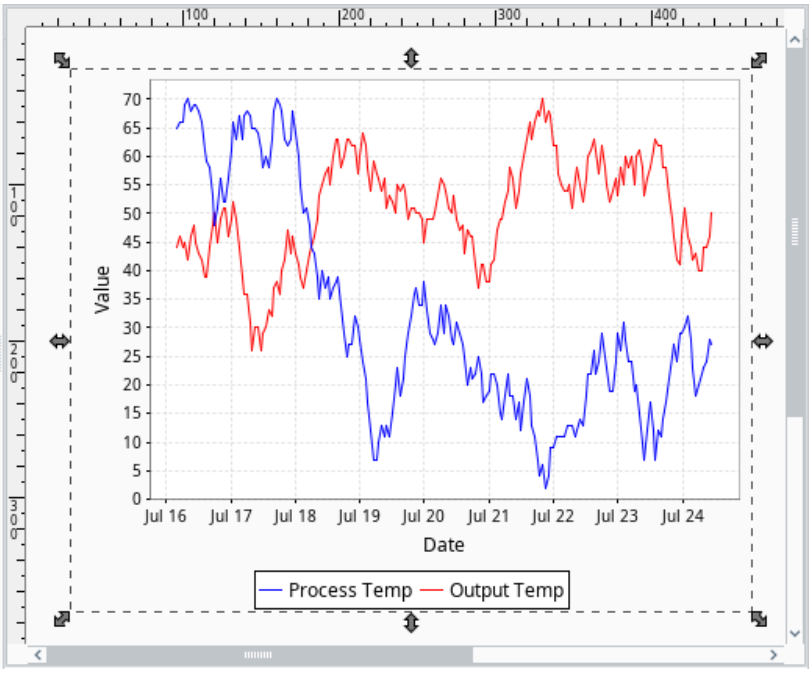
Binding Techniques

The Classic Chart can be used to make almost any kind of chart, with some effort. Historical, realtime, dynamic pen selection, etc., is all possible. Your job is just to fill the datasets with the pertinent data, and the chart will display it. The most common idea is to make the chart dynamic by varying the date range that the dataset's SQL Query bindings run. This is easy to do by adding a Date Range component and using Indirect Bindings.

Chart Type: XY vs Category

The Classic Chart is typically in XY Plot mode. This means that the X-axis is either date or numeric, and the Y-axes are numeric. If your X-axis is categorical (names, not numbers), you can switch the Chart Type property to Category Chart in the Property Editor. Don't be surprised when you get a few errors - you'll need to go and switch your X-axis to be a Category Axis, and fill your dataset in with valid category data, that is, String-based X-values. This is most often used with the Bar Renderer (see the Chart Customizer).

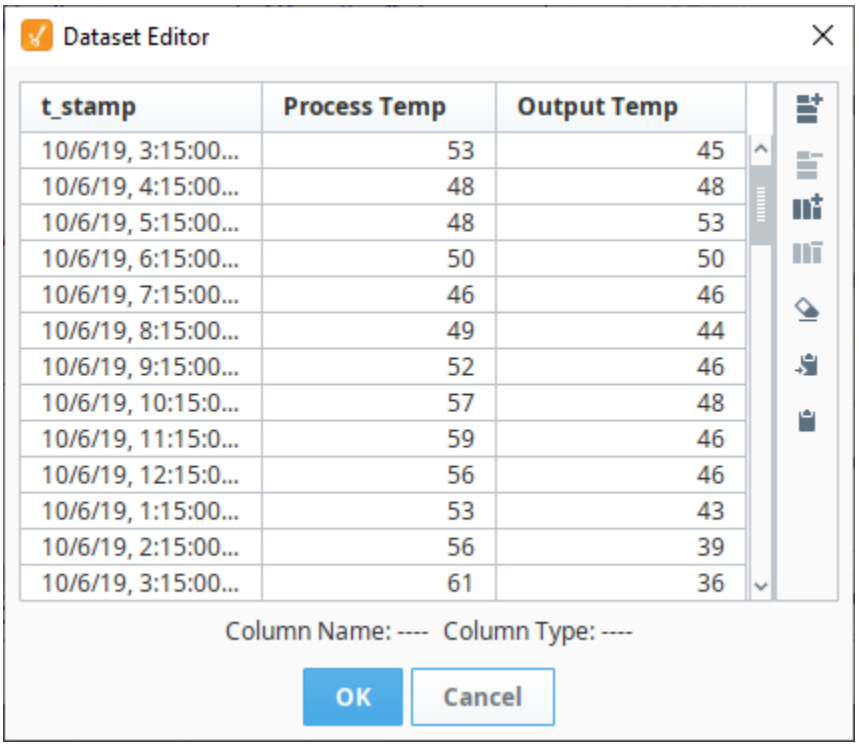




Populating a Chart with Data

Populating the Classic Chart involves using Custom Properties to collect data. New Custom Properties are added to the component via the Chart Customizer. Multiple dataset properties may be configured on the component, and each additional dataset will show as a new subplot. Populating the Chart with historical data involves populating one of these Custom Properties.

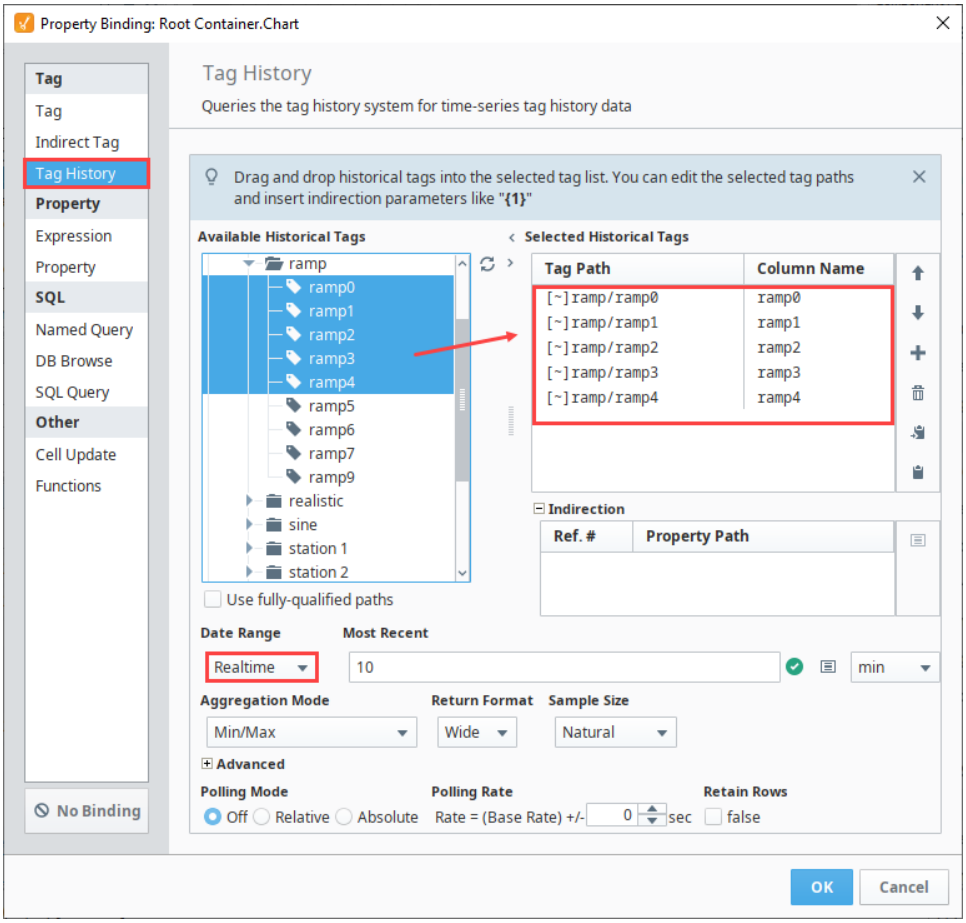
The default configuration of the component expects a dataset where the first column is a timestamp (the first column **always** acts as the domain for the chart), and sequential columns are pens that should be drawn on the chart. The default dataset contains a **t_stamp** column with values for the domain, and two columns (**Process Temp** and **Output Temp**) with values that will be drawn against the range.



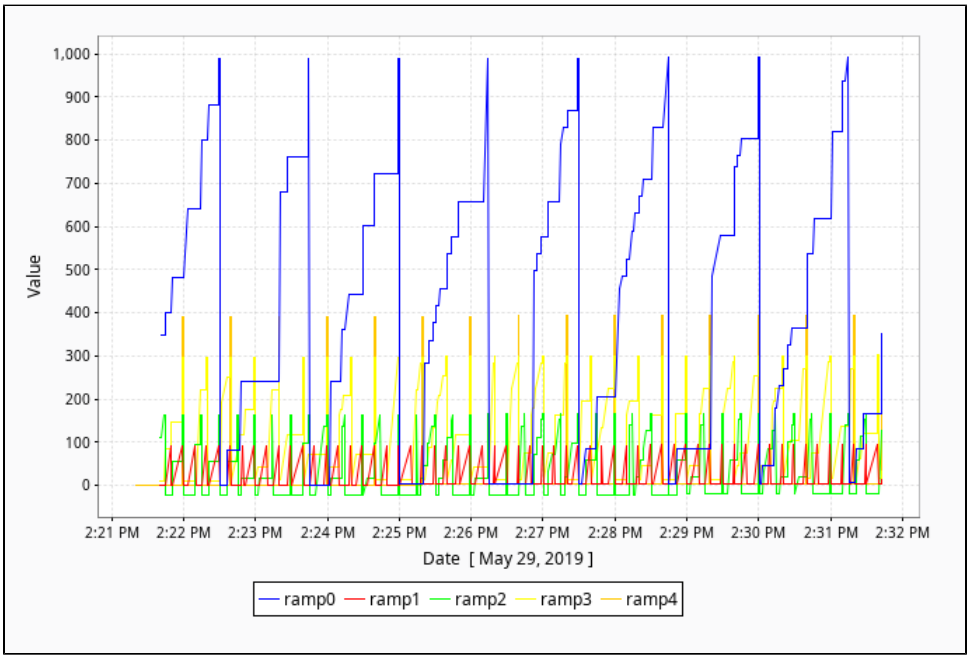
Example - Tag History

The Classic Chart is initially configured in a manner that easily displays Tag History with a Tag History Binding.

1. Drag a Chart component onto a window.
2. In the Vision Property Browser, scroll down to the **Data** property. It should be located at the bottom of the Property Editor when sorted by section. Click the **Binding** icon.
3. Select **Tag History** from the Tag section of the Binding window.
4. Browse the **Available Historical Tags**. For this example, we chose some ramp Tags. Drag the Tags over to the **Selected Historical Tags** table on the right of the window.
5. Change the **Date Range** to **Realtime** with a **Most Recent** set to 10 minutes.



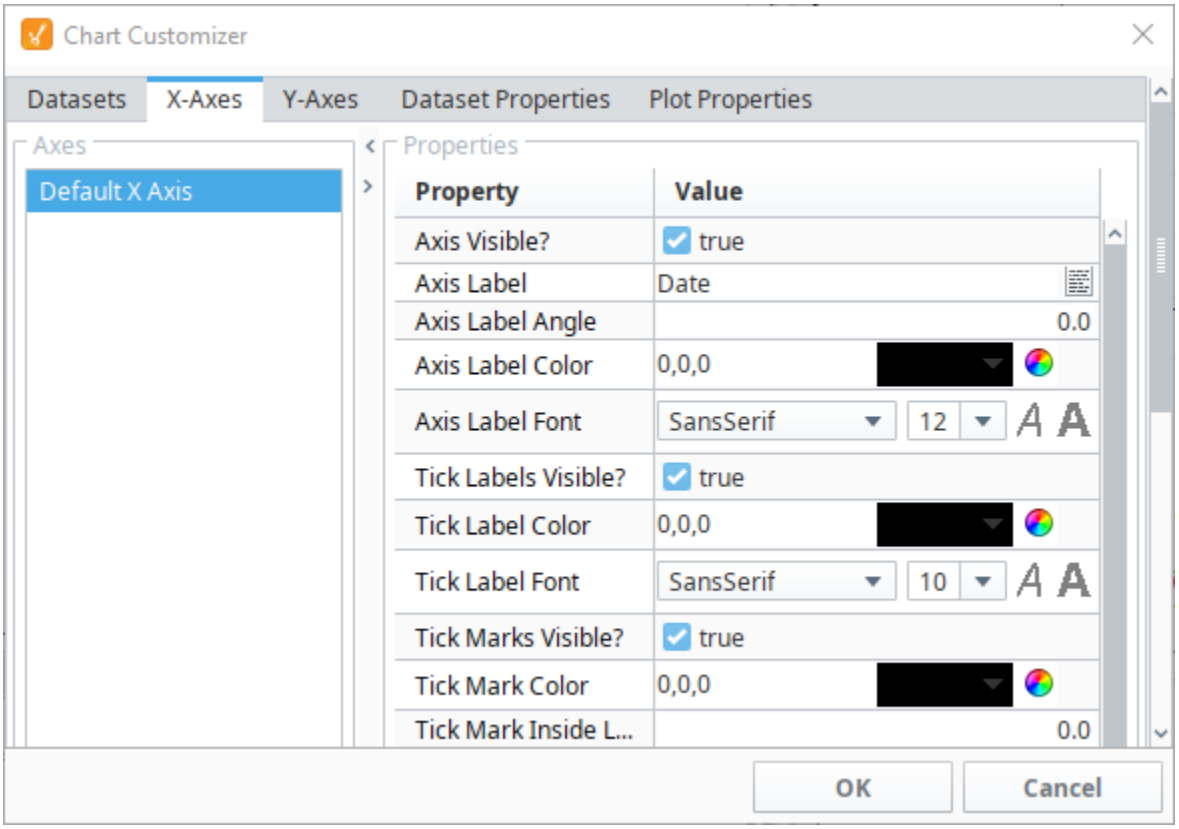
6. Click the **OK** button. The Chart will now show Tag History from the Tags you selected.



Note that the timestamp has automatically been included as the first column, so the domain has been automatically configured for you. If you wish to change the orientation of the axes, use the **Chart Orientation** property in the Property Editor to swap the position of the Domain and Range axes.

Customizing a Chart

Once trends are present on the chart, additional customization can be achieved through the Chart Customizer. When you open the customizer, you'll notice five tabs at the top: Datasets, X-Axes, Y-Axes, Dataset Properties, and Plot Properties. Each tab has its own set of properties and defaults.

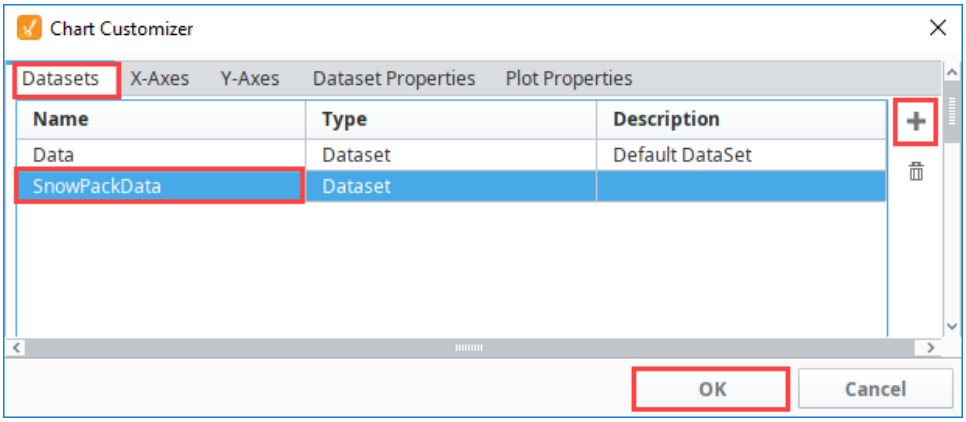


An overview of each section in the Chart Customizer is listed below:

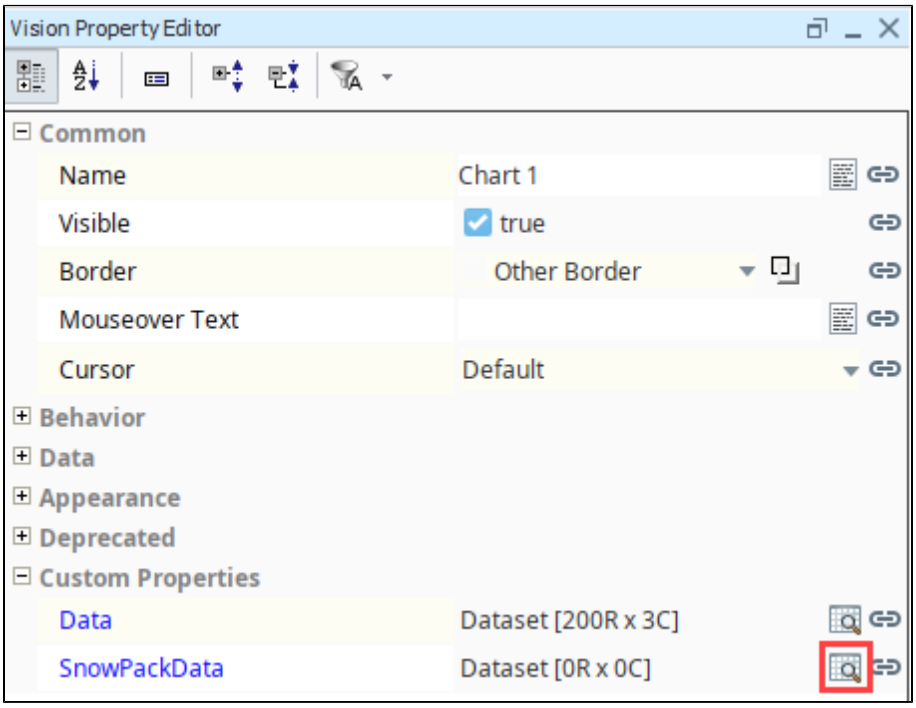
- **Datasets**: Allows additional datasets to be added to the component. Each dataset will appear as a separate custom property on the component. Data from multiple charts can be drawn on the same chart.
- **X-Axes** and **Y-Axes**: Allows for the creation of new X and Y Axes on the chart. There are six types of axes to choose from when configuring a chart, each having its own list of properties: Number Axis, Date Axis, Category Axis, Logarithmic Axis, Elapsed Axis, and Symbols Axis. Most of the X and Y axes properties are used in the customizer, and some properties are specific to the axis type and have their own unique properties. See the Chart Customizer page for more information on types of axis, associated properties, and examples.
- **Dataset Properties**: Specify which axes should a be used with each dataset. Also allows you to specify which subplot each dataset should be shown on.
- **Plot Properties**: Configure the look of each plot.

This example walks you through adding a new dataset to a chart, and modifying the existing chart properties using the Chart Customizer. When you first drag a chart to your window, you'll notice that it will display some data, that's because it's using the default dataset provided. For this example, you can add a new dataset by either copying and pasting the one below or adding your own.

1. Let's add a new dataset (i.e., SnowPackData) that measures the Snow Pack Level for the month of February. Double click on the chart to open the Chart Customizer.
2. Go to the Datasets tab and click the **Add** ✚ icon to add a new dataset.
3. Enter the Dataset name and then click **OK**.



4. In the Vision Property Editor, click on the **Dataset Viewer** icon.

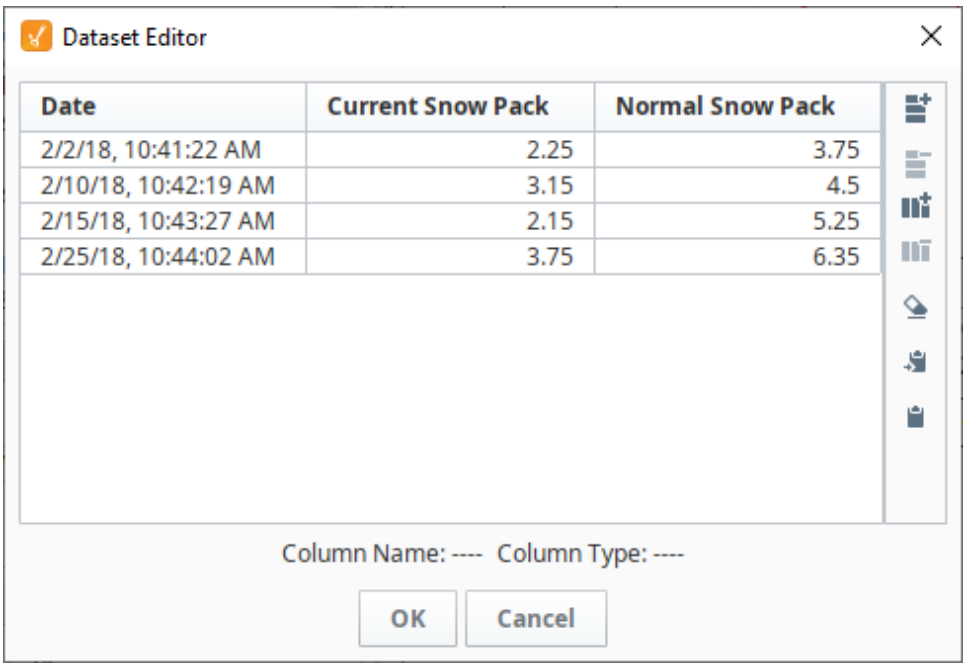


5. Highlight and copy the dataset below, then click the **Paste Dataset from Clipboard** icon.

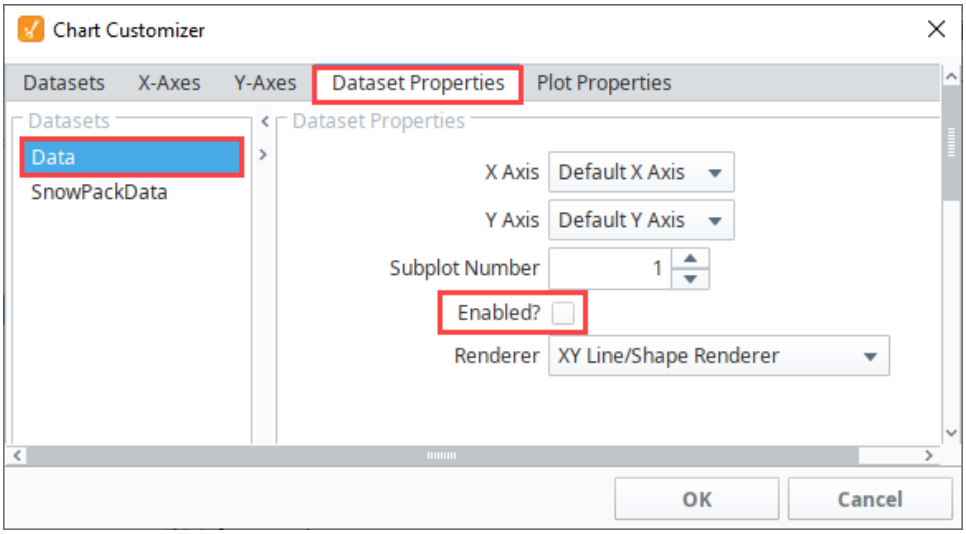
Classic Chart - SnowPack Dataset

```
"#NAMES"
"Date","Current Snow Pack","Normal Snow Pack"
"#TYPES"
"date","F","F"
"#ROWS","4"
"2018-02-02 10:41:22","2.25","3.75"
"2018-02-10 10:42:19","3.15","4.5"
"2018-02-15 10:43:27","2.15","5.25"
"2018-02-25 10:44:02","3.75","6.35"
```

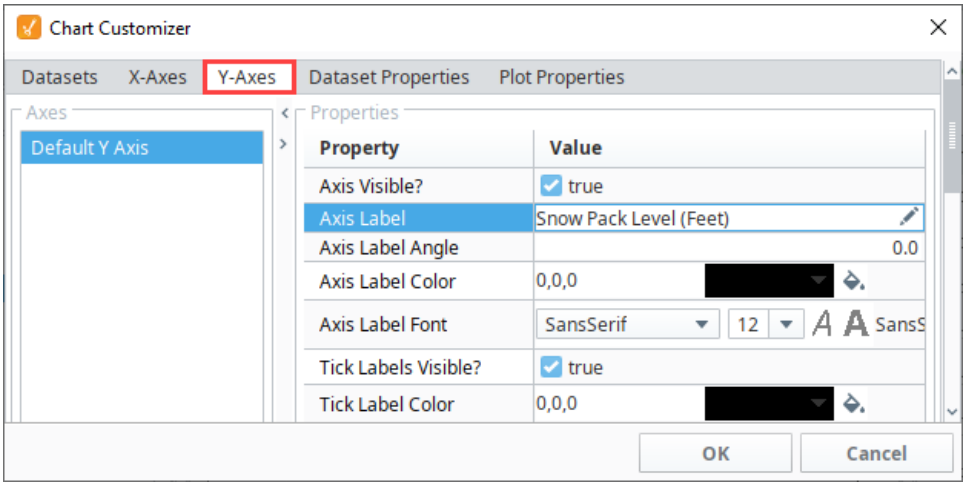
6. The data will appear in the Dataset Viewer. Click **OK** to save the dataset.



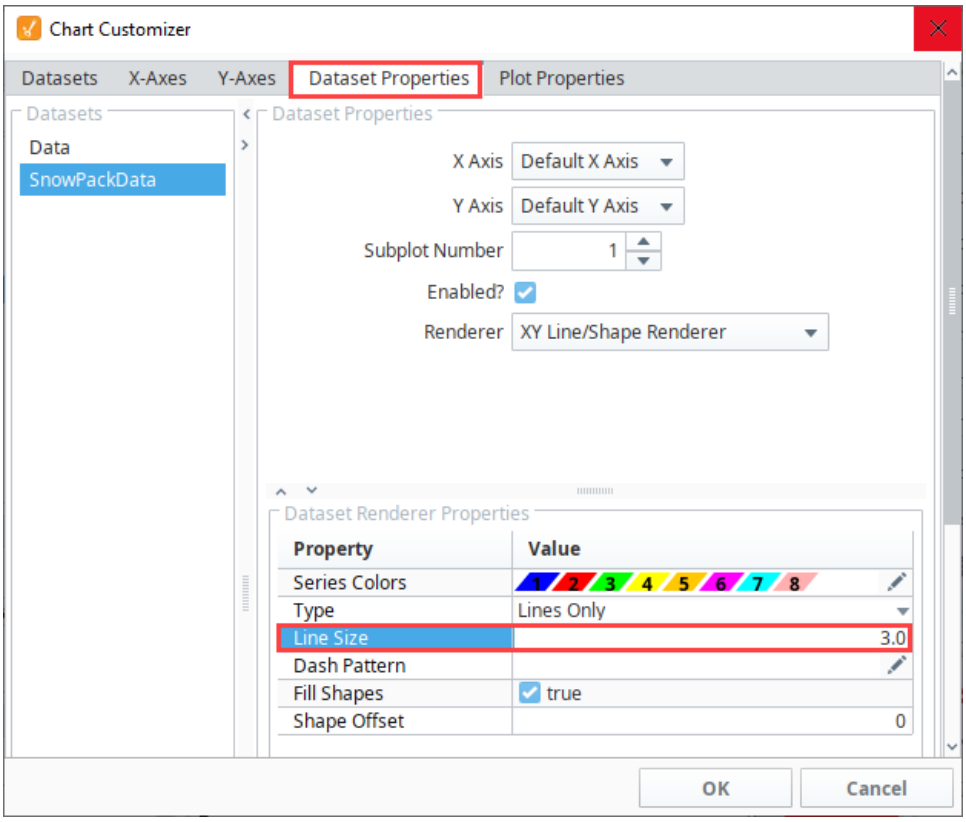
7. The Chart component is now displaying both the default dataset and **SnowPackData** dataset. Let's disable the default database. Open the Chart Customizer and go to the Dataset Properties tab.
8. Select the default dataset (i.e., Data) and uncheck the **Enabled?** box.



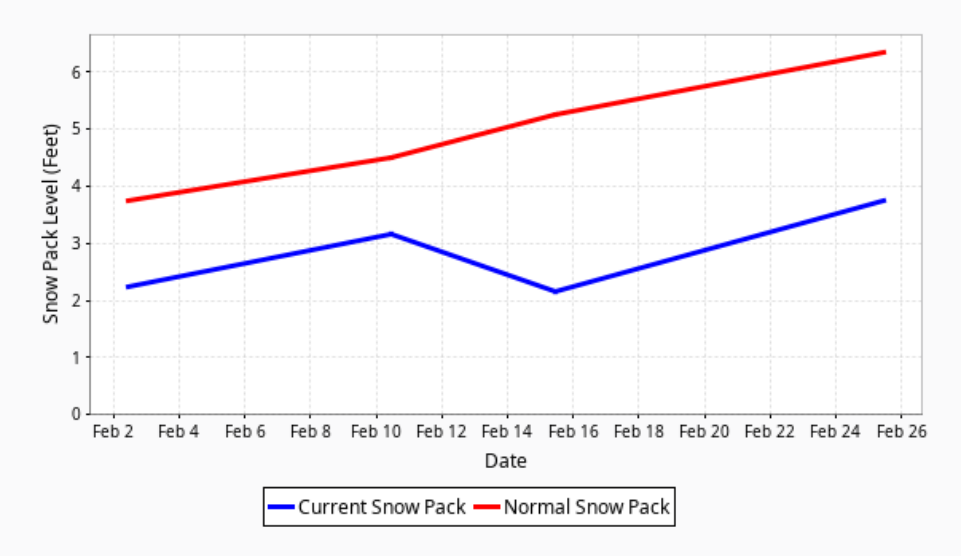
9. Click **OK**. You'll notice the chart is now only displaying the values for the new dataset (i.e., SnowPackData).
10. This step uses the default Number Axis type, but if you want to add a new Axis Type, go the **X and Y Axes tabs** and select a new Axis type**.**
11. Next, let's change a few visual properties on our chart. Go to the **Y-Axes tab** and select Default Y Axis, and change the **Axis Label** (i.e, Snow Pack Level (Feet)).



12. Lastly, let's make the plot lines thicker on the chart. Go to the Dataset Properties tab, and change the **Line Size** renderer property from 1.0 to **3. 0**.



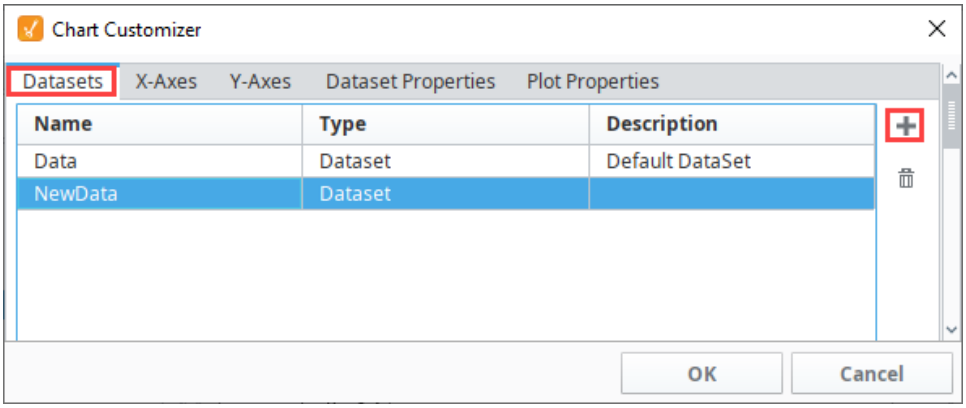
13. Here is the chart with the new dataset and updated properties. As you can see, you can easily configure additional properties for a dataset, as well as choose from a host of visual style properties to design your charts using the Chart Customizer.



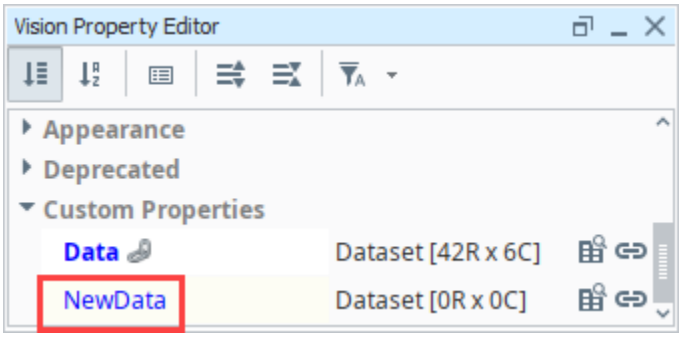
Adding a Second Dataset to a Chart

In this example, we'll add an additional dataset to the chart.

1. Double click on the Chart component to open the **Chart Customizer (**or right-click and select **Customizers > Chart Customizer)**.
2. Once the Chart Customizer is open, make sure the **Datasets** tab is selected, and click the **Add** ✚ icon to add a new dataset.
3. A new row will appear. Give the new dataset a name by typing into the cell under the **Name** column. We'll call it **NewData**.



4. Click the **OK** button.
5. Check the bottom of the Property Editor. The newly created dataset will appear and may now be populated with data.



6. Now that we have a new dataset, you can add data however you like and it will show up in a second subplot.

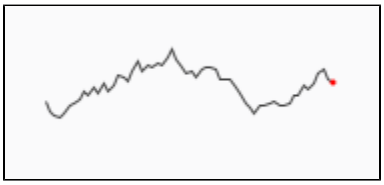
Other Vision Trending Charts

Along with the Easy Chart and Classic Chart, there are several other types of charts contained in the Vision Module that can also visualize a trend and track a change in condition, output, or process through a single data point or multiple data points over time. This page describes the Sparkline and Status Charts and how they can be used.

On this page ...

- Sparkline Chart
  - Usage
- Status Chart
  - Series Data
  - Color Mapping

Sparkline Chart



The Sparkline Chart is a minimalistic chart that displays a line-chart history for a single data point that fits well with High Performance HMI screens. It's a great way to show a lot of contextual information in a very small amount of space. It provides a fast way to display the recent history of a single data point so the viewer can quickly discern the most recent trend.

Note, the Sparkline Chart can not show multiple trends, however, multiple Sparkline Charts can be stacked. This chart is often laid over an image or level display to show a simple trend with the current value.

Sparkline Chart

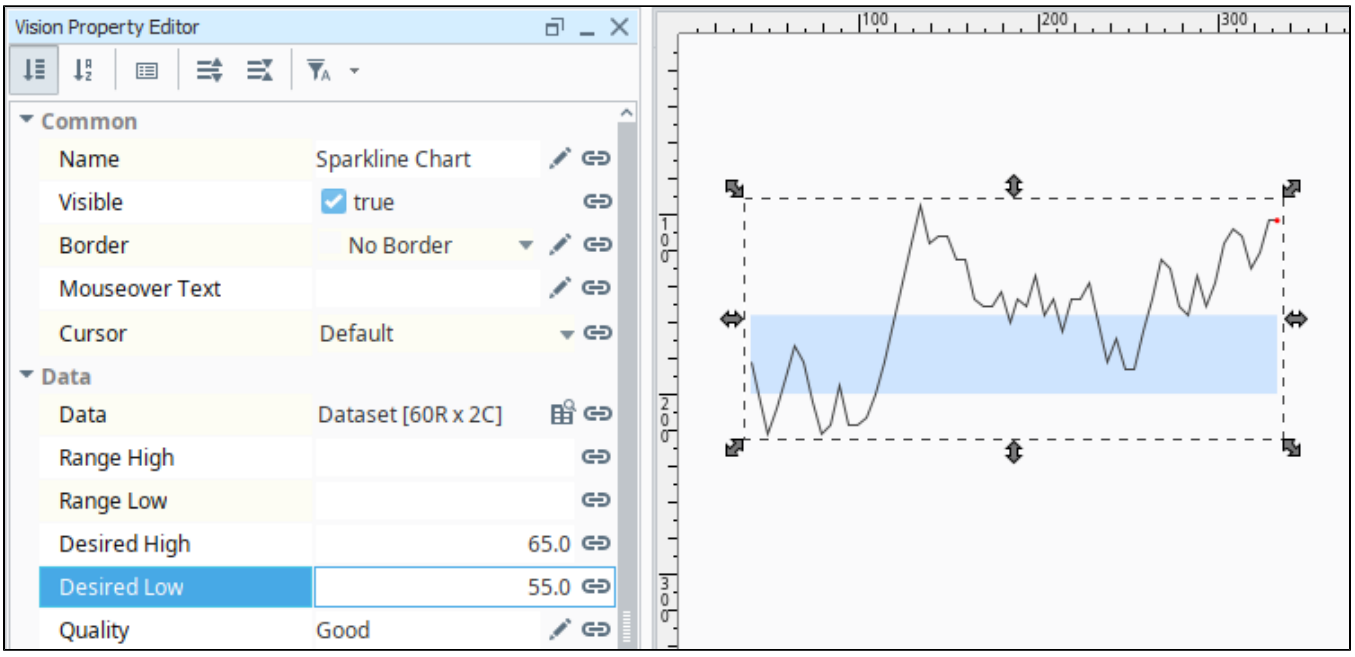
Watch the Video

Usage

The trend on the Sparkline Chart is determined by the **Data** property located in the Property Editor. Bind the Data property to either a Tag Historian realtime query, or to a database query. The dataset should contain two columns: the first being a **date**, and the second a **number/value**.

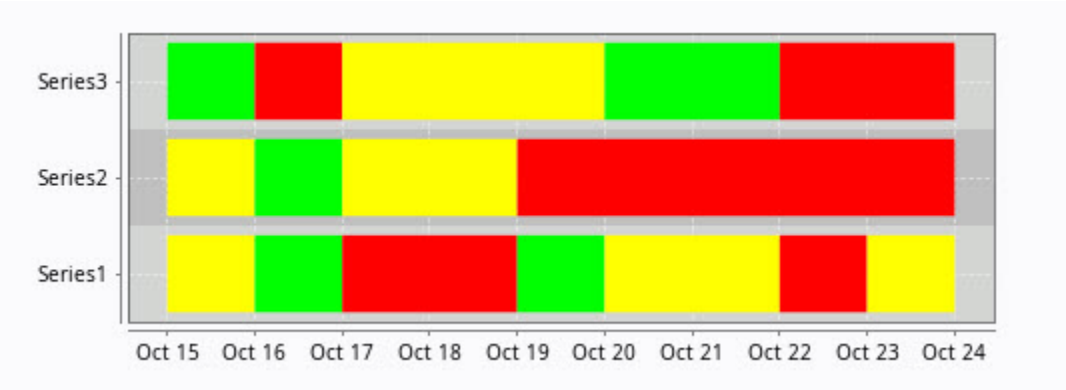
Any additional columns in the dataset are ignored. Thus, the Sparkline Chart will only show a single pen per instance of the component. The data should be sorted by date in ascending order.

The red dot on the Sparkline Chart represents the most recent value. The Sparkline Chart can display a band of color across the back of the chart which indicates the desired operating range of the datapoint. Setpoints can be displayed on the chart by setting a value for the **Desired High** and **Desired Low** properties. This allows an easy way to view when the value fell out of bounds. Note that both properties must have a value for the **Desired Range Color** to appear.



Status Chart

The Status Chart allows you to visualize the status of one or more discrete data points over a time range. The X-axis is always a time series axis, and the Y-axis is a category axis, with one entry per data series. The Status Chart is populated with the **Series Data** property in the **Property Editor**. This chart is good for showing machine states or HOA values over time. It is recommended to include some sort of Legend when using this chart.



Series Data

The first column of the **Series Data** property must contain datetime values. Each additional column should be numeric (the default columns are doubles). The order of the columns (left-to-right) determines the order of the entries on the chart (bottom-to-top). Because of this, re-ordering the entries would involve changing the order of the columns as they appear in the Series dataset by modifying the mechanism that is populating the dataset (i.e., changing the order of columns in a query).

In **Wide** format, all of the columns but the first must be numeric. These "series" columns' headers will be used as the names on the y-axis. In **Tall** format, there should be exactly three columns. The first is the timestamp, the second is the series name, and the third is the value. For example:

Wide Format

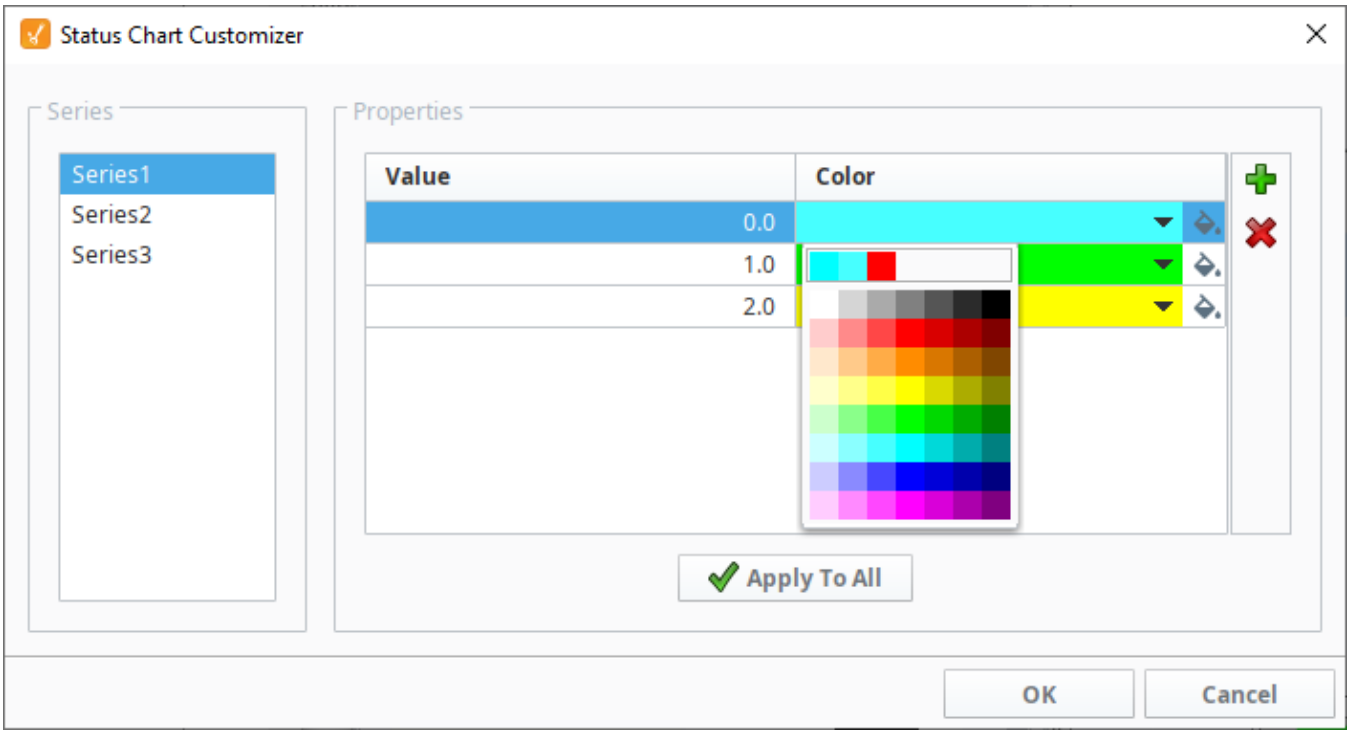
t_stamp	Valve1	Valve2
2010-01-13 8:00:00	0	2
2010-01-13 8:02:00	0	2
2010-01-13 8:04:00	1	2
2010-01-13 8:06:00	1	1
2010-01-13 8:08:00	0	1

Tall Format

t_stamp	Name	Value
2010-01-13 8:00:00	Valve1	0
2010-01-13 8:00:00	Valve2	2
2010-01-13 8:02:00	Valve1	0
2010-01-13 8:02:00	Valve2	2
2010-01-13 8:04:00	Valve1	1
2010-01-13 8:04:00	Valve2	2
2010-01-13 8:06:00	Valve1	1
2010-01-13 8:06:00	Valve2	1
2010-01-13 8:08:00	Valve1	0
2010-01-13 8:08:00	Valve2	1

Color Mapping

Apart from getting the data into the series chart, the only other commonly configured option is the mapping of discrete values to colors. This is done in the Status Chart Customizer. Each named series can have its own mapping of colors, if desired. These mappings are stored in the expert-level dataset property **Series Properties Data** so they can be altered at runtime.



Vision Client Tags

Client Tags, as the name implies, are only available for use in Vision Clients. Their values are isolated to a Client runtime. All clients will have the same list of client Tags, however, the actual values are unique and independent for each running Client. In other words, even though client Tags are created in the Designer, each client will create their own instances. This makes them very useful as in-project variables for passing information between screens, and between other parts of the clients, such as scripting.

Client Tags support most of the data types that standard Tags do (including datasets). Additionally, Client Tags do not have a Tag Group property, so the value will only update when the polling property executes, or a reference in the Client Tag's expression updates.

Often, users will create parameterized windows that take in a line ID or machine name. A Client Tag can store this value for indirection to be used across multiple windows without users on different clients fighting over the current value as they would with a standard Tag.

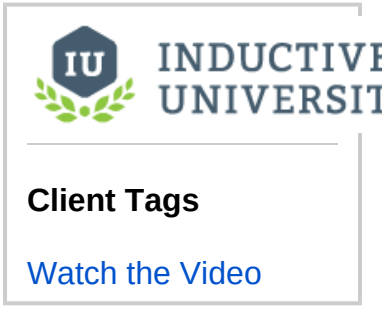
Note: Client Tags are only available to Vision resources, and are unavailable to Perspective and Gateway scoped resources. If you are using Perspective see the Session Properties page for a similar system.

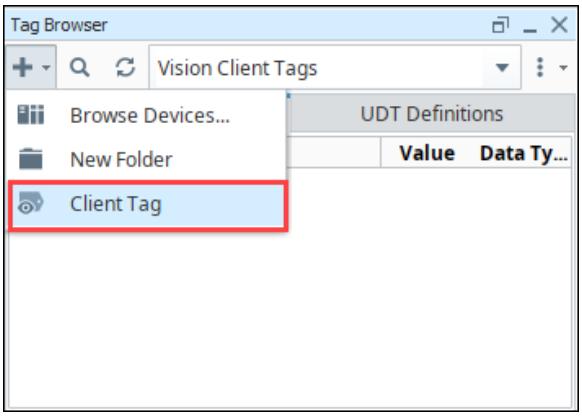
The following feature is new in Ignition version **8.1.26**
Click here to check out the other new features

Note: Writes to Client Tags are still allowed when the Vision Client is set to read-only mode.

On this page ...

- Expression Type
- Create a Client Tag
- Using Vision Client Tags
- Overriding Vision Client Tags
- System Client Tags





Expression Type

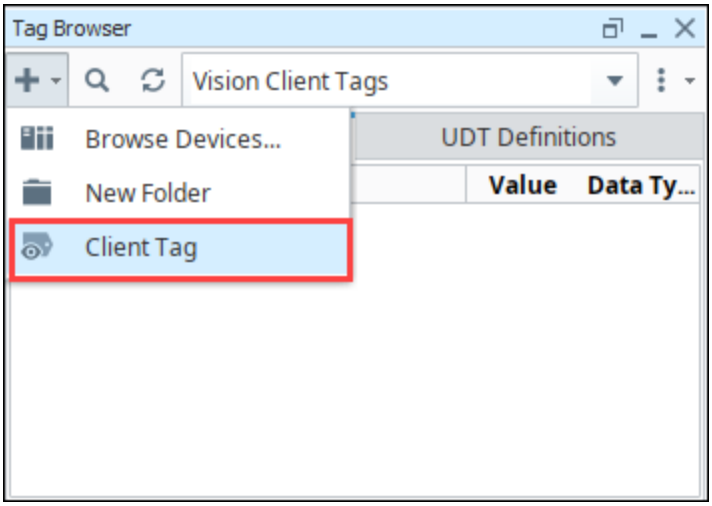
Client Tags can be configured in one of several ways, based on the **Expression Type** property. The following options are available.

- **None**: Causes the Tag to behave like a Memory Tag.
- **Expression**: Allows the Tag to utilize Ignition's Expression Language, much like an Expression Tag.
- **Query**: Executes a SQL Query, similar to a Query Tag.
- **Named Queries**: Client Tags may call Named Queries

Create a Client Tag

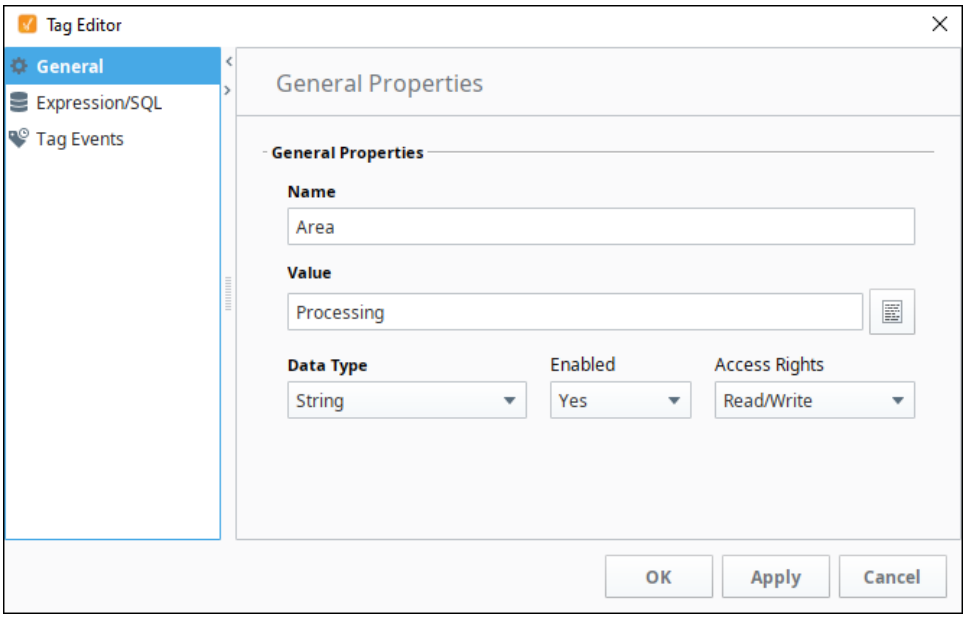
This example shows how to create a client Tag.

1. In the Tag Browser, select Vision Client Tags in the Tag Provider Selector.

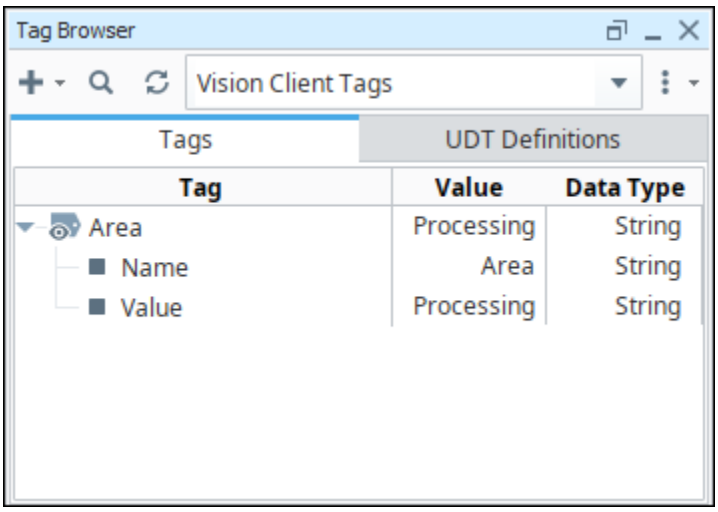


2. Click the **Add** ✚ icon and select **Client Tag**. The Tag Editor will open.
3. In the **Tag Editor** enter the Tag name and the following general properties, then click **OK.**

   Name: **Area**
   Value: **Processing**
   Data Type: **String**



4. A new Tag called **Area** is created in the **Client** folder.



Using Vision Client Tags

Once a Client Tag is created, it can be used just like any other Tag. You can drag-and-drop, bind to it, use it in scripting, or add it to a Transaction Group.

Overriding Vision Client Tags

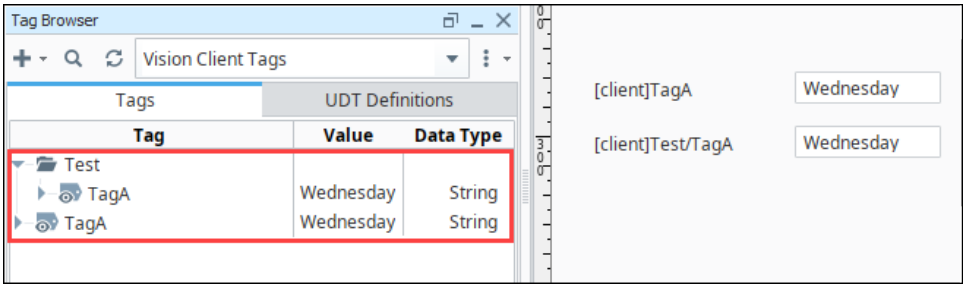
When you open a Client, your Vision Client Tags default to the value that was saved in the Designer. If you want to open a Client with different values in your Client Tags, you can override them in the Vision Client Launcher (either for all applications, or an individual application). A Client Tag can be overridden in the **Client Tag Overrides** section of the Vision Client Launcher.

- Use the **Settings** button to override a client Tag in all applications.
- Select **Manage** on the individual application to override a tag on that specific application.

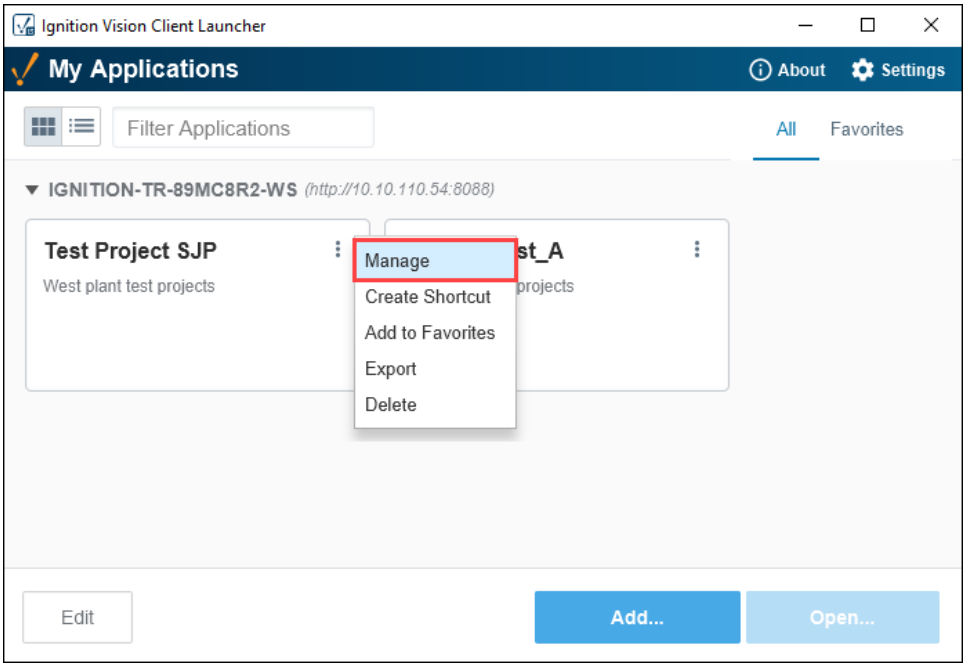
Client Tags can be overridden within a folder. You can also pass client Tag overrides with spaces by using a '+' icon as an escape character. This happens automatically if you are configuring overrides in the Client Launcher.

The following example shows how to override client Tags on a single application.

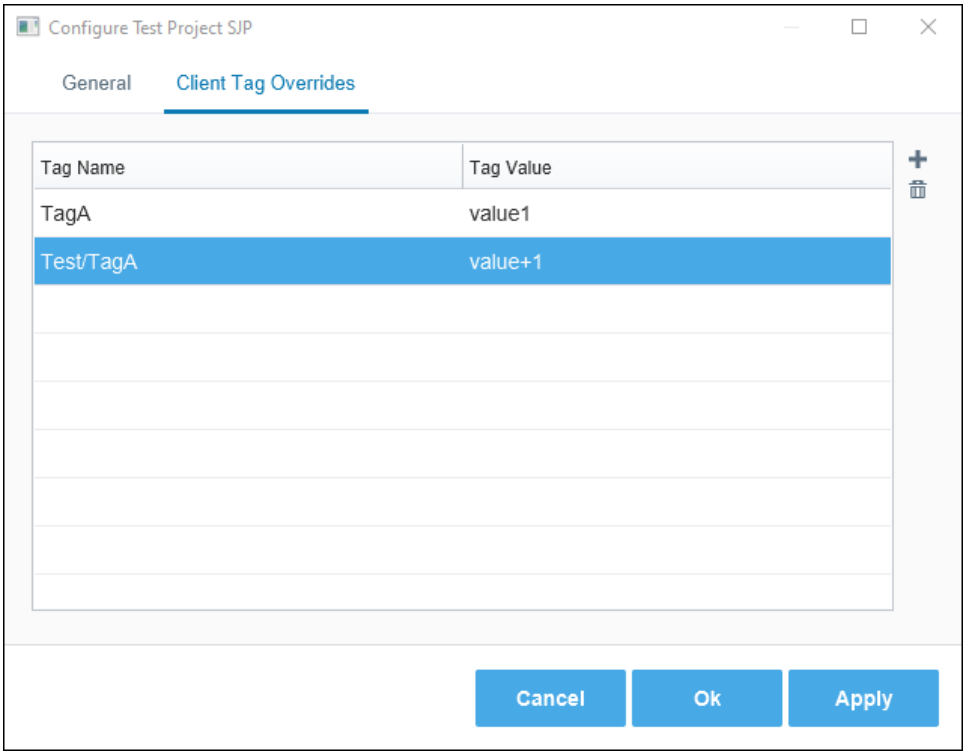
1. In the **Designer**, create a client Tag inside a folder and bind it to something that can be displayed on a window. This example has two Tags: one Tag is directly under Vision Client Tags folder, and another one under a nested folder. Each **TagA** tag is bound to a different Label components.



2. Open the **Vision Client Launcher**. On your application, click the **More Options** ⋮ icon to open the **Manage** screen.

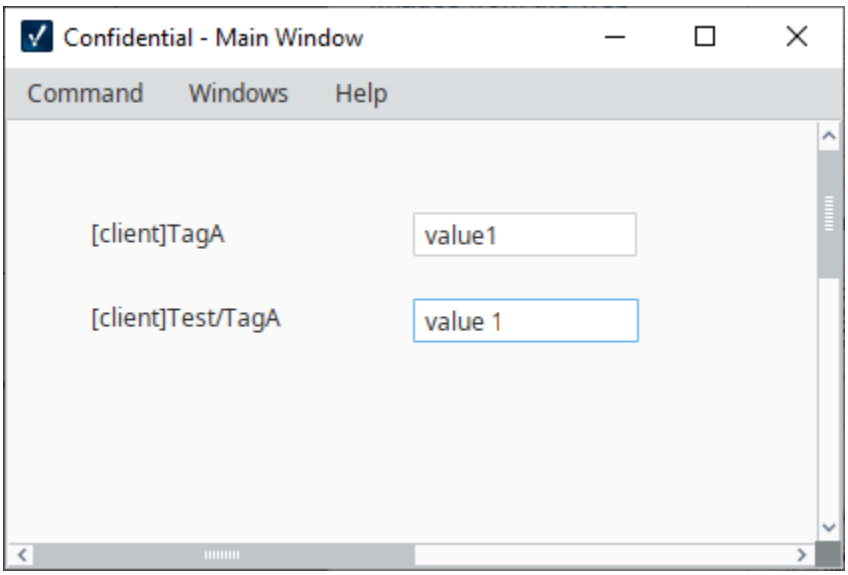


3. Select the Client Tag Overrides tab.
4. Click the **Add** ✚ icon on the right side of the table to add a Tag override. In this example, let's configure two client Tag overrides. The first TagA is for a client Tag directly under Vision Client Tags. Enter the **Tag Name** and **Tag Value.**
5. The second override is for the nested TagA Tag. In the Tag Name, the folder name must precede the Tag name followed by a forward slash. Notice how the value has a **'+'** that denotes a space. The launcher will automatically enter the space for you in the client launcher when you save your updates. Enter the **Tag Name** (including the folder name) and **Tag Value.**



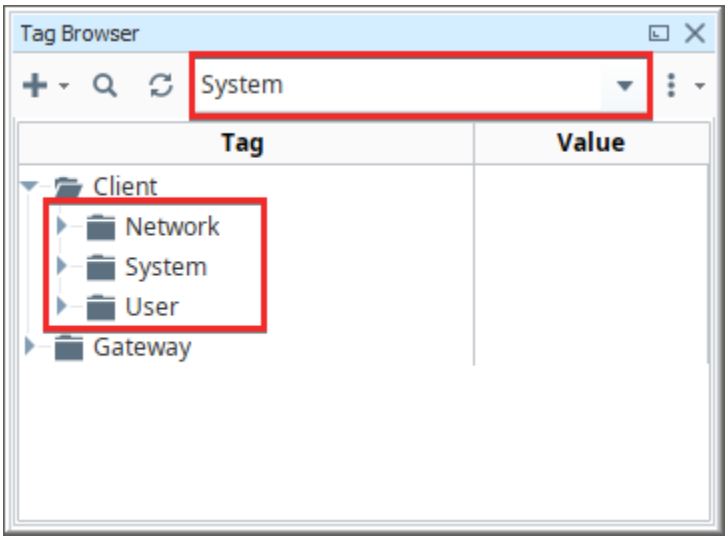
6. Click the **OK** button. This will take you back to your **My Applications** screen.

7. Select your project and click the Open button. You will see the new values that you overrode in the Client Launcher.



System Client Tags

System Tags provide status about the Ignition system, such as memory usage, performance metrics, and so on. Every individual Vision client is going to have its own values like IP address, hosting name, username, and more. You cannot modify System Client Tags. For more information, see System Tags.



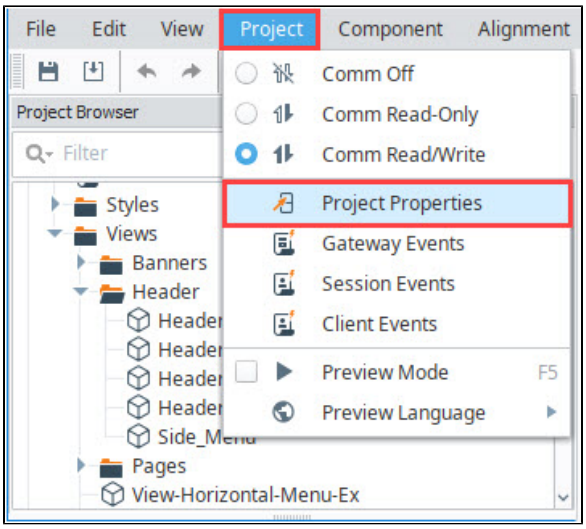
Related Topics ...

- Vision Client Launcher

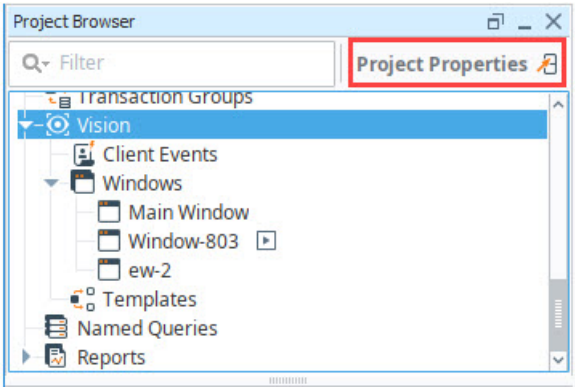
Vision Project Properties

There are a number of properties you can set for your Vision projects within the Designer. For example, there are properties for setting the Touch Screen mode, customizing a client's auto-login, or configuring how the clients receive updates, and more.

To access the Project Properties, in the Designer, click on **Project** tab on the menu bar. Then select **Project Properties**.



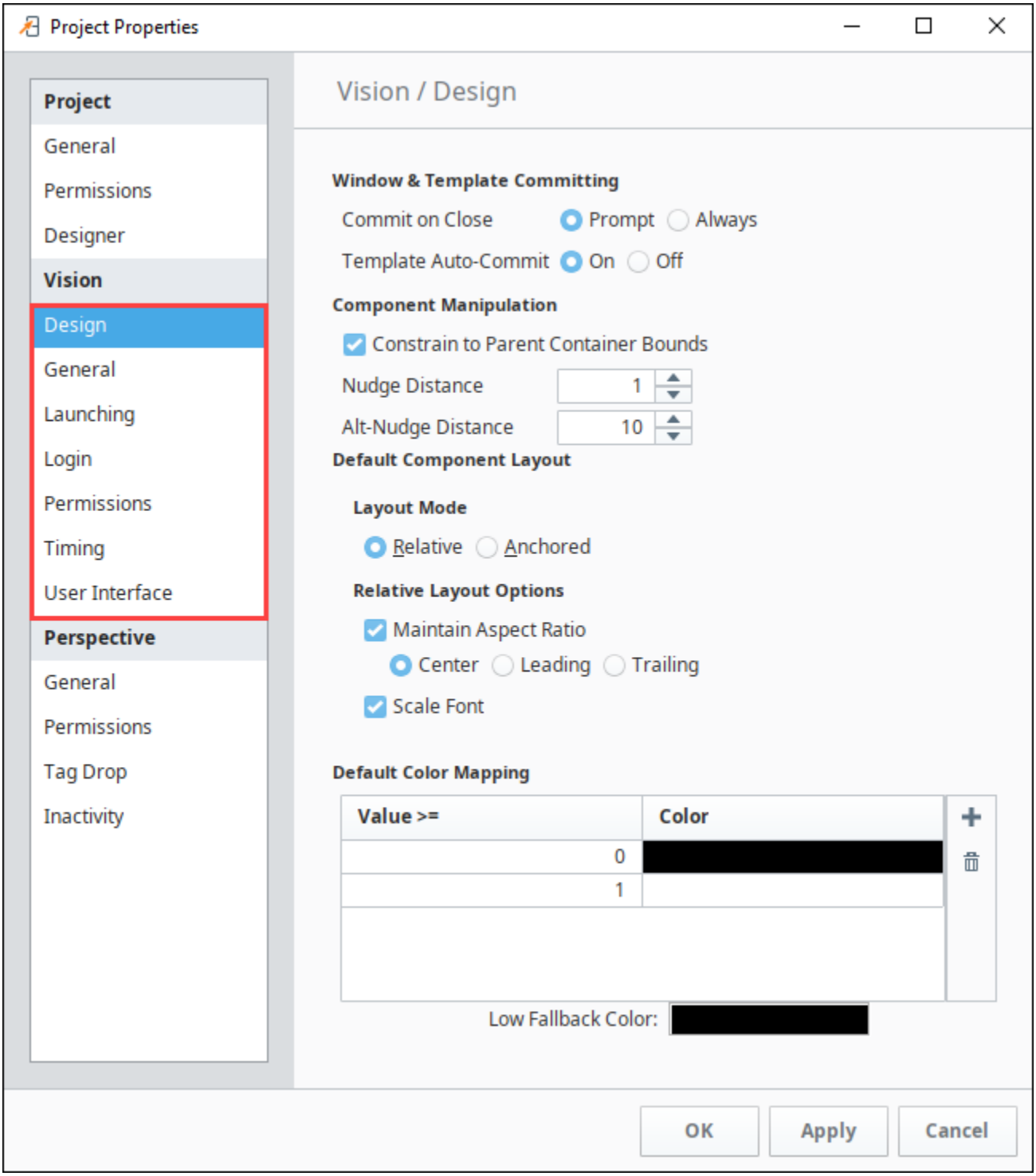
Alternatively, you can click on the **Project Properties** icon at the top of the Project Browser.



On this page ...

- Vision Design Properties
- Vision General Properties
- Vision Launching Properties
- Vision Login Properties
- Vision Permissions Properties
- Vision Timing Properties
- Vision User Interface Properties

Project properties span several functional areas each containing settings applicable to that area. Scroll down to the **Vision** section.



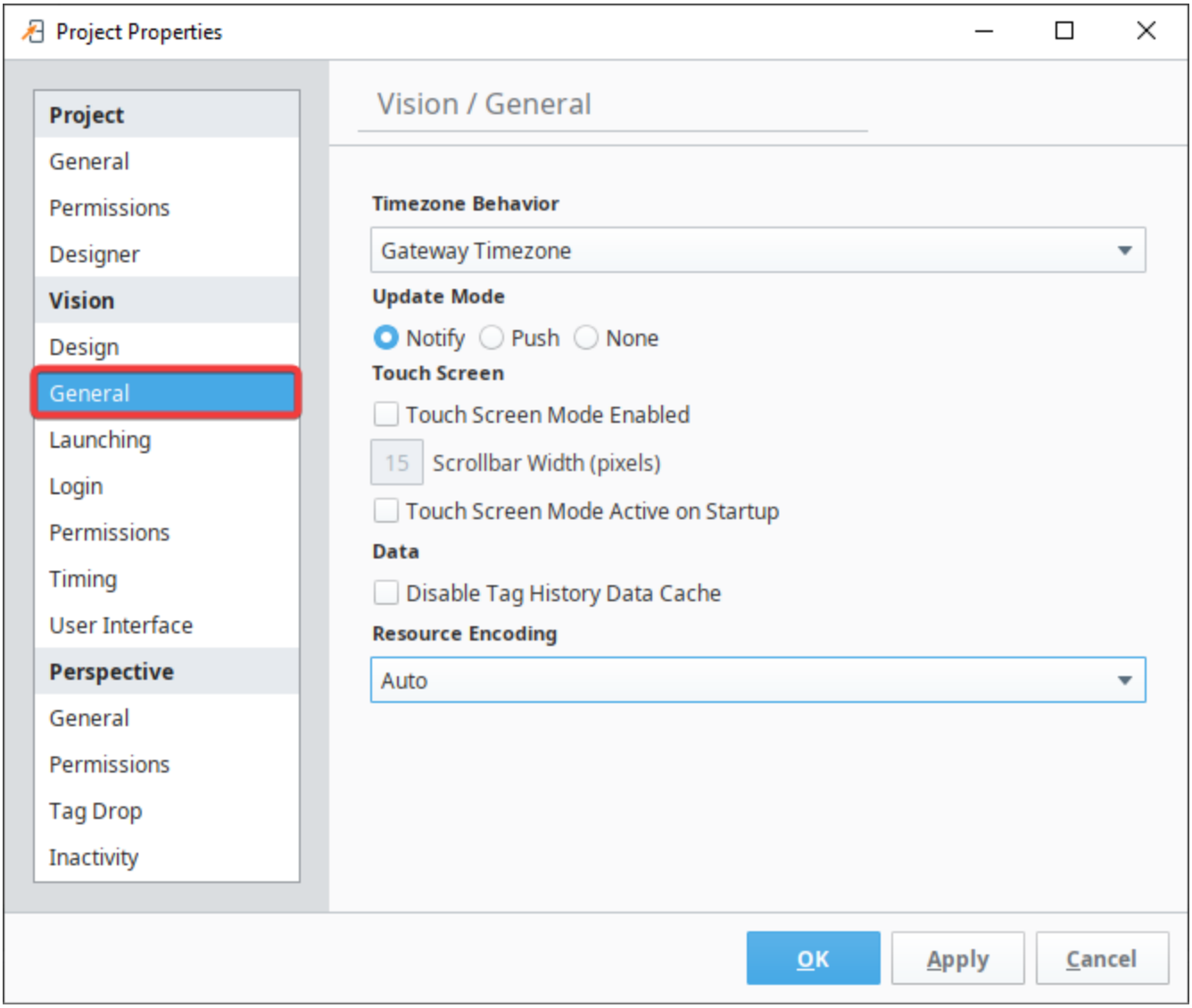
Vision Design Properties

This section of properties apply to the Vision Client in general.

Window & Template Committing	
Property	**Description**
Commit on Close	**Prompt** - Prompt whether the user wants to commit changes when closing a window or template. **Always** - Always commit changes when closing a window or template.

Template Auto Commit	**On** - Always commit changes when switching to a different window or template. **Off** - Do not automatically commit changes when switching to a different window or template.
Component Manipulation	
Property	**Description**
Constrain to Parent Container Bounds	Disabling the constraint on parent bounds allows you to position components outside of their parents bounds, which can be helpful in advanced layouts.
Nudge Distance	The number in this box is the distance (in pixels) that a nudge moves (when using the arrow keys) or resizes a component.
Alt-Nudge Distance	The number in this box is the distance (in pixels) that an alt-nudge moves (when using the arrow keys plus the Alt key) or resizes a component.
Default Component Layout	
Property	**Description**
Layout Mode	**Relative** - All newly created components will be configured with a Relative layout, further configured by the **Relative Layout Options** property. **Anchored** - All newly created components will be configured with an Anchored layout, further configured by the **Anchored Layout Options** property.
Relative Layout Options	When the Layout Mode property is set to **Relative**, these options determine the layout options of new components. For more information, refer to Component Layout.
Anchored Layout Options	When the Layout Mode property is set to **Anchored**, these options determine the layout options of new components. For more information, refer to Component Layout.
Default Color Mapping	The initial color mapping when configuring a new **number-to-color** binding.

Vision General Properties



General	
Property	**Description**
Timezone Behavior	The Vision Client can emulate any timezone. By default, it will appear to be in the same timezone as the **Gateway Timezone**. This has the effect of all Clients behaving the same, regardless of the timezone setting on the Client's host operating system. Depending on your project's requirements, this may not be optimal. You can have the Client use the host's timezone by choosing the **Client Time zone** option, or you may specify any explicit timezone for all Clients to emulate.

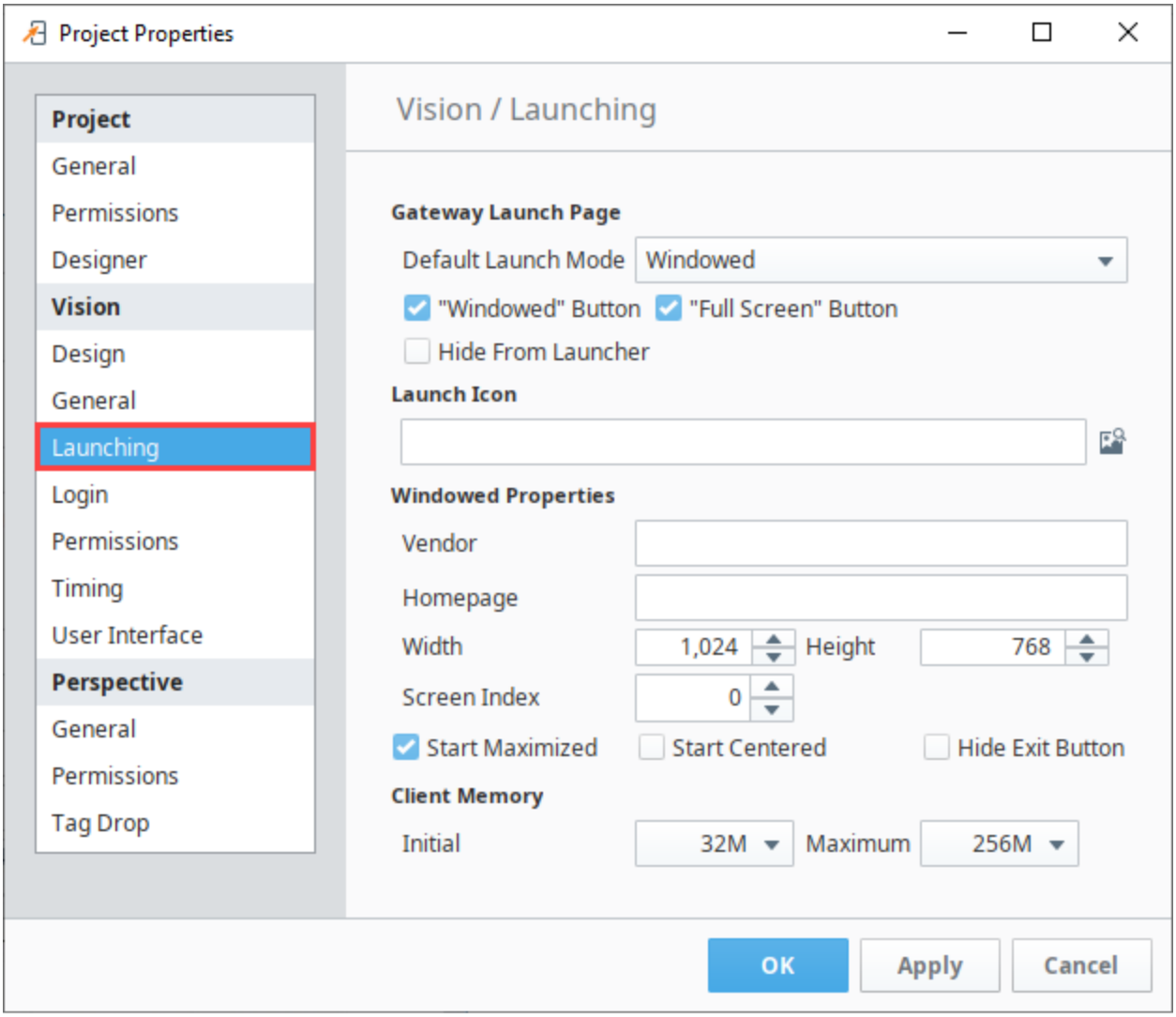
Update Mode	This feature was changed in Ignition version **8.1.24**:
	
	Updates sent to the Client when a Designer saves the projects can push updates in three ways:
	
	• Notify
	  ◦ This option will notify the user that there is a new Vision Client project update by showing an update banner at the top of the Client.
	• Push
	  ◦ This option pushes any updates to the Vision Client without warning the users.
	• None
	  ◦ Introduced in Ignition 8.1.24, this option does not push updates, but instead allows users to update Vision Client projects via a Client System Tag and companion scripting function. See Client Update Modes.

Touch Screen	

Property	Description
Touch Screen Mode Enabled	All Clients can operate in Touch Screen mode. When enabled, clicking on editable numeric and text entry fields (i.e., Text Fields, Numeric Text Fields, etc.) will pop up on-screen keyboards that can be used for data entry. You can optionally set the width of any scrollbars (number of pixels wide/tall).
Touch Screen Mode Active on Startup	Configures the Clients to start up with the Touch Screen mode active. More details can be found on the Using Touch Screen Mode page.
Data	
Property	**Description**
Disable Tag History Data Cache	The Clients normally maintain a cache of data retrieved from Tags History, improving repeat operations on graphs and tables. When this option is disabled, no data is cached, and the full queries execute against the Gateway each time data is required.
Resource Encoding	The following feature is new in Ignition version **8.1.25** Click here to check out the other new features The encoding format that Vision Windows, Templates, and Template Drop Target configurations should use when saving, helping to make them more accessible and human readable. This project property is anterograde, meaning new serializations will be affected, but old serializations will not. Options available are: • Auto (defaults to binary) • Binary • XML (plain text XML)

Vision Launching Properties

These properties affect the Vision Client's launching process.

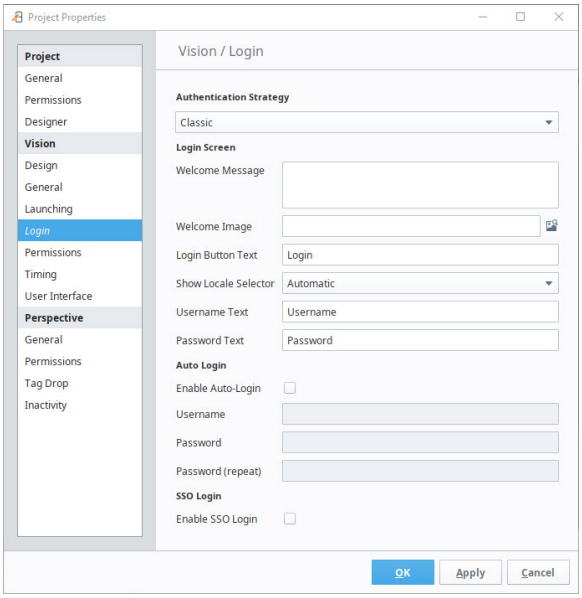


Gateway Launch Page	
Property	**Description**
Default Launch Mode	Determines the mode for a Client launched from the **Launch** button that appears next to the project in the Client Launcher. Available modes are Windowed or Full Screen.
Windowed Button/Full Screen Button	Each launch mode can also be enabled individually, which allows that mode to appear in the Dropdown list next to the **Launch** button on the Gateway **Home** page. This feature was changed in Ignition version **8.1.13**: These settings were removed in 8.1.13. The Window Modes settings on applications in the Vision Client Launcher provide similar functionality.
Hide From Launcher	This option hides the project from the Client Launcher and prevents the project from being selected in the Vision Client Launcher.
Launch Icon	
Property	**Description**
Launch Icon	The image specified here is used to represent the project on the launch page and desktop shortcut. This needs to be a path to an image that has been uploaded to the Gateway . Use the browse button to choose or upload a new image.

	The following feature is new in Ignition version **8.1.12** Click here to check out the other new features Supported image formats now include PNG, SVG, GIF, and JPEG files.
Windowed Properties	
Property	**Description**
Vendor	This property is displayed as the project is launching through Java Web Start, as well as in the JWS application manager.
Homepage	A URL that is displayed in the JWS application manager.
Width	The width of the Client window when launched in Windowed mode.
Height	The height of the Client window when launched in Windowed mode.
Screen Index	The number here is the index of the screen to use for Full Screen mode, it starts at 0. This property is not supported on all operating systems.
Start Maximized	If the box is selected, when the Client is launched in Windowed mode, it starts maximized. Note that this is not the same thing as Full Screen mode, which is only available when the Client is launched in Full Screen mode. In Full Screen mode, the width, height, and start maximized properties have no effect. When launched in Full Screen mode, the user is given an **Exit** button on the login screen by default. For terminals where the application should not be exited, this button can be removed by checking the **Hide Exit Button** box .
Start Centered	If the box is selected, when the Client is launched in Windowed mode, it starts centered.This property is ignored if **Start Maximized** is enabled.
Hide Exit Button	If the box is selected, when the Client is launched in Full Screen mode, the exit button is hidden to prevent the application from closing.
Client Memory	
Property	**Description**
Initial	Governs how the Client use RAM resources on its host machine. The initial memory setting is how much memory the Client will require on startup. While this is typically left alone, boosting it a bit can improve performance somewhat.
Maximum	Governs how the Client use RAM resources on its host machine. The maximum memory setting sets a cap on how much memory the Java VM is allowed to use. When you launch a Client on a machine with plenty of RAM, you'll also need to boost this setting to allow the Client to use more RAM.

Vision Login Properties

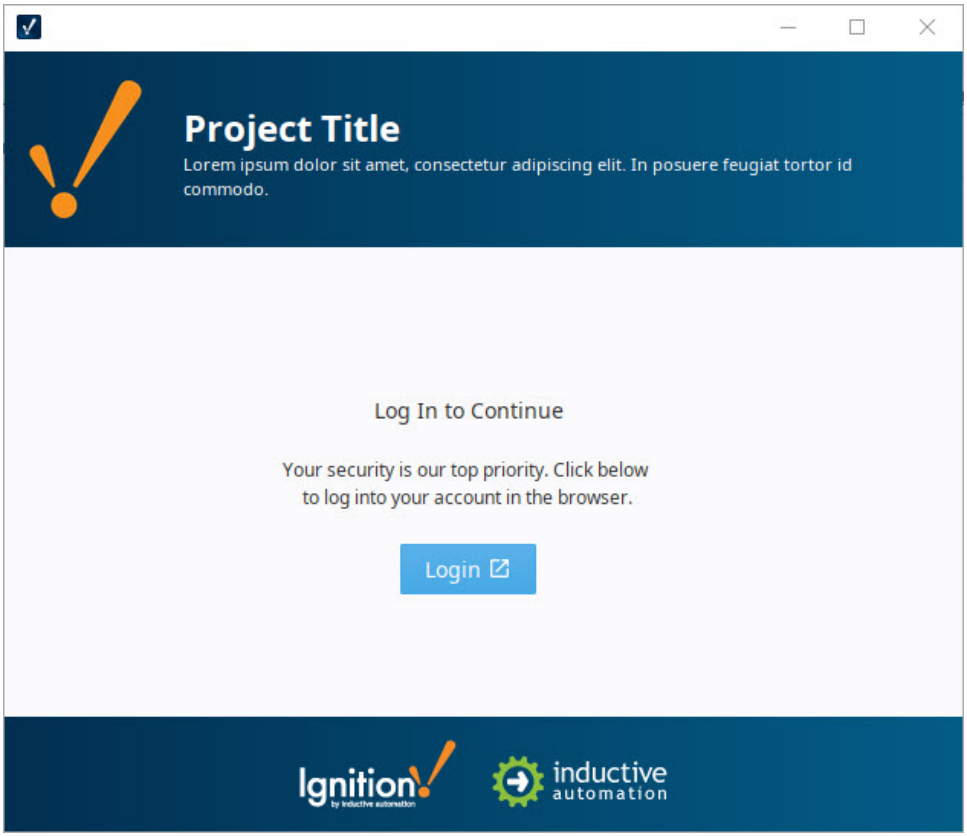
These properties affect how the Vision Client's login process behaves and appears.



<table>
<tr><th colspan="2">Login Screen</th></tr>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Authentication Strategy</td><td>
The following feature is new in Ignition version 8.1.0
Click here to check out the other new features
Authentication strategy setting. Options are Classic or Identity Provider.
<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>Classic</td><td>User Source system that works as it did before release 8.1.</td></tr>
<tr><td>Identity Provider</td><td>Identity Provider mode uses the system Identity Provider (IdP) to log into the Vision Client.</td></tr>
</table>
</td></tr>
<tr><td>Welcome Message</td><td>The message that appears in the upper-left corner of the Login screen. If left blank, no message is displayed. (HTML formatting is allowed).</td></tr>
<tr><td>Welcome Image</td><td>The image that appears in the upper-left corner of the Login screen. If left blank, uses a default image. Images are resized/forced to fit into a square format. If you use a more rectangular image, the scaling on the image will automatically be adjusted.

 
</td></tr>
<tr><td>Login Button Text</td><td>The text that appears on the Login button.</td></tr>
<tr><td>Show Locale Selector</td><td>Determines if the Locale Selector should appear on the Login screen. This property interacts with Ignition's Localization system.

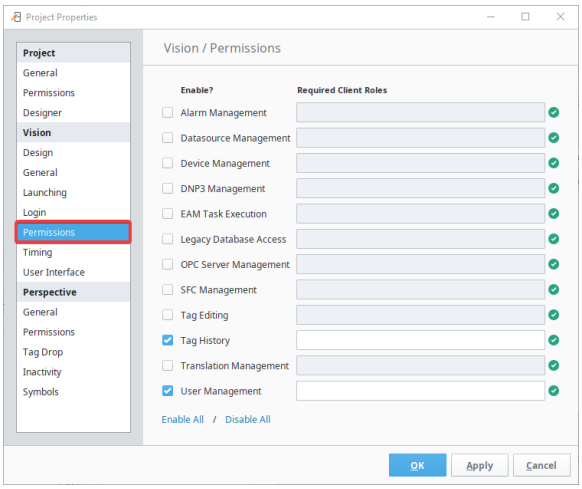
• Automatic - Show the selector if more than a single Languages exists in the project.
• Show - Always show the selector, regardless of how many Languages exist.
• Hide - Never show the selector.</td></tr>
<tr><td>Username Text</td><td>(Classic authentication strategy only) The text that appears next to the username field.</td></tr>
<tr><td>Password Text</td><td>(Classic authentication strategy only) The text that appears next to the password field.</td></tr>
</table>

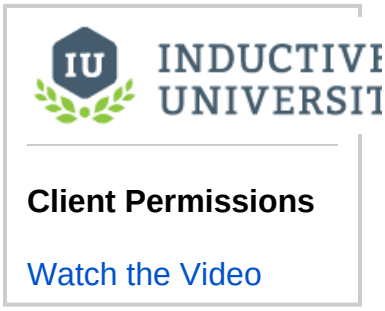
Login Prompt	The following feature is new in Ignition version **8.1.17** Click here to check out the other new features (*Identity Provider strategy only*) Prompt that appears on the Login screen when the Authentication Strategy is set to Identity Provider. Default is "Log In to Continue."
Login Message	The following feature is new in Ignition version **8.1.17** Click here to check out the other new features (*Identity Provider strategy only*) Message that appears on the Login screen when the Authentication Strategy is set to Identity Provider. Default is "Your security is our top priority. Click below to log into your account in the browser." 
Auto Login	
Enable Auto-Login	(*Classic authentication strategy only*) By enabling auto-login, you can have the launched Client skip the login process. The Client will log in behind the scenes using the credentials supplied here. If they fail, the login screen will be presented. **Property** / **Description** **Username** — User name of the user to automatically log in when Client is launched. **Password** — Password of the user to automatically log in when Client is launched. **Password (repeat)** — Password of the user to automatically log in when Client is launched. See also Setting Up Auto Login.
SSO Login	
Enable SSO Login	(*Classic authentication strategy only*) Enable Single Sign-On for the project. The Project's default Authentication Profile must use Active Directory, and SSO must be enabled in the Profile. See Active Directory Authentication for more details.
Auth Token	

Inactivity Timeout	The following feature is new in Ignition version **8.1.24** Click here to check out the other new features (*Identity Provider strategy only*) The number of minutes which must elapse before expiring a designer user's auth token due to inactivity caused by a disconnected session. Must be greater than zero. Default value is 10.
Time-To-Live (TTL)	The following feature is new in Ignition version **8.1.24** Click here to check out the other new features (*Identity Provider strategy only*) The maximum number of minutes a designer user's auth token may exist before it expires. If set to any number less than or equal to zero, auth tokens may live forever, as long as the auth token has not expired due to inactivity.

Vision Permissions Properties

These properties allow you to limit the Client's ability to perform certain tasks. The tasks are grouped by category. Access can be configured statically for all users, or require specific roles.





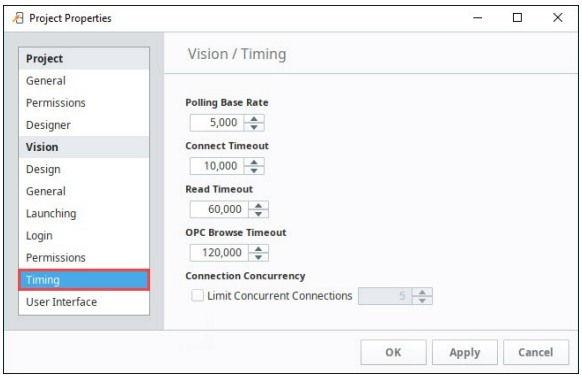
Permissions	
Property	**Description**
Enable?	Determines if the Client has access to this category. If unselected, the category will be disabled in the Client for all users. If selected, the **Required Client Roles** text field will also be used to determine if the Client has access.
Required Client Roles	A **comma separated list** of Managing Users and Roles and/or Security Zones that have access to the category while the **Enabled** ch eckbox is selected. While these fields have focus, the **down arrow key** on the keyboard may be pressed to make a list of available roles appear. There are several ways to use roles and zones in this field, and they can be used together in the comma separated list. • **Blank Field** - anyone can use this, regardless of their roles. • **Role name** - users with this role can use this regardless of their zone. • **roleName@zoneName** - users must have this role AND be logged in from this zone. **Caution:** When creating a new project, all of these settings will be disabled by default.
Category Descriptions	The following is a list of the initial categories. Note that the categories you see in your Designer are dependent on which modules are installed on the Gateway. Additionally, third-party modules can add to this list.

Category	Description

Alarm Management	Allows the Client to cancel, shelve, and acknowledge alarms. Applies to both local and remote Alarms.
Datasource Management	Allows the Client to modify Gateway datasource connections.
Device Management	Allows the Client to modify device connections.
DNP3 Management	Allows the Client to freeze DNP3 operations via scripting.
Legacy Database Access	Allows Clients to run queries directly against the database. This doesn't effect named queries.
OPC Server Management	Allows the Client to modify OPC server connections.
SFC Management	Allows the Client to start or stop Sequential Function Charts.
Tag Editing	Allows the Client to add, edit, or delete Tags through scripting.
Tag History	The following feature is new in Ignition version **8.1.21** Click here to check out the other new features Allows the Client to query/modify Tag History.
Translation Management	Allows the Client to modify translations in the localization system.
User Management	Allows the Client to modify schedules, holidays, and users through scripting or components.

Vision Timing Properties

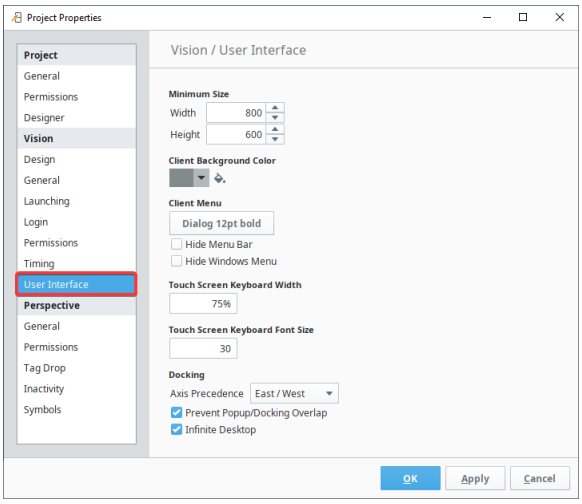
These properties affect the Vision Client polling rate and timeout settings.



Timing	
Property	**Description**
Polling Base Rate	The base rate, in milliseconds, for all polling bindings.
Connect Timeout	The maximum amount of time to wait for connections to the Gateway to be established. Specified in milliseconds.
Read Timeout	The maximum amount of time for socket connection to the Gateway to remain open. Specified in milliseconds.
OPC Browse Timeout	Maximum amount of time, in milliseconds, to wait for the response to a request. (default: 120,000)
Connection Concurrency	By default, Clients are not limited by the number of concurrent connections to the Gateway. These connections are used to send Tag writes, return database results, as well as any other action that requires information to be passed between the Gateway and the Client . Depending on what is running in the Client, your network's bandwidth could be hindered. Enabling this property will limit the amount of concurrent connections the Client can maintain. Note that this may negatively impact Client performance, but is usually preferable on busy networks.

Vision User Interface Properties

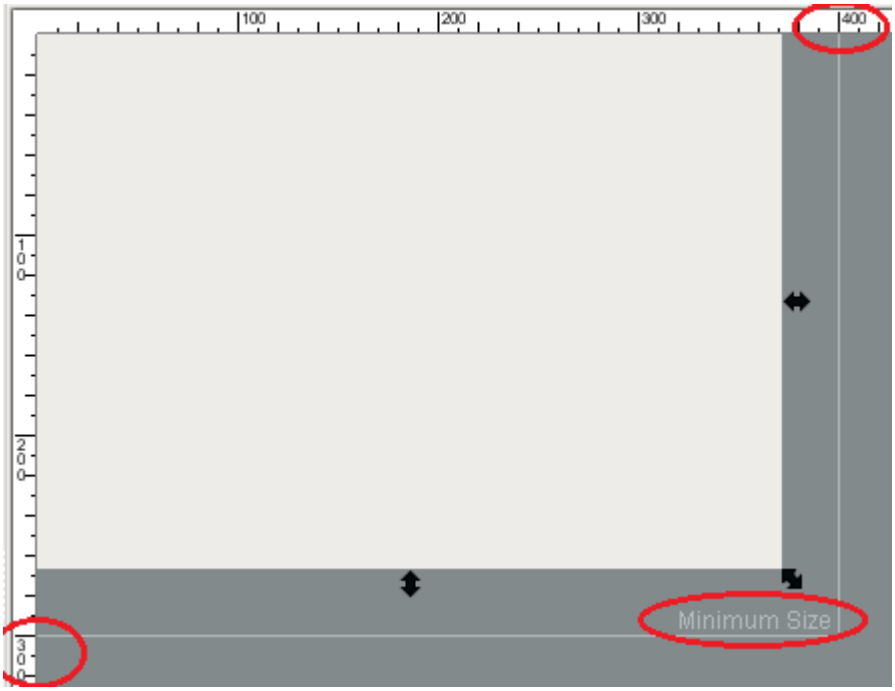
These properties affect how the Vision Client appears and behaves while it is running.



INDUCTIVE UNIVERSITY

Setting Client Minimum Size

Watch the Video

User Interface	
Property	**Description**
MInimum Size	Typically, a Vision Client is designed to run on multiple different resolution and sizes of monitors. The various component layout features help design elastic screens, but sometimes you need to set a lower bound as to how small you'll allow the Client's usable area to shrink. This is what the Minimum Size settings are for. You can see these settings visually represented in the Designer as lines on the Vision workspace when the Root Container is smaller than the configured Minimum Size.. Whenever the usable space shrinks smaller than these bounds, scrollbars will appear, capping the **Width** and **Height** to these minimums. This defaults to 800 x 600. In the image below, the project was set to a minimum size of 400 x 300. Since the window is smaller, the outline is visible. 
Client Background Color	This option allows you to specify the color of the Vision workspace which will be visible when not obscured by windows.
Client Menu	
Property	**Description**

Menu Font	Changes the font type, font style, and font size of the Menu Bar.
Hide Menu Bar	Hides the entire Menu Bar in the Client. Usually enabled in situations where users should not be able to close the client. Changes to this setting are applied on client startup, meaning clients will need to be relaunched after changing this setting.
Hide Windows Menu	Hides the automatically-generated "Windows" menu that lets users switch between open windows. Enabled when users should not be able to close windows in the Client. See also Menubar Scripts .
Touch Screen Keyboard Width	Determines how wide the Touch Screen Keyboard should appear in the Client. Percentage of the client window the touch screen keyboard displays. The following feature is new in Ignition version **8.1.34** Click here to check out the other new features You can also use the `-Dignition.touchscreen.keyboardWidth` system property for local overrides.
Touch Screen Keyboard Font Size	The following feature is new in Ignition version **8.1.34** Click here to check out the other new features Customize touchscreen keyboard font size. You can also use the `-Dignition.touchscreen.keyboardFontSize` system property for local overrides. Default is 30.
Docking	
Property	**Description**
Axis Precedence	Defines which axis takes precedence for docked windows. (i.e., East/ West or North/South). When windows are docked on adjacent sides, this property determines which sides should take precedence. When set to "East/West", windows docked to the East or West sides will expand vertically from the top to the bottom of the Client , and will push any North or South docked windows out of the way.
Prevent Popup /Docking Overlap	By default, popup windows are not allowed to overlap any docked window. Disabling this property will allow Popup windows to be placed on top of docked windows in the client.
Infinite Desktop	If true, the desktop area will be expanded if floating windows are dragged out of frames. If false, popups are prevented from being dragged beyond the bounds of a window so they don't get distorted.

Related Topics ...

- Client Update Modes
- Using Touch Screen Mode
- Working with Components
- Docked Windows - Axis Precedence

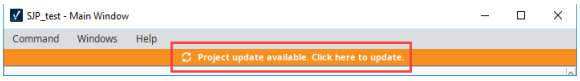
In This Section ...

Client Update Modes

Notify Versus Push Versus None

When a Client is launched, the most recent version of the project is used. Vision Client projects can receive updates in three modes:

- **Notify**
  This is the default mode to automatically notify operators when project updates are available. In this mode, every time you update a project in the Designer and save the changes, Clients will display an orange information bar at the top of their display. This orange bar notifies the operator that an update is available. B**y clicking on the notification banner, the new project modifications are downloaded and applied.**



- **Push**
  When you save your changes in the Designer, this mode automatically pushes all project changes and updates to all running clients with no operator interaction, that is, the new version is downloaded and applied automatically. This is often desirable when a Client is running in a situation where keyword and mouse access is inconvenient, such as in a large overhead display.

- **None**

  The following feature is new in Ignition version **8.1.24**
  Click here to check out the other new features

  Selecting this option will defer updates to any currently active Vision Clients. The Vision Client project will update upon restart of the active Vision Client, or through using the system.vision.updateProject() system function. This option may be preferred when trying to avoid interruptions to currently active users or Vision Clients.

Note: This property (and many other project properties) are only read when a client is launched. When changing from one Update Mode to another, client updates will not get pushed when you save. Clients will need to be re-launched to start using the new Update Mode.

On this page ...

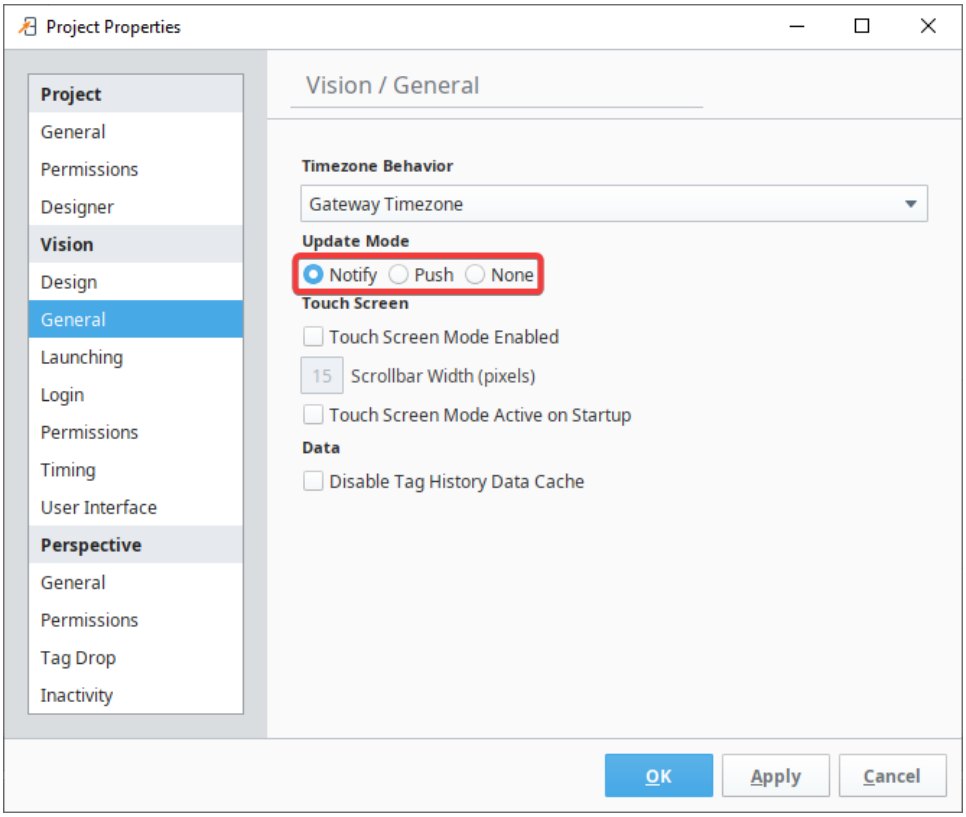
- Notify Versus Push Versus None
  - Setting Client Update Modes

Setting Client Update Modes

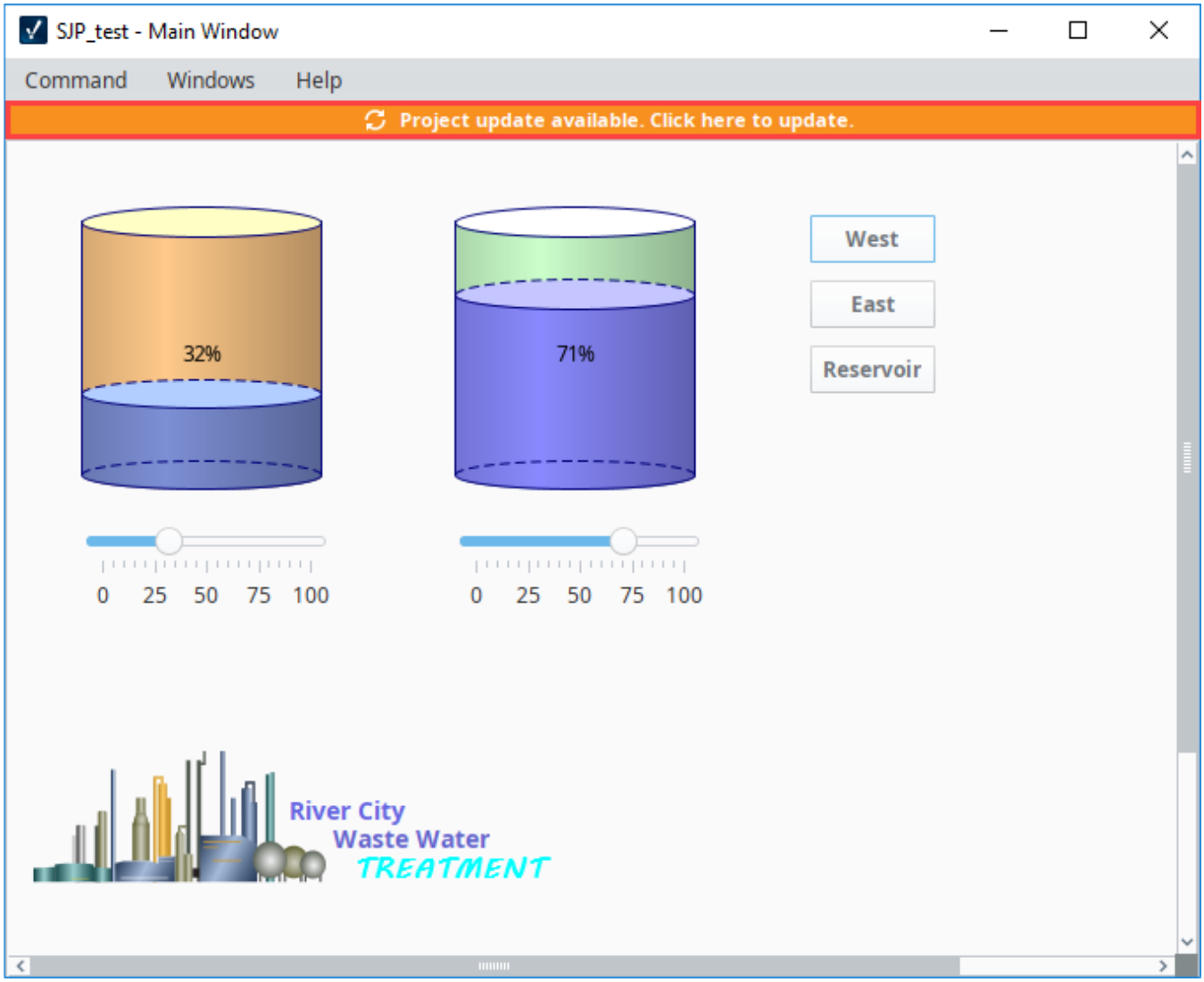
This example shows how to change the Client Update Mode.

1. In the Project Browser of the Designer, select **Project > Properties** from the top menubar. The **Project Properties** window will open.
2. Under the **Vision** folder, select **General**, set **Notify**, **Push**, or **None** as your Client Update Mode. The default is set to **Notify**.

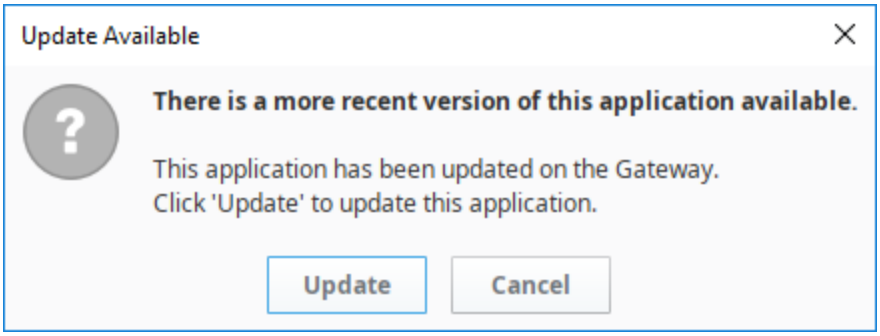
3. Click **OK.**


Project Properties
Project
General
Permissions
Designer
Vision
Design
General
Launching
Login
Permissions
Timing
User Interface
Perspective
General
Permissions
Tag Drop
Inactivity
Vision / General
Timezone Behavior
Gateway Timezone
Update Mode
Notify
Push
None
Touch Screen
Touch Screen Mode Enabled
15
Scrollbar Width (pixels)
Touch Screen Mode Active on Startup
Data
Disable Tag History Data Cache
OK
Apply
Cancel

Now, in the Client (with Update Mode set to **Notify**), the operator will have to click the banner to update the Client.



Then, the operator must click **Update** on the confirmation message.



Now the client version is updated with the most recent version of the application.

Related Topics ...

- Using Touch Screen Mode
- Vision Project Properties

Setting Up Auto Login

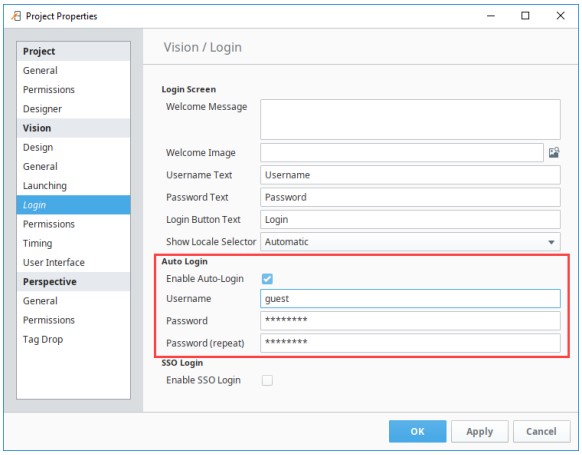
Client Auto Login

Clients can log in automatically when launched, once you specify the **Auto Login** settings in the **Client /Login** section of the **Project Properties** window. This is typically used in situations where the client should login with a low access user, such as a 'guest' account.

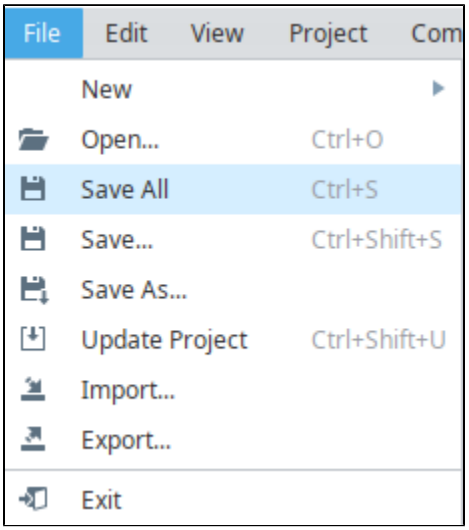
Set the Clients to Automatically Login

In this example, we will set the project properties to allow Ignition clients to automatically login.

1. In Designer, go to **Project > Properties**.
   The **Project Properties** window is displayed.
2. Scroll down to **Vision > Login**.
3. Click the **Enable Auto-Login** checkbox.
4. Enter a Username and Password.
5. Click **OK.**



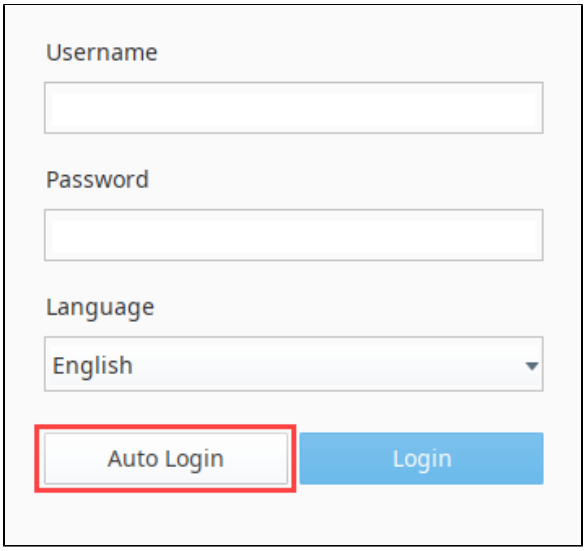
6. Select the **File > Save All** to save your changes to the Gateway and your project.



When in a client that has auto login set up, if the user accidentally logs out there is a button on the login screen to allow the user to enter in the auto login credentials. That way there is no need for them to know any login credentials.

On this page ...

- Client Auto Login
  - Set the Clients to Automatically Login



Related Topics ...

- Gateway Backup and Restore

Using Touch Screen Mode

It is very common to deploy Ignition Vision projects on touchscreen computers, such as industrial panel-PCs acting as Human Machine Interface (HMI) or Operator Interface Terminal (OIT). In situations where the PC does not have a keyboard attached, Touch Screen mode can be used to assist with user input. For this reason, all of the Input components in Vision can be enabled for touch screen.

Under normal circumstances, you don't have to do anything special other than enable Touch Screen mode on your project. This will allow the operator to activate Touch Screen mode when they log in. You can also enable Touch Screen mode via scripting.

Touch Screen-enabled **Input** components all have an expert level property called **Touch Screen Mode**. This property has three settings:

- **Single-Click**: The keyboard will appear on a single click
- **Double-Click**: The keyboard will only appear after a double-click
- **None**: Disable touch screen support on the component. The component will no longer invoke the touch screen keyboard.

On this page ...

- Enabling Touch Screen Mode
- Invoke the Touch Screen Keyboard with Scripting
- Change the Size of the Touch Screen Keyboard

Using Touchscreen Mode

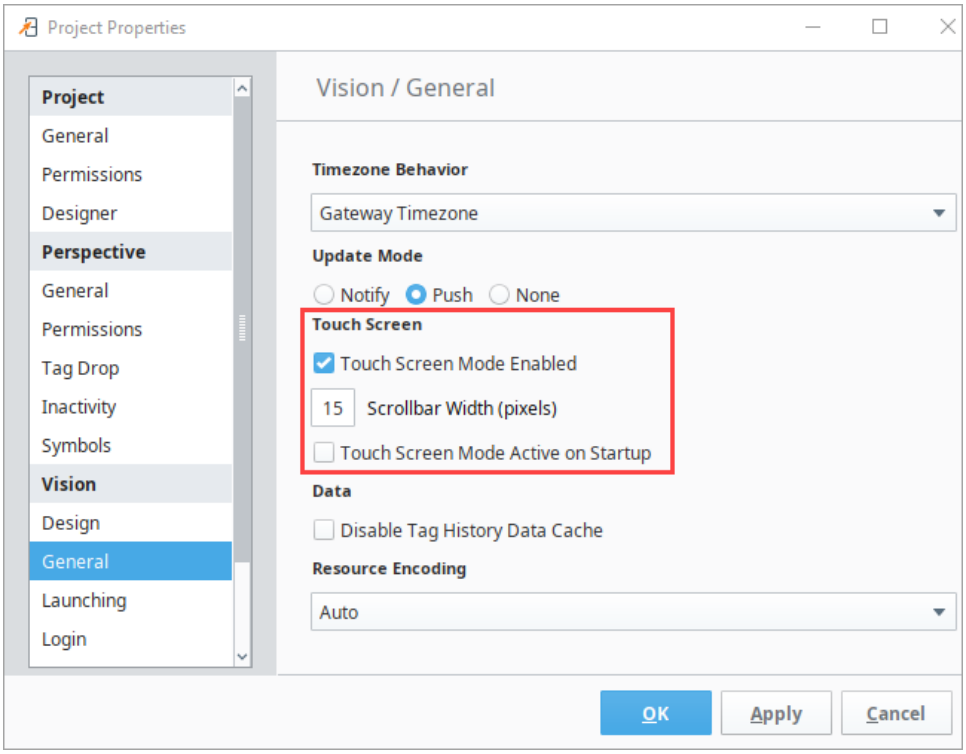
Watch the Video

Enabling Touch Screen Mode

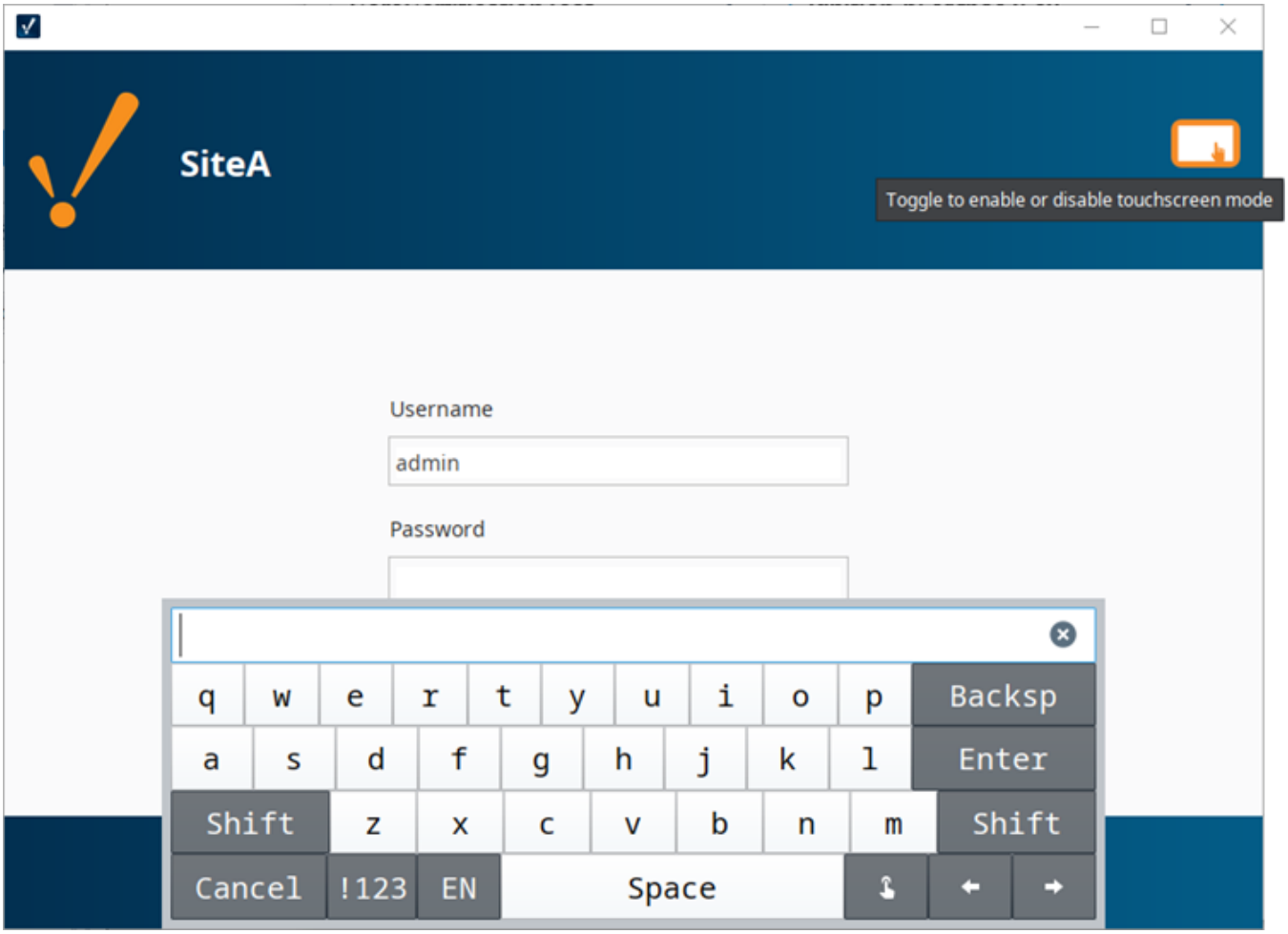
Touch screen support is built into Ignition. Turn it on through the Project Properties or scripting in the Designer.

1. From the Project Browser, click on the Project Properties icon, or from the menubar go to **Project > Project Properties**. The **Project Properties** window is displayed.
2. Scroll down to **Vision > General** page, to see the **Touch Screen** options.
   - **Touch Screen Mode Enabled:** By default, this is enabled, which means an operator can activate the mode on the Startup screen. All Clients can operate in touch screen mode. When Touch Screen mode is enabled, clicking on numeric and text entry boxes will pop up on-screen keyboards that can be used for data entry. You can optionally set the width of any scrollbars (number of pixels wide/tall)
   - **Touch Screen Mode Active on Startup:** This option sets the Clients to start up with the Touch Screen mode active.
3. Change the settings as desired then click **OK**.

> **Note:** Make sure Comm Read/Write is enabled in Project Properties of the Designer.



These settings are helpful for mixed-use projects, that is, those that are launched on both touch screen devices and traditional computers and laptops. Once Touch Screen mode is enabled through Project Properties, the Touch Screen icon will appear on top right corner of the project login screen. Click in any of the login fields to bring up the Touch Screen keyboard.



Invoke the Touch Screen Keyboard with Scripting

To handle touch screen logic via scripting, the general pattern is to respond to a mouse event, popup up a keyboard, and then set the component's value to whatever was entered in the keyboard. For example, for a text field, you would write a script like this:

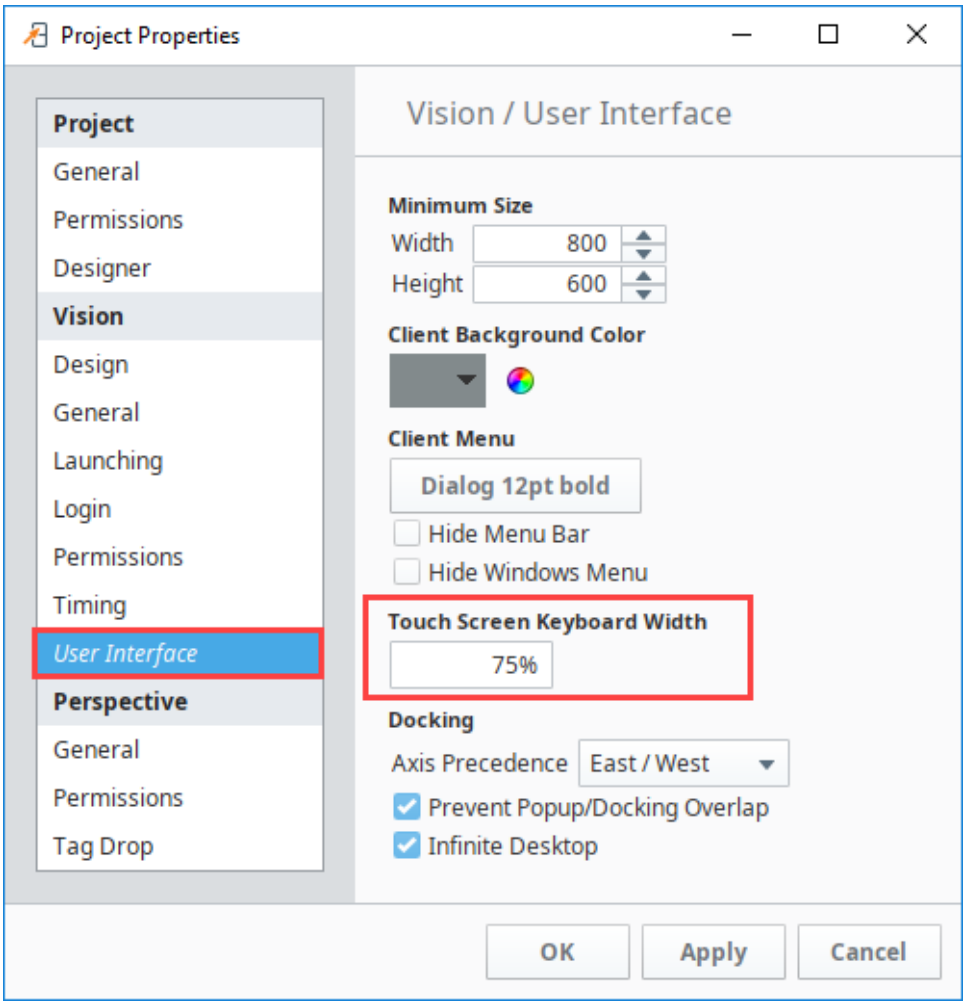
```
if system.gui.isTouchscreenModeEnabled():
        currentText = event.source.text
        newText = system.gui.showTouchscreenKeyboard(currentText)
```

See also system.gui.setTouchscreenModeEnabled.

Change the Size of the Touch Screen Keyboard

You can control the size of the Touch Screen keyboard that is displayed in a Vision client.

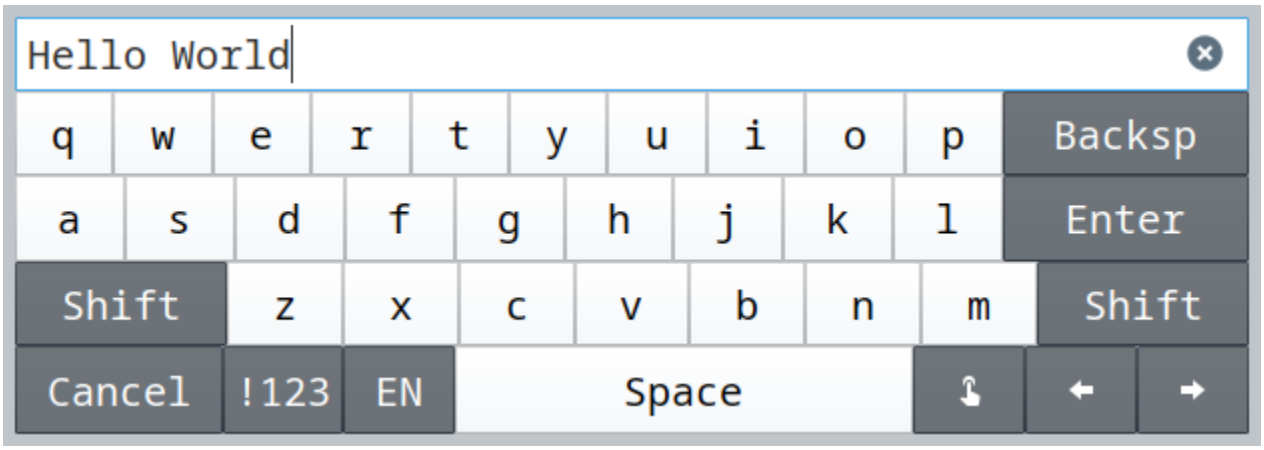
1. In the Designer, go to **Project Properties > Vision > User Interface**.
2. Change the percentage for the Touch Screen Keyboard Width property to anything you want.



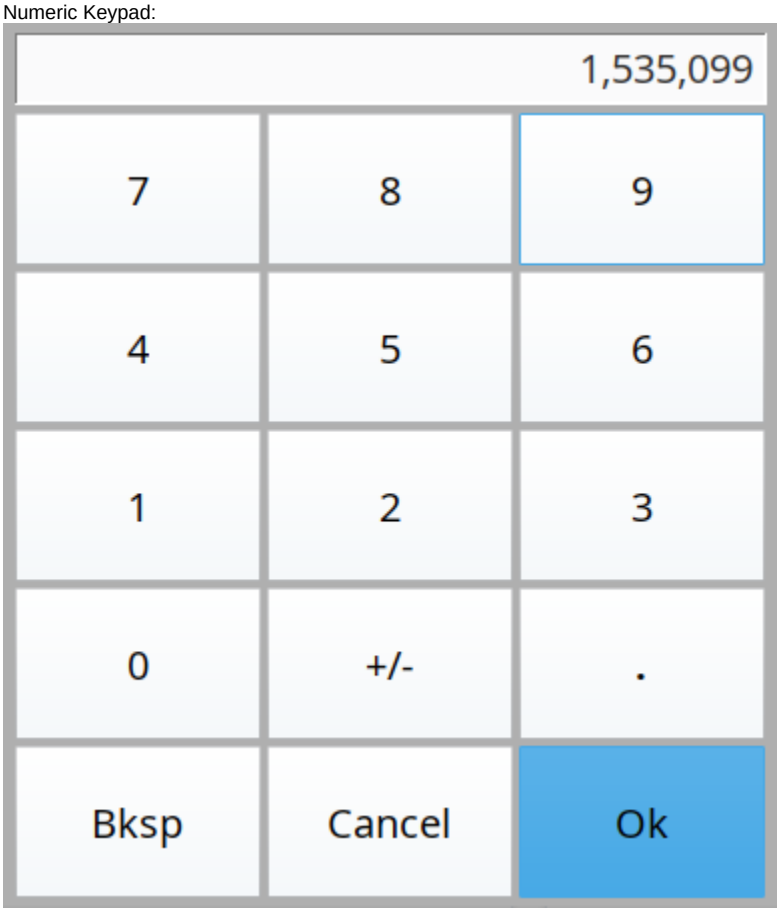
3. When you launch a Vision client, the Touch Screen keyboard will be displayed in the default percentage of 75%. After you're logged into the client, the Touch Screen keyboard will be set at the value you entered into the Touch Screen Keyboard Width property field.

The Touch Screen Keyboard Width setting controls both the alphanumeric keyboard and the numeric keypad.

Alphanumeric Keyboard:



Numeric Keypad:



Related Topics ...

- Setting Up Auto Login

Common Tasks in Vision

This section contains examples for items we've identified as "common" tasks: undertakings that many users are looking to utilize when first starting out with a specific module or feature in Ignition . Additionally, this section aims to demystify some of the more complex or abstract tasks that our users may encounter.

The examples in this section are self-contained explanations that may touch upon many other areas of Ig nition. While they are typically focused on a single goal or end result, they can easily be expanded or modified after the fact. In essence, they serve as a great starting point for users new to Ignition, as well as experienced users that need to get acquainted with a new or unfamiliar feature.

Below is a list of common tasks related to this section of the manual.

On this page ...

- Component Animation
- Custom Input Template
- Client Tags for Indirection
- High Performance HMI Techniques
- Open Dynamic Windows on Startup
- Tank Cutaways

Component Animation

Creating animation within a Vision project. The Component Animation page walks through the different ways to animate graphics on a window. The different methods vary in complexity and offer different solutions for different needs.

Custom Input Template

Creating a simple template that can be used many times to create simple user input fields. The Custom Input Template is relatively simple to put together, but can be a powerful tool that can quickly build out screens that are heavy on user input.

Client Tags for Indirection

Vision Client Tags can be used as a variable across all windows to indirectly point to a set of Tags, such as an area of the plant. For example, setting up Client Tags for Indirection can enable the user to choose an area of the plant from a Dropdown List component, and have Ignition display the correct windows for that area.

High Performance HMI Techniques

Using techniques that relay information more quickly than standard a P&ID display. High Performance HMI Techniques help create simpler screens that have much less noise and useless "fluff" that takes up screen space. The suggestions and examples help guide better practices to increase efficiency and performance across any industry.

Open Dynamic Windows on Startup

It can sometimes be necessary to Open Dynamic Windows on Startup, so that certain users see one set of windows while other users with potentially different privileges sees a different set. While it is easy to set a window to open on startup, that same window will always open for anybody that logs in to the project. By using scripting with the Client Event Startup Script, you can customize which windows get opened based on any criteria.

Tank Cutaways

The Tank Cutaways in Symbol Factory may seem like just a random shape, but they can be used to remove parts of other SVG diagrams to create a way to "look inside" the tank or other machine and view information like how much material is available.

In This Section ...

Component Animation

Animation can be a useful tool to help visualize what is happening at any given time. Animations make it easy to tell if a machine is running, or a conveyor is moving with a quick glance at the screen, and it can help highlight which components aren't currently in use. There are two main ways of creating animation: actually moving all or part of a component, or cycling through a few different static images of a component that give the illusion of moving. Each method has its advantages. For example, making a dump truck move forward and backwards may be better accomplished with actually moving the component. It would be fairly simple to make the whole component move forward on its x coordinate, and then back the same amount. But, if the truck was instead stationary, and just its bed were moving up and down to simulate the truck dumping its cargo, it may be difficult to seamlessly move the component, since it would not just be moving on one axis, but would require careful rotation and movement combined. In this instance, it would be far easier to create a few static images of the truck with its bed in various states, and then cycle through those images fairly quickly so it looks like it is moving. Let's go over both methods below.

On this page ...

- Animation in Scripting
- Components that Actually Move
- Animating a Component

INDUCTIVE UNIVERSITY

Component Animation

Watch the Video

Animation in Scripting

The best way of moving components around on the screen is using the scripting function system.gui.transform. This function can move and resize components from a Python script. It provides many options for moving and resizing components, all in one simple scripting function. This can be called on a property change when a user enters a new value, or based on other conditions that happen on the window.

Components that Actually Move

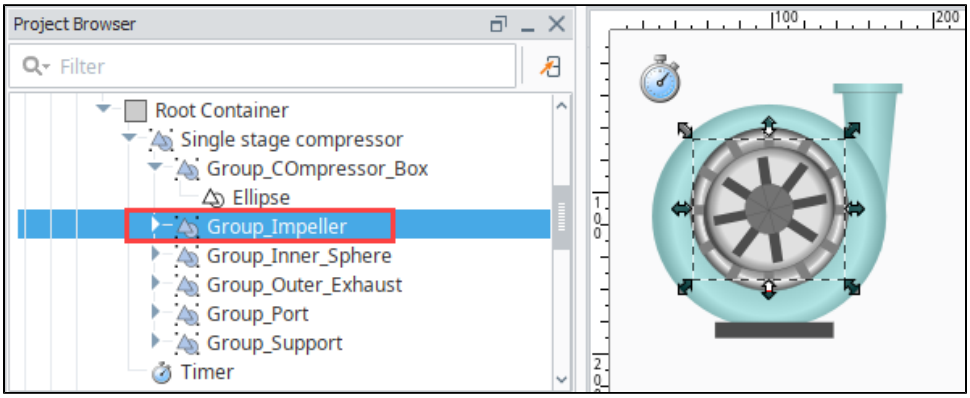
To actually move components around during runtime usually involves binding all or part of an image or SVG to the value property of a timer or signal generator. The Symbol Factory Enhanced components all contain an Angle property. Binding the Angle property of a spinning part of a motor can help us visualize when the motor is running or not.

Note:

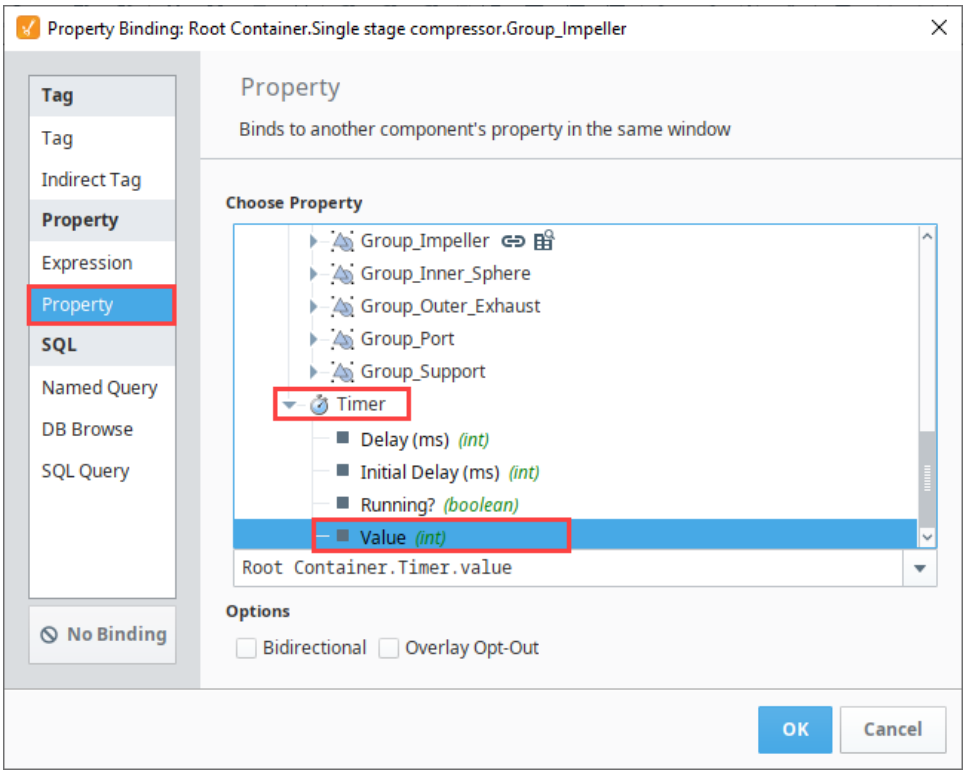
When searching Symbol Factory, make sure to select the Enhanced radio button in the search window. The enhanced symbols have groupings that enable you to more easily animate them

1. Pick out a component that you would like to animate. For this example, we used the **Single Stage Compressor** from Symbol Factory. You will also want to drag a **Timer** component onto your window.

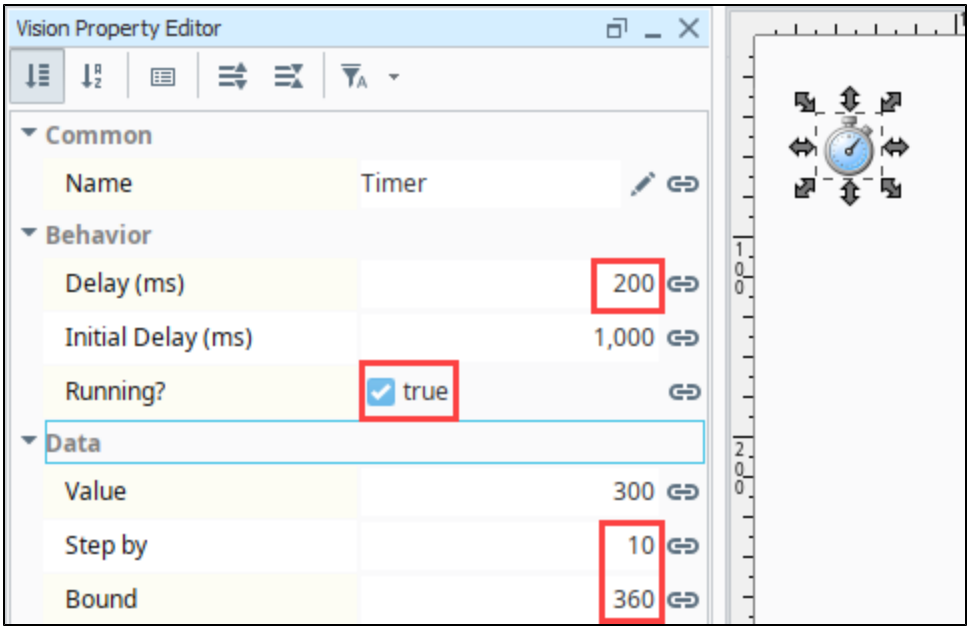
1. In the Project Browser, click the **Expand** ▶ icon to expand the Single Stage Compressor component. You will notice that the whole component is made up of many smaller pieces. We want to select the piece called Group_Impeller.



2. In the Property Editor, bind the **Group_Impeller's Angle** property to the **Value** property of the **Timer**.



3. We now need to modify the properties of the Timer to ensure a good rotation. Select the **Timer** component, and set
   Delay (ms): **200**
   Step By: **10**
   Bound: **360**
   Running?: **True**



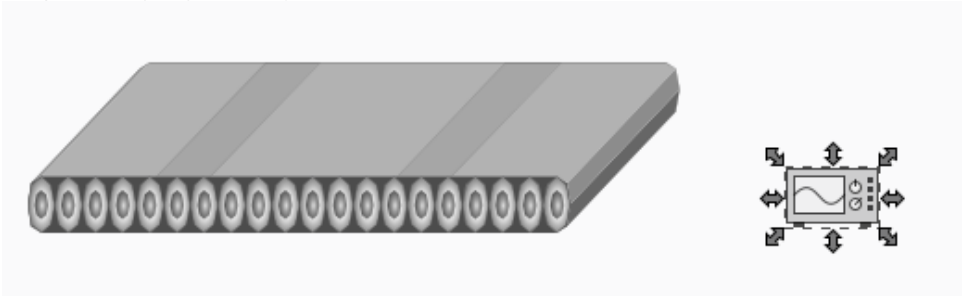
4. Put the Designer in **Preview Mode** and your Compressor symbol component will now animate. You can adjust the Delay (ms) to be lower or higher to adjust the speed at which the component rotates.

Animating a Component

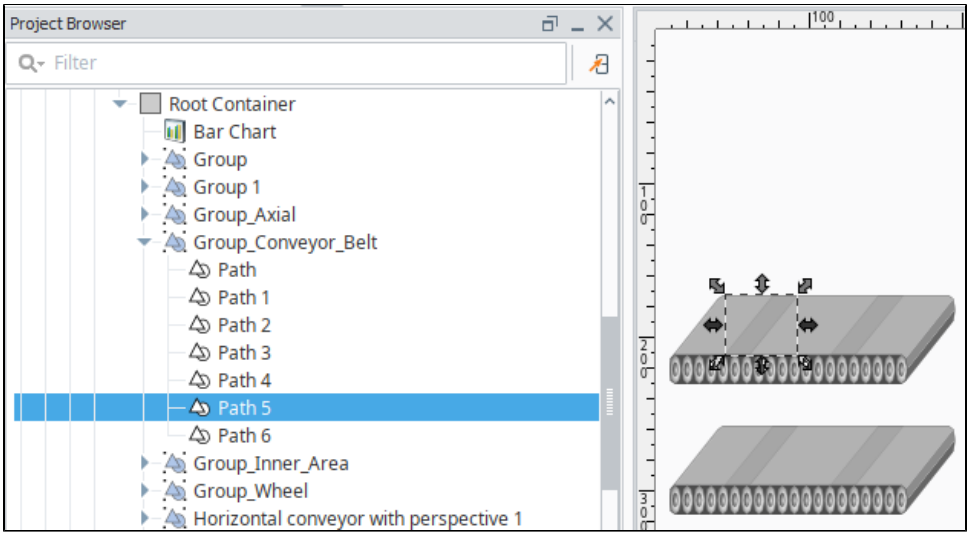
In this example, we will add some animation. We will duplicate a graphic multiple times and modify each graphic to be a little different than the others, and then show and hide them in the correct order to make it seem like they move.

The following example uses a Signal Generator component to drive the animation, but any incrementing value can be used, such as the Value property on a Timer component, an accumulating value in a PLC, or the current time in seconds.

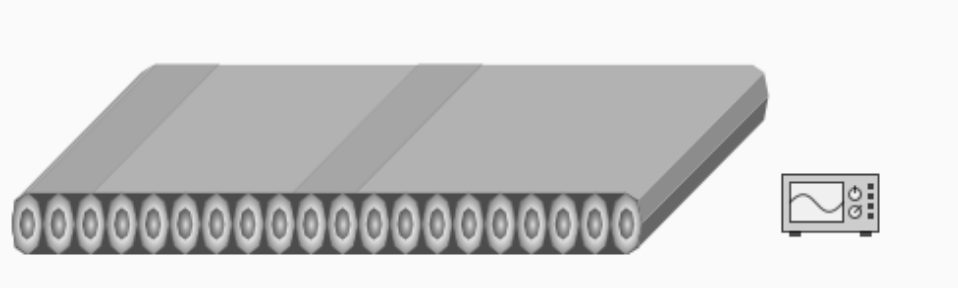
1. Pick out a component to animate. For this example, we chose the **Horizontal Conveyor with Perspective** from the Symbol Factory. We also placed a signal generator symbol on the window.



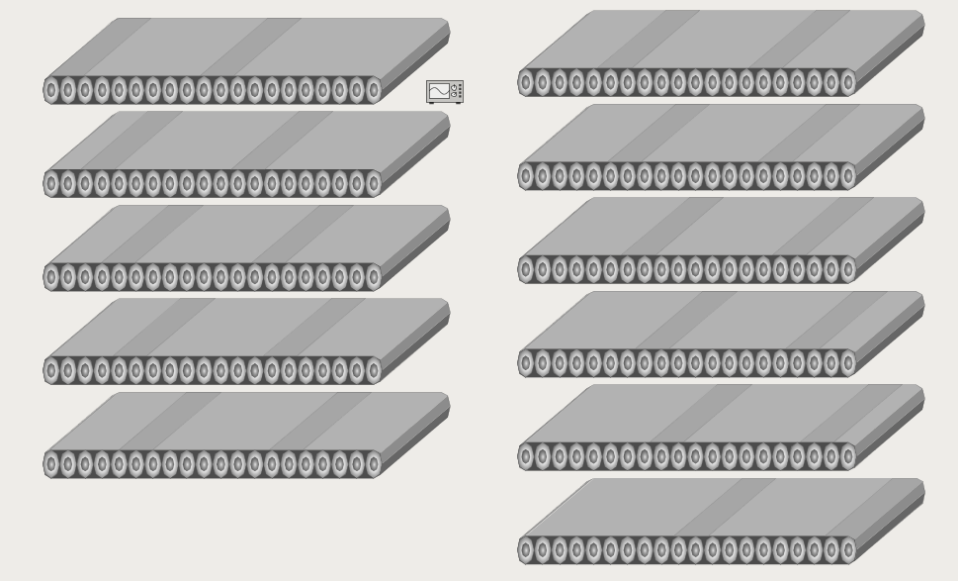
2. Duplicate this component 10 times for a total of 11 conveyors on the window.
3. Select the first instance of the component then choose **Component > Ungroup**.
4. In the Project Browser, expand the Group_Conveyor_Belt. Select **Path 5**.



5. Move **Path 5** to the left and then move **Path 4** the same distance to the left. In this example, we moved them both an equal distance left so that the leftmost component is on the left edge of the conveyor.



6. Repeat this with the rest of the components, except move them slightly to the right of the previous instance. Notice how the gray bars slowly move from left to right over all the components when viewed in sequence.



7. Select each Conveyor image individually select **Component > Group**.
8. Next, select the first Conveyor from the Project Browser (the one that has the bars on the far left), and place an expression binding on its **Visible** property that looks like this:

```
if({Root Container.Signal Generator.value} = 0, 1, 0)
```

Duplicated this across all of the conveyors, but increment the first number by **1** each time. The last conveyor (the one that has the bars on the far right) should have the following expression:

```
if({Root Container.Signal Generator.value} = 10, 1, 0)
```

9. Repeat **Step 8** across all of the conveyors, but increment the first number by **1** each time. The last conveyor (the one that has the bars on the far right) should have the following expression:

```
if({Root Container.Signal Generator.value} = 10, 1, 0)
```

10. Next, stack all of the conveyer images them all on top of each other exactly. This is easily done by selecting all of the conveyors, going into the Alignment menu, then selecting **Align Centers Horizontal**  icon and **Align Centers Vertical**  icon.
11. Select the **Signal Generator** component. Set the **Signal Type** to **Ramp**, the Period to **1000**, the Values/Period to **11**, the **Upper Bound** to **11** , and the **Lower Bound** to **0**.
12. Set the **Running?** property of the Signal Generator to **True** and it will now cycle through showing all of the conveyors, which will make it look like it is moving.
13. **Save** the project.

Custom Input Template

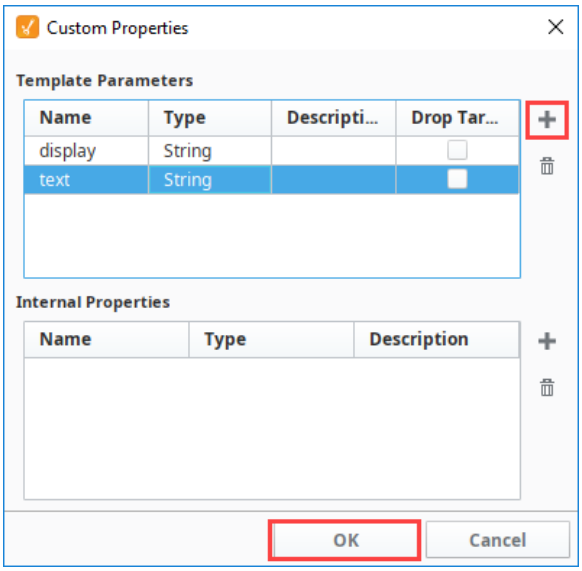
Sometimes you may need to have text fields in your project for the user to input data. Rather than copying and pasting these text fields into each window, you can create a template that includes a single label and text field. Parameters can then be passed in so that the label and text can be used for different types of input. The template can then contain an expression to validate that there is data in the text field. This template can be reused many times on multiple windows to allow users to input data.

Custom Input Template Example

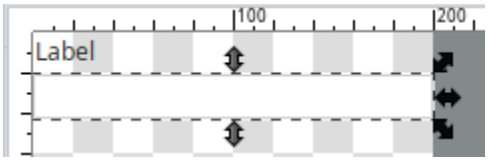
In this example, we'll create a template containing a Label and Text Field components. We'll add two parameters to be passed into the components. We will also copy the expression in the code block below to let the user know that there is data in the text field. Once you get the template created, you can copy the template to a window and test it out.

1. In the Project Browser, right-click **Templates** and select **New Template**.
2. Right-click on ***New Template*** and click **Rename** to change its name it to something meaningful such as **Text Entry**.
3. To add a parameter, right-click the checkered area and select **Customizers > Custom Properties**.
4. In the **Custom Properties** window, add two **Template Parameters** by clicking the Add ✚ icon twice and entering the following:

   1st Parameter
   Name: **display**
   Type: **String**
   2nd Parameter
   Name: **text**
   Type: **String**



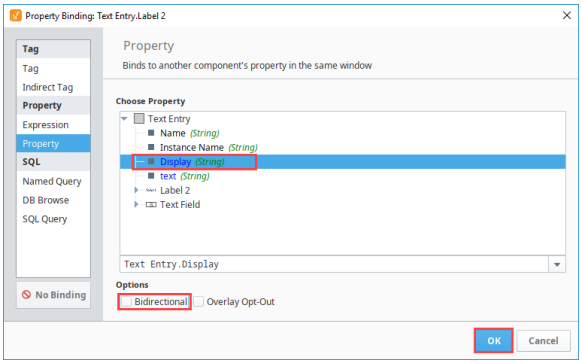
5. Click **OK.**
6. From the Component Palette, drag a **Text Field** and a **Label** to the checkered area of the template, resizing so that the label and the text box occupy the majority of the space.



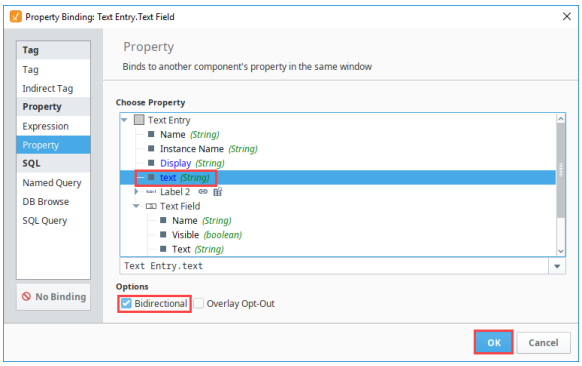
7. Select the **Label** component, go to the **Property Editor**, and click on the **Binding** 🔗 icon of the **Text** property. The Property Binding window is displayed.
8. In the **Property Binding** window, click on the **Property** binding type, choose the **display** propert y. Make sure the **Bidirectional** box is not selected, and click **OK.** Don't worry if the label on the template disappears. It is simply displaying the value of the display custom property, which is currently blank.

On this page ...

- Custom Input Template Example



9. Now select the **Text Field** component, go to **Property Editor**, and click on the **Binding** icon of the **Text** property.
10. In the **Property Binding** window, click on the **Property** binding type, choose the **text** property. Make sure the **Bidirectional** box is selected, and click **OK**.



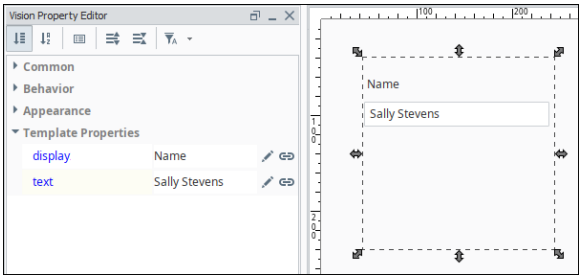
11. Next, make the background color of the Text Field change depending on whether the user entered a text value or not. While the **Text Field** component is still selected, go to **Property Editor**, and click on the binding icon of the **Background** property.
12. In the **Property Binding** window, click on the **Expression** binding type, and copy and paste in the following expression.

Expression - Input Validation

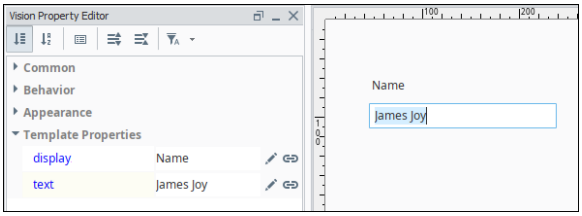
```
// Inside the if statement is the len and the trim functions that
are available when doing expression bindings.
// The trim function will trim the blank spaces from the text
therefore validating that there is actual text rather than spaces.
// The len function will count the length of the recently trimmed
text.
// The if statements asks the question: Is the length of the trimmed
text greater than zero?


if(len(trim({textBox.Text Field.text}))>0,color(255,255,255),color
(255,0,0))
```

13. Click **OK**.
14. Close the template, click **Save** on the top menubar, and click **Save** again.
15. To test it out, drag a few of the **Text Entry** templates from **Project Browser** to your window. At first, text field will be red as it is expecting data.

16. In the **Property Editor**, go to the **display** custom property and enter something like **Name**. Do something similar for the other templates to fill the label. Note that the background color changed to white.



17. Put the Designer into **Preview Mode**, and enter values into the text fields. Now, the information that was entered into the **text** property can also be bidirectionally bound so that it writes to a Tag or can be used in a script.



Client Tags for Indirection

Vision Client Tags can be used as a variable across all windows to indirectly point to a set of Tags, such as an area of the plant. Suppose that a window has a Dropdown List component that allows a user to select different areas of the plant. If the drop-down is bidirectionally bound to a Client Tag's selected string value property, the user can change the drop-down's value, therefore resulting in the indirect binding throughout the project reflecting the change in the Client Tag.

The Client Tag can be bound to a custom property on a window's Root Container. This Custom property can be bound to the Client Tag. The components inside the window can have indirect bindings on their properties that leverage the Custom property on the window. Therefore, the components on the window will be dependent on the Client Tag. Client Tags are managed inside the Client, therefore each Client will be independent of other Clients.

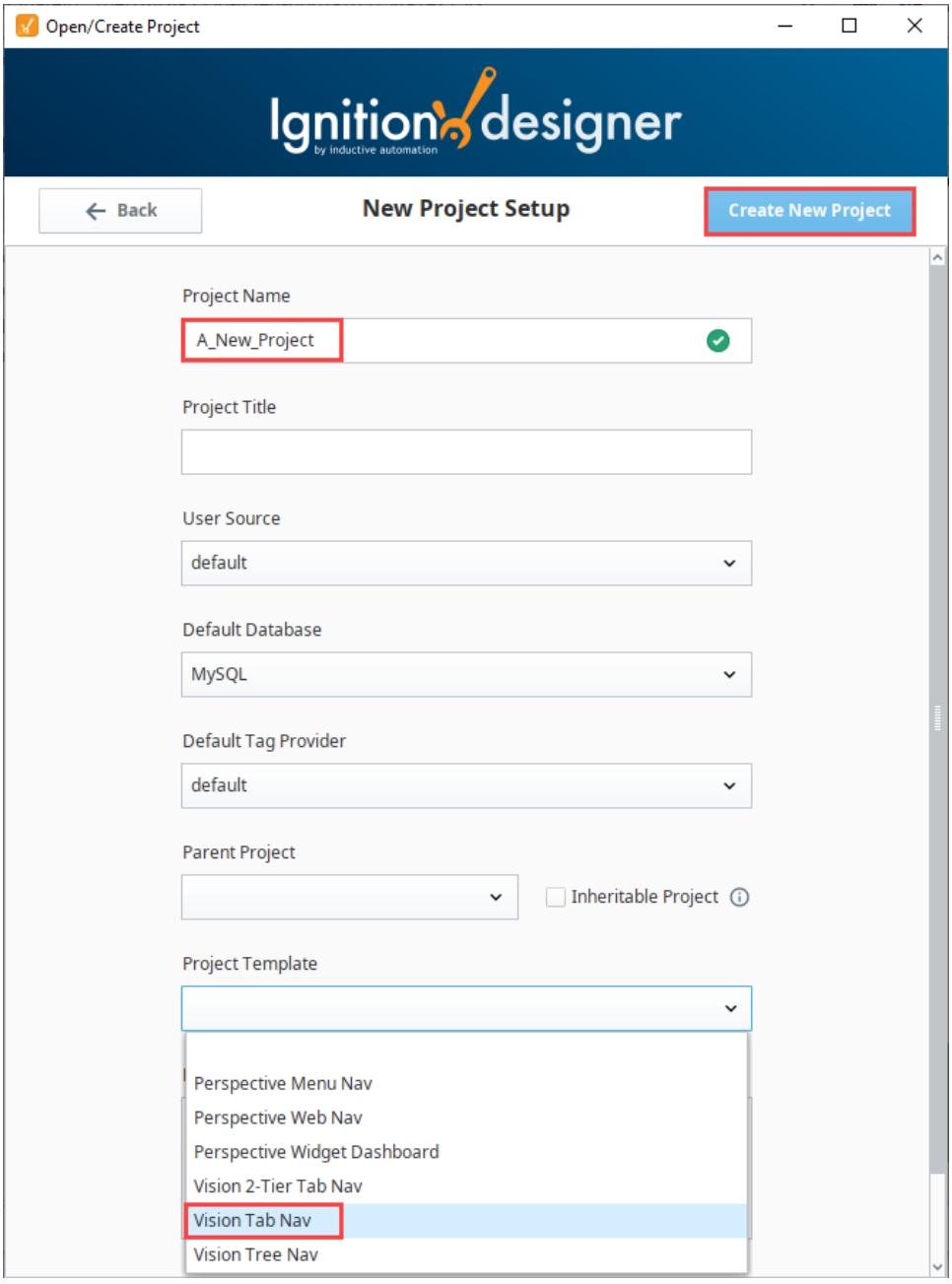
On this page ...

- Client Tag Indirection Example
  - Testing Your Work

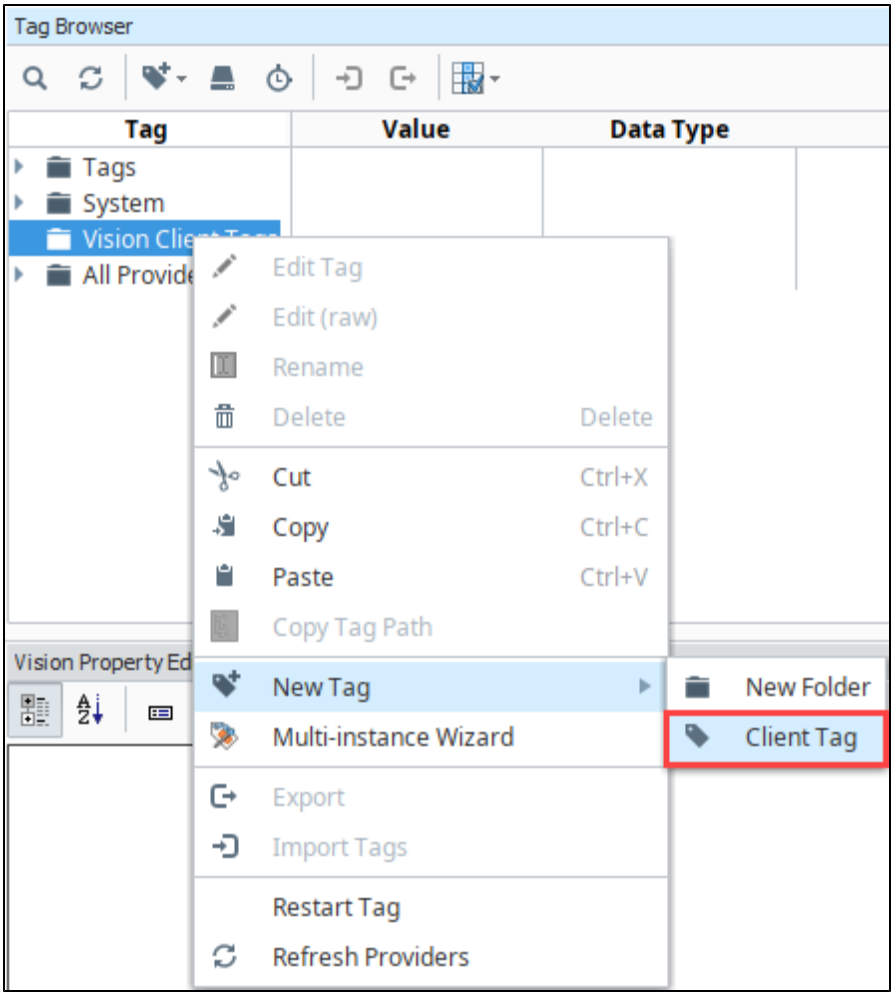
Client Tag Indirection Example

In this example, we will set up Client Tags for indirection which can be shared between different windows in our project.

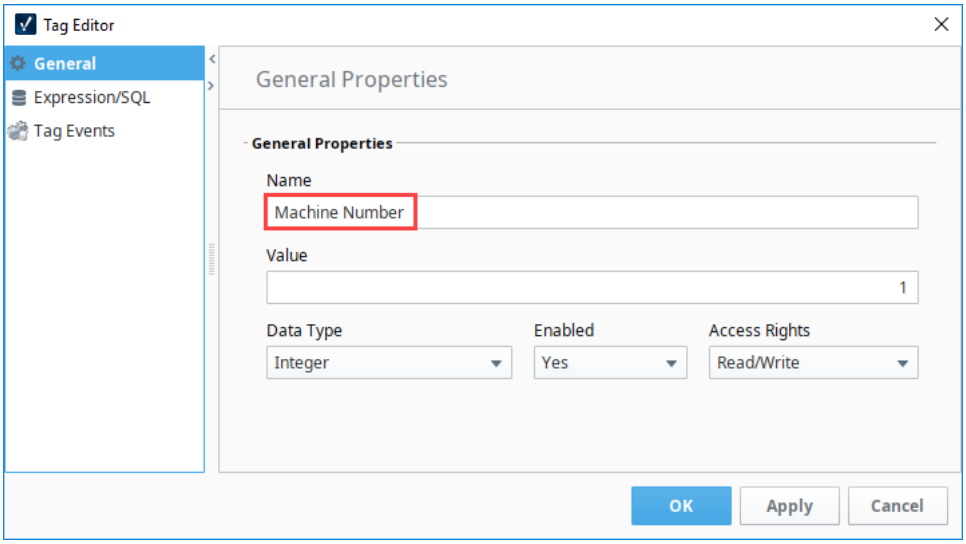
1. We need two windows and a way to navigate between them. If you don't already have a project that meets this criteria, create a new project using the **Vision Tab Nav** project template.



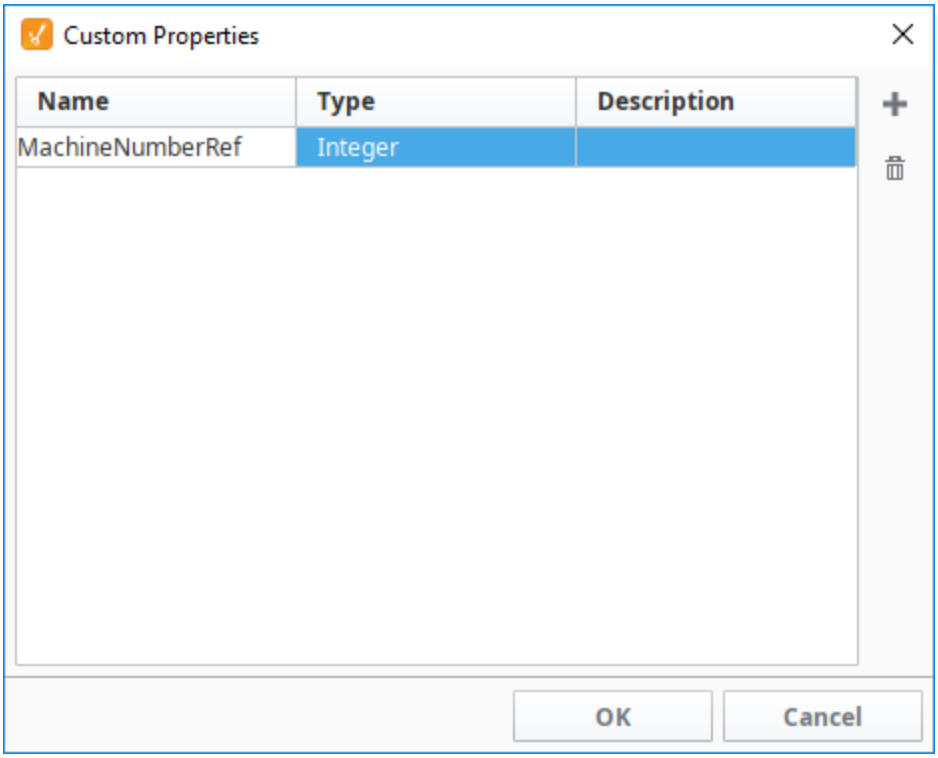
2. Add a Client Tag. In the Tag Browser, right click on **Vision Client Tags**, then choose **New Tag > Client Tag**.



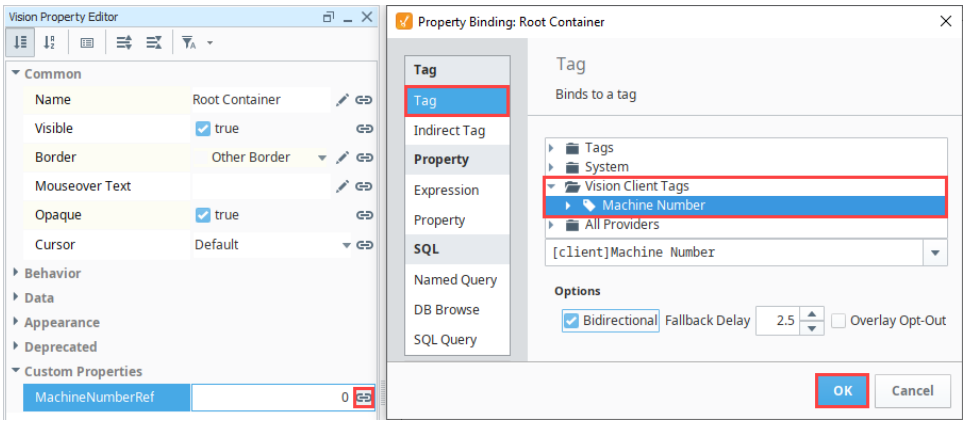
3. Name the Tag **Machine Number** and set the **Value** to **1**.



4. Add a window to show data based on the Client Tag. Create a new window named **Machine Overview**.
5. Next, we'll add a Custom Property to the Root Container of the window.
    a. In the Project Browser, select the **Root Container** for the Machine Overview window.
    b. Right click on the desktop and choose **Customizers > Custom Properties.**
    c. Click the **Add** ✚ icon.
    d. Enter **MachineNumberRef** as the name, and make sure the Type is **Integer**. Click **OK** to save the new custom property.

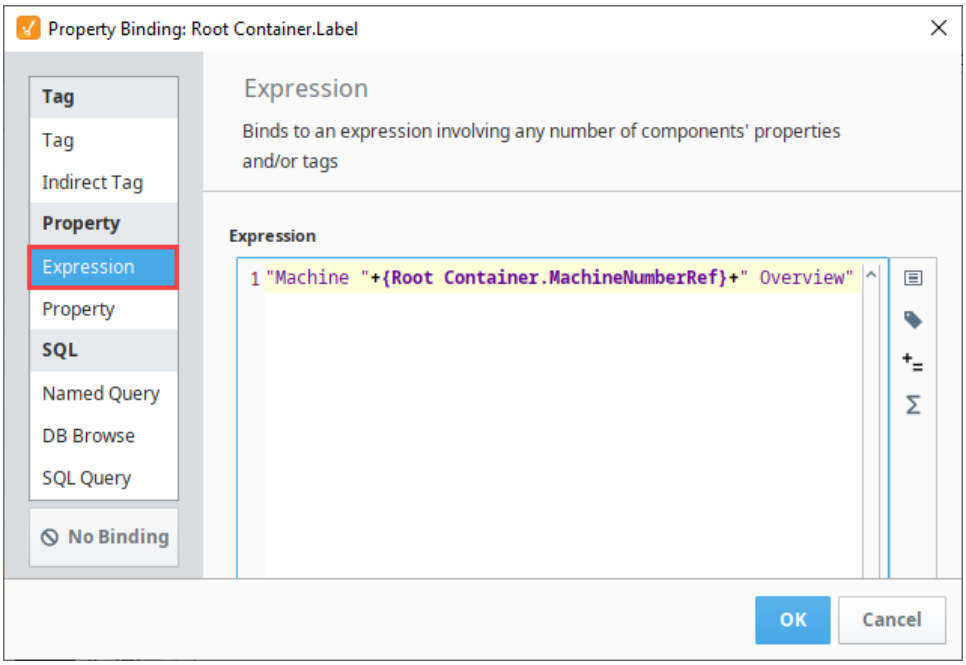


6. Next we'll bind the custom property to the Client Tag. In the Project Browser, select the Root Container for the Machine Overview window.
   a. In the Vision Property Editor, click on the **Binding** icon next to the **MachineNumberRef** custom property.
   b. Choose the **Tag** binding type.
   c. Select the **Machine Number** Vision Client Tag.
   d. Click **OK** to save the binding.

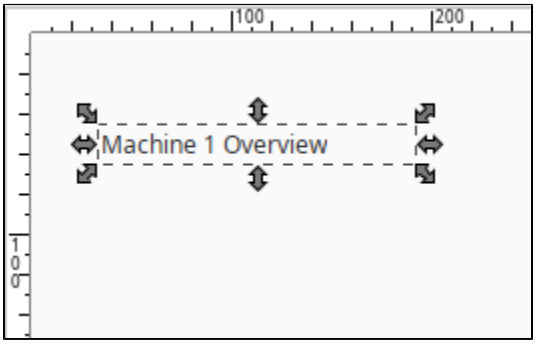


7. Drag a **Label** component onto the window. Next we'll add an Expression binding to show the page title.
   a. In the Vision Property Editor, click on the **Binding** icon next to the **Text** property.
   b. Choose the **Expression** binding type. Enter the following expression:

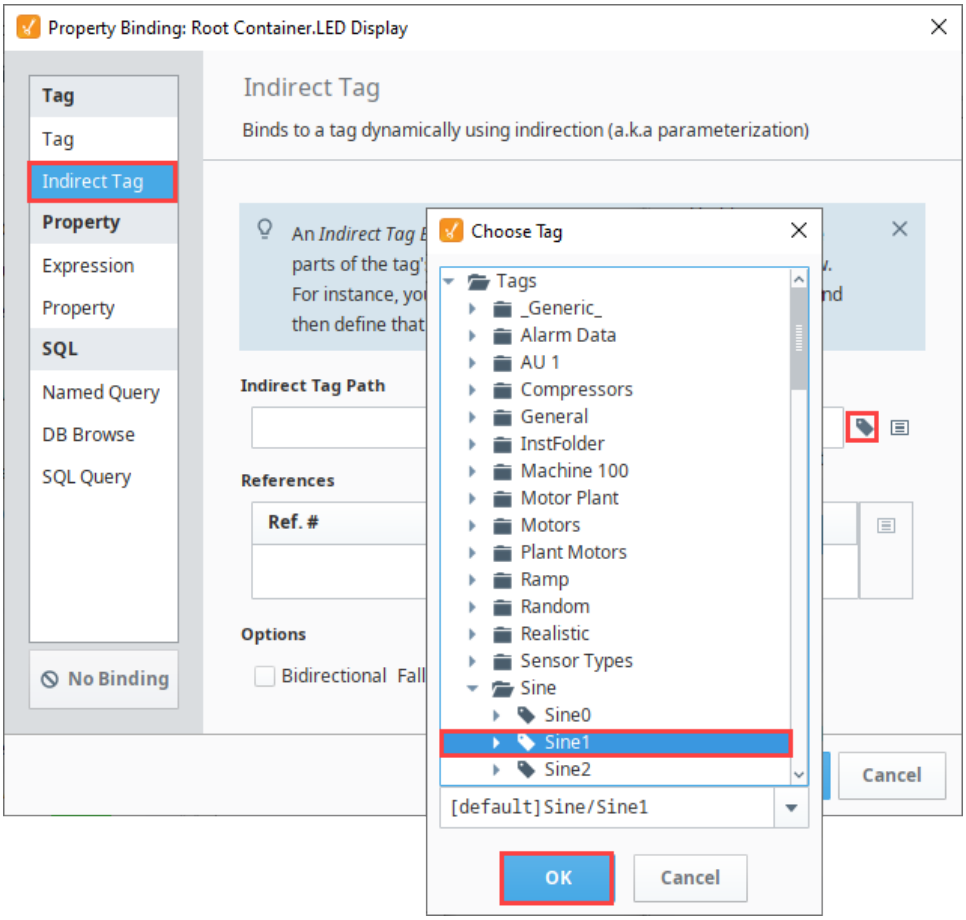
```
"Machine "+{Root Container.MachineNumberRef}+" Overview"
```



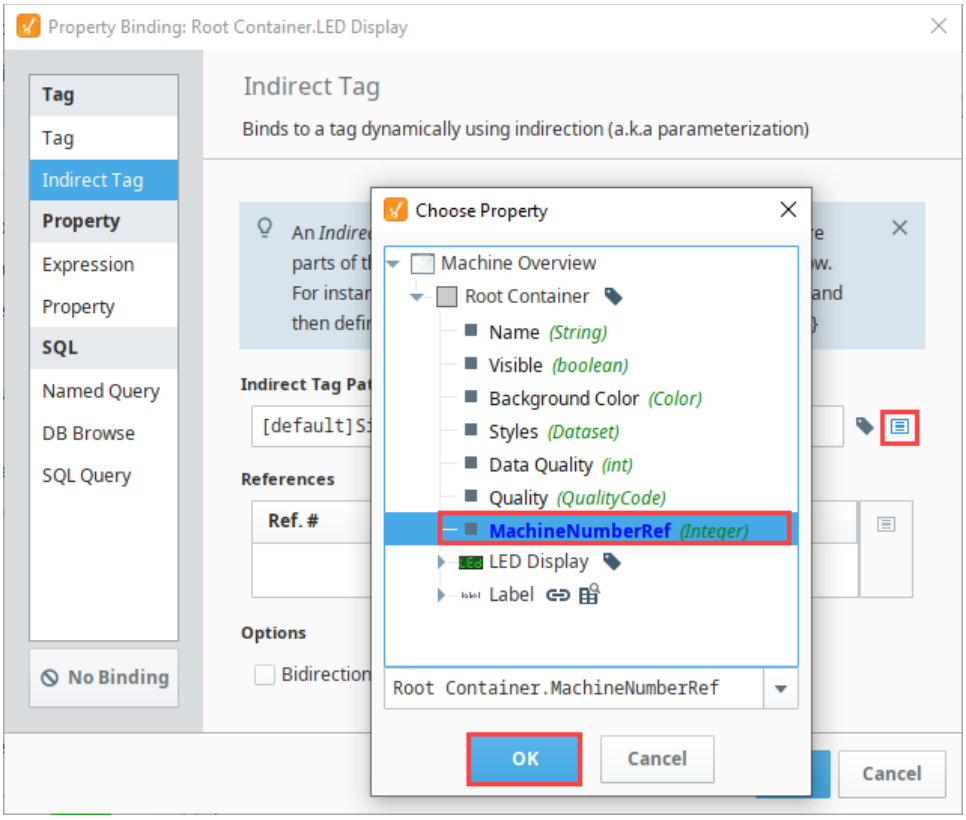
c. Click **OK** to save the binding.



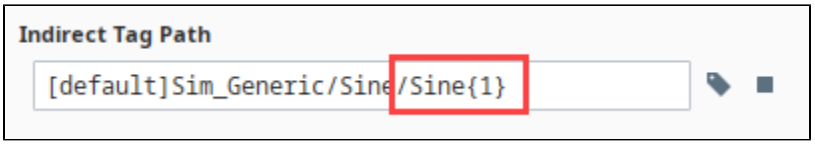
8. Next we'll set up an LED Display component for the value.
    a. Drag an **LED Display** component onto the window.
    b. Click on the **Binding** icon next to the **Value** property.
    c. Select the **Indirect Tag** binding type.
    d. Click on the **Tag** icon. Navigate to a Tag you want to use. We used the **Sine1** tag from the simulator.
    e. Click **OK** to save the Tag path.



f. Next, click the **Insert Property Value** icon.
g. Choose the **MachineNumberRef** property from the root container. Click **OK**.

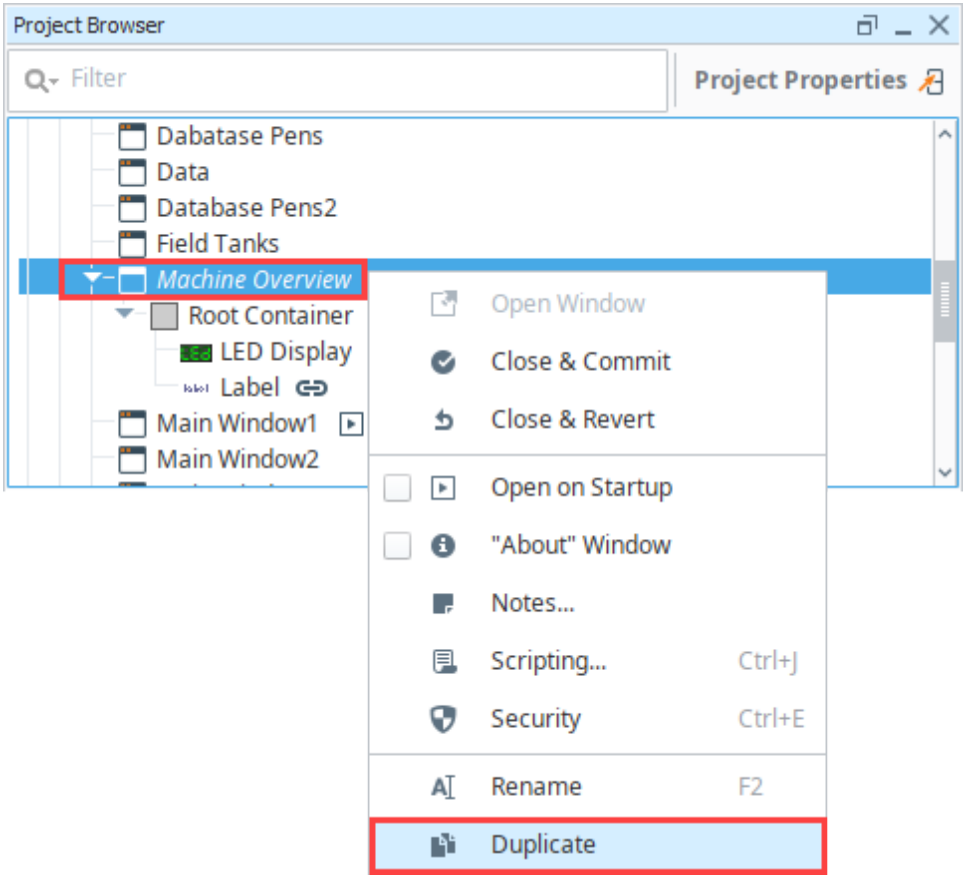


h. In the Indirect Tag Path field, delete the "1" before the {1}.



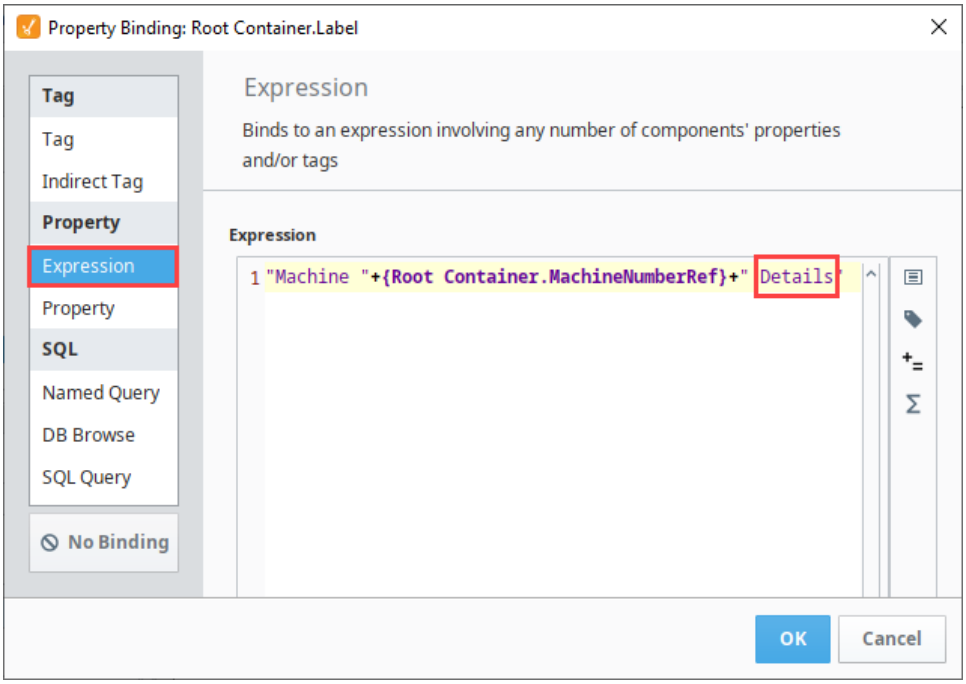
i. Click **OK** to save the binding.
j. **Save** your project.

9. Next we'll make a details screen so we can switch between the two. In the Project Browser, right click on the **Machine Overview** window and select Duplicate.

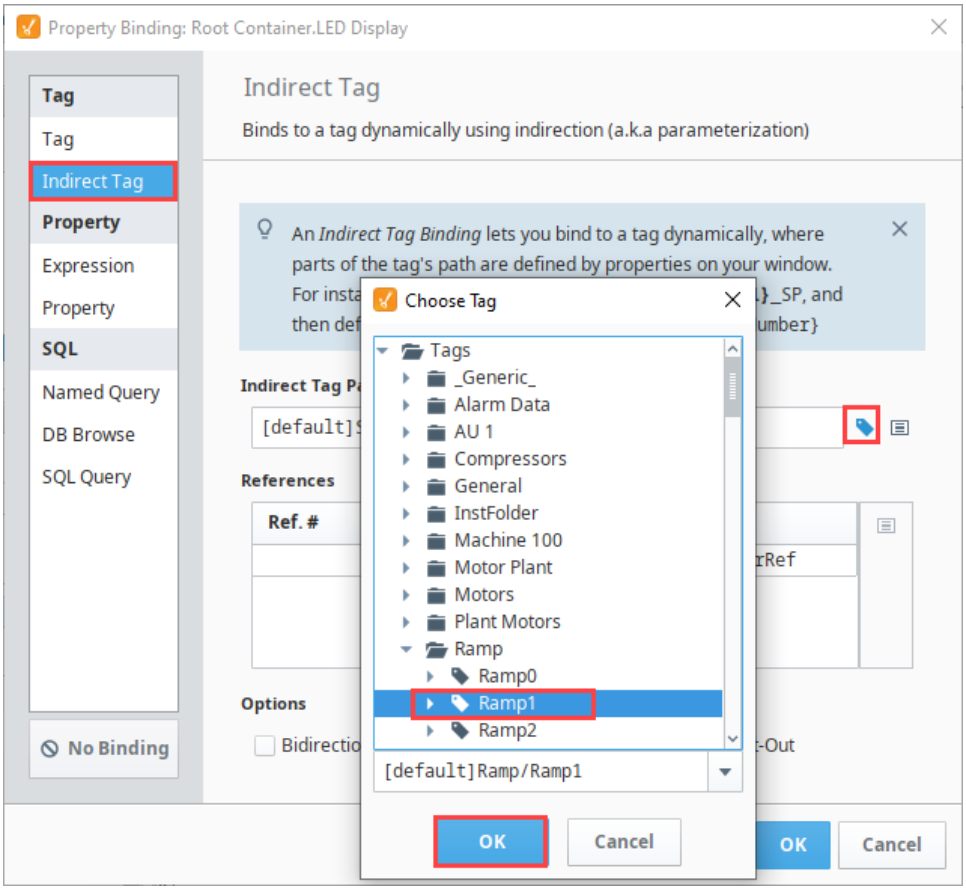


10. **Rename** the copy as "Machine Details."
11. On the Machine Details window, select the **Label** component.
12. Click on the **Binding** icon next to the **Text** property. Change the word **"Overview"** to **"Details**." Edit the Expression binding as follows:

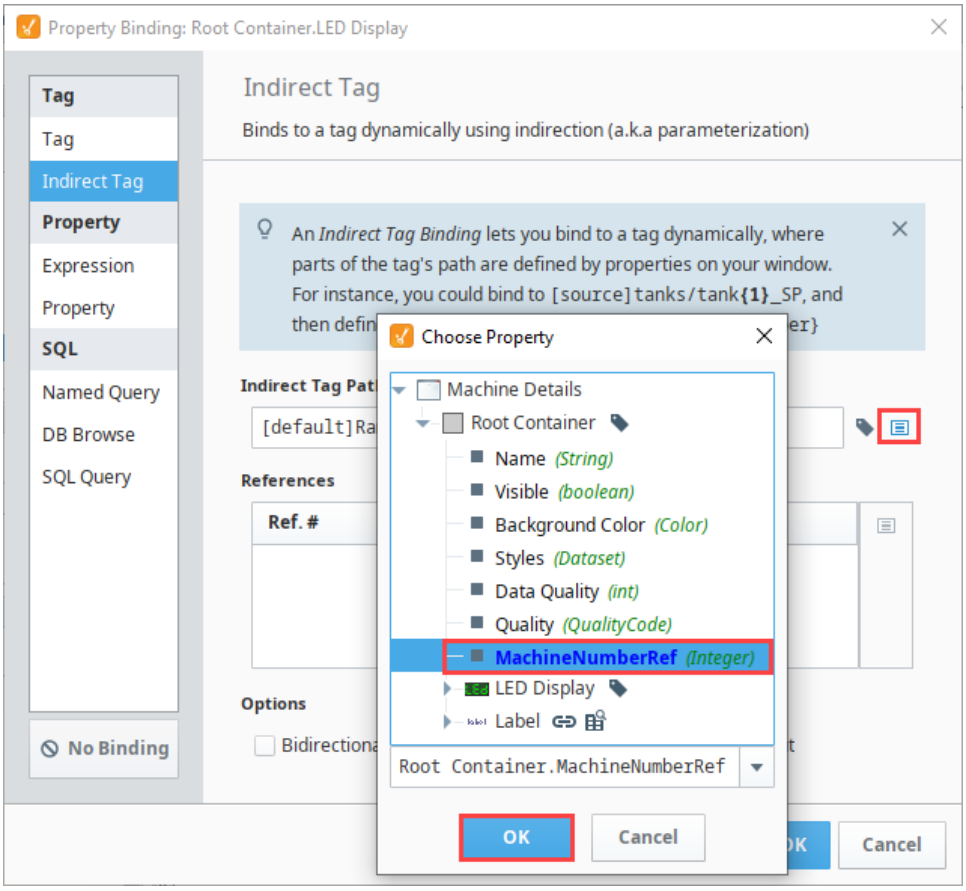
```
"Machine "+{Root Container.MachineNumberRef}+" Details"
```



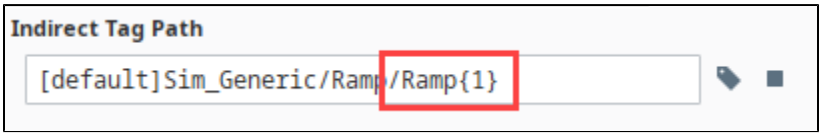
13. On the LED Display, update the Indirect Tag binding on the **Value** property to point to the **Ramp** Tag instead of the Sine Tags.
    a. Select a **Ramp** tag from the Tag selector and click **OK**.



    b. Click the Insert **Property Value** icon. Choose the **MachineNumberRef** property from the root container and click **OK**.

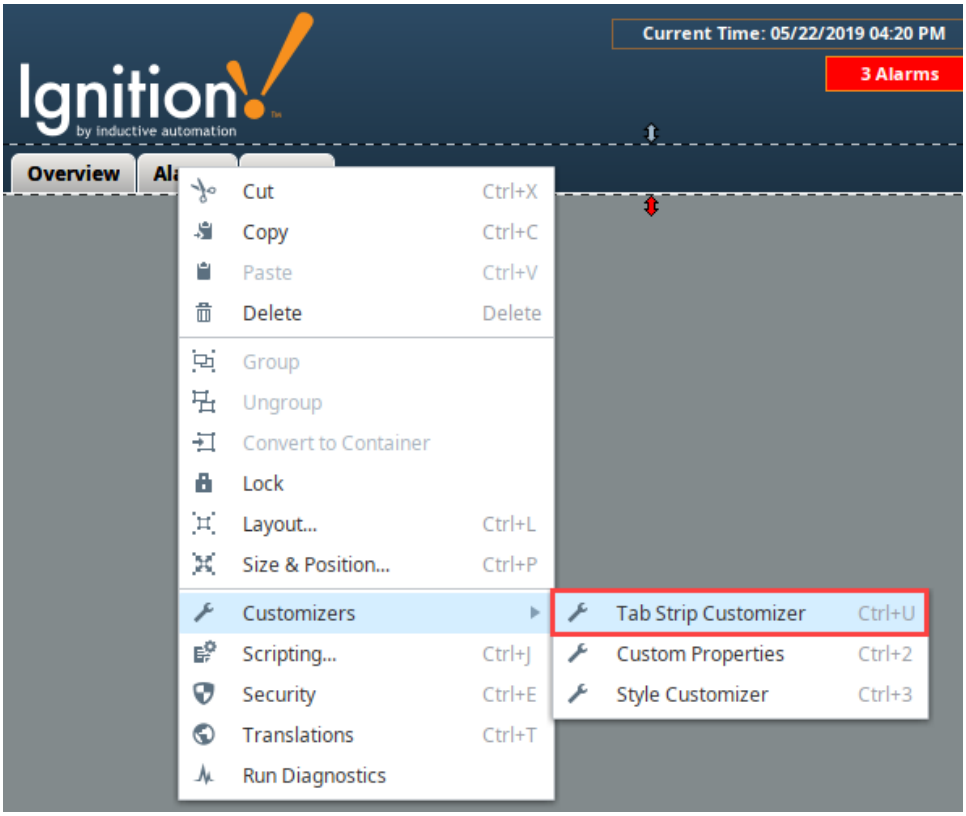


c. In the Indirect Tag Path field, delete the "1" before the {1}. Click **OK** to save the binding.

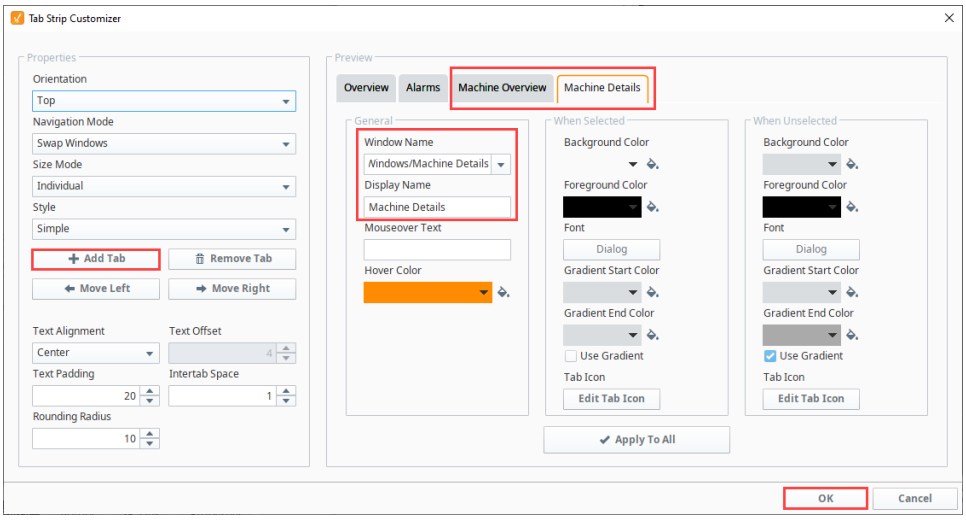


14. Next we'll add the two new windows to the Navigation. Open the Navigation window.

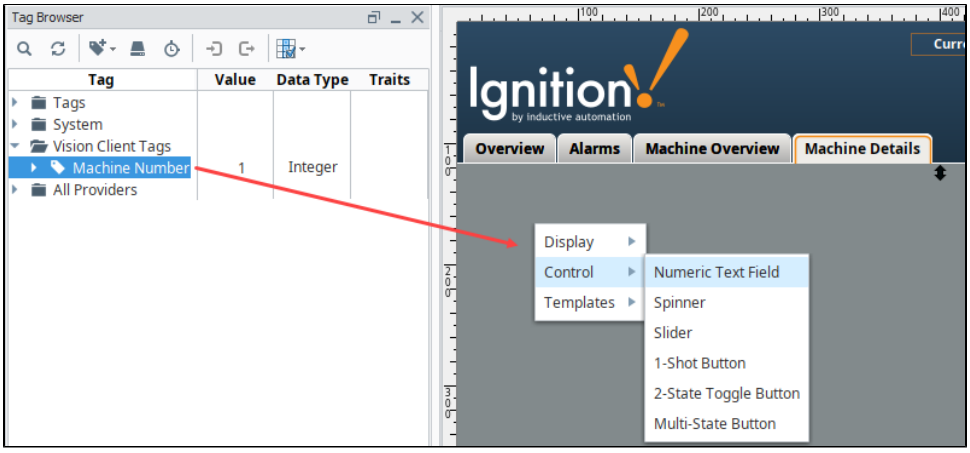
a. Right click on the **Tab Strip** component and select Tab Strip Customizer.



b. In the Tab Strip Customizer, click on the **Empty** tab. Change the Window name and the display name to **"Machine Overview**."
c. Click the **Add Tab** button. Change the Window name and the display name to "**Machine Details**."
d. Click **OK** to save the changes to the Tab Strip.



15. Add a way to change the Client Tag value in the Client. There are two ways to do this:
    a. Drag a **Numeric Text Field** component onto the Navigation window, and then drag the **Machine Number** Client Tag onto the **Numeric Text Field** component.
    b. If your Header doesn't have a background component, you can just simply drag the **Machine Number Tag** directly onto the window and select **Control > Numeric Text Field**.

c. 

16. **Save** the project.

Testing Your Work

Remember that the point of a Client Tag is to have different values across multiple Clients, so there is no way to test it in the Designer. To test the functionality of this example, do the following:

1. Open two Vision Clients.
2. In the first Client, change the Client Tag Value from "1" to "2" in the header.
3. Click between the Machine Overview and Machine Details screen. Notice that they are both showing the Tags for Machine 2.
4. Switch to the second Client. Notice that the Machine Overview and Machine Details screens still show the Tags for Machine 1.

High Performance HMI Techniques

About High Performance HMI Techniques

High performance HMI techniques and practices call for designs and displays which help the viewer make the best decision in the shortest amount of time after interacting with the HMI.

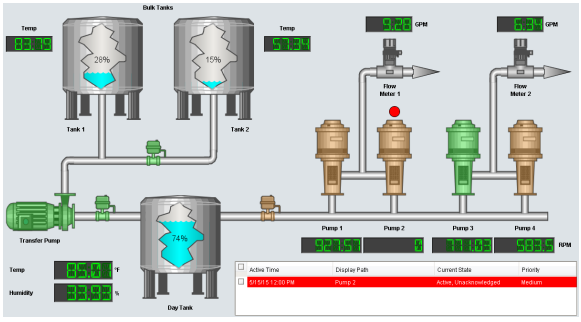
High performance HMIs often look basic and simplistic. They typically use gray-scale colors rather than the traditional graphics and bright colors for their displays. Conceptually, the High Performance HMI operates under the idea of visually contrasting critical and non-critical states. The power of this design philosophy is when something does go wrong, a high performance HMI will quickly guide the user to the source of the problem.

Here is a comparison of a traditional HMI next to a high performance HMI.

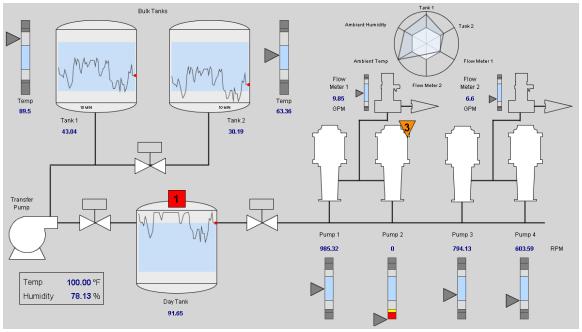
On this page ...

- About High Performance HMI Techniques
  - Traditional HMI
  - High Performance HMI
- Use of Color
  - Colors and Alarm Indicators
  - Technical Considerations with Colors
  - Accommodating Color Blind Viewers
- Alarm Indicator
  - Reducing Ambiguity

Traditional HMI

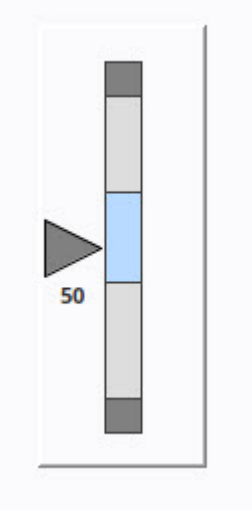


High Performance HMI

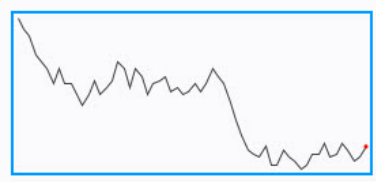


In Ignition, you create the high performance HMIs by using components such as moving analog indicators, sparkline charts, and radar charts.

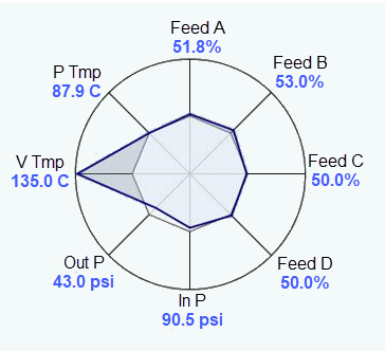
Moving Analog Indicator



Sparkline Chart



Radar Chart



Use of Color

Colors are an important consideration when designing a high performance HMI. Gray-scale colors are used instead of the traditional bright red, green, and blue colors.

The person that designs the HMI must understand the diverse audience that may view the HMI. For example, an operator may look at a motor on a traditional HMI. If it the HMI has colored the motor green, the operator may conclude that the motor is running. However, if a maintenance technician looks at the same motor, he may conclude that the motor is not faulted. These are logical conclusions. They reflect the respective interest of the person viewing the HMI where the operator wants the motor to run and the maintenance technician wants it to be working.

In reality, the motor is simply "scheduled" to run, meaning when it runs depends on its control mechanism. For example, the motor may run when there is product on the line or boxes on the conveyor. In other words, the motor may be periodically starting and stopping automatically.

A high performance HMI can eliminate this confusion by introducing a color that signifies a state of "scheduled." A common high performance HMI practice is to use a dark gray to signify a "scheduled" state for equipment. This color should never compete with more alerting colors that "pop" from the HMI resulting in the viewer's eye's being drawn to the portion of the HMI where the problem may be occurring.

INDUCTIVE UNIVERSITY

Use of Color

Watch the Video

Colors and Alarm Indicators

Color can play an important role in how an operator responds when problems do occur. High performance HMI design refrains from coloring equipment when the equipment is in a state of fault. For example, some equipment may still run when faulted. Instead, an optimized solution is to place an alarm indicator near the equipment in such a way that when the equipment is undergoing some fault, the alarm indicator renders with the appropriate color and shape. The object and the consequential color should signify the most important alarm state occurring for the equipment at that current time.

Technical Considerations with Colors

Some HMIs in industrial settings may temporarily lose their ability to render color because of various environmental factors. High performance HMI design incorporates this possibility by encouraging the use of descriptive text with color. For example, motors of two different colors may look the same on a color deficient HMI resulting in confusion for the viewer. Even worse, the viewer may misinterpret the motor state and assume everything is fine. However, if each motor has a descriptive text such as "Motor 1 is Faulted" and "Motor 2 is Running", the problem associated with a faulty HMI failing to display color is largely reduced by the HMI's high performance design.

Accommodating Color Blind Viewers

Common color combinations such as red and green and blue and purple cannot adequately be distinguished by those with color blindness. High performance HMI design accommodates color blind uses by combining colors with descriptive text as well as incorporating alarm indicators in unique shapes.

Alarm Indicator

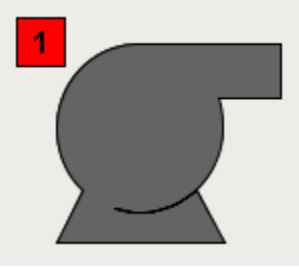
The high performance HMI design techniques make use of an object called the **Alarm Indicator** which displays a colored shape when there is a problem. This works well with the high performance HMIs as color is only used when there is a problem. The Alarm Indicator can contain descriptive text in addition to the shape and color, and is usually placed near the component that is causing the problem. You can import the Alarm Indicator, shown in the example below, from Ignition Exchange.

The Alarm Indicator represents different levels of alarm with different shapes, color, and descriptive text. For example, a motor that exists in an industrial setting is monitored by a high performance HMI. There are two alarms on the motor. The first is a critical alarm associated with the motor becoming seized, will display as a red rectangle with the number 1. The second is a high priority alarm associated with the motor when overheating, will display as a yellow triangle with the number 2. An Alarm Indicator is placed near the motor, positioned in such a way to clearly show motor that the indicator is referring to.

Given this scenario, a high performance HMI will show the Alarm Indicator as in the following examples:

Example 1

The motor is critically faulted. This is the highest priority.

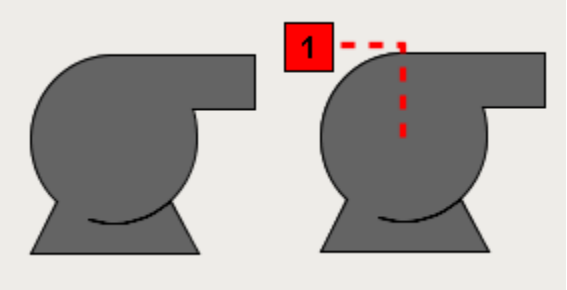


Example 2

Motor is overheating resulting in a high alarm.

Reducing Ambiguity

A high performance HMI design technique to reduce ambiguity incorporates a line from the Alarm Indicator to the object experiencing the alarm. A simple dotted line can ensure that the viewer associates the correct Alarm Indicator with the correct motor.



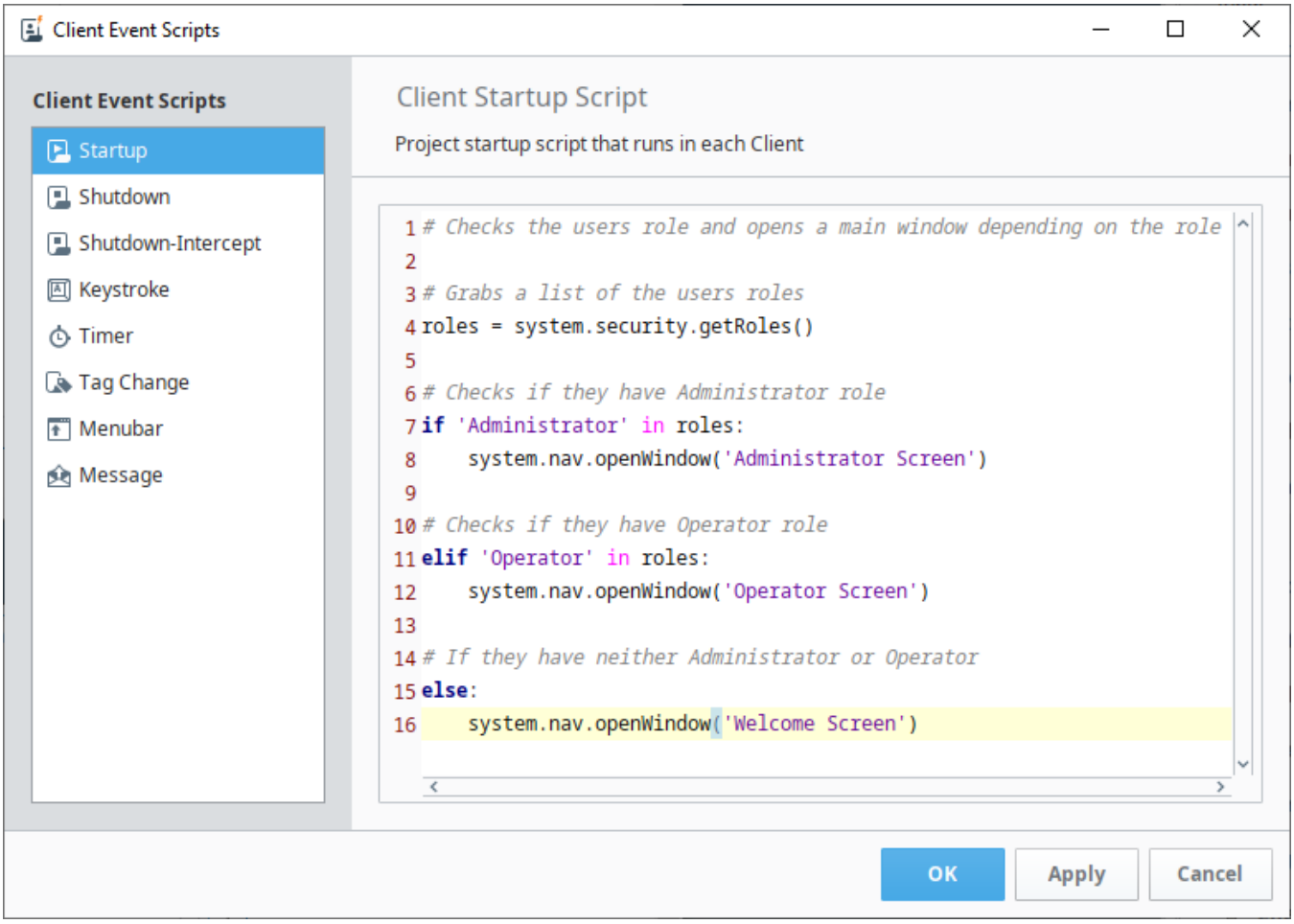
Open Dynamic Windows on Startup

Sometimes a project needs to change its startup windows depending on who logged in, what security roles they have, or what computer the Client is launched on. In these cases, rather than setting a static startup window, you can write a Client Startup Script that uses the `system.nav` library to open a dynamic set of windows based on hostname, IP address, and user who logged in.

This means you will remove the Open on Startup option from some or all of your windows and use a Client Startup Script to determine which windows will be opened. Typically, you will set your navigation window to Open on Startup, but decide on a main window in the startup script. The example below checks the users role before opening a window.

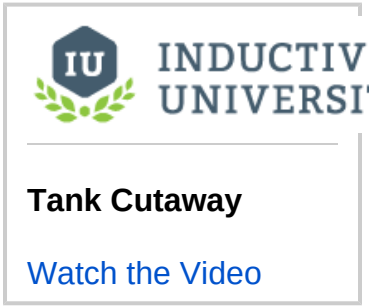
Code Snippet - Client Startup Role Check

```
# Checks the users role and opens a main window depending on the role

# Grabs a list of the users roles
roles = system.security.getRoles()

# Checks if they have Administrator role
if 'Administrator' in roles:
        system.nav.openWindow('Administrator Screen')

# Checks if they have Operator role
elif 'Operator' in roles:
        system.nav.openWindow('Operator Screen')

# If they have neither Administrator or Operator
else:
        system.nav.openWindow('Welcome Screen')
```

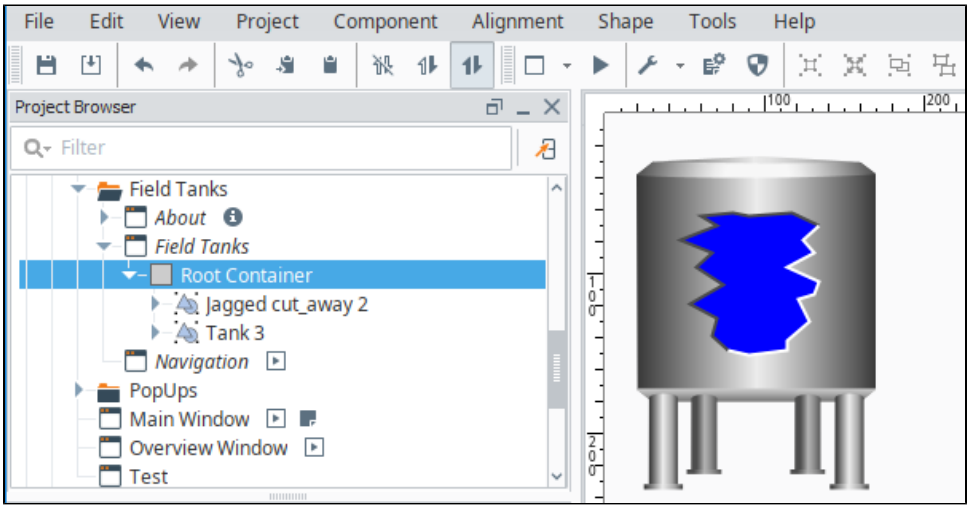

Client Event Scripts
Client Event Scripts
Startup
Shutdown
Shutdown-Intercept
Keystroke
Timer
Tag Change
Menubar
Message
Client Startup Script
Project startup script that runs in each Client
1 # Checks the users role and opens a main window depending on the role
2
3 # Grabs a list of the users roles
4 roles = system.security.getRoles()
5
6 # Checks if they have Administrator role
7 if 'Administrator' in roles:
8 system.nav.openWindow('Administrator Screen')
9
10 # Checks if they have Operator role
11 elif 'Operator' in roles:
12 system.nav.openWindow('Operator Screen')
13
14 # If they have neither Administrator or Operator
15 else:
16 system.nav.openWindow('Welcome Screen')
OK
Apply
Cancel

Tank Cutaway

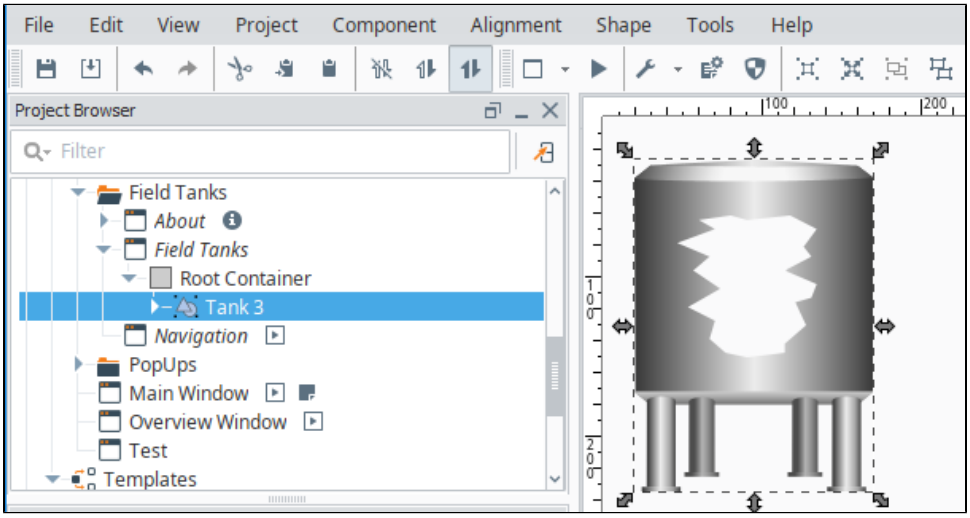
The Symbol Factory images in the Basic > Tank Cutaways category work well when combined with the other symbols, especially tanks from the Tank Category. Use the following technique to make a dynamic cutaway tank display:



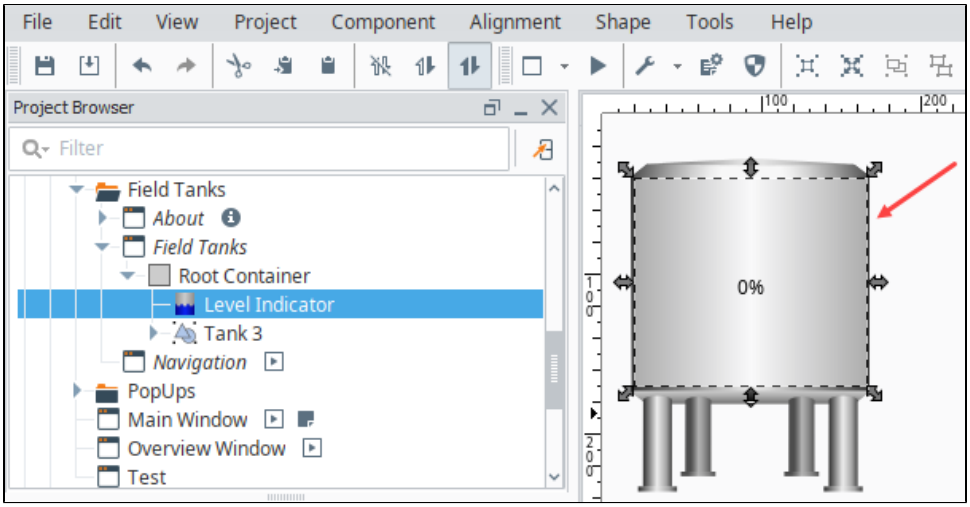
1. From your Symbol Factory SVG symbols, drag a **Tank** and a **Cutaway** symbol onto the window. (We used *tank 3* and *jagged cut-away 2*.)
2. Align the cutaway symbol on the tank where you'd like the cutaway to be placed.



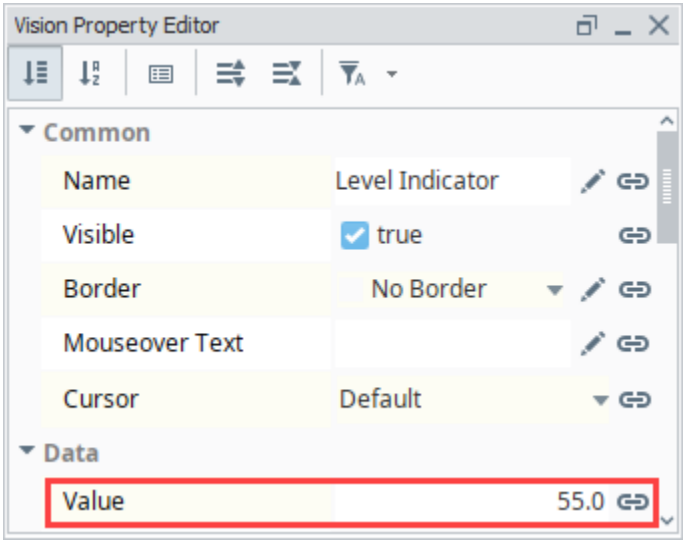
3. Select the tank symbol **first**, and then select the cutaway while holding **CTRL** to select both symbols.
4. Click the **Difference** icon to use the cutaway symbol to make it appear that the area of the tank is cut away.



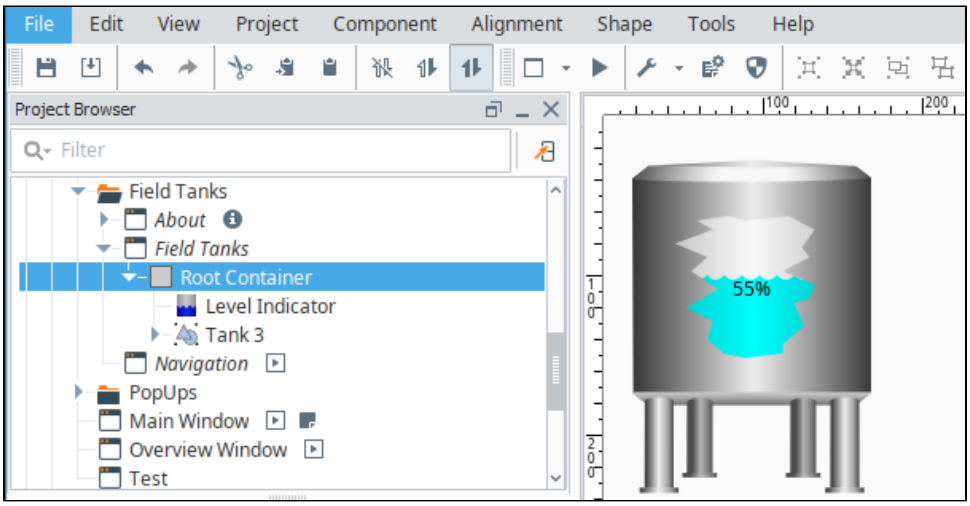
5. Place a **Level Indicator** component (drag from the Component Palette) on the area removed by the cutaway.



6. With the Level Indicator selected, in the Property Editor, enter a value for the **Value** property, or use a binding to put a value on the **Level Indicator**.



7. Choose **Alignment > Move Back** to put the Level Indicator behind the tank.



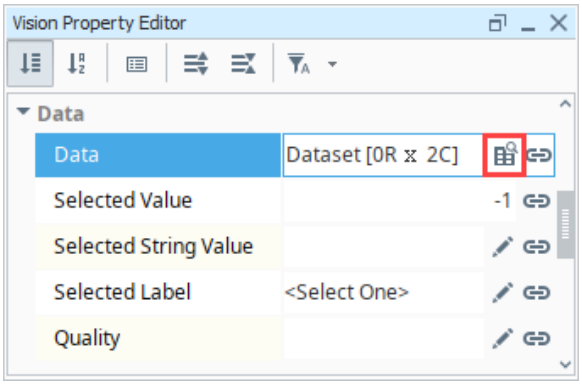
8. This is an optional step, but you can select the tank, including all the graphics, right click and Group them so now they can move around as one unit. You can even make a template out of it, so you can use it multiple times.

Related Topics ...

- Symbol Factory

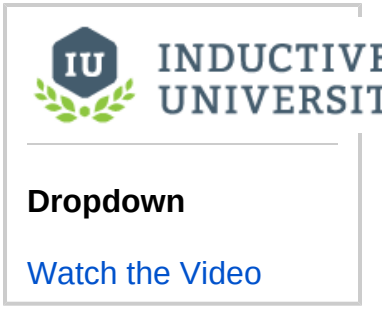
Dropdown List Example

Dropdown lists are used when you want to select a single item from a list of options. The Dropdown component is under the Input section of the component palette. Simply drag it on to your window. The most important property of a Dropdown component is the Data property. It is a Dataset that contains one or more rows of data. Each of the rows are different options that you see on the component. Select the Dropdown component and click on the **Dataset Viewer** icon for the Data property to manually add some options.



On this page ...

- Data Property's Dataset Modes
  - Number/Label Pair
  - Single Label Column
  - Code/Label Pair
- Setting Dropdown Options
- Displaying Multiple Columns in a Downdown List



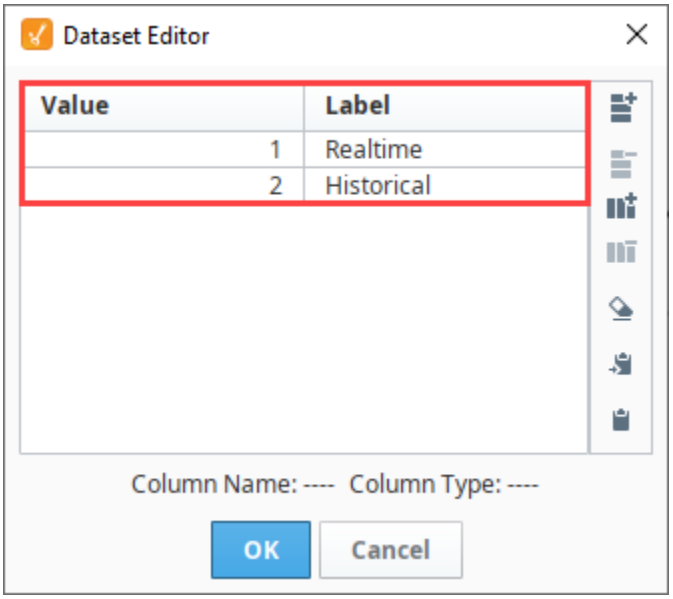
Data Property's Dataset Modes

There are three modes you can use the **Data** property's dataset: a number/label pair, a single label column, and a code/label pair. Which mode is used depends on what columns are in the **Data** property's dataset, and will determine the values of the **Selected Value**, **Selected String Value** and **Selected Label** properties. Any additional columns that are added to the dataset will not affect these properties.

Number/Label Pair

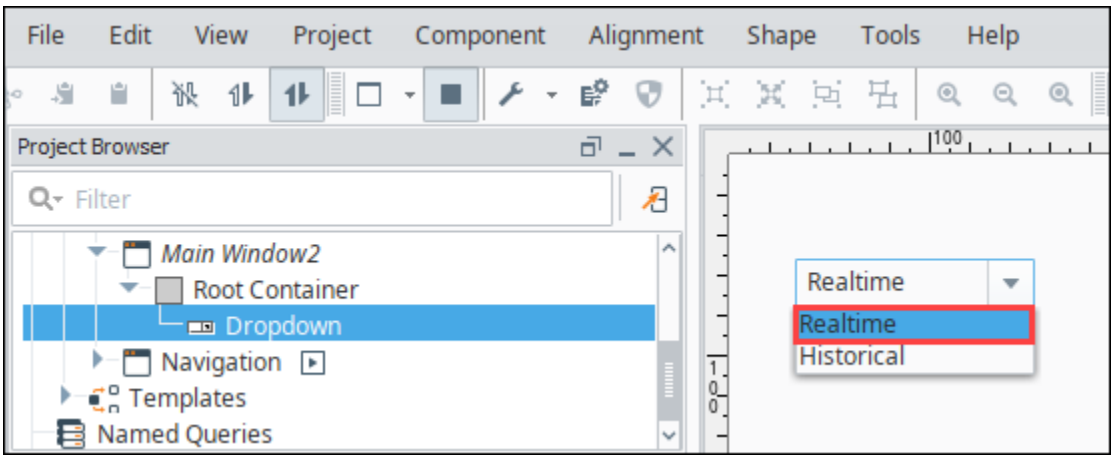
In the Number/Label Pair mode, the first column of the Dataset is an integer (often an id) and the second column is a string (often a label). The first column is the **Value** column which is invisible to the user and is usually used in binding. The second column is the **Label** column and is visible to the user in the dropdown list.

In this example, the dataset has two rows. The **Value** column includes the integers **1** and **2.** Under the **Label** column, **Realtime** and **Historical** are displayed, respectively.



In **Preview Mode**, select from the list of dropdown options. Notice, you can only see the options in the **Label** column. Select the **Realtime** option. Now, under the Data property section, the **Selected Value** is **1**, which is the corresponding integer in the Value column of the dataset. The **Sel**

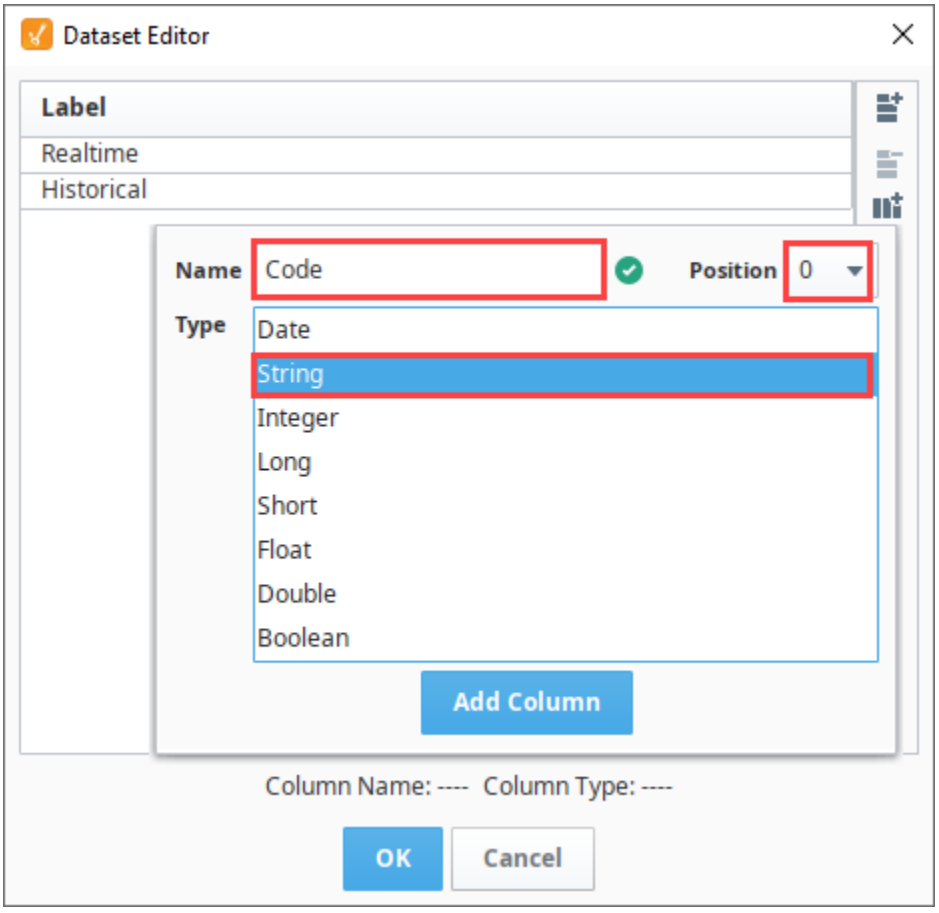
ected String Value and the **Selected Label** both display **Realtime**, which is the corresponding string in the Label column of the dataset.



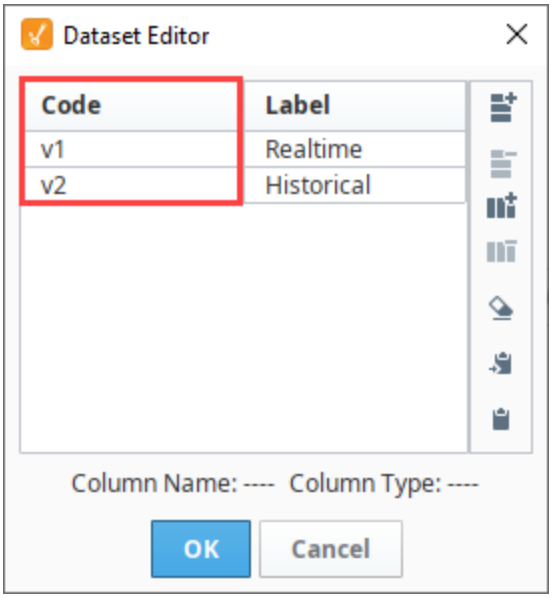
Single Label Column

The Single Label Column mode you can use is a string/string combination which are two columns that are both strings.

1. In the Dataset Editor, remove the **Value** column from our example by selecting a cell in that column and clicking on the Vertical Delete ic on.
2. Add another column by clicking on the Vertical Add icon.
3. Call the new column **'Code,'** set the column position to **'0'** and make it a **'String.'** Click **OK**. Both columns are now strings.

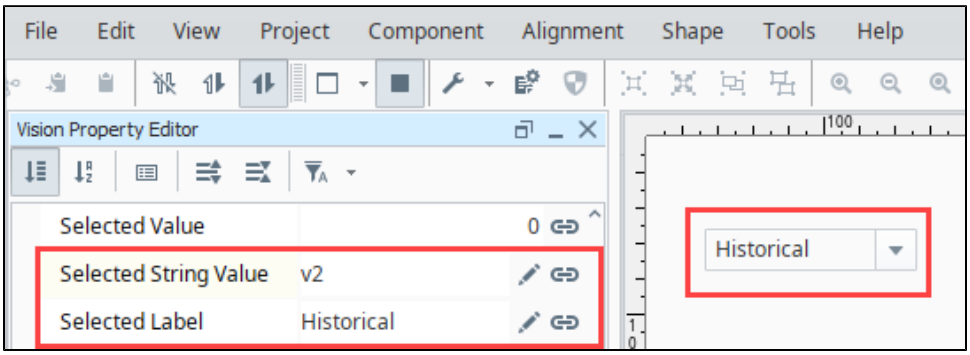


4. Under the **'Code'** column enter **'v1'** and **'v2.'** Click **OK.**



Note: Because we added the 'Code' column in position '0,' the users will see the same two options in the dropdown list: 'Realtime' and 'Historical.' If you see 'v1' and 'v2' instead, that means your 'Code' column is not the first column.

5. In Preview Mode, click on **'Historical'** and you can see the Selected String Value is **'v2'** because the value is in the first column and the Selected Label is**'Historical.'**

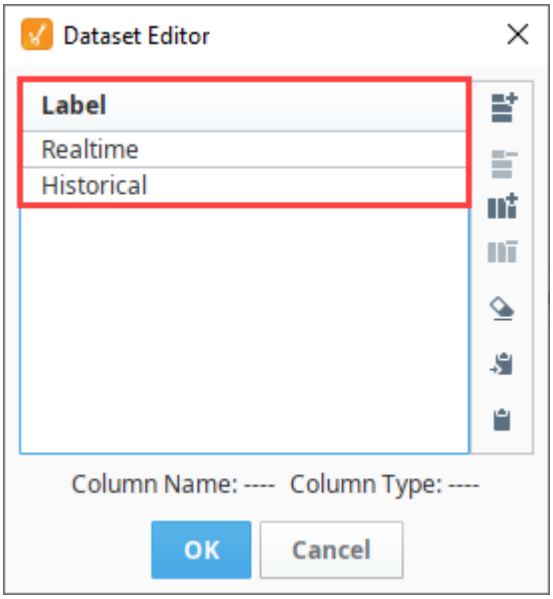


Code/Label Pair

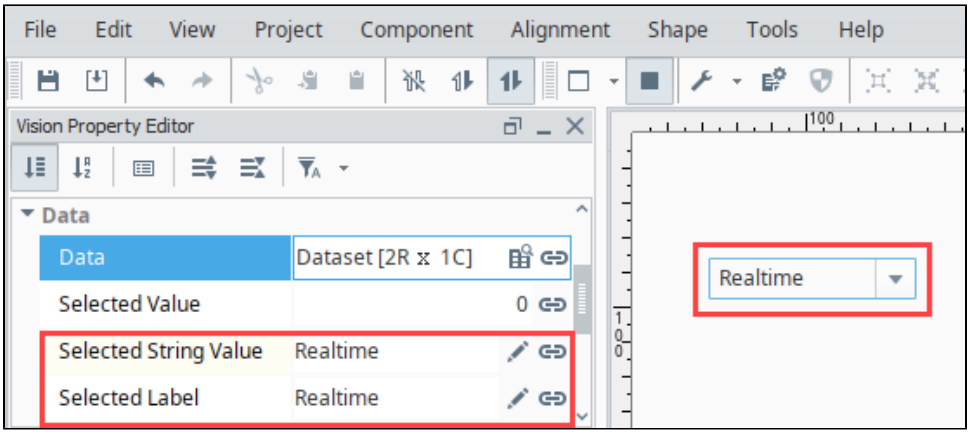
The Code/Label Pair mode simply uses a one string column.

1. In the Dataset Editor, remove the **Code** column from our example by selecting a cell in that column and clicking on the **Vertical Delete** ic on.
2. You are going to see the same two options in the Dataset Viewer: **'Realtime' and 'Historical.'**

Note: This only applies if there is exactly one column in this dataset.



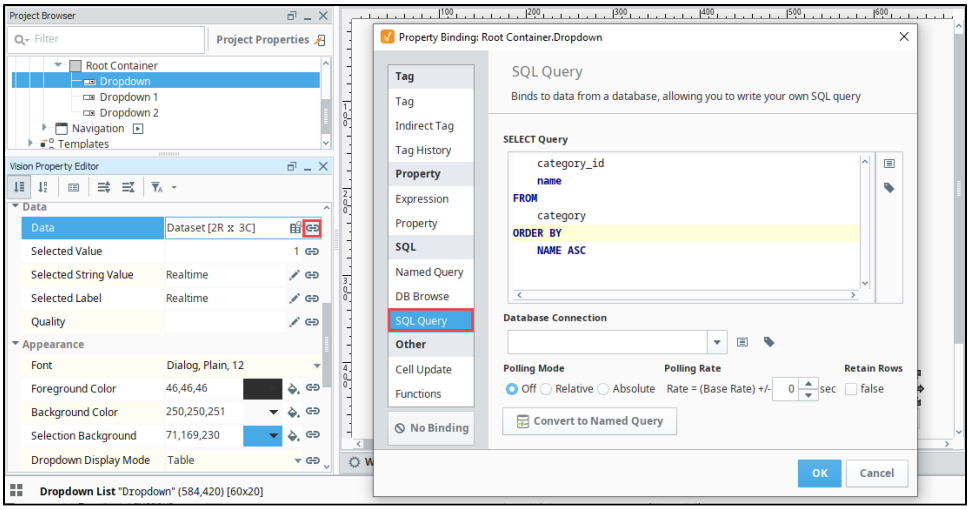
3. In **Preview Mode**, select the **'Realtime'** option. You will see the same value in the **Selected Label** and **Selected String Value** properties since there is only the one column.



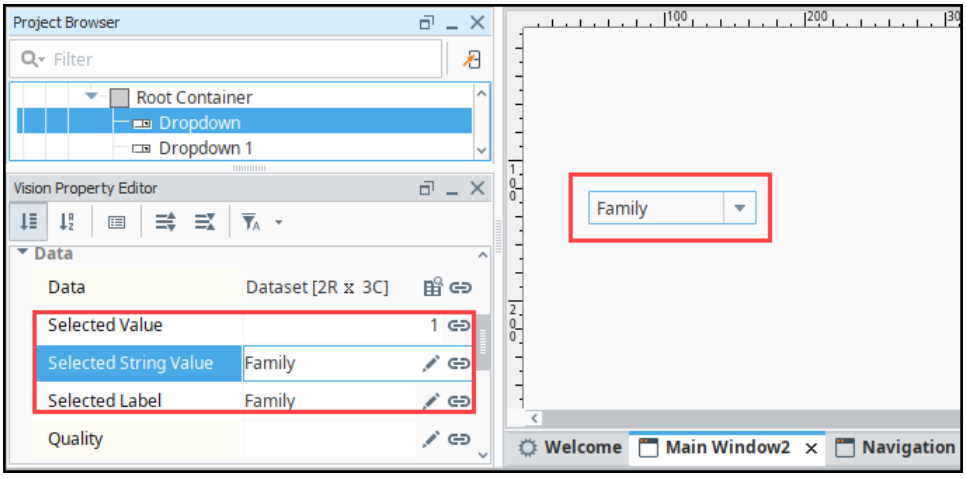
Setting Dropdown Options

Now, you can set these dropdown options manually or bind the Data property. In this example, you can take a Dropdown List on the window and bind the Data property using a SQL query.

1. Select the Dropdown component, and click on the **binding** icon for the Data property.
2. Select the **SQL Query Binding Type** and enter a query that brings back an ID and Name from one of your tables in the database.



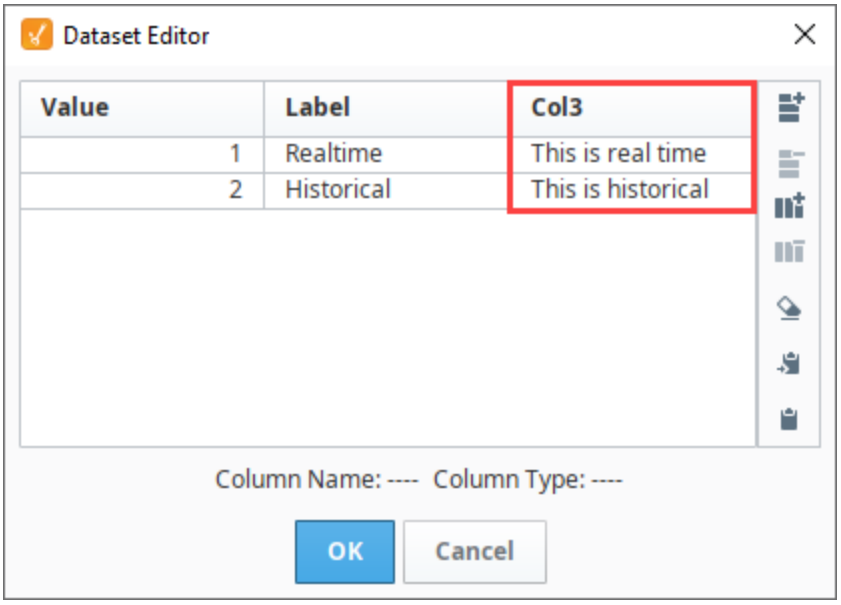
3. Click **OK** to see all the options that came back from the database.
4. In Preview Mode, select any one of these options and you can see the **Selected Value**, **Selected String Value**, and **Selected Label** as shown in the following example.



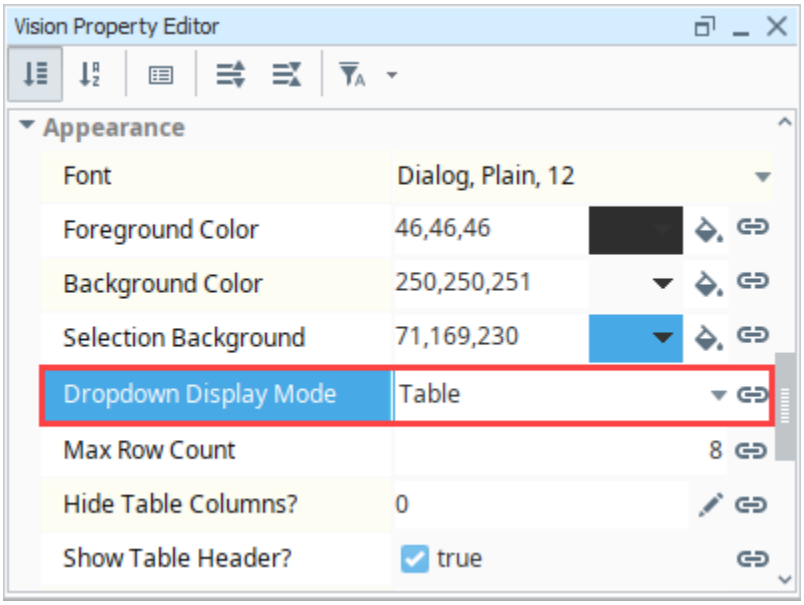
Displaying Multiple Columns in a Downdown List

Another feature of the Dropdown List component is you can show more than one column to a user.

1. Drag another Dropdown component to your window.
2. Go to the **Dataset Viewer** and add some options manually. Instead of having only one or two columns you can add as many as you want. The first column needs to be either an **integer** or a **string**. Any additional columns will show up in the dropdown.
    a. Open the Dataset View, under **Value**, enter **'1'** and **'2.'**
    b. Under **Label**, enter '**Realtime**' and '**Historical**.'
    c. Add another column and call it '**Col3**' in '**Position 2'** , and make it a **string** and click **Add Column.**
    d. Under Col3, enter **'This is real'** and **'This is historical'**.
    e. Click **OK**.

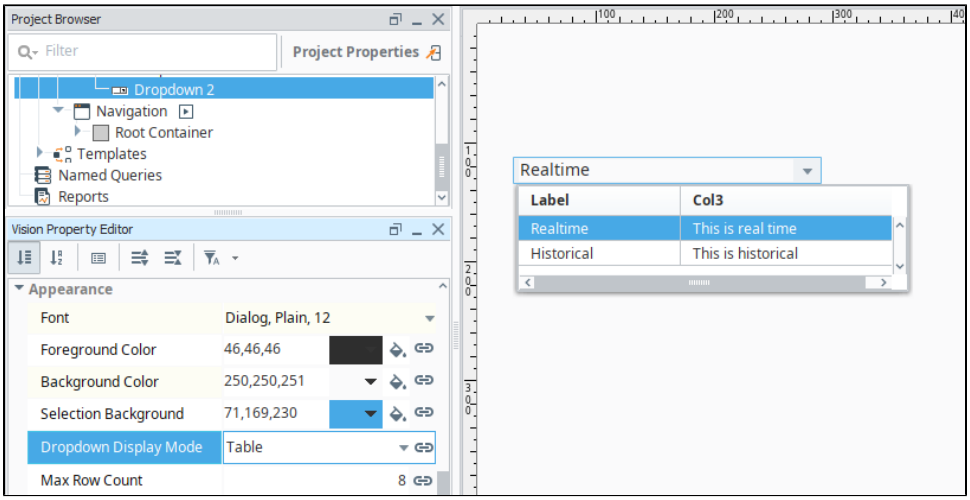


3. In the Property Editor, set the **Dropdown Display Mode** to **'Table'** to see both columns.



4. Now you can see **Label** and **Col3** and select between those different rows. It's a nice way to show more information in the Dropdown list. The **Selected Value**, **Selected String Value**, and **Selected Label** properties will behave the same as previous examples ignoring any

columns beyond the second.



Multi-Monitor Clients

Multiple Desktops

In some situations, such as control rooms, or workstations with multiple monitors, it may be preferable to have clients open on several monitors so that different windows are simultaneously in view. Instead of opening several different clients, it is possible to open a single client, and spawn multiple desktops through scripting. Desktops are additional workspaces where windows may be opened. They are similar in functionality to a standalone Client in that they may be positioned and resized independently of other desktops and Clients, but share a session ID with the initial Client that launched the desktop.

Client Tags and Property Values

The value of Client Tags are shared between each desktop. This provides an easy method to change values on one desktop from another without interacting with other Clients: simply write to a Client Tag.

Desktops act as separate clients in regard to property values. For example, if a Text Field is placed on a window, and multiple Desktops open that same window, values entered into one Desktop will not overwrite the other Text Fields. If synchronization on these components is preferred, then simply bind the property to a Tag.

Handles

When creating a new desktop, an optional handle may be assigned to the desktop. This acts as a name, or reference to the desktop. If a handle is not provided, then the desktop may be referenced by the screen index. Handles and indices are useful when trying to interact with specific desktops from a Python script.

On this page ...

- Multiple Desktops
  - Client Tags and Property Values
  - Handles
  - Spotting the Primary Desktop
  - Project Updates
- Opening Another Desktop
  - Navigating Windows in Desktops
- Opening a Desktop on Each Monitor

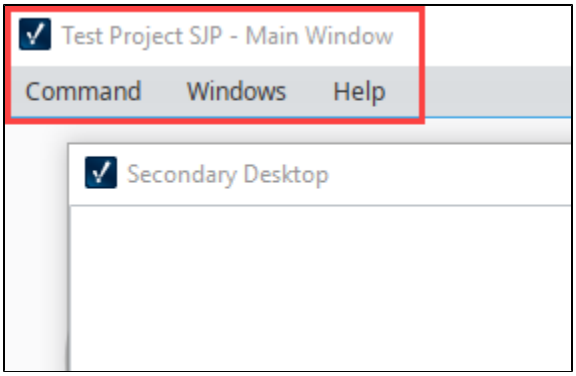
Multi-Monitor Clients

Watch the Video

Spotting the Primary Desktop

When multiple desktops are open, it is important to know that only the primary desktop will have a menu bar. Additionally, the title bar on the client will show the name of the project. Additional Desktops will instead show the handle or index of the Desktop by default. However, a custom title may be used when the desktop is invoked.

Below we see two desktops. The highlighted desktop is displaying the title of the project. Because this is the primary desktop, a menu bar is present. The other desktop was given a title of "Secondary Desktop". A menu bar is not present because this desktop is not the primary.



Project Updates

When a project update is pushed to a Client, the Update banner will only appear on the Primary Desktop (assuming the Update Mode of the project is set to **Notify**). Updating the Primary Desktop will also push the changes to all other local desktop, so there is no need to update each desktop individually.

Opening Another Desktop

Another Desktop may be opened by calling system.gui.openDesktop:

```
#This will open a new desktop without any windows, and a name of "0"
system.gui.openDesktop()
```

However, without specifying which windows to open, the desktop will open without any opened windows. It is recommended to specify at least one window, a title, and a handle for the new desktop. Assuming a window exists at the path "Main Windows/Main Window", the following would open a new desktop, open the specified window, and specify a title and handle for the window.

```
#Create a list of window paths to open in the new desktop
windowToOpen = ["Main Window"]

#Defines a name for the Desktop, which will be used as both the the title and handle of the window.
name = "Secondary Desktop"

#Creates a new desktop. The desktop will open the windows listed above.
system.gui.openDesktop(windows=windowToOpen, title=name, handle=name)
```

Navigating Windows in Desktops

Functions for the **gui** and **nav** scripting modules will execute in the Desktop that originated the call: If the Primary Desktop calls system.nav.swapTo, then the Primary Desktop will swap to a new window, but all other desktops will remain unaffected. However, it is possible for a script on one Desktop to force a navigation or GUI change on another Desktop with the following functions:

- system.gui.desktop
- system.nav.desktop

```
# to open a popup in your second desktop
# if you are identifying desktops by number, they are zero indexed
system.nav.desktop(1).openWindow('Popups/Popup')
```

Additional scripting functions that interact with desktops exist in the **gui** and **nav** scripting modules. Please see the System Functions in the Appendix for more details.

Opening a Desktop on Each Monitor

Sometimes you may want your client to open a new desktop on each of your other monitors. It's pretty simple to get all of your monitors and open a client on each, but then you will have two on your main monitor. The following code block shows you how to skip the primary monitor and even how to open specific windows on each new desktop. This example assumes you have a '**Main Window/Overview**' window, and that window has a custom string property in the root container named '**Display**' to pass values into. Bind a label component to that custom property to easily check your script. This script is best placed in a Client Startup Script to open a client for each monitor on startup.

```
# Get the screen information for all of your monitors.
screensDataset = system.gui.getScreens()

# Open the first window of the project in the (current) primary monitor.
screenIndex = screensDataset[0][0]
monitorNum = screenIndex + 1
primaryScreenText = 'This is Monitor %d' %monitorNum
system.nav.swapTo('Main Windows/Overview', {'Display':primaryScreenText})

# Step through all of the screen information, starting with index 1 instead of 0.
for screenDetails in screensDataset[1:]:
        # unpacks the tuple that is returned for each of the monitors present. Consists of screen index,
width, and height of the screen.
        screenIndex, screenWidth, screenHeight = screenDetails
        monitorNum = screenIndex + 1
        screenText = "This is Monitor %d" %monitorNum

        # Open an empty frame on the next monitor.
        # Assign a handle and apply the width/height for the monitor you are opening on
        handleName = "Monitor %d" %monitorNum
```

```
system.gui.openDesktop(screen=screenIndex, handle=handleName, width=screenWidth, height=screenHeight)

# Open the Main Window on this new desktop and pass the parameters needed.
system.nav.desktop(handleName).swapTo('Main Windows/Overview', {'Display':screenText})
```

Related Topics ...

- system.gui.openDesktop

Local Client Fallback

Ignition Clients are fully dependent on being able to communicate with a Gateway. If Gateway communication is lost, the Client suspends operation while it attempts to reconnect with the Gateway. This can be a problem when you need the Client to monitor critical operations on a plant floor.

Ignition provides a local Vision client fallback mechanism that lets you use a Gateway running on the machine where the client is running. In normal operation, your Client can connect to a central Gateway located somewhere on the network. The central Gateway would be responsible for all data aggregation, such as storing historical data in a database. But if communication to the central Gateway is lost, the Clie nt can automatically retarget to a project that you specify in the local Gateway. This project should contain the minimal realtime information that you need to keep your operation running.

Note: In order to use local client fallback, **port 6501** must be open on the local machine.

On this page ...

- Enable Fallback
- Test Local Fallback
- Automatically Transfer Back

Enable Fallback

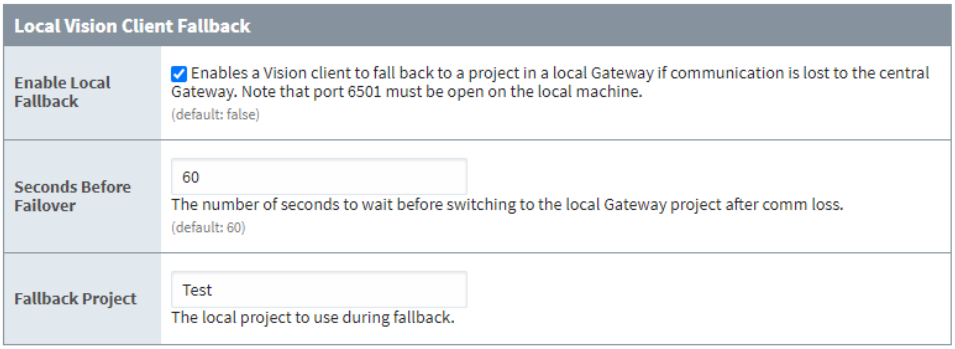
To enable local Vision client fallback, do the following steps:

1. Go to **Config** > **System** > **Gateway Settings** in the local Gateway.
2. Scroll down to the Local Vision Client Fallback section and select **Enable Local Fallback**.
3. Enter the name of the Fallback Project.

   Note: The selected project must be published in the local Gateway, and it must have at least one main window.

4. Optionally, you can change the **Seconds Before Failover** setting to a value other than 60 seconds.

   This setting controls the number of seconds to wait before fallback automatically starts. During comm failure, you can also click a button to load the local fallback project immediately.



When local Vision client fallback is enabled, the Client attempts to open port 6501 on the local machine. If the port can be opened successfully, the Client reads fallback settings from the local Gateway and shows a Fallback Project button on the bottom of the Gateway Connection Lost window. You can click this button at any time to load the fallback project, or simply wait for the fallback project to automatically load. You may want to set the local Client to automatically log in to avoid typing in a username and password when the Client loads. This can be set in the Login section of the project's properties.

Test Local Fallback

Testing local Vision client fallback is highly recommended before you start to depend on it in a production setting. The easiest way to test fallback is to simply unplug the network cable to the Client machine, or disable the network card on the machine. If the Fallback Project button is not visible on the Gateway Connection Lost window, check your local Gateway console and verify that the message "Started Fallback Socket on port 6501" is present in the console. Any other error message related to the Fallback Socket Controller indicates that some other problem has occurred (most likely the port cannot be reserved) and local fallback is not available to Clients.

Automatically Transfer Back

Local Vision Client Fallback will not automatically transfer back to the main Gateway when it is running again, as simplicity was key in this system. You can, however, provide your own solution to automatically switch back. One example is to add a retarget script to your Timer Client Event Scripts to silently try to reconnect.

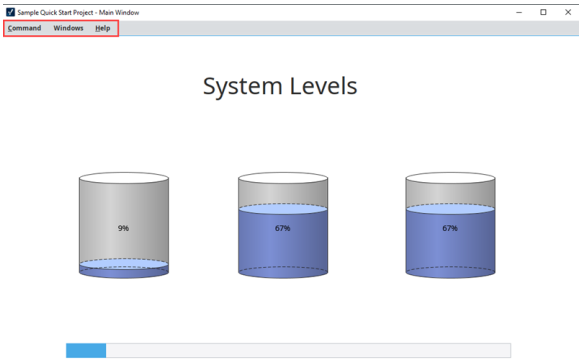
```
# add this to a Client Timer Script running every 30 seconds
# change the ipaddress and project names to match your system
```

```
# allow the main copy of Ignition 10 seconds to give a response
status = system.util.getGatewayStatus("ipaddress:8088/main", 10000)
# if it's running again, retarget
if status == "RUNNING":
    system.util.retarget("my_project", "ipaddress:8088")
```

Vision Client Interface

Vision Clients, launched in either windowed or fullscreen mode, contain a menubar above the configured project windows. By default, this menubar includes a Command, Windows, and Help menus. To partially or completely hide this menubar, access **Project Properties** > **Vision** > **Client Menu** and check either Hide Menu Bar or Hide Windows Menu within the Designer. Navigation within the menubar can be further configured using Client Event Scripts. See the Navigation - Menubar page for an example of how to set up a new menu.

Vision Clients launched in windowed mode, will also display the project title and current window displayed at the top of the Vision Client window.

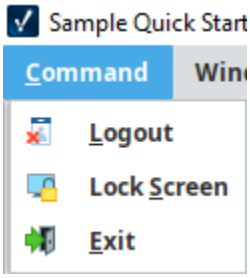


On this page ...

- Command Menu
- Windows Menu
- Help Menu
  - Diagnostics Popup - Vision Client Logs

Command Menu

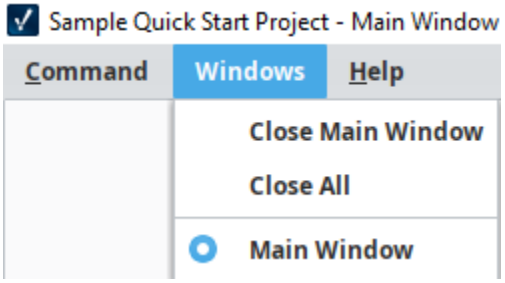
The Command menu lists three options by default. You can add to the Command menu using Client Menubar Scripts.



Function	Description
Logout	Logs out the current users and displays the project log in screen.
Lock Screen	Locks the screen and requires the user who set the lock to enter their password to resume activity. A popup screen will appear when selected with a Screen Locked message and field to enter a password and unlock, or log out.
Exit	Closes the Vision Client.

Windows Menu

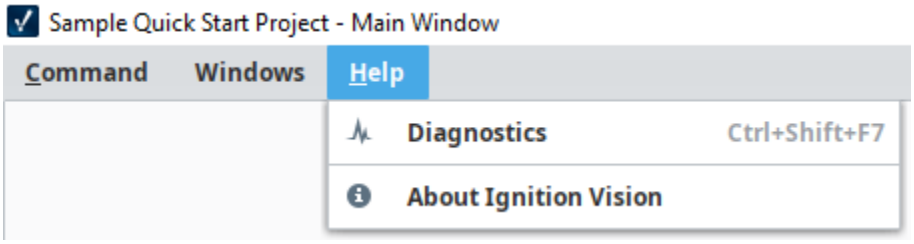
The Windows menu displays all currently opened windows, based on window titles, and allows user navigation between windows.



Function	Description
Close "Main Window"	Closes the displayed window. The window title will change based on the currently opened window.
Close All	Closes all opened windows.
"Main Window"	Lists of all currently opened windows by window title. The list will update as windows are opened and closed. Users can switch windows by clicking between the listed windows. A blue selection icon indicates which window is currently displayed.

Help Menu

The Help menu lists options for users to view Vision Client performance, logs, and other module details.



Function	Description
Diagnostics	Accesses the Diagnostics popup, which includes Performance, Console, Log Viewer, Logging Levels, Thread Viewer, Connections, and Scripts tabs. Tabs display related information and allow user configuration for the Vision Client instance.
About Ignition Vision	Accesses the About Ignition Vision popup that displays Ignition Vision Client details, such as versions, licenses, Gateway information, and Time Zone settings.

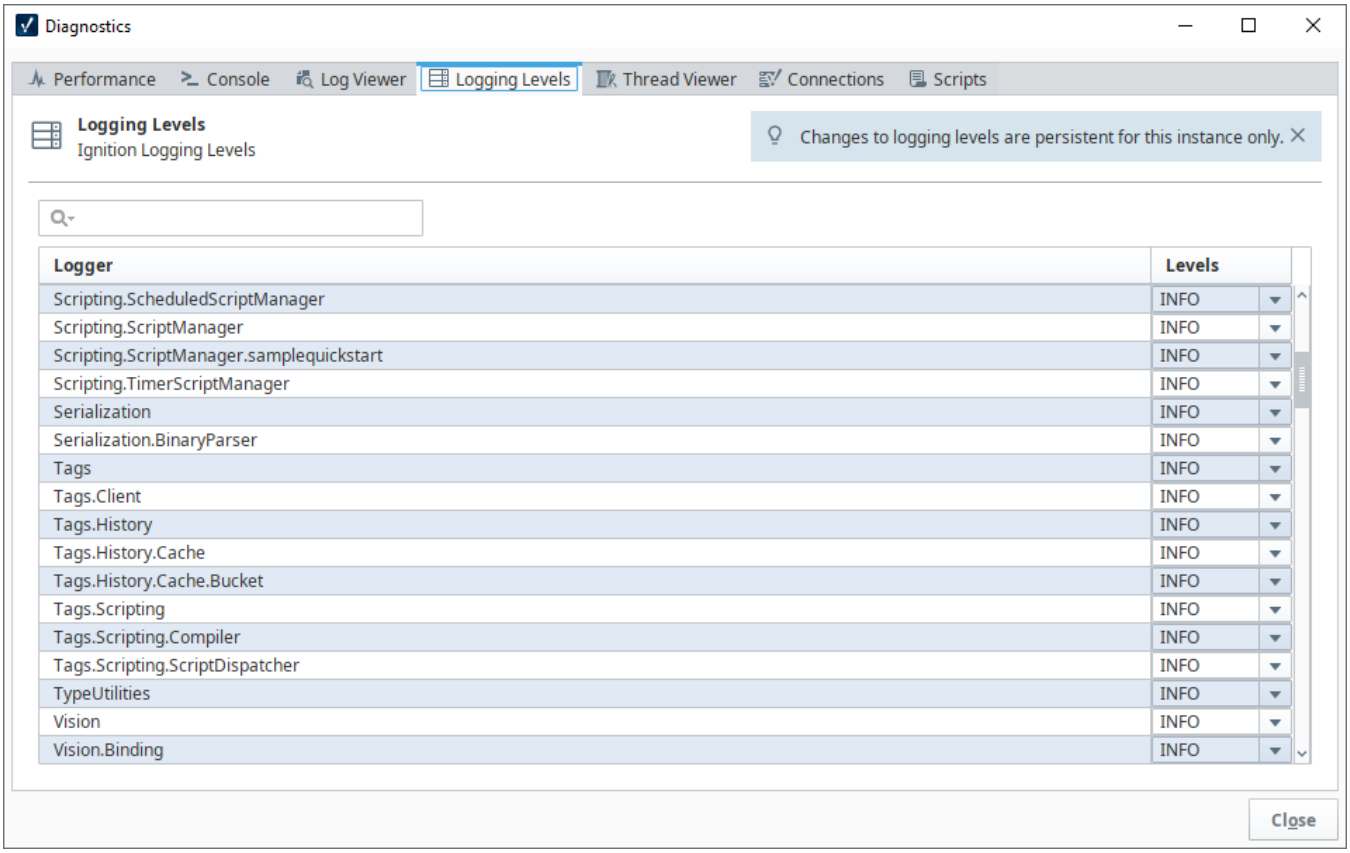
Diagnostics Popup - Vision Client Logs

Vision Clients have their own log files that are only available during runtime. Like the Vision Clients log display on the Gateway, these logs contain information about the operation and errors of the Vision Client. However, the Vision Client log files can only be obtained from the runtime Diagnostics window. There is no way to view them using the Gateway's Web interface.

These Client-scoped loggers, such as bindings loggers, can be extremely useful when troubleshooting issues that occur in the Vision Client and are not Gateway-scoped. Users can set logger levels in the Logging Levels tab and see results in the Log Viewer tab.

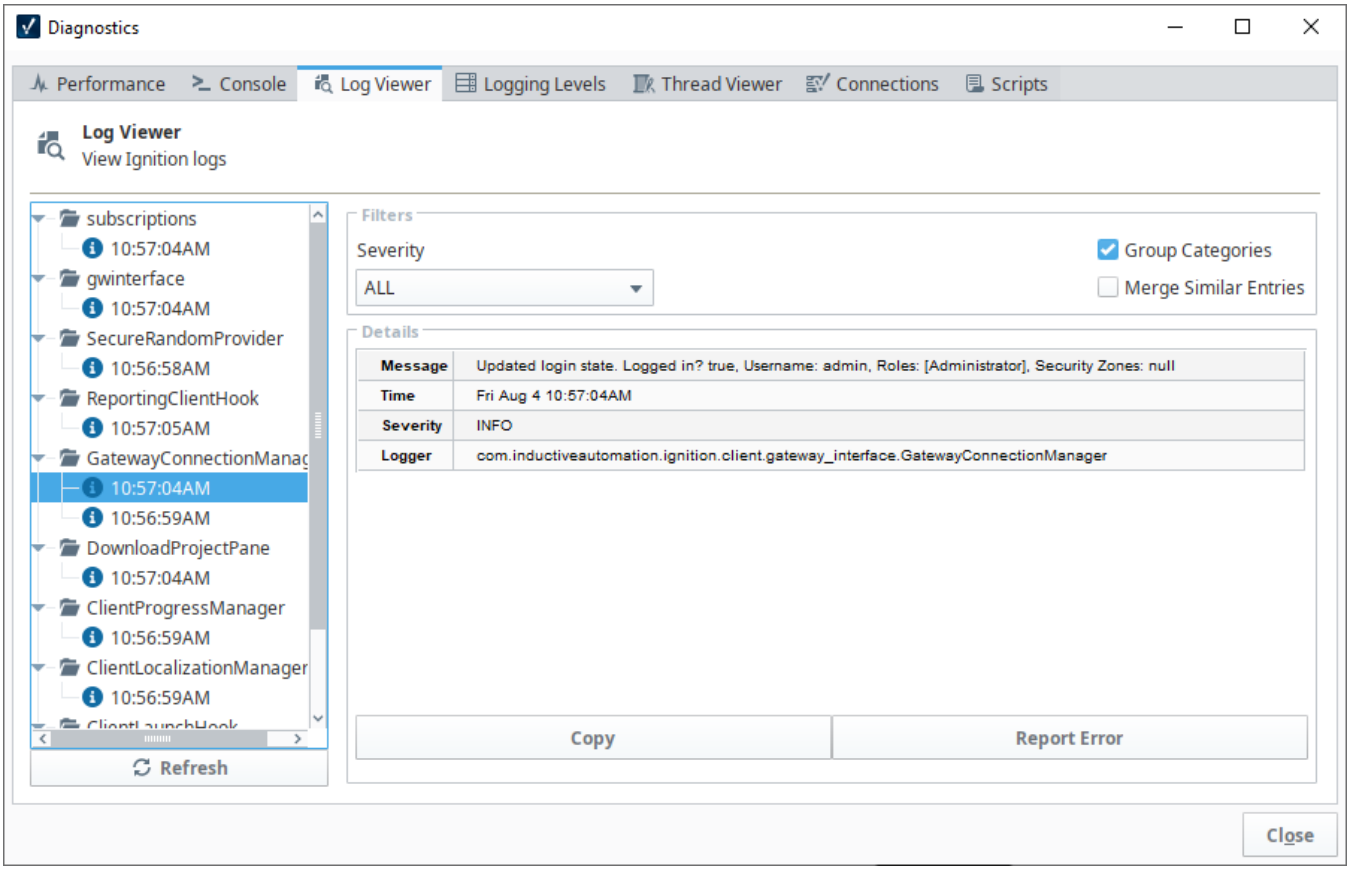
Logging Levels Tab

The Logging Levels tab lists all available loggers for the Vision Client instance. Use the search bar to quickly filter loggers and the level dropdowns to set different logger levels. By default, loggers are set to info, but loggers can be switched to trace, debug, warn, error, and fatal as desired. Changes in logger levels are immediately reflected on the Log Viewer. All changes made to the logging levels are persistent for the current instance only and will need to be reset upon restart.



Log Viewer Tab

The Log Viewer Tab displays certain logs by default, such as subscriptions, hooks, and managers. The displayed logs will update based on logger level settings and can be refreshed at the bottom of the list. Selecting a log will access the full log details including Message, Time, Severity, and Logger name. Use the filters available at the top of the Log Viewer page to adjust severity and grouping options.


Diagnostics
Performance
Console
Log Viewer
Logging Levels
Thread Viewer
Connections
Scripts
Log Viewer
View Ignition logs
subscriptions
10:57:04AM
gwinterface
10:57:04AM
SecureRandomProvider
10:56:58AM
ReportingClientHook
10:57:05AM
GatewayConnectionManag
10:57:04AM
10:56:59AM
DownloadProjectPane
10:57:04AM
ClientProgressManager
10:56:59AM
ClientLocalizationManager
10:56:59AM
Refresh
Filters
Severity
ALL
Group Categories
Merge Similar Entries
Details
Message
Updated login state. Logged in? true, Username: admin, Roles: [Administrator], Security Zones: null
Time
Fri Aug 4 10:57:04AM
Severity
INFO
Logger
com.inductiveautomation.ignition.client.gateway_interface.GatewayConnectionManager
Copy
Report Error
Close