Ignition 8.3 User Manual — Platform: Tags, Alarming, Security and Scripting

Generated 2026-10-12 from https://www.docs.inductiveautomation.com/docs/8.3/

Tags

What Is a Tag?

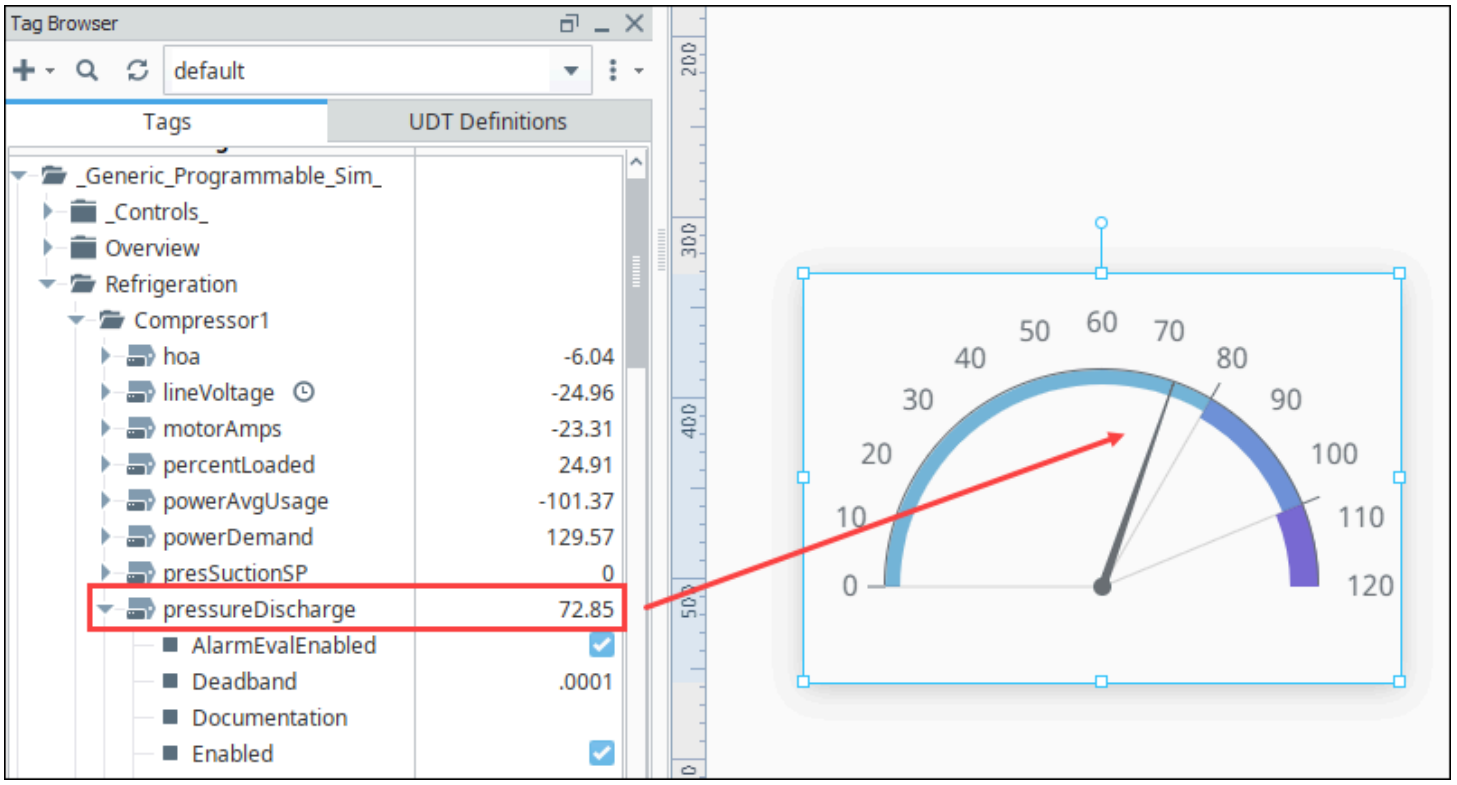
Tags are points of data and may have static values or dynamic values that come from an OPC address, an expression, or a SQL query. The values can be used on screens and in transaction groups.

Tags provide a consistent data model throughout Ignition, and offer the easiest way to get up and running creating realtime status and control systems. Despite their fast initial learning curve, however, tags offer a great amount of power in system design and configuration. The ability to aggregate tags from a variety of installations means that you can build widely distributed SCADA systems more easily than ever before with a high level of performance and relatively easy configuration.

While the goal of tags in Ignition is to create an easy yet powerful model, the variety of options and terminology can sometimes make configuration confusing. Tags are created and controlled using both the Gateway and the Designer for configuration.

- In the Designer, you create or import the tags. There are several types of tags such as an OPC tags and Memory tags. Each tag has many properties and other functionality such as alarming, history, etc. Once your tags are created, you can use them in your windows, views, and reports.
- On the Gateway, you create and modify Tag Providers. You can create these Realtime Providers to store groupings of tags for use in your projects either locally in Ignition or share them externally with connected Gateways. There are also Historian Providers used to store historical data for the tags, but these are automatically created for each data source you have. These Tag Provider configurations in the Gateway apply globally to all your projects.

The following example shows a pressureDischarge tag in the Tag Browser and a Gauge component in the Designer. The value on the Designer component is bound to the tag and updates in realtime. This is just a simple example of how tag values can be represented in your SCADA designs.



Tag Providers

There are two types of Tag Providers: Internal and Remote. By default, a fresh Ignition installation will have an internal Tag Provider. This can be thought of as a standard internal tag database, and stored in the Ignition Gateway. Additionally, it is possible to create remote **Tag Providers**, linking one installation of Ignition to the tags on another Ignition. This ability opens up some very flexible architectures.

Tag User Defined Types

Tag **User Defined Types** (UDTs) provide an object-oriented approach to tag building, allowing you to define parameterized data types, extend and override types, and then rapidly generate instances. A change to the type definition is then inherited by all instances, drastically saving time when making routine changes. The UDTs are fully supported by Vision templates, which means you can configure templates for your custom data types and take advantage of drag-and-drop binding to rapidly build complex screens.

Tag Features

Tags work naturally and easily with Ignition to offer the following features:

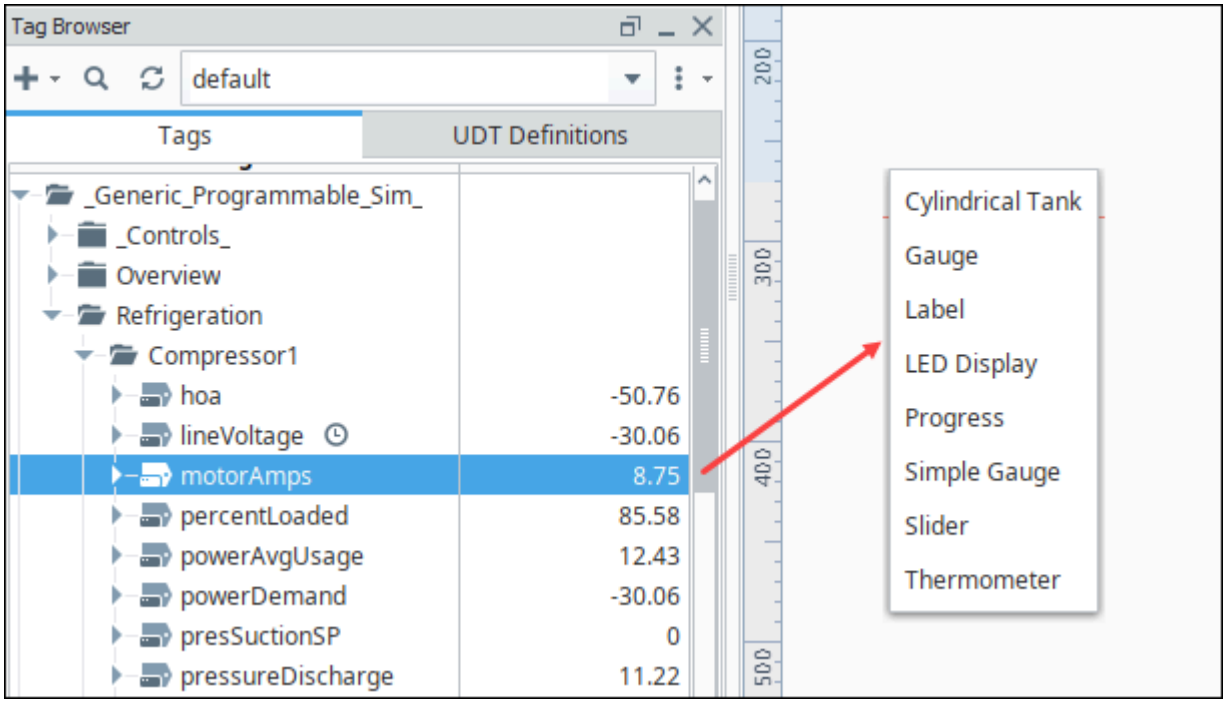
Performance and Scalability Tags offer great performance on the Gateway, in Perspective Sessions, and in the Vision Client. On the Gateway, the system can support many thousands of value changes per second and millions of tags. In runtime, Tags improve efficiency with their lightweight subscription

architecture. Adding additional Clients creates a nearly negligible effect on the database and the Gateway performance.

Object-Oriented Design Use **Tag UDTs** (User Defined Types) to design re-usable, parameterized, and extendable data types. You can create and configure new instance tags in seconds, saving a great amount of time over traditional tag systems.

Powerful Alarming Model Each tag can have any number of **alarms** configured on it. There are many different alarm modes accommodating simple digital alarms, analog high/low value alarms, as well as more specialty alarms like bad data quality and bit-packed alarms. The settings for alarms can bound to other tags, making the alarm configuration dynamic.

Drag-and-Drop Screen Design You can drag and drop tags onto a window or view to automatically create new bound components. Drag tags onto existing components or properties to quickly bind them to the data.

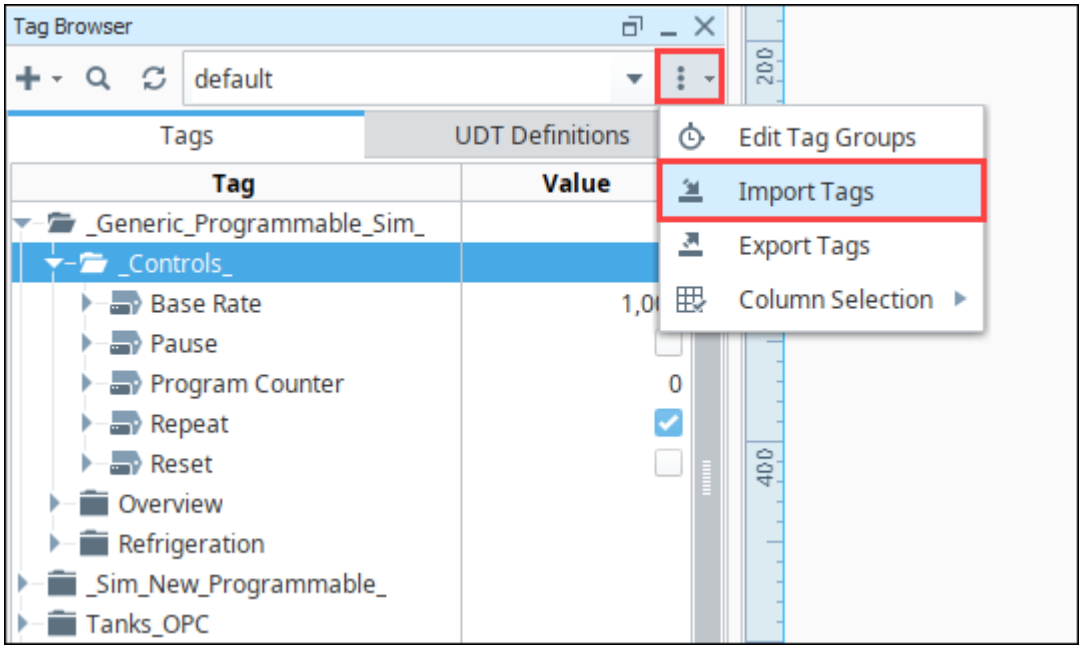


Historical Logging The **Tag Historian Module** makes it easier than ever to store and use historical data. When you simply select a check box on a tag, historical data is stored in an efficient format in your SQL database. This data is then available for querying through scripting, historical bindings, and reporting. Also, you can drag-and-drop tags directly onto an many components to create trends or display historical values. The Tag Historian's robust querying provides you great flexibility in how you retrieve the data.

Importing and Exporting Tags

Ignition tags can easily be imported and exported from the Designer by selecting the More Options menu, then either the Import Tags icon or Export Tags icon. You'll be prompted to choose

the tags or folders that you want. See the **Exporting and Importing Tags** page for more information.



Tag Naming

Tags names are flexible and to not have to match data source names (like an OPC path) or tag codes (such as N7, F8, etc.). It is not necessary that tag's name be related at all to its underlying data source (OPC path, for instance). This provides a level of indirection that is convenient for systems with many repeat tag structures.

It is important to give tags a meaningful structure and arrange them in hierarchical tag folders so that they are easy to understand, identify, and locate for all developers. By default, Ignition tags are named after their OPC Server address when a tag is dragged into the Tag Browser. You can change this name to just about anything that you want. We recommend using names that mean something to your process, such as "Motor 3 Amps." Alternatively you could create folders in your Tag Browser such as "Motor 3/Amps.". When renaming tags and folders, there is really only one question to ask: "does this structure make sense?"

Another important concept to consider when naming and organizing your tags, is to do this early in your project. If you rename or move any of your tags to another folder, and your tag is being used in other places, chances are you are going to break the reference to the tag on your screen. So keeping your tags organized and defining your tag structure early on in your project is critical.

When you choose a new name for your tags and folders, there are some rules that must be followed. The first character of the tag name must be one of the following:

- Letter - specifically, a letter as recognized by Unicode's Letter (L) category.
- Number

- Underscore

The second character, and every character after that can then be one of the following:

- Letter
- Number
- Underscore
- Space
- Any of the following special characters: `'`, `-`, `:`, `(`, `)`

All other special characters are not allowed in a tag name.

Tag Browser

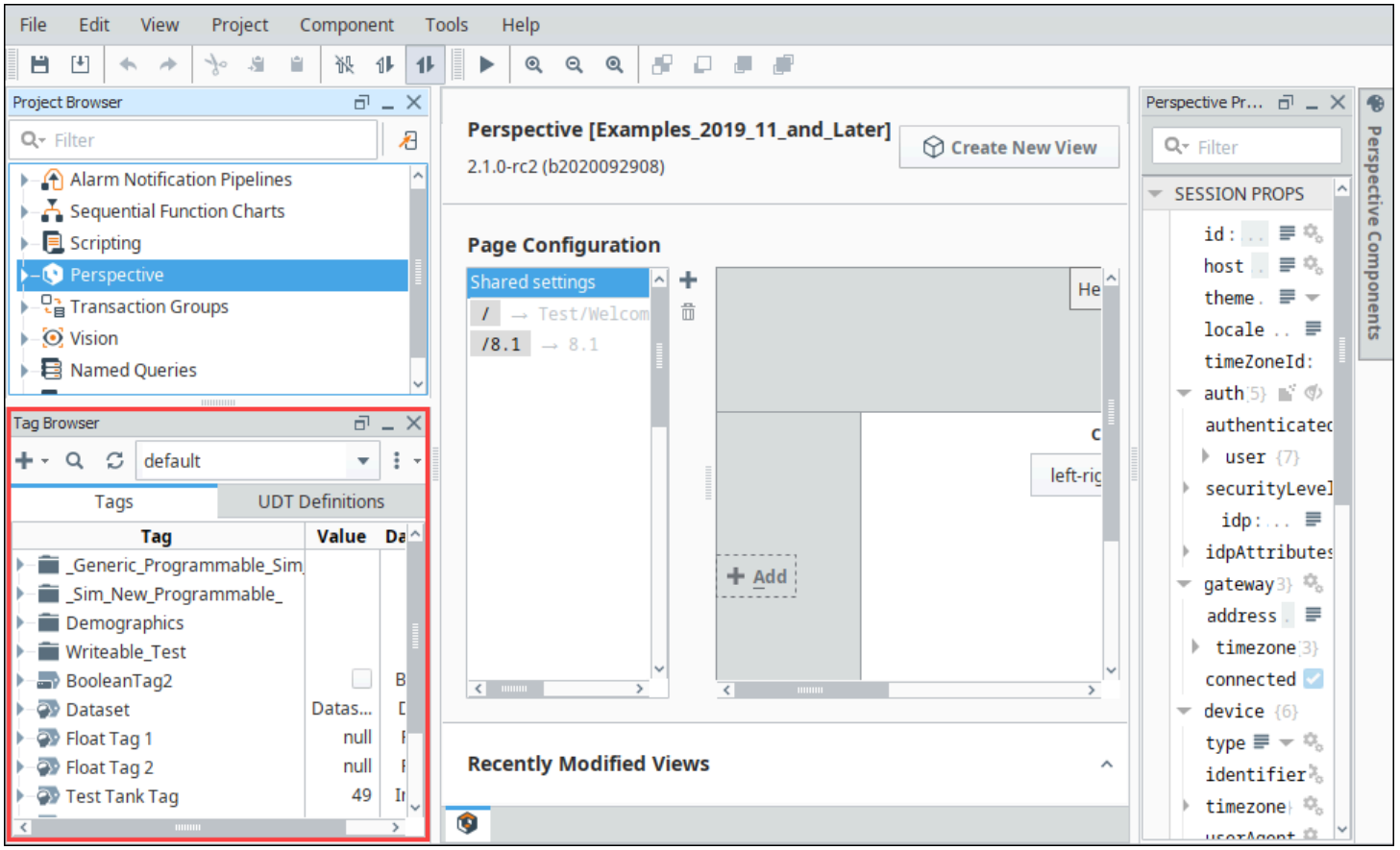
INDUCTIVE UNIVERSITY

Tags in Ignition

Watch the video

The Tag Browser is the central location for interaction with all types of tags on your system. It gives you full view of the tags including the current value, datatype, and any traits. When panels are in their default configuration in the Designer, the Tag Browser appears on the left side.

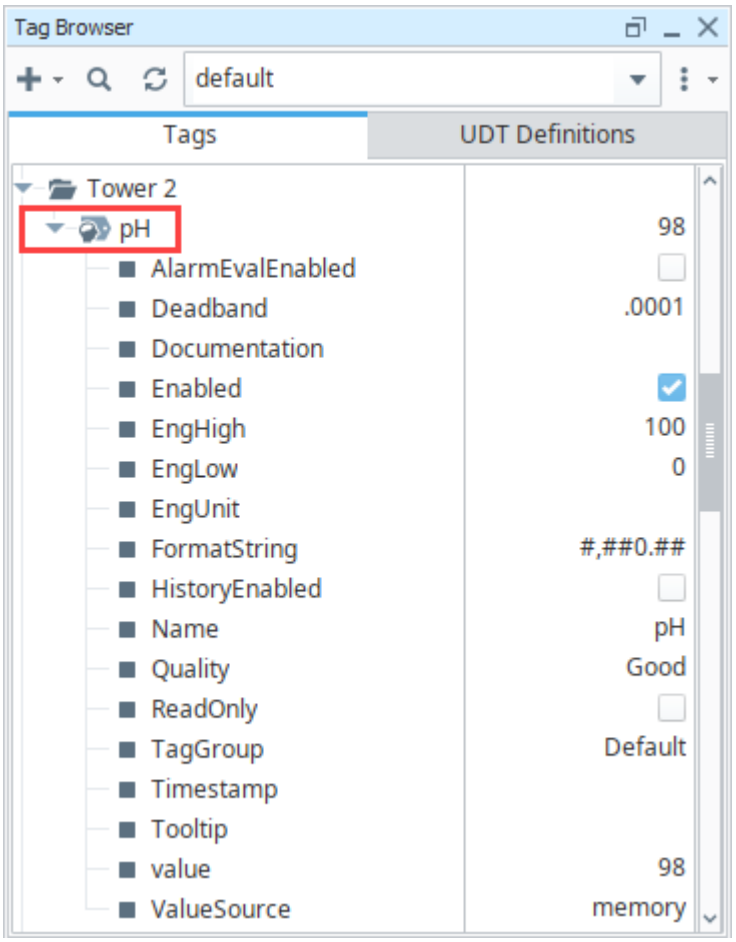
Tags and **UDT Definitions** have their own tabs. The Tag Provider Selector enables you to view tags for a specific Tag Provider. There is an interface for **Creating tags**. Each **tag type** will also have their own icons.



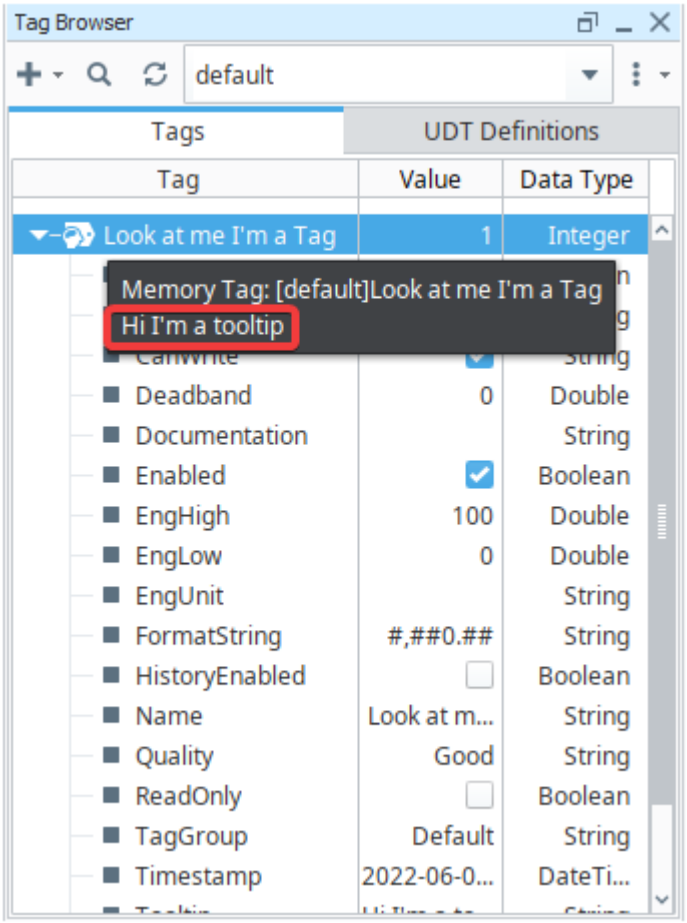
Tag Browser Tree

The Tag Browser is set up in an interactive tree structure with folders that can be expanded or collapsed to view more Tags.

Click on the **Expand** ▶ icon to expand any folder or the **Collapse** ▼ icon to collapse the folder. In the example below, the pH tag for Tower2 was expanded.



If you create a tooltip on a tag by filling in the **Tooltip property**, hovering over the base tag at the top of the Tag Browser Tree will display the tooltip.

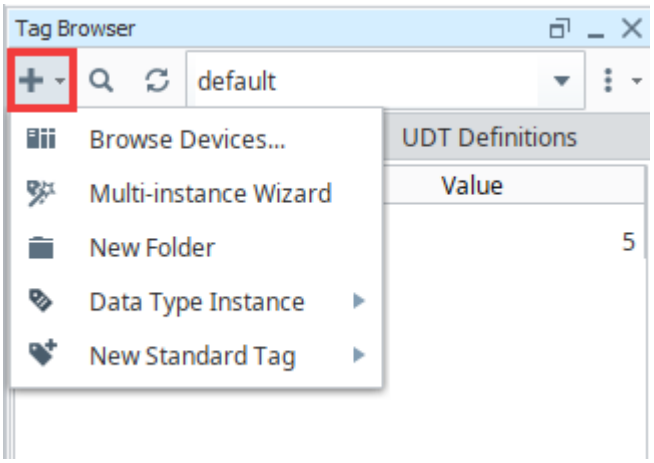


Tag Browser Toolbar

The Tag Browser toolbar has several options for working with tags.

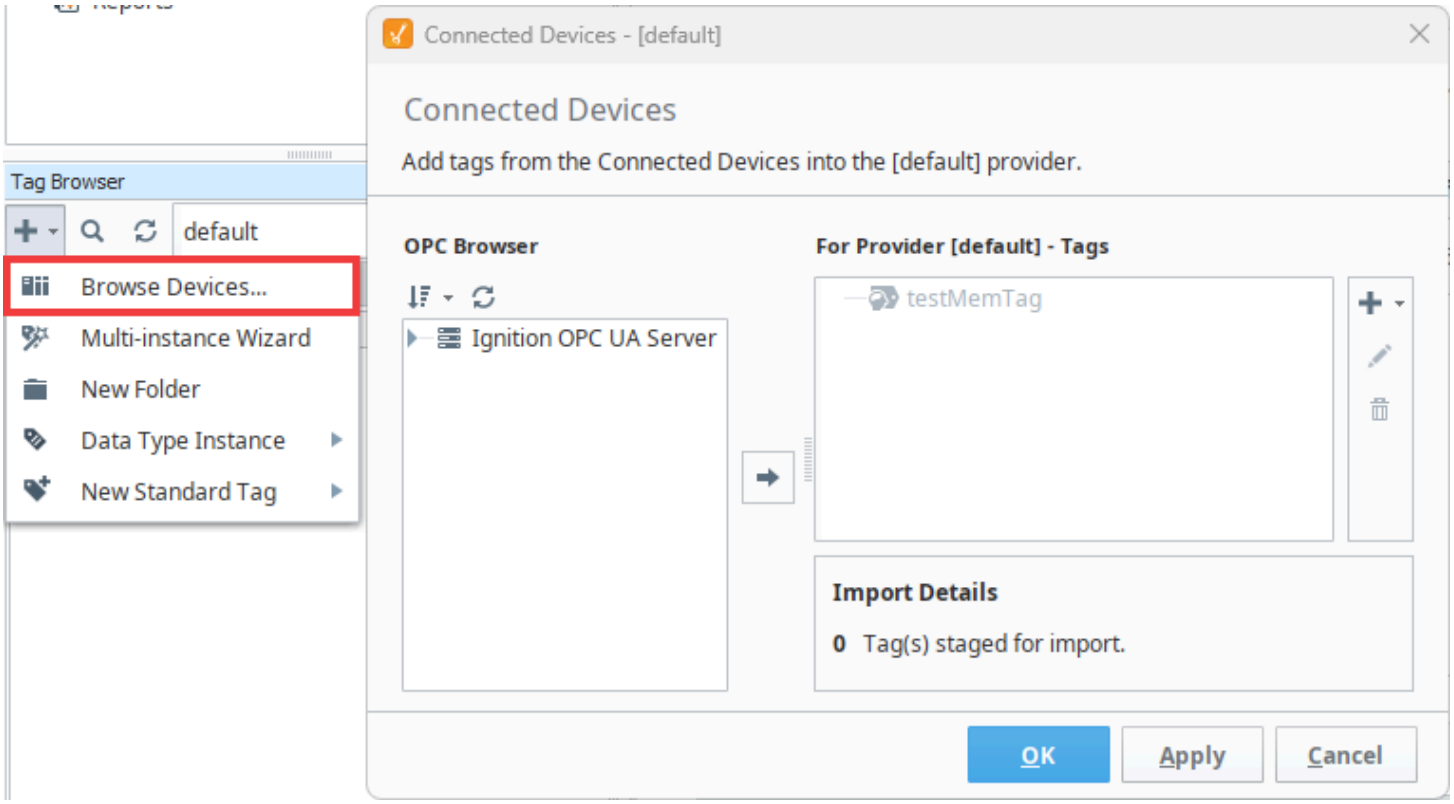
Add Tag

The **Add** + icon opens a context menu showing all the options to Browse Devices, add a tag, Folder, UDT Instance or a UDT Definition. The new object is added under the Folder you selected, or as a sibling to the tag you selected. This button is disabled if there is no selection.



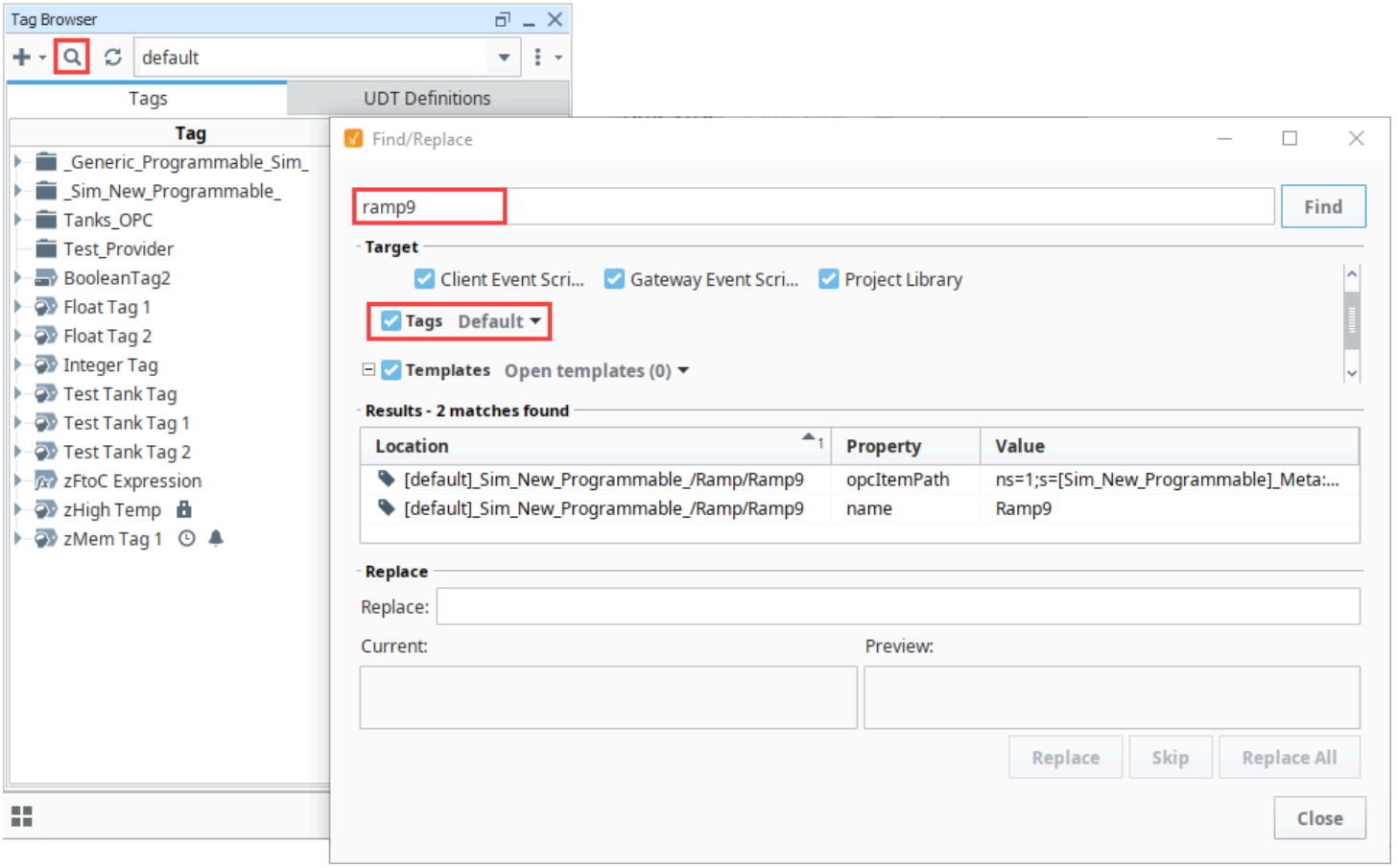
Browse Devices

With the Browse Devices, you can browse to find external PLC or OPC tags. Click the **Add** icon at the top of the Tag Browser to open the Connected Devices window. You can then select Tags and move them to the Tag Browser to be used in the Ignition system. For complete information, see Creating Tags.



Find/Replace

Clicking on the **Search** icon in the Tag Browser will open up the Designer's global **Find/Replace** screen. In the the example below, we searched for the Ramp9 tag and limited the search to the default tags. Results are shown at the bottom of the screen. For additional information, see **Find and Replace**.



Refresh Providers

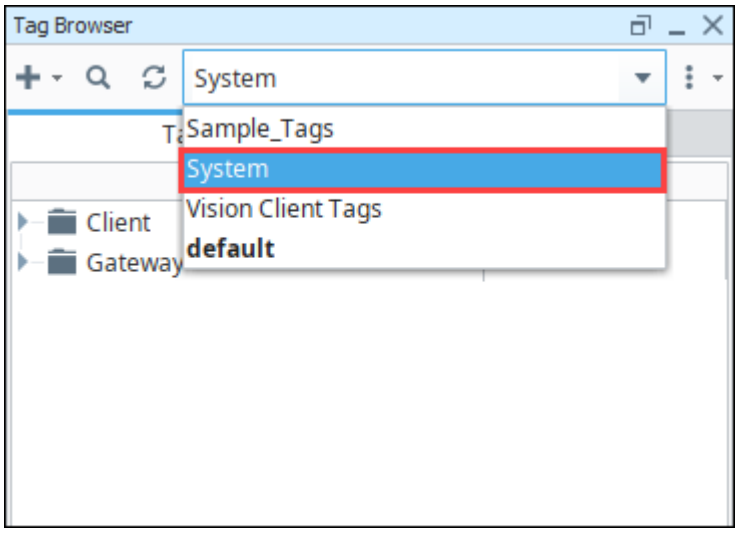
The **Refresh Providers** icon refreshes all of the Providers in the Tag Browser. This is useful if you or others have modified tags and do not see an update. In general, this button is not used very often.

Tag Provider Selector

The Tag Provider Selector is a dropdown menu with a list of available Tag Providers. Use this selector to view the System tags, Vision Client tags, default tags, or other **Tag Providers** you have in your project.

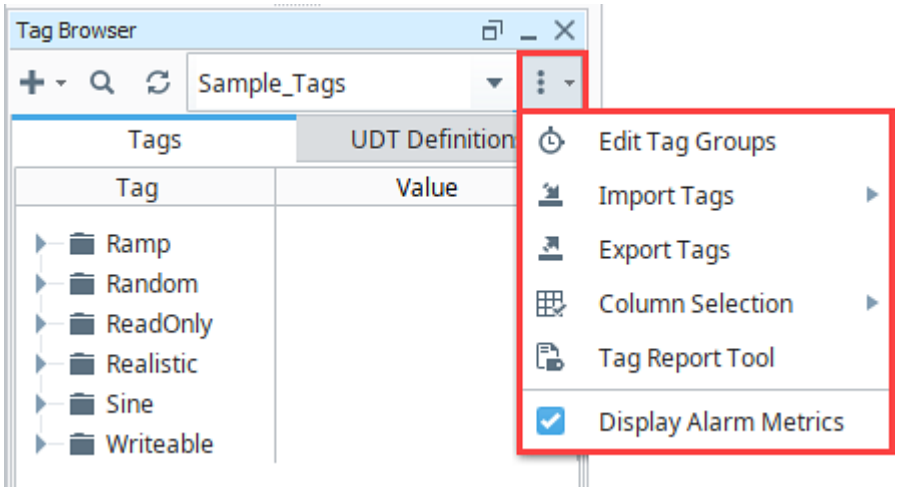
On a new install of Ignition, a single "default" Tag Provider is created for you, but there are no tags added to the "default" provider initially. You'll notice the "default" provider is in bold. The bold entry in the dropdown list is determined by the Default Provider for the project. You can change the default provider in **Project Properties > Project > General > Tag Settings > Default Provider**. Save and restart the Designer, and the next time you open the Tag Provider in the dropdown of the Tag Browser, a different Default Provider will be displayed in bold.

When a provider is selected to `System`, the Browsing Devices under the **Add** icon is disabled.



More Options Menu

A **More** options icon on the upper right of the Tag Browser, opens a dropdown menu of additional options.



Tag Groups

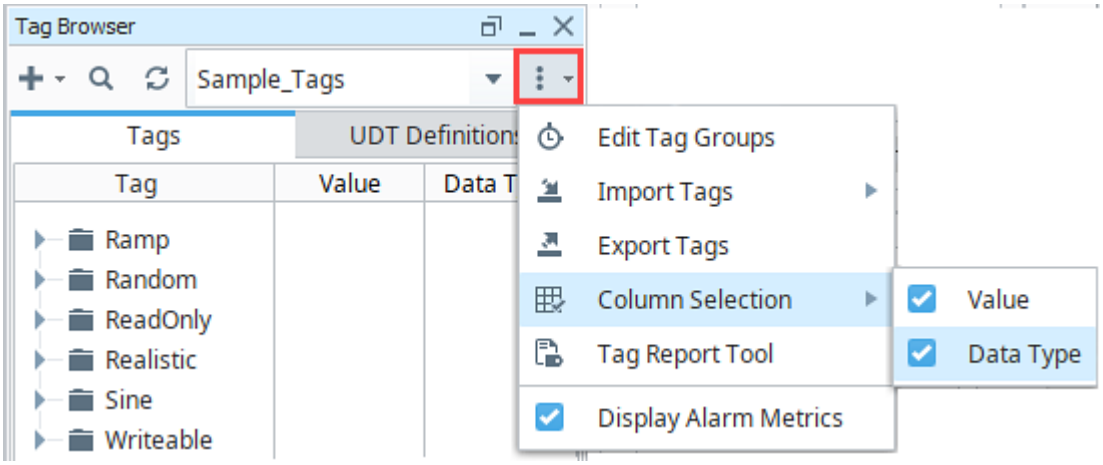
In the Tag Browser, the **Edit Tag Groups** opens the **Tag Group Editor** window. Tag Groups dictate the rate of execution of tags. This is where you set up your Tag Groups and scan rates. See **Tag Groups** for more information.

Import/Export

Ignition can export and import tag configurations to and from the JSON (JavaScript Object Notation) file format. Use the **Import** icon or **Export** icon to import and export tags in the Gateway. See **Exporting and Importing Tags** for more information.

Column Selector

The Tag Browser displays the **Value** and **Data Type**. The Value type is set by default. To toggle any of the options, click on the **Column Selection** icon, then click the checkbox. In the example below, the Data Type is enabled.



Tag Report Tool

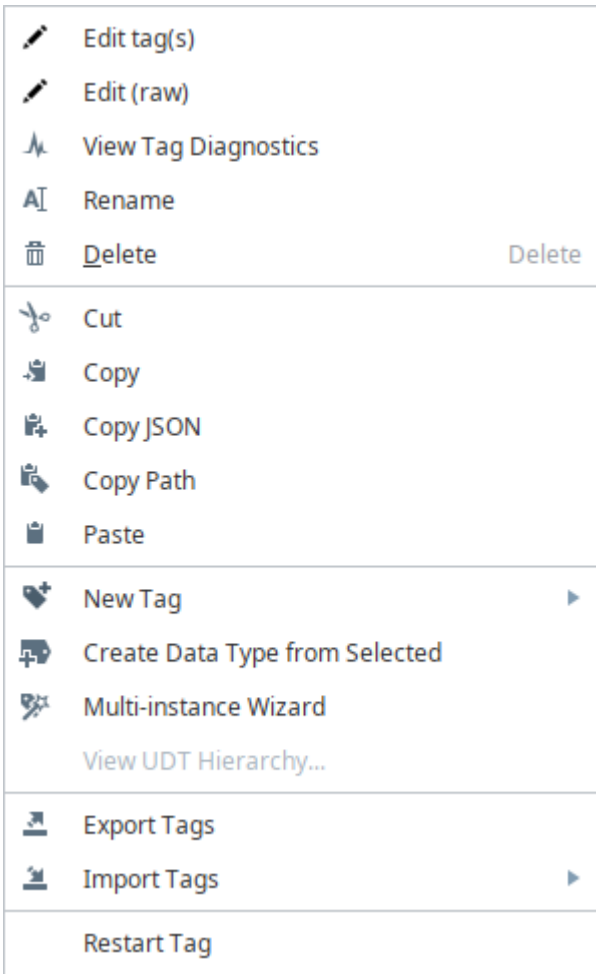
Ignition allows users to search for tags by tag path, quality, type, traits, and values through the **Tag Report Tool**. Click on **Tag Report Tool** to open the editor and begin configuring a search within the currently selected Tag Provider.

Display Alarm Metrics

The Tag Browser displays the **Alarm Metrics** values by default. To change the visibility of Alarm Metrics, toggle the **Display Alarm Metrics** checkbox as needed.

Right-Click Menu

Editing tags is done mostly through the **Tag Browser**. The Tag Browser allows you to right click on a tag or folder to perform any of the following functions. Different objects will have different options available. The special **Data Types** folder is slightly different than a regular folder and will have even fewer options.

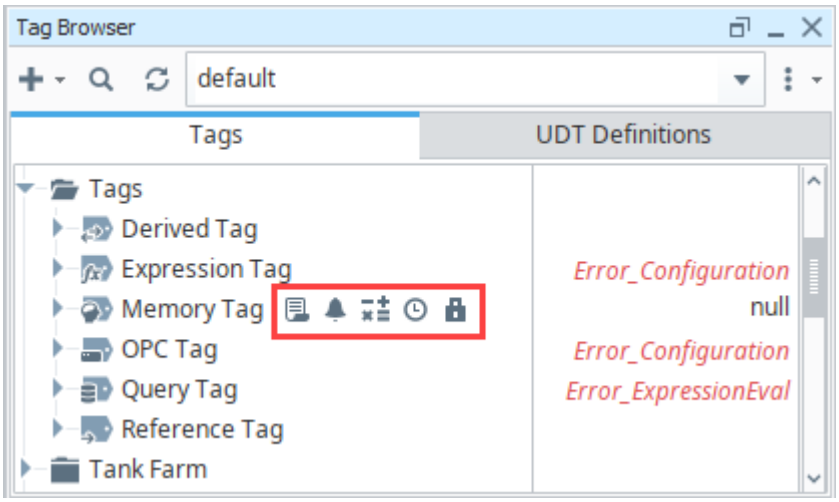

Edit tag(s)
Edit (raw)
View Tag Diagnostics
Rename
Delete
Delete
Cut
Copy
Copy JSON
Copy Path
Paste
New Tag
Create Data Type from Selected
Multi-instance Wizard
View UDT Hierarchy...
Export Tags
Import Tags
Restart Tag

Function	Description
Edit Tag	Disabled when a Folder is selected. Opens the Tag Editor window so the tag can be edited.
Edit (raw)	Disabled when a Folder is selected. Opens a JSON editor, allowing you to view and edit the underlying JSON that makes up the tag. **Note**: For UDTs, the default **Overwrite** policy replaces the entire definition and removes members not in the import. Use **MergeOverwrite** to keep existing members and values. See the **Exporting and Importing Tags** page for a list of Collision Policy options.
View Tag Diagnostics	Opens the Tag Diagnostics window, which also accesses Active Subscriptions and Reference Log data.
Rename	Renames the current selection.
Delete	Deletes the current selection.
Cut	Cuts the current selection into the clipboard.
Copy	Copies the current selection into the clipboard.
Copy JSON	Available for non-client tags only. Copies the JSON for the selected tags into the system clipboard. In addition, pasting the JSON into a different provider/designer will create or overwrite tags.
Copy Path	Copies the currently selected tag path into the clipboard.
Paste	Pastes the content in the clipboard into the selected context.
New Tag	Disabled when a tag is selected. For Folders, this option opens a sub-menu to create a tag or tags. • New Folder - Creates a new Tag folder • Data Type Instance - Creates a new instance of an existing data type. The instance is linked to the parent type so when the parent changes, the instances are overwritten with the parent type changes. Sub Menu - based on Data Types • New Standard Tag - Creates different types of Tags such as Derived, Expression, Memory, OPC, Query, and Reference tags. Sub Menu -

Function	Description
	Standard Tag Types
Multi-instance Wizard	Creates many instances of a UDT at the same time.
Export Tags	Exports the selected tags.
Import Tags	Imports tags into the project.
EAM	Sends tag to EAM Agents.
Restart Tag	Attempts to restart the selected tag. "Restarting" a tag effectively refreshes the following traits: • Value generation (Expressions for expression tags, queries for Query tags, OPC value for OPC tags, etc). • Scaling • Engineering Limits • Alarms • Deadbands • Event Scripts If a folder is restarted, then all tags under the folder will restart.

Tag Traits

Certain settings or tag configurations are visually represented next to the tag in the Tag Browser.



The following icons enable you to note some important settings on the Tag at a glance. A description of the icons are listed below.

Icon	Setting	Description
	Scaling	The Scale Mode property under the Numeric **Tag Properties** section of the Tag Editor has been set to a value other than "Off." The value on the tag will be scaled to some degree.
	Alarming	At least one **alarm** has been configured on this tag.
	Tag History	This tag has been configured to log data into the **Tag Historian** system.
	Tag Event Script	At least one **Tag Event Script** has been enabled on this tag.
	Lock	Shows the tag has permissions enabled.
	Inheritance	Denotes inheritance. Displays the Parent Type, name of the UDT and instance name (i.e., Parent Types: Motor UDT > Complex Motor). Refer to **UDT Inheritance Traits**.
	Override	Denotes that the property in the UDT instance overrides the parent property. Refer to **Overriding Properties of the Parent UDT**.

Types of Tags

There are many different types of tags in Ignition: standard tags, **System Tags**, and **Vision Client Tags**. All these tag types are available in the Tag Browser.

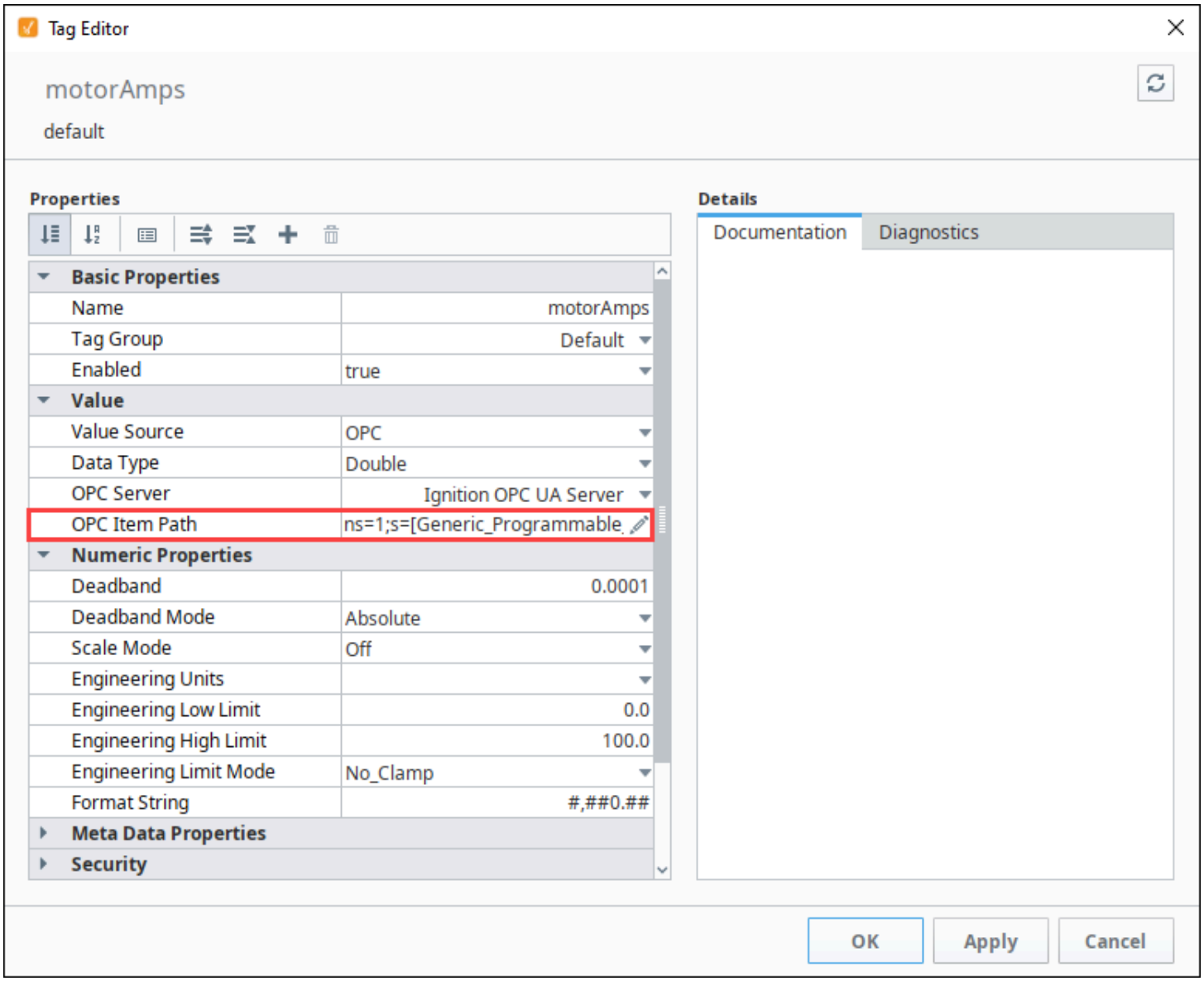
Tags executed in the Gateway support all of the primary features of tags: scaling, alarming, history, and role-based permissions. These tags run in the Gateway, and the values of the tags are shared among all running sessions and clients. They are identical in their configurations, apart from defining how the value is generated. Each tag type has its own icon in the Tag Browser.

Icon	Tag Type
	Memory tag
	OPC tag
	Expression tag
	Query tag
	Reference tag
	Derived tag
	System Client tags, Vision Client tags

OPC Tags

An OPC tag is driven by an OPC Item Path and OPC server. The OPC Item Path is a string path to a particular device connection. The exact path is defined by the driver and OPC server used to communicate with the device. Many drivers support browsing, allowing you to automatically create OPC tags by dragging-and-dropping from the OPC Browser. However, in cases where browsing isn't supported, OPC tags can manually be created.

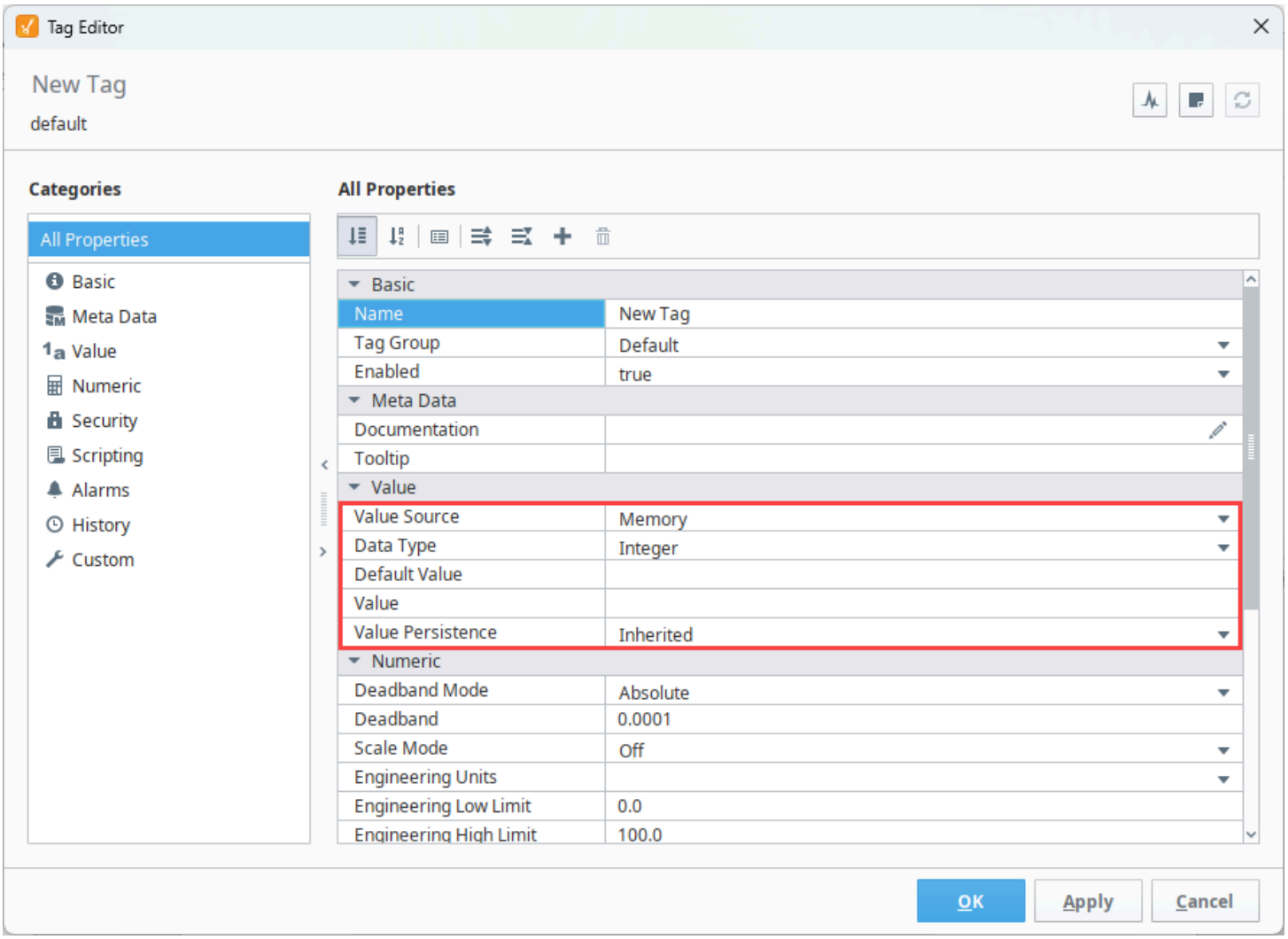
In the **Tag Browser**, double click on any existing OPC tag, to see the the **OPC Server** name and **Item Path**.



Memory Tags

Memory tags are simple tags, that do not automatically poll or update their value. They hold the same value until some other user-created mechanism (most likely a script or binding) changes their value. They're useful in situations where a value must be stored outside of a PLC or database.

The **Value Persistence** setting determines how the Memory tag values are stored. This option will impact whether the tag value is retained across Gateway restarts.

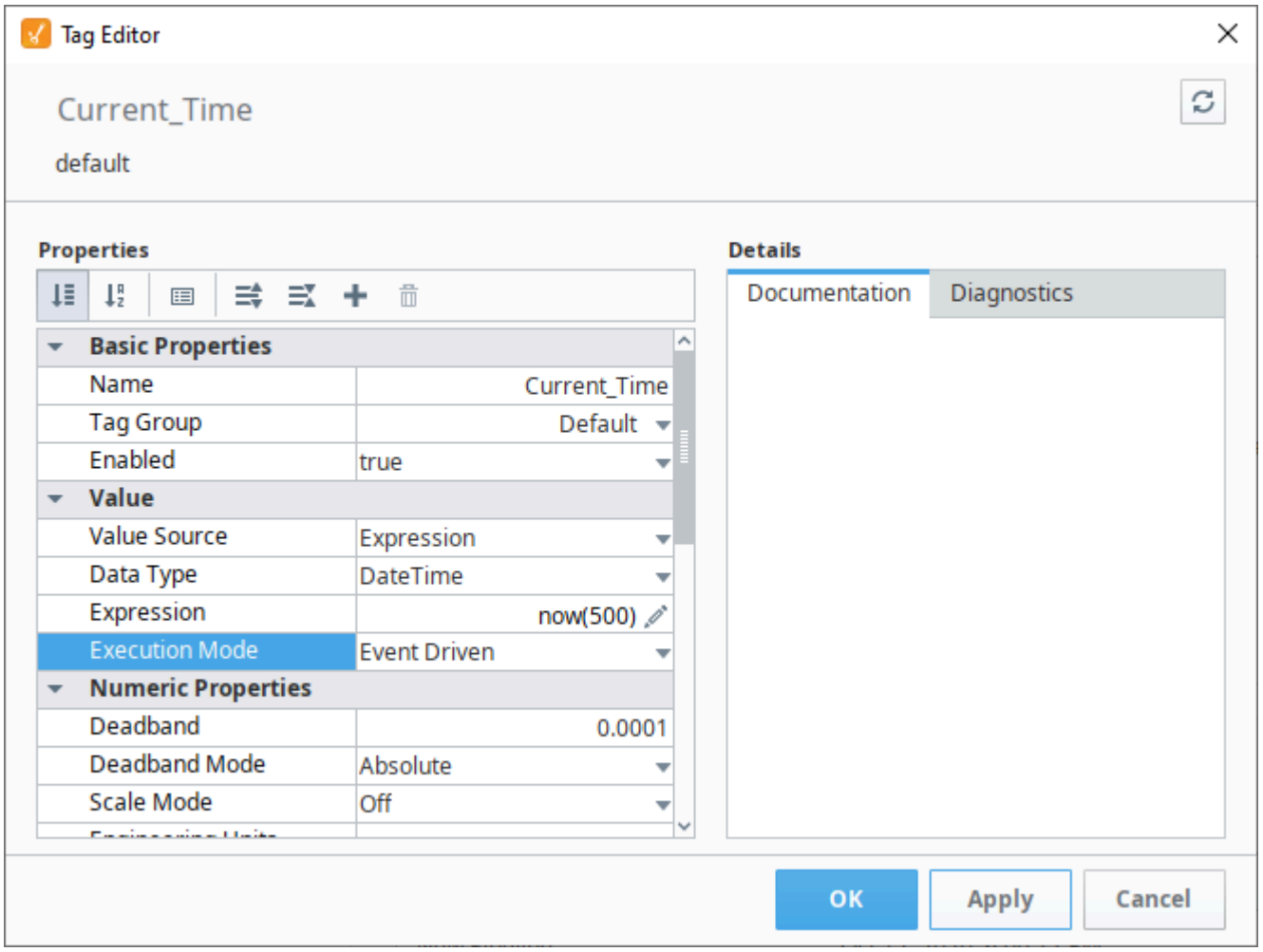


Expression Tags

Expression tags are driven by an **expression**, allowing their values to be determined from a calculation.

The **Expression** property on Expression tags determines their value. The expression can reference values and properties on other Gateway-scoped tag values. However, due to scoping, they can not reference property values on Vision Client and Perspective Session components.

The expression on an Expression tag executes based off of the **Execution Mode**. More information on Execution Mode can be found on the **Tag Properties** page.

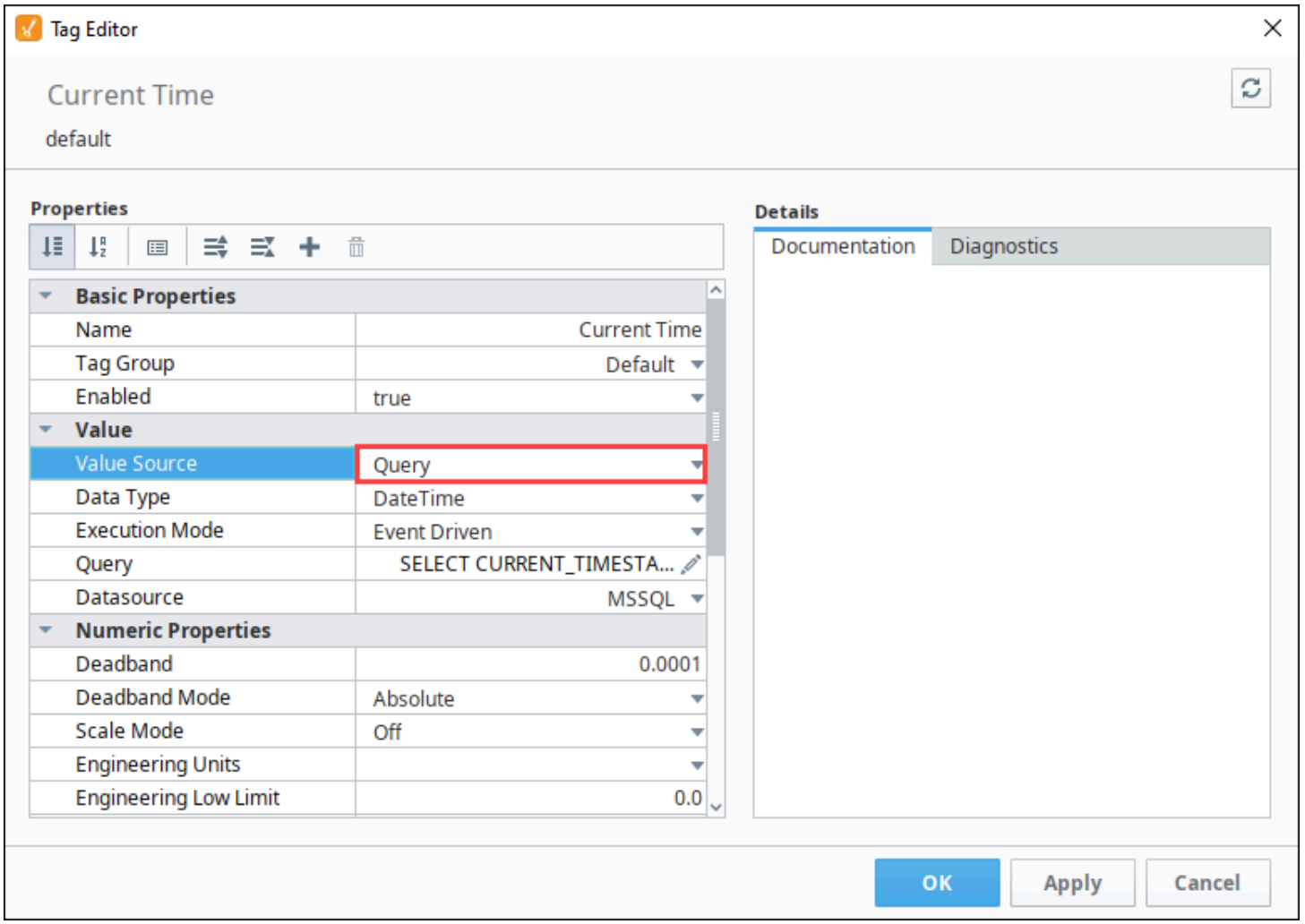


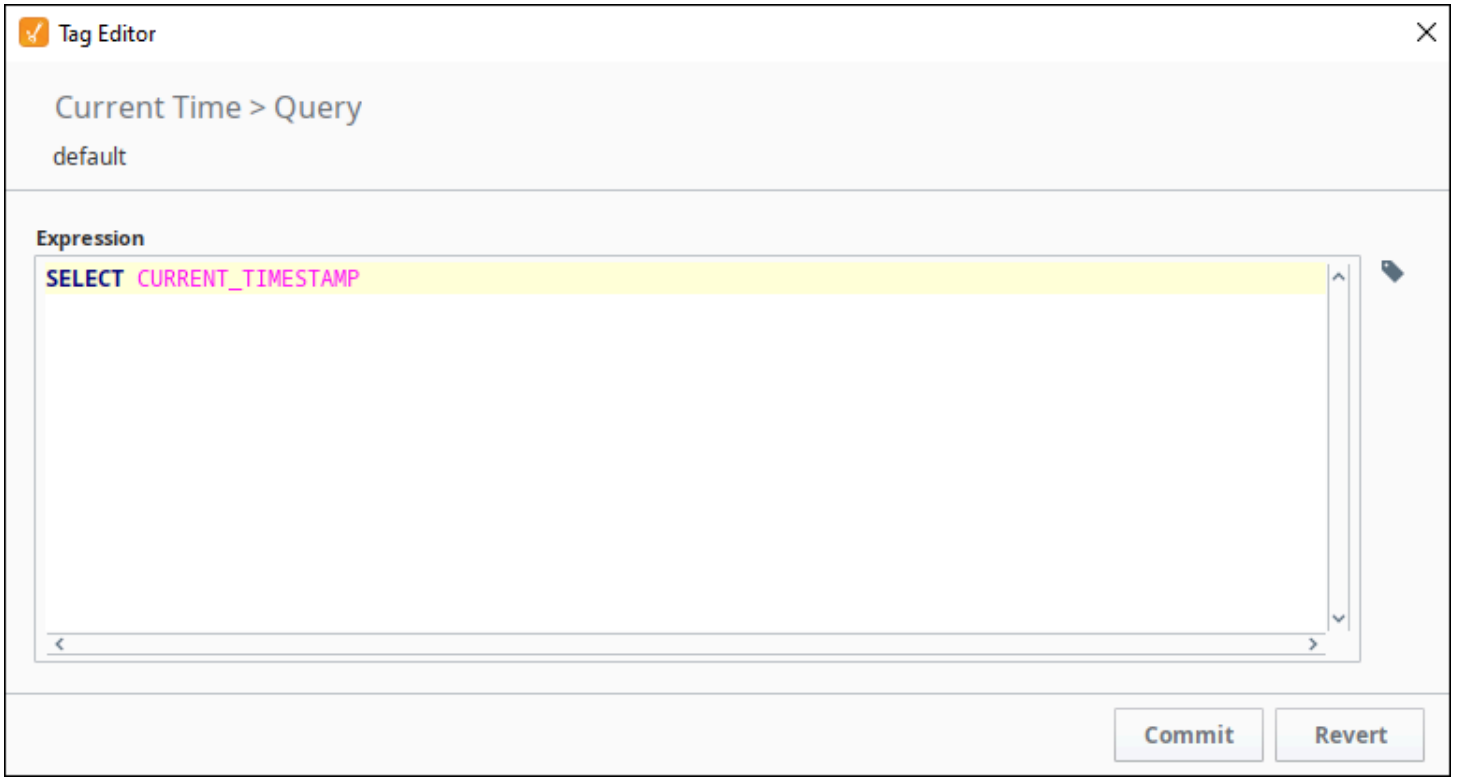
NOTE

It may be helpful when troubleshooting or testing Expression tags to increase the default thread pool count. Refer to the Gateway Configuration File Reference - Thread Pool Counts for more information.

Query Tags

A Query tag executes a SQL Query; the result of that query is returned to the value on the tag. Query tags can reference other Gateway-scoped tags to build dynamic queries. The **Query** property dictates the query that will execute, and the **Execution Mode** determines how often the query will run. Furthermore, the **Datasource** property determines which database connection the query will execute against.


Tag Editor
Current Time
default
Properties
Details
Documentation
Diagnostics
Basic Properties
Name
Current Time
Tag Group
Default
Enabled
true
Value
Value Source
Query
Data Type
DateTime
Execution Mode
Event Driven
Query
SELECT CURRENT_TIMESTA...
Datasource
MSSQL
Numeric Properties
Deadband
0.0001
Deadband Mode
Absolute
Scale Mode
Off
Engineering Units
Engineering Low Limit
0.0
OK
Apply
Cancel


Tag Editor
Current Time > Query
default
Expression
SELECT CURRENT_TIMESTAMP
Commit
Revert

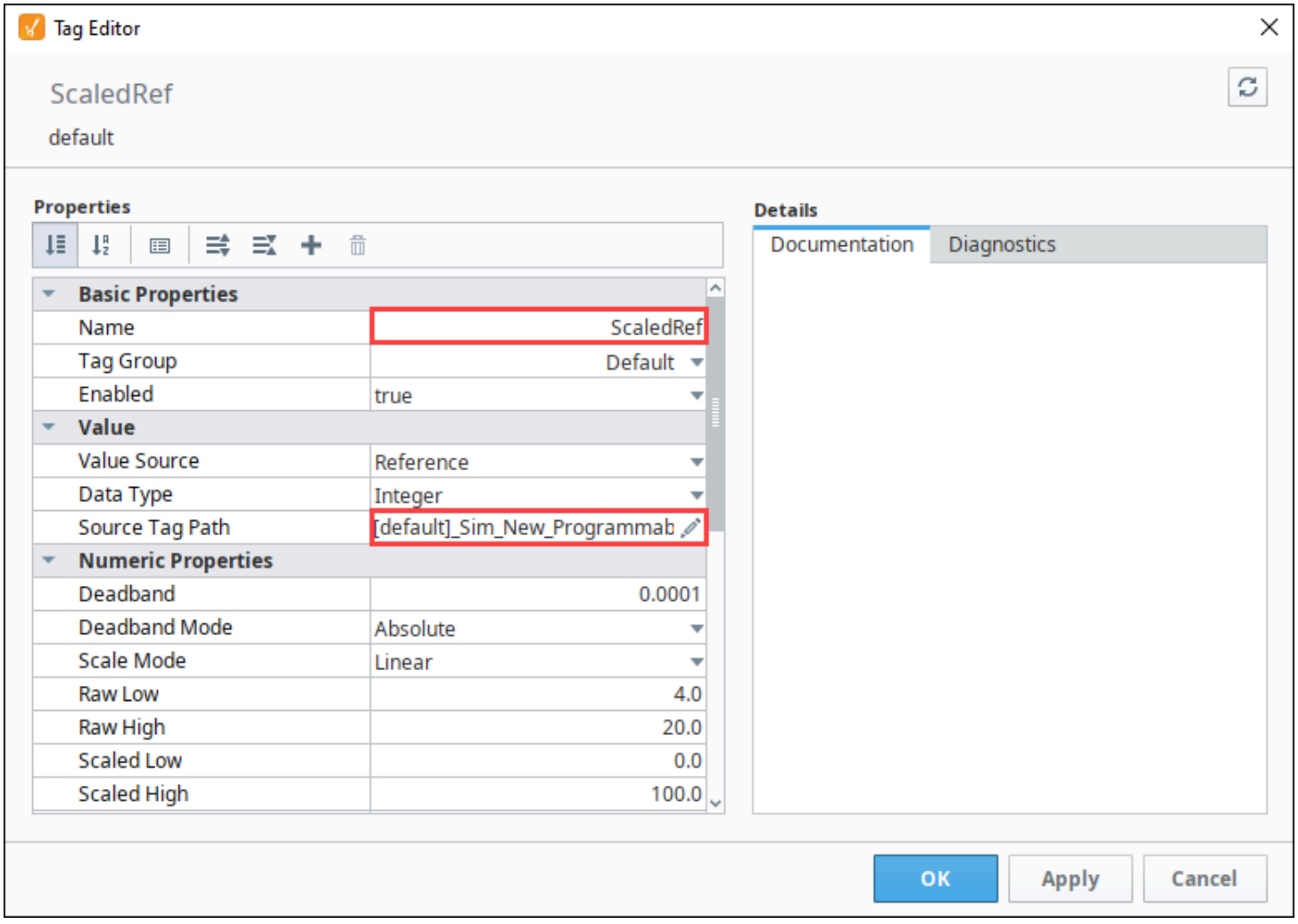
NOTE

It may be helpful when troubleshooting or testing Query tags to increase the default thread pool count. Refer to the Gateway Configuration File Reference - Thread Pool Counts for more information.

Reference and Derived Tags

Reference Tags

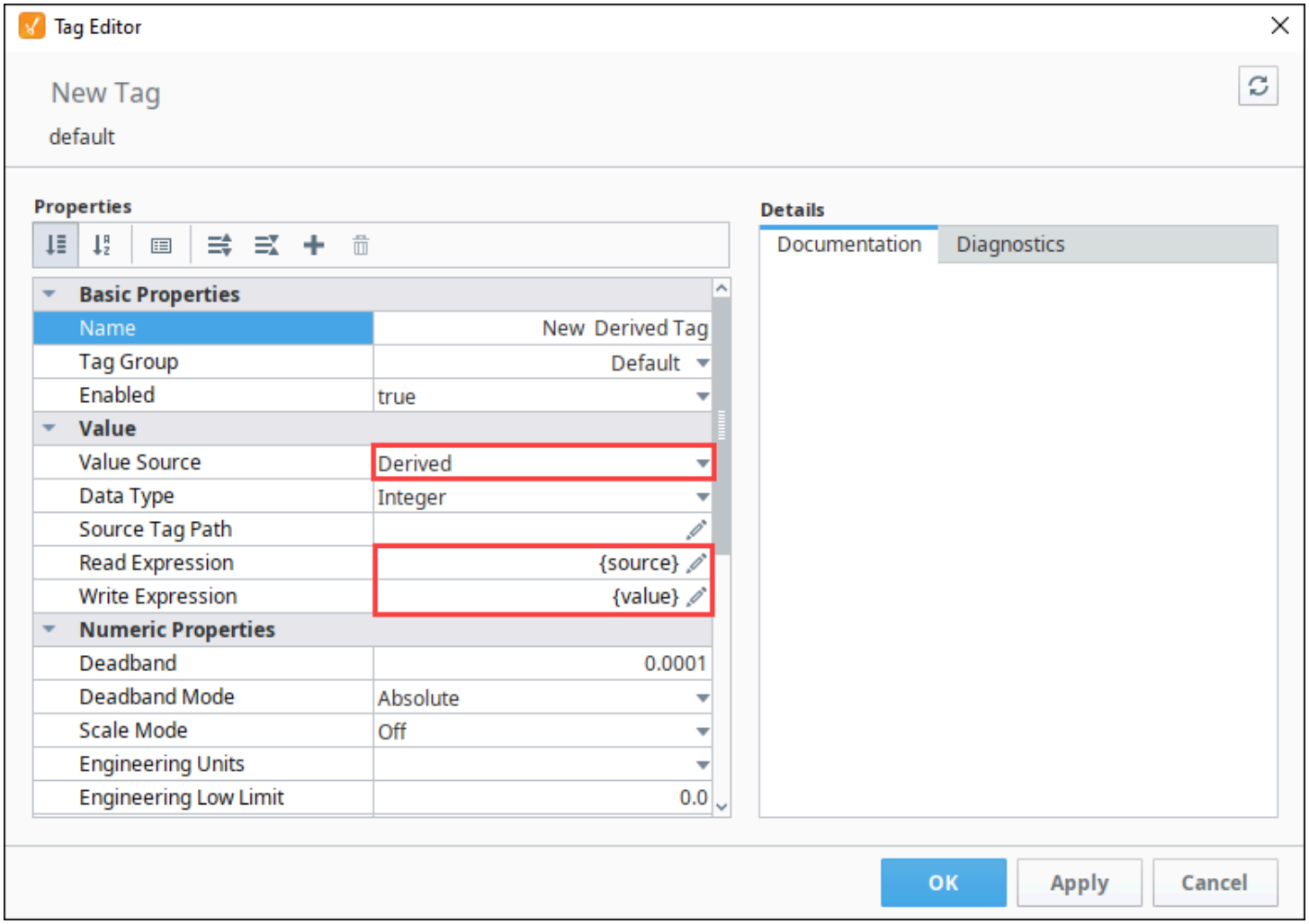
A Reference tag simply refers to an existing tag, using the **Source Tag Path** property to determine which other tag to reference. Writes targeting the Reference tag will send a write request to the source tag.



Derived Tags

A Derived tag is an abstracted tag that refers to another tag. They are similar conceptually to Reference tags in that that value is dependent on the **Source Tag Path** property, but Derived tags have some additional functionality. Namely, they can apply expressions to the referenced value with the **Read Expression** property, allowing the value on the Derived tag to differ from the source tag.

In addition, The **Write Expression** property will apply an expression to the value of any write requests targeting the Derived tag, allowing the expression to modify the value of the incoming write before it's applied back to the source tag.



Property	Description
Read Expression	Determines what value should appear on the Derived tag. The current value of the source tag may be referenced with the `{source}` reference.
Write Expression	When writing to the Derived tag, this expression determines what value should be written to the source tag. The current value of the source tag may be referenced with the `{source}` reference. The raw value passed to the Derived tag may be referenced with the `{value}` reference.

This interface provides an opportunity to scale the written and read value. For example, if the source tag was in Fahrenheit, and the derived tag should be Celsius, we could use the following expressions:

```
//Read Expression
({source}-32)*(5/9)

//Write Expression
{value}*(9/5) + 32
```

Changing the Source

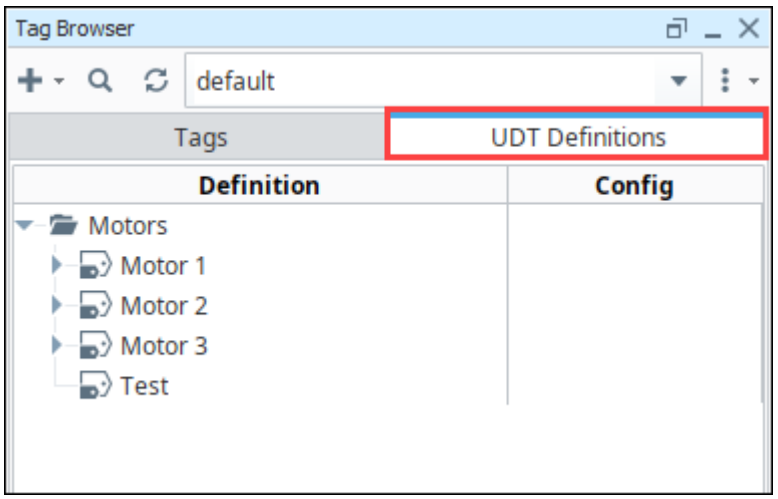
Aside from editing the tag from the Designer, the source tag path on a Derived tag may be modified by writing directly to the **SourceTagPath** property on the tag, or using a Tag Binding on a Vision component.

Additionally, the **SourceTagPath** property may be changed through scripting:

```
#Example of writing to the SourceTagPath attribute via Python Script
system.tag.writeBlocking(["Derived Example/Derived Tag.SourceTagPath"], ["
[.]Folder/New Source Tag"])
```

User Defined Types (UDTs)

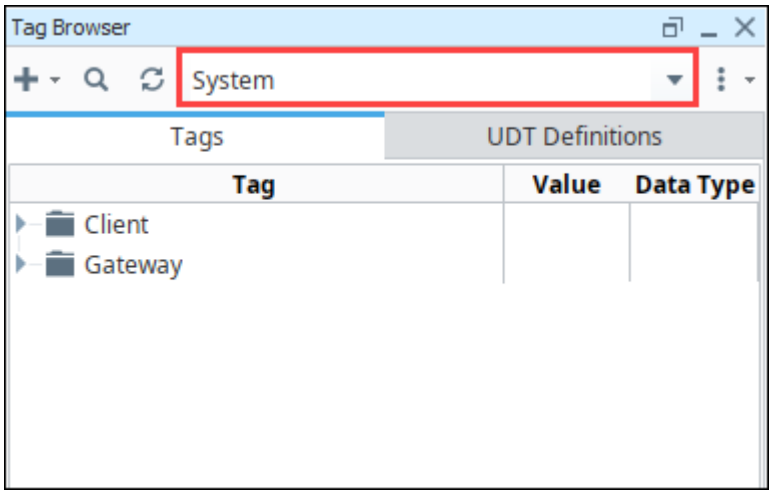
UDTs are created out of standard tag types, but they offer a variety of additional features. You can think of them as a way to create "data templates", where a particular structure of tags is defined, and can then be created as if it were a single tag. This UDT example shows two Motor instances, the data type Motor, and all the parameters and tags that make up the structure (i.e., Amps and HOA). For more information, see **User Defined Types - UDTs**.



System Tags

System tags provide status about the Ignition system. They're generally cannot be modified, but provide use information about how the system is performing.

More information can be found on the **System Tags** page.

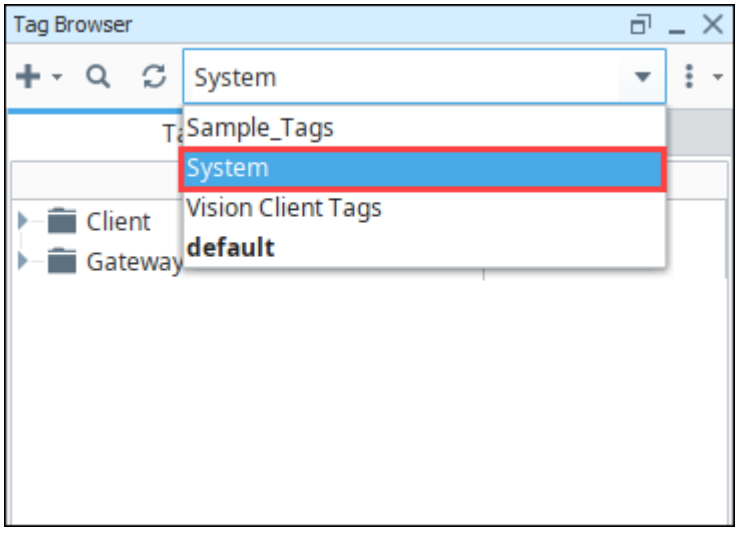


Vision Client Tags

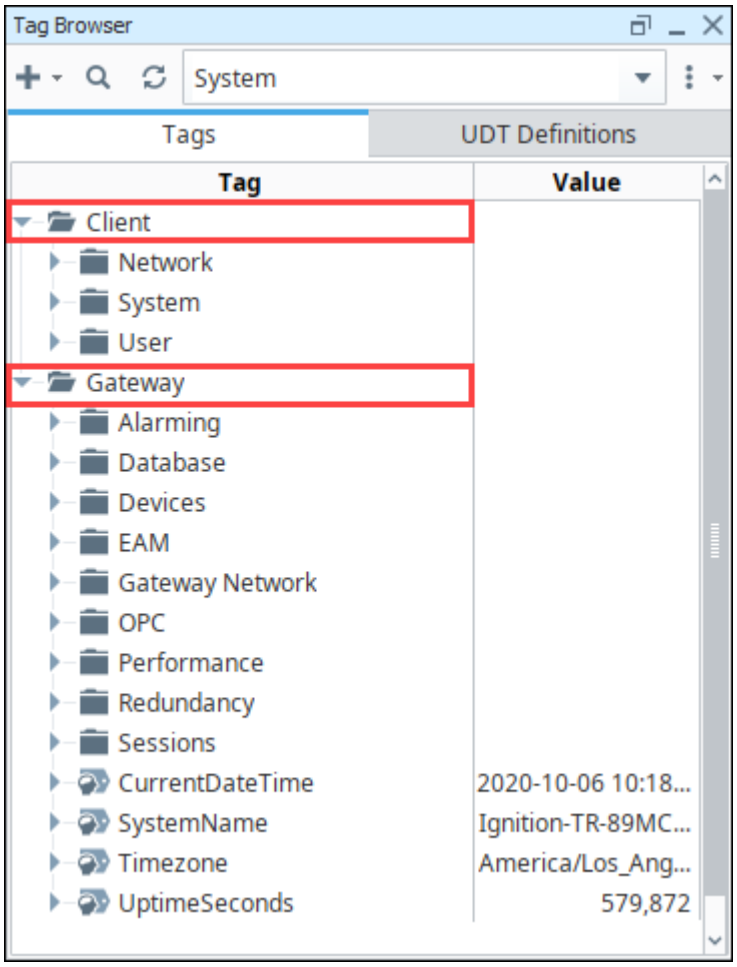
Within the Vision module, you can also have Vision Client tags that are specific to a Vision Client. Their values are isolated to a Client runtime. For more information, see **Vision Client Tags**.

System Tags

System tags provide status about the Ignition system, such as memory usage, performance metrics, and so on. System tags cannot be deleted or modified. To view the System tags in the Tag Browser, go to the Tag Provider Selector and select **System**.

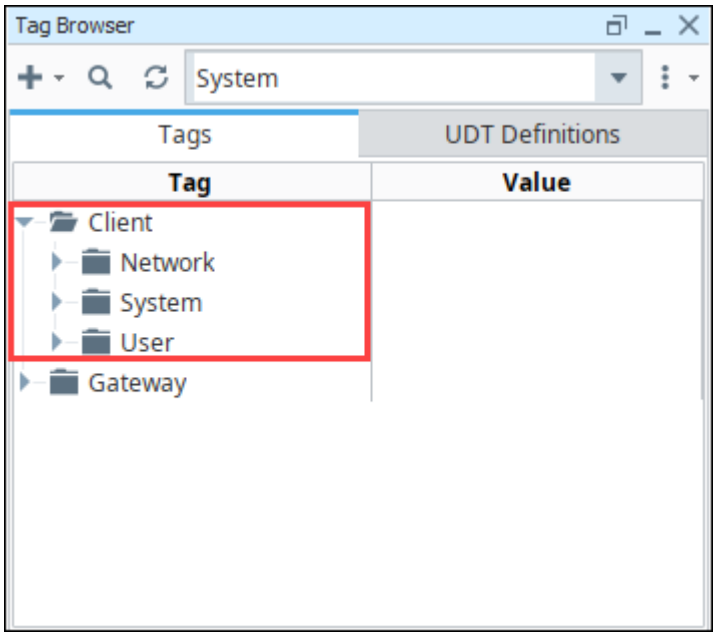


The System tags folders are displayed: **Client** and **Gateway**. The scope for each is slightly different.



System Client Tags (Vision Only)

Client-scoped System tags provide status information about the client's system. They can be used with the Vision module for any Vision Client. Every individual client is going to have their own values, such as IP address, host name, username, and more. There are three folders within the System > Client folder: **Network**, **System**, and **User**. You cannot modify Client System tags.



Vision System Client Tags

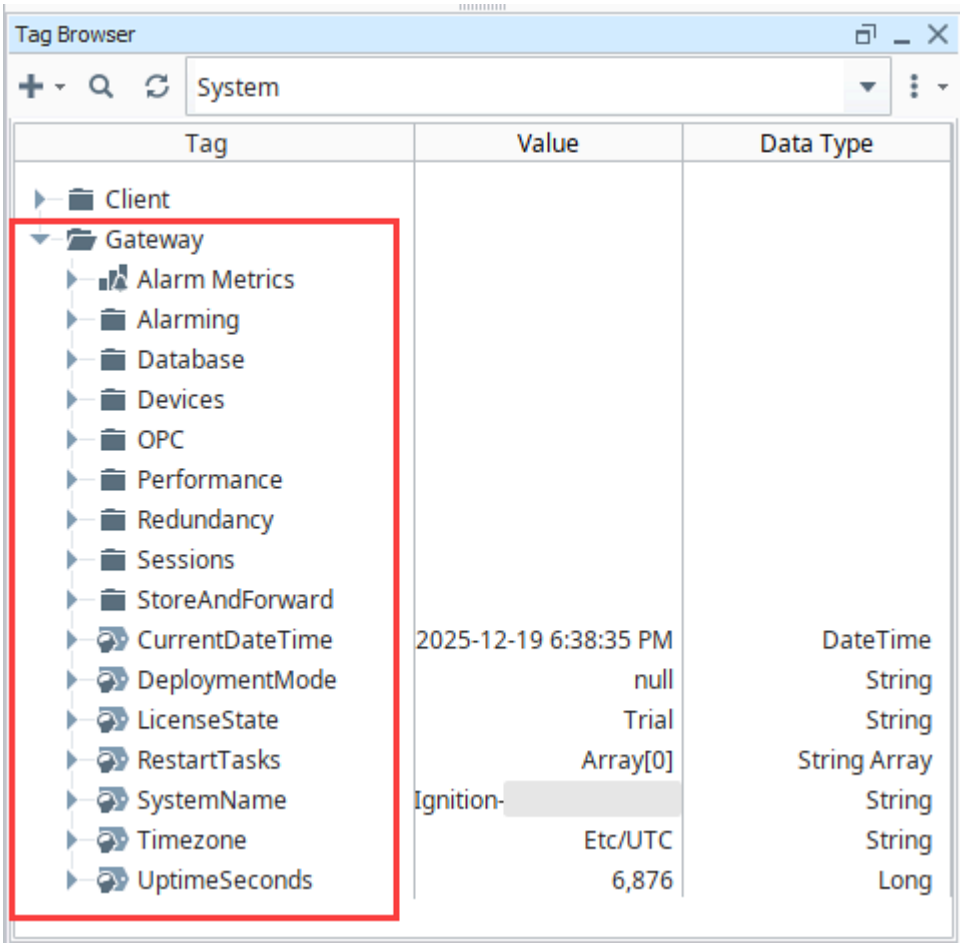
Name	Folder	Description	Data Type
GatewayAddress	Network	Gateway URL address.	string
GatewayRedundancyRole	Network	Redundancy State of the Gateway that the client is connected to. Independent, Master, Backup.	string
Hostname	Network	Hostname (name) of the computer that the Client is running on.	string
IPAddress	Network	IP Address of the computer that the Client is running on.	string
MACAddress	Network	MAC Address of the computer that the Client is running on.	string
CurrentDateTime	System	Current system date and time. Format is yyyy-MM-dd hh:mm:ss a.	DateTime
DefaultDatabase	System	Name of the default database connection used by the project.	string
DefaultTagProvider	System	Name of the default Tag Provider used by the project.	string
FPMIVersion	System	Current Ignition version in use.	string
JavaVersion	System	Current Java version in use by the client.	string
LastProjectUpdate	System	Date and time the local Vision Client's project was last updated.	DateTime
OperatingSystem	System	Operating system of the computer that the Client is running on.	string
ProjectName	System	Name field for the current project.	string

Name	Folder	Description	Data Type
ProjectTitle	System	Title field for the current project.	string
ProjectUpdateAvailable	System	Whether or not there is a project update available for the currently open Vision Client project. Intended to be used alongside the `None` option in **Vision's Project Properties**.	boolean
SystemFlags	System	A byte array of **flags** for the current state of the Client.	integer
UserSource	System	Name of the user source for the current Client.	string
Country	User	Two letter country code according to operating system. for example: US.	string
CurrentWindow	User	The current main window open in the project (top most Floating, Maximized window).	string
DateFormatFull	User	Full date format according to the operating system. Format: EEEE, MMMM d, y.	string
DateFormatLong	User	Long date format: MMMM d, y.	string
DateFormatMedium	User	Medium date format: MMM d, y.	string
DateFormatShort	User	Short date format: M/d/yy.	string
DateTimeFormatFull	User	Full date and time format: EEEE, MMMM d, y 'at' h:mm:ss a zzzz.	string
DateTimeFormatLong	User	Long date and time format: MMMM d, y 'at' h:mm:ss a.	string

Name	Folder	Description	Data Type
DateTimeFormatMedium	User	Medium date and time format: MMM d, y, h:mm:ss a zzzz.	string
DateTimeFormatShort	User	Short date and time format: M/d/y, h:mm a.	string
HomeFolder	User	Home folder according to OS. For example: C:\Users\psmith.	string
Language	User	Language according to OS. For example: "en" for English.	string
OSUsername	User	OS user name, for example: PSmith.	string
RolesDataSet	User	Dataset with Roles for currently logged in user. For example: Dataset[2R x 1C].	dataset
RolesString	User	Comma separated string with Roles for currently logged in user. For example: Administrator, Operator.	string
TimeFormatFull	User	Full time format according to the operating system. Format: h:mm:ss a zzzz.	string
TimeFormatLong	User	Long time format: h:mm:ss a z.	string
TimeFormatMedium	User	Medium time format: h:mm:ss a.	string
TimeFormatShort	User	Sort time format: h:mm a.	string
Timezone	User	Current timezone, for example, America/Los Angeles.	string
Username	User	Currently logged in username, for example, PSmith.	string

Gateway System Tags

Gateway System tags exist in the Gateway scope. There are several folders within the Gateway tags folder which report metrics on various Gateway level systems.



The following Gateway-scoped System tags are available.

Root System Tags

Name	Description	Data Type
CurrentDateTime	Current system date and time. Format is yyyy-mm-dd hh:mm:ss a.	DateTime
DeploymentMode	**New in 8.3.3** Current Gateway deployment mode. If there is no active mode, the value will be null.	String
LicenseState	Returns the license state of the Gateway. If a valid license is applied, the tag will return "Activated". This includes Emergency Activation mode and Maker licensing. For any other license state, the tag will return "Trial".	String
RestartTasks	Displays the number of pending tasks requiring a Gateway restart to complete.	String Array
SystemName	Returns the name of the system where the Ignition Gateway is installed.	String
Timezone	Timezone on the Gateway computer. For example, America/Los Angeles.	String
UptimeSeconds	Number of seconds since Ignition was started.	Long

Alarming System Tags

Name	Description	Data Type
Active and Acked	Number of alarms currently active and acknowledged.	Integer
Active and Unacked	Number of alarms currently active and unacknowledged.	Integer
Clear and Acked	Number of alarms cleared and acknowledged.	Integer
Clear and Unacked	Number of alarms cleared and unacknowledged.	Integer

Database System Tags

> **NOTE**
>
> There will be a subfolder for each database connection, or none if there are no connections. Each subfolder will have the following tags.

Name	Description	Data Type
ActiveConnections	Number of active connections in the pool to this database connection.	Integer
Available	Indicates whether this data source is available.	Boolean
AvailableThroughFailover	Indicates if any database along the failover chain attached to this data source can be reached.	Boolean
AvgQueryTime	Average time, in seconds, that it is taking database queries to run.	Integer
ConnectionSaturation	Percentage of possible query throughput that is being used (ratio of currently active connections to maximum possible connections).	Double
QueriesPerSecond	Number of queries running per second.	Integer

Device System Tags

> **NOTE**
>
> For each device configuration on the Gateway, a separate subfolder will exist. The contents of each subfolder are listed below.

Name	Description	Data Type
Description	Description of the device connection as configured on the device connection on the Gateway.	String
Enabled	Boolean representing whether or not the device connection is enabled.	Boolean
Name	Name of the device connection.	String
Status	Status of the connection.	String

EAM System Tags

> **NOTE**
>
> System tags are created on the Controller for each agent, allowing you to build EAM monitoring clients, set alarms on individual tags, and many other tag-related operations. A folder is created for each agent. As with other System tag values, all EAM system tag values are read-only.

Metrics Folder

The metric folder contains multiple subfolders.

Database Folder Each database connection on the agent will be represented by a different folder.

Name	Description	Data Type
Active Connections	Number of active connections in the pool to this database connection.	Integer
Active Queries	Number of active queries.	Integer
Active Query Duration	Average time, in seconds, that it is taking database queries to run.	Integer
Average Throughput Per Second	Number of records forwarded to the data sink per second. Throughput will be -1 if idle.	Integer

Logging Folder

Name	Description	Data Type
Errors Per Hour	Number of errors logged per hour.	Integer
Errors Per Minute	Number of errors logged per minute.	Integer

Session Folder

Name	Description	Data Type
Clients	Number of currently running Vision clients.	Long
Designers	Number of currently running Designers.	Long
Perspective Sessions	Number of currently running Perspective sessions.	Long

System Folder

Name	Description	Data Type
CPU Usage	CPU Utilization as reported to the Java Virtual Machine.	Float
Memory Usage	Amount of RAM currently in use by the Gateway, in whole percentages from 0-100.	Float

Module Versions Folder

The ModuleVersions folder contains tags for each module installed on the agent where the tag name reflects the module name and the tag value displays the module version number.

Root Tags

The following tags are located in the root of the EAM folder

Name	Description	Data Type
AgentGroup	Name of the group that the Agent belongs to.	String
AgentName	Name of the Agent.	String
BrowserURL	Agent Gateway URL address.	String
IsApproved	Whether this network connection has been approved.	Boolean
IsConnected	Whether the Gateway Network connection is active.	Boolean
IsRunning	Whether the Agent Gateway is considered "healthy". Healthy Agent Gateways may not be connected to the Controller, but not enough time has passed to consider the Agent Gateway as "down".	Boolean
LastCommunication	A timestamp representing the last known communication received from the Gateway.	DateTime
LicenseKey	Returns the license key of the Gateway.	String
NodeRole	Role of the connected Agent. • **Independent** - Redundancy is not enabled and this Ignition system runs as an independent node. • **Master** - This is the Master node, who listens for a connection from the Backup node, and is in charge of managing system synchronization. • **Backup** - This is the Backup node, who will connect to the Master and receive system updates.	String
PlatformEdition	Ignition Edition running on the Agent Gateway. Values include `standard`, `edge`, and `maker`.	String
RunningState	String representation of the Running State. Possible values for RunningState are: `Disconnected`, `Running`, `Warned`, `Errored`	String

Name	Description	Data Type
RunningStateInt	Integer representation of the Running State.	Integer
ServerId	ID of the Agent Gateway.	String
Version	Version of Ignition software running on the Agent Gateway.	String

Gateway Network System Tags

Each system connected over the Gateway Network receives a special folder designated by the following nomenclature:

- Independent Gateways: 0:0
- Redundant Master Gateway: 0:1
- Redundant Backup Gateway: 0:2
- "Virtual" route aliased to either the Master or Backup Gateways: 0:0

Each folder contains the following tags.

Name	Description	Data Type
IsAvailable	Whether the remote Gateway is available or not.	Boolean
LastComm	A timestamp representing the last known communication received from the remote Gateway.	DateTime

Licenses System Tags

This tag folder will only appear if your Gateway is using a Leased License. For each leased license on the Gateway, a separate subfolder will exist. The contents of each subfolder are listed below.

Name	Description	Data Type
ConsecutiveErrorCount	Rolling counter for how many times the license tried to renew itself, if failed.	Long
ErrorCode	Displays an error code if the license failed to renew.	Integer
Expiration	Time when the session expires.	DateTime
ExpiresInSeconds	Amount of time left before the current lease period expires, in seconds.	Long
NextCheck	Time when the license will attempt to renew.	DateTime
Problem	Description of why the license failed to renew or activate.	String
Status	Current status of the leased license.	String

OPC System Tags

> **NOTE**
>
> There will be a subfolder for each OPC UA Server. Each subfolder will have the following tags.

Name	Description	Data Type
Connected	Whether the OPC UA server is connected to Ignition.	Boolean
Enabled	Whether the OPC UA server connection is enabled.	Boolean
State	The state name of the connection. For example: Connected, Faulted, Connecting.	String

Performance System Tags

Name	Description	Data Type
Available Disk Space (MB)	Available disk space on the computer Ignition is installed on, in megabytes.	Long
CPU Usage	CPU Utilization as reported to the Java Virtual Machine.	Double
Disk Utilization	Percentage of hard disk that is in use.	Double
Max Memory	Maximum amount of RAM the Gateway can use, in bytes.	Long
Memory Usage	Amount of RAM currently in use by the Gateway, in bytes.	Long
Memory Utilization	Current memory usage/maximum memory usage.	Double

Redundancy System Tags

Name	Description	Data Type
Connection, Is Connected	Whether this Gateway is connected to another for redundancy.	Boolean
Connection, PeerId	The ID of the Gateway connected to, empty string if not connected.	String
ActivityLevel	Indicates where the Gateway is in the redundant state. Can be undecided, cold, warm, or active.	String
IsActive	Whether the Gateway is running.	Boolean
IsMaster	Whether the Gateway is the master. False if the backup is in control.	Boolean
Role	Named role of the Gateway. Options: Independent, Master, Backup.	String

Sessions System Tags

Name	Description	Data Type
SessionCount	Number of active sessions on this Gateway. Note this value does not account for Perspective Client Sessions.	Integer

Store and Forward System Tags

Each database connection will have their own set of System Tags.

Name	Description	Data Type
TotalDroppedCount	The total number of dropped datapoints within the Store and Forward engine.	Integer
TotalPendingCount	The current total number of pending datapoints within the memory buffer and disk cache waiting to be pushed through the Store and Forward engine.	Integer
TotalQuarantinedCount	The current total number of datapoints that are in quarantine.	Integer

In addition to the tags above, each database connection contains multiple subfolders. The subfolders are listed as tables below.

ForwardMetric

Name	Description	Data Type
FifteenMinuteRate	The rate at which datapoints are forwarded to the database or remote Gateway at 15 minute intervals.	Integer
FiveMinuteRate	The rate at which datapoints are forwarded to the database or remote Gateway at five minute intervals.	Integer
OneMinuteRate	The rate at which datapoints are forwarded to the database or remote Gateway at one minute intervals.	Integer

StoreMetric

Name	Description	Data Type
FifteenMinuteRate	The rate at which datapoints are stored to the Store and Forward engine at 15 minute intervals.	Integer
FiveMinuteRate	The rate at which datapoints are stored to the Store and Forward engine at five minute intervals.	Integer
OneMinuteRate	The rate at which datapoints are stored to the Store and Forward engine at one minute intervals.	Integer

Creating Tags

Tags are created in the Designer in one of two ways; either using the Connected Devices window or creating tags manually in the Tag Browser.

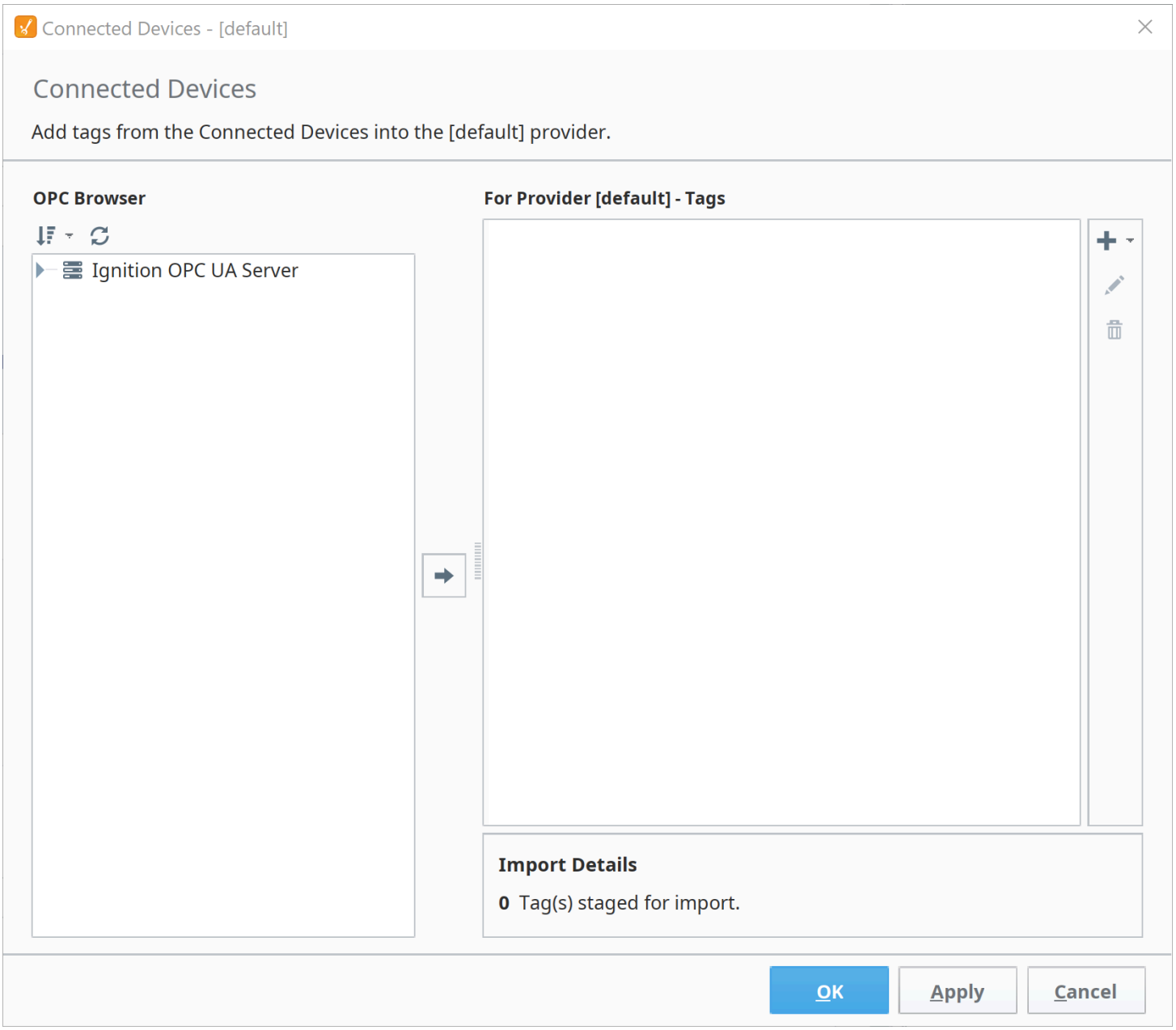
The Connected Devices window allows you to browse your connected devices and drag OPC tags into your Tag Provider, as well as create folders and other tags manually. You can also make immediate changes to the tag properties in the Connected Devices window.

Or you can manually create tags in the Tag Browser. The Tag Browser allows you to choose from a list of these standard tag types and change the tag properties directly in the Tag Editor.

This page describes both methods for creating tags.

Connected Devices Window

The Connected Devices window lets you browse all your connected devices and OPC Servers for datapoints. When the tag you're interested in is not available for browsing, you can manually add a tag and change any tag properties directly in the Connected Devices interface. This approach is common for devices like Modbus and Siemens where memory addresses aren't browsable. This interface allows you to create any standard type of tag. Note that the Connected Devices window is not available when the **System** or **Vision Client Tags** providers are selected.



Tags that are grayed out in the **For Provider** area are tags that already exist in your Tag Browser. The Tag Provider represented here is dependent on the Provider selected by the Tag Browser before the Connected Devices window was opened.

New tags added by the Connected Devices window will appear in black font. These new tags are effectively in a "staging" area, meaning they won't be added to the Tag Provider until you press **OK** or **Apply**. In the Connected Devices window, you can also select the Preview icon to see a summary of new tags you are adding.

You have the option of editing your new tags in the Connected Devices interface by selecting the tag you want to edit in the **For Provider** area and double clicking or clicking on the edit (pencil) icon on the right. The Connected Devices edit window will open. Here you can edit any properties of the tag. When you're finished, click **Commit** to accept your changes. When you are done adding tags, click the **OK** or **Apply** buttons in the lower right corner to add your tags to the Tag Browser.

Creating Tags

INDUCTIVE UNIVERSITY

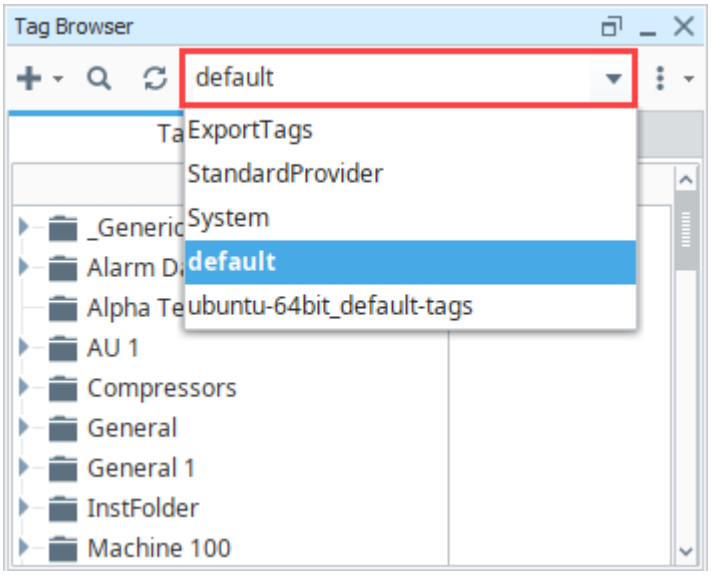
Creating OPC Tags

Watch the video

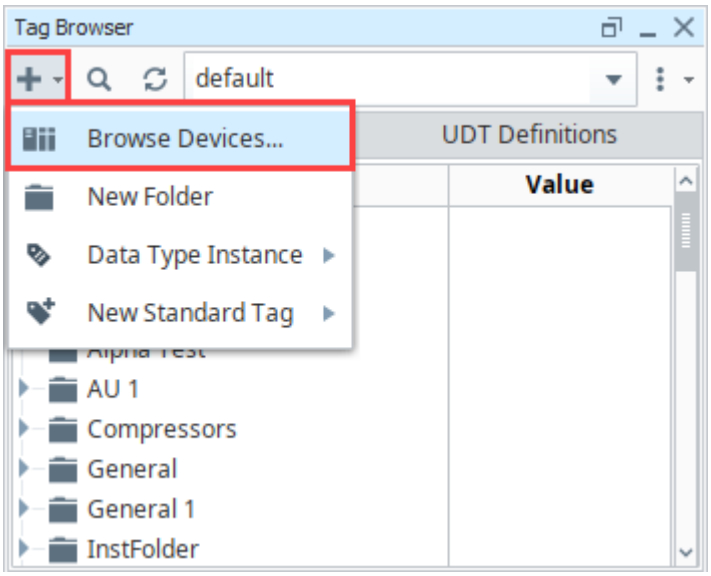
The example below describes creating tags using the Connected Devices browser.

> **NOTE**
>
> In order to have any items appear under the OPC Browser tree, you must have a **device connection**.

1. In the Tag Browser, choose a Tag Provider from the dropdown list where you want to place your new tags. The project's default provider will appear in bold.



2. In the Tag Browser, click the Add ✚ icon to open the context menu. Here you can Browse Devices to add OPC tags, or simply create a tag or folder. Select Browse Devices...

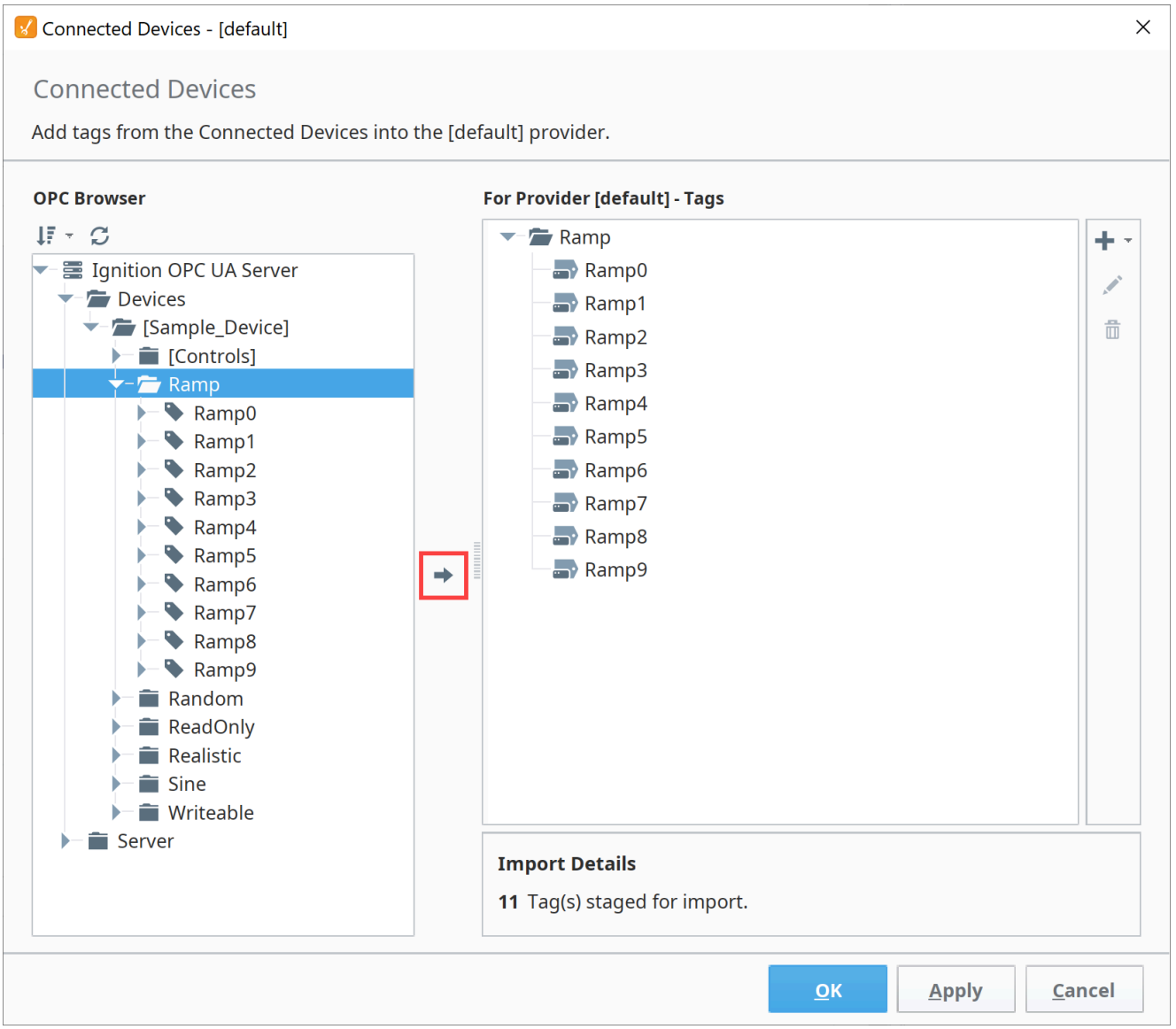


3. The Connected Devices window will open where you can either add an OPC tag or create a new standard tag. To add an OPC tag:

    i. Browse the OPC tags area to find tags that you're interested in.

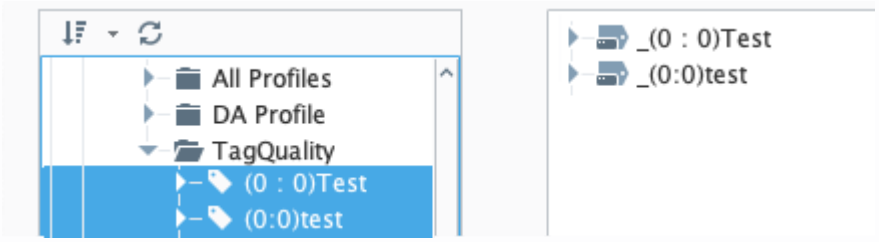
    ii. Select the folder(s) or tag(s) you want to add. To unselect a row, click out on a blank area, thus clearing any selected row.

    iii. Move the selected OPC tags into your selected For provider area by clicking the Right Arrow icon or dragging them into the right panel.



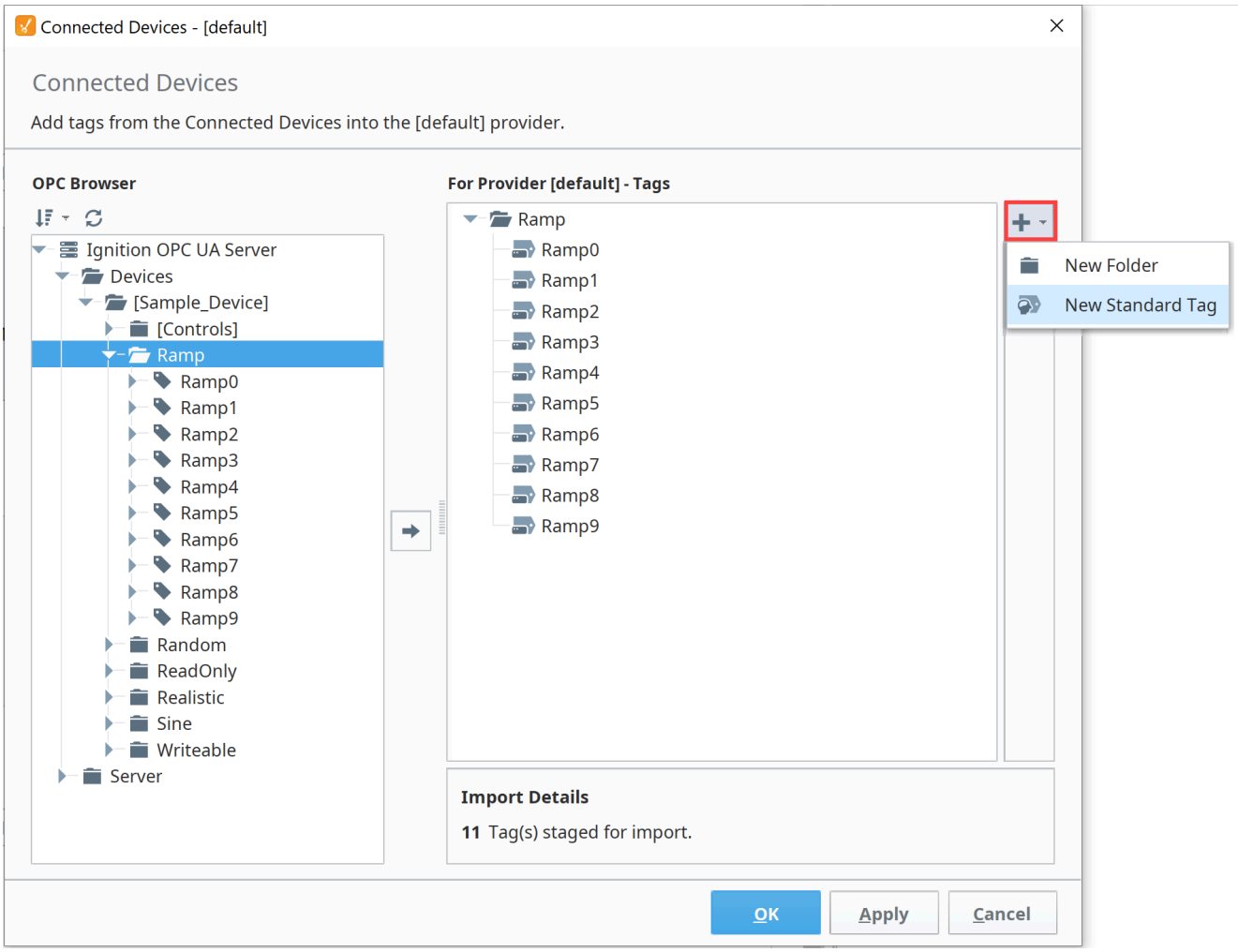
NOTE

The following characters are invalid and will be replaced by an underscore when dragging and dropping an OPC item into the Tag Browser: ! @ # $ ^ & * + [ ] Parentheses are valid characters but are not valid as a first character in a tag name. If an OPC item name starts with a parenthesis, the tag name will be prefixed with an underscore:



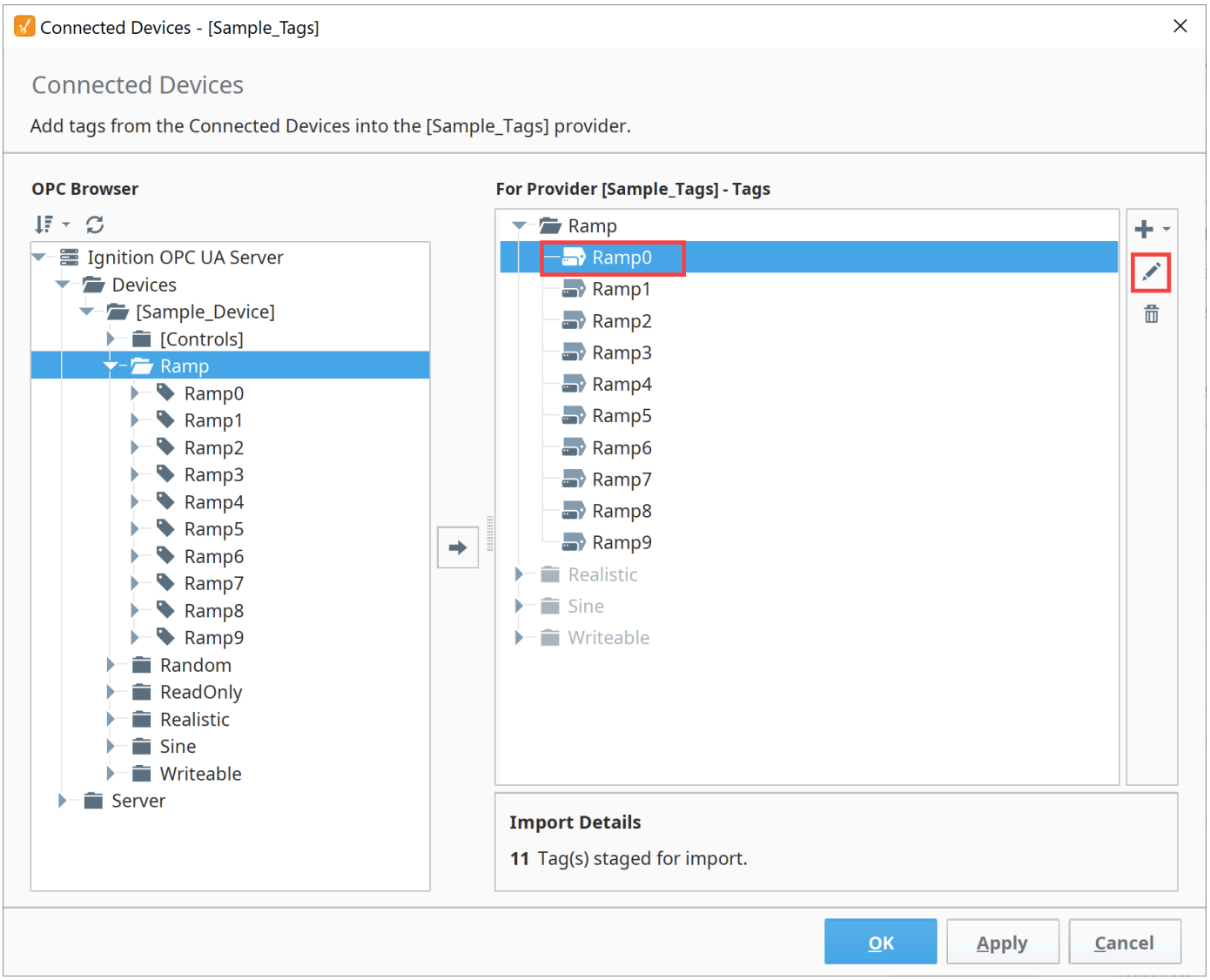
iv. To create a New Standard tag, click the Add ✚ icon.

v. Click New Standard Tag. A new blank tag will appear in the For provider area in the currently selected folder (if one is selected), or at the bottom of the tag list.

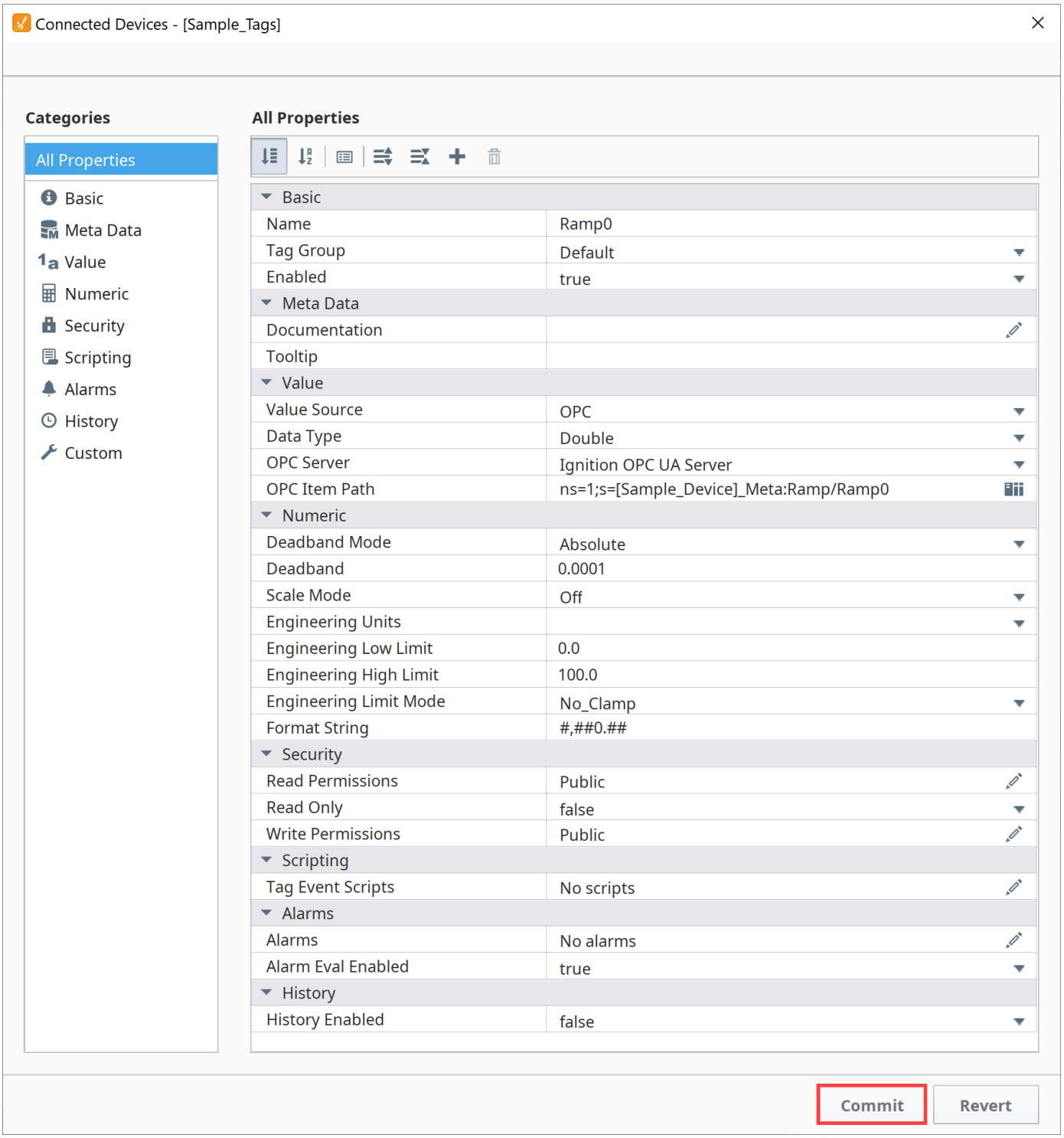


4. You have the choice of editing your tags in the Connected Devices window or editing them in your Tag Browser. To edit tags in the Connected Devices window, select a tag and click on the Edit icon. Here you can change the Data Type, Item Path, set alarms or store history, and anything

else that you can do to modify a tag.

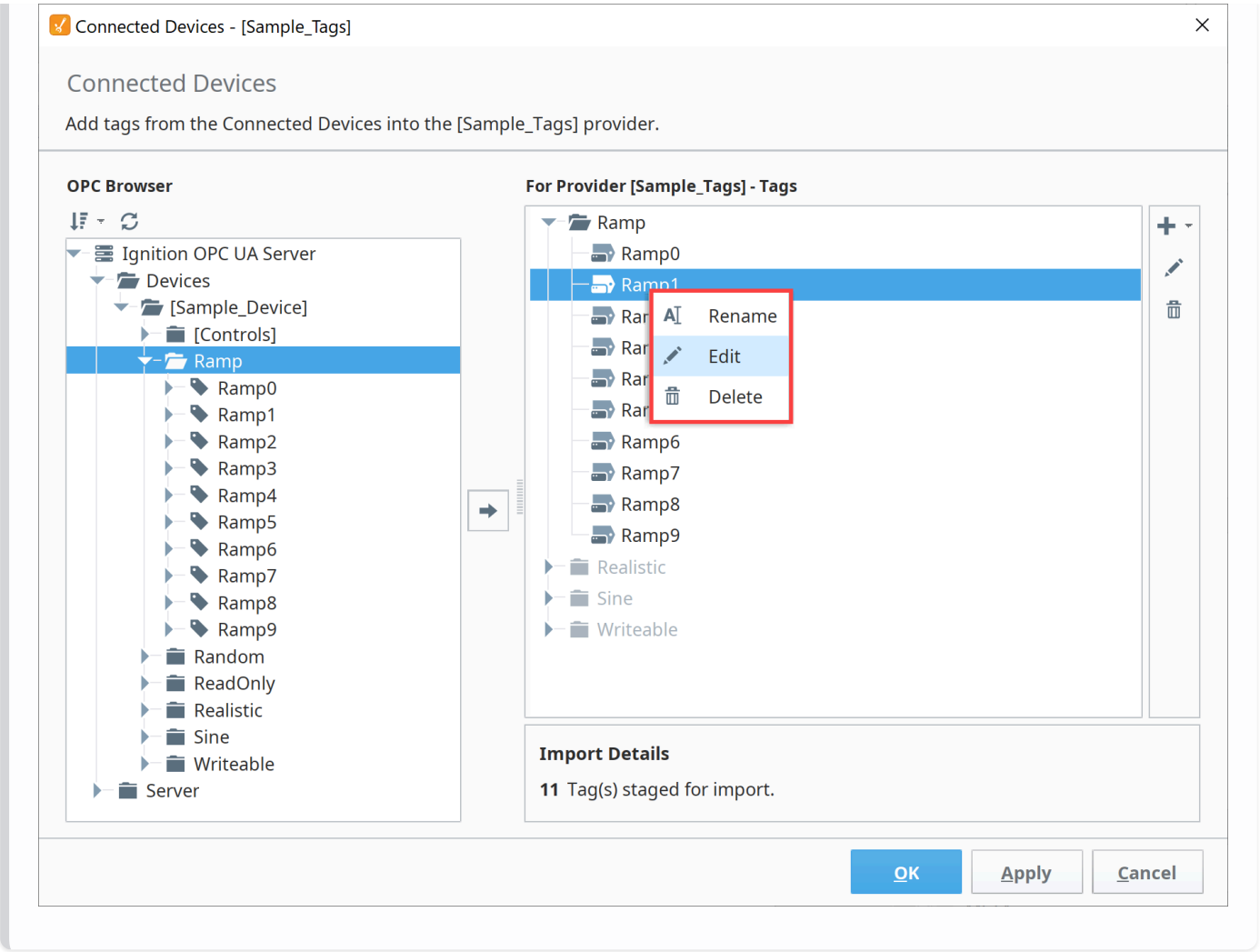


5. Click **Commit** to submit your changes.

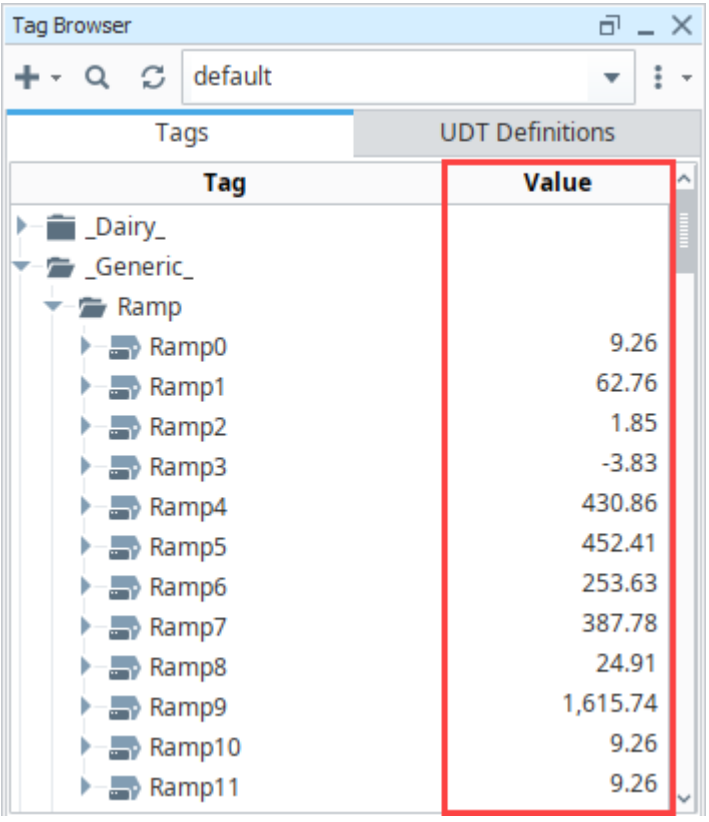


NOTE

The For Provider area features a right-click menu that also has Edit, Rename, and Delete options.



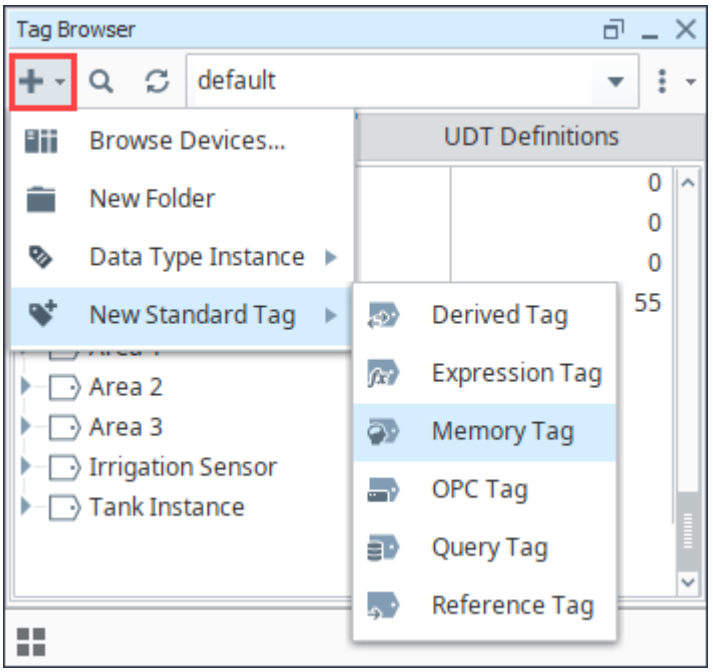
6. When finished, click the Apply button to add your new tags to the Tag Browser. As soon as the tags are in the Tag Browser, the values will start updating automatically.



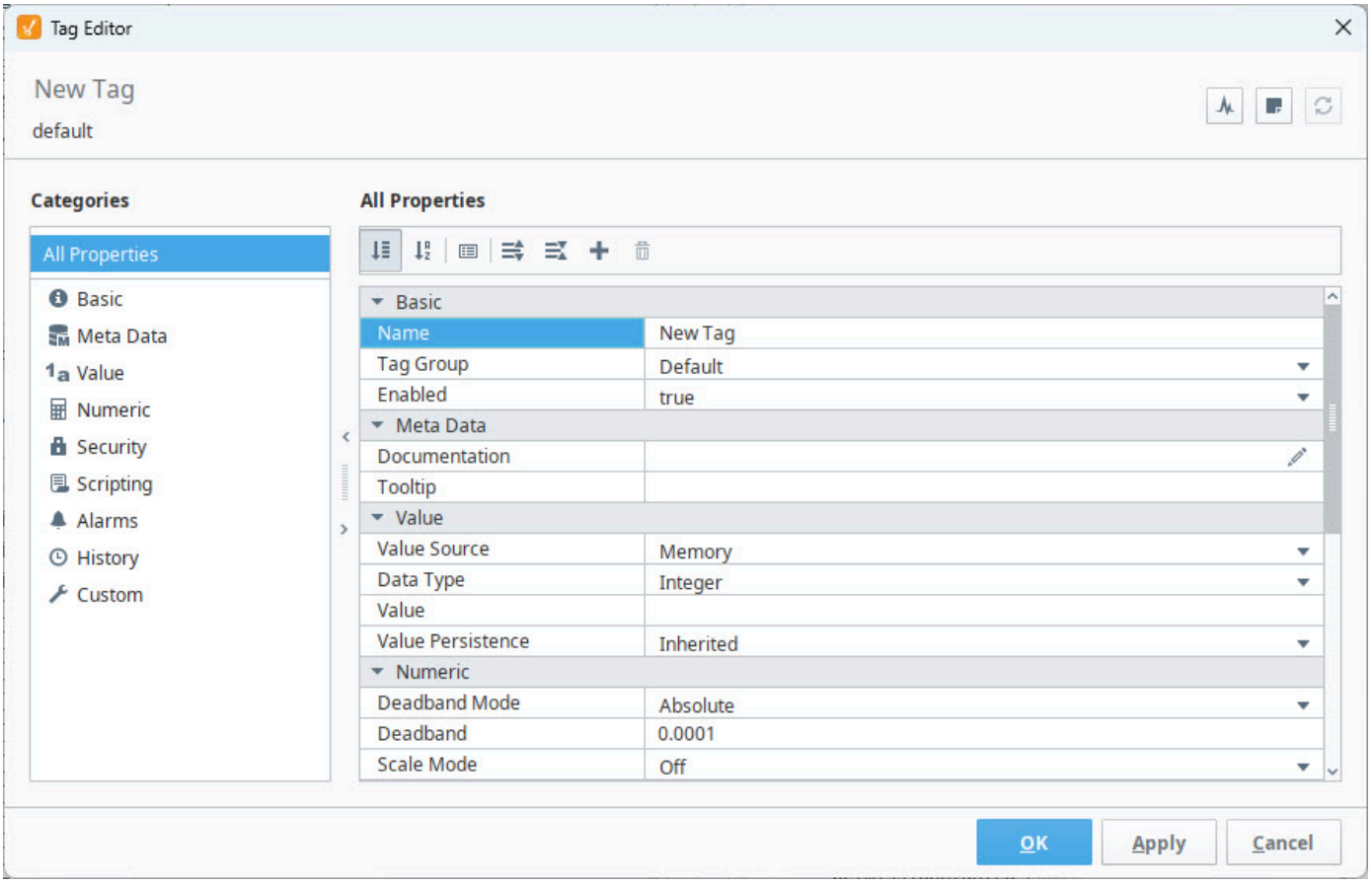
Creating Tags in the Tag Browser

Tags can be created manually in the Tag Browser. You can create any of the Standard Tag Types or a Data Type Instance. This example creates a Memory tag.

1. You can create tags in two different ways: click on the Add ✚ icon to open the context menu and select New Standard Tag and choose the tag type, or right click any where in the Tag Browser and select New Tag and choose the tag type.

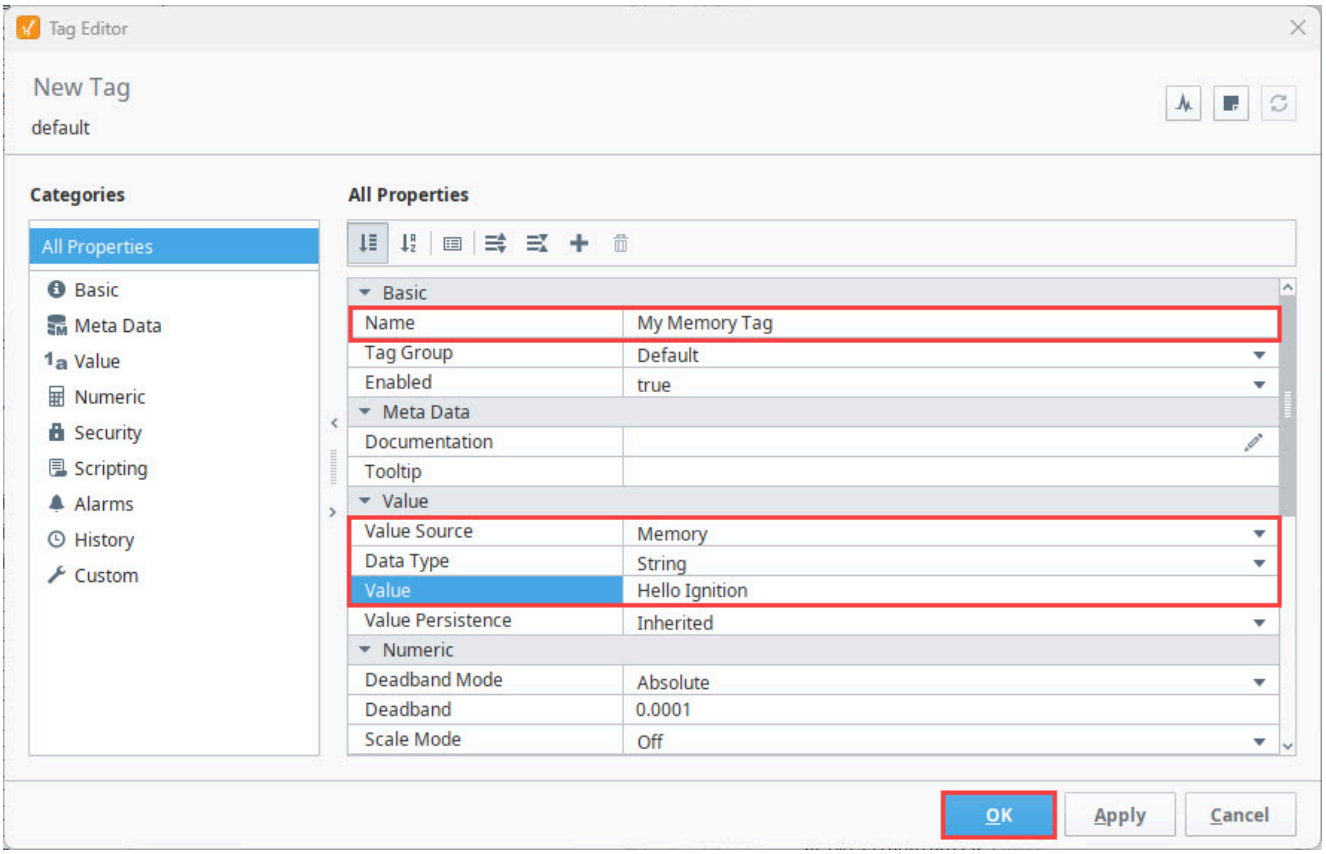


2. The Tag Editor will open to edit the tag's properties.



3. Set the properties as follows:

- Name: **My Memory Tag**
- Value Source: **Memory**
- Data Type: **String**
- Value: **Hello Ignition**



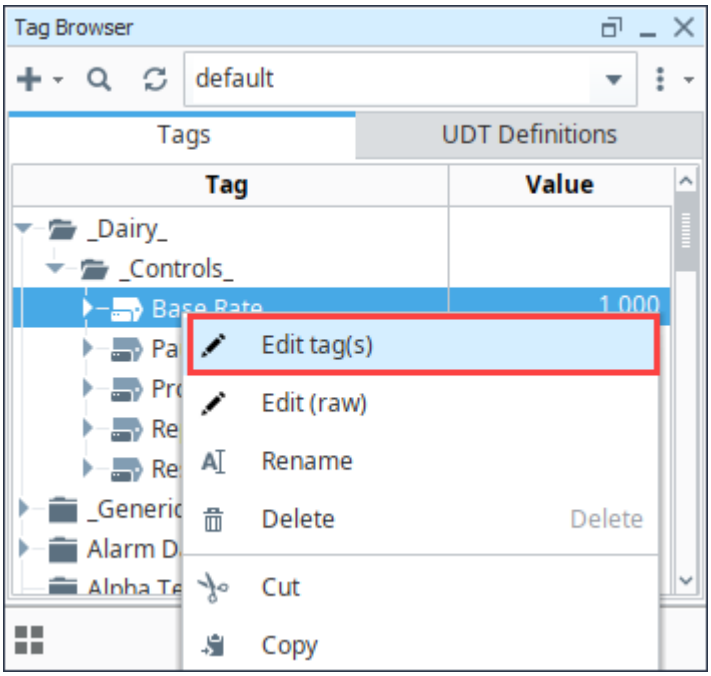
4. When finished, click **OK**. Your new tag will be placed in the tag column of the Tag Browser. Since this is a Memory tag, its value will not change unless you write to it.

Editing Tags

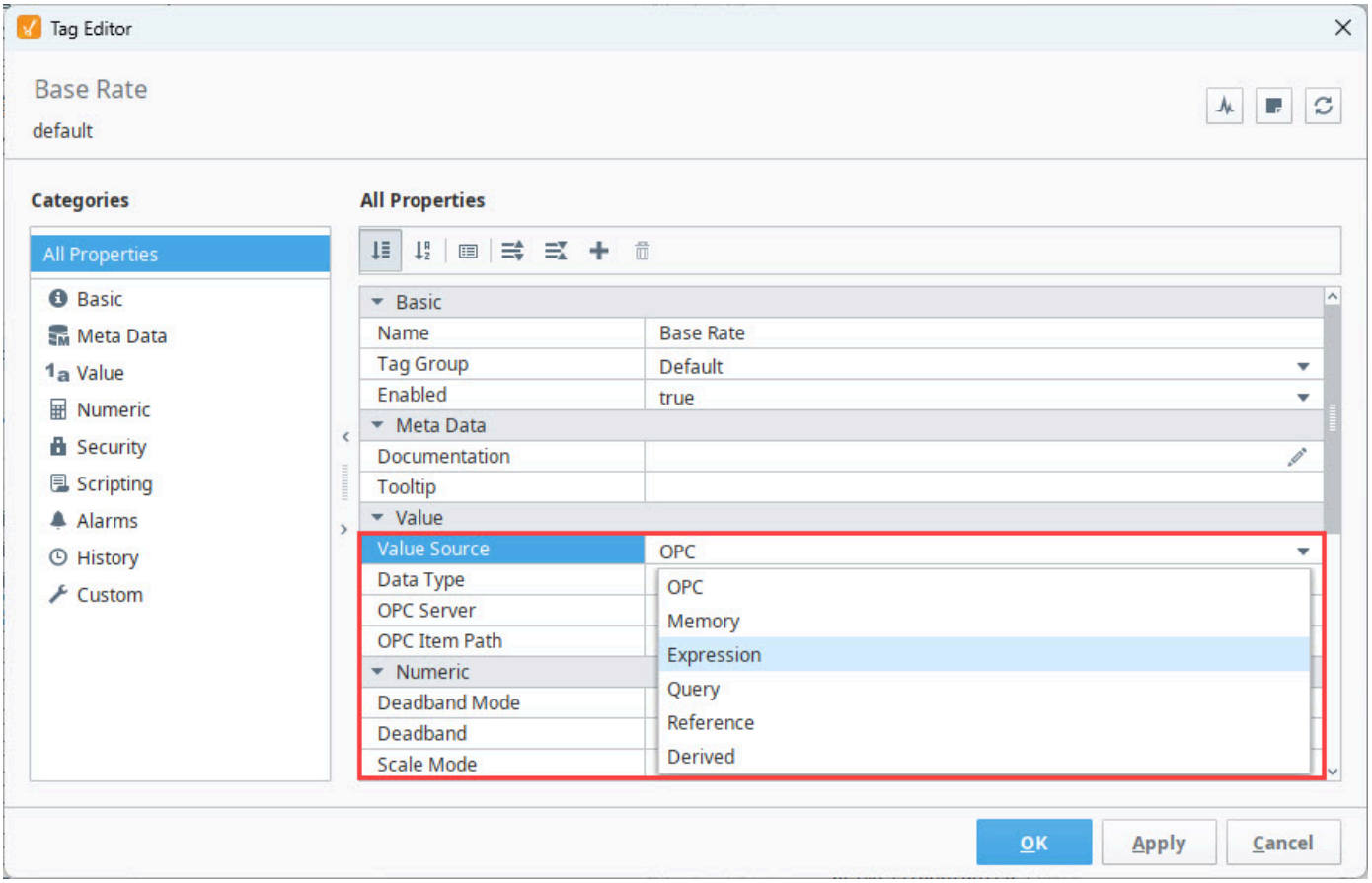
The Tag Editor is a powerful tool used when creating tags and for editing them. The properties displayed in the Tag Editor are custom to the type of tag you've selected. The Tag Editor is accessed using the tag right-click menu. Beyond the examples below, you can find additional information on the tag right-click menu here.

Edit a Tag

1. To edit an existing Tag, right-click on the Tag, and select the **Edit Tag** icon.



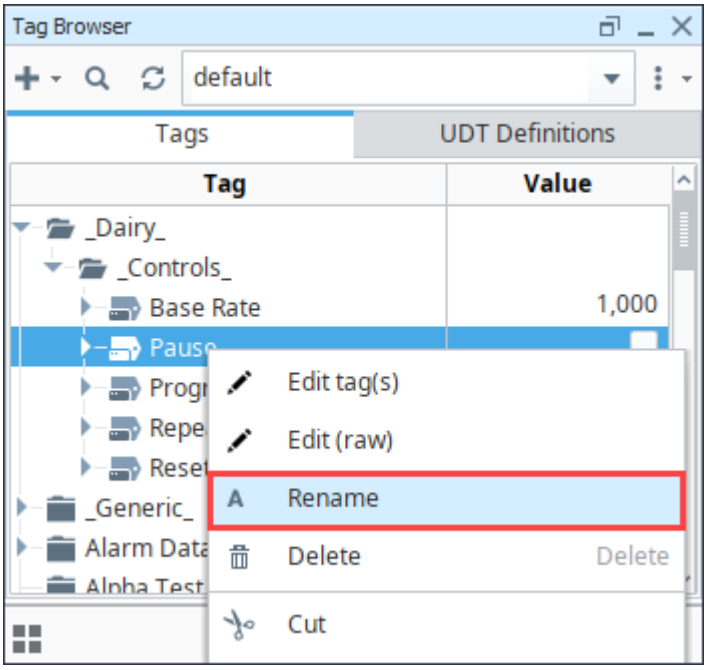
2. Once in the Tag Editor, you can update the Tag properties. For example, if you want to change the Tag to a different type – such as from OPC to Expression – go to the Value Source property, click the **Expand** icon, and choose the type of Tag (OPC, Memory, Expression, Query, Reference, or Derived) that you want.



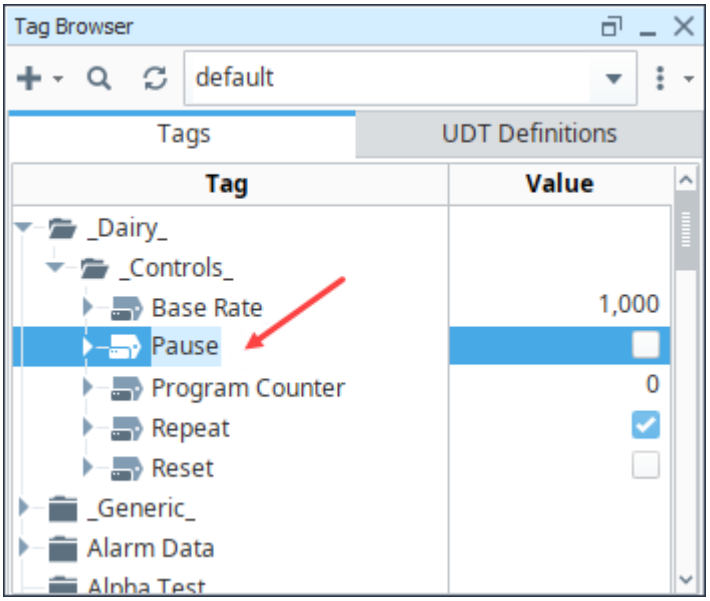
Rename a Tag

Tags names are flexible. For naming conventions, see **Tag Browser**.

1. To rename an existing Tag, right click on the Tag in the Tag Browser and select the **Rename** option.



2. The cursor will now blink inside the Tag name and you can type the new name.

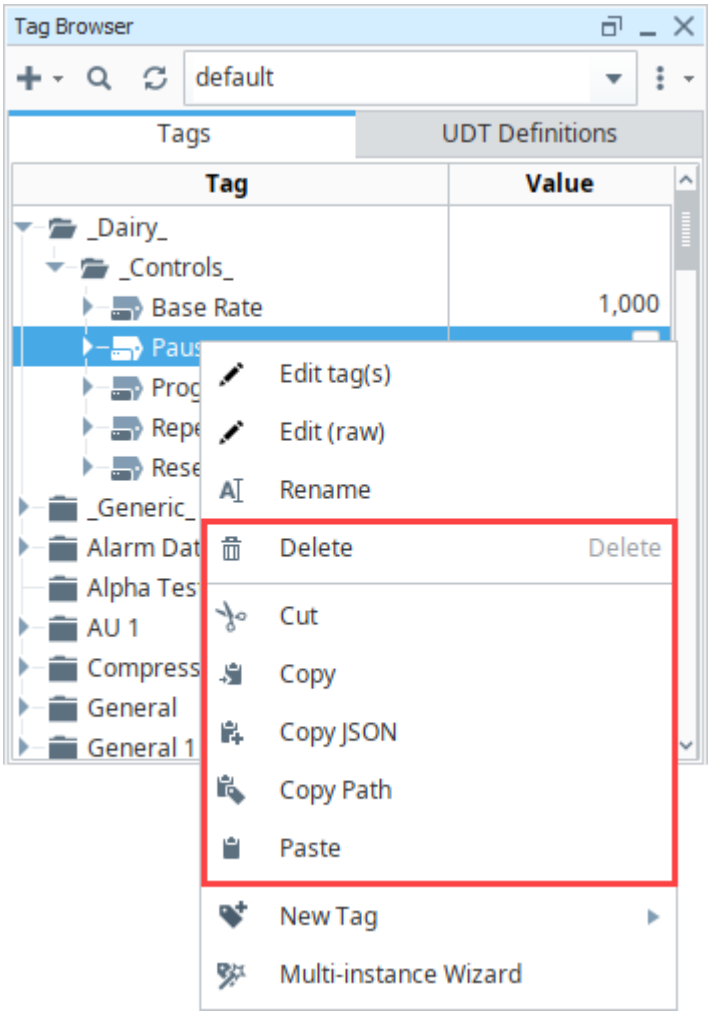


Cut, Paste, or Copy a Tag

You can also cut, paste, and copy Tags within the Tag Browser. Right click on the Tag in the Tag Browser. Choose the command you want.

- **Delete**: Completely removes the Tag.
- **Cut**: Delete the Tag from the current location, but leave it in the clipboard to be pasted elsewhere in the browser.

- **Copy**: Make a copy of the Tag and leaves it in the clipboard to be pasted elsewhere in the browser.
- **Paste**: Pastes the Tag you've cut or copied into the currently selected location in the Tag Browser.



Addressing Bits

In bindings and scripting there are ways to read a single bit of a word, but you can also have a Tag to read and write to a single bit. In order to address individual bits in Ignition, you must create a separate OPC Tag pointing directly to the specific bit in the PLC.

When the integer values that come from the OPC Tags are a series of binary bits, it is then possible to address each bit. For example, an integer value can have a 16-bit binary representation as shown here:

Integer	Bit Level Representation	How It Works
4096	0,0,0,0,0,0,0,0,0,0,0,1,0,0,0	`2^12 = 4096`
1025	1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0	`2^0 + 2^10 = 1025`

> **NOTE**
>
> Addressing bits may work differently depending on the type of device you are addressing. Most commonly you will either use `/<bit>` like `/0` or `/1`, or `[<bit>]` like `[0]` or `[1]`, or will use a . like .0 or .1. IE:
>
> - Micrologix: `[device]N7:1/0`
> - ControlLogix: `[device]Folder/Tag.0`
> - Siemens: `[device]I0.0`

Address an Individual Bit in a Micrologix

In this example, we want to address bits from a known integer value: 1025. This is represented as above, with the first (0) and eleventh (10) bits true and all others false. Our Gateway is connected to a MicroLogix PLC named MLX. To address an individual bit, do the following:

1. From the **Tag Browser** window, create an OPC Tag to have a specific value:
   - Data Type: **Integer**
   - OPC Item Path: `[MLX]B3:0`
2. Once created, set the Tag value to **1025**.
3. Then create another new OPC Tag to show only the value of the first bit of our previous Tag:
   - Data Type: **Boolean**
   - OPC Item Path: `[MLX]B3:0/0` (for Micrologic, you can specify the bit as: 0/0 or 0.0. That is, with a slash/ or a period.) `[MLX]B3:0/0` has a value of "1" or a Boolean value of "True" because the first bit is true (integer value is odd).
4. You can create a Tag for any of the other individual bits. For example, create a new OPC Tag with a Boolean value to the second bit of the original Tag as follows:
   - Data Type: Boolean
   - OPC Item Path: `[MLX]B3:0/1`
     `[MLX]B3:0/1` has a value of "0" or a Boolean value of "False" because the second bit is false.

Once you have your Tags created, try changing the boolean values and see the Integer Tag change.

User Defined Types - UDTs

UDTs (User Defined Types), also referred to as complex tags, offer the ability to leverage object-oriented data design principles in Ignition. UDTs are extremely important in Ignition. With UDTs, you can dramatically reduce the amount of work necessary to create robust systems by essentially creating parameterized "data templates".

By defining UDTs and using these “data templates”, you can generate tag instances to rapidly build complex screens. A change to the type definition is then inherited by all instances, drastically saving time when making routine changes.

The UDT data types are fully supported by **Vision Templates**, which means you can configure templates for your custom data types and take advantage of drag-and-drop binding to rapidly build complex screens.

Primary UDT Features

- **Object Oriented**: Use small or large groups of tags to create a single object. Create objects that match your real world devices or the existing structures in your PLCs.
- **Central Definition**: Once you define your data type, you can then create instances of it. If at a later time you want to change some aspect of the type, you can simply edit the type definition, and all instances of it are automatically updated.
- **Parameterized Settings**: Define custom parameters on your data type, and then reference them inside some or all of your member tags. When it comes time to create instances, you can simply modify their parameter values in order to change where the underlying data comes from.
- **Extendable**: Data types can inherit from other data types in order to add additional members or override settings. Instances can also override settings, allowing for flexibility when dealing with irregularities and corner cases.

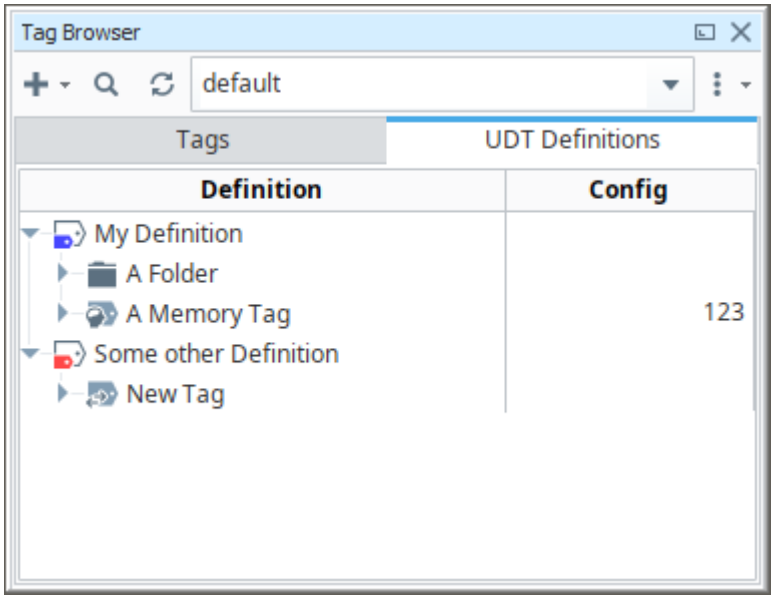
UDT Terminology

Many terms are frequently used when discussing UDTs:

Definition

A Definition represents the structure of a UDT. Definitions don't run, so tags inside of a Definition won't poll or subscribe to anything. Rather they represent a tag structure which Instances will inherit from. Changes made to a Definition are automatically applied to any Instances of that Definition.

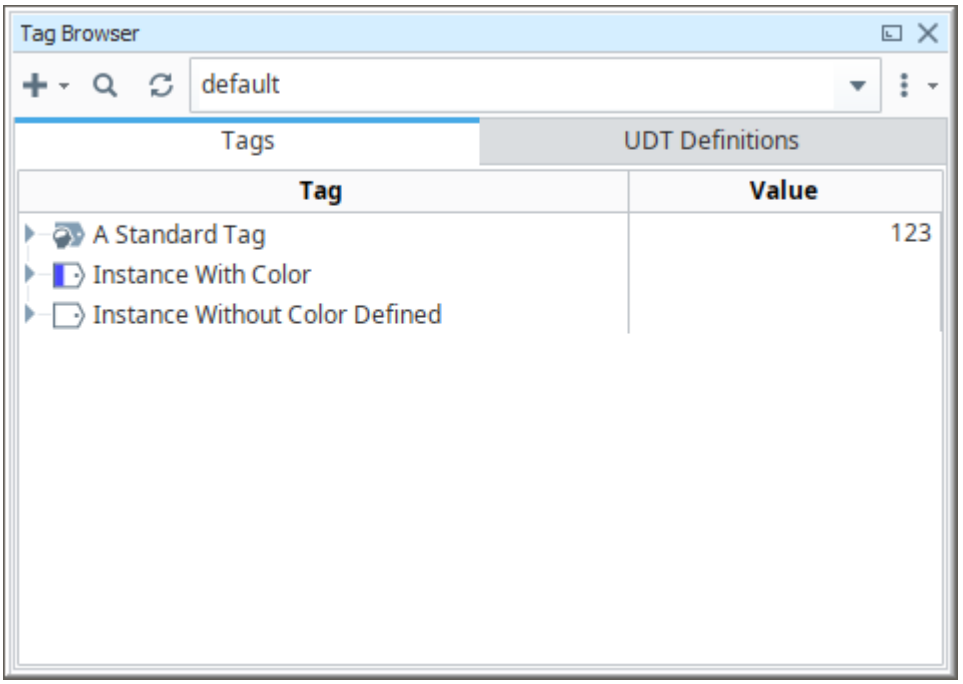
In the Tag Browser, UDT Definitions are always located under the **UDT Definitions** tab.



Instances

Instances are running copies of a Definition. All Instances have a "Parent Type", which is the definition that the Instance is inheriting from. The structure of an Instance is defined by its parent Definition, so you can not add new Members to an Instance. However, you can override the values on properties in any Member.

In the Tag Browser, UDT Instances can be found under the Tags tab, and are signified by either a plain white tag icon, or a tag icon with a vertical stripe. Furthermore, you can expand the UDT Instance and find the members (other tags) in the UDT.



Parameters

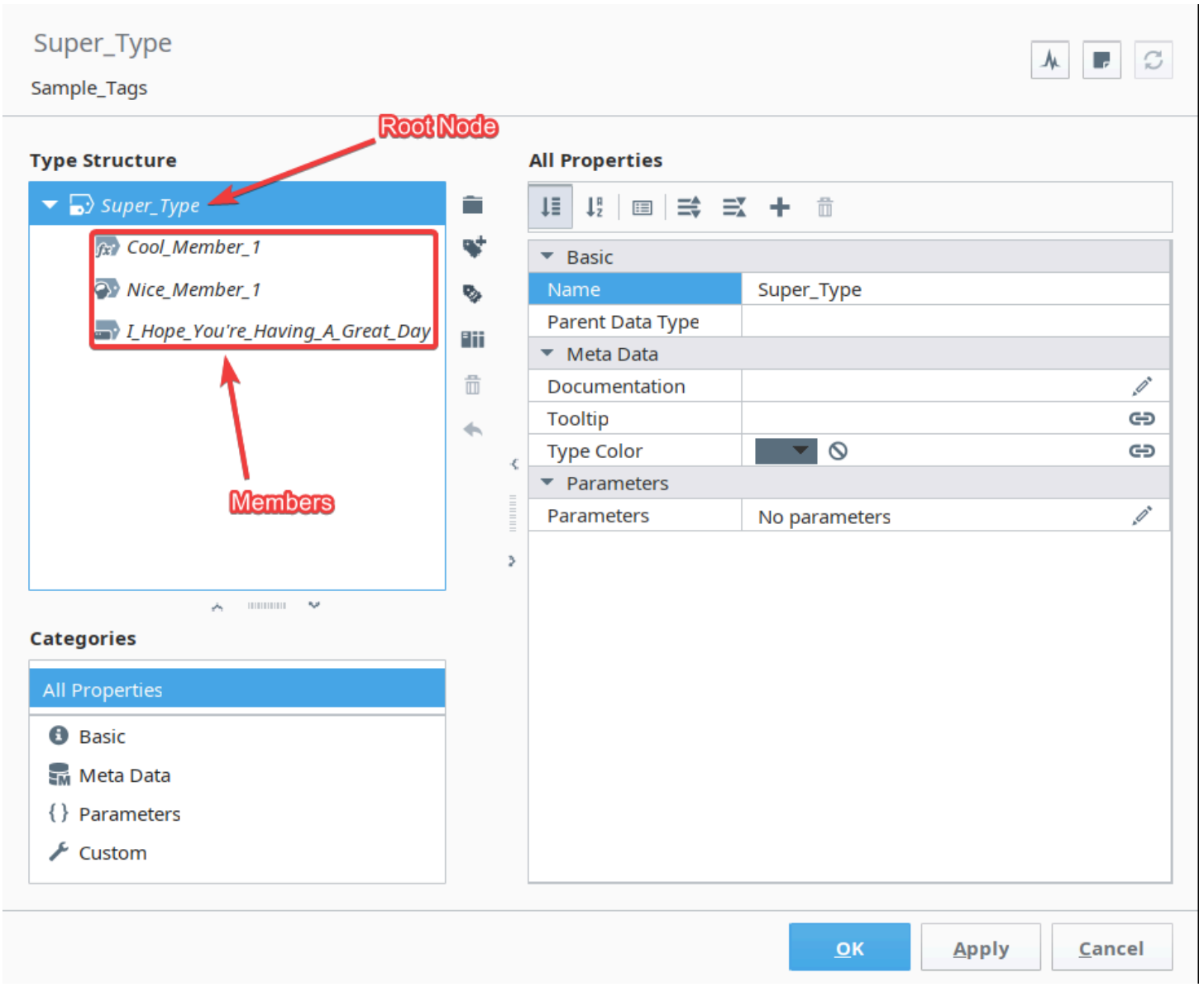
Parameters are user created properties that can be used to create parameterized data templates. Parameters are configured on Definitions, and their values can be overridden on individual Instances. You can replace values on a member in a UDT with a reference to a parameter, allowing for For example, if a data type consists of three OPC tags that only differ by a number in the path, you can use a parameter for the "base address", allowing instances to be created with only one setting.

Root Node

The top level item in a UDT.

Members

Members are the tags inside of a data type or instance. Members are always under a Root Node. Members can be standard tag types or an instance of another UDT.



Override

Instances are copies of a Definition, but in some cases you may wish to change the value of a property on a particular member (tag) in a UDT instance. This is called overriding the property, allowing the property to have a value that deviates from the Definition.

Creating a UDT Definition and Instance

INDUCTIVE UNIVERSITY

Creating UDT Definitions

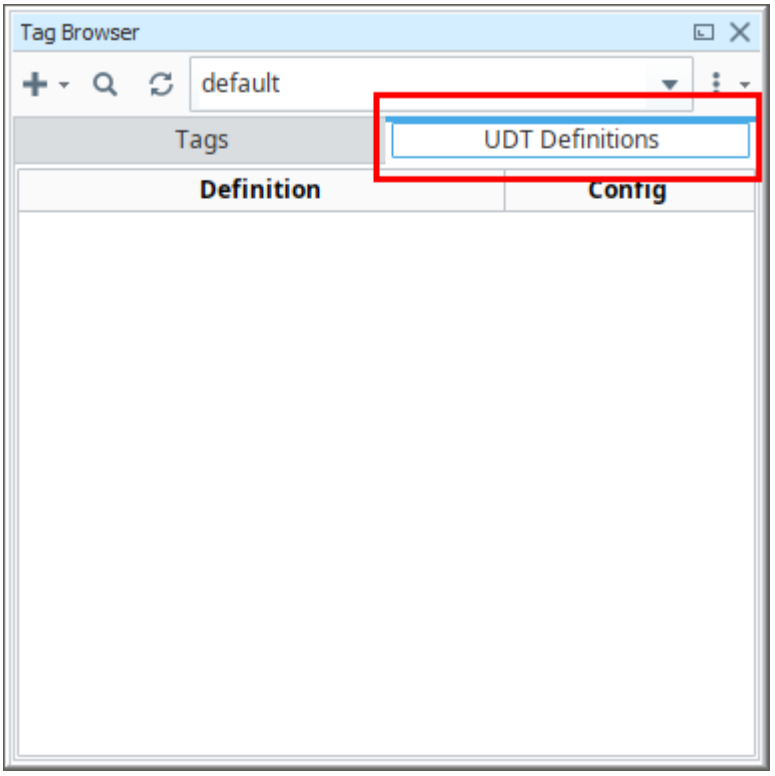
Watch the video

In these series of examples, we will create a very simple UDT Definition, make an instance of it, and make some additional configuration changes.

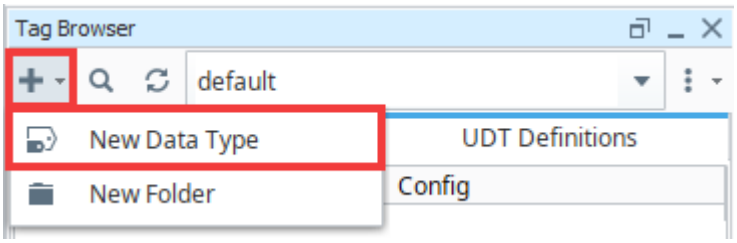
Creating a Definition

In this example, we'll demonstrate how to create a UDT Definition.

1. To create a UDT Definition, first switch click on the **UDT Definitions** tab in the Tag Browser. This is the only section where you can make UDT Definitions.

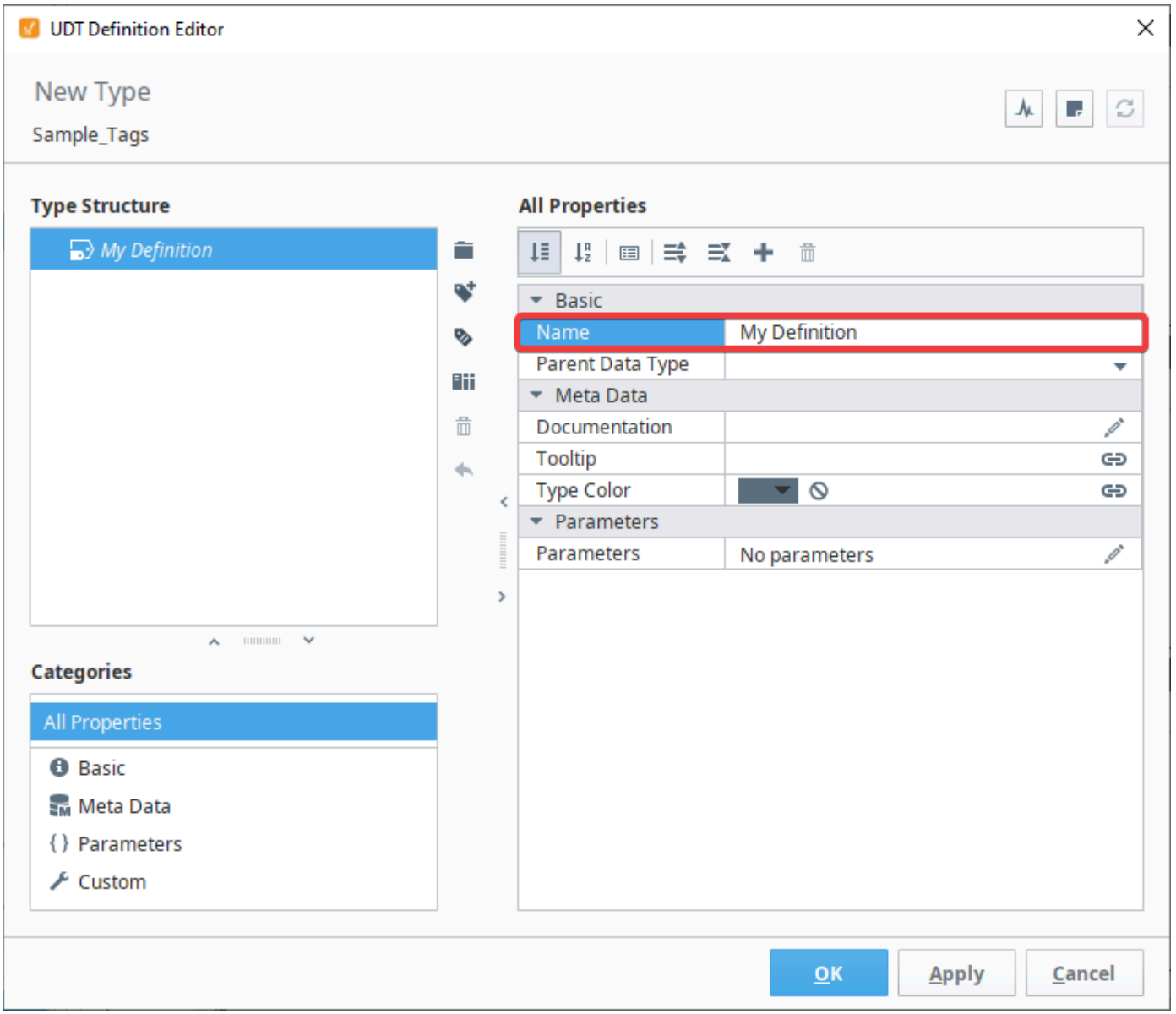


2. Next click the **Add** ✚ icon, and select **New Data Type** from the dropdown.

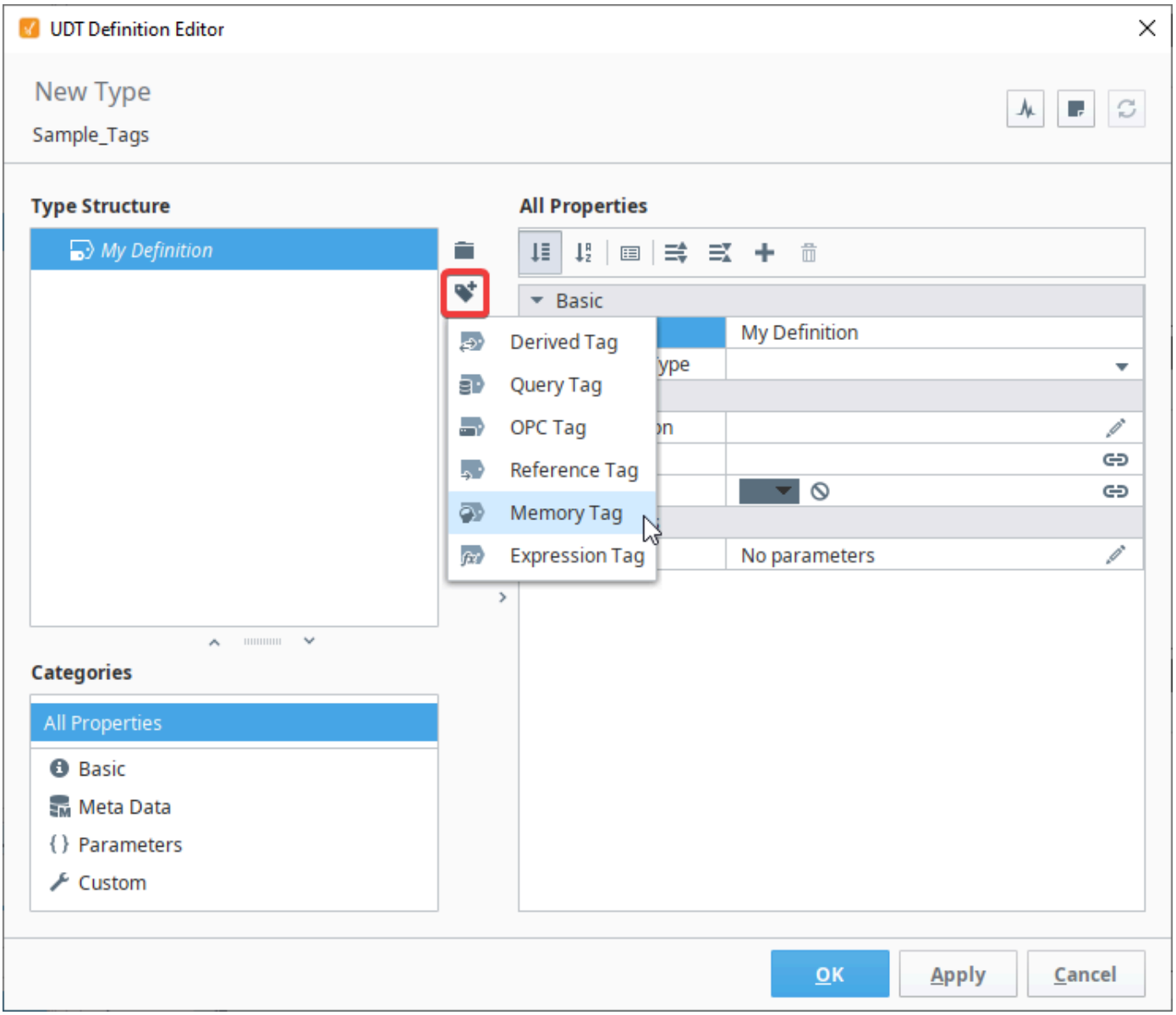


3. The Tag Editor will open, allowing you to create a new UDT Definition. To start, enter a name into the **Name** property. For this example, we used **My Definition**. This name is how the UDT will be identified by the rest of the system.

> **NOTE**
>
> Names can be changed later, but doing so after you've already made instances will create what's called an "orphaned UDT instance:" an instance that is no longer associated with a definition. It's generally advised to avoid changing the name of a definition unless you're also willing to update any of the instances.

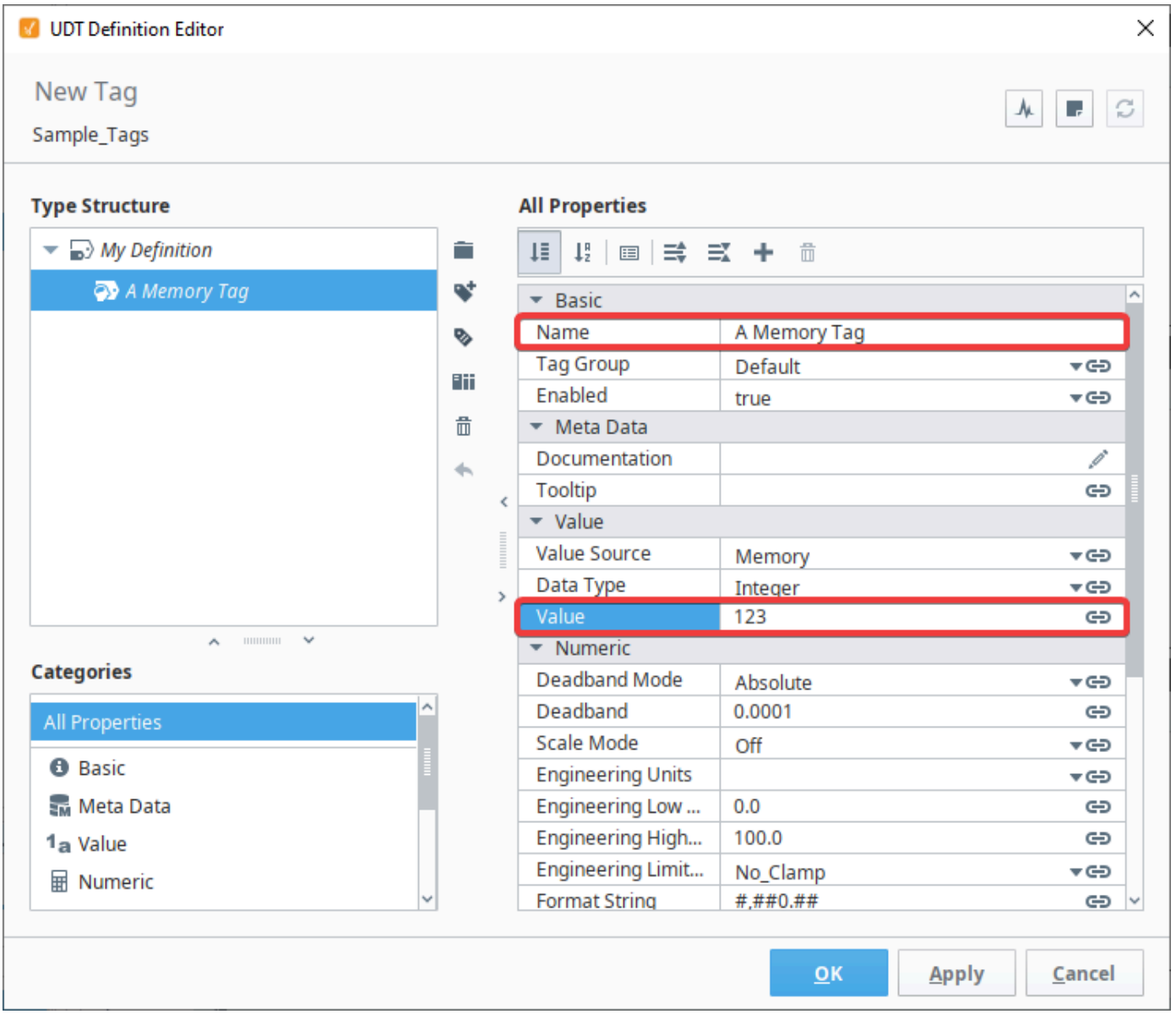


4. We'll keep this demonstration simple, and only add two members. We'll create a Memory tag and add a tag from a connected device. Let's create the Memory tag first. Click the **New Tag** icon, and select **Memory Tag**.



5. The Tag Editor will now add a Memory tag to the **Type Structure** tree and select it, allowing the **Properties** table to show settings for the selected tag.

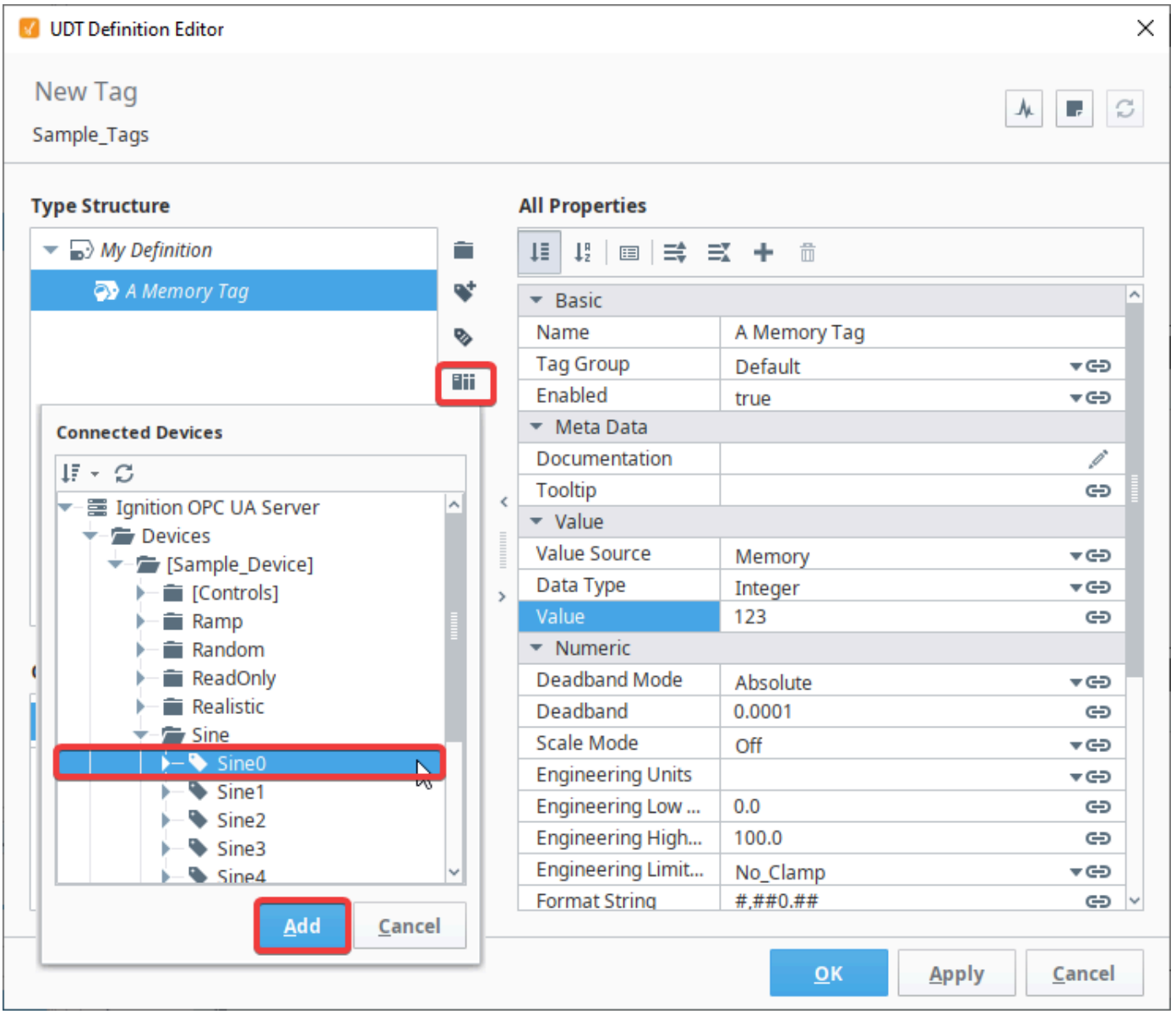
6. Change the Name to **"A Memory Tag"**, and set a Value of **123**.



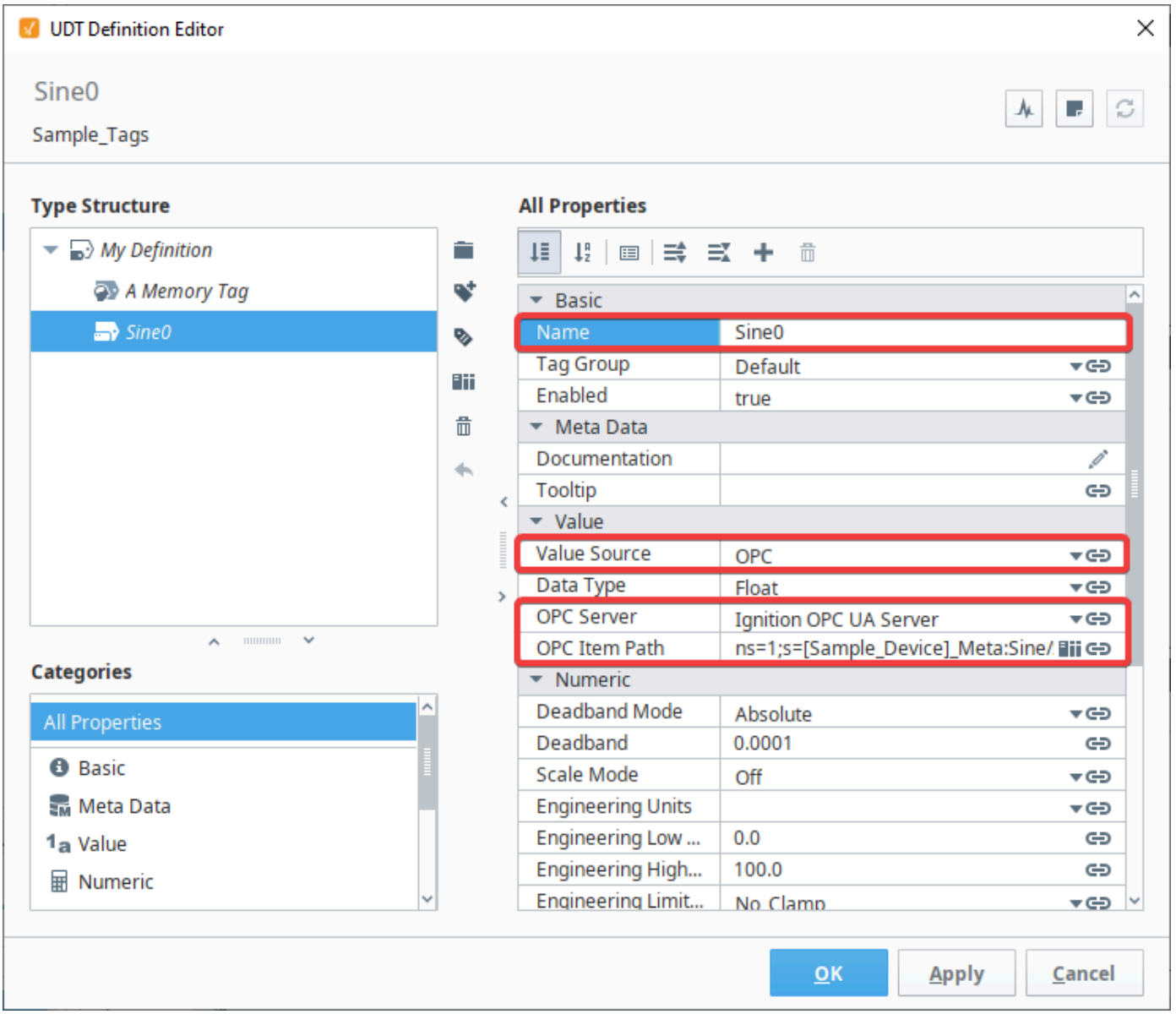
7. Now, let's add a tag from the OPC UA server. Click the **Connected Devices** icon, and a window will open showing you your Connected Devices. Expand **Ignition OPC UA Server>Devices>Generic>Sine**, and select **Sine0**. Click **Add**.

   You can browse OPC devices from within the Tag Editor in order to add OPC nodes to UDT definitions.

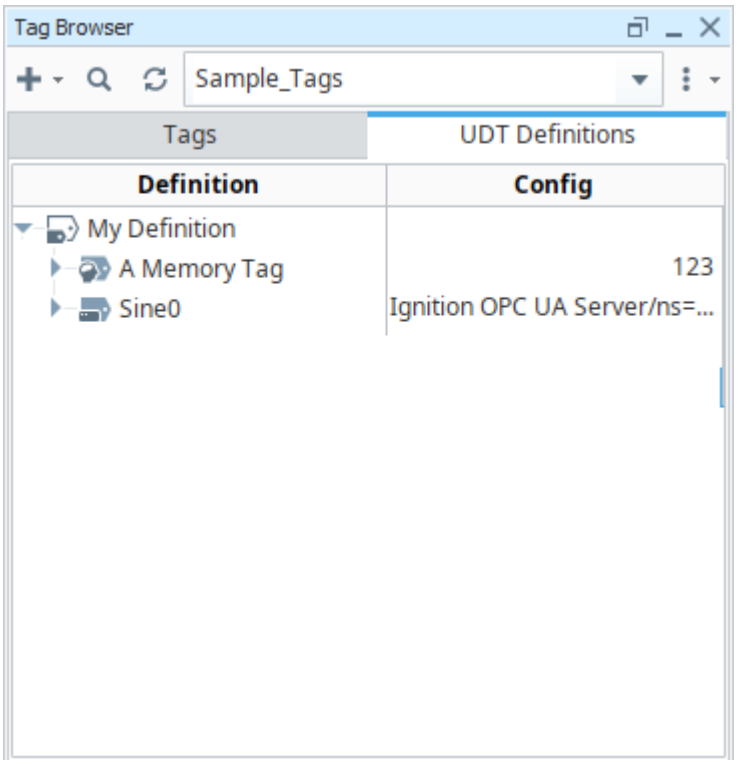
> **NOTE**
>
> When adding tag(s) from a device, it requires that you have some connected devices such as the Ignition OPC UA Server as in this example.



8. You'll see that the **Sine0** tag was added to the **Type Structure**. Select the **Sine0** tag and you'll see all the property settings for that tag.



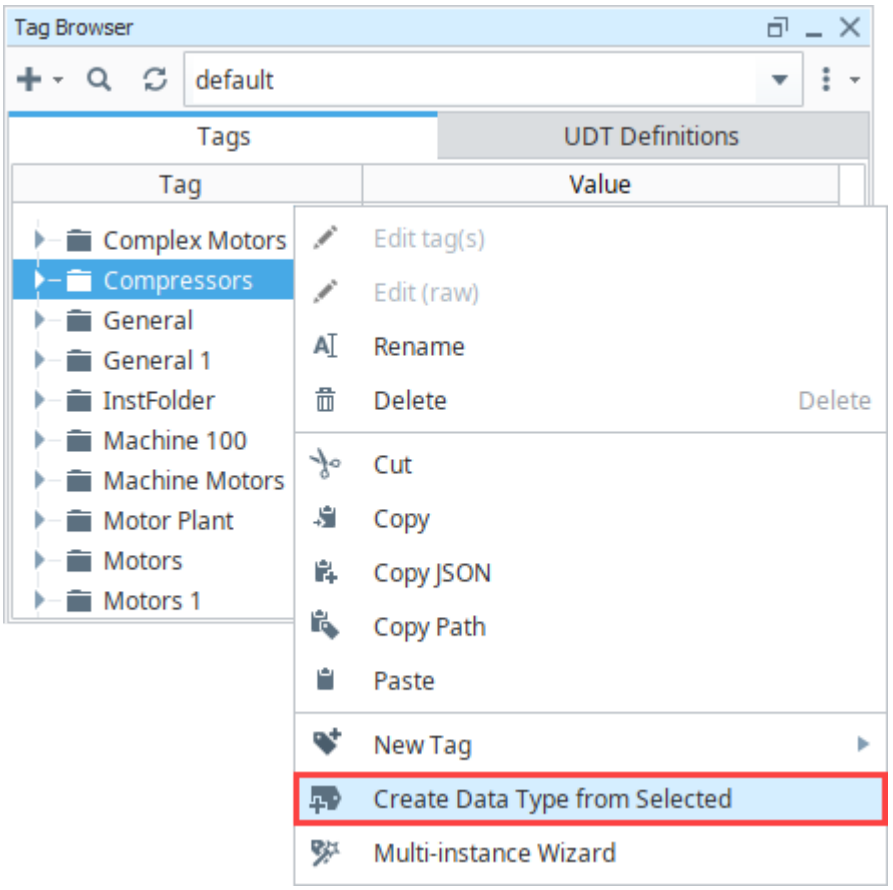
9. Click **OK**. This will close the Tag Editor, and apply your changes. The Tag Browser will now show your UDT Definition. You can expand the "My Definition" item to make the Memory and Sine0 tags visible.



Creating a Data Type from Existing Tags

Data types can be created from existing tags. To create a data type from existing tags, simply select the tags or folders you wish to include, right-click and select **Create Data Type from Selected**. The Tag Editor window is displayed with the selected tags pre-populated as members. From here you can modify the tags, add parameters, and more. The original tags will not be affected.

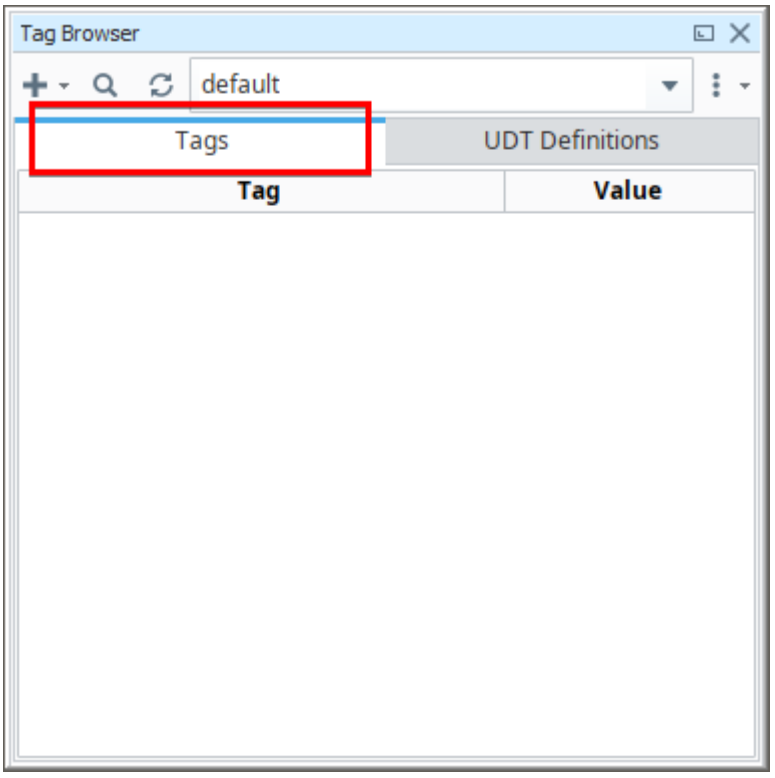
> **TIP**
>
> If you select a single folder as the root to create the type from, its sub-members will be added, and its name will be the basis for the type (that is, the folder itself won't be included in the structure).



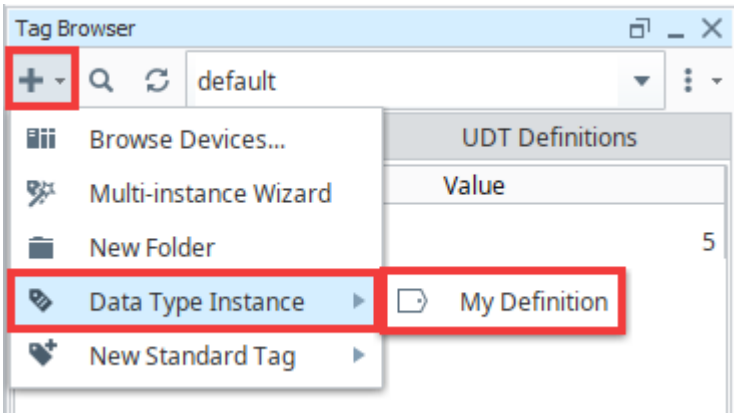
Creating an Instance

Now that we have a UDT Definition, we can create a UDT Instance. You can use the **Multi-Instance Wizard** to make many instances quickly, but you can also create a single instance.

1. In the Tag Browser, switch back to the **Tags** tab. This section of the Tag Browser is where you create UDT instances. Definitions can not be placed in this section.

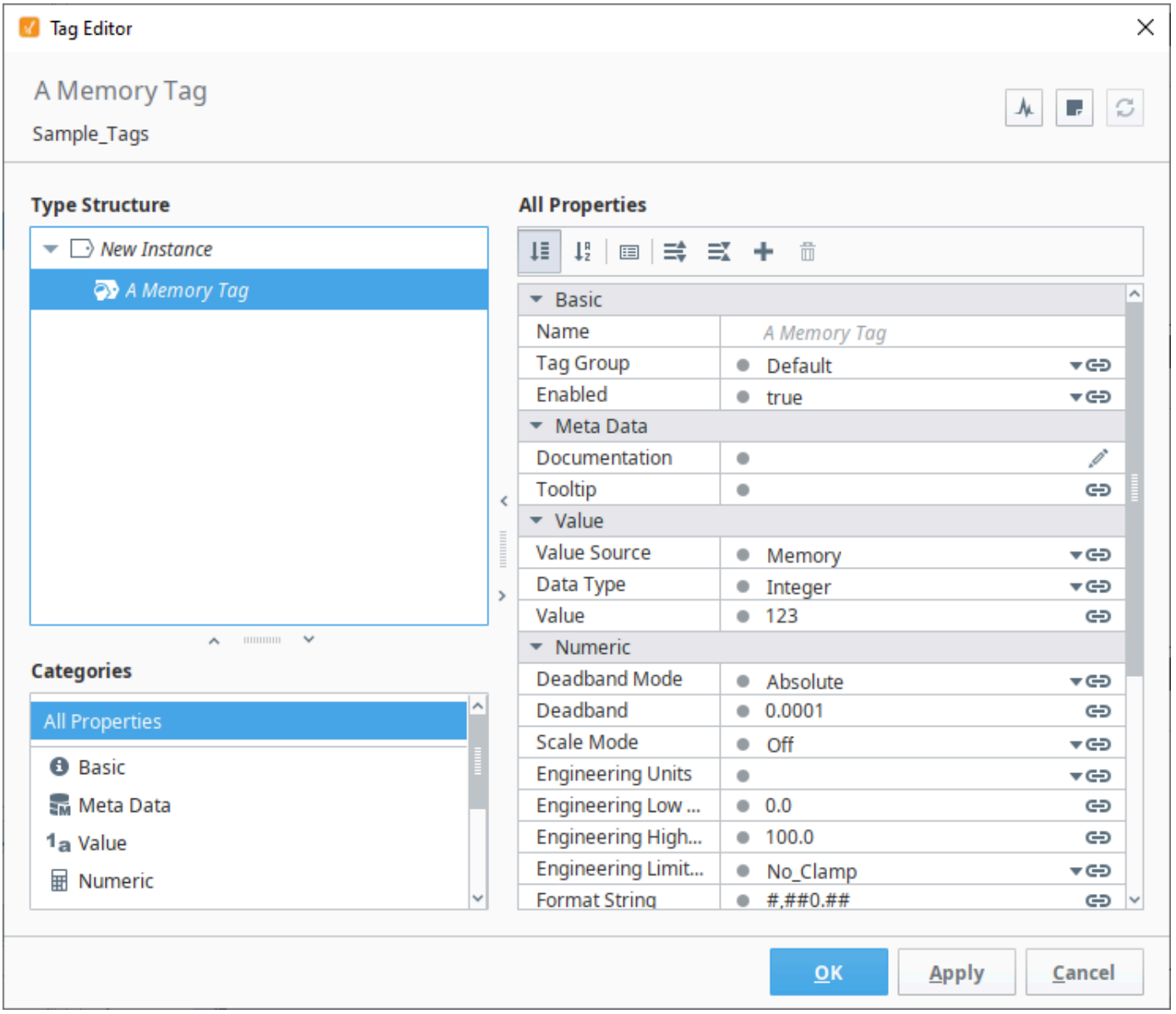


2. Click the Add icon and hover the mouse cursor over **Data Type Instance**. This will expand and show all of the UDT Definitions that exist within the active **Tag Provider**. Select the definition we created in the previous example.

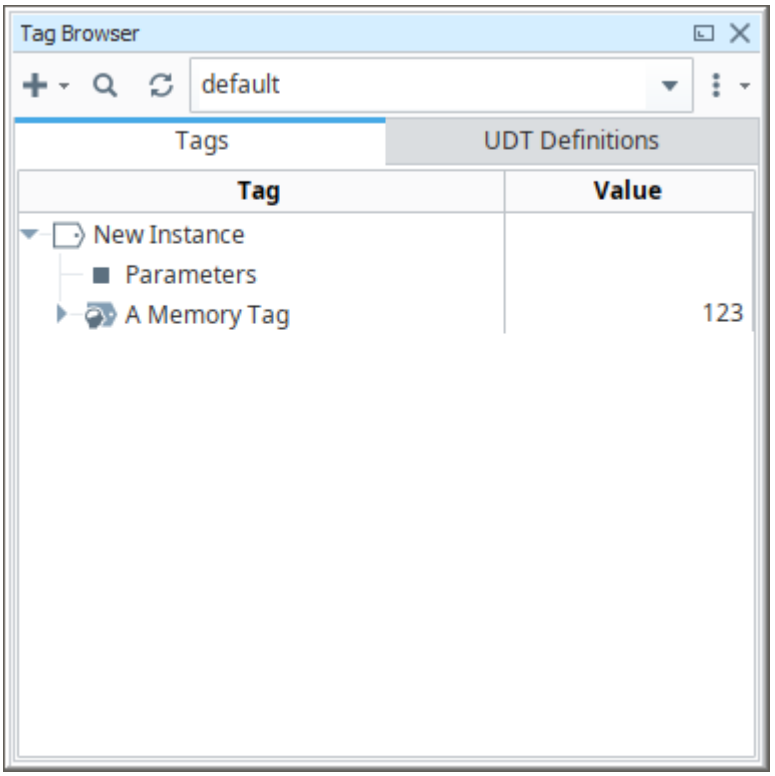


3. Again, the **Tag Editor** will open. This time allowing you to edit an instance of the UDT. From here you can name the Instance. In this case we'll use the name "New Instance".

4. If you select the member ("A Memory Tag"), you'll notice that many of the properties have a grey dot. This signifies that the properties are inheriting their values from the definition.



5. We won't make any changes here. Just click **OK**, which closes the Tag Editor and creates a UDT instance.



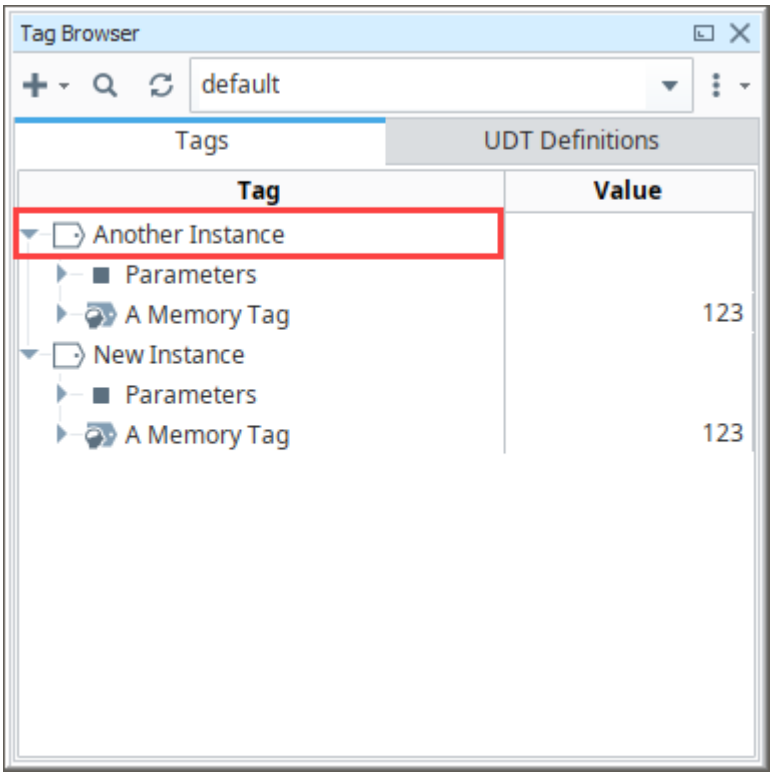
Override Instance Properties

Instances inherit their structure and properties from the Definition, but property values can be overridden on an instance.

1. Now that we have an instance, let's create another. You can use the same steps in the last example, or just copy and paste **New Instance** from the Tag Browser's right-click menu.

2. Name the new instance "Another Instance".

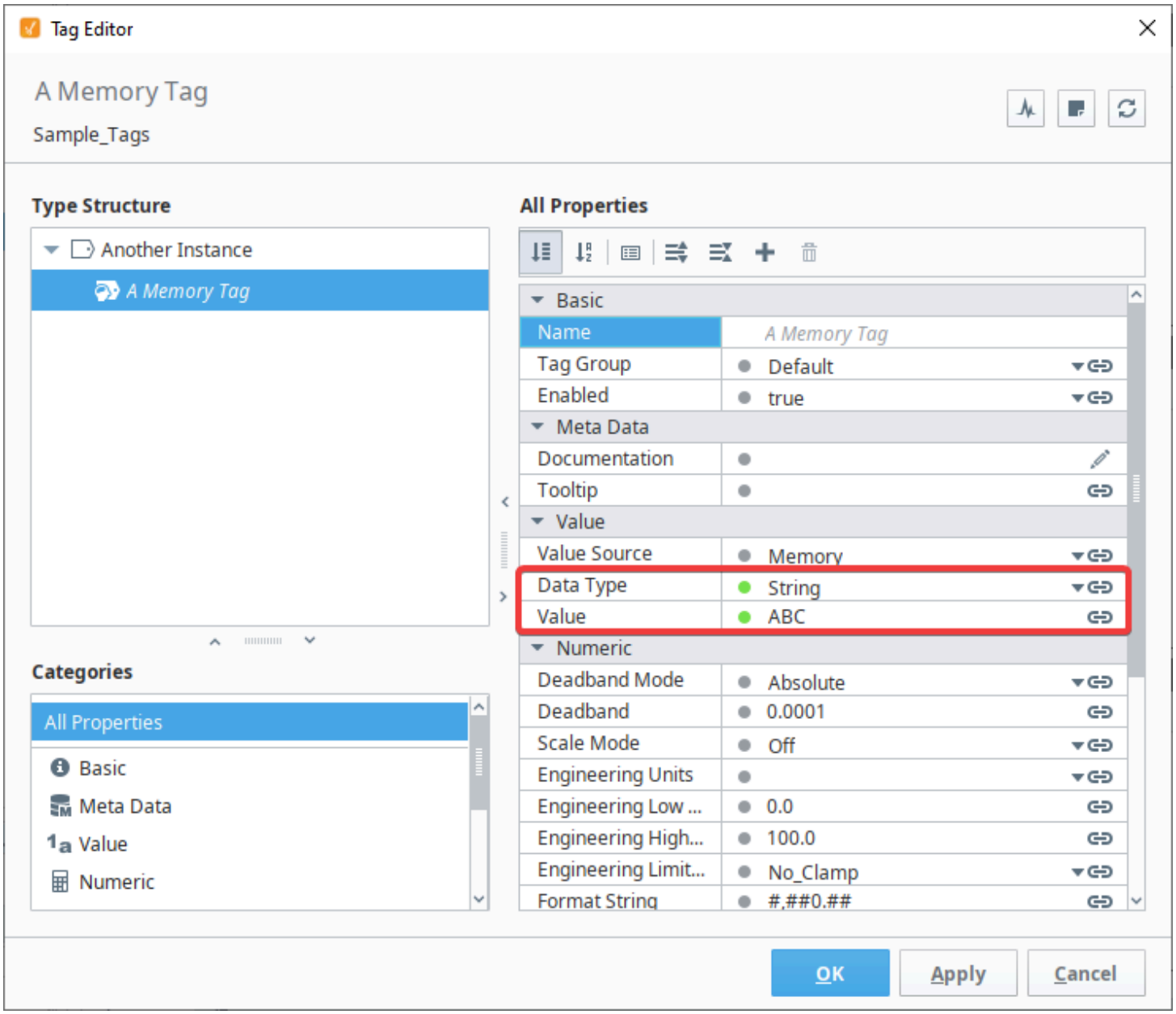
3. If you need to make changes to a UDT Instance, you can open the **Tag Editor** by double clicking on the Instance, or any of its members.



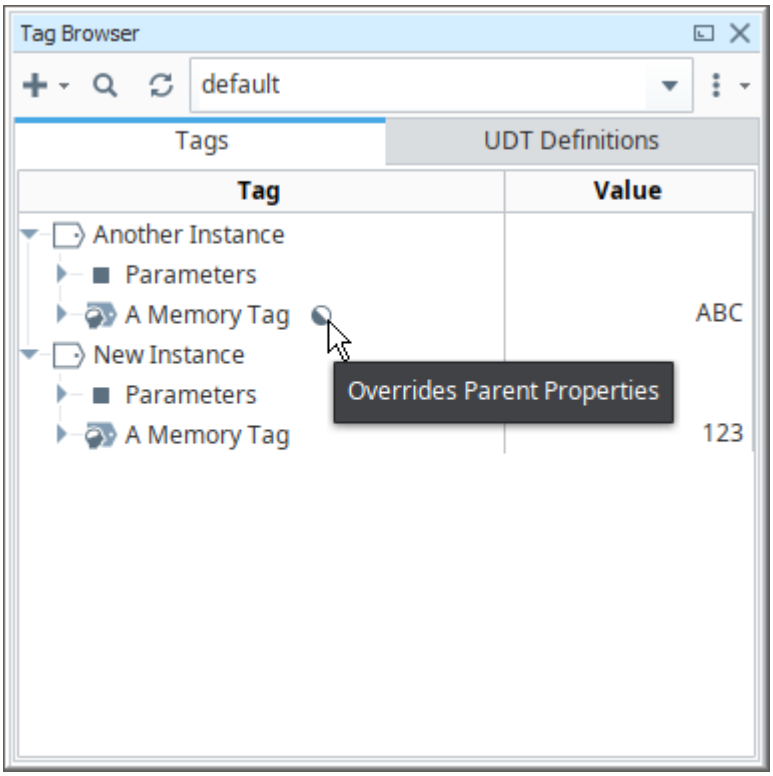
4. With both instances expanded, we can see that aside from the names, they're identical. Let's change that. Double click on **Another Instance** or any of its members to open the Tag Editor.

5. With the Tag Editor open and focused on **Another Instance**, select the Memory tag. Change the Data Type to a **String**, and the Value to **"ABC"**.

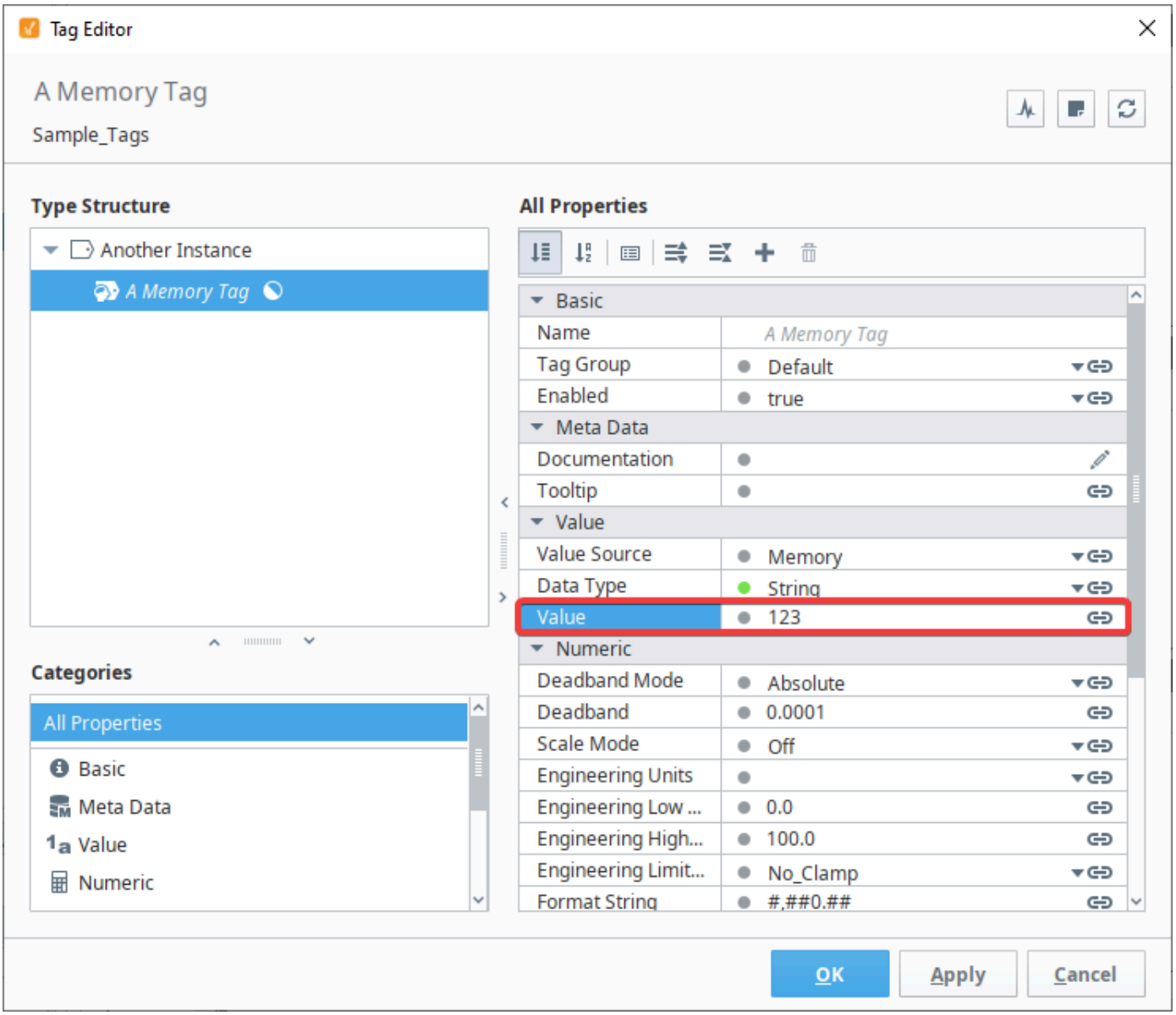
You'll notice that as you make changes to the properties, the grey dots become green signifying that the property values now differs from the Definition. This concept is known as "Overriding" a UDT member property.



6. Click **OK** to close the Tag Editor and apply your changes. You'll notice that the mMmory tag in the instance we edited now has an attribute icon, signifying that one or more properties on the member are overridden.



7. If you wish to remove the overrides, simply edit the tag again, and click on the green dot. This will remove the override and cause the property to revert to the value on the Definition.



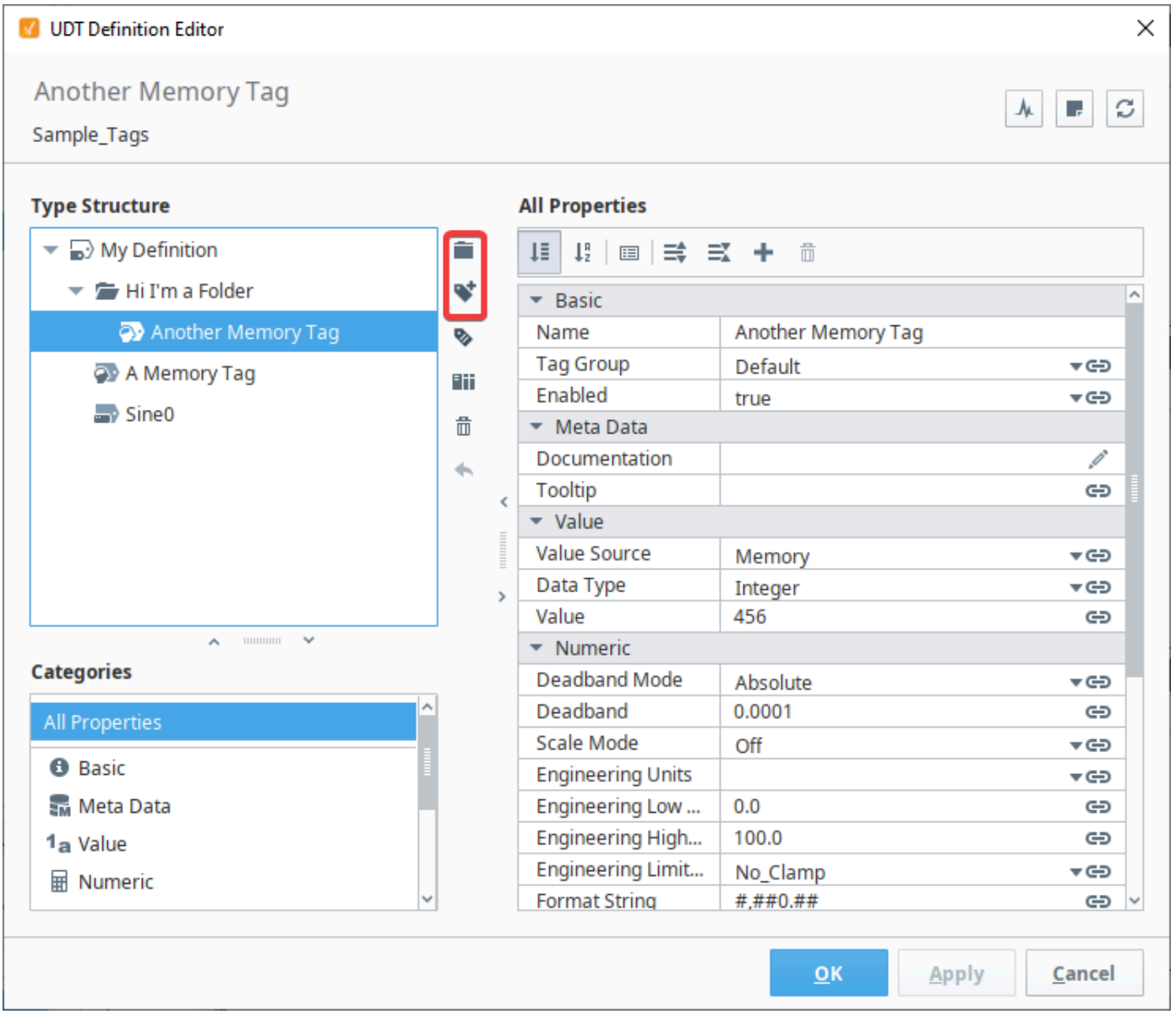
Make Changes to the Definition

Now that we have some instances, we can make a modification to the Definition, which will propagate the change down to the Instances.

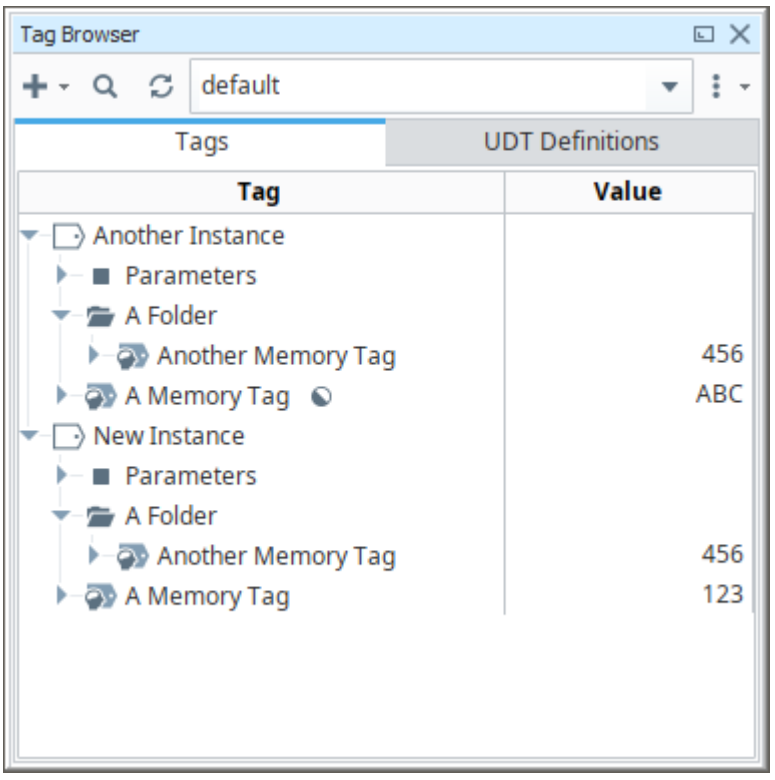
1. Switch back to the **UDT Definition** tab in the Tag Browser.

2. Edit the **My Definition** UDT.

3. Use the **Add Folder** and **Add Standard Tag** icons to add a new folder, and a member inside of that folder.

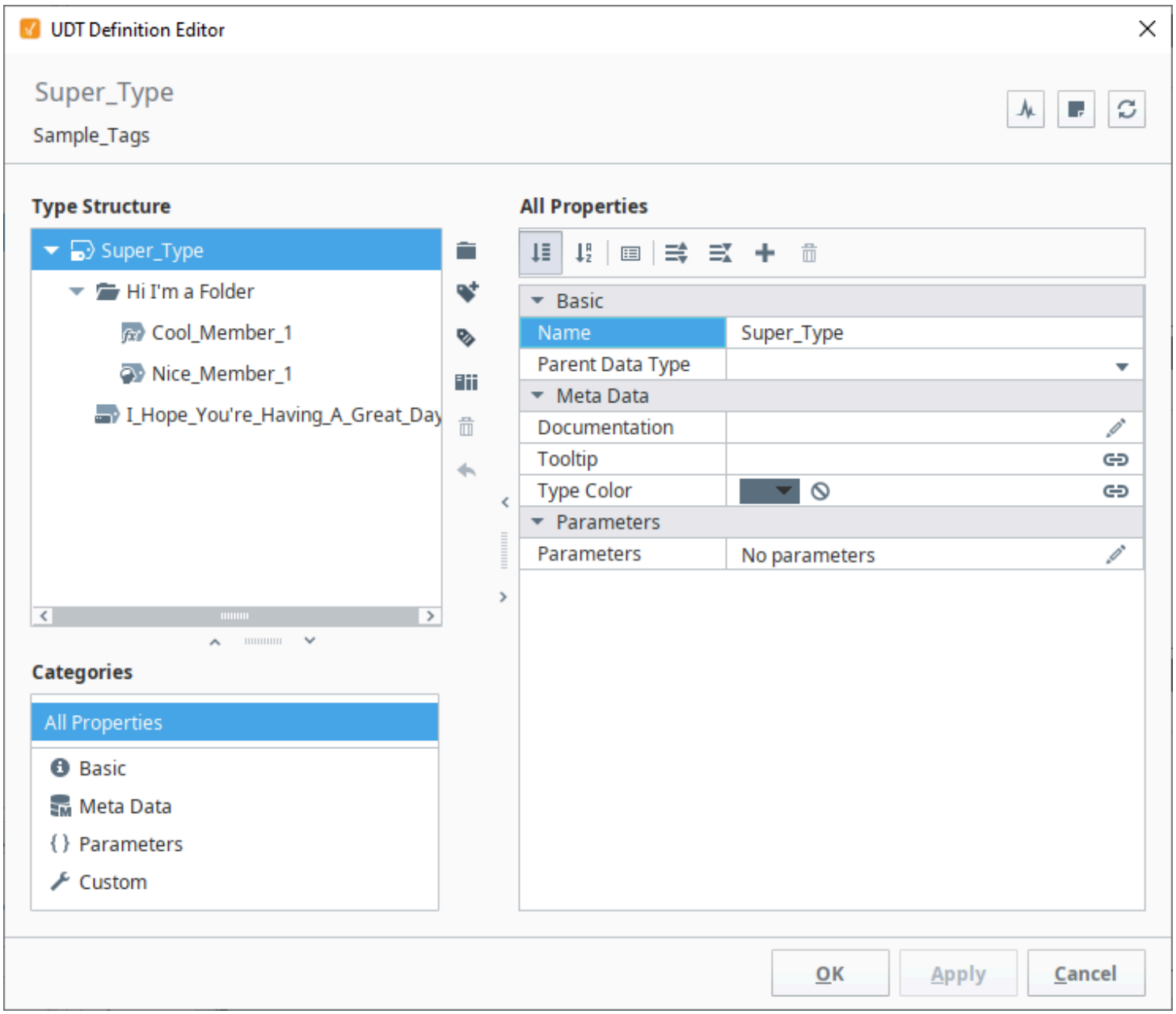


4. Click **OK** to switch back to our instances,. You'll see they now both have new members.

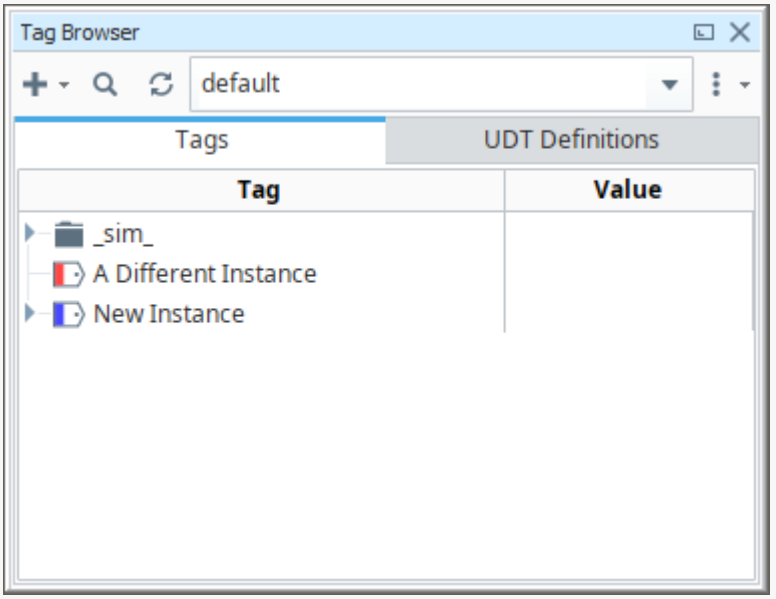


UDT Root Node Properties

While editing a UDT Definition or Instance, the Tag Editor will show some unique properties on the Root Node.



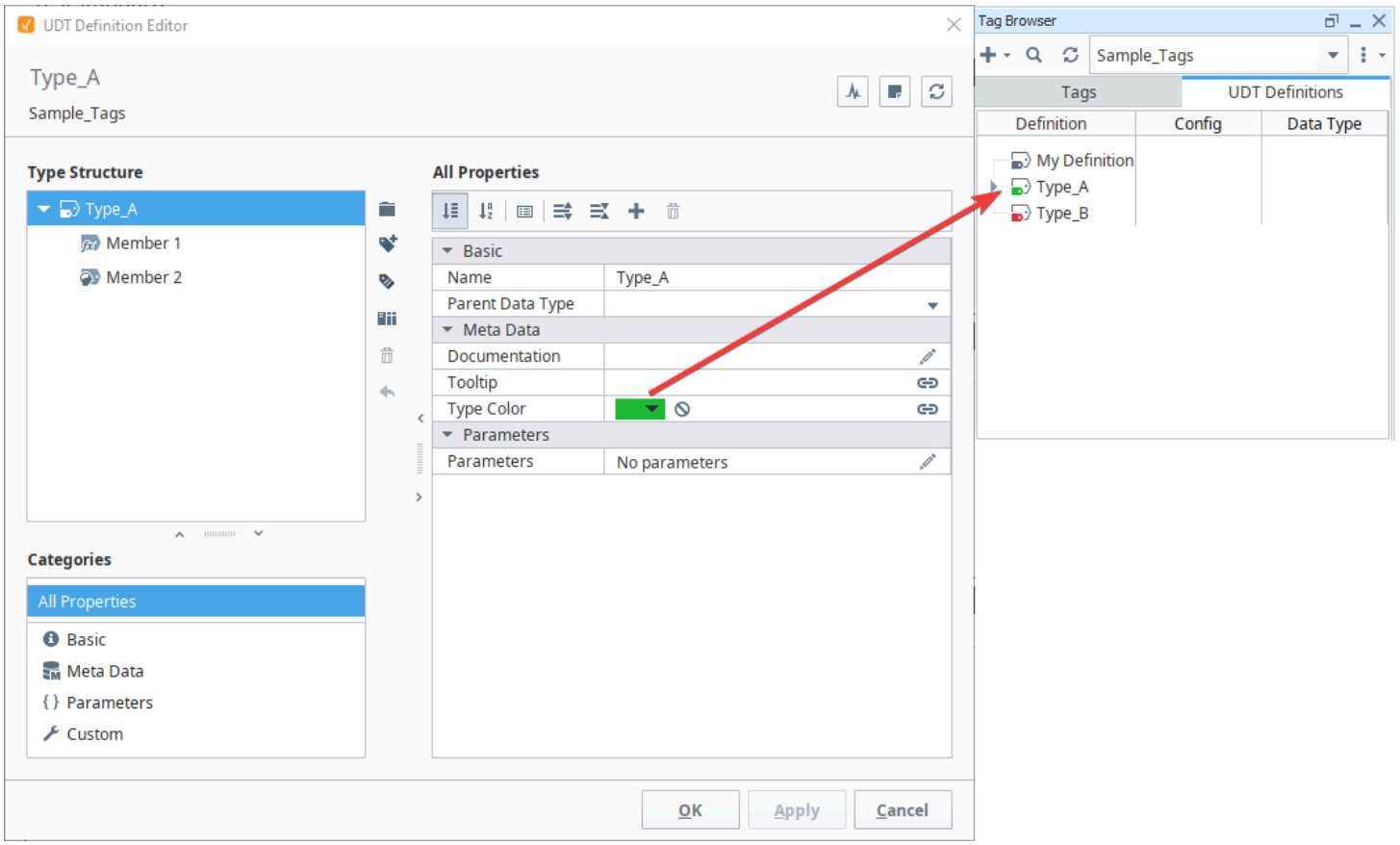
Property	JSON Name	Description
Name	`name`	The name of the UDT Definition.
Parent Data Type	`typeId`	Both Instances and Definitions have this property, but the implications of the property are different. **On a Definition** - The name of the UDT Definition that this Definition is inheriting from. If blank, then the UDT being edited does not inherit from another UDT. **On an Instance** - The name of the UDT Definition this UDT is an instance of. Changing the Parent Data Type of an instance is not supported.
Documentation	`documentation`	A freeform text property for information about the tag or UDT.
Tooltip	`tooltip`	The tooltip provides a hint to visual components as to what should be displayed when the user hovers their mouse cursor over the component that is being driven by the value of this tag or UDT.
Parameters	`parameters`	A collection of **parameters** configured on the Definition. Note that you can only add or remove parameters on Definitions.

Property	JSON Name	Description
Type Color	`typeColor`	A color that will be applied to the Definition and any Instances. This property is only cosmetic, but can be useful to have certain UDTs stand out from others in the Tag Browser. 

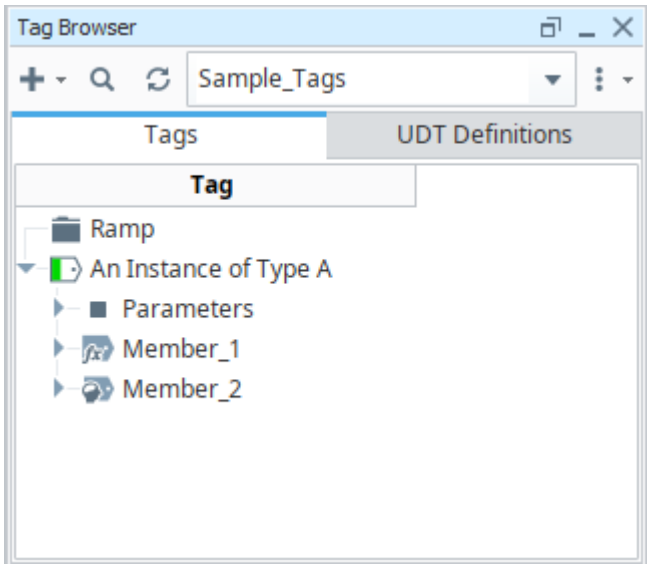
Assigning Colors to UDTs

UDTs can be color coded, which applies a color to the Root Node. This is purely a cosmetic change, but can be helpful in systems with a large number of instances, as the colors can make certain UDTs stand out from one another.

Color is applied to the Definition, via the **Type Color** property.



Any instances of that type will apply the color to the Root Node.

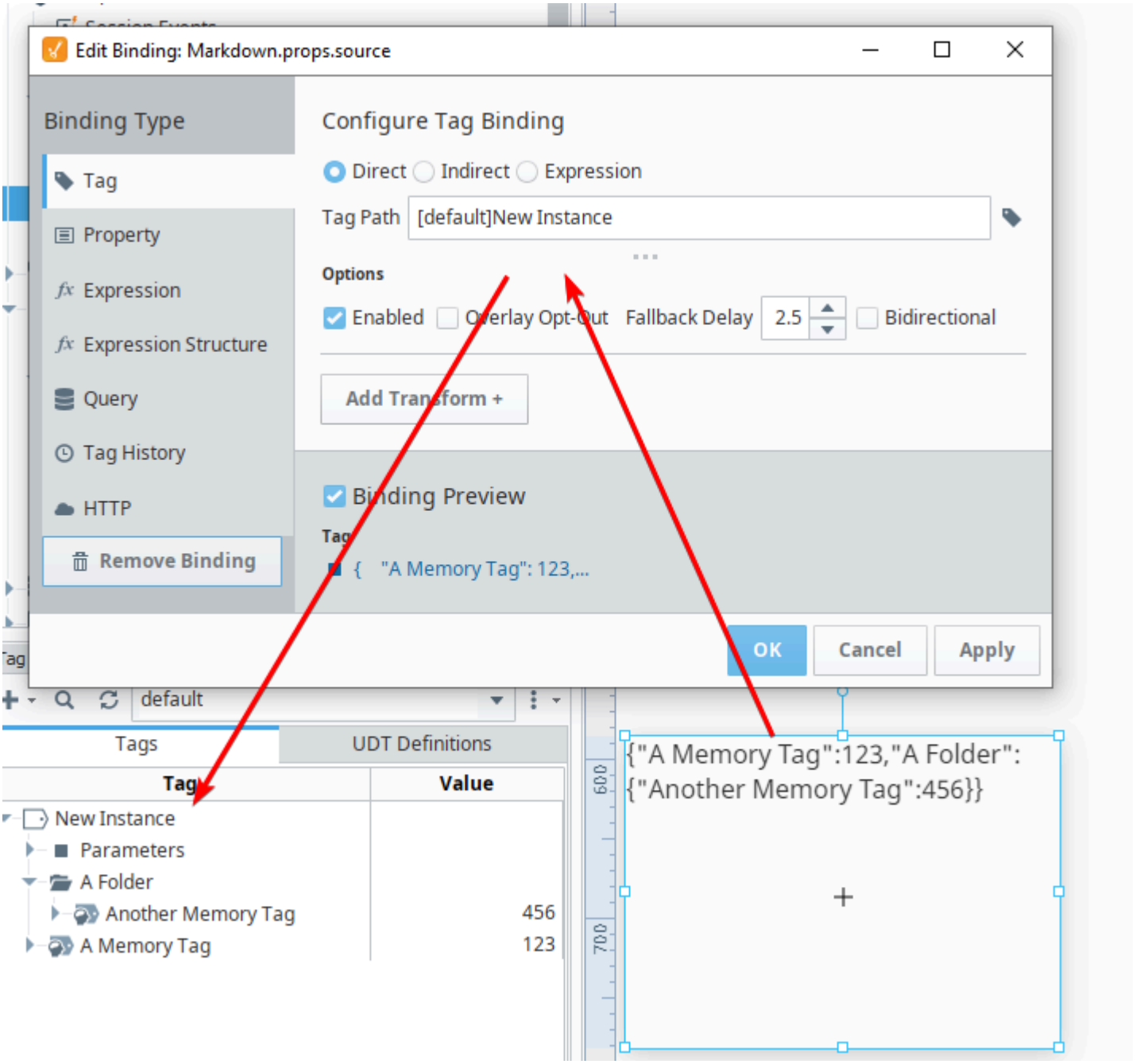


Binding to UDTs

Both Perspective and Vision feature a "binding" system, that allows components in those modules to display live values on tags. In regards to UDT instances, component bindings can bind to members just like any standard tag, but they can also bind directly to the UDT instance, which results in the binding receiving live live values from all members of the UDT.

In the image below, a Perspective Markdown component has a Tag binding on its **source** property. The binding is leading to the root of a UDT instance. As a result, the member values are shown on the

component. They're also live values, so any value changes on any member will appear on the Markdown component.



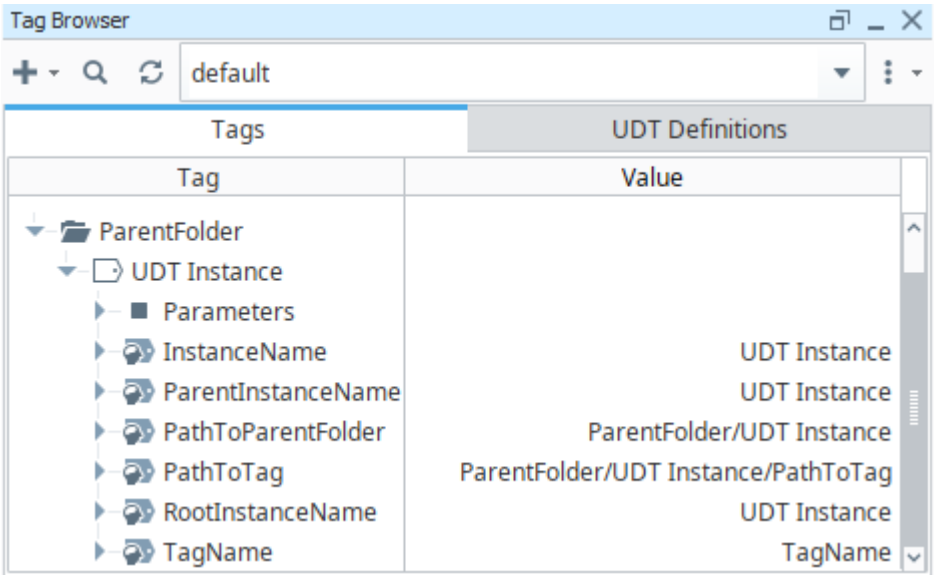
UDT Parameters

Parameters effectively act as variables that can be referenced by properties on members. A common use case for a parameters in UDTs is to make the OPC Item Path on OPC tag members dynamic, allowing you to replace parts of the OPC Item Path with the parameter's value. However, parameter values can be referenced by other member properties, such as expressions on Expression tags.

UDT Parameters are configured on UDT Definitions. Instances of a UDT can override the value of a parameter, much like any other property on an Instance.

Pre-Defined Parameters

UDTs have a few parameters already defined to make things easier for you. They give you access to the name and various paths associated with a UDT member tag. These parameters can be accessed from anywhere in a tag that a normal parameter can be used. Each of these parameters uses that tag it is in as a starting point for its path.

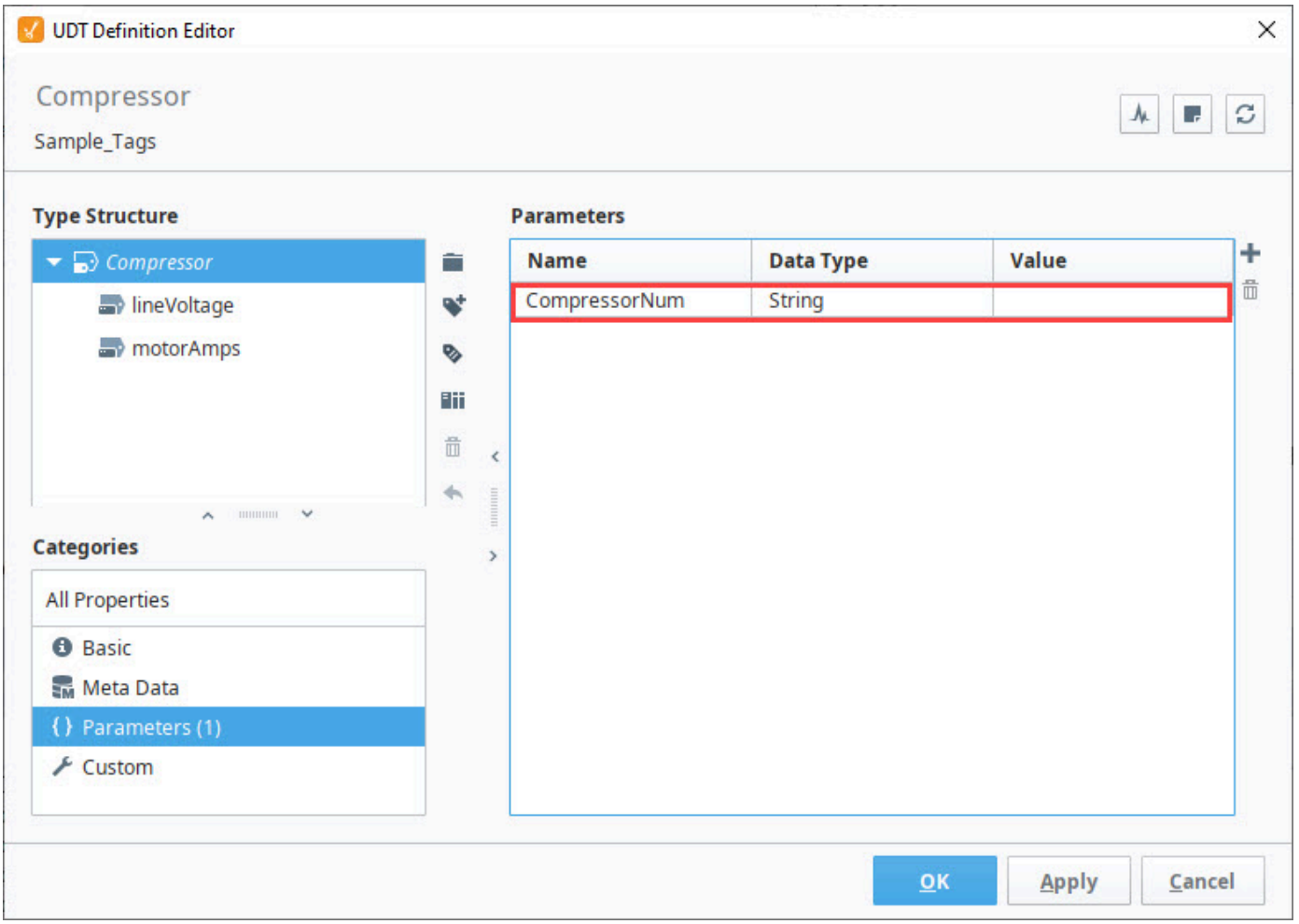


Parameter Name	Description
`{InstanceName}`	The name of the UDT Instance that this tag is inside. In cases where UDTs are nested, this parameter will return the name of the UDT the member belongs to. Thus referencing this parameter in a nested UDT will return the name of the nested UDT.
`{ParentInstanceName}`	The instance name of the parent UDT Instance. This is similar to InstanceName, except the name of parent UDT Instance will be returned when binding from UDT Instance parameters.
`{PathToParentFolder}`	The full path to the folder that this tag is in.
`{TagName}`	The name of the tag that is using this parameter.
`{PathToTag}`	The full path to the tag using this parameter.
`{RootInstanceName}`	Returns the top-most UDT Instance name.

Adding a Parameter to a UDT

In this example, our plant has multiple compressors. We created a Compressor UDT, but we want each instance of the Compressor UDT to reference a different set of tags. Our Compressor UDT has two OPC tags: lineVoltage and motorAmps. These two tags are pointing to a specific address in the PLC. In order to reference a different set of tags for each instance, we need to add a parameter to our Compressor UDT that we called **"CompressorNum."**

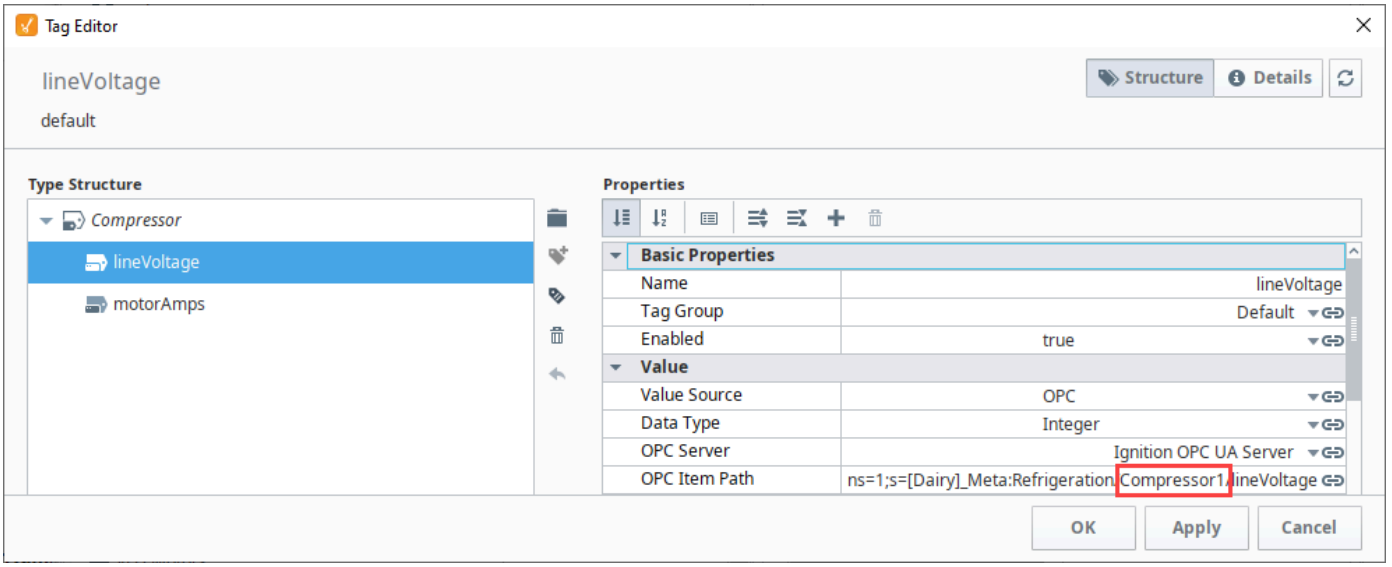
1. In the Tag Browser, we clicked on the **UDT Definitions** tab to find our Compressor UDT.
2. Double click on the Compressor UDT to open it in the Tag Editor and click on the edit icon next to the **Parameters** property. The Parameters pane will open.
3. Click the Add icon to create a parameter. Enter the parameter **Name** and Data Type of **String**.



Referencing Parameters from Member Properties

Continuing with the example from above, this next example will show you how parameters are referenced from member properties.

1. In the Tag Editor, replace the Compressor Number for each tag with the new parameter **CompressorNum**. Select the **lineVoltage** tag and click on the binding icon and click **Edit**.



2. This opens the OPC Item Path window for editing the lineVoltage tag. Place your cursor at the end of `Compressor1`, delete the `1`, and enter `{CompressorNum}`. Click **Commit**.

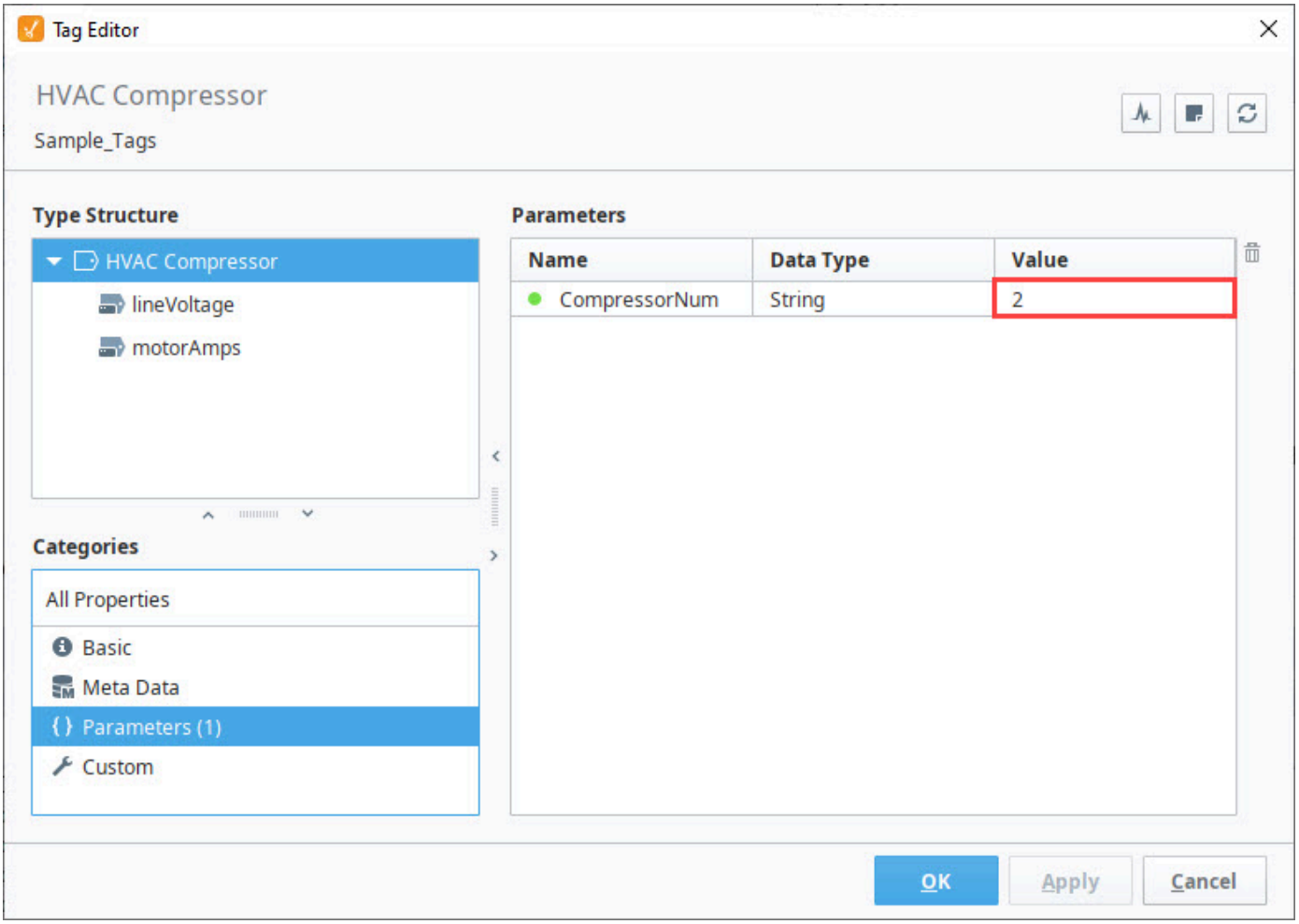
   You may also edit tag paths inline in the Tag Editor rather than opening the OPC Item Path window. For this example, you would highlight `1` in the OPC Item Path and replace it with the new parameter, `{CompressorNum}`.

3. Repeat Step 2 for the **motorAmps** tag. .1 Both tags will now show the **CompressorNum** parameter in the OPC Item Path.

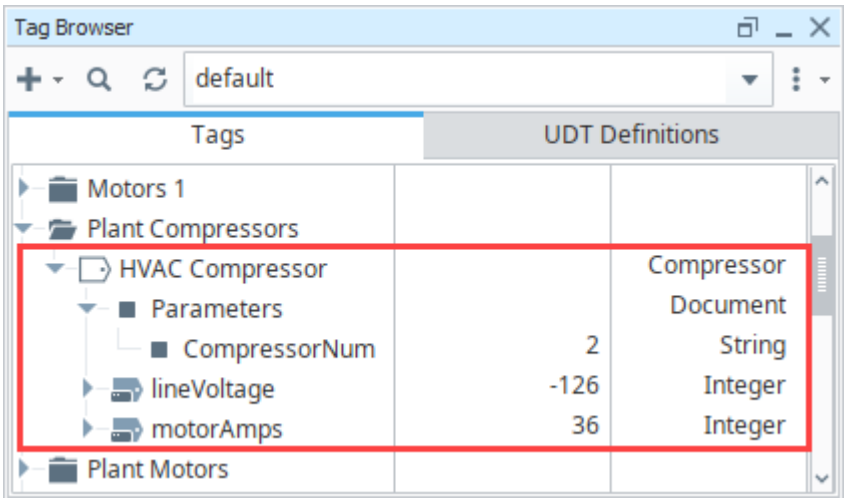
4. Now, let's create an instance of the Compressor UDT using the CompressorNum parameter under the **Tags** tab of the Tag Browser. To keep UDT instances organized, we created a Plant Compressors folder.

5. Right-click on the Plant Compressors folder and select **New Tag > Data Type Instance > Compressor UDT**.

6. Enter a Name for the Instance (i.e., HVAC Compressor), then click on the pencil icon next to the **Parameters** property and enter a value (i.e., **2**). Click **OK** to save the HVAC Compressor instance.



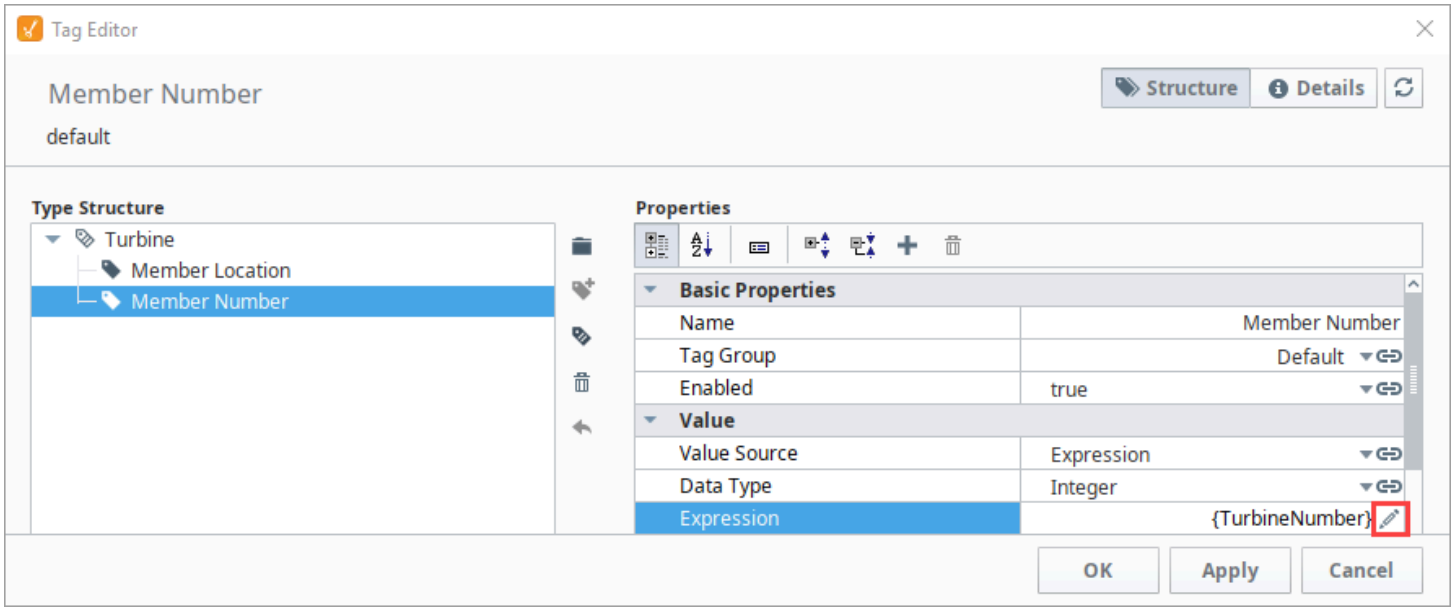
7. Under the **Tags** tab, you'll see the HVAC Compressor was created showing the Parameter that was used and the values for the OPC tags listed in the OPC Item Path.



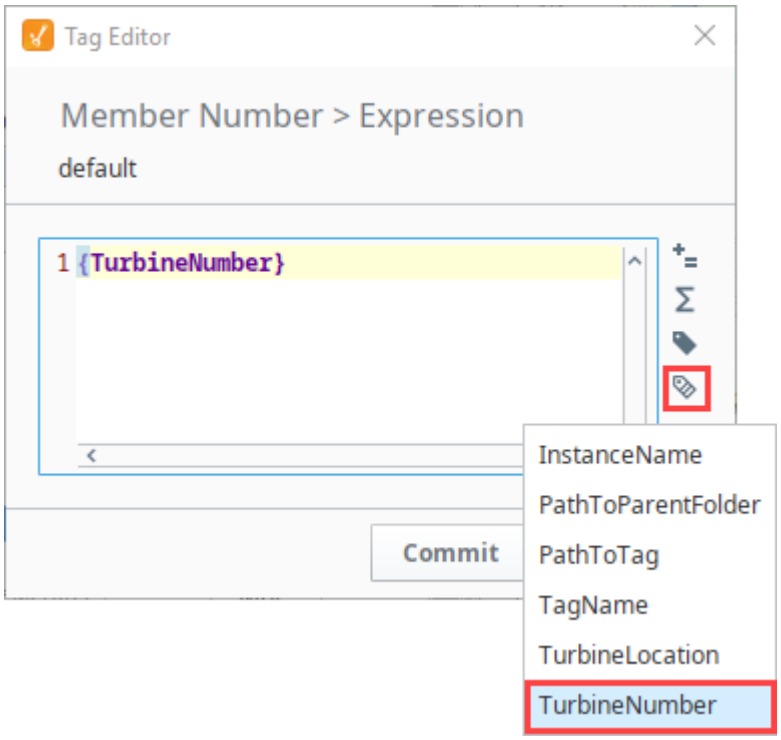
Data Type Parameters in Expressions

It is possible to use the value of data type parameters directly in expression bindings within a UDT. Parameter references can be quickly inserted into an expression.

While a UDT member is selected in the Tag Editor, you can edit bindable properties, such as the Expression on Expression tags by clicking the **Edit** icon next to the property.

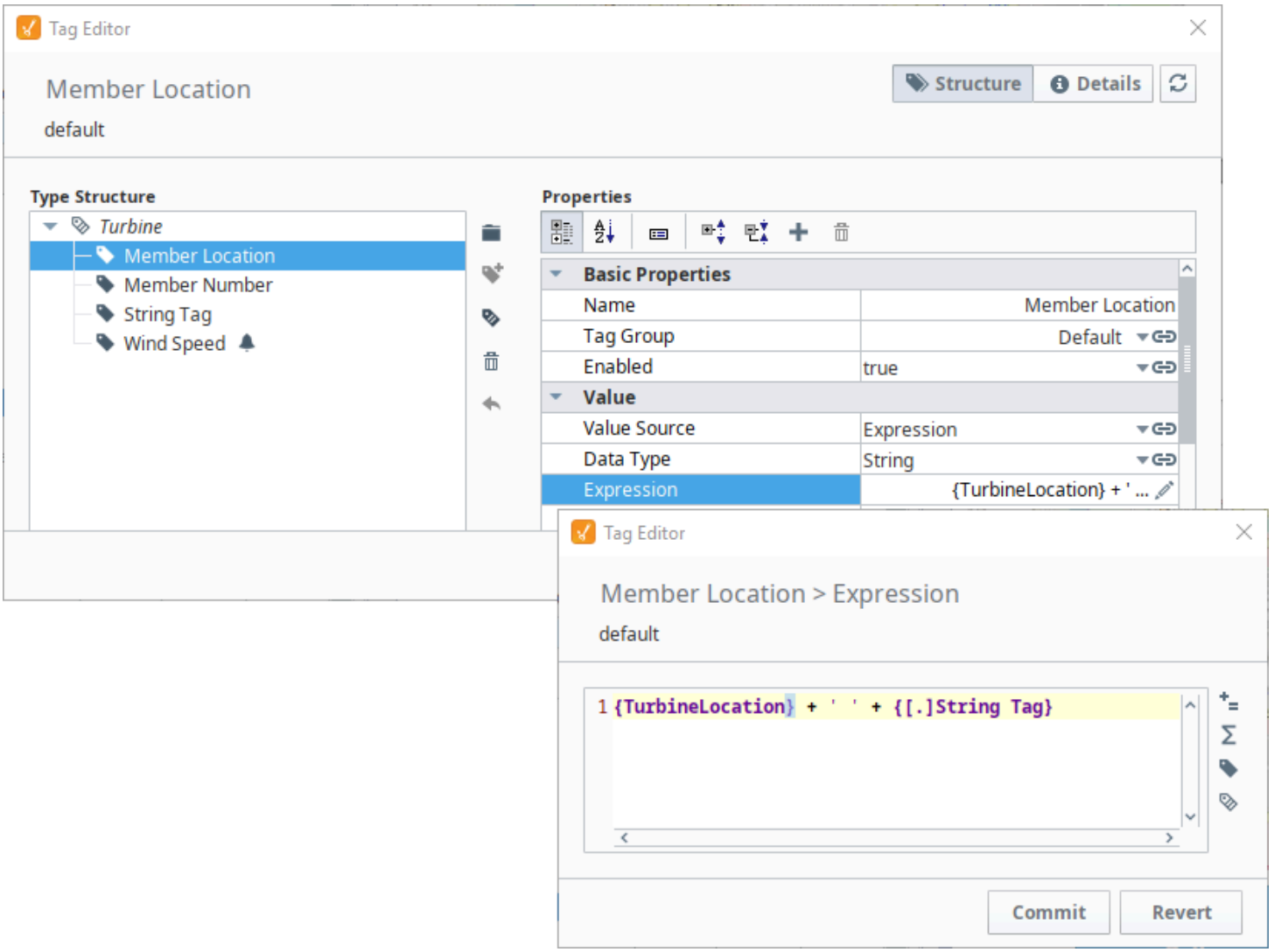


This opens the Expression window. Click on the **UDT Parameters** icon on the right of the expression area, select a parameter, then click **Commit**.

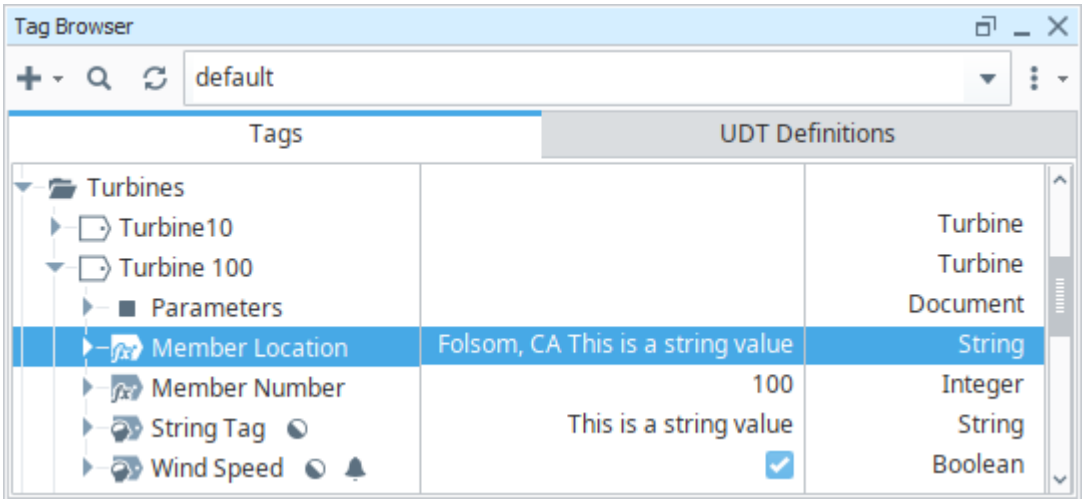


Combining Parameters and Tag References

Because parameter and tag references differ in syntax, some consideration must be made when attempting to use both in the same expression. Tag references must not be placed inside of quotes. After adding a string tag to the Turbine UDT, a reference to the tag can be added to Member Location's expression. Single quotes were added to create a space between the Member's Location and the string value.



Here's what it looks like in the Tag Browser.



Attribute Referencing and Parameterized Types

As mentioned above, many properties in the member tag configuration can reference the parameters available in the data type. When instances are created, these references are replaced with the values defined for the type. Parameter references also support basic offsets and numerical formatting, providing a great deal of flexibility. To reference a parameter, use the syntax `{ParameterName}`.

To offset a value, use the form `{ParameterName+offset}`. To format a value, use the form `{ParameterName|format}`. The format pattern is the same as that used for the `numberFormat` expression function. In short, "0" can be used to require a digit, and "#" can be used for optional digits. ie: **##0**

Example:

For this example, we'll assume that we've parameterized **OPC Item Path**, and that the data type has an integer attribute named `BaseAddress` defined. We'll pretend the **OPC Server** provides tags named like `DataPoint1`.

Standard Referencing

OPC Item Path: `DataPoint{BaseAddress}`

Offset

Imagine that our data type had three fields, and these were laid out sequentially in the device. Instead of specifying each address for each tag, we can simply offset from the base address:

Member 1: `DataPoint{BaseAddress+0}`

Member 2: `DataPoint{BaseAddress+1}`

Member 3: `DataPoint{BaseAddress+2}`

Formatting

Continuing from the example above, imagine that our OPC server actually provided addresses in the form `DataPoint001`, in order to stay consistent up to "DataPoint999". This can be accommodated using number formatting in the reference:

Member 1: `DataPoint{BaseAddress+0|000}`

Member 2: `DataPoint{BaseAddress+1|000}`

Member 3: `DataPoint{BaseAddress+2|000}`

This format of three zeros means "three required digits". If our instance has a base address of 98, the resulting paths will be `DataPoint098`, `DataPoint099`, `DataPoint100`.

Parameters support more mathematical operators in addition to offsets and formatting. There is a simple expression language available that can be used in conjunction with formatting. The following

table shows all available operators in their order of operations (they are evaluated starting at the top of the table).

Operator	Description	Example
()	Parenthesis. These operators are used for grouping any number of values. Also used to change the order of operations.	`{Baseaddress*(2+3)}`
^	Power. This operator is used to raise a number to a power.	`{BaseAddress^2}`
-	Negative. Used to create a negative value from one number.	`{BaseAddress*-2}`
*	Multiplication. Multiply two numbers.	`{BaseAddress*2}`
/	Division. Dividing the first number by the second number.	`{BaseAddress/2}`
%	Modulus. This operator returns the remainder of a division operation. IE: 7/3 = 2 with a remainder of 1, so 7%3 = 1	`{BaseAddress%2}`
+	Addition. Add two numbers.	`{BaseAddress+2}`
-	Subtraction. Subtract two numbers	`{BaseAddress-2}`
|	Used to define a formatting pattern. Patterns are defined with 0 and # characters.	`{BaseAddress|##0.00}`

Example

```
# This dynamic OPC Item path takes in three parameters to determine the tag path
ns=1;s=[DeviceName]Path/to/tag{BaseAddress+(ParamNum*Multiplier)|0000}

# The OPC Item path resolves to the following assuming the following values:
# BaseAddress = 5
# ParamNum = 8
# Multiplier = 2
ns=1;s=[DeviceName]Path/to/tag0021
```

Calculations and Numerical Parameter Names

If the parameter names are purely numerical values (we don't recommend this: it gets confusing), then quotation marks must encase the parameter to run any sort of calculations on the value of the parameter.

For example, if a UDT contains a parameter named 0, and its value is 10:

```
// This will evaluate to 0, because it thinks you mean the integer 0, not the parameter named "0"
{0 * 1000}

// This will evaluate to 10000, because the quotation marks denote a parameter named "0"
{"0" * 1000}
```

Nonexistent UDT Parameters

Parameter references in bindings that do not evaluate to an existing UDT instance parameter will be returned as a string. For example, say a binding on a member used the following:

```
SomeText/{myParameter}
```

If the UDT definition does not contain a parameter named "myParameter", then the binding above would be treated as a string literal, returning a value of `SomeText/{myParameter}`.

Null Values on Parameters

If the value of a UDT parameter is null, then any bindings that reference to the value of that parameter will instead return a string literal, similar to how referencing nonexistent parameters works. For example, say a UDT definition has the a parameter named "NullParameter". If the value of that parameter is null, then references to it would return the string "`{NullParameter}`". Thus, if we had a binding that included the parameters:

```
SomeText/{NullParameter}
```

The binding would return a value of:

```
"SomeText/{NullParameter}"
```

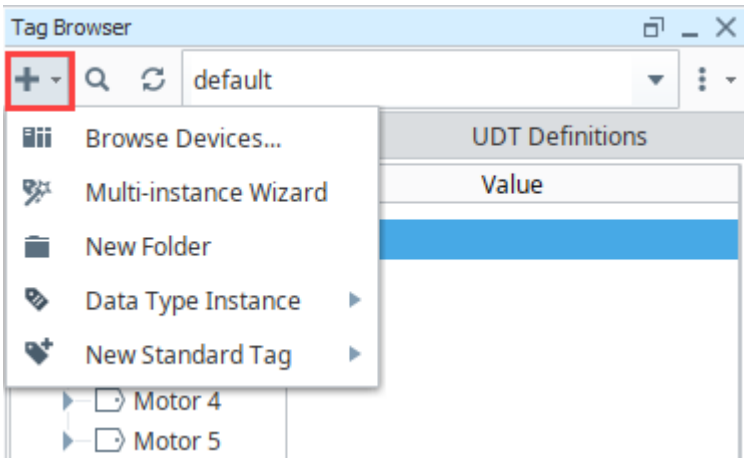
UDT Multi-Instance Wizard

The Multi-Instance Wizard provides a streamlined way to generate multiple instances of a UDT.

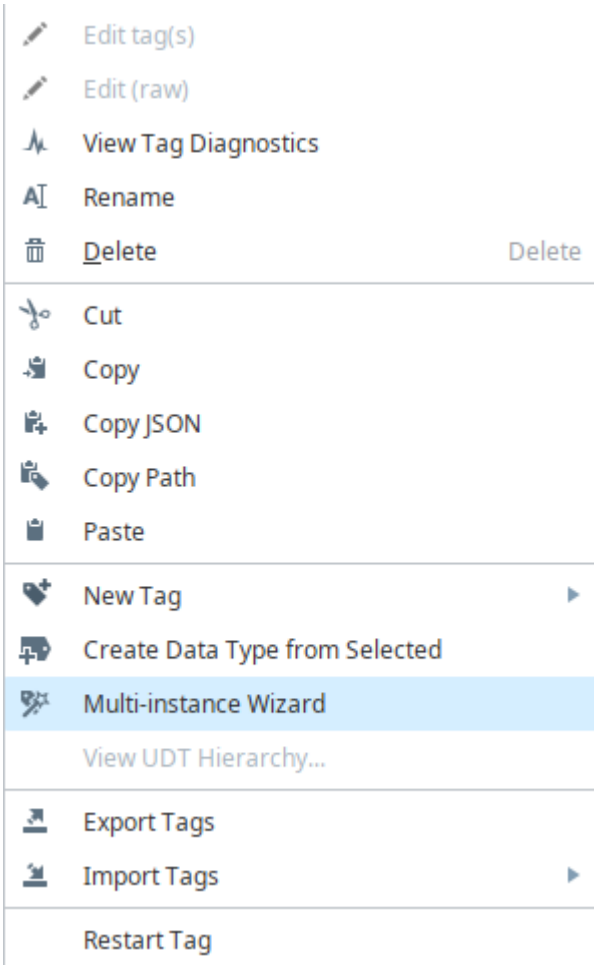
Accessing the Multi-Instance Creation Wizard

Users can access the Multi-Instance Creation Wizard through:

- The **Add** ✚ menu in the Tag Browser toolbar.



- The right-click menu in the Tag Browser tree.



Selecting the **Multi-Instance Wizard** option from either location will open the Multi-Instance Creation Wizard.

Instance Naming and Pattern Configuration

The Multi-Instance Creation Wizard has two tabs: **Create Instances** and **Configure Parameters**.

Create Instances

This section defines how the instance names will be generated and where they will be placed.

- **Data Type**: A dropdown selector used to choose the UDT definition to instantiate.
- **Folder Location**: A folder selector used to define where the new instances will be created in the Tag Browser.
- **Base Instance Name**: An input field used to define the base name for each instance. You can use reference tokens enclosed in curly braces to define dynamic naming patterns (e.g., `motor{x}`).

- **Reference Table**: When a reference is defined in the Base Instance Name using curly braces, a row is added to the reference table.

Column	Description
Reference	The name of the reference token (e.g., `x` in `{x}`).
Pattern Type	**Changed in 8.3.9** Defines the type of pattern to apply (Numeric, Alpha (Upper), Alpha (Lower), or List of Names). Prior to 8.3.9, List of Names was not an available Pattern Type.
Number Format	Optional format mask for numeric patterns (e.g., `#`, `##`, `000`).
Step Count	Defines the numeric step increment between values.
Start From	The starting value for the pattern.
End At	The last value in the sequence. Only applicable to numeric references.

Configure Parameters

If the selected UDT includes parameters, the **Configure Parameters** tab becomes available. By default, parameters inherit values from the UDT definition. To override them, you can enable and configure them using the **Parameter Table**.

Column	Description
Parameter	The name of the UDT parameter. Includes a checkbox to enable overriding the default value.
Type	The data type of the parameter.
Parameter Value	The value to assign. This field becomes editable when the parameter is enabled.
Pattern Type	Defines how the parameter value will be applied. Options include None, Numeric, and Manual Entry.
Step Count	Applicable to numeric patterns. Defines the interval between values.
Start From	The starting value for the numeric pattern.
Placement	Specifies whether the value should appear before or after the base string value (e.g., `1Motor` vs. `Motor1`). Applies to string concatenation.

Use this table to specify parameter values per instance or apply patterns dynamically.

Preview Table

The preview table displays a live preview of the instances that will be created. This includes:

- **Instance Name**: The full name of each instance as it will appear in the Tag Browser.
- `<parameterName>`: A column for each UDT parameter. Shows inherited or overridden values per instance.

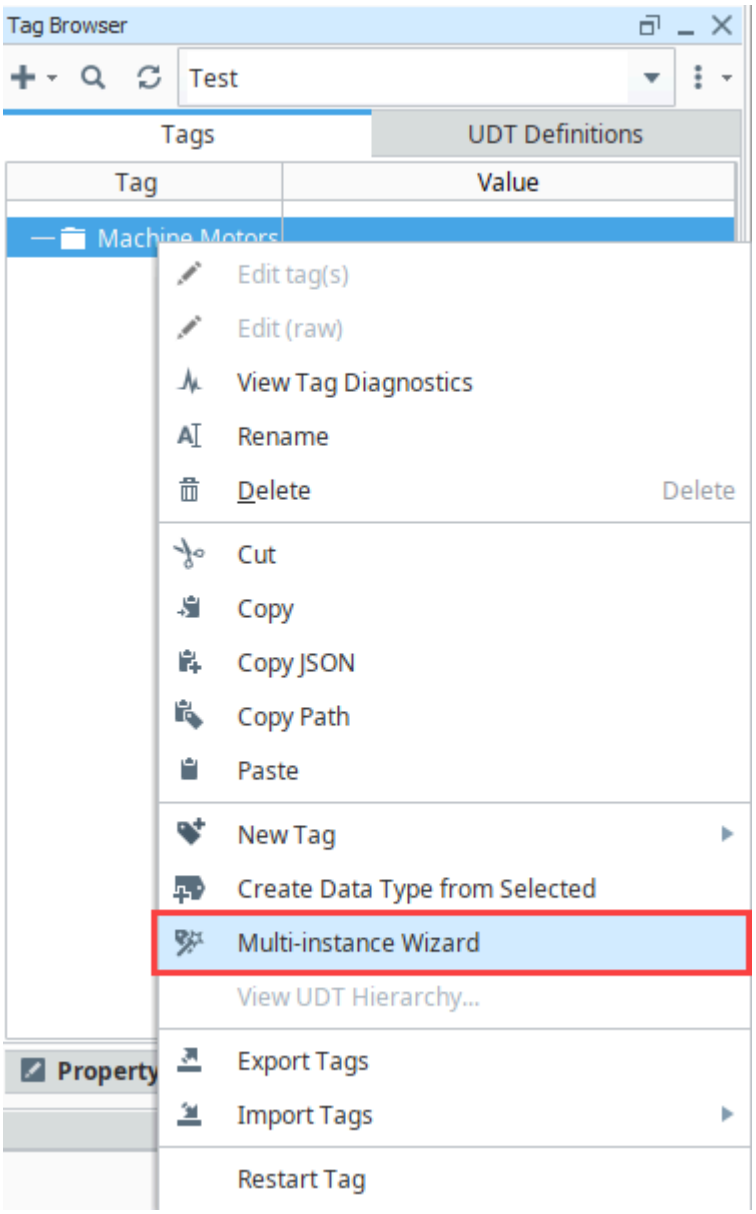
This preview updates in real time as you modify the base name, pattern configuration, or parameter values.

How to Make New Instances of a UDT

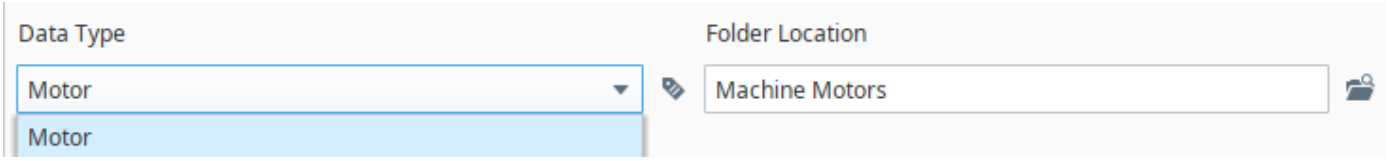
Once you have a UDT Definition created, you can make multiple instances of it with the Multi-instance Creation Wizard.

1. In the **Tag Browser**, click the Add ✚ icon to create a new folder. In this example, we created a folder named **Machine Motors**.

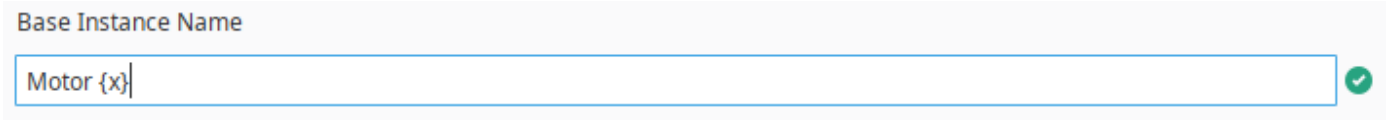
2. Right-click on the **Machine Motors** folder, and select **Multi-Instance Wizard** to open the Multi-Instance Creation Wizard.



3. Select a UDT (i.e., **Motor**) from the **Data Type** dropdown menu.



4. In the **Base Instance Name** field, enter:
   - `Motor {x}` (Include a space after "Motor" to separate numbers.)



5. In the Reference Configuration Table, configure the pattern:

- **Pattern Type**: Numeric
- **Start From**: `1`
- **End At**: `5`

Reference	Pattern Type	Number Format	Step Count	Start From	End At
x	Numeric	0	1	1	5

This configuration generates **five** instances:

- Motor 1
- Motor 2
- Motor 3
- Motor 4
- Motor 5

6. The Preview table dynamically updates, showing how instance names and parameter values will be applied.

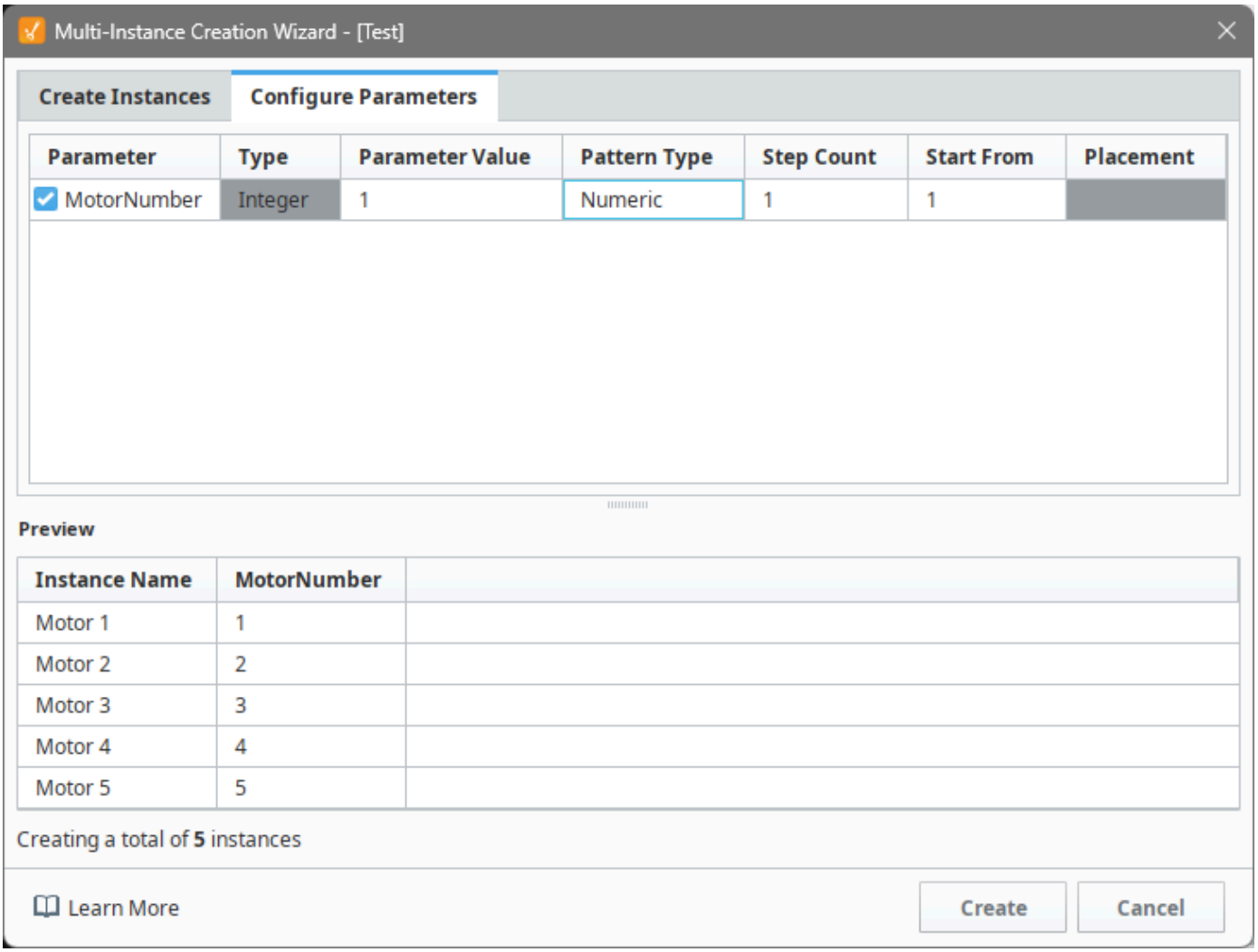
Preview

Instance Name	MotorNumber	
Motor 1	1	
Motor 2	1	
Motor 3	1	
Motor 4	1	
Motor 5	1	

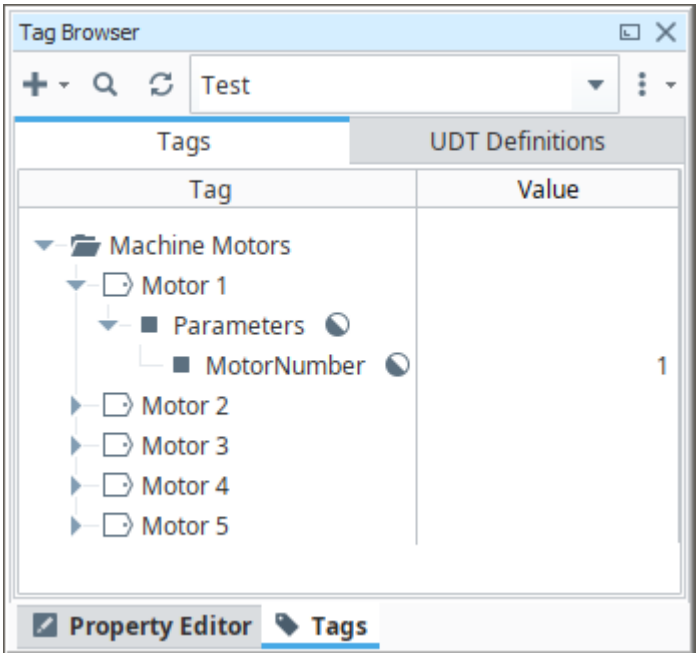
Creating a total of **5** instances

7. If the selected UDT has parameters, the **Configure Parameters** tab will be visible. To configure:

- Check the parameter to enable configuration.
- Choose a Pattern Type (Numeric).
- Enter a custom value or pattern.



8. Verify the created instances in the **Tag Browser**.



NOTE

The **Multi-Instance Creation Wizard** cannot edit existing instances. It is only for creating multiple instances of a UDT. To modify instances, see How to Edit an Existing UDT.

UDT Inheritance

Once you have a single data type created, it is possible to set up UDT inheritance where data types extend to other data types, to add additional members, or override default values. For example, you can create a new data type and using the inheritance feature it will inherit all tags from the parent data type including the parameters. Then you can add additional tags and/or override any settings in your new data type. UDT Inheritance is a way to extend to a class of data types to add more functionality to that class.

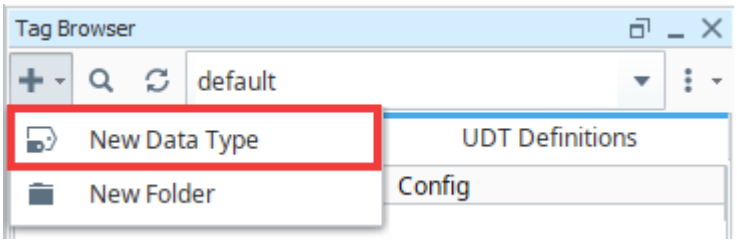
For example, you may have a simple motor and a complex motor. The complex motor can inherit from the simple motor, which means all simple motor values will be in the complex motor and you can add more.

Nesting (using one or more UDTs to make up a larger UDT) is different from inheritance and can be found under **UDT Nesting**.

Inheriting Property Values from an Existing UDT

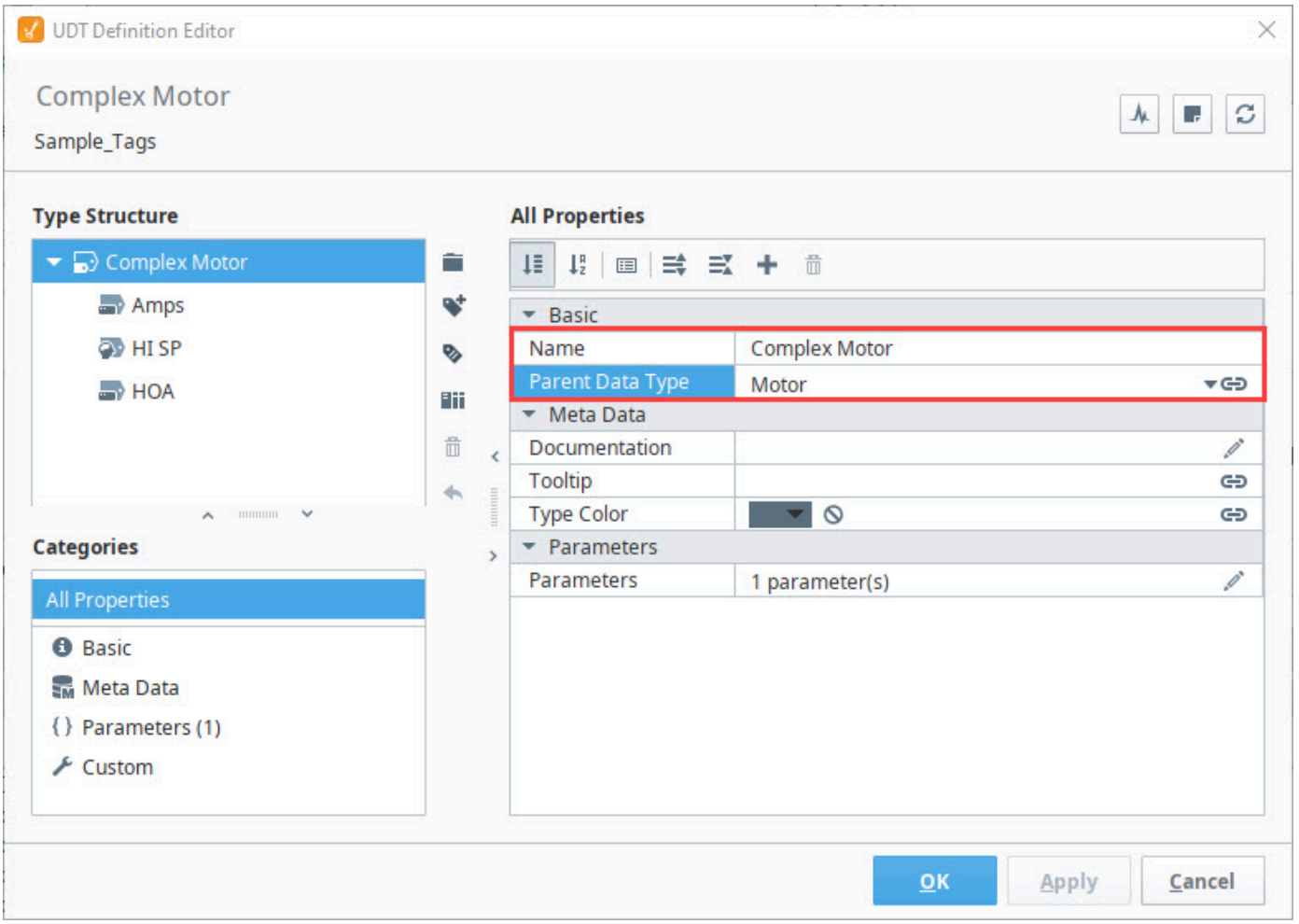
Let's use our data type Motor from the previous sections to create another data type. We'll set the parent to Motor so our new data type automatically inherits all the properties of Motor.

1. In the **Tag Browser**, click on the **UDT Definitions** tab and click the Add ✚ icon to create a **New Data Type**. The Tag Editor window will open.

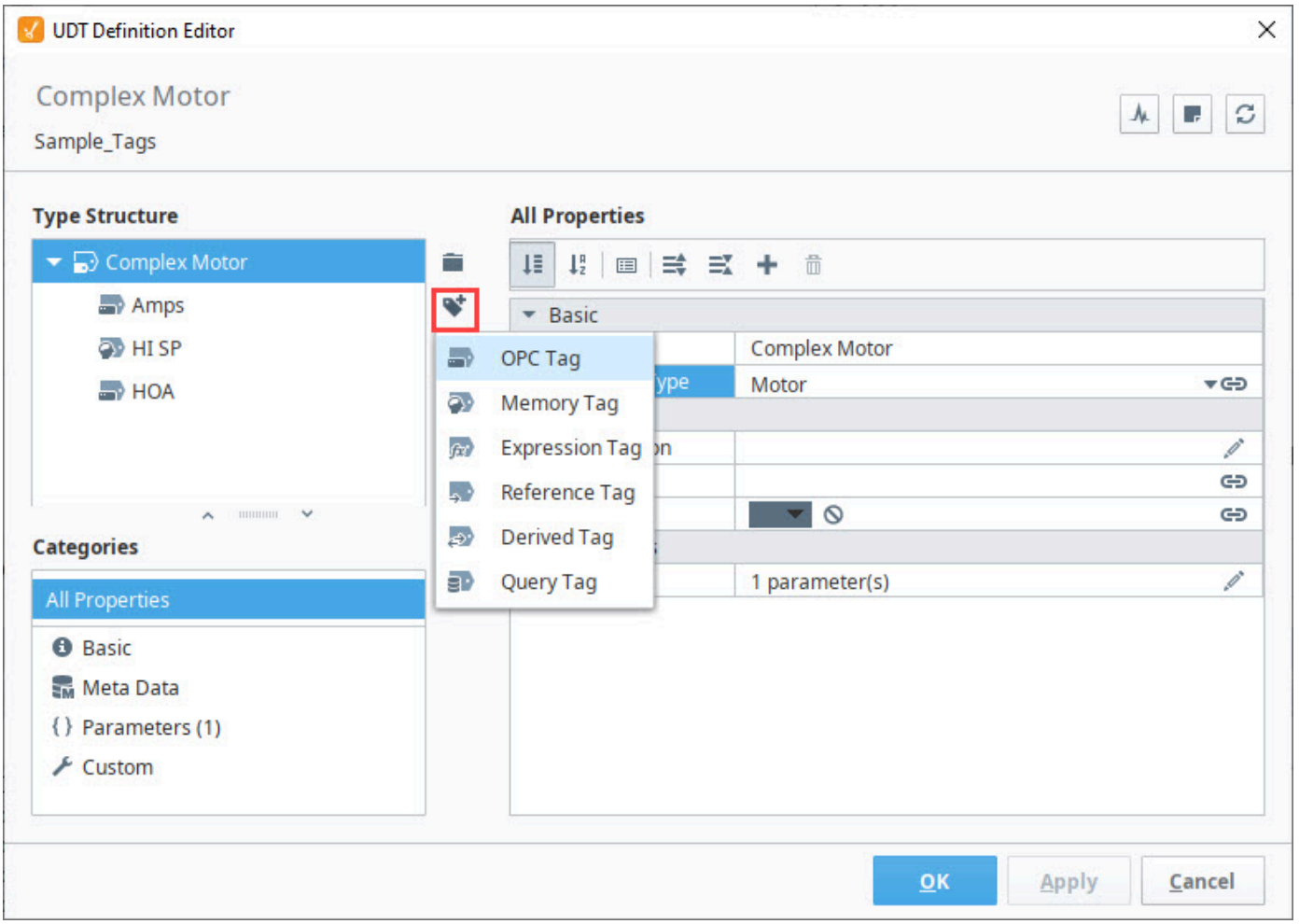


2. In the **Name** field, enter name for your new UDT (i.e., Complex Motor). Under **Parent Data Type** property go to the dropdown and select the **Motor** data type and click **Apply**. Now your new

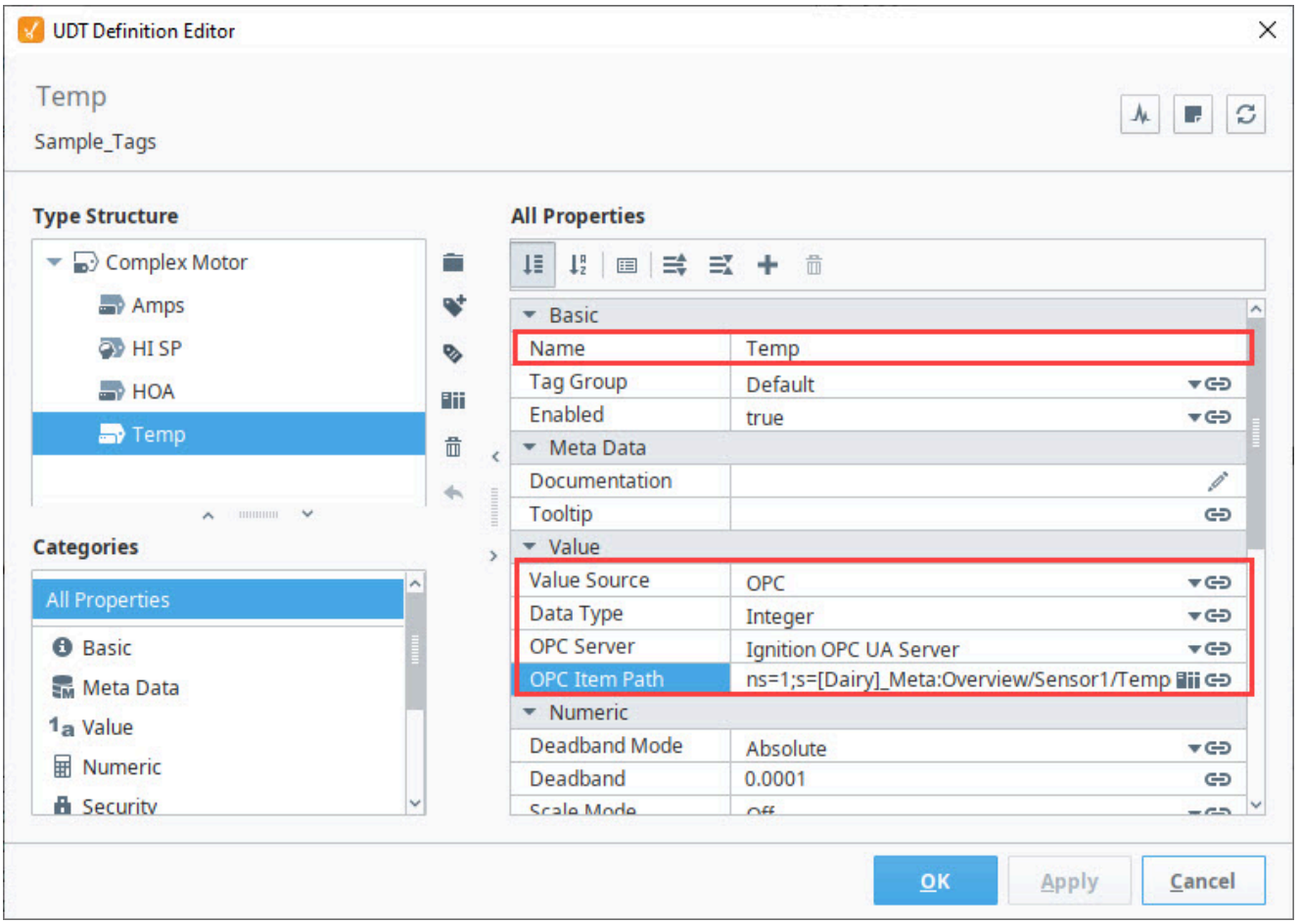
Complex Motor UDT is inheriting from all the properties of the parent Motor UDT: Amps, HI SP and HOA.



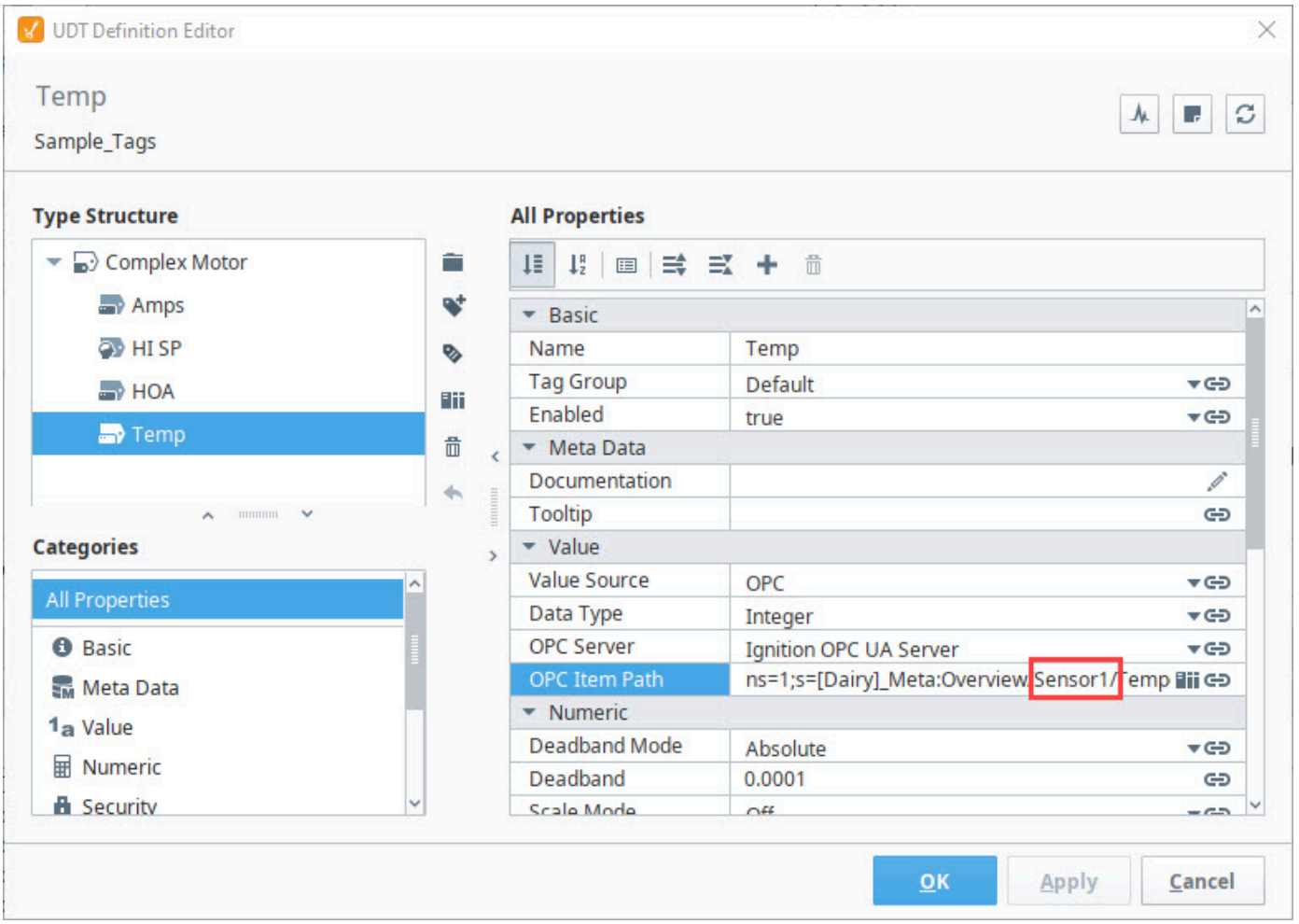
3. With the Tag Editor still open, let's add an **OPC Tag** to the Complex Motor UDT. Click on the **Add Tag** and select **OPC Tag**.



4. Enter the following properties for your new tag and click **Apply**. You will see the new tag was added to the Complex Motor UDT.
    - **Name**: Temp
    - **Value Source**: OPC
    - **Data Type**: Integer
    - **OPC Server**: Click on the binding (🔗) icon and select **Ignition OPC UA Server**
    - **OPC Item Path**: Browse the **OPC** and find the Tag you want to use. This example uses a **Temperature** Tag from a Sensor in the Dairy program.

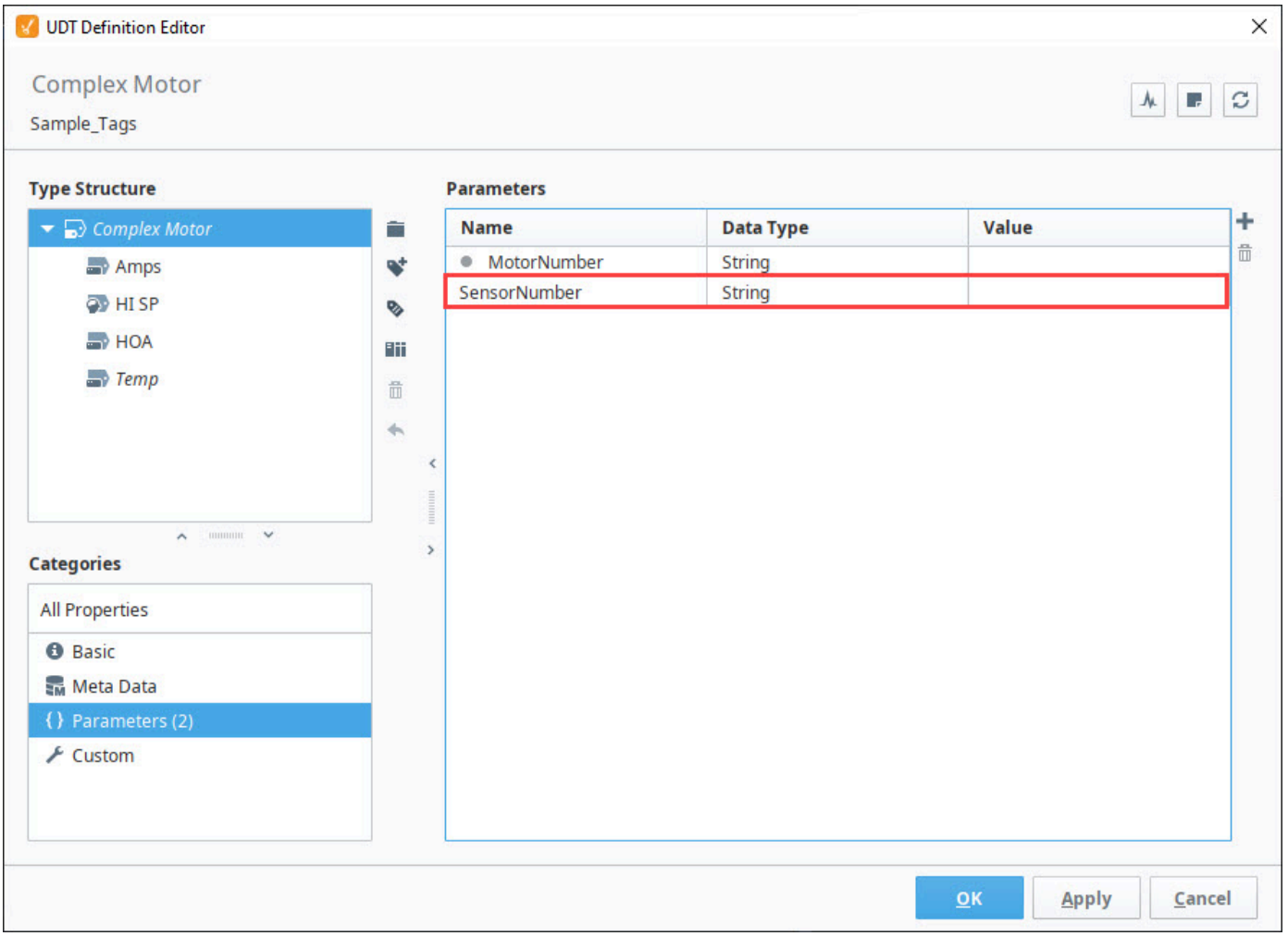


5. In the image below the Temp tag is pointing to a specific address in the PLC. Because we're creating a new tag in our UDT, we don't want to point to one specific set of **'Temp'** tags. We want each instance of the Complex Motor UDT to reference a different set of **'Temp'** tags. To do that, we need to add a parameter to the Complex Motor data type that we will call **'SensorNumber'**.

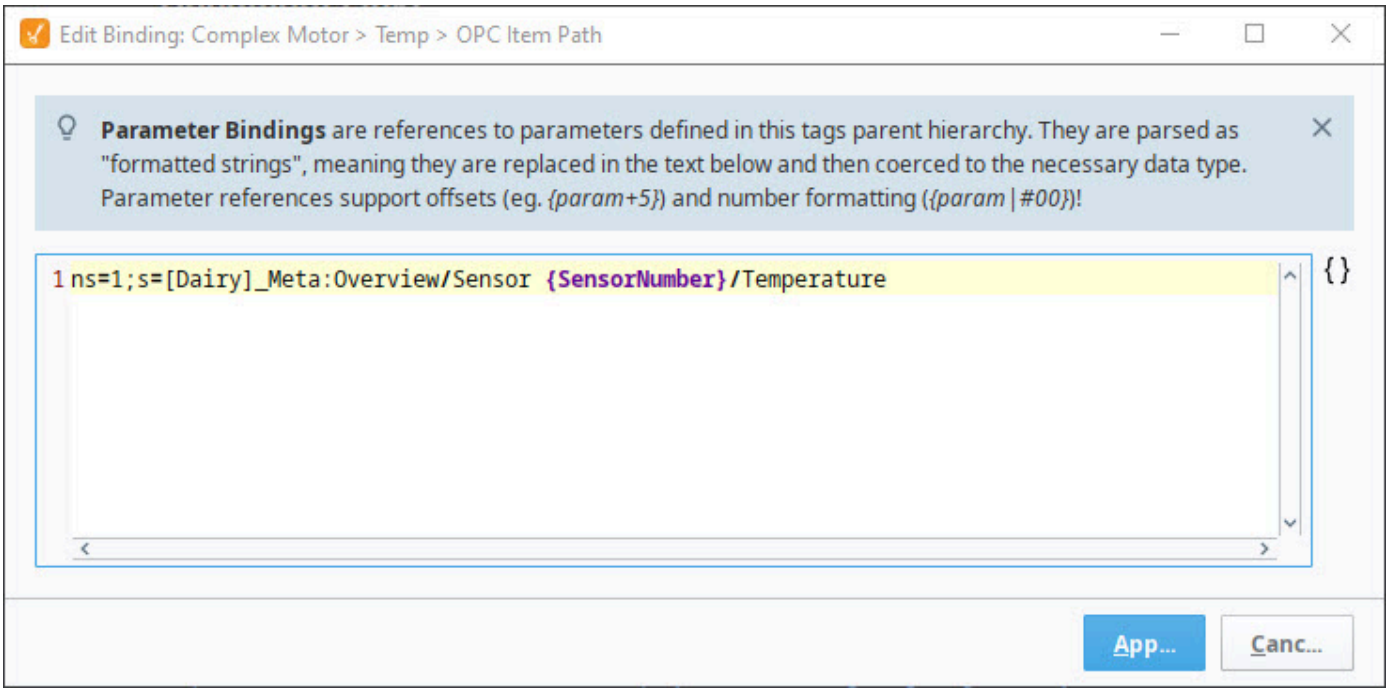


6. With the Complex Motor UDT **Tag Editor** open, let's create a new UDT parameter. Right click the **Edit** icon next to the **Parameters** property. The Parameters pane will open.

7. Click the Add icon and add the new parameter, **'SensorNumber'**.



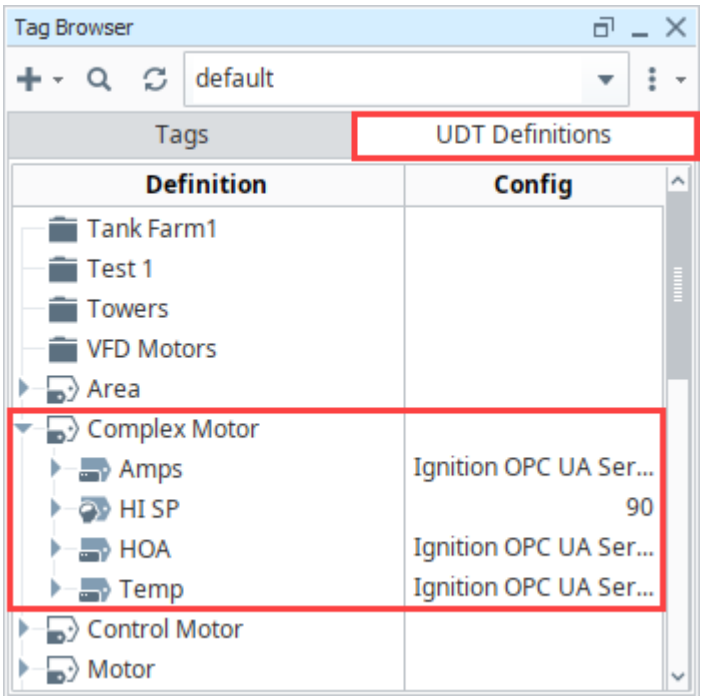
8. With the Tag Editor still open, select the **'Temp'** tag. In the **OPC Item Path** field, click the **binding** icon, select **Edit**, and the **Temp > OPC Item Path** window will open. Place your cursor at the end of **'Sensor1'**, delete the **'1'**, add a space, and enter `{SensorNumber}`. Don't forget the curly braces. Click **Apply** to save your updates and go back to the previous window.



You may also edit tag paths inline in the Tag Editor rather than opening the OPC Item Path window. For this example, you would highlight '1' in the OPC Item Path and replace it with the

new parameter, `{SensorNumber}`.

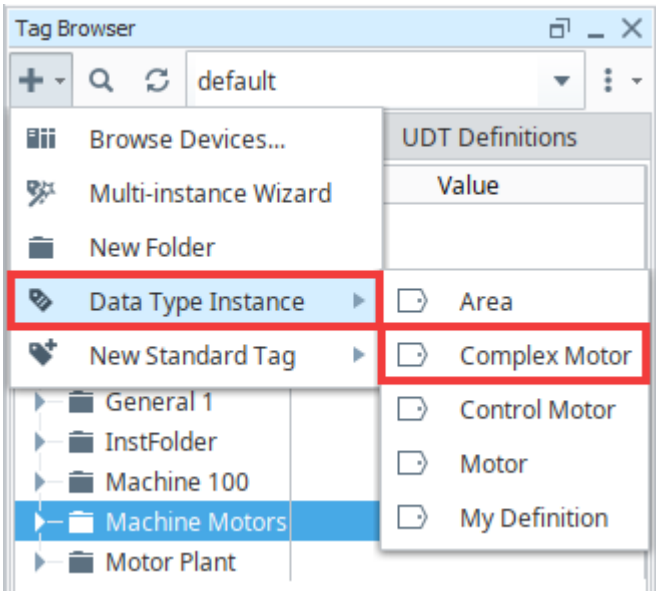
9. In the **Tag Editor** window, click **OK** to save your UDT. In the Tag Browser, the new data type is now visible in the , "Complex Motor". You can see all the inherited tags from the Motor data type and the newly added Temp tag.



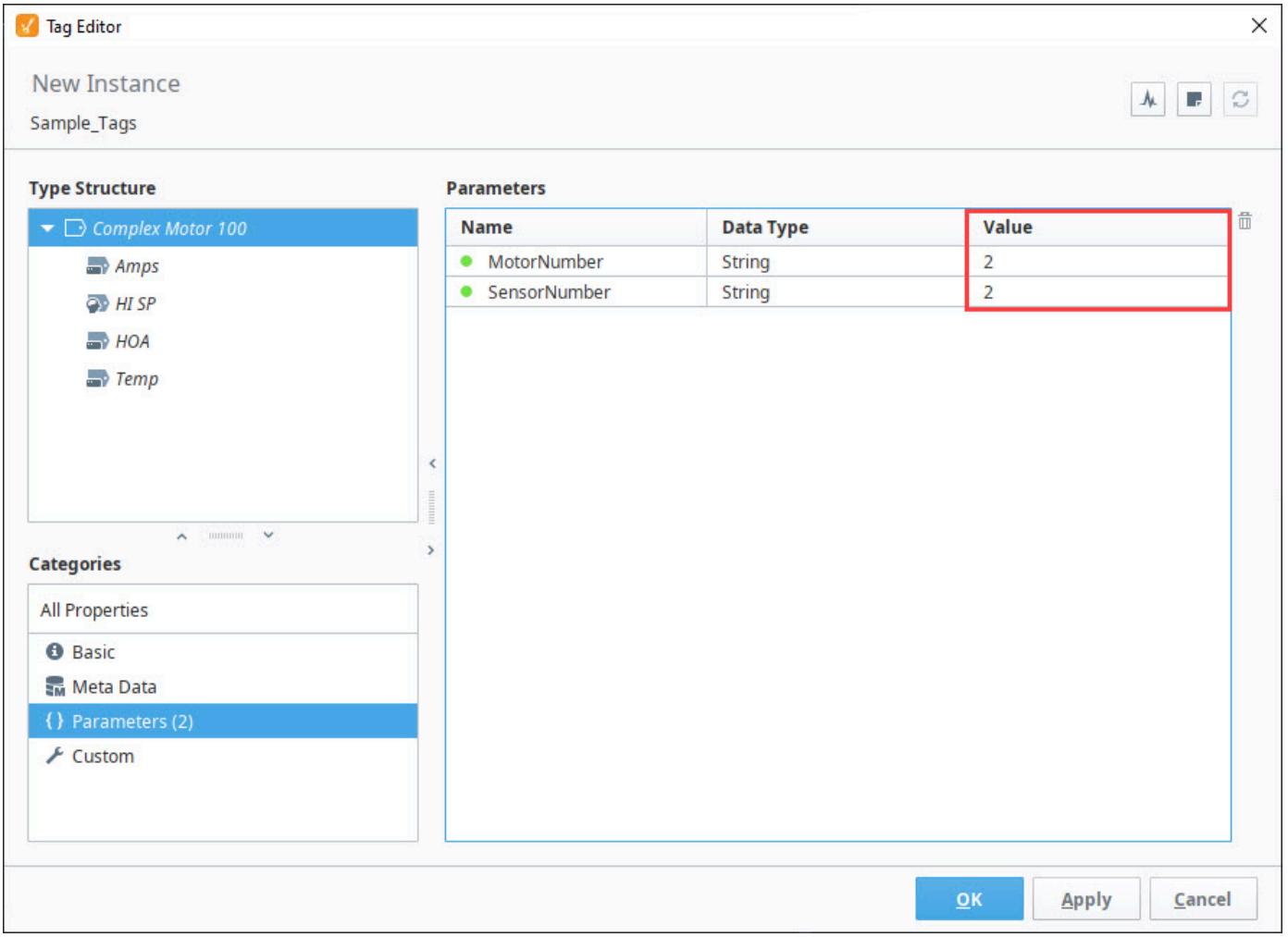
Creating the Data Type Instance

Now that our Complex Motor UDT is set up, let's create a data type instance of the Complex Motor.

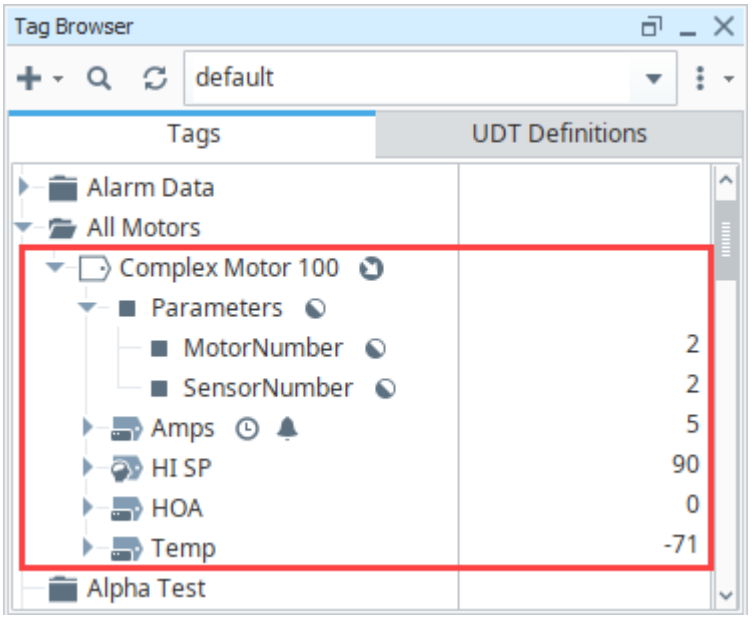
1. Click on the **Tag** tab, and from the context menu, select the **Data Type Instance** > **Complex Motor**.



2. Enter the **Name** for the instance (i.e., Complex Motor 100). Click the **Edit** icon next to the **Parameters** property and enter the parameter value of '2' for **MotorNumber** and **SensorNumber**.



3. Click **OK** to create the instance. Now, you'll be able to see all the values for Complex Motor 100 including the Temp Tag that was added.



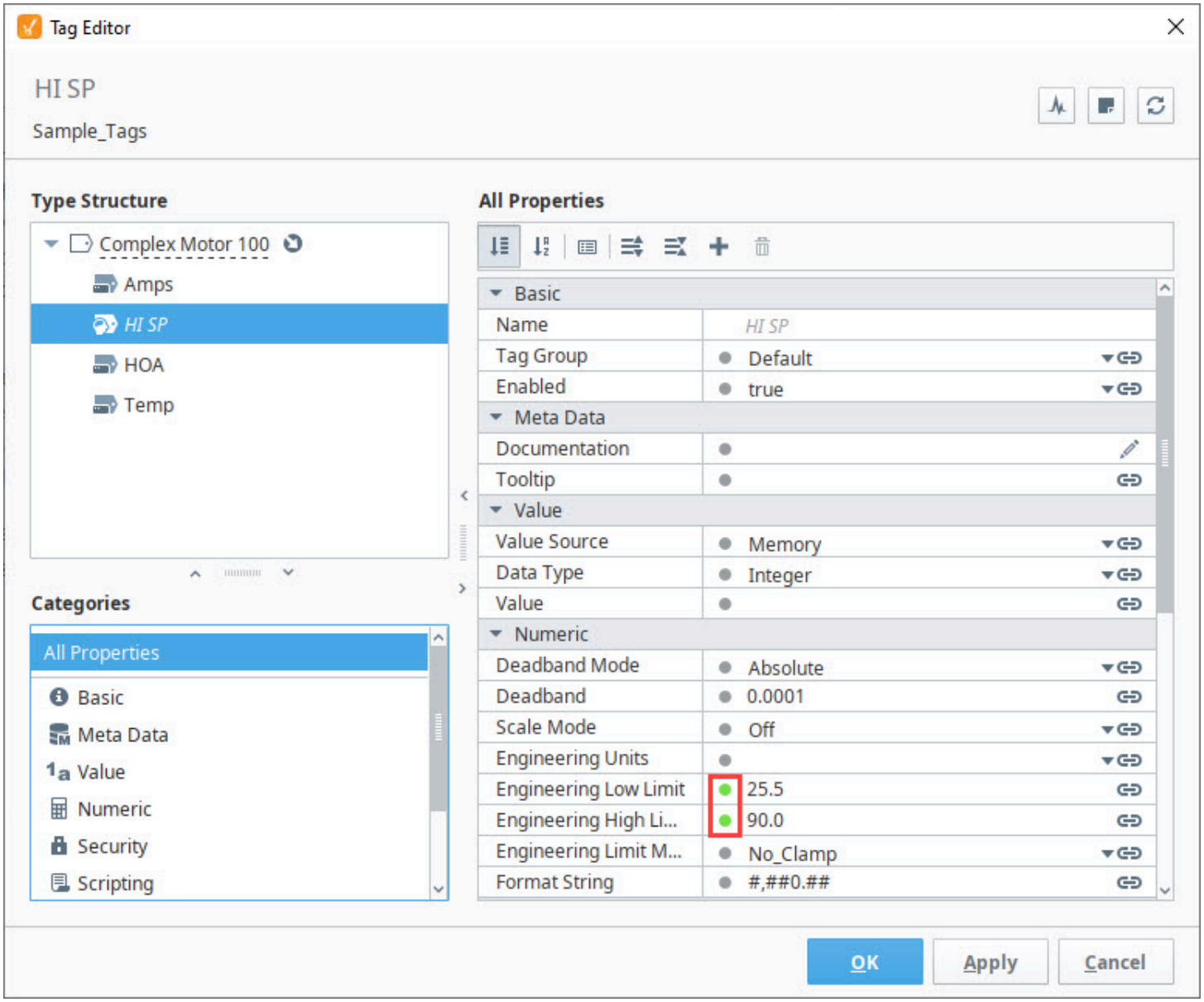
Overriding Properties of the Parent UDT

INDUCTIVE UNIVERSITY

Overriding Properties in UDT Instances

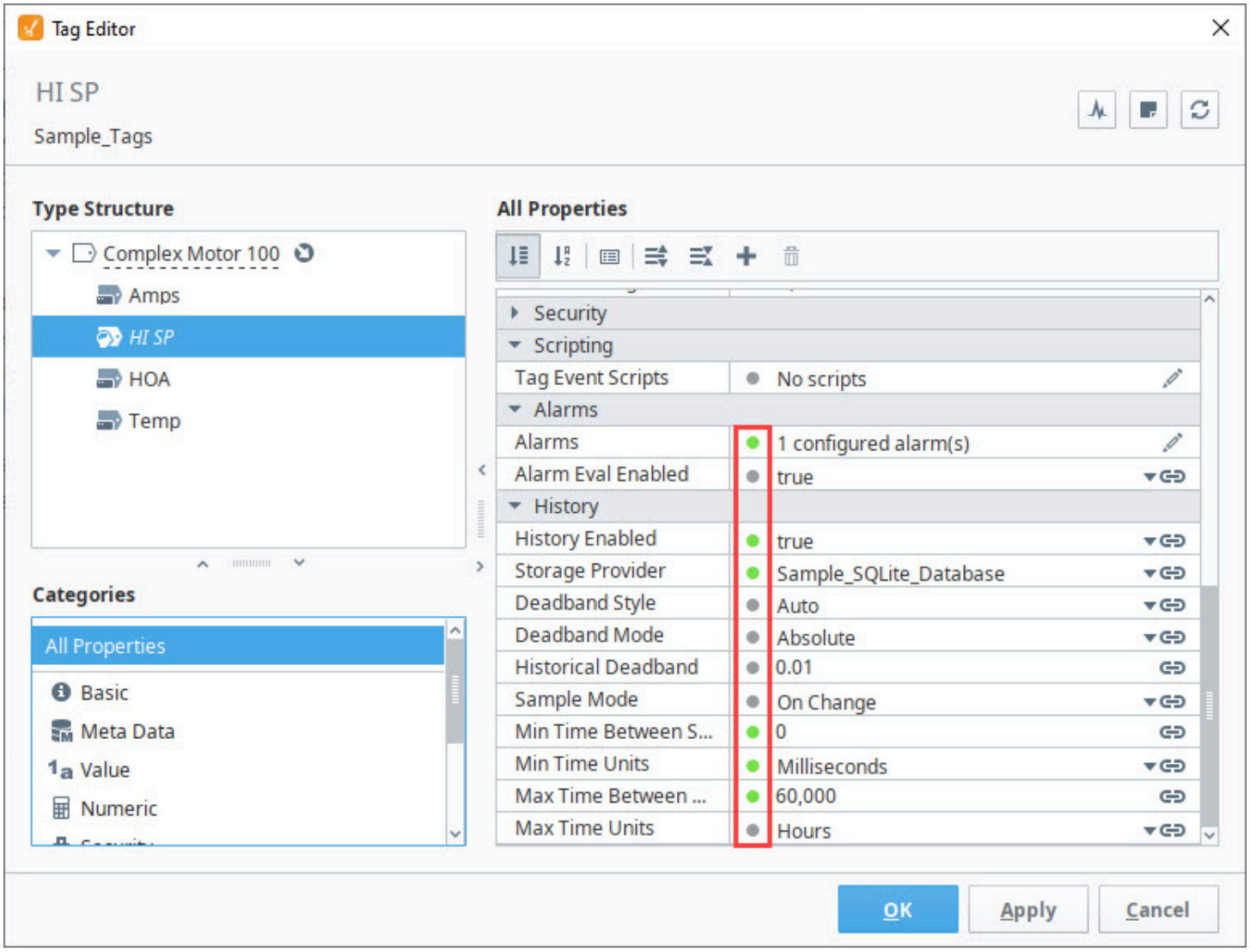
Watch the video

Another benefit of UDT inheritance is it allows you to override some of the properties of the parent. For example, since the Complex Motor has Motor as the parent, you can go to any of the tags and override any of the settings of that data type. Click the circle to the right of the property and enter a new value, or change a property's value and the green circle changes to green automatically. This overrides the property inherited from the parent.



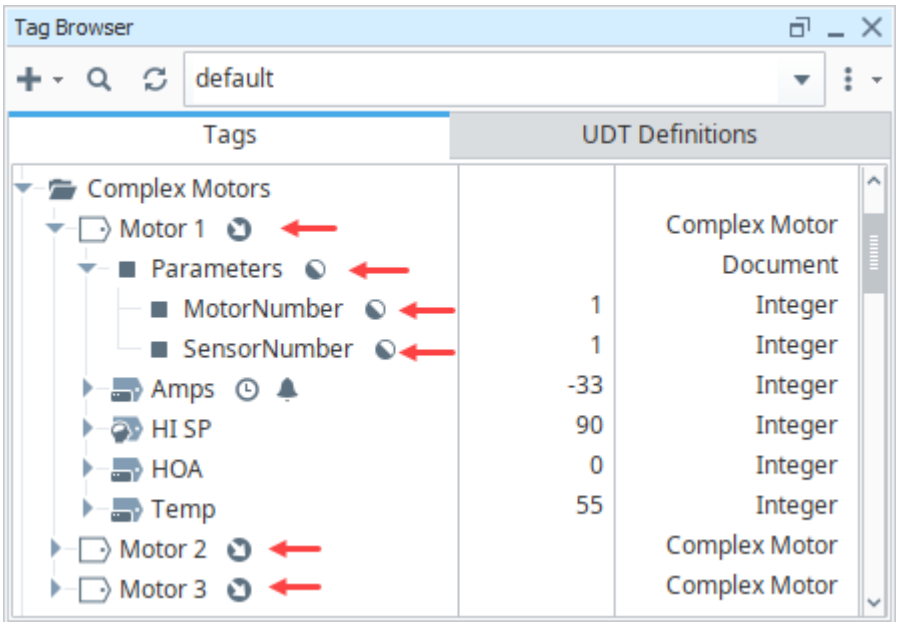
You can also turn on Alarming and History that wasn't initially turned on in the parent UDT by simply using the override feature. Next to the Alarm property, click the green circle to change it to green, and

click the pencil ( ) icon to configure the alarm if it is not already configured. If you want to turn on History, click the green circle or change any of the History properties which will cause any of the green circles to change from gray to green.



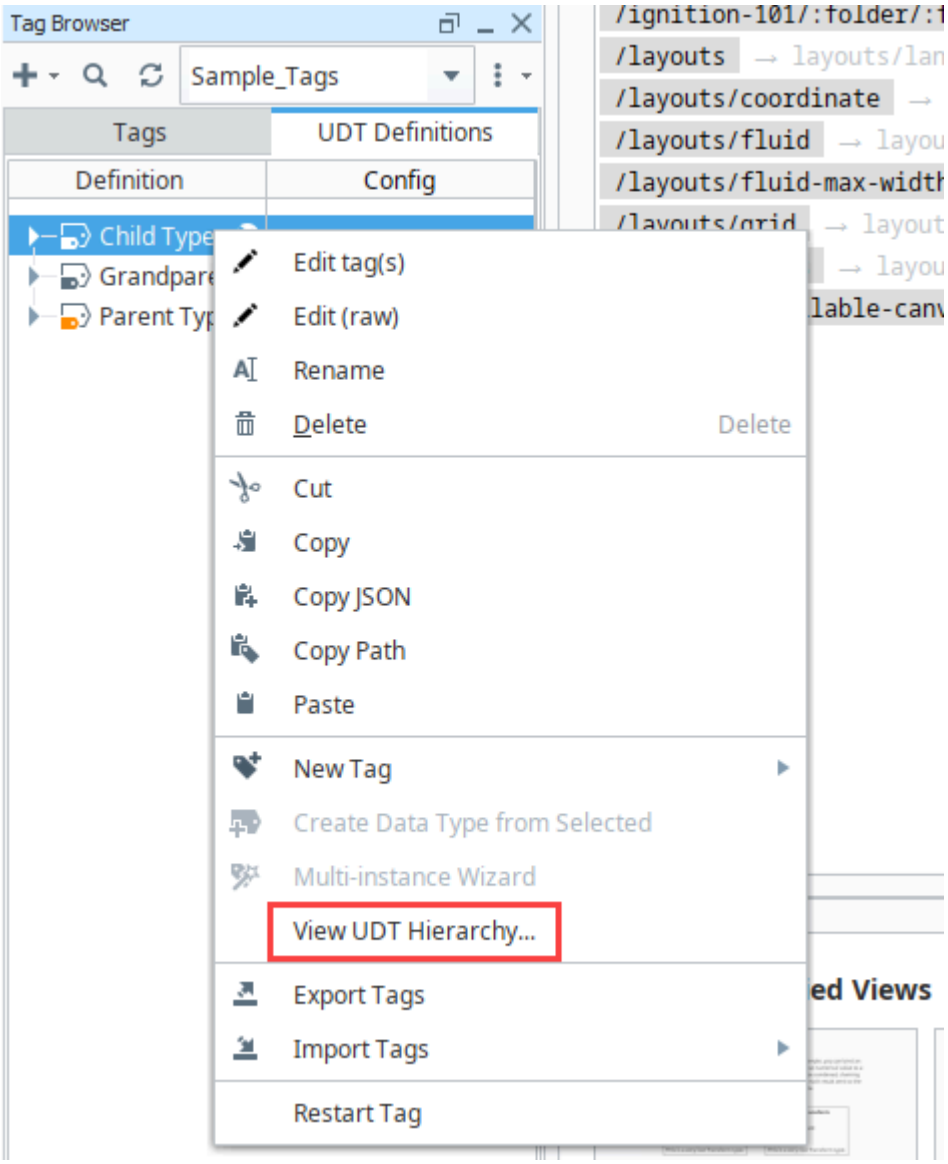
UDT Inheritance Traits

UDT instances that are inherited and have properties that override the parent properties have a visual representation next to them.



UDT Hierarchy Tool

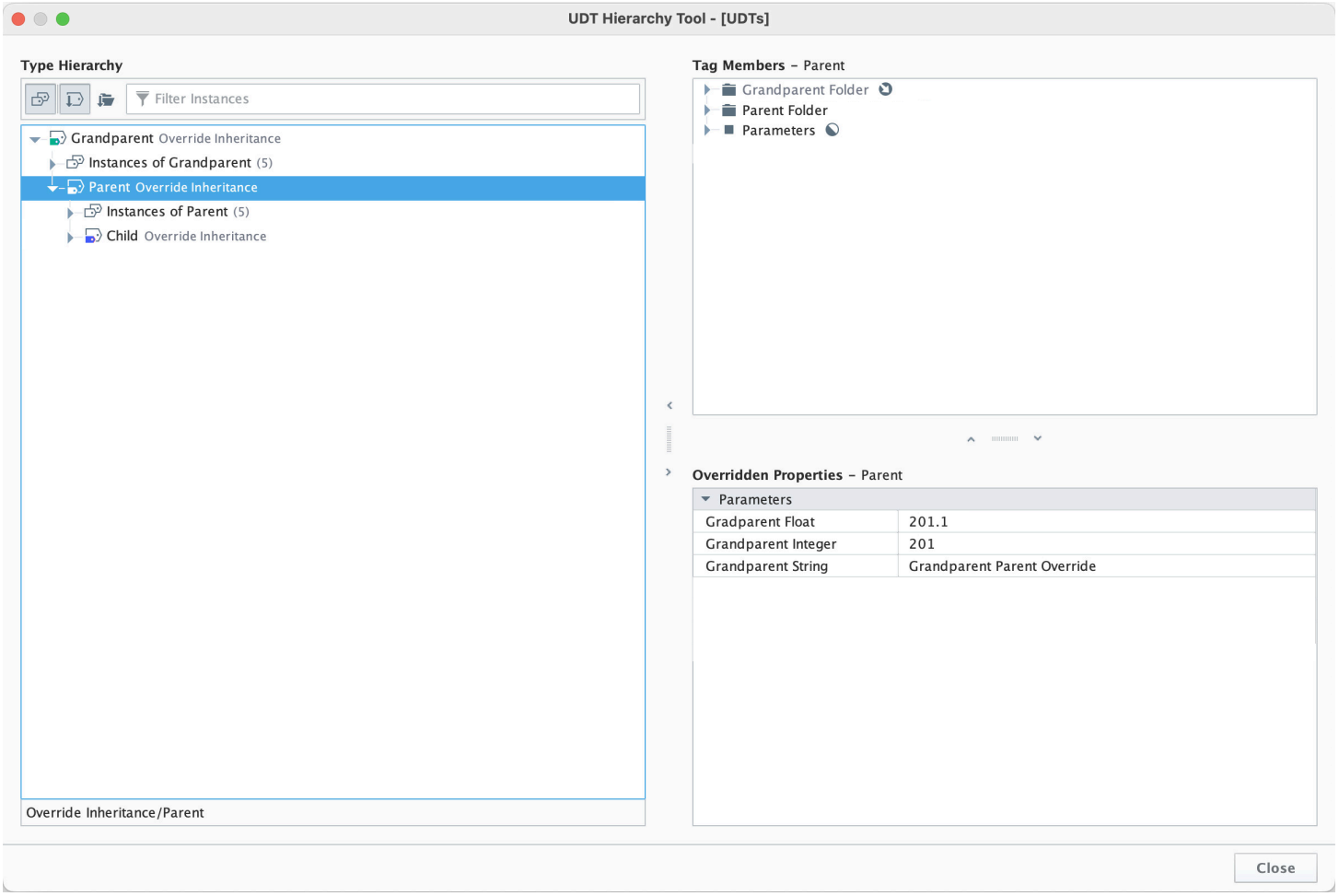
The UDT Hierarchy Tool displays the relationships between connected UDT definitions and instances. It shows inheritance between UDT definitions, overrides of inherited properties, and instances directly related to selected definitions. Right click a UDT Definition or UDT Instance and select **View UDT Hierarchy** to access the UDT Hierarchy Tool.



The Type Hierarchy panel shows the related definitions. When you have the Show Instance test one two icon selected, you can sort instances by typing into the filter bar and choose between grouping by folder path or name with the respective Sort Instances by Path or Sort Instances by Name icons. If you expand the **Instances of** folders, you will see UDT instances that directly inherit the definition as a parent. Note, nested UDT Instances are not visible in this list.

If you need additional filtering for instances, such as specific overrides, right click the **Instances of** text for the specific UDT instances to open the **Tag Report Tool** with the search pre-populated for that specific set.

Information about member tags and overridden properties for the selected UDT Definition or UDT Instance are visible on the right side of the tool. As shown in the example below, the inherited tag member types will be grayed out and followed by the Inherited icon.


UDT Hierarchy Tool - [UDTs]
Type Hierarchy
Filter Instances
Grandparent Override Inheritance
Instances of Grandparent (5)
Parent Override Inheritance
Instances of Parent (5)
Child Override Inheritance
Override Inheritance/Parent
Tag Members – Parent
Grandparent Folder
Parent Folder
Parameters
Overridden Properties – Parent
Parameters
Gradparent Float
201.1
Grandparent Integer
201
Grandparent String
Grandparent Parent Override
Close

UDT Nesting

It's possible to set up UDT nesting in Ignition where you are putting one UDT inside of another UDT. The UDT is nested as an instance within another UDT. It facilitates quicker development of projects since you're able to piece together multiple UDT definitions as needed without having to build everything from the ground up as with each UDT definition. This is particularly useful because it promotes rapid development if you are expanding a plant or facility where all you have to do is make a few Tag changes to existing parameters and property settings.

For example, you may have a production line that is built out of several different machines. You don't need to re-create a motor for each line, instead you can create it once and use it in every line.

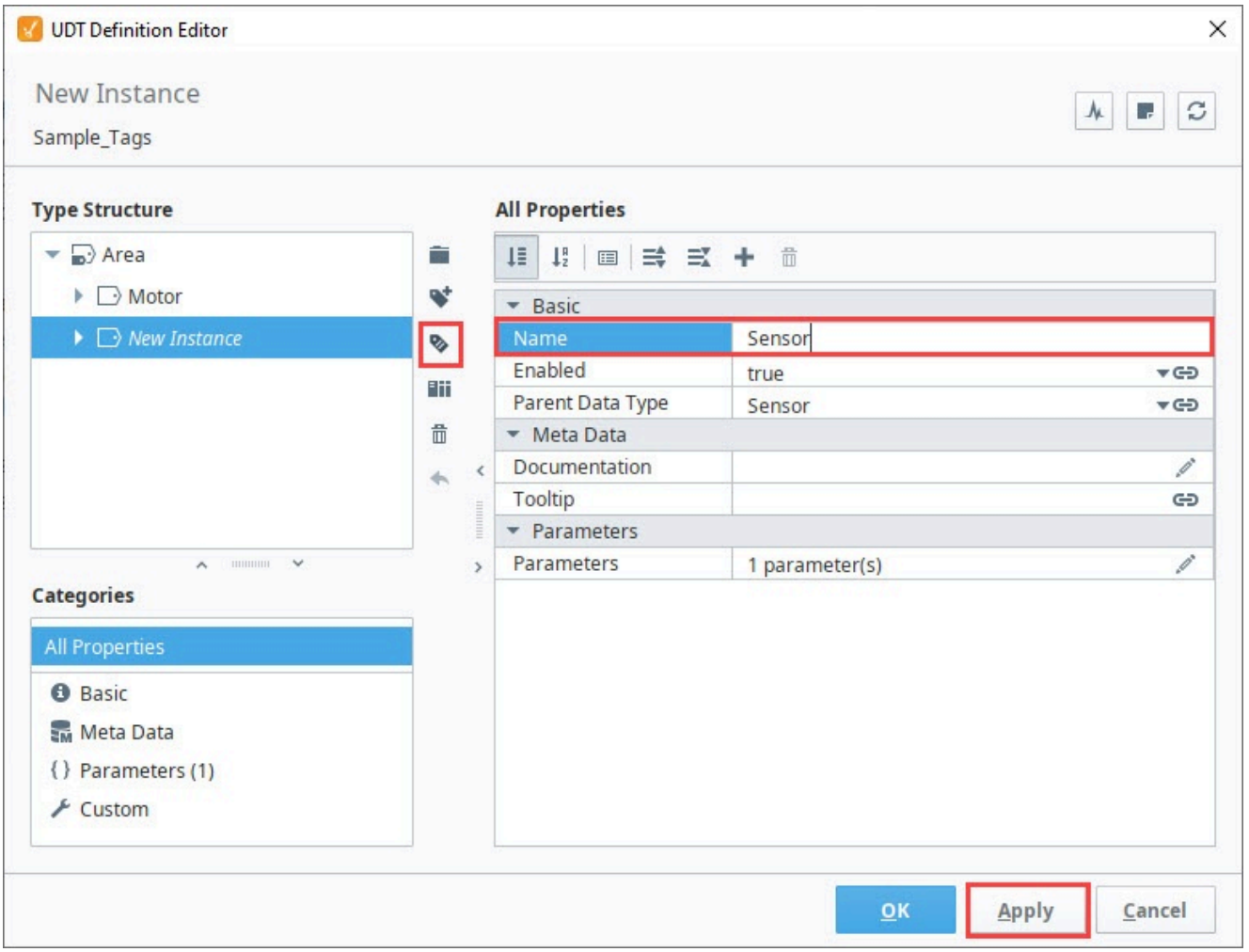
Inheritance (having simple and complex version of similar objects) is different from nesting UDTs, and can be found under **UDT Inheritance**.

Set Up UDT Nesting

In this example, let's use our Motor and Sensor data types that were created and used in previous sections of this manual. We are going to create a third UDT called Area that will contain the Motor and Sensor data types inside of it.

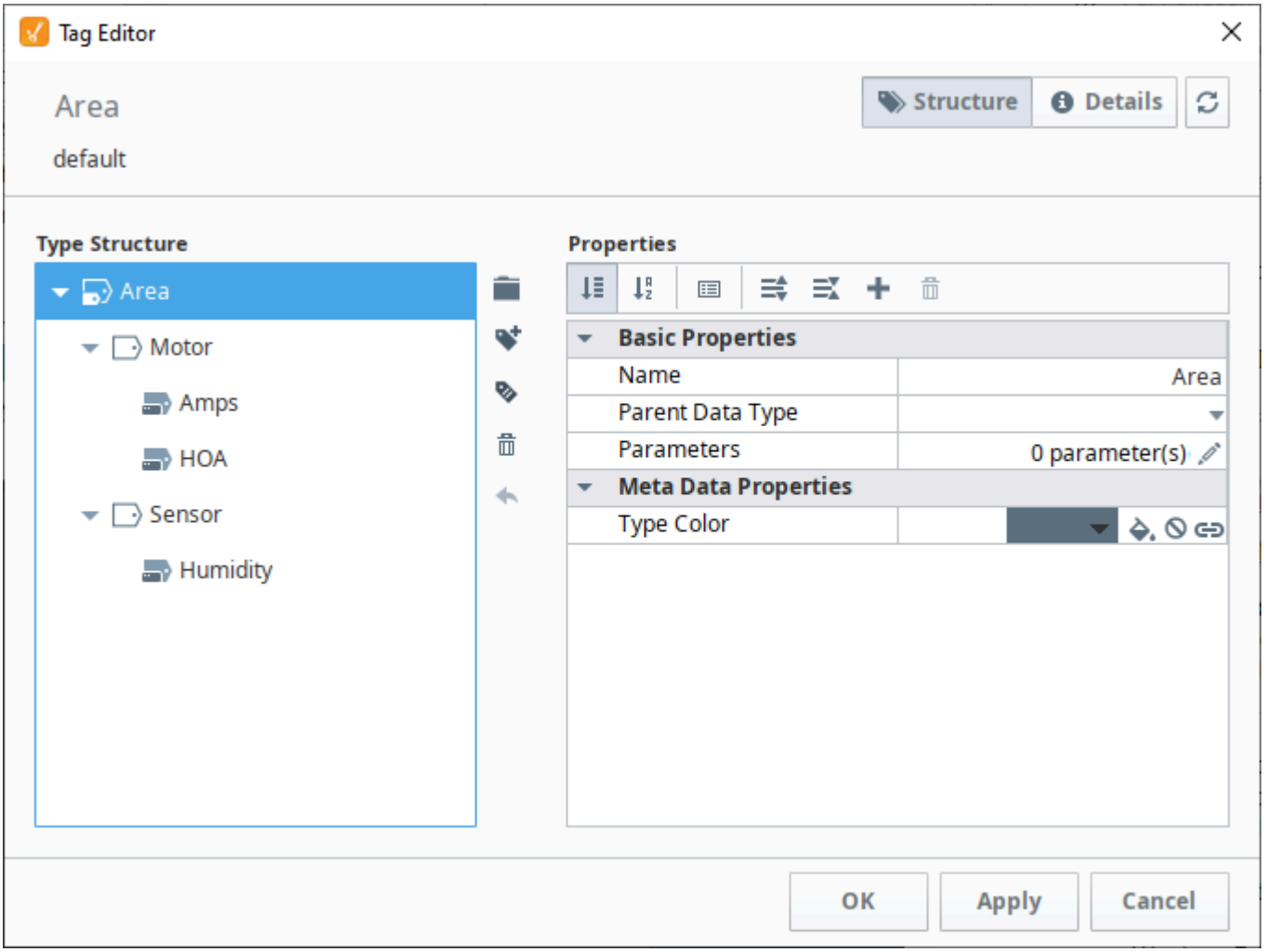
1. In the Tag Browser, right click on **Data Types** and select **New Tag > New Data Type**. The Tag Editor window will open. Assign the new data type a Name called **Area**.
2. Inside of the **Area** data type, create two data type instances, one for Motor and the other for Sensor:
   - Click on the **New UDT Instance** icon, select **New UDT Instance > Motor/Motor1**. Click **OK**.
   - Rename the new instance 'Motor'. Click Apply.
   - Click on the **New UDT Instance** icon, select **New UDT Instance > Sensors/Sensor**. Click **OK**.

- Rename the new instance **'Sensor'**. Click **Apply**.

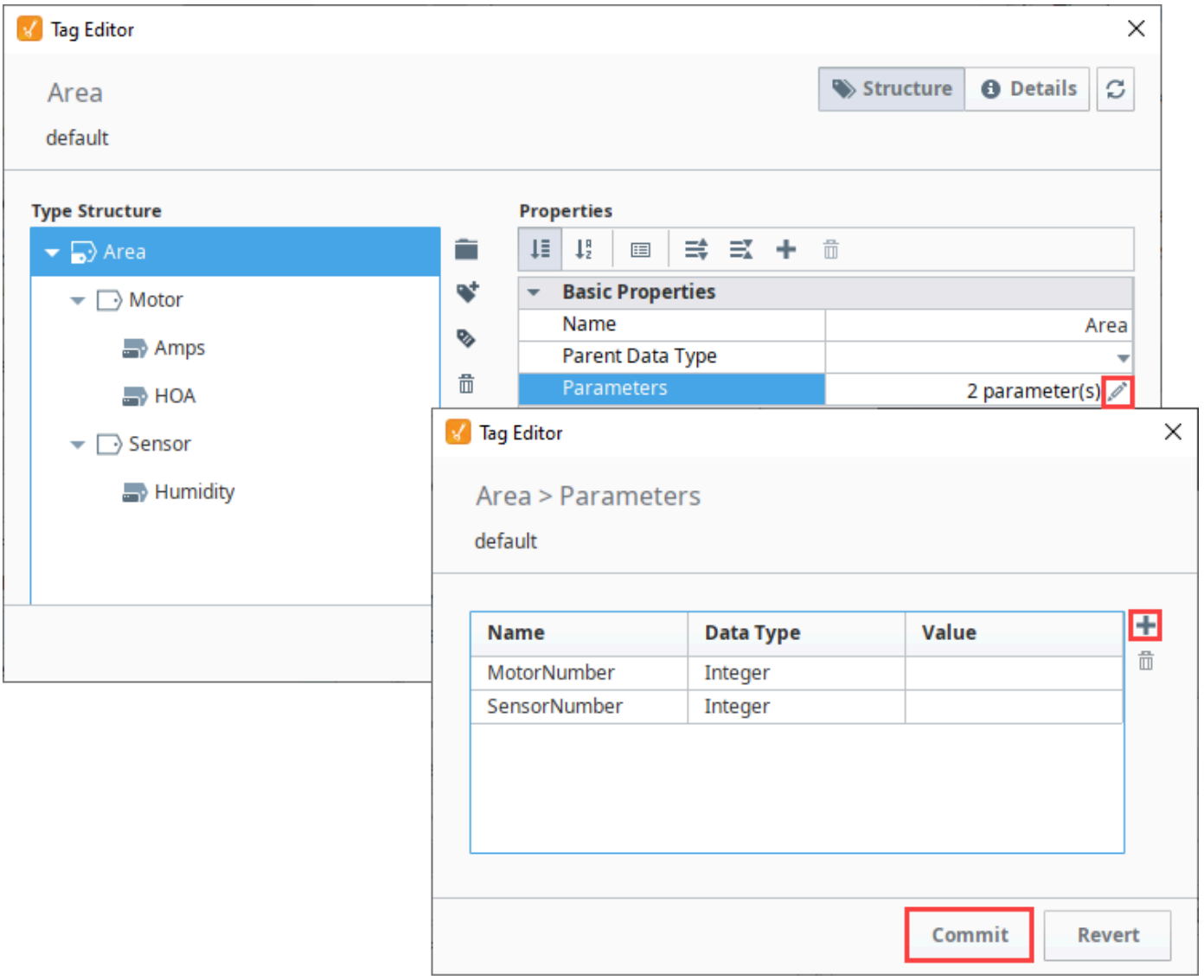


3. With the Tag Editor still open, you'll notice that both the Motor and Sensor UDTs were added. For every UDT that you add inside another UDT, those UDT instances have parameters that need to be specified. In this example, the Motor UDT has the **'MotorNumber'** parameter, and the Sensor UDT has the **'SensorNumber'** parameter. You must pass a value into the each of these UDTs (Motor and Sensor) from the parent UDT (Area). To view the parameters for each UDT instance,

select each UDT instance (Motor and Sensor) and click the **Edit** icon.

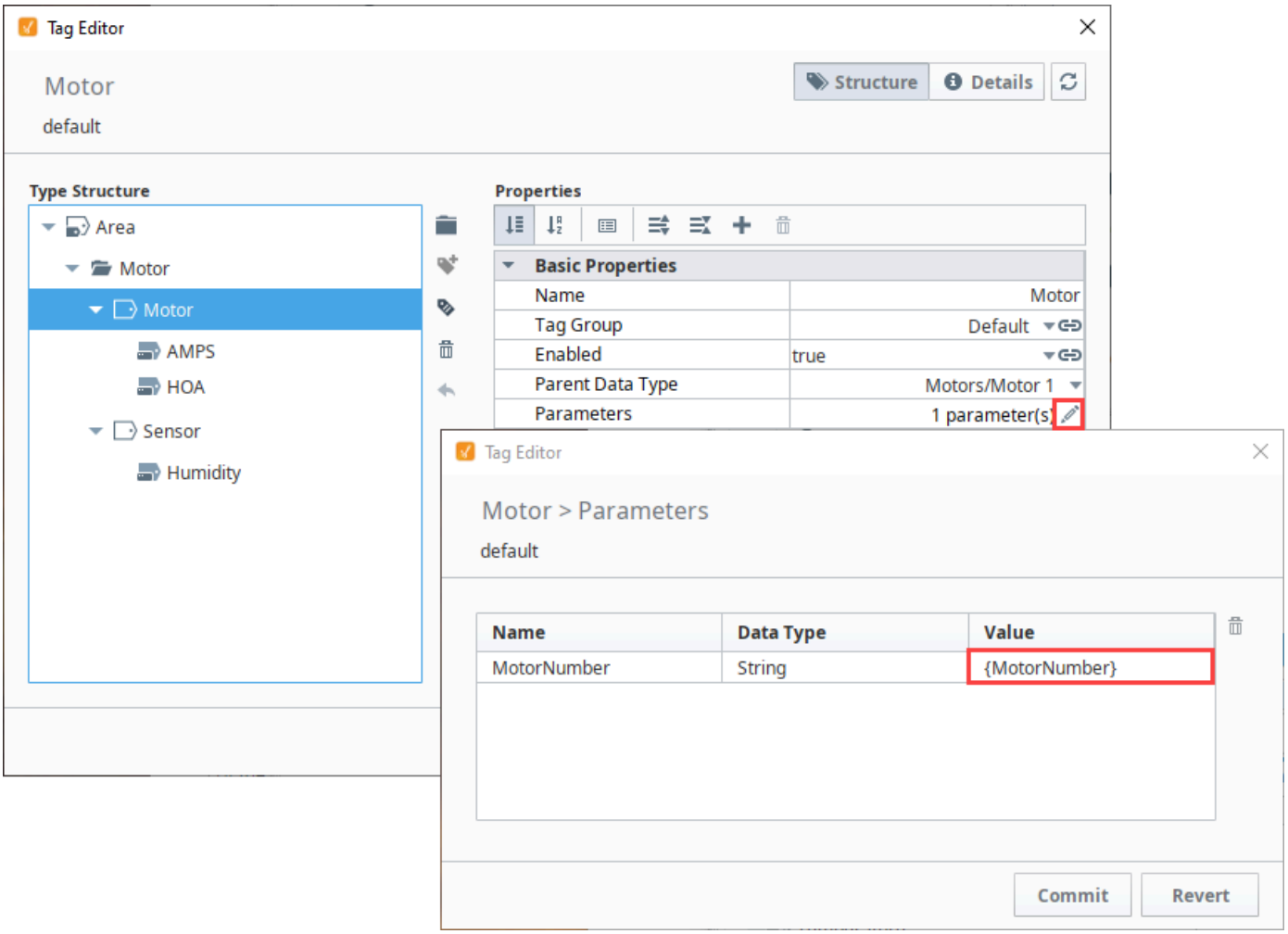


4. Now that you know what parameters are in each UDT instance, go to the Area UDT, and click the **Edit** icon next to the **Parameters** property. The Parameters window will open.
5. Click on the **Add** icon to add the 'MotorNumber' and 'SensorNumber' parameters, then click **Commit**.



6. Now we need to pass these values into the UDT instances by adding a reference. Select the **Motor** UDT, and click the **Edit** icon next to the **Parameters** property.
7. Enter the reference for MotorNumber: `{MotorNumber}`. Click **Commit**.

> **IF MULTIPLE DATA TYPES USE THE SAME PARAMETER**
>
> In the event data types use the same parameter, you only need to enter it once in the new data type (i.e., Area).



8. Do the same steps to add a reference to the **Sensor** UDT. Select the Sendor UDT and click the **Edit** icon next to the **Parameters** property. Enter the reference for SensorNumber: `{SensorNumber}`. Click **Commit**.
9. Click **OK**.

Tag Groups

INDUCTIVE UNIVERSITY

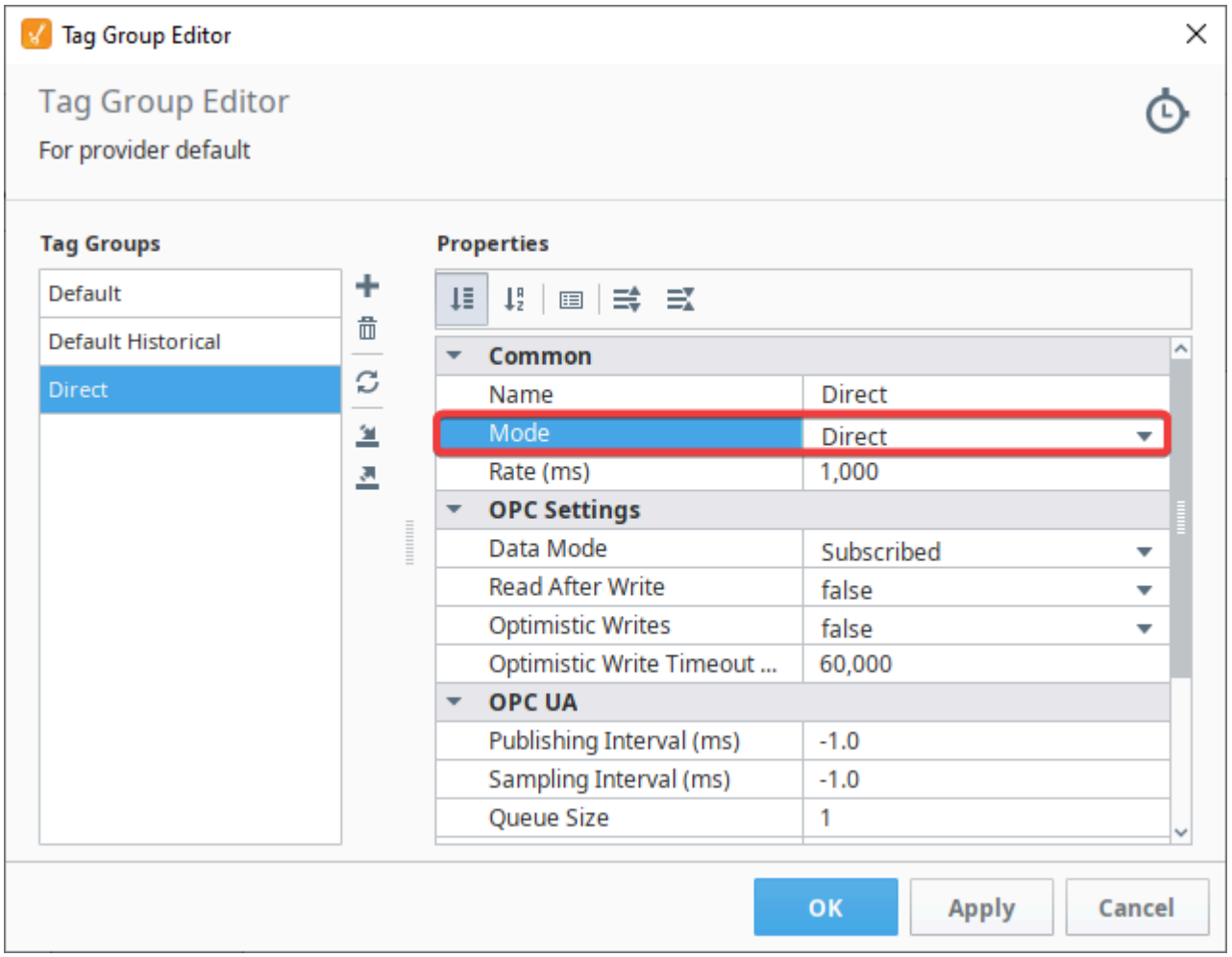
Tag Group Overview

Watch the video

What Is a Tag Group?

Tag Groups dictate the rate of execution of Tags, and therefore play a crucial role in the design of large and high-performance systems. Highly optimized systems generally use multiple Tag Groups which allows for each data point to be polled only as fast as it needs to be. For example some Tags may need to updated every 500ms, while others may only need to be updated every 30 seconds.

Creating different Tag Groups allows you to organize your Tags into groups that subscribe or poll at different rates.



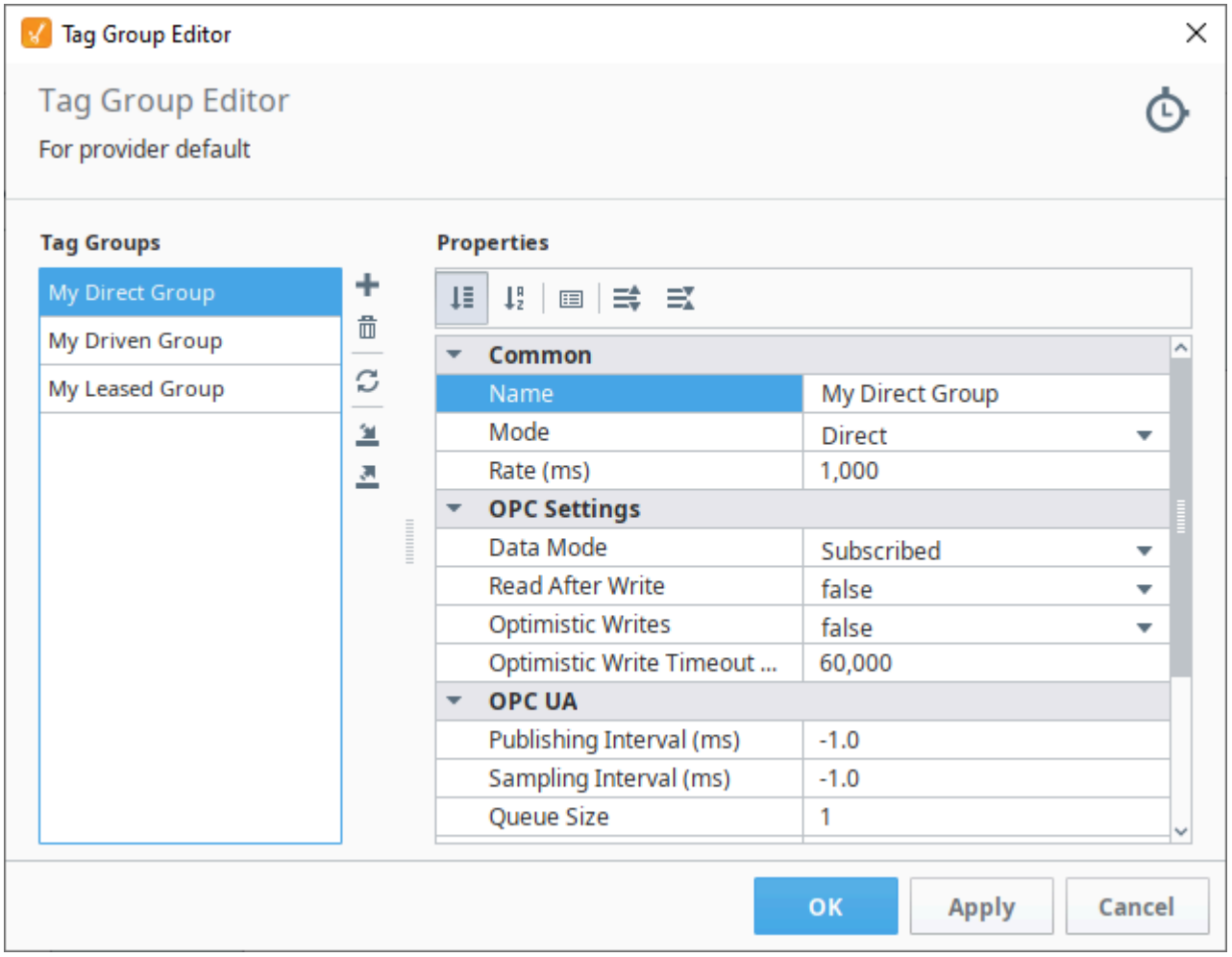
Tag Groups Modes

Tags are assigned to a Tag Group that determine how often they update. For example, how often an OPC value is polled from the PLC, how often an Expression Tag calculates its expression, and how often a Query Tag runs its query. It's easy to create Tag Groups in Ignition for just about any scenario you can think of. Tag Groups are extremely powerful and flexible, and you can create them based entirely on your individual business requirements.

There are three different Tag Group modes in Ignition that you can use. Each mode works a bit differently.

Direct

Executes at one fixed rate, as determined by the **Rate** property in milliseconds. This is the most simplistic option, as the rate of the group doesn't intentionally change between two different rates. The default tag group on newly installed Ignition systems uses this mode. It's the simplest mode in that there is only a single rate for this mode.



See the **Direct Tag Group Example** page on steps to add a Direct Tag Group.

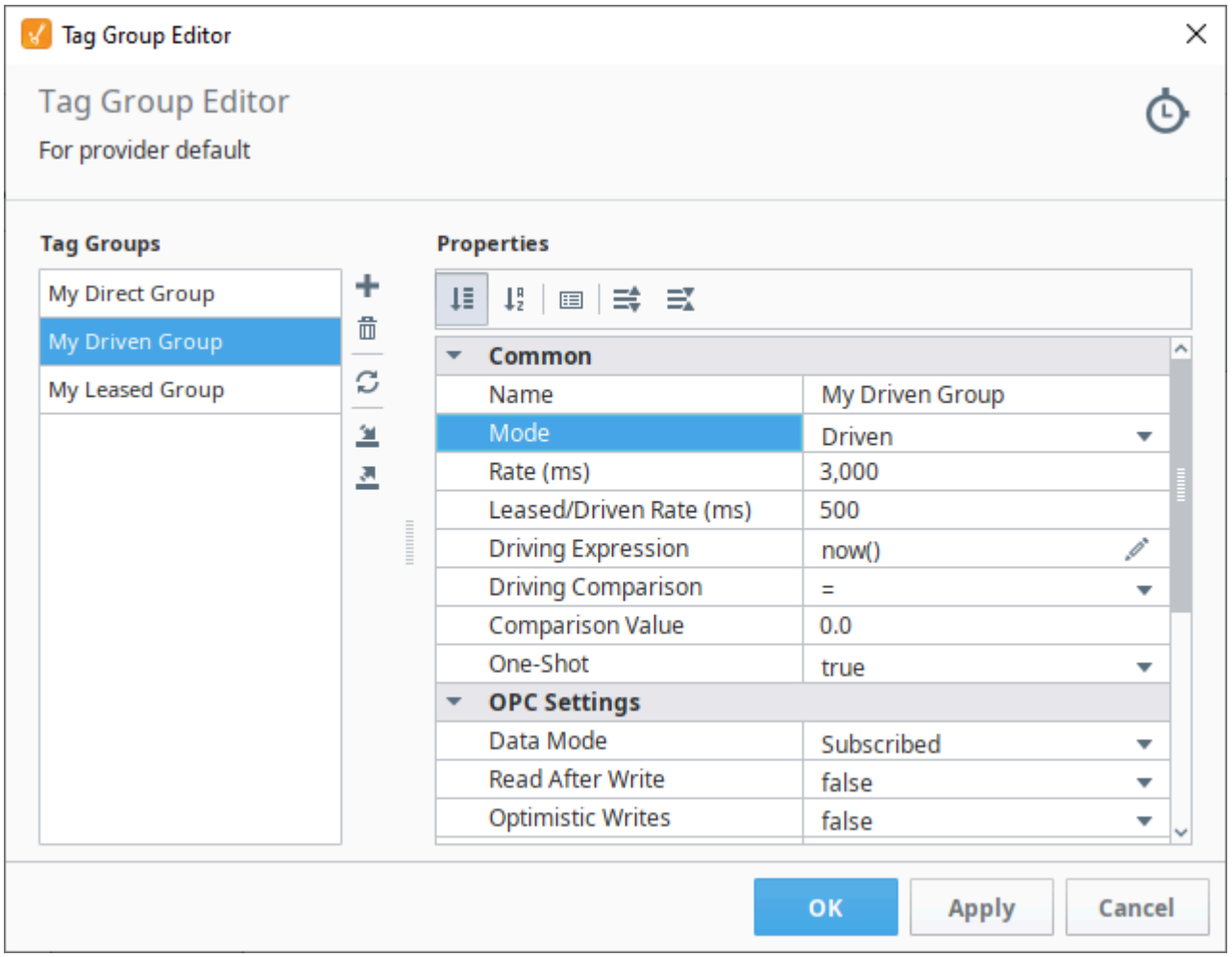
Driven

Driven Tag Groups switch between two different rates based on a condition you configure directly onto the Tag Group. The condition in this case is an **expression** on the **Driving Expression** property. The result of the expression is then compared to the **Comparison Value** property. If the comparison of the two values is equivalent, then the entire tag group executes at the **Leased/Driven Rate** in milliseconds, otherwise it executes at the **Rate**.

Thus, you can have a tag group that switches between a fast and slow rate under certain conditions.

Furthermore, Driven Tag Groups have a **One Shot** property. When set to true, means the group only executes every time the comparison is equivalent. Furthermore, it only executes once every time that comparison transitions from false to true: this is sometimes called "a rising edge".

The One Shot property allows for more interesting tag executions. Instead of running at set rates, you can trigger an execution for all tags on the group by incorporating a tag value as part (or all) of the driving expression, allowing a single tag value change to cause many other tags to update.



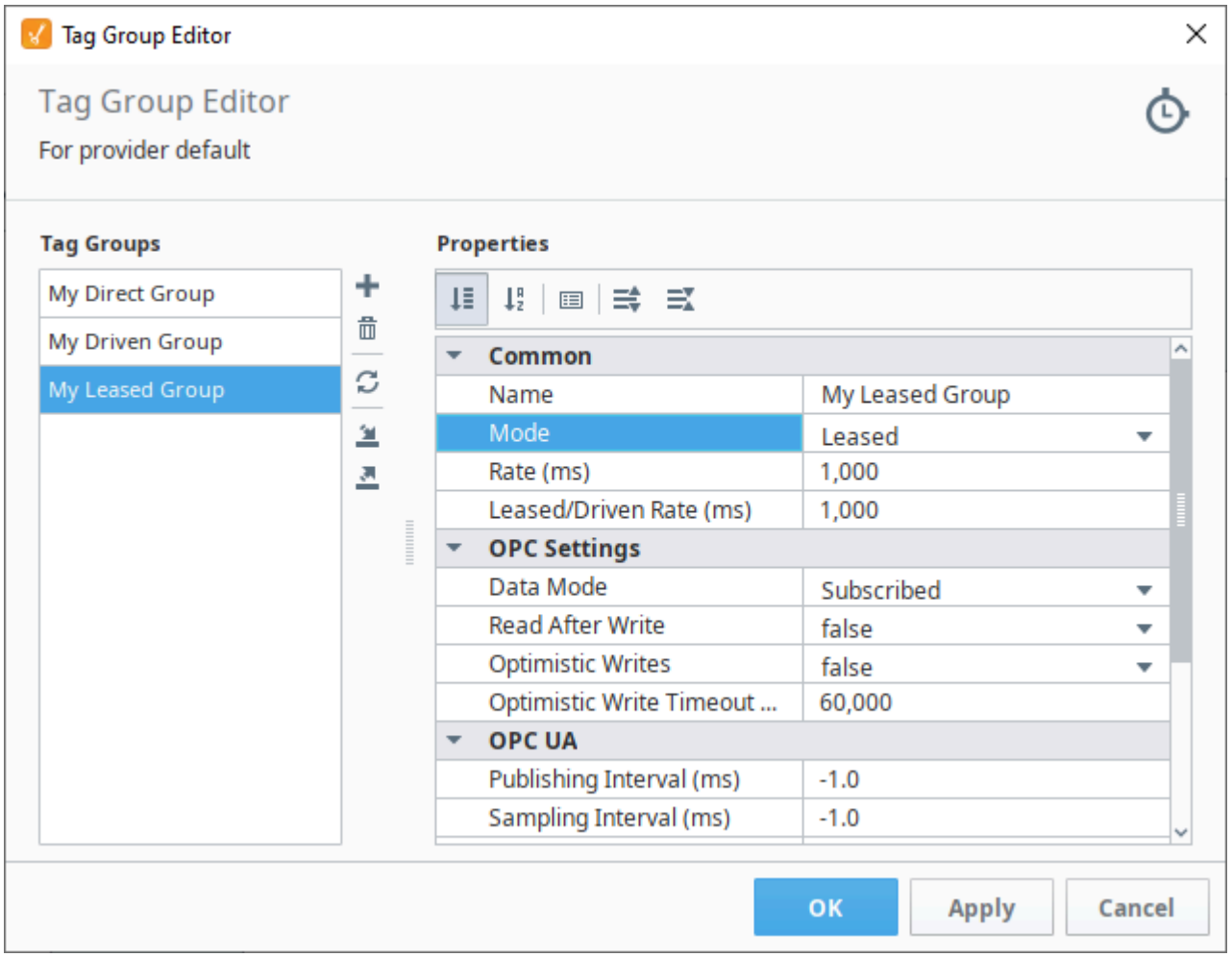
See the **Driven Tag Group Examples** page on steps to add a Driven Tag Group.

Leased

Leased groups have two different execution rates, much like a Driven group. However Leased groups don't rely on a comparison to determine rate. Instead, the driving mechanism is whether a Tag is being displayed on an open window or view: generally via a Tag Binding of some sort.

Tags that are displayed (leased) will run at the **Leased/Driven Rate**, while those Tags on the same group that aren't leased will run at the **Rate**.

Leased groups are unique in that Tags on the same group may execute at different rates while being in the same group. For example, if Tags **A** and **B** are both on the same Leased group, and all Designer, Vision Client, and Perspective Sessions are closed, then both tags execute using the **Rate** value. If a user launches a Perspective Session and switches to a view where the value of **A** is displayed on a component binding, then **A** will switch to the **Leased/Driven** Rate. However, since **B** isn't displayed anywhere, it will continue to execute using the **Rate** value.



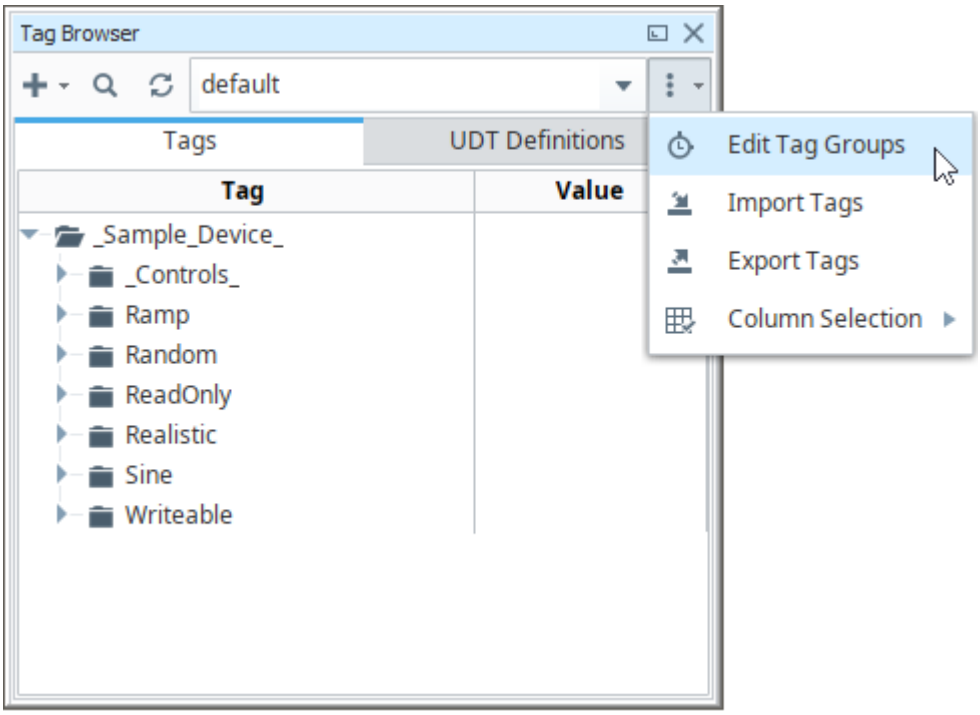
See the **Leased Tag Group Example** page on steps to add a Leased Tag Group.

> **NOTE**
>
> Viewing the Tag only in the Tag Browser will not cause the Tag Group to execute at the Leased Rate.

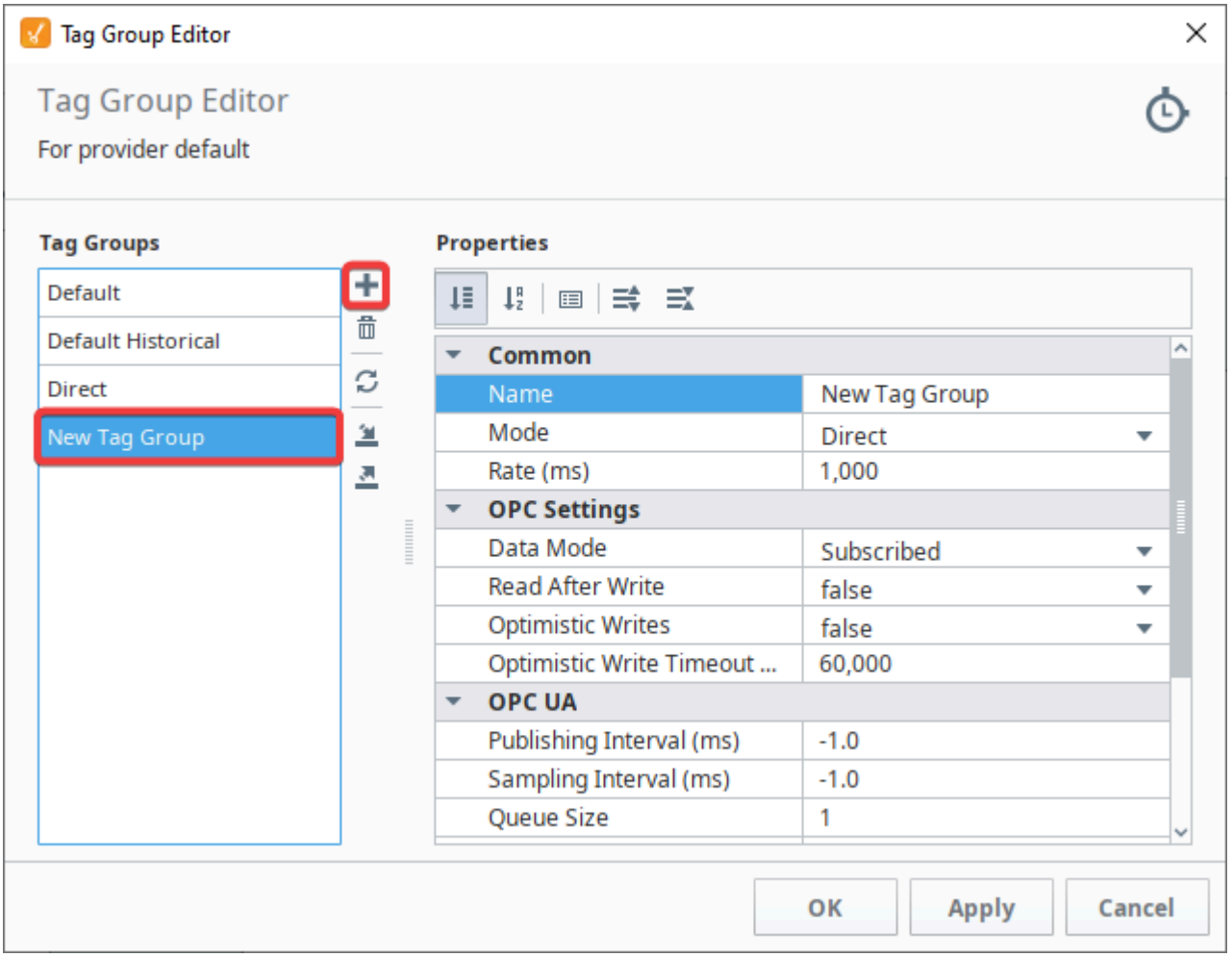
Adding and Editing Tag Groups

Adding and editing Tag Groups is easy in the Designer once you understand how the different Tag Group modes work. It's just a matter of choosing which Tag Group mode you want to use for your Tag, and entering the properties for your Tag Group.

1. In the Tag Browser, click on the **Edit Tag Group** icon under the Options Menu to open the Tag Group Editor window.



2. A list of already configured Tag Groups appear on the left side of the window and configuration settings on the right. To add a Tag Group, click the **Add** ✚ icon. (Alternatively, you can click on an existing Tag Group to edit it.)



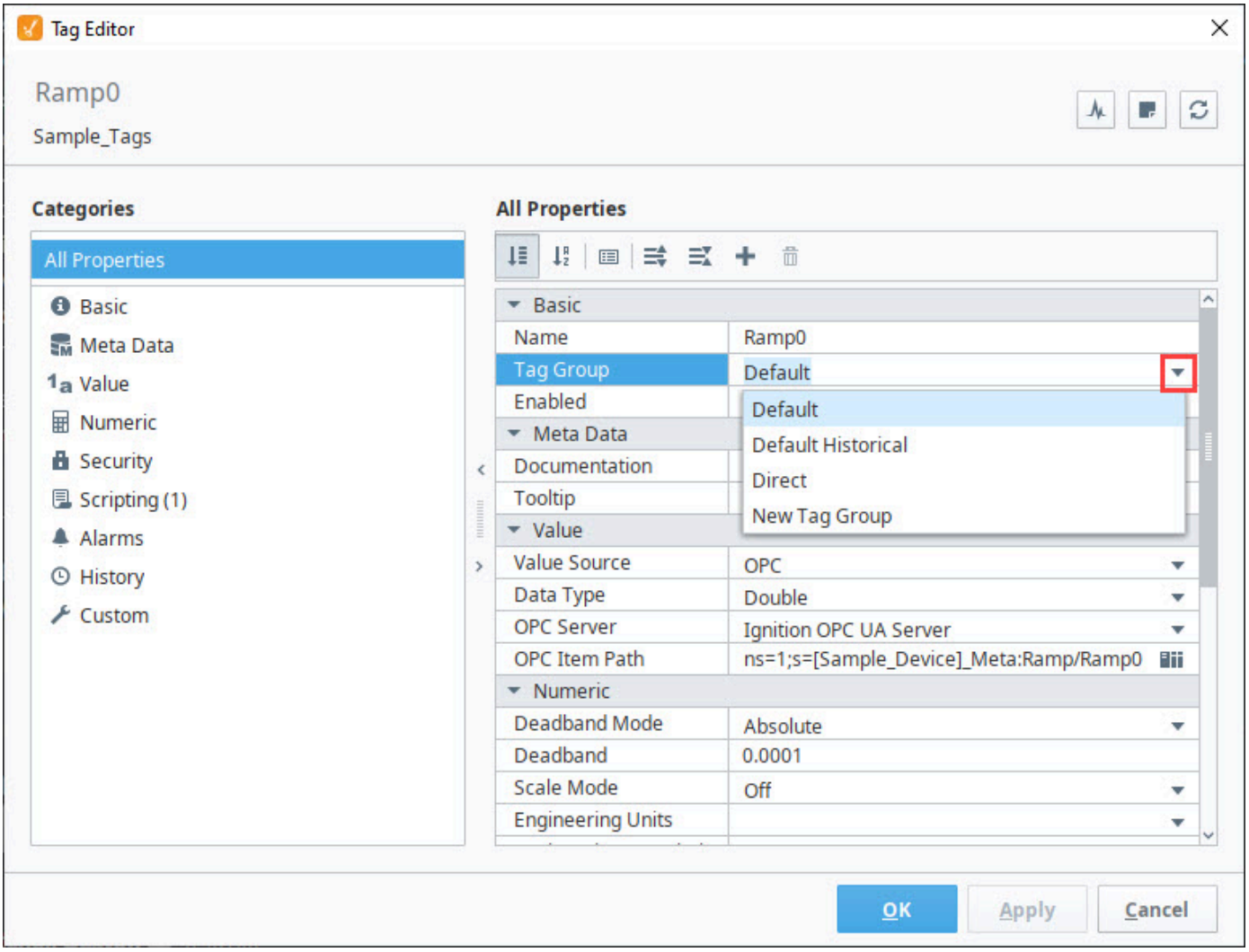
When a Tag Group in the list is selected, the properties for the group will appear on the Properties Table. Each mode will have slightly different settings that will need to be configured. You can find

descriptions for those properties in the **Tag Group Properties** section below.

Setting a Tag Group on a Tag

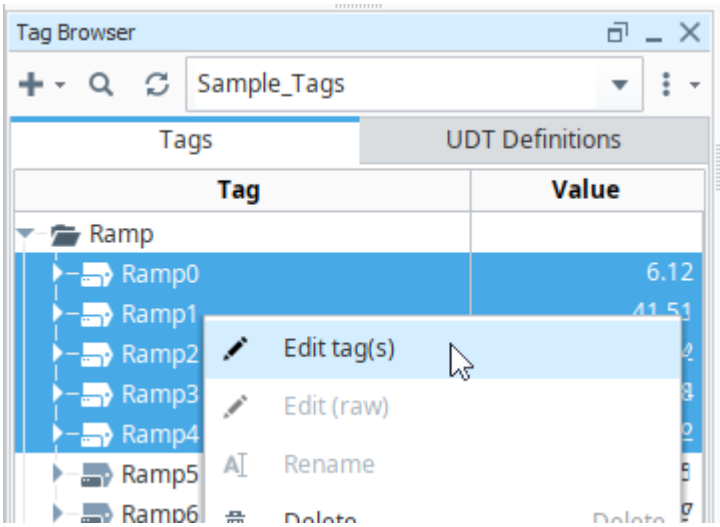
The example below demonstrates how to find the Tag Group on a tag, and change it to something else.

1. In the **Tag Browser**, right-click on any Tag, and click the **Edit tag** icon. The **Tag Editor** window opens. A list of Tag properties is displayed.
2. Under **Basic Properties**, on the right side of the **Tag Group** property, click the dropdown list and a list of available Tag Groups will appear. Again, Tag Groups are configured per Tag Provider, so this list will only contain Tag Groups that are present in the Tag's Tag Provider.



3. If you wish to change the Tag Group. Select the new option, and click **OK** or **Apply** to apply the new Tag Group. The Tag will switch rates and start using the new Tag Group.

As a reminder, you can select multiple Tags in the Tag Browser by right clicking to edit the Tags. This opens the Tag Editor, and sets the Tag Group for all the selected Tags at the same time.



Tag Group Properties

The following table lists the properties for Tag Groups. Note that some properties are available only for specific modes.

Common

Name	Category	Description	Applicable Modes
Name	Common	Unique name of the Tag Group.	All
Mode	Common	Defines how the Tag Group executes: • **Direct**: Executes at one fixed rate, as determined by the **Rate** property. • **Driven**: Executes based on conditions set by **Driving Expression**, **Driving Comparison**, and **Comparison Value**. **Leased/Driven Rate** applies when conditions are met. • **Leased**: Executes at base **Rate** until requested by a client or designer session, then switches to **Leased/Driven Rate** for displayed tags only. **Note**: Historical logging records executions. If your base **Rate** is 0, tags will not log until triggers occur.	All
Rate	Common	Base update rate for tag execution, in milliseconds. **Note:** A rate of 0 means the Tag Group will not execute.	All
Leased/Driven Rate	Common	Rate used when **Leased** or **Driven** triggers are active. Ignored if **Driven Mode** with **One Shot** is enabled.	Leased, Driven
Driving Expression	Common	The resulting value of this expression is compared to the **Comparison Value** using the **Driving Comparison** to determine when tags in this group will be executed at the **Leased/Driven Rate**, instead of the base **Rate**. Note that this expression can reference a single tag's value instead of a calculation.	Driven

Name	Category	Description	Applicable Modes
Driving Comparison	Common	Operator of comparison (=, !=, <, <=, >, >=) used between the **Driving Expression** and **Comparison Value** to determine when tags will be executed at **Leased/Driven Rate**. If **Any Change** operator is selected, the Tag Group will execute immediately whenever the Driving Expression changes value, and the Tag Group no longer uses the base **Rate** or **Leased/Driven Rate** properties.	Driven
Comparison Value	Common	Static value that is compared to the **Driving Expression** using the **Driving Comparison** to determine when tags in this group will be executed at the **Leased/Driven Rate**, instead of the base **Rate**.	Driven
One Shot	Common	Executes once when conditions are true, waits for false then true to re-execute.	Driven

OPC Settings

Name	Category	Description	Applicable Modes
Data Mode	OPC Settings	Dictates how OPC values are obtained: • **Subscribed:** OPC tags subscribe to updates, reducing traffic. However, if the PLC or tag source sends too many updates, it can overload the network. If subscription requests fail, the mode will failover to polling, making it important to have a polling rate set. • **Polled:** Tags are read at each Tag Group execution, offering predictable network load. Useful for slow connections or avoiding subscription-induced overload. Combined with **Driven Mode** and **One Shot**, Polled mode can enable manual refresh buttons in client sessions.	All
Read After Write	OPC Settings	Value changes from Ignition to an OPC device will send a write request, followed by a read request. This varies from **Optimistic Writes** because value changes will not show in the Ignition tag until hearing back from the read request. **Note**: Read After Write doubles network traffic for each value change. **Caution**: Do not use Read After Write with OPC tags in subscription mode due to a race condition that can leave you with Ignition tag values that don't match PLC tag values.	All
Optimistic Writes	OPC Settings	Only applies to OPC tags in the Tag Group. Value changes will be applied immediately to Ignition tags. The time out duration is determined by **Optimistic Write Timeout**. So although writes may appear to execute	All

Name	Category	Description	Applicable Modes
		quicker on slow connections, the correct OPC tag value isn't guaranteed to show on the Ignition tag. Using a slow polling Tag Group example of 30 seconds where an operator makes a manual change, this option would allow the operator to see the new value immediately, even though it won't be confirmed for 30 seconds. If both **Optimistic Writes** and **Read After Write** were enabled, the operator would see the new value immediately and it would be confirmed as soon as the read request comes back (earlier than 30 seconds).	
Optimistic Write Timeout (MS)	OPC Settings	The timeout duration for **Optimistic Writes**. A value of 0 will effectively disable this option. Keep **Optimistic Write Timeout** greater than **Rate**. Otherwise this will happen: a value changes from 0 to 1 will show 1 immediately, then after the timeout revert back to 0, then after polling rate will change to 1 (which would be confusing for users).	All

OPC UA

Name	Category	Description	Applicable Modes
Publishing Interval (ms)	OPC UA	The rate at which data is delivered to the OPC-UA client. A value of -1 means automatic, allowing the OPC-UA client to determine the rate.	All
Sampling Interval (ms)	OPC UA	Allows the Requested Sampling Interval column on the Gateway **Status > OPC Connections > Clients > Nodes** page to be requested independently of the Tag Group Rate . The default value for this is -1, which indicates that the value will be inherited from the Tag Group Rate value. **Note:** Setting the Sampling Interval value to 0 against a driver that supports it will show the requested value on the Gateway. Setting the value to 0 against a driver that doesn't support it will show a Revised Sampling Interval of 500. Setting the value below 0 will show the the interval rate of the Tag Group setting in the Requested/Revised Sampling Interval columns.	All
Queue Size	OPC UA	The OPC-UA specifications states that in cases where the sampling interval (the rate as which the server checks the data source for changes) is faster than the publishing interval (rate at which the the data is delivered to the client), the samples may be queued or batched together before publishing. This setting determines the maximum size of that queue. When the maximum is reached and a publish has not yet occurred, oldest samples are dropped first.	All

Name	Category	Description	Applicable Modes
		Currently, there are not many features in Ignition that utilize multiple entries in the queue, but 3rd party OPC-UA clients may be able to take advantage of this setting. Note that values on Ignition Tags will only ever show one value at a time, regardless of what this property is set to. (If the value is 0, nothing will be queued.) Support for this feature is dictated by the driver. See the **DNP3 page** for details on how buffered data and Sequence of Events works	
Include Timestamp-Only Changes	OPC UA	If true, allows the OPC-UA client to receive timestamp-only changes for tags in this group. By default, the OPC-UA client only receives value and quality changes.	All

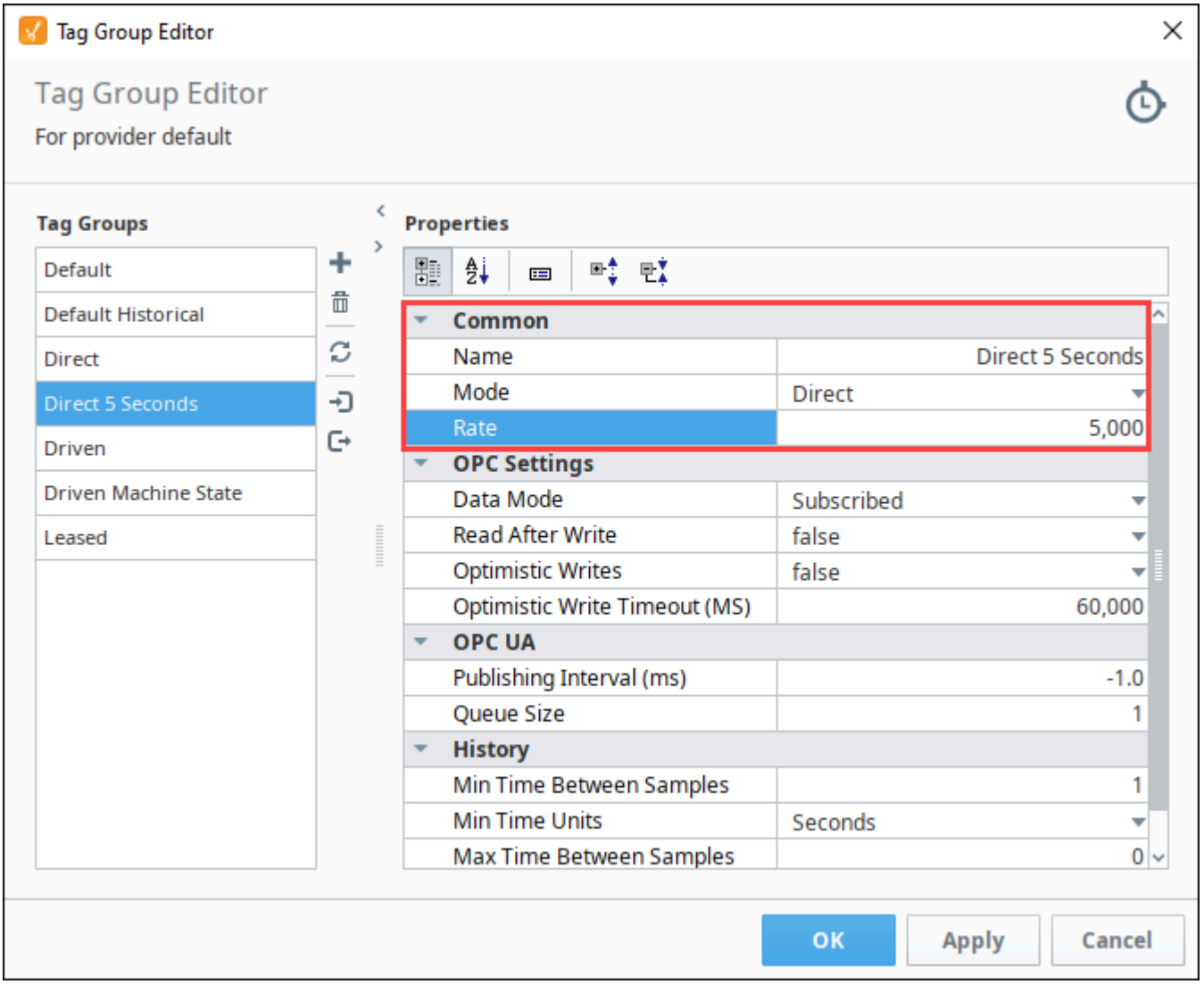
History

Name	Category	Description	Applicable Modes
Min Time Between Samples	History	Minimum time between samples (integer).	All
Min Time Units	History	Minimum time in units is defined as: Milliseconds, Seconds, Minutes, Hours, Days, Weeks, Months, and Years.	All
Max Time Between Samples	History	Maximum time between samples (integer). When a Tag uses the "Tag Group" sample mode, and is set to this Tag Group, it will ignore it's own Max Time Between Settings setting (configured directly on the Tag), but only if this Max Time Between Samples setting (on the Tag Group) is using non-default values. Defaults to 0.	All
Max Time Units	History	Maximum time in units is defined as: Milliseconds, Seconds, Minutes, Hours, Days, Weeks, Months, and Years.	All

Direct Tag Group Example

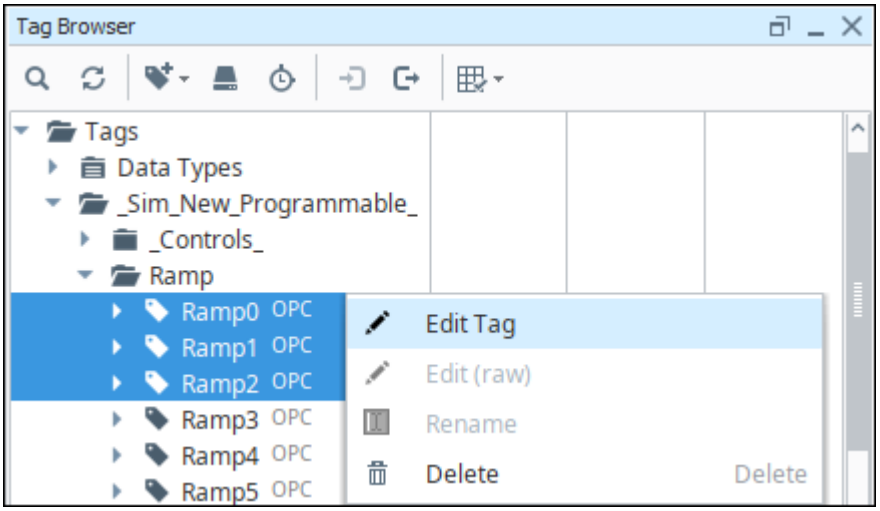
Add a Direct Tag Group

1. In the Tag Browser, click on the **Tag Groups** icon to open the **Tag Group Editor**.
2. On the bottom left side, click the Add icon to create a new Tag Group, and enter the values for the following properties:
   - Name - Enter a unique name for the Tag Group: **Direct 5 Seconds**
   - Select the Mode: **Direct**
   - Enter Rate: **5,000**
3. Click **OK** to save.

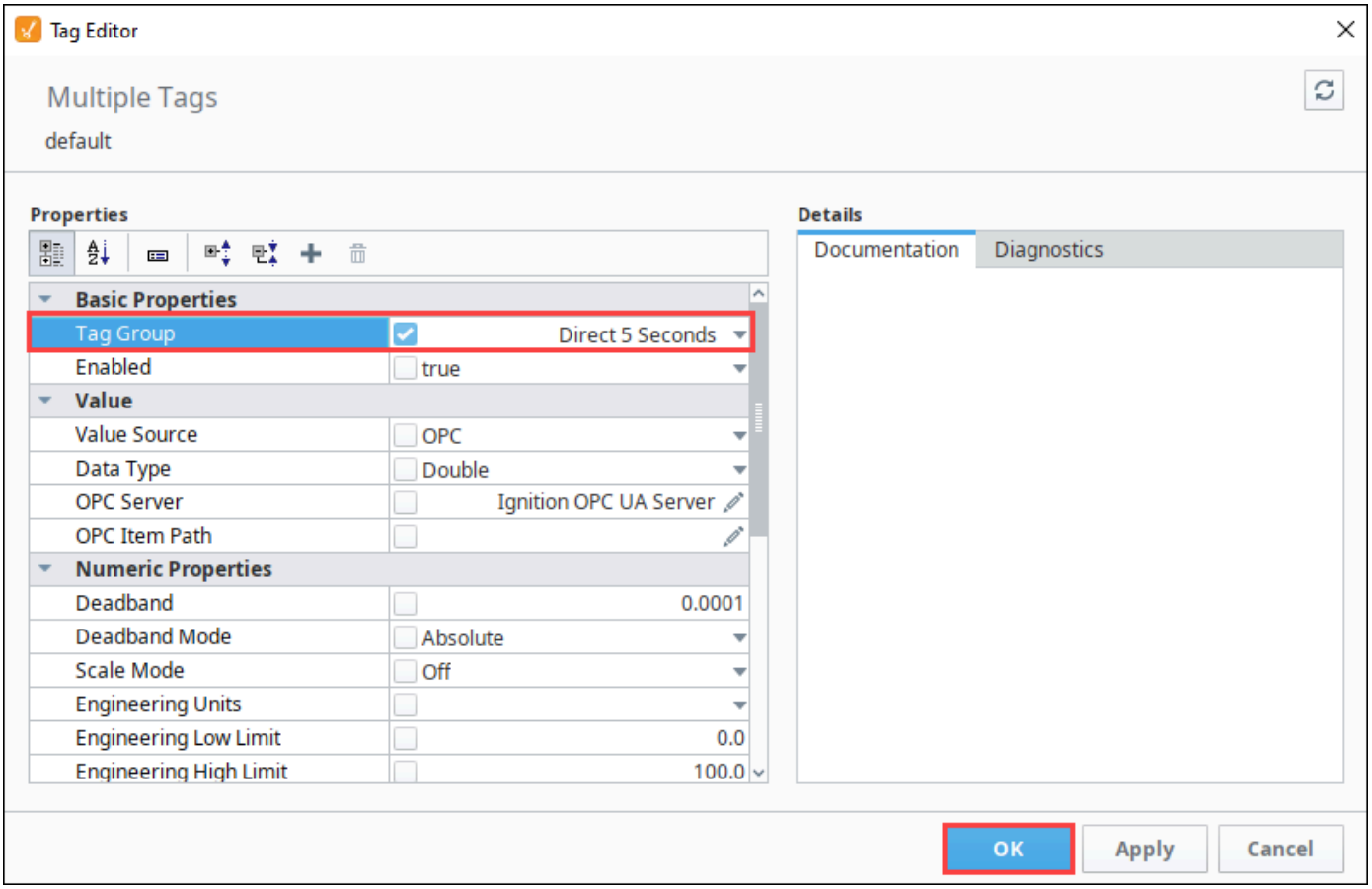


4. Now that you have your Tag Group created, let's add multiple Tags to the Tag Group. Go to your **Tag Browser**, find some Tags you want to add to the Tag Group. This example uses several

Ramp Tags. Right click on the selected Tags, and click on **Edit Tag** .



5. This opens the **Tag Editor** window. It also shows you that you have multiple Tags selected. Select the **Direct 5 Seconds** Tag Group from the dropdown list.
6. Click OK.

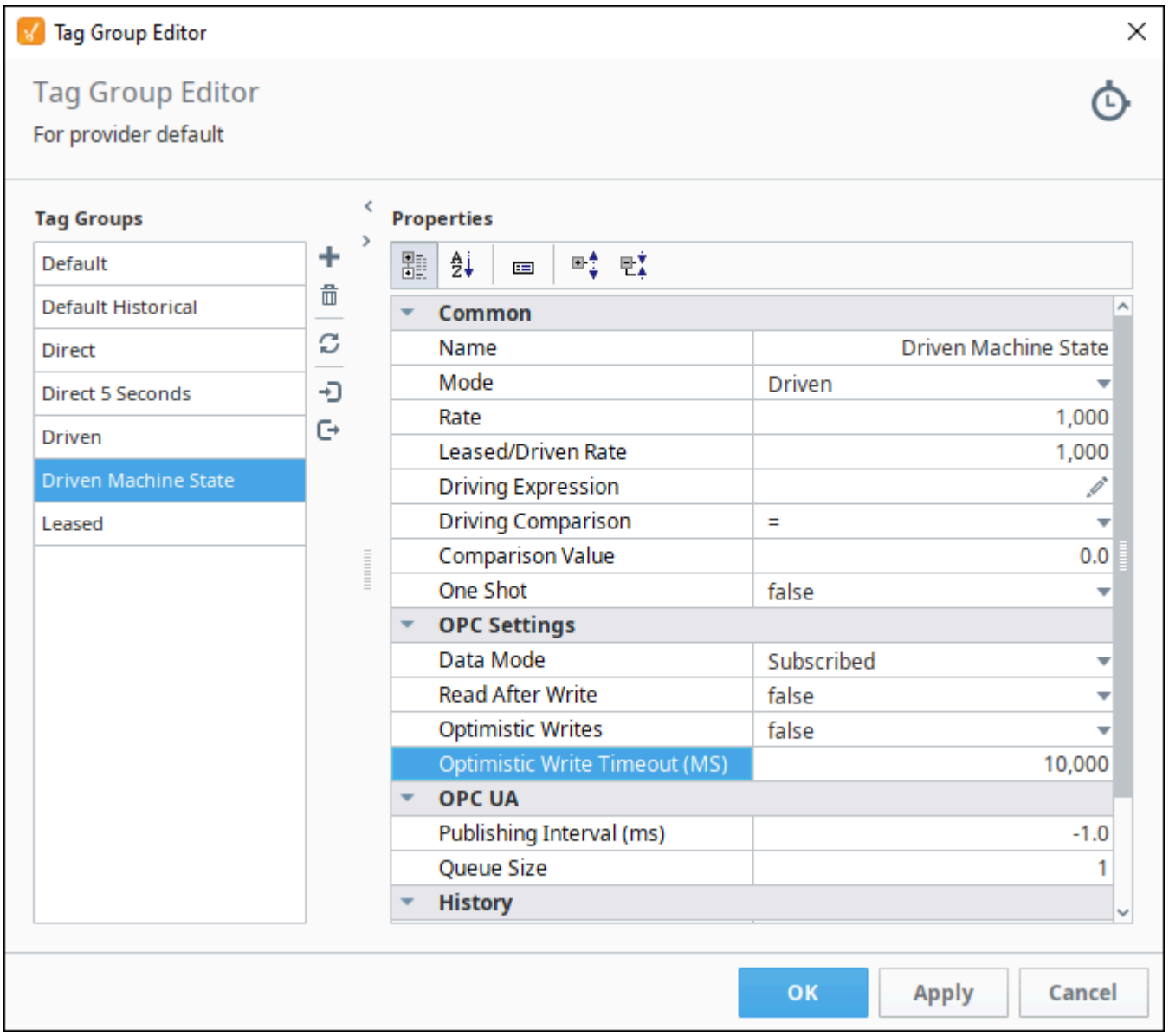


You created a new Direct 5 Second Tag Group and added your Tags. Just make sure you want to poll the 5 second values all the time (24/7) when you use the Direct 5 Seconds Tag Group.

Driven Tag Group Examples

Driven Tag Group - Machine State

Driven Tag Groups that are based on a machine can be very important when you only want to poll values differently based on machine state. A machine state Tag Group is when the Tags are polled at one rate with the machine ON, and a different rate when the machine is OFF. You can easily see the rate change in the the Tag Browser using this type of condition.



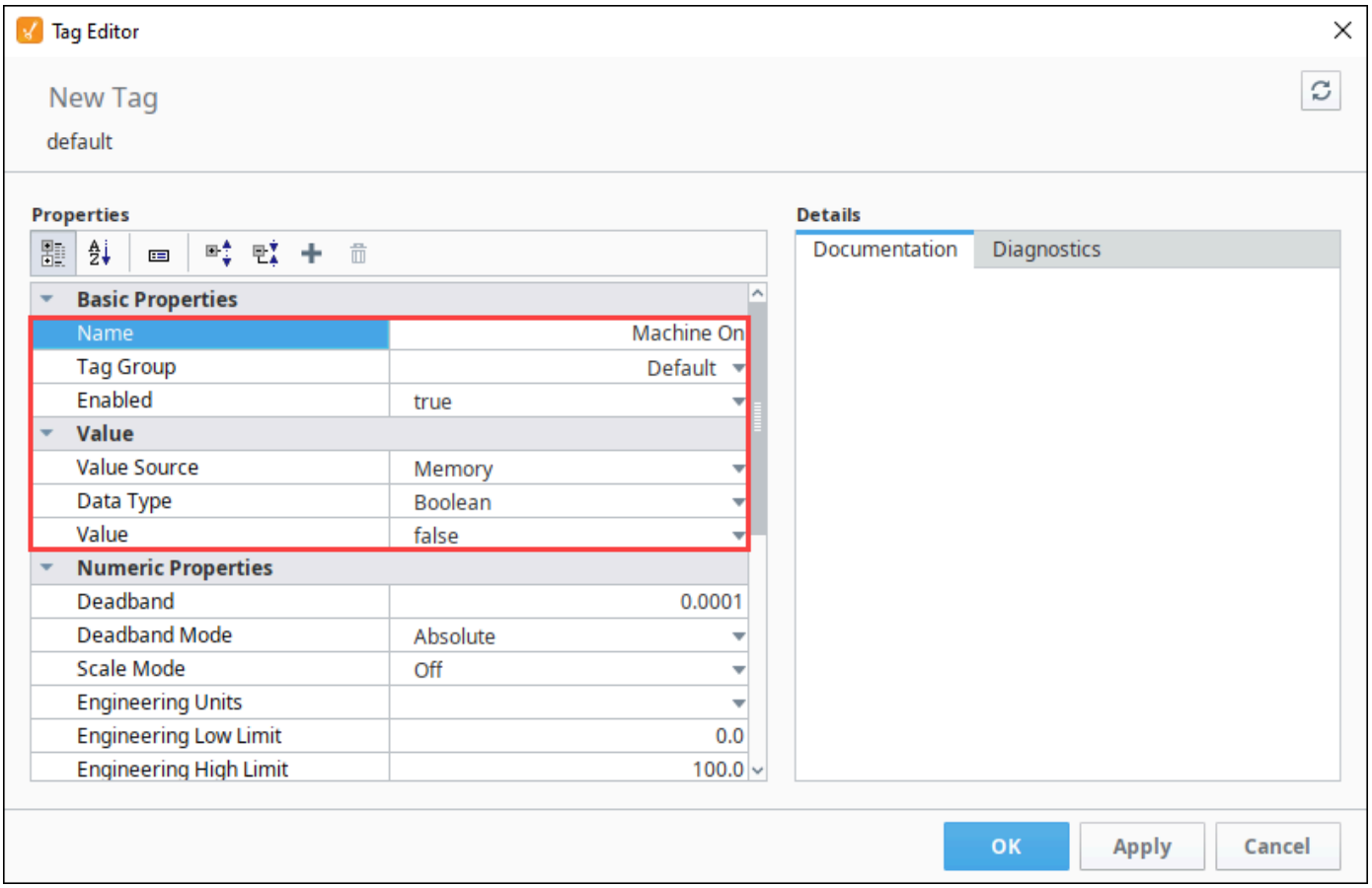
Create a Machine State Tag

First, you need to create a driving Tag if you don't already have one.

1. In the **Tag Browser**, right-click on the **Tags** folder, then go to **New Tag > Memory Tag** to create a memory Tag.

The Tag Editor window is displayed.

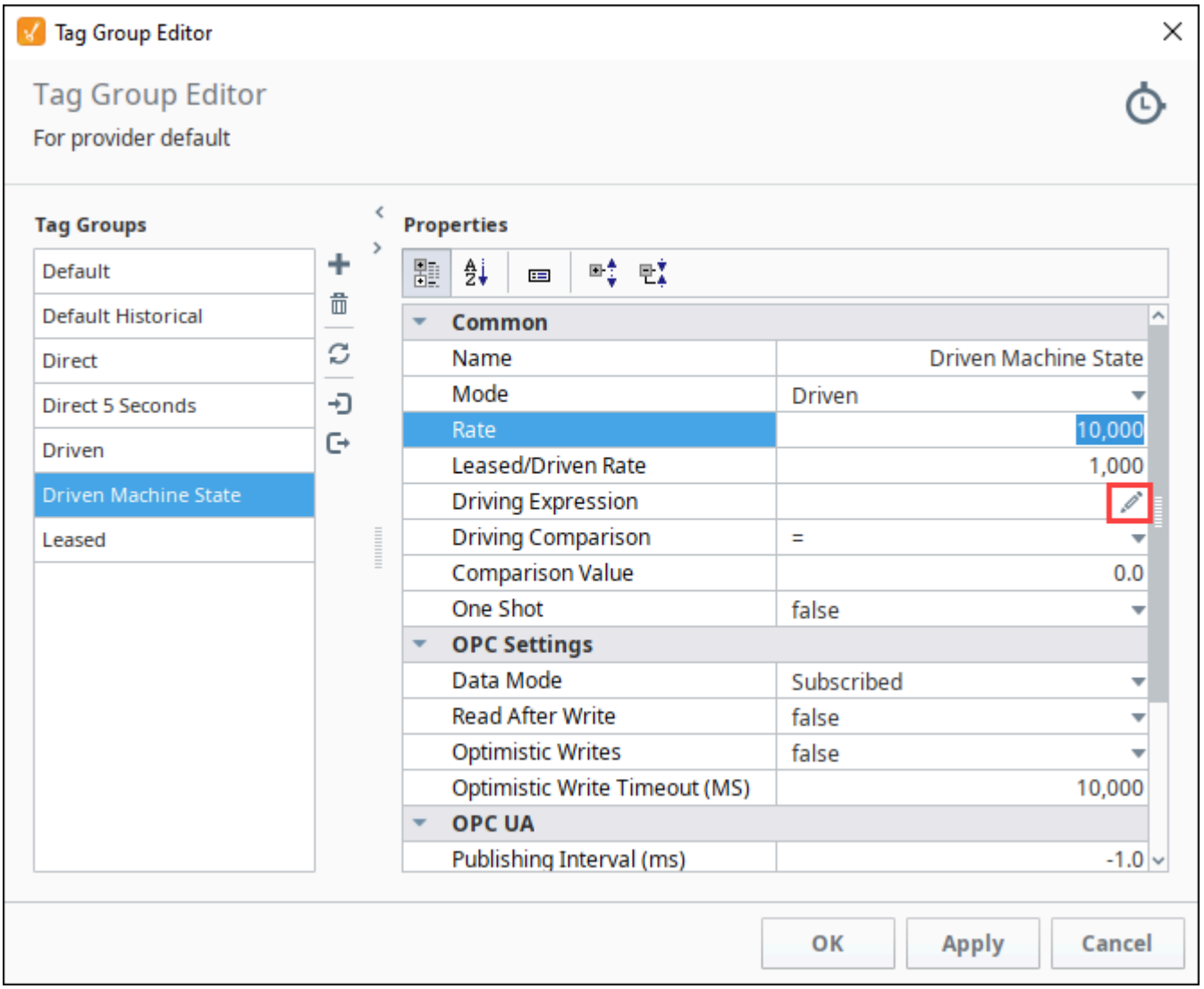
2. In **Tag Editor**, enter the following:
    - Name: **Machine On**
    - Tag Group: **Default**
    - Enabled: **true**
    - Data Type: **Boolean**
    - Value: **false**
3. Click **OK** to add the **Machine On** Tag to the Tag Browser.



Add a Driven Tag Group Based on the Machine State

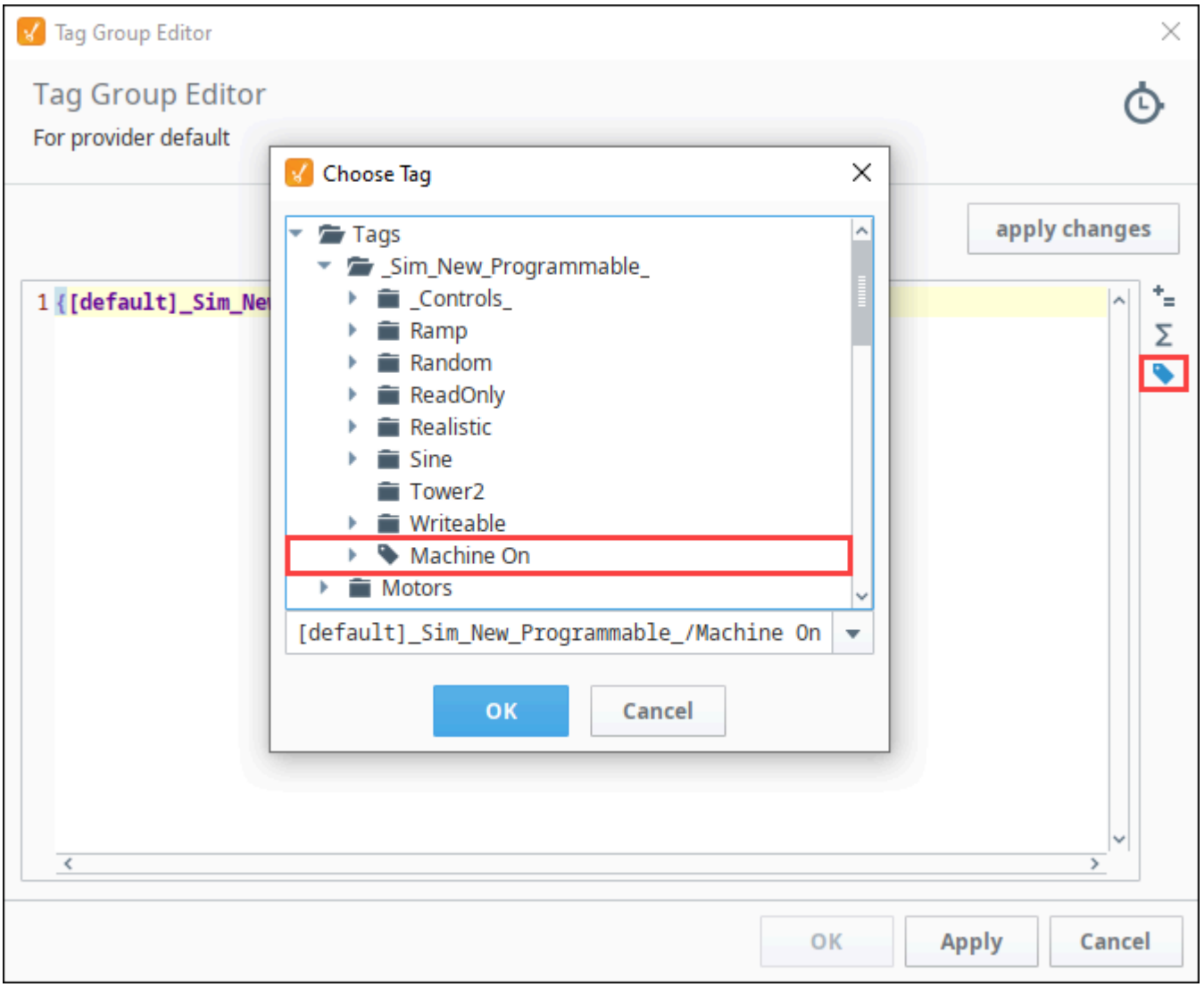
Once you have your driving Tag created, add a Driven Tag Group that updates the Tags based on when the machine is ON.

1. In the **Tag Browser**, click on the **Edit Tag Groups** icon to open the Tag Group Editor.
2. To create a new Tag Group, click **Add** icon.
3. Enter the name of the Tag Group. For example, you can name it **Driven Machine State**, and set the **Mode** to **Driven**.
4. Set the **Rate** to 10,000 (10 seconds) and the **Leased/Driven Rate** to 1000 (1 second).
5. This Tag Group executes based on the condition you set on the **Driving Expression**. Set the **Driving Expression** by clicking the **Edit** icon.



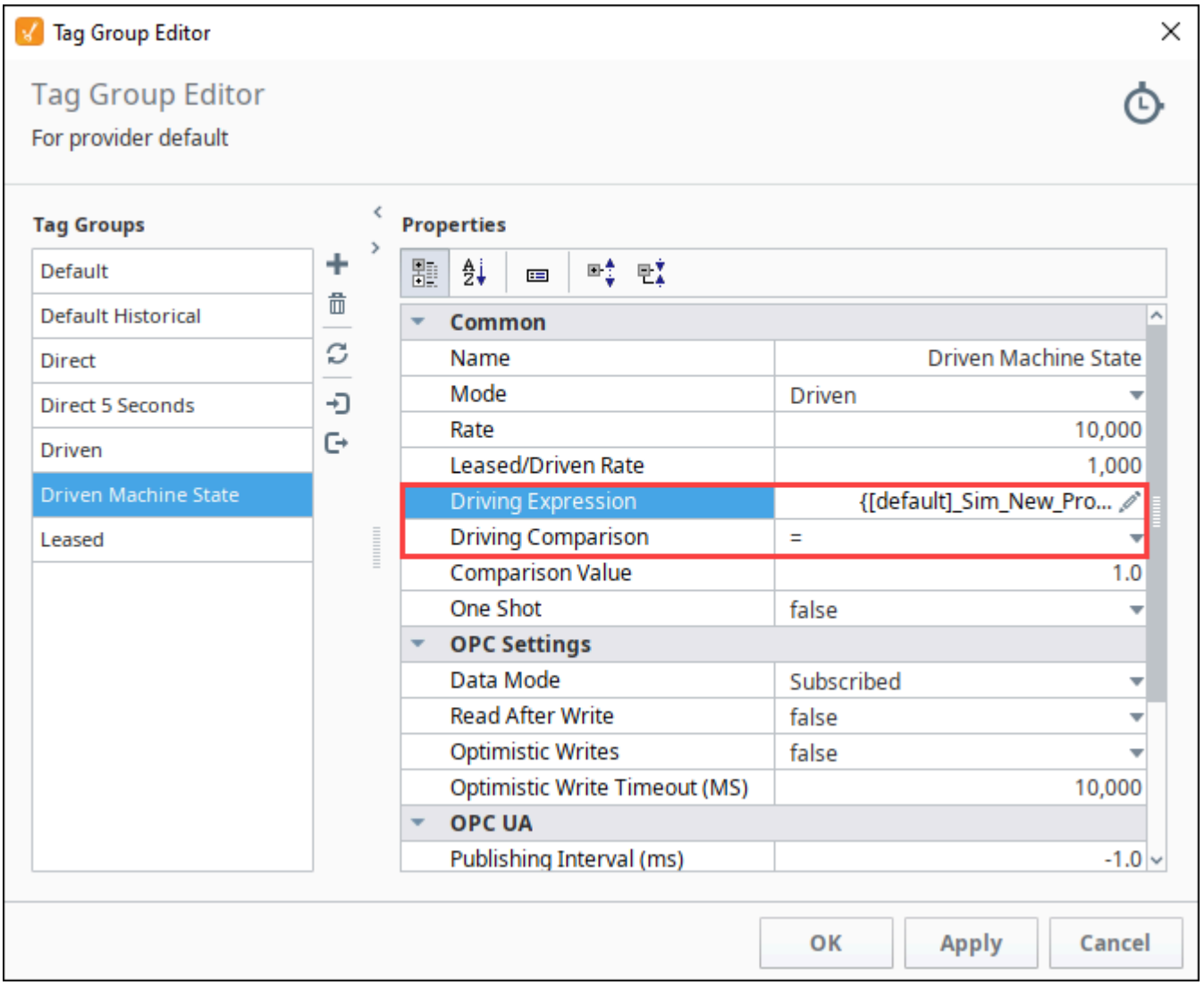
6. This Tag Group executes based on the condition you set on the **Driving Expression**. Set the **Driving Expression** by clicking the **Edit** icon.
7. Click the **Tag** icon and choose the Tag you want to set the condition on (i.e., Machine On).

8. Click **OK** on the popup and **apply changes** in the upper right of the Tag Group Editor.

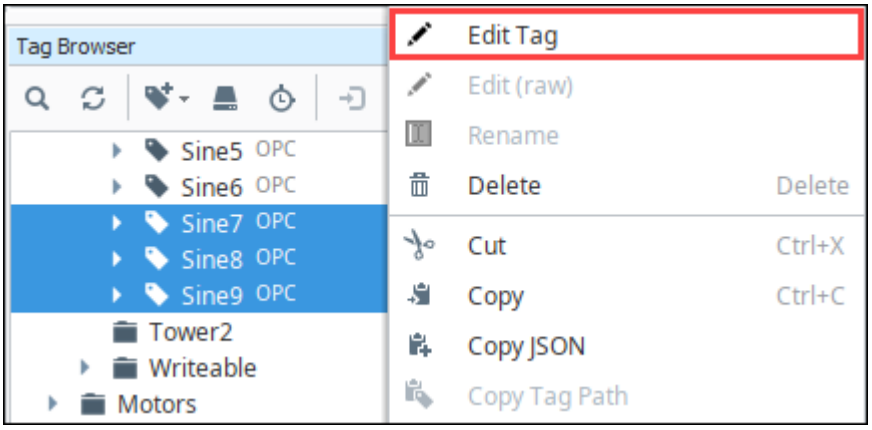


9. While still in the Tag Group Editor, set the following values:
    - Driving Comparison: **=**
    - Comparison Value: **1**

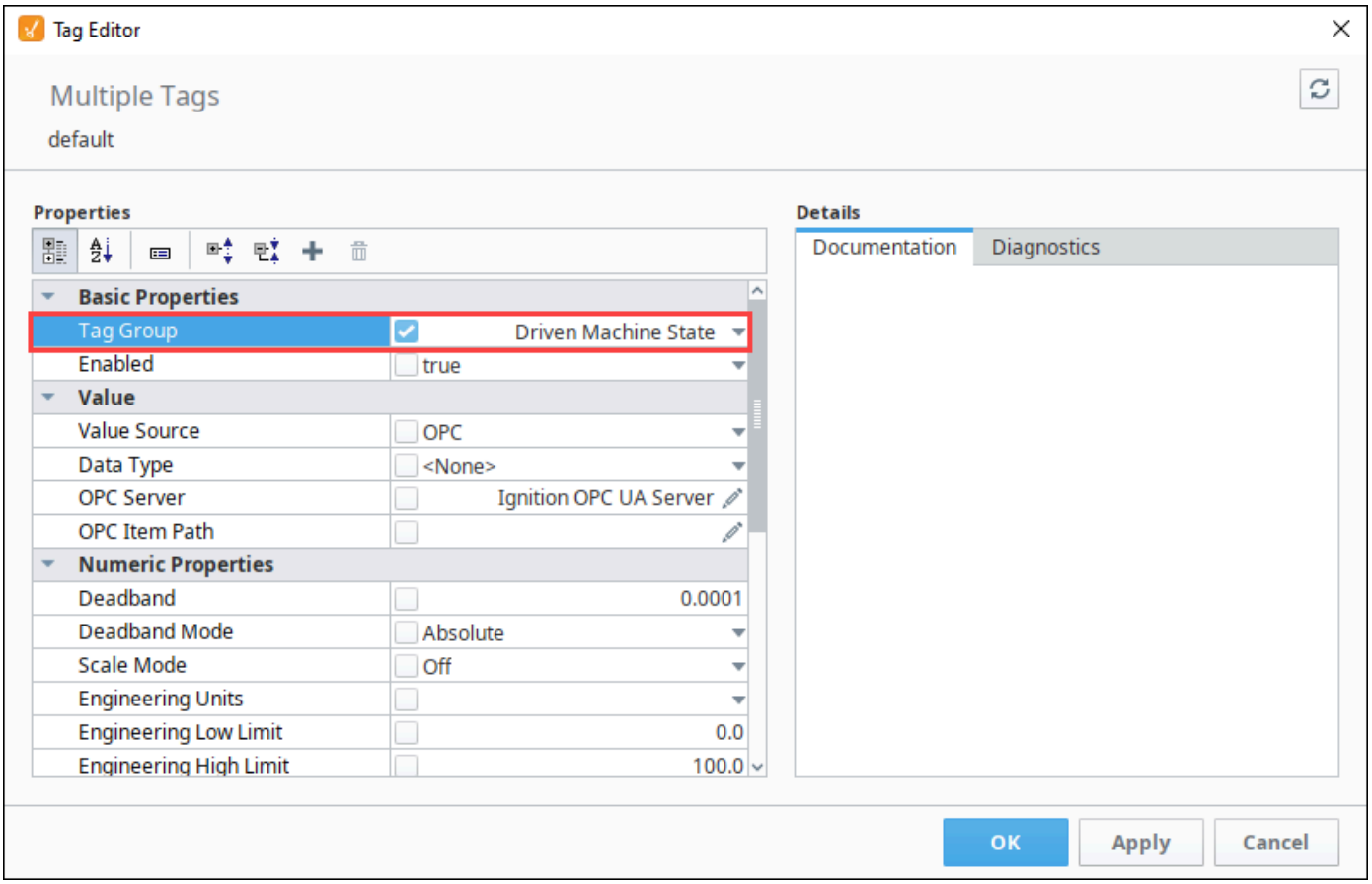
10. Click **OK** in the lower right of the Tag Group Editor.



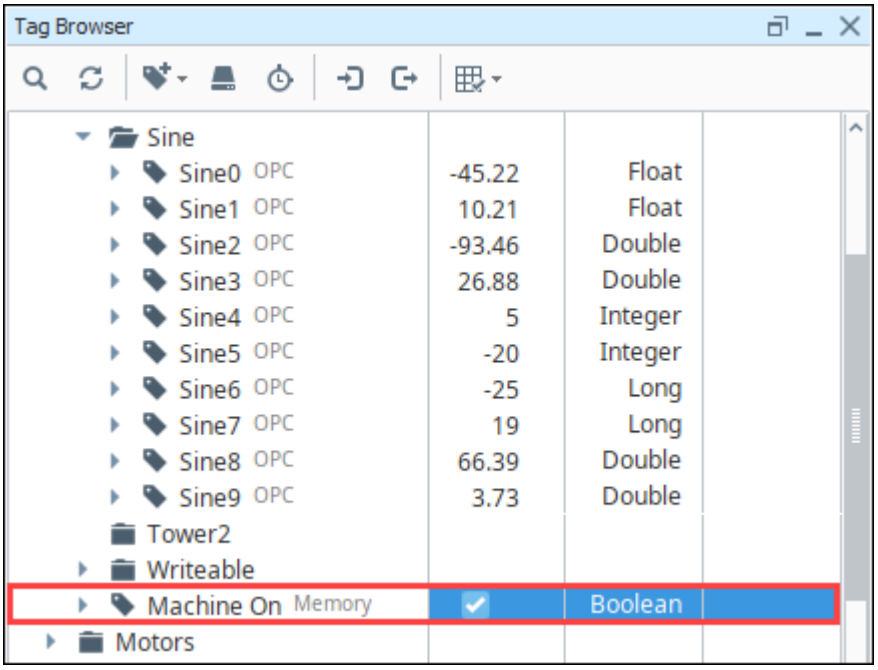
11. When the machine is on, it will use the Leased/Driven Rate (1 second). When the machine is off, it will use the Rate (10 seconds)
12. Next, select all the Tags that you want to use for the Drive Machine State Tag Group. This example uses **Sine7**, **Sine8**, and **Sine9** Tags.
13. Right click on the selected Tags and choose **Edit Tag**.



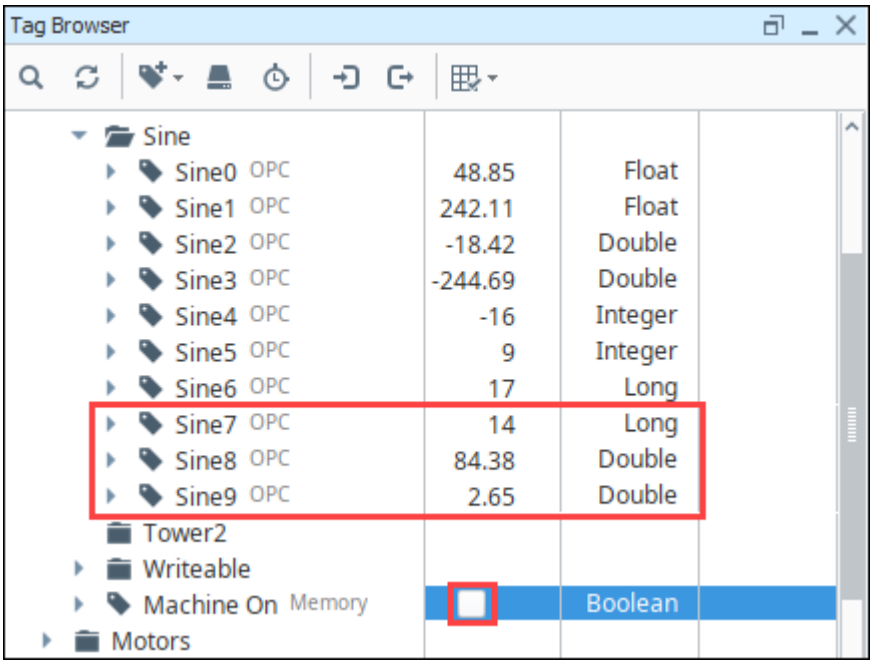
14. The Tag Editor will open showing you are editing multiple Tags. From the Tag Group dropdown list, choose **Drive Machine State**, and click **OK**.



15. Now let's turn the machine on. In the Tag Browser, mark the checkbox next to the **Machine On** Tag that we created earlier.
    Now, when the machine is on, polling will be a at the 1 second rate for the Sine Tags. When the machine is off, polling will be at the 10 second rate for the Sine Tags. Test it by toggling the Machine On Tag on and off.



    You can also set the **Rate** (slow rate) to a value of 0, which means polling is stopped when the condition is false. So for example, when the machine is off, you will see the Tags stay at the last known value.

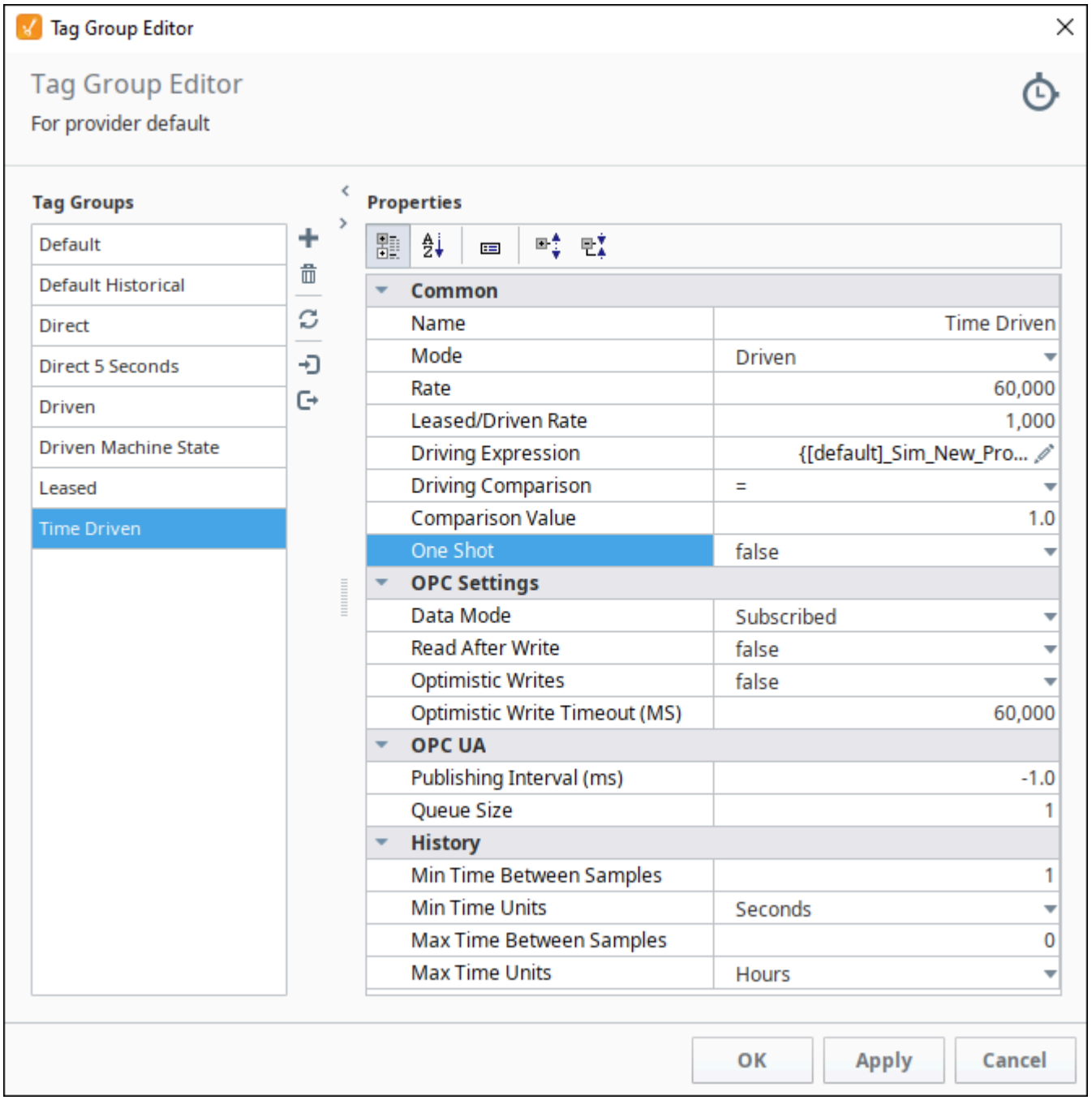


Once the machine is turned back on, the Tags resume polling.

Driven Tag Group - Time of Day

Driven Tag Groups can be used as the polling rate for Tags to trigger at different rates at different times of the day. They can also be used as a one-shot event at a specific time of the day. You can accomplish this by setting the condition of the Driven Tag Group to be true at certain times of the day, and false at other times.

Keep in mind, that if you use a Driven Tag Group based on the time of day with a 0 Slow Rate for history, the Tag will only store history during those specified times.



Add a Driven Tag Group Based on the Time of Day

Let's add a Driven Tag Group that updates the Tags based on a time of day. We will use an Expression to drive the Tag Group. You can use the functions that are in the expression language to poll the PLC during the hours of 8am to 5pm.

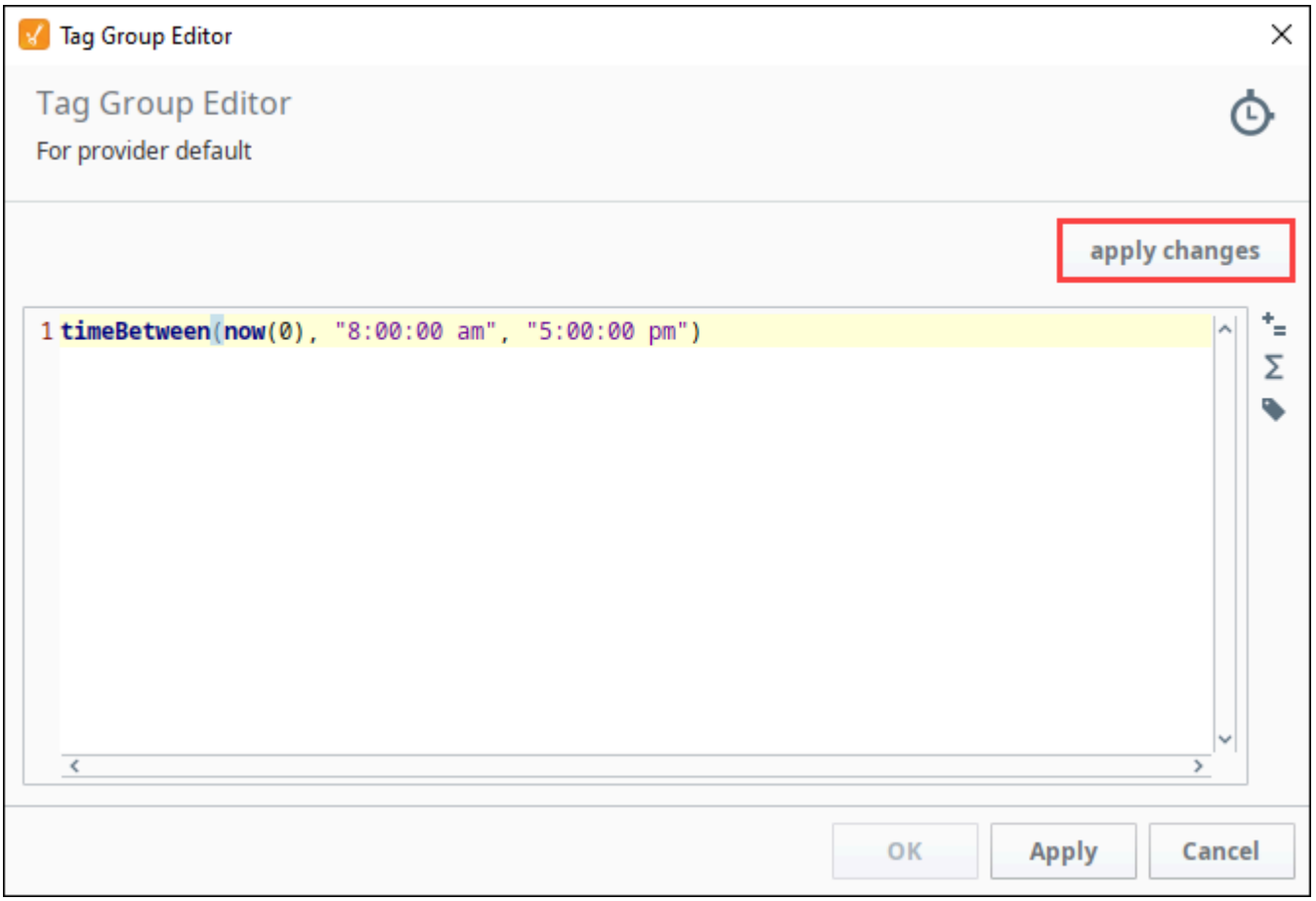
1. In the Tag Browser, click on the **Tag Groups** icon to open the Tag Group Editor.
2. To create a new Tag Group, click on the **Add** icon.
3. Enter the name for the Tag Group. In this example, we named it **Time Driven**, and set the Mode to **Driven**. This Tag Group executes based on the condition of the **Driving Expression**. In this example, we'll use the Poll Time Tag we created above.

4. Set the **Rate** to 60,000ms so it polls at a slow rate, and the **Leased/Driven Rate** to 1,000ms so it polls at a faster rate.

5. Enter the **Driving Expression**. Click on the **Edit** icon, and copy and paste the following expression in the expression box, then click **Apply Changes**.

timeBetween and Machine On expression

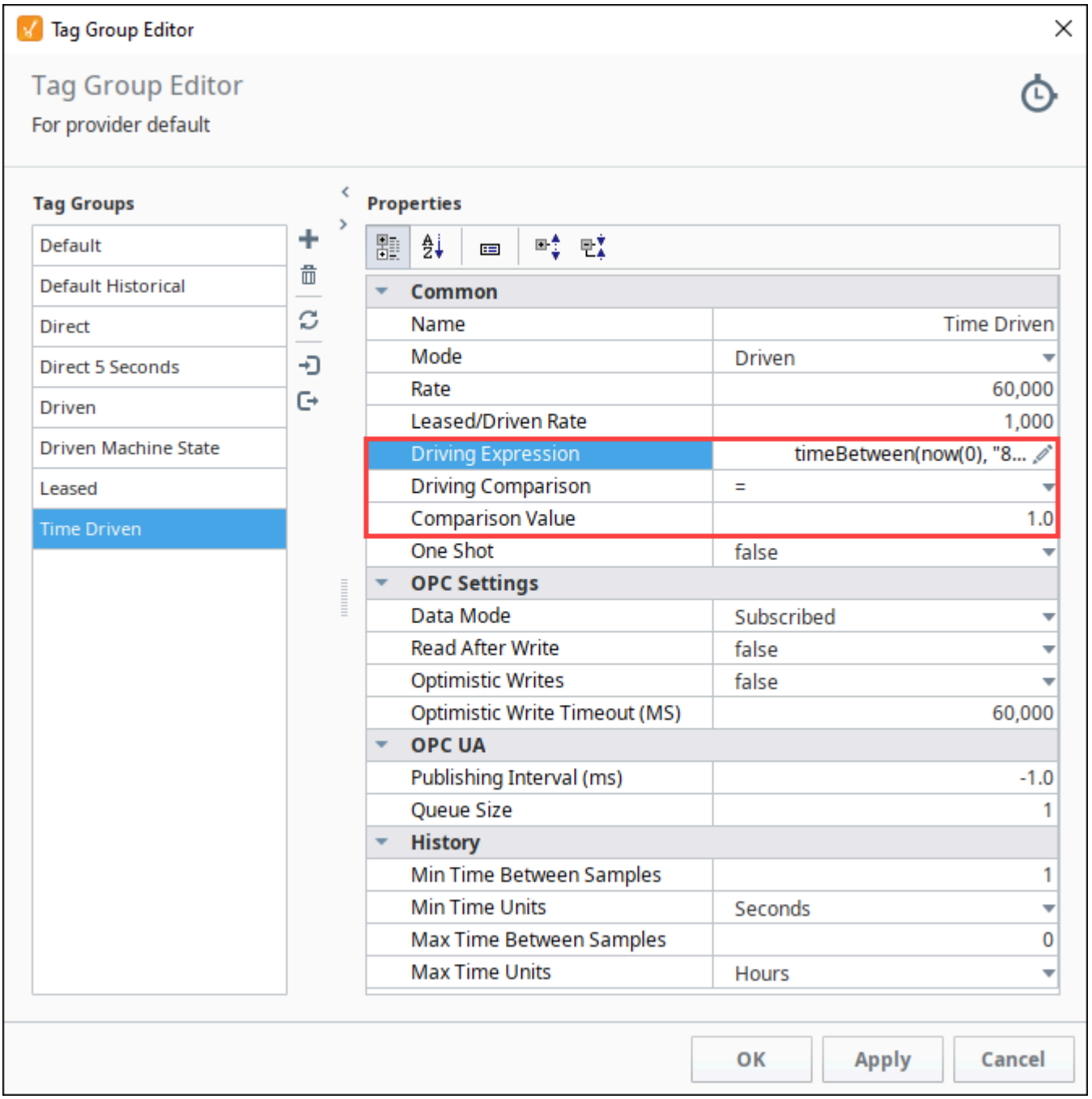
```
timeBetween(now(0), "8:00:00 am", "5:00:00 pm")
```



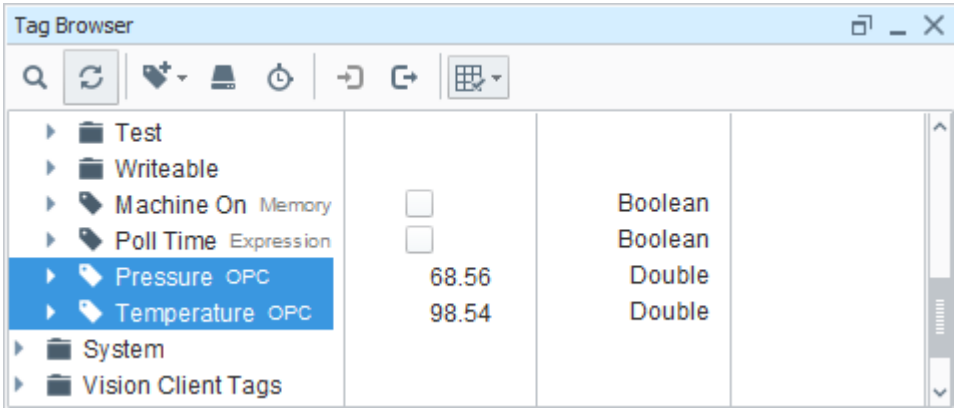
6. While still in the Tag Group Editor, set the following values:

   - Driving Comparison: **=**
   - Comparison Value: **1**

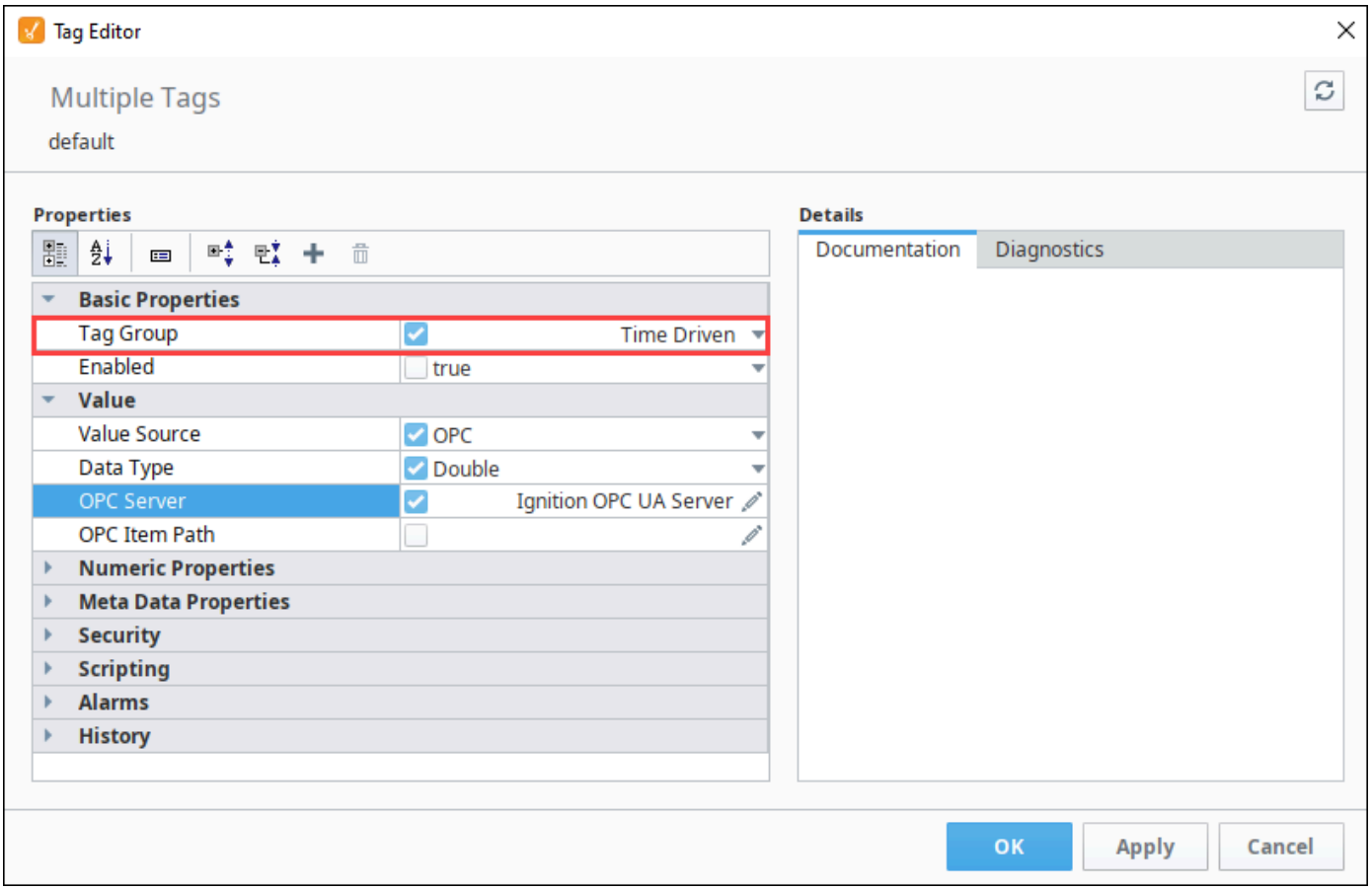
7. Click **OK**.



8. Next, select all the Tags that you want to specify in the Tag Group (i.e., **Pressure** and **Temperature**). Right-click on the selected Tags, and choose **Edit Tag**.



9. The Tag Editor will open showing you are editing multiple Tags. To change the Tag Group, choose **Time Driven** from the dropdown, and click **OK**.



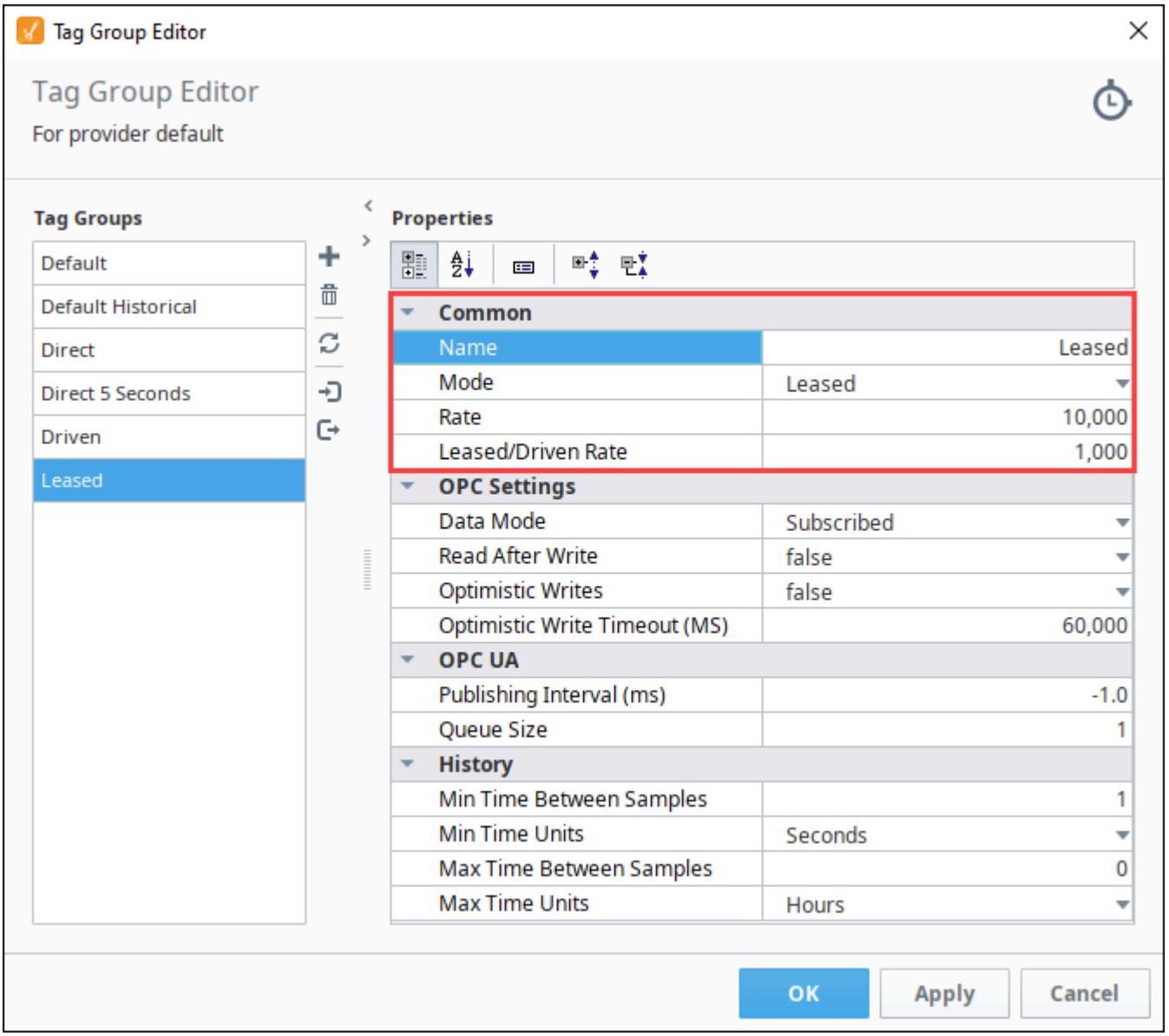
10. In the Tag Browser you can look at the Tags to see they are updating at the correct rate. Try adjusting the time range in your expression to change the rate of polling. When the driving conditions are true, that is the time between hours of 8am and 5pm, polling is at the Leased/Driven Rate of 1000 ms. When not, polling is stopped, and the last known value is displayed.

Leased Tag Group Example

Add a Leased Tag Group

Let's add a new Leased Tag Group that polls a tag from the PLC at a one second rate when someone needs to view that tag in the Client. If the Client is closed, the tags will poll at the standard Rate value.

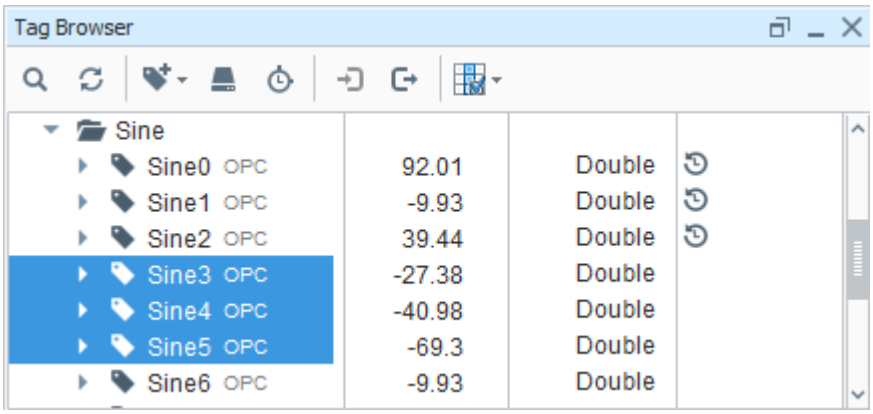
1. In the **Tag Browser**, click on the **Tag Group** icon to open the **Tag Group Editor**.
2. On the left side of the Tag Group Editor window, you can see all existing Tag Groups. Click on the **Add +** icon on the lower left side of the window to add a new Tag Group.
3. Enter the name of the Tag Group. For example, you can name this new Tag Group **Leased**, and set the **Mode** to **Leased**.
4. Set the **Rate** to 10,000ms (10 seconds) and the **Leased/Driven Rate** to 1,000ms (one second).
5. Click **OK**.



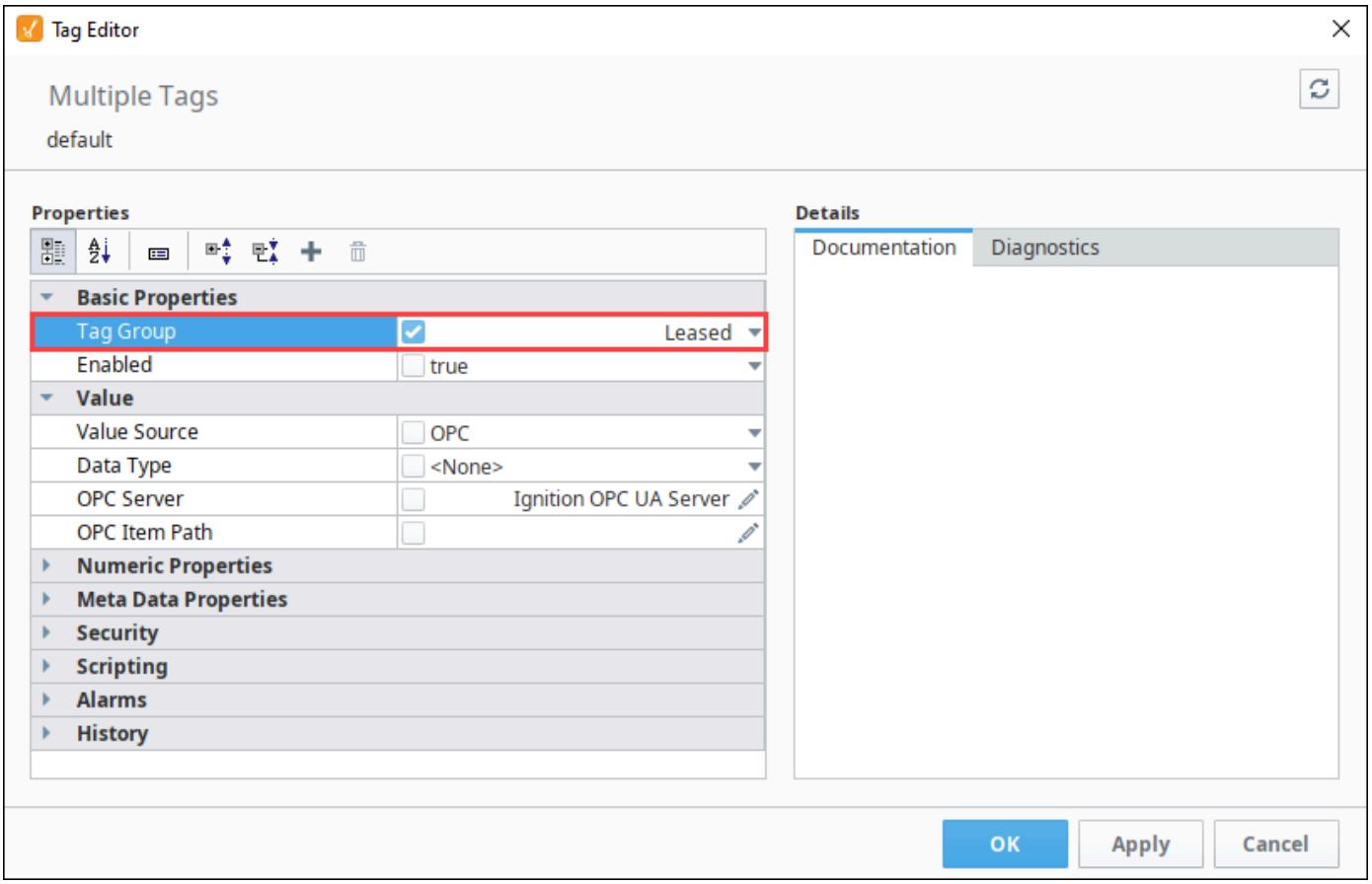
6. Next, select the tags you want to use on the new Leased Tag Group.

    i. Go to your Tag Browser, and select some tags you want to add to the Leased Tag Group. This example uses three Sine tags: Sine3, Sine4, and Sine5.

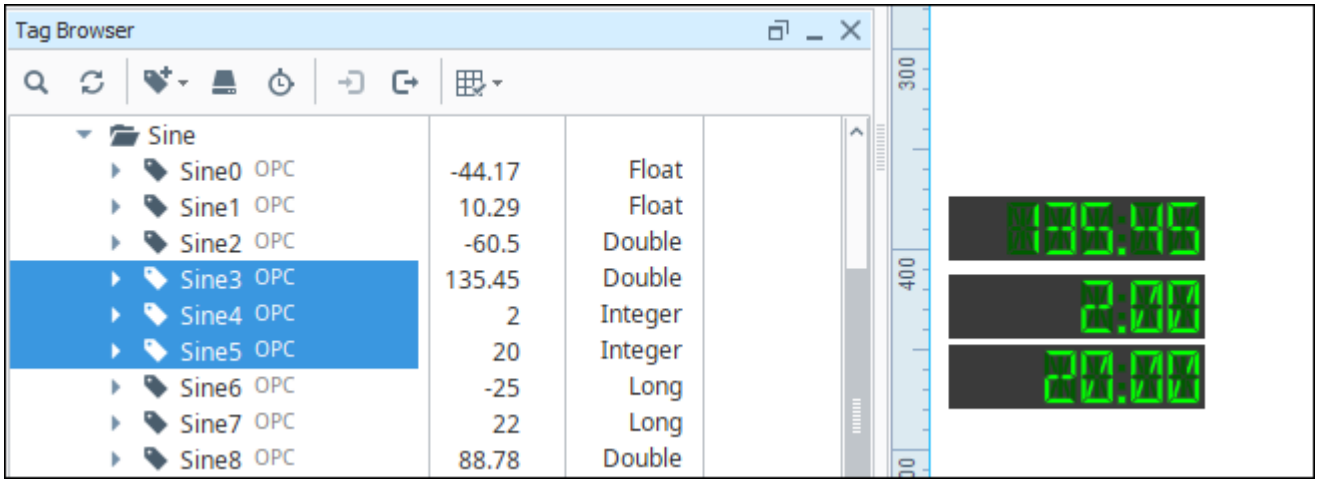
    ii. Right click on the selected tags, and click on **Edit tags**.



7. The Tag Editor will open showing you are editing multiple tags. From the Tag Group dropdown list, choose **Leased**, and click **OK**.



You just created a new Leased Tag Group and added some tags. These tags are only polled quickly when a component that is bound to it is showing in a Client, Session, or when the tags are showing in the Designer. For example, these three Sine tags that are part of the Leased Tag Group are each bound to an LED component.



NOTE

Remember that Tag History is stored based on the current Ignition tag value. If your Leased Tag Group has a **0** slow rate, you will not get updated data to store for history. Always make sure your Tag Group slow speed is at least as fast as your History Tag Group speed.

Tag Providers

At the highest level of tag configuration is the Tag Provider. A provider is a tag database (a collection of tags) and a name. An Ignition Gateway can have any number of Tag Providers, and therefore the name is used to distinguish which provider a tag comes from. Tag Providers can be set up with security or even disabled independent of each other.

Every copy of Ignition has its own tags, spread across one or more Tag Providers. With the Remote Tag Provider, Ignition can also see the tags on another Gateway, as long as the two Gateways are connected through a Gateway network.

All tags reside in a Tag Provider and have realtime values. Additionally, there is the concept of **Tag Historian Providers**, which can store and query historical data for tags. Each tag can optionally have a historian provider assigned to it to whom it will report value changes for historical storage.

Realtime Provider Types

There are two types of Realtime Tag Providers that you can choose from:

Standard Tag Provider

Standard Tag Providers store all configuration and do any execution (read, write, history, alarms) through the local Ignition Gateway. Every new Ignition installation automatically creates a Standard Tag Provider named **default**. You can add as many Standard Tag Providers as you want. This provider can be exposed or hidden from other Gateways on the network through the Gateway's OPC UA settings.

Remote Tag Provider

Remote Tag Provider

Watch the video

Remote Tag Providers connect a remote installation of Ignition and access those tags. The Remote Tag Provider works by creating a link from the local Gateway to a Tag Provider on a remote Gateway using a Gateway Network connection. The local Ignition may be allowed to read and write to the remote tags, but any execution is handled by the remote Gateway. So, things like writing to a PLC, alarms, and history will still be handled by the remote Ignition.

Put another way, "tags" in a Remote Tag Provider are simply representations of the tags as they exist on the remote system. By default, a Remote Tag Provider will fall under the **Default Security Zone** and be read only.

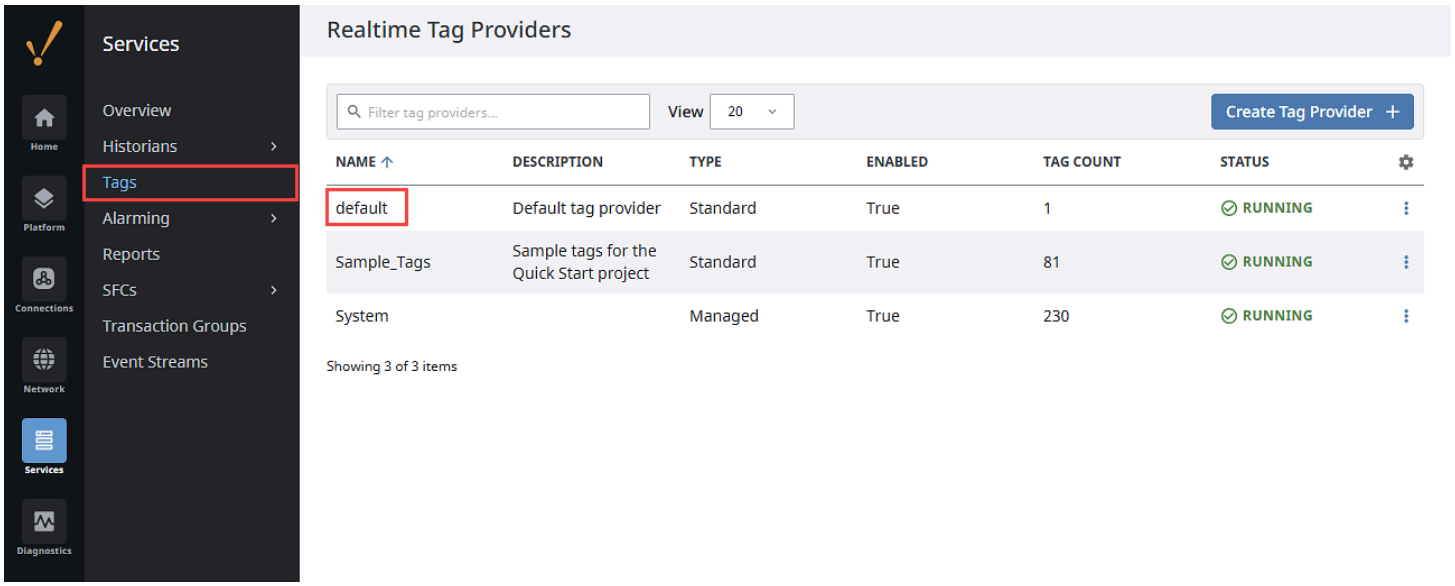
> **NOTE**
>
> Annotations to remote Gateways via Remote Tag Providers are not supported. If you want to send annotations to a remote Gateway, you can instead use a **Remote Historian provider**.

Managed Tag Provider

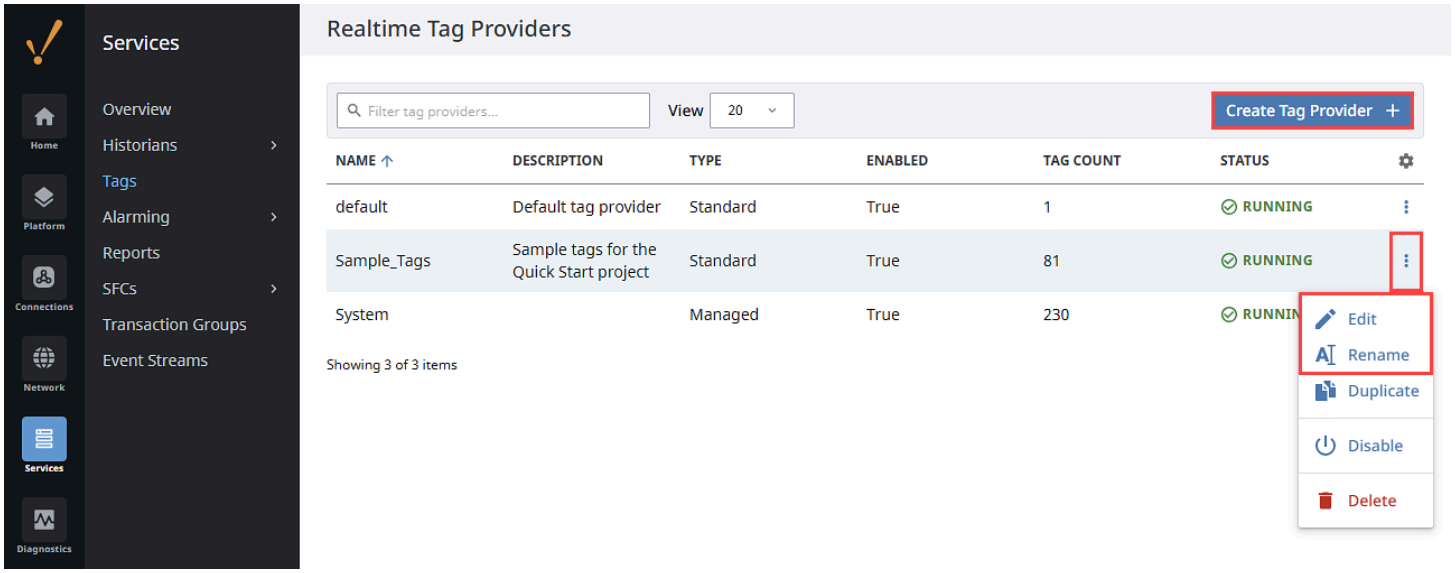
Every Gateway also includes a System Tag Provider, which is configured as a Managed type. The System Tag Provider stores **Client and Gateway tags** that provide status about the Ignition system. Additional Managed Tag Providers cannot be created by users, and the System Tag Provider cannot be deleted or modified. The only option available to edit is enabling or disabling the **Tag Reference Tracker Store**.

Configuring Realtime Providers

Realtime Tag Providers are configured in the Gateway's Services section under **Tags**. The Ignition Gateway will start with a System Tag Provider defined.



You can edit a Tag Provider's name or settings by clicking on the three dots menu to the right and selecting **Rename** and **Edit**, or create new providers by selecting **Create Tag Provider** above the table.



> **NOTE**
>
> When setting up a Realtime Tag Provider, naming your Provider the same name as the database connection is **not** recommended. Doing so may cause no values to be returned when querying tag data.

Standard Tag Provider Settings

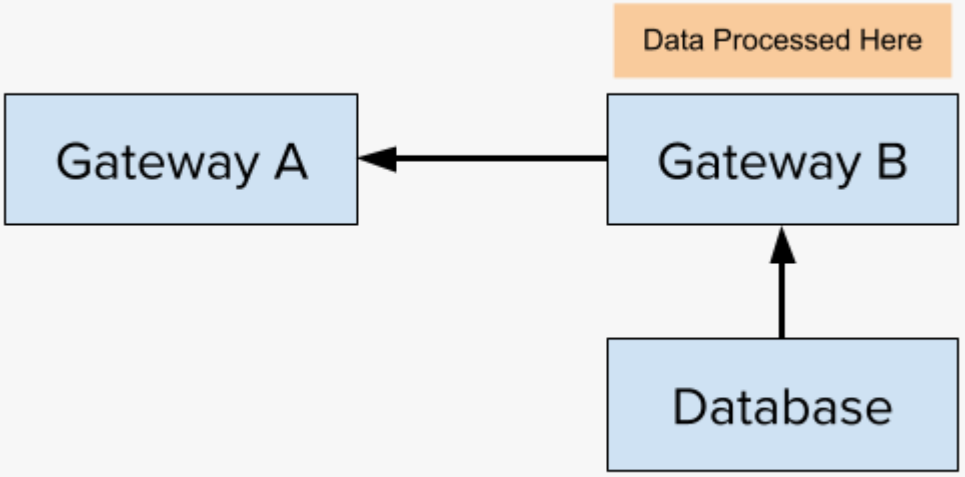
Tags are stored inside of Ignition and executed by the system.

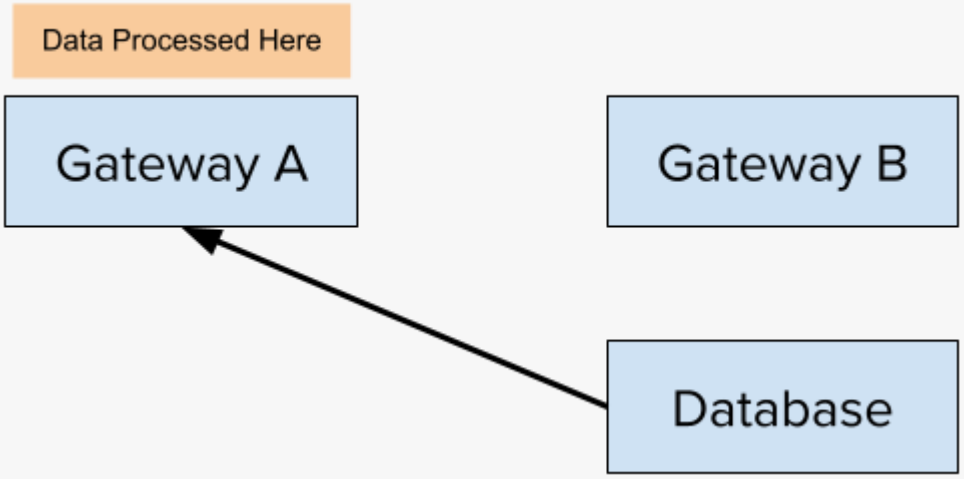
Setting	Description
Name	The name of the provider.
Description	The description of the provider.
Enabled	If true, Tag Provider is enabled. Default is true.
Default Datasource	The default database connection to use for Expression tags that run SQL queries. All Query tags with default database providers selected with run their queries against this database source.
Value Persistence	The behavior for how Memory tag values will be stored. Possible values include: • **None**: Memory tag value changes will persist in the memory only and on Gateway restarts, the value will revert to the configuration value. • **Database**: Memory tag value changes will persist to a SQLite DB and be reloaded on start, but will not modify the configuration value when changed. Note that the configuration file will be modified with an edit to the tag, such as adding an alarm. • **Configuration**: Memory tag value changes will persist to the tag configuration. This option should be used sparingly to avoid performance hits. **Note**: This setting can be overridden for individual Memory tags in the Tag Editor.
Tag Read Permissions	Any actor attempting to read any tag within this tag provider must have these permissions in addition to the target tag's permissions. Multiple security level paths can be specified by separating them with commas. If blank, tag reads for this provider will not be restricted by default. For example: `Authenticated/Roles/Administrator, SecurityZones/MyZone` When multiple security levels are provided, the radio buttons determine if the user needs all of the listed security levels, or at least one.

Setting	Description
Read Only	If enabled, writes to any tag within this Tag Provider will be rejected, regardless of the target tag's permissions. Disabled by default.
Tag Write Permissions	Any actor attempting to write to any tag within this Tag Provider must have these permissions in addition to the target tag's permissions. Multiple security level paths can be specified by separating them with commas. If blank, tag writes for this provider will not be restricted by default. See the example for the Tag Read Permissions setting above.
Tag Editing Permissions	Determines the roles required to edit, create, or delete tags in the provider. Expects a path to a Security Level. Multiple levels can be specified, separated by a comma. For example: `Authenticated/Roles/Administrator, SecurityZones/MyZone` When multiple security levels are provided, the radio buttons determine if the user needs all of the listed security levels, or at least one. Edge Gateways can access the Tag Editing Permissions setting.
Allow Back-fill Data	If enabled, data will be allowed to arrive out of order from the source. Data from the past will be stored to history, but will not be used for alarms, scripts, or subscriptions. If false (default behavior), each value will be processed fully as it arrives. Default is false.
Enable Tag Reference Tracker Store	Enables the storing of Tag Reference entries to a database on the local Gateway for analysis in a Designer. Default is true.

Remote Tag Provider Settings

Tag Provider from one Gateway is brought in to another Gateway.

Setting	Description
Name	The name of the provider.
Description	The description of the provider
Enabled	If true, the Remote Tag Provider is enabled. Default is true.
Gateway	The name of the Gateway on the Gateway Network that this provider is coming from.
Provider	The name of the provider as it is on the remote Gateway. This does not have to be the same as its name on the new Gateway.
History Access Mode	This setting dictates how Tag history is queried for remote tags. Ultimately determines where records are processed. When set to **GatewayNetwork**, history requests will go through the Gateway Network. Meaning the remote Gateway will query records from it's database, process the records, then send the results over to the requesting Gateway.  When set to **Database** history requests will run against the database directly. In this scenario, the requesting Gateway then processes the

Setting	Description
	data. 
History Datasource	The data source to query when History Access Mode is set to Database.
History Driver	If querying the database directly, this is the Gateway name of the remote system. It is used to identify data from that system in the database.
History Provider	If querying the database directly, this is the name of the Tag Provider on the remote system. It is used along with driver name to identify the correct tags in the database.
Alarms Enabled	If true, alarms configured on the remote Gateway will be enabled on the new Gateway.
Alarm Mode	How alarm states should be monitored. Options include: • **Queried**: The alarm state will be queried through the Gateway network when necessary. • **Subscribed**: The alarm state will be subscribed, and updates will be sent asynchronously. Subscribed provides better performance, but uses more memory.
Allow Back-fill Data	If enabled, data will be allowed to arrive out of order from the source. Data from the past will be stored to history, but will not be used for alarms, scripts, or subscriptions. If false (default behavior), each value will be processed fully as it arrives. Default is false.

Setting	Description
Enable Tag Reference Tracker Store	Enables the storing of Tag Reference entries to a database on the local Gateway for analysis in a Designer. Default is true.

Tag Event Scripts

Scripts can be attached to tags. When you edit a tag, you can navigate to the Tag Events section and click on the **Edit** icon to see a list of all of the available events. Those events are

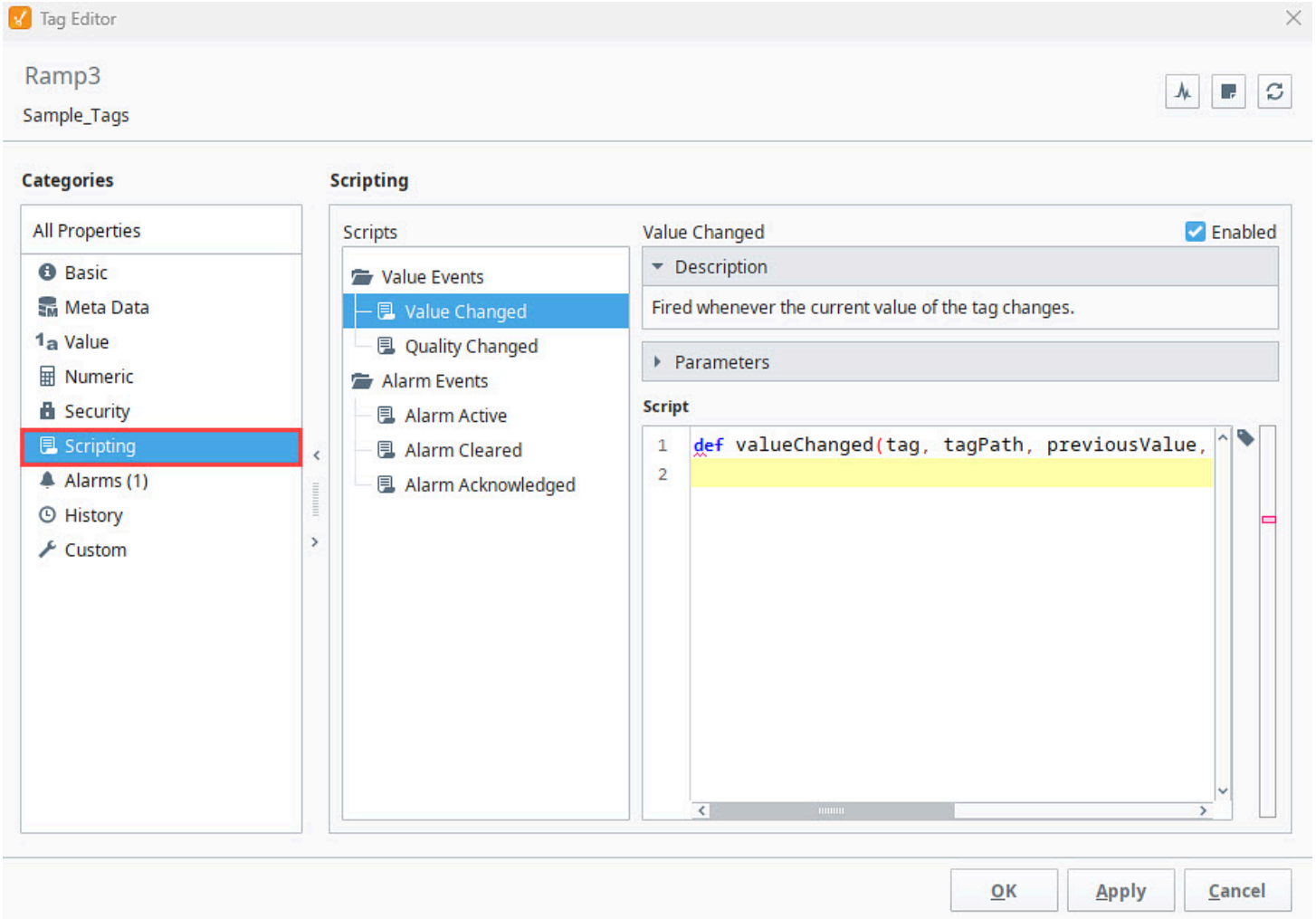
- Value Changed
- Quality Changed
- Qualified Value
- Alarm Active
- Alarm Cleared
- Alarm Acknowledged

Because tags are stored in the Gateway, they aren't scoped to a specific project. In these cases, some system functions may require that you specify a project or other resource, such as a database for the **system.db.runPrepUpdate** function.

> **NOTE**
>
> Because these scripts are Gateway scoped, certain things like print statements will not print to the Designer console, but will print instead to the wrapper log file in Ignition's installation directory. The preferred approach to adding logging statements to Tag Event Scripts is to use the **system.util.getLogger** function, which will send the messages to the **Gateway's Logs** page.

Event Options

Once an event has been selected, note that the top of the text area is **defining a function**. As a result, all code that should execute when this event occurs should be indented at least once to be included in the definition.



Value Changed

The Value Changed event is fired whenever the value of the particular tag changes.

This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- Object previousValue: The previous value. This is a qualified value, so it has value, quality, and timestamp properties.
- Object currentValue: The current value. This is a qualified value, so it has value, quality, and timestamp properties.
- Boolean initialChange: A boolean flag indicating whether this event is due to the initial subscription or the first execution after a tag update. The valueChanged event will have an initialChange of true if either:
  - The tag's previous value was null, as in the first evaluation of an expression tag after a Gateway restart
  - Its previous quality was **Uncertain_InitialValue**. This is typically an OPC tag's initial value before a true value is delivered from a subscription.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

> **NOTE**
>
> The currentValue and previousValue arguments are qualified values: objects that contain a value, timestamp, and quality. This means that to get to the value of the currentValue, your script would need to access `currentValue.value`. Refer to **Scripting Object Reference** for more information about qualified values.

Quality Changed

The Quality Changed event is fired whenever the quality of the tag changes. This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- Object previousValue: The previous value. This is a qualified value, so it has value, quality, and timestamp properties.
- Object currentValue: The current value. This is a qualified value, so it has value, quality, and timestamp properties.
- Boolean initialChange: A boolean flag indicating whether this event is due to the initial subscription or the first execution after a tag update.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

Qualified Value

The Qualified Value event is fired whenever the value, quality, or timestamp of the tag changes, depending on the change trigger enabled on the script:

- Value
- Quality
- Timestamp

This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- Object previousValue: The previous value. This is a qualified value, so it has value, quality, and timestamp properties.
- Object currentValue: The current value. This is a qualified value, so it has value, quality, and timestamp properties.
- Boolean initialChange: A boolean flag indicating whether this event is due to the initial subscription or the first execution after a tag update.

- Object changed: Information about which aspects of the qualified value (value, quality, timestamp) changed to trigger the event.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

Alarm Active

The Alarm Active event fires whenever a new active alarm event occurs. This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- String alarmName: The name of the alarm. This does not include the full alarm path.
- Object alarmEvent: The full alarm event object. The properties available to this object are:
  - eventId
  - source
  - name
  - priority
  - displayPath
  - displayPathOrSource (the display path if it is set, otherwise the source)
  - state (the current state, active/cleared + acked/unacked)
  - eventState (the last transition, active, clear, acknowledged)
  - isClear (a boolean if the alarm is currently cleared)
  - isAcked (a boolean if the alarm is currently acknowledged)
  - isShelved (a boolean if the alarm is currently shelved)
  - notes
- String alarmPath: The full alarm path.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

Alarm Cleared

The Alarm Cleared event fires whenever an alarm becomes cleared on the tag. This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- String alarmName: The name of the alarm. This does not include the full alarm path.
- Object alarmEvent: The full alarm event object. See the **Alarm Active alarmEvent object** for the full list of available properties.

- String alarmPath: The full alarm path.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

Alarm Acknowledged

The Alarm Acknowledged event fires whenever an alarm event becomes acknowledged on the Tag. This event has a variety of arguments available for use in the script:

- String tagPath: The full path to the tag. Example: `[tagProvider]Folder/Folder/Tag`
- String alarmName: The name of the alarm. This does not include the full alarm path.
- Object alarmEvent: The full alarm event object. See the Alarm Active alarmEvent object for the full list of available properties.
- String alarmPath: The full alarm path.
- String ackedBy: The full path to the user that acknowledged the alarm.
- Boolean missedEvents: A flag indicating that some events have been skipped due to an event overflow.

Using the Project Library in a Tag Event Script

Scripts defined in a project script can be called from a Tag Event Script. However, only scripts defined in the Gateway Script Project may be used. For more information on configuring the Gateway Scripting Project, see the **Project Library** page.

UDT Parameters in Tag Event Scripts

Parameters on UDTs can be interacted with from Tag Event Scripts by accessing the parameter values as dictionary values. The script will access the most recent parameter values on a UDT, where the parameters can then change without needing to edit the UDT or restart the tag.

For example, if you are trying to access the value of a parameter named "myParam" from a Tag Event Script within the UDT, the script would look like:

```
# Reminder: "tag" is a built-in argument on all Tag Event Scripts.
# Accessing the "parameters" key on the tag argument will
# provide access to all UDT parameters.
paramValue = tag['parameters']['myParam']
```

Using the approach described above, value changes to UDT parameters will be reflected in subsequent calls.

Tag Script Examples

Printing All Parameters

```
# This script will fetch all of the possible parameters in the Tag Changed Script.
# Note that this will not print out to the console, but it will print to the
wrapper logs located on the Gateway.

path = tagPath
prev = previousValue
cur = currentValue
init = initialChange
missed = missedEvents
print path, prev.value, cur.quality, init, missed
```

Automatic Reset

```
# This code can be used on a Value Changed script, and will automatically reset
the value of the tag to 0 after it goes above 3000.
# This can be useful for counter tags.
if currentValue.value > 3000:
    system.tag.writeBlocking(["[default]IntegerCounterTag"], [0])
```

Copy to Another Tag

```
# This code can be used on a Value Changed script, and will record the highest
value of the current tag onto another memory tag.
# This can be useful for recording the highest temperature.
highestRecordedTemp = system.tag.readBlocking(["[default]HighestTempTag"])
[0].value
if currentValue.value > highestRecordedTemp:
    system.tag.writeBlocking(["[default]HighestTempTag"], [currentValue.value])
```

Troubleshooting

It may be helpful when troubleshooting or testing Tag Event Scripts to increase the default thread pool count, as reaching the thread pool limit may cause scripts to not run as expected. Refer to the

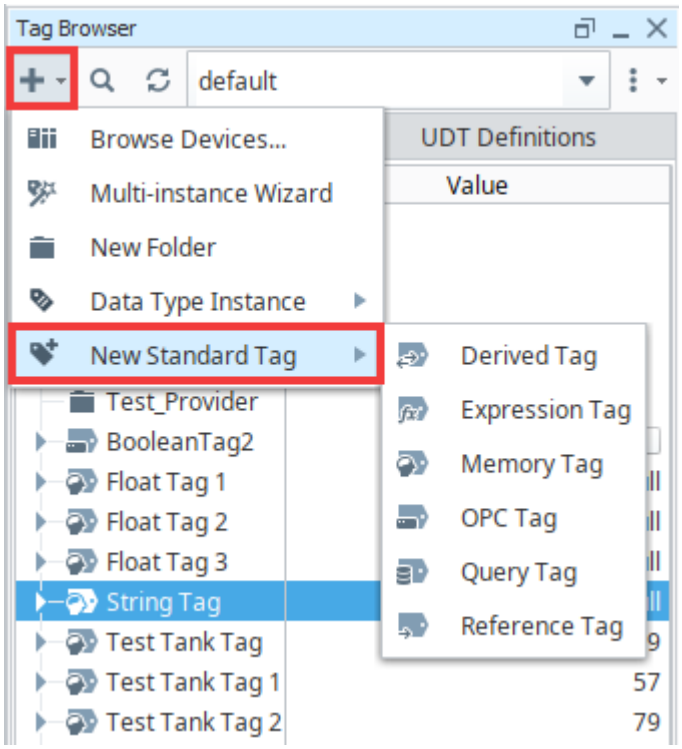
Gateway Configuration File Reference - Thread Pool Counts for more information.

Tag Properties

Tags are points of data and may have static values or dynamic values that come from an OPC address, an Expression, or a SQL query. The values can be used on screens and in Transaction Groups. Additionally, tags offer a core set of features above and beyond simple values, such as scaling, alarming, and history logging. Depending on the specific **type of tag**, even more options are available. In general, tags provide a common interface for tying together many types of data in Ignition.

Tag Configuration in the Designer

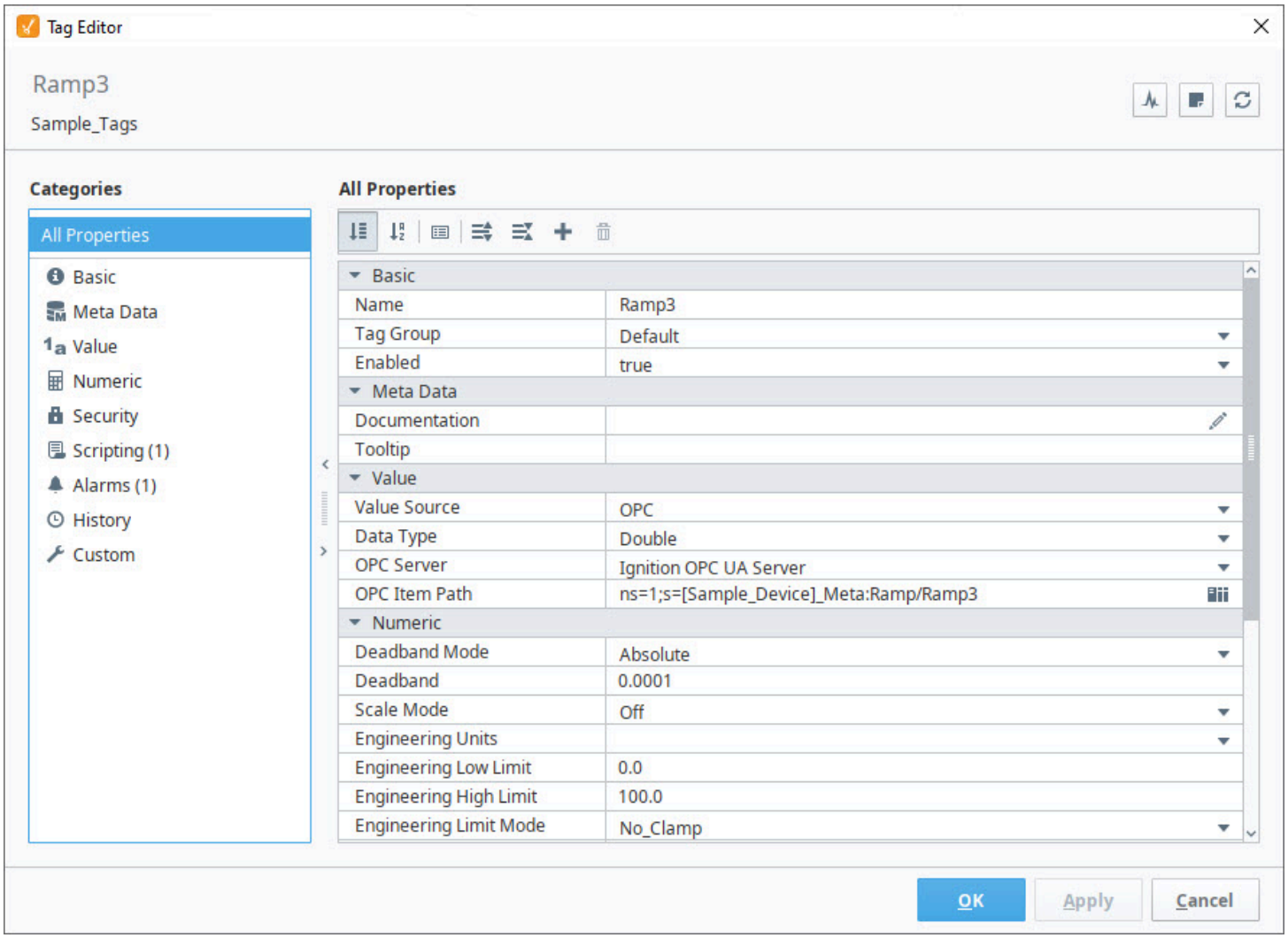
Tags are managed in the Tag Editor. To configure a tag, right-click on it and select **Edit Tag**. Or create a new tag by right-clicking on the Tags folder in the Tag Browser and use the **New Tag** option to select a new tag type.



Once the **Tag Editor** window is displayed you can set the properties for that tag. The **Tag Editor** window has the following sections depending on the type of tag you are creating:

- Basic Properties
- Value
- Numeric Properties
- Meta Data Properties
- Security

- Scripting
- Alarms
- History



Tag Configuration on the File System

Tag configurations are grouped by the folder they exist in and are stored in the following locations:

- Atomic Tags: `/data/config/core/ignition/tag-definition/<provider>/<path to folder>/tags.json`
- UDT Instances: `/data/config/core/ignition/tag-definition/<provider>/<path to folder>/udts.json`
- UDT Definitions: `/data/config/core/ignition/tag-type-definition/<provider>/<path to folder>/udts.json`

Tag Object Types

Some features, such as **system.tag.browse**, can access the Object Type of the tag (sometimes called "tagType"). Below is a table representing the possible types.

Tag Object	Description
`Property`	A single value underneath an node.
`Node`	An entity that may have a value and may have children. Node is a generic term for other objects in this table, such as a Folder or AtomicTag.
`Folder`	Represented by a folder in the Tag Browser. Folders generally have child nodes, but don't have values or other properties that make up a tag.
`AtomicTag`	A *normal* type of tag. Objects with this type can be one of the following (based on the Value Source property): • OPC tag • Query tag • Expression tag • Derived tag • Reference tag • Memory tag
`UdtInstance`	An instance of a complex tag (UDT Instance). It's important to note that UdtInstances contain other nodes, so this type is generally only seen at the root of a UDT instance. Thus, nodes under a UdtInstance are not considered to have a type of `UdtInstance`, unless the child node is actually a UdtInstance, in other words, a nested UDT instance.
`UdtType`	Represents the root of a complex tag definition (UDT Definition). Similar to UdtInstance, nodes under a UdtType have their own object type, so a UdtType represents the root of a complex tag.
`Provider`	Represents a Tag Provider.

Standard Tag Properties Table

This following table provides detail on each Tag Property, including the binding name, description, data type, and the tag types that use the property.

Basic Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Name	`name`	How the tag will be presented and referenced in the system. The tag path will be the provider, the folder structure, and this name. Valid names begin with either a letter or underscore and can contain: • Letters • Digits • Underscores • Spaces • Parentheses • Single quotes • Dashes • Colons Additionally, the name must be less than 256 characters in length.	String	OPC, Query, Expression, Derived, Client, Reference, Memory
Tag Group	`tagGroup`	The Tag Group that will execute the tag. The Tag Group dictates the rate and conditions on which the tag will be evaluated. For more details, see Tag Groups.	String	OPC, Query, Expression, Derived, Reference, Memory
Enabled	`enabled`	Whether the tag will be evaluated by the Tag Group. If false, the tag	Boolean	OPC, Query, Expression, Derived,

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		will still be present, but will not return a value or good quality. Default value is **Enabled**.		Reference, Memory

Value Properties

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
Tag Type (unlisted)	`tagType`	The type of the node, automatically determined by the tag type created. See the **Tag Object Types** table for more information. Commonly used values are folder, UdtInstance, UdtType, and AtomicTag. Default value is **AtomicTag**. **Note**: This property does not appear in the Tag Editor, but is accessible via scripting.	String	OPC, Query, Expressio Derived, Reference Memory
Type ID	`typeId`	Returns a path representing which UDT this instance is derived from. If the node is not a UDT, then this property will return a None object. **Note**: This property appears in a UDT Definition/Instance editor as **Parent Data Type**. Additionally, parent	String	OPC, Query, Expressio Derived, Reference Memory

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		data types can only be set from UDT Definitions within the same provider.		
Value Source	`valueSource`	Specifies how the tag determines its value. In other words, sets the type of the tag (Memory, OPC, Expression, etc). Below is a list of possible values and their JSON representation: • **Derived** - `derived` • **Expression** - `expr` • **Memory** - `memory` • **OPC** - `opc` • **Query** - `db` • **Reference** - `reference` Default value is **Memory**.	String	OPC, Query, Expressio Derived, Reference Memory
Data Type	`dataType`	The data type of the tag. It is important that this be set as correctly as possible with regards to the tag's underlying data	String	OPC, Query, Expressio Derived, Reference Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		source. The tag system will attempt to coerce any raw incoming value (for example, from OPC or a SQL query) into the desired type. For detailed information and a list of possible values, see **Tag Data Types**. Default value is **Integer**. **Note**: Regarding Array data types, Alarming, Scaling, and Historical settings applied to an array tag are propagated down to elements in the array.		
Default Value	`defaultValue`	The default value of an individual Memory tag, which typically only applies on creation when a value doesn't exist within the Value Persistence Database or Configuration behavior. This property will always	Object (depends on the data type of the tag)	Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		apply if Value Persistence is set to None, since no value is ever stored. Note that if the Default Value is not set and the Value property is set, the Default Value will inherit the Value property.		
Value	`value`	The value of the tag. Can only be modified if the tag allows value writing and the user has sufficient privileges. If the Default Value is set for individual tags, and this Value is not, the Default Value will be inherited. **Note**: In UDT instances, certain types of tag values, such as DB and OPC, can get stored within the definition, causing a tag to appear to have an override when it actually doesn't.	Object (depends on the data type of the tag)	Memory

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
Value Persistence	`valuePersistence`	The behavior for storing a Memory tag's value. This property defaults to **Inherited**, which means the memory tag will use the Value Persistence option set on the Tag Provider. Possible values to override the Tag Provider setting include: • **None**: Memory tag value changes will persist in the memory only and on Gateway restarts, the value will revert to the Default Value property. • **Database**: Memory tag value changes will persist to a SQLite DB and be reloaded on start, but will not modify the Default Value in the configuration	String	Memory

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		file when changed. Note that the configuration file will be modified with an edit to the tag, such as adding an alarm. • **Configuration**: Memory tag value changes will persist to the tag configuration. This option should be used sparingly to avoid performance hits.		
OPC Server	`opcServer`	The server against which to subscribe the data point. The list displayed is the **OPC Clients in Ignition** section.	String	OPC
OPC Item Path	`opcItemPath`	The path to the node to subscribe to via the OPC Client. The point will be subscribed at the rate dictated by the Tag Group.	String	OPC

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		It's possible to escape curly braces in the item path by using additional curly braces. For example: `{{device_name}}` would evaluate to `{<device_name value>}`, allowing you to include braces in the Item Path.		
Source Tag Path	`sourceTagPath`	The path to the tag that this tag is referencing.	String	Derived, Reference
Execution Mode	`executionMode`	Determines how and when the tag executes. Possible values are listed below along with JSON names and descriptions: • **Event Driven** `EventDriven`: Updates when something happens (i.e., value event or alarm event) within the expression. • **Fixed Rate** `FixedRate`:	String	Expressio Query

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		Tag will be executed at the set or fixed rate. Adds the Execution Rate property, which determines how often the tag executes in milliseconds. • **Tag Group** `TagGroupRate`: Tags are executed by Tag Groups, which dictate the rate of execution. Default value is **Event Driven**. **Note**: In certain tag types such as DB, the default value may not make sense as there are no events generated via a change. In these cases, a different execution mode should be set.		
Expression	`expression`	The expression the tag will use to determine its value.	String	Expressio

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
Read Expression	`deriveExpressionGetter`	The expression that determines how the value on the Derived tag is read from the source tag.	String	Derived
Query	`query`	The SQL query to be run, which drives the tag's value. Queries doing database reads and writes are possible, see the **Query Type** property description for details.	String	Query
Write Expression	`deriveExpressionSetter`	The expression that determines how the value on the Derived tag is written to the source tag.	String	Derived
Datasource	`datasource`	The database connection that the Query tag will execute against. The list displayed is the **DB connections** section.	String	Query
Query Type	`queryType`	Defines whether the query is executing a	String	Query

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		database read or a database write. Important for determining the **value** behavior of the tag. Possible values are: • **AutoDetect** - Query type is determined from the query itself. • **Select** - Dictates that the query is reading data from the database. The query result set will be stored on the tag's value. • **Update** - Dictates that the query is writing data to the database (but does not require an UPDATE statement in the query, specifically). The value on the Query tag will be the		

Property	JSON/Scripting Name	Description	Data Type	Applicabl Tag Type
		number of affected rows.		
Preserve Source Timestamp	`preserveSourceTimestamp`	**New in 8.3.1** If true, the derived value will use the timestamp from the source value. If false, the derived value will use the current time as the timestamp.	Boolean	Derived

Numeric Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Deadband	`deadband`	A numerical value used to prevent unnecessary updates for tags whose values change by small amounts.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Deadband Mode	`deadbandMode`	Defines how the deadband value is used. Possible values along with JSON names and descriptions are listed below: • **Absolute** `Absolute` : The deadband setting is considered to be an absolute value. • **Percent** `Percent` : The actual deadband is calculated as a percent of the tag's engineering unit span, including **Engineering Low** and **Engineering High**. • **Off** `Off` : The deadband setting is the equivalent to a value of 0.0, so that all values pass through if their timestamp has changed. Default value is **Absolute**.	String	OPC, Query, Expression, Derived, Reference, Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Scale Mode	`scaleMode`	If and how the tag value will be scaled between the source, and what is reported for the tag. A listing of possible values along with their JSON names and numerical representation can be found below: • **Off** `Off`: 0 • **Linear** `Linear`: 1 • **Square Root** `SquareRoot`: 2 • **Exponential Filter** `ExponentialFilter`: 3 • **Bit Inversion** `BitInversion`: 4 Default value is **Off**. **Note**: When using system functions like system.tag.configure to set the Scale Mode, the JSON string value must be used.	String	OPC, Query, Expression, Derived, Reference, Memory
Raw Low	`rawLow`	Start of the "raw" value range. Only present if **Scale Mode** is set to **Linear** or **Square Root**.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Raw High	`rawHigh`	End of the "raw" value range. Only present if	Numeric	OPC, Query, Expression,

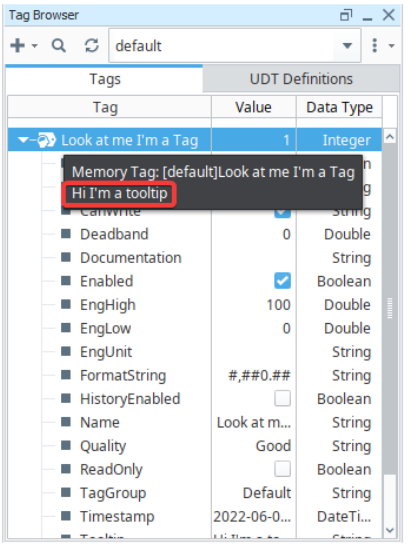
Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		Scale Mode is set to **Linear** or **Square Root**.		Derived, Reference, Memory
Scaled Low	`scaledLow`	Start of "scaled" value range. Raw low will map to Scaled low for the tag. Only present if **Scale Mode** is set to **Linear** or **Square Root**.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Scaled High	`scaledHigh`	End of "scaled" value range. Raw high will map to Scaled high for the tag. Only present if **Scale Mode** is set to **Linear** or **Square Root**.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Clamp Mode	`clampMode`	How values that fall outside of the ranges will be treated. Clamped values are only present if **Scale Mode** is set to **Linear** or **Square Root**, and will be adjusted to the low/high scaled value as appropriate. Possible values along with their JSON names and numerical representation are listed below: • **No_Clamp** `No_Clamp` : 0 • **Clamp_Low** `Clamp_Low` : 1 • **Clamp_High** `Clamp_High` : 2	String	OPC, Query, Expression, Derived, Reference, Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		• **Clamp_Both** `Clamp_Both` : 3 Default value is **No_Clamp**.		
Scale Factor	`scaleFactor`	The factor parameter for the equation, used when the **Scale Mode** property is set to **Exponential Filter**	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Engineering Units	`engUnit`	The engineering units of the value, which can be manually entered or selected from a list. Some Vision components, such as the Numeric Label, will automatically show these units.	String	OPC, Query, Expression, Derived, Reference, Memory
Engineering Low Limit	`engLow`	The lowest expected value of the tag. This property can be leveraged for use by: • Out of Range alarms • Engineering Limit modes • History Deadband mode calculations In some cases, using drag and drop onto Vision components may automatically bind this property to the `Minimum` property of the component.	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Engineering High Limit	`engHigh`	The highest expected value of the tag. This property can be leveraged for use by: • Out of Range alarms • Engineering Limit modes • History Deadband mode calculations In some cases, using drag and drop onto Vision components may automatically bind this property to the `Maximum` property of the component.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Engineering Limit Mode	`engLimitMode`	Dictates how the engineering range should be enforced on the tag. If not "Off", the tag will change to bad quality ("limit exceeded"), when the value falls outside the specified range. Possible values along with their JSON names and numerical representations are listed below: • **No_Clamp** `No_Clamp` : 0 • **Clamp_Low** `Clamp_Low` : 1 • **Clamp_High** `Clamp_High` : 2	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		• **Clamp_Both** `Clamp_Both` : 3		
Format String	`formatString`	How the value should be formatted when converted to a string (only applies to numerical data types). Uses # and 0 characters to describe the format. • #: If the number in this position is non-zero, then do not show the position. Otherwise, show the number. Useful when you only want to show a decimal position if the value is non-zero. • 0: If the number in this position is non-zero, then show that number. Otherwise, show a zero. Useful to add leading and trailing zeros to a value. See **Data Type Formatting Reference** for more information. In some cases, using drag and drop onto Vision components may automatically bind this property to the `Decimal Format` property of the component.	String	OPC, Query, Expression, Derived, Reference, Memory

Meta Data Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Tooltip	`tooltip`	The tooltip provides a hint to visual components as to what should be displayed when the user hovers their mouse cursor over the component that is being driven by the value of this tag. In some cases, using drag and drop onto Vision components may automatically bind this property to the `Mouseover Text` property of the component. Hovering over the tag itself in the **Tag Browser** will also display this hint. 	String	OPC, Query, Expression, Derived, Reference, Memory
Documentation	`documentation`	A freeform text property for information about the tag.	String	OPC, Query, Expression, Derived, Reference, Memory

Security Data Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Read Permissions	`readPermissions`	Defines the security levels required in order to read values from a tag. Roles under authenticated will show up based on property **Security Levels** being configured. For more information, see **Tag Security Properties**. See the **Permission Values** section for a description of possible values.	JSON Object	OPC, Query, Expression, Derived, Reference, Memory
Read Only	`readOnly`	Defines whether a tag is read-only or writeable. For more information, see **Tag Security Properties**.	Boolean	OPC, Query, Expression, Derived, Reference, Memory
Write Permissions	`writePermissions`	Defines the security levels required in order to read values from a tag. Roles under authenticated will show up based on property **Security Levels** being configured. For more information, see **Tag Security Properties**.	JSON Object	OPC, Query, Expression, Derived, Reference, Memory

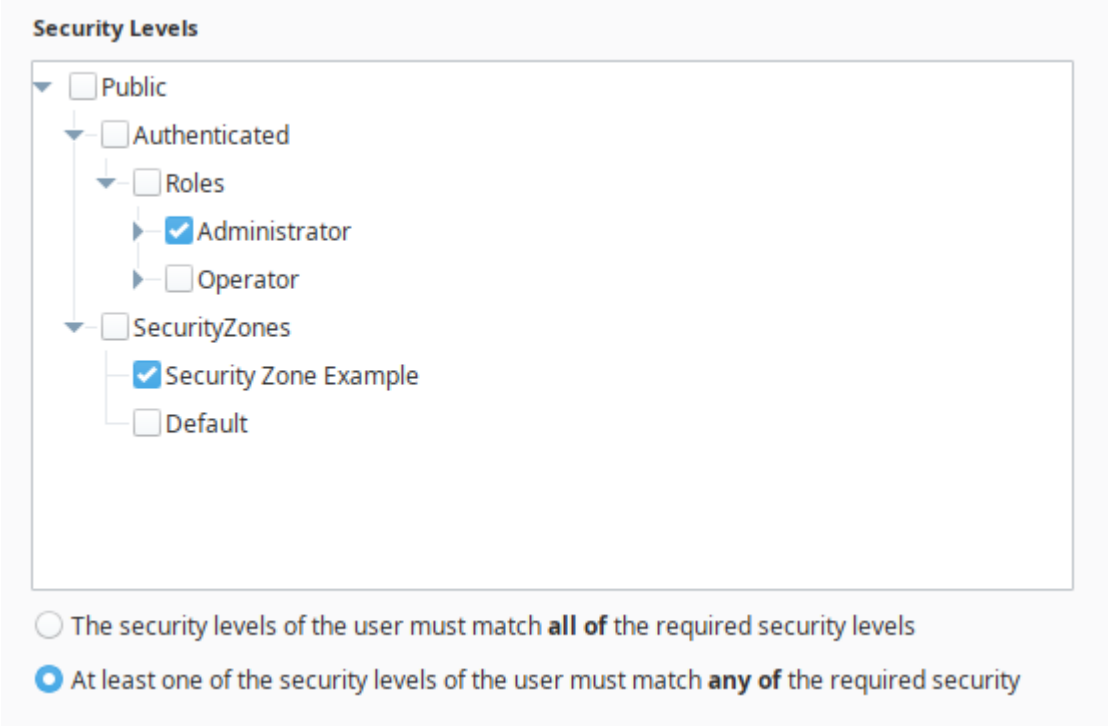
Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
		See the **Permission Values** section for a description of possible values.		

Permission Values

Mode	JSON/Scripting Name	Description
Type	`type`	Represents the selected radio button on the security level UI, determining if all of the elements in the securityLevels array are required, or if any of the elements are allowed. Possible values are: • `AnyOf` • `AllOf` In other words, a user logging in must satisfy either any of the selected criteria or all criteria selected.
Security Levels	`securityLevels`	Represents allowed security levels for this permission. Each level is represented as a JSON object, containing a "name" value that represents the name of a security level, and a "children" array which represents any levels under the current. The actual "selected" levels are any levels that have an empty "children" object. See the example below for more information.

JSON Example

The JSON in this example uses the configuration shown in the image below. Permission is granted if the security levels on the request are from either an "Administrator" user, or if the request originated from the "Zone A" Security Zone.




```
"readPermissions": {
    "type": "AnyOf",
    "securityLevels": [
      {
        "name": "Authenticated",
        "children": [
          {
            "name": "Roles",
            "children": [
              {
                "name": "Administrator",
                "children": []
              }
            ]
          }
        ]
      },
      {
        "name": "SecurityZones",
        "children": [
          {
            "name": "Zone A",
            "children": []
          }
        ]
      }
    ]
  }
```

Scripting Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Tag Event Scripts	`eventScripts`	Each tag has the option to have Tag Event Scripts on it. When you edit a tag, you can navigate to the Tag Events screen to see a list of all of the tag scripts. You can then select which event you would like to write a script for. You can even write a script for multiple events if you like. For detailed information, see **Tag Event Scripts**. When interacting with a tag from a script, the Tag Event Scripts are represented as an array of JSON objects. Each JSON object is described below in the **Key Description** section.	JSON Array	OPC, Query, Expression, Derived, Reference, Memory

Key Description

Key	Description
eventid	A value representing the type of event script. See values below for a list of possible events. • Quality Changed: `qualityChanged` • Value Changed: `valueChanged` • Alarm Active: `alarmActive` • Alarm Cleared: `alarmCleared` • Alarm Acknowledged: `alarmAcked`
script	A value representing the content of the script

Alarms Properties

Property	JSON/Scripting Name	Description	Data Type	Applicable Tag Type
Alarms	`alarms`	Tags have the ability to define any number of alarms. Each alarm is a condition that will be evaluated when the value of the tag changes. When the condition becomes true, the alarm is said to be active. When it becomes false, the alarm is said to be cleared. For detailed information, see **Tag Alarm Properties**.	JSON Array of JSON objects. For detailed information, see **Tag Alarm Properties** .	OPC, Query, Expression, Derived, Reference, Memory
Alarm Eval Enabled	`alarmEvalEnabled`	Determines if alarms will be evaluated on this tag.	Boolean	OPC, Query, Expression, Derived, Reference, Memory

History Properties

Property	JSON/Scripting Name	Description	Data Type	App Tag
History Enabled	`historyEnabled`	Whether the tag will report its history to the Tags Historian system.	Boolean	OPC Que Expr Deri Refe Mem
Storage Provider	`historyProvider`	Which Tag Historian data store the tag will target. A particular tag can only target one history store. For more information, refer to **History Providers** on the Tag History Gateway Settings page. **Note**: The Storage Provider dropdown displays the provider names as they are written at the time of tag configuration. If the Storage Provider name is updated later, this setting will need to be adjusted to match the new Storage Provider name.	String	OPC Que Expr Deri Refe Mem

Property	JSON/Scripting Name	Description	Data Type	App Tag
Deadband Style	`historicalDeadbandStyle`	There are three styles to choose from: Auto, Analog, or Discrete. When set to Auto, this setting will automatically pick from Analog or Discrete, based on the data type of the tag: • If the data type of the tag is set to a float or double, then Auto will use the Analog Style • If the data type of the tag is any other type, then the Discrete style will be used. More information on the Analog and Discrete types can be found on the **Configuring Tag History** page. A list of possible values along with their JSON name are below: • **Auto** `Auto` • **Analog** `Analog_Compressed` • **Discrete** `Discrete`	String	OPC Que Expr Deri Refe Mem
Deadband Mode	`historicalDeadbandMode`	Defines how the deadband value is used.	String	OPC Que

Property	JSON/Scripting Name	Description	Data Type	App Tag
		Possible values along with JSON names and descriptions are listed below: • **Absolute** `Absolute` : The deadband setting is considered to be an absolute value. • **Percent** `Percent`: The actual deadband is calculated as a percent of the tag's engineering unit span. • **Off** `Off`: The deadband setting is the equivalent to a value of 0.0, so that all values pass through if their timestamp has changed. See the **Numeric Properties** section above for the Percent calculation.		Expr Deri Refe Mem
Historical Deadband	`historicalDeadband`	A deadband that applies only to historical evaluation.	Numeric	OPC Que Expr Deri Refe Mem

Property	JSON/Scripting Name	Description	Data Type	App Tag
Sample Mode	`sampleMode`	Determines how often to check if a historical record should be collected. Possible values include: • OnChange • Periodic • TagGroup	String	OPC Que Expr Deri Refe Mem
Sample Rate	`historySampleRate`	When the **Sample Mode** property is set to "Periodic", this property (working in conjunction with the **Sample Rate Units** property) determines how often a record should be collected.	Numeric	OPC Que Expr Deri Refe Mem
Sample Rate Units	`historySampleRateUnits`	When the **Sample Mode** property is set to "Periodic", this property (working in conjunction with the **Sample Rate** property) determines the unit of time that will be use in record collection. For a list of units, see the **Units of Time** section.	String	OPC Que Expr Deri Refe Mem
Historical Tag Group	`historyTagGroup`	When the Sample Mode property is set to **Tag Group**, this property determines which Tag Group will be used to collect records. See **Tag**	String	OPC Que Expr Deri Refe Mem

Property	JSON/Scripting Name	Description	Data Type	App Tag
		Groups for more information.		
Min Time Between Samples	`historyTimeDeadband`	Minimum time between records. Prevents multiple consecutive tag changes from triggering consecutive record collections. Works in conjunctions with the **Min Time Units** property. The Value is calculated off of the value timestamp. Useful in restricting the number of records collected when the **Sample Mode** is set to **On Change**. However, the behavior will differ slightly when using **On Change**, as the latest timestamp **after** the **Min Time Between Samples** threshold is used instead of timestamps from before the minimum time.	Integer	OPC Que Expr Deri Refe Mem
Min Time Units	`historyTimeDeadbandUnits`	Units of time to use with the Min Time Between Samples property. For a list of units, see the Units of Time section.	String	OPC Que Expr Deri Refe Mem

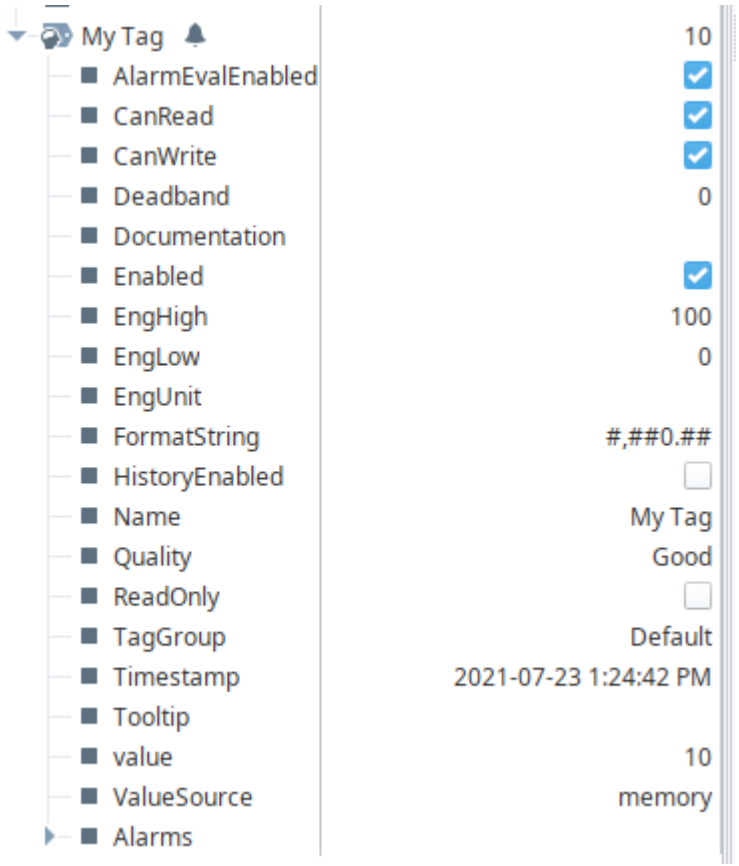
Property	JSON/Scripting Name	Description	Data Type	App Tag
Max Time Between Samples	`historyMaxAge`	Maximum time between samples. Works in conjunction with the **Max Time Units** property. If a sample has not been collected by the time range specified by these two properties, then a record will be collected on the next sample interval. This setting cannot be less than 1000ms. Setting this value to 0 will disable automatic record collection. Default is 0. **Note**: This setting will be ignored if the Sample Mode is set to Tag Group, and the targeted Tag Group is using non-default values for its Max Time Between Samples setting. The implication being that non-default values on the Tag Group settings take precedence over this setting.	Integer	OPC Que Expr Deri Refe Mem
Max Time Units	`historyMaxAgeUnits`	Maximum time in units. For a list of units, see the **Units of Time** section.	String	OPC Que Expr Deri Refe Mem

Units of Time

Unit of Time	JSON/Scripting Name
Milliseconds	`MS`
Seconds	`SEC`
Minutes	`MIN`
Hour	`HOUR`
Day	`DAY`
Week	`WEEK`
Month	`MONTH`
Year	`YEAR`

Runtime Properties

In addition to properties listed in the Tag Editor, some properties are exposed as runtime properties in the Tag Browser. These properties are valid targets for component bindings and tag reads/writes.



Most runtime properties are representations of properties that can be configured in the Tag Editor. However there are some properties only listed in the Tag Browser:

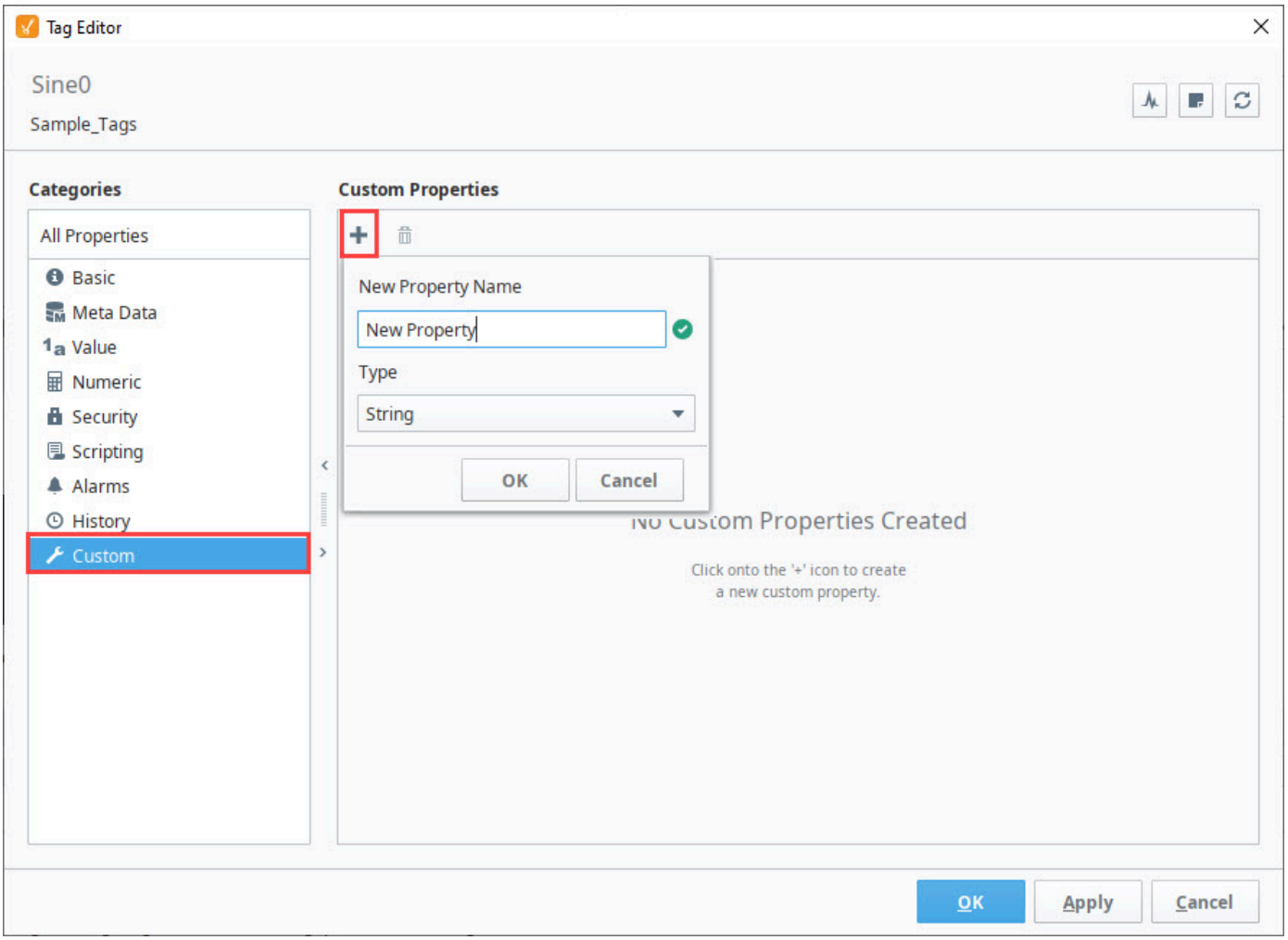
Property	Description
CanRead	A read-only property that represents whether or not this tag can be read from the current security context. This is determined by looking at the read permission settings on the tag and the Tag Provider's permission settings.
CanWrite	A read-only property that represents whether or not this tag can be written to from the current security context. This is determined by looking at the write permission settings on the tag, the Read Only property, and the Tag Provider's permission settings.

Custom Tag Properties

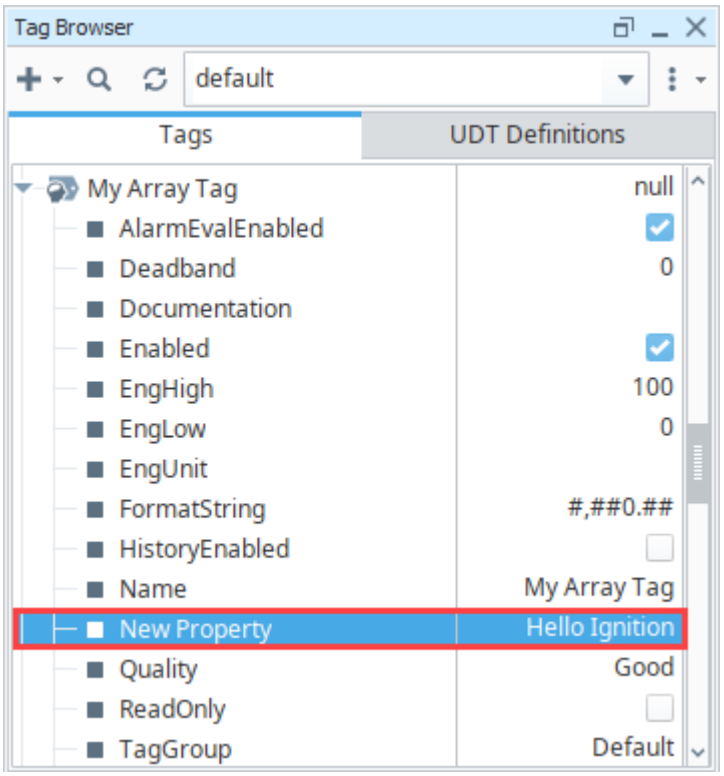
Custom tag properties allow application designers to configure their own properties on tags to store unique values on any tag. Once added, a custom property can be referenced like any other tag property via bindings, expressions, and scripts.

Both the Perspective and Vision visualization systems can bind to custom properties. In the following example, we already have Array tag. Now let's add a custom property.

1. Open an existing tag and select the Custom properties category.
2. Click on the **Add** ✚ icon in the Tag Editor. This will open the Custom Property dialog box.
3. Enter a **Name** for your property, select the **Data Type**, and click **OK**.



4. When you open your Tag Browser and expand the My Array Tag, you'll see your Custom Property.



Vision Client Tags

Client tags have the ability to be used as either Expression or SQL Query tags. There is an **Expression/SQL** page in the Tag Editor that allows you to select what type it is.

Property	Binding/Scripting Name	Description	Data Type	Applicable Tag Type
OPC Server	`OPCWriteBackServer`	The server against which to subscribe the data point. **Note**: this property is only present on legacy tag imports, from 7.9 and earlier	String	Query, Expression
OPC Item Path	`OPCWriteBackItemPath`	The path to subscribe to on the server. **Note**: this property is only present on legacy tag imports, from 7.9 and earlier	String	Query, Expression
Query	`Expression`	Text area to build your query or expression. This is the code used by the tag: either a SQL Query for Query tags, or an Expression for Expression tags.	String	Query, Expression
Query Type	`QueryType`	When the TagType property is set to 1, this property determines if the tag should be a Memory, Expression, or Query tag. A mapping of supported values is listed below:	Integer	Query, Expression, Memory

Property	Binding/Scripting Name	Description	Data Type	Applicable Tag Type
		• **Memory**: 0 • **Expression**: 1 • **Query**: 2 .		
Datasource	`SQLBindingDatasource`	The default data source of the Tag Provider.	String	Query

Tag Alarm Properties

Alarm Property Reference

The table in this section provides a description of alarm properties. The descriptions of these properties can also be viewed in the Tag Editor > Alarms popup by selecting the Show/Hide Description Area icon available above the property list.

Alarms in Scripting

When interacting with the tags system in via scripting, such as with the **system.tag.configure** function, alarms are represented as a JSON array of JSON objects, where each object contains the configurations for a single alarm. Thus, any scripting names here are assumed to exist under the alarms array.

Main Alarm Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Name	`name`	Identifier of the alarm. Combined with the location of the alarm, this will be the unique alarm ID. For dynamic values, used Label or Display Path.	String	OPC, Query, Expression, Derived, Reference, Memory
Enabled	`enabled`	This boolean determines whether or not the alarm will be evaluated. A disabled alarm's condition will not be evaluated, and thus will not generate any alarm events.	Boolean	OPC, Query, Expression, Derived, Reference, Memory
Priority	`priority`	The priority or severity of this alarm. Alarm priorities can be examined by many other systems in Ignition, including the visualization modules, pipelines, and even scripting. Possible values are along with their JSON names are listed below: • **Diagnostic** `Diagnostic` • **Low** `Low` • **Medium** `Medium` • **High** `High`	String	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		• **Critical** `Critical`		
Timestamp Source	`timestampSource`	Indicates where the timestamp for the alarm event should come from: the system time of when the event was generated (i.e., the Gateway's time), or the timestamp of the value that triggered the the event (i.e., the timestamp of the value from the OPC server). See the list below for listing of possible values, including their JSON name and numeral representation: • **System** `System` : 0 • **Value** `Value` : 1	String	OPC, Query, Expression, Derived, Reference, Memory
Label	`label`	An optional name that will be used for display purposes. Provides a dynamic alternative to the static **name** property. If left blank, the **name** will be used.	String	OPC, Query, Expression, Derived, Reference, Memory
Display Path	`displayPath`	Optional text that will be used for display and browsing purposes. If this is blank, this property will show the	String	OPC, Query, Expression, Derived,

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		path to the tag and the name of the alarm instead.		Reference, Memory
Ack Mode	`ackMode`	Dictates how acknowledgement works for the alarm. Possible values are listed below, along with their JSON name and numerical representation: • **Unused** `Unused` - 0: Alarm is always marked as acknowledged. • **Auto** `Auto` - 1: Alarm is acknowledged when cleared. • **Manual** `Manual` - 2: Alarm is acknowledged by the user. **Note:** Since one of the default alarm dropout conditions is an acknowledged alarm, setting the Ack Mode to **Unused** prevents the alarm from ever entering the alarm pipeline. However, you can set the acknowledged dropout condition to	String	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		false to receive a notification.		
Notes	`notes`	A place for any free-form documentation about the alarm that can be displayed to users.	String	OPC, Query, Expression, Derived, Reference, Memory
Ack Notes Required	`ackNotesReqd`	If this setting is true, the operators will be required to provide some explanation when the alarm is acknowledged.	Boolean	OPC, Query, Expression, Derived, Reference, Memory
Shelving Allowed	`shelvingAllowed`	If this setting is false, the shelving feature will be unavailable for this alarm.	Boolean	OPC, Query, Expression, Derived, Reference, Memory

Alarm Mode Settings Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Mode	`mode`	This setting controls what condition this alarm is evaluating. See the listing below for a description of possible modes: • **Equal** `Equality` - Active when the tag's value equals the alarm's setpoint. • **Not Equal** `Inequality` - Active when the tag's value does not equal the alarm's setpoint. • **Above Setpoint** `AboveValue` - Active when the tag's value is above the alarm's setpoint. • **Below Setpoint** `BelowValue` - Active when the tag's value is below the alarm's setpoint. • **Between Setpoints** `BetweenValues` - Active when the tag's value is	String	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		between the low and high setpoints. If any change is true, an event will be generated for each value change between the setpoints. • **Outside Setpoints** `OutsideValues` - Active when the tag's value falls outside the low and high setpoints. If any change is true, an event will be generated for each value change outside the setpoints. • **Out of Range** `OutOfEngRange` - The same as Outside Setpoints, but uses the tag's Engineering High and Engineering Low as the high and low setpoints. • **Bad Quality** `BadQuality` - Active if the tag value becomes a bad quality, for example, on communication loss.		

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		• **Any Change** `AnyChange` - An alarm event is generated every time the tag value changes. **Note**: This alarm will never be "active" because each active event is paired with a matching clear event, instantly. • **Bit State** `Bit` - This alarm mode is used to alarm when a specific bit out of an integer tag becomes high. You must specify which bit position to use, with zero being the least significant bit. The On Zero property is used to invert the logic and alarm when the bit is low. • **On Condition** `OnCondition` - This free-form alarm mode is used for when you want to specify the condition using an expression or another tag. To do		

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		this, bind the "Is Active" property to an appropriate expression or tag. • **When True** `WhenTrue` - Active when the tag value is True for tags with Boolean data types. If the tag's data type is integer, the alarm will be active when the tag value is a non-zero number. • **When False** `WhenFalse` - Active when the tag value is False for tags with Boolean data types. If the tag's data type is integer, the alarm will be active when the tag value is zero.		
Setpoint/Low Setpoint	`setpointA`	Used to determine if the alarm is active by comparing this value to the the tag value. Some modes under the **Mode** property allow for multiple setpoints (i.e., a low setpoint and a high setpoint). In these cases, this property is	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		considered to be the Low setpoint. **Note**: Setpoints must be numerical values, not string representations of numbers. If a string is passed to a setpoint then it will be converted to a numeric value of 0.		
Inclusive	`inclusiveA`	If true, the Setpoint value is used inclusively for the condition to alarm.	Boolean	OPC, Query, Expression, Derived, Reference, Memory
High Setpoint	`setpointB`	The high value used to initiate an alarm when the alarm mode calls for two setpoints. Available for modes: Between Setpoint, Outside Setpoints. **Note**: Setpoints must be numerical values, not string representations of numbers. If a string is passed to a setpoint then it will be converted to a numeric value of 0.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
High Inclusive	`inclusiveB`	If true, the High Setpoint value is used inclusively for the condition to alarm. Available for	Boolean	OPC, Query, Expression, Derived,

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		modes: Between Setpoint, Outside Setpoints.		Reference, Memory
Any Change	`anyChange`	If true, will alarm on each value change given the alarm mode conditions are met. **Note**: This alarm will never be "active" because each active event is paired with a matching clear event, instantly. Available for modes: Above Setpoint, Below Setpoint, Between Setpoints, and Outside Setpoints.	Boolean	OPC, Query, Expression, Derived, Reference, Memory
On Zero	`bitOnZero`	If true, will alarm when the specified bit is not high (when the bit is 0).	Boolean	OPC, Query, Expression, Derived, Reference, Memory
Bit Position	`bitPosition`	The position of the bit, starting at 0 that will be watched. Available for modes: Bit State.	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Is Active	`activeCondition`	When this property is active, the alarm will be active. Typically has a binding of some sort	Boolean	OPC, Query, Expression, Derived,

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		that will be used to determine when the alarm goes active. If the expression evaluates to True, the alarm is active. If the expression evaluates to False, the alarm is not active.		Reference, Memory

Deadbands and Time Delay Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Deadband	`deadband`	The value for the deadband, interpreted according to the Deadband mode. **Note**: All alarms are only evaluated after the tag's value changes, which means that the tag's own deadband will be considered first. When the deadband is positive, an active alarm condition needs to clear its setpoint(s) by the amount of the deadband for the alarm to clear. For example, suppose you had a Between Setpoints alarm with a low setpoint of 50 and a high setpoint of 70, and with a deadband of 2. The alarm will go active if the value is between 50 and 70, but will only clear if the value falls	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		below 48 or rises above 72.		
Deadband Mode	`deadbandMode`	Defines how the deadband value is used. Possible values are listed below along with their JSON name and a description. • Absolute `Absolute` : The deadband setting is considered to be an absolute value. • Percent `Percent` : The actual deadband is calculated as a percent of the tag's engineering unit span. • Off `Off` : The deadband setting is the equivalent to a value of 0.0, so that all values pass through if their timestamp has changed.	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Active Delay	`timeOnDelaySeconds`	The time, in seconds, before the alarm will be considered active after the alarm's condition becomes true. Also known as a "rising edge time deadband."	Numeric	OPC, Query, Expression, Derived, Reference, Memory
Clear Delay	`timeOffDelaySeconds`	The time, in seconds, before an active alarm will be considered clear after the alarm's condition becomes false. Also known as a "falling edge time deadband." **Note**: When troubleshooting alarms that appear to be missing notifications, you may run into a situation where the alarm condition becomes true again before the Clear Delay elapses. In this case, the clear is never committed. The original active alarm event is retained, and no new alarm event is	Numeric	OPC, Query, Expression, Derived, Reference, Memory

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
		generated. Because of this, no new notification is sent.		

Notification Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Active Pipeline	`activePipeline`	The name of an alarm notification pipeline to put this alarm into when it becomes active in order to send out active alarm messages. Many alarms may share a single pipeline.	String	OPC, Query, Expression, Derived, Reference, Memory
Clear Pipeline	`clearPipeline`	The name of an alarm notification pipeline to put this alarm into when it becomes cleared in order to send out cleared messages.	String	OPC, Query, Expression, Derived, Reference, Memory
Ack Pipeline	`ackPipeline`	The name of the alarm notification pipeline to put this alarm into when the alarm has been acknowledged.	String	OPC, Query, Expression, Derived, Reference, Memory

Phone Call Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Custom Message	`voip.customMessage`	The voice message you want to play when sending the Alarm. See **Configuring Messages** for more information on custom phone messages.	String	OPC, Query, Expression, Derived, Reference, Memory

Email Notification Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Custom Subject	`CustomEmailSubject`	A string that will be used as the subject line of an email notification message. If blank, the message settings defined on the notification block that sent the email out will be used instead.	String	OPC, Query, Expression, Derived, Reference, Memory
Custom Message	`CustomEmailMessage`	A string that will be used as the body of this alarm's email notification message. If blank, the message settings defined on the notification block that sent the email out will be used instead. Supports HTML tags such as `<html>, <body>, <h1>,` etc.	String	OPC, Query, Expression, Derived, Reference, Memory

SMS Notification Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
Custom Message	`CustomSmsMessage`	If specified, will be used for the SMS message. If blank, the message defined in the notification block will be used.	String	OPC, Query, Expression, Derived, Reference, Memory

Associated Data Properties

Property Name	JSON/Scripting Name	Description	Datatype	Applicable Tag Type
User Defined Data	N/A	Associated Data are custom alarm properties that can be added to any alarm. These properties will often be bound to other Tags that represent associated contextual data that may be related to the alarm. A snapshot of the values of these properties will be taken when the alarm becomes active. These values will be attached to the alarm event as it moves through the rest of the alarming system, meaning that the values will be available from the alarm status system, the alarm journal system, and in the alarm notification system.	String	OPC, Query, Expression, Derived, Reference, Memory

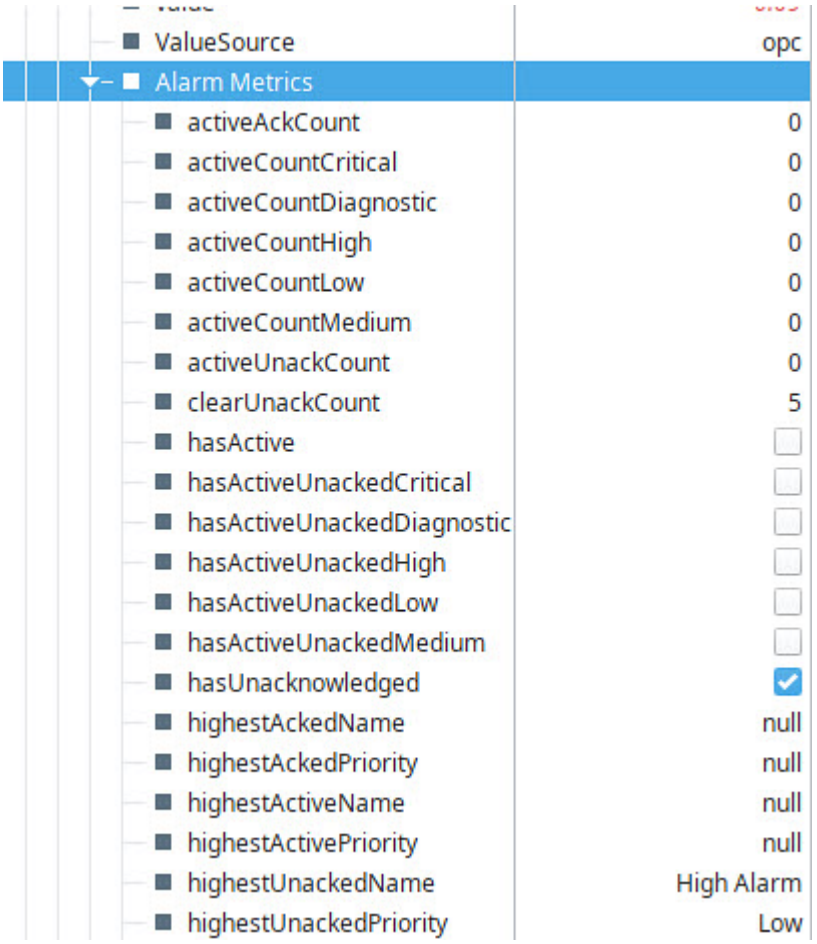
Runtime Alarm Metrics Properties

The Alarm Metrics folder is available within the Tag Browser under tag folders and UDTs, and will appear under individual tags when at least one alarm is configured on the tag. This folder contains properties that report the realtime operational status of those alarms.

New in 8.3.9 An Alarm Metrics folder at the tag provider level is now available in the Tag Browser.

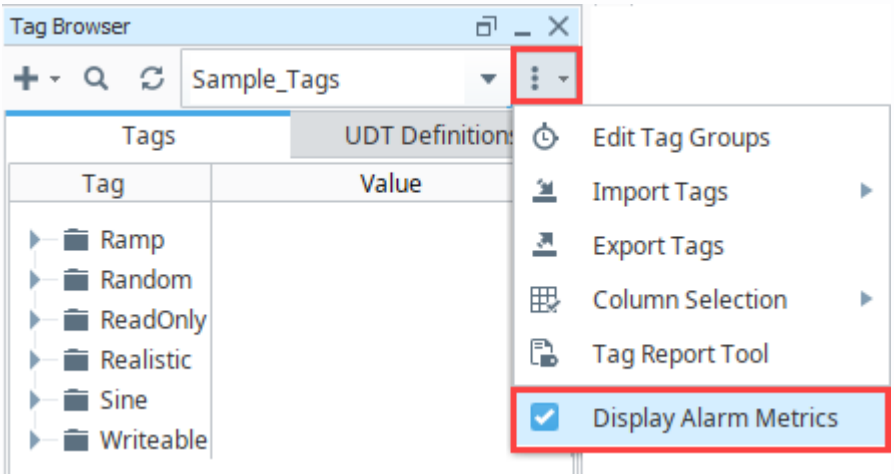
You are able to bind or subscribe to these alarm metrics, but direct reads are not supported to maintain efficiency and protect Gateway performance. However, creating a Reference tag for a specific alarm metric can then be used to read the value via scripting if necessary.

These properties include both standard alarm metric data, as well as total aggregated values. The aggregated alarm metric properties are aggregated by priority. Using these properties, you can list the specific aggregates, such as `ActiveCountCritical` in their own row, or use the wildcard notation to serve as the priority, such as `ActiveCount(*)`.



HIDE ALARM METRICS

The Tag Browser **More** icon includes an option to show/hide the Alarm Metrics. They are displayed by default, so the **Display Alarm Metrics** option will need to be unchecked if you don't want them to be visible.



Property Name	Description
ActiveAckCount	The number of alarms on the tag that are both active and acknowledged.
ActiveCount(Pri*)	The number of active alarms in each alarm priority. This aggregates all Active alarms, regardless of of Acked or Unacked. Includes the following count options: • ActiveCountCritical • ActiveCountDiagnostic • ActiveCountHigh • ActiveCountLow • ActiveCountMedium
ActiveUnackCount	The number of alarms on the tag that are both active and unacknowledged.
ClearUnackCount	The number of alarms on the tag that are both clear and unacknowledged.
HasActive	True, if the tag has at least one active alarm. False, if there are zero alarms.
HasActiveUnacked(Pri*)	A boolean value to indicate that at least one alarm in a priority is in alarm, but not acknowledged. The value will be true when there are any Active or Unacked alarms and false when there are no Active and Unacked alarms. Includes the following count options: • HasActiveUnackedCritical • HasActiveUnackedDiagnostic • HasActiveUnackedHigh • HasActiveUnackedLow • HasActiveUnackedMedium
HasUnacknowledged	True, if the tag has at least one unacknowledged alarm. False, if there are zero unacknowledged alarms.
HighestAckedName	The Name of the highest acknowledged alarm, ranked by Priority.

Property Name	Description
HighestAckedPriority	The highest Priority of all acknowledged alarms on the tag.
HighestActiveName	The Name of the highest active alarm, ranked by Priority.
HighestActivePriority	The highest Priority of all active alarms on the tag.
HighestUnackedName	The Name of the highest unacknowledged alarm, ranked by Priority.
HighestUnackedPriority	The highest Priority of all unacknowledged alarms on the tag.
LastActiveTime	A timestamp representing the last time an alarm went active on the tag.
ShelvedCount	The number of currently shelved alarms on the tag.
UnackCount(Pri*)	The number of unacknowledged alarms in each priority. Includes the following count options: • UnackCountCritical • UnackCountDiagnostic • UnackCountHigh • UnackCountLow • UnackCountMedium

Syntax Examples

The aggregated alarm properties can also be accessed using the dropdown in the Binding Editor for the Tag Path field, or entered directly using the syntax: `<TagPath>/Alarm Metrics.<Alarm Metric property>`.

For instance, if your Tag Browser has the tag and folder structure shown below, you can use the folder and tag data to collect alarm metrics anywhere from the Pumps level to the Outpress level. If looking for the Outpress level, the tag path would be your `[Tag Provider]` followed by `Pumps/Distribution Pumps/Plant A/Pump 1/Outpress` and `Pumps/Distribution Pumps/Plant A/Pump 2/Outpress` for Pump 1 and Pump 2.

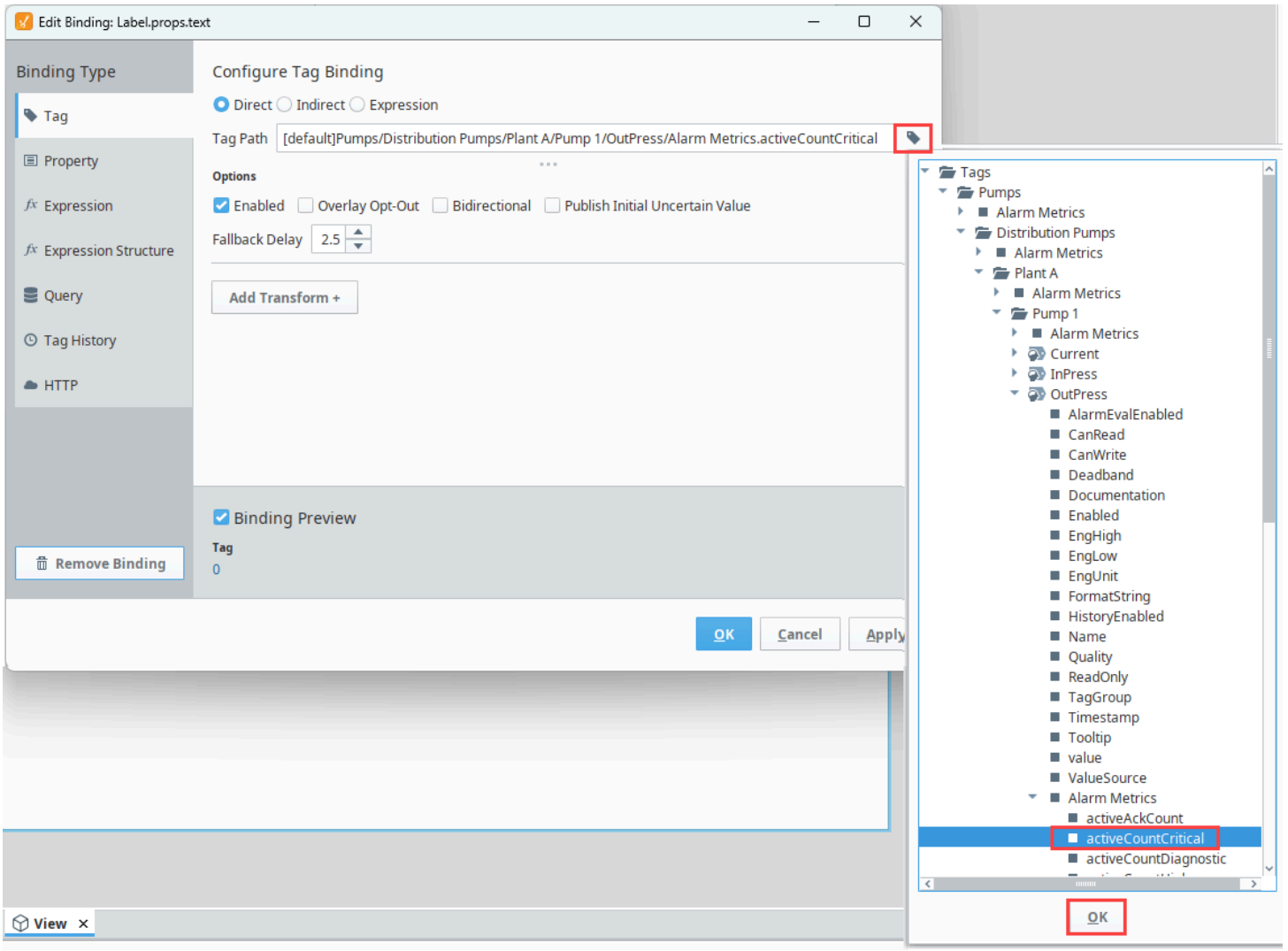
```
Pumps
 Distribution Pumps
     Plant A
         Pump 1
             Current
             InPress
             OutPress
             RPM
             Temperature
             Vibration
         Pump 2
             Current
             InPress
             OutPress
             RPM
             Temperature
             Vibration
```

The following examples list the tag paths to use in the Binding Editor to get the count of all active critical alarms at each level.

Tag Path Syntax Examples

```
## Pump 1 Outlet Press
Pumps/Distribution Pumps/Plant A/Pump 1/Outpress/Alarm Metrics.activeCountCritical

## Pump 1 All Alarms
Pumps/Distribution Pumps/Plant A/Pump 1/Alarm Metrics.activeCountCritical

## Plant A, All Pump Alarms
Pumps/Distribution Pumps/Plant A/Alarm Metrics.activeCountCritical

## All Distribution Pump Alarms
Pumps/Distribution Pumps/Alarm Metrics.activeCountCritical

## All Pump Alarms Tag Path
Pumps/Alarm Metrics.activeCountCritical
```

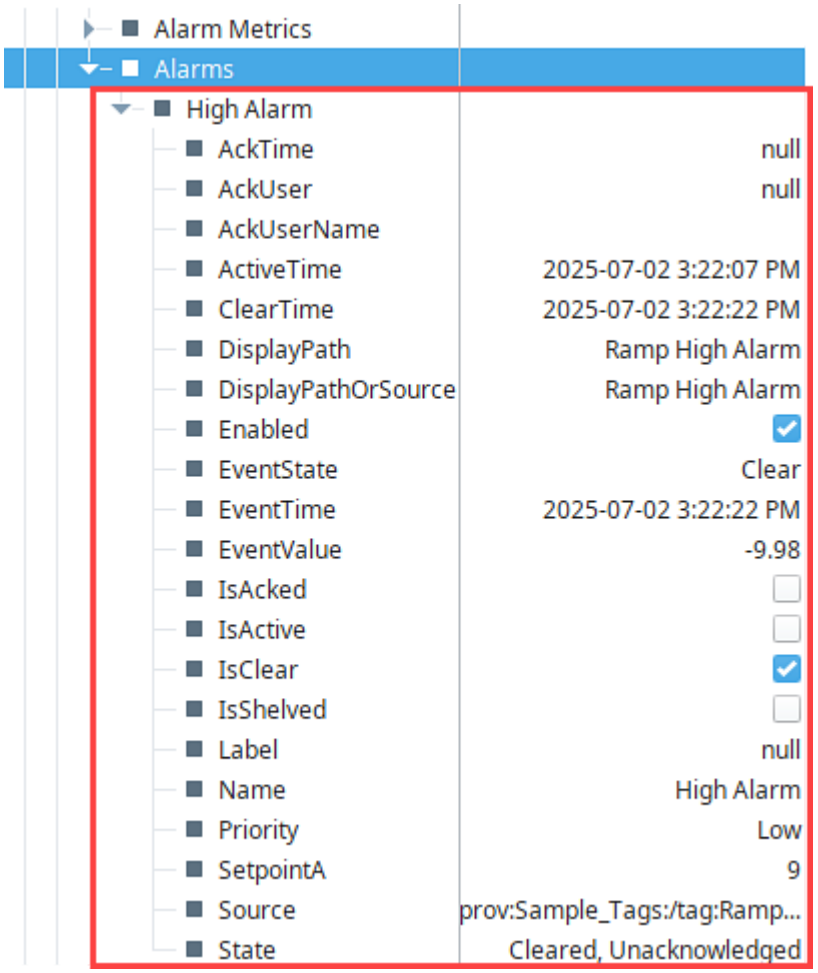
To use the Binding Editor to populate these paths instead, simply select the Browse Tags icon and navigate to the folder, UDT, or tag you want to select the Alarm Metric for. Once you've made your selection, click OK and it will fill the Tag Path field.



Runtime Alarm Metrics for Individual Alarms

In addition to the metrics above, each alarm configured on a tag has further properties that are available. These properties are located in the **Alarms** folder. For each alarm on a tag, additional expandable items will be visible in the property editor. The name on these items will match the name of their associated alarm.

In the image below, the parent tag has one alarm configured, named **High Alarm**. When the Alarms folder is opened, the High Alarm folder can be expanded to show the individual alarm properties. If there are multiple alarms on a tag, the Alarms folder will show multiple folders, sorted by the name of each alarm.



The additional properties revealed are listed in the table below.

Property Name	Description
AckTime	If the most recent alarm event for this alarm has been acknowledged, then this property will show a timestamp representing when acknowledgement occurred. If the most recent alarm event has not yet been acknowledged, the value of this property will be null.

AckUser	If the most recent alarm event for this alarm has been acknowledged, then this property will show the username that acknowledged the alarm event. Otherwise the value of this property we be null. If the alarm event was acknowledged by the system, such as when the Alarm Mode is set to Auto, then the property will show a null value.
	
	If the user that acknowledged the alarm event was authenticated via a User Source, then the value will follow the pattern below, where X is the name of the User Source, and Y is the name of the user.
	
	`usr-prov:X:/usr:Y`
	
	If the user that acknowledged the alarm event was authenticate via an Identity Provider, then the value will follow the pattern below, where Y is the name of the user.
	
	`usr:Y`

AckUserName	If the most recent alarm event for this alarm has been acknowledged, then this property will show the username that acknowledged the alarm event. Otherwise the value of this property we be an empty string.If the alarm event was acknowledged by the system, such as when the Alarm Mode is set to Auto, then the property will show an empty string. This property is effectively a simplified version of AckUser, containing only the username.
ActiveTime	Shows a timestamp representing when the most recent alarm event became active.
ClearTime	Shows a timestamp representing when the most recent alarm event transitioned from an active state to a clear state.
DisplayPath	Represents the value of the Display Path property on the alarm.

Property Name	Description
DisplayPathOrSource	Shows the display path if one has been configured. Otherwise shows the source path.
Enabled	Represent whether or not this alarm is enabled.
EventState	Represents the most recent state transition. A listing of states and their numerical values are listed below: • **Active** : 0 • **Clean** : 1 • **Acknowledge** : 2
EventTime	A timestamp representing when the event last changed state.
EventValue	Represents the last value that made the alarm event transition to either active or cleared.
IsAcked	A boolean representing whether or not the most recent alarm event has been acknowledged.
IsActive	A boolean representing whether or not the most recent alarm event is active.
IsClear	A boolean representing whether or not the most recent alarm event is cleared.
IsShelved	A boolean representing whether or not the alarm has been shelved.
Label	A string representing the Label property on the alarm. Shows a null value if a Label wasn't defined.
Name	The name of the alarm.
Priority	The priority of the alarm. A listing of priorities and numerical values are listed below: • **Diagnostic** : 0 • **Low** : 1

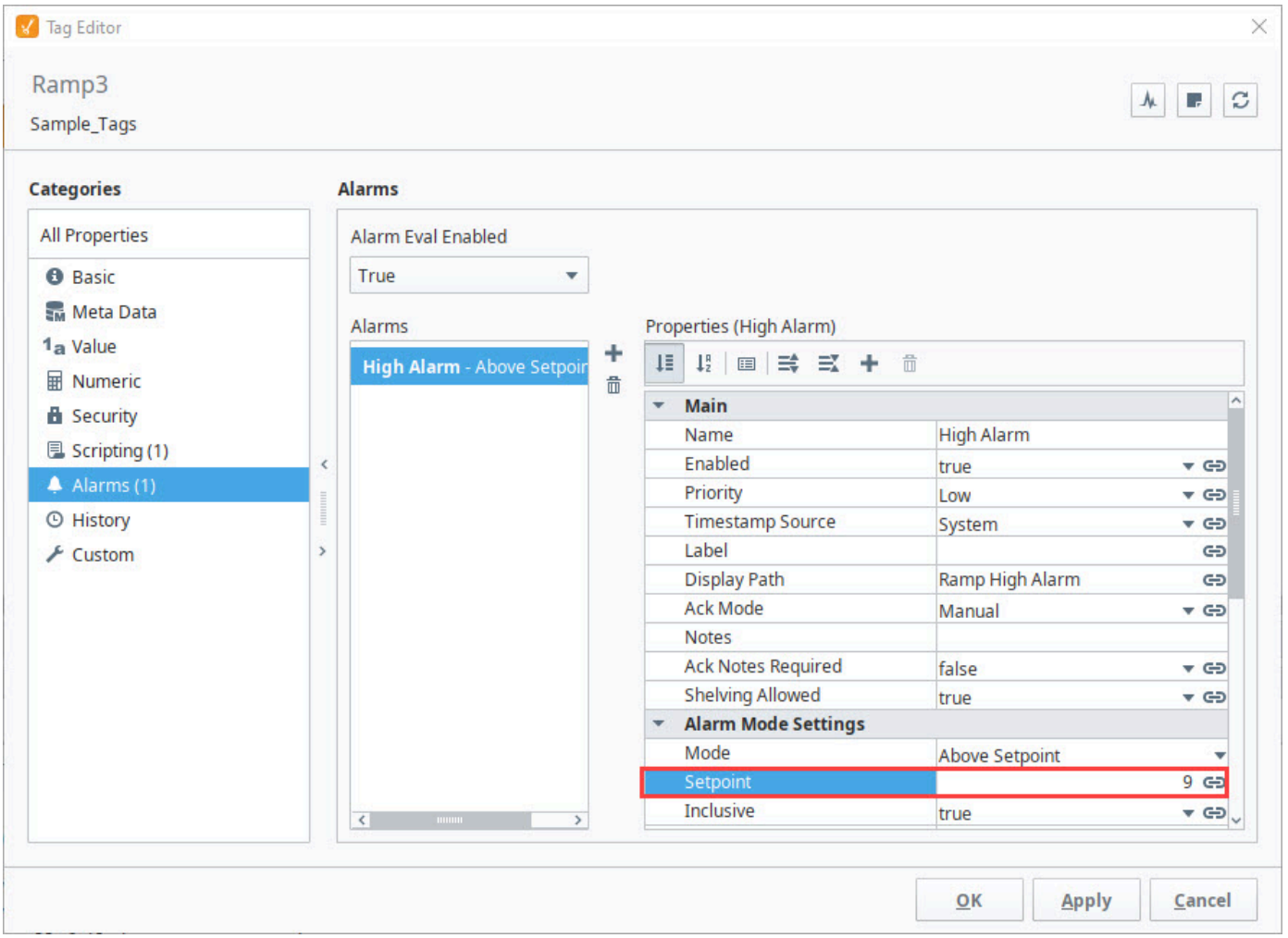
Property Name	Description
	• **Medium** : 2 • **High** : 3 • **Critical** : 4
SetpointA	When the Mode property on the alarm has a single setpoint, this property shows the setpoint value. When the Mode property has multiple setpoints, this represents the Low Setpoint property.
SetpointB	When the Mode property on the alarm has a single setpoint, this property is hidden. When the Mode property has multiple setpoints, this represents the High Setpoint property.
Source	The source path of the alarm.
State	The state of the alarm. A listing of states and their numerical values are listed below: • **Clear and Unacked** : 0 • **Clear and Acked** : 1 • **Active and Unacked** : 2 • **Active and Acked** : 3

Binding

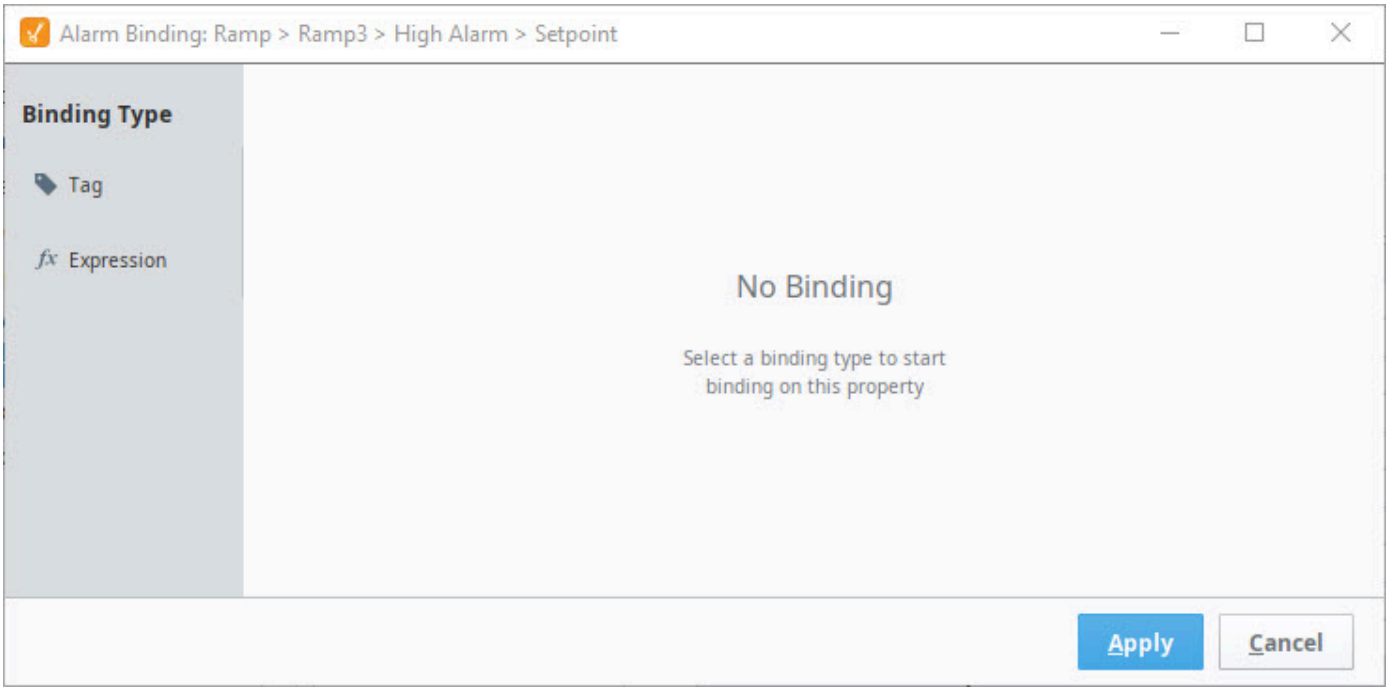
Many alarm properties are bindable, which means they can be bound to other tags in the system, or expressions. For example, you might bind the enabled property to another tag which represents whether or not your process is running, thereby, disabling the alarm when production is stopped. Another example is you might bind the setpoint of an alarm to a tag that operators can manipulate, thereby, letting the setpoint be changed at runtime. For more information, see **Configuring Alarms**.

To bind an alarm property of a tag:

1. Click on the **binding** icon. An Alarm Binding popup window will appear.

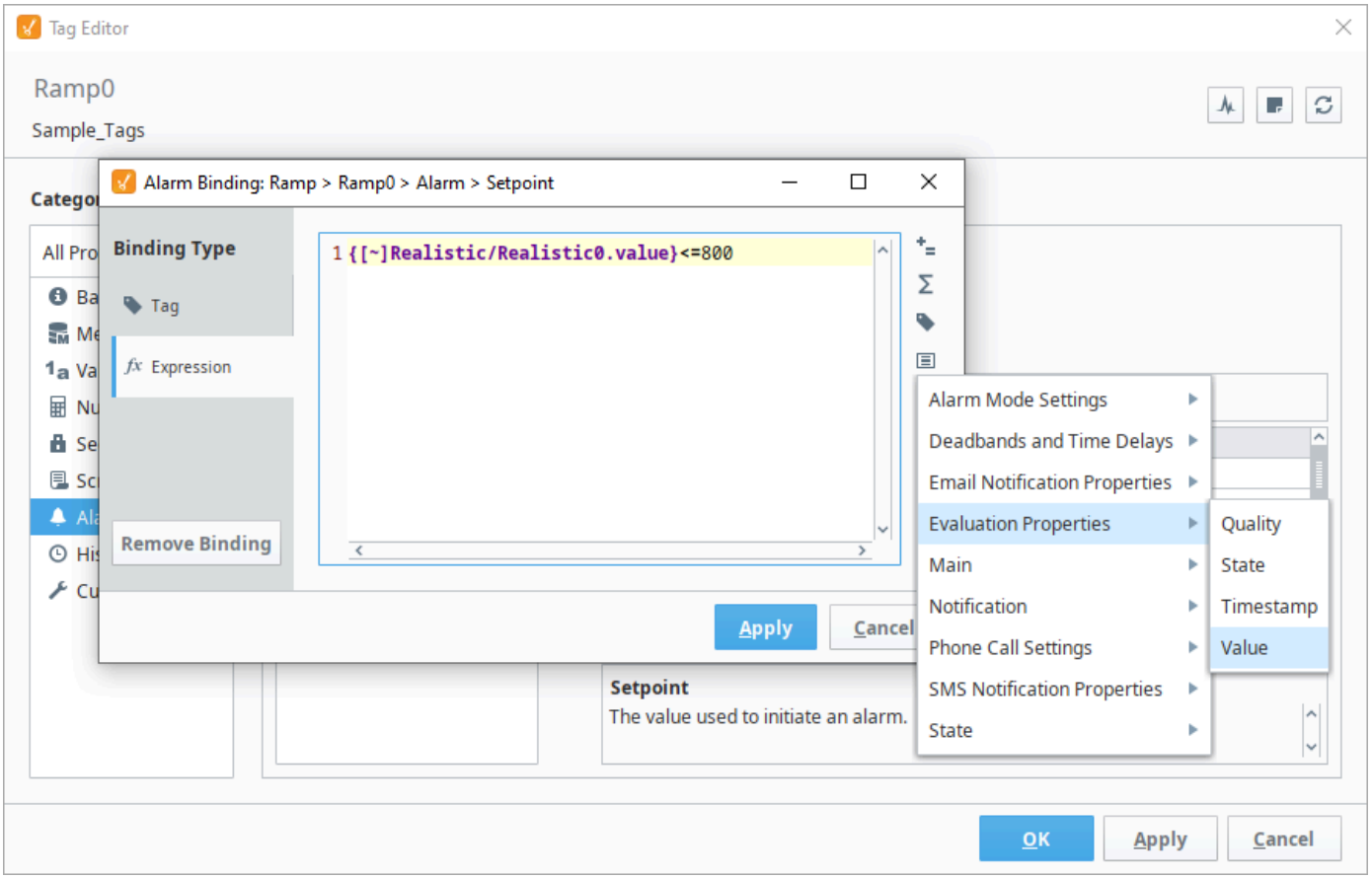


2. From this window, you can select a Binding Type to configure a Tag or an Expression binding. For this example, let's select Expression. Binding to an Expression can reference many useful values such as the tag's value and other settings of the alarm.



3. Now, enter your expression, using the Operators , Functions , Tags , and Alarm Properties selections as needed.

> **NOTE**
>
> The alarm properties that can be selected using the Alarm Properties icon are the properties included in the **Tag Editor > Alarms** window or under the tag in the Tag Browser. See the **Reference Table** section and the **Runtime Alarm Metric Properties** section for descriptions of these properties. The State properties will only be available to view after alarming is enabled on the tag.

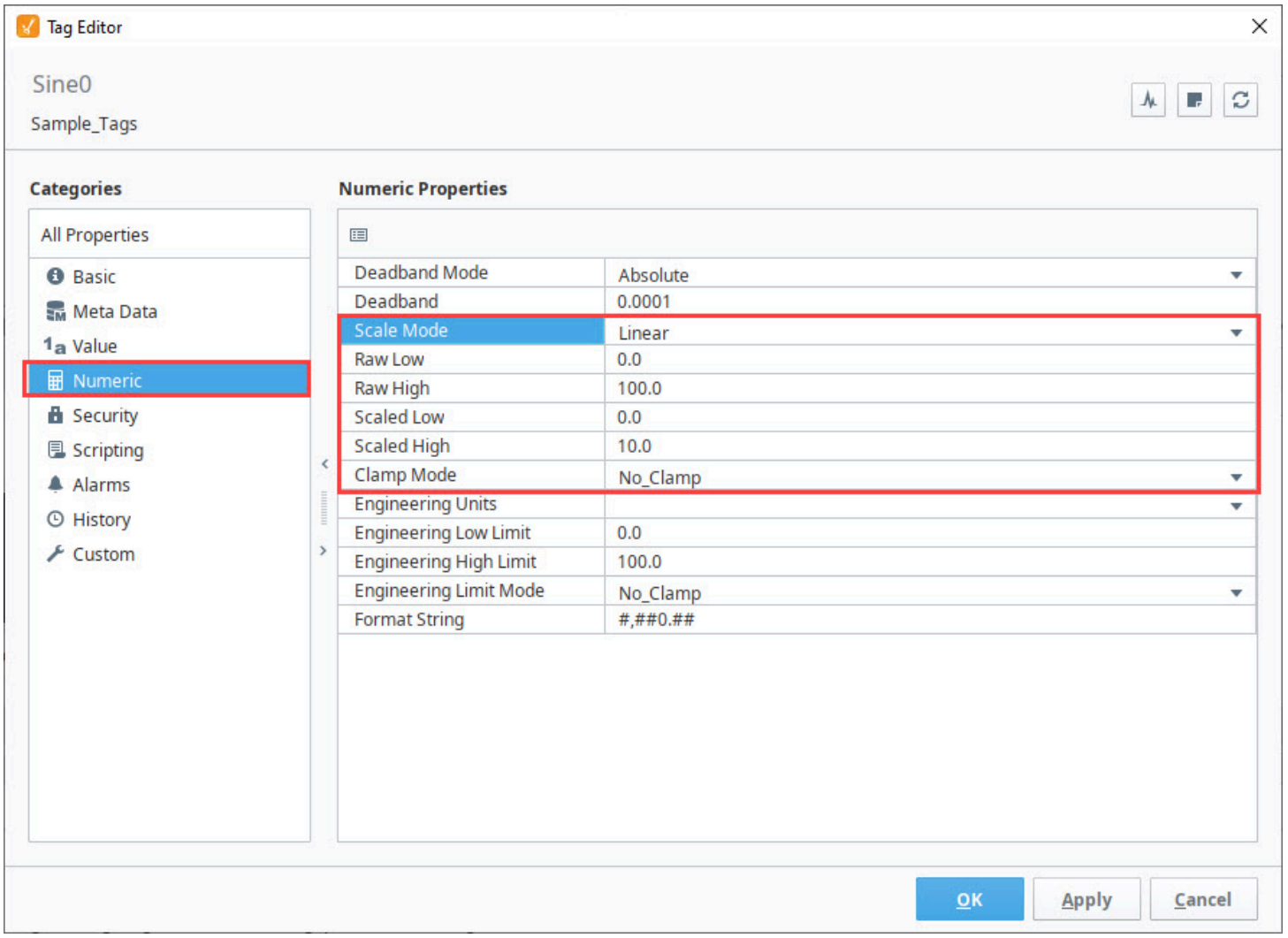


4. When you configured the binding to your liking, click **Apply**.

Tag Scaling Properties

Configuring a Tag's scaling will condition the data for use within the Ignition Designer. Scaling will take the raw PLC value driving a Tag, do some math, and use the resulting value as the value of that Tag. Scaling works both ways. When you write to that Tag, Ignition will scale it in the opposite direction before writing to the PLC.

For example, if the capacity of a tank is 5250 gallons, but the tank's fill level is better represented in the Designer as percentage of 0 through 100, configuring the Tag's scaling property, will result in the Tag displaying 0 through 100, while the actual tank fill moving is between 0 and 5250 gallons. For this example, you can double-click on the Tag to open the **Tag Editor**, and expand **Numeric Properties** to configure the scaling of the Tag. When scaling between a Raw Low and and High, and Scaled Low and High, select the **Linear** Scale Mode. So what Ignition is actually doing, is setting up the calculations behind the scenes to scale the value appropriately.



IF YOU ARE USING SCALING...

The numbers and units don't have to match-up. Scaling is straight from one number to another number. It doesn't matter what the units are, and it doesn't matter what the conversion is. What is important, is that the data type of the Tag must match the data type of the scale value (i.e., dividing an integer in the PLC by 10 probably means your Ignition Tag should be a float).

Scaling is not available on Memory Tags: Memory Tags are not driven by an external source such as a PLC or SQL query, so scaling will never be applied. In these scenarios, it is recommended to scale the mechanism that is writing to the Memory Tag instead. Numerical properties of Tags can be scaled allowing automatic bi-directional conversion outside of the PLC. Scaling types include **Linear** scaling, **Square Root** scaling, and **Exponential Filter** scaling. The numerical properties are available to **OPC**, **Expression**, **Database**, and **Client** Tags whose data types are numeric. For a complete list of all of the Tag Scaling properties, see **Tag Properties**.

Linear Scaling

The value will be scaled linearly between the low and high values, and clamped as appropriate. The linear equation is:

```
ScaledValue = ΔS * (Value-RL)/ΔR + SL
```

Where:

- ΔS = (ScaledHigh-ScaledLow)
- ΔR = (RawHigh - RawLow)
- RL = RawLow
- SL = ScaledLow

Square Root Scaling

The equation for square root scaling is:

```
ScaledValue = ΔS * √((Value-RL)/ΔR) + SL
```

Where:

- ΔS = (ScaledHigh-ScaledLow)
- ΔR = (RawHigh - RawLow)
- RL = RawLow
- SL = ScaledLow

Exponential Filter Scaling

This mode implements a simple linear recursive filter to smooth values. The scale factor corresponds to the weight of the smoothing effect, and is a value between 0.0 and 1.0. The smaller the factor, the greater the degree of smoothing.

The equation for the filter is:

```
y(t) = (1-f)*y(t-1)+f*x(t)
```

Where:

- y(t) = the output at time t
- y(t-1) = the previous output
- x(t) = the input value (current value)

- f = the scale factor, with 0.0<=f<=1.0

> **NOTE**
>
> Only **good** quality values are considered for the filter. **Bad** quality values are ignored.

Bit Inversion Scaling

This simple scaling mode will generate the complement of a binary value. If the current value is coming in as 0001_0101 (21), this will return a 1110_1010 (-22) instead. A popular use for this scale mode is that it can be used to invert modbus values if your device stores them in reverse bit order. Note that Bit Inversion Scaling uses a little-endian format.

Scaling Examples

Simple Divide by Ten

This is common when storing a single decimal point of precision as an Integer in the PLC. This is to save space by not using a Float type.

- **Raw Low**: 0.0
- **Raw High**: 100.0
- **Scaled Low**: 0.0
- **Scaled High**: 10.0

4-20 Milliamp Signal to Percent

This is common when using a simple pressure sensor. The sensor is calibrated to send 4 milliamps (minimum value) when the tank is empty, and 20 milliamps (maximum value) when the tank is full.

- **Raw Low**: 4.0
- **Raw High**: 20.0
- **Scaled Low**: 0.0
- **Scaled High**: 100.0

4-20 Milliamp Signal to Gallons

This is common when using a simple pressure sensor. The sensor is calibrated to send 4 milliamps (minimum value) when the tank is empty, and 20 milliamps (maximum value) when the tank is full (5000 gallons).

> **NOTE**
>
> There is no direct conversion between amps and gallons. In scaling it doesn't matter.

- **Raw Low**: 4.0
- **Raw High**: 20.0
- **Scaled Low**: 0.0
- **Scaled High**: 5000.

Tag Security Properties

Tag security is often the best way to configure security for data access. By defining security on a Tag, you affect the Tag across wherever it is used, as opposed to configuring component security on each component that displays or controls that Tag.

There are three properties on Tags that can restrict access.

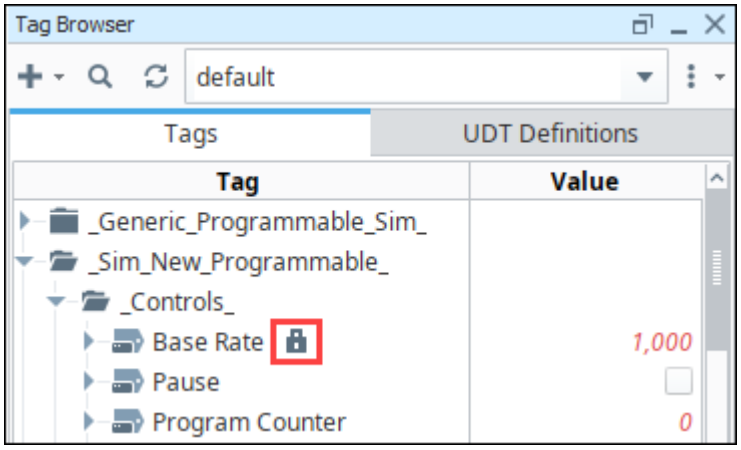
- Read Permissions: Defines the security levels required in order to read values from a Tag
- Read Only: Defines whether a Tag is read-only or writable
- Write Permissions: Defines the security levels required in order to write values to a Tag

Users with specific roles and zones can be given read/write access to a Tag, while other users with other roles are excluded from modifying the Tag.

If a user opens a Perspective view or a Vision client window that has components that are bound to a Tag they do not have permissions for, the user will see an overlay on top of the component. For more information, see **Quality Codes and Overlays**. The following example shows a tank displayed in a session, but the user does not have read permission for the Tag it is bound to.

Read Only Security

When a Tag is set to read only, a **Lock** icon is displayed next to the Tag in the Tag Browser.



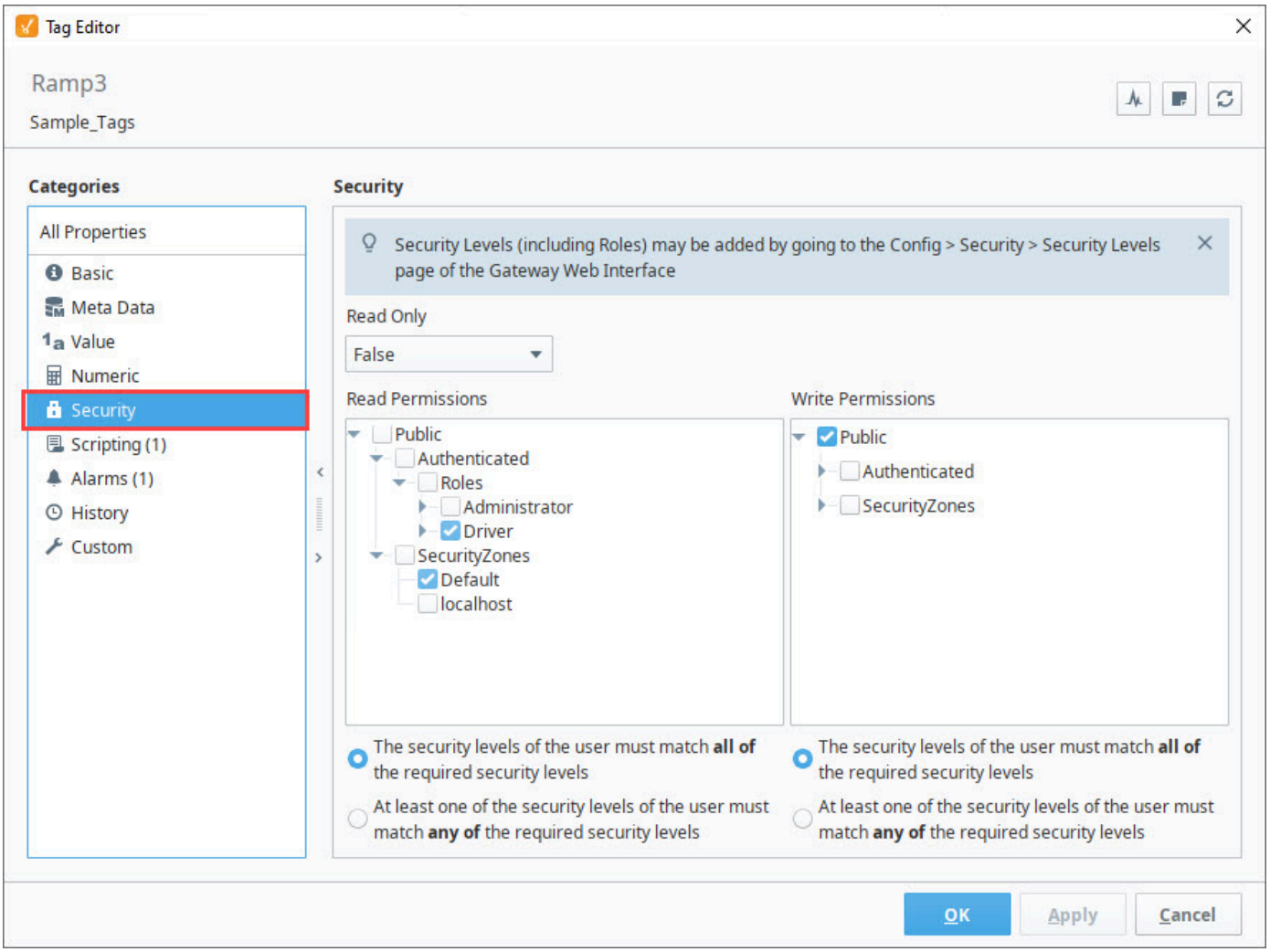
Read and Write Permissions

Instead of making a tag Read Only for all users, you can conditionally provide read and write access based on **Security Levels**. Doing so involves a adjusting the security settings on the tag in question. The checkbox tree you are presented with will show you all of the security levels configured in the Gateway Config > Security > Security Levels page.

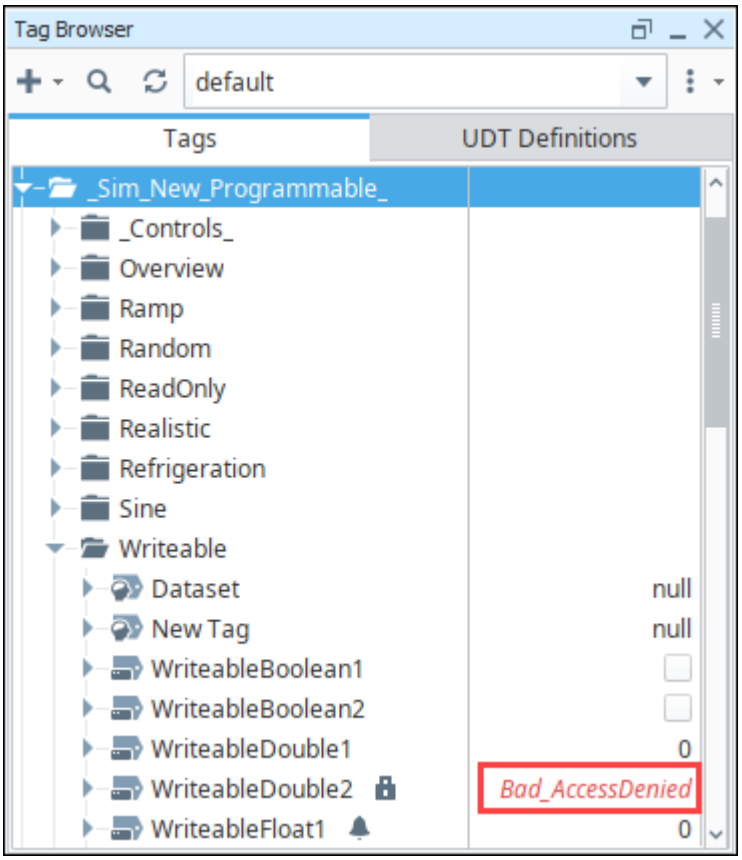
Read Permissions

Read permissions define the security levels required in order to read values from a Tag. By default, Tags have Read Permissions set to "Public". You can change the Read security using the Tag Browser in the Designer.

1. In the Tag Browser, right-click on the **Tag**, and select the **Edit** icon.
2. Select the Security section.
3. On the screen, choose the security levels you want to have Read permissions for this Tag. In this example, only users with role of "Driver" will be able to see the Tag value.



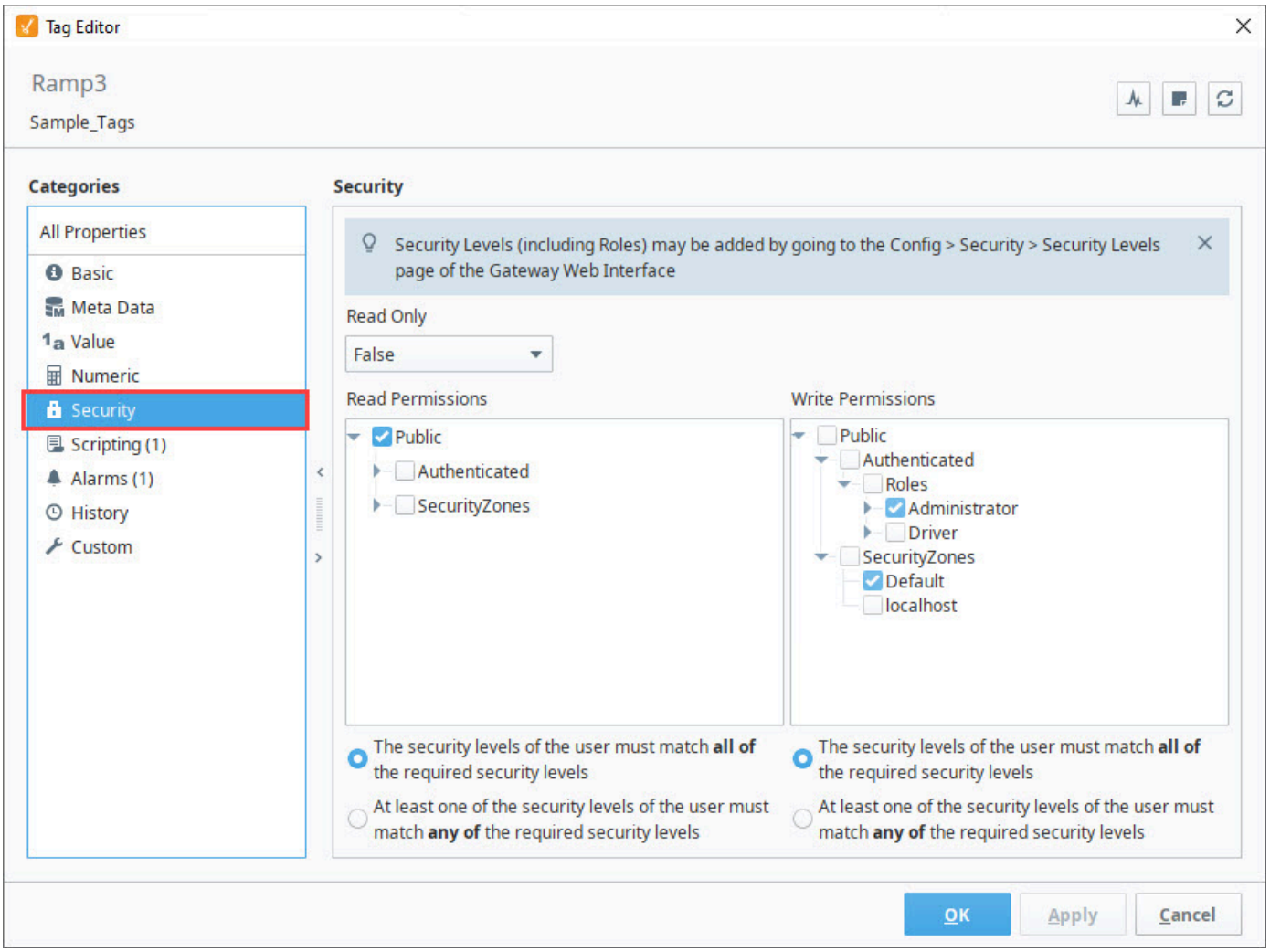
4. Click **Commit** to accept the settings.
5. Click **OK** to save the changes to the Tag.
6. If you are logged in as a user other than Driver, you will now see the "Bad_AccessDenied" in the Tag Browser instead of the Tag's value.



Write Permissions

Write permissions define the security levels required in order to write values a Tag. By default, Tags have Write Permissions set to "Public". You can change the Write security using the Tag Browser in the Designer.

1. In the Tag Browser, right-click on the **Tag**, and select the **Edit** icon.
2. Select the Security section.
3. On the screen, choose the security levels you want to have write permissions for this Tag. In this example, only users with role of Administrator will be able to write to the Tag value.



If any security levels under the Read Permissions or Write Permissions sections are deleted from the Gateway, they will appear grayed-out with a red warning underline. A warning indicator ⚠ icon will also appear in the upper right-hand corner with the number of selected security levels that no longer exist. If you follow the security levels tree up to the parent levels, you'll notice affected levels includes a dotted underline.

The deleted security level and all warning indications are removed when the deleted security level is unchecked and new settings are saved.

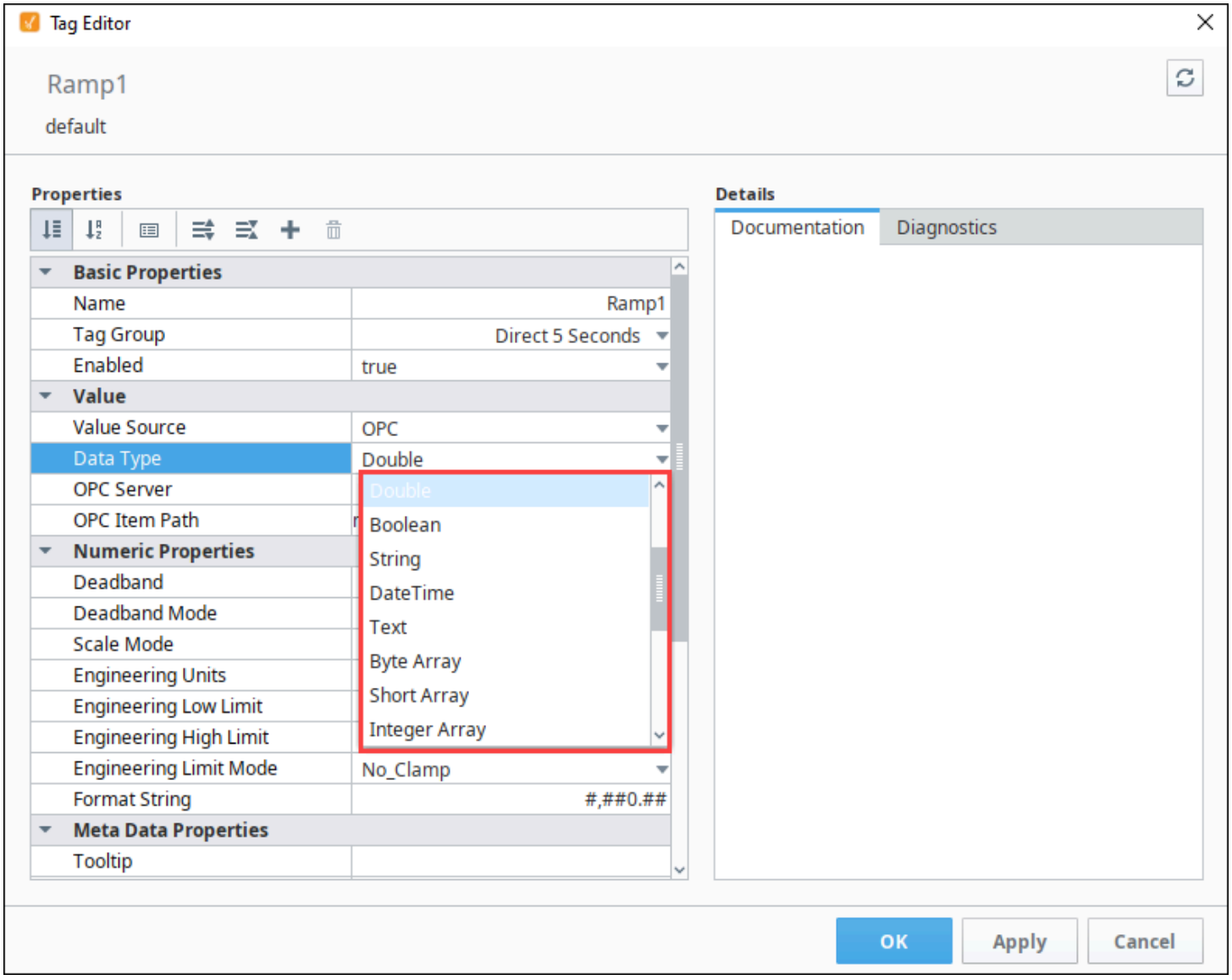
Using Security Zones

In addition to setting up security on individual Tags, you can set up security policies specific to Security Zones. This is useful in cases where you want to make all tags in a provider Read Only from network locations. See the **Security Zones** page for more details about Tag Access options.

Tag Data Types

This page describes the data types that can be applied to standard tags.

The data type of a tag is determined by the Data Type property in the Tag Editor The tag system will attempt to coerce incoming raw values (for example, from OPC or a SQL query) to the configured type.



The following table lists all data types available for tags.

Data Type	String Value	Integer Value
Byte	`Int1`	0
Short	`Int2`	1
Integer	`Int4`	2
Long	`Int8`	3
Float	`Float4`	4
Double	`Float8`	5
Boolean	`Boolean`	6
String	`String`	7
DateTime	`DateTime`	8
Text (Deprecated)	`Text`	10
Byte Array	`Int1Array`	17
Short Array	`Int2Array`	18
Integer Array	`Int4Array`	11
Long Array	`Int8Array`	12
Float Array	`Float4Array`	19
Double Array	`Float8Array`	13
Boolean Array	`BooleanArray`	14
String Array	`StringArray`	15
DateTime Array	`DateTimeArray`	16
Binary Data	`ByteArray`	20

Data Type	String Value	Integer Value
Dataset	`DataSet`	9
Document	`Document`	29

> **NOTE**
>
> The Siemens driver supports the **DateTime** data type for the following devices:
>
> - S7-300
> - S7-400
> - S7-1500

Array and Dataset Data Types

The Array and Dataset data types allow multiple data points to be stored in a single tag. To configure a tag as an array or dataset, change the Data Type property in the Tag Editor.

> **NOTE**
>
> Most OPC servers do not support Array or Dataset data types. These types work best when used on Query tags or Memory tags. When working with arrays through OPC UA device drivers, it is recommended to create separate Ignition tags for each array element instead of using a single tag.

Array Types

For OPC servers and drivers that support Array type tags, each element in the array can be represented using the Array data types in Ignition. Since all elements share the same core data type, configurations applied to the tag, such as tag history, alarming, or scaling, are inherited by each element.

Array tag types accept and coerce JSON array values, allowing methods such as `jsonGet`.

Write-Back

OPC array tags support writing values back to the device. How this occurs depends on the OPC server implementation. Some OPC servers support writing to individual array elements, allowing

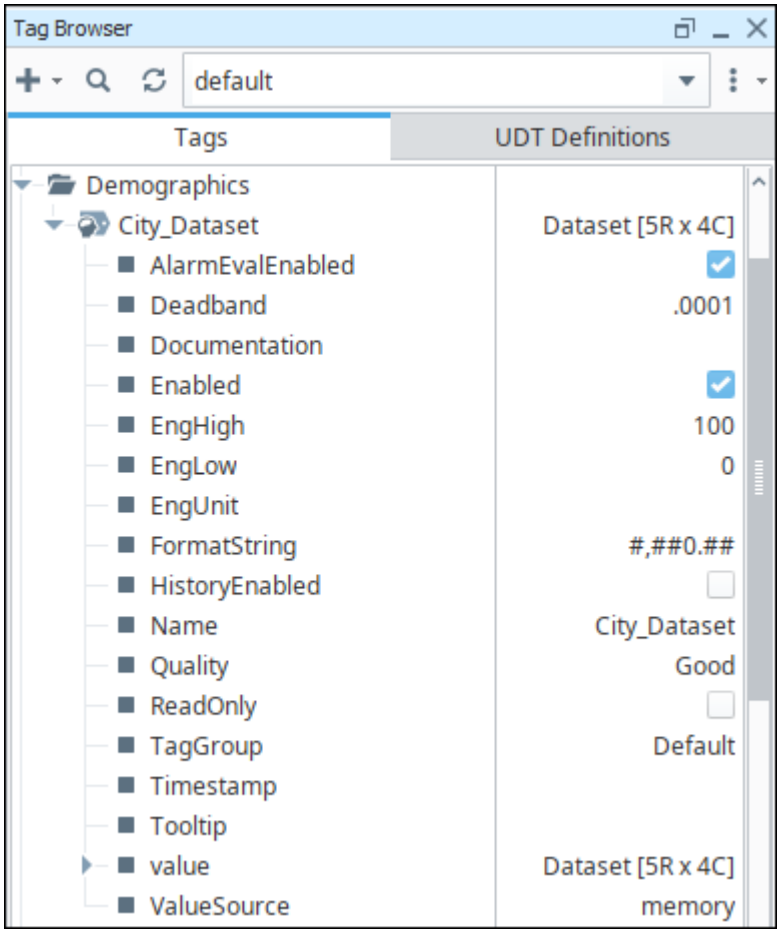
writes to occur just like any other tag write. However, other OPC servers do not support individual element writes. In these cases, the entire array must be written back to the array tag even if only a single element changes.

Dataset Types

Dataset types allow multiple rows and columns of data to be stored in a single tag. Each column appears as a separate folder in the tag. For example, the **name** folder shown in the image below.

Dataset types can be driven by a query, making it possible to retrieve multiple columns in a single query. This is more efficient than using multiple Query tags that each execute separate queries.

While datasets are convenient, note that the Tag History system and Alarm system do not support Dataset tag types.



Valid data types that can be stored in Dataset type tags include:

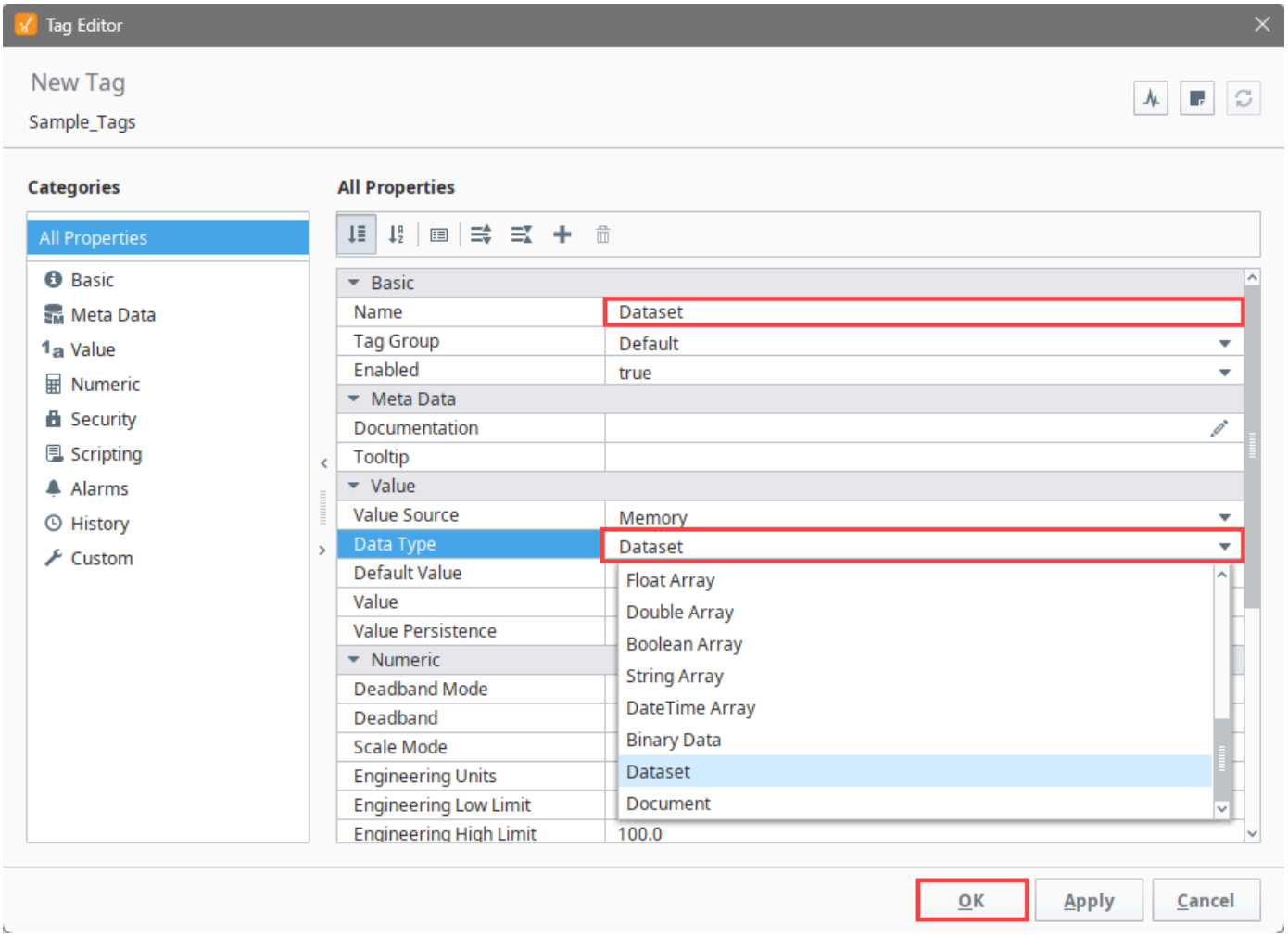
- Float
- Short
- Long
- Date
- Integer

- Boolean
- String
- Color
- Double
- Timestamp
- Byte Arrays

Dataset Tag Example

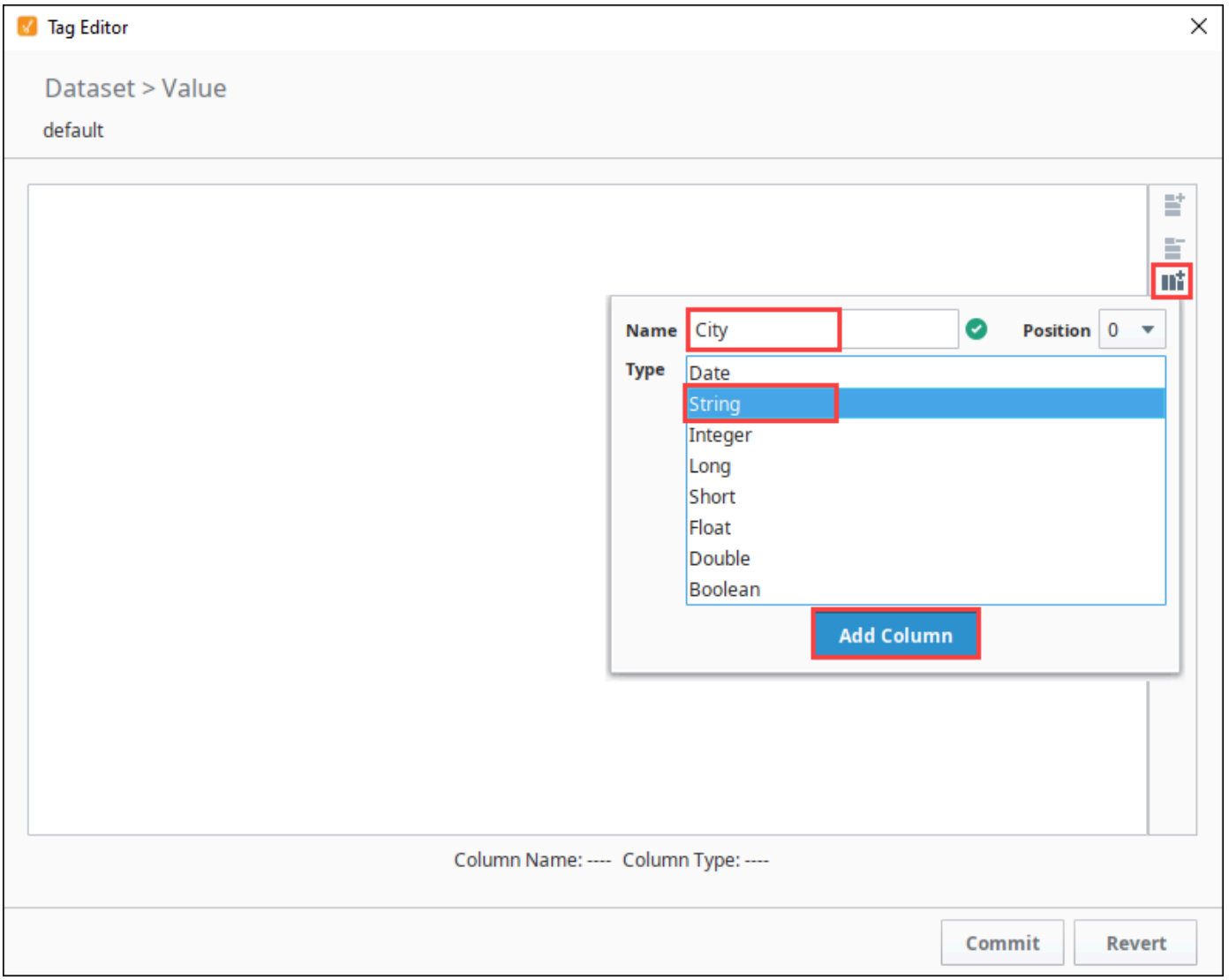
The following example will create a dataset Memory tag and display the contents in a Table component.

1. Create a new **Memory Tag**. Name it **Dataset**, and set the Data Type to **Dataset**. The dataset will be empty by default.



2. Click the **Edit** icon next to **Value** to edit the dataset. For this example, create a simple dataset with four columns and five rows.

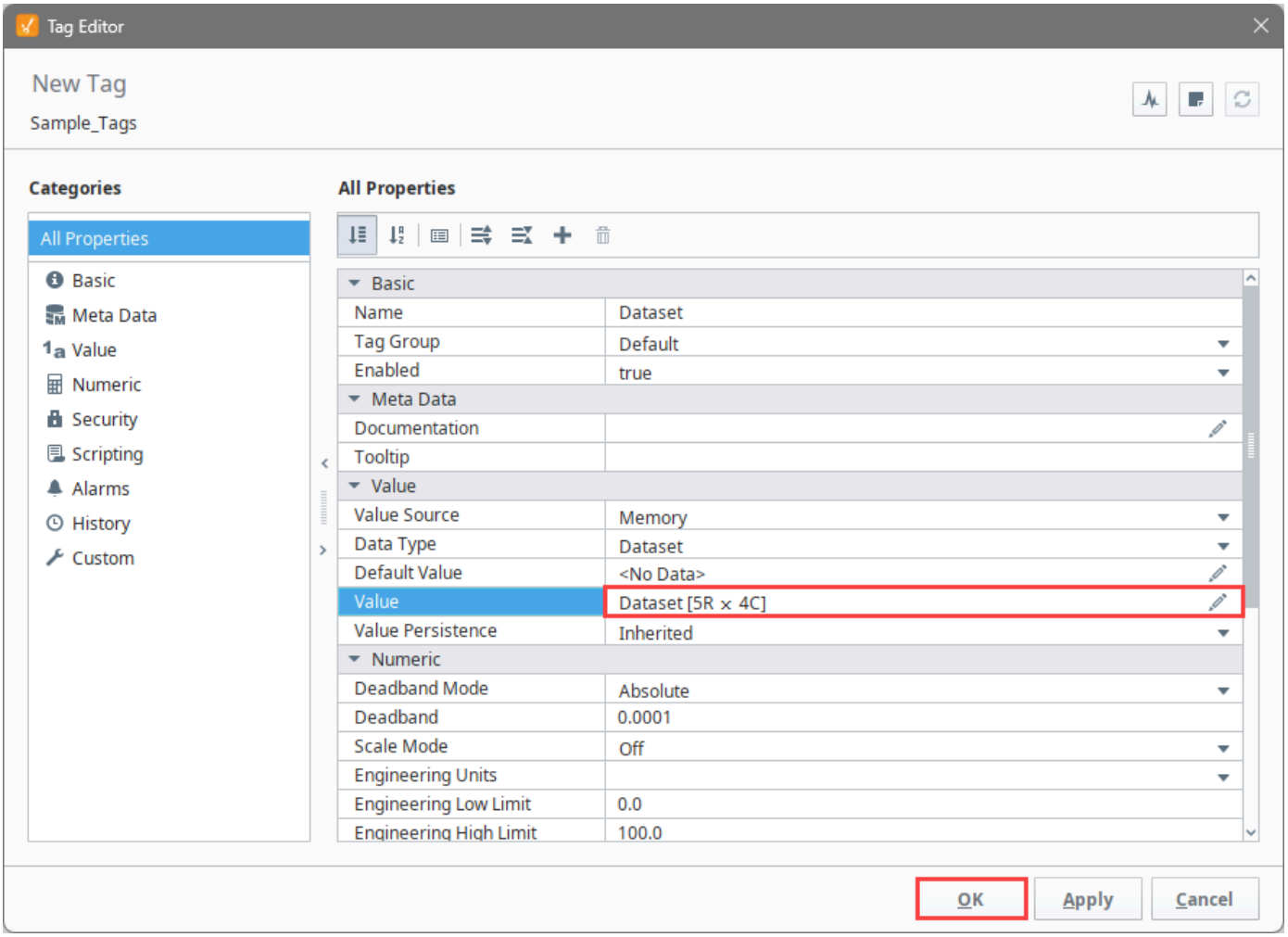
3. Click the **Add Column** icon. Name the first column **City** and set the type to **String**.



4. Add the following columns:
    - Name: **Population**
        - Type: **Integer**
    - Name: **TimeZone**
        - Type: **String**
    - Name: **GMTOffset**
        - Type: **Integer**
5. Click the **Add Row** icon and enter the following values.

New York	8368710	EST	-5
Los Angeles	3833995	PST	-8
Chicago	2853114	CST	-6
Houston	2242193	CST	-6
Phoenix	1567924	MST	-7

6. Click **Apply**.
7. Click **OK** to save the tag.



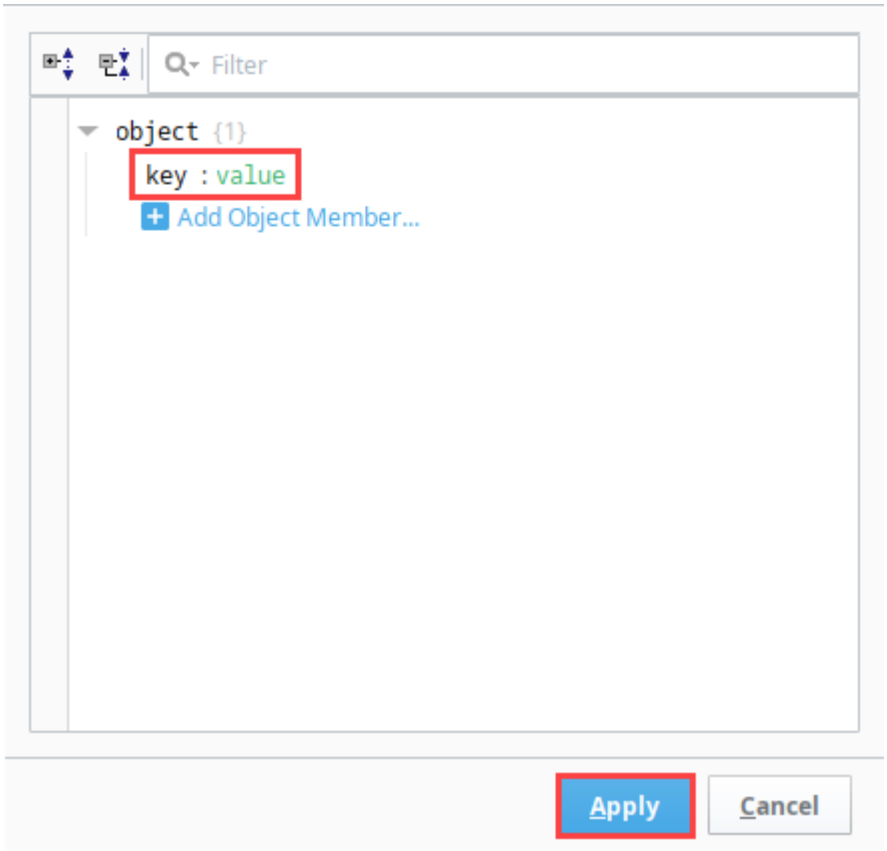
The tag now contains rows, columns, and values based on the configuration created in this example. The dataset value can be bound to components in Vision and Perspective.

Document Type

The Document type allows a tag to store a JSON document as its value.

> **NOTE**
>
> The Tag History system and most OPC servers do not support the Document data type. This type works best when used on Query tags or Memory tags

Click the **Edit** icon for the Value field to open the JSON Editor, which will allow you to manually write JSON values directly. Then, click **Add Object Member** to add new members to the JSON object. When finished, click **Apply**.



The tag value now appears as a JSON document in the **Tag Browser**.

The Document type will now accept and coerce PyDictionary values.

Tag Paths

Tags and their properties can be referenced by a string-based path in many areas of Ignition, such as **expressions** and **scripts**. Each tag has a unique absolute path and often has many equivalent relative paths when referenced from other tags. In most cases these paths are generated automatically via helper buttons. However, it's a good idea to understand how tag paths work, particularly if you need to configure an Indirect Tag Binding, or access a tag from an expression or script.

A tag path looks something like this: `[Tag Provider]folder/path/tag.property`

The `folder/path/tag.property` portion of the path may contain the following:

- A tag
- Any number of nested folders followed by a tag, separated by forward slashes (/)
- A period (.) followed by a property name after the tag. Omitting this is equivalent to using the .value property

The `[Tag Provider]` portion surrounded by square braces can have the following options:

Source Option	Meaning
`[Tag Provider Name]`	The name of the Tag Provider that hosts the tag.
`[]` or not specified	The default Tag Provider for the current project. If used in the Gateway scope, this notation can (generally) result in an invalid path, as the Gateway doesn't have a default Tag Provider.
`[.]`	Relative to the folder of the tag that is being bound. This is especially useful in UDT definitions.
`[~]`	Relative to the Tag Provider of the tag that is being bound (root node).
`[Client]`	Refers to the Vision Client Tag Provider, which contains only Vision Client Tags.
`[System]`	Refers to a System Tag.

Using Relative Tag Paths

Tag paths that begin with `[.]` or `[~]` are known as *relative* paths. They are used inside tags that bind to other tags, and they are relative to the host tag's path. Using the relative path syntax helps to avoid problems caused by moving tags and renaming providers.

`[.]` refers to the tag's current folder. By using [.], tags can be moved from folder to folder without problem (provided that all of the applicable tags are moved together). Additionally, you can use "`..`" (two periods) to go back one folder from the current relative position, for example `[.]../../tag` allows you to reference a tag that is two folders up.

`[~]` refers to the Tag's Provider root. It replaces an explicit provider name and thus protects the tag path from "breaking" if the provider is renamed or if the tag is imported/exported/moved between different providers.

Tag Path Manipulation

Ignition provides a great deal of flexibility for tag addressing since tag paths and tag properties are string-based. The underlying strings that compose a valid tag path can be assembled from many different parts in which the eventual construction results in a valid tag path.

The following scripting demonstrates this concept. Suppose there was a tag path to a level indicator in a tank. In this case it is the default Tag Provider, Tanks folder, Tank 1 Folder, and the Level tag.

```
tagPath = "[default]Tanks/Tank 1/Level"
```

But suppose that there was more than just Tank 1, and instead there was Tank 2, Tank 3, Tank 4, etc. Dynamically changing the tag paths is simple because Ignition's tag paths are string representations. The following takes the tank number and inserts it into a new tag path. The tankNumber variable changes the eventual creation of the tagPath. Using this method in scripting or in an expression binding will look slightly different.

In the following example, `tankNumber` is a variable for a python script:

Python Dynamic Tag Path

```
tankNumber = 2
tagPath = "[default]Tanks/Tank %i/Level" % tankNumber
```

Then, in this next example, `tankNumber` is a separate tag being referenced within an expression tag of the same folder:

Expression Dynamic Tag Path

```
tag("[default]Tanks/Tank "+{[.]tankNumber}+"/Level")
```

The result of the tagPath variable will be `[default]Tanks/Tank 2/Level` which is a valid tag path to the level sensor for Tank 2.

Array Type Tag Paths

When a path leads to an array type tag, individual elements can be accessed using square brackets, and the index offset.

```
[default]Folder/myArrayTag[0]
```

Historical Tag Paths

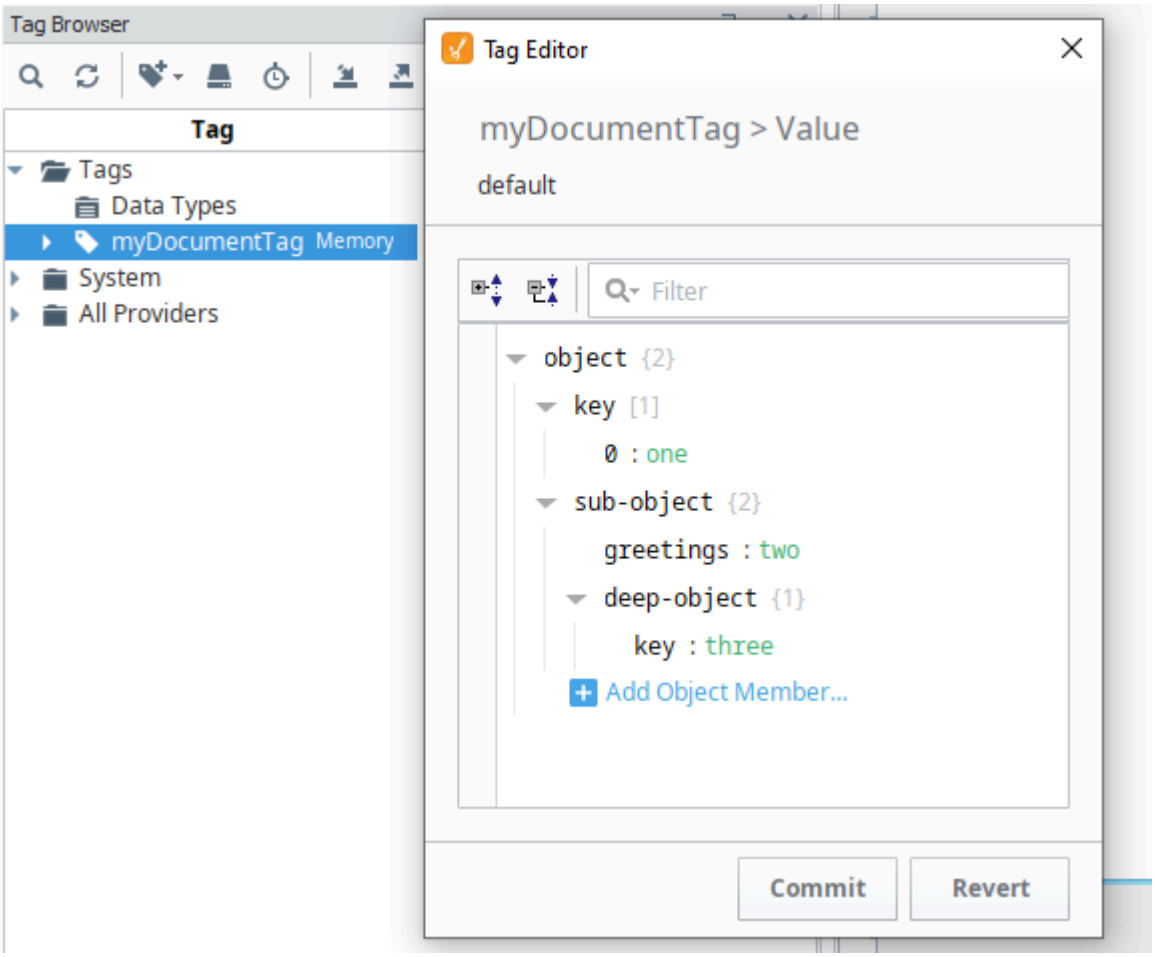
Components like the **Power Chart** and the **Tag Browse Tree** use historical tag paths when historical values are needed. The **system.historian** functions also support historical paths.

Historical Tag Path Syntax

```
histprov:test:/sys:myGateway:/prov:default:/tag:_Simulator_/Ramp/Ramp0
```

Document Type Tag Paths

When a path leads to a document type tag, individual objects within the tag's value can be accessed by using square brackets wrapped around a set of quotations marks that lead to the object. For example, say we have a documentation tag with the following value:



The following would return "**one**", which is in the first element of the "**key**" array:

```
[default]myDocumentTag['key[0]']
```

JSON Values are references with a '.' character, so the following would retrieve the value of "two" inside of the "sub-object":

```
[default]myDocumentTag['sub-object.greetings']
```

For more complex structures, you can continue adding to the JSON string. The following would return "**three**" from the key in "**deep-object**":

```
[default]myDocumentTag['sub-object.deep-object.key']
```

Writing to Document Type Tags

The paths above aren't only for reading. Using the same addressing mentioned above on a bi-directional component binding would allow the binding to write back to the specific JSON object in the document.

Using the Keyword

Tags have a built-in "this" keyword, that can be used as a reference to the tag. The keyword is useful in cases where an expression is being configured on a tag property and you want to reference the value of another property.

```
// The expression below always returns the name of the tag
{this.name}
```

Using in Alarms

The "this" keyword is also available for use on expressions on alarms. However when used like this, the keyword still refers to the tag, not the alarm. Thus, "`this.name`" on an alarm property expression would return the name of the tag, not the name of the alarm.

For more information on Ignition's Expression language, see **Expression Overview and Syntax**.

Wildcards in Tag Paths

New in 8.3.4

Gateway Event Tag Change Scripts and the **Event Streams Tag Event source** accept glob-style wildcard operators in the tag paths you supply. A single wildcard path can subscribe to many tags at once, so you can target an entire folder, a naming pattern, or a set of UDT instances without listing each tag individually.

Note that wildcards were always supported for Tag Change Scripts at the folder level, but they will now work at both the folder level and the tag level. This page describes the supported operators and common patterns.

When to Use Wildcards

Use a wildcard path when you would otherwise list many tags that share a naming pattern or a common folder. Typical cases include wanting to monitor or respond to:

- Any tag change under a line or area.
- Every `Alarm_*` tag across a Tag Provider.
- UDT-wide subscriptions where the instance structure is repeated across the plant.

If you only need to cover one or two specific tags, use a normal path. Explicit paths are easier to implement and are the default for targeted subscriptions.

Supported Operators

Wildcards are resolved against the tag path segment by segment.

Operator	Description	Example
`*`	Matches any characters within a single path segment.	`[default]Folder/*` subscribes to every tag directly inside `Folder`.
`**`	Matches any number of intermediate folders.	`[default]Folder/**/Tag` matches `Tag` at any depth below `Folder`.
`?`	Matches exactly one character.	`[default]Tag?` matches `Tag1` and `Tag2`, but not `Tag10`.
`??`, `???`, ...	Matches a fixed count of characters.	`[default]Tag??` matches `Tag10` and `Tag42`, but not `Tag1`.

Operators can also be combined in a single path. For example, `[default]Folder/**/Tag_*` matches any `Tag_<suffix>` at any depth below `Folder`. Wildcards can appear as part of a tag name, not only as a full segment, so `[default]Tag_*` subscribes to every tag in the Tag Provider root whose name starts with `Tag_`.

Wildcards are supported on non-default Tag Providers using the same syntax. For example, `[second]Folder/**/Tag_*` applies the pattern to the `[second]` Tag Provider, and `[default]**/Instance*/**` matches every tag inside every UDT instance whose name starts with `Instance`, at any depth.

Entering Wildcard Paths

Wildcard paths go in the Tag Path(s) field, one path per line. The **Tag Browser** icon on the right is useful for starting with an explicit tag path and then editing it to introduce wildcards.

A single Tag Event source or Tag Change Script accepts any combination of explicit and wildcard paths. Each line is resolved independently.

> **NOTE**
>
> When you hover over a wildcard path, the tooltip displays **Glob pattern detected, unable to validate.** This is expected. Wildcard paths cannot be validated at edit time, so the pattern is evaluated on the Gateway at runtime against the live tag tree.

> **TIP**
>
> Wildcard subscriptions have been verified against a folder containing 10,000 tags with no subscription issues. The match set is resolved on the Gateway, so downstream handlers will see one event per matching tag. Design for fan-out the same way you would for explicit subscriptions.

Common Patterns

The following patterns cover the majority of integration use cases:

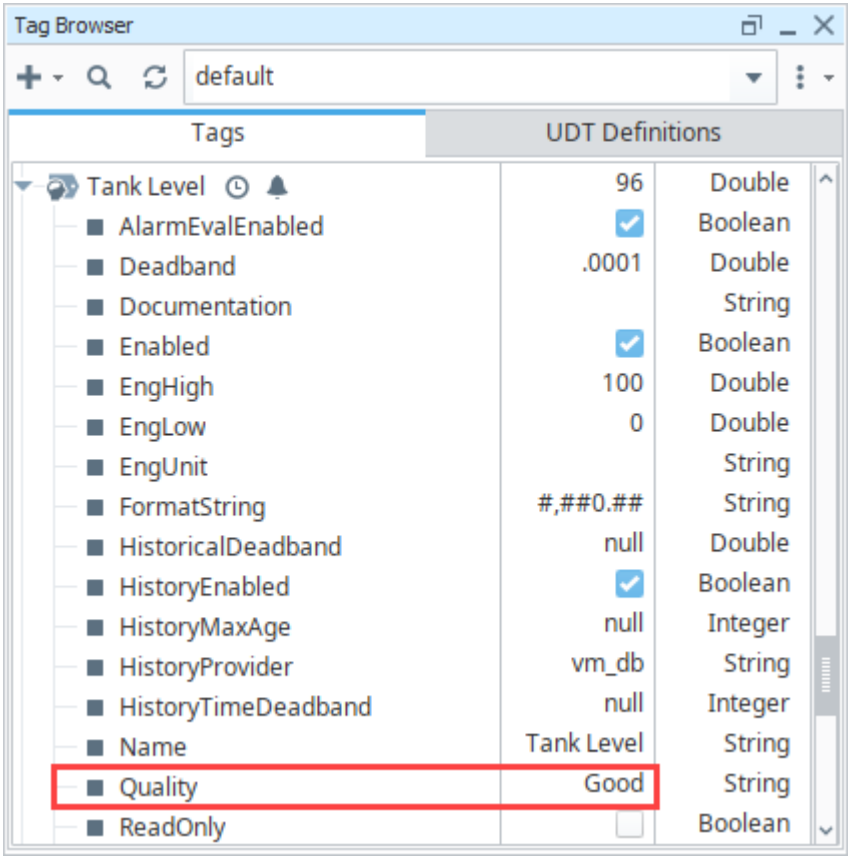
- One source per line, many tags: `[default]Line<n>/**/Alarm_*` subscribes every alarm on a line.
- UDT-wide monitoring: `[default]**/Motor*/**` catches every tag inside every `Motor` UDT instance at any depth.
- Naming-convention filters: `[default]Area1/Tag_???` selects fixed-length tag names, which is useful when tag IDs are zero-padded.
- Per-provider routing: `[default]Folder/Tag_*` and `[second]Folder/Tag_*` subscribe the same glob on `[default]` and `[second]` that then branch downstream on the Tag Provider.

Quality Codes and Overlays

A Quality Code represents a level of confidence in a value. Quality codes are made up of a level (Good, Bad, Error, or Uncertain) and an integer. If a value's quality is not Good, the value generally should not be trusted.

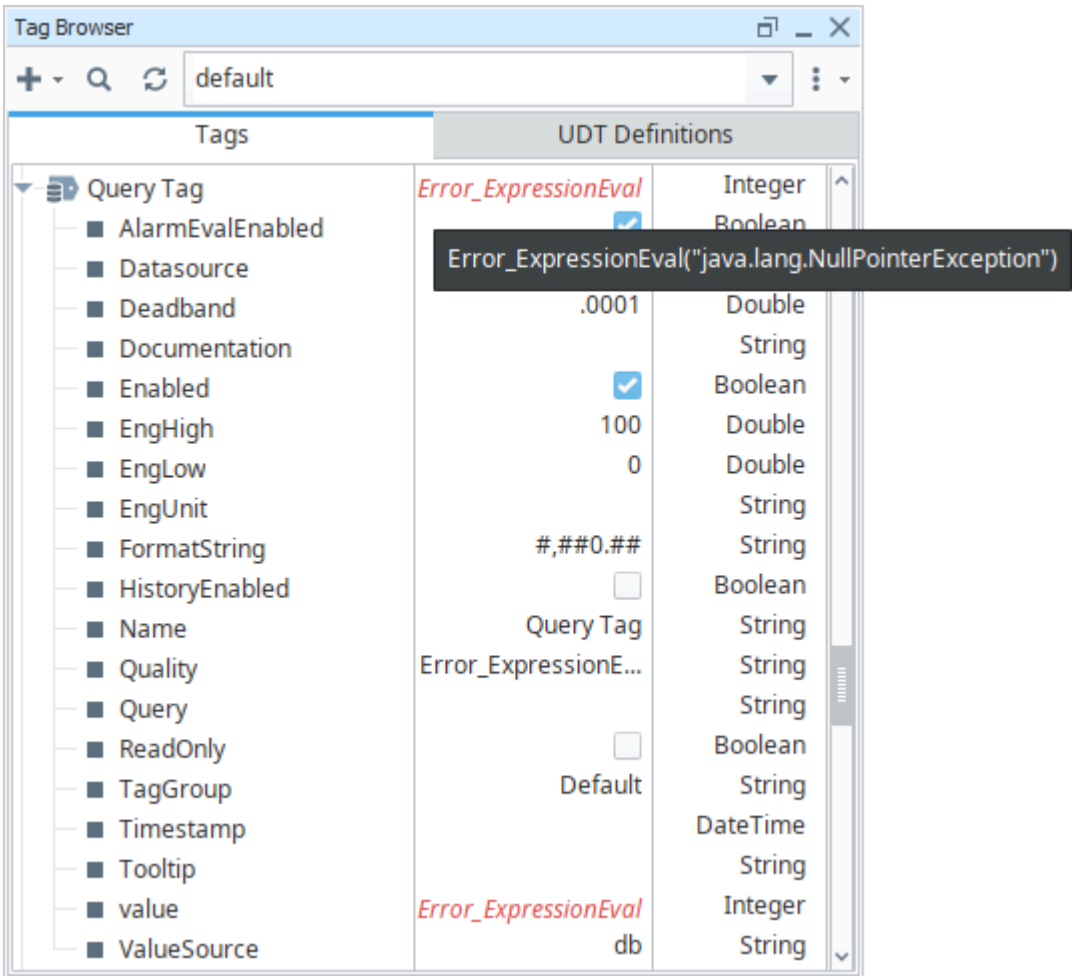
Tag Quality in the Designer

In the Tag Browser, find your Tag, expand it, and scroll down to the meta property called **Quality**. Here, you can check the quality of the Tag. This example shows a Good Quality Tag, meaning the Tag can be trusted.



One obvious indicator if the Tag is of bad quality is if there is a red error message next to the Tag in the Value column. Hover over the error message to see if there is any additional information about the

error. You can also expand the Tag to see the quality issue. This example shows the Query Tag with with a Error Expression Evaluation which helps you resolve the issue promptly.



Component Overlays

HMI screens allow users to quickly gauge the health and accuracy of what is displayed at a glance. In a highly distributed system like Ignition, it is especially important as the client may be located at quite a distance (maybe across the world) from the physical process it is monitoring and controlling.

For these reasons, Perspective and Vision components display visual overlays for various reasons to indicate that the data they are displaying is not good, or pending a reply from the device. Each data binding that drives a component is evaluated for quality. If any of these qualities becomes poor, the Perspective or Vision component will show an overlay. The different overlays can mean different things, denoting their underlying cause. What they indicate is based on the Quality properties of Tags.

Component overlays appear in the Designer workspace, Perspective Session, and Vision Client to let designers and operators know when there is a problem with one of the bindings on a component. What is cool about component overlays is that they not only tell you that there is a problem, but they also help diagnose the problem. Vision and Perspective overlay systems are similar, but each look a little bit different.

The sections below describe in detail Perspective and Vision overlays. Each module has its own Tag Quality Code Reference Table displaying the error codes and what they mean.

Quality Code Reference Table

There are four primary quality codes which quickly inform the user of the quality of the Tag: Good, Uncertain, Bad, and Error.

OPC UA clients are now allowed to write to exposed Tags, which include a StatusCode. This StatusCode is converted to its Qualitycode equivalent, while the write value is passed to the Tag system as a QualifiedValue.

> **NOTE**
>
> For users who have upgraded from 7.9, be aware that old code values have been adapted to match the quality codes below. Because the quality the old values are adapted, tag history quality results and bindings can still be trusted in your upgraded versions. However, it is important to check areas where manual entry may prevent adapted values from populating correctly such as scripts for live checks of tag quality.

Each quality code has a range of subcodes that provide more specific information about the value of the Tag. The following tables outline the primary data qualities. Quality codes are broken up into several ranges. Note that not every value within a range has a code: the space is there for future codes to be added.

- 0-225: Good quality. The value is generally considered reliable.
- 256 - 511 : Uncertain quality. Generally represents a value that was good, but the reliability is somewhat questionable. These are generally used when the system hasn't received a new value in a timely manner.
- 512 - 767: Bad quality. There's a problem with the value, but it's an "expected" or well recognized type of the problem is. For example, the trial expired or read access was explicitly denied.
- 768 - 1023: Error quality. There is a problem with the value, and the problem was completely unexpected. For example, a query on a query tag couldn't execute, but the Tag doesn't know why. More details on such errors are usually sent to a console somewhere, such as the Gateway's console.

Good Quality (subcodes 0-255)

Quality	Subcode	Description
Good_Unspecified	0	A generic "good" code. Generally used in conjunction with a matching good quality subcode, (1,2, or 192).
Good_WritePending	2	Used when a write is in progress. Generally, values use this code until the system knows the write through successfully, which would then result in a 192 code.
Good	192	This data has met all criteria for being considered reliable.
Good_Provisional	200	Good data that should not be considered valid long-term.
Good_Initial	201	Indicates that the value is an initial/seed value for a system that is starting up.
Good_Overload	202	Represents good data that is being sampled slower than requested due to a resource limitation.
Good_Backfill	203	Used to indicate good quality values that have arrived out of order. Different systems can choose to process them accordingly.

Uncertain Quality (subcodes 256-511)

Quality	Subcode	Description
Uncertain	256	An unspecified degree of uncertainty exists in this value.
Uncertain_LastKnownValue	257	The current value is unavailable and represents the last known value.
Uncertain_InitialValue	258	Indicates that a subscription has been made and a good value should be arriving shortly.
Uncertain_DataSubNormal	259	Insufficient good-quality sources required for the derivation of this value.
Uncertain_EngineeringUnitsExceeded	260	Indicates that a value has gone beyond its configured engineering units.
Uncertain_IncompleteOperation	261	An asynchronous operation is currently pending and its result is unknown.

Bad Quality (subcodes 512-767)

Quality	Subcode	Description
Bad	512	General code for a bad value.
Bad_Unauthorized	513	An unauthorized request was made for data that requires authorization.
Bad_AccessDenied	514	Data requested that requires credentials not held by the requesting user.
Bad_Disabled	515	Data source is currently not enabled.
Bad_Stale	516	Data is out-of-date based upon the requested refresh interval.
Bad_TrialExpired	517	The Trial Mode's timer expired.
Bad_LicenseExceeded	518	The license limit has been exceeded.
Bad_NotFound	519	Object requested was not found.
Bad_ReferenceNotFound	520	Derived or referenced value required an object which was not found.
Bad_AggregateNotFound	521	Requested aggregate was not found.
Bad_NotConnected	522	A connection required for this value is not currently connected.
Bad_GatewayCommOff	523	Connection to the Ignition Gateway is currently turned off. See **General Designer Interface**.
Bad_OutofRange	524	This value exceeded its allowed range.
Bad_DatabaseNotConnected	525	A database connection required for this value is not connected.
Bad_ReadOnly	526	A write was attempted on a read only target.

Quality	Subcode	Description
Bad_Failure	527	A "failure" code was received from the underlying system. Additional details may be in the diagnostic message. This generally does not indicate an exception, which would be handled by Error_Exception, but instead a simple failure from a system that can return success or failure.
Bad_Unsupported	528	The operation is not supported by the target.

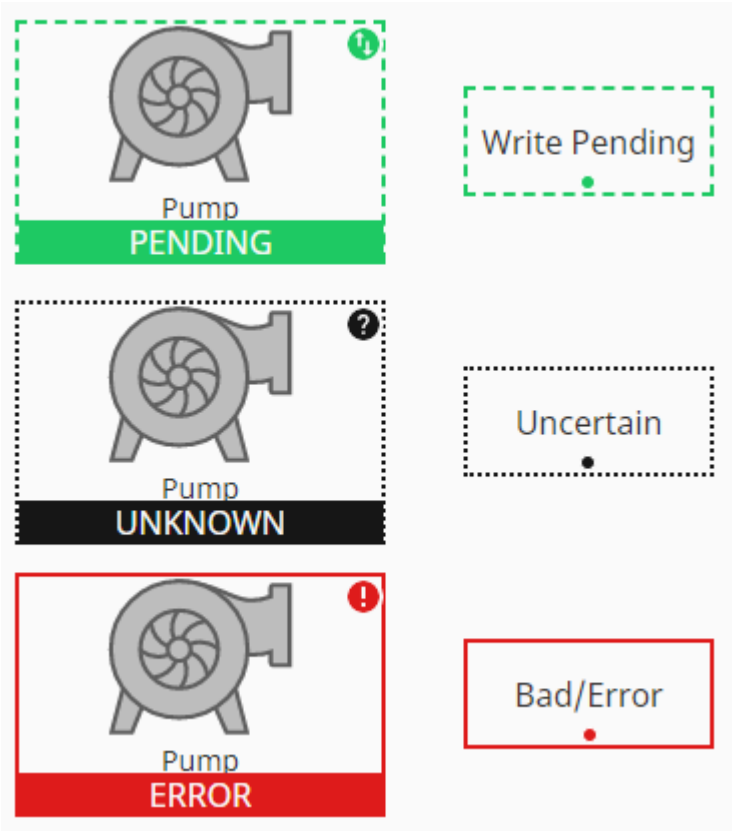
Error Quality (subcodes 768-1023)

Quality	Subcode	Description
Error	768	An unexpected error occurred while retrieving or calculating this value.
Error_Configuration	769	The source of this value is not configured correctly.
Error_ExpressionEval	770	The source expression was unable to be executed.
Error_TagExecution	771	The source Tag could not be executed.
Error_TypeConversion	772	The actual value was not able to be coerced into the configured data type for the source of this value.
Error_DatabaseQuery	773	A database query required for this value caused an error upon execution.
Error_IO	774	An input/output error occurred while attempting to retrieve or calculate this value.
Error_TimeoutExpired	775	An asynchronous operation failed due to a timeout.
Error_Exception	776	An exception was caught, and logged in the relevant system.
Error_InvalidPathSyntax	777	A path (i.e., Tag path, property path, etc.,) was not able to be parsed because the syntax is invalid.
Error_Formatting	778	Attempted formatting (i.e., numeric, date formatting) failed.
Error_ScriptEval	779	A script needed to create this value failed to execute.
Error_CycleDetected	780	Calculating the value involved an execution cycle.

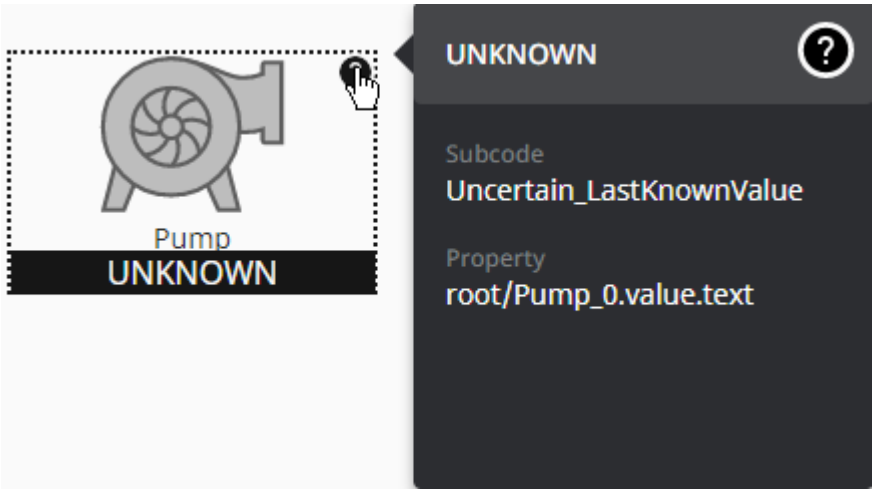
Perspective Component Overlays

Component properties may show a quality overlay if the source of data has a certain code. In Perspective, there are three types of quality overlays, each with a large and small variant.

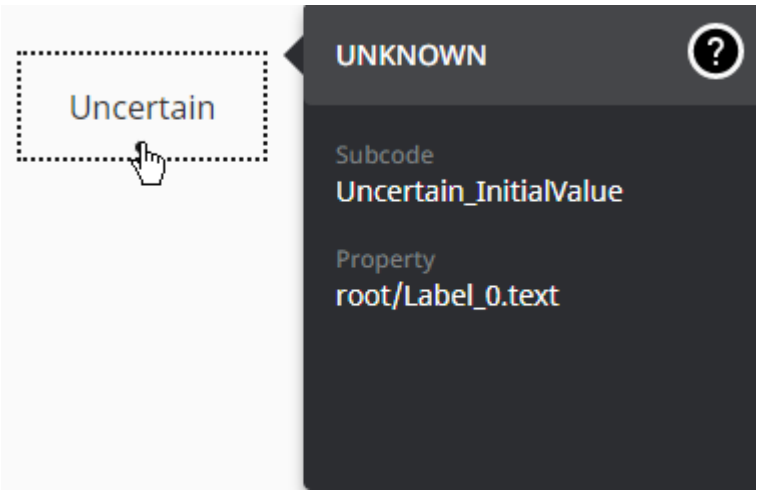
- The Pending overlay is displayed for Good quality, subcode 2 only
- The Unknown overlay is displayed for any Uncertain quality subcode
- The Error overlay is shown for any Bad and Error quality, regardless of subcode



For any large overlay, clicking the icon in the upper right will yield diagnostic information.



For the smaller overlays, clicking on the dot will show the diagnostic information.



> **NOTE**
>
> All Perspective components, regardless of type, will use the three overlays described above. Unlike Vision, details for Perspective overlays are not explicitly displayed.

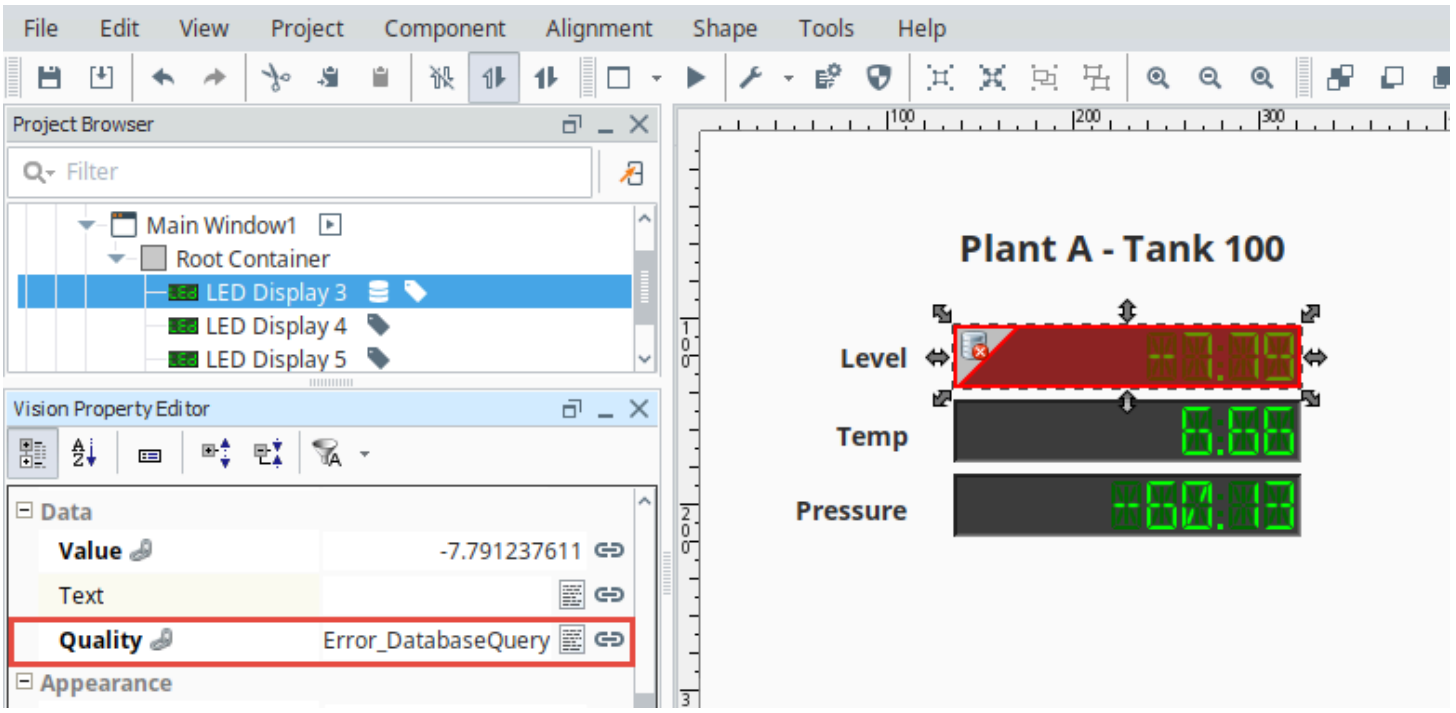
Vision Component Overlays

An overlay on a Vision component lets the operator know that they could be looking at a bad value for that Tag. When the overlay goes away and the values start coming in again, the operator knows that it's a valid Tag, and the values can be trusted.

Component Overlays in Designer Mode

In the following example, you see a red overlay with an icon in the top left corner of the selected LED Display component. The icon gives you a clue to the source of the problem. In this example, it is an SQL Database error. In the Vision Property Editor, the Quality property is highlighted and you'll notice there is a "Error_DatabaseQuery" error message.

The overlays table in the next section show all the possible Vision overlays and what they mean.



Component Overlays in Preview Mode

Let's switch from the Designer mode to the Preview Mode. To put your active view in **Preview Mode**, press the Preview / Designer mode icon ▶ in the top menubar. Components that have a problem will have a red overlay and an icon in the top left of the component overlay to indicate the problem. The overlay is identical to the overlay that is displayed in the Designer, but the component cannot be selected.

Component Overlays in the Vision Client

Component overlays in a Vision Client work the same way as they do in Preview Mode of the Designer. You have to look at the icon on the overlay to help you diagnose the problem. Go back to the Designer to correct the problem.

Vision Component Overlay Chart

For legacy reasons, Vision has different subcodes. Possible subcodes are listed below.

Good_WritePending
Good_WritePending (2)

Good_Provisional
Good_Provisional (200)

Good_Initial
Good_Initial (201)

Good_Overload
Good_Overload (202)

Good_Backfill
Good_Backfill (203)

Uncertain
Uncertain (1073742080)

Uncertain_LastKnownValue
Uncertain_LastKnownValue (1073742081)

Uncertain_InitialValue
Uncertain_InitialValue (1073742082)

Uncertain_DataSubNormal
Uncertain_DataSubNormal (1073742083)

Uncertain_EngineeringUnitsExceeded
Uncertain_EngineeringUnitsExceeded (1073742084)

Uncertain_IncompleteOperation
Uncertain_IncompleteOperation (1073742085)

Bad
Bad (-2147483136)

Bad_Unauthorized
Bad_Unauthorized (-2147483135)

Bad_AccessDenied
Bad_AccessDenied (-2147483134)

Bad_Disabled
Bad_Disabled (-2147483133)

Bad_Stale
Bad_Stale (-2147483132)

Bad_TrialExpired
Bad_TrialExpired (-2147483131)

Bad_NotFound
Bad_NotFound (-2147483129)

Bad_ReferenceNotFound
Bad_ReferenceNotFound (-2147483128)

Bad_AggregateNotFound
Bad_AggregateNotFound (-2147483127)

Bad_NotConnected
Bad_NotConnected (-2147483126)

Bad_GatewayCommOff
Bad_GatewayCommOff (-2147483125)

Bad_OutOfRange
Bad_OutOfRange (-2147483124)

Bad_DatabaseNotConnected
Bad_DatabaseNotConnected (-2147483123)

Bad_ReadOnly
Bad_ReadOnly (-2147483122)

Bad_Failure
Bad_Failure (-2147483121)

Bad_Unsupported
Bad_Unsupported (-2147483120)

Error
Error (-1073741056)

Error_Configuration
Error_Configuration (-1073741055)

Error_ExpressionEval
Error_ExpressionEval (-1073741054)

Error_TagExecution
Error_TagExecution (-1073741053)

Error_TypeConversion
Error_TypeConversion (-1073741052)

Error_DatabaseQuery
Error_DatabaseQuery (-1073741051)

Error_IO
Error_IO (-1073741050)

Error_TimeoutExpired
Error_TimeoutExpired (-1073741049)

Error_Exception
Error_Exception (-1073741048)

Error_InvalidPathSyntax
Error_InvalidPathSyntax (-1073741047)

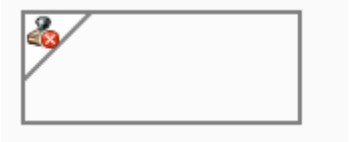
Error_Formatting
Error_Formatting (-1073741046)

Error_ScriptEval
Error_ScriptEval (-1073741045)

Error_CycleDetected
Error_CycleDetected (-1073741044)

Template Overlay

In addition, there is a quality overlay for an unknown template. This can occur when the Template Path property on a template instance does not resolve to an existing template. This can happen due to a name change on the template definition, or if a project import file was recently imported, and included usages of a template instance but not the definition.

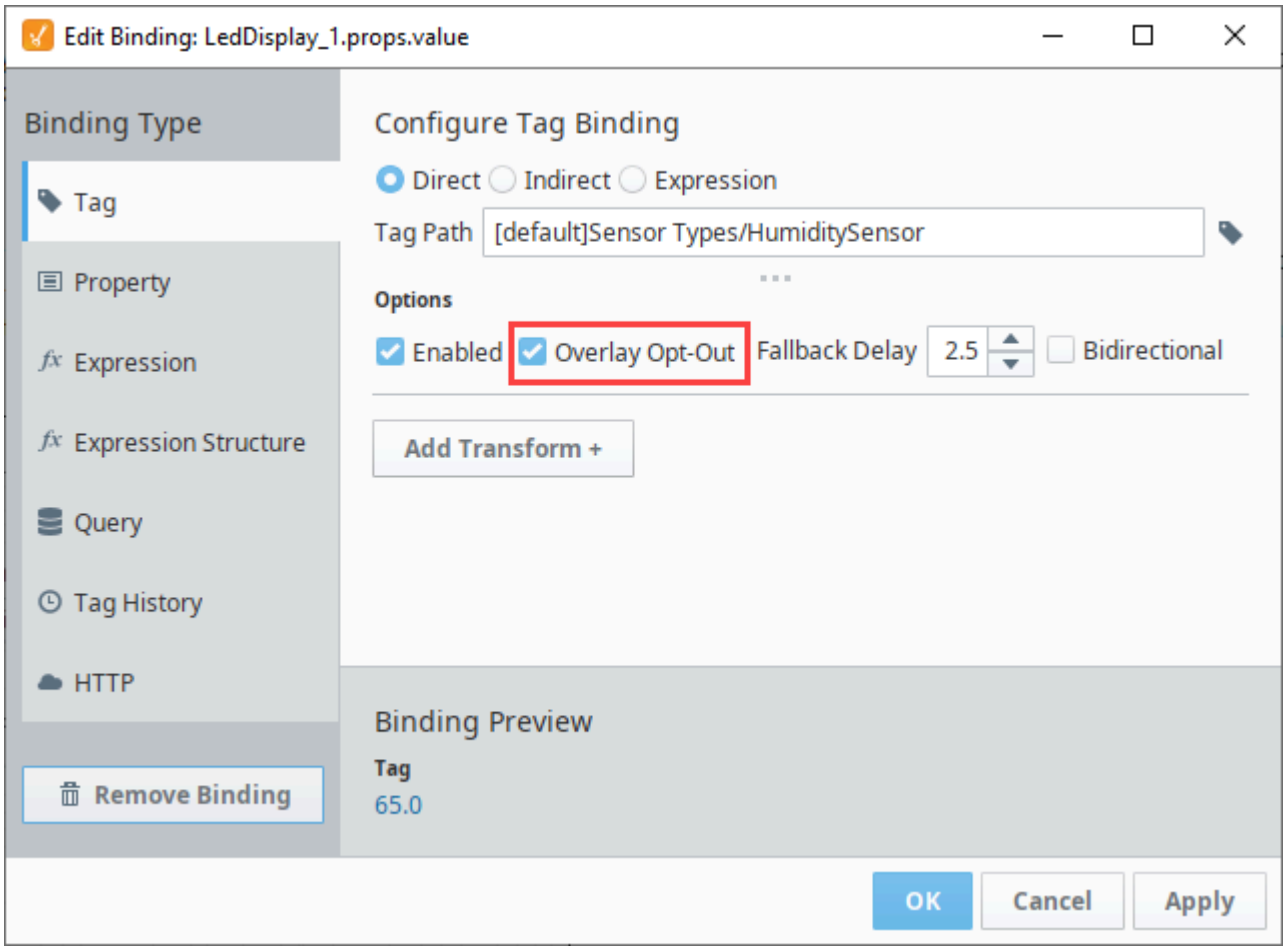


Tag Quality and Referenced Tags

When Tags reference other Tags, such as in expressions, they will often pass the worst sub-quality up as their own. For example, even though a particular Tag's expression executes without problem, if the expression references a Tag whose quality is "Bad", the expression Tag will also report "Bad."

Overlay Opt-Out

Choosing the Overlay Opt-Out option will ignore the quality of the chosen Tag, making it have no effect on the component's quality overlay. The Overlay Opt-Out option is located in the Tag bindings for both Perspective and Vision components. If this option is enabled, the operator will not see any overlays and will have no indication that the underlying Tag quality is something other than good. A word of caution when you use the Opt-Out option because you always want to give the operator some indication that the values they are seeing on the screen can be trusted, and by opting out, you are removing that indicator for the operator.



Exporting and Importing Tags

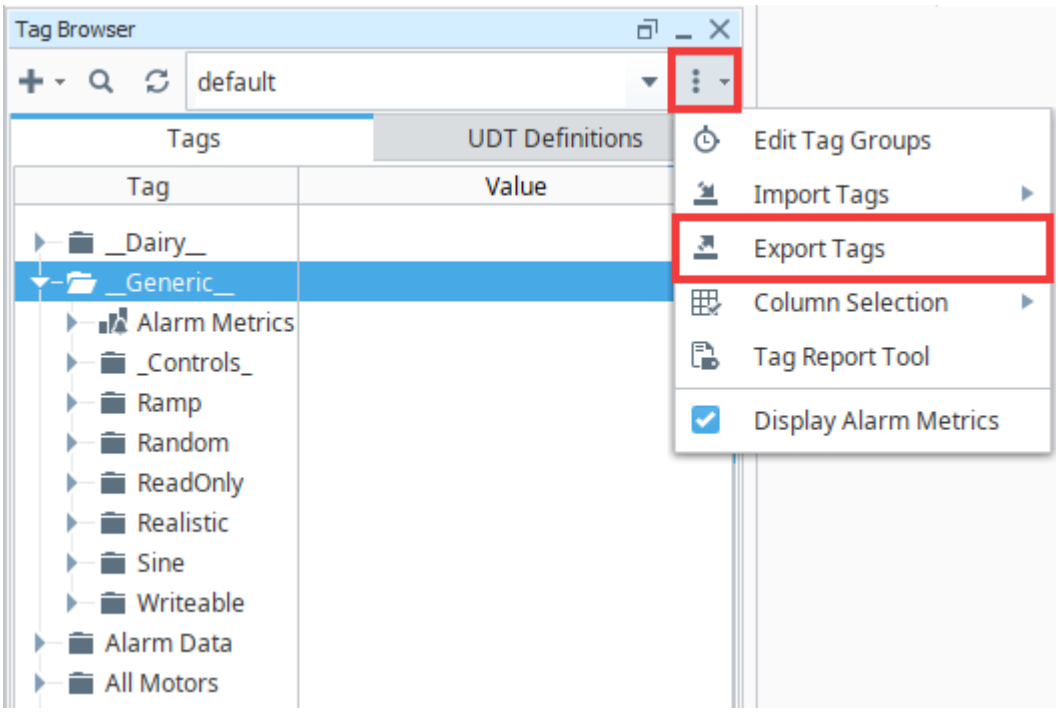
Ignition can export and import tag configurations to and from the **JSON** (JavaScript Object Notation) format. You can import **XML** (Extensible Markup Language) or **CSV** (Comma Separated Value) file formats as well, but Ignition will convert them to **JSON** format while in a tag provider. Tag export files can be edited directly in any text editor, allowing you to make bulk edits to tags before importing them back into a tag provider.

Export Tags

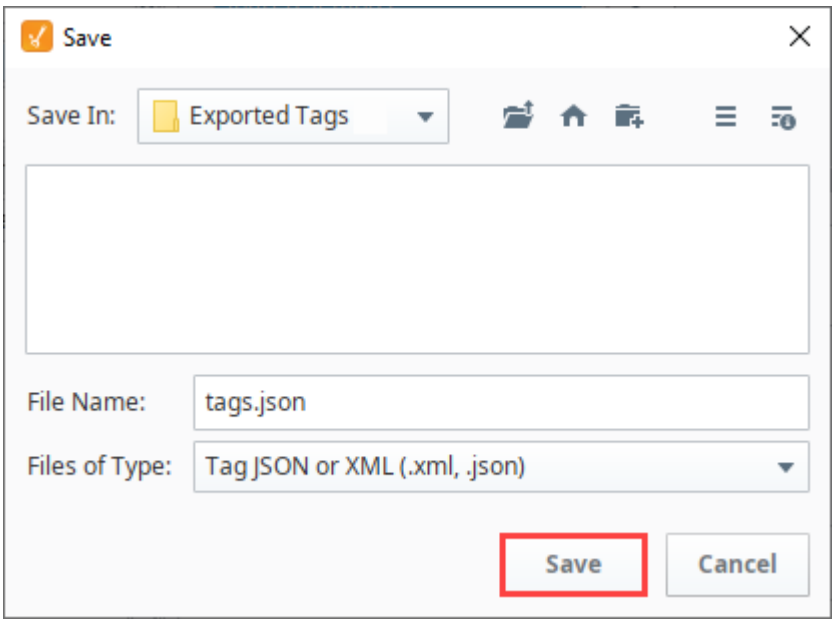
You can export tags in the Designer **Tag Browser** by selecting the Tags tab to export all your tags or selecting an individual folder that contains tags you want to export. You can even export individual tags as long the individual tags are in the same folder.

> **EXPORTING UDTS AND UDT INSTANCES**
>
> If you select a UDT instance to export, the UDT definition is not automatically included. You must export the definition as a separate file by clicking on the UDT Definitions tab.
>
> When later importing these UDTs, it is recommended to import UDT definitions before importing any instances.

1. On the **Tag Browser** toolbar, click the More Options menu to open the dropdown. Select **Export Tags**.



2. Specify the folder you want to save your exported tag files in the **Save** window, and then click **Save**. Ignition will export tags by default to a .**json** file.



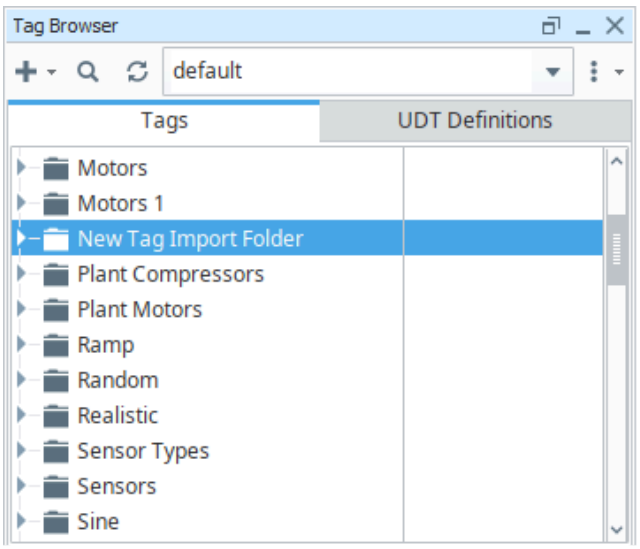
Import Tags

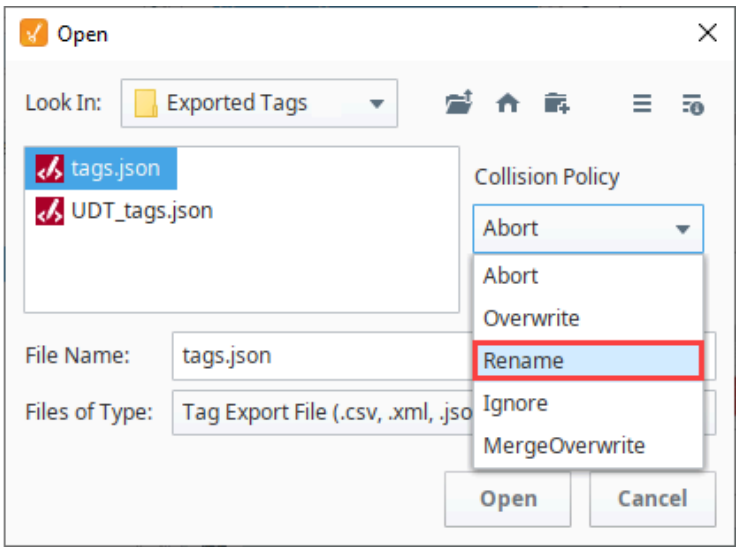
You can import tags to an individual folder or under the Tags tab. When importing tags, you need to set the Collision Policy to tell Ignition how to handle duplicate tags. If any of the tags being imported already exist in the folder you specify, Ignition can abort the import, overwrite the tags, rename them, ignore them, or merge them.

Collision Policy	Description
Abort	Aborts the import if duplicate tags are found.
Overwrite	Overwrites any tags in the folder that have the same name as tags being imported. Note this a complete overwrite of the tag. Importing UDT definitions with this collision policy will remove any members that are not included in the import file.
Rename	Renames any duplicate tags.
Ignore	Ignores duplicate tags and imports only those that are unique.
MergeOverwrite	Overwrites the tag with the exception of any properties that aren't defined in the import folder. Those properties will be merged.

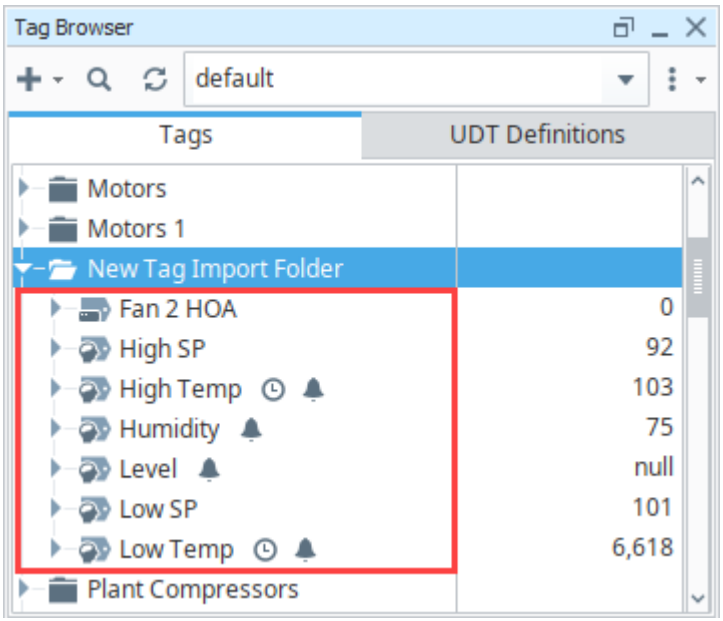
To import tags:

1. In the Tag Browser toolbar, right-click an existing folder and select **Import Tags** > **Direct**.
2. Specify the folder you want to import your tags from and choose a previously exported file, either **.json** , **.xml** , or **.csv** file type.
3. Choose a Collision Policy.
4. Click **Open** to import the tags.



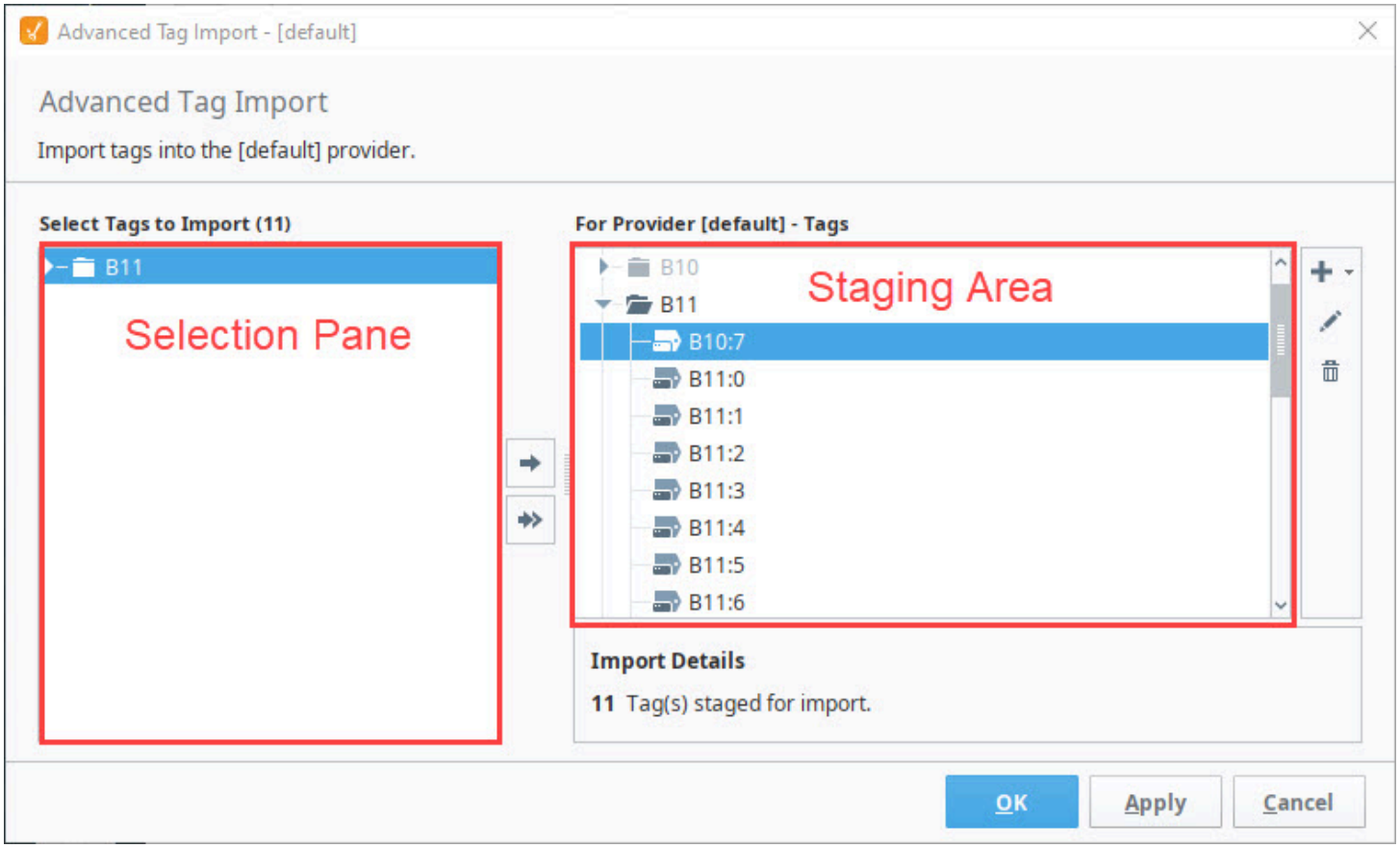


The tags are now displayed in the Tag Browser.



Interactive Tag Import

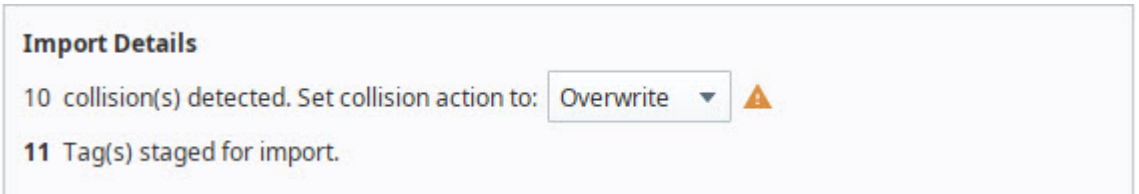
The Interactive tag import option opens the Advanced Tag Import tool, which allows users to edit tag properties in the Designer before import. It combines the functionality of the basic Import Tags tool with the Tag Editor window.



Interface Element	Description
	Adds the selected tag, folder, or UDT to the Staging Area.
	Adds all tags, folders, and UDTs in the Selection Pane to the Staging Area.
	Creates a new folder or tag at the selected node in the Staging Area, or the root if a node is not selected.
	Opens a Tag Editor window for the selected tag or UDT.
	Deletes the selected tag or UDT from the Staging Area.

1. From the Tag Browser, click the **More Options** menu (or right-click on a folder) and select **Import Tags** > **Interactive**.
2. Select the **.json**, **.xml**, or **.csv** file to import.
3. Move the desired tags and UDTs from the Selection Pane to the Staging Area using the **Add** or **Add All** options.
4. Edit tags in the Staging Area by selecting a tag and clicking **Edit**
5. If any collisions are detected, you will be prompted to select a Collision Policy from the dropdown menu.

> **NOTE**
>
> Unlike the basic Import Tags tool, you may only choose to **Overwrite** or **Ignore** duplicate tags imported via the Advanced Tag Import tool.



6. Click **OK** to import the tags and UDTs

The tags will now appear in the Tag Browser.

> **NOTE**
>
> If your tags are not appearing as expected for a large tag import, the Designer's memory allocation may need to be increased. Access your Gateway and navigate to the **Config** > Gateway Settings > **Designer Memory** to adjust memory limitations. The default size is 1.0 GB, with available dropdown options from MB128 to 4.0 GB.

System Tag Alarms and Configurations

Unlike regular tags, you cannot use the Tag Browser's built-in export and import tool for System tags. You can instead use scripting to export and import the exported System tags' configurations on a different system. See the **Exporting and Importing System Tag Alarms** page for examples on how to do this.

Tag File Formats

Tags can be imported from CSV, JSON, and XML. Tags can only be exported in XML or JSON. There are many configuration settings for tags than what is displayed in a JSON or XML export file. The tag export feature only exports the configuration properties that have been edited in at least one of the tags in the selected export folder. Therefore, to ensure the desired configuration setting is available in the export file, at least one tag within the selected export folder must have that configuration property changed

CSV Format

Importing

Ignition supports importing tags from a CSV format. Details of the format are below. This format can contain tag types, OPC paths, and most tag properties. One difference between the CSV format and the XML and JSON format is that the CSV format does not include support for alarm configurations. Alarms can certainly be added to tags in the Ignition Designer after tags have been imported from CSV, but alarms cannot be defined directly in the CSV.

Below is an example of the legacy CSV format, which contains the following tag types, purely for demonstrative purposes:

- An OPC tag
- A folder
- An OPC tag located in a folder

- A Derived tag
- An Expression tag
- A Memory tag
- A Query tag

```
Path,Name,Owner,TagType,DataType,Value,Enabled,AccessRights,OPCServer,OPCItemPath,Sca
# version=1,,,
```

Property Name	Type	Description
Value	*varies*	The value of the tag, dependent on the data type.
Data Type	Int	The data type of the tag. Possible values are: • 0: Int1 • 1: Int2 • 2: Int4 • 3: Int8 • 4: Float4 • 5: Float8 • 6: Boolean • 7: String • 8: DateTime • 9: DataSet
Enabled	Boolean	Determines if the tag should be enabled or disabled upon import.
Tagtype	Int	Determines the type of the tag. A value of one is a "DB Tag", which is either a Memory tag, Query tag, or Expression tag, depending on the value of the ExpressionType field. Possible values are: • 0: OPC Tag • 1: DB Tag (see ExpressionType) • 2: Client Tag • 6: Folder • 10: UDT Instance • 13: Derived Tag
ExpressionType	Int	Used in conjunction when the TagType is set to 1 (DB Tag), otherwise this field is ignored. Possible values are: • 0: None • 1: Expression • 2: SQL Query

Property Name	Type	Description
AccessRights	Int	If custom, will be defined by a **Permissions** tag. Possible values are: • 0: Read Only • 1: Read/Write • 2: Custom
OPCServer	String	The server against which to subscribe the data point. Only present for OPC tags.
OPCItemPath	String	Represents the OPC item path for the tag. Only present for OPC tags.
OPCWriteBackServer	String	The configured write back OPC server for Expression tags.
OPCWriteBackItemPath	String	Represents an OPC item path on the configured write back server, in cases where expression tags are configured.
ScaleMode	Int	If and how the tag value will be scaled between the source, and what is reported for the Tag. • 0: Off • 1: Linear • 2: Square Root • 3: Exponential Filter
ScaleFactor	Float	A numerical value representing the scale factor
RawLow	Float	A numerical value representing the raw low value. Only used when ScaleMode is set to Linear or Square Root.
RawHigh	Float	A numerical value representing the raw high value. Only used when ScaleMode is set to Linear or Square Root.

Property Name	Type	Description
ScaledLow	Float	A numerical value representing the scaled low value. Only used when ScaleMode is set to Linear or Square Root.
ScaledHigh	Float	A numerical value representing the scaled high value. Only used when ScaleMode is set to Linear or Square Root.
ClampMode	Int	Which clamp mode to use. Possible values are: • 0: None • 1: Low • 2: High • 3: Both
Deadband	Float	A numerical value used to prevent unnecessary updates for tags whose values change by small amounts. Setting to 0.0 disables the deadband mode functionality.
DeadbandMode	Int	While Deadband is set to a non-zero value, determines which mode to use. Possible values are: • 0: Absolute • 1: Percentage
FormatString	String	How the value should be formatted when converted to a string (only applies to numerical data types). Uses # and 0 characters to describe the format. **#** : If the number in this position is non-zero, then do not show the position. Otherwise, show the number. Useful when you only want to show a decimal position if the value is non-zero. **0** : If the number in this position is non-zero, then show that number. Otherwise, show a zero. Useful to add leading and trailing zeros to a value.
EngUnit	String	The engineering units of the tag.

Property Name	Type	Description
EngLow	Float	The lowest expected value of the tag.
EngHigh	Float	The highest expected value of the tag.
EngLimitMode	Int	Dictates how the engineering range should be enforced on the tag. If not Off, the tag will change to bad quality (limit exceeded), when the value falls outside the specified range. Valid values are as follows: • 0: None • 1: Low • 2: High • 3: Both
Tooltip	String	The tooltip provides a hint to visual components as to what should be displayed when the user hovers their mouse cursor over the component that is being driven by the value of this tag.
Documentation	String	A freeform text property for information about the tag.
DriverName	String	The name of the driver. Only used by tags configured for tag-driving providers.
ScanClass	String	The export will only include the name of the ScanClass, not the configuration of the Scanclass itself. A Scanclass with the same name needs to already exist on the Gateway that the tags are being imported to, prior to importing them.
HistoryEnabled	Boolean	Determines if the Historian Core module should record values from this tag.
PrimaryHistoryProvider	String	The history provider to use if storing history with the Historian Core module.

Property Name	Type	Description
HistoricalDeadband	Float	A numerical value used to prevent unnecessary updates for tags whose values change by small amounts. Setting to 0.0 disables the HistoricalDeadbandMode functionality.
HistoricalDeadbandMode	Int	While HistoricalDeadbandDeadband is set to a non-zero value, determines which mode to use. Possible values are: • 0: Absolute • 1: Percentage
HistoricalScanclass	String	The export will only include the name of the ScanClass, not the configuration of the Scanclass itself. A Scanclass with the same name needs to already exist on the Gateway that the tags are being imported to, prior to importing them.
InterpolationMode	Int	How values are interpolated. 2 exists for backwards compatibility (and is equivalent to 1), but only 0 or 3 should be used in the future. • 0: Discrete • 2: Analog (deadband) • 3: Analog (compressed)
HistoryTimestampSource	Int	Possible values are: • 0: System • 1: Value
HistoryMaxAgeMode	Int	Possible values are: • 0: Unlimited • 1: Limited
HistoryMaxAge	Int	Max cycles between storage.
UDTParentType	String	The path to the parent UDT type. Used by sub-types and instances.

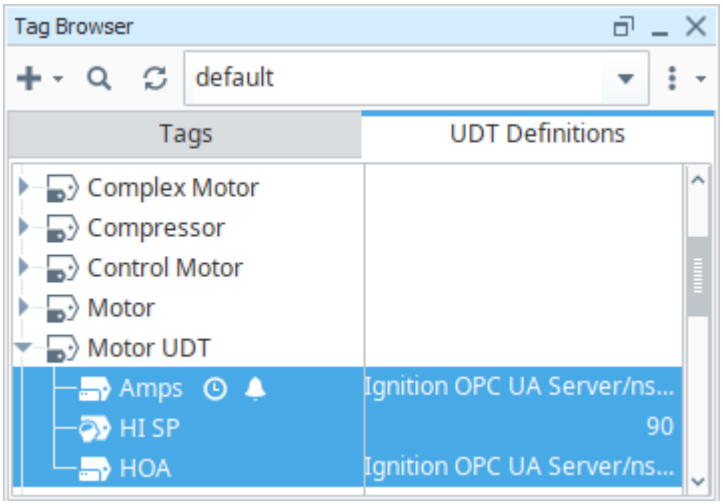
Exporting

Although Ignition can import tags from a CSV format, Ignition does not export tags to a CSV format. Since multiple alarms can be configured per tag, the XML or JSON formats provide a much better format to allow for the tree structures needed to fully represent Ignition tags.

JSON Example

In this example, we exported three tags from the Motor UDT in JSON format.

- Amps: Expression tag with an Alarm and History enabled
- HI SP: Memory tag which is bound to a parameter to an OPC tag
- HOA: OPC tag



JSON Export Format Example

The following exported tag file is in JSON format. As you browse through the JSON file, you will see the tag properties and configuration settings for each of the three tags listed above. Descriptions on the various properties can be found on the **Tag Properties** page.

Exported Tags in JSON Format

```
{
  "tags": [
    {
      "opcItemPath": {
        "bindType": "parameter",
        "binding": "ns\u003d1;s\u003d[Dairy]_Meta:Overview/Motor
{MotorNumber}/Amps"
      },
      "valueSource": "opc",
      "historyProvider": "MySQL",
      "alarms": [
        {
          "mode": "BelowValue",
          "setpointA": 25.0,
          "name": "Low Amps",
          "priority": "Critical",
          "displayPath": {
            "bindType": "Expression",
            "value": "Motor{MotorNumber}"
          }
        },
        {
          "mode": "AboveValue",
          "name": "High Amps",
          "priority": "Critical",
          "setpointA": {
            "bindType": "Expression",
            "value": "{[.]HI SP}"
          }
        }
      ],
      "name": "Amps",
      "historyEnabled": true,
      "tagType": "AtomicTag",
      "opcServer": "Ignition OPC UA Server"
    },
    {
      "valueSource": "memory",
      "name": "HI SP",
      "value": 90,
      "tagType": "AtomicTag"
    },
    {
      "opcItemPath": {
        "bindType": "parameter",
        "binding": "ns\u003d1;s\u003d[Dairy]_Meta:Overview/Motor
{MotorNumber}/HOA"
      },
```

```
      "valueSource": "opc",
      "name": "HOA",
      "tagType": "AtomicTag",
      "opcServer": "Ignition OPC UA Server"
    }
  ]
}
```

XML Example

In this example, we exported the same three tags from our Tag Browser, that were also used in the JSON example. As you browse through the XML file, you will see the tag properties and configuration settings for each of the same three tags. Descriptions on the various properties can be found on the **Tag Properties** page.

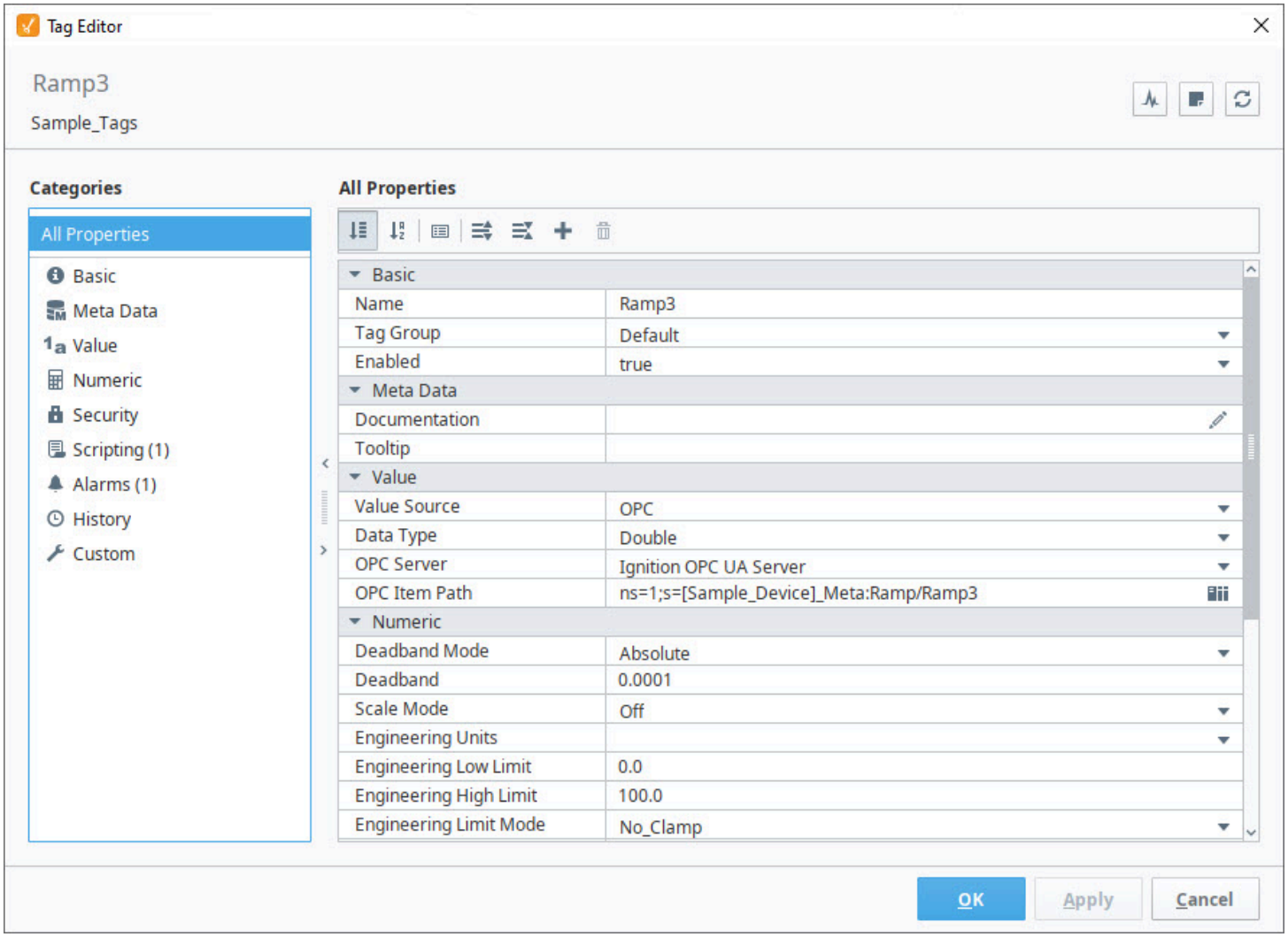
Exported Tags in XML Format

```
<Tags MinVersion="8.0.0" locale="en_US">
   <Tag name="Amps" type="AtomicTag">
      <Property name="opcItemPath" boundValueType="parameter">ns=1;s=
[Dairy]_Meta:Overview/Motor {MotorNumber}/Amps</Property>
      <Property name="valueSource">opc</Property>
      <Property name="historyProvider" datatype="String">MySQL</Property>
      <CompoundProperty name="alarms">
         <PropertySet>
            <Property name="mode">3</Property>
            <Property name="setpointA">25</Property>
            <Property name="name">Low Amps</Property>
            <Property name="priority">4</Property>
            <Property name="displayPath" bindtype="Expression">Motor{MotorNumber}
</Property>
         </PropertySet>
         <PropertySet>
            <Property name="mode">2</Property>
            <Property name="name">High Amps</Property>
            <Property name="priority">4</Property>
            <Property name="setpointA" bindtype="Expression">{[.]HI SP}</Property>
         </PropertySet>
      </CompoundProperty>
      <Property name="historyEnabled" datatype="Boolean">true</Property>
      <Property name="opcServer">Ignition OPC UA Server</Property>
   </Tag>
   <Tag name="HI SP" type="AtomicTag">
      <Property name="valueSource">memory</Property>
      <Property name="value">90</Property>
   </Tag>
   <Tag name="HOA" type="AtomicTag">
      <Property name="opcItemPath" boundValueType="parameter">ns=1;s=
[Dairy]_Meta:Overview/Motor {MotorNumber}/HOA</Property>
      <Property name="valueSource">opc</Property>
      <Property name="opcServer">Ignition OPC UA Server</Property>
   </Tag>
</Tags>
```

Tag Editor

The Tag Editor is a robust interface that contains all the properties that can be configured for tags. In the Tag Editor, you set the tag's name, value, numeric and meta data properties, security, alarming, history, and more. For information an explanation on all possible tag configurations, see **Tag Properties**.

> **INFO**
>
> 8.3 tags cannot be edited by 8.1 systems. Attempting to edit an 8.3 tag from 8.1 will cause the Tag Editor to throw an error.



Diagnostics, Documentation, and Refresh icons are included on the upper right-hand side of the Tag Editor to access more tag information:

- The **Diagnostics** icon opens the **Tag Diagnostics** window.
- The **Documentation** icon displays a Documentation pane at the bottom of the Tag Editor.

- The **Refresh** icon refreshes the editor with the current definition of the tag from the Gateway.

The Tag Editor requires fewer clicks and keeps relevant tag information visible while modifying bindings, alarms, and event scripts.

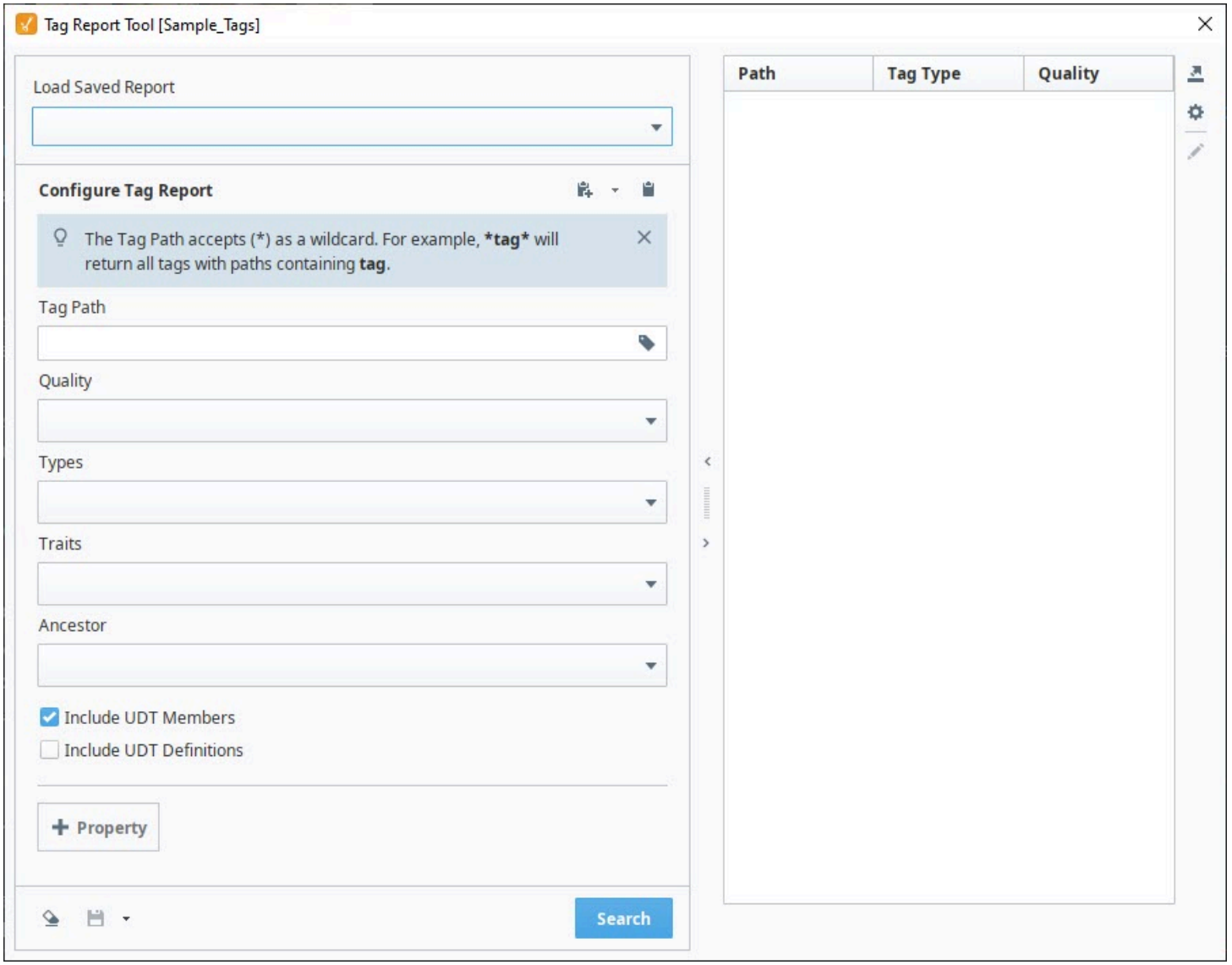
Some features include:

- Expression bindings require opening the dialog to change. Tag and Parameter bindings can be edited in-line within the Tag Editor and are not shortened.
- Sorting in UDT Editor sorts folders first, then tags.
- Users can add tags in the UDT Editor without having to click the root of the UDT/folder.
- UDT Parameters show when they are overridden.
- UDT properties are added in-line to bindings.
- The Expression editor accepts drag and drops from the UDT Editor or Tag Browser.

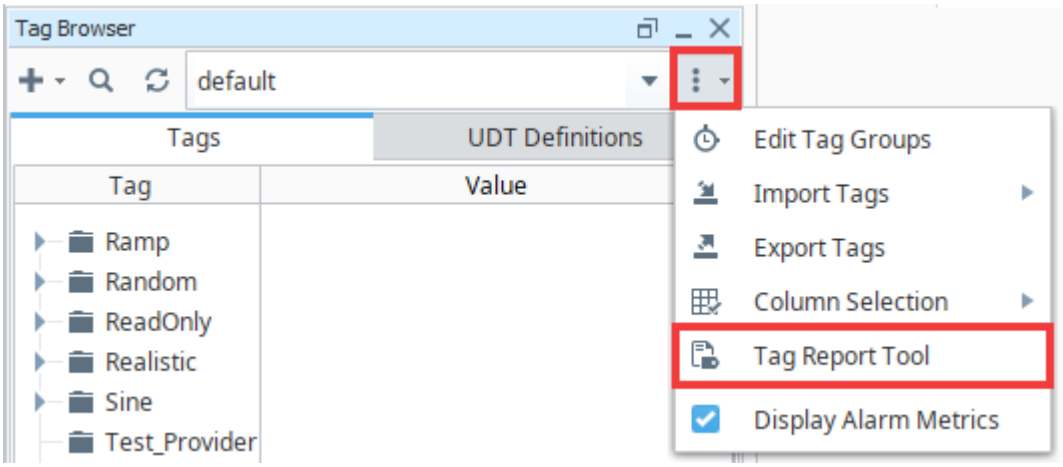
Tag Diagnostics data is displayed in its own window.

Tag Report Tool

The Tag Report Tool allows users to search for tags by multiple criteria, including **Tag Path**, **Quality**, **Type**, **Traits**, and values on specific **Properties**.



The Tag Report Tool is accessible through the **More Options** menu on the Tag Browser. Select the Tag Provider you wish to search, then click the icon and select **Tag Report Tool** to begin configuring your report.



> **NOTE**
>
> The Tag Report Tool can only search within one Tag Provider at a time. To search within a different Tag Provider, exit the Tag Report Tool and select a new Tag Provider from the drop-down menu.

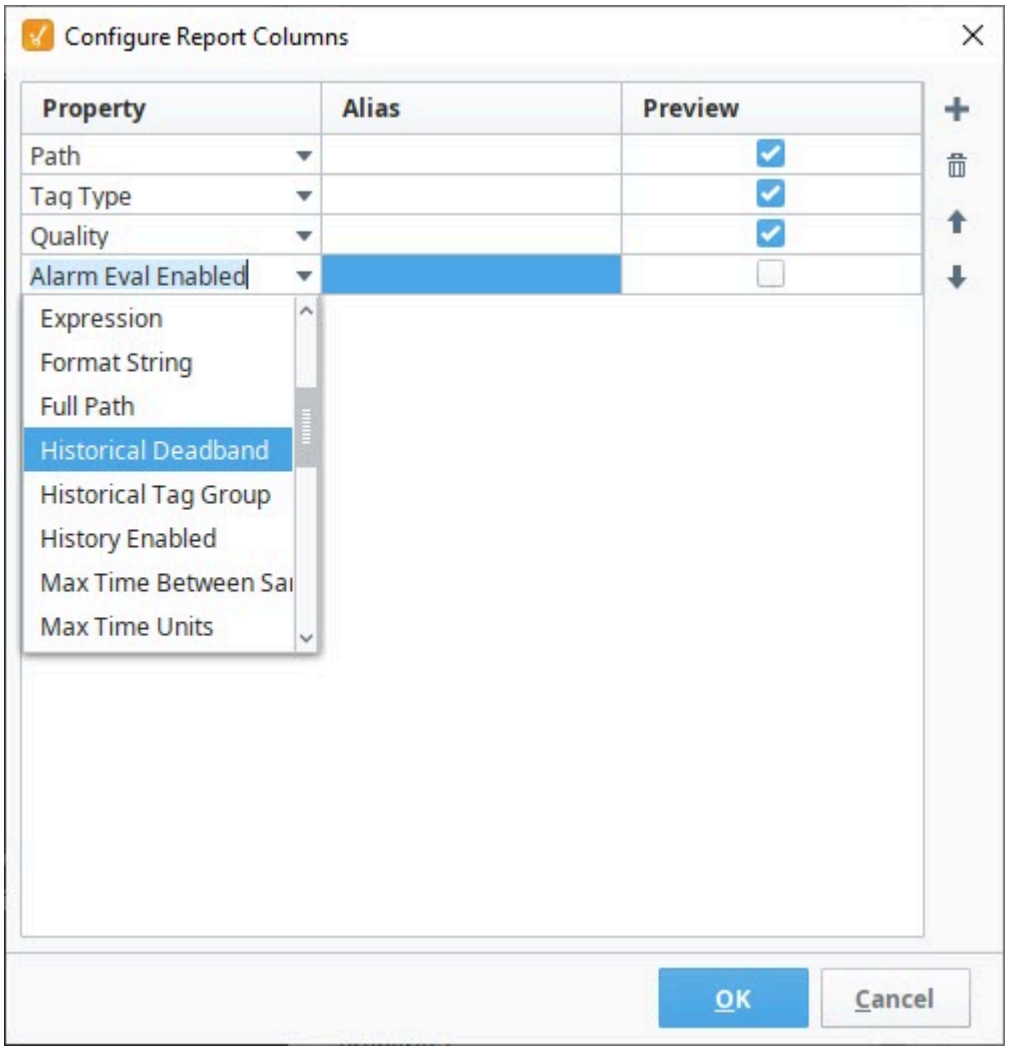
Configure, Save, and Copy Reports

Save your query as a Saved Report to easily repeat searches with complex criteria. Saved reports are accessible to all users and all designers on the same Gateway, and can be reused to generate reports for different Tag Providers.

You can also **Copy** a report as either a JSON object or as a script. Copying a report as a JSON object allows you to **Paste** the same query into the Tag Report Tool on a separate Gateway. Copying a report as a Script will allow you to paste a system.tag.query() script anywhere in Ignition where scripting can be used.

Export Reports

Export reports as CSV files to analyze data in Excel or other platforms outside of Ignition. Configure Report Columns to add, remove, reorder, or assign aliases to columns displayed in the report.



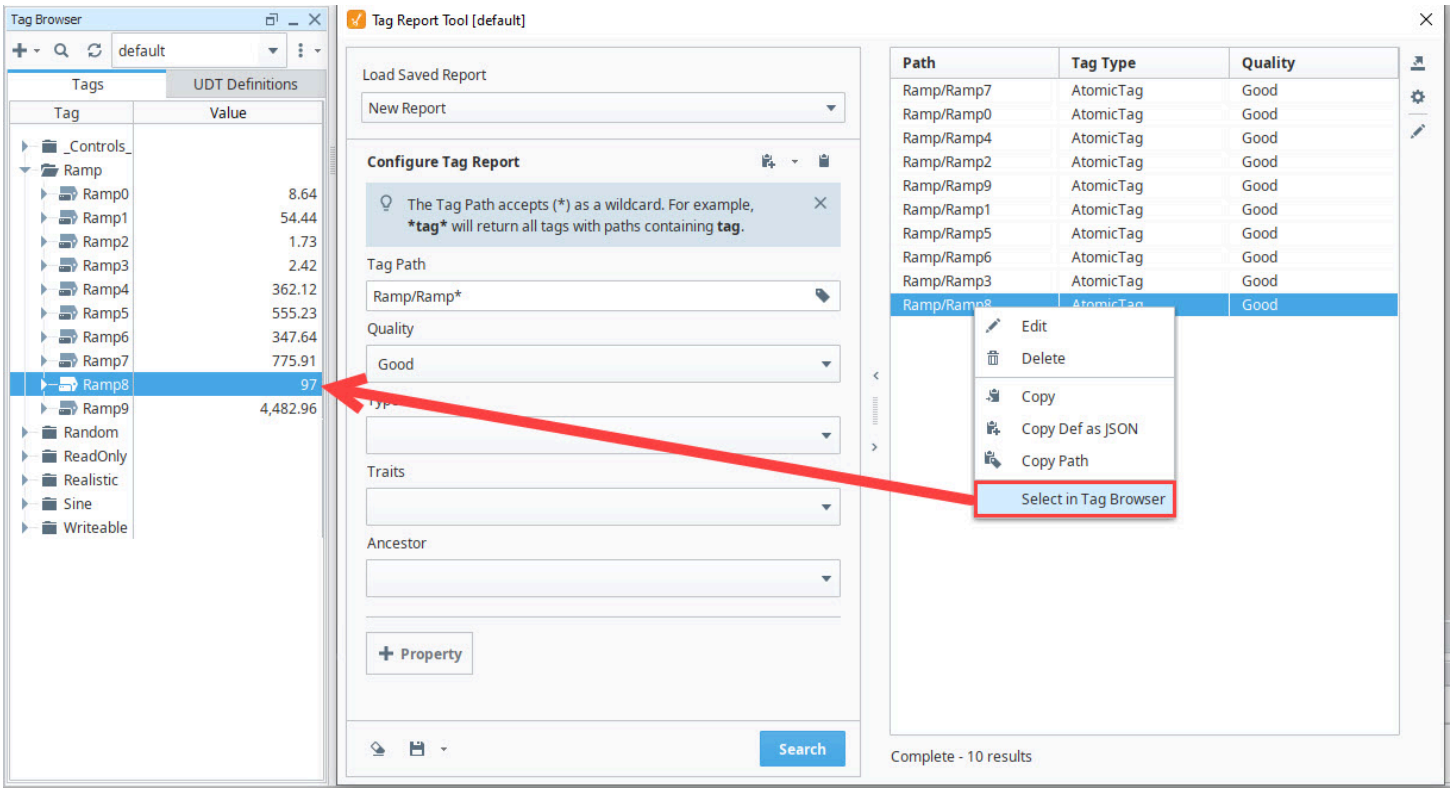
Other columns available for export:

Property	Description
Overrides	Lists all overridden properties on the tag.
Timestamp	A timestamp representing when the stored value was seen by the system. This property can also be used to **filter reports**.

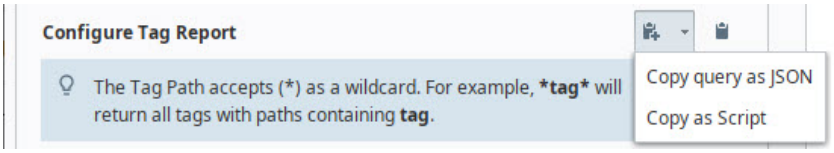
View Tags in Context

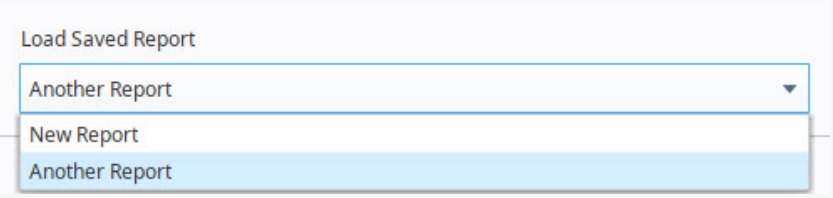
Right-click on any tag in the Selection Pane to view additional options, such as **Edit**, **Delete**, or **Copy Path**. Choose **Select in Tag Browser** to highlight the selected tag in the Tag Browser.

> **CAUTION**
>
> Changes made within the Tag Report Tool will edit tags directly. Deleting a tag in the Tag Report Tool will also delete the tag from the Tag Browser, not just from the generated report.


Tag Browser
default
Tags
UDT Definitions
Tag
Value
Controls
Ramp
Ramp0 8.64
Ramp1 54.44
Ramp2 1.73
Ramp3 2.42
Ramp4 362.12
Ramp5 555.23
Ramp6 347.64
Ramp7 775.91
Ramp8 97
Ramp9 4,482.96
Random
ReadOnly
Realistic
Sine
Writeable
Tag Report Tool [default]
Load Saved Report
New Report
Configure Tag Report
The Tag Path accepts (*) as a wildcard. For example, *tag* will return all tags with paths containing tag.
Tag Path
Ramp/Ramp*
Quality
Good
Traits
Ancestor
Property
Search
Path
Tag Type
Quality
Ramp/Ramp7 AtomicTag Good
Ramp/Ramp0 AtomicTag Good
Ramp/Ramp4 AtomicTag Good
Ramp/Ramp2 AtomicTag Good
Ramp/Ramp9 AtomicTag Good
Ramp/Ramp1 AtomicTag Good
Ramp/Ramp5 AtomicTag Good
Ramp/Ramp6 AtomicTag Good
Ramp/Ramp3 AtomicTag Good
Ramp/Ramp8 Good
Edit
Delete
Copy
Copy Def as JSON
Copy Path
Select in Tag Browser
Complete - 10 results

Interface Elements

Icons	Interface Elements	Description
	Copy Query	Allows the user to copy the configured query: • **Copy query as JSON** - Copies the query as a JSON object that can be pasted into the Tag Report Tool in another Gateway. • **Copy as Script** - Copies the query as a script (**system.tag.query**) 
	Paste JSON Query	Paste a JSON query into the Tag Report Tool.
	Download Tag Report	Exports the Tag Report as a CSV file.
	Configure Report Columns	Add, remove, reorder, or assign aliases to columns displayed in the report.
	Edit tag	Opens the Tag Editor for the selected tag. **Caution**: made within the Tag Report Tool will edit tags directly. Deleting a tag in the Tag Report Tool will also delete the tag from the Tag Browser, not just from the generated report.
	Clear	Clears the configured query.

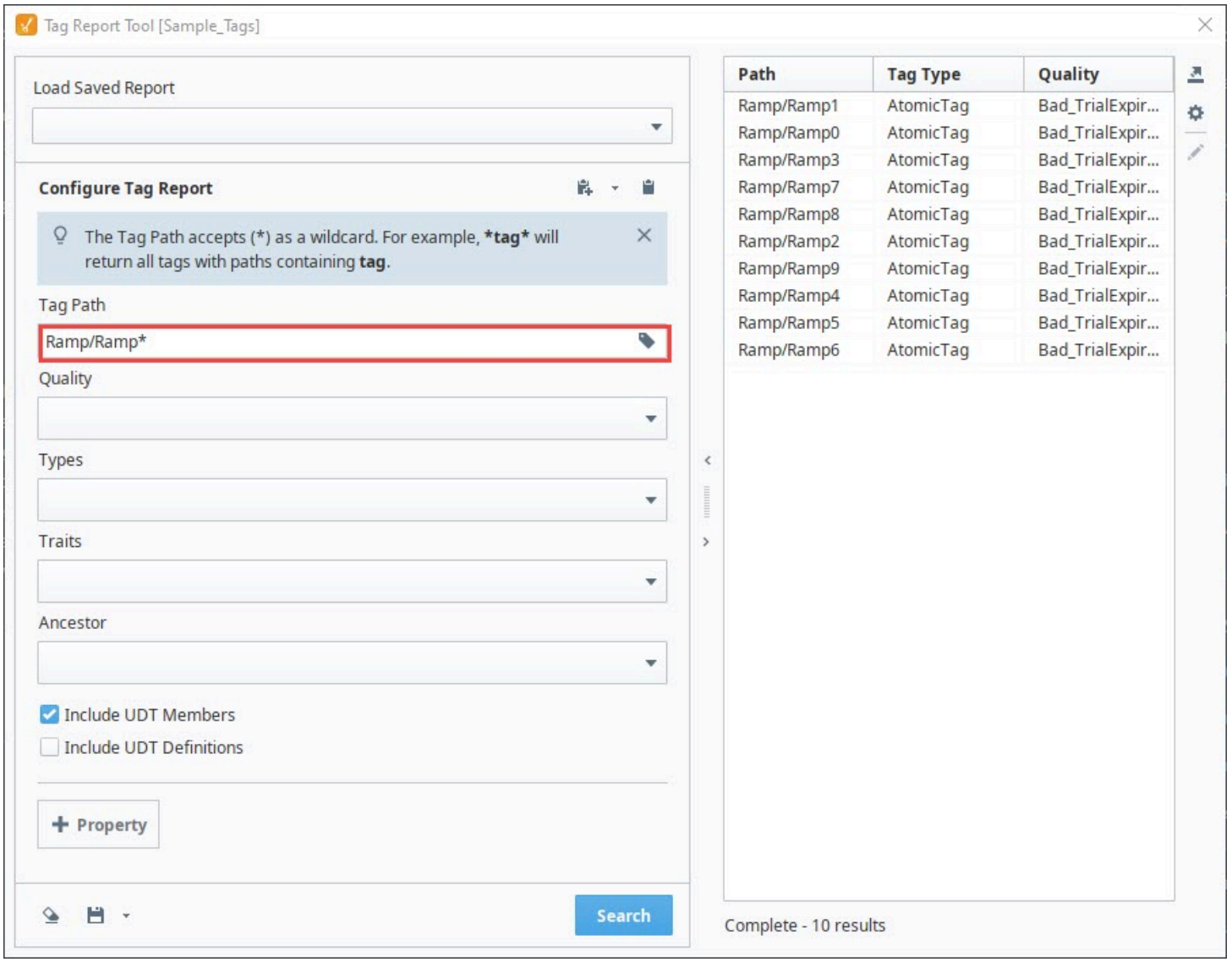
Icons	Interface Elements	Description
	Save	Save options for the configured query: • **Save** - Saves changes to an existing saved report. • **Save As** - Save as a new saved report. • **Manage saved reports** - Opens a new indow
	Load Saved Report	Select and load a previously saved report. Saved reports are accessible to all users and all designers on the same Gateway. 

Configure Tag Report

Configure a search query using the criteria on the left-hand panel and click **Search** to generate a new report.

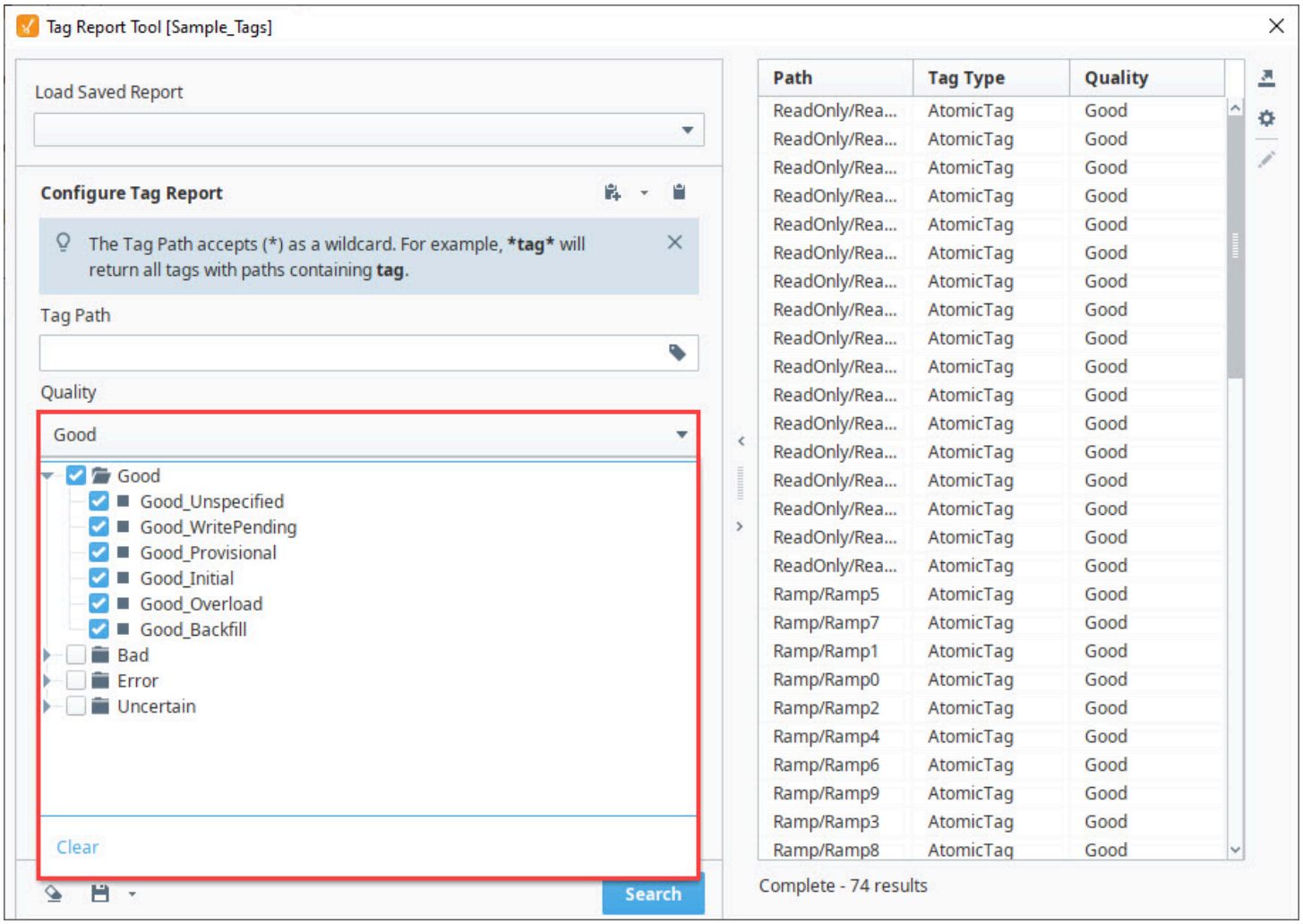
Tag Path

Search for tags by full or partial Tag Path. The Tag Path accepts (*) as a wildcard anywhere in the search string.



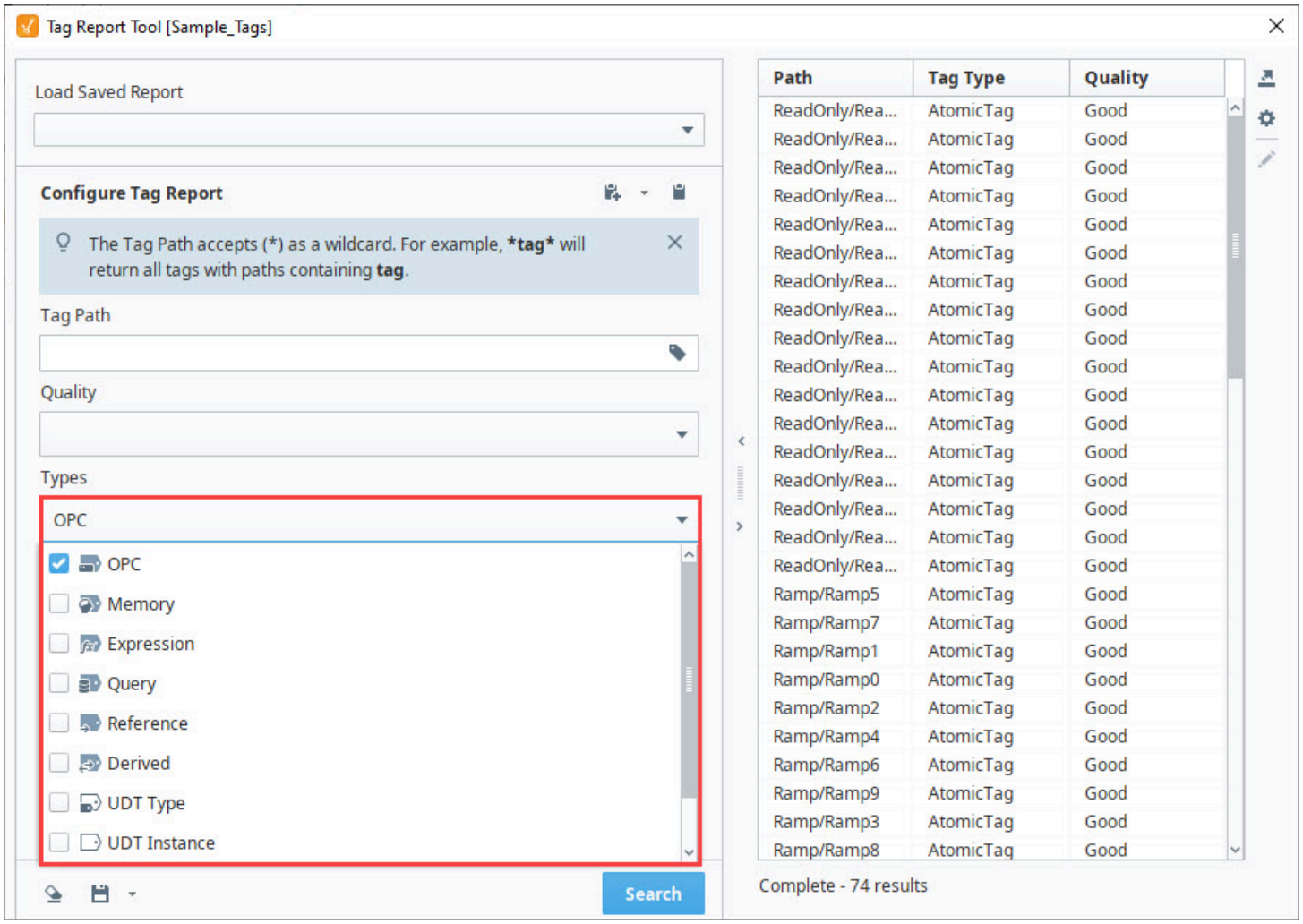
Quality

Search for tags by Quality. You may search by **Good**, **Bad**, **Error**, **Uncertain** or narrow your search down to specific qualities like **Good_Initial** or **Bad_Stale**. For more information on tag qualities, see **Quality Codes and Overlays**.



Types

Search for tags by their value source.

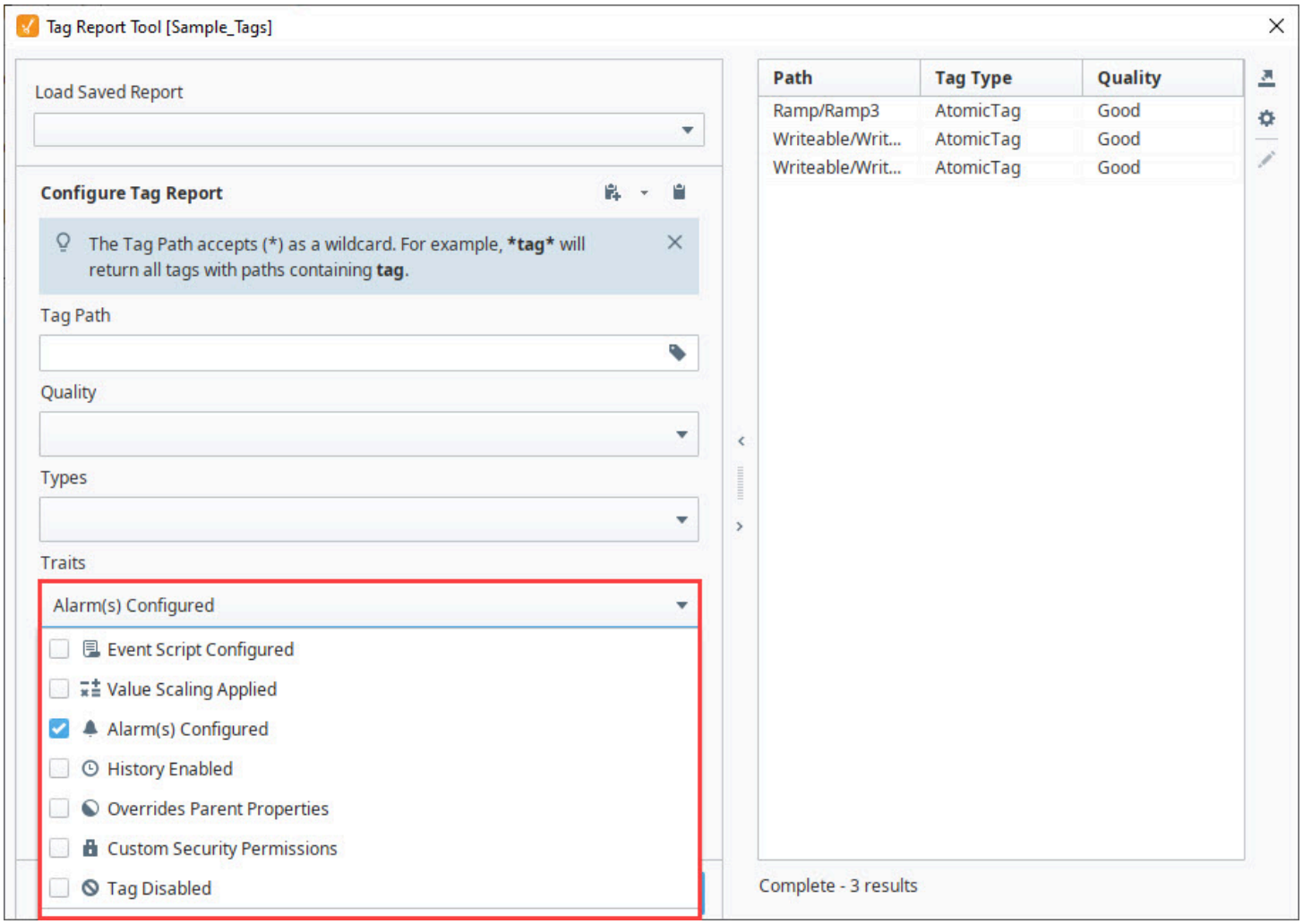


Traits

Search for tags by specific traits. Searchable traits include the following:

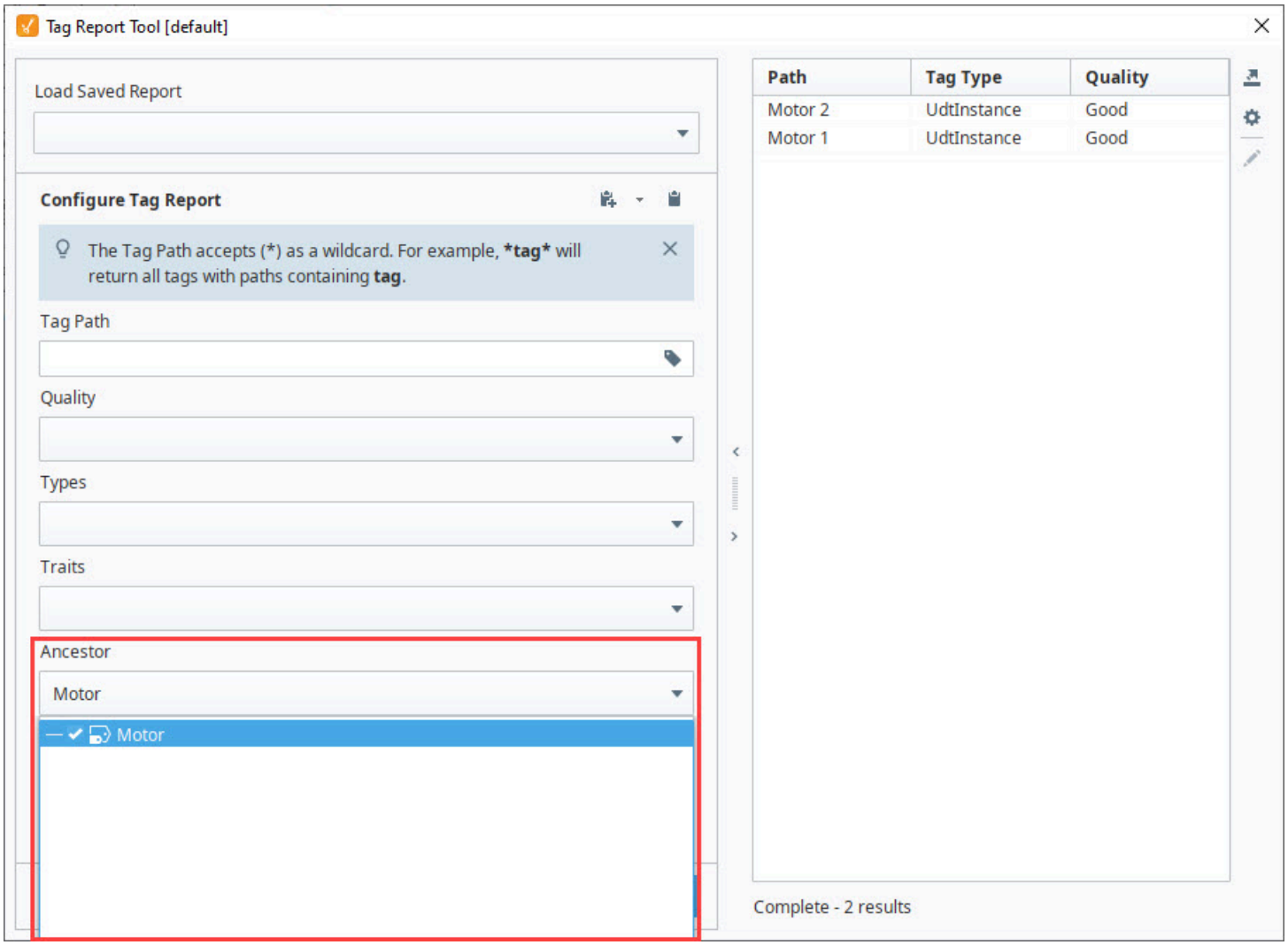
- Event Script Configured
- Value Scaling Applied
- Alarm(s) Configured
- History Enabled
- Overrides Parent Properties
- Custom Security Permissions
- Tag Disabled

> **NOTE**
>
> Trait searches configured in the Tag Report Tool are AND queries. Selecting multiple criteria will return a report including only tags that match all selected criteria.



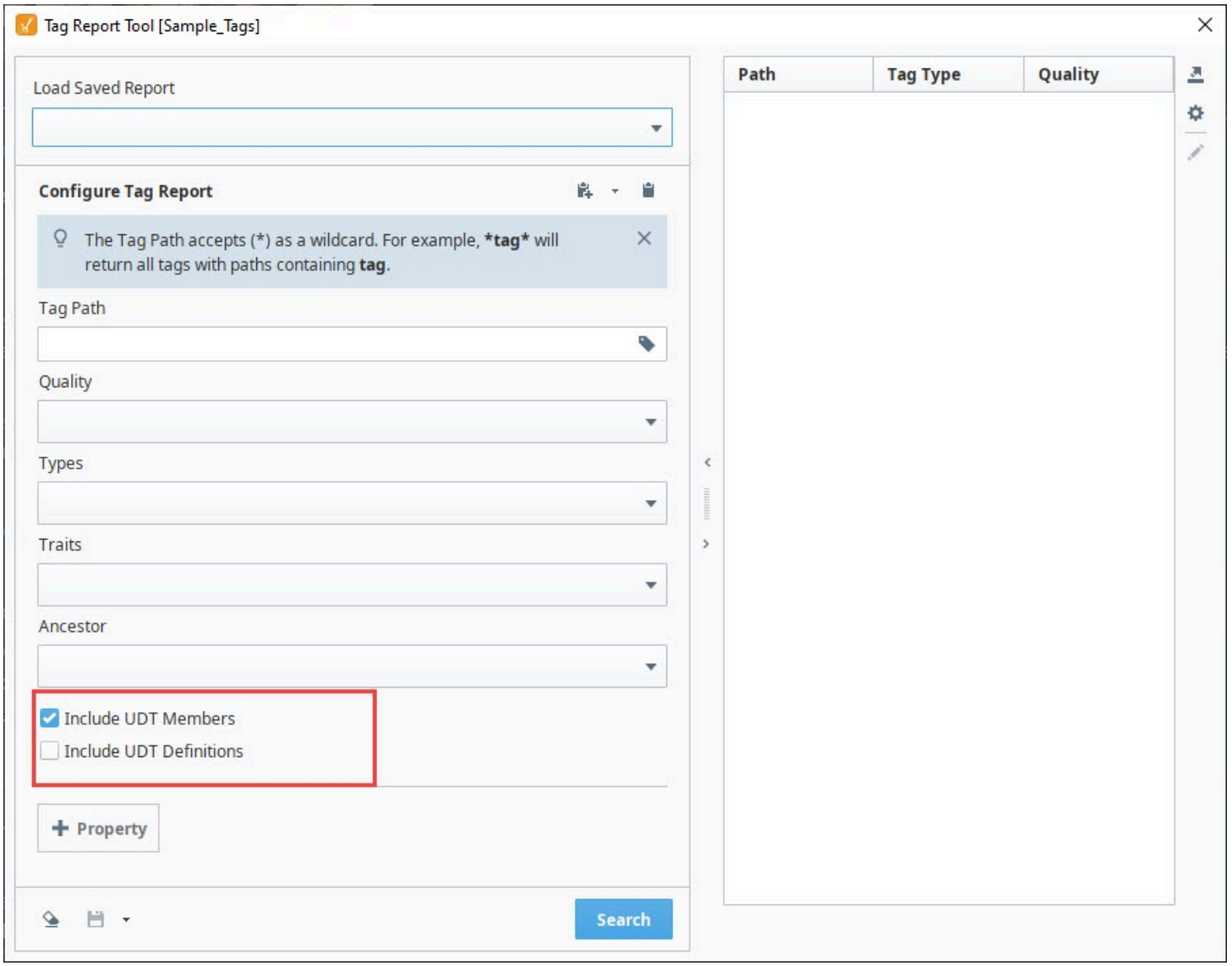
Ancestor

Search for tags by UDT Parent types.



UDT Members and Definitions

The Tag Report Tool provides the option to include UDT Members or UDT Definitions in the query's search results.

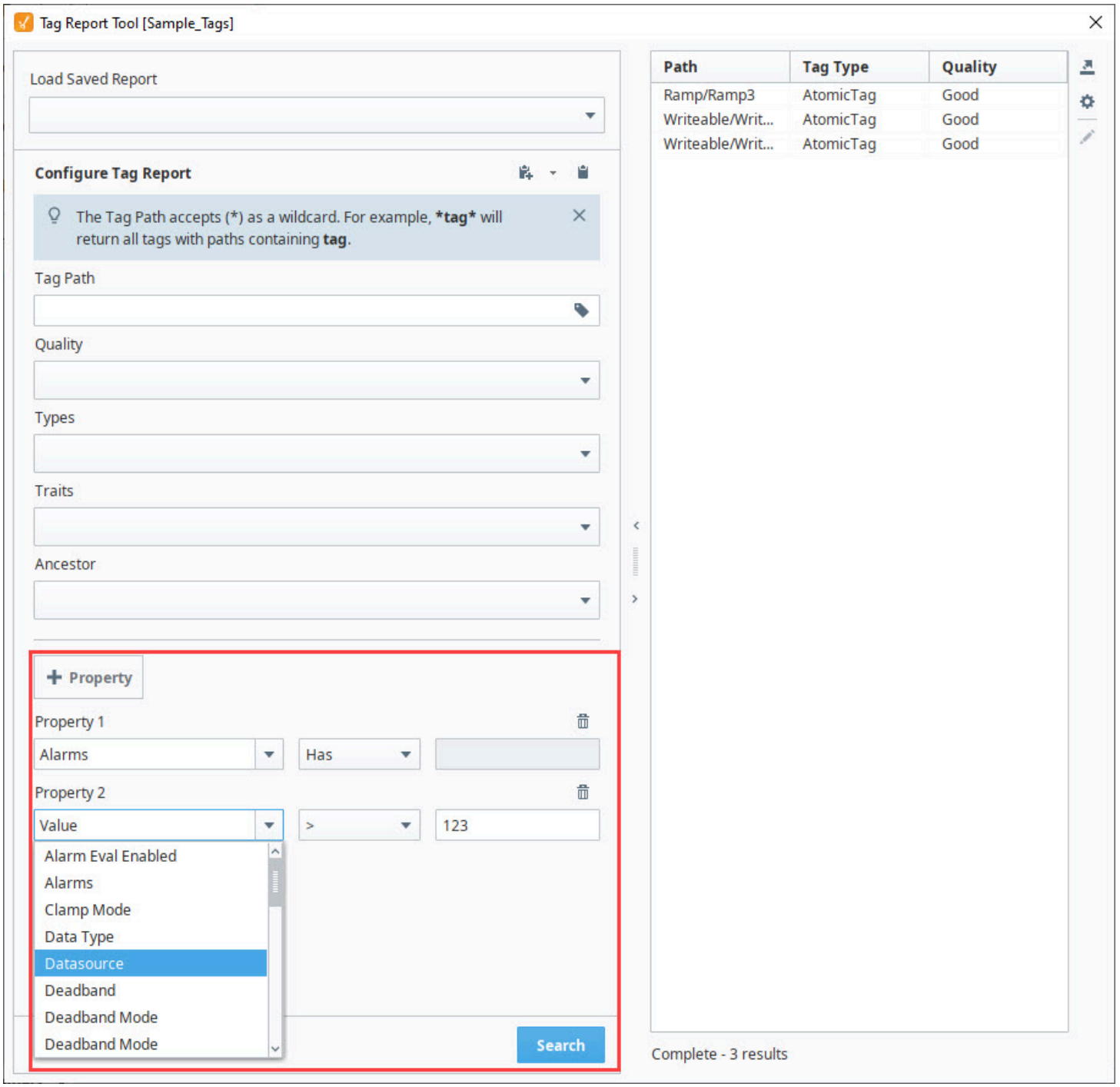


Properties

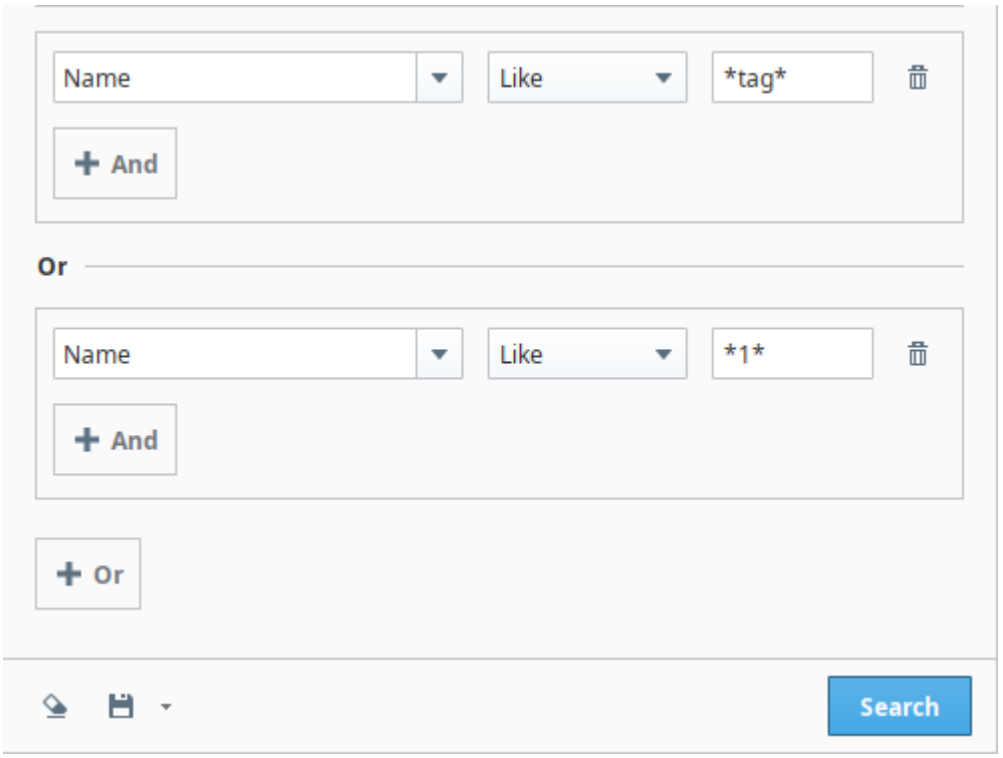
Search for Tags by values on specific properties. All tag properties are available to query on using the following operators:

> **NOTE**
>
> Properties contained within a set or properties that are a set require wildcards around search terms when using **Like** and **Not Like**. These properties are listed below.
>
> - Alarms (Only returns results when searching for alarm names)
> - Parameters
> - Qualified Value
> - Read Permissions
> - Tag Event Scripts
> - Value (If the value data type is a set)
> - Write Permissions

Operator	Description
`Has`	Has been configured to a value other than the default value.
`Has Not`	Has not been configured.
`Overridden`	Value of this property has been overridden from the parent property.
`Like`	Is like. This argument accepts a wildcard (*) in the search string.
`Not Like`	Is not like. This argument accepts a wildcard (*) in the search string.
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`=`	Equals
`!=`	Does not equal



You can search for multiple Tags with different properties using the `Or` operator. Additionally, you can nest multiple `And` operators within an `Or` operator.



> **NOTE**
>
> You will not be able to use the `Or` operator to query a remote Tag Provider that is from an older Gateway, since Gateways before version 8.1.28 will not have `Or` operator functionality.

You can also use the `Or` operator for specific system functions that use JSON queries. The following is a list of system functions that can use `Or`:

- **system.alarm.queryJournal**
- **system.alarm.queryStatus**
- **system.tag.query**

See the system function pages above on examples of how to use the Or operator.

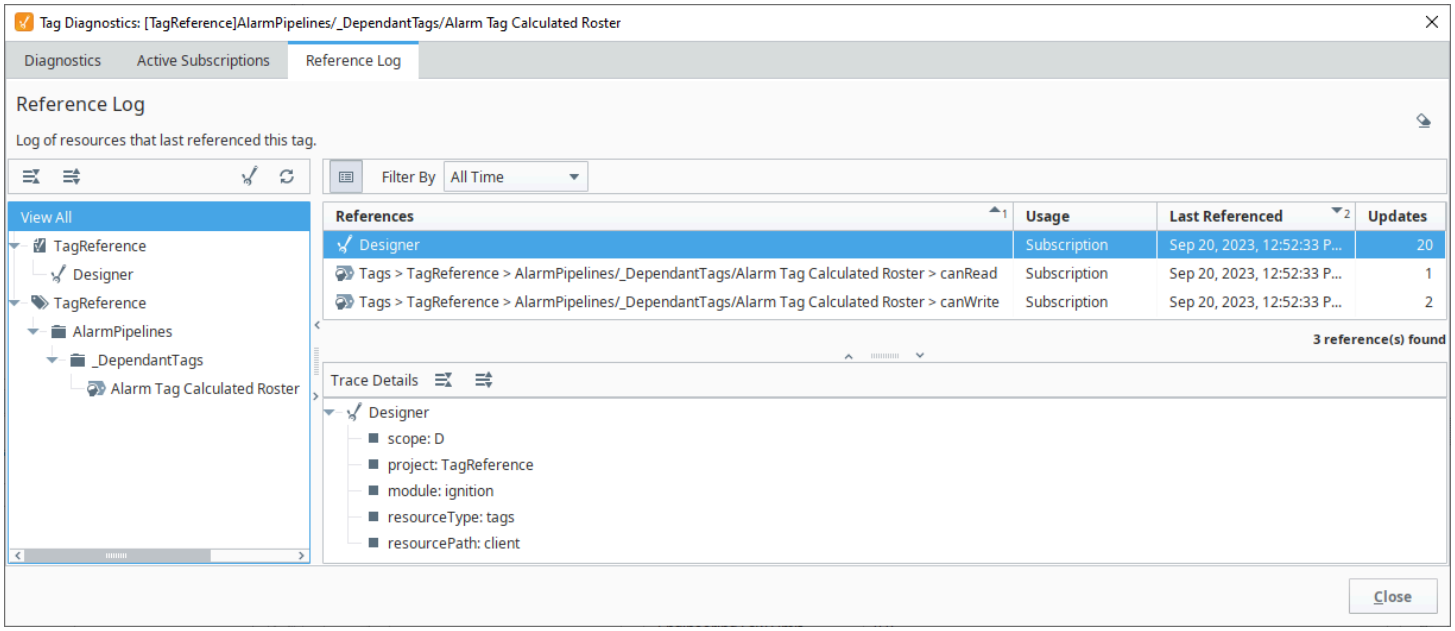
Tag Diagnostics

INDUCTIVE UNIVERSITY

Tag Diagnostics

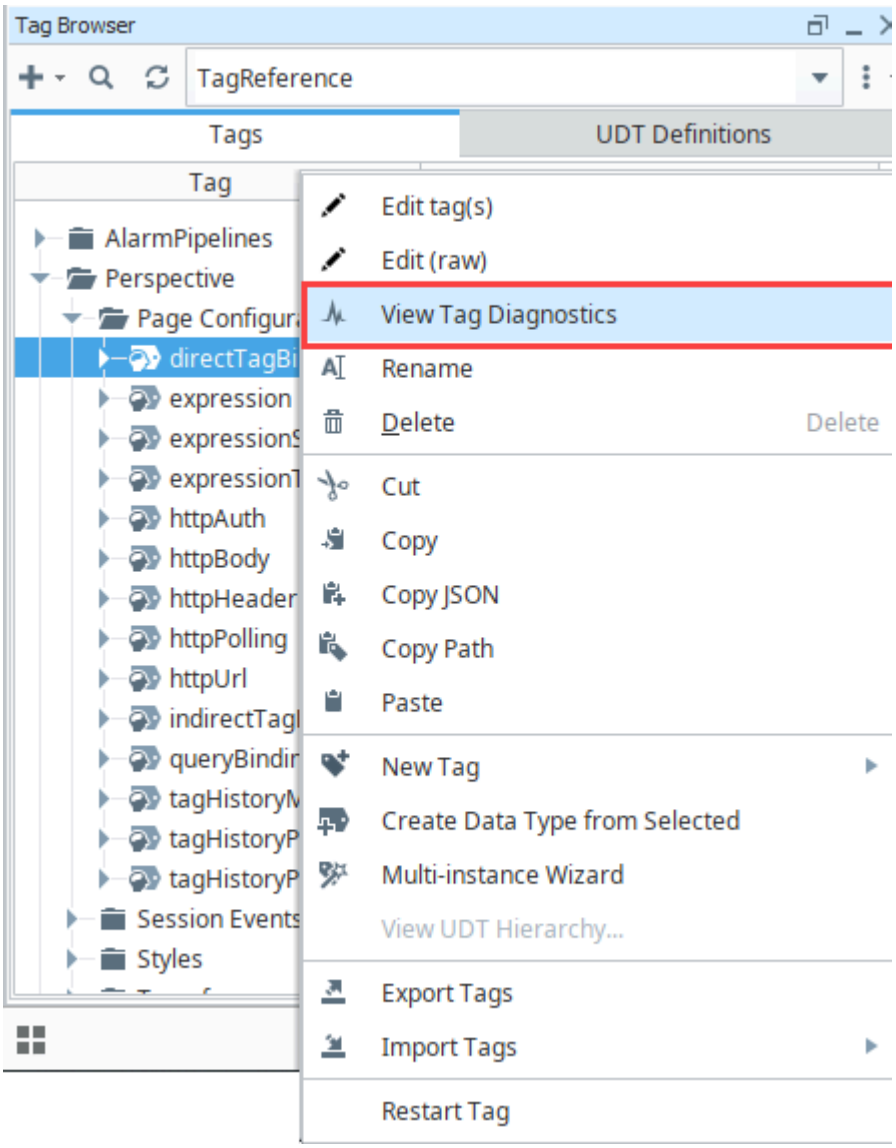
Watch the video

The Tag Diagnostics popup window maintains quick access to relevant tag information and introduce the Tag Reference Tracker. Note that the Tag Editor does not display tag diagnostics information. Pages detailing features of the previous Tag Editor can be found in **Deprecated Ignition Features**.



Accessing the Tag Diagnostics Window

To access Tag Diagnostics for a specified tag, right-click on the tag in the Tag Browser and select **View Tag Diagnostics**. This will open the Tag Diagnostics window, which contains a Diagnostics, Active Subscriptions, and Reference Logs tabs.



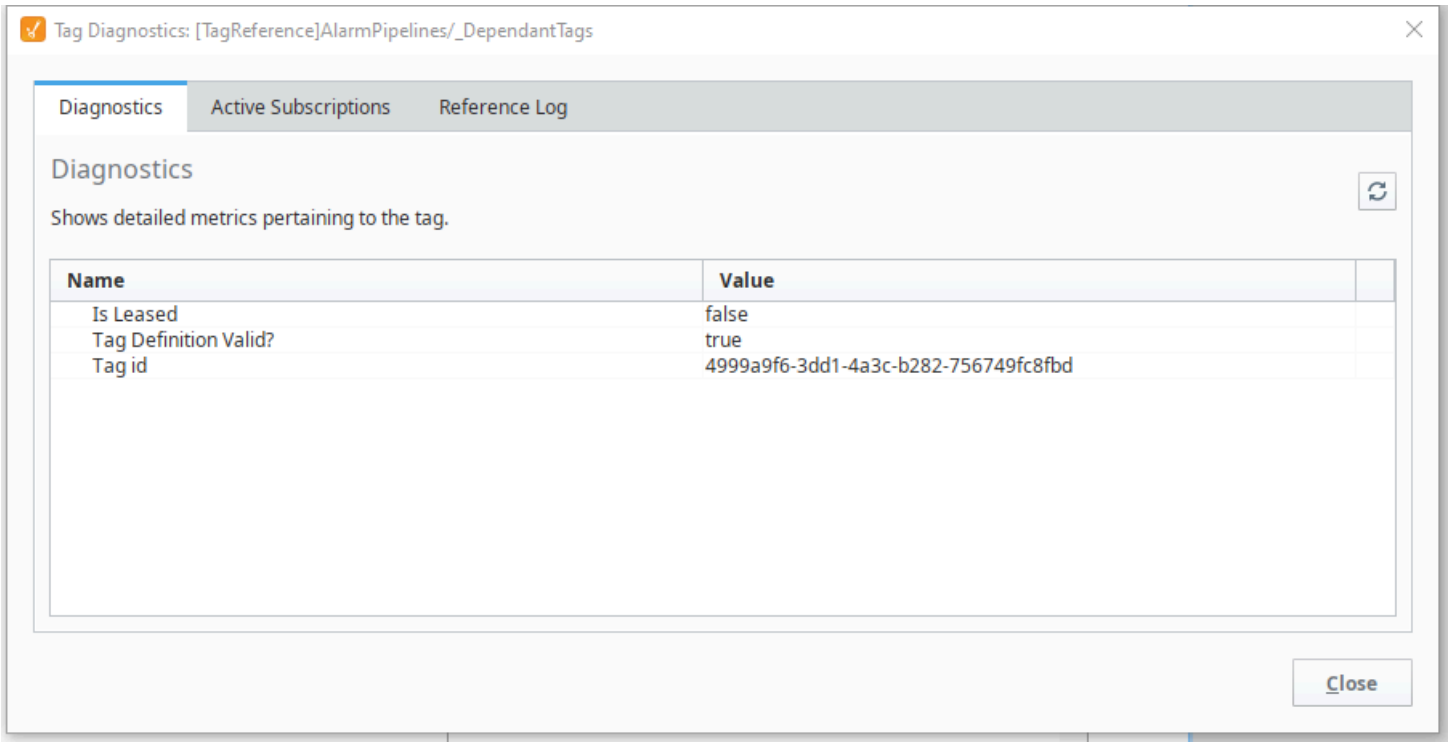
Tag Diagnostic Values

The Diagnostics tab provides detailed metrics about the specified tag including:

- Current value, quality and timestamp information
- Error messages
- Status of dependent data sources

The diagnostic values do not update while the Tag Diagnostics window is open. The displayed information is collected when opened and will remain static until the window is closed and opened again.

You can use the Refresh icon in the Diagnostics tab to update the displayed information while the window is still opened.


Tag Diagnostics: [TagReference]AlarmPipelines/_DependantTags
Diagnostics
Active Subscriptions
Reference Log
Diagnostics
Shows detailed metrics pertaining to the tag.
Name
Value
Is Leased
false
Tag Definition Valid?
true
Tag id
4999a9f6-3dd1-4a3c-b282-756749fc8fbd
Close

Name	Description
Alarm*	A subcategory for details pertaining to alarms configured on the tag. Each alarm on a tag will receive its own subcategory, containing the following items: • **Last State** - Displays the last state of the alarm. • **Live Event Count** - The number of active or unacknowledged events for the alarm. The maximum count is determined by the **Gateway's Live Event Limit setting**.
Deadband	A subcategory for details pertaining to the deadband settings on the tag. These metrics are only available when the Numeric category's Deadband is set to a value larger than 0. Contains the following: • **Last value** - The last value on that tag, as recognized by the Numeric deadband. • **Limit** - The Numeric deadband value on the tag.

History	A subcategory for details pertaining to the history settings on the tag.
	• **Deadband (Analog)** - A subcategory of diagnostic metrics that are only available when the History category's **Deadband Style** is set to **Analog**. For more information, see the **Analog Deadband Values** section.
	• **Deadband (Discrete)** - A subcategory of diagnostic metrics that are only available when the History category's **Deadband Style** is set to **Discrete**.
	  ◦ **Last Value** - The last value on that tag, as recognized by the History category's **Deadband** property.
	  ◦ **Limit** - The **Historical Deadband** value on the tag.
	  ◦ **Mode** - The Historic category's **Deadband Style** used by the tag. Useful in cases where a style of "Auto" is used, as this diagnostic value will display which option was selected by the system.
	• **Min/Max Age**
	  ◦ **Max Age Limit (ms)** - A milliseconds representation of the Max Time Between Samples and Max Time Units properties.
	  ◦ **Minimum Age Limit (ms)** - A milliseconds representation of the Min Time Between Samples and Min Time Units properties.
	  ◦ **Pending Value** - Represents one or more pending values that can't yet be written to the tag due to the Minimum Age Limit.

Name	Description
	◦ **System Time of Last Storage** - A date representing the the last time a historical record value was stored for this tag. ◦ **Timestamp of Last Stored Value** - A date representing the timestamp associated with the last collected historical record. • **Stored Value Count (since startup)** - The total number of historical records collected for the tag since history on the tag was enabled.
OPC	A subcategory for details pertaining to OPC details on the tag. These diagnostic values are only available when **(Value Source)** is set to **OPC**. • **Last Subscription Value** - A qualified value representing the last subscribed value. The value on this setting ignores the deadband. • **Server Name** - The name of the OPC server the tag is subscribing to. • **Subscription Name** - The name of the subscription, used by the OPC server. • **Valid Tag Group? -** States whether or not the tag group the tag is configured to use is valid.
Alarm Acknowledged	Displays any errors from the **Alarm Acknowledged** tag event script. Only visible if a script is defined on the corresponding tag event.
Alarm Active	Displays any errors from the **Alarm Active** tag event script. Only visible if a script is defined on the corresponding tag event.
Alarm Cleared	Displays any errors from the **Alarm Cleared** tag event script. Only visible if a script is defined on the corresponding tag event.
Datasource Status	States which data source is configured for the tag. Only available when **Value Source** is set to **Query**.
Is Leased	If true, indicates that the tag is using a leased tag group, and is currently executing at the leased rate.
Is Scaled	If true, indicates that **Scale Mode** has been enabled.
Last published value	Represents the last value on the tag when the diagnostics tab was last updated.

Name	Description
Range	When **Deadband Mode** is set to **Percent**, indicates the range of values between the **Engineering Low Limit** and **Engineering High Limit**.
Quality Changed	Displays any errors from the **Quality Changed** tag event script. Only visible if a script is defined on the corresponding **tag event**.
Quality Diagnostics	Reports any quality errors about the tag. This is typically only present when there is a configuration issue with the tag, such as when **Value Source** is set to **Query** and a **Datasource Status** value has not been set.
Tag Definition Valid?	A flag that indicates whether or not a tag configuration is valid.
Tag id	A unique identifier for the tag. Used by the tag system to distinguish one tag from another.
Value Changed	Displays any errors from the **Value Changed** tag event script. Only visible if a script is defined on the corresponding **tag event**.

Analog Deadband Values

The following metrics appear under the **Deadband (Analog)** category.

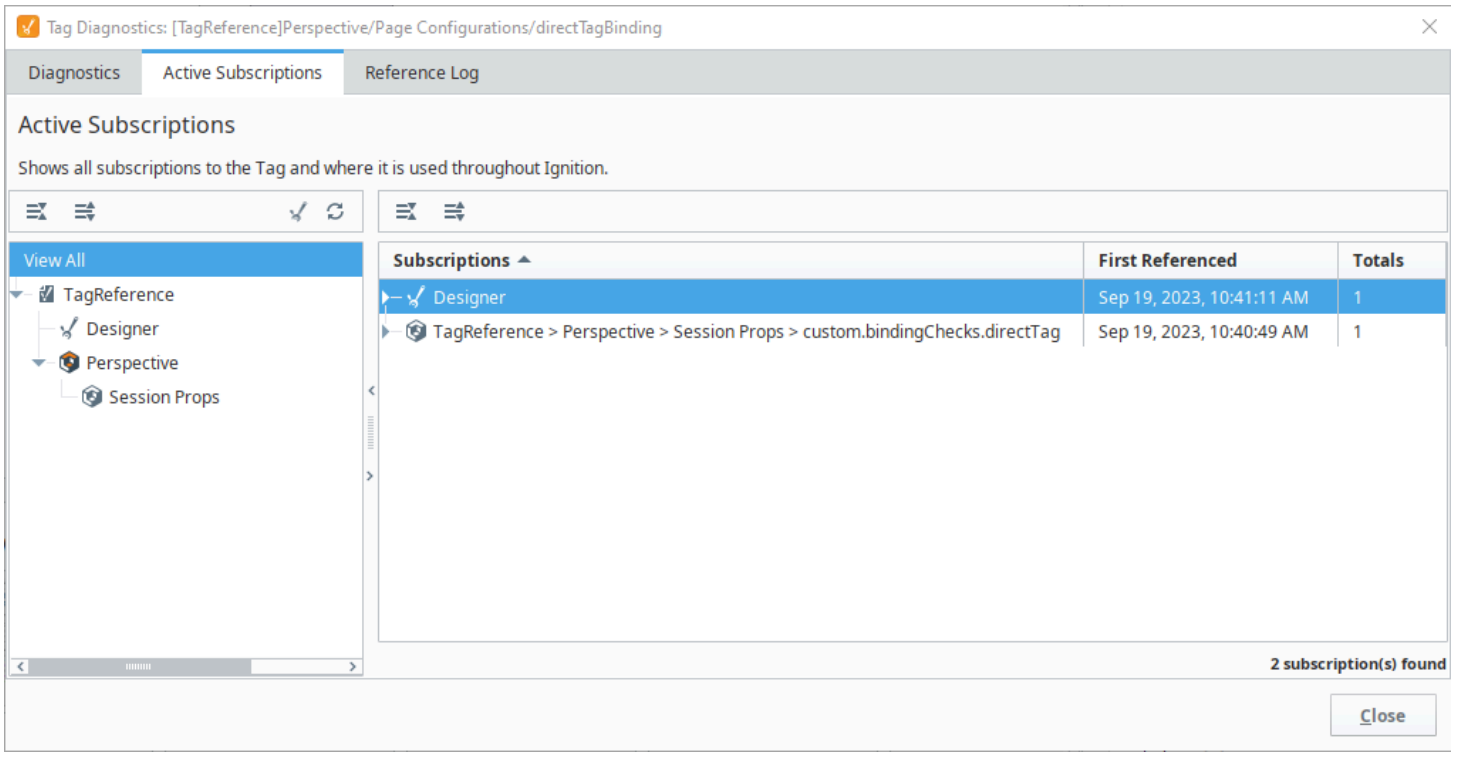
Name	Description
Current Lower Slope and Current Upper Slope	These properties represent the slope for each window. These values are calculated every time the tag evaluates, based upon the tag's sample mode. The resulting numbers are then compared to these current slopes. If the new upper slope is less than the Current Lower Slope, or the new lower slope exceeds the Current Upper Slope, then the new value is stored. In either case, the new slope values replace the value on these Current Slope properties. See **Deadband and Analog Compression** for more information.
Last Stored Time	Represents the previously stored timestamp, after the most recent timestamp. Works in a similar manner to **Last Stored Value** except this stores the time of the storage, instead of the value.
Last Stored Value	Represents the value stored before the most recently stored value. For example, say a historian system stores the following for a single tag, in order: • 1000 • 1100 • 1200 The most recently stored value would be 1200. The value of this **Last Stored Value** metric would be 1100, since that was the previously stored value.
Last value	Represents the last **QualifiedValue** on the tag, regardless of whether or not that value was stored. In other words, the value most recently written to the tag.
Mode	The Historic category's **Deadband Style** used by the tag. Useful in cases where a style of "Auto" is used, as this diagnostic metric will display which option was selected by the system.
Tolerance	The tolerance used to calculate each new slope. The value of this metric is determined by the History category's **Deadband** setting on the tag.

Active Subscriptions

The Active Subscriptions tab displays real-time tracking of a tag subscription by sharing the following information:

- **Subscriptions**: Displays the path to a resource that is currently subscribed to the selected tag.
- **First Referenced**: Displays the first logged time a tag subscription at the given location was created. The logged time is reflective of the first subscription for the current grouping of a specific subscription.
- **Totals**: Displays the number of currently active subscriptions from the referenced location. For example, if there are multiple Perspective Sessions with the same View bound to the selected tag loaded, this number would increase.

Additionally, the Reference Browser on the left-hand side lists all locations where the tag has an active subscription. Use the Collapse All and Expand All icons to adjust how much information is displayed. You can also update the active subscription information by clicking the Refresh icon. The Designer icon is available to show/hide references to the specified tag from the Designer, including in the tag browser and Vision components when in Preview mode.



Tag Reference Tracker

IU INDUCTIVE UNIVERSITY

Tag Diagnostics - Tag Reference Tracker

Watch the video

The Reference Log tab shows the data stored in the Tag Reference Tracker Store as a way to find where tags are being used in a project. The Tag Reference Tracker Store provides visibility to the areas that rely on any given tag as it displays what resources have subscribed, read, written to or configured specified tags.

> **CAUTION**
>
> The Reference Log can only show references that are in use or have been in use since the reference tracking began. It will miss potential references that haven't been active since tracking started. The longer is it in use, the more reliable the tracker becomes as the history will grow and continue to log with each new session that is opened.

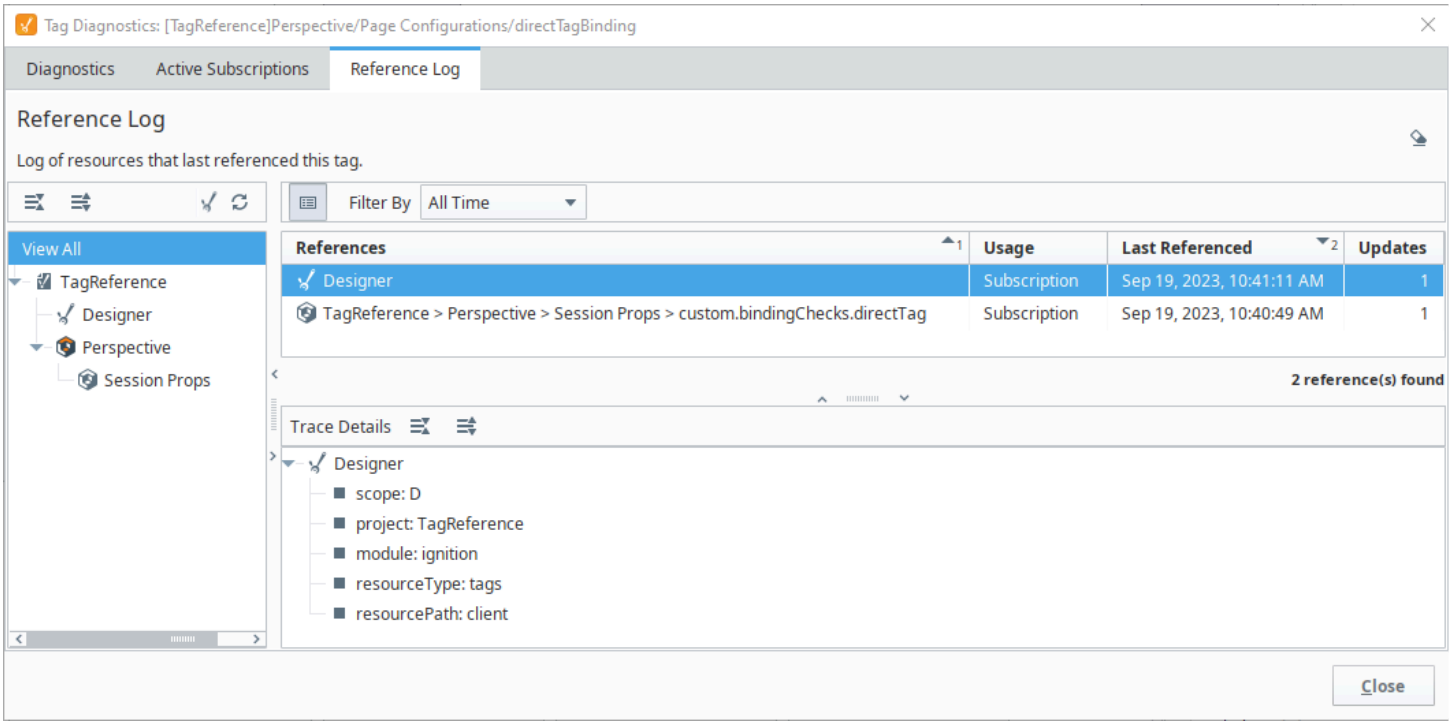
The bulk of the Tag Reference Tracker information includes:

- **References**: The tag path to display where the tag was used.
- **Usage**: The type of reference to this tag. The usage type listed will match the tracking type recorded in the Ignition database file. The four tracking types for tag usage include reads, writes, subscriptions, and configuration changes.
  - **Configuration**: Resource made a configuration change to the tag definition.
  - **Subscription**: Resource subscribed to the tag to display information on a particular property.
  - **Read**: A read event occurred to a property on the tag.
  - **Write**: A write event occurred to a property on the tag.
- **Last Referenced**: The timestamp for when the reference was last updated.
- **Updates**: The number of times a reference with a specific usage type occurred.

> **NOTE**
>
> Double-clicking on a reference line will open the tag reference root location or the Tag Editor depending on the reference type. If it is in a Tag Value Change Script or an Expression used within the Tag definition, it will open the Tag Editor.

Like the Active Subscriptions tab, the Reference Browser on the left-hand side will show where the tag is used in any project. You can use the reference tree or Collapse All and Expand All icons to locate where in the Designer a tag is used for each project, the Designer icon to show/hide Designer references, and the Refresh icon to refresh the browser list. Refer to the Trace Details

section at the bottom for more information that might be helpful in understanding tag usage. Raw tracking logs can also be found by right-clicking on the Reference Log tab and selecting **Show Raw**. The Raw Tag Logs Browser will now be visible on the Tag Diagnostics window. Clicking the Export icon to download the raw logs.



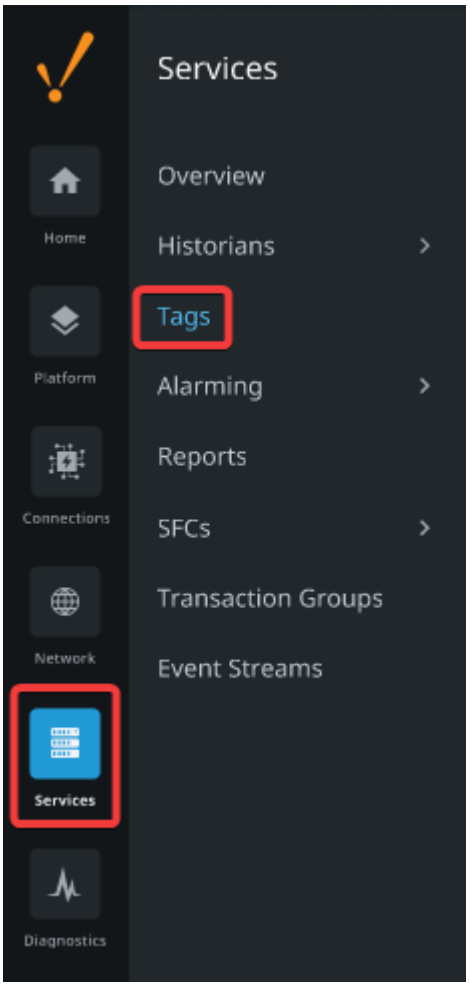
Enabling/Disabling the Tag Reference Store

The Tag Reference Store is enabled by default on new installations. There are two options to enable or disable the Tag Reference Store depending on the provider type.

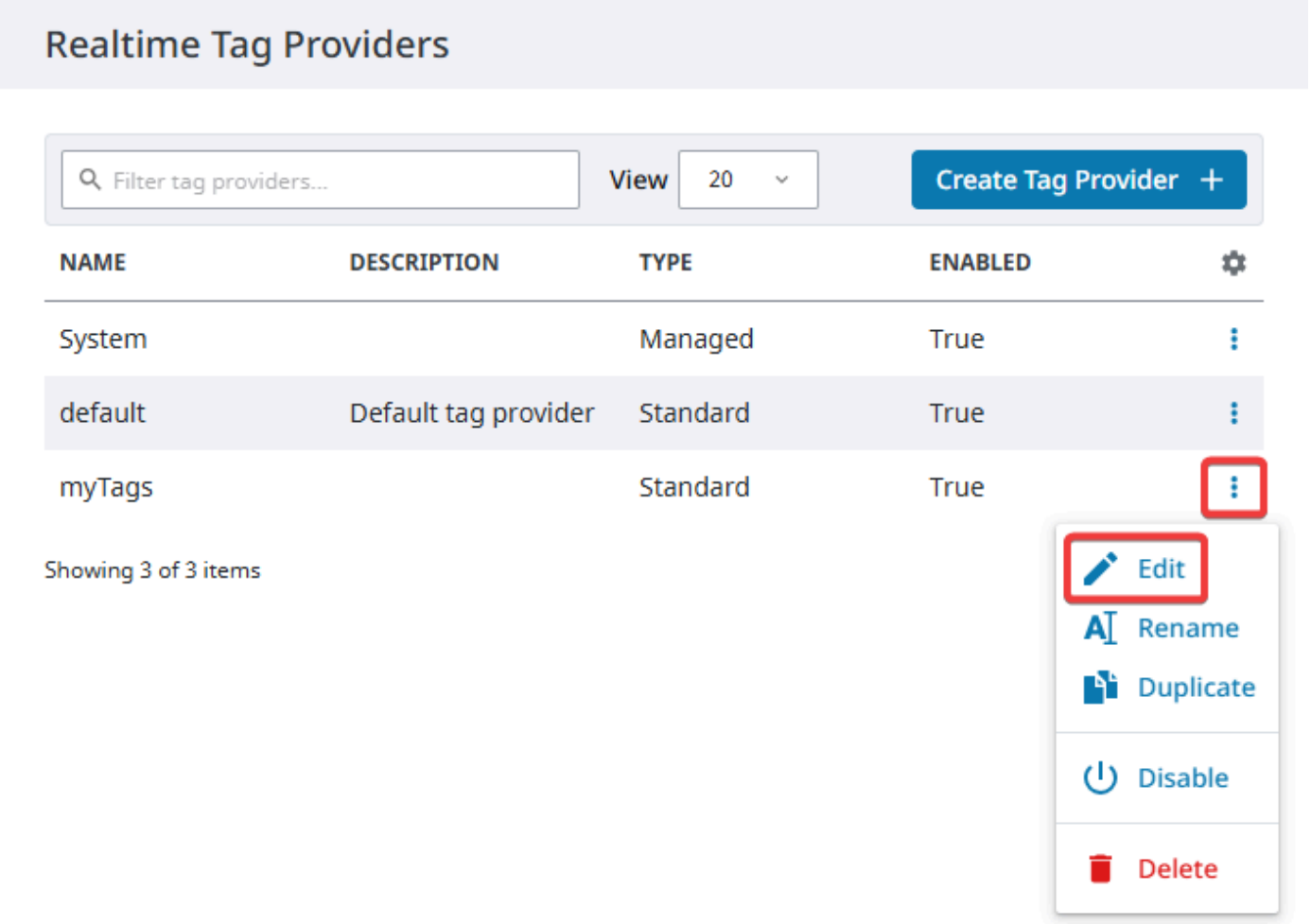
Standard and Remote Tag Providers

For both standard and remote Tag Providers, follow the steps below.

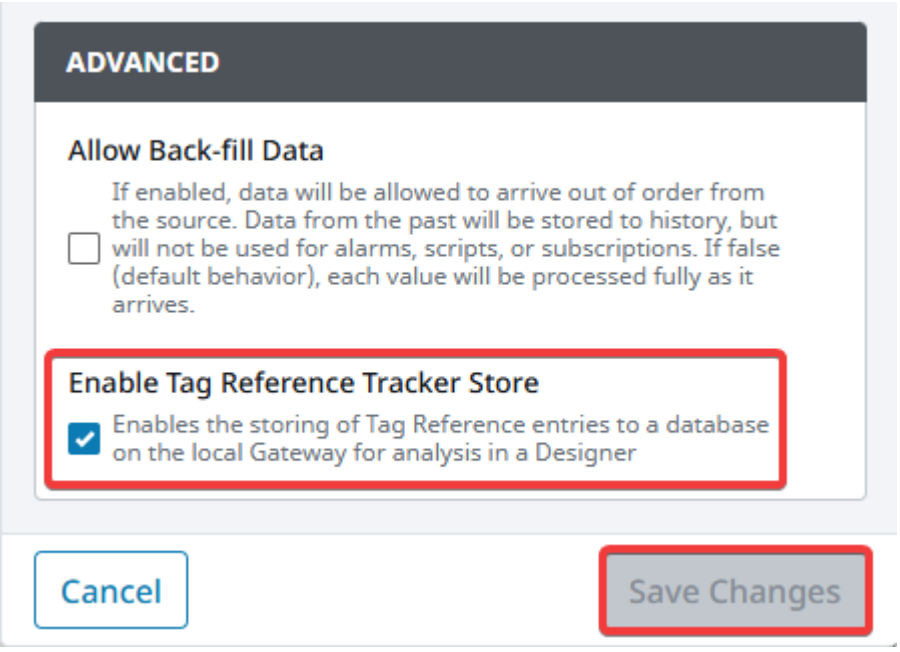
1. Navigate to the **Services** > **Tags** section on the Gateway to access the Realtime Tag Providers page.



2. Click on the three dots menu for the provider you want to enable/disable the Tag Reference Tracker Store setting for, then click **Edit**.



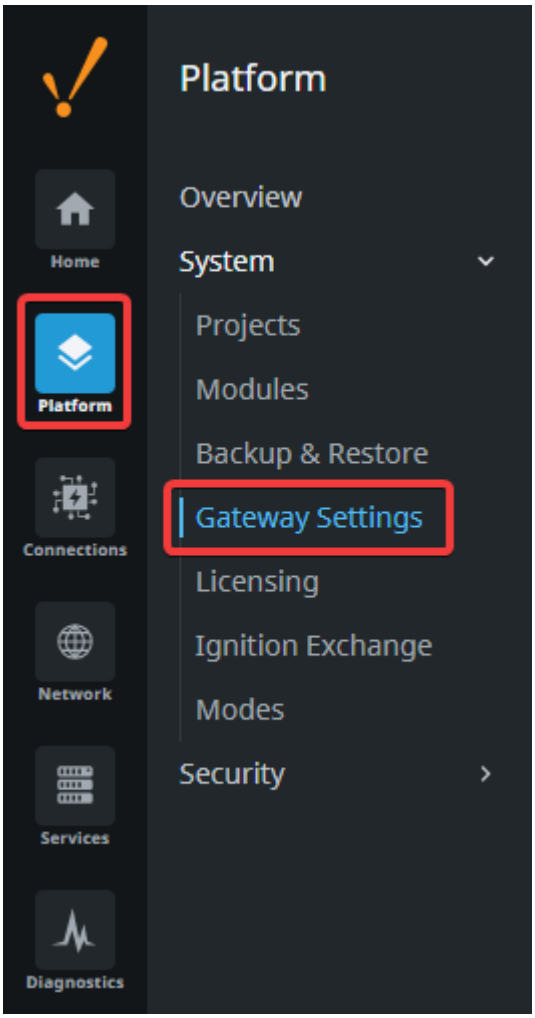
3. Scroll down the settings to the bottom to locate the **Enable Tag Reference Tracker Store** setting. If this property is checked, it is enabled. If you need to change the setting, toggle the check box, then click **Save Changes**.



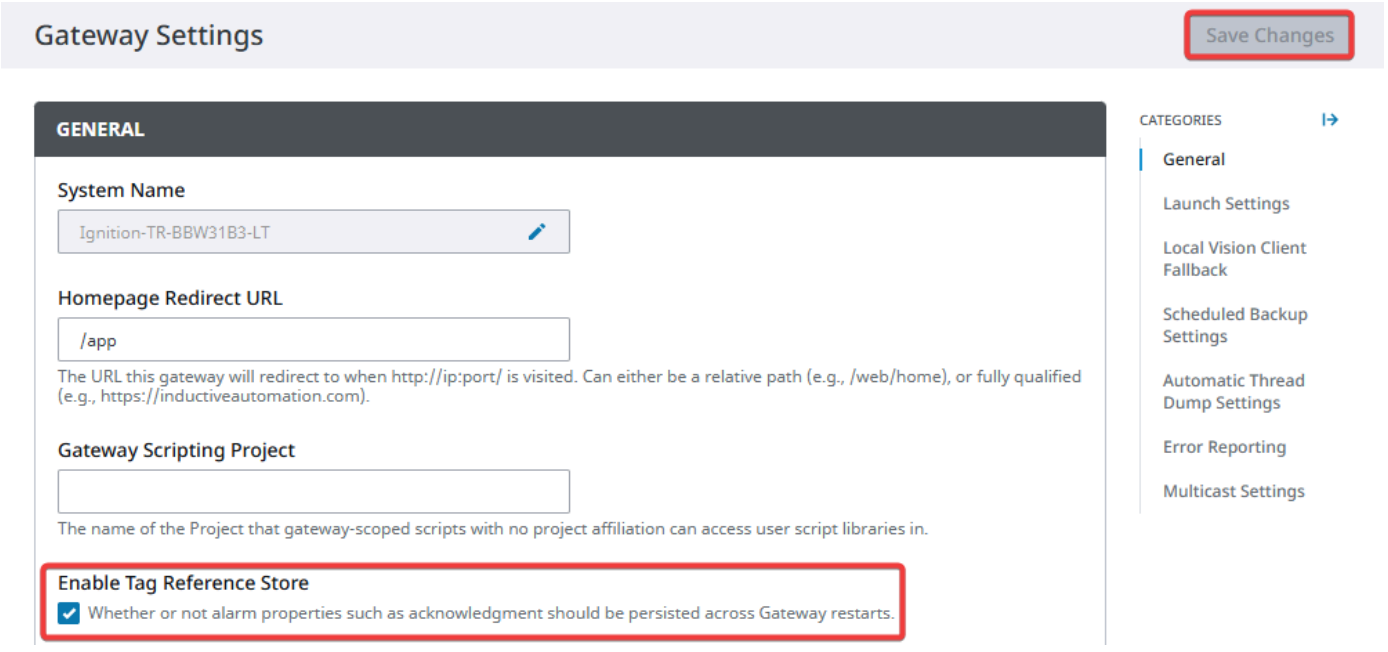
Managed Providers

For managed providers, like third party modules and the system provider, follow the steps below.

1. Navigate to **Platform** > **System** > **Gateway Settings** on the Gateway to access the **System Settings** page.



2. Find the **Enable Tag Reference Tracker Store** property. If it is checked, the property is enabled. If you need to change the setting, toggle the check box, then click **Save Changes**.



3. Be prepared that this setting impacts all managed providers throughout the Gateway, so a large number of third party modules can result in large data collections.

> **NOTE**
>
> Disabling the Tag Reference Tracker will not impact the Diagnostics or Active Subscriptions tabs, but the Reference Log will no longer track tag information.

Collecting Data Backups

Gateway backups do not include Tag Reference Store data. This information is saved instead in a database that creates a new Diagnostics folder within the Data directory of an Ignition installation. Make sure to collect and add the reference data from this folder to any backups so no information is lost.

The default folder location for each operating system are listed below:

Windows

`C:\Program Files\Inductive Automation\Ignition\data\diagnostics\tags`

MacOS

`/usr/local/ignition/data/diagnostics/tags`

Linux

`/usr/local/bin/ignition/data/diagnostics/tags`

Alarming

Alarming in Ignition is flexible and highly customizable to your needs. You can configure alarms with one alarm on a tag or multiple alarms on a tag using the **Alarm settings** in the Tag Editor. Alarms have multiple properties with values that can be assigned dynamically and they support filtering, shelving, notes, and separate pipelines based on **alarm event** states. Alarms can also be added to UDTs, so that every instance of a tag will automatically have alarms configured when created.

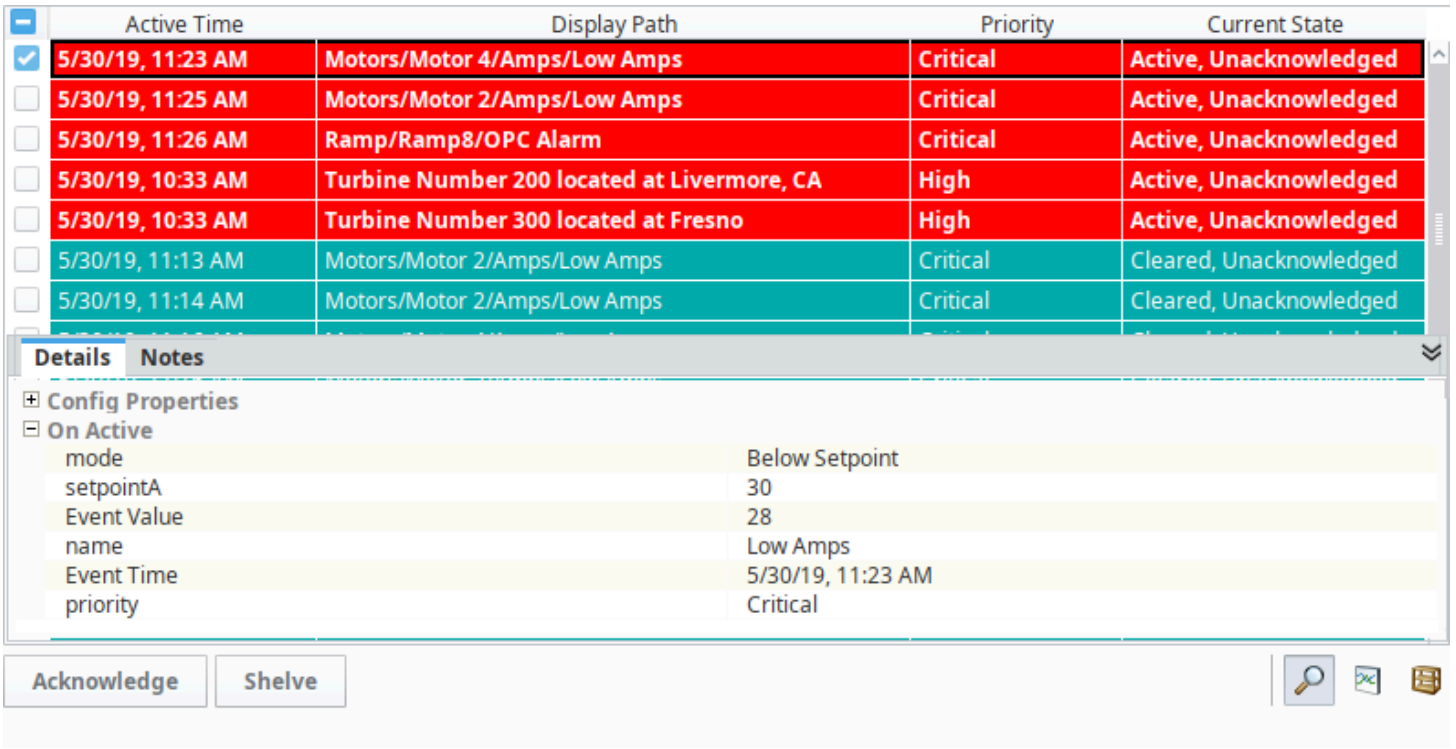
Once alarms are configured, components and module functionality can be utilized to display alarm states, track realtime alarm statuses and alarm history, and create Notification Pipelines with user roles, schedules, and On-Call Rosters to automatically inform necessary personnel of alarm statuses.

This page describes the possible alarm states and an overview of how these states can be shared and displayed. More information about configuring alarms can be found on the **Configuring Alarms** page.

Alarm Events

Alarm events are determined by how the source tag's value compares to configured alarm criteria, which is defined in the **Alarm Mode settings**. Each alarm event contains several attributes that store metadata about the source tag at the time the alarm event was created. Since tags can have multiple alarms configured on them, multiple alarm events can be created from a single value change on a tag.

Although alarms are monitored by the Gateway and considered Gateway-scoped, realtime alarm values can be monitored from **Vision** and **Perspective** using their respective Alarm Status Table components. In addition, the **system.alarm.queryStatus** function can be used to retrieve realtime alarm statuses.



Conditions

The state of an alarm event is determined by the Active or Acknowledged condition.

Active

Determines if the alarm is active, or "live". All alarm events start with an Active condition, and then move to a Cleared condition. Alarm events never transition from Cleared to Active. Instead, a new alarm event would be created should the value on a tag meet the alarm criteria.

- **Active**: The value on the source tag meets the setpoint configured on the alarm. If the alarm is used to signal that there's a problem, then the active condition means the problem is currently occurring.
- **Cleared**: The alarm event is no longer active. The value on the source tag no longer meets the setpoint configured on the alarm.

Acknowledged

Alarms feature an Acknowledgement functionality, which is a flag that can be marked on an alarm event. The Acknowledged condition is commonly used as a way for users to "claim" an alarm when multiple operators are monitoring alarm events, signaling to the others that the event is being handled.

Alarm event acknowledgement is generally performed using the Vision or Perspective Alarm Status Table components, or the **system.alarm.acknowledge** function.

- **Unacknowledged (Unacked)**: The alarm event has not yet been flagged as Acknowledged.

- **Acknowledged (Acked)**: The alarm event has been flagged as Acknowledged.

States

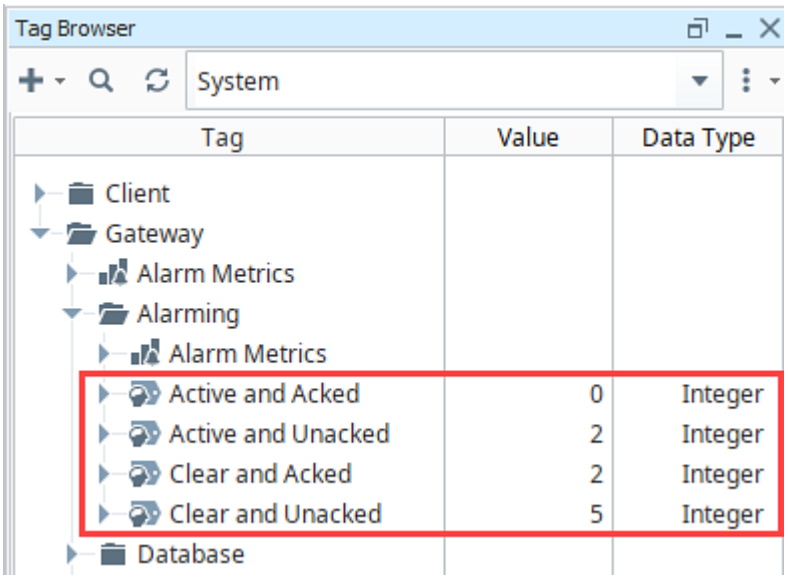
The two conditions above are combined to form one of four possible states of an event:

- Active and Unacknowledged
- Active and Acknowledged
- Cleared and Unacknowledged
- Cleared and Acknowledged

Alarming State Tags

Ignition provides a set of **System tags** to view information about the Ignition server, including four tags that count the number of alarms in each state. A quick way to see if any alarms are currently active and get an alarm count is to create Label components bound to these tags.

To view these tags, open the **System** Tag Provider in the Tag Browser of the Designer, and navigate to the **Gateway** > **Alarming** folder.



Filtering Alarms

The **Perspective** and **Vision** Alarm Journal Table and Alarm Status Table components, along with **Alarm Journals** on the Gateway, all contain filter properties to further determine which alarms appear or are logged.

By default, all alarms within the system are included, but in many cases it can be desirable to only show a subset of alarms. Strings can be entered in these fields, including wildcards, and you can

enter multiple comma-delimited values to filter by. The filtering properties for components can also be bound so the paths are dynamically created.

Display Path

The default value for an alarm's Display Path is the tag path that leads to the name of the alarm. For example, if there is an alarm named **High Alarm** on a **Ramp3** tag within a **Ramp** folder, the Display Path would be `Ramp/Ramp3/High Alarm`. However, the Display Path can be customized when configuring the alarm by entering the desired path into the **Tag Editor** > **Alarms** > **Display Path** field. This is generally utilized to display messages describing the issue the alarm is tracking.

To filter entries on the table by the Display Path, simply set a value on the Display Path Filter property. The `*` wildcard can be used in the Display Path string.

Source Path

The Source Path is also a path to the alarm, but it uses a different syntax and includes the Tag Provider the alarm is located in. If the name of the Tag Provider is **Sample_Tags**, then the source path using the same tag as the Display Path Filtering example would resolve to `prov:Sample_Tags:/tag:Ramp/Ramp3:/alm:High Alarm`. Unlike the Display Path, the Source Path on an alarm can never be overridden.

To filter entries on the table by Source Path, simply set a value on the Source Filter property. The `*` wildcard can be used in the Source Filter. See the **Source Path Filter Examples** table below for more use cases.

Source Path Filter Examples

Since the Source Path is made up of the alarm name, the tag path to the host tag, and the Tag Provider the tag resides in, the Source Filter property can be used to filter on tag folders and Tag Providers. For example, you may want the table to only show alarms that contain a certain term or folder path. The table below demonstrates some filters as well as some hypothetical results.

Example Filter	Result
`prov:tagProvider:/tag:Inputs/PS_1:/alm:MyAlarm`	Retrieve alarm information from the alarm `prov:tagProvider:/tag:Inputs/PS_1:/`
`*PS_1:/alm:MyAlarm`	Retrieves alarm information from any path The following paths would be returned: `prov:tagProvider:/tag:Inputs/PS_1:/` `prov:tagProvider:/tag:anotherFolder`
`prov:tagProvider:/tag:PS_*`	Retrieves alarm information from any sour `prov:tagProvider:/tag:PS_`, such as: `prov:tagProvider:/tag:PS_1:/alm:MyA` `prov:tagProvider:/tag:PS_2/Tag2:/al`
`*MyAlarm*`	Retrieves any alarm information that has

Shelving Alarms

Shelving an alarm tells the system that it should ignore the alarm, so that no alarm notifications are generated and all corresponding alarm events are hidden from the Vision and Perspective Alarm Status Table components. Shelving is useful to ignore alarms for a period of time, such as when maintenance is being performed.

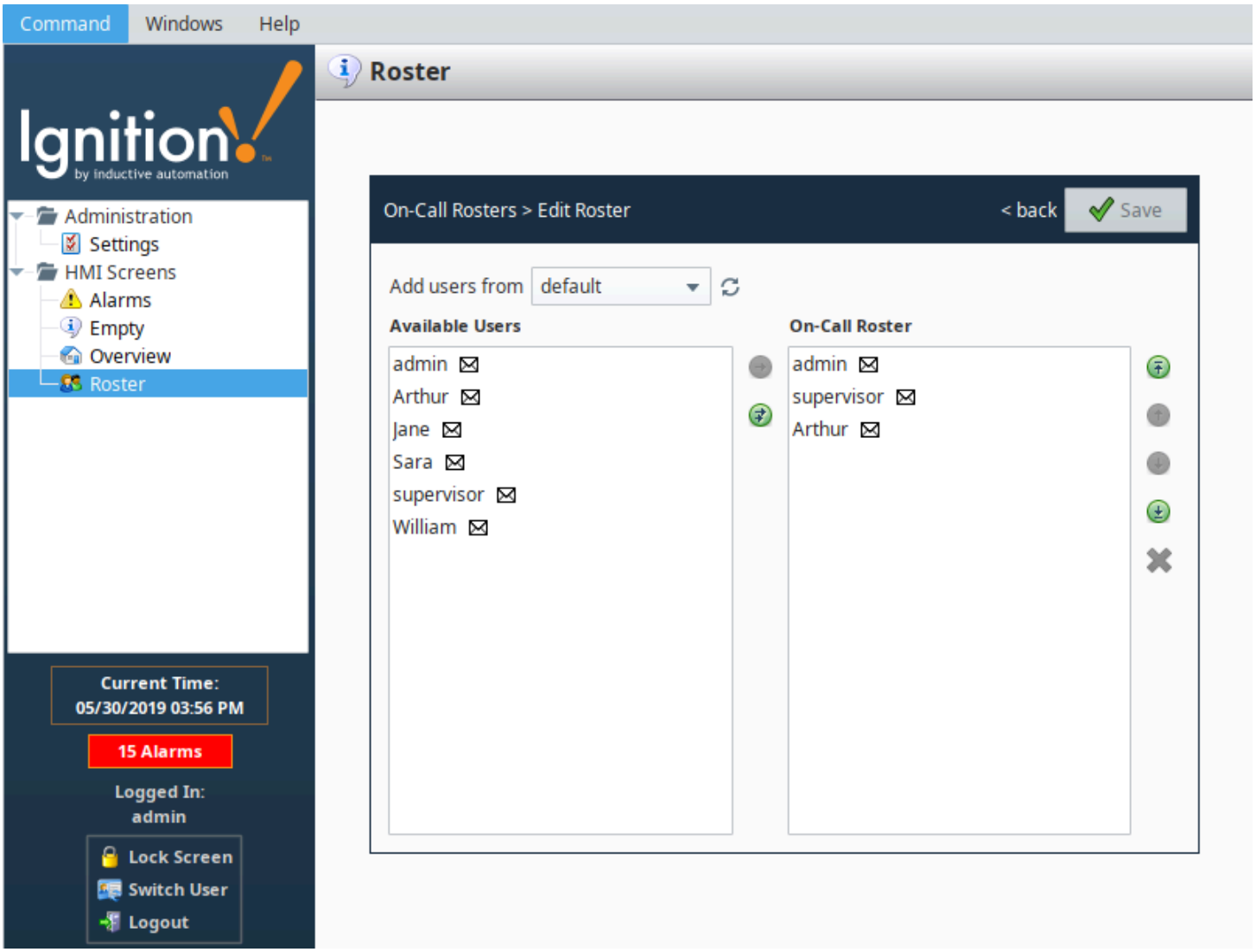
There is always a duration associated with a shelved alarm, and new alarm events for this alarm are temporarily prevented. At the end of the shelving period, the alarm will evaluate its source tag, potentially creating a new alarm event.

Generating Alarm Notifications

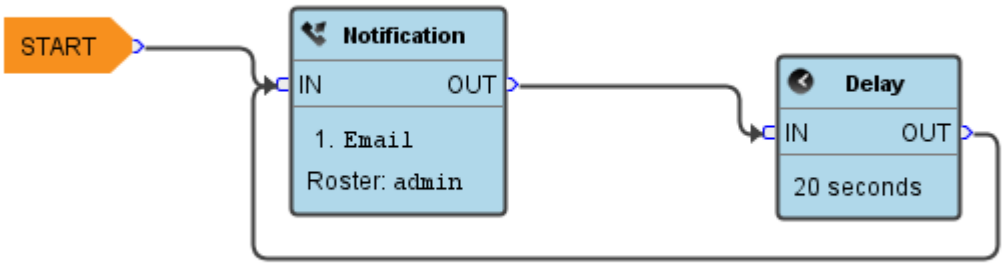
Alarms can generate notifications, which then can be delivered to users to immediately communicate events and problems. **Alarm Notification Pipelines** control how and when notifications are sent to users. The delivery channel for how alarms are sent can be either **Email**, **SMS**, or **Voice**.

By designing your own alarm notification routing, you have control of what happens when an alarm goes active, when an alarm is sent out, and who receives the alarm notification. This is done through a drag and drop interface and the configuration of **On-Call Rosters** and **Schedules**.

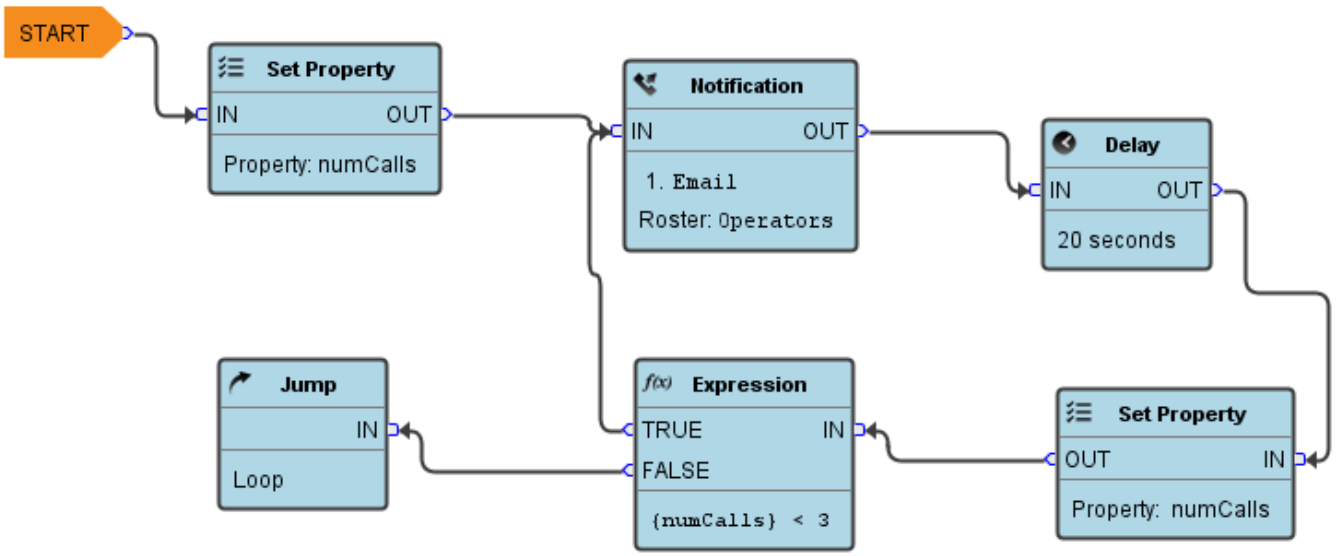
The On-Call Roster is a collection of users to notify when an alarm occurs based on their active schedule. For example, the Roster Management component below shows a list of six available users approved to handle an alarm, but only three will receive the alarm notification since they are included in the On-Call Roster.



Utilizing this On-Call Roster in a simple Notification Pipeline could result in the logic shown below. When an alarm is triggered, the people listed in the On-Call Roster are notified via Email. If no one acknowledges the alarm in 20 seconds, the alarm notification is routed back to the same users listed in the On-Call Roster.



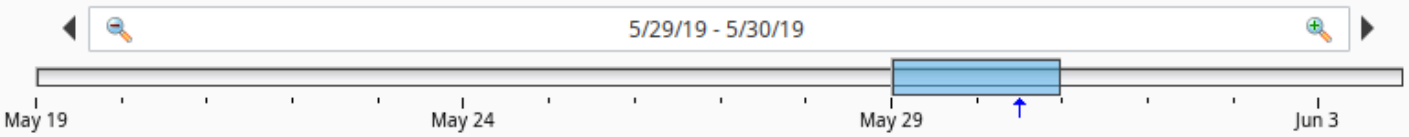
In a more complex Notification Pipeline, if an operator doesn't respond to the alarm after three attempts, the alarm jumps it to another pipeline, referred to as an escalation pipeline.



Viewing Alarm History

The **Alarm Journal** stores historical information about alarms in a database. It stores basic data about alarms that have occurred, such as their source and timestamp, associated data on an alarm, and the values of an alarm's properties at the time the event occurred. It captures all status changes for each alarm to an external SQL database of your choosing. To begin viewing alarm history, all you need to do is create an Alarm Journal Profile on the Gateway webpage.

Like the Alarm Status Table, both **Vision** and **Perspective** have components that can be used to view entries in an Alarm Journal.



Event Time	Display Path	Event State	Priority	Current State	Label
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Active	Critical	Active, Unacknowledged	Low Amps
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Ack	Critical	Cleared, Acknowledged	Low Amps
5/30/19, 11:45 AM	Ramp/Ramp8/OPC Alarm	Active	Critical	Active, Unacknowledged	OPC Alarm
5/30/19, 11:45 AM	Ramp/Ramp8/OPC Alarm	Ack	Critical	Cleared, Acknowledged	OPC Alarm
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Active	Critical	Active, Unacknowledged	Low Amps
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Ack	Critical	Cleared, Acknowledged	Low Amps
5/30/19, 11:45 AM	Ramp/Ramp8/OPC Alarm	Active	Critical	Active, Unacknowledged	OPC Alarm
5/30/19, 11:45 AM	Ramp/Ramp8/OPC Alarm	Ack	Critical	Cleared, Acknowledged	OPC Alarm
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Active	Critical	Active, Unacknowledged	Low Amps
5/30/19, 11:45 AM	Motors/Motor 3/Amps/Low Amps	Ack	Critical	Cleared, Acknowledged	Low Amps
5/30/19, 11:44 AM	Motors/Motor 3/Amps/Low Amps	Active	Critical	Active, Unacknowledged	Low Amps
5/30/19, 11:44 AM	Motors/Motor 3/Amps/Low Amps	Ack	Critical	Cleared, Acknowledged	Low Amps
5/30/19, 11:44 AM	Ramp/Ramp8/OPC Alarm	Active	Critical	Active, Unacknowledged	OPC Alarm
5/30/19, 11:44 AM	Ramp/Ramp8/OPC Alarm	Ack	Critical	Cleared, Acknowledged	OPC Alarm
5/30/19, 11:44 AM	Motors/Motor 1/Amps/Low Amps	Active	Critical	Active, Unacknowledged	Low Amps
5/30/19, 11:44 AM	Motors/Motor 1/Amps/Low Amps	Ack	Critical	Cleared, Acknowledged	Low Amps
5/30/19, 11:44 AM	Ramp/Ramp8/OPC Alarm	Active	Critical	Active, Unacknowledged	OPC Alarm

7,287 events

Alarm Journal

INDUCTIVE UNIVERSITY

Create Alarm Journal Profile

Watch the video

By default, current alarm data is only stored in memory, and a finite number of events are retained for each alarm. Ignition can easily be configured to store alarm data into a SQL database with an Alarm Journal Profile. The journal can store basic data about alarms that have occurred, such as their source and timestamp, associated data on the alarm, and the values of the alarm's properties at the time the event occurred.

The Gateway can have more than one Alarm Journal. Alarm Journals have options to filter which Alarms are stored in the journal, therefore, by having more than one Alarm Journal configured on the Gateway, it is possible to store some alarms in one journal, and different alarms in another journal. Once configured, the journal can be accessed by the **Alarm Journal Table** component, scripting functions, or direct database queries.

> **NOTE**
>
> You must have an Alarm Journal Profile created and have a valid database connection to use the Alarm Journal Table.

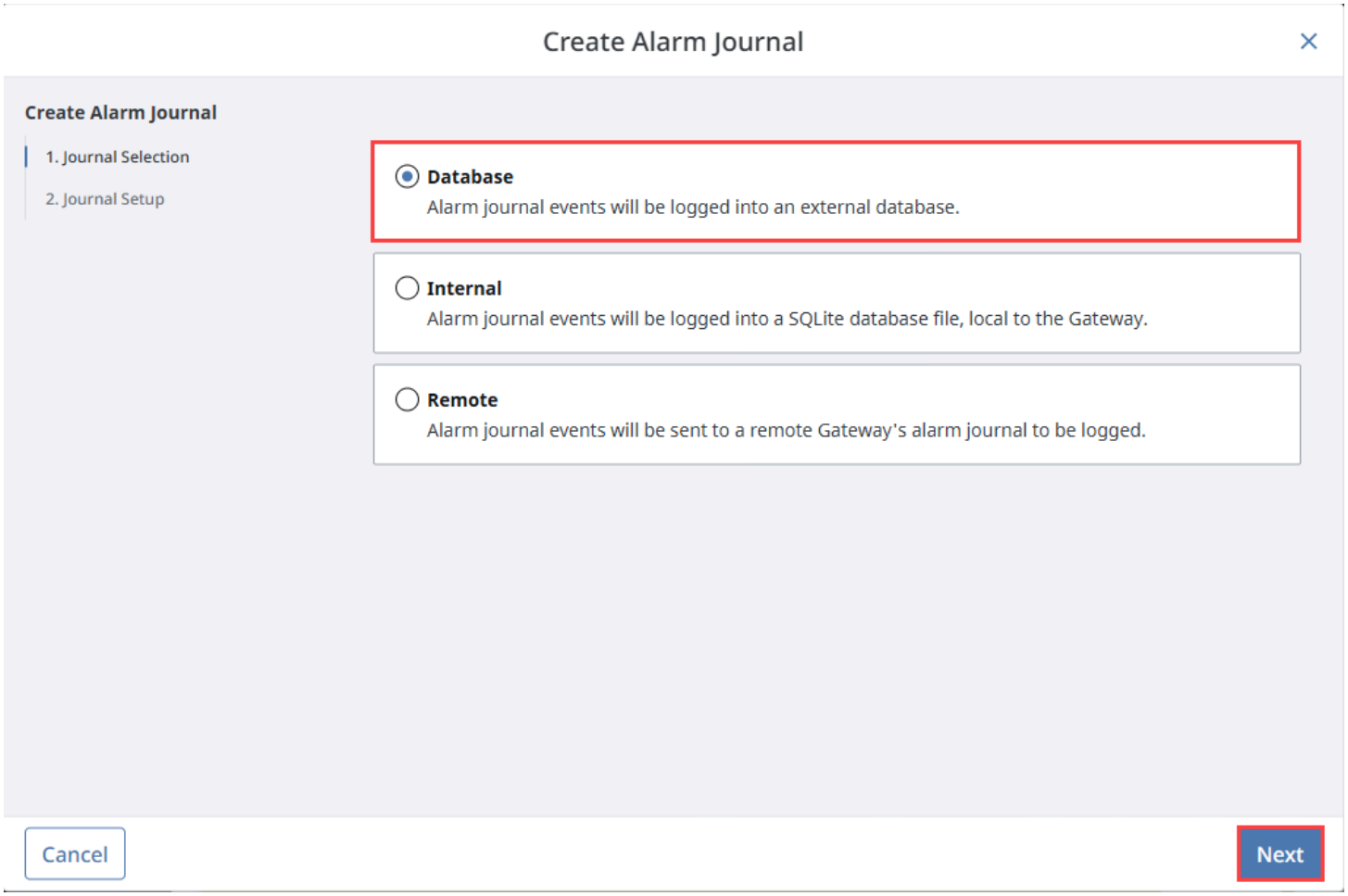
Alarm Journals can store data in one of three ways, and store data indefinitely unless a **Data Pruning** value is set:

- In a database, using an existing database connection from the Gateway
- Remotely, using another Ignition Gateway's Alarm Journal Profile
- Internally, storing alarm information into the Ignition install directory

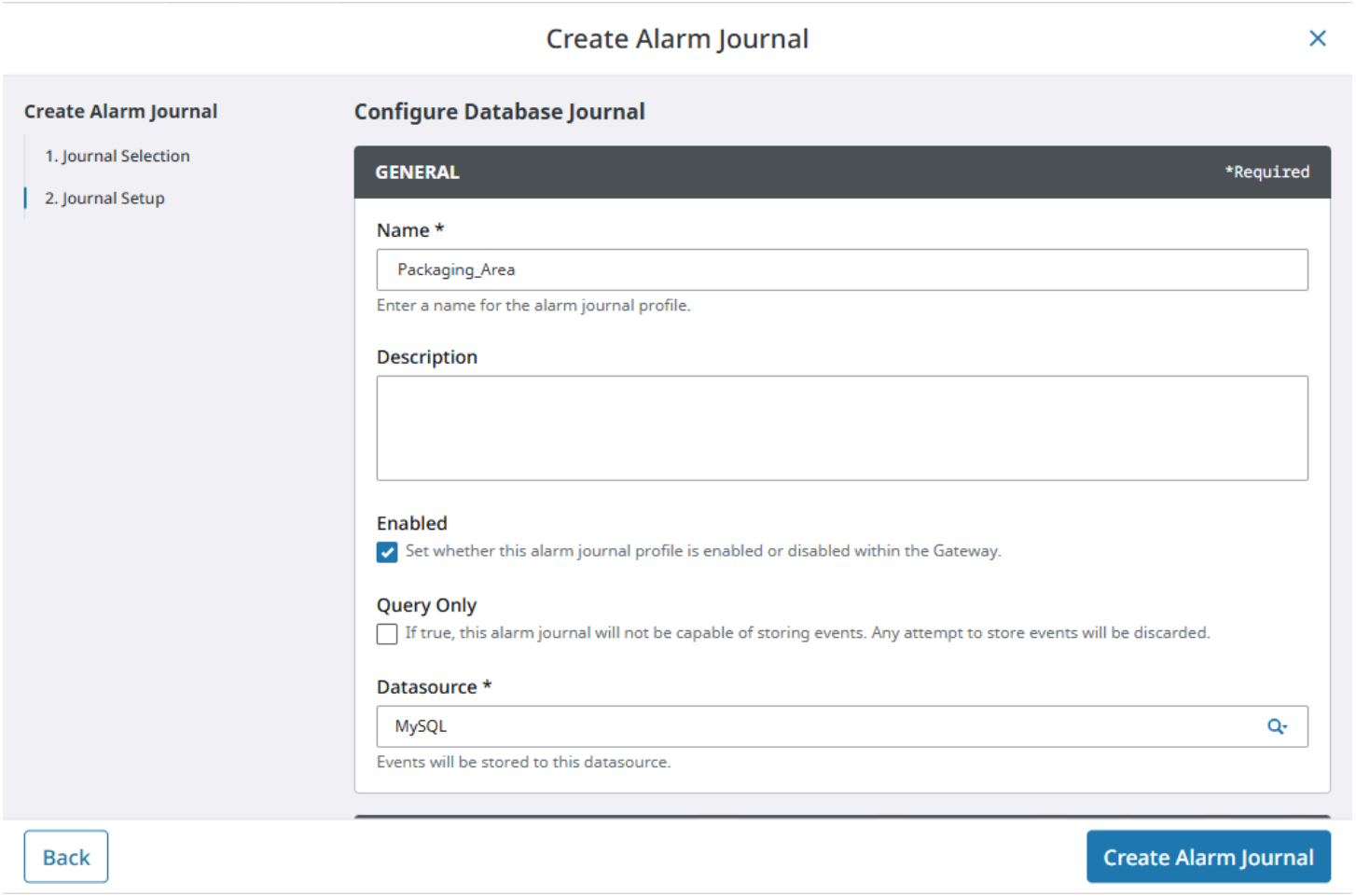
Create an Alarm Journal to Log Events to an External Database

In Ignition, an Alarm Journal stores historical information about alarms in a database. It can store basic data about alarms that have occurred, such as source and timestamp, along with associated data on the alarm, and the values of the alarm's properties at the time the event occurred.

1. Go to the **Services** section of the Gateway.
2. Choose **Alarming > Journals** from the menu on the left.
3. Click **Create Alarm Journal +**. The Create Alarm Journal form will be displayed.
4. You have the option of logging Alarm Journal events to an external database, logging locally, or sending them to a remote Gateway's Alarm Journal. In this example, select **Database**, and click **Next**.



5. Enter a **Name** for your Alarm Journal Profile and update other settings as required. Most of the fields have default settings that do not need to be changed. Refer to the **journal properties table** below for setting descriptions.



6. Once you are finished, click **Create Alarm Journal** at the bottom of the form.

> **NOTE**
>
> If you only have one Alarm Journal specified on your Gateway, then you do not need to specify the journal name on the Journal Name property. Ignition will set this for you. If you have more than one Alarm Journal created, then you need to provide the name of the journal you'd like to query in the Journal Name component property of the Property Editor.

Create a Remote Alarm Journal Profile

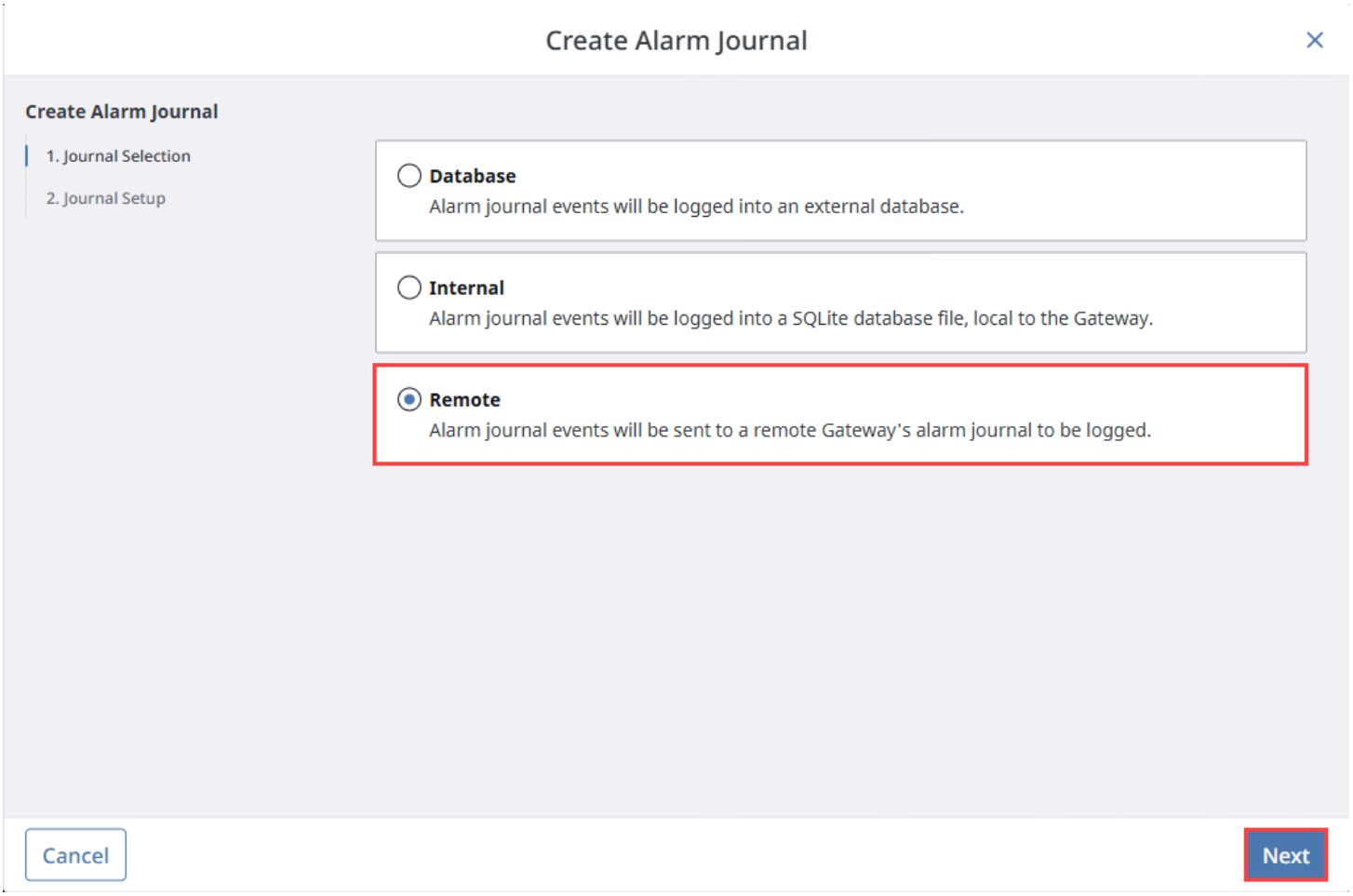
Utilizing the Gateway Network, remote Alarm Journal Profiles allow one Ignition Gateway to send local alarm events to a remote Gateway for journal logging. This type of profile is useful in cases where alarm events need to be recorded by multiple Alarm Journal Profiles. In addition, this type of profile is useful in **Hub and Spoke architectures** as it allows the hub to record the alarm events from each spoke.

> **CAUTION**
>
> Due to serialization updates, 8.3 Gateways will not be able to store Alarm Journal data to a remote 8.1 Gateway. It is recommended to first upgrade the central server hosting data to 8.3

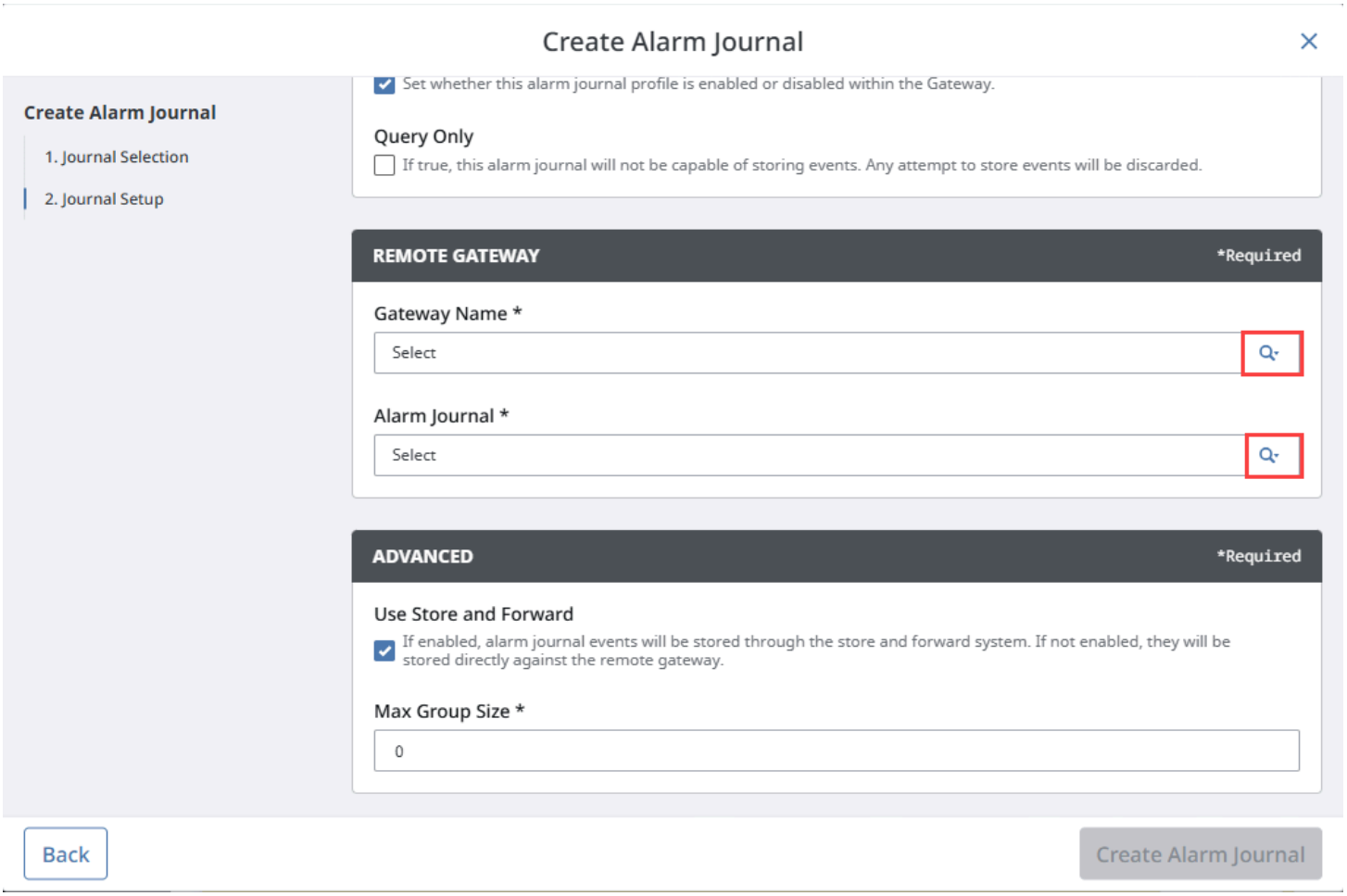
before upgrading any remote 8.1 Gateways. See the 8.3 Upgrade Guide for more information.

Just like configuring Alarm Journal events to be logged into a database, creating a Remote Alarm Journal Profile is done from the Gateway **Services > Alarming > Journals** page.

1. Click **Create Alarm Journal +**. The Create Alarm Journal form will be displayed.
2. Select **Remote**, and click **Next**.



3. A list of known Gateways and Alarm Journals will be available through the selection dropdowns. If you don't see a Gateway that you expected to see, check your Gateway Network settings to verify that the connections are valid.



4. When all settings are configured, click **Create Alarm Journal**.

You will receive a successful message stating your new Alarm Journal Profile is created.

Remote Gateway Alarm Journal Properties Table

Main

Name	Description
Name	The default name, is the name of the remote Gateway and Alarm Journal.
Description	Description of the journal profile. Optional
Enabled	Whether the journal profile is enabled. Defaults to true.
Query Only	If set to true, allows the Alarm Journal to opt out of being used for storage. When set to true, all alarm events will be discarded by the given journal.

Remote Gateway

Name	Description
Gateway Name	Name of the remote Gateway.
Alarm Journal	Name of the Alarm Journal on the remote Gateway.

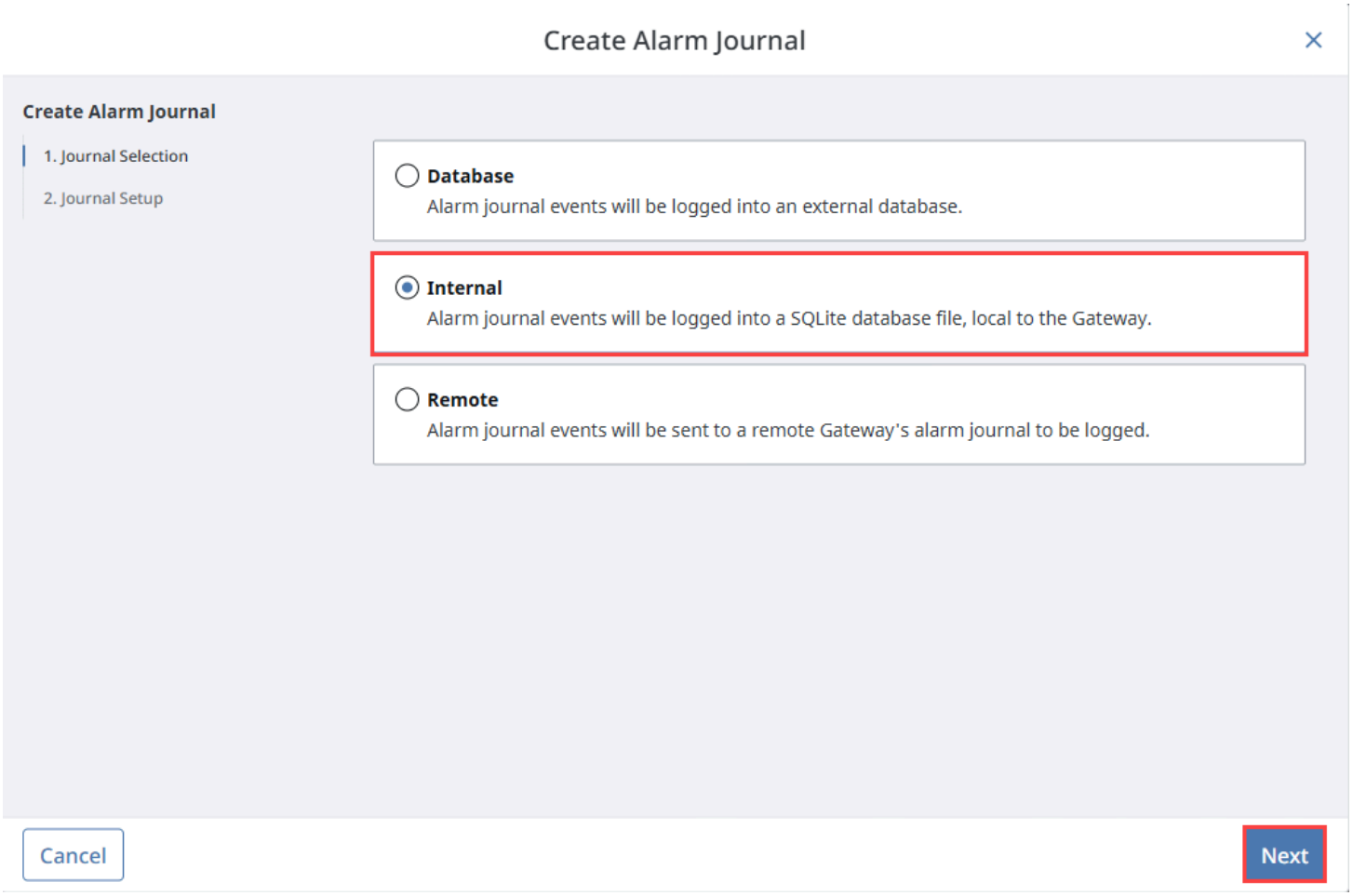
Advanced

Name	Description
Use Store and Forward	If enabled, Alarm Journal events will be stored through the Store and Forward system. If not enabled, they will be stored directly against the remote Gateway. Default is true.
Max Group Size	The maximum number of data points that can be sent per request. This value is used in conjunction with the Store and Forward setting to dictate how much data is sent at once. The default value of 0 means unlimited data points per request.

Create an Internal Alarm Journal to Log Events Locally

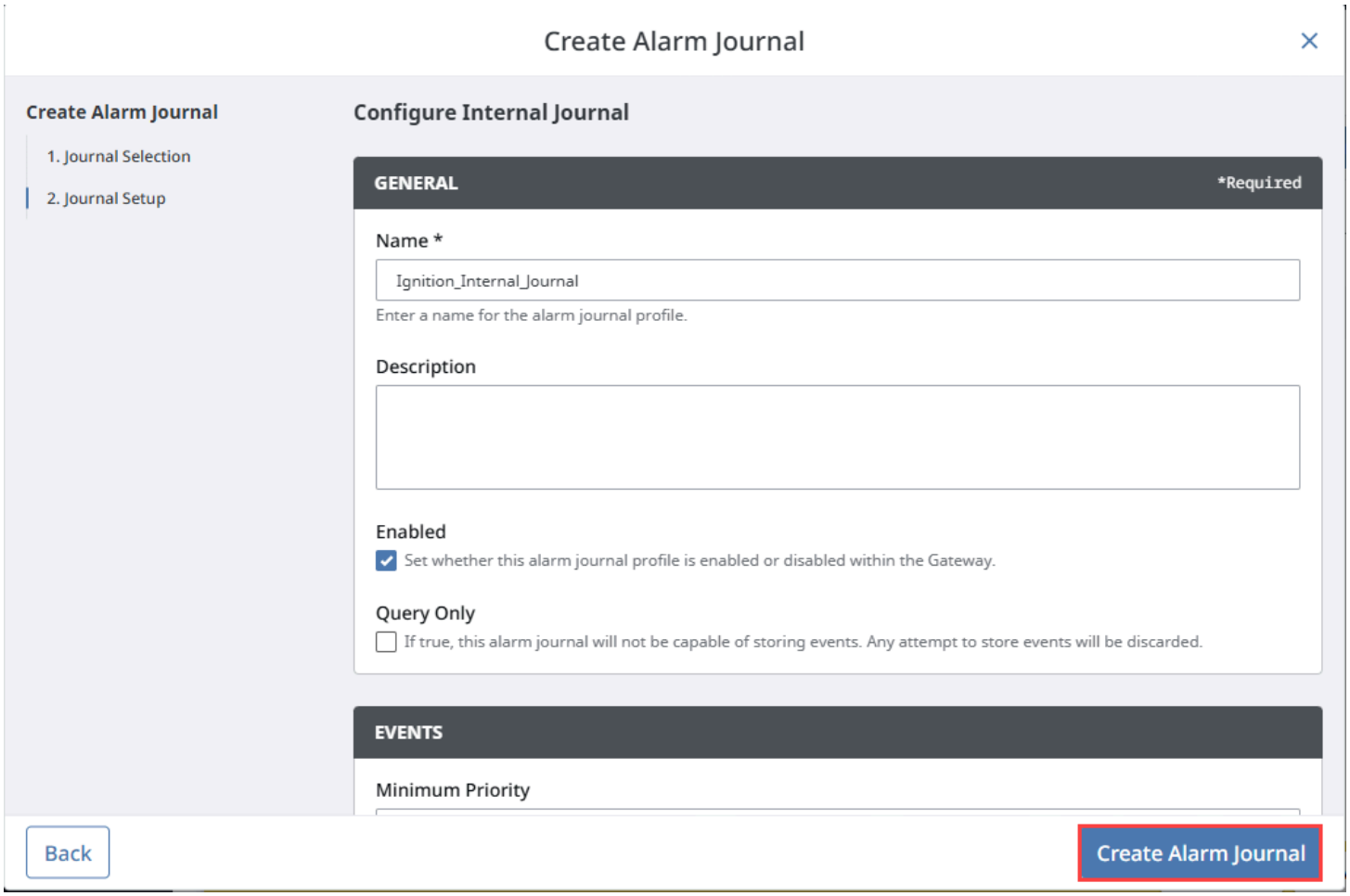
Ignition Gateways can also create an Internal Alarm Journal Profile that stores journal entries locally. Go to the Gateway **Services > Alarming > Journals** page to create an Internal Alarm Journal Profile.

1. Click **Create Alarm Journal +**.
2. Select **Internal** to have your alarm journal events logged locally, then click **Next**.



3. Enter the name of your Alarm Journal Profile and update any settings as required, then click **Create Alarm Journal**.

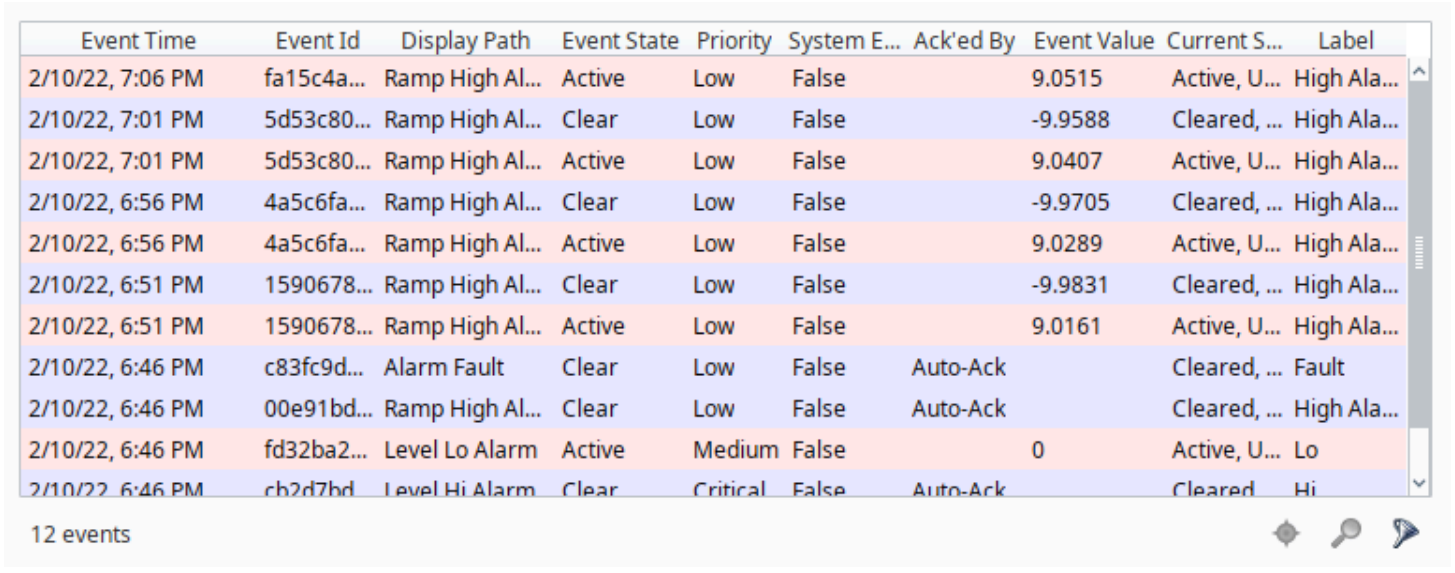
> **NOTE**
>
> It is strongly encouraged to set a Data Pruning value for Internal Alarm Journal Profiles to avoid running out of hard drive space.



You will receive a successful message stating your new Alarm Journal Profile was created.

Alarm Journal Component

The Vision and Perspective modules feature built-in components that can automatically retrieve events recorded in an Alarm Journal. See **Vision - Alarm Journal Table** and **Perspective - Alarm Journal Table** for more information.



Event Time	Event Id	Display Path	Event State	Priority	System E...	Ack'ed By	Event Value	Current S...	Label
2/10/22, 7:06 PM	fa15c4a...	Ramp High Al...	Active	Low	False		9.0515	Active, U...	High Ala...
2/10/22, 7:01 PM	5d53c80...	Ramp High Al...	Clear	Low	False		-9.9588	Cleared, ...	High Ala...
2/10/22, 7:01 PM	5d53c80...	Ramp High Al...	Active	Low	False		9.0407	Active, U...	High Ala...
2/10/22, 6:56 PM	4a5c6fa...	Ramp High Al...	Clear	Low	False		-9.9705	Cleared, ...	High Ala...
2/10/22, 6:56 PM	4a5c6fa...	Ramp High Al...	Active	Low	False		9.0289	Active, U...	High Ala...
2/10/22, 6:51 PM	1590678...	Ramp High Al...	Clear	Low	False		-9.9831	Cleared, ...	High Ala...
2/10/22, 6:51 PM	1590678...	Ramp High Al...	Active	Low	False		9.0161	Active, U...	High Ala...
2/10/22, 6:46 PM	c83fc9d...	Alarm Fault	Clear	Low	False	Auto-Ack		Cleared, ...	Fault
2/10/22, 6:46 PM	00e91bd...	Ramp High Al...	Clear	Low	False	Auto-Ack		Cleared, ...	High Ala...
2/10/22, 6:46 PM	fd32ba2...	Level Lo Alarm	Active	Medium	False		0	Active, U...	Lo
2/10/22, 6:46 PM	cb2d7bd...	Level Hi Alarm	Clear	Critical	False	Auto-Ack		Cleared	Hi

12 events

Alarm Journal Profile Properties

The following properties are available to both Database and Internal type Alarm Journal Profiles, unless otherwise noted. Since Remote Alarm Journal Profiles simply configure the alarm events to be logged to an existing profile, they have separate **settings**.

Main

Name	Description
Name	The name of the Alarm Journal Profile.
Description	Description of the journal profile.
Enabled	Whether the journal profile is enabled. Defaults to true.
Query Only	When enabled, the Alarm Journal will not store alarm events.
Datasource	Events are stored to this data source. (Only available on Database type profiles)

Events

Name	Description
Minimum Priority	Only events equal to or greater than the specified priority will be stored. The default is Low. You can set the priority to be: Diagnostic, Low, Medium, High, and Critical.
Store Shelved Events	Not enabled by default. If enabled, events generated by shelved alarms will still be written to the journal system.
Store Enabled & Disabled Events	When enabled, events generated by enabling or disabling alarms will be stored in the journal. This includes cases where the **Enabled** property on an alarm is toggled, as well as cases where a tag's **Alarm Eval Enabled** property is changed. This property additionally relies on setting the Perspective/Vision Alarm Journal Table properties for the enabled and disabled events.

Event Data

Name	Description
Static Config	By default, it is not selected. If selected, will store the values of static alarm configuration. That is, the alarm properties that are not bound. These do not change during evaluation, only when a user modifies them in the Designer, and so they are not stored by default.
Dynamic Config	If selected, will store the values of dynamically bound alarm configuration properties. The value of these properties can change at any time, and the values at the time of the alarm are captured on the alarm event.
Static Associated Data	If selected, will store the values of non-bound associated data (properties created by the user) properties on alarm that do not change during execution.
Dynamic Associated Data	If selected, will store the values of dynamically bound associated data (properties created by the user) properties.

Data Filters

The three data filter properties (**Filter by Alarm Source**, **Filter by Display Path**, and **Filter by Display Path or Source**) interact via logical AND, meaning an alarm must meet the criteria for all given values for it to be logged in the Journal. It is recommended to avoid using all three properties simultaneously on the same Journal.

For example, if a journal has values for all three properties, and an alarm only meets the requirements for **Filter by Alarm Source** and **Filter by Display Path or Source**, but not **Filter by Display Path**, then the alarm will not be logged to the Journal.

Name	Description
Filter by Alarm Source	Only events matching the source will be stored. Multiple sources to match can be comma separated. Leave blank to store events from all sources.
Filter by Display Path	Only events matching the display path will be stored. Multiple display paths to match can be comma separated. Leave blank to store events from all display paths.
Filter by Display Path or Source	Only events matching the display path, if defined, will be stored. Multiple matches can be comma separated. If no display path is defined, only events matching the source will be stored. Leave blank to store all events.

Refer to the **Filtering Alarms** section on the Alarming page for more information on Display Path and Source syntax.

Data Pruning

Name	Description
Enable Data Pruning	If selected, the system will automatically delete data after the specified time period as set by the Prune Age and Units below. Default is false. **Note**: If using the Database type to store directly in a database, an administrator is free to manually delete data at any time.
Prune Age	The number of Prune Age Units to store data for. i.e., 1 year, 5 hours, etc. The default is 1.
Prune Age Units	The type of Prune Age Unit. Default is Years. You can choose the unit to be Milliseconds, Seconds, Minutes, Hours, Days, Weeks, Months, or Years.

Advanced Properties

These settings apply to Database Alarm Journal Profiles only.

Name	Description
Table Name	The table name for the core event table. The default is `alarm_events`.
Event Data Table Name	The table name for event data associated with alarms. The default is `alarm_event_data`.
Use Store and Forward	Enabled by default, which means the Alarm Journal events will be stored through the **Store and Forward system**. If not enabled, they will be stored directly against the database. This system protects data from being lost due to temporary database connectivity issues.

Database Table Definitions

The Alarm Journal system will automatically create the necessary tables for you, and scripting functions can be used to query the system without having to know about the table structure. However, understanding the structure of the Alarm Journal tables can be useful for accessing the data in situations where SQL queries are more convenient.

Alarm Events (alarm_events)

This table stores the core data for each event that occurs. An event is a transition for an alarm between active, cleared, or acknowledged. Additionally, other events may be stored in this table that aren't directly related to an alarm, such as a system shutdown event. This table defines a primary key `id`, that is used as a foreign key by the Alarm Event Data table, which stores additional information for each event.

> **FILTERING ALARMS**
>
> You can filter which alarms are logged to the Alarm Journal and Alarm Events table. See the Data Filters section for more information.

Column Name	Data Type	Description
id	integer	A unique integer id for each event entry event
eventid	string	The UUID of the alarm event that this individual event is related to. Each alarm event (one particular active/clear/ack cycle of a defined alarm) receives a unique id in order to distinguish it from other events from the same source.
source	string	The qualified path of the entity that generated the alarm event. See below for more information about qualified paths.
display path	string	The value set for the Display Path of the alarm. Generally a user defined, friendlier version of the source.
priority	integer	The priority or severity of the alarm: • 0: Diagnostic • 1: Low • 2: Medium • 3: High • 4: Critical
eventtype	integer	The type of transition represented by this event: • 0: Active • 1: Clear • 2: Acknowledged • 4: An alarm was enabled • 5: An alarm was disabled
eventflags	integer	A numeric bitmask flag field providing additional information about the event. See the **example** below for how to adjust the eventflags value into a more readable format. • Bit 0: System Event - One of the designated system events. (System Startup, System Shutdown) • Bit 1: Shelved Event - The alarm was shelved at the time that the event occurred. Shelving alarms does not prevent execution, so if the Journal is configured to store shelved

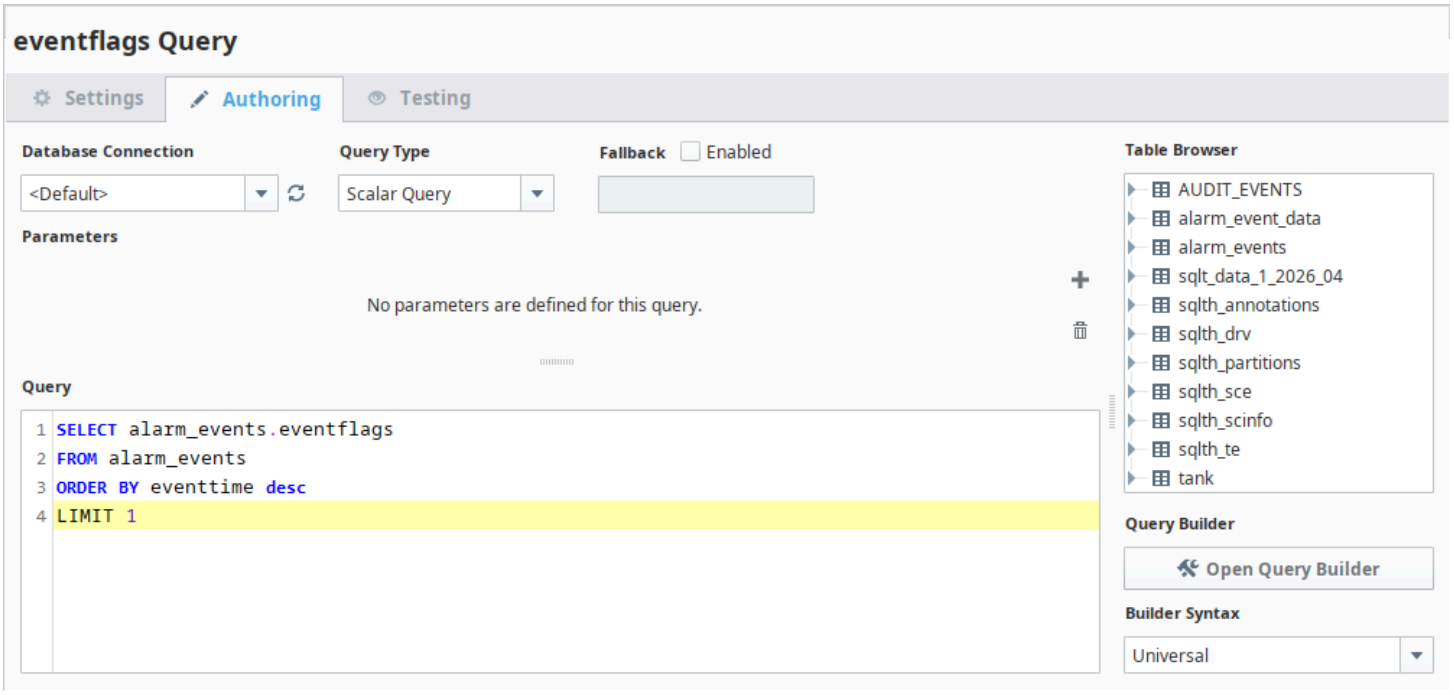
Column Name	Data Type	Description
		events, they will be stored even if they're not sent to the notification system, or shown to users. • Bit 2: System Acknowledgement - When the Live Event Limit (defined in general alarm settings) is exceeded, the system will automatically acknowledge overflow events, and the acknowledgment event will have this flag set. • Bit 3: Acknowledged Event - The event was acknowledged at the time of the event. Used for events that are cleared after being acknowledged. • Bit 4: Cleared Event - The event was cleared at the time of the event. Used for alarms that are acknowledged after being cleared. • Bit 5: Enabled - Signifies that the enabled state on the alarm was changed.
eventtime	datetime	The time of the event.

Example - eventflags Property

The following example helps format the `eventflags` property's value into a more readable format. The script uses a Named Query to get the most recent alarm's `eventflags` value and prints the result as a list of strings, corresponding to their bit value. This example assumes you have at least one Alarm Journal alarm event.

Get the most recent eventflags value using a scalar Named Query

```
SELECT alarm_events.eventflags
FROM alarm_events
ORDER BY eventtime desc
LIMIT 1
```



TIP

Notice that we are using a Scalar Query for the Query Type, since errors for the script below may be thrown if the wrong Type is selected.

Use the Named Query in a script to get the eventflags as a string

```
# Import the OrderedDict data type from the collections module
from collections import OrderedDict

# Declare a variable for Named Query result
event_flags = system.db.runNamedQuery("eventFlagIntValue")

def convert_event_flags(eventflags):
    """
    Converts an integer representing event flags to a binary string and interprets
each bit.

    The function formats the `eventflags` integer into a 6-bit binary string. Bits
are then assessed
    from least significant to most significant (right to left) to determine the
status of various flags.

    Parameters:
    - eventflags (int): The integer value of the event flags.

    Returns:
    - OrderedDict: A dictionary that includes the original integer value, the
binary representation,
      and boolean status for each event type based on the corresponding bits in
the binary string.
    """

    eventflags_bin = format(eventflags, '06b')
    return OrderedDict([
        ("int_value", eventflags),
        ("binary_value", eventflags_bin),
        ("System Event", bool(int(eventflags_bin[5]))),
        ("Shelved Event", bool(int(eventflags_bin[4]))),
        ("System Acknowledgement", bool(int(eventflags_bin[3]))),
        ("Acknowledged Event", bool(int(eventflags_bin[2]))),
        ("Cleared Event", bool(int(eventflags_bin[1]))),
        ("Enabled", bool(int(eventflags_bin[0])))
    ])

# Print the result
print convert_event_flags(event_flags)
```

Alarm Event Data (alarm_event_data)

This table stores the properties associated with an alarm event. The individual event is referenced through the `ID` column, against the alarm event table.

Column Name	Data Type	Description
id	integer	The id that corresponds to the alarm event in the alarm_events table.
propname	string	The name of the property. May be one of the common alarm properties (a configuration setting), or the name of an associated data property.
dtype	integer	The data type of the property, indicating which data column should be used: • 0: Integer • 1: Float • 2: String
intvalue	integer	The corresponding value columns for the property. Unused columns will receive `null` values.
floatvalue	float (double)	The corresponding value columns for the property. Unused columns will receive `null` values.
strvalue	string	The corresponding value columns for the property. Unused columns will receive `null` values.

Qualified Paths

A qualified path in Ignition is a path to an object, described by various annotated components. Each component has a type identifier and a value, separated by a colon (`:`), and each component is separated by colon-forward slash (`:/`). For example, an alarm is identified by `alm:Alarm Name`. It usually exists under a tag, in which case, its fuller path would be `tag:Path/To/Tag:/alm:Alarm Name`. Paths can be built up further depending on the level of specificity required by the situation.

Configuring Alarms

This page covers configuring alarms on tags and OPC items, alarm property reference, and alarm property bindings.

Alarms can be configured on Memory tags, Query tags, Expression tags, tags inside a UDT, OPC items in **SQL Bridge (Transaction Groups)**, and **System tags** such as Gateway Performance and CPU Usage.

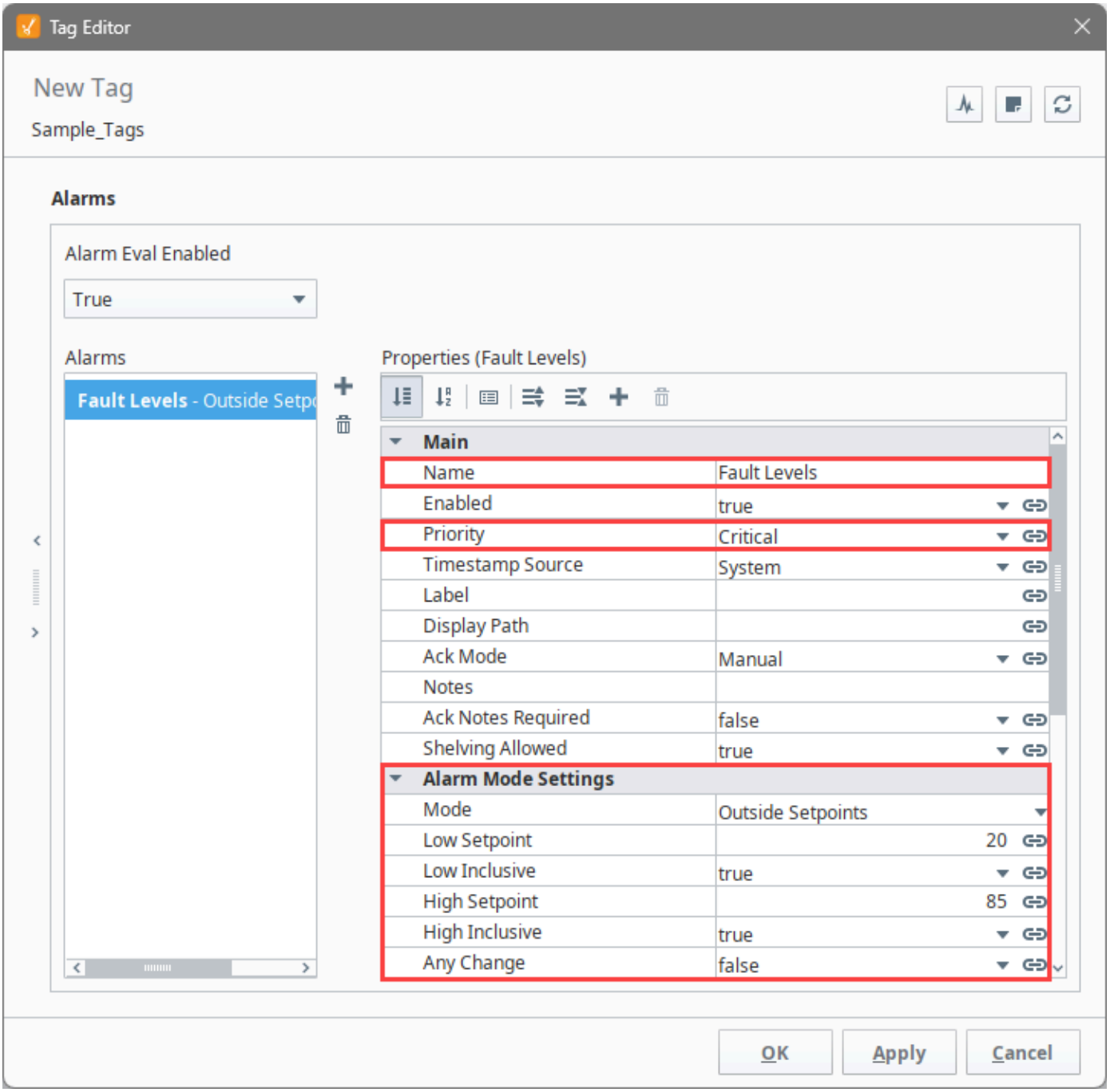
> **NOTE**
>
> Dataset type tags are not supported by the Alarming system.

Configuring an Alarm on a Memory Tag

This example uses a Memory tag with an Integer data type.

1. In the Tag Browser, double-click a Memory tag to open the **Tag Editor**.
2. In the **Categories** panel, select **Alarms**.
3. Click the **Add** + icon in the bottom-left corner of the window, or double-click the **New Alarm** item to add a new alarm.
4. Configure the following alarm settings:
    - Name: **Fault Levels**
    - Priority: **Critical**
    - Mode: **Outside Setpoints**
    - Low Setpoint: **20** (alarm triggers when the value falls below 20)
    - Low Inclusive: `true` (alarm also triggers when the value equals 20)
    - High Setpoint: **85** (alarm triggers when the value rises above 85)

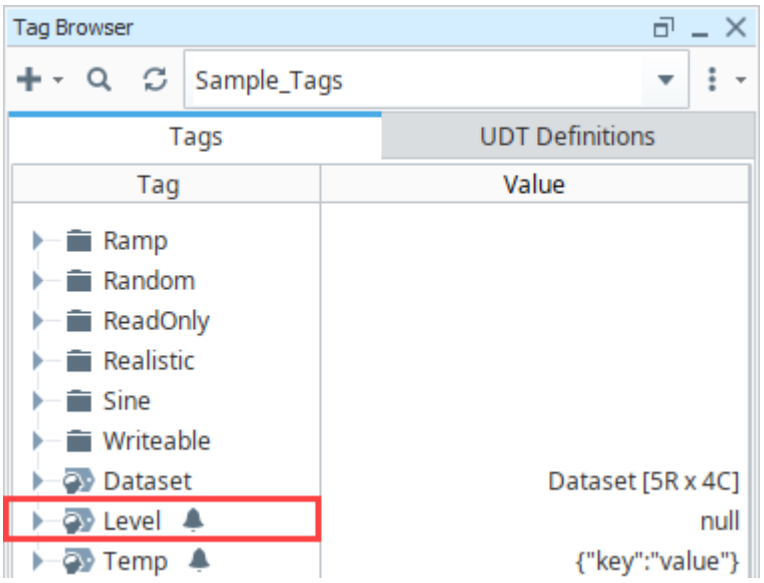
- High Inclusive: `true` (alarm also triggers when the value equals 85)



5. Click **Apply** to save your alarm.

6. Click **OK** to save your tag edits.

Ignition will start monitoring the alarm immediately. The Alarm icon appears in the Tag Browser next to the tag name, indicating at least one alarm is configured on the tag.

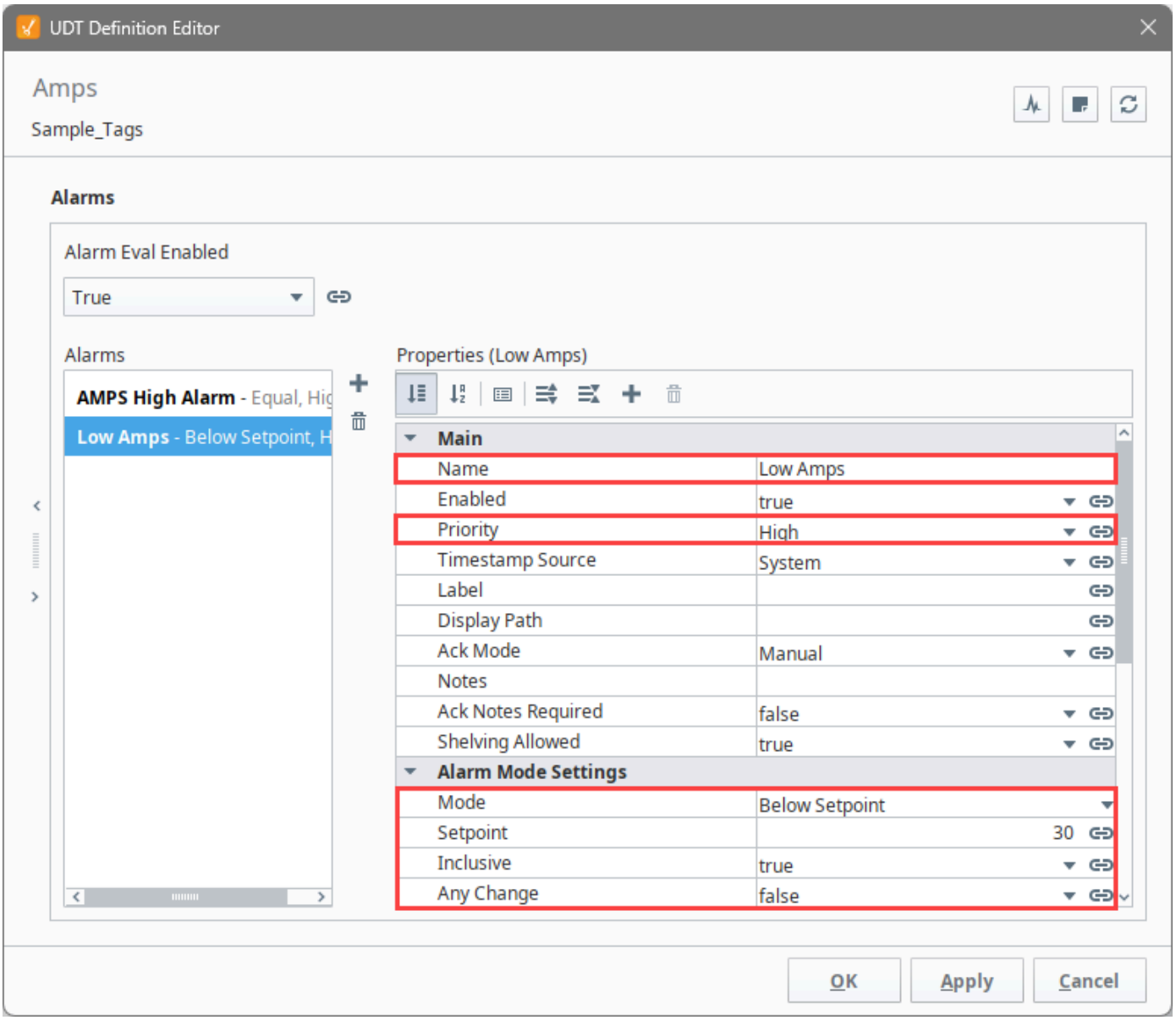


Configuring Alarms on Tags in a UDT

Alarms can be configured on tags inside a UDT so that when you create instances of that UDT, the alarms are automatically applied.

This example configures an alarm on the Amps tag of a Motor UDT.

1. In the **Tag Browser**, double-click a UDT to open the **Tag Editor**.
2. In the Type Structure area, click a tag.
3. In the **Categories** panel, select **Alarms**.
4. Click the **Add** ✚ icon.
5. For this example, configure the following settings:
    - Name: **Low Amps**
    - Priority: **High**
    - Mode: **Below Setpoint**
    - Setpoint: **30**
    - Inclusive: `true` (alarm triggers when the value equals 30 or falls below)



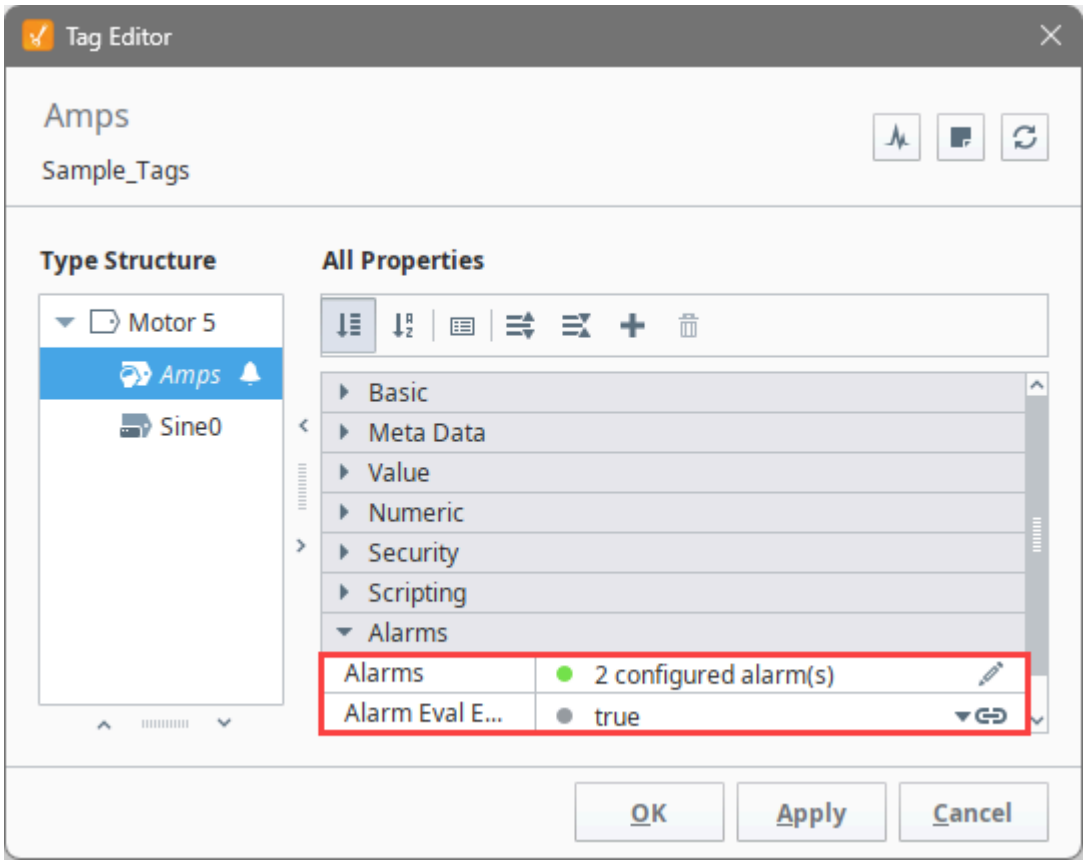
6. Click **Apply** to save your alarm.

7. Click **OK** to save the UDT.

Once you create UDT instances, Ignition will start monitoring the alarm immediately.

Updates to the UDT and Overriding Alarm Settings

If an alarm in the UDT is updated, instances automatically receive the update. When alarm property values need to be unique for a specific instance, you can override alarm settings. Click the override button on the alarm UI in the Tag Editor (it turns green when active) and edit the properties and their values as needed. For more information, refer to **Overriding Properties of the Parent UDT**.



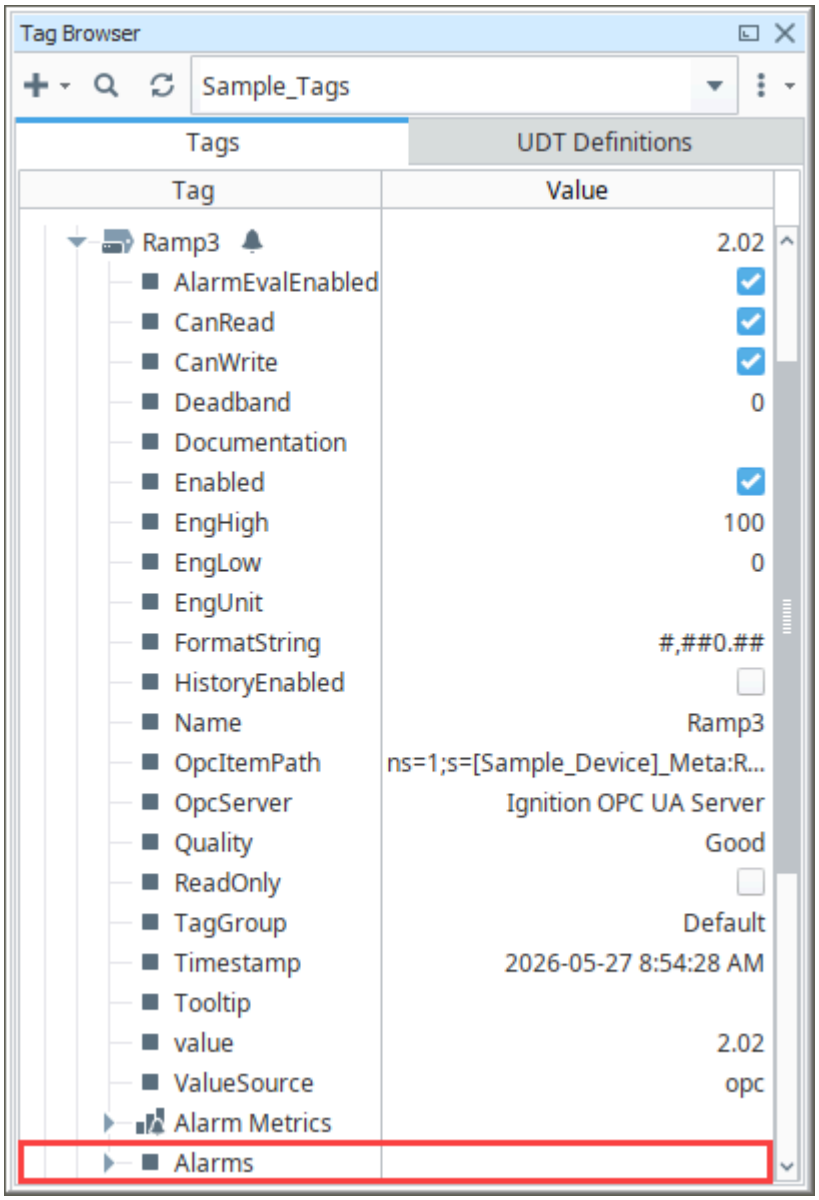
Configuring Multiple Alarms on a Single Tag

Tags can have multiple alarms configured, each with unique setpoints and other settings. Adding additional alarms follows the same process as adding the first. Click the **Add** + icon in the alarm UI and configure the new alarm. This allows separate alarms to monitor different thresholds, such as both high and low setpoints on the same tag.

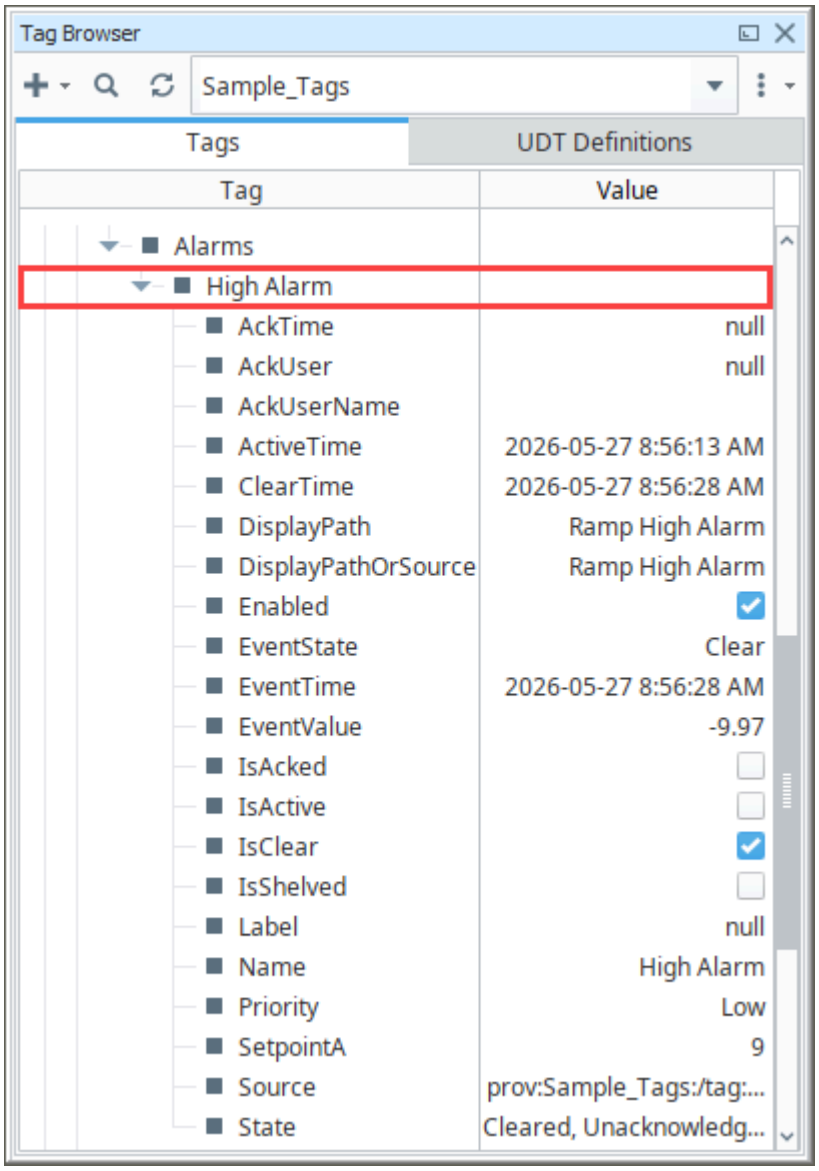
Alarm Properties

Alarm properties control the behavior of each alarm. For a complete reference, see the **Tag Alarm Properties** page.

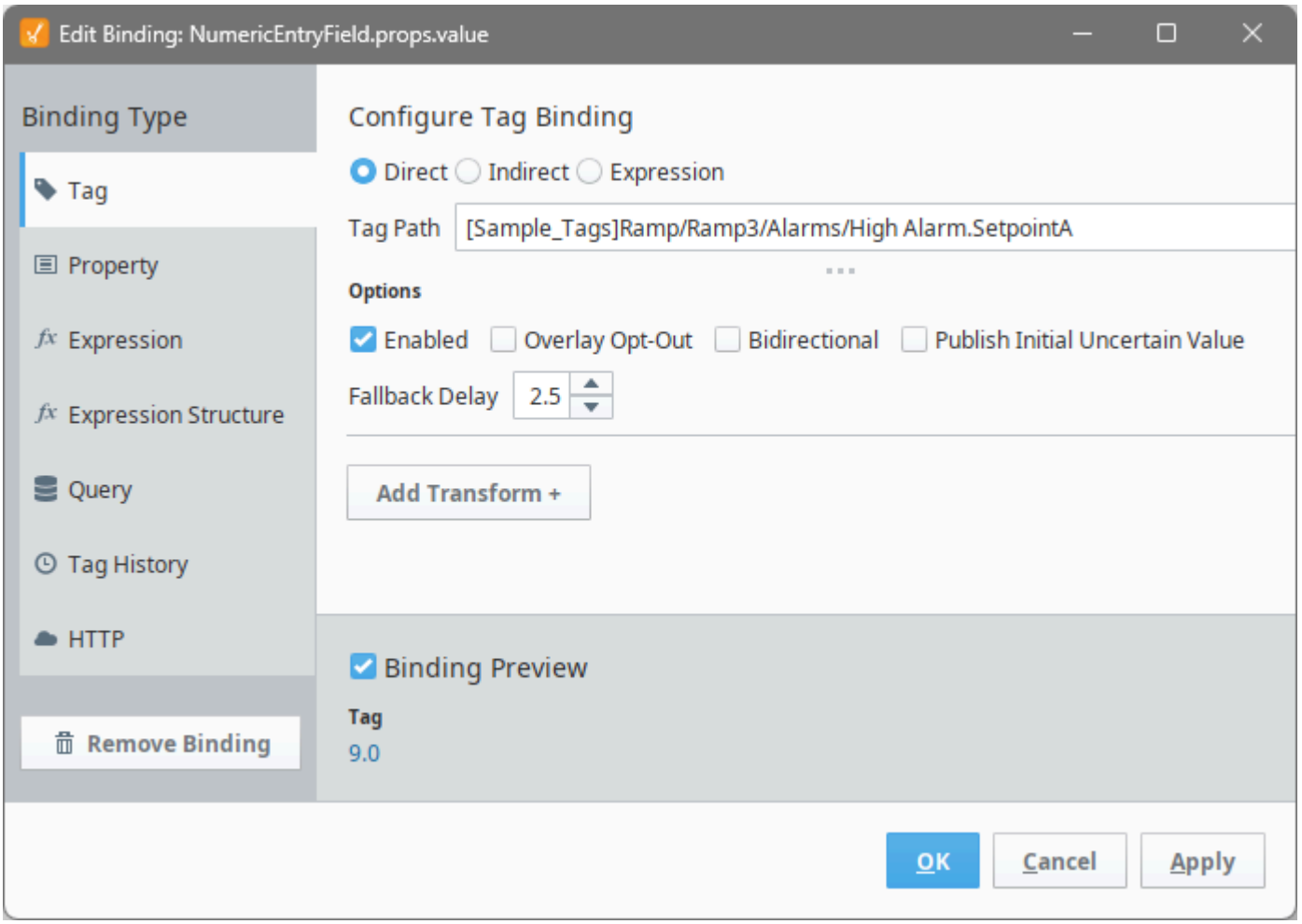
Configured alarms appear under the **Alarms** node of an alarmed tag in the Tag Browser.



Expanding an individual alarm reveals its runtime attributes, which are accessible to Tag bindings and scripts.



Setpoint and Enabled properties can be modified directly through tag attribute writes at runtime. For example, a Perspective component can bind directly to an alarm's setpoint attribute.

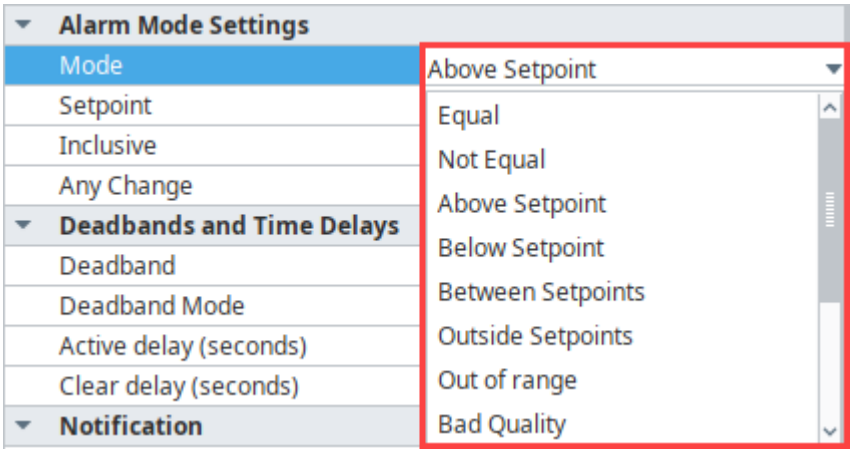


Property	Description
Name	A unique identifier for the alarm. Supports forward slashes (`/`), though only the characters after the rightmost slash are displayed in alarm table components, the Tag Browser, and alarm scripting functions. To avoid confusion, avoid using forward slashes in alarm names.
Priority	The priority level of the alarm. Options include Diagnostic, Low, Medium, High, and Critical.
Display Path	An organizational path used to group alarms in alarm display components.
Notes	Supplemental notes or information about the alarm.
Enabled	Determines whether the alarm is active. Can be bound to a tag to toggle alarming at runtime. For example, this can disable alarms when production is stopped.

Alarm Modes and Setpoints

Each alarm is configured with one mode and usually one or more setpoints. The mode determines how alarm activity is evaluated, while setpoints are the thresholds that determine when a tag is in an alarm state. Both work together to control when alarms become active and when they clear.

The available modes can be selected from the Mode dropdown.



Property	Description
Mode	The condition that triggers the alarm (e.g., Above Setpoint, Below Setpoint, Between Setpoints, Outside Setpoints, Bit State, Bad Quality). For the full list, see the **Alarm Mode Settings** table.
Setpoint	The threshold compared against the tag's value. Usage varies by mode.
Low Setpoint	The lower threshold for range-based modes such as Between Setpoints or Outside Setpoints.
High Setpoint	The upper threshold for range-based modes.
Inclusive	When `true`, the alarm triggers when the tag value equals the setpoint.
Bit Position	Used with **Bit State** mode. Specifies which bit of an integer tag to monitor.

A common PLC pattern is to pack multiple alarm conditions into a single integer tag, where each bit represents one condition. Ignition can read that integer as a tag through its OPC UA server, and the **Bit State** alarm mode lets you configure a separate alarm for each bit. The **Bit Position** property specifies which bit the alarm monitors.

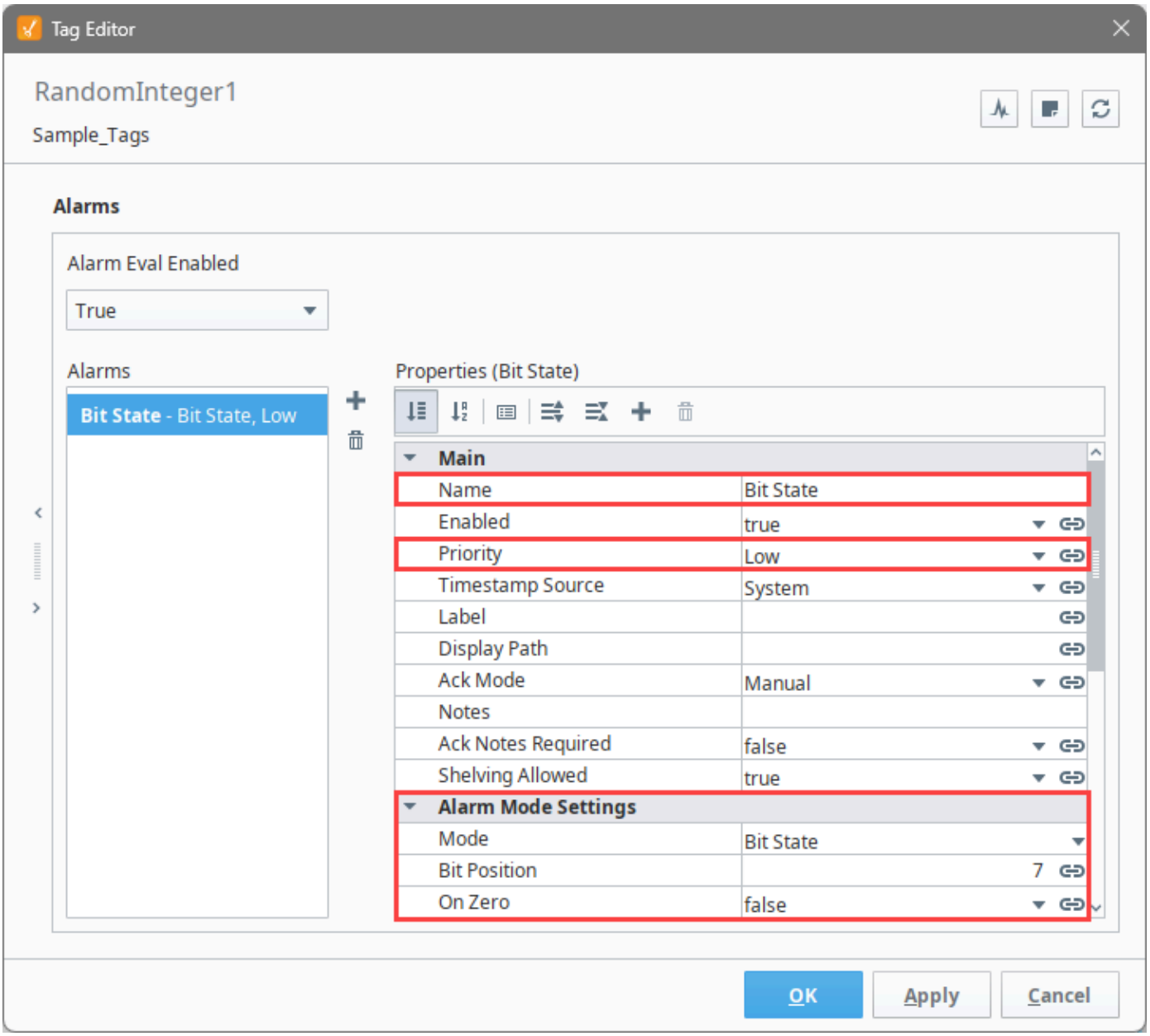
For example, an 8-bit integer with a value of 142 has the binary representation `10001110`. A tag holding this value can have up to eight alarms configured, one for each bit, each triggering based on

the state of its assigned bit.

Binary	Decimal
10001110	142

Configuring Alarms on Individual Bits

1. In the **Tag Browser**, double-click an integer tag to open the **Tag Editor**. In this example, we will use the RandomInteger1 tag.
2. In the **Categories** panel, select **Alarms**.
3. Click the **Add** ➕ icon to add a new alarm.
4. Configure the following alarm settings:
    - Name: **Bit State**
    - Priority: **Low**
    - Mode: **Bit State**
    - Bit Position: **7**



5. Click **Apply** to save your alarm.

6. Repeat steps 3 through 5 for each additional bit, incrementing the **Bit Position** value as needed.

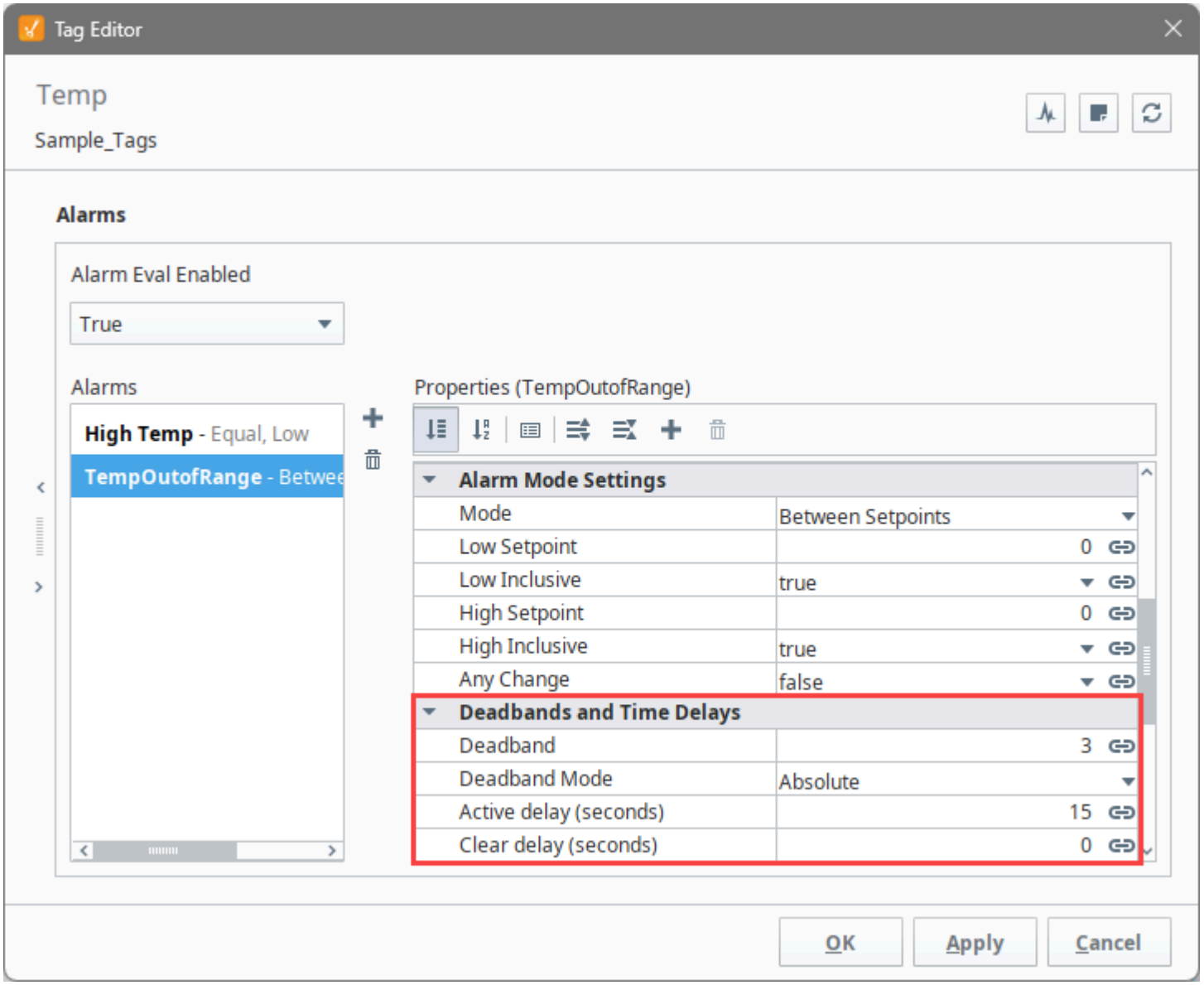
Deadband and Time Delays

The deadband value is interpreted by the Deadband Mode. All alarms are only evaluated after a tag's value changes, meaning the tag's own deadband is applied first. When the deadband is positive, an active alarm must clear its setpoint(s) by the deadband amount before the alarm transitions to cleared.

For example, a Between Setpoints alarm with a Low Setpoint of 45, a High Setpoint of 65, and a Deadband of 3 becomes active when the value is between 45 and 65, but only clears when the value falls below 42 or rises above 68.

The **Active Delay** property prevents alarms from triggering until a tag has remained in an alarm state for a set period of time. This is useful when a tag briefly and frequently enters and leaves an alarm

state.



Property	Description
Deadband Mode	Determines how the deadband value is applied.
Deadband	The amount by which an active alarm condition must clear its setpoint(s) before the alarm transitions to cleared.
Active Delay	The duration a tag must remain in an alarm state before the alarm is triggered.
Clear Delay	The duration a tag must remain in a cleared state before the alarm is marked as cleared.

For more information, refer to the **Reference Table** on the Tag Alarm Properties page.

Associated Data

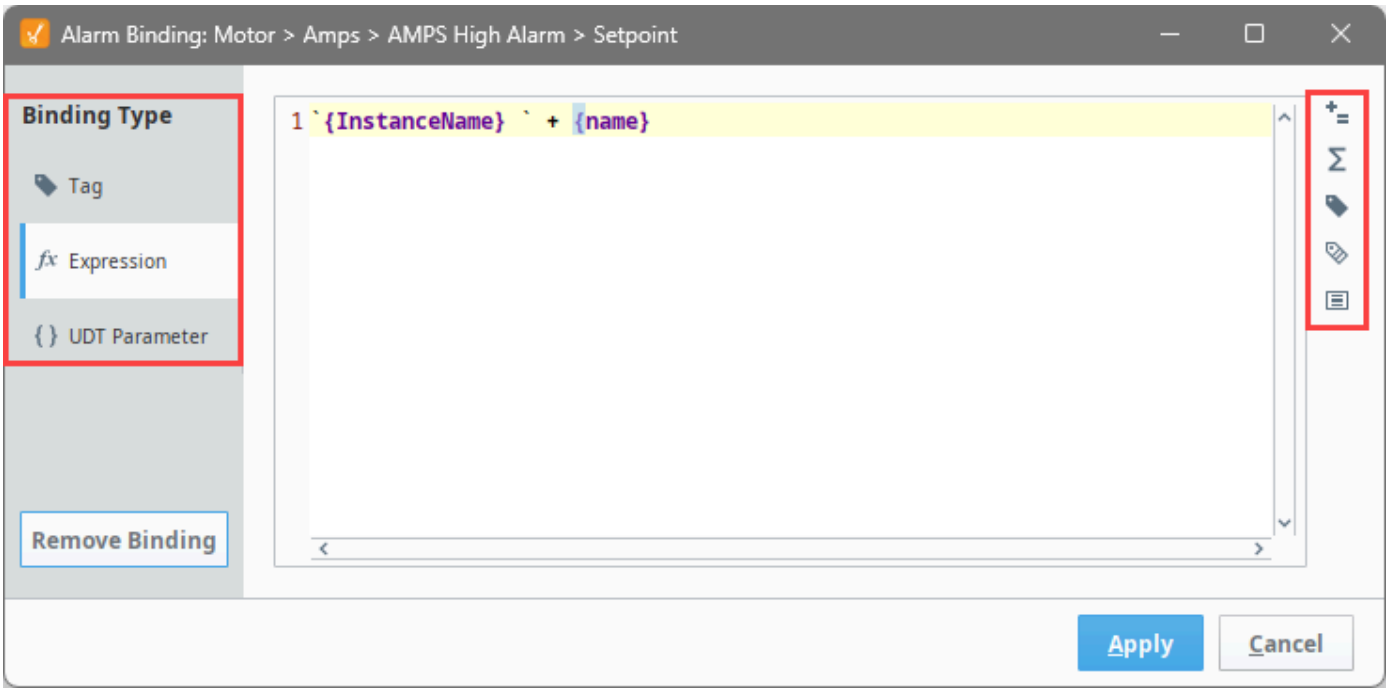
You can extend alarm properties by adding your own **Associated Data**, which are custom properties attached to a configured alarm. Values can be static or dynamic. Static properties are useful for filtering alarms; dynamic properties can be driven by another tag or evaluated with an expression. Associated Data values are attached to the alarm event, visible in **real time**, and recorded in the **Alarm Journal**.

Alarm Property Bindings

Many alarm properties are bindable and can be bound to other tags in the system, expressions, or UDT parameters. For example, you might bind the **Enabled** property to a tag representing whether your process is running, thereby disabling the alarm when production is stopped. Or, bind a setpoint to a tag that operators can adjust, allowing the setpoint to change at runtime.

This example shows how to bind an alarm property from the Tag Browser in Perspective.

1. Double-click a tag to open the **Tag Editor**.
2. In the **Categories** panel, select **Alarms**. If no alarms exist, first create one using the steps in **Configuring an Alarm on a Memory Tag**.
3. Select the alarm. The screen refreshes with all alarm properties.
4. Find the property you want to bind and click the binding icon to open the binding editor.
5. Select a binding type from **Tag**, **Expression**, or **UDT Parameter**. The image below shows an Expression binding. The expression can reference the tag's value and other alarm settings. Enter your expression.



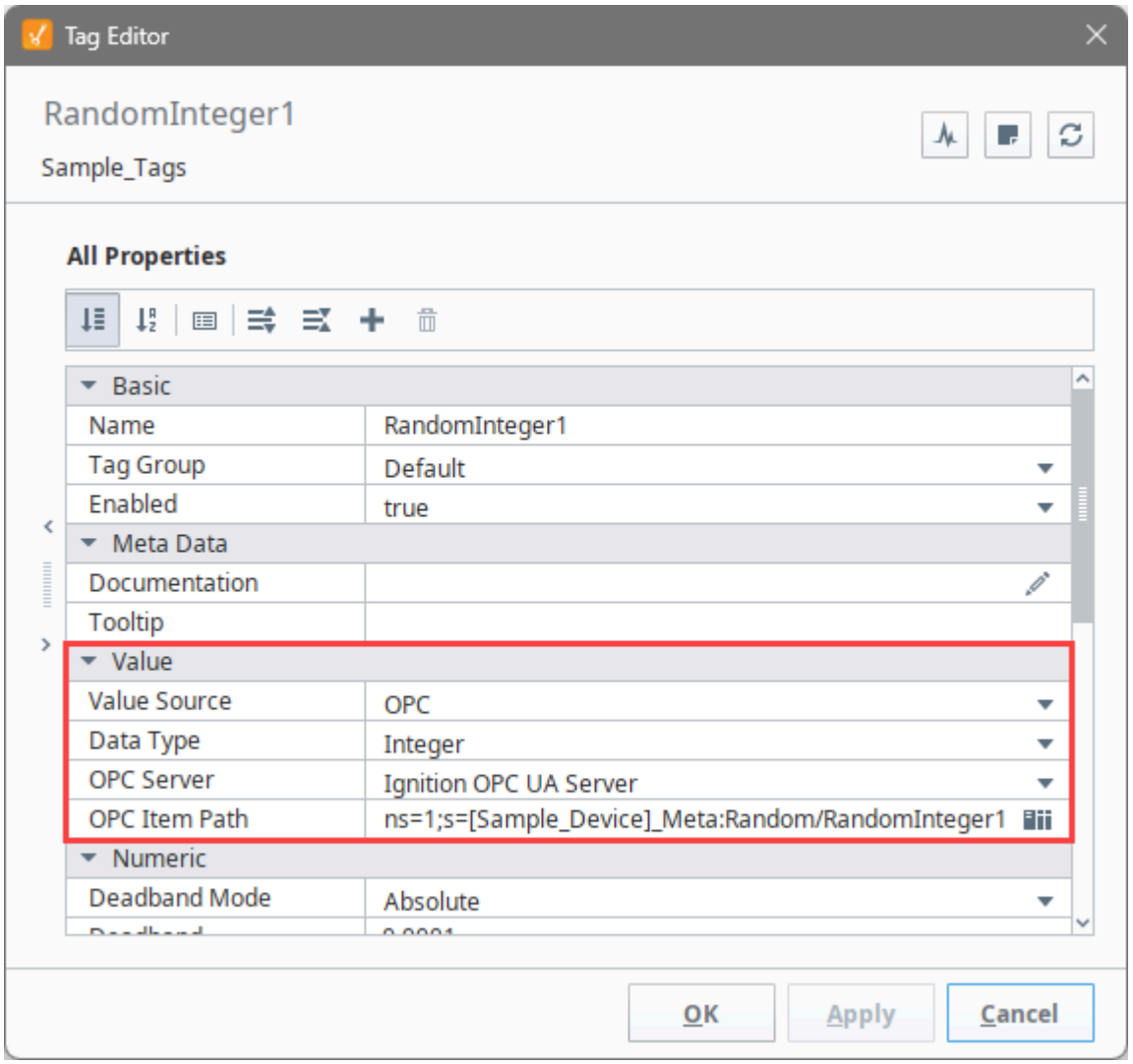
6. Once the binding is configured, click **Apply**.

7. Click **OK** to save the changes to the tag.

For more information on property bindings, see **Property Bindings in Perspective** and **Property Bindings in Vision**.

Alarms in Transaction Groups

Alarms can be added to OPC items in Transaction Groups, allowing alarms to be used without creating a tag in Ignition. An OPC item references a value on an OPC server through its OPC Item Path, the same path used when configuring an OPC tag with `OPC` as its **Value Source**.



In a Transaction Group, selecting an OPC item exposes an Alarming section in the item's properties. Adding and configuring an alarm on the item follows the same steps as adding an alarm to a tag.

Transferring System Tag Alarm Configurations

Although System tag configurations and alarms cannot be imported or exported through the Tag Browser, scripting can be used to extract them into a JSON format. The resulting `.json` file can be transferred to another system and used to configure the System tags there. See the **Exporting and Importing System Tag Alarm** page for examples.

Dynamic Alarm Attributes

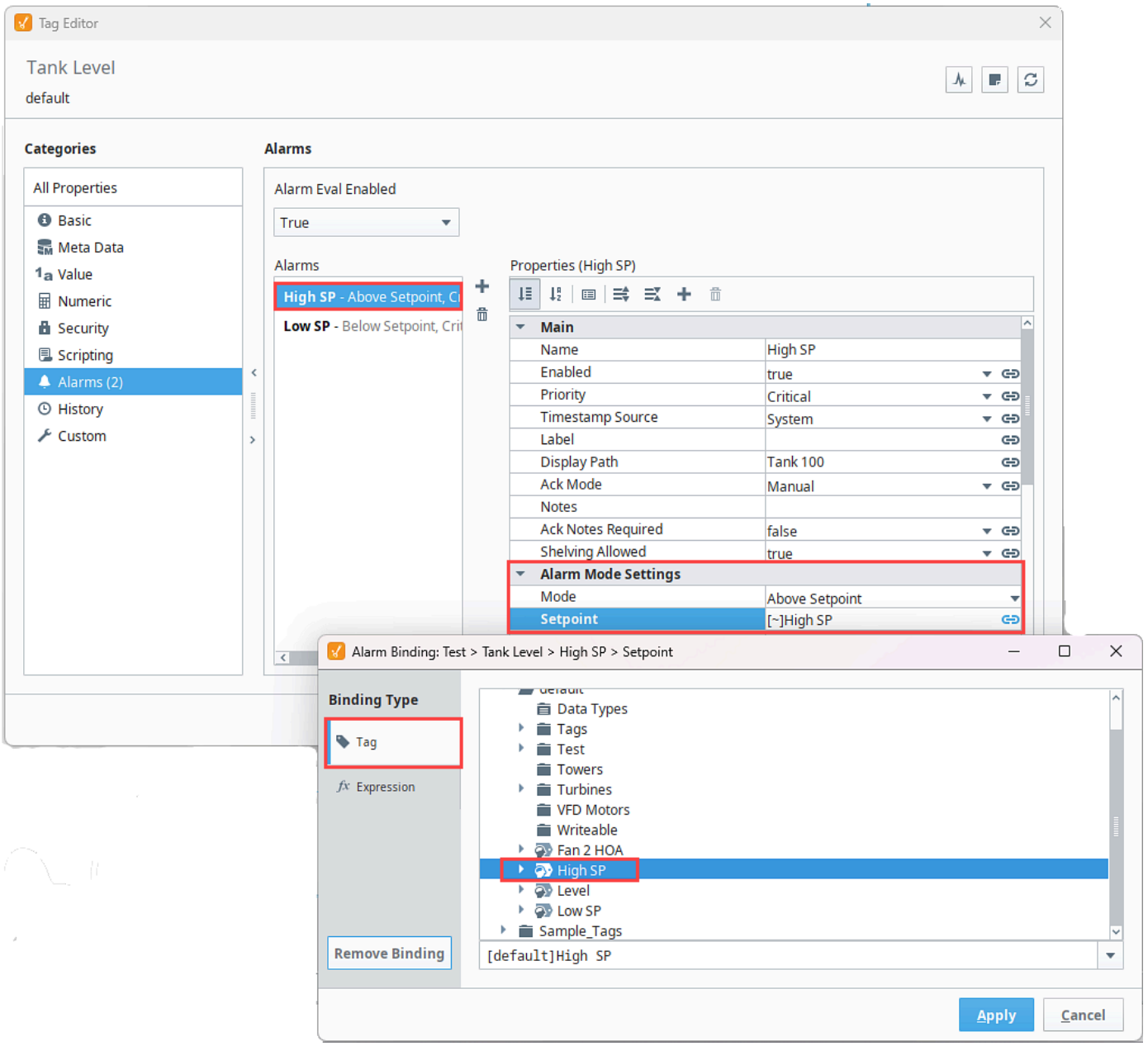
Dynamic Setpoints

Dynamic attributes allow alarm properties to change based on tag values or expressions. Using dynamic setpoints in alarming is useful in situations like when an operator needs to frequently change the high or low setpoints for an alarm evaluation. This way, the setpoint values based on the Tag or Expression bindings can be updated as needed, and the alarm evaluation will automatically update to match the adjusted values.

To Configure Dynamic Setpoints on an Alarm

Consider that you have a tank you'd like to configure an alarm for when the levels go beyond a high or low setpoint. Rather than hardcoding values in the Setpoint properties, you can bind alarms to tags in the system to make them dynamic. This example uses an OPC tag called **Tank Level**, and two Memory tags: a **High SP** tag with a value of 85 and a **Low SP** tag with a value of 20.

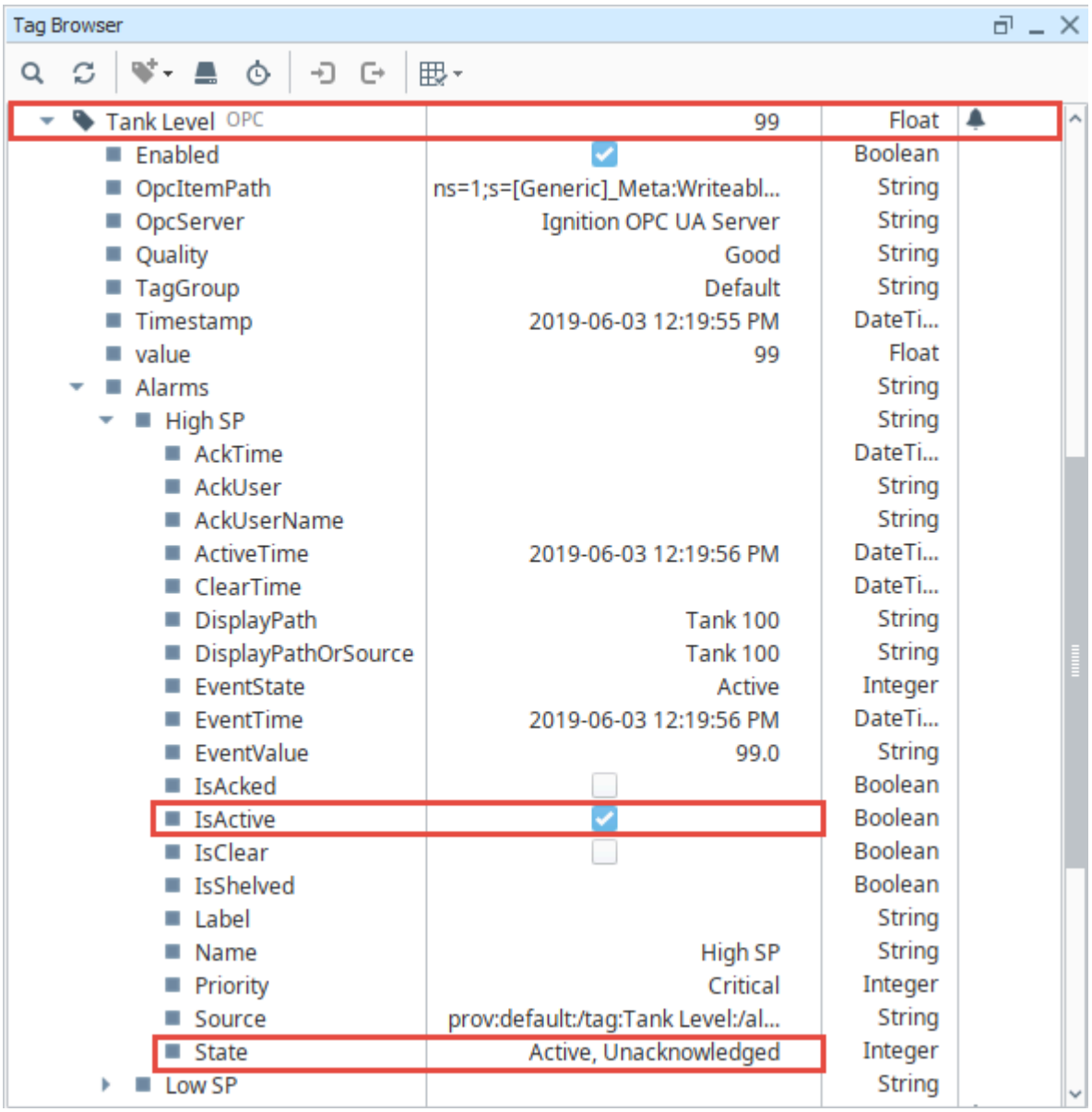
1. Double-click on the **Tank Level** tag to open the Tag Editor.
2. Select Alarms, and click on the **Add** + icon to create an alarm named **High SP**.
3. Under Alarm Mode Settings, set the Mode to **Above Setpoint**.
4. Click on the **Binding** icon for the Setpoint property.
5. Select the **Tag** tab and navigate the tag tree to bind the **Setpoint** property to the High SP tag.
6. Click **Apply**.



7. Repeat steps 2-6 using the Low SP tag to create a Low SP alarm.

8. Click **OK**.

Now, the tank alarm will be evaluated based upon the high and low setpoints of each tag. To test it, change the value of the Tank Level tag to 99, and you'll notice it activates the alarm.



Dynamic Enabling and Disabling

Enabling Based on Machine State

Allowing a dynamic condition to determine if an alarm is enabled or disabled is possible inside the alarm's properties, such as to indicate whether or not a machine is on. This example uses two tags: one for the alarm and one to determine the running state of the machine. The tag that will determine whether the machine is on or off will be named **Machine On** in this example and must be a boolean data type.

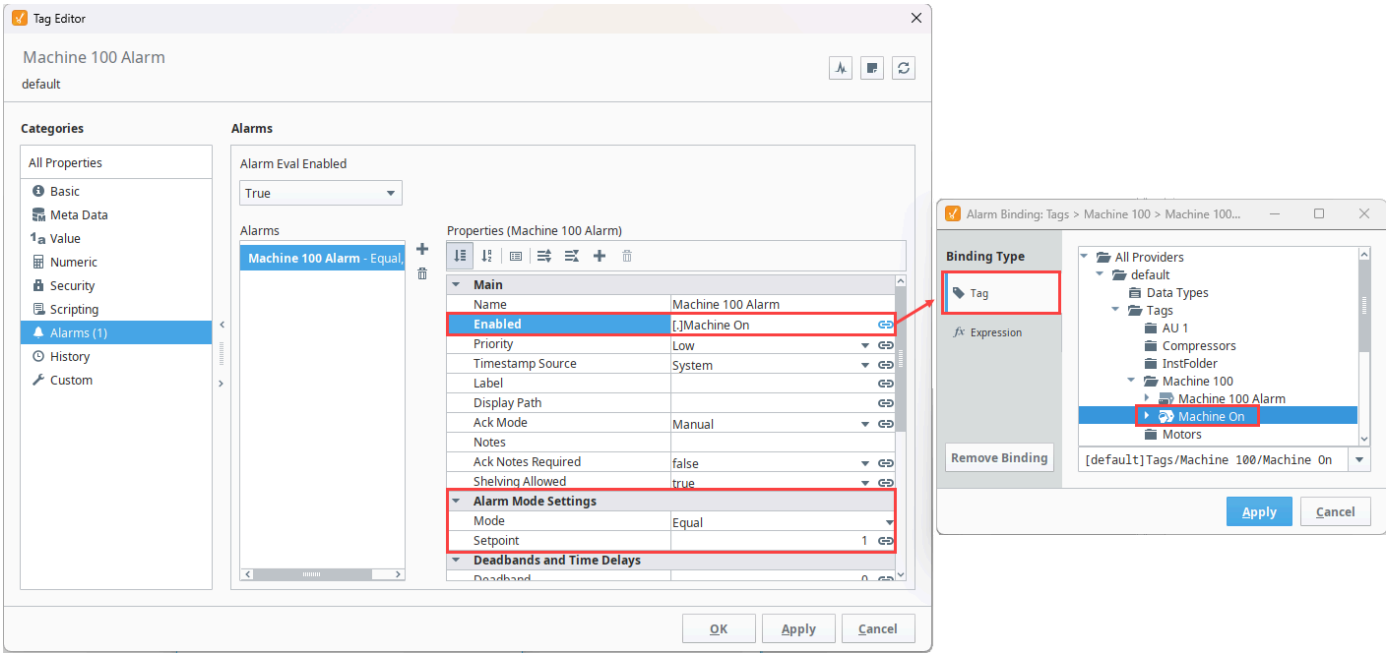
1. Select a tag that has the alarm you want to configure. The one we will use for this example is named **Machine 100 Alarm**.

2. Select Alarms, and click on the **Add** + icon to create an alarm named **Machine 100 Alarm**.

3. Click the **Binding** icon for the Enabled property and bind it to the **Machine On** tag.

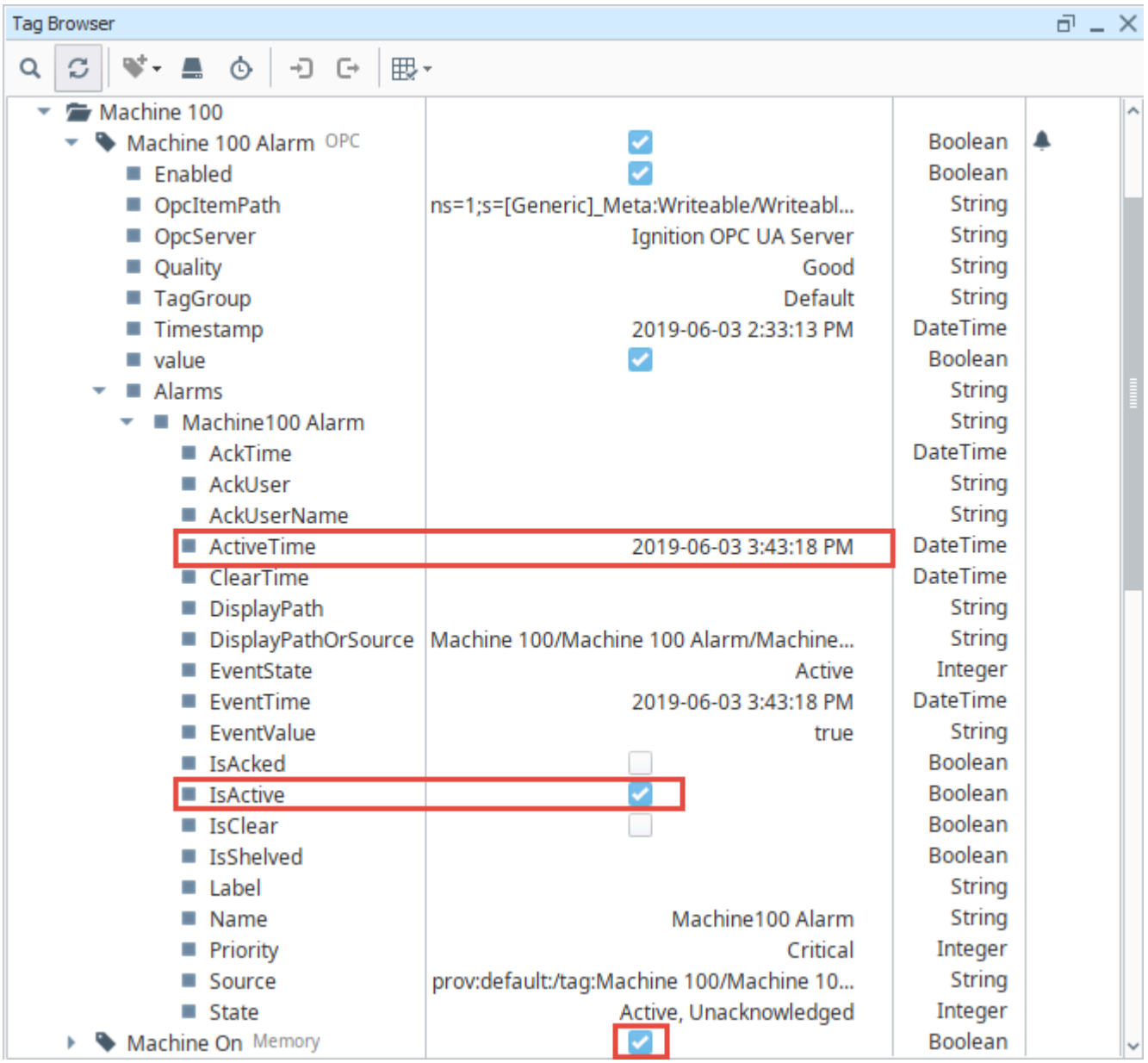
4. Set the Mode to `Equal` and the Setpoint to `1`.

5. Click **Apply**.



6. Click **OK** to save your changes.

Now, the alarm will only be evaluated based upon the state of the Machine On tag. Set the Machine to `true`. In the Tag Browser, expand your Machine 100 Alarm tag, and you'll notice the `IsActive` is now active.



Enabling Based on Time of Day

There may be occasions when you only want alarms to be evaluated at certain times of the day. This example demonstrates how to automatically enable and disable alarms for specific times of the day. Although we will use an expression here, this can also be achieved by binding the alarm's Enabled property to a tag.

1. Select a tag from the Tag Browser to configure the alarm. The one we will use for this example is named **Machine 100 Alarm**.

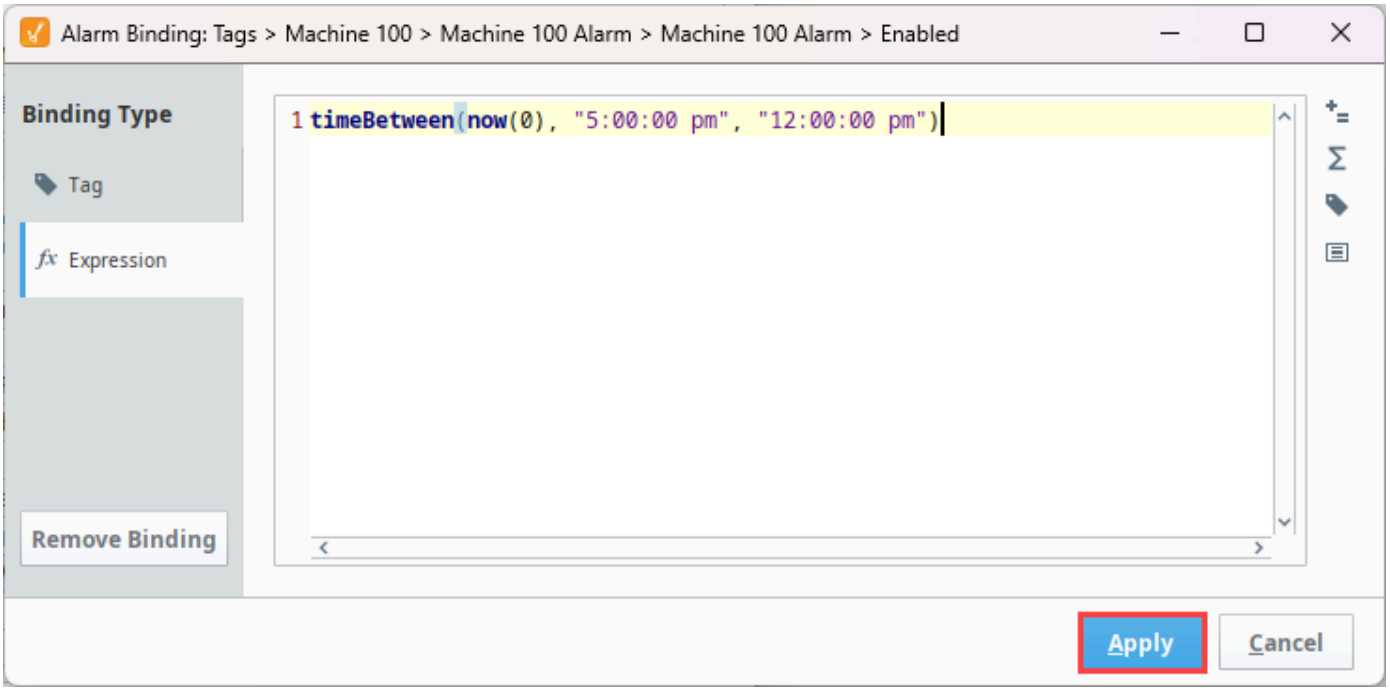
2. Select Alarms, and click on the **Add** + icon to create an alarm named **Machine 100 Alarm**.

3. Click the **Binding** icon for the Enabled property and select the **Expression** tab on the Alarm popup.

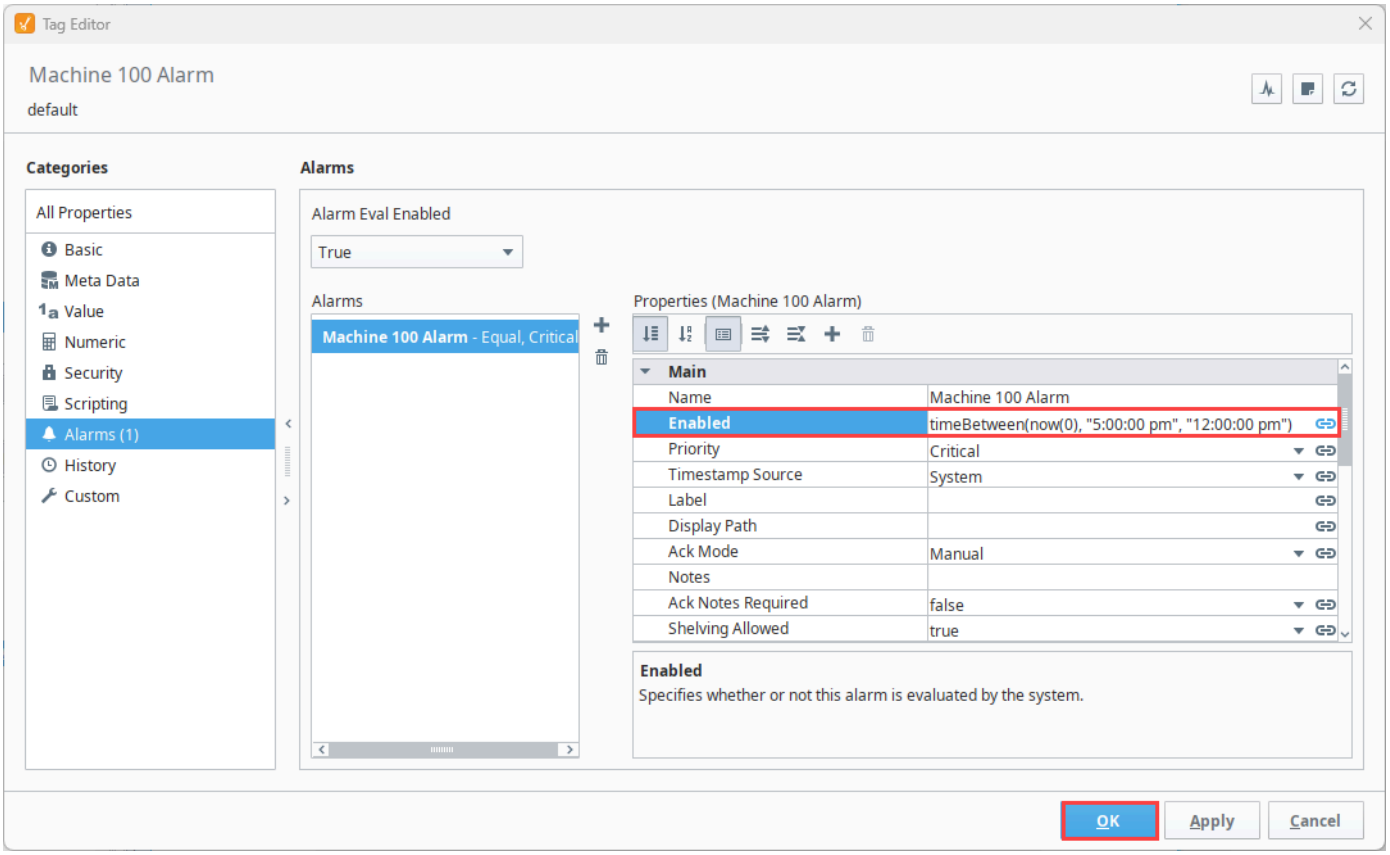
4. Enter the code block below to return a value of `true` if the time is between the hours of 5pm and 12pm, and `false` if it is not.

```
timeBetween(now(0), "5:00:00 pm", "12:00:00 pm")
```

5. Click **Apply**.



6. Click **OK** to save your edits.



The alarm will now be evaluated when the system clock falls between the time specified in the expression.

Alarms in UDTs

INDUCTIVE UNIVERSITY

Configure Alarm in UDT

Watch the video

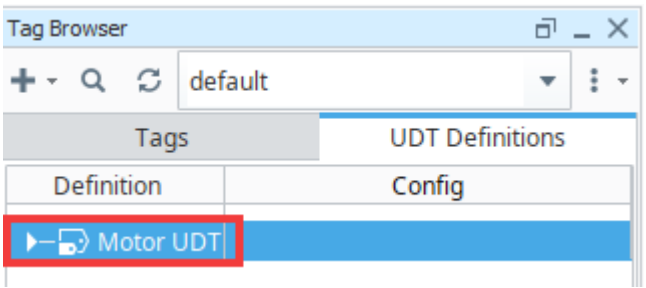
Configuring an Alarm on a UDT Member

When you configure the contents inside of a UDT definition, every instance of that UDT will automatically inherit that same configuration. This can be useful in cases where you need to create many tags, but don't want to manually create each tag one by one. The same concept works for alarms on a UDT. If an alarm is configured inside a UDT, every instance of that UDT will automatically have that same alarm configuration. Even if a new instance is created, it will automatically get that same alarm configuration.

Configuring an Alarm on a UDT

This example uses a Motor UDT. The Motor UDT contains two OPC tags: Amps and HOA. Let's configure an alarm on the Motor UDT when the Amps setpoint goes under 25.

1. In the **Tag Browser**, go to the **UDT Definitions** tab, and double-click on a UDT (i.e., Motor UDT) to edit the definition.

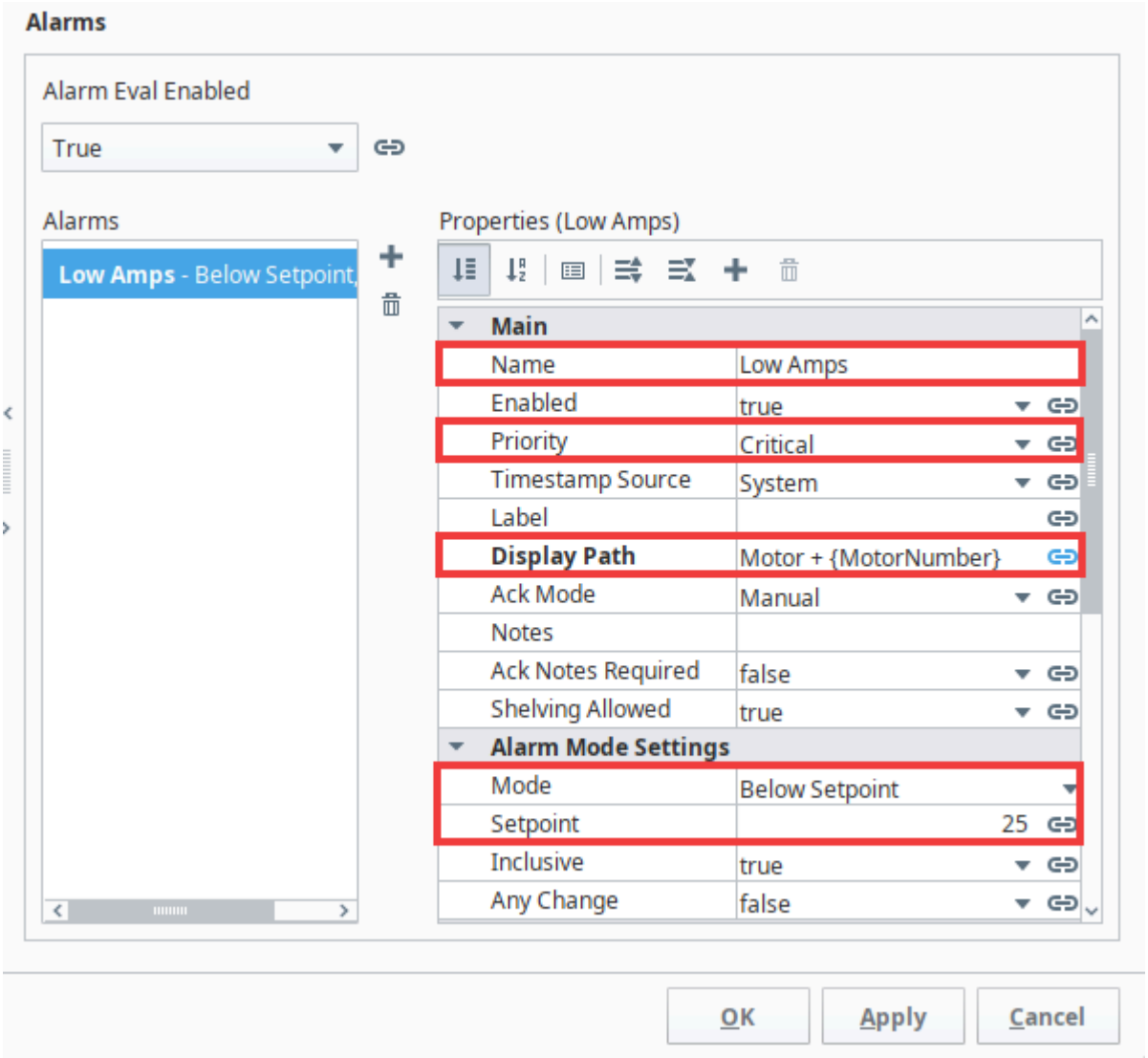


2. Double-click your tag (i.e., Amps), and scroll down to Alarms.

3. Click on the **Edit** icon next to your alarm to open the Alarm UI.

4. Click on the **Add** icon to create a new alarm.

5. Enter an **Alarm Name**, **Display Path**, **Priority**, **Mode**, and **Setpoint**. UDT configurations are set up the same way as the normal **alarm configuration on a tag**.



> **DISPLAY PATH**
>
> In this example, since you have multiple instances of a Motor, you need to make sure that the Display Path is unique for every instance. Otherwise, every instance of the UDT will have the same Display Path name, and the operator won't know which Motor alarm to respond to. There are a couple of ways to address this. You can leave the Display Path blank, in which case, Ignition will use the full Display Path to the instance of that tag, or you can bind a property to an expression involving a UDT parameter (i.e., `Motor` + `{MotorNumber}`).

6. Click **Apply** to save your alarm edits.

7. Click **OK** to save your Motor UDT edits.

Now, all instances of the UDT will have a similar alarm setup, but each alarm can be unique to that UDT instance by changing the **Below Setpoint** value.

Dynamic Setpoints in UDTs

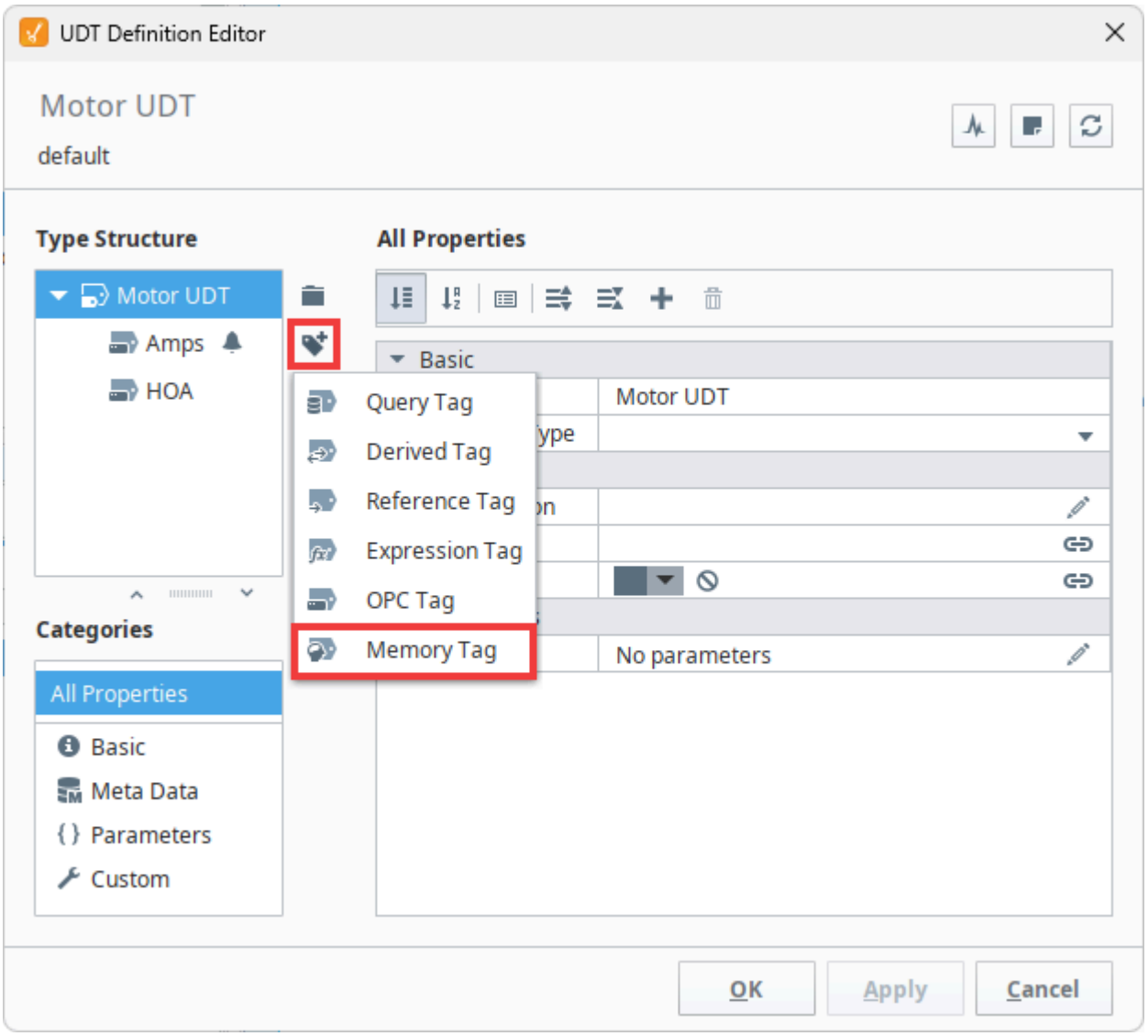
Instead of using a hardcoded setpoint, you can configure alarms with dynamic setpoints inside of a UDT definition. It is similar to configuring a UDT alarm, however, configuring a UDT alarm with a dynamic setpoint requires additional tags to serve as these setpoints in all the deployed UDTs.

Configuring Dynamic Setpoints Inside a UDT

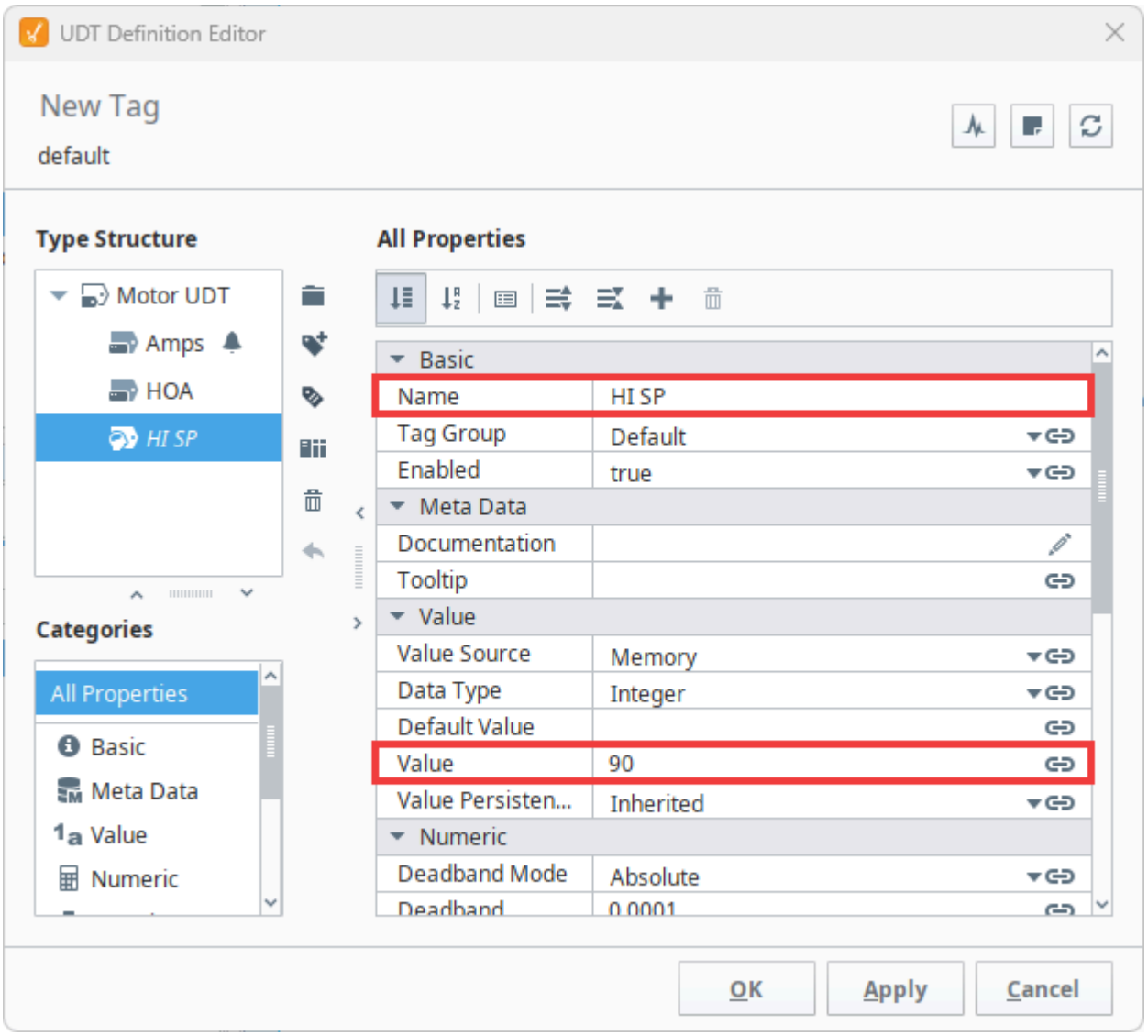
In this example, we'll use the Motor UDT in the above section to create a **Memory tag** inside the UDT definition to serve as the setpoint.

1. In the **Tag Browser**, go to the **UDT Definitions** tab, and double-click on your UDT to edit the definition.

2. Click the **Add Tag** icon and select **Memory Tag** from the dropdown.



3. Enter the **Name** of the tag (HI SP) and a **Value** (90).



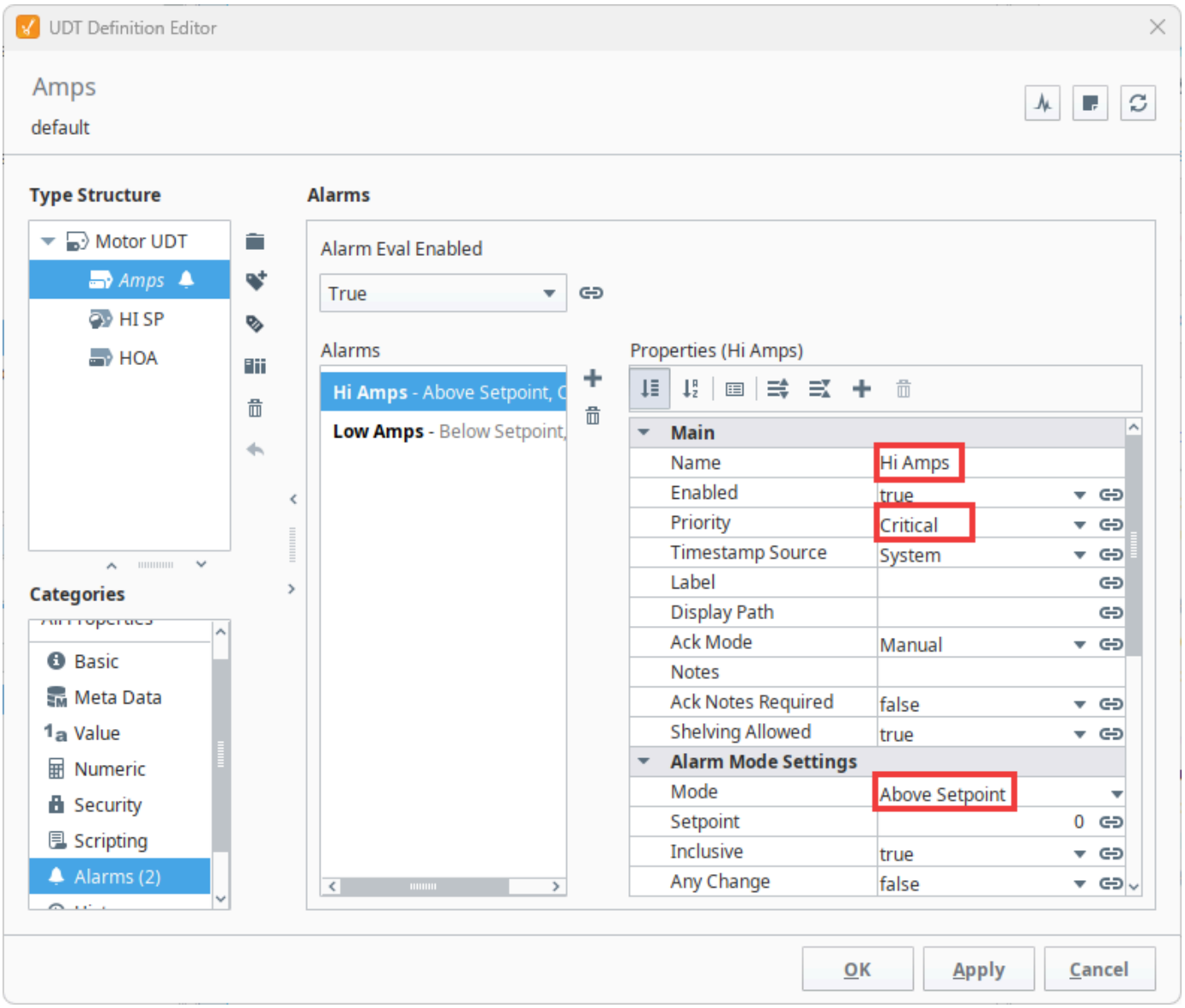
4. Click **Apply** to save your changes.

5. In the **Type Structure** area, select the tag you want to associate with the dynamic setpoint (i.e., Amps).

6. Under the Properties table, scroll down to Alarms. Click on the **Edit** icon next to Alarms.

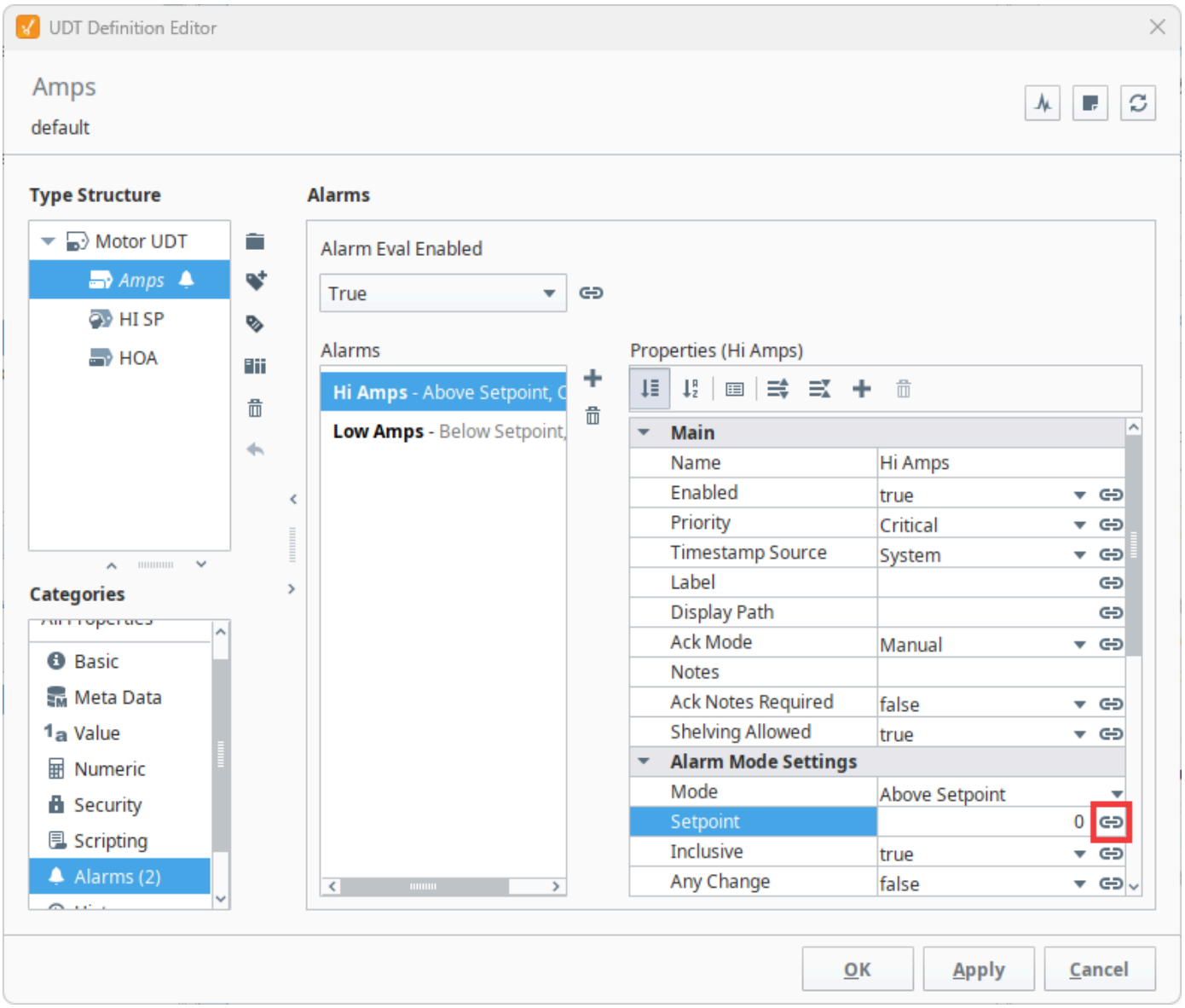
7. Click on the Add icon to create a new alarm for the high setpoint and name it **Hi Amps**.

8. Set the **Priority** as **Critical** and the **Alarm Mode** as **Above Setpoint**.

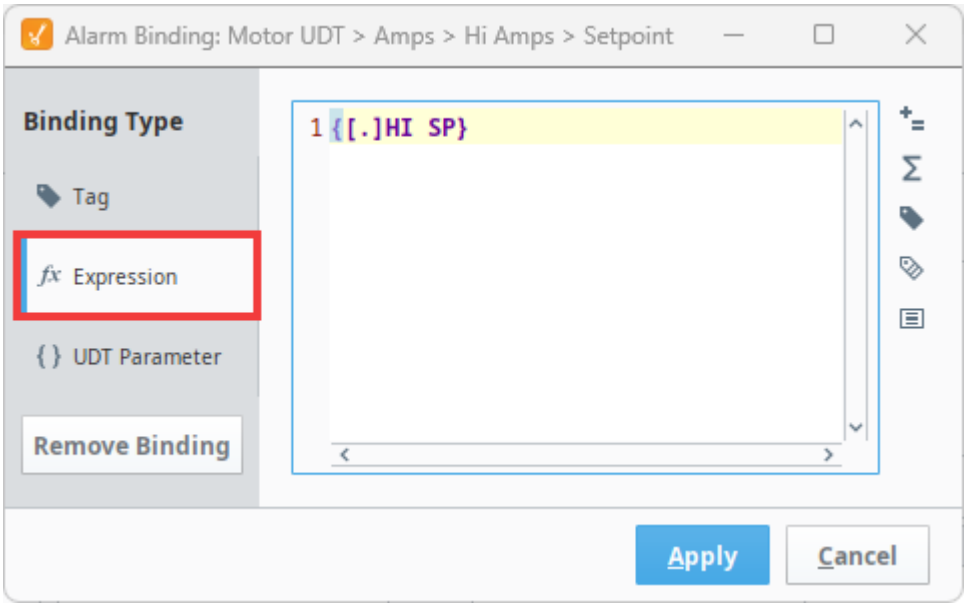


9. Click **Apply** to save your changes.

10. Bind the **Setpoint** property to the new UDT Memory tag (HI SP) by clicking on the **binding** icon.



11. You can either create an Expression binding or use a Tag binding to select the HI SP Memory tag. This example uses an expression as shown in the image below.



12. Click **Apply** to save the expression.

13. Click **Apply** again to save your alarm edits.

14. Click **OK** to save all your UDT updates.

Now you're ready to create instances of your UDT. All of the UDT instances will now have Memory tags that serve as alarm setpoints. You can test it out by dragging an Alarm Status Table component into a window and locating your alarm.

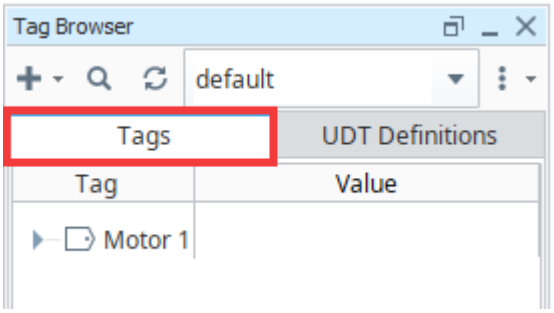
Overriding UDT Alarm Properties

After creating your UDT instances, there may be situations where you want to change some alarm properties for your UDT members. This process is similar to overriding inherited tag properties.

Configuring an Alarm Setpoint Override

This example will use the HI SP Memory tag configured in the section above and assumes you have already created instances of your UDT. To learn more about creating UDT instances, see **Creating an Instance**.

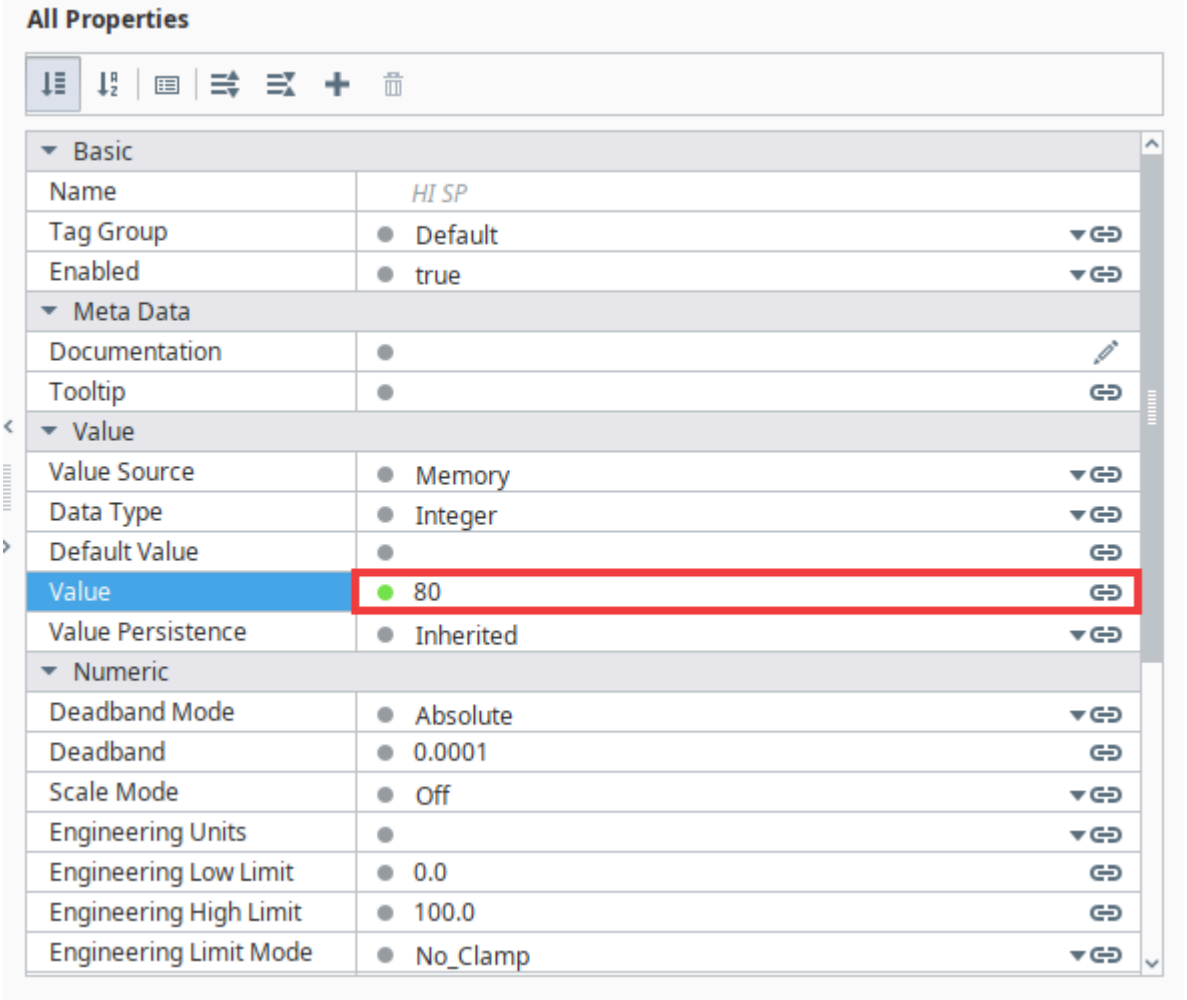
1. In the **Tag Browser**, go to the Tags tab and double-click your UDT instance to edit it. In this example, we will edit Motor 1.



2. In the **Type Structure** area, select the tag you want to override the properties for. We will chose **HI SP**.

3. Go down to the **Value** property and change it to the desired value. We will change our Value to 80.

4. Note that the gray circle next to the Value property has turned green. This indicates that Ignition detects an overridden value for that property.



5. Once you finish configuring your properties, click **Apply** and then **OK**.

Our Motor 1 instance is now using a different high setpoint than our other Motor instances. You can also confirm that the the HI SP tag has an override because of the half-filled circle in the Tag Browser ( ).

Alarm Associated Data

Associated data lets you attach additional context values to an alarm event. Every alarm in Ignition already carries built-in properties as it moves through Ignition, such as active time, clear time, who acknowledged it, acknowledgement time, priority, and display path. Associated data (also called custom properties) extends this list with values of your own.

Associated data values can be static or dynamic:

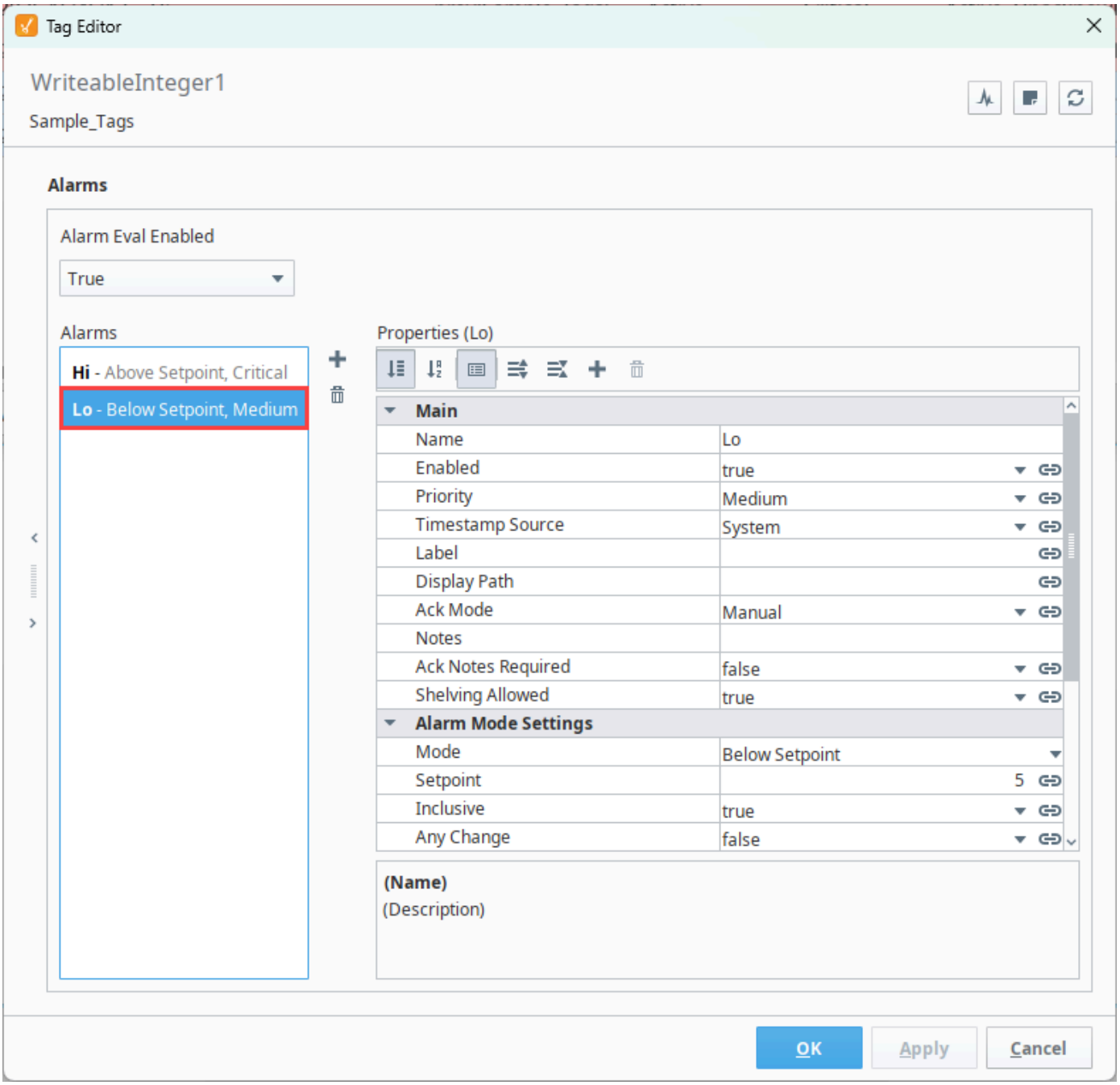
- **Static** values are fixed and work well for filtering alarms.
- **Dynamic** values are bound to a tag or expression and update accordingly.

Either way, the value is attached to the alarm event as it moves through Ignition and is available from the **Perspective Alarm Status Table**, the **Vision Alarm Status Table**, the **Alarm Journal**, and the **Alarm Notification** module.

Creating Associated Data

You can add associated data to any existing alarm, or **create a new alarm** first. The steps below add one static property and one dynamic property to an alarm. This example uses the `WriteableInteger1` OPC tag from the Sample_Tags provider and its built-in `Lo` alarm, but any alarm-capable tag will work. Ignition records associated data with each alarm event as it occurs, so add your properties before the events you want them to appear on; existing history is not updated retroactively.

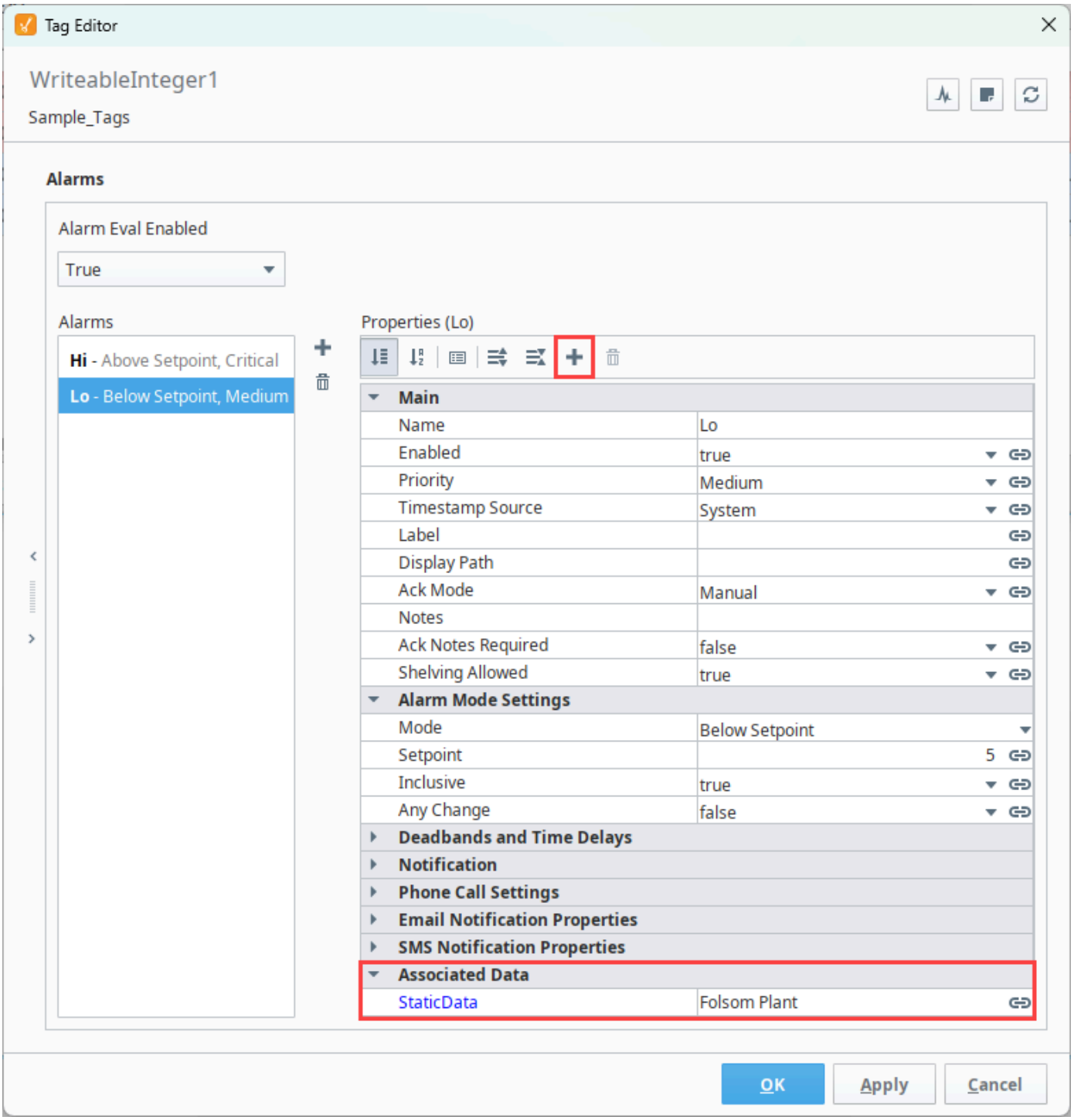
1. Right-click your tag and select **Edit Tag**.
2. In the **Tag Editor**, scroll to **Alarms** and click the **Edit** icon.
3. Select your alarm to open the alarm editor.



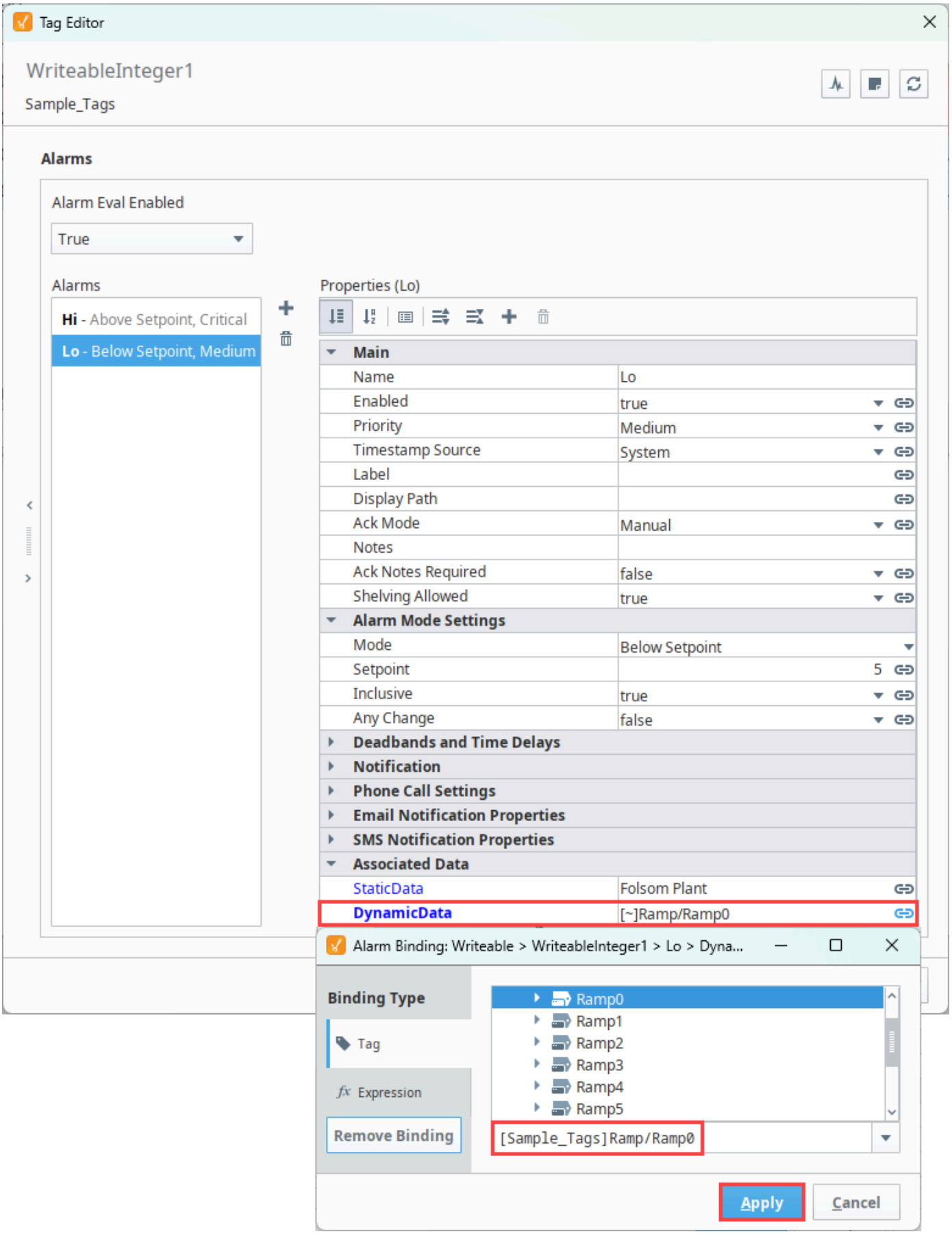
4. Click the **Add** + icon above the alarm properties. A new property named `New Data` is added at the bottom of the list.

5. Double-click the property name and rename it to `StaticData`.

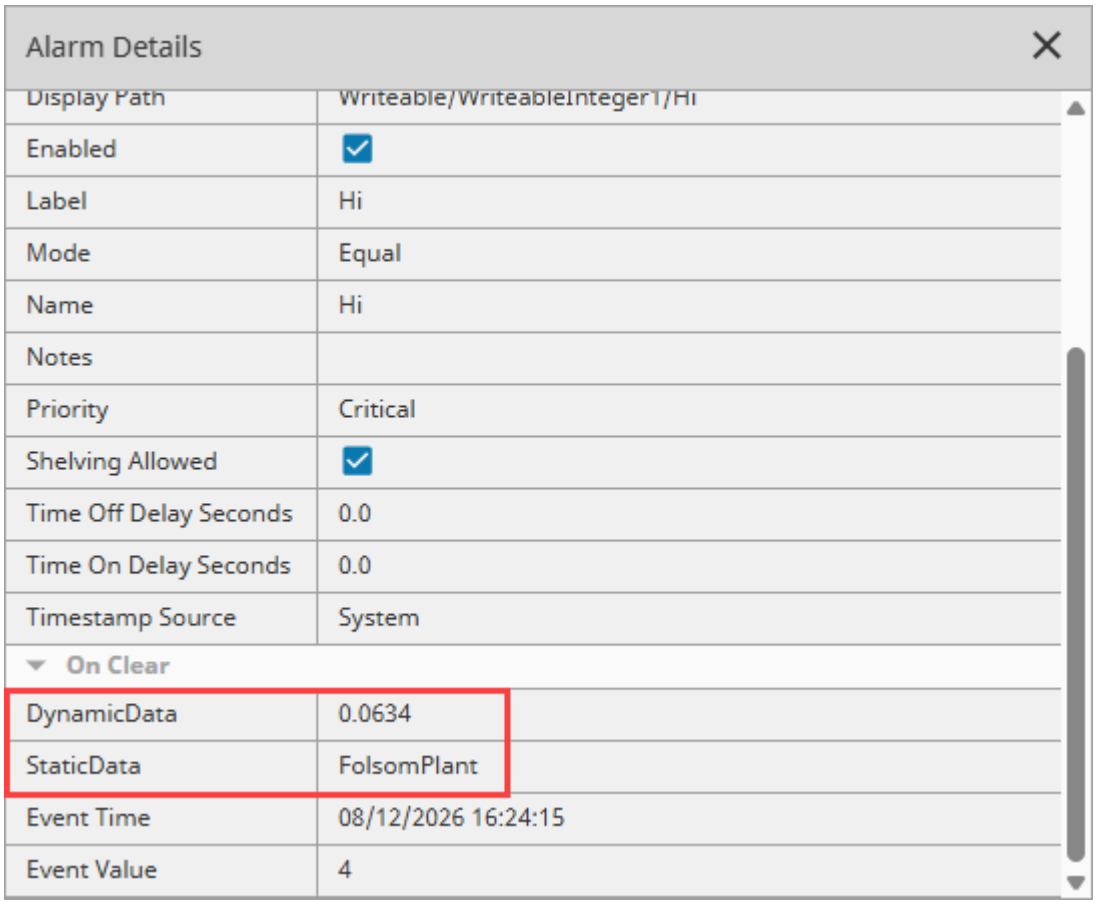
6. Set its value to `Folsom Plant` and press **Enter**.



7. Click the **Add** ✚ icon again to add a second property.
8. Rename the new `New Data` property to `DynamicData`.
9. Click the **Binding** 🔗 icon and bind the property to a tag. This example binds to the `Ramp0` OPC tag.
10. Click **Commit** to confirm the binding, then click **OK** to save the tag.



When the alarm goes active, Ignition stores the associated data values with the alarm event. To view them in the **Alarm Status Table**, select the alarm and click the gray arrow at the end of its row to open **Alarm Details**. The associated data appears under the **On Active** and **On Clear** sections, alongside the event properties.



Retrieving Associated Data in Scripts

Associated data values are always stored as strings. When you retrieve a value for use in an expression or a script, typecast it to the data type you need:

- In expression bindings, use these **type casting functions**.
- In Python scripting, use these **type casting functions**.

To read an associated data value from an alarm event in a script, call `alarmEvent.get("PropertyName")`, then cast the returned string to the type you need.

Alarm Grouping

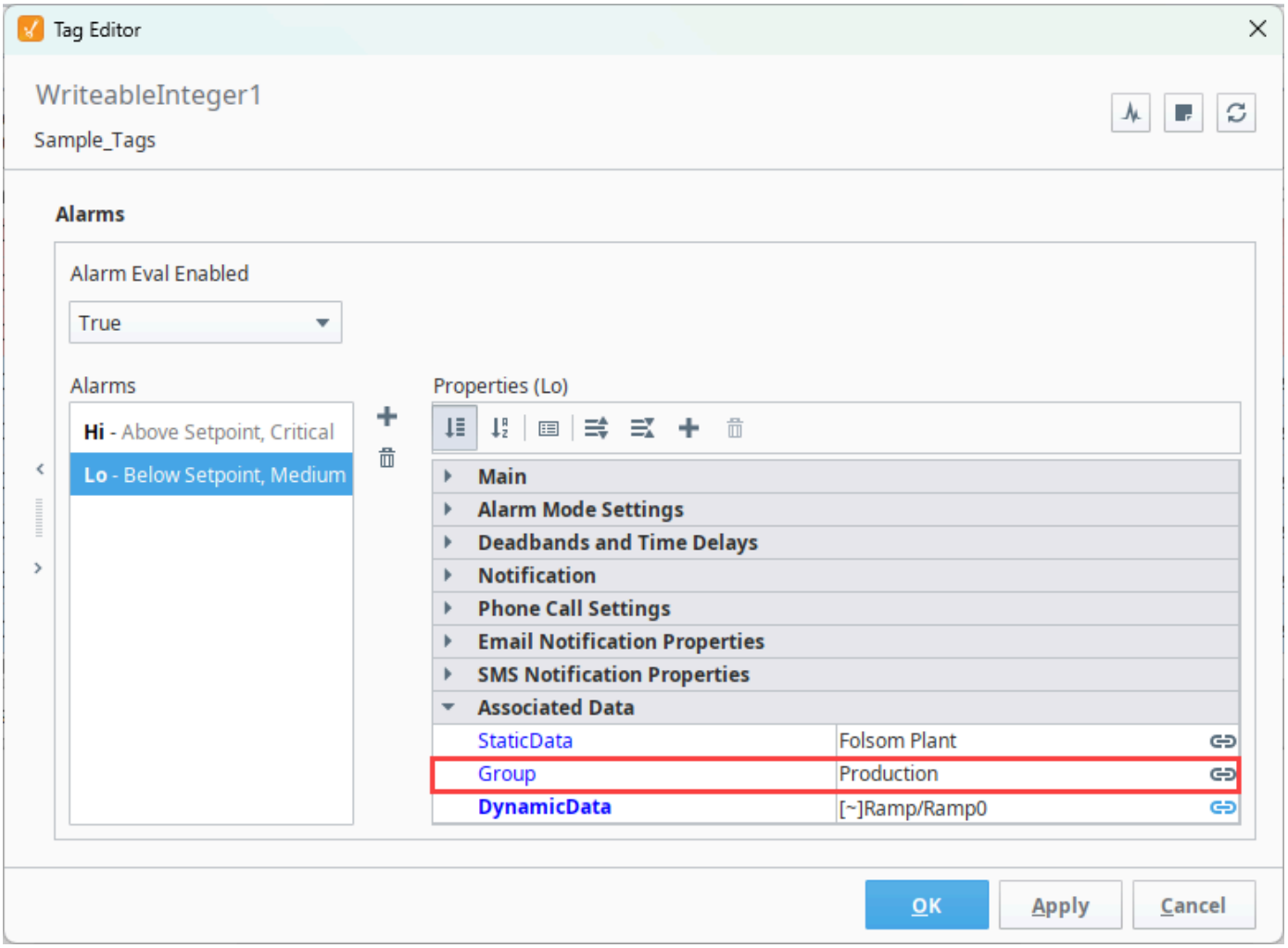
Grouping alarms is an important design concept. Instead of viewing every alarm in a single list, you often want to see alarms for a specific plant area or set of equipment.

There are several ways to group alarms: organizing tags into a folder hierarchy, using the **Display Path** field in the alarm configuration, or using associated data. Associated data is the most flexible and recommended approach, and it is common practice to base alarm groupings on it.

Creating an Alarm Grouping

This example reuses the `WriteableInteger1` tag from the previous example and adds a `Group` property that can be used for filtering and notification.

1. Right-click your tag and select **Edit Tag**.
2. In the **Tag Editor**, scroll to **Alarms**, click the **Edit** icon, and select your alarm.
3. Click the **Add** icon above the alarm properties.
4. Rename the new `New Data` property to `Group` and set a value that represents the group, such as `Production`.



5. Click **Commit** to save your alarm edits, then click **OK**.

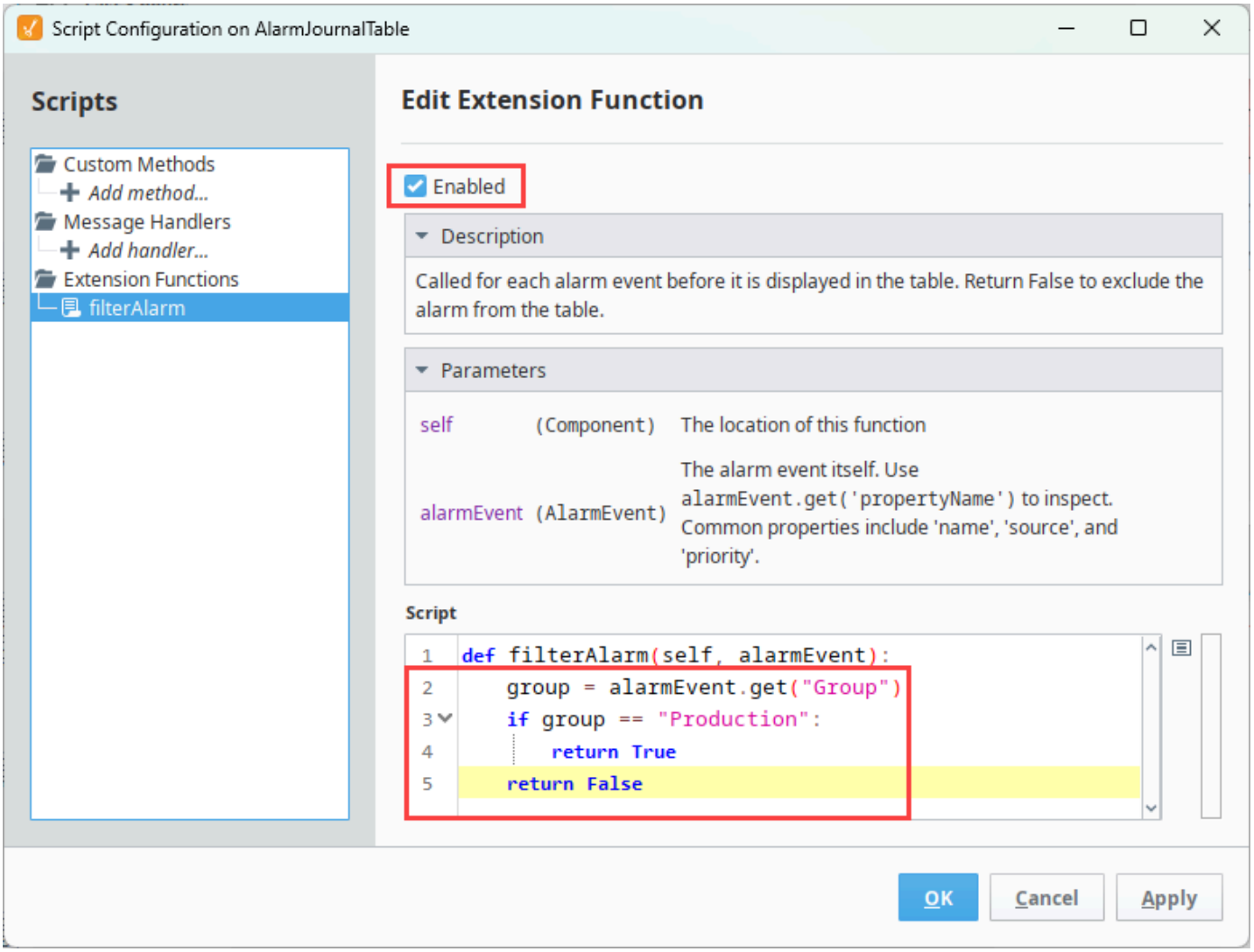
The alarm now always carries the `Group` designation, which you can use for filtering or in alarm pipeline notifications.

To filter the Alarm Journal on this new property, enable and edit the `filterAlarm` extension function:

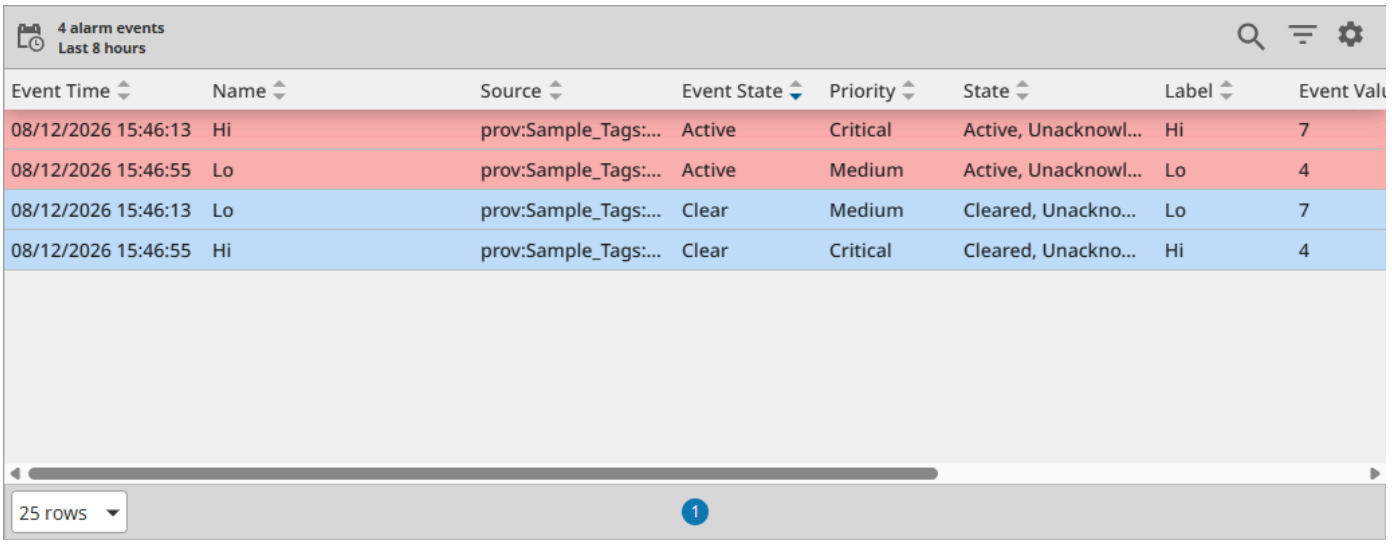
1. Right-click the **Alarm Journal** component and select **Configure Scripts...**.
2. Under **Extension Functions**, click **filterAlarm** and select the **Enabled** checkbox.
3. Edit the script to check your associated data property and value. For example:

filterAlarm scripting function

```
group = alarmEvent.get("Group")
if group == "Production":
    return True
return False
```



4. Open the **Alarm Journal Table** component and filter on your associated data. The table now displays only the alarms associated with the `Production` group.


4 alarm events
Last 8 hours
Event Time
Name
Source
Event State
Priority
State
Label
Event Valu
08/12/2026 15:46:13 Hi prov:Sample_Tags:... Active Critical Active, Unacknowl... Hi 7
08/12/2026 15:46:55 Lo prov:Sample_Tags:... Active Medium Active, Unacknowl... Lo 4
08/12/2026 15:46:13 Lo prov:Sample_Tags:... Clear Medium Cleared, Unackno... Lo 7
08/12/2026 15:46:55 Hi prov:Sample_Tags:... Clear Critical Cleared, Unackno... Hi 4
25 rows
1

Alarm Event Properties Reference

An alarm event is made up of many pieces of information. The state, value, time, and configuration data for the alarm are collectively known as the properties of the event. Alarm event properties fall into two categories:

- Alarm event properties, which are created when an alarm becomes active, cleared, or acknowledged.
- Alarm runtime properties, which exist only while the alarm event is in memory.

You can extend either category with your own associated data, which can be configured on any alarm that already exists in a project. To learn more, see **Alarm Associated Data**.

For the properties you set when configuring an alarm, see the **Tag Alarm Properties** page. For the alarm settings that apply to the entire Gateway, see the **Gateway General Alarm Properties** page.

Alarm event properties define how an alarm behaves, and you can reference them in:

- Messages, such as the body of an email or SMS message.
- Expressions, such as the binding of a different property, or an Expression block.
- Alarm pipelines, where they can also be created as temporary variables, such as a counter.
- Scripts, such as a Script block in an alarm pipeline.

Reference a property by wrapping its binding or scripting name in curly brackets. For example, `{name}` returns the name of the alarm, and `{eventValue}` returns the value associated with the event. Expressions can also retrieve a property with the `getProperty()` function.

Properties are always referenced the same way, but the value you get back depends on where that value comes from, and how long it lasts. An alarm event holds active event properties, clear event properties, acknowledged event properties, and runtime properties. The active, clear, and acknowledged properties are created when that type of event occurs, so the same property can exist more than once within a single alarm event. A bound configuration property or associated data, for example, is captured on both the active and clear events. When you reference a property, the alarm event returns the most recent value, so referencing a bound property named MyData while the alarm is active can return a different result than referencing it after the alarm clears. The **Alarm Journal** stores the individual values separately.

Alarm Event Properties

Event Properties	Binding/Scripting Name	Description	Datatype
Name	name	The name of the Alarm.	String
Enabled	enabled	Specifies whether or not this alarm is evaluated by the system. Set to False to turn off the alarm state and all associated actions.	Boolean
Priority	priority	The priority (or severity) of the alarm. Used for sorting/filtering. Numerical values are associated with each priority to make comparison easier. This property can also be referenced as a string with the following priority names. • Diagnostic (0) • Low (1) • Medium (2) • High (3) • Critical (4)	Integer or String
Display Path	displayPath	The unique path of the Alarm state. Used for display and browsing purposes.	String
Active Pipeline	activePipeline	The pipeline (if any) that will be used to process active events generated by the alarm.	String
Clear Pipeline	clearPipeline	The pipeline (if any) that will be used to clear events generated by the alarm. Used when the alarm goes into the Clear State.	String

Event Properties	Binding/Scripting Name	Description	Datatype
Active Delay	timeOnDelaySeconds	The amount of consecutive seconds that the alarm state must be True before the tag enters this alarm state.	Double
Clear Delay	timeOffDelaySeconds	The amount of consecutive seconds that the alarm state must be False before the tag exits this alarm state.	Double
Notes	notes	Free-form notes for the alarm state.	String

Alarm Runtime Properties

Runtime properties exist only while the alarm event is live, meaning it has not yet cleared or been acknowledged. They are not stored in the Alarm Journal by default. In addition to the properties you create through the Set Property block, the system defines the runtime properties below. The system uses these internally, but they are still regular properties, so you can access and modify them through the normal means.

Runtime Property	Description	Datatype
IsInitialEvent	Set to true when the event is caused by the initial state of the alarm.	Boolean
SystemAck	Set to true when the alarm has been acknowledged by the system, due to an overflow of the live event queue. Live events are alarm events that are active or not acknowledged, and are limited for each alarm by the general alarm settings.	Boolean
ShelfExpiration	When the shelf will expire for this event.	Integer
IsShelved	Whether the alarm is currently shelved.	Boolean
EventCanceled	If set, the event will drop out as soon as possible from the pipelines.	Boolean
EventId	The unique id (uuid) of this alarm event. Each event gets a completely unique id.	String
Source or Source Path	The qualified path to the item that generated this event. Includes the tag provider, tag path, and the name of the alarm. Example: prov:tagProviderName:/tag:folder/tagName:/alm:alarmName	String
DisplayPathOrSource	Gets the display path if defined, otherwise, returns the source.	String
State	The current overall state of the alarm. States include: • Clear and Unacked (0) • Clear and Acked (1) • Active and Unacked (2) • Active and Acked (3)	Integer
EventState	The transitional state that caused the current event. States include: • Active (0)	Integer

Runtime Property	Description	Datatype
	• Clear (1) • Acknowledged (2)	
EventValue	The value associated with the current event.	Integer
AckUser	The user who acknowledged this event.	String
IsAcked	True if the event has been acknowledged.	Boolean
IsActive	True if the event is still active.	Boolean
IsClear	True if the event is not still active.	Boolean
ActiveTime, ClearTime, AckTime	The timestamp for each event.	Date
PipelineTransitionCount	How many transitions the event has made inside of the pipelines.	Integer

The PyAlarmEvent Object

Some system functions, such as **system.alarm.queryStatus()**, return PyAlarmEvent objects. Each PyAlarmEvent object contains methods that retrieve additional information about an individual alarm.

Many of these methods return a complex object instead of a standard Python datatype. In these cases, normal Python type casting turns the object into a native Python type. For example, `getDisplayPath()` returns a `StringPath` object, which the `str()` function converts to a Python string.

The following example prints the display path, state, and acknowledgement status of each active alarm:

Read Properties from Active Alarms

```
results = system.alarm.queryStatus(state=["ActiveUnacked", "ActiveAcked"])

for alarm in results:
    print str(alarm.getDisplayPath()), alarm.getState(), alarm.isAcked()
```

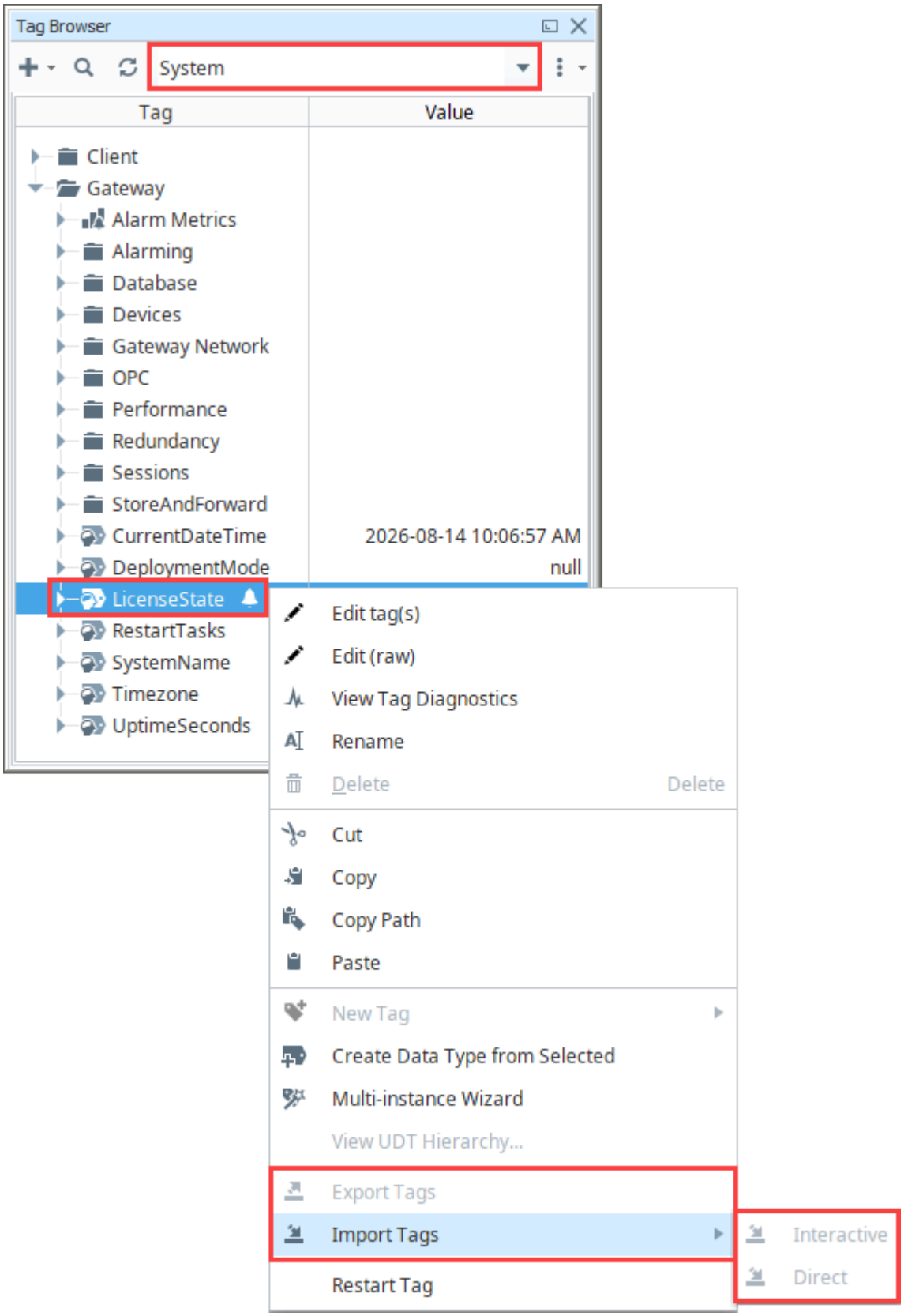
Function	Description	Returned Object
getDisplayPath	Returns the Display Path of the alarm.	StringPath
getDisplayPathOrSource	If the Display Path for the alarm is in an empty string, then returns the Source Path, otherwise, returns the Display Path	Python Unicode
getId	Returns the UUID of the alarm event.	java.util.UUID
getLastEventState	Returns the last state of the alarm. Possible return values are **Active**, **Acknowledged**, or **Cleared**. The last event is always returned, so if you're looking to see if an alarm is has been both Acknowledged and Cleared, use `getState()` instead.	AlarmStateTransition
getName	Returns the name of the alarm.	Python Unicode
getNotes	Returns the value of the Notes property. Returns a None-type object if a Note has not been configured on the Alarm.	Python Unicode or None
getPriority	Returns an AlarmPriority object representing the Priority of the alarm. Can easily be converted to an Integer with .intValue. Default is String.	AlarmPriority
getSource	Returns the Source path of the alarm.	QualifiedPath
getState	Returns the current state of the alarm: i.e., Active, Unacknowledged	AlarmState
isAcked	Returns a boolean flag indicating that the alarm has been Acknowledged.	Python Boolean
isCleared	Returns a boolean flag indicating that the alarm has been Cleared.	Python Boolean

Function	Description	Returned Object
isShelved	Returns a boolean flag indicating that the alarm has been Shelved.	Python Boolean

For the full list of methods, including the methods available to Script blocks in alarm pipelines, see the **PyAlarmEvent** section of the Scripting Object Reference page.

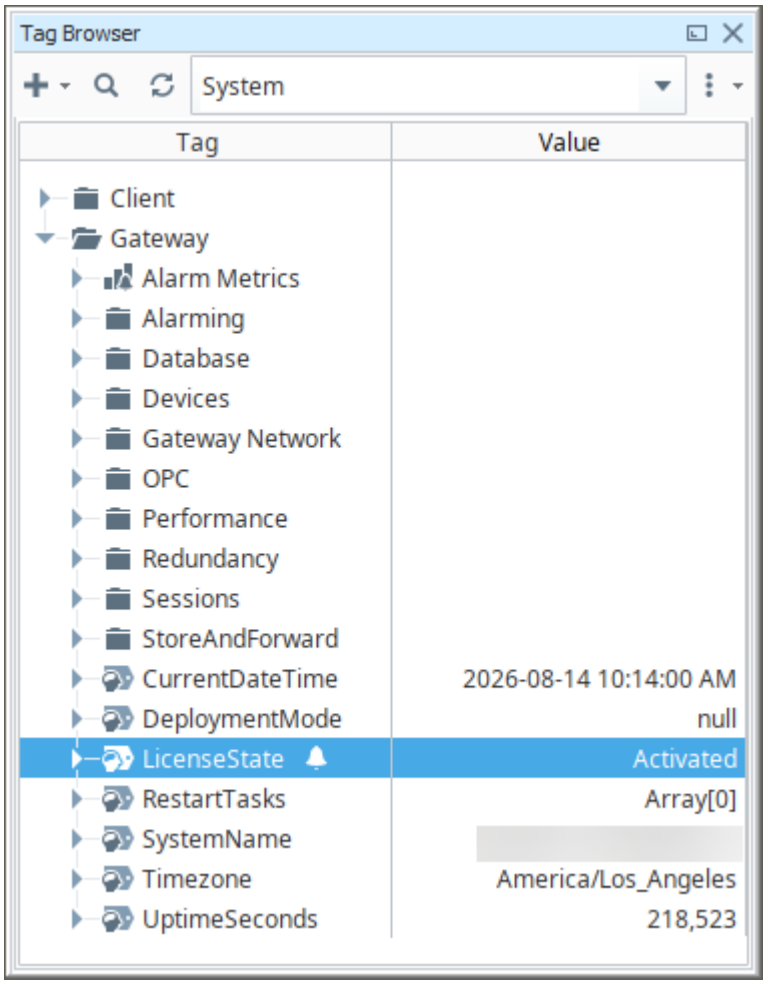
Exporting and Importing System Tag Configurations

There may be times when you want to export or import the configuration of a **System tag**. Unlike regular tags from a tag provider, the built-in import and export functions are disabled for System tags.



Instead, you can use scripting functions to **import** or **export** the configurations you want. This is useful when migrating a Gateway to new hardware, backing up and restoring alarm configurations, or replicating settings across development, testing, and production environments.

Each example below uses a single alarm configured on the **LicenseState** tag, found in the Tag Browser under [System] > Gateway > LicenseState.

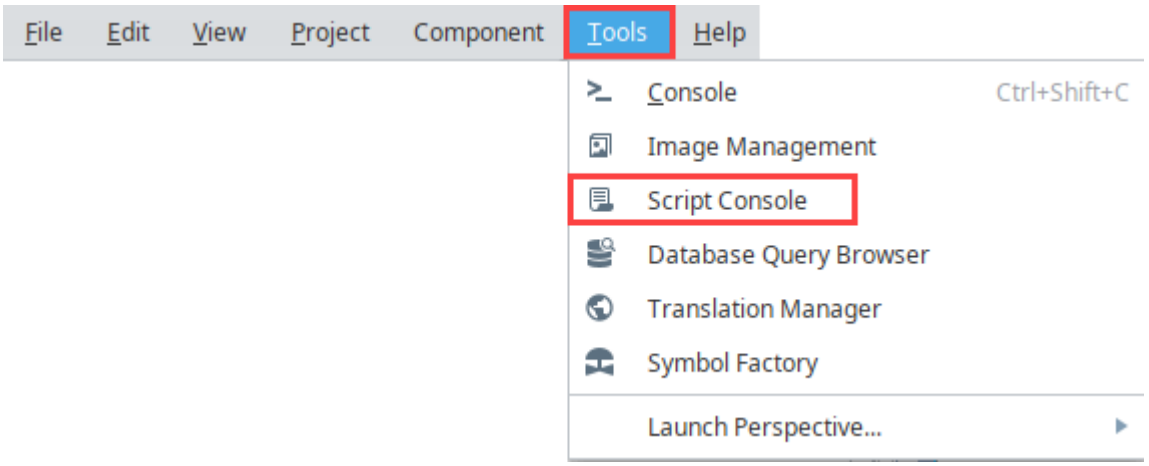


Exporting System Tag Configurations

Exporting Tags in the System Provider

This example exports the configurations for all the System tags in the System provider. To export a single tag instead, see **Exporting a Targeted System Tag**.

1. Open the **Script Console** in the Ignition Designer.



2. Use the **system.tag.exportTags()** function script below to export your System tags, replacing the `filePath` parameter with the path you want to export to:

Export System Tags Script

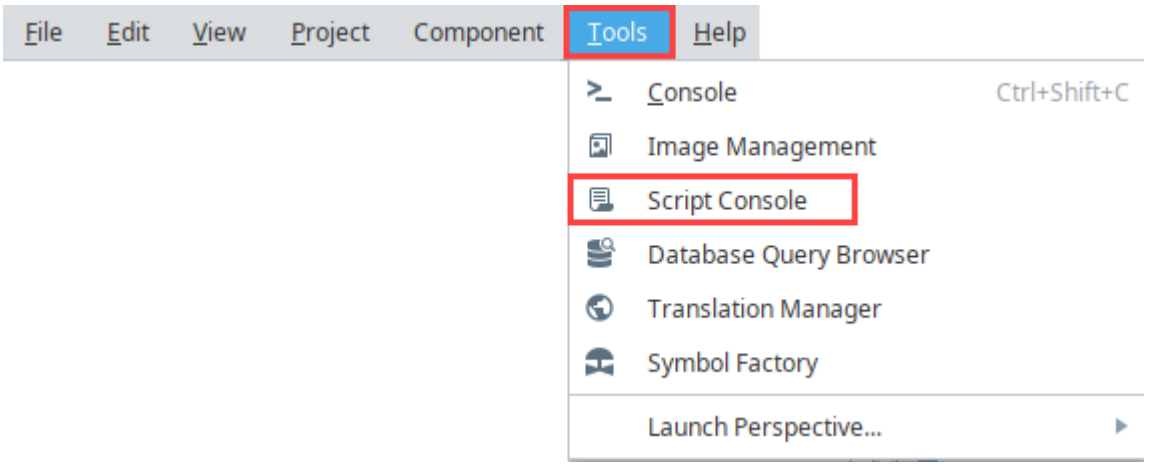
```
filePath = "C:\\Users\\myUser\\Desktop\\mySystemTags\\allSystemTags.json"
tagPaths = ["[System]"]

system.tag.exportTags(filePath, tagPaths)
```

The export creates a JSON file named **allSystemTags.json** at the specified path.

Exporting a Targeted System Tag

You can also export a single System tag or a group of tags. This example exports only the LicenseState tag.

1. Open the **Script Console** in the Ignition Designer.



2. Use the **system.tag.exportTags()** function script below to export the LicenseState tag, replacing the `filePath` parameter with the path you want to export to:

Export LicenseState Tag Script

```
filePath = "C:\\Users\\myUser\\Desktop\\mySystemTags\\targetedSystemTag.json"
tagPaths = ["[System]Gateway/LicenseState"]

system.tag.exportTags(filePath, tagPaths)
```

The export creates a JSON file named **targetedSystemTag.json** at the specified path.

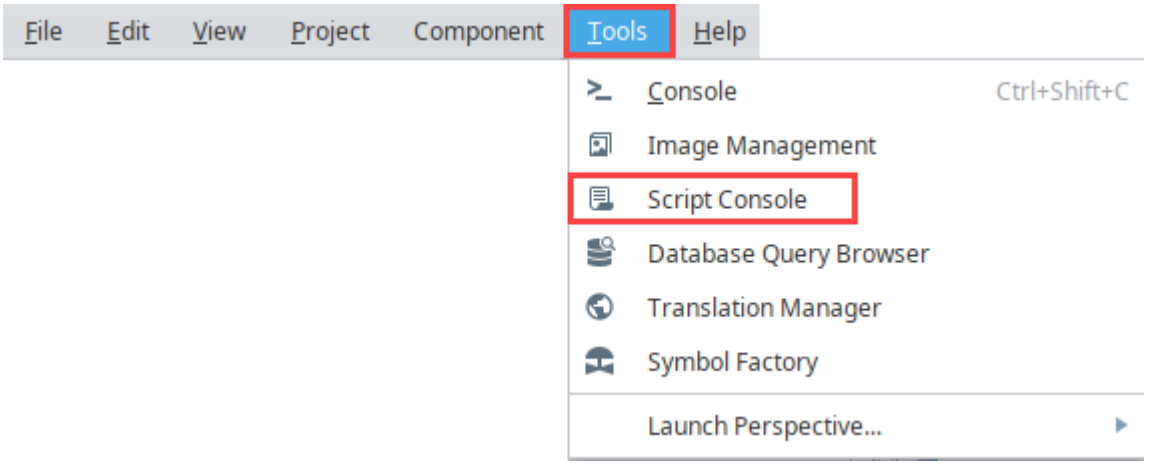
Importing System Tag Configurations

Importing a configuration overwrites the existing settings on any matching System tags. The examples below use the JSON files created in the export examples, so make sure the file is accessible on the Gateway you are importing to.

Importing Tags from the System Provider

This example imports the **allSystemTags.json** file from **Exporting Tags in the System Provider**. To import a single tag instead, see **Importing a Targeted System Tag**.

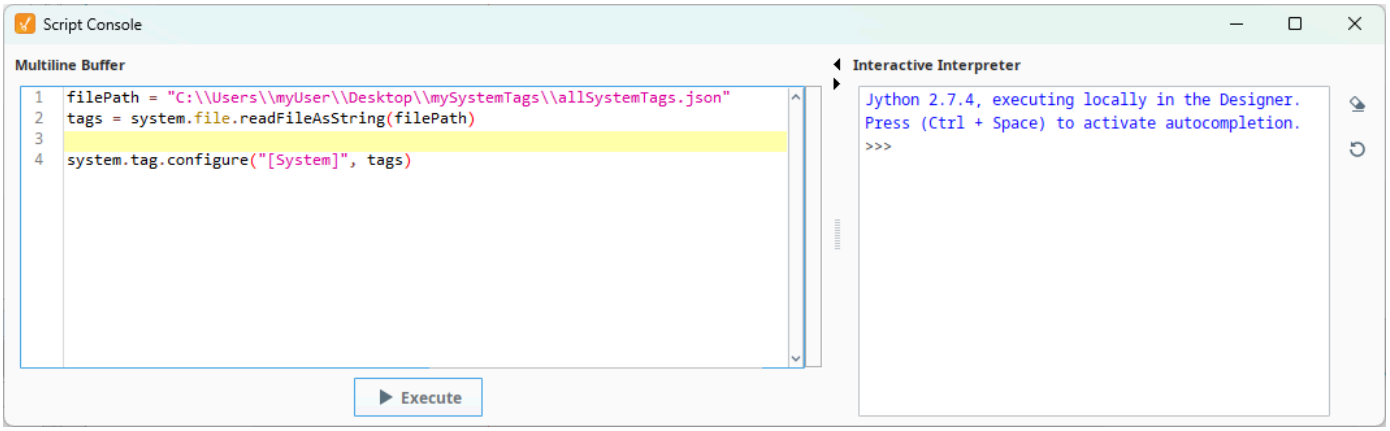
1. Open the **Script Console** in the Ignition Designer.



2. Use the **system.tag.configure()** function script below to import the **allSystemTags.json** file, replacing the `filePath` parameter with the path to your .json file:

Import allSystemTags.json File Script

```
filePath = "C:\\Users\\myUser\\Desktop\\mySystemTags\\allSystemTags.json"
tags = system.file.readFileAsString(filePath)

system.tag.configure("[System]", tags)
```



Your System tags are now updated with the imported configurations.

> **NOTE**
>
> You may receive an error saying *[Bad_AccessDenied("Insufficient Tag Provider Edit Permissions")]* after importing all your configurations. This happens because the imported

configurations may affect tags that:

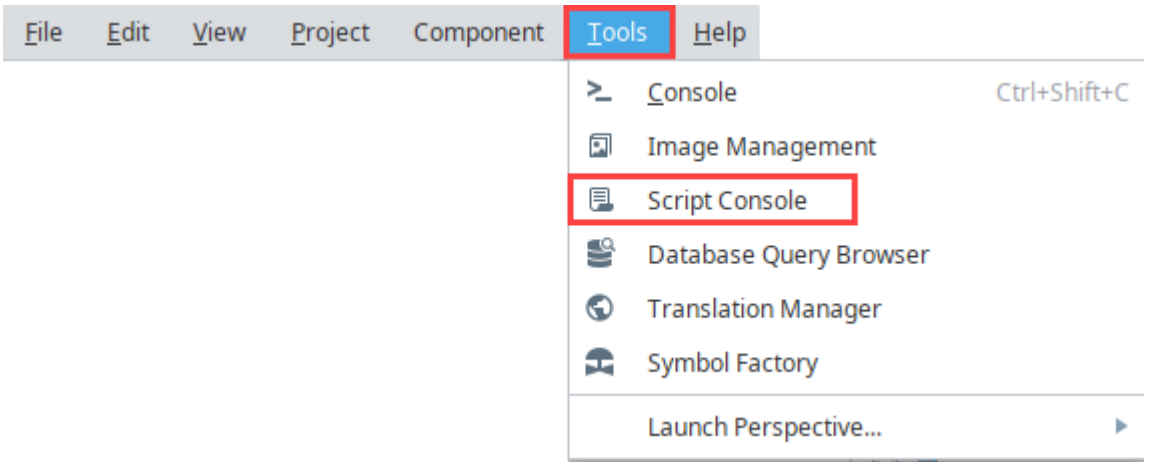
- Do not exist
- Are not in the same location as the export system
- Cannot be edited, or are in folders that cannot be edited

The imported configurations still affect the System tags that are available and in the same directory.

Importing a Targeted System Tag

This example imports the **targetedSystemTag.json** file from **Exporting a Targeted System Tag**.

1. Open the **Script Console** in the Ignition Designer.



2. Use the **system.tag.configure()** function script below to import the **targetedSystemTag.json** file, replacing the `filePath` parameter with the path to your .json file:

Import targetedSystemTag.json File Script

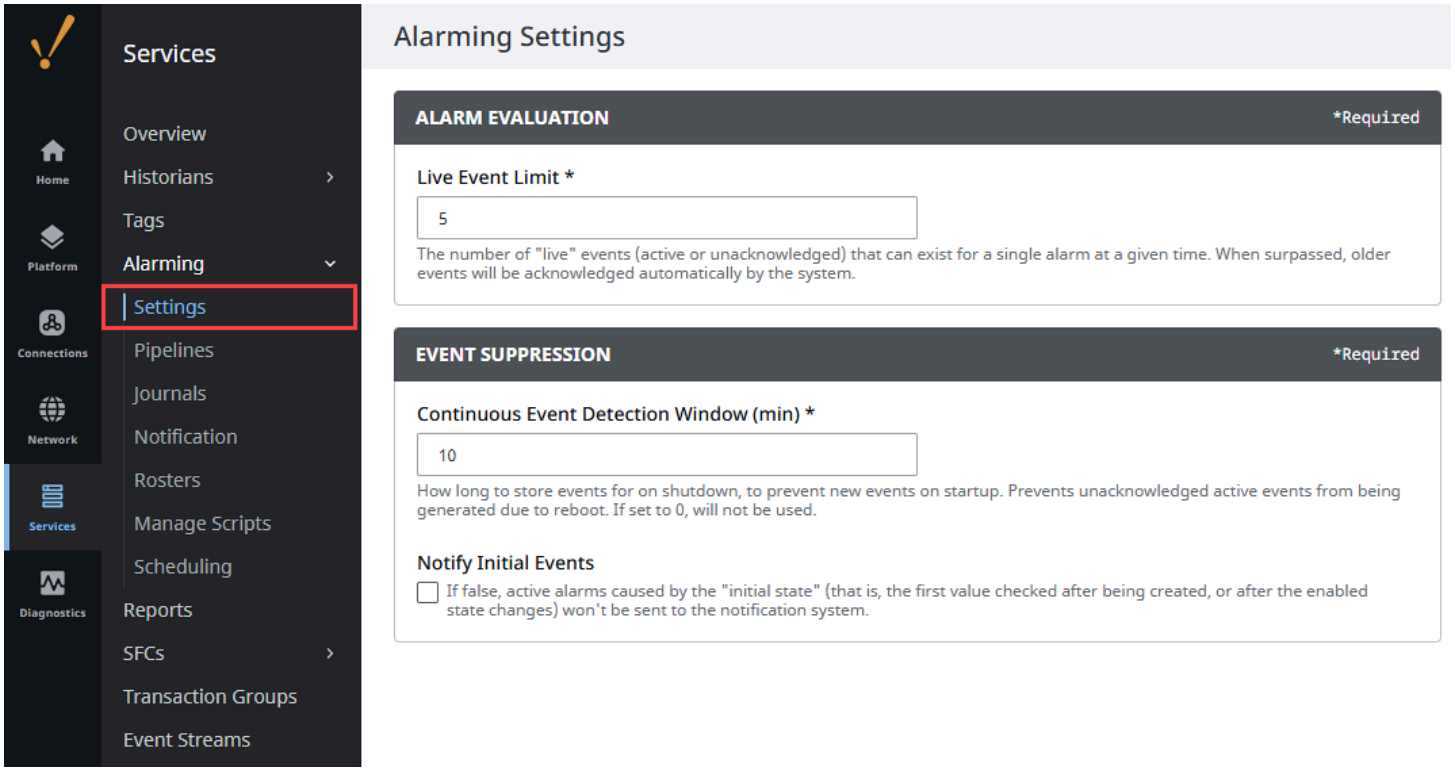
```
filePath = "C:\\Users\\myUser\\Desktop\\mySystemTags\\targetedSystemTag.json"
tags = system.file.readFileAsString(filePath)

# Unlike the previous example where all the System tags were imported to the [System] provider,
# you need to specify the exact directory where the targeted tag lives.
system.tag.configure("[System]Gateway", tags)
```

The specified System tag is now updated with the imported configuration.

Gateway General Alarm Properties

The alarm system has several settings available on the Gateway. These properties can be found under **Services > Alarming > Settings**.

> **NOTE**
>
> The Alarm Settings page is not available on Edge Gateways.



Properties

Alarm Evaluation

Name	Description
Live Event Limit	The number of "live" events (active or unacknowledged) that can exist for a single alarm at a given time. When surpassed, older events will be acknowledged automatically by the system. This means as an alarm cycles on and off, Ignition will keep track of the last five times the alarm event happened until the user acknowledges them. This does not store history for those events. Default is 5.

> **CAUTION**
> Increasing the Live Event Limit may cause your system to use more memory if alarms are frequently active.

Event Suppression

Name	Description
Continuous Event Detection Window (min)	The amount of time to store events before shutdown to prevent new duplicate events from being created on startup. This setting prevents unacknowledged active events from being generated due to reboot. If set to 0, will not be used. Default is 10.
Notify Initial Events	If false, active alarms caused by the "initial state" (that is, the first value checked after being created, or after the enabled state changes) won't be sent to the notification system. This means if you add an alarm to a tag, a notification won't be immediately sent when the new state is created. Default is false.

Alarming Schedules

The Alarm Notification system utilizes schedules to determine which users should be notified about an active alarm by looking at the users defined in the **notification block**. This means that notification messages are sent only to active users based on the defined schedules.

You can set a schedule for each user in the Alarm Notification system and utilize **On-Call Rosters** to create user groups. For example, suppose you have alarms that should be sent to all supervisors. You can put all of the supervisors in one On-Call Roster and the scheduling system will automatically notify only those supervisors who are on shift (based on their individually set schedules) when the alarm goes active.

You can also utilize Alarm Notification Profiles to determine how to send alarms to users, such as over email, SMS, voice, or Twilio. See the **Notification Profile Types** page for more information.

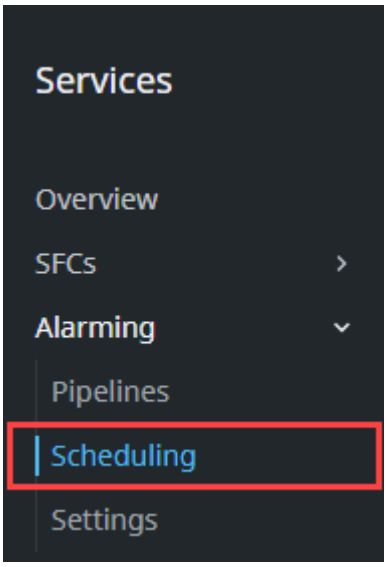
When you first access the **Alarming** > **Scheduling** page, you will see an Always and an Example schedule.

- **Always**: A built-in schedule that is active 24/7.
- **Example**: A pre-existing example schedule consisting of Monday to Friday, 8am to 5pm, with a lunch break.

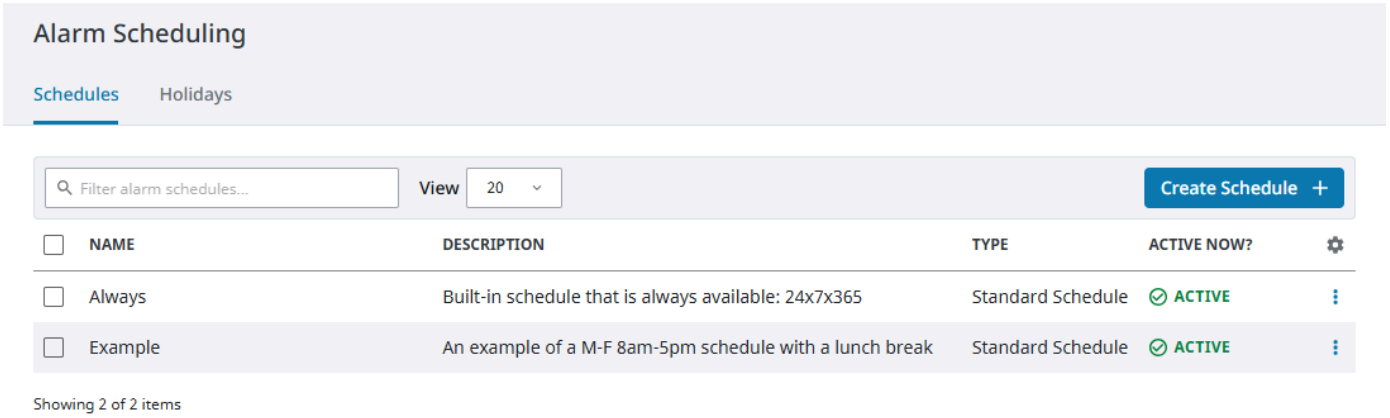
Click the three dots menu icon and select **Edit** for a schedule to access the schedule's Edit panel. From the **Alarm Scheduling** > **Schedules** tab, you can also see any available descriptions for the alarm schedules, the type of schedules, and if they are currently active under the Active Now column. An alarm will be listed as Inactive when the current day and time don't match the schedule settings, or if **Observe Holidays** is enabled on a configured holiday. Holidays can be configured on the **Alarm Scheduling** > **Holidays** tab, or by using the **system.user holiday** functions.

Define a New Schedule

1. Go to the **Services** section of the Gateway Webpage and select **Alarming** > **Scheduling**.

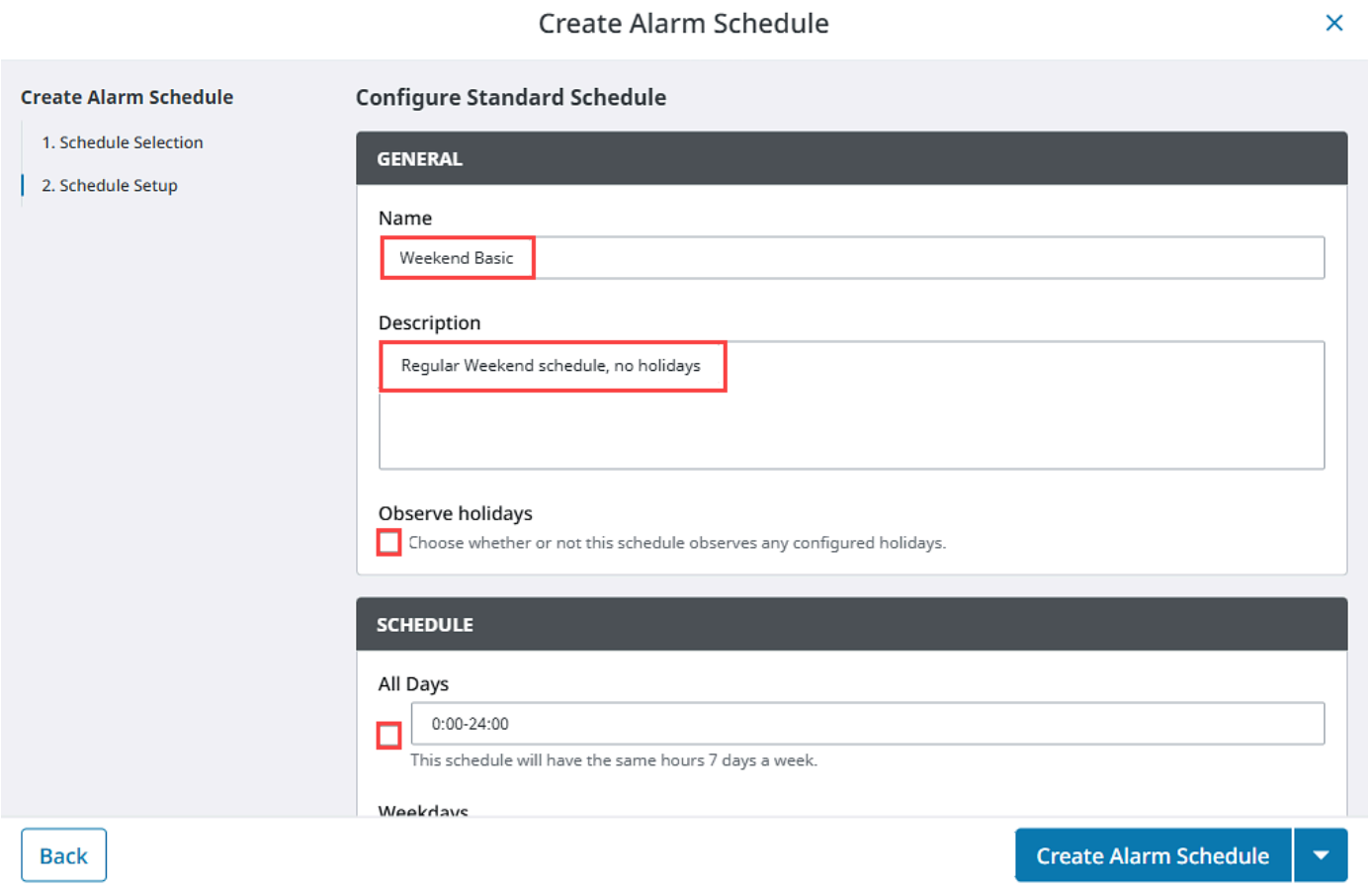


2. The Alarm Scheduling page is displayed. Click on **Create Schedule +**.



3. For our example, we'll set up a new **Standard** schedule. The other option available is a **Composite schedule**. Enter a schedule name, description, and set the hours. See the **Standard Schedule Properties** table below for a full description of the available properties.

    - Name: **Weekend Basic**
    - Description: **Regular Weekend schedule, no holidays**
    - Observe Holidays: **No** (Unselect this option)
    - All days: **No** (Unselect this option)
    - Saturday: **Yes** (Select this option)
    - Sunday: **Yes** (Select this option)



4. Click **Create Alarm Schedule**.

> **SCHEDULE RESTRICTED SETTINGS**
>
> If you want to ensure users are unable to log in when their assigned schedule is not active, you can select the **Schedule Restricted** option for the user's corresponding User Source. The user will then be denied any login access for attempts made outside active hours.

Standard Schedule Properties

A schedule defined by hours of operation on each day, with repeating options.

General

Name	Description
Name	The name of the alarm schedule.
Description	The description of the alarm schedule.
Observe Holidays	When enabled, this schedule will be inactive on any **configured holidays**. By default, this setting is enabled.

Schedule

At least one schedule option is required to be configured to create the alarm schedule. By default, all selections will be set to 24 hours a day.

> **NOTE**
> A schedule is active based on the Gateway's system time, not a user's local time.

Name	Description
All Days	This schedule will have the same hours seven days a week. This option is selected by default. No other options can be selected when this option is selected.
Weekdays	This schedule will have the same hours Monday-Friday, but will have no hours on the weekends. No other options can be selected when this option is selected.
Monday	Determines the Active hours of the alarm for every Monday. This option can be configured with any of the other individual day selections.
Tuesday	Determines the Active hours of the alarm for every Tuesday. This option can be configured with any of the other individual day selections.
Wednesday	Determines the Active hours of the alarm for every Wednesday. This option can be configured with any of the other individual day selections.
Thursday	Determines the Active hours of the alarm for every Thursday. This option can be configured with any of the other individual day selections.
Friday	Determines the Active hours of the alarm for every Friday. This option can be configured with any of the other individual day selections.
Saturday	Determines the Active hours of the alarm for every Saturday. This option can be configured with any of the other individual day selections.
Sunday	Determines the Active hours of the alarm for every Sunday. This option can be configured with any of the other individual day selections.

Repeating

Name	Description
Repeat/Alternate	Repeats an alternating schedule, determined by the Daily or Weekly settings below. Default is Off. If you set to anything other than Off, and you do not specify a starting day, the schedule will never be active.
# Days/Week On	Sets the number of days or weeks on. Use the arrows to adjust the amount. Default is 1.
# Days/Week Off	Sets the number of days or weeks off. Use the arrows to adjust the amount. Default is 1.
Starting At	Pick a date that defines the starting day or week for the repetition. For example, suppose you had a one week on, one week off schedule, and you picked a Monday of some week. The schedule would be active for that week, and would inactive on the following Monday. This pattern will then repeat indefinitely.

Composite Schedule Properties

A schedule defined by the composite of two other schedules. This type of schedule will be active whenever either of the two assigned schedules are active.

Name	Description
Name	The name of the alarm schedule.
Description	The description of the alarm schedule.
Schedule One	The first schedule configurations to apply to this schedule.
Schedule Two	The second schedule configurations to apply to this schedule.

Configure Holidays

Holidays associated with a schedule's Observe Holidays setting can be created, edited, and deleted on the Holidays tab of the Alarm Scheduling page. To create a holiday, select **Create new Holiday**

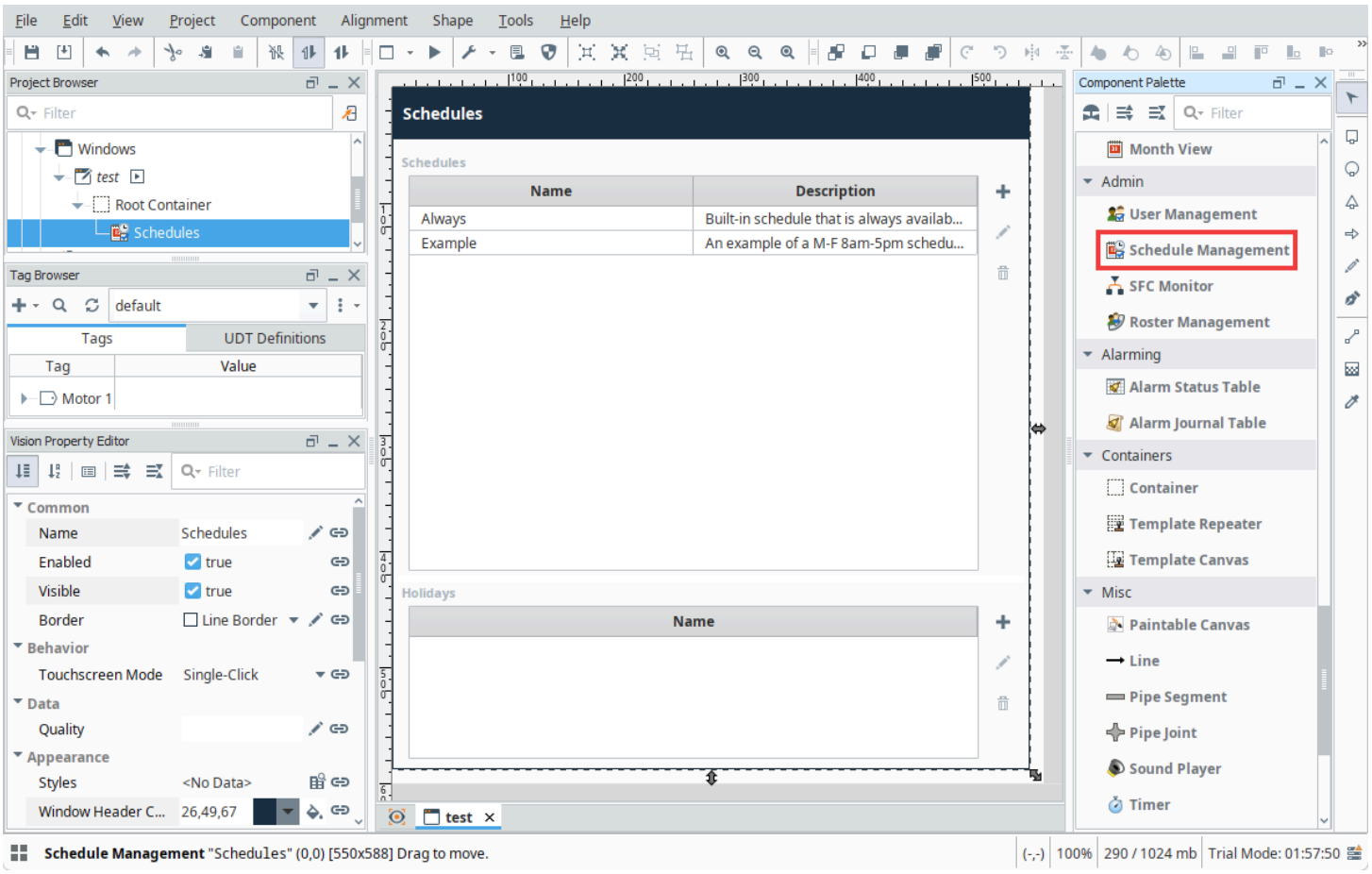
and enter the relevant Name, Date, and Repeat Annually values. Holidays can be edited or deleted at any time using the **Edit** and **Delete** buttons.

Holiday Properties

Name	Description
Name	The name of the holiday.
Date	The date of the holiday. Accepted format is MM-DD-YYYY.
Repeat Annually	Repeats this holiday on the same day each year when checked. By default, this setting is checked.

Manage User Schedules from the Vision Client

There are a few ways to manage user schedules from the Vision Client. The first is to use the **Schedule Management** component on a window. This component allows you to quickly and easily manage the schedules from the Vision Client.



Alternatively, you may instead want to use scripting to manage the schedules. This may offer a more granular control at the click of a button. There are a number of system functions that allow you to create, read, edit, and delete schedules or holidays from a user source using scripting (i.e., **system.user.addSchedule**, **system.user.getHoliday**, etc.). To learn what system functions are available for user scheduling and holiday scripting, refer to the **system.user** functions in the Appendix.

```
# This code creates a new schedule by using an old schedule but setting observe
holidays to true.
mySchedule = system.user.getSchedule("WeeklySchedule")
if mySchedule != None and mySchedule.getType() == "basic schedule":
    mySchedule.setObserveHolidays(True)
    mySchedule.setName("NewWeeklySchedule")
    system.user.addSchedule(mySchedule)
```

Localization and Languages

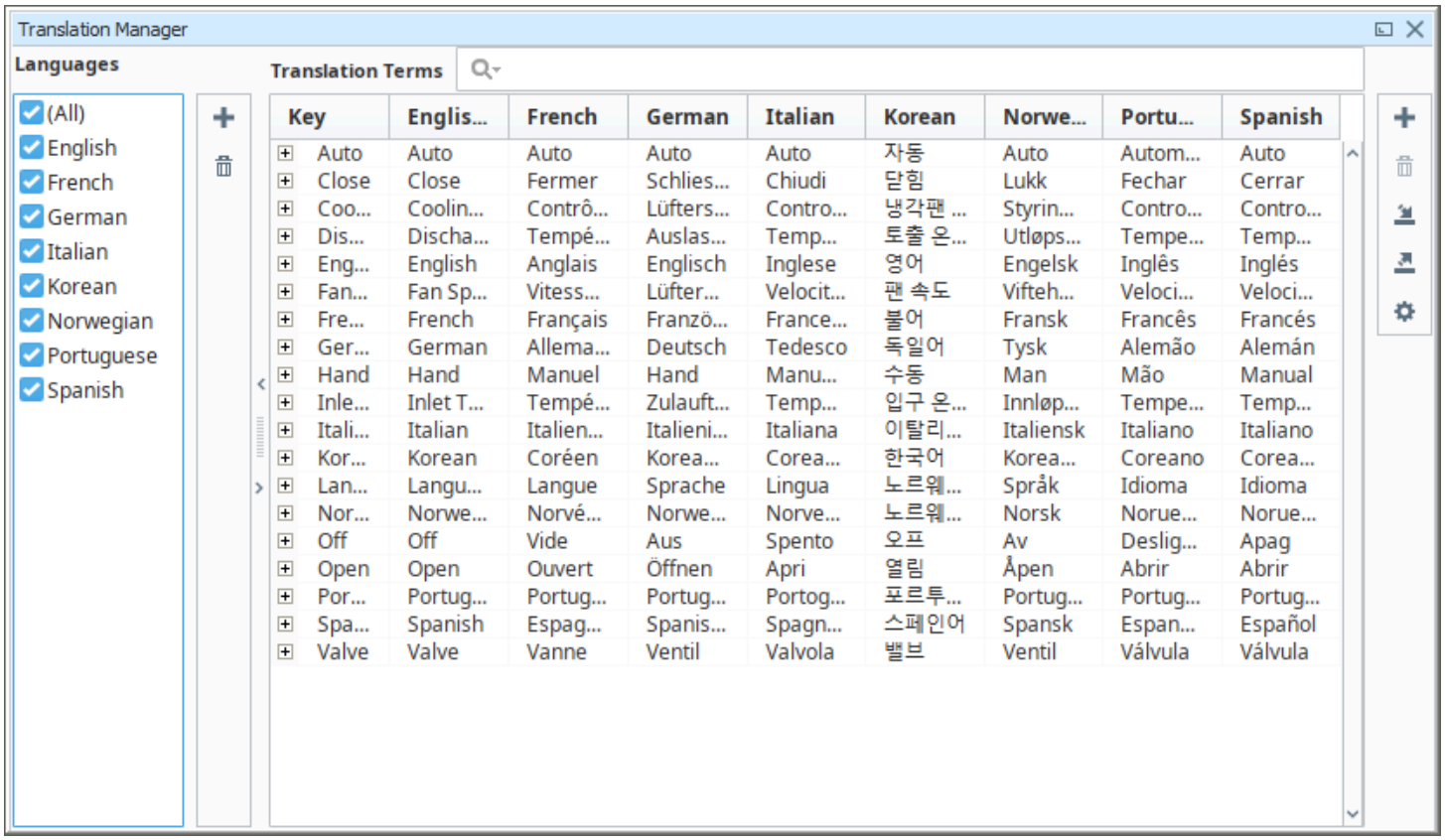
Translating Ignition

Localization lets you display projects in multiple languages. Translations are stored in the Gateway and can be used in Perspective sessions, Vision clients, alarms, and scripts. This allows users in different regions to view the same project in their preferred language without additional setup in each project.

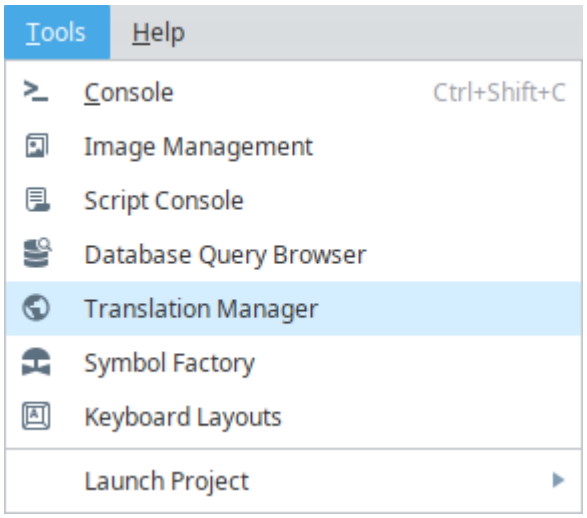
In Perspective, localization is configured by mapping component text to translation keys and setting the session's locale. When the locale changes, the text updates automatically. For example, you can provide a dropdown that switches the session from English to Spanish. Vision also supports localization through the same translation database, and it can define component-level terms through the **Translatable Terms panel**. Vision projects can also use tools such as the **Language Selector** component.

Translation Manager

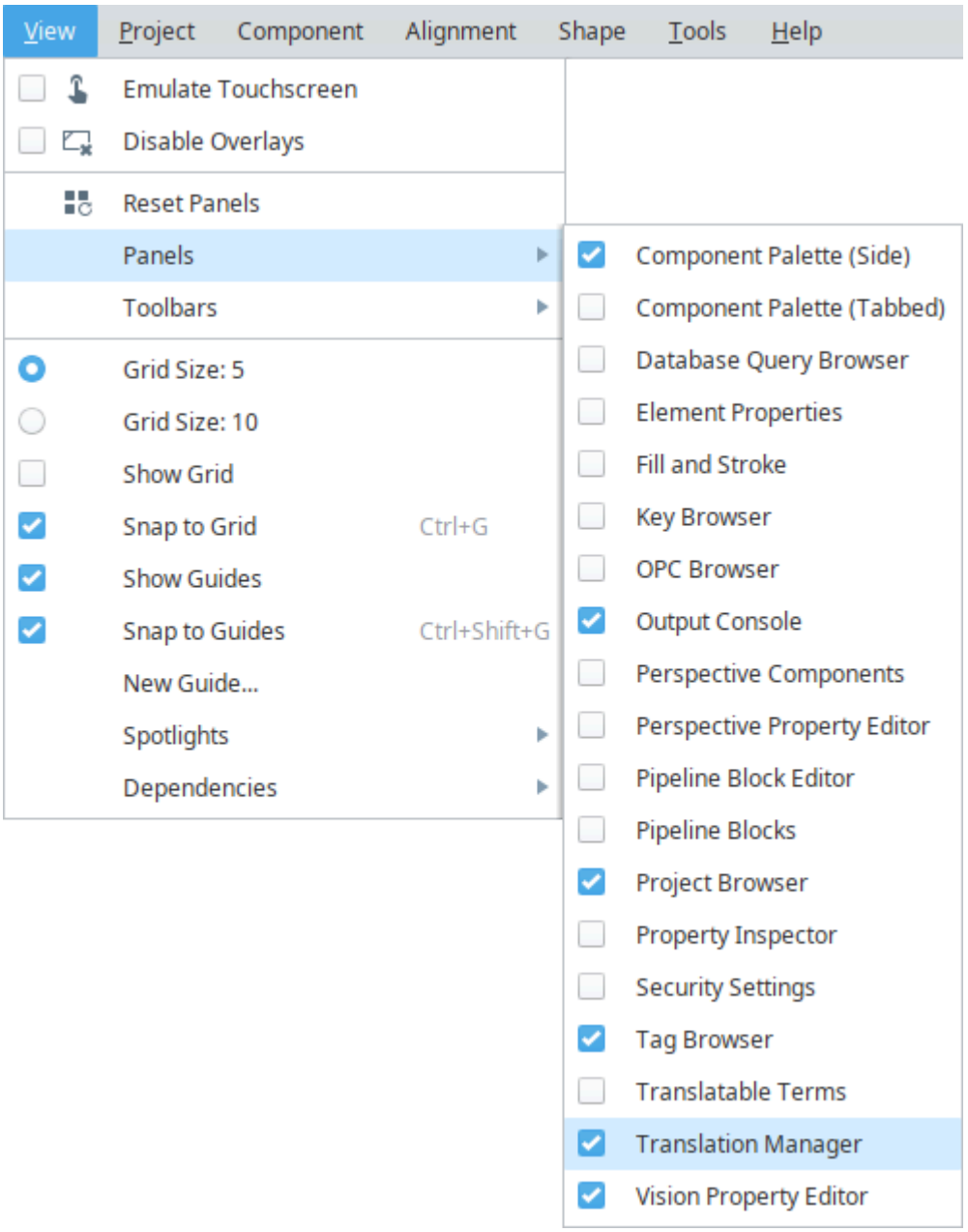
All translations are stored in a shared database that makes it possible to manage terms in one place and apply them to any project. Translation functionality is available in Perspective session, Vision Clients, Gateway scripts, and alarm messages. Translation support is built into component text properties and other text-based properties, such as with alarm messages. In most cases, once a translation is defined, no additional configuration is needed for it to be applied. You can view and edit the translation database in the Translation Manager panel in the Designer. Use the Translation Manager to add, edit, or remove terms. You can create translations for single words or longer phrases, define new languages, and **import and export** translation lists.



To open the Translation Manger panel, go to the menu bar and select **Tools** > **Translation Manager**.



To add the panel to your Designer interface, go to the menu bar and select **View** > **Panels** > **Translation Manager**.



Translation Database and Term Lookup

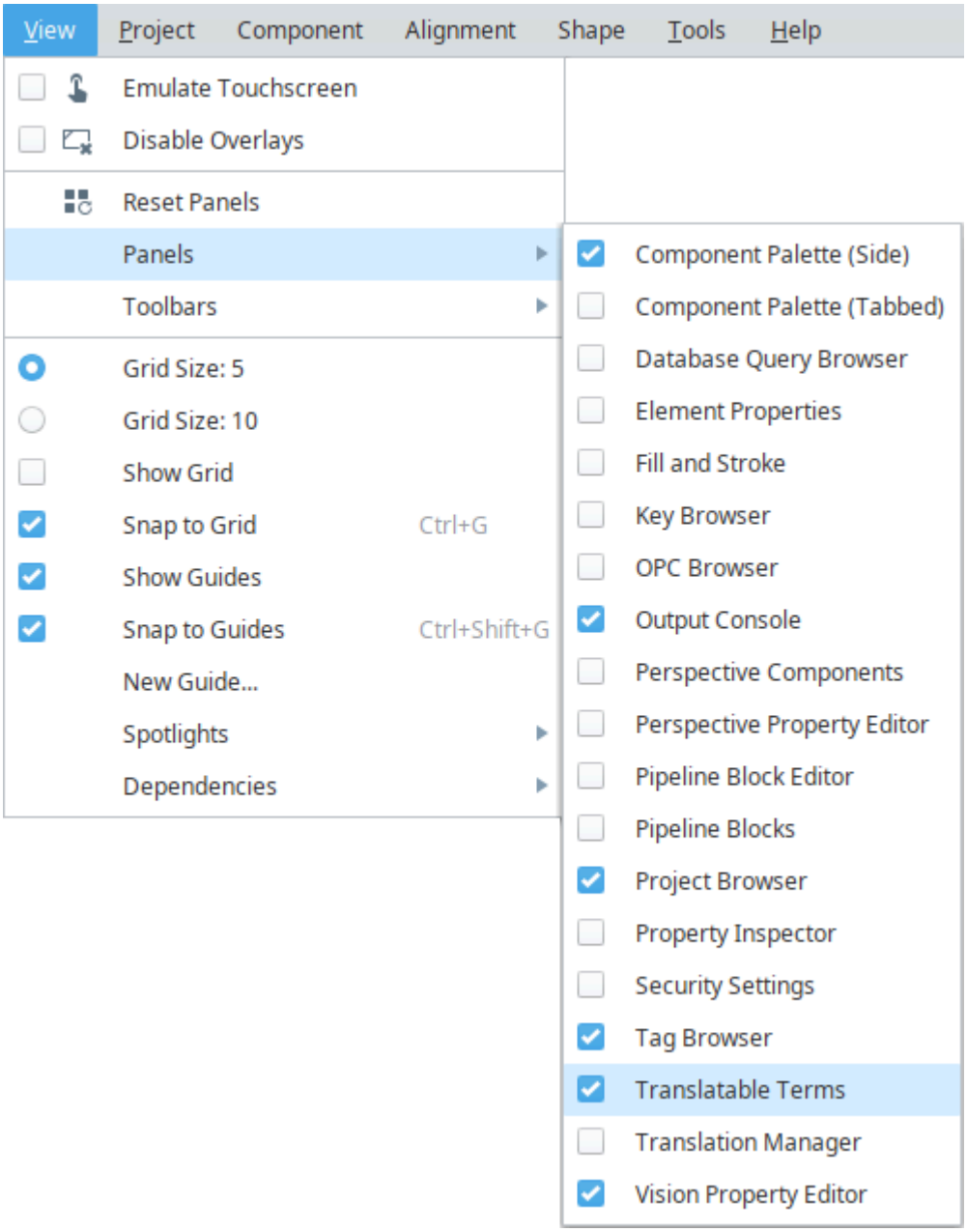
Translations are stored in the Gateway's database as **keys**. A key is the base term Ignition uses to look up the correct translation. Keys can be simple text such as **Start**, or coded identifiers such as **START_COMMAND** or **#start**. Using coded identifiers helps avoid accidental matches and makes terms easier to reuse across projects.

When a component property references a key, Ignition checks the translation database for a matching entry in the active language. If no match is found, the English value is displayed by default.

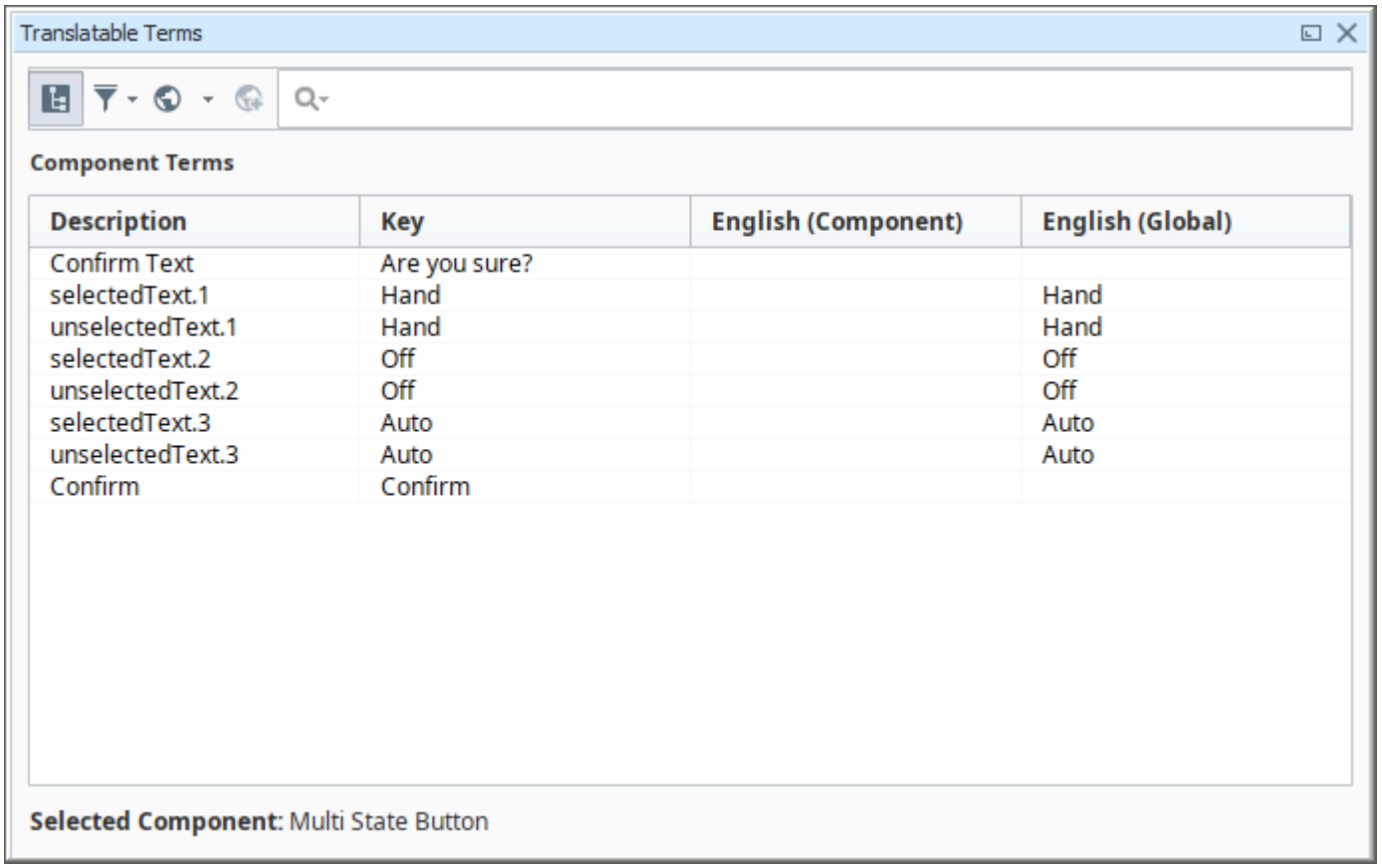
In Perspective, you assign a key to a component property in the Property Editor. When the session's locale changes, all components bound to that key automatically update to display the correct translation.

Using the Translatable Terms Panel in Vision

In Vision, you can view and edit translation keys directly in the Translatable Terms panel. To open it, go to the menu bar and select **View** > **Panels** > **Translatable Terms**.

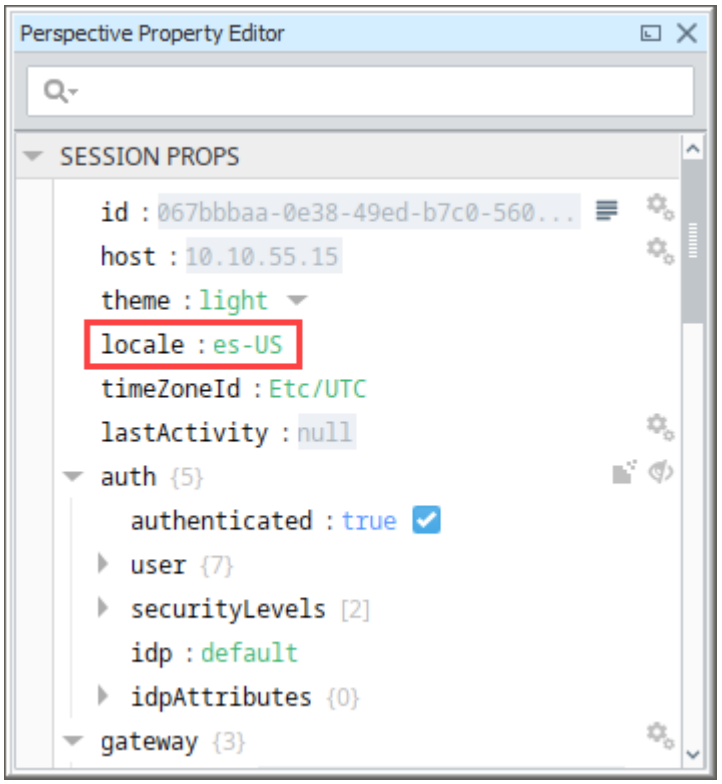


The panel shows each term for the components in the current window, along with the translations available for the active language.

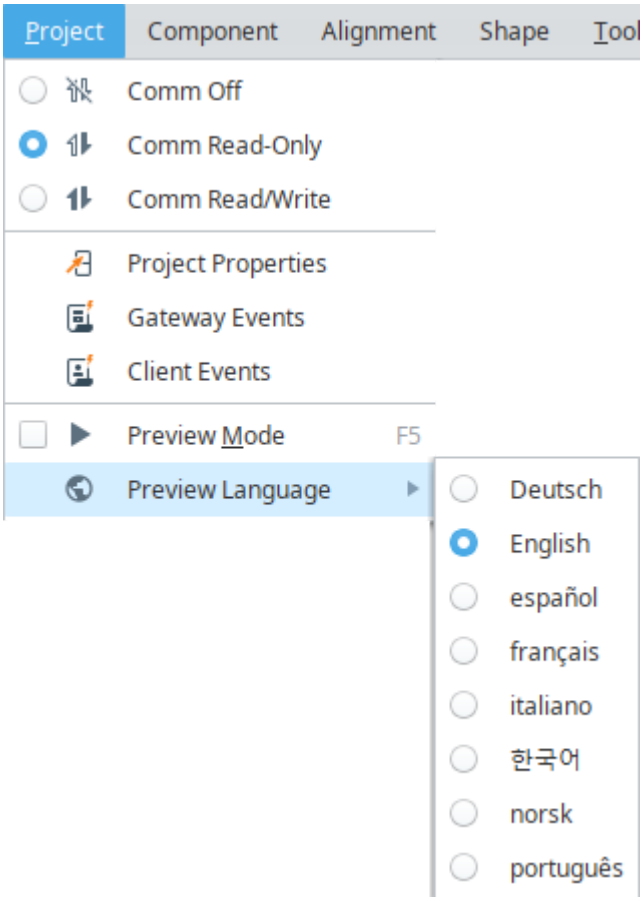


Previewing Translations

In Perspective, you can preview translations by changing the session's locale. In the Designer, select **Perspective** in the Project Browser to open the **Session Properties**, then set **locale** to the desired language. You can update the locale in a running session through scripting or bindings. When the locale changes, all text mapped to translation keys updates automatically.



In Vision, you can preview translations by setting a Preview Language in the Designer. Go to the menu bar, select **Project** > **Preview Language** and choose a language. When you switch to Preview Mode, text in the window updates to the selected language. You can also use the Language Selector component or a script to temporarily change the language. The Designer remembers your preview language even after a restart, but it reverts whenever Preview Mode is disabled.



Creating Translation Lists

INDUCTIVE UNIVERSITY

Introduction to Project Translation

Watch the video

Project Translation Overview

This page explains how to create new languages, add translation terms, and build lists you can reuse across projects. Translations are managed in the **Translation Manager**, a global database stored in the Gateway and shared across all projects.

You can create translations for single words or longer phrases, edit or remove existing terms, and import or export translation lists for use in other environments. Once a translation is defined, it can be applied anywhere in your projects without additional configuration.

TRANSLATION NOTES

- Vision Table components do not support translation. However, the Vision **Alarm Status Table** and **Alarm Journal Table** components do.
- For text properties with bindings, translations occur after the binding is evaluated.

Component vs Global Terms

The translation system supports two types of terms, **global terms** and **component terms**.

- **Global terms** are stored in the Gateway's shared translation database. They can be applied across all projects, windows, and components. Global terms are recommended for most cases because they only need to be defined once and are automatically reused whenever the same key appears.

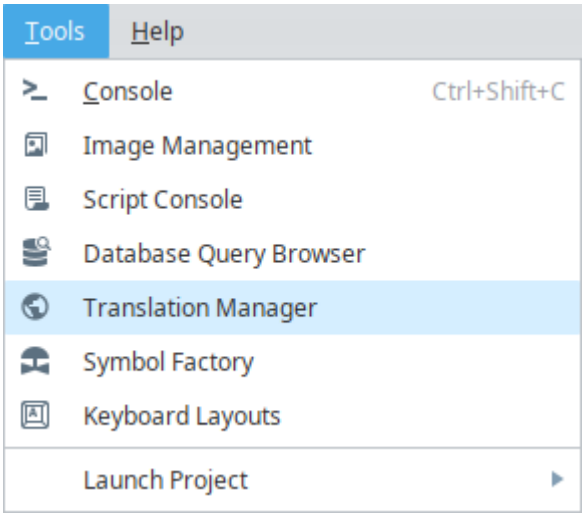
- **Component terms** are available only in Vision and are defined in the Translatable Terms panel for a specific component or window. A component term always takes precedence over a matching global term. Component terms are useful when a translation must differ in a specific context. For

example, you may have a global term for "Tank", but in one Vision window you want it to display as "Barrel". Adding a component term ensures that window shows the context-specific translation.

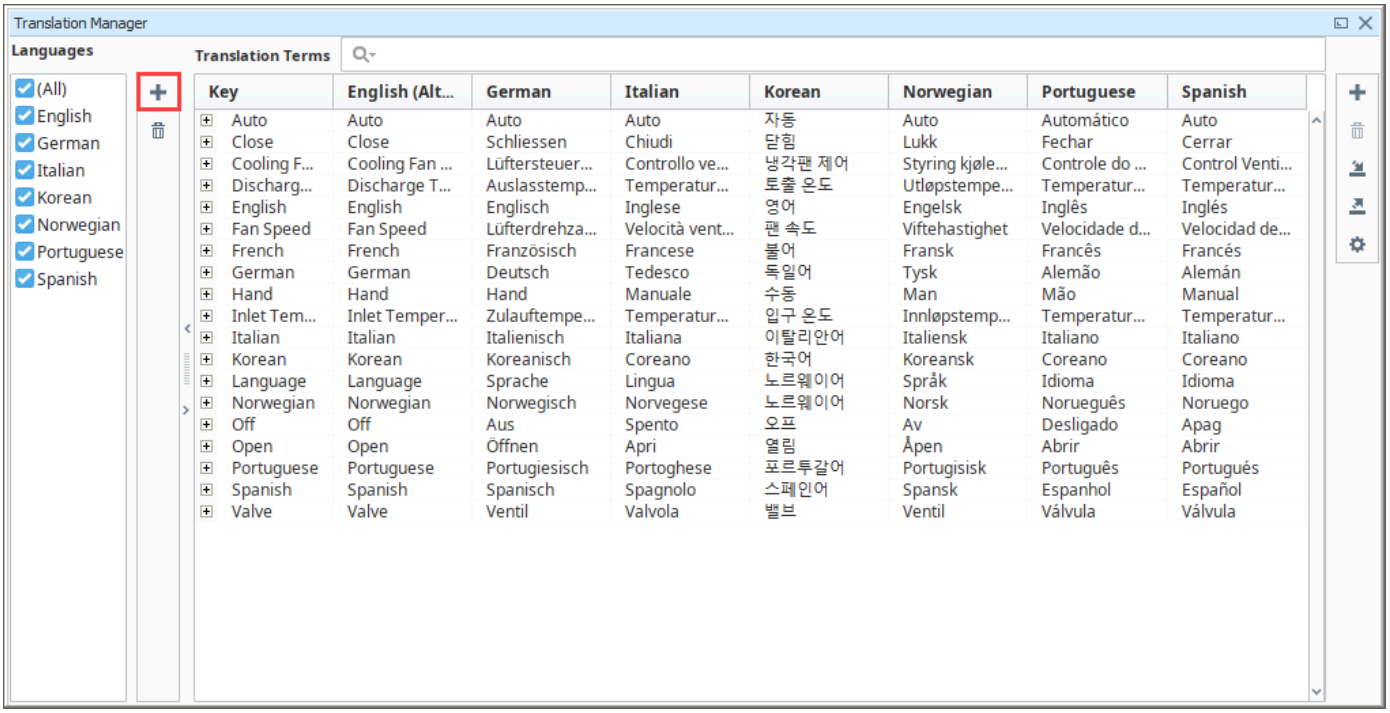
Add a New Language

Before creating translations, you must define the languages that your project will support. The Translation Manager comes with English, French, German, Italian, Korean, Norwegian, Portuguese, and Spanish defined by default, but you can add any number of additional languages that are available. Once added, a language is available for all projects.

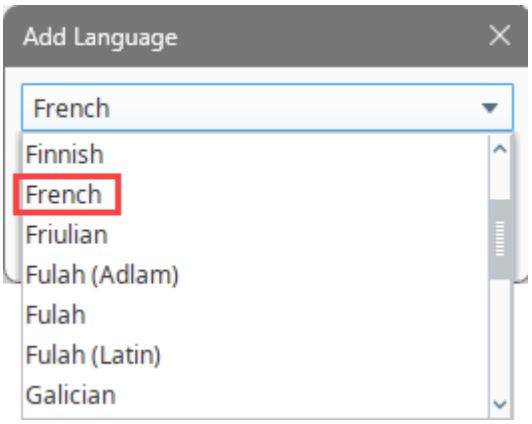
1. In the Designer, go to the menu bar and select **Tools** > **Translation Manager**.



2. In the **Translation Manager**, click the **Plus** icon next to Languages on the left side.

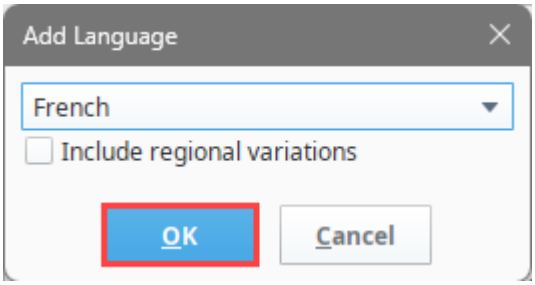


3. In the **Add Language** dialog, select the language you want from the dropdown.

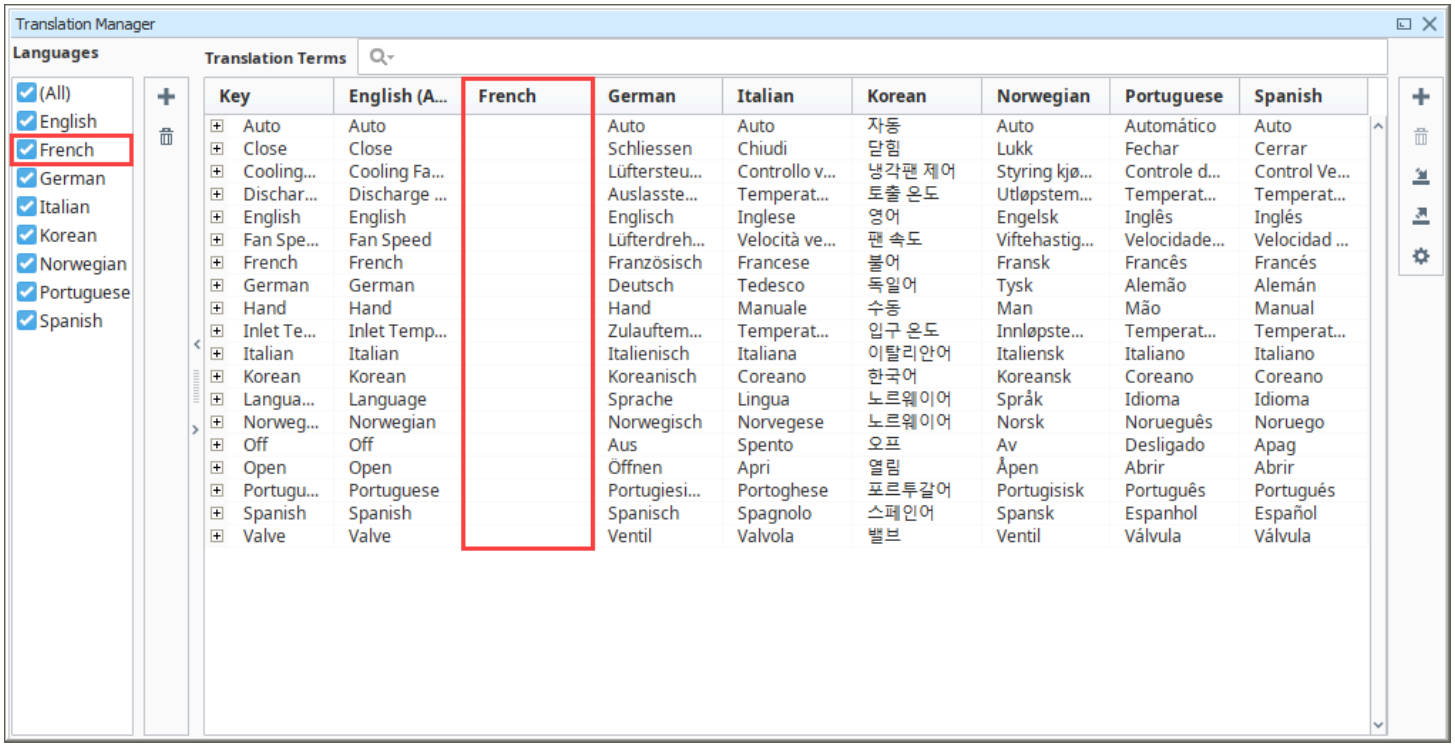


Optionally, check **Include regional variations** to make region-specific codes available (for example, French (Canada) vs. French (France)).

4. Click **OK**.



The new language will appear in the Languages panel and as a new column in the Translation Terms table.



Add Global Translation Terms

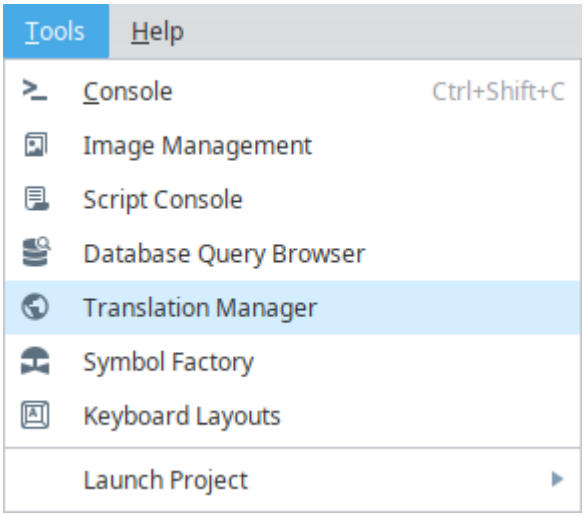
Global translation terms are shared across all projects and are the preferred way to manage most translations. Component-level terms (Vision only) should only be used when a translation needs to override a global definition in a specific context.

There are two ways to add global terms:

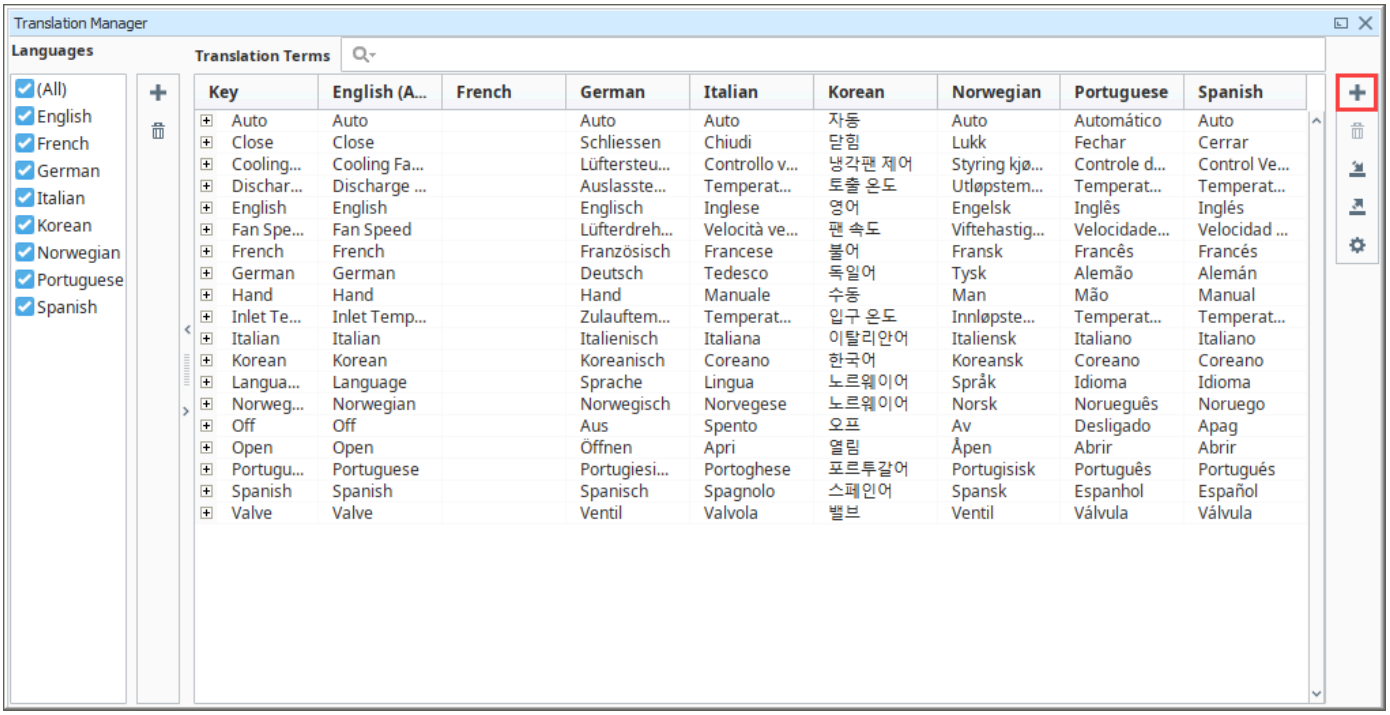
- Directly in the Translation Manager
- From the Translatable Terms panel in Vision

The steps below show how to add a global term directly in the Translation Manager.

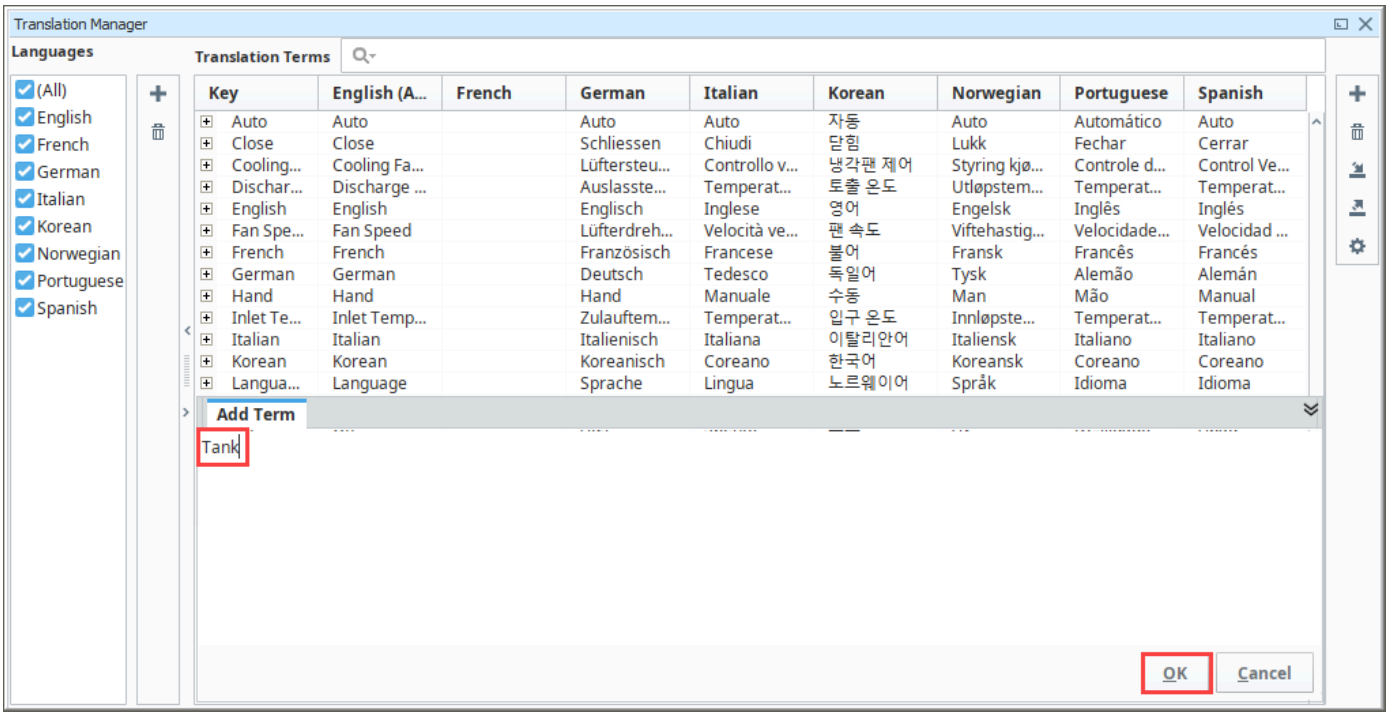
1. In the Designer, go to the menu bar and select **Tools** > **Translation Manager**.



2. In the Translation Terms table, click the **Plus** icon on the right side.

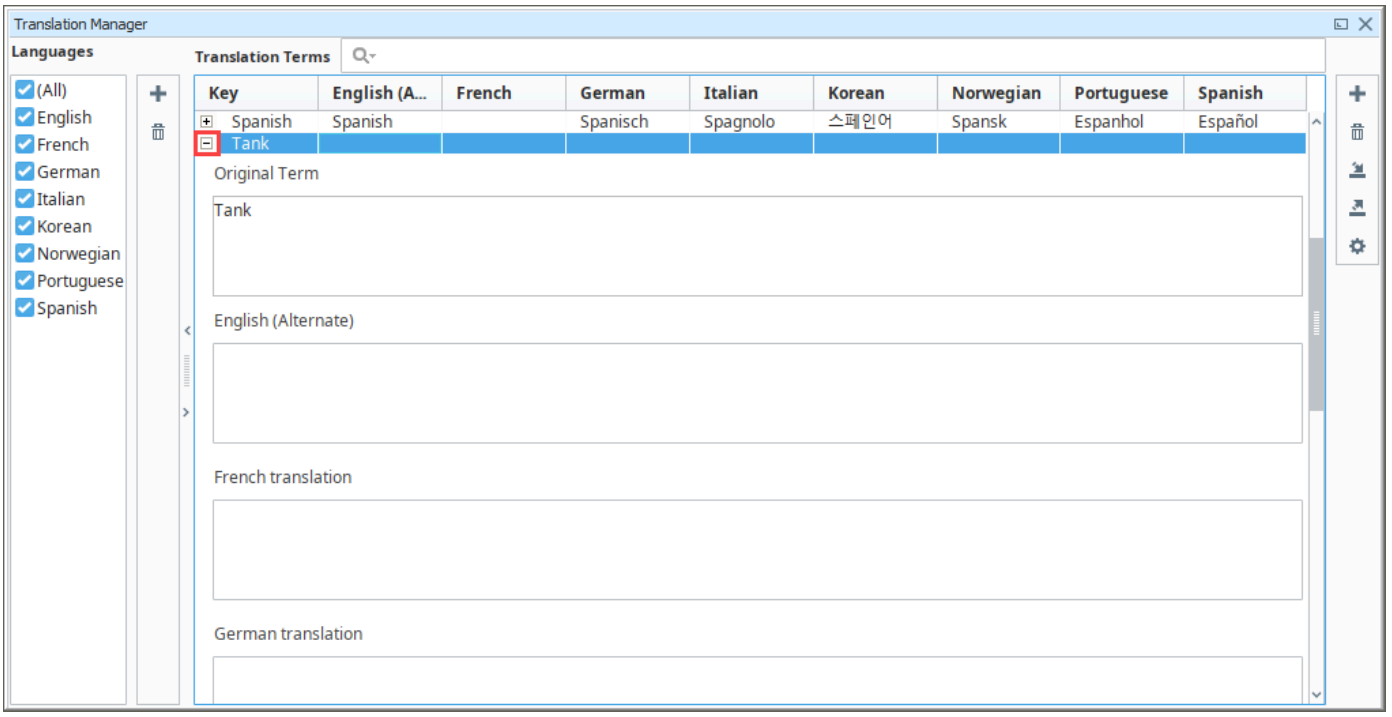


3. A dialog will appear at the bottom of the table. Enter the base term for the **Key** column. In this example, we will use **Tank**.

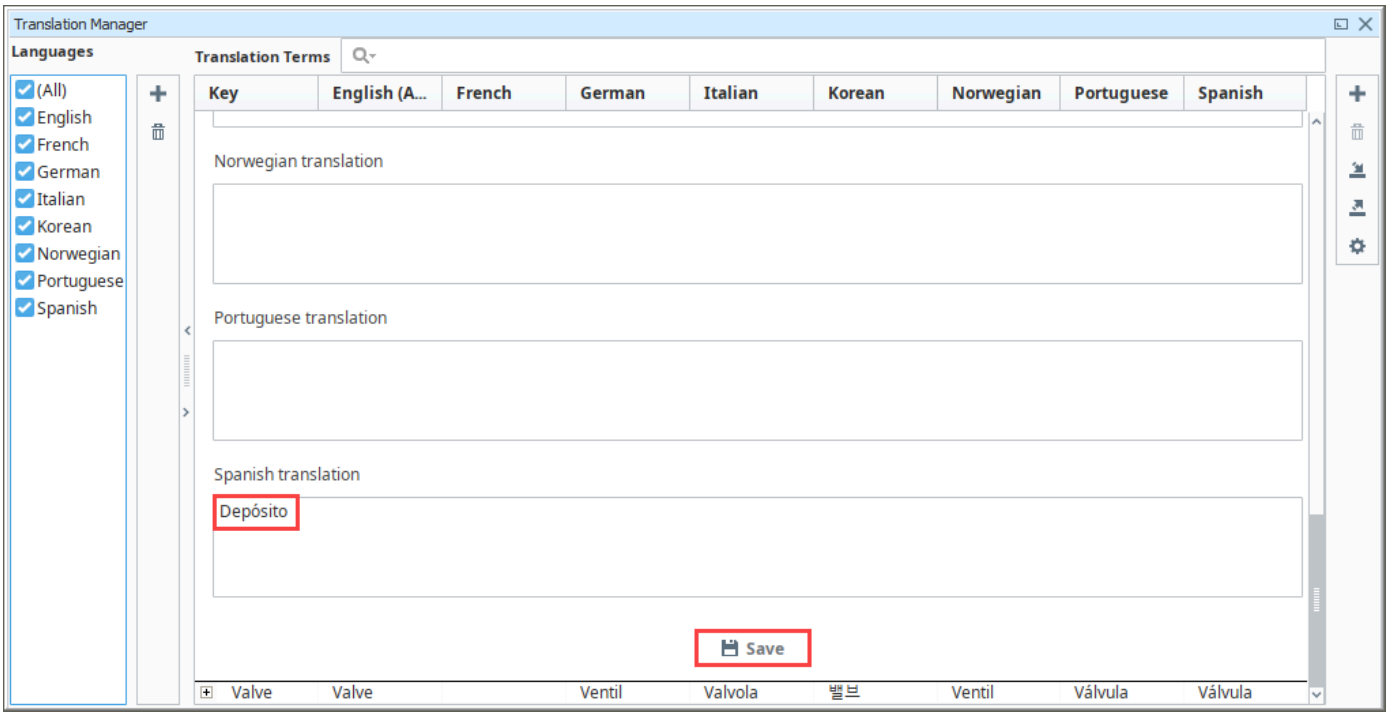


4. Click **OK**.

5. Click the + next to the new key to expand the translation fields for each language that has been added.

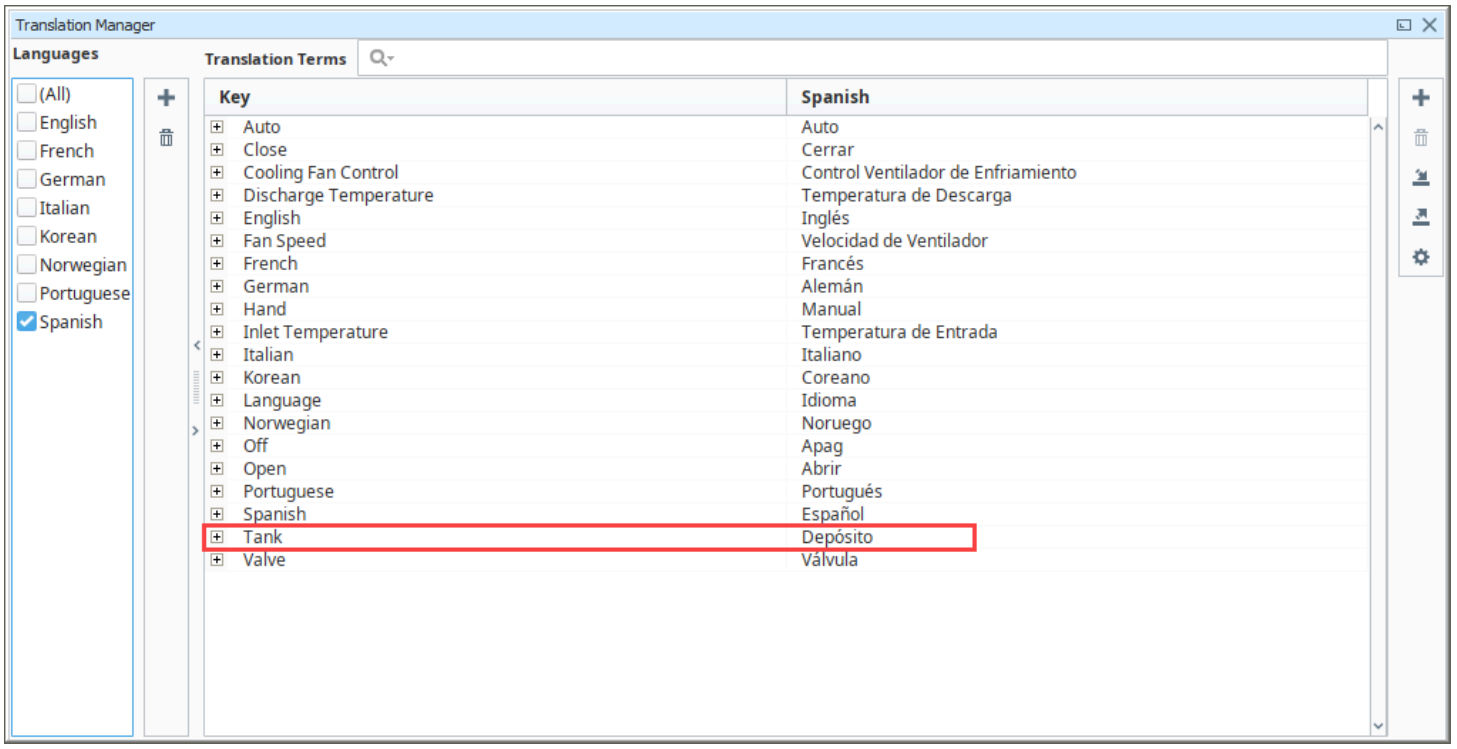


6. Enter translations for each available language. In this example, we will use "Depósito".



7. Click **Save**.

The term is now available globally across projects.

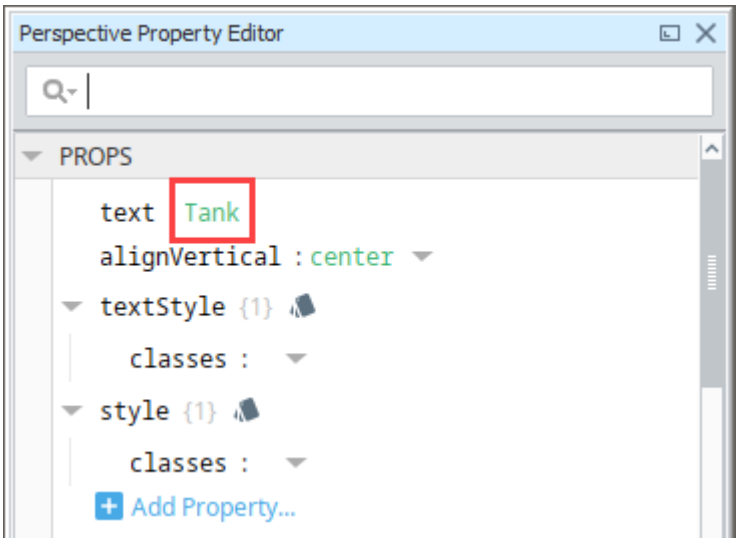


Testing Global Terms

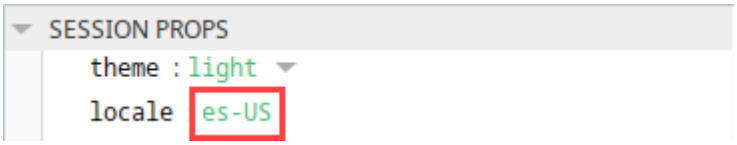
After you add a global translation term, you can test it in your projects to make sure the translations apply correctly. The steps differ slightly depending on whether you are working in Perspective or Vision.

Perspective

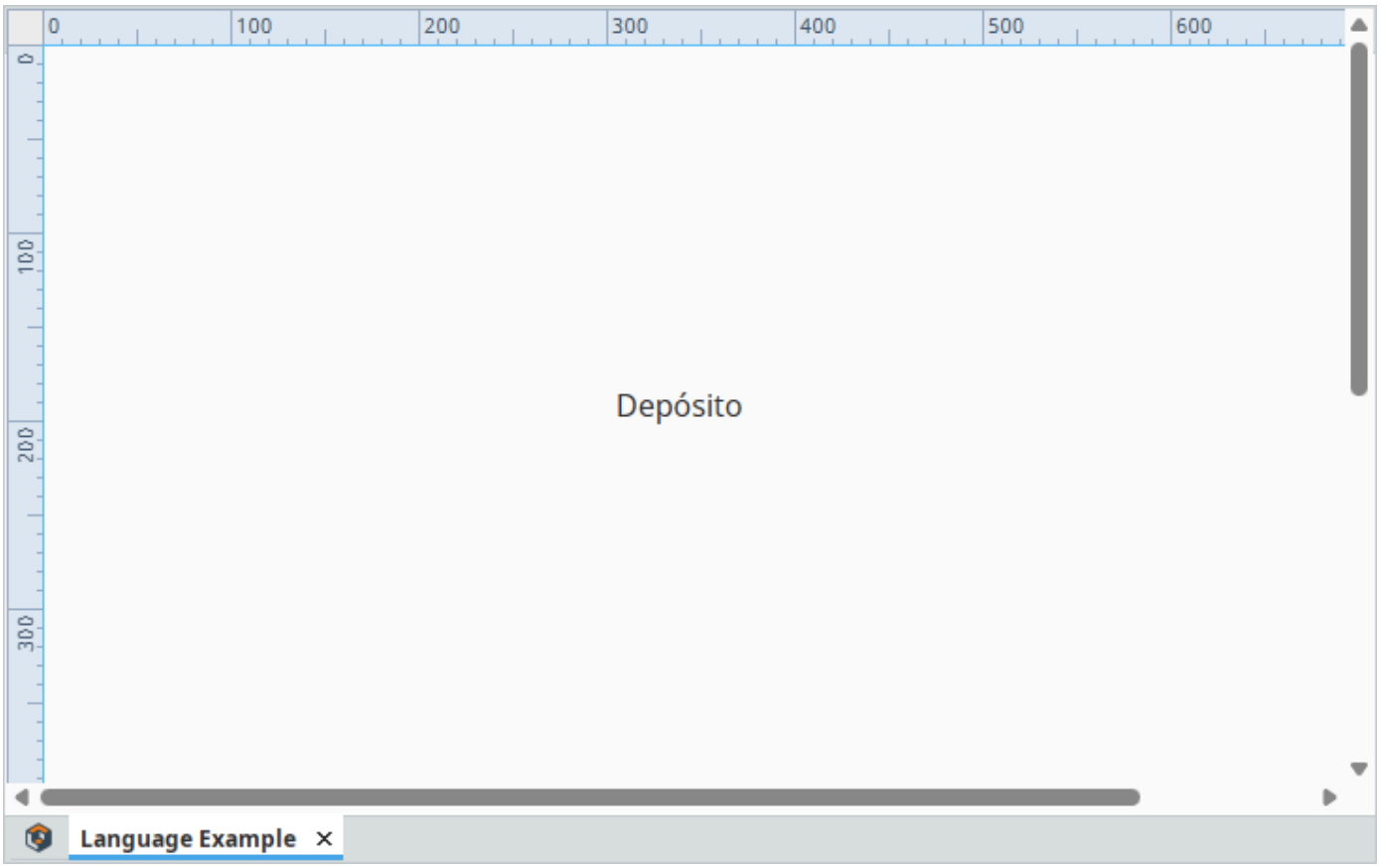
1. In the Designer, open a View in the Project Browser.
2. Add a Label component and set the `text` property in the Property Editor to the key you created in the Translation Manager.



3. Change the session's `locale` property (for example, `es-US` for Spanish).

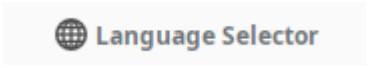


4. Verify that the component text updates to the correct translation.



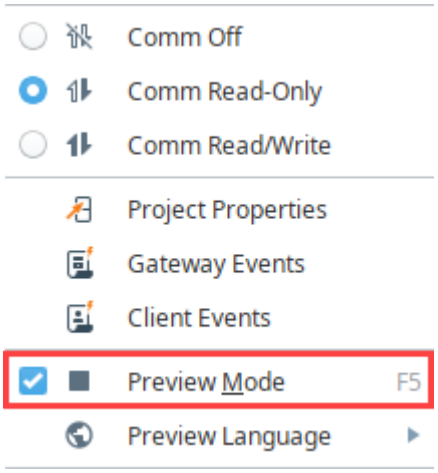
Vision

1. In the Designer, add a Language Selector component to a window.

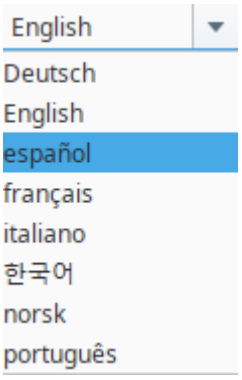


2. Add a Label component and set the `text` property in the Property Editor to the key you created in the Translation Manager.

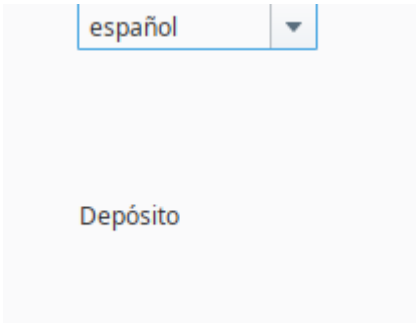
3. Switch to **Preview Mode**.



4. Use the Language Selector dropdown to toggle between languages.



5. Verify that the component text updates to the correct translation.



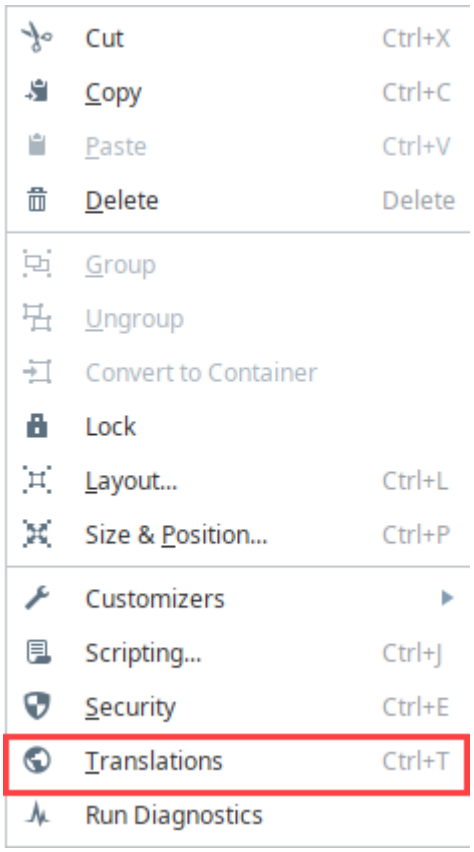
Add Component Level Translation Terms

> **NOTE**
>
> Component level translation terms are only available in Vision.

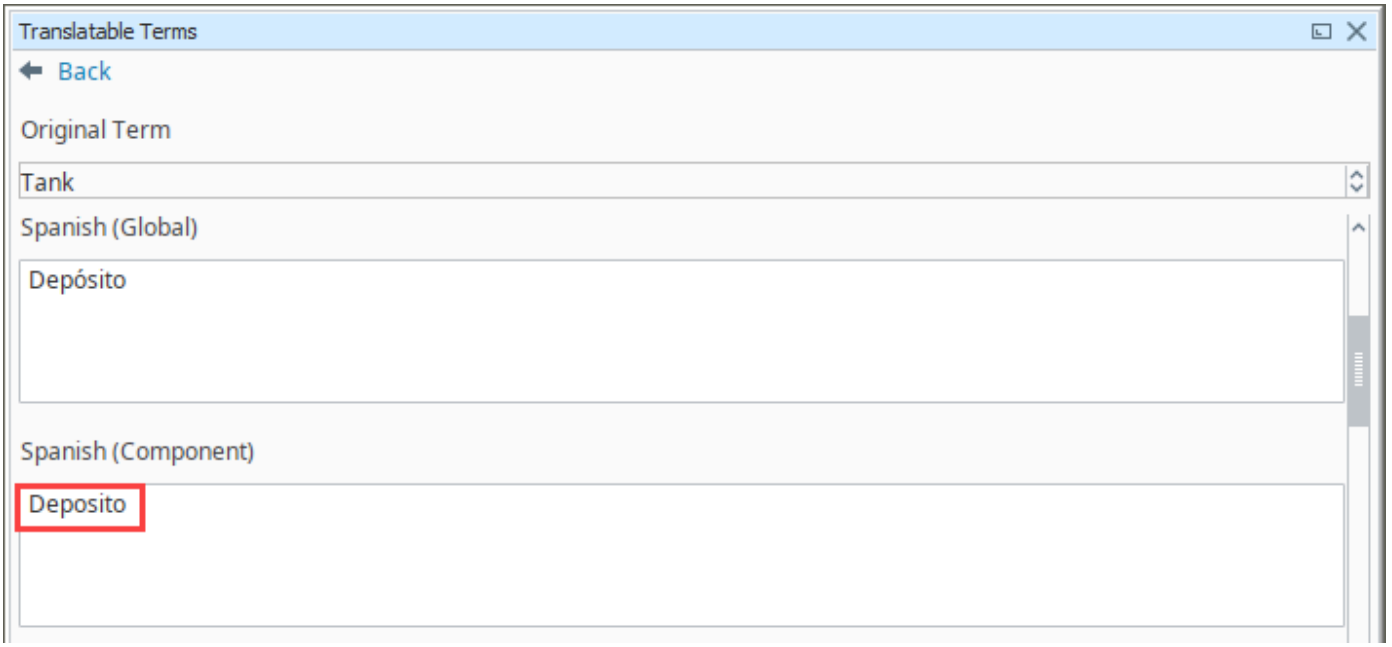
Component terms are defined in the Translatable Terms panel. This panel lists all the terms used by a component, or an entire window if opened at that level. Component terms always override global terms with the same key, allowing you to provide translations in context when a translation needs to differ in a specific window or component.

1. In the Designer, add a Label component to a Vision window.

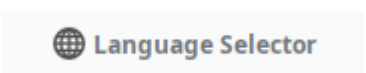
2. Right-click the component and select Translations to open the Translatable Terms panel.



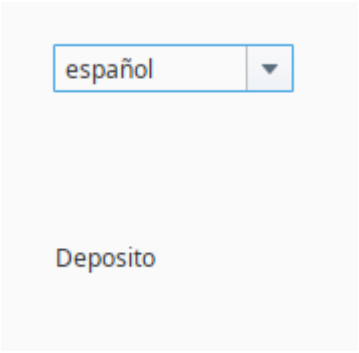
3. Double-click the component column of the chosen language to open the and enter the translation to the Component field. In this example, we added an alternate translation in Spanish (Component).



4. Add a Language Selector component to the window.



5. Switch to Preview Mode and switch languages to verify the text updates.



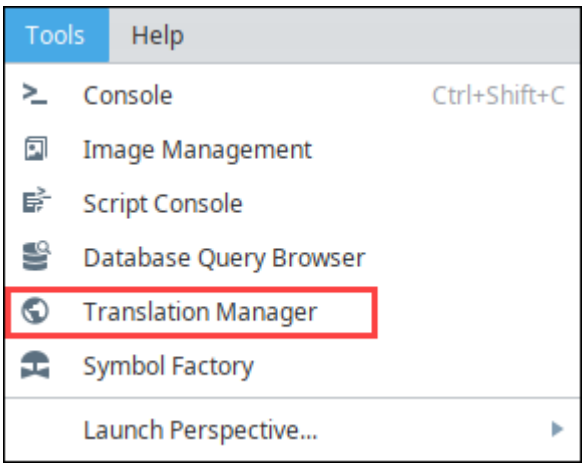
Translation List Import and Export

Importing and exporting translation lists makes it easier to manage large projects. You can quickly share terms across environments, send lists to third parties for translation, and keep multiple projects up to date.

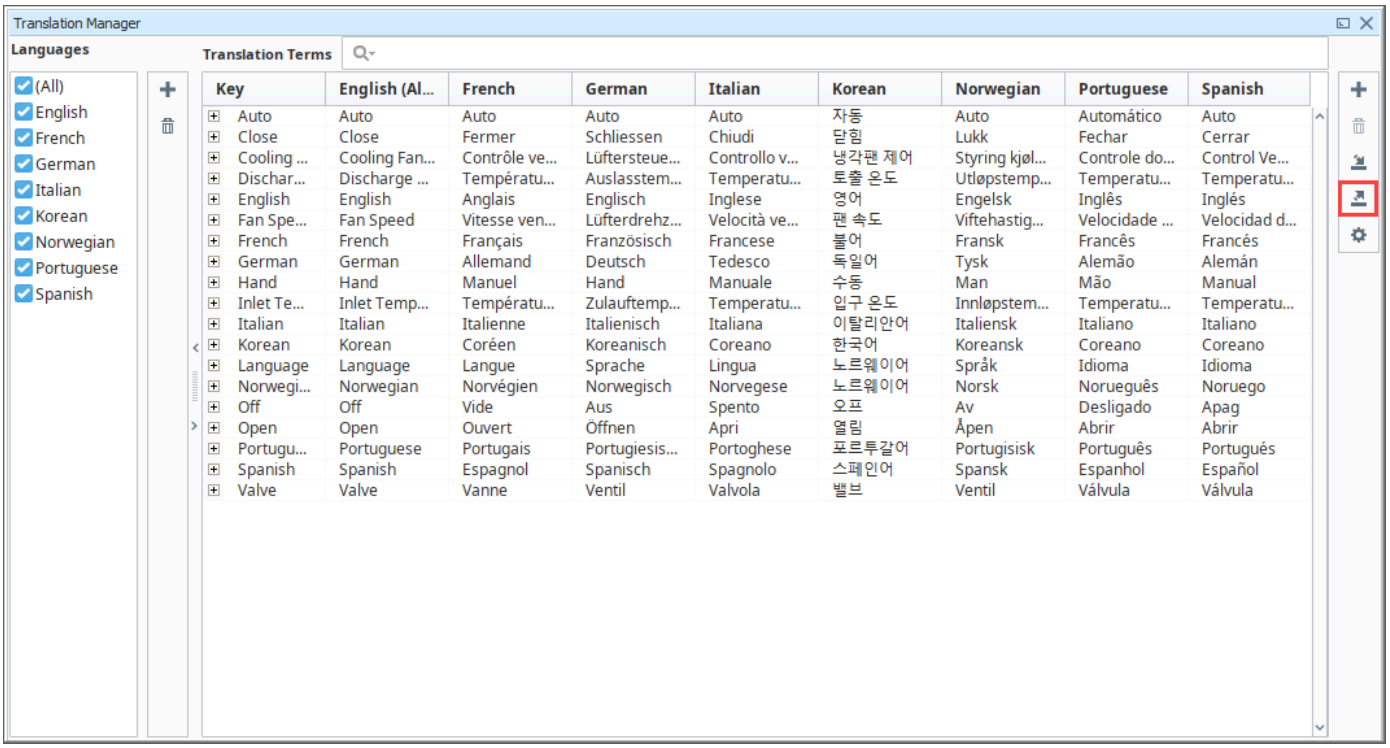
Export Terms

You can export all terms in the Translation Manager or only selected terms.

1. In the Designer, go to the menu bar and select **Tools** > **Translation Manager**.



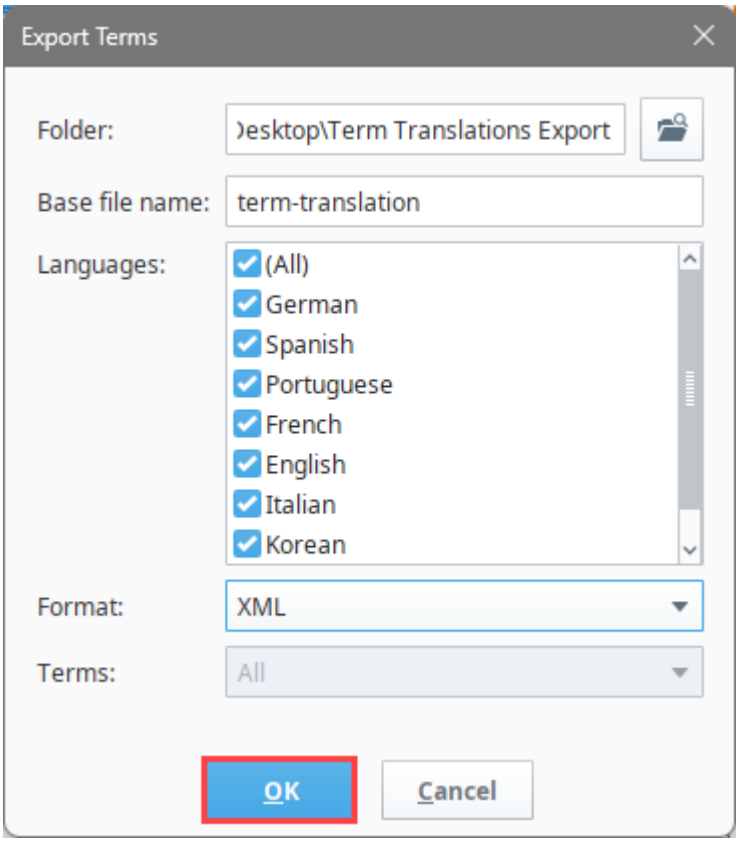
2. Click the **Export Terms** icon on the right side of the Translation Manager.



3. In the Exported Terms dialog, configure the following:

    - **Folder**: Choose where to save the exported files.
    - **Base file name**: Enter the base name for use for the exported files.
    - **Languages**: Select the languages you want to export. A separate file will be created for each language.
    - **Format**: Choose **PROPERTIES** or **XML**. XML supports UTF-8 encoding.
    - **Terms**: **Select All** or **Selected** depending on whether you want to export every term or only those highlighted.

4. Click **OK** to generate the files.



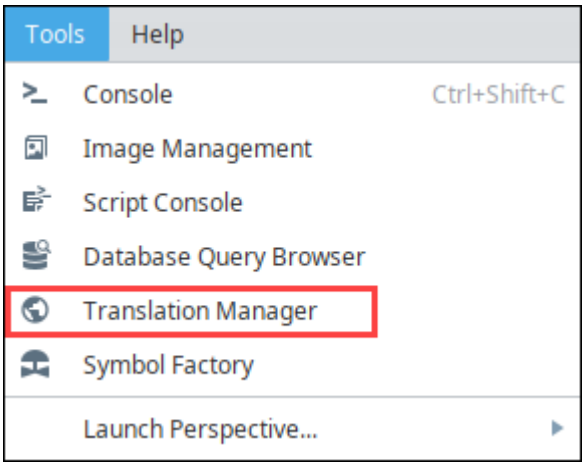
5. Go to the export folder and confirm the files were created and verify the contents. For this example, an XML file was exported.

```
▼<properties>
   <comment>Locale: en</comment>
   <entry key="Italian">Italian</entry>
   <entry key="Korean">Korean</entry>
   <entry key="German">German</entry>
   <entry key="Fan Speed">Fan Speed</entry>
   <entry key="Hand">Hand</entry>
   <entry key="Off">Off</entry>
   <entry key="Open">Open</entry>
   <entry key="Auto">Auto</entry>
   <entry key="English">English</entry>
   <entry key="Cooling Fan Control">Cooling Fan Control</entry>
   <entry key="French">French</entry>
   <entry key="Inlet Temperature">Inlet Temperature</entry>
   <entry key="Portuguese">Portuguese</entry>
   <entry key="Language">Language</entry>
   <entry key="Valve">Valve</entry>
   <entry key="Spanish">Spanish</entry>
   <entry key="Close">Close</entry>
   <entry key="Norwegian">Norwegian</entry>
   <entry key="Discharge Temperature">Discharge Temperature</entry>
 </properties>
```

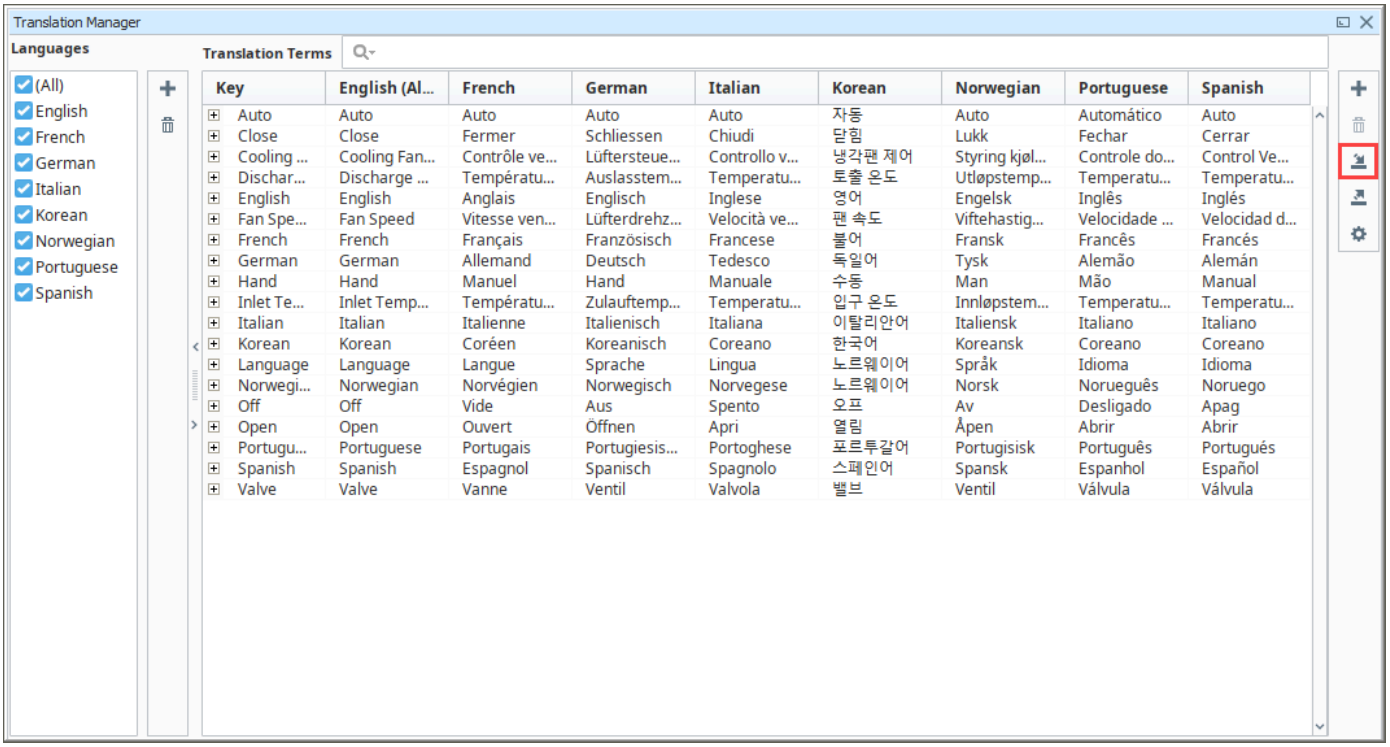
Import Terms

You can import translation files back into the Translation Manager to quickly populate terms in another project or environment.

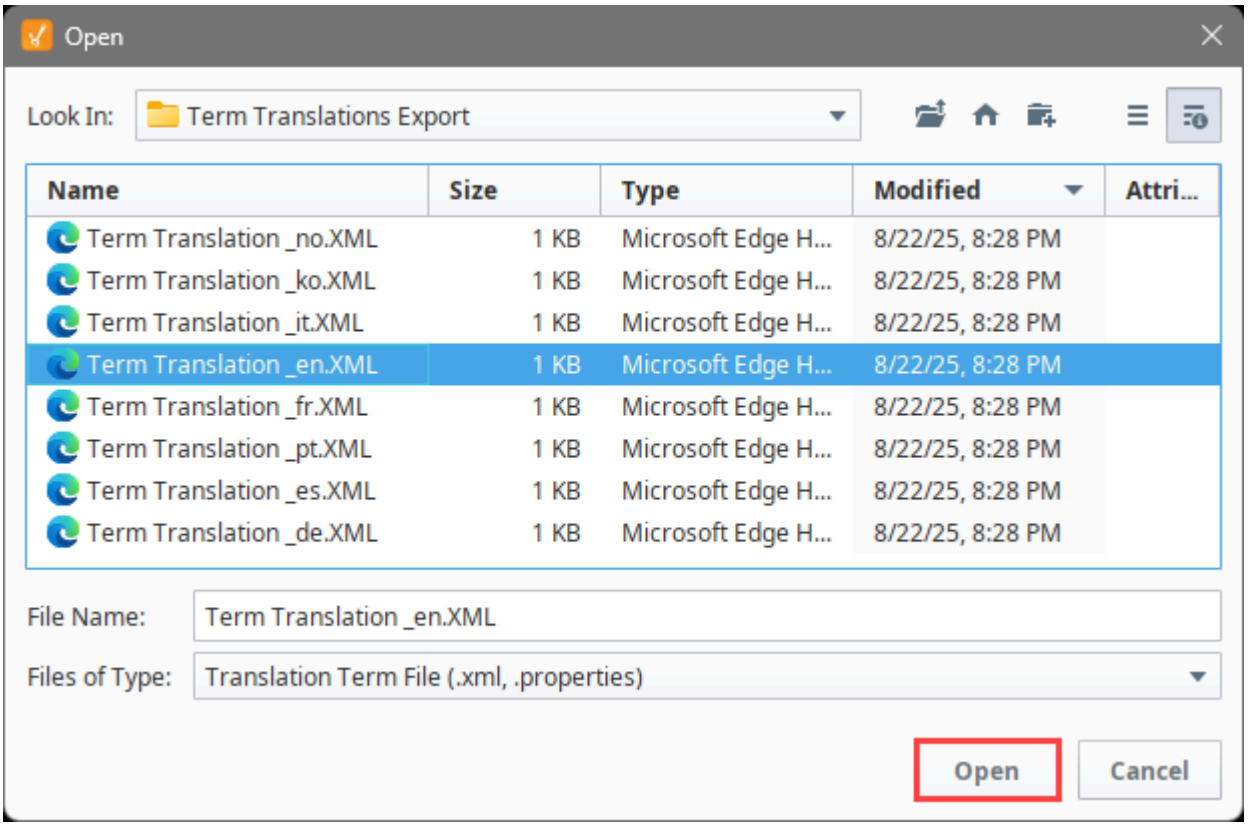
1. In the Designer, go to the menu bar and select **Tools** > **Translation Manager**.



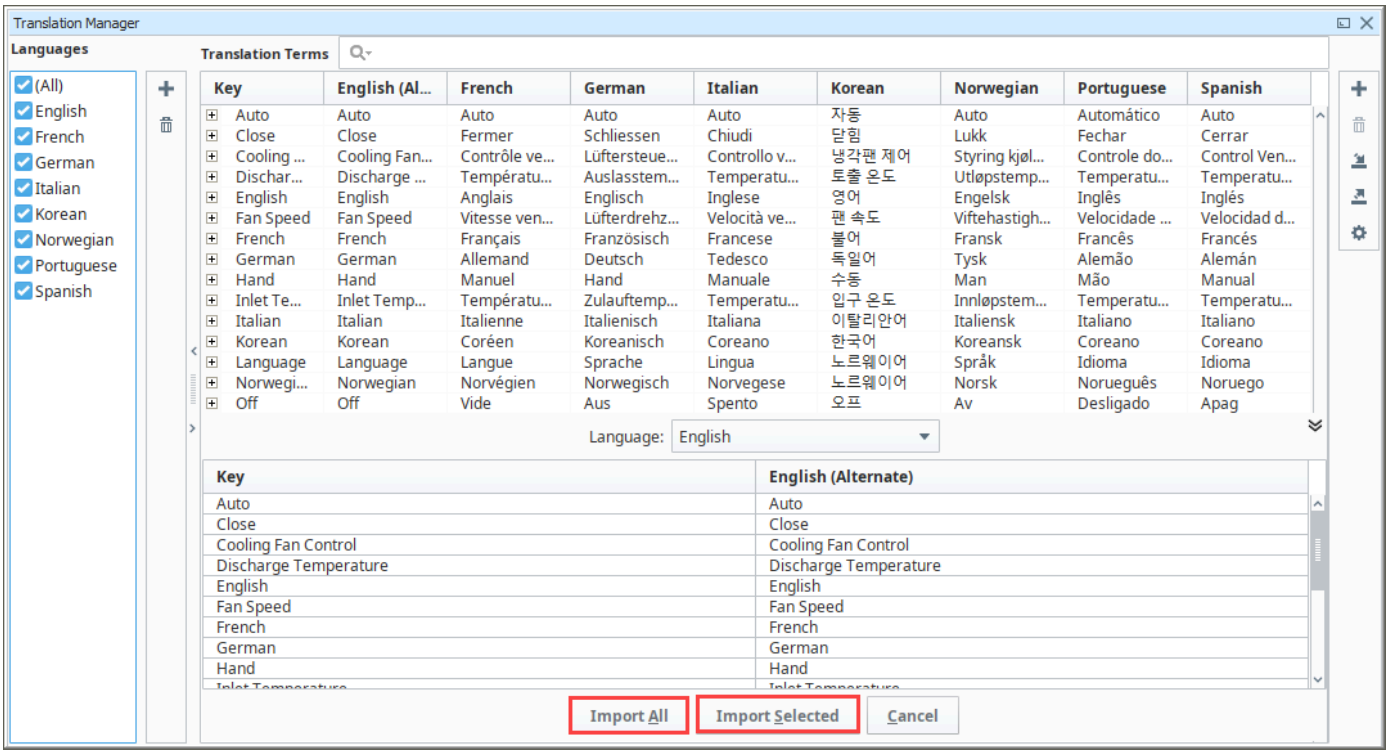
2. Click the **Import Terms** icon on the right side of the Translation Manager.



3. In the file browser, select the translation file you want to import, and click **Open**.



4. In the Import Terms dialog, choose either **Import All** or **Import Selected** depending on whether you want to import every term or only those highlighted.



5. Verify the imported terms appear in the Translation Manager.

> **IMPORTED TERMS OVERWRITE EXISTING TERMS**
>
> If a term already exists in the Translation Manager, importing will overwrite it with the values from the file.

Switching the Current Language

Language Selection

Because Ignition has multiple visualization systems, each system has separate ways to switch the current language.

Vision

In Vision, the language can be selected using the language selector dropdown before logging in to the client, or can be added to the client in the form of the language selector component.

Perspective

In Perspective, the language is selected by changing the locale of the session by modifying the session.props.locale. This can be written to using a script when the session first opens, or something can be bound to it that the user can change.

Expression and Scripting Functions

You can look up translations using the following functions:

- Expression Function
  - `translate()`
- Scripting Functions
  - `system.util.translate()`
  - `system.util.modifyTranslation()`

Translations are matched by looking for the base language value in the translation database. This is especially useful for message boxes and other warnings or errors that you show in your scripts.

System Considerations During Translation

The inherent operating system may affect Ignition's ability to provide a complete translation in certain circumstances.

For example, suppose a user selected the Spanish option from their Language Selector component. An English to Spanish translation term exist for the word "Information" as well as "Start." The message box script takes two parameters: one for the content while the other parameter is optional. If the second parameter is not included in the function then the resulting title of the message box will include the English word "Information." The following code is executed on a button clicked event handler.

```
message = system.util.translate("Start")
system.vision.showMessage(message)
```

What results is a message box with a translated content while the title remains in English. In addition, the accept button is translated as well even though the English word for "OK" does not exist in the Translation manager. This is due to the system level translations that exist for inherent language support.

A similar occurrence exist with the file open dialog window. The following code is executed on a button clicked event handler.

```
system.file.openFile()
```

The result is a translation for the title as well as the open and cancel buttons while the references to the "File Name" and the "Files of Type" remain in the operating system's language and thus are beyond the scope of Ignition's ability to translate.

Similar occurrences will appear in print and error dialog boxes.

Localization Keys

Ignition's localization features allow you to present your project in multiple languages without duplicating views or windows. This page outlines best practices for working with project terms, component overrides, naming keys, and managing long text.

Project Terms vs Component Terms

Project terms are shared across all components in the project. Use them for any text that appears in multiple places to avoid duplication and ensure consistency.

Component terms apply only to a single component. Use them when a component needs wording that differs from the project-wide translation. If both a project term and a component term exist with the same key, the component term takes precedence.

> **IMPORTANT**
>
> Review component definitions if translations do not appear as expected.

Using Naming Conventions with Keys

Always define a **key** for each translation term instead of using literal text. Keys act as stable identifiers that map to the correct translation, even if the visible text changes. Use unique keys that are unlikely to conflict with regular words, such as by prefixing with `#` or using dot notation like `dialog.save_changes`. Once a key is in use, keep it stable. Renaming a key breaks its mapping to translations. Keys are especially helpful for long text strings such as introductions or paragraphs, since they prevent accidental matches across multiple components.

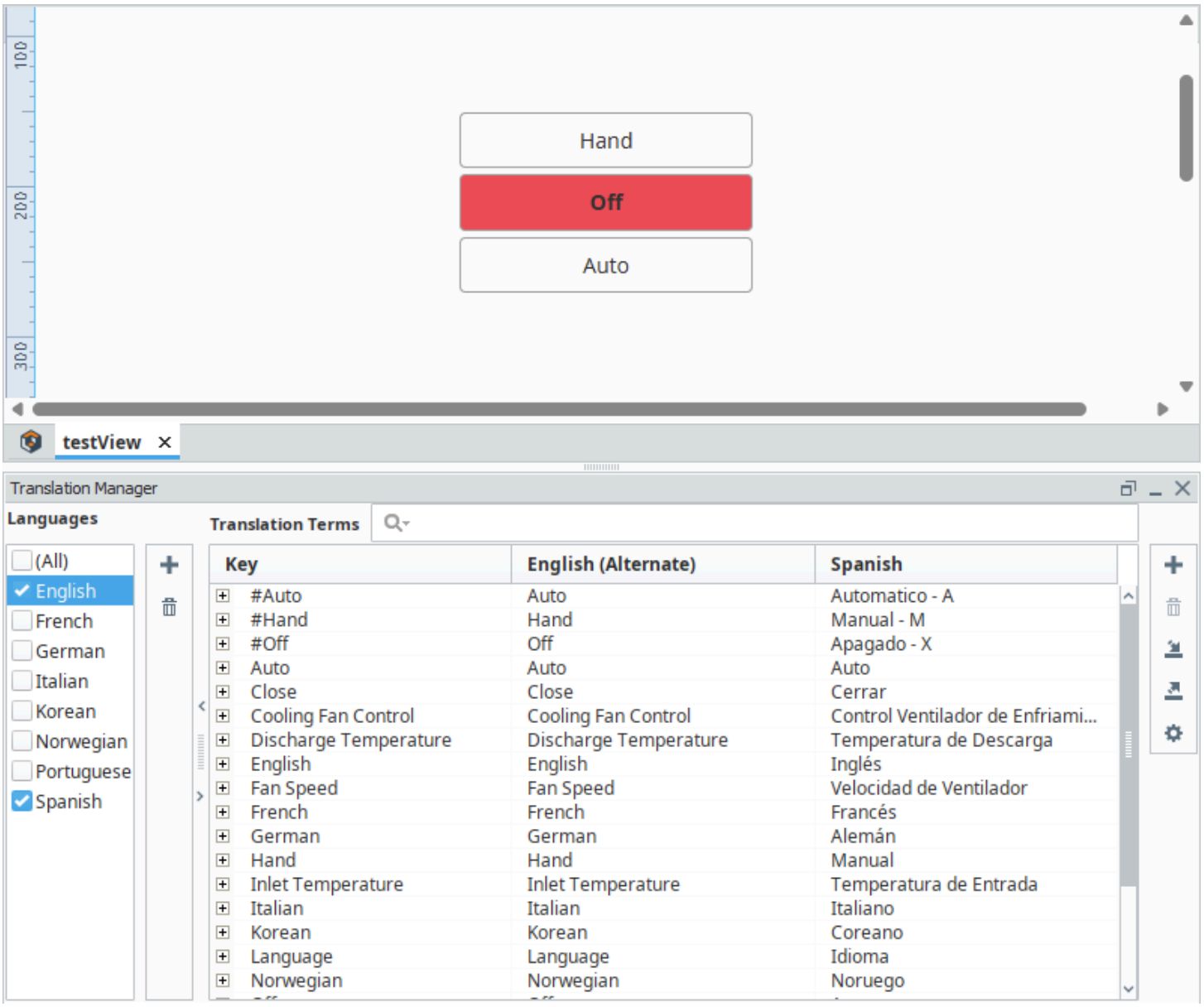
Preventing Unwanted Translations

Using common words as keys (such as “Auto” or “Off”) can sometimes trigger unintended translations if a project term with the same text exists. To avoid this, you can define the values as unique keys by prefixing them with `#`. This convention applies in both Perspective and Vision, although the configuration steps differ between the two modules.

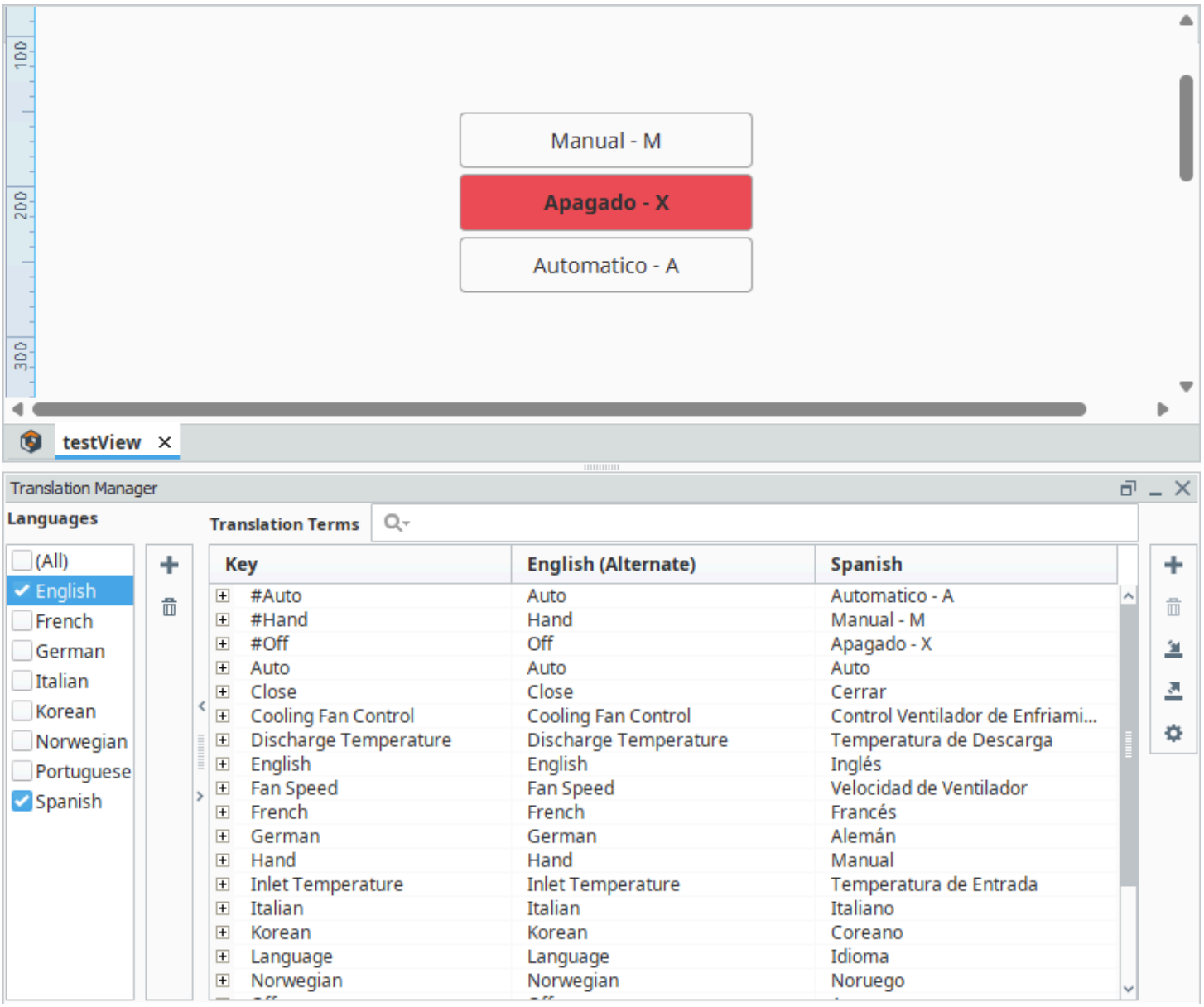
Perspective

In Perspective, the Multi-State Button component's states, **Hand**, **Off**, and **Auto** are defined in the `states[#].text` property as `#Hand`, `#Off`, and `#Auto`. Using the prefix ensures these words are treated as keys rather than matching existing project terms.

The keys appear in the Translation Manager, where translations are defined in English and Spanish. When `session.props.locale` is set to `en-US`, the button displays the English values.

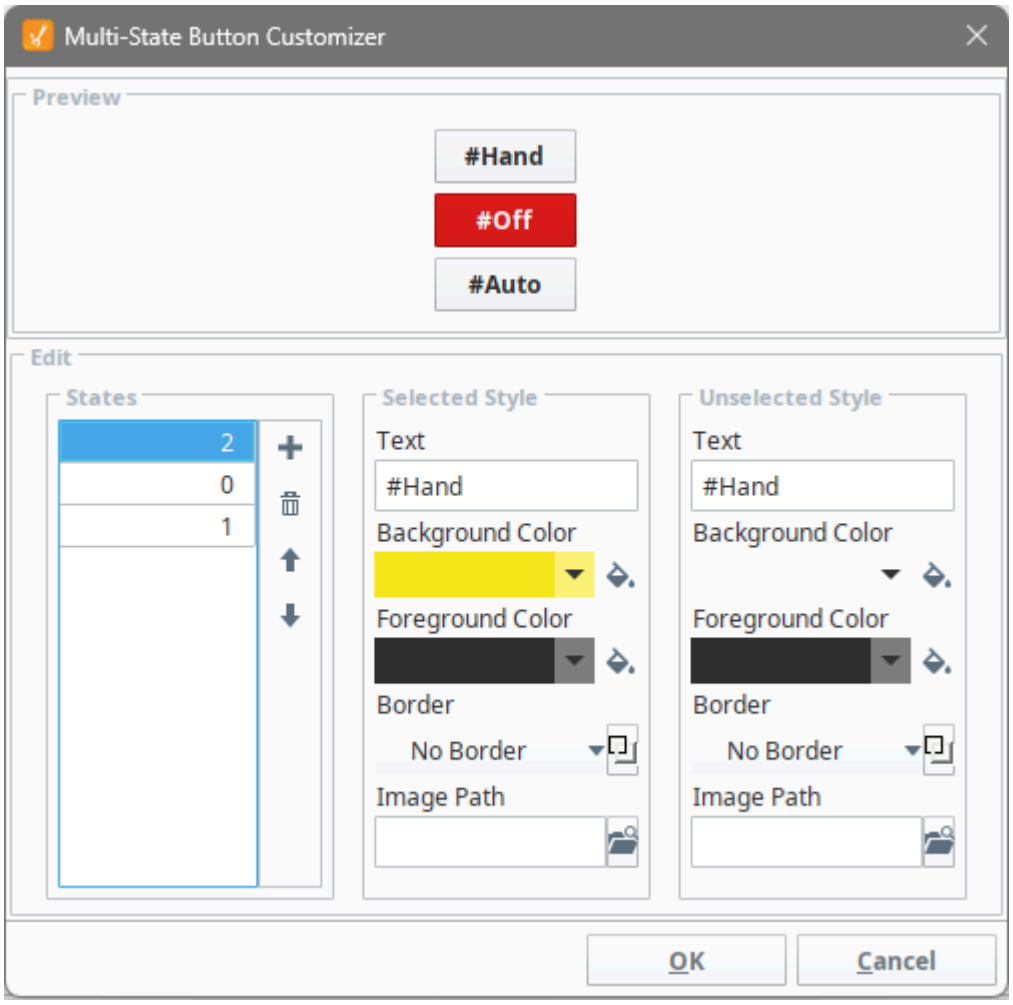


When `session.props.locale` is switched to `es-US`, the button displays the Spanish translations.



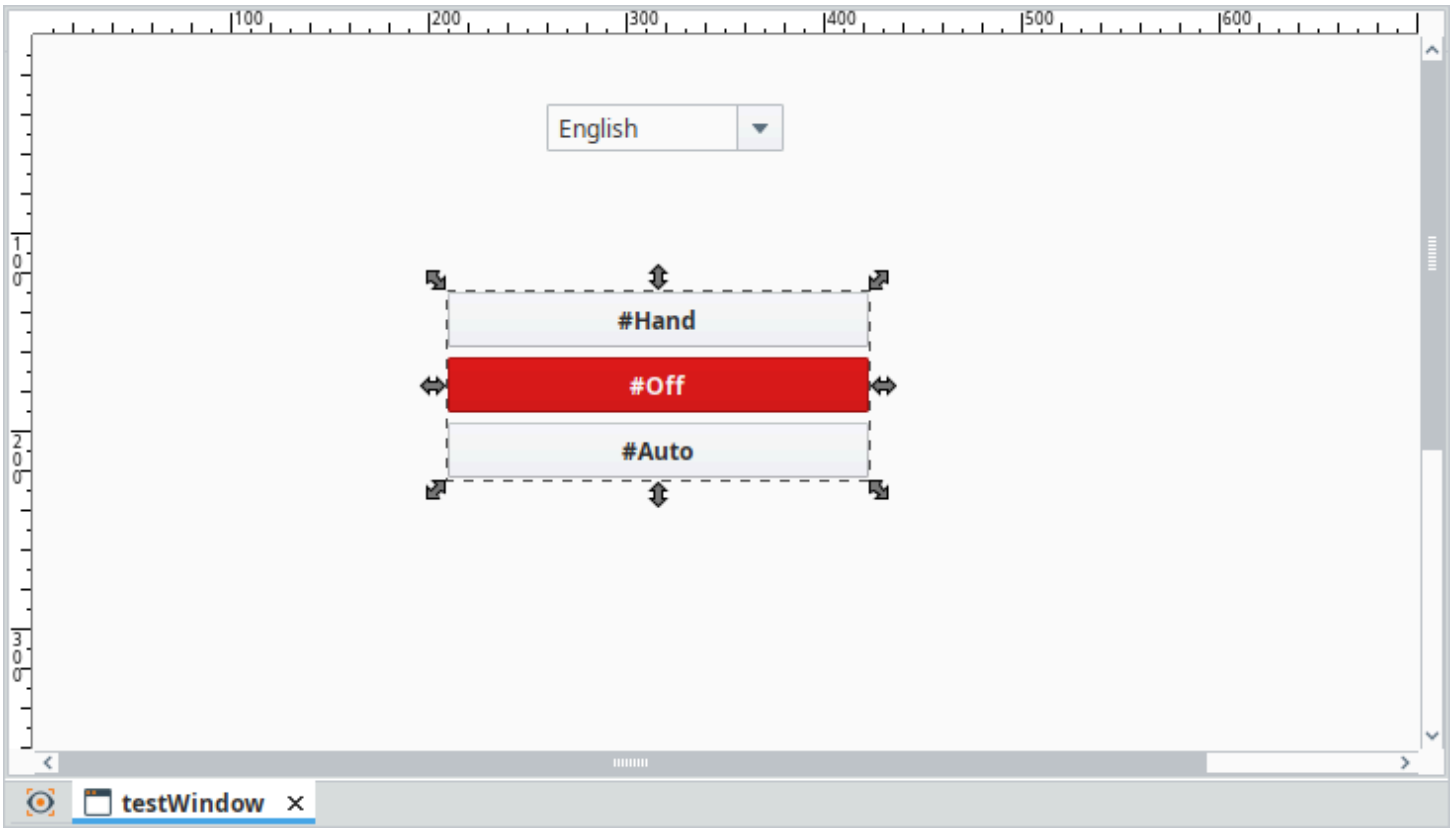
Vision

In Vision, the Multi-State Button component's states, **Hand**, **Off**, and **Auto** are defined in the **Multi-State Button Customizer** as `#Hand`, `#Off`, and `#Auto`. Using the prefix ensures these words are treated as keys rather than matching existing project terms.

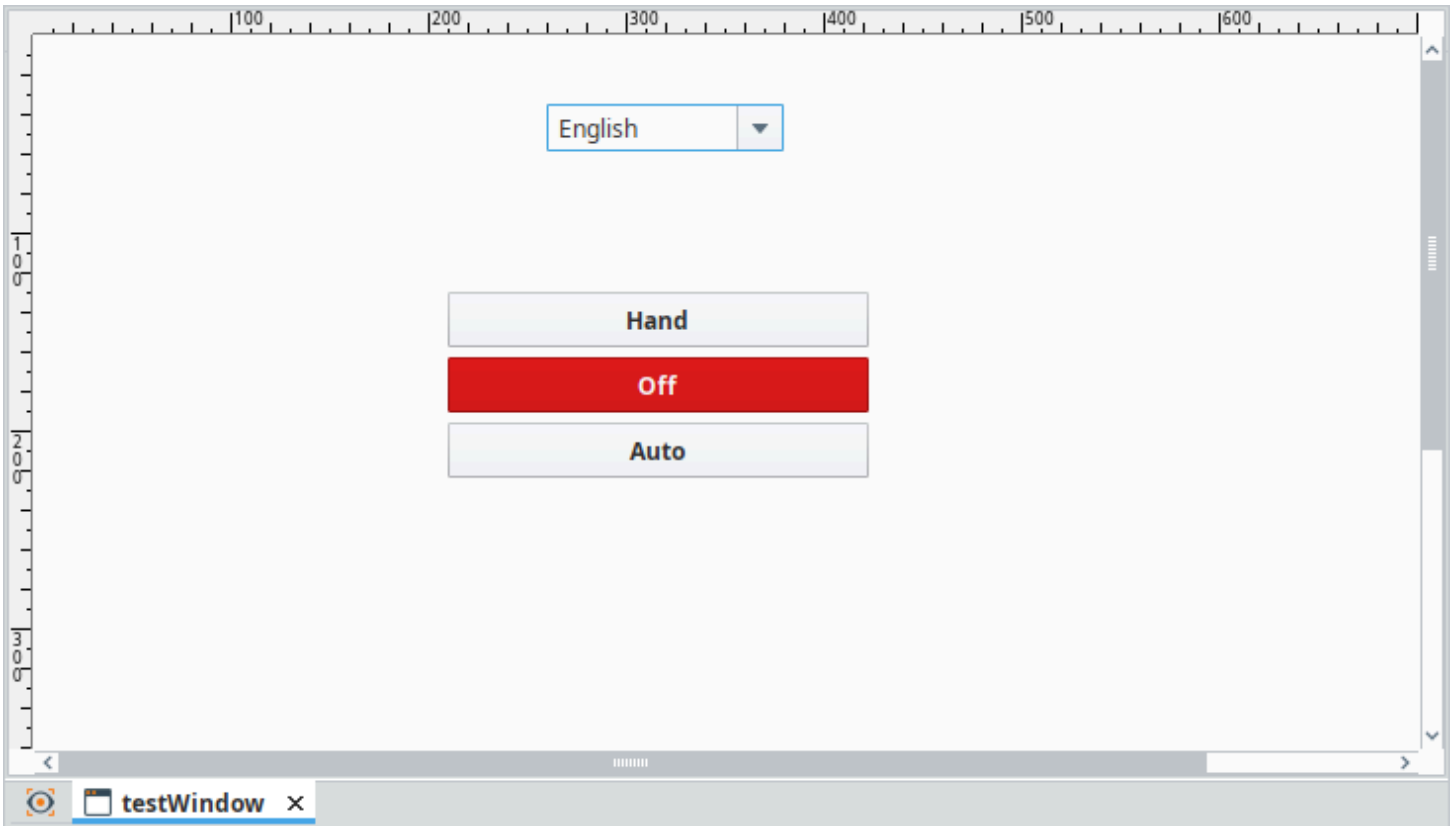


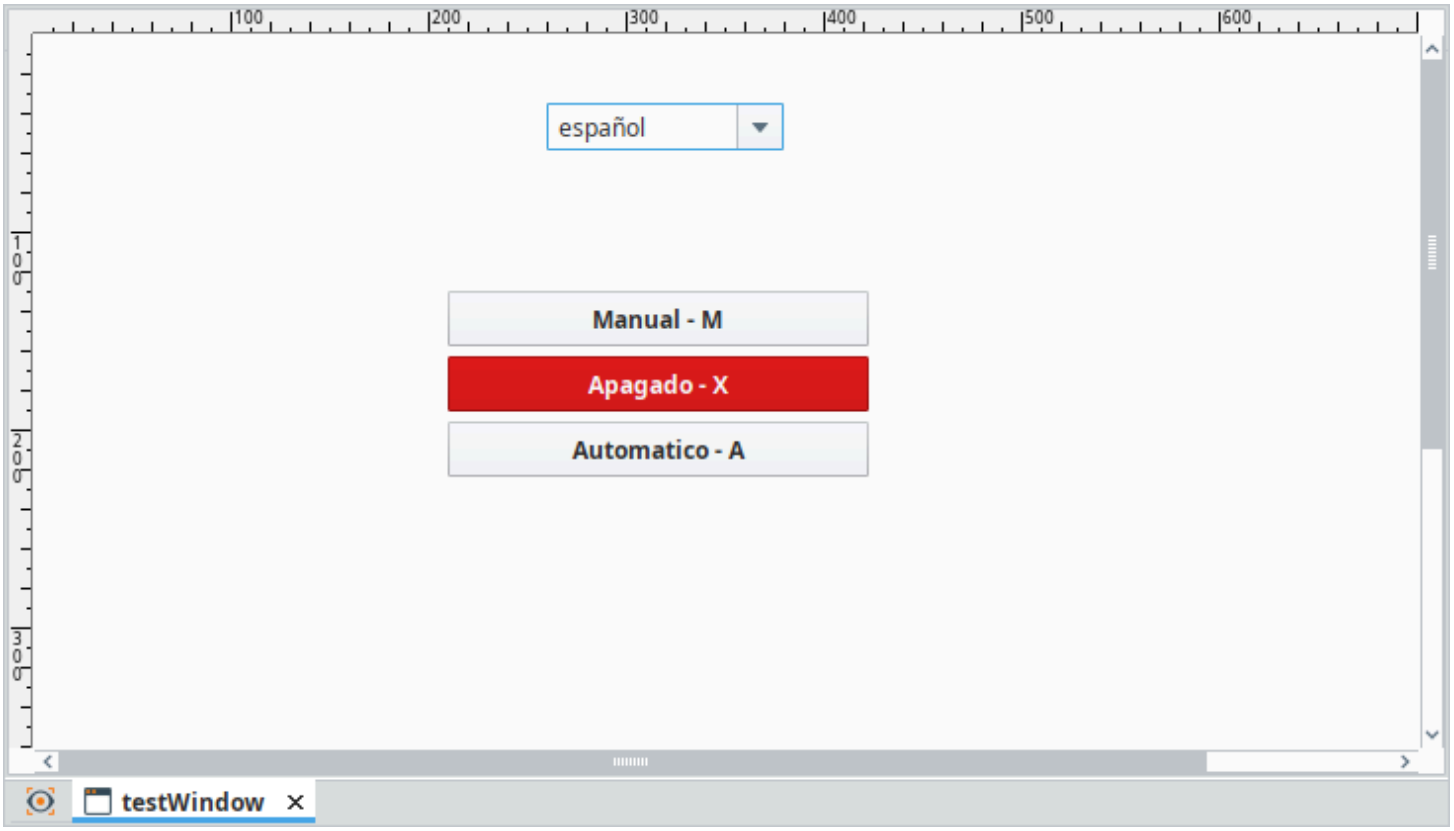
The keys appear in the Translatable Terms panel, where translations are defined in both the component and global columns.

In Design Mode, the button displays the untranslated keys.



In Preview Mode, the button displays the translated states in English and Spanish.

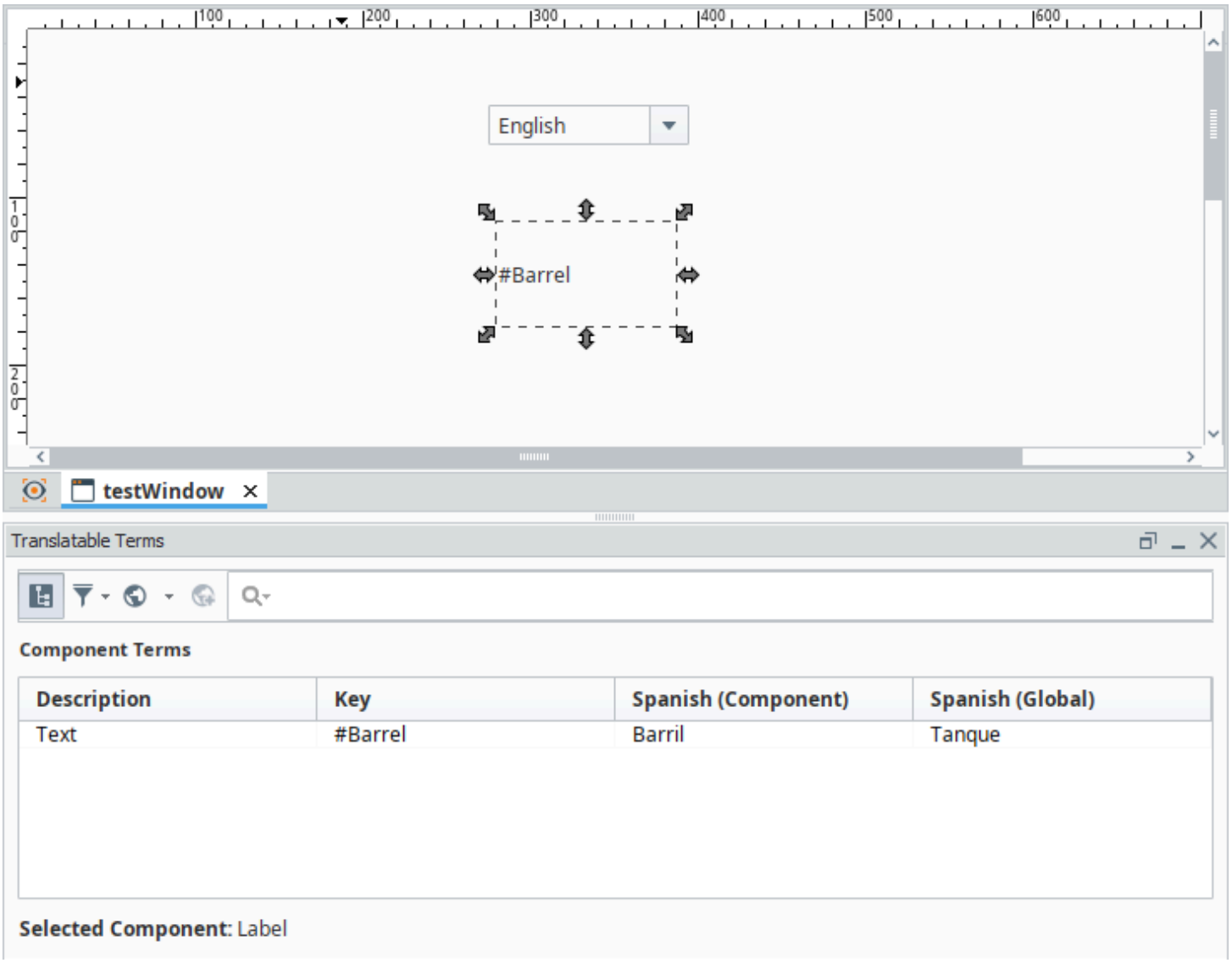




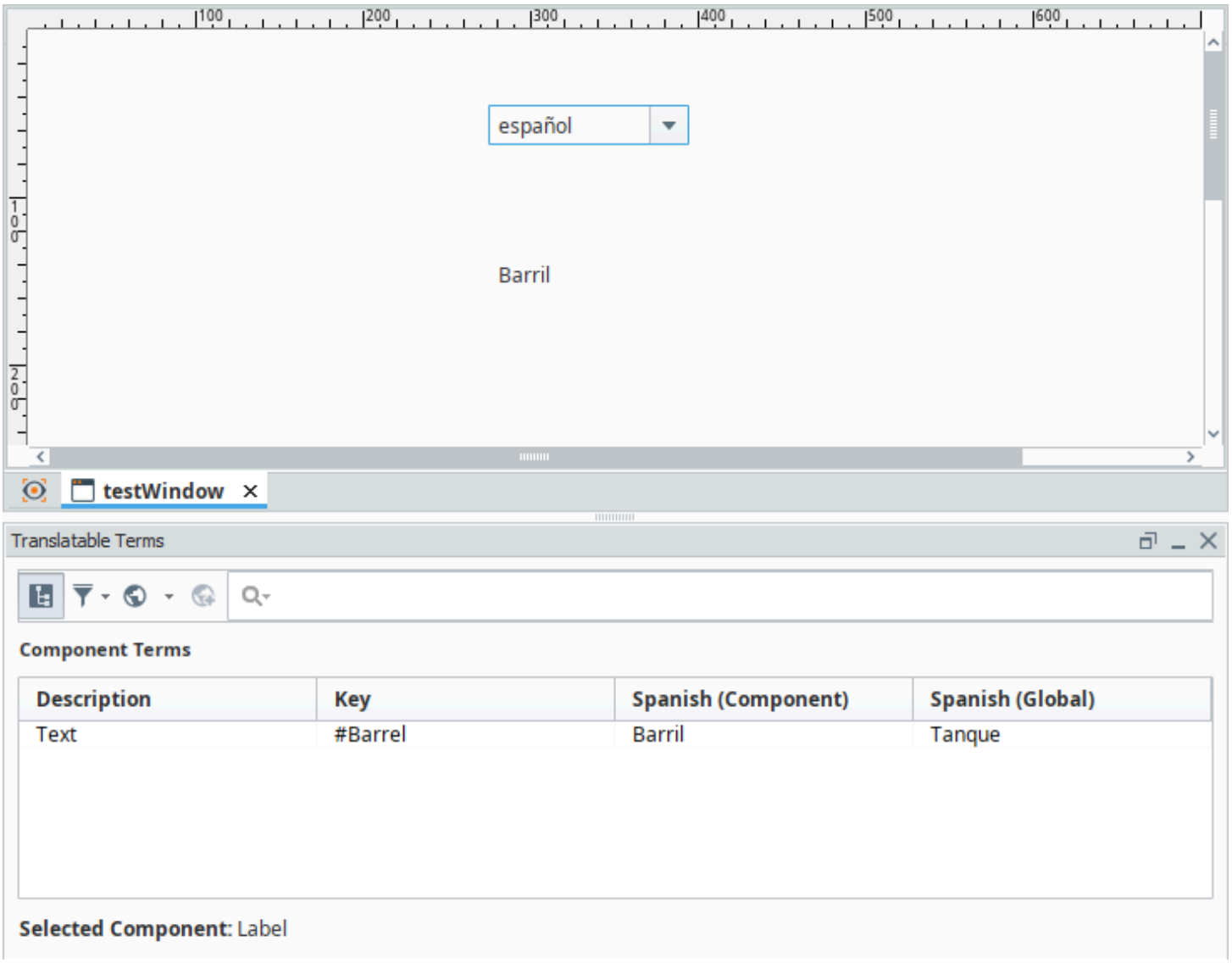
Overriding Vision Components

In this example, a Label component was added to the window and its text was set to `#Barrel`. This created a key in the Translatable Terms panel. The global term translates `#Barrel` to **Tanque**, while a component term with the same key translates it to **Barril**. Because component terms take precedence, the Label displays **Barril** instead of the project-wide value.

In Design Mode, the Label shows the raw key.



In Preview Mode, the Label displays the component translation, overriding the global term.



Overriding Built-in Vision Messages

In addition to project and component terms, some system dialogs in Vision also use fixed text that can be overridden. These messages are not exposed as component properties, but you can still localize them by adding their text as keys in the Translation Manager. The most common examples are the **Access Denied** and **Screen Locked** dialogs.

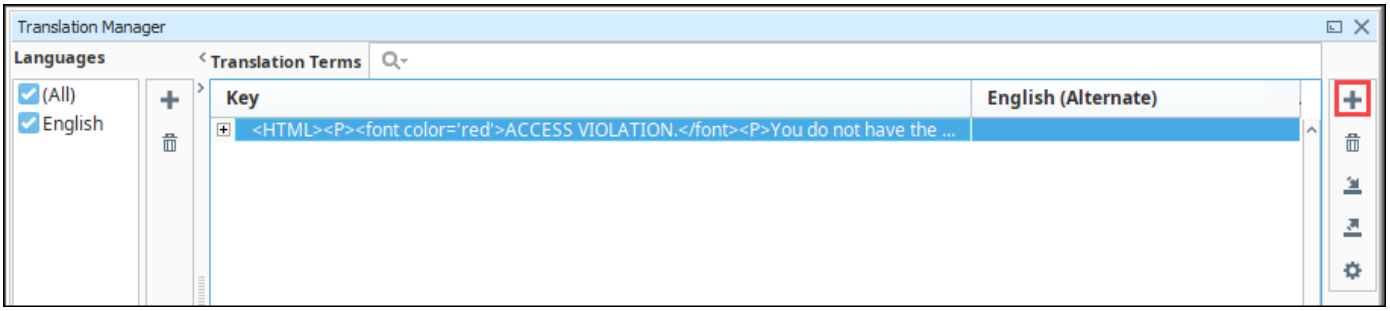
Access Denied Message

When a Vision window is restricted with **Do Not Open**, an Access Denied popup appears. The text in this popup is fixed, but you can override it by adding a key in the Translation Manager.

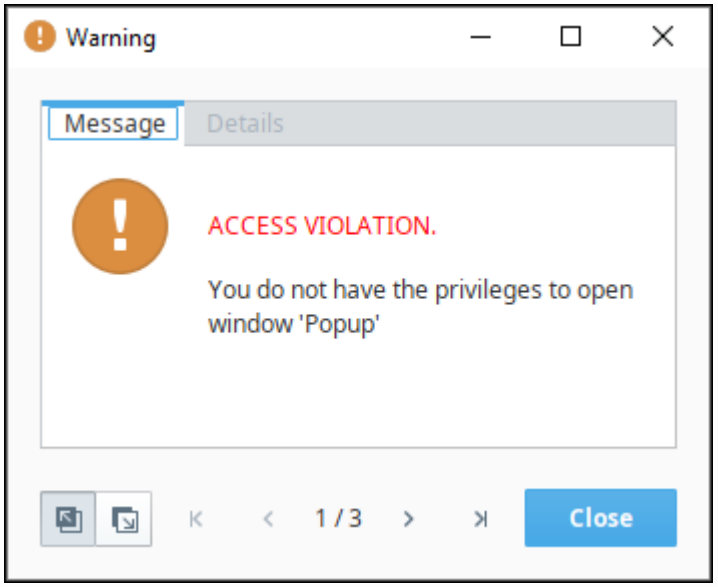
1. In the Designer, go to **Tools** > **Translation Manager**.
2. Click the **plus** ➕ icon, paste the following value into the **Add Item** field, and click **OK**.

```
Access Violation - Key
```

```
<HTML><P><font color='red'>ACCESS VIOLATION.</font><P>You do not have the
privileges to open window '%s'.
```



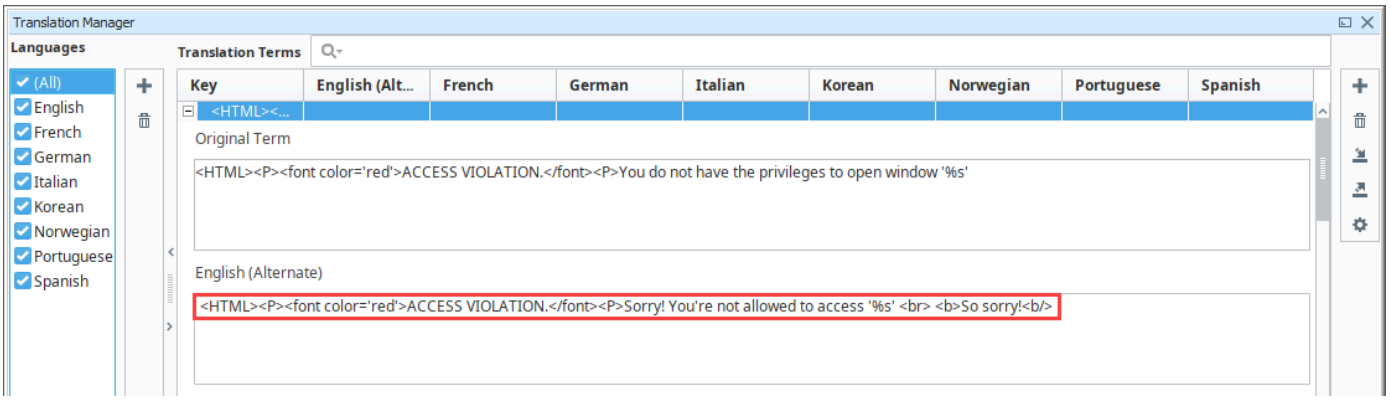
> **NOTE**
>
> The `%s` placeholder is replaced with the window name at runtime.



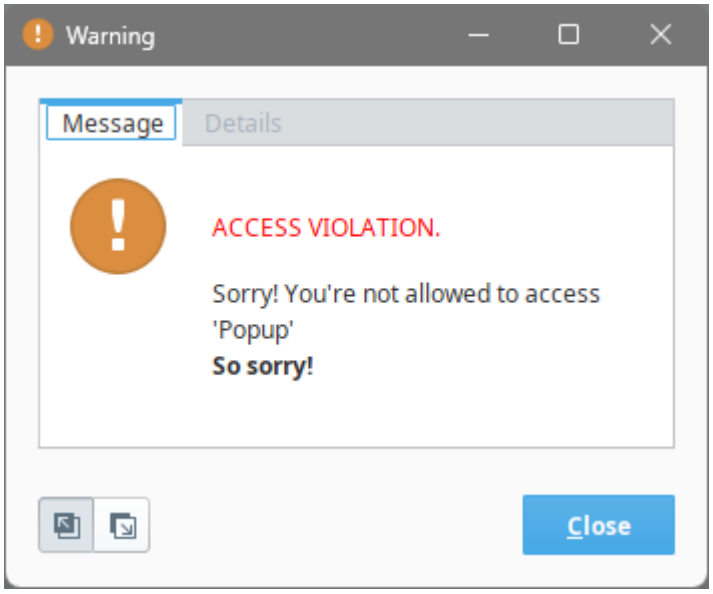
3. Add a translation for the key. For example:

Access Violation - Translated Term

```
<HTML><P><font color='red'>ACCESS VIOLATION.</font><P>Sorry! You are not
allowed to access '%s'<br><b>So sorry!</b>
```



Now, when a user without the required permission attempts to open the window, the popup displays the translated text.



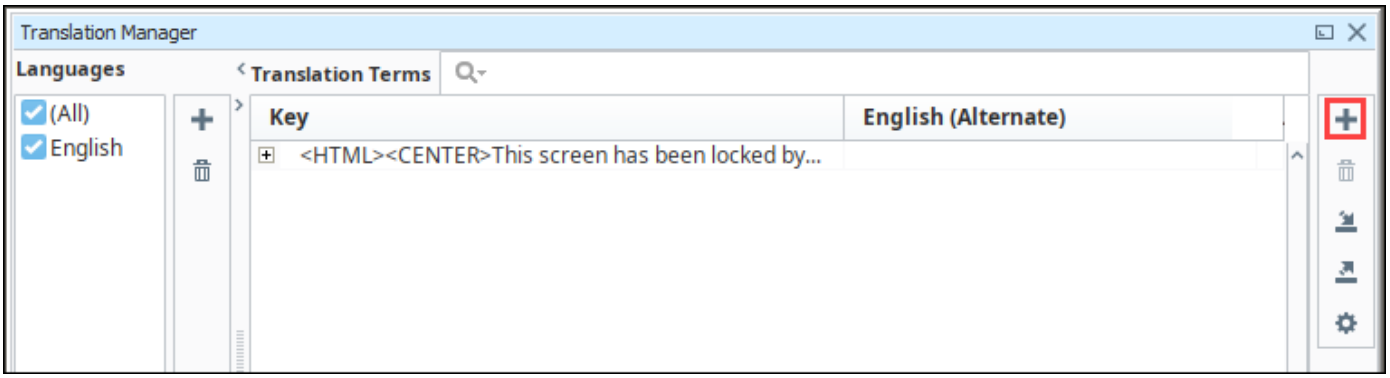
Screen Locked Message

The **system.vision.lockScreen** function in a **Client Timer Script** displays a popup when the client is locked. The message text is not configurable, but you can override it with a key in the Translation Manager.

1. In the Designer, go to **Tools** > **Translation Manager**.
2. Click the **plus** ✚ icon, paste the following value into the **Add Item** field, and click **OK**.

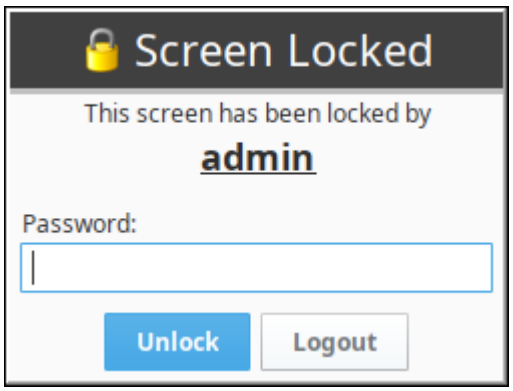
Screen Locked - Key

```
<HTML><CENTER>This screen has been locked by<BR><font size=+1><b><u>%s</u></b>
```



NOTE

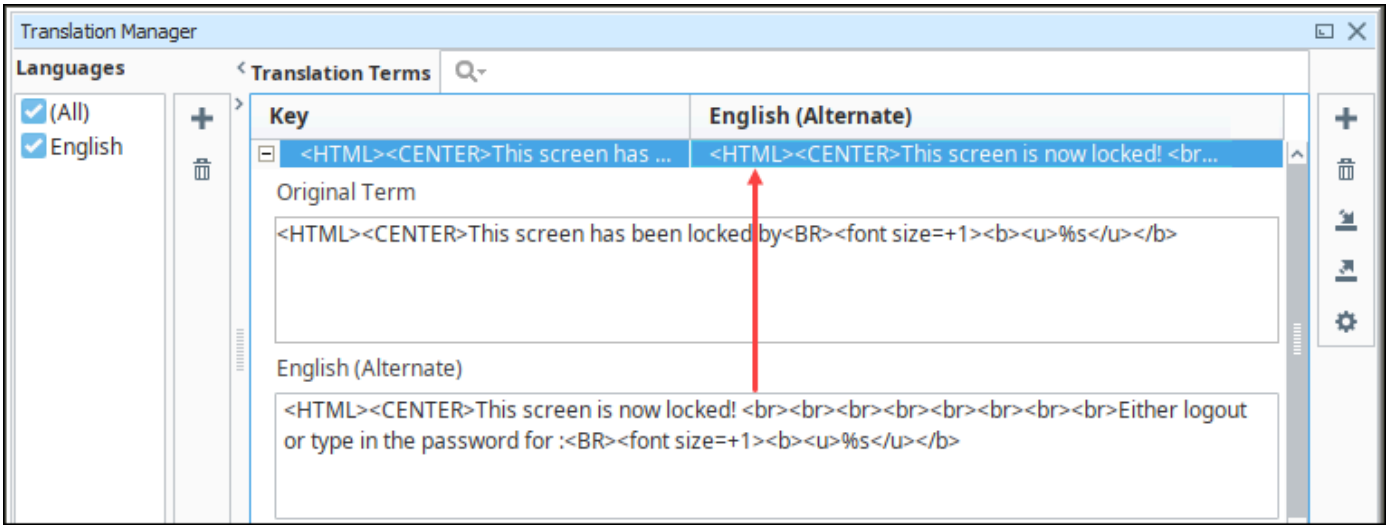
The `%s` placeholder is replaced with the current username at runtime.



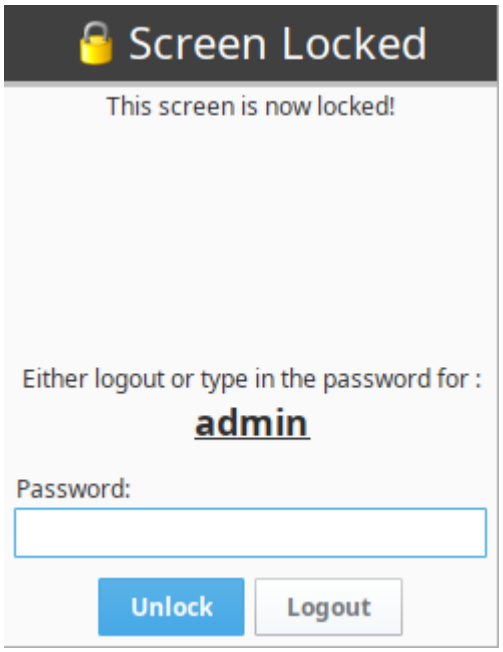
3. Add a translation for the key. For example:

Screen Locked - Translated Term

```
<HTML><CENTER>This screen is now locked!<br><br>Either log out or type in the password for :<BR><font size=+1><b><u>%s</u></b>
```



Now, when the screen is locked in the client, the popup displays your translated text.



Wrapping Long Text in Vision

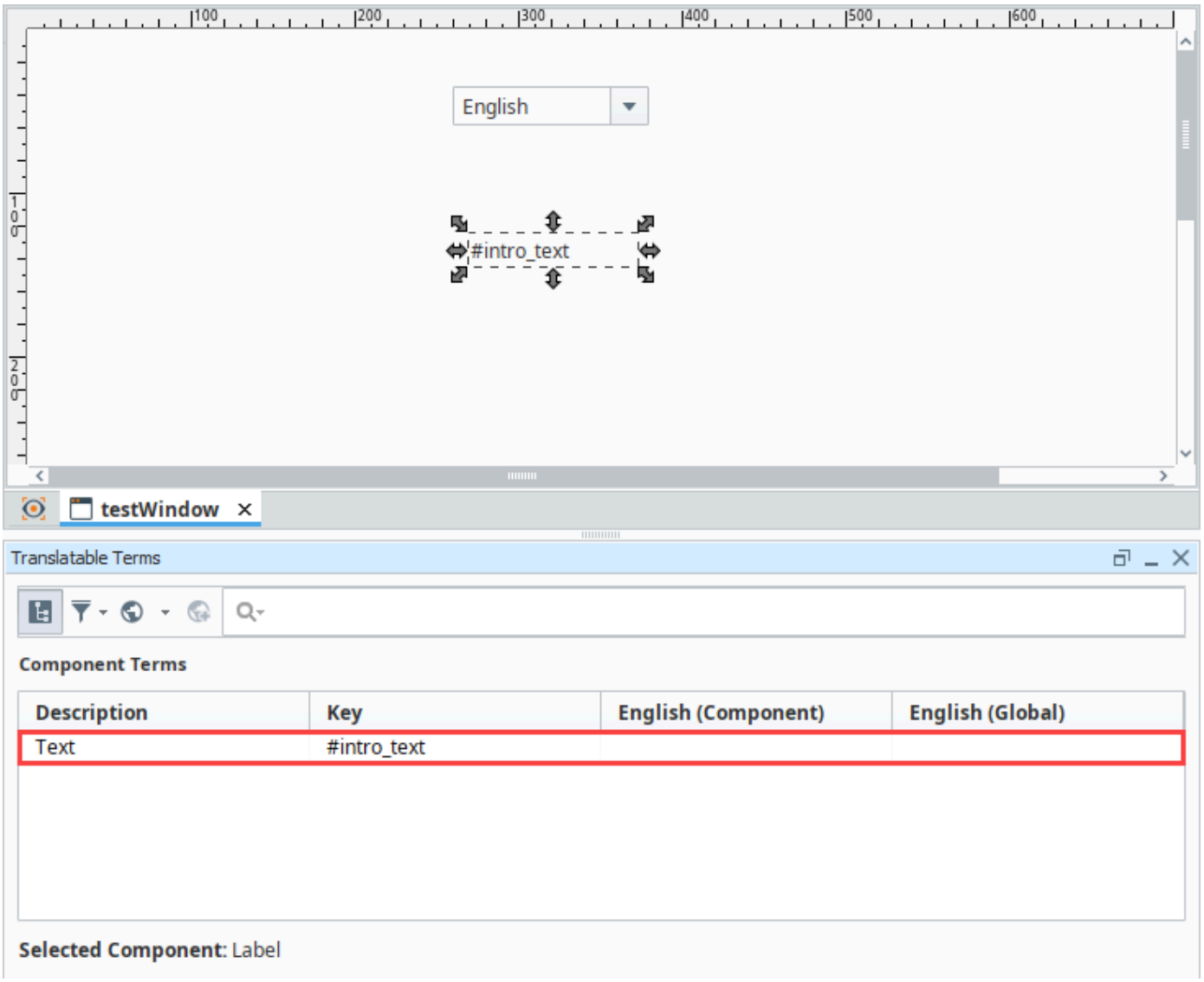
In Vision you can use basic HTML tags to wrap long text values inside components that display text. This allows translations with long sentences or paragraphs to break across multiple lines.

In this example, a Label component is configured with the key `#intro_text`, and HTML wrapping is used so its text displays neatly in both English and Spanish.

1. In the **Property Editor** of a Label component, set the `text` property to a unique key. In the example below, we use `#intro_text`.

2. Open the **Translatable Terms** panel and double-click the term row to add translation values for each language.

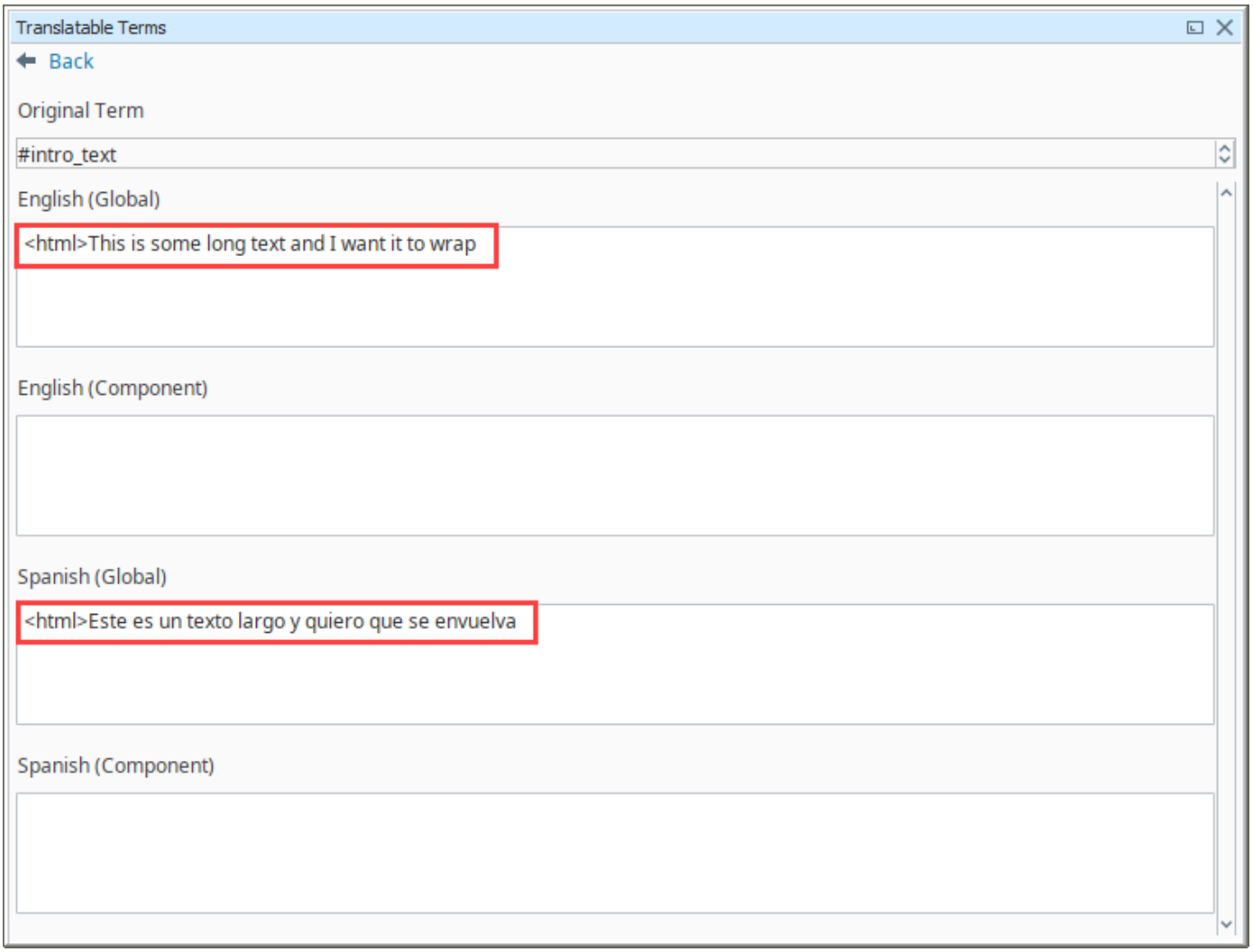
> **NOTE**
>
> You can open the **Translatable Terms** panel by right-clicking the component and selecting **Translations**.



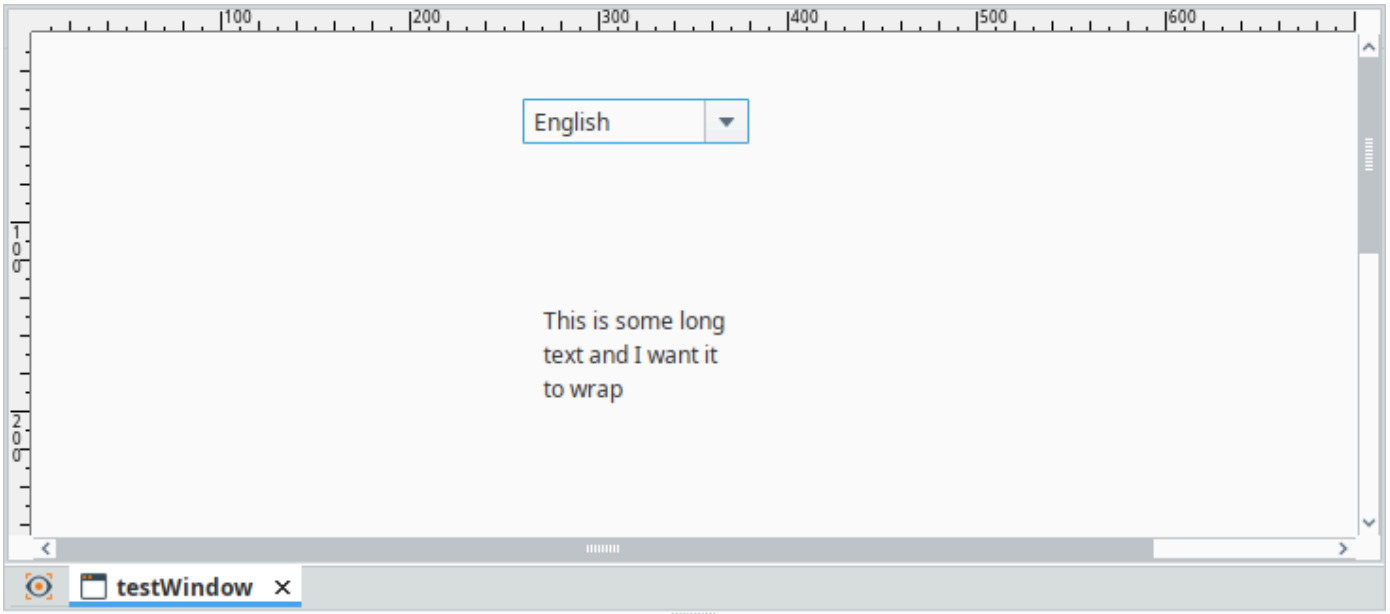
3. Begin the value with `<html>` to enable wrapping.

Translation Values

```
# English
<html>This is some long text and I want it to wrap

# Spanish
<html>Este es un texto largo y quiero que se envuelva
```



4. In **Preview Mode**, the Label displays the text across multiple lines in English and Spanish. If the component is not large enough to fit the text, only part of the wrapped content will be visible.

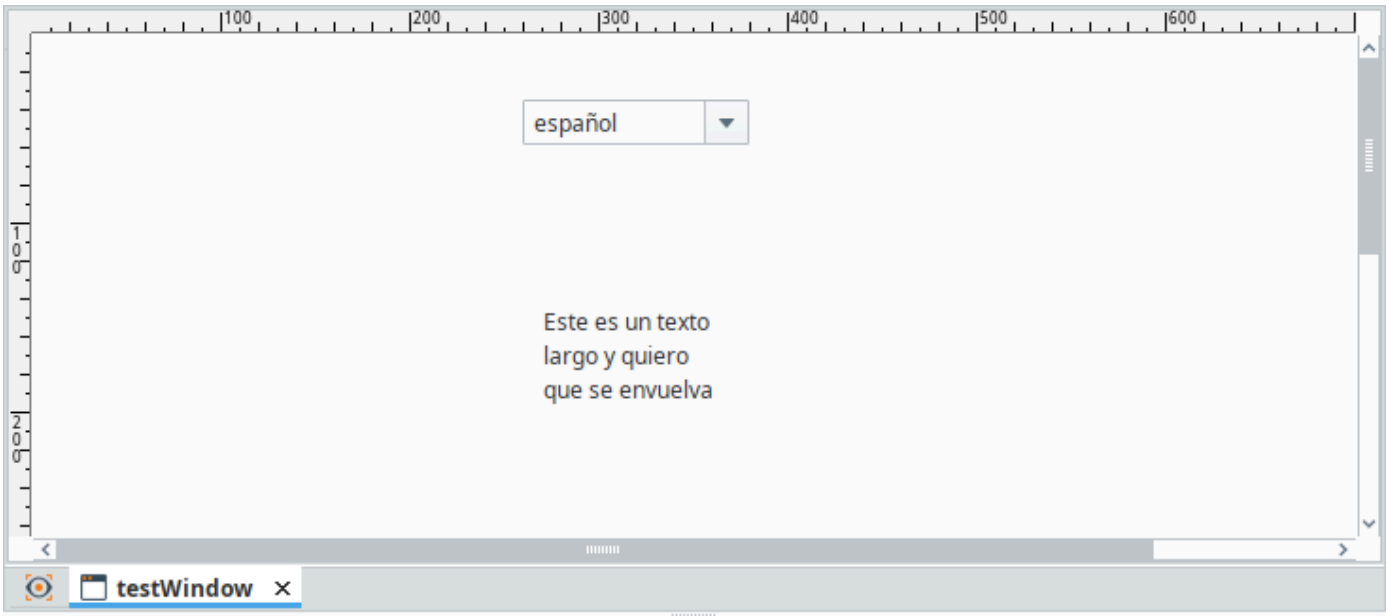


Translatable Terms

Component Terms

Description	Key	English (Component)	English (Global)
Text	#intro_text		<html>This is some long t...

Selected Component: Label



Translatable Terms

Component Terms

Description	Key	español (Component)	español (Global)
Text	#intro_text		<html>Este es un texto lar...

Selected Component: Label

Locale Formats and Language Tags

Several scripting functions, expression functions, and properties in Ignition accept or return a **locale** as a string. Because Ignition has to remain backwards compatible with older projects, more than one string format is accepted, and different parts of the product prefer different formats. This page explains the formats, where they are used, and recommended formats when writing new code.

Locale Objects and String Formats

Since Ignition runs on Java, Ignition locales are Java `Locale` objects. A `Locale` object determines how text, dates, and number values are translated based on a specified language. Regions can also be supplied to further define translation. For example, just English can be provided as the language, or it can be combined with a region, like the United States.

The two string formats that can be passed include BCP-47 language tags and the legacy underscore format.

BCP-47 Language Tag (Recommended)

A **BCP-47 language tag** uses a hyphen (`-`) to separate the language subtag from the region subtag:

- `en` — English
- `en-US` — English (United States)
- `es-ES` — Spanish (Spain)
- `fr-CA` — French (Canada)
- `en-IE` — English (Ireland)

The language subtag comes from the ISO 639 standard, and the region subtag from ISO 3166. Language tags are an open standard supported across the web, within Java, and within other systems that exchange data with Ignition. This is the recommended format to use when writing new code.

Legacy Underscore Format

The legacy format uses an underscore (`_`) instead of a hyphen:

- `en`
- `en_US`
- `es_ES`
- `fr_CA`

This is a Java-specific idiom rather than a general purpose, interoperable standard. It is still accepted in many places for backwards compatibility, but it is **not recommended** for new code. Some functions return this format, so you may encounter it when reading a locale back from Ignition.

> **NOTE**
>
> For a language with no region, such as `en` on its own, the two formats look identical because there is no separator to differentiate them. The distinction only appears once a region is included (`en-US` versus `en_US`).

Where Formats are Used

The table below summarizes what format each function or property accepts as input and returns as output. Note that when both formats are accepted, the language tag format is always preferred.

Location	Accepts	Returns	Preferred
`system.util.translate`	Both	—	Language tag
`system.util.modifyTranslation`	Both	—	Language tag
`translate` **expression function**	Both	—	Language tag
`system.vision.getLocale`	—	Underscore	—
`system.vision.getAvailableLocales`	—	Underscore	—
Perspective session `locale` **property**	Language tag	Language tag	—
User `Language` attribute via `system.user.editUser`	Both	Both	Language tag

CONVERTING FORMATS

If you need to compare or pass values returned from Vision functions in the underscore format to areas that expect language tags, simply convert the underscore to a hyphen.

Language Names Are Not Locale Identifiers

A locale identifier is a code such as `es` or `es-ES`, not the English (or native) name of the language. Passing a display name like `"Spanish"` or `"Español"` where a locale is expected will not match a defined locale, and the translation will silently fall back to the original term.

```
# Incorrect - "Spanish" is a language name, not a locale identifier
system.util.translate("Hello", "Spanish")

# Correct - use the language tag
system.util.translate("Hello", "es")

# Also correct - include a region when you need a specific variant
system.util.translate("Hello", "es-ES")
```

If you are unsure which locale codes are defined on your system, the locales you configure appear in the Translation Manager. See **Creating Translation Lists** for how languages are defined.

Expression Language and Syntax

Expression Language

The expression language is used to define dynamic values for component properties and **Expression tags**. Expressions often involve one or more other values that are used to calculate a final value. In most cases, expressions only return a value.

Expressions are also used for bindings in both Vision and Perspective. See the **Expression Bindings in Vision** or **Expression Bindings in Perspective** pages for more information.

Syntax

As its name suggests, everything in the expression language is an "expression". This means that everything returns a value: 5 is an expression, so is 5+1, and so are `{MyTags/TankLevel}` and `{MyTags/TankLevel}+1`. Expressions can be combined in many powerful ways. Let's take a look at how expressions are written.

More formally, an expression is any one of the following:

- Number
- Boolean
- String
- Bound tag
- Bound property
- Function call
- Dataset access
- Equation involving any of these

The sections below detail various syntax used in the expression language.

Literal Values

Literal values are things like numbers, booleans, and strings that are represented directly in the language. In the expression language, numbers can be typed in directly as integers, floating point

values, or using hexadecimal notation with a 0x prefix, such as:

```
42
8.456
0xFFC2
```

Strings are represented by surrounding them with double or single quotes. You can use the backslash character to escape quotes that you want to be included in the string, such as:

```
"This is a regular string"
'This one uses single quotes'
"This string uses \"escaping\" to include quotes inside the string"
```

In addition, the following escape characters are available:

Character	Description
\n	New line
\t	Tab
\r	Carriage return

```
// The words "Hello" and "User" will be placed on separate lines
"Hello\nUser"

// Each "\t" will inject a tab
"Lots\tOf\tSpace"
```

Boolean values are represented with True and False (without quotation marks). Note that boolean values are case insensitive, so true is the same as True. For simplicity, we recommend writing True and False, since Python uses similar casing for boolean values.

```
// Multiple ways to specify a true boolean value.
True
true
tRuE
TRUE
```

Another value you can specify is a null literal, using one of the following:

```
null
None
none
```

You can also use scientific notation to specify numeric constants, such as `1.3e5`.

Operators

You can use these arithmetic, logical, and bit-shifting operators to combine expressions.

Operator	Name	Description
//	Comments	Allows for comments following this operator.

-	Unary Minus or Subtraction	If both preceded by a number, then returns a value by subtra from the operand on the left of the operator:
		
		If preceded by anything else (or nothing, such as the start o followed by a number, this operator will negate the number.
		
		`// This will return 6.`
		`10 - 4`
		
		`// This will return -10.`
		`- (``15 - 5``)`

!	Not	Logical opposite of a boolean.
^	Power	Raises a number to the power of another number.
%	Modulus	Modulus or remainder of two numbers. a%b is the remainde
*	Multiply	Multiplies the number on the left of the operator by the numb operator.
/	Divide	Divides the number on the left of the operator by the numbe operator.

+	Add or Concatenation	If both operands are numbers, this will add them together. C arguments as strings and performs concatenation.
		
		`// This will return 10.`
		`4 + 6`
		
		`// This will return 'FirstSecond'.`
		`'First' + 'Second'`
		
		`// This will return '2Alarms'.`
		`2 + 'Alarms'`

Operator	Name	Description
-	Subtraction	Subtracts the number on the right of the operator from the n operator.
~	Bitwise NOT	Examines the bits of an operand, and performs performs log value of 0 become 1, and vice versa.

&	Bitwise AND	Examines the bits of two operands, and performs a logical A comparing the bits in each position from both sets. Returns where the bits being compared are true.
		`//  0101`
		`//  AND 0011`
		`//  =  0001`
		
		`// Performs the bitwise operation above, resultir`
		`5 & 3`

|	Bitwise OR	Examines the bits of two operands, and performs a logical O comparing the bits in each position from both sets. Returns where either bit being compared is true.
		`//  0101`
		`//  OR  0011`
		`//  =  0111`
		
		`// Performs the bitwise operation above, resultir`
		`5 | 3`

xor	Bitwise XOR	Examines the bits of two operands, and performs a logical e comparing the bits in each position from both sets. Returns where only one of the bits are true.
		`//  0101`
		`//  XOR 0011`
		`//  =  0110`
		
		`// Performs the bitwise operation above, resultir`
		`5 xor 3`

Operator	Name	Description
<<	Left Shift	A signed bitwise left shift.
>>	Right Shift	A signed bitwise right shift.
>	Greater Than	Logical greater-than test between two numbers. Returns a b
<	Less Than	Logical less-than test between two numbers. Returns a bool
>=	Greater Than or Equal To	Tests if the operand on the left is greater or equal to the ope Returns a boolean.
<=	Less Than or Equal To	Tests if the operand on the left is less than or equal to the op Returns a boolean.
=	Equal	Tests for equality between two operands.
!=	Not Equal	Tests for equality, returning true when not equal.
&&	Logical AND	Returns true when both operands are true. Anything non-ze
||	Logical OR	Returns true when either operand is true. Anything non-zero
like	Fuzzy String Matching	Compares the left-hand value with the pattern on the right si consist of %,*, and ? wildcards.

Bound Values

Bound values are paths to other values enclosed in braces. These will appear red in the Expression Editor. When you are writing an expression for a **Expression Binding in Vision**, you can reference tag values and property values using the brace notation. When you are writing an expression for an Expression tag, you can only reference other tag values. You can use the **Insert Property** icon and **Insert Tag** icon to build these references for you.

Dataset Access

If you have an expression that returns a dataset, you can pull a value out of the datatset using the dataset access notation, which takes one of these forms:

```
Dataset_Expression ["Column_Name"]    //Returns the value from the first row at
the given column name.
Dataset_Expression [Column_Index] //Returns the value from the given column at the
first row.
Dataset_Expression [Row_Index, "Column_Name"] //Returns the value from the given
row at the given column name.
Dataset_Expression [Row_Index, Column_Index] //Returns the value from the given
row at the given column index.
```

For example, this expression would pull a value out of a Table at row 6 for column "ProductCode":

```
{Root Container.Table.data}[6, "ProductCode"]
```

Note that you'll often have to convince the expression system that what you're doing is safe. The expression language can't tell what the datatype will be for a given column, so you may have to use a type-casting function to convince the expression language to accept your expression, like this:

```
toInt({Root Container.Table.data}[6, "ProductCode"])
```

Collection and Mapping access

When working with collections such as sequences or arrays, individual elements can be addressed with square brackets and the index of the desired element.

```
// Returns the third elements in the array
{An Array}[2]
```

Keys in maps (dictionaries and JSON objects) can be access with square brackets and string name of the desired key:

```
// Returns the value of "myKey"
{A Map}["myKey"]
```

Expression Functions

The expression language's functions are where much of the real power lies. A function may take various arguments, all of which can themselves be any arbitrary expression. This means that you can

use the results of one function as the argument to another function. In general, the syntax for a function call is:

```
functionName(expression1, expression2, ...)
```

Whitespace and Comments

Whitespace, such as spaces, tabs and newlines, are largely ignored in the expression language. It is often helpful to break your expression up onto multiple lines for clarity. Comments are delimited by two forward slashes. This will make the rest of that line be ignored. This example shows an if function spread over four lines with comments annotating the arguments.

```
if( {Root Container.UseTagValueOption.selected},
    {MyTags/SomeValue}, // Use the tag value.
    "Not Selected"     // Use default value if the user doesn't check the box.
)
```

Tag Paths

While referencing a tag path in expressions, you may see some special notation, such as `~` and `[.]`. More information on this notation can be found on the **Tag Paths** page.

Additional Examples

The following headings demonstrate many simple examples that better display the expression language's syntax.

String Concatenation

You can use an Expression binding to concatenate strings, resulting in a new string.

The function starts with CONCAT() and inside the function is a list of strings. They can be manually typed like "42" or they can come from Tags or properties.

Example 1

```
concat("The answer is: ", "42") //returns "The answer is: 42"
```

Example 2

You have a date, and need to extract the year, and concatenate the word "Vintage" to the end for a label display. Bind a label's text property to:

```
dateExtract({Root Container.VintageDate}, 'year') + ' Vintage'
```

Celsius to Fahrenheit

Suppose you had a Temperature tag that was in Celsius. If you wanted to display that tag in Fahrenheit on a Label component, you would use an Expression binding on the Label's `text` property using the following expression:

```
1.8 * {My Temperature Tag} + 32
```

Every time the Temperature tag changes, the expression will re-calculate the value and push it into the Label's text property. Now, let's say that you wanted to append **°F** to the end of the label so the user knew the units of the temperature. Similar to the example above, you could simply use a string concatenation in your expression, like this:

```
(1.8 * {Some Folder/Some Tag} + 32) + " °F"
```

Format Date

You can format a date in an Expression binding by using the `dateFormat` and `now` functions.

To make a label that updates to show the current time:

1. Drag a **Label** component onto the window.
2. Select the label's **Text** property binding icon and select **Expression** binding.
3. Enter the following code into the expression, and click **OK**.

```
dateFormat(now(1000), "MMM d, yyyy hh:mm:s a")
```

The `dateFormat` function takes two arguments. The first argument is any date type variable. This can include another function (like `now`) that returns a date type. The second argument refers to the date format that you want returned. The `now` function returns the current time and in this case, it will update every second. For more information on the date formatting expression, see the **dateFormat** Expression Functions page.

Date Manipulations

You can manipulate dates in Expression bindings such as a date addition/subtraction with the `dateArithmetic` function. This is important when you want to use the Expression bindings to select a date that is offset by a certain amount.

The following example will return the time 15 minutes ago by using the dateArithmetic expression function:

```
dateArithmetic(now(), -15, "minute")
```

Bit Functions

You can use various bit functions in Expression bindings like `getBit` to return individual bits of a word.

Example 1

Assuming a tag path 'Folder/BitTag', the following would return the binary representation of the tag's value at the 0 position

```
getBit({Folder/BitTag}, 0)
```

For more details on the getBit function, please see the **getBit** page in the Appendix.

Example 2

You have 3 bits in a PLC, only one of which will be on at a time. You want to turn these 3 bits into a single integer (0,1,2) to drive a component's Styles. Bind a custom integer property to:

```
binEnum({MyTags/Bit1}, {MyTags/Bit2}, {MyTags/Bit3})
```

Switch

You can use the switch function in Expression bindings to evaluate conditional statements. This function acts like the switch statement in C-like programming languages. It takes the value argument and compares it to each of the case1 through caseN expressions.

The following example returns the string `Running` when it is given the value of 1. Its options are 0,1, and 2. And when comparing the value to the options the switch statement returns one of the corresponding results. If a result cannot be found, a fail-over option is returned.

```
switch(
1,
0, 1, 2,
"Off","Running","Fault",
forceQuality("!BAD STATE!",0))
```

Checking Conditions

You can use Expression bindings to return true or false based on different conditions.

Example 1

Consider the following expression that references the tag with a path of 'Folder/Machine State':

```
{Folder/Machine State} = 0
```

The above expression simply tests the value of the Machine State tag. If the value of the Machine State tag is ever equal to 0 then the above expression would return true. In every case where Machine State is not equal to 0, then the expression would return false.

Example 2

It is possible to check for multiple conditions in the same expression. If you have two boolean tags and you only want the expression to return true if both tags are true, then the binding would look like:

```
{boolTag1}=True && {boolTag2}=True
```

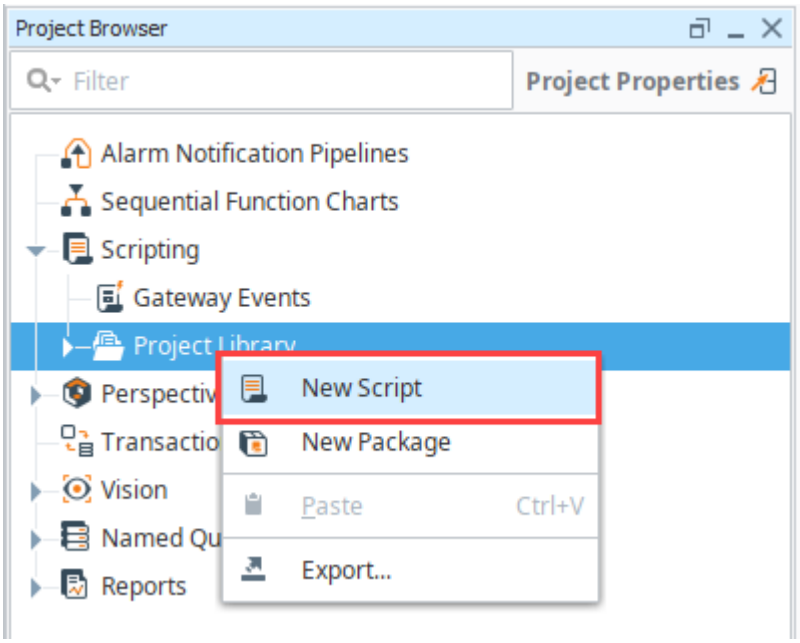
Tag History

You can use Expression tags to display tag history data that is stored in a database. The example below demonstrates how to pull a single tag's average history on an Expression tag using a Project Library script and the runScript function. The expression used here can be altered to incorporate multiple tag values if desired. The tag history used for this example is from the Sine0 tag in the Quick

Start configuration. This was enabled in our project by accessing the **default** Tag Provider in Tag Browser dropdown, selecting **Browse Devices** to import the Sine0 tag, and selecting **History** > **History Enabled** > **true** and a **Storage Provider** in the Sine0 Tag Editor.

> **CAUTION**
>
> Since this method uses both project library scripting and requires access to tag data, make sure your project name is defined in the Gateway Scripting Project property on your Gateway Webpage.

1. In your Project Browser, expand the **Scripting** dropdown.
2. Right-click **Project Library** and select **New Script**.



3. In the name field, enter **TagHistory**.
4. Select **Create Script**.
5. Paste the following script:

```
def tagHistory(tagpath):
    start = system.date.parse("2025-06-30 4:00:00")
    end = system.date.now()
    results = system.historian.queryAggregatedPoints(paths=[tagpath],
startTime=start, endTime=end)
    return results.getValueAt(0,1)
```

> **NOTE**

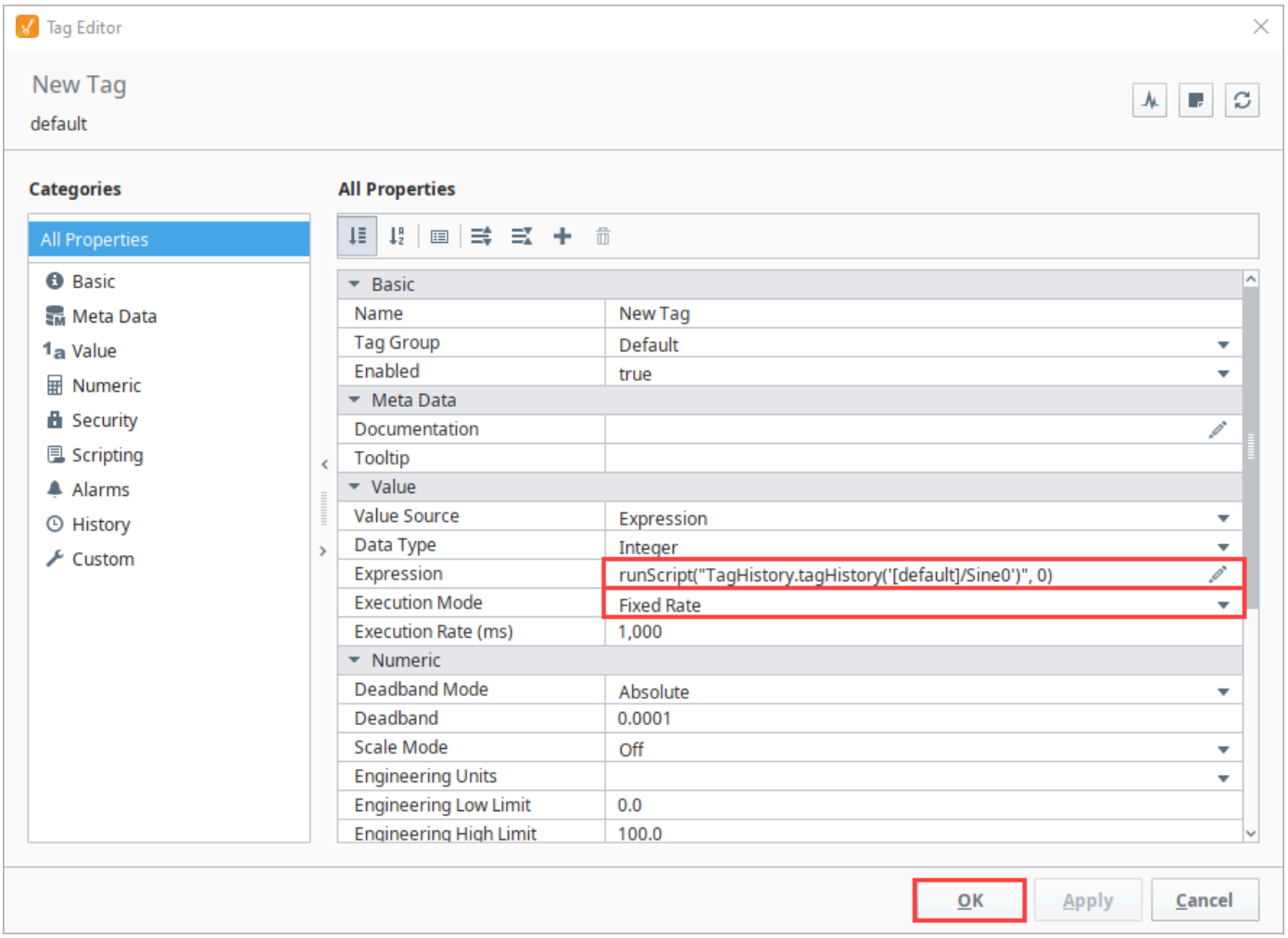
> See the **system.historian.queryAggregatedPoints** page for more information on the function.

6. Save your project.
7. Navigate to the Tag Browser and select the **Add Tag** + icon.
8. Select **New Standard Tag** > **Expression Tag**.
9. Click the Expression **Edit** icon and paste the following expression into the pop-up field:

```
runScript("TagHistory.tagHistory('[default]/sine0')", 0)
```

> **NOTE**
>
> See the **runScript** page for more information on the function.

10. Click **Apply**.
11. Change the Execution Mode to **Fixed Rate**.
12. Click **OK**.

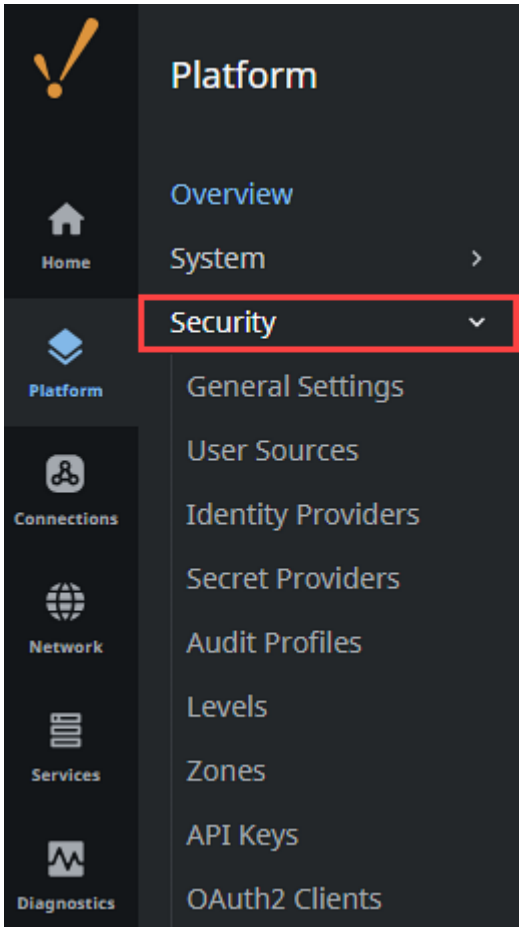

Tag Editor
New Tag
default
Categories
All Properties
Basic
Meta Data
Value
Numeric
Security
Scripting
Alarms
History
Custom
All Properties
Basic
Name
New Tag
Tag Group
Default
Enabled
true
Meta Data
Documentation
Tooltip
Value
Value Source
Expression
Data Type
Integer
Expression
runScript("TagHistory.tagHistory('[default]/Sine0')", 0)
Execution Mode
Fixed Rate
Execution Rate (ms)
1,000
Numeric
Deadband Mode
Absolute
Deadband
0.0001
Scale Mode
Off
Engineering Units
Engineering Low Limit
0.0
Engineering High Limit
100.0
OK
Apply
Cancel

Security

Security options in Ignition provide many ways to safeguard access to your data and applications. You control not only who accesses your systems, but when and where they can access them. Ignition offers two authentication strategies: **Classic Authentication Strategy** or **Identity Provider Authentication Strategy**.

Gateway Security

Security in Ignition falls into a few categories, tying into the various scopes (Designer, Gateway, Vision Clients and Perspective Sessions). In the Gateway scope, the bulk of security setup happens under the **Platform** section of the Gateway Webpage, under the Security header, you'll find pages for authentication, role mappings, and zones.



The primary purpose of Gateway security is to protect access to the two most critical areas of Ignition: the Designer and the Gateway. Many important resources are configured in these areas, so access to each Gateway section, as well as the Designer, can be limited by Security Level.

Authentication Strategies

In regard to authentication and permissions, there are two approaches.

Classic Authentication Strategy (Designer and Vision Only)

Classic Authentication Strategy involves a concept known as a User Source, which is a configuration that contains multiple roles and users. Users are assigned roles, and security restrictions within a project can be used to check if a user has one or more roles. User Sources can be "internal", meaning all users and roles are contained within an Ignition Gateway, or externally stored in an SQL database. Furthermore, User Sources offer integration with **Active Directory**.

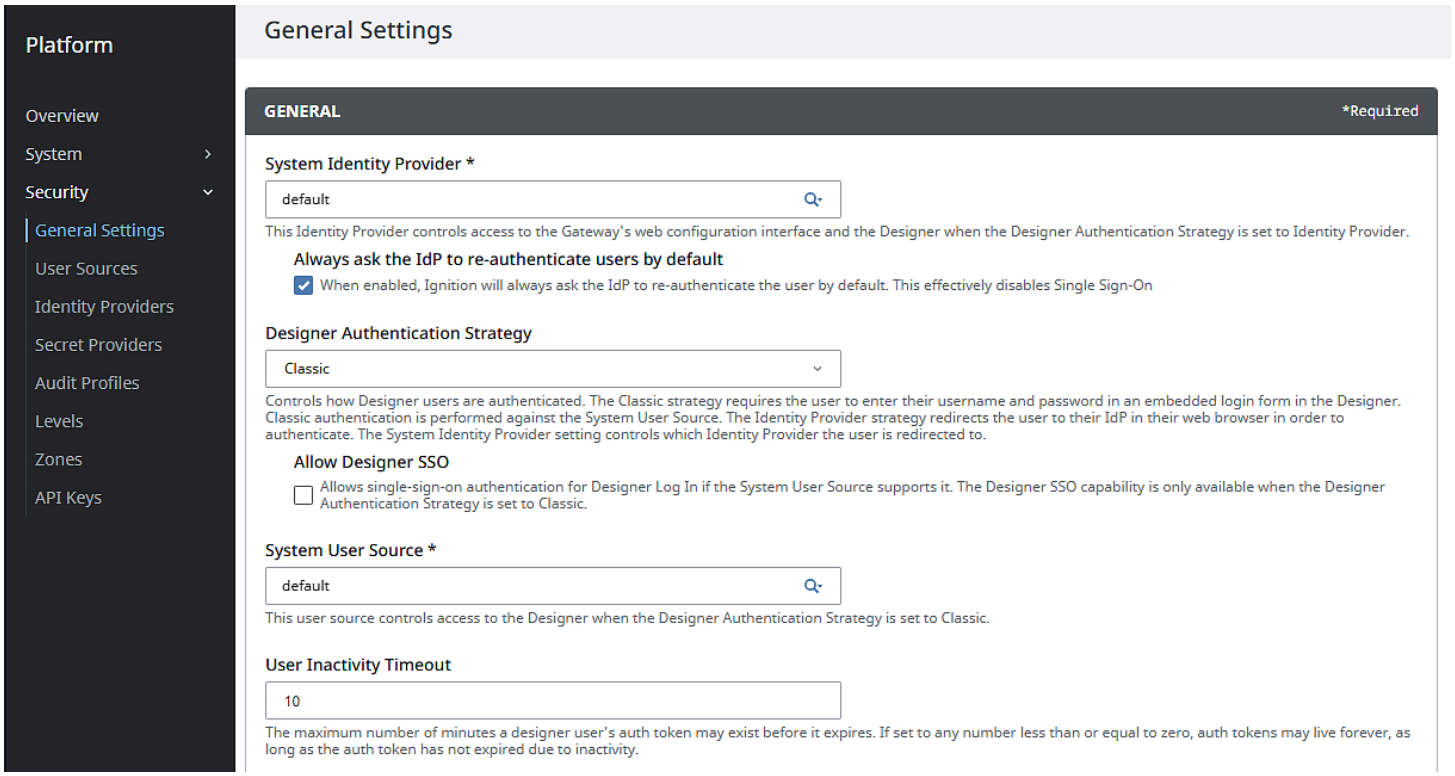
Identity Provider Authentication Strategy

Ignition can also integrate with Federated **Identity Providers** (IdP), allowing users to authenticate against a trusted third party. The Identity Provider Authentication Strategy works by assigning Security Level restrictions to various features within Ignition, and utilizing **User Attribute Mapping** and **Security Level Rules** to assign Security Levels to users.

Ignition can integrate with both OpenID Connect and Security Assertion Markup Language (SAML) providers. In addition, Ignition can act as an Identity Provider for isolated systems.

Gateway General Security Settings

This page determines security permissions for the Gateway and Designer.



Security General Settings Table

General

Setting	Description
System Identity Provider	Dropdown list to select the Identity Provider that controls access to the Gateway's web configuration interface and the Designer (only when the Designer Authentication Strategy is set to **Identity Provider**). An additional option is included below the dropdown that determines whether or not to always ask the IdP to re-authenticate users by default. When enabled, Ignition will always ask the IdP to re-authenticate the user by default. This effectively disables Single Sign-On.
Designer Authentication Strategy	Controls how the Designer authenticates users. Options are **Classic** or **Identity Provider**. • **Classic**: The Classic strategy requires the user to enter their username and password in an embedded login form in the Designer. Classic authentication is performed against the System User Source. • **Identity Provider**: The Identity Provider strategy redirects the user to their IdP in their web browser in order to authenticate. The System Identity Provider setting controls which Identity Provider the user is redirected to. Additional options on this screen will change depending on the Designer Authentication Strategy that is selected here.
User Inactivity Timeout	The number of minutes which must elapse before expiring a user's Gateway web interface session to inactivity. Sessions will not timeout if set to any number less than or equal to zero.
Allow User Administration	Allows the administration of the Gateway's system user source from the Designer and client. Unless this is enabled, the Vision module's User Management component will be prevented from altering the Gateway's system user source and scripts will not be able to alter users or roles. (Default is false.)

Setting	Description
Gateway Audit Profile	Dropdown list to select the name of the audit profile that Gateway-scoped actions will log to.

Designer Authentication Strategy

The following settings will be applied to Designer users based on the selected authentication strategy.

Classic

Setting	Description
Allow SSO	Allows single-sign-on authentication for Designer Log In if the System User Source supports it. Default is false.
System User Source	This user source controls access to the Designer.
Role(s)	Users must belong to one of these roles in order to log in to the Designer. Multiple roles can be specified by separating them with commas.

Identity Provider

Setting	Description
Auth Token Inactivity Timeout	The number of minutes which must elapse before a Designer user's auth token will expire due to inactivity caused by a disconnected Session. Must be greater than zero. Default value is 10.
Auth Token Time-To-Live	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, unless the auth token expires due to inactivity.
Permissions	Check the security levels required for Designer access. If Public, then all users will be given access permissions. This setting also requires confirmation for whether users need to match at least one or all of the checked levels to receive these permissions.

Roles and Permissions

Setting	Description
Gateway Write Permissions	Check the security levels required for Write capabilities to the Gateway. If Public, then all users will be given Write permissions, meaning they can interact with all Gateway pages and settings. Users who receive these permissions will also receive Read permissions regardless of settings applied to that field. This setting also requires confirmation for whether users need to match at least one or all of the checked levels to receive these permissions.
Gateway Read Permissions	Check the security levels required for Read capabilities to the Gateway. If Public, then all users will be given Read permissions, meaning they can view all Gateway pages and settings. Users who receive these permissions will also receive Access permissions regardless of settings applied to that field. This setting also requires confirmation for whether users need to match at least one or all of the checked levels to receive these permissions.
Gateway Access Permissions	Check the security levels required for general access to the Gateway. If Public, then all users will be given Access permissions, which means they will be able to view the Gateway pages within the Home section, with the exception of the Perspective Sessions and Brand Settings pages. This setting also requires confirmation for whether users need to match at least one or all of the checked levels to receive these permissions.

CREATE PROJECT PERMISSION DEPRECATION

The Create Project Permission setting was deprecated in version 8.3.7 in favor of the existing Designer Roles and Permissions settings. See the **Create Project Permission** page in the Deprecated Manual for more information.

Classic Authentication Strategy

The Classic Authentication strategy authenticates users against a User Source. Both the Designer and Vision Client can authenticate users with this strategy.

> **NOTE**
>
> If you have Ignition 8.3 with the Perspective module, authentication is handled instead by the **Identity Provider Authentication Strategy**.

User Sources

User sources are a collection of users, roles, and other user data, such as contact information or schedule. When a **new user or role is created**, it is applied and stored in the user source. Projects and the Gateway are assigned a User Source to authenticate against. This determines which users have access to which project(s).

There are several types of user sources: single-storage types with varying storage mediums, "hybrids" that combine features of the other types, and a cache type used in Local Client Fallback systems.

Single-Storage

Users and roles are stored in a single location. The single-storage users sources are:

- **Internal Authentication** - Users and roles are stored internally to Ignition.
- **Database Authentication** - Users and roles are stored in a SQL database. Managing users is done via direct interaction with the database.
- **Active Directory Authentication** - Users and roles are managed by Active Directory. Users are authenticated through the LDAP protocol.

Hybrid

Users in hybrid user sources authenticate against Active Directory, meaning that user names and passwords are checked against those stored in Active Directory. However, roles are stored either internally in Ignition or in a SQL database, so it is possible to make a role change without have to contact your Active Directory administrator. This way, Active Directory can be consulted to see if a user is valid, but the management of roles does not require coordination with the IT department, who

typically control the Active Directory system. This "best of both worlds" approach is popular for many users of Active Directory.

- **Active Directory-Internal Hybrid** - Users managed by Active Directory and roles stored to Ignition internally.
- **Active Directory-Database Hybrid** - Users managed by Active Directory and roles stored in an SQL database.

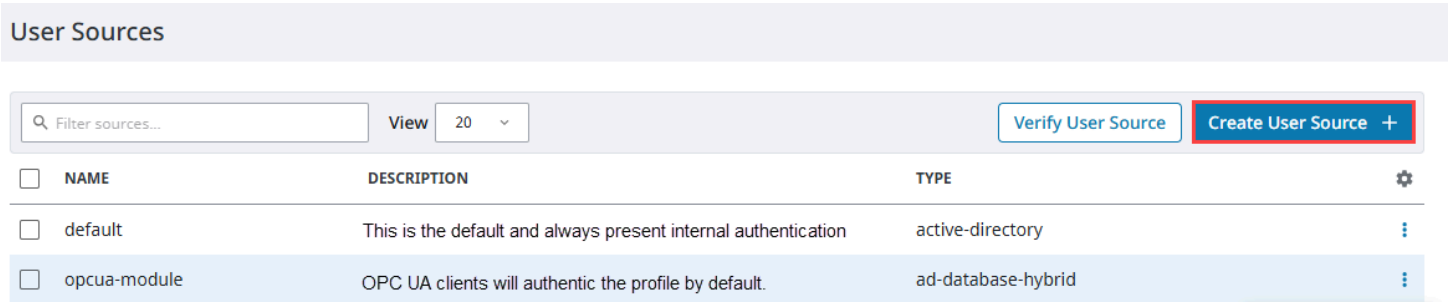
Fallback Cache

This User Source was developed specifically for a system that is using Local Client Fallback, and allows you to cache the login credentials from a remote user source. This means your users can still log in with their normal username/password on a Local Client Fallback project, even when the network connection is unavailable.

More information can be found on the **Fallback Cache Authentication** page.

Creating a User Source

All of the described User Source types can be created by navigating to the **Platform > Security > User Sources** section of the Gateway. To begin the process, select **Create User Source +** and choose the User Source type you need. Once you've finished entering the User Source properties, click **Create User Source**.



> **NOTE**
>
> The User Source page also includes a **Verify User Source** option. See the Verify a User on a User Source page for information on how to verify an existing user's profile.

Shared Functionality

Regardless of type, all User Sources have the following functionality:

- **Failover Source**: If the User Source is unavailable for authentication, then a backup User Source can be specified. The type of the fail-over User Source can differ from the primary, so configurations where an internal-type fails over to a database-type are possible.
- **Schedule Restrictions**: The User Source can prevent users from logging in when they are off schedule, meaning that the schedule assigned to the user determines when the user may login.

General Properties

All User Sources have a section of **General** properties that are listed before their unique type properties. Below are the descriptions of the General properties. See the individual User Source pages for the remaining property descriptions.

Name	Description
Name	The name of the User Source. This is how other systems in Ignition reference the user source. Note that every User Source **must** have a unique name.
Description	An optional description of the user source. Useful for noting which database connection or AD server the User Source may be referencing.
Enabled	Disabling a User Source profile will prevent it from being used completely, including authentication and user/role management. User Sources are enabled by default.
Schedule Restricted	Forces schedule restrictions on users. If true, users are only able to log in when their assigned **schedule** is active. This means a login will fail for users attempting to log into a client while they are off schedule.
Failover Source	Allows authentication attempts against this User Source to failover to another User Source in the event of a network outage, or some other connection issue. Useful with database or Active Directory user sources, as connection failures to the database/AD server will prevent users from logging in. This property in initially set to **None**, meaning a failover User Source is not configured.
Failover Mode	When a **Failover Source** is configured, this property determines when the failover User Source should be consulted. The following options are available: **Hard**: The Failover User Source is only consulted when this User Source is unreachable. **Soft**: The Failover User Source will be consulted if the user's credentials fail authentication, meaning that the user typed in credentials that are unrecognized or incorrect.
Cache Validation Timeout	The amount of time in minutes between cache updates of the User Source. If you set this value to less than 1, the cache validation timeout is disabled. Default is 15.

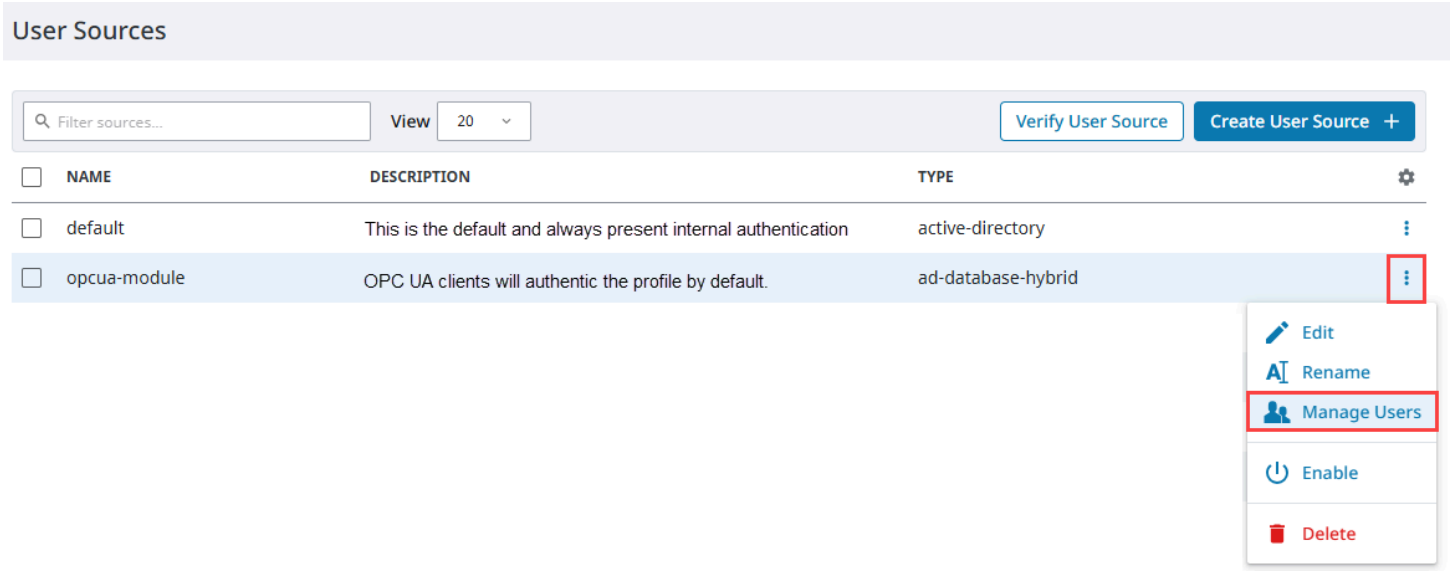
Name	Description
Lockout Enabled	Lock out a user's account after more than the maximum allowed number of failed authentication attempts occur within the lockout window. Default is true. Note that access can be restored to all locked out users by editing the user source, and clicking the **Save Changes** button.
Lockout Attempts	Maximum number of failed authentication attempts allowed within the lockout window before locking the user out. Default is 5. If this value is set to something less than zero (for example, -1), then the lockout functionality will be entirely disabled, regardless of what the **Lockout Enabled** property is set to.
Lockout Window	The duration of the lockout window in minutes. Default is 15. Setting this property to a value of less than zero (for example, -1) will disable the lockout functionality entirely, regardless of what the **Lockout Enabled** property is set to.

Details on the Password Policy Properties can be found on the **Internal Authentication** page.

The Default User Source

When Ignition is installed for the first time, an internal User Source named 'default' is created. You can manage the default User Source by navigating to the **Platform > Security > User Sources** section of the Gateway.

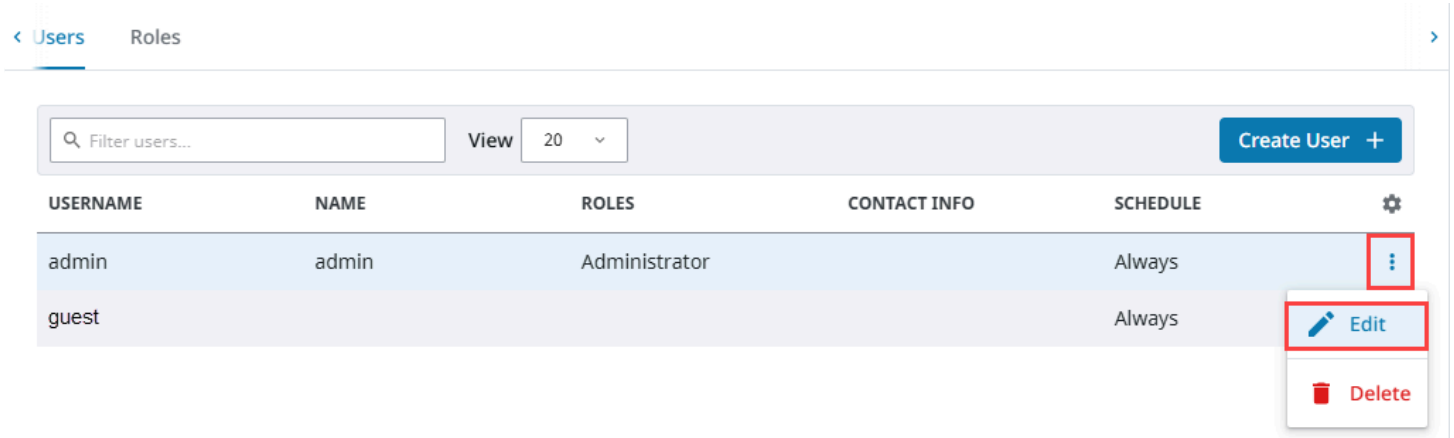
The **Manage Users** option under the user source **three dots menu** dropdown allows you to add new users, modify roles and passwords for existing users, remove users, and add/remove roles from the user source.



When you open the **default** user source for the first time, you will see the first user that was created at installation. This is the administrator account that has full privileges. If this user source has been modified before, a list of existing users is displayed.

Editing a User

Clicking **Edit** on the **User Source > Manage Users** page will access the corresponding Manage Users panel, which allows you to make any necessary changes to that user.



Fill out the fields for that user then click **Save Changes**.

User Properties

Name	Description
Username	The name of the user.
Change Password?	Check this box to change the existing password.
Password	New password. The **password policies** set for the internal User Source this user belongs to will is listed under the password field for better requirement visibility when creating user passwords. If any password requirement changes are desired, they must be made to the User Source.
Password	Re-type password for verification.
First Name	First name of the user.
Last Name	Last name of the user.
Roles	Role(s) assigned to this user. Check the box next to each role you want this user to have.
Schedule	Schedule for the user. Choose from a dropdown list of schedules that are already defined.
Language	Language to be used for the user. Choose from a dropdown list of languages that are already defined.
Notes	Any notes for this user.
Badge	A string that represents the value set for the user's badge.

CAUTION

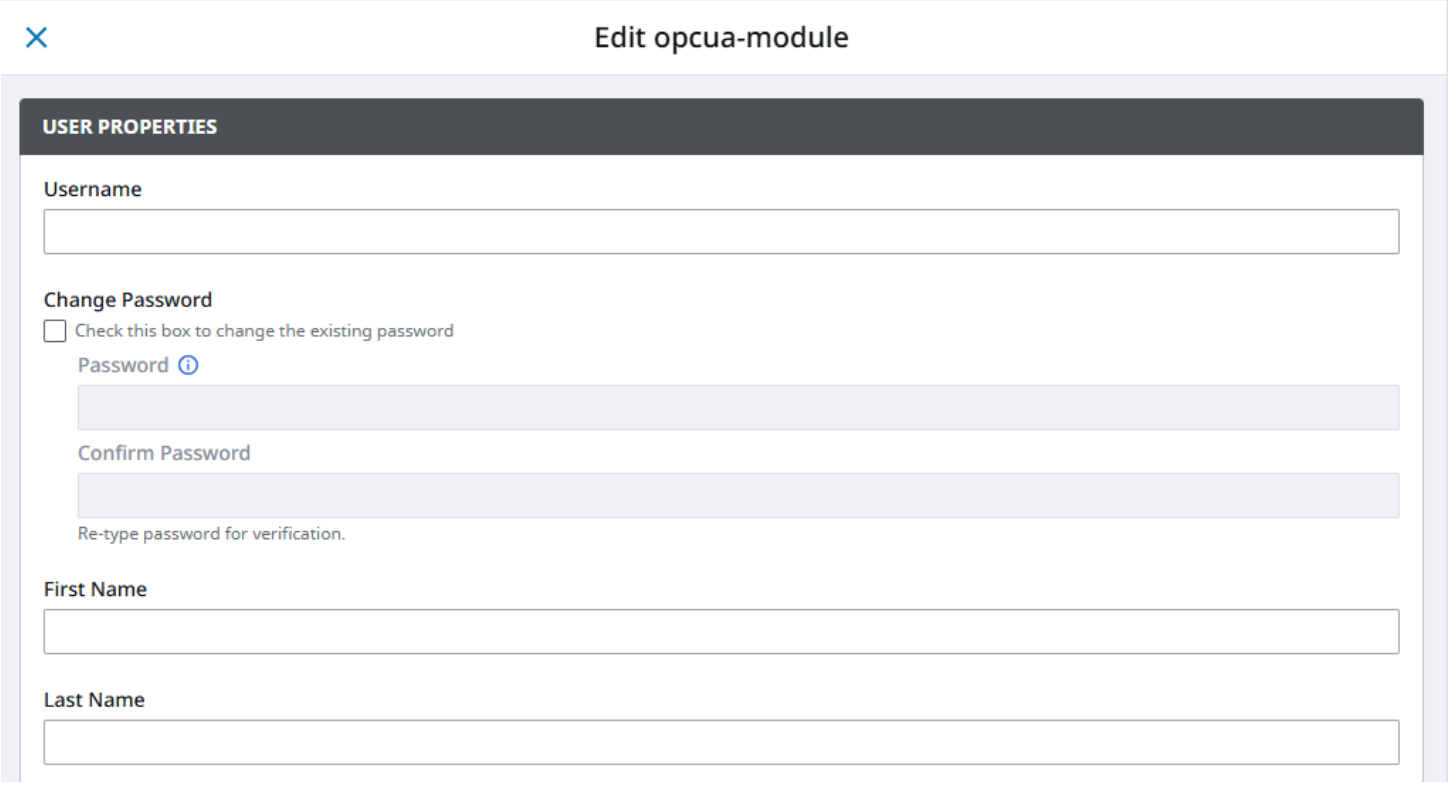
When scanning a badge to log in using Classic Authentication, a new line character may be appended to the scanned value, which can cause logging in to fail. If you encounter login issues with a badge, check the badge value for extra characters to remove, such as a trailing line or whitespace after the badge number.

Contact Info

Name	Description
Type	Choose email or SMS.
Value	The email value or SMS number.

Extended Properties

Name	Description
Security Pin	The pin assigned to a user to confirm authorization when receiving an alarm notification. This property is only applicable to the **Voice** and **Twilio Voice** Notification Profiles, and will only be visible if the Voice Notification or Twilio Integration and Alarm Notification modules are enabled.



Managing Users and Roles

Users and Roles

Security is based on the roles that are assigned to specific users. Roles do not have any structure or hierarchy by default, but it can be created. You can create a hierarchy based on users with a greater role being assigned all matching lesser roles.

There isn't a built-in restriction to the number of roles a user can have, so each user can be assigned many roles or none at all.

It's important to think about the different roles in your project and how they affect the security of your project. For instance, what level of access a particular area of a project needs may determine the functional type roles that you create, and the different users assigned to each role.

> **DANGER**
>
> When using role-based security in a project, the project stores the name of the role as a string. This means that if you were to modify the name of the role in the Gateway, the role-based security in your project will not update to reflect the new name, and instead will try searching for a role with the original name. Be very careful when modifying the names of roles.

You can manage users and roles using either the Gateway interface, or using the **User Management component** inside the Designer or Client. This section shows how to manage users and roles using the Gateway interface under **Platform > Security > User Sources**. Click on **Manage Users** link for the **User Source** you want to administer.

> **NOTE**
>
> Adding or modifying a role in the Users, Roles section of the Gateway will not impact or change roles located in the **Security Levels** section, as they are separate entities within Ignition. You will need to update the roles in the Security Levels section separately from the roles in Users, Roles.

Manage Users

User Sources support managing the users and roles from within Ignition to varying degrees. Some User Sources are fully manageable, meaning that you can administer the users, roles, contact info, and so on from within the Ignition Gateway, as well as inside a Vision Client. Other User Sources do

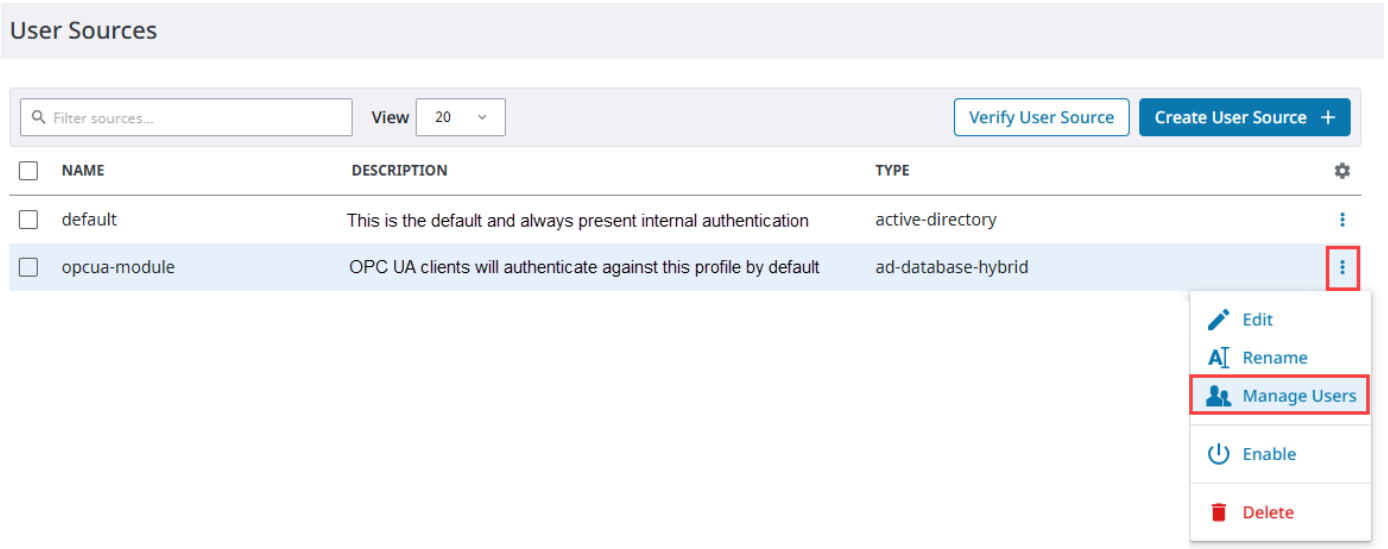
not support this at all or only partially support it. Make sure you understand how and where the administration takes place before you choose a User Source type.

Often, it is desirable to let some management or administrative users of a Vision project manage other users without having to log into the Gateway's Configure section. To do this, you can simply use the built-in User **Management Component** that comes with the Vision Module.

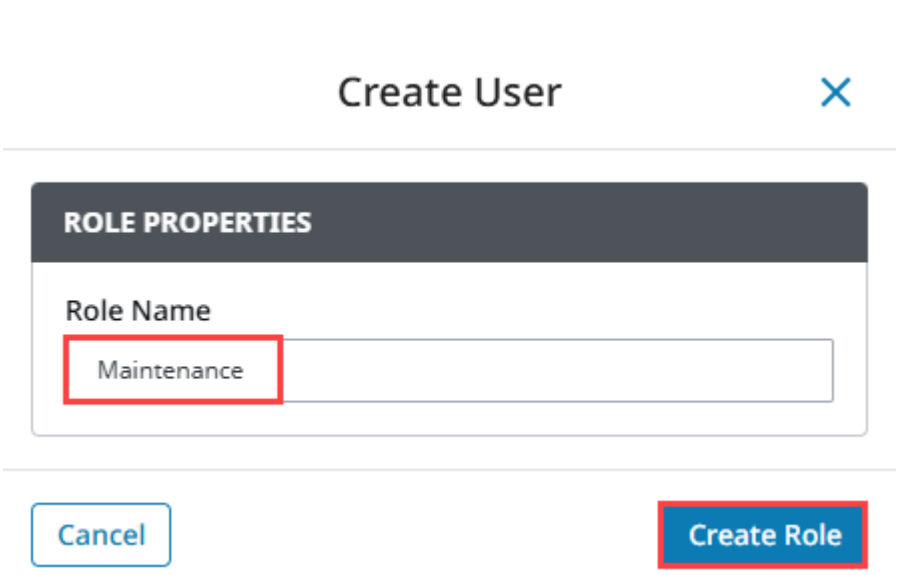
Create a Role

When a project is first created, the **Administrator** role is the only role available. Additional roles can be added from the User Sources page on the Gateway.

1. On the Gateway Webpage, go to the **Platform** section.
2. Navigate to **Security > User Sources**.
3. Click on the three dots menu and select **Manage Users** for the User Source you want to manage.



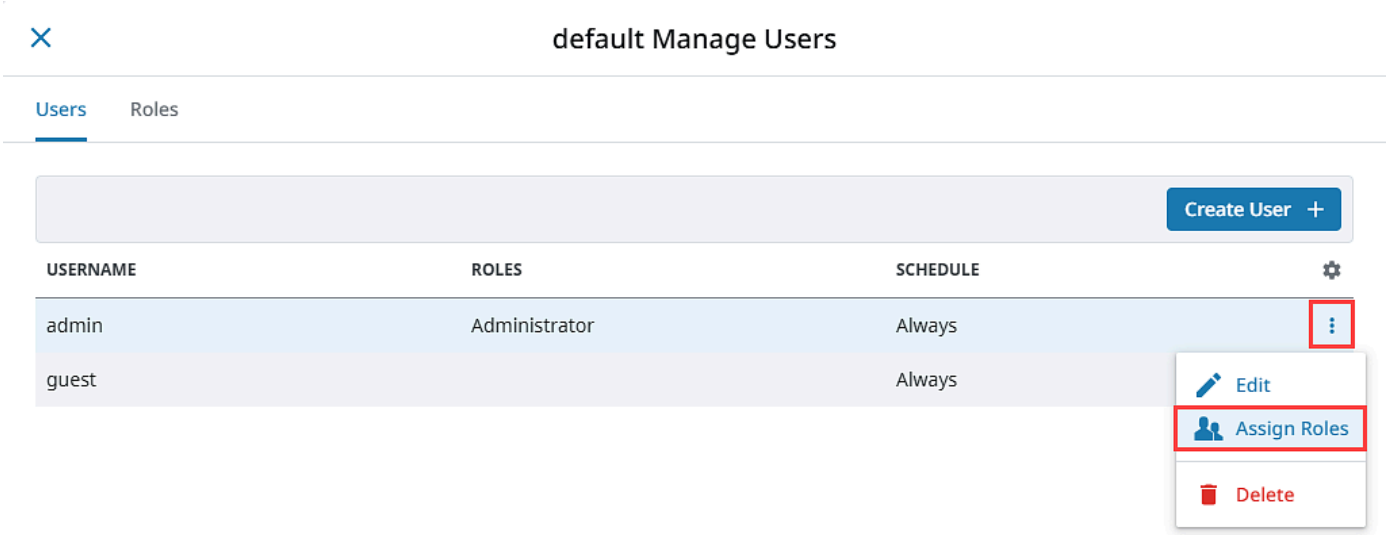
4. Click the **Roles** tab and click **Create Role +**.
5. Name the role by entering it in the **Role Name** field, and click **Create Role**. The role is now available to be associated with specific users.



Assigning Roles

Although it is not required for a user to have a role, be aware that users without role assignments might not have access to an area of the project that requires a role.

1. On the Gateway Webpage, go to the **Platform** section, and navigate to **Security > User Sources** to access the User Sources page.
2. Click the three dots menu and select **Manage Users** for the User Source you want to manage.
3. Click the three dots menu and select **Assign Roles** for the User you want to edit.
    - If you need to create a User for a role to be assigned to, select the **Create User +** button. Enter the **user properties** you want this user to have, click **Create User**, and then select **Assign Roles** from the three dots dropdown for the User to continue this procedure.



4. Select the role(s) that you want this user to have from the supplied Roles list.

5. Click **Save Changes**. The user now has the privileges associated with the selected role(s).

Role Hierarchy

Often you might want to have one role that includes all the permissions for another role, i.e., Supervisor can do everything that Administration and Maintenance roles can do. In the Designer, access to components can be restricted to specific security roles. You can give any Supervisor both of the Administration and Maintenance.

User Management Component

Ignition has a special User Management component in the Vision Module that allows you to add, modify, and delete users and roles (and more) inside the Designer and the Client. By default, changes to a User Source may not be made from this component to prevent users from locking themselves out of the Gateway, or give themselves access to the Gateway. You can override this behavior by enabling the **Allow User Admin** property located under the **Gateway's general security settings**. Once the Allow User Admin property is enabled, the User Management component is simple to set up and use.

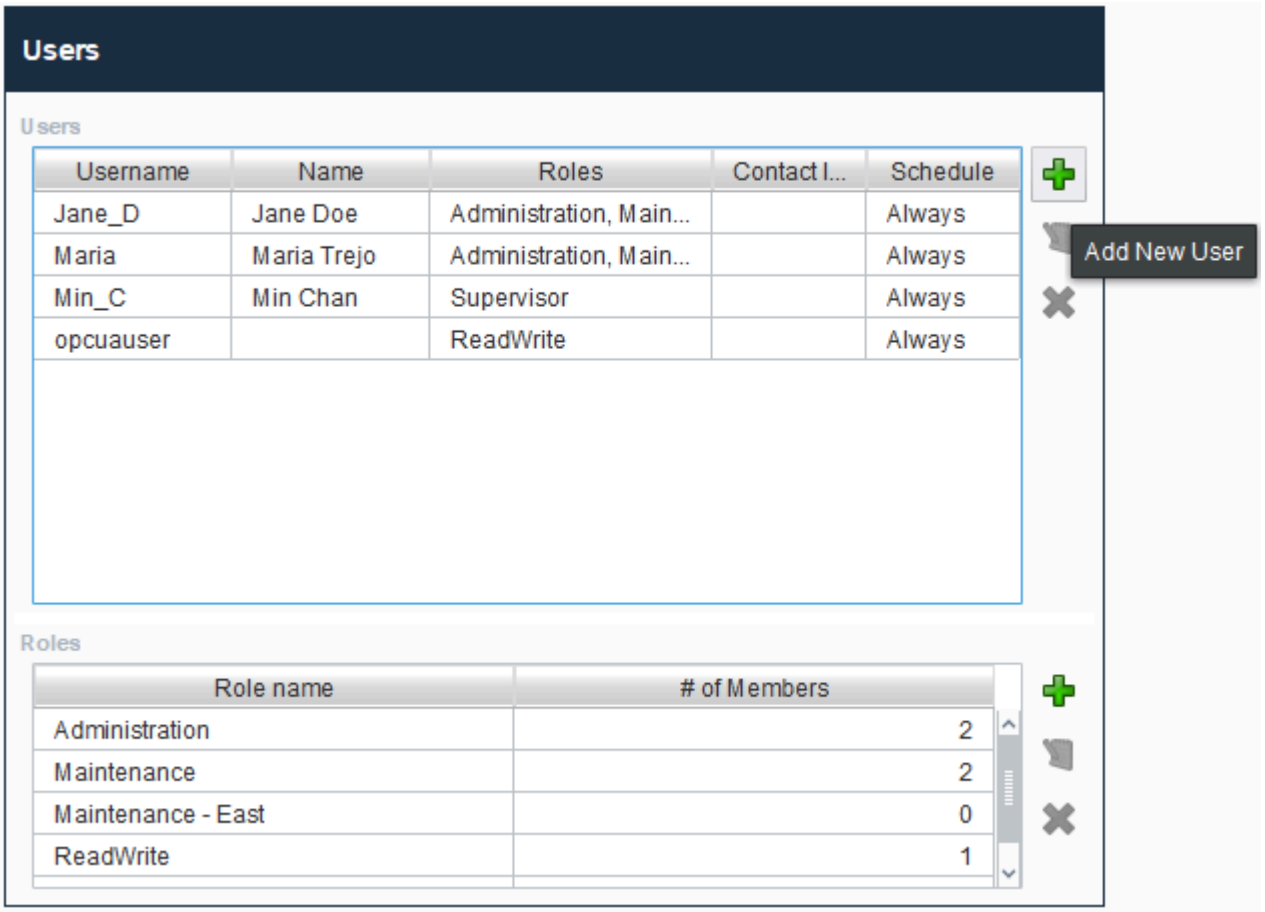
1. In Designer, go to the Project Browser and then to Vision.

2. Create a new Window or open an existing one.

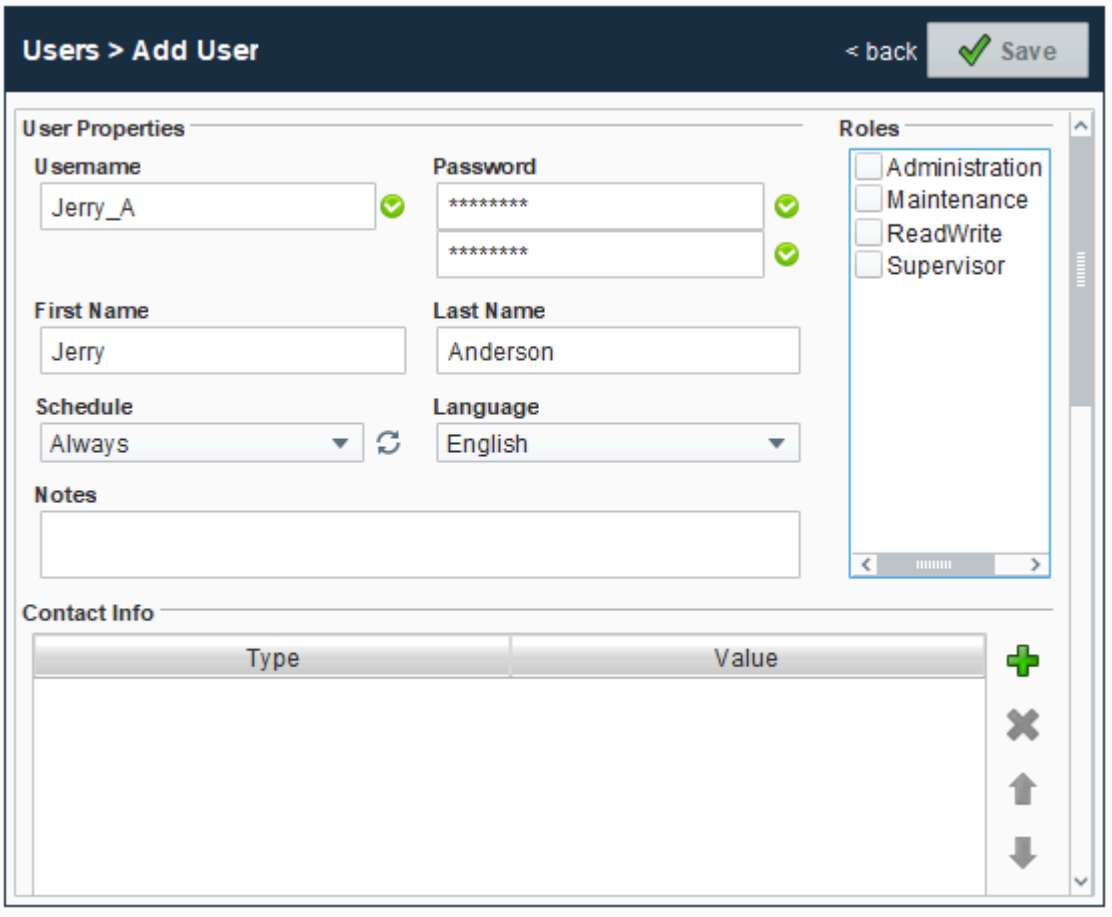
3. Drag a **User Management** component to your window. This component will automatically point to the default user source being used by your project. You can change the User Source property if needed.

4. If you already have some users and roles set up using the Gateway Webpage, you will see them in the User Management component. If you don't have any users or roles set up, you can create them here. Use the icons on the right side to add, edit, or delete a user or role.

5. To add a new user, put the Designer in **Preview Mode**. Click the **Plus** icon next to the user section.

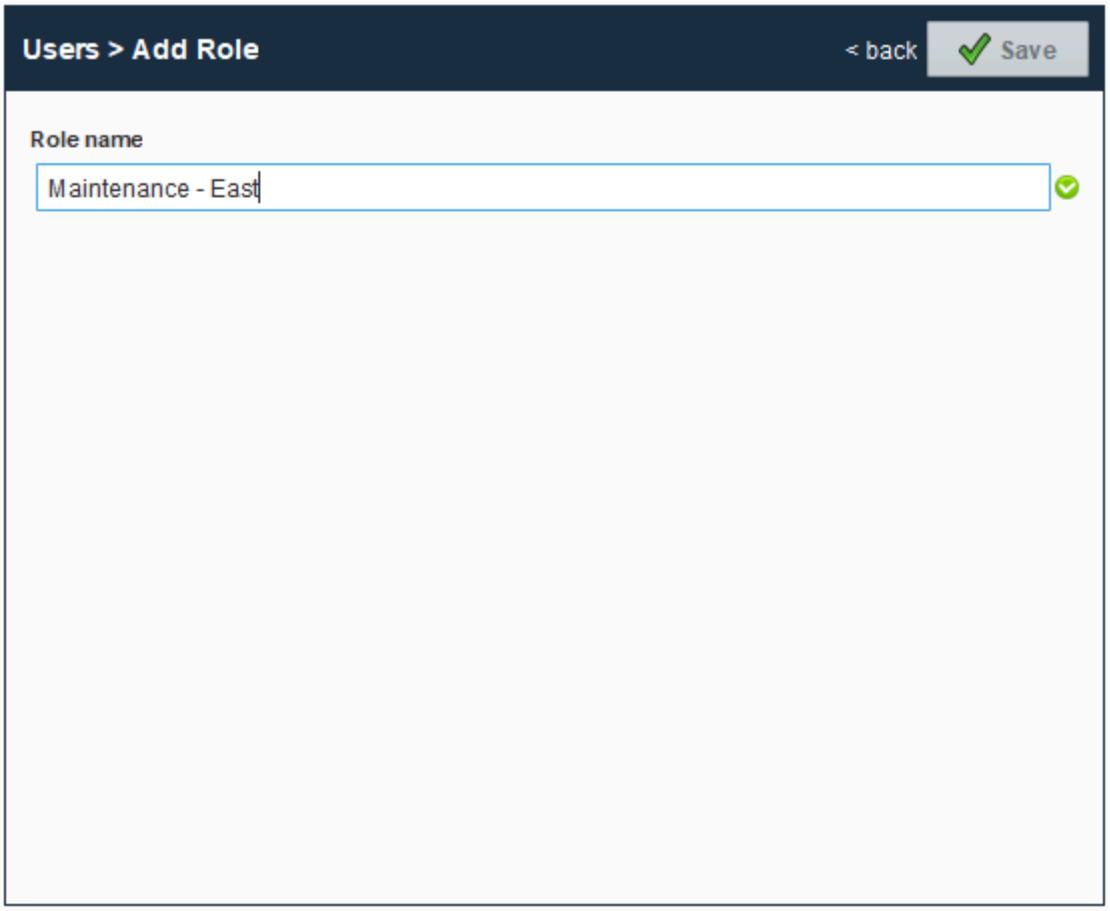


6. The Add User window will open. At a minimum, enter the **Username** and **Password**. All other properties are optional. When finished, click **Save**.



7. To add a new role, make sure the Designer is in **Preview Mode**. Click the **Plus** icon next to the role. The Add Role window will open.

8. Enter the name of the new role. Click **Save**.



9. Now you can see the user and role that were just added in the User Management window.

Users

Users

Username	Name	Roles	Con...	Sch...
Jane_D	Jane Doe	Administration, Maintenance, ...		Alw...
Jerry_A	Jerry Anders...	Maintenance - East		Alw...
Maria	Maria Trejo	Administration, Maintenance, ...		Alw...
Min_C	Min Chan	Supervisor		Alw...
opcuauser		ReadWrite		Alw...

Roles

Role name	# of Members
Administration	2
Maintenance	2
Maintenance - East	1
ReadWrite	1

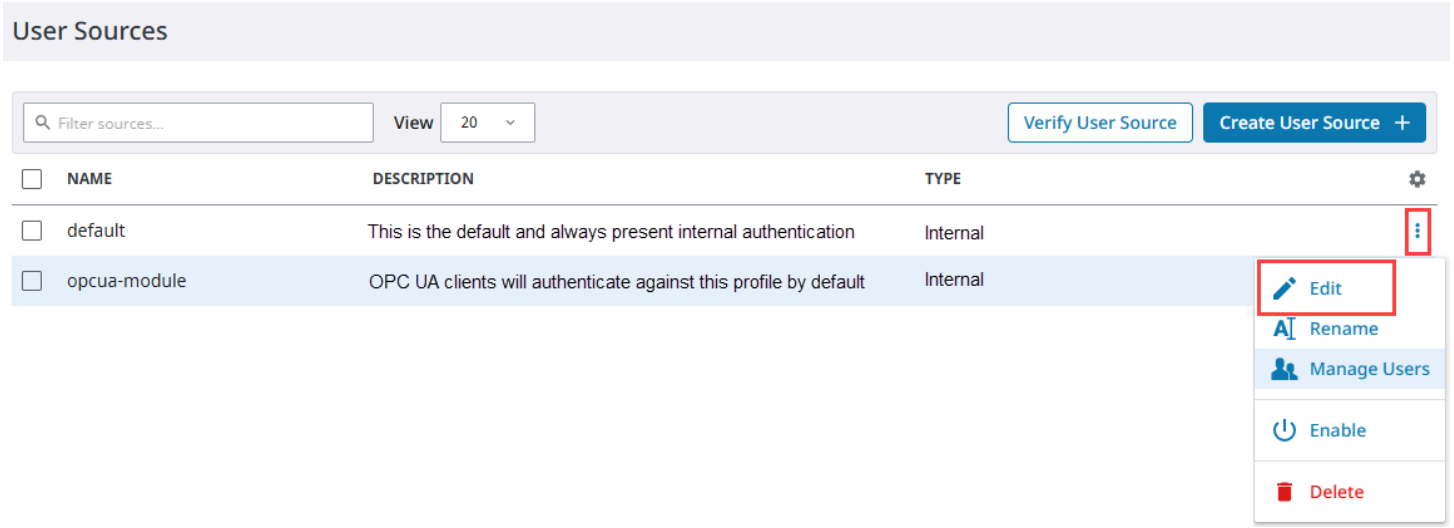
Internal Authentication

Internal User Sources

An Internal type User Source stores user information internally in the Gateway's database. This means that Internal User Sources are included in Gateway Backup files, and don't require an external SQL database, or other external user management system.

When Ignition is first installed, the **default User Source** that initially grants access to the Gateway and Design is an Internal type User Source. You can, of course, continue to use this default internal User Source for your project(s), or you may choose to use other User Sources instead.

The Internal User Source is fully **manageable** from within Ignition. You can access User Sources from the Gateway Webpage under the **Platform** section, **Security > User Sources**. Click the three dot menu and select **Edit**.



Property Reference

This section details Internal User Source properties, organized by category.

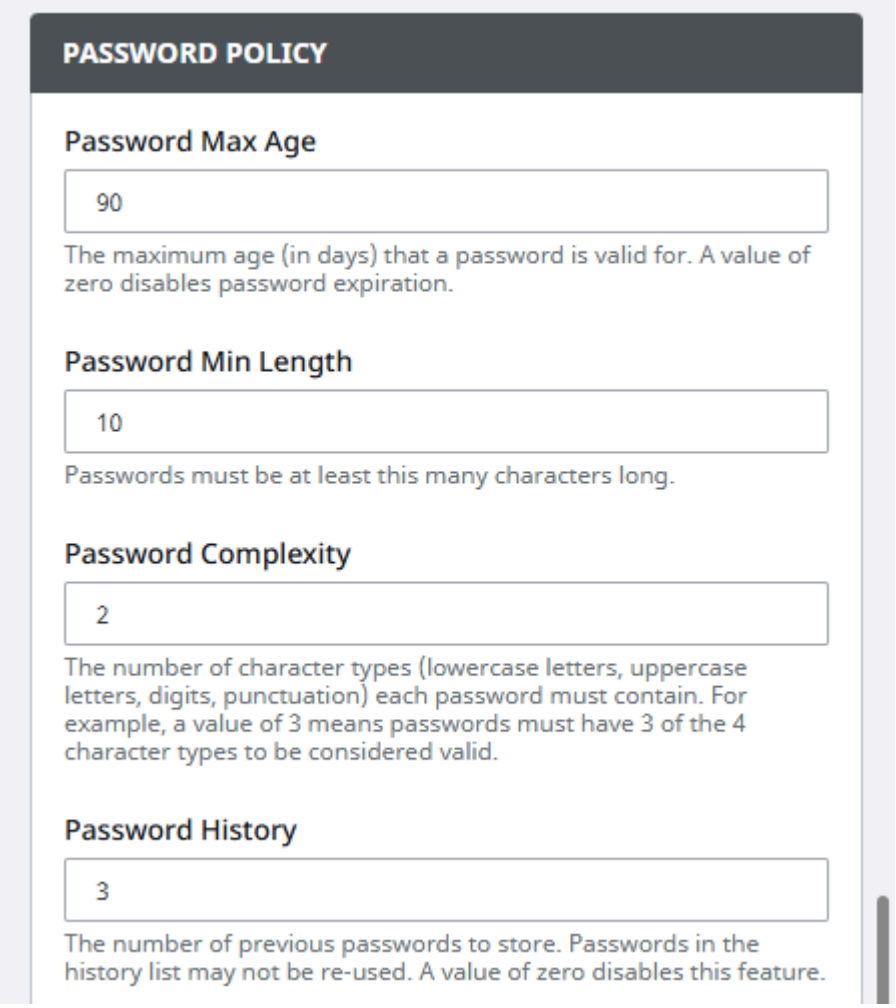
General Properties

Details on the General Properties can be found on the **Classic Authentication Strategy** page.

Password Policies Properties

The Internal User Source has password policies that are configurable from within the Gateway to provide an extra layer of security by ensuring that good password practices are used.

1. From the **Platform** tab on the **Gateway Webpage**, select **Security > User Sources**.
2. Click the three dot menu and select **Edit** for the User Source you want to update.
3. Scroll down to the Password Policy section. You can change the default password policies by entering the appropriate password values to support your password policies.



Below is a description of the Password Policy properties

Name	Description
Password Maximum Age	The maximum age in days that the password will still be valid. After the number of days has past, when the user tries to login, it will prompt them to change their password. A value of 0 will disable this feature. This property will also affect password age and expiration when you are using Ignition's Internal Identity Provider (IdP). If you try to log in after the expiration period, you will be redirected and prompted to update your password.
Password Minimum Length	The minimum amount of characters that a password must contain to be considered valid. If the user tries to make a shorter password, it will not allow it, and let them know that it does not meet the minimum length requirements. Default is 8.
Password Complexity	This determines how complex a password must be. There are four character types: lowercase letter, uppercase letter, digits, and special characters. The value here determines how many of those character types must be present at least once in the password for it to be considered valid. Default is 3.
Password History	Determines the number of previously used passwords to store. When users make a new password, old passwords can not be re-used. A value of zero disables this functionality. History is only stored while this setting is turned on, so any passwords used while this is off can be re-used when history is turned back on. Default is 0 (disabled).
Prohibit Password	If enabled, passwords may not contain any form of the word "password". This includes: • password • pass • word • pwd • passwd • pword Default is true.

Name	Description
Prohibit Username	If enabled, passwords may not contain any form of the username. Default is true.
Maximum Consecutive Repeated Characters	The maximum number of consecutive repeated characters allowed in a password. Set to 0 to disable. Default is 3.

Database Authentication

Database User Source

The Database Authentication type uses an external database instead of storing data inside Ignition. Managing users is done via direct interaction with the database. This section addresses how to set up a database user source. The Database Authentication type requires you have a connection to an existing database, like SQL Server, Oracle, or MySQL. It stores all users, roles, schedules, and and more in the database, and uses queries to check login credentials. When you create a database user source, you have the option of setting it up in Automatic or Manual mode.

Automatic Mode

In Automatic mode, Ignition will create and manage the database tables for you. You can specify a prefix for the tables that are created automatically for you, but their names after the prefix are chosen by the user source. In this mode, the user source will be fully **manageable** in Ignition.

Manual Mode

In Manual mode, you must provide SQL queries for various functions of the user source. In this mode, the user source will not be manageable from the Gateway or the Clients. You'll have to manage the users directly through the database. Examples for each of the queries are given on the user source setup page. Read each query description carefully to make sure you design your queries to return all the columns that are defined in the query's description as shown below.

Property Reference

Database User Sources have the following properties, organized by category.

General Properties

Details on the General Properties can be found on the **Classic Authentication Strategy** page. The Database User Source also has the following properties:

Name	Description
Database	The database connection this User Source will retrieve user information from.
Mode	How the Gateway should manage the database tables. Has two settings: **Automatic**: The Gateway will automatically create the database tables necessary, and all interactions with the table will use the built-in queries. When this option is set, the **Tablename Prefix** property is utilized. **Manual**: The Gateway will not automatically create any database tables, nor will it automatically modify users or roles. When set to manual, it is assumed that you want to manually write the queries that update the tables, or are utilizing another system. When Mode is set to this option, the **Manual Mode** properties are used to determine how the Gateway should query user data.

Automatic Mode Properties

Name	Description
Tablename Prefix	When set to **Automatic** mode, this property determines the prefix that will be used on all automatically created tables. Useful when multiple database User Sources are connected to the same database scheme.

Manual Mode

Name	Description
Authentication Query	A query that **must** return a row if the given username and password combination provided is valid. The query will run as a prepared statement, so use the question mark character (?) to represent username first and then password. The returned row may contain the user's basic properties under the column names: [firstname, lastname, schedule, language, notes] Note that the Gateway will pass both the username and password the user typed in, so this query **MUST** utilize exactly two question marks, otherwise an exception will occur.
Badge Authentication Query	A query that must return a row if the given badge provided is valid. The query will be run as a prepared statement, so use question mark (?) to represent the badge. The returned row must contain the username. Example: `SELECT username FROM USERS WHERE badge = ?`
List Roles Query	A query that returns all possible roles that any user could be a member of. The role names must be returned in the first column of the query's results.
User's Roles Query	A query that returns all of the roles that the provided user belongs to. The roles must be strings and must be in the first column of the query's results. The query will be run as a prepared statement with one parameter: the username.
Find User Query	A query that returns a row if a user with the given username exists. The query will be run as a prepared statement, so use question mark (?) to represent username. There must be at least one column: the username. Other columns are optional, supported columns are: [username, firstname, lastname, schedule, language, notes, badge]. Example: `SELECT username, firstname, lastname, schedule, FROM USERS WHERE username = ?` When the User Source is paired with an **Ignition Identity Provider**, enabling this option can provide a modest performance improvement during session login.

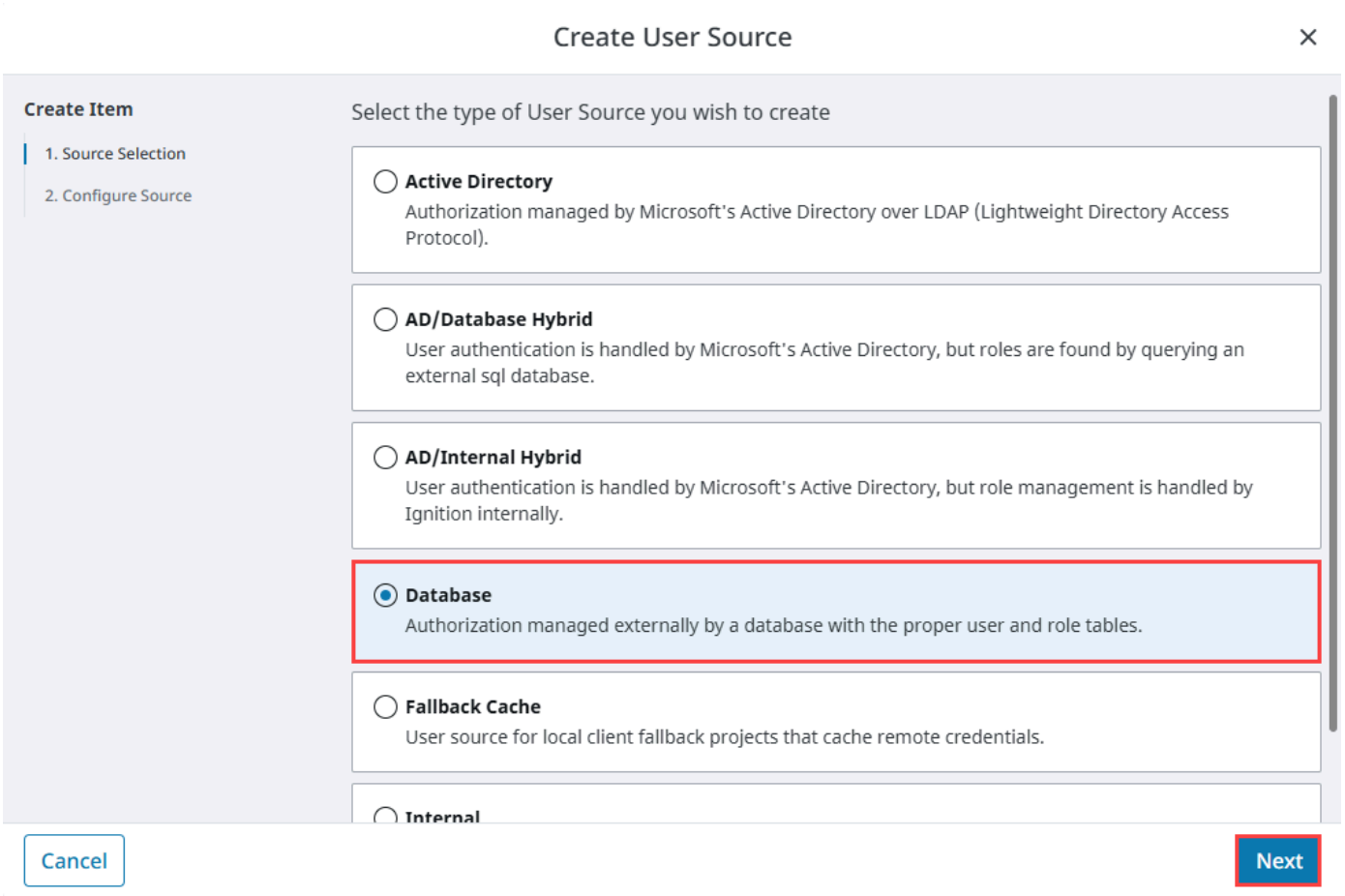
Name	Description
List Users Query	A query that returns a row containing each username. There must be at least one column: the username. Other columns are optional, supported columns are: [username, firstname, lastname, schedule, language, notes].
Contact Info Query	A query that returns all of the contact info for the user. The first column must be the contact type, the second column the contact value. Optional, may be blank.
Schedule Adjustment Query	A query that returns the upcoming schedule adjustments for the user. This property is optional, and may be left blank. The results set expects the following columns: • **Start**(date) • **End**(date) • **Available**(boolean) • **Note**(string)
Extra Properties Query	A query that returns name, value pairs of extra properties for the user. Will be run with one parameter: the username. Optional, may be blank.

To Create a Database User Source

1. On the Gateway Webpage under the **Platform** tab, go **Security > User Sources**. The User Sources page will be displayed.

2. Click **Create User Source +**.

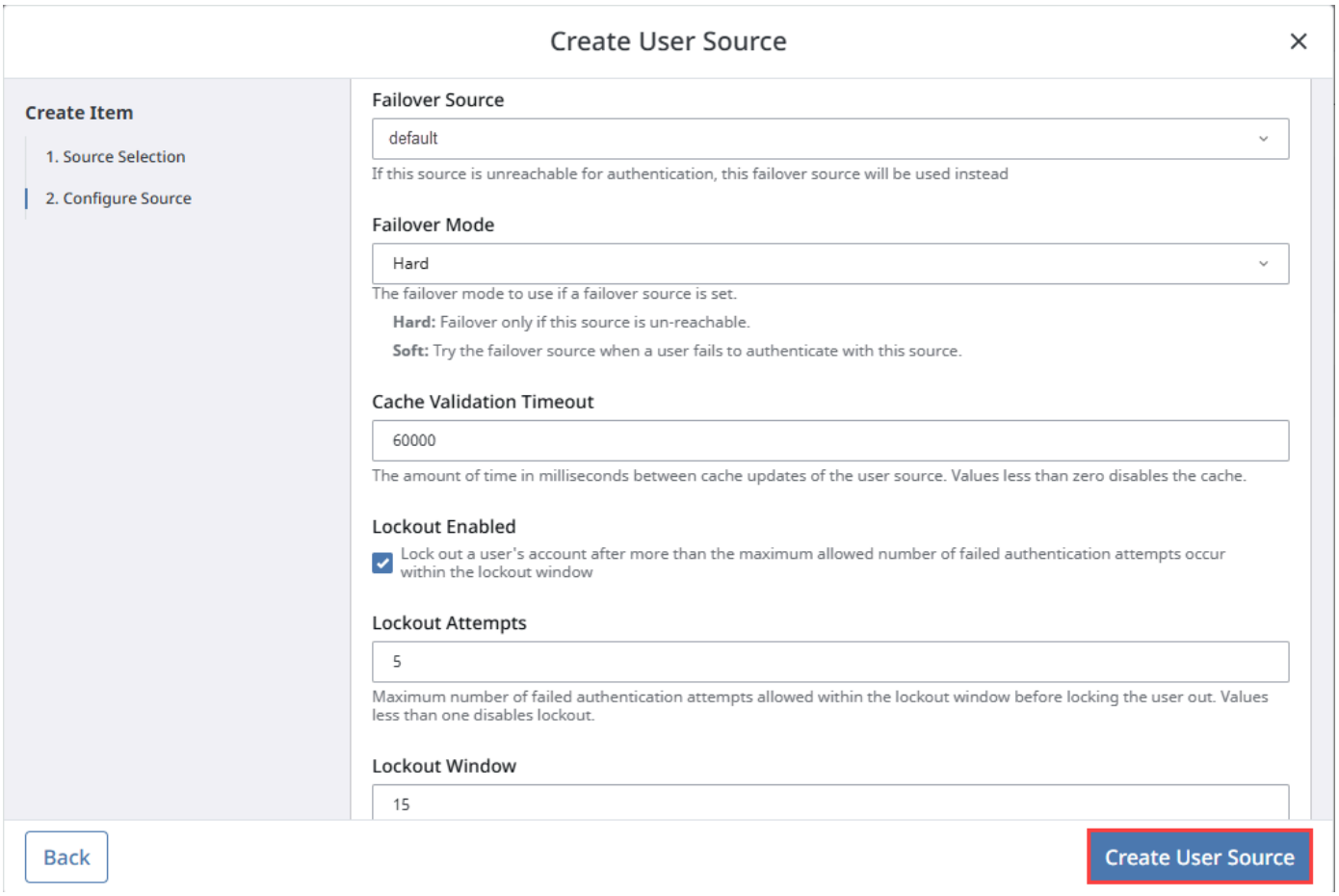
3. Choose the **Database** authentication type, and click **Next**.



4. The Configure Source form will open. Some properties are optional, but if you're using Automatic mode, enter the following properties as appropriate.

    - **Name**: DBAuth
    - **Failover Source**: default
    - **Failover Mode**: Hard
    - **Database**: MySQL
    - **Mode**: Automatic
    - **Tablename Prefix**: auth_ (You can leave this field blank, but if you use a prefix when the tables get created, they will contain the specified prefix in their name).

    When finished, click **Create User Source**.

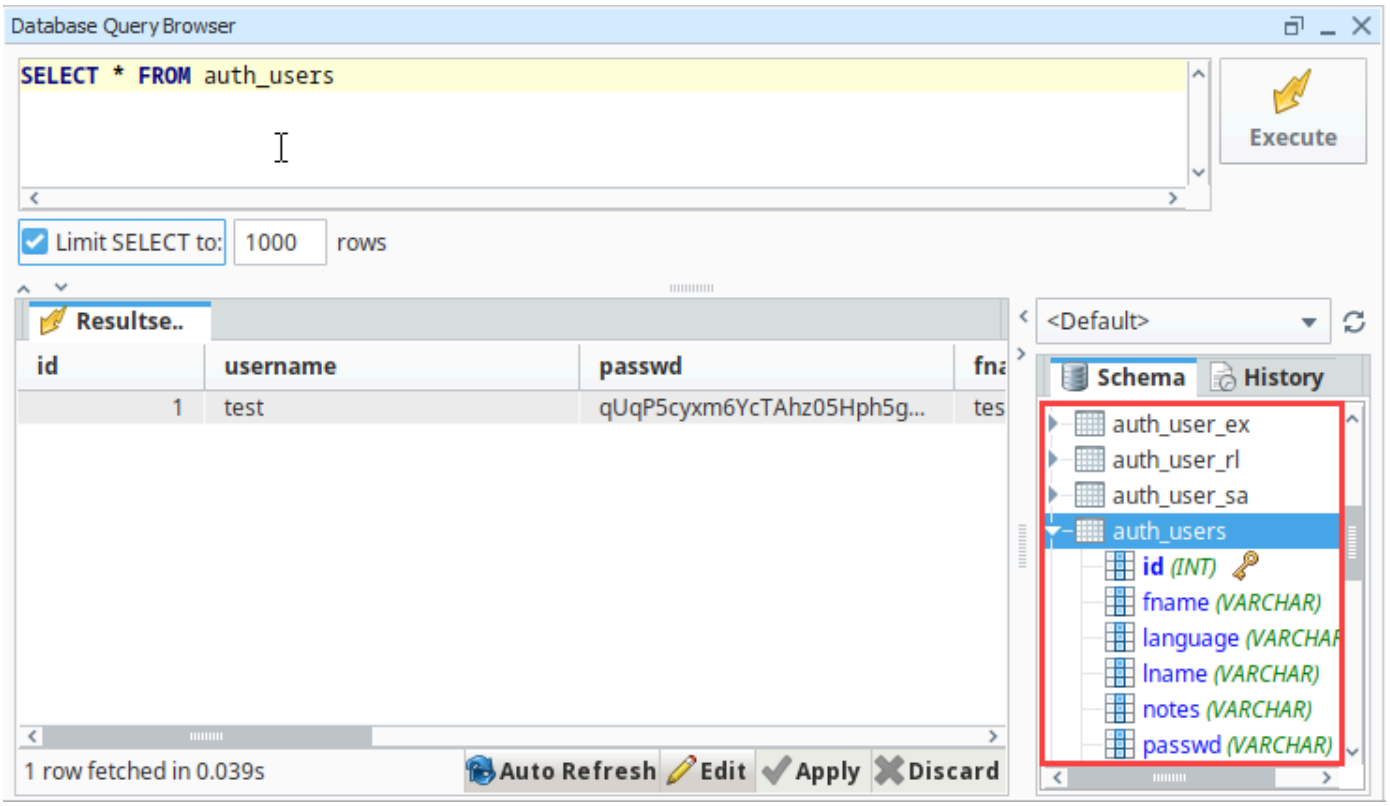


> **TABLE CREATION**
>
> The tables in the database will not be created in the database until they are needed. For example, as soon as a user or role is added, the associated tables will automatically get created.

5. Now that your Authentication profile is created, add a user. On the right, click on the **three dot** menu and select **Manage Users**. Click **Create User +** and fill in the required fields.

6. To view and verify the tables, go into **Designer** and from the menu bar, select **Tools** > **Database Query Browser**.You will see all the tables that were created beginning with 'auth_' when the user and role get created.

7. Double click on any of tables beginning with '**auth_**', and click **Execute**. In this example, you will see the tables associated with 'roles' and 'users' displayed in the Schema area.



Password Encryption

When a database user source Mode is set to Automatic, Ignition encrypts the user's password in the database using SHA1. However, it is possible to allow users to change their passwords using an update query. The script below takes a password string, encrypts it, and updates the password in the database.

Simple Password Encryption Script

```
# Get the current logged in user's username.
username = system.tag.readBlocking(["[System]Client/User/Username"])[0].value

# Take the user's new password from a component. We are using the Password Field
component.
newPass = event.source.parent.getComponent('Current Password').text

# Import SecurityUtils from com.inductiveautomation.ignition.common.util.
from com.inductiveautomation.ignition.common.util import SecurityUtils

# Encrypt the password from the second line using SecurityUtils.
# This will encrypt your password as SHA1.
encryptPass = SecurityUtils.sha1String(newPass)

# Add our update query.
query = "UPDATE scada_users SET passwd = ? WHERE username = ?"

# Execute our query to update the password in the database.
system.db.runPrepUpdate(query, (encryptPass, username), 'database_connection')
```

This is a simplified script, but you can add additional logic to your script. For example, the following script also validates the user's current password before allowing them to update their password.

Password Encryption Script with User Validation

```
# Get the current logged in user's username.
username = system.tag.readBlocking(["[System]Client/User/Username"])[0].value

# Create a variable for the user's current password. In the client this is where
the user enters their current password.
password = event.source.parent.getComponent('Current Password').text

# If the field isn't empty, continue; otherwise warn the user.
if password != "":
    # Validate the username and password for the current logged-in user.
    confirmPassword = system.security.validateUser(username, password)

    if confirmPassword:
        # Get new password entries (must match).
        pass1 = event.source.parent.getComponent('Password 1').text
        pass2 = event.source.parent.getComponent('Password 2').text

        if pass1 == pass2:
            # Encrypt the new password as SHA1.
            from com.inductiveautomation.ignition.common.util import SecurityUtils
            newPass = SecurityUtils.sha1String(pass1)

            # Update the password in the database.
            query = "UPDATE support_users SET passwd = ? WHERE username = ?"
            system.db.runPrepUpdate(query, (newPass, username),
'database_connection')

            # Inform the user.
            system.vision.showMessage("Your password has been updated.")
        else:
            system.vision.showWarning("Password 1 and Password 2 do not match")
    else:
        system.vision.showWarning("Incorrect password")
else:
    system.vision.showWarning("Enter current password")
```

Active Directory Authentication

Active Directory User Source

The Active Directory Authentication profile uses Microsoft's Active Directory over **LDAP** (Lightweight Directory Access Protocol) to store all the users, roles, and more that make up an Authentication profile. Active Directory **Groups** are used for Ignition's **roles** and user-role mappings.

While using an Active Directory User Source, administration of users and roles is through Active Directory itself, and not manageable within Ignition. Thus adding new users to an Active Directory User Source, or modifying pre existing users, requires the modifications be made from Active Directory, usually through an AD Administrator.

Active Directory User Sources supports SASL (Simple Authentication and Security Layer). SASL is a framework for authentication and data security in Internet protocols such as LDAP.

Property Reference

Active Directory User Sources have the following properties shown in the table below, organized by category.

> **NOTE**
>
> Certain properties in the Active Directory User Source allow you to filter users, such as the **User List Filter**. These filters only determine which users will be displayed on screen. They are not authentication filters, so even if a user does not show in the list they can still authenticate and may have access to unintended areas. Be sure to configure **project security** appropriately to prevent this from happening!

General Properties

Details on the General Properties can be found on the **Classic Authentication Strategy** page.

Active Directory Properties

Name	Description
Domain	The Windows Domain your Active Directory server is running on. If you aren't sure of your domain, ask your network administrator. Leave blank to set advanced properties manually.
Gateway Username	The login name for the Gateway to use when querying Active Directory. Used for retrieving the list of users and roles via LDAP.
Password	The password selections available for the Active Directory. Options include None, Embedded, and Referenced. See the **Secrets Management** page for more information on using Referenced passwords.
Primary Domain Controller Host	The IP address or hostname of your primary domain controller. Example: "192.168.1.4" or "MainServer"
Primary Domain Controller Port	The port number for the primary domain controller's LDAP interface.
Secondary Domain Controller Host	The IP address or hostname of your secondary domain controller (optional). Example: "192.168.1.4" or "MainServer"
Secondary Domain Controller Port	The port number for the secondary domain controller's LDAP interface.
Use SSL	Disable to use "**ldap://**" protocol, enable to use "**ldaps://**"
SSO Enabled	This setting was disabled and deprecated in 8.1.17 to protect against a potential security vulnerability. While the property is still visible, it cannot be enabled without setting a special system property. This is not recommended. See the **SSO Enabled Deprecated Property** page for more information.
SSO Domain	The domain that Windows users must match in order to use SSO. If blank, the main "Domain" property will be used. Not case-sensitive.

Name	Description
Nested Groups	If this option is enabled, nested group membership lookups will be performed. Enabled by default.

LDAP Search Properties

The following table describes various configurable LDAP properties. See Microsoft's **official documentation** on LDAP Syntax Filters for more in-depth information about LDAP.

Name	Description
Username Prefix	This prefix will be prepended to the username before an Active Directory bind is attempted for authentication.
Username Suffix	This suffix will be appended to the username before an Active Directory bind is attempted for authentication.
Automatic Suffix	If this option is checked, and the suffix is left blank, then the suffix will automatically be assigned a value of `@<domain>`.
Use prefix and suffix for Gateway username	If this option is checked, the username prefix and suffix will be applied to the Gateway username before a bind is attempted. This option is checked by default.
User Search Base	The base folder to search for users under, such as: `DC=MyCompany,DC=com` The entire subtree under this folder will be searched using the User Search Filter. Multiple subtrees can be specified by putting them in parenthesis, like so: `(OU=Administrators,DC=MyCompany,DC=com)` `(OU=Operators,DC=MyCompany,DC=com)`
User Search Filter	The LDAP search filter that will be used to find a specific user. Use the placeholder 0 as a stand-in for the login name.
User List Filter	The LDAP search filter used when querying for the list of all users. Should restrict the type to user.
User Name Attribute	The attribute on the User object to define the username. Default is `sAMAccountName`.
User Role Attribute	Attributes of this name on the User object will define the user's roles.
User Role Name Attribute	The attribute of this name on the Role object will define the role's name. Leave blank to use the raw value of the attribute defined by **User Role Attribute**.
Full Name Attribute	The attribute on the User object to define the full name of the user.

Name	Description
Phone Attribute	The attribute name on the user object that represents the user's phone number.
Email Attribute	The attribute name on the user object that represents the user's email address.
SMS Attribute	The attribute name on the user object that represents the phone number that this user receives text messages on.
Badge Attribute	The attribute on the User object to define the badge. This setting is required to enable badge-based authentication.
Connect Timeout	**New in 8.3.2** The connect timeout in milliseconds for LDAP operations. A value smaller than one will force the LDAP operation to wait until the underlying network protocol's timeout period has been reached. Default is 0.
Read Timeout	The read timeout in milliseconds for LDAP operations.
Results Page Size	The number of entries returned per page of results in a query.
Role Search Base	The base folder to search for roles under, such as: `OU=Roles,DC=MyCompany,DC=com` The entire subtree under this folder will be searched using the Role Search Filter. If you specify the root of your tree structure, the search may take a very long time. Multiple subtrees can be specified by putting them in parenthesis, like so: `(OU=Builtin,DC=MyCompany,DC=com)(OU=Users,DC=MyCompany,DC=com)` If you leave this blank the whole subtree of the domain controller will be searched.
Role Search Filter	The LDAP search filter that will be used to locate roles.
Badge Search Filter	The LDAP search filter to use to find a specific user given a badge. Use the placeholder 0 as a stand-in for the user's badge. For example, `(&(objectClass=user)(badge=0))`.
Allow Anonymous	If enabled, authentication attempts with blank passwords will be passed through to LDAP, which may choose to accept them.

Name	Description
	Caution: It is highly recommended to disable this setting unless you know it is required. AD servers may allow logging in as any user with a blank password when Security Authentication is set to “None” or “Simple” (even if a provided username does not exist in AD), which is a major security risk.
Security Protocol	Specifies the security protocol between the Gateway and AD server. The following options are available: • **AUTO**: No security protocol is explicitly used or requested by the Gateway. • **SSL**: SSL should be used for the connection.
Security Authentication	This property specifies how usernames and passwords are used to bind to LDAP. The following options are available: • **AUTO**: Unspecified from the Gateway side, meaning the LDAP implementation will choose. • **NONE**: Anonymous access. **(Not recommended due to security risks)** • **SIMPLE**: Plaintext username and passwords will be used. **(Not recommended due to security risks)** • **STRONG**: Usernames and passwords will be encrypted. • **SASL**: Simple Authentication and Security Layer. See the SASL Properties table below for additional SASL authentication configuration settings.
Referral	Specifies how referrals will be processed. The following options are available: • **follow**: Follow referrals automatically. • **ignore**: Referrals are ignored. • **throw**: Throws a ReferralException when a referral is encountered.
Legacy Naming Enabled	**New in 8.3.7** Determines how user and role search base entries are parsed. When checked, the 8.1 default behavior is enabled. This setting is available so that Active Directory User Sources are properly migrated from 8.1 to 8.3 Gateways. Default is true when upgrading from 8.1. Default is false when upgrading from a prior 8.3 version. It is

Name	Description
	recommended to only enable this setting when experiencing issues parsing user and role entries as valid LDAP names.

LDAP Attributes

A list of additional LDAP attributes to retrieve and store on the returned user.

Name	Description
Attribute	The name of the attribute to retrieve from LDAP.
Property	The property name that the attribute value will be stored under on the returned User object. If no property name is given, the attribute name will be used as the property name.
Binary	If checked, the value(s) will be treated as a byte[] array.
Multi Valued	If checked, the property will be treated as a multi-valued property and be retrievable as a list.

SASL Properties

These settings are utilized when Security Authentication is set to **SASL**.

Name	Description
Mechanism	An ordered list of space-separated mechanism names. The LDAP provider will use the first mechanism for which it finds an implementation. A blank value will leave this setting unspecified. (Default is `DIGEST-MD5` `CRAM-MD5`).
Realm	A realm defines the namespace from which the user is selected. A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is blank.
Quality of Protection	A comma-separated list of Quality-of-Protection (QoP) values, the order of which specifies the preference order. There are three well-known values: "auth" (authentication only), "auth-int" (authentication with integrity protection), and "auth-conf" (authentication with integrity and privacy protection). A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. (Default is `auth-conf`, `auth-int`, `auth`).
Protection Strength	A comma-separated list of privacy protection strength values, the order of which specifies the preference order. The three possible strength values are "low", "medium", and "high". A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is `high`, `medium`, `low`.
Mutual Authentication	Enable or disable mutual authentication. This setting will only be used by mechanisms which support it. Default is disabled.

To Create an Active Directory User Source

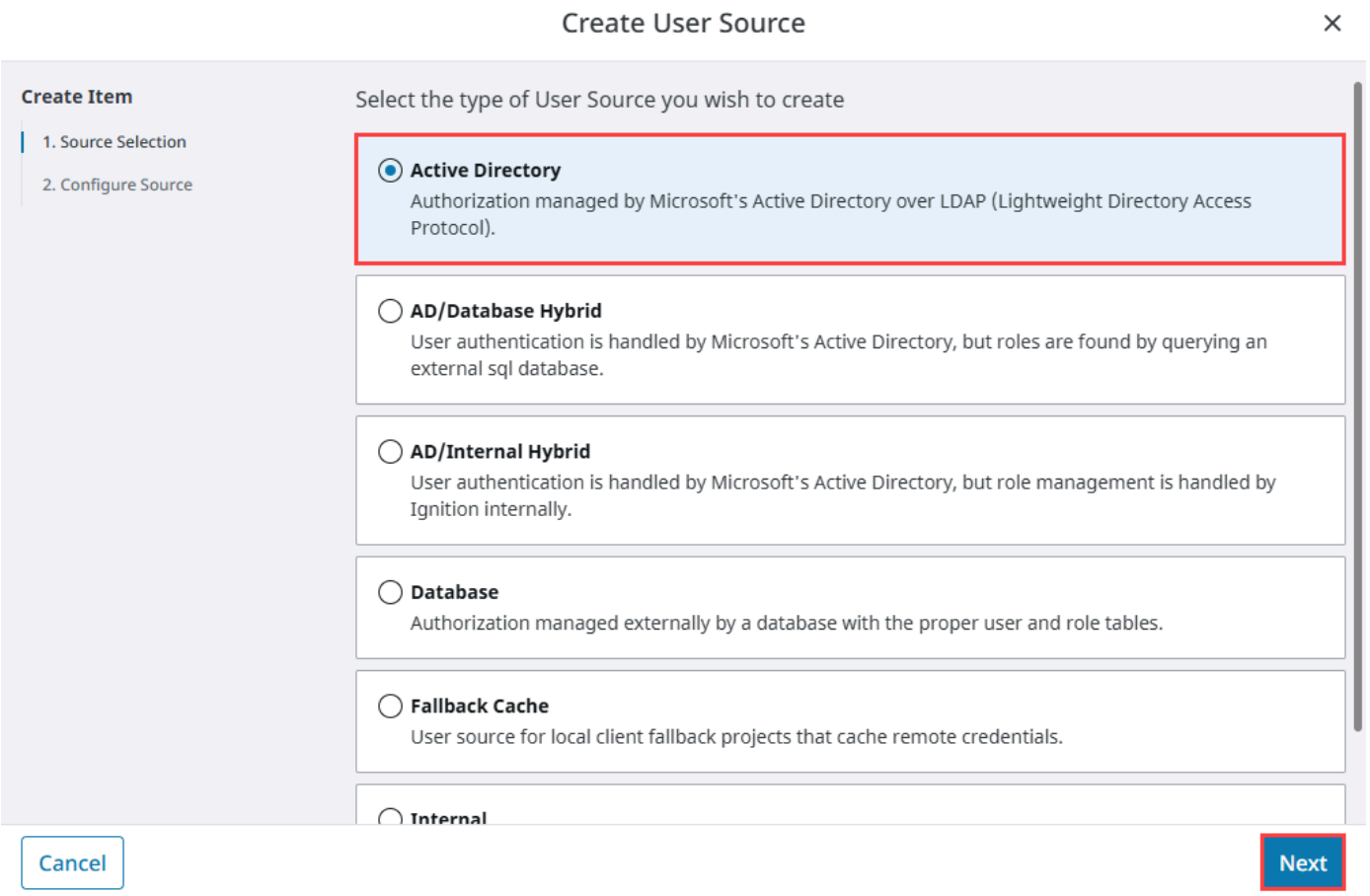
To configure an Active Directory User Source, you must specify the host that is acting as your primary domain controller. You can also use a secondary domain controller in case the primary is unavailable. You'll also need to specify the name of the domain and credentials for the Gateway itself to use: the Gateway needs a user account to interact with the AD server, even when it's simply querying for a list of roles.

> **NOTE**
>
> When using Active Directory User Source, you may need to consult with your internal IT Department to get the required information to complete your user source setup. These settings

are common to AD (not specific to Ignition), and your IT department will know what values to supply to each property.

1. On the Gateway Webpage, under the **Platform** section, go to **Security** > **User Sources**. The User Sources page will be displayed. Click **Create User Source +**.

2. Choose the **Active Directory** authentication type, and click **Next**.



3. The Configure Source form will be displayed. Some properties are optional. In the very least, you must specify the following: **Domain, Gateway Username, Password, Primary Domain Controller Host**.

4. Click **Create User Source** to create the User Source.

Connect AD over SSL

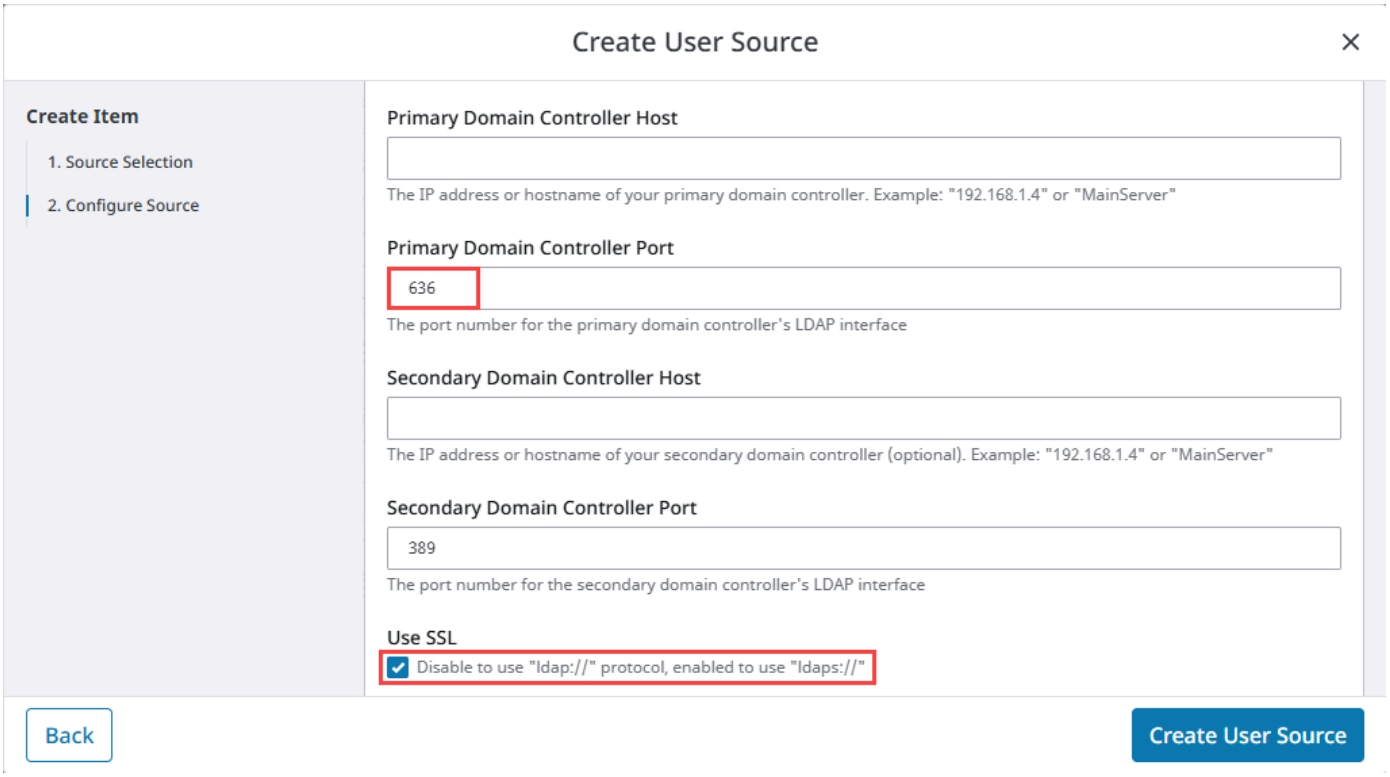
For additional security, you can adjust the Active Directory settings to enable SSL since LDAP is not encrypted by default. The default port for LDAP is port 389, but LDAPS uses port 636 and establishes SSL upon connecting.

1. On the Gateway Webpage, under the Platform section, go to **Security** > **User Sources**.

2. Click **Create User Source +** on the User Sources page.

3. Choose the **Active Directory** authentication type, and click **Next**.

4. Change the **Primary Domain Controller Port** to 636.



5. Check **Use SSL** to enable “ldaps://”.

6. Click the **Show Advanced +** button to expand advanced properties.

7. Under LDAP Search Properties, change the **Security Protocol** to SSL.

8. Fill in the remaining required properties as needed and click **Create User Source**.

If you try to query or authenticate against the AD server at this point, you will receive the following error:

```
Caused by: javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorException: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
```

This error indicates Ignition was unable find a valid certificate generated from the AD server, and therefore cannot validate the AD server's identity. Work with IT to obtain a certificate from the AD server. This certificate must be added to the data/certificates/supplemental directory and then imported into the Java cacerts keystore. Once the certificate is added, restart the Gateway.

AD Internal Hybrid

AD/Internal User Source

The Active Directory/Internal Hybrid authentication profile type combines the Internal User Source type with the Active Directory User Source type. Active Directory is used to find all of the users, and to check their credentials when they attempt to log in. However, it allows assigning of roles, contact info, and other meta-information about a user through Ignition, then stores all this information as if it were an Internal User Source. This way, Active Directory can be consulted to see if a username/password is valid, but the management of roles does not require coordination with your IT Department, who typically controls the Active Directory system. This "best of both worlds" approach is popular for many users of Active Directory.

The AD/Internal Hybrid User Source is partially manageable in Ignition. Users cannot be added or removed, and their usernames and passwords cannot be changed. This is because this information resides in Active Directory, not within Ignition. Other information, such as user roles, contact info, schedules, are **manageable** in Ignition.

> **GATEWAY SETTINGS**
>
> Before you can use the User Management component to manage roles, contact info, etc., you first have to go into Gateway Settings, and mark the checkbox to **'Allow User Admin.'** This allows for the administration of the Gateway's system user source from the Designer and the Client. Unless this is enabled, the Vision Module's User Management component is prevented from modifying the Gateway system's user source.

Property Reference

This **User Source** shares many properties with the AD User Source. Please see the **Active Directory Authentication** page for a list of properties.

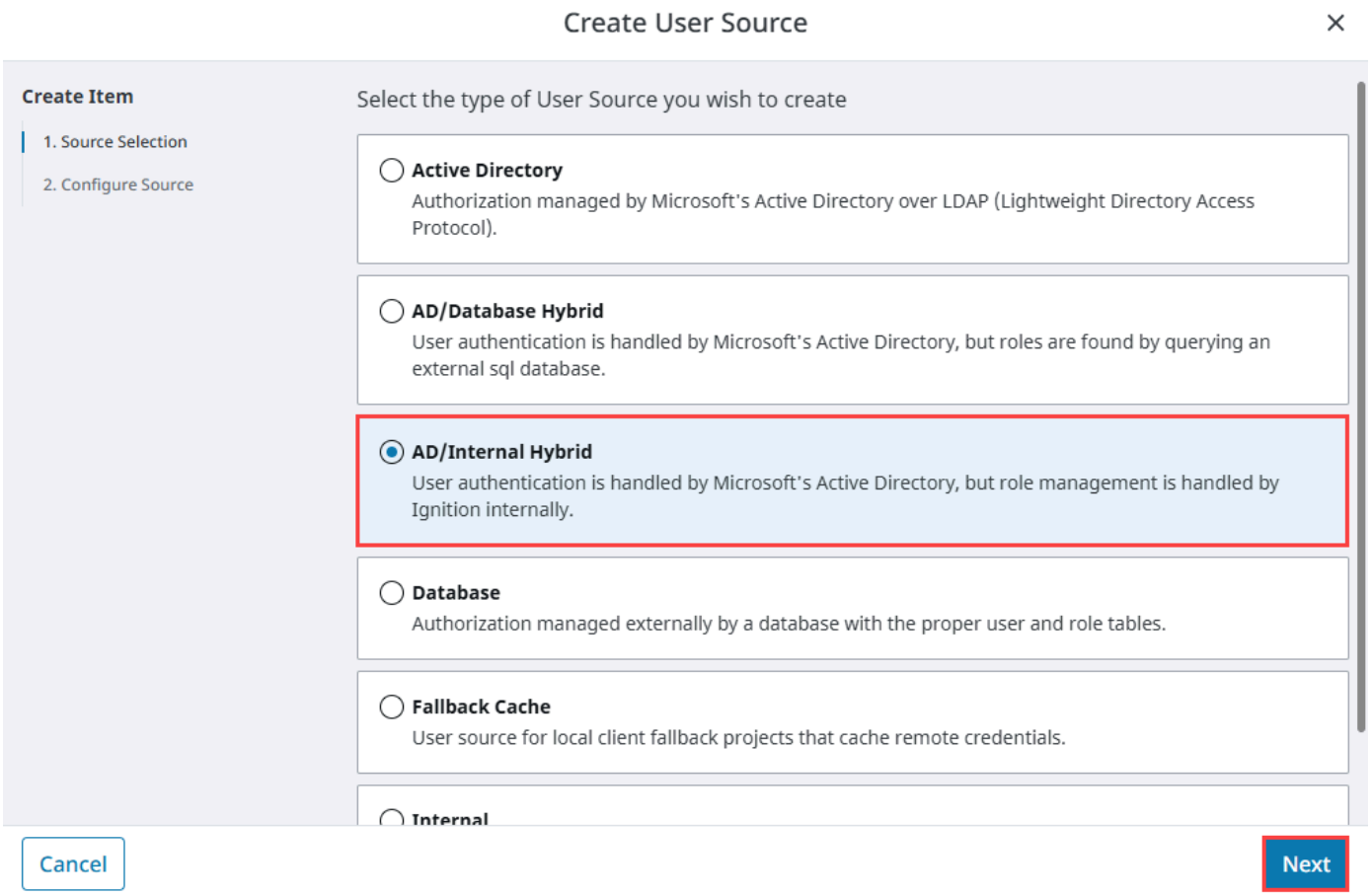
Creating an AD/Internal Hybrid User Source

To set up an AD/Internal Hybrid User Source, you must specify the host that is acting as your primary domain controller. You can also use a secondary domain controller in case the primary is unavailable. You'll also need to specify the name of the domain and credentials for the Gateway itself to use for authentication for when it queries the list of roles.

> **CONTACT YOUR IT DEPARTMENT**
>
> When using AD/Internal Hybrid User Source, you may need to consult with your internal IT Department to get the required information to complete your user source setup.

1. On the **Gateway webpage**, under the Platform tab, go to **Security > User Sources**. The User Sources page will be displayed. Click **Create User Source +**.

2. Choose the **AD/Internal Hybrid** authentication type, and click **Next**.



3. The Configure Source form will open. Some properties are optional depending on how you set up your profile. Details on the General Properties can be found on the **User Sources** page. Active Directory properties are listed in the table below.

4. Click **Create User Source** to save the new user source.

Active Directory Properties

Name	Description
Domain	The Windows Domain your active Active Directory server is running on. If you aren't sure of your domain, ask your network administrator. Leave blank to set advanced properties manually.
Primary Domain Controller Host	The IP address or hostname of your primary domain controller. Example: "192.168.1.4" or "MainServer"
Primary Domain Controller Port	The port number for the primary domain controller's LDAP interface.
Use SSL	Disable to use "**ldap://**" protocol, enable to use "**ldaps://**"
List Users from Active Directory	If true, Active Directory will be queried for the list of all users. If false, users must be added manually. Default is true.
Populate Users On-Demand	If true, a user record will be created in the Internal Database from the AD entry of any user who successfully authenticates into Active Directory if the record does not already exist in the Internal Database at the time of authentication. When the User Source is paired with an **Ignition Identity Provider**, enabling this option can provide a modest performance improvement during session login.
Gateway Username	The login name for the Gateway to use when querying Active Directory. Used for retrieving the list of users and roles via LDAP.
Password	The password selections available for the Active Directory. Options include None, Embedded, and Referenced. See the **Secrets Management** page for more information on using Referenced passwords.
SSO Enabled	Whether or not to use Single-Sign-On (SSO) to authenticate AD users. Note that projects must also have this option enabled for SSO to work. Default is false.

Name	Description
	This setting was disabled and deprecated to protect against a potential security vulnerability. While the property is still visible, it cannot be enabled without setting a special system property. This is not recommended. Check out the **Active Directory SSO Disabled** article for more information.
SSO Domain	The domain that Windows users must match in order to use SSO. If blank, the main "Domain" property will be used. Not case-sensitive.

Advanced Properties

The following table describes several configurable LDAP properties. See Microsoft's **official documentation** on LDAP Syntax Filters for more in-depth information about LDAP.

Name	Description
Secondary Domain Controller Host	The IP address or hostname of your secondary domain controller (optional). Example: "192.168.1.5" or "BackupServer"
Secondary Domain Controller Port	The port number for the secondary domain controller's LDAP interface. Default: 389.
Connect Timeout	**New in 8.3.2** The connect timeout in milliseconds for LDAP operations. A value smaller than one will force the LDAP operation to wait until the underlying network protocol's timeout period has been reached. Default is 0.
Read Timeout	The read timeout in milliseconds for LDAP operations. Default is 60,000.
Results Page Size	The number of entries returned per page of results in a query. Default is 1,000.
Username Prefix	This prefix will be prepended to the username before an Active Directory bind is attempted for authentication.
Username Suffix	This suffix will be appended to the username before an Active Directory bind is attempted for authentication.
Automatic Suffix	If this option is checked, and the suffix is left blank, then the suffix will automatically be assigned a value of `@<domain>`. Default is true.
Use prefix and suffix for Gateway username	If this option is checked, the username prefix and suffix will be applied to the Gateway username before a bind is attempted. This option is checked by default.
User Listing Base	The base folder to search for users under, such as `DC=MyCompany,DC=com`. The entire subtree under this folder will be searched using the User List Filter. Multiple subtrees can be specified by putting them in parenthesis, like so: `(OU=Administrators,DC=MyCompany,DC=com)` `(OU=Operators,DC=MyCompany,DC=com)` If you leave this blank the whole subtree of the domain controller will be searched.
User List Filter	The LDAP search filter used when querying for the list of all users. Should restrict the type to user. Default is `(&(objectClass=user)(!`

Name	Description
	`(objectClass=computer)))`.
User Search Filter	The LDAP search filter to use to find a specific user. Use the placeholder 0 as a stand-in for the login name. Default is `(&(objectClass=user)(sAMAccountName={0}))`.
User Name Attribute	The attribute on the User object to define the username. Default is sAMAccountName.
Full Name Attribute	The attribute on the User object to define the full name of the user.
Phone Attribute	The attribute name on the user object that represents the user's phone number.
Email Attribute	The attribute name on the user object that represents the user's email address.
SMS Attribute	The attribute name on the user object that represents the phone number that this user receives text messages on.
Allow Anonymous	If enabled, authentication attempts with blank passwords will be passed through to LDAP, which may choose to accept them. **Caution**: It is highly recommended to disable this setting unless you know it is required. AD servers may allow logging in as any user with a blank password when Security Authentication is set to “None” or “Simple” (even if a provided username does not exist in AD), which is a major security risk.
Security Protocol	Auto or SSL. Default is Auto.
Security Authentication	This property specifies how usernames and passwords are used to bind to LDAP. The following options are available: • **AUTO**: Unspecified from the Gateway side, meaning the LDAP implementation will choose. • **NONE**: Anonymous access. • **SIMPLE**: Plaintext username and passwords will be used. • **STRONG**: Usernames and passwords will be encrypted.

Name	Description
	• **SASL**: Simple Authentication and Security Layer. See the SASL Properties table below for additional SASL authentication configuration settings.
Referral	Specifies how referrals are to be processed. Possible options are: • **Follow**: Always automatically follow referrals. This is the default option. • **Ignore**: Ignores referrals. • **Throw**: Throws a ReferralException whenever a referral is encountered.
Legacy Naming Enabled	**New in 8.3.7** Determines how user and role search base entries are parsed. When checked, the 8.1 default behavior is enabled. This setting is available so that Active Directory User Sources are properly migrated from 8.1 to 8.3 Gateways. Default is true when upgrading from 8.1. Default is false when upgrading from a prior 8.3 version. It is recommended to only enable this setting when experiencing issues parsing user and role entries as valid LDAP names.

SASL Properties

These settings are utilized when Security Authentication is set to **SASL**.

Name	Description
Mechanism	An ordered list of space-separated mechanism names. The LDAP provider will use the first mechanism for which it finds an implementation. A blank value will leave this setting unspecified. Default is `DIGEST-MD5` `CRAM-MD5`.
Realm	A realm defines the namespace from which the user is selected. A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is blank.
Quality of Protection	A comma-separated list of Quality-of-Protection (QoP) values, the order of which specifies the preference order. There are three well-known values: "auth" (authentication only), "auth-int" (authentication with integrity protection), and "auth-conf" (authentication with integrity and privacy protection). A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is `auth-conf`, `auth-int`, `auth`.
Protection Strength	A comma-separated list of privacy protection strength values, the order of which specifies the preference order. The three possible strength values are "low", "medium", and "high". A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is `high`, `medium`, `low`.
Mutual Authentication	Enable or disable mutual authentication. This setting will only be used by mechanisms which support it. Default is disabled.

LDAP Attributes

A list of additional LDAP attributes to retrieve and store on the returned user.

Name	Description
Attribute	The name of the attribute to retrieve from LDAP.
Property	The property name that the attribute value will be stored under on the returned User object. If no property name is given, the attribute name will be used as the property name.
Binary	If checked, the value(s) will be treated as a byte[] array.
Multi Valued	If checked, the property will be treated as a multi-valued property and be retrievable as a list.

AD Database Hybrid

AD/Database User Source

This AD/Database Hybrid User Source is not **manageable** from within Ignition. Users/passwords must be administered through **Active Directory**, and roles, contact info, and so on, must be administered directly through the database. The way AD/Database Hybrid works, is it has all the same information requirements as the other authentication profiles, but it also has a number of Database properties. You need to specify a database (i.e., MySQL) to store information, and set up queries that you want to use. You must also specify the host that is acting as your primary domain controller, and a secondary domain controller in case the primary is unavailable. You'll also need to specify the name of the domain and credentials for the Gateway itself to use for authentication when it queries the list of roles.

AD/Database User Sources now support Simple Authentication and Security Layer (SASL). SASL is a framework for authentication and data security in Internet protocols such as LDAP.

Property Reference

This **User Source** shares many properties with both the AD User Source and Database User Source.

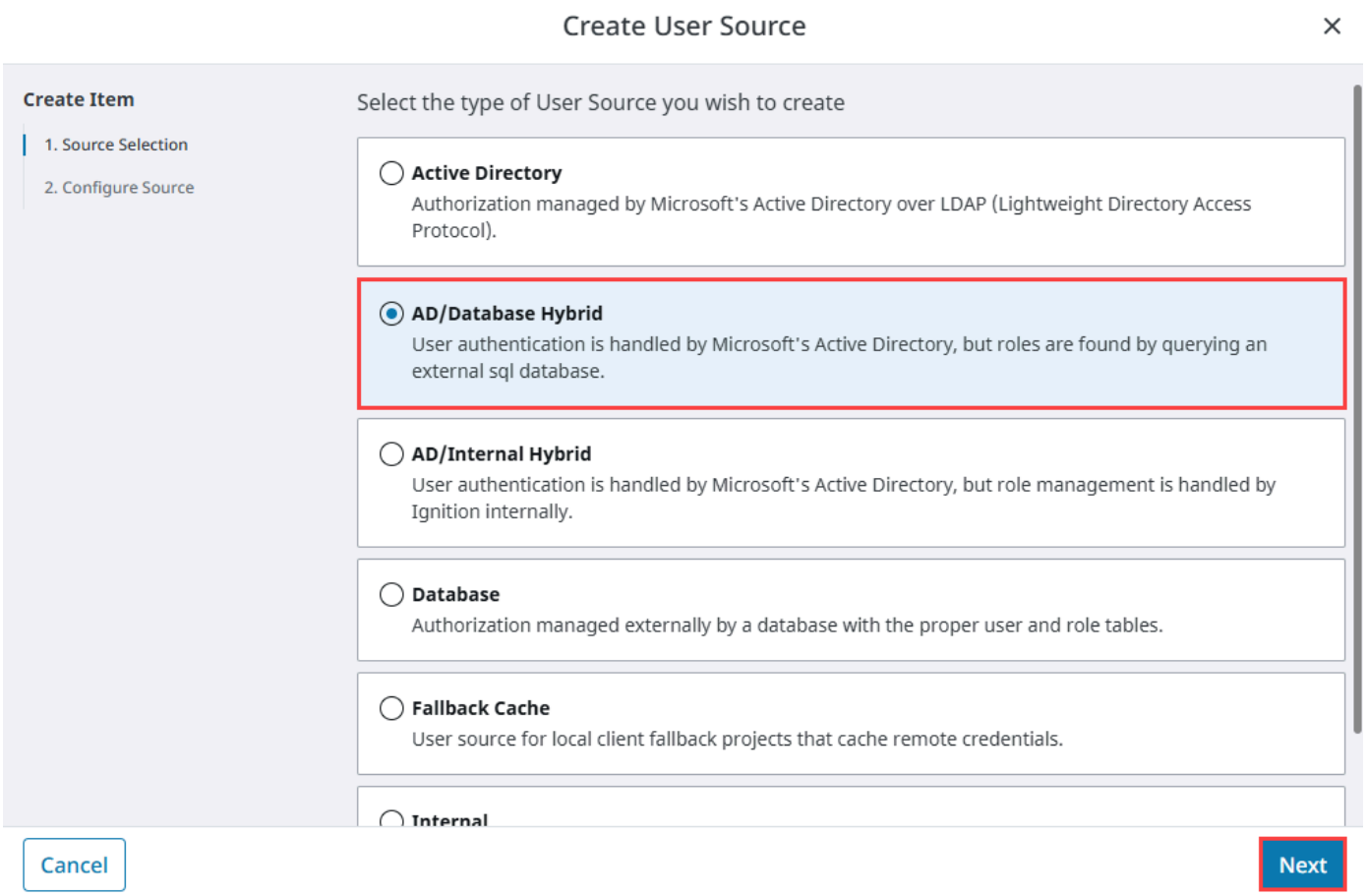
- See the **Active Directory Authentication** page for a list of Active Directory User Source related properties.
- See the **Database Authentication** page for a list of Database User Source related properties.

Creating an AD/Database Hybrid User Source

1. On the Gateway Webpage, under the **Platform** section, go to **Security** > **User Sources**. The User Sources page will be displayed.

2. Click **Create User Source +**.

3. Choose the **AD/Database Hybrid** authentication type, and click **Next**.



> **NOTE**
>
> When using AD/Database Hybrid User Source, you may need to consult with your internal IT Department to get the required information to complete your user source setup.

4. The User Source Setup window will display configuration settings. Some properties are optional depending on how you set up your profile. Details on the General Properties can be found on the **User Sources** page. The Active Directory Properties and Database properties are listed in the tables below.

5. Click **Create User Source** to save the new user source.

Active Directory Properties

Name	Description
Domain	The Windows Domain your active Active Directory server is running on. If you aren't sure of your domain, ask your network administrator. Leave blank to set advanced properties manually.
Primary Domain Controller Host	The IP address or hostname of your primary domain controller. Example: "192.168.1.4" or "MainServer"
Primary Domain Controller Port	The port number for the primary domain controller's LDAP interface.
List Users from Active Directory	If true, Active Directory will be queried for the list of all users. If false, the database will be queried instead. Default is true.
Gateway Username	The login name for the Gateway to use when querying Active Directory. Used for retrieving the list of users and roles via LDAP.
Password	The password selections available for the Active Directory. Options include None, Embedded, and Referenced. See the **Secrets Management** page for more information on using Referenced passwords.
SSO Enabled	Whether or not to use Single-Sign-On (SSO) to authenticate AD users. Note that projects must also have this option enabled for SSO to work. Default is false. This setting was disabled and deprecated to protect against a potential security vulnerability. While the property is still visible, it cannot be enabled without setting a special system property. This is not recommended. Check out the **Active Directory SSO Disabled** article for more information.
SSO Domain	The domain that Windows users must match in order to use SSO. If blank, the main "Domain" property will be used. Not case-sensitive.

Database Properties

Name	Description
Database	Dropdown list. Choose the database connection this authentication profile will use.
User Properties Query	A query that returns the basic properties for a single user. Supported return columns are [username, firstname, lastname, schedule, language, notes].
Role List Query	A query that returns all possible roles that any user could have. The role names must be returned in the first column of the query's results.
User's Roles Query	A query that returns all of the roles that the provided user belongs to. The roles must be strings (i.e., the role names), and must be in the first column of the query's results. The username will be inserted into this query as a parameter.
Contact Info Query	A query that returns all of the contact info for the user. The first column must be the contact type, the second column the contact value, and the third column the name of a schedule. Optional, may be blank.
Schedule Adjustment Query	A query that returns the upcoming schedule adjustments for the user. Columns must be Start(date), End(date), Available(boolean), Note(string). Optional, may be blank.
Extra Properties Query	A query that returns name, value pairs of extra properties for the user. Will be run with one parameter: the username. Optional, may be blank.
Find User Query	A query that must return a row if a user with the given username exists. The query will be run as a prepared statement, so use question mark (?) to represent username. There must be at least one column: the username. Other columns are optional, supported columns are: [username, firstname, lastname, schedule, language, notes, badge]. Example: `SELECT username, firstname, lastname, schedule, FROM USERS WHERE username = ?` When the User Source is paired with an **Ignition Identity Provider**, enabling this option can provide a modest performance improvement

Name	Description
	during session login.
List Users Query	A query that returns a row containing each username. Only used if "List Users from Active Directory" is false. There must be at least one column: the username. Other columns are optional, supported columns are: username, firstname, lastname, schedule, language, notes.
Badge Authentication Query	A query that must return a row if the given badge provided is valid. The query will be run as a prepared statement, so use question mark (?) to represent the badge. The returned row must contain the username. Example: `SELECT username FROM USERS WHERE badge = ?`

Advanced AD Properties

The following table describes several configurable LDAP properties. See Microsoft's **official documentation** on LDAP Syntax Filters for more in-depth information about LDAP.

Name	Description
Secondary Domain Controller Host	The IP address or hostname of your secondary domain controller (optional). Example: "192.168.1.5" or "BackupServer"
Secondary Domain Controller Port	The port number for the secondary domain controller's LDAP interface. Default: 389.
Connect Timeout	**New in 8.3.2** The connect timeout in milliseconds for LDAP operations. A value smaller than one will force the LDAP operation to wait until the underlying network protocol's timeout period has been reached. Default is 0.
Read Timeout	The read timeout in milliseconds for LDAP operations. Default is 60,000.
Results Page Size	The number of entries returned per page of results in a query. Default is 1,000.
User Listing Base	The base folder to search for users under, such as `DC=MyCompany,DC=com`. The entire subtree under this folder will be searched using the User List Filter. Multiple subtrees can be specified by putting them in parenthesis, like so: `(OU=Administrators,DC=MyCompany,DC=com)` `(OU=Operators,DC=MyCompany,DC=com)` If you leave this blank the whole subtree of the domain controller will be searched.
User List Filter	The LDAP search filter used when querying for the list of all users. Should restrict the type to user. Default is `(&(objectClass=user)(!(objectClass=computer)))`.
User Search Filter	The LDAP search filter to use to find a specific user. Use the placeholder 0 as a stand-in for the login name. Default is `(&(objectClass=user)(sAMAccountName={0}))`.
User Name Attribute	The attribute on the User object to define the username. Default is sAMAccountName.
Username Prefix	This prefix will be prepended to the username before an Active Directory bind is attempted for authentication.

Name	Description
Username Suffix	This suffix will be appended to the username before an Active Directory bind is attempted for authentication.
Automatic Suffix	If this option is checked, and the suffix is left blank, then the suffix will automatically be assigned a value of `@<domain>`. Default is true.
Use prefix and suffix for Gateway username	If this option is checked, the username prefix and suffix will be applied to the Gateway username before a bind is attempted. This option is checked by default.
Allow Anonymous	If enabled, authentication attempts with blank passwords will be passed through to LDAP, which may choose to accept them. **Caution**: It is highly recommended to disable this setting unless you know it is required. AD servers may allow logging in as any user with a blank password when Security Authentication is set to “None” or “Simple” (even if a provided username does not exist in AD), which is a major security risk.
Security Protocol	Auto or SSL. Default is Auto.
Security Authentication	This property specifies how usernames and passwords are used to bind to LDAP. The following options are available: • **AUTO**: Unspecified from the Gateway side, meaning the LDAP implementation will choose. • **NONE**: Anonymous access. • **SIMPLE**: Plaintext username and passwords will be used. • **STRONG**: Usernames and passwords will be encrypted. • **SASL**: Simple Authentication and Security Layer. See the SASL Properties table below for additional SASL authentication configuration settings.
Referral	Specifies how referrals are to be processed. Possible options are: • **Follow**: Always automatically follow referrals. This is the default option. • **Ignore**: Ignores referrals.

Name	Description
	• **Throw**: Throws a ReferralException whenever a referral is encountered.
Legacy Naming Enabled	**New in 8.3.7** Determines how user and role search base entries are parsed. When checked, the 8.1 default behavior is enabled. This setting is available so that Active Directory User Sources are properly migrated from 8.1 to 8.3 Gateways. Default is true when upgrading from 8.1. Default is false when upgrading from a prior 8.3 version. It is recommended to only enable this setting when experiencing issues parsing user and role entries as valid LDAP names.

SASL Properties

These settings are utilized when Security Authentication is set to **SASL**.

Name	Description
Mechanism	An ordered list of space-separated mechanism names. The LDAP provider will use the first mechanism for which it finds an implementation. A blank value will leave this setting unspecified. Default is `DIGEST-MD5` `CRAM-MD5`.
Realm	A realm defines the namespace from which the user is selected. A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is blank.
Quality of Protection	A comma-separated list of Quality-of-Protection (QoP) values, the order of which specifies the preference order. There are three well-known values: "auth" (authentication only), "auth-int" (authentication with integrity protection), and "auth-conf" (authentication with integrity and privacy protection). A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is `auth-conf`, `auth-int`, `auth`.
Protection Strength	A comma-separated list of privacy protection strength values, the order of which specifies the preference order. The three possible strength values are "low", "medium", and "high". A blank value will leave this setting unspecified. This setting will only be used by mechanisms which support it. Default is `high`, `medium`, `low`.
Mutual Authentication	Enable or disable mutual authentication. This setting will only be used by mechanisms which support it. Default is disabled.

LDAP Attributes

A list of additional LDAP attributes to retrieve and store on the returned user.

Name	Description
Attribute	The name of the attribute to retrieve from LDAP.
Property	The property name that the attribute value will be stored under on the returned User object. If no property name is given, the attribute name will be used as the property name.
Binary	If checked, the value(s) will be treated as a byte[] array.
Multi Valued	If checked, the property will be treated as a multi-valued property and be retrievable as a list.

Fallback Cache Authentication

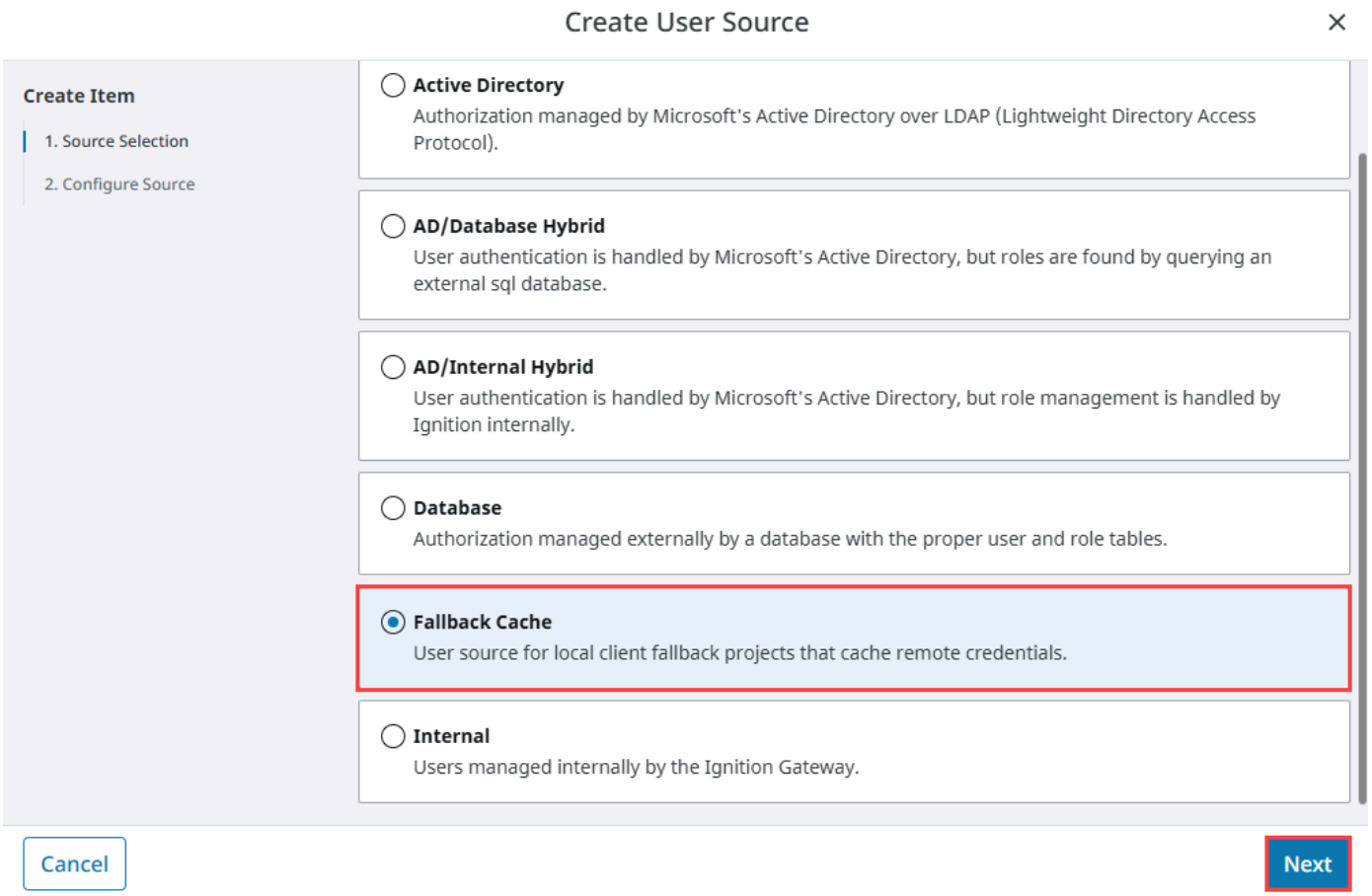
This User Source was developed specifically for a system that is using **Local Vision Client Fallback**, and allows you to cache the login credentials from a remote user source. This means your users can still log in with their normal username/password on a Local Vision Client Fallback project, even when the network connection is unavailable.

> **NOTE**
>
> Fallback Cache Authentication does not work with **Identity Provider Authentication Strategy**. This User Source will only function with Vision clients and user sources using **Classic Authentication Strategy**.

Creating the Fallback Cache User Source

The Fallback Cache User Source is created in a similar fashion to any other User Source:

1. On the **Gateway Webpage**, go to the **Platform** tab. Select **Security > User Sources**.
2. Click on **Create User Source +**.
3. Select the **Fallback Cache** option and click the **Next** button.



4. Type in a name for the new User Source and fill in the remaining properties as needed. Details on the General Properties can be found on the **User Sources** page.

5. Click **Create User Source**.

Fallback Cache Properties

Property	Description
Cache Retention	Number of days that the cache will retain recently used credentials. This property determines the number of days credentials will be stored in the cache. Once exceeded, the credentials will be removed from the cache. (Default: 15)

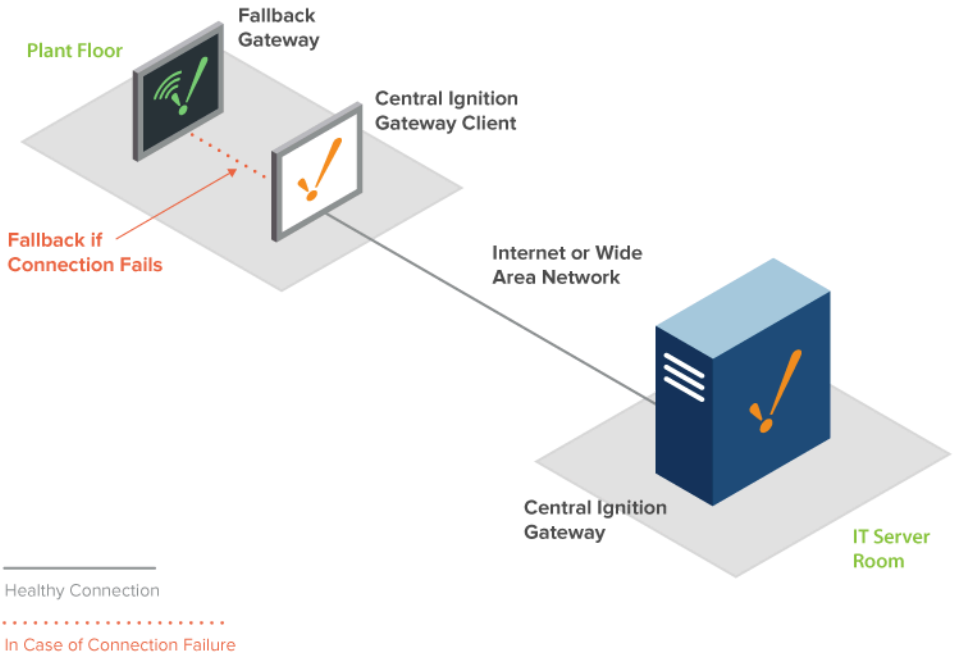
Populating the Cache

Users and Roles can not be manually added to the Fallback Cache. Instead, they are automatically copied from remote Gateways. This type of User Source is normally configured on an **Edge Gateway**, but can be utilized on an Ignition Gateway.

- Before the Fallback Cache will populate, both the **Central Ignition Gateway** and **Fallback Gateway** must be connected over the **Gateway Network**. The credentials are passed from a User Source on the **Central Ignition Gateway** to the Fallback Cache over the Gateway Network.

- A Fallback Cache User Source must exist on the **Fallback Gateway**.

- A client must be launched on the **Fallback Gateway** from the **Central Ignition Gateway**. If the user successfully authenticates against the **Central Ignition Gateway's** User Source, then the credentials are cached into the Fallback Cache.

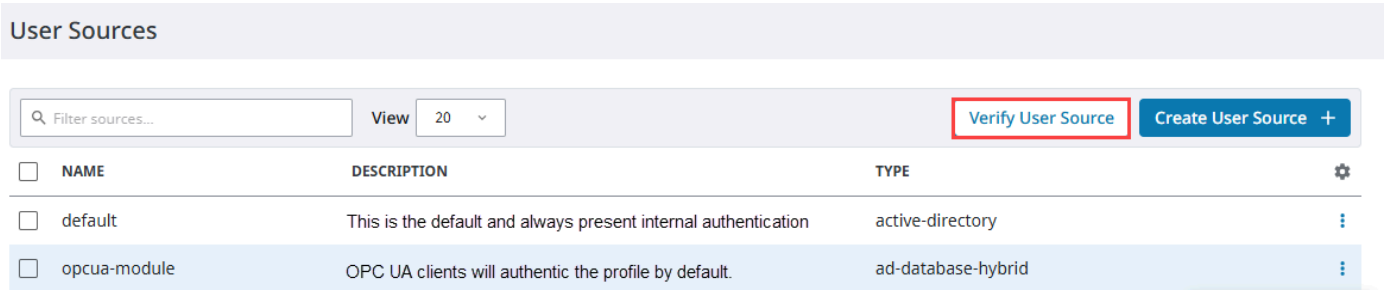


Verify a User on a User Source

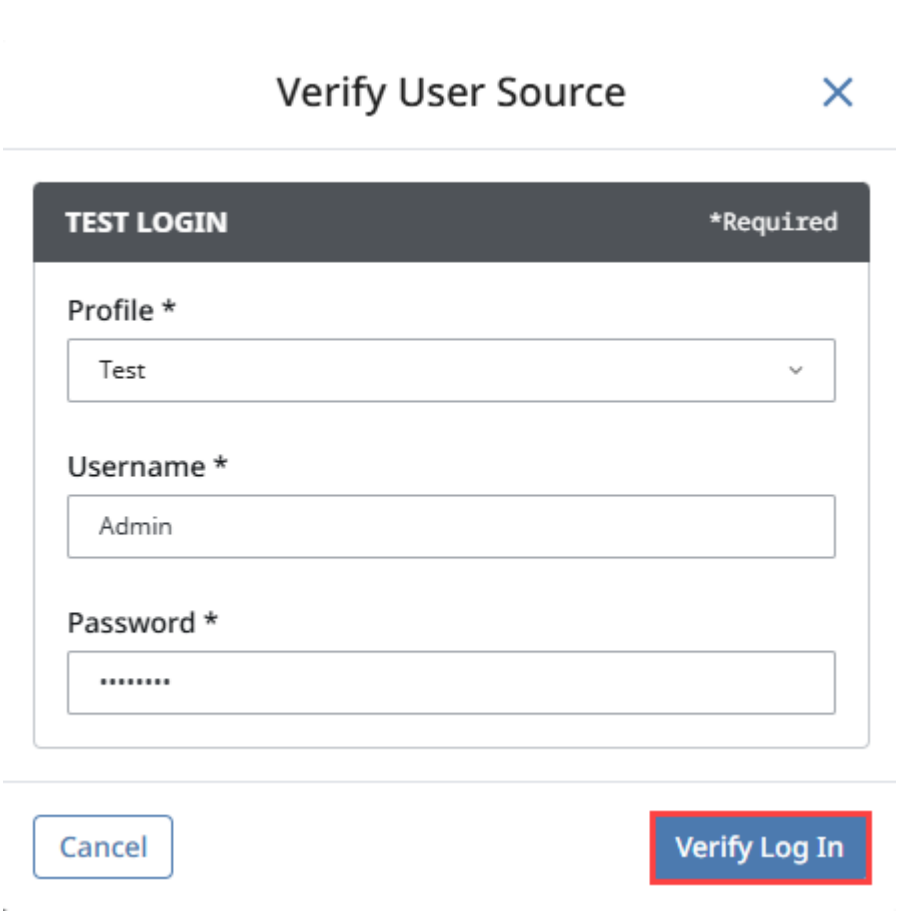
You can verify that a user exists in a given **User Source**, if the password is correct, what roles a user has, and any other information added about the user.

Verify a User

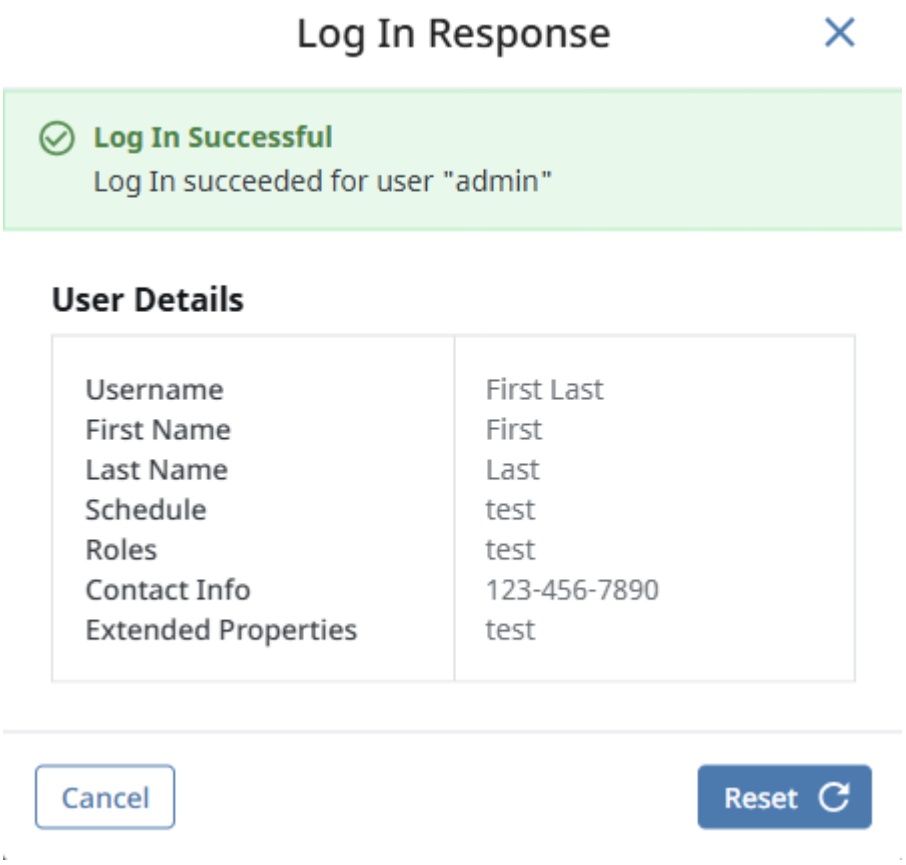
1. On the Gateway Webpage, go to the **Platform** tab.
2. Choose **Security > User Sources** from the menu on the left. The **User Sources** page is displayed.
3. Select **Verify User Source**.



4. The Verify User Source form will appear. Choose a **Profile** from the dropdown list. Enter a **Username** and **Password**. Click **Test Login**. Ignition will test the credentials then display the results of the validity test.



If the Login is incorrect, an error message will appear stating that the Login failed for a specified user. If the Login is correct, a successful message will appear for a specified user along with their information.



Identity Provider Authentication Strategy

An Identity Provider (IdP) offers a way for users to log in to Ignition using credentials stored outside of Ignition. An IdP creates, maintains, and manages identity (login) information while providing authentication services to Ignition. This provides a secure login that allows Ignition to use SSL and two-factor authentication (2FA).

> **NOTE**
>
> Launching a project from an IdP-initiated SSO response is not currently supported.

Identity Providers (IdPs) offer user authentication as a service. An IdP creates, maintains, and manages identity information for principals while providing authentication services to relying party applications within a federation or distributed network. Authentication of the user is handled by the IdP. Ignition can connect to these three different types of IdPs:

- **Internal**
- **OpenID Connect 1.0**
- **Security Assertion Markup Language 2.0**

Your organization's IT may have some sort of existing integration with an Identity Provider. Some popular Identity Providers are listed below.

- **Ping Identity**
- **Okta**
- **Active Directory Federation Services**
- **Duo**

IdPs are set up at the Gateway level. **Security Levels** are also set through the Gateway. The Security Levels enable you to define a hierarchy of access inside a Perspective Session.

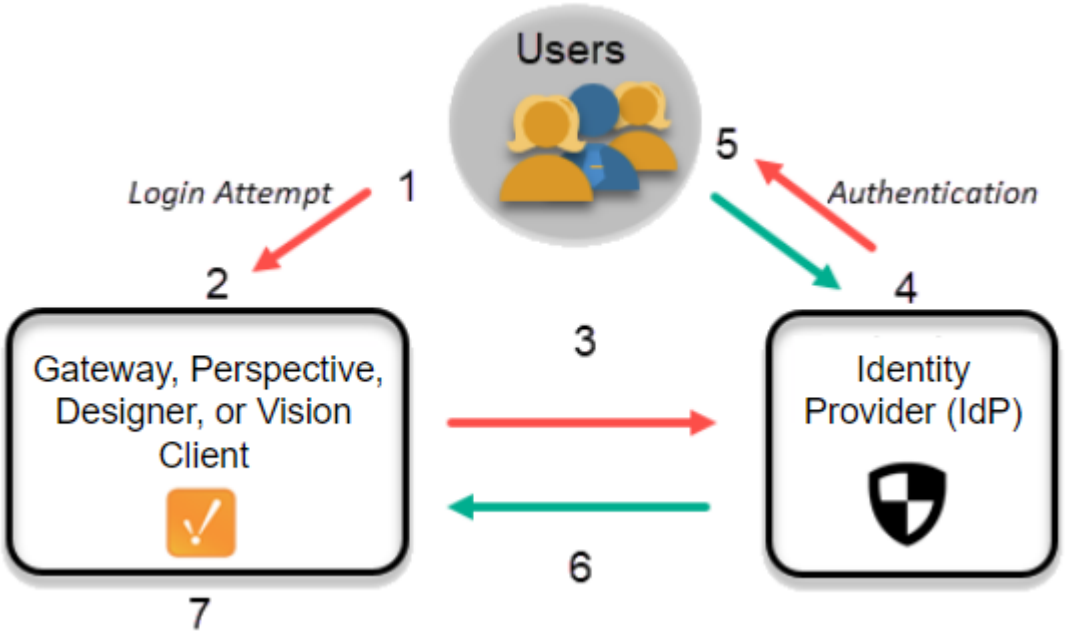
Besides Perspective, Identity Providers can also be used in the Vision module, the Designer, and on the Gateway. The Identity Provider strategy redirects the user to their IdP in their web browser in order

to authenticate. The System Identity Provider setting controls which Identity Provider the user is redirected to.

> **NOTE**
>
> If your browser is not supported, you will get an error message.

Identity Provider Authentication Workflow

The following diagram illustrates how IdP authentication works.



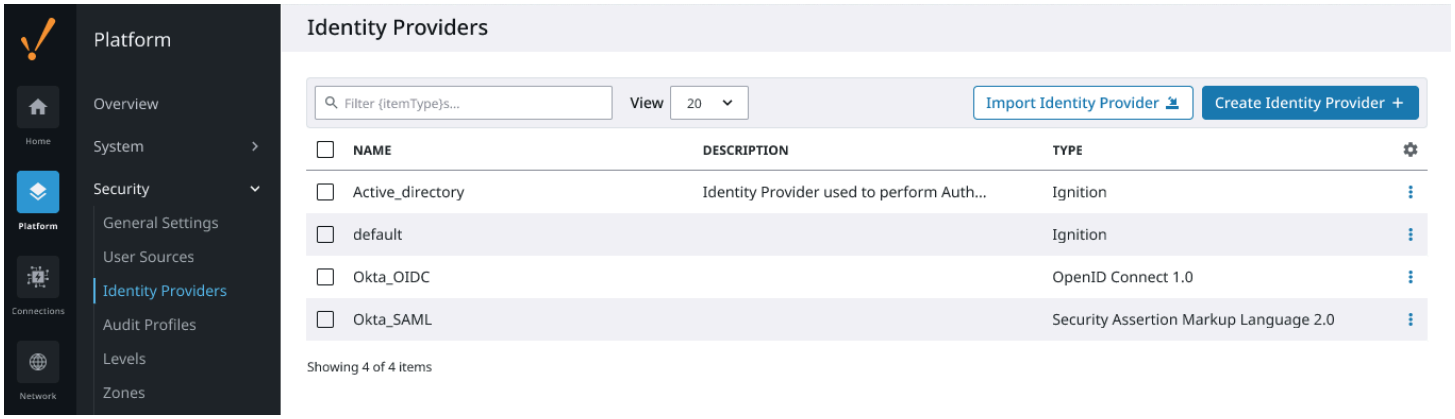
1. **User make a login attempt to the Gateway, Perspective, the Designer, or a Vision Client.**
2. **Ignition sees that IdP authentication is required.**
3. **Ignition redirects the User to Identity Provider for authentication of their credentials.**
4. **IdP Authenticates the User**: The IdP prompts the user with a security challenge, such as requesting a username and password. The extent of the challenge depends entirely on the provider, but many providers may offer support for multi-factor authentication (MFA).
5. **User Responds**: The user correctly responds to the security challenge.
6. **Redirect back to Ignition**: If the IdP successfully validates the user, it will redirect the user back to the Perspective Session. Some IdPs may have an additional workflow they will guide the user through, such as re-verifying an email address or replacing an expired password. The IdP will also return information about that user to the Session. This provides some context about the user that the Session can use to assign Security Levels.

7. **Update the User's Security Level**: Once back at the session, the user will be mapped to the specified Security Level, giving the user access to the restricted action.

Using Identity Providers

The first step in using Identity Providers is to configure them. For the steps for configuring Internal Ignition IdP, OpenID Connect 1.0, or Security Assertion Markup Language (SAML), go to **Configuring Identity Providers**.

Once an Identity Provider has been configured, there are a few things that can be done to test and adjust how it works. You can **map the attributes** that are returned in the IdP response document to more familiar user properties that are available to use within the project. You can add **rules to custom security levels** that determine when a user falls into the level. Overrides can be given to users in the form of **User Grants**, so that they are granted certain security levels regardless of the rules. Finally, you can **test out the IdP** by logging in with a user to confirm what is returned in the response document.



Auth Token Connection Recovery

After logging into the IdP, a special auth token is generated with the session on the Gateway and is saved in the Designer and Vision Client instance memory after authenticating with an IdP. If a connection is lost and later recovered, Designers and Vision Client instances may securely resume their sessions without having to completely restart by passing the Gateway a valid auth token. Note that auth tokens are not included in Gateway Backups. Any existing auth tokens are cleared when a Gateway Backup is restored.

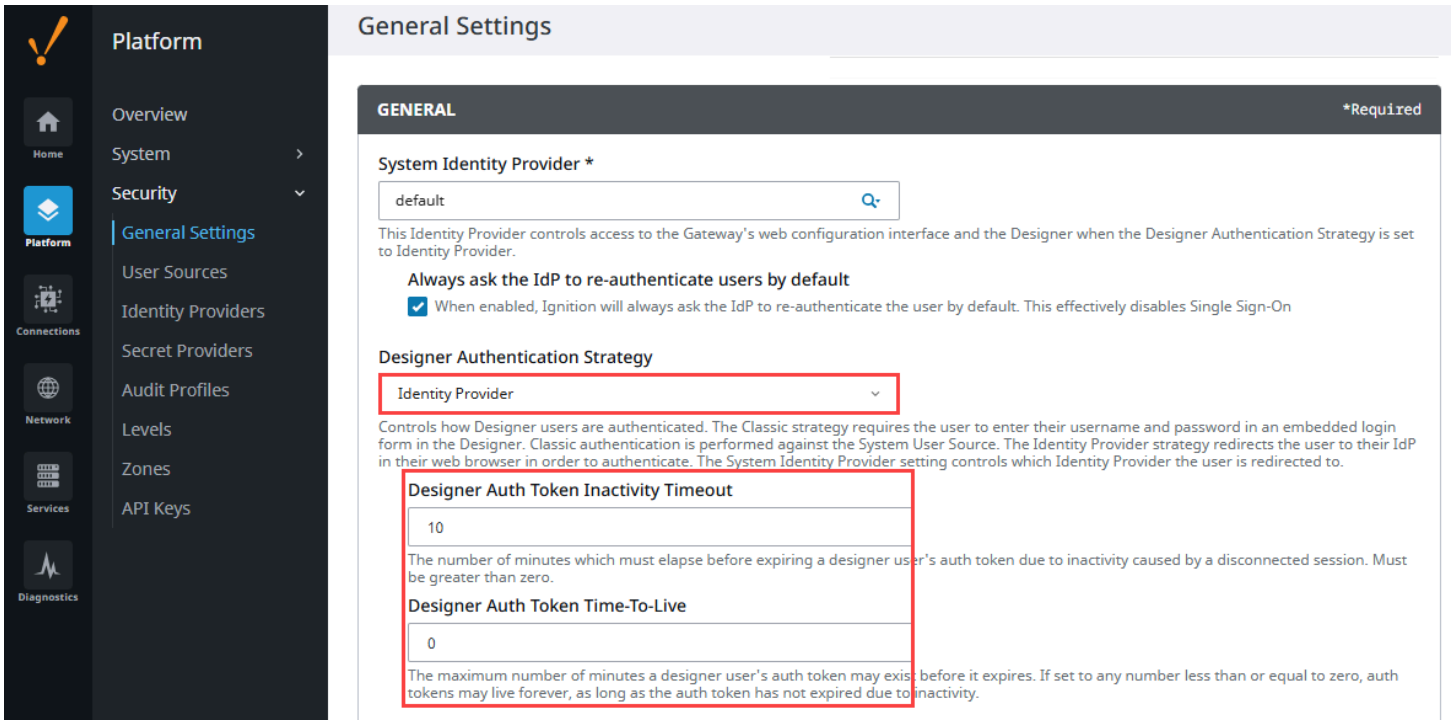
You can further configure auth tokens by adjusting settings that control the auth token lifecycle. To see these settings, make sure **Identity Provider** is selected as the Authentication Strategy as these settings do not apply to the Classic Authentication Strategy.

- User Inactivity Timeout: The number of minutes which must elapse before expiring a user's auth token due to inactivity caused by a disconnected session. Must be greater than zero. Default: 10

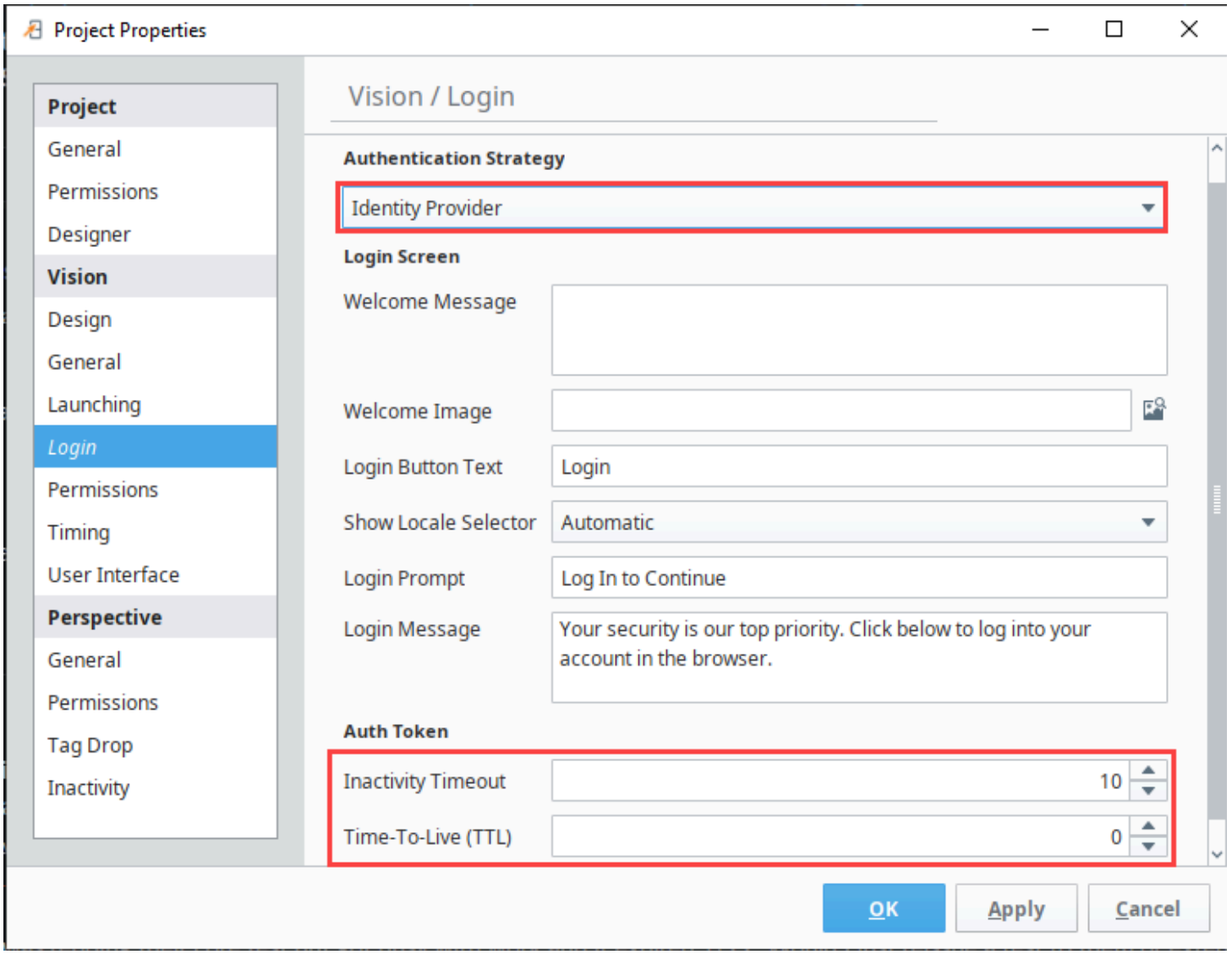
minutes.

- Time-To-Live (TTL): The maximum number of minutes a user's auth token may exist before it expires. If set to any number less than or equal to zero, auth tokens will not expire, as long as the auth token has not expired due to inactivity. Default: 0 minutes (does not expire).

For Designer Auth Tokens, these settings can be found on the Gateway **General Security Settings** page by navigating to **Platform > Security > General Settings**.



For Vision Client auth tokens, these settings can be found in the Designer by opening the **Project Menu**, selecting **Project Properties** and navigating to **Vision > Login**.



When redundancy is enabled, Vision Client auth tokens are synchronized from the Master to the Backup so that IdP-authenticated Vision Client sessions may be resumed seamlessly during failover by using an auth token.

Configuring Identity Providers

Registering the Ignition Gateway

Before configuring an Identity Provider on the Ignition Gateway, it must first be registered as an Identity Provider Client. Your Identity Provider will have a workflow to register, and it will most likely request something called a **return URL** or **redirect URI**. The paths provided utilize your Gateway's address/hostname, and they change depending on the type of provider.

> **NOTE**
>
> The same redirect URI is used for login and logout.

OpenID Connect Providers (OP)

OpenID Connect Providers
`http://yourGatewayAddress:Port/data/federate/callback/oidc`

SAML Providers

SAML Providers
`http://yourGatewayAddress:Port/data/federate/callback/saml`

Secure Integration with IdPs

You should always use the secure versions of those redirect URIs (https) in production environments. To do this you must **enable SSL/TLS** in Ignition and install a valid certificate. This is the best practice for maintaining a secure integration with third party Identity Providers.

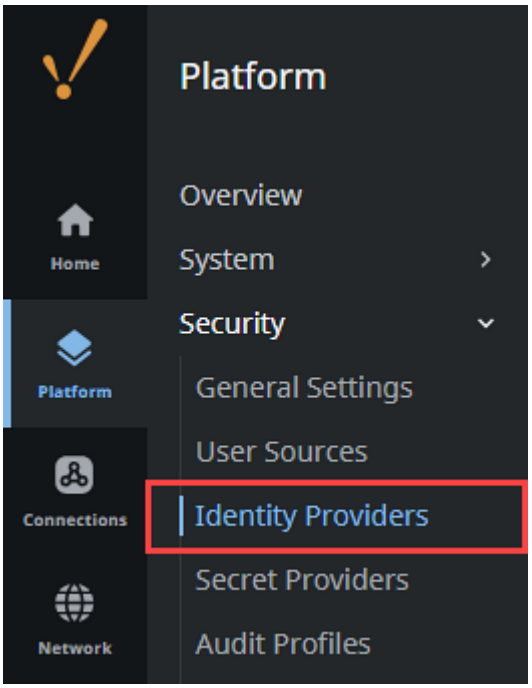
Configure an Identity Provider

Configuring Identity Providers

Watch the video

Although there are several types, the general workflow for creating an Identity Provider is the same.

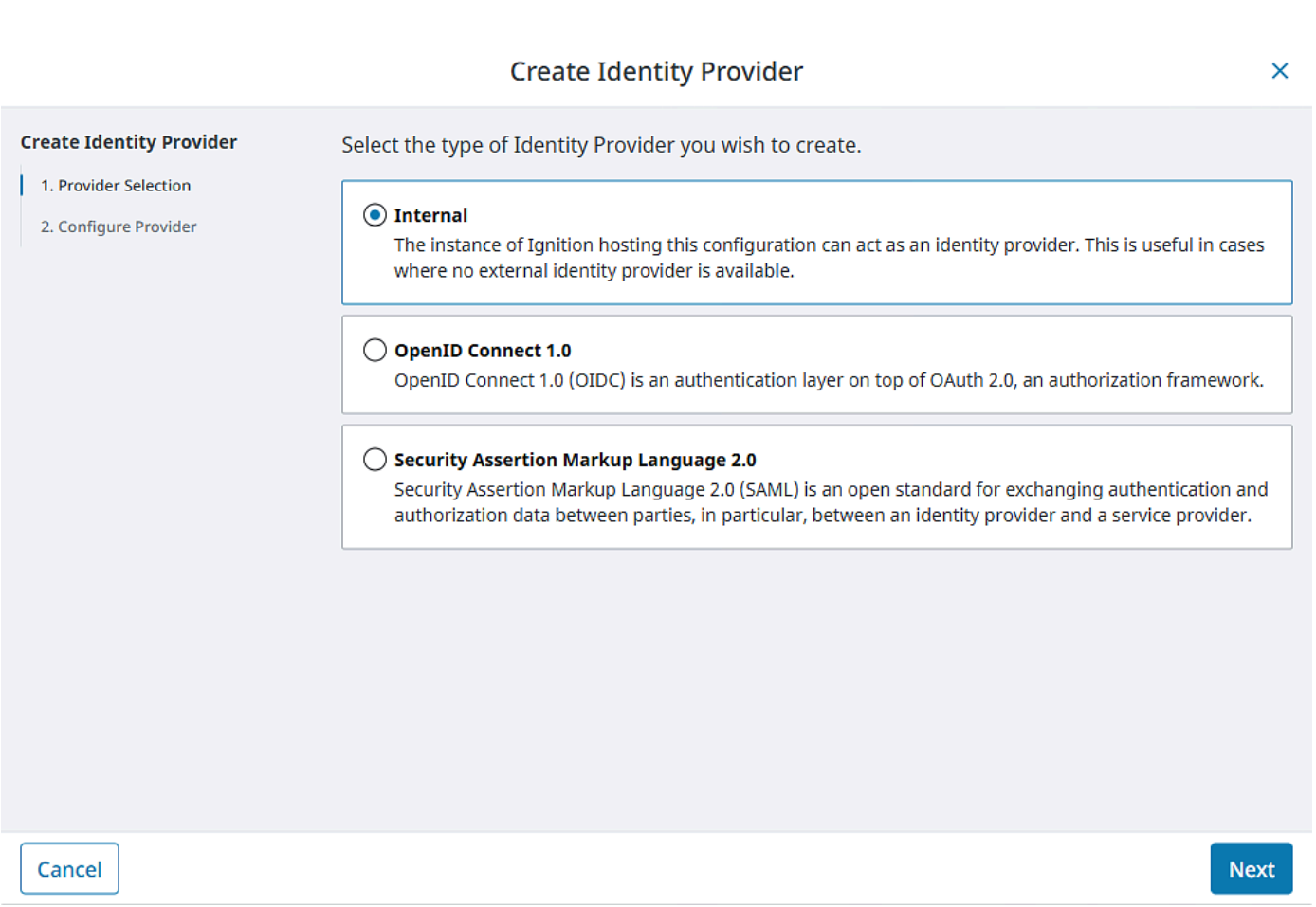
1. On the Gateway Webpage, click on the **Platform** tab.



2. Under the Security section, click on **Identity Providers**. The Identity Providers page is displayed. This page will list all IdPs that have been configured. You can filter by name or adjust the number of IdPs displayed in the view.

3. Click on **Create Identity Provider +**.

4. Choose the type of provider. The current options are **Internal, OpenID Connect 1.0** or **Security Assertion Markup Language 2.0 (SAML)**.

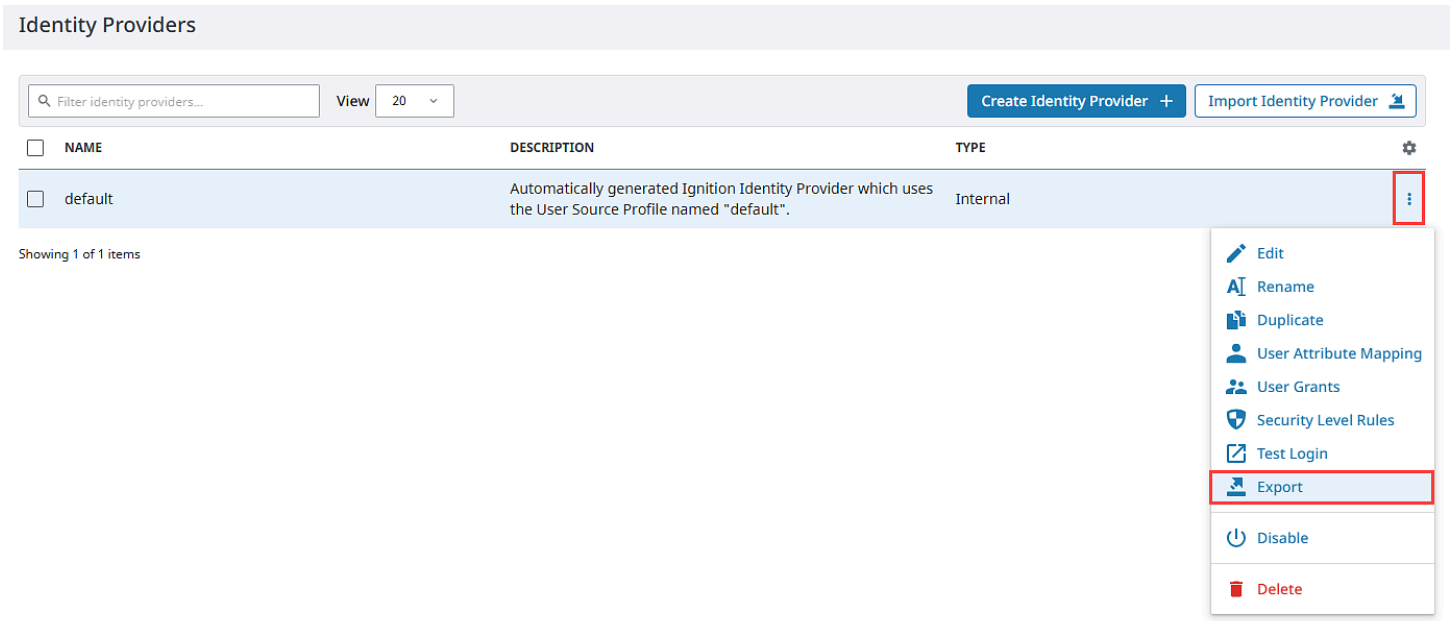


5. Click the **Next** button.
6. Configure the adapter. This step varies based on the type of provider. Please see the reference tables below for a description of properties.
7. Once you've filled in the properties, click **Create Identity Provider**.

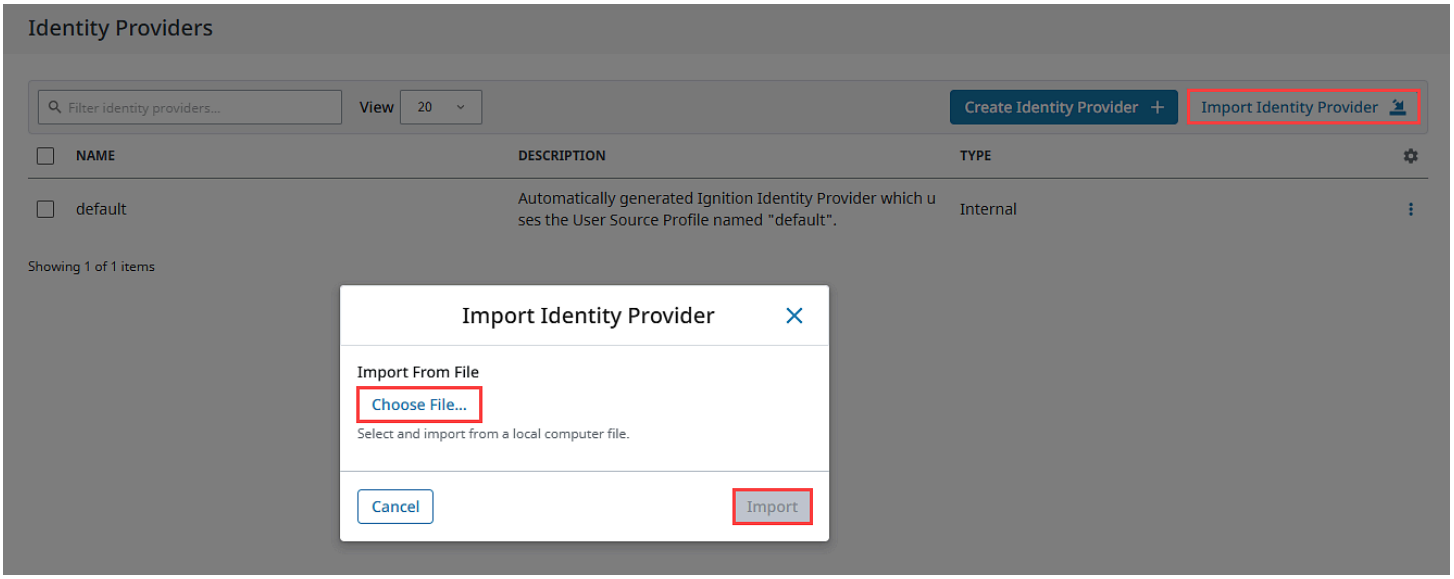
Import/Export an Identity Provider

Another option for adding an Identity Provider to your Gateway is to import one. You must first have a file of an existing configuration, which can be taken from a Gateway by expanding the three dots menu for the desired Identity Provider and selecting **Export**.

New in 8.3.3 8.1 Gateway Identity Provider exports can now be successfully imported on an 8.3 Gateway. Prior to 8.3.3, importing Identity Providers from an 8.1 Gateway was not supported.



Now, from the Identity Providers page on the Gateway you want to import on, you can select the **Import Identity Provider** option. Click **Choose File...** from the Import Identity Provider popup. Once you have your file loaded, select **Import**.



A success message will appear at the top of the screen and the Identity Provider will now be listed on the page. Note that if your import file has any missing configuration values, they will be imported as the default values.

General Properties

All Identity Provider types share the following properties:

Property Name	Description
Provider Name	The name of the adapter. Adapter names must be unique, so no two adapters on the same Gateway may have the same name. The naming conventions for IdPs are as follows: • IdP names must begin with an underscore or alpha character. • The remaining characters in the name must be either underscores or alphanumeric. • IdP names are **not** case sensitive.
Description	A description of the provider.
Enabled	Sets whether this IDP is enabled or disabled within the Gateway.
Provider Type	The type of Identity Provider. The value for this field comes from the previous screen. It cannot be changed here.

Internal Ignition Identity Provider

The Internal Ignition Identity Provider has the following properties:

Property Name	Description
User Source	The User Source for this IdP. In order to properly authenticate users, the Ignition Identity Provider must be able to query the list of users from the underlying user source profile.
Session Inactivity Timeout	The number of minutes which must elapse before expiring a session due to user inactivity. Sessions will not timeout if set to any number less than or equal to zero.
Session Expiration	The maximum number of minutes a session may exist before it is expired. Sessions will not have a max lifetime if set to any number less than or equal to zero.
Remember Me Expiration	The maximum number of hours a user will be remembered if they elect to be remembered. Remember Me is disabled when this value is set to any number less than or equal to zero. For more information on this option, see the Remember Me section below.
Authentication Methods	You can opt into Badge-based authentication for the IdP by enabling the **Badge** method. The Default Method setting determines which option users first see when attempting to authenticate against the IdP.

Badge Settings

Property Name	Description
Badge Secret	Choose whether or not the user is required to enter a secret (password) along with their badge scan. Check to require the user enter their password in addition to scanning their badge. Default is false.

Built-In Attributes

The following attributes are available in the Ignition IdP.

Attribute	Type	Description	Example

auth_time	Date	Represents the time the user last authenticated.	`// Check if it has been with`
			`// 15 minutes since the last`
			`//  authentication attempt`
			`dateDiff({idp-attributes:aut`
			`    now(), "minutes") <= 15`

challenged	Boolean	Signifies if the user provided credentials at the last login.
		
		If true, then the user was asked to re-validate their credentials the last time they attempted to login.
		
		If false, then they were not challenged to re-validate their credentials during the last login attempt. This can happen when a login request was made after a user was already authenticated. For example, if a user was already authenticated in a Perspective Session, and a separate call to **system.perspective.login** function was made with the forceAuth parameter set to false, meaning the user did not provide credentials during the last authentication challenge.
			`// Returns True or False,`
			`// depending on whether or n`
			`// provided credentials at t`
			`{idp-attributes:challenged}`

Remember Me Example

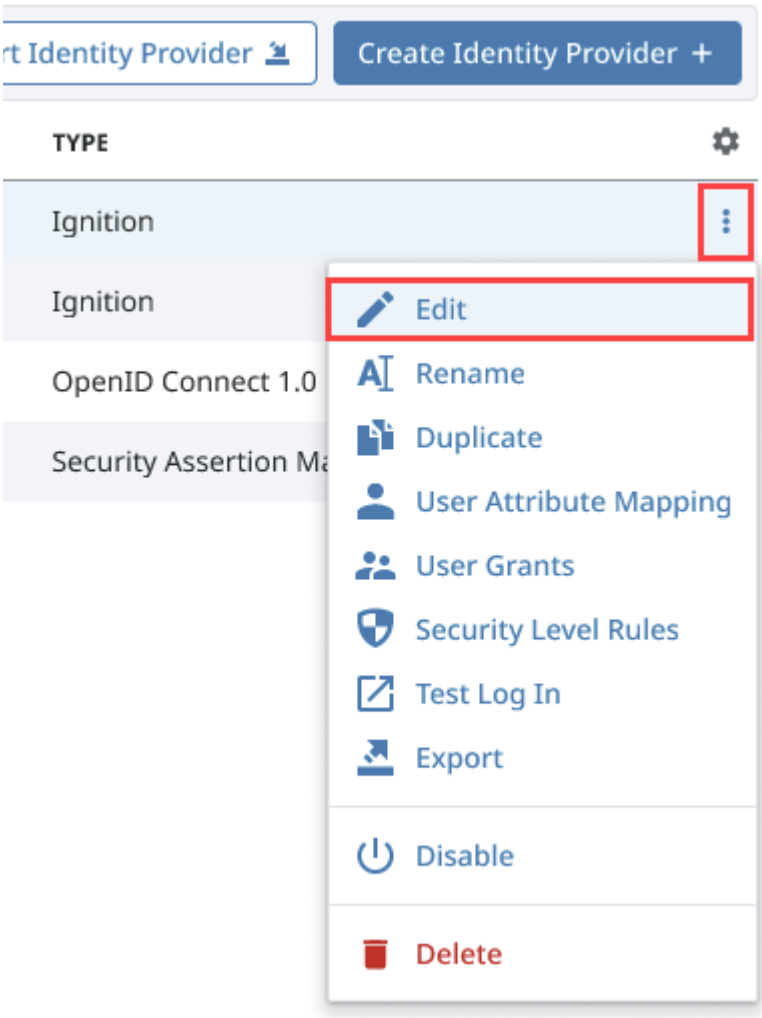
The Remember Me option allows your login to be remembered for a set amount of time, even if you close your browser or restart your Gateway. When set, you will be remembered on this device for the specified number of hours without needed to log in again.

> **NOTE**
>
> The following items will prevent this feature from working successfully:
>
> - Updating the Ignition Gateway to a new version will cause the device to "forget" the remembered user for some update versions.
> - Checking the **Always ask the IdP to re-authenticate users by default** option for the General Gateway Security Settings > System Identity Provider setting will still require re-authentication regardless of Remember Me time allotments.

> **CAUTION**
>
> This option is not recommended if you are using a public or shared device.

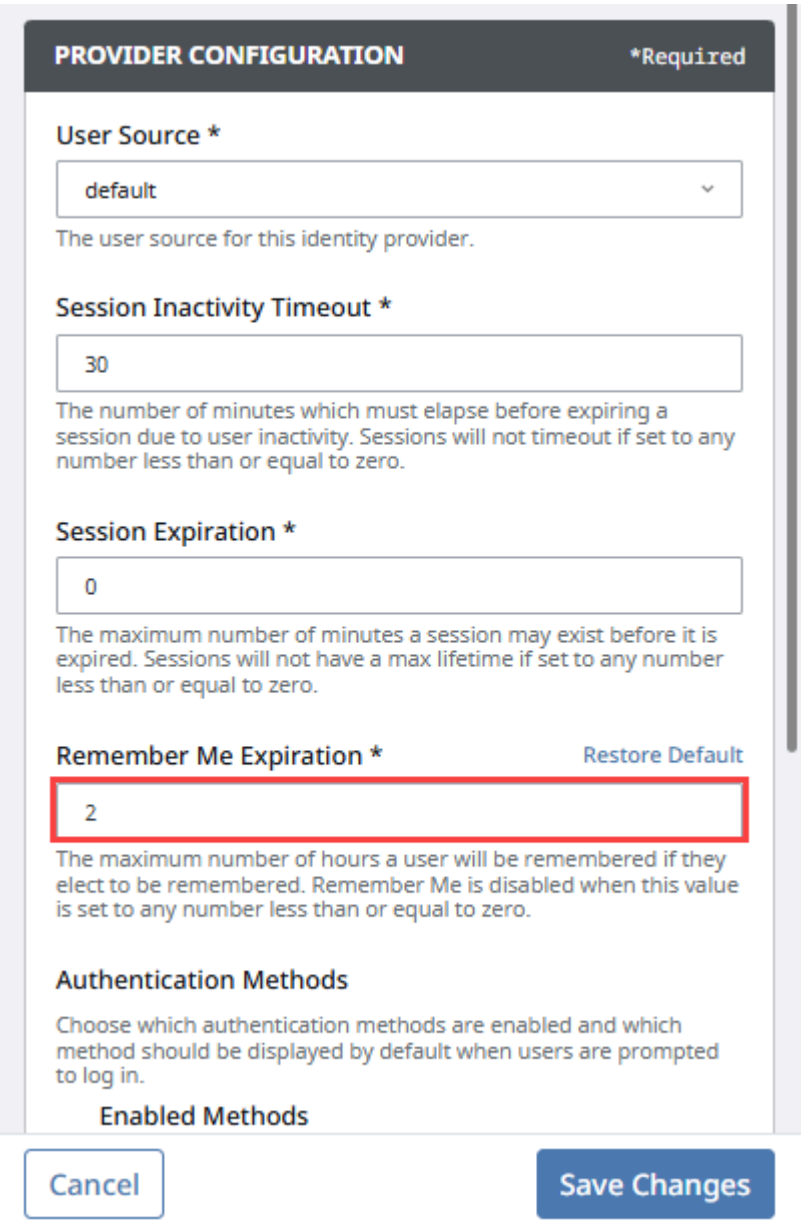
To set up Remember me, do the steps that follow:

1. On the Gateway Webpage, click on the **Platform** tab. Scroll down to **Security** > **Identity Providers**.
2. For the Ignition Identity Provider you'd like to configure, click on the three dots menu and choose **Edit**.



3. On the Edit panel, scroll down to the **Provider Configuration** section.

4. For the Remember Me Expiration option, enter a value greater than zero. For this example, we set the option to **two** hours.



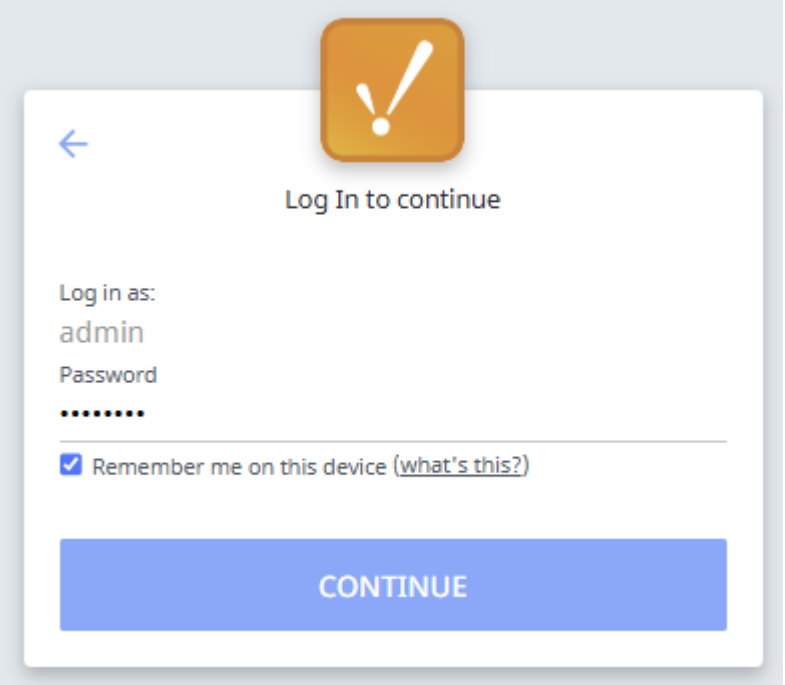
5. Click Save Changes.

To enable Remember Me for your login, do a test login:

1. On the Gateway Webpage, click on the **Platform** tab. Scroll down to **Security > Identity Providers**.

2. For the Ignition Identity Provider you'd like to configure, click on the three dots menu and choose **Test Login**.

3. Enter your password and select the Remember Me option.



4. Click the **Continue** button.

5. Your login will now be remembered for the amount of hours that were specified in the Remember Me Expiration setting, which in this example was 2 hours.

OpenID Connect Providers

OpenID Connect Providers (OP) properties are listed in the following tables. The values on many of these properties may require you to refer to information from your third party IdP. See the **OpenID Connection 1.0 Example** page for more information on creating a connection to this type.

Importing Metadata from the Provider

This method is preferred because of its ease-of-use and accuracy. After importing, you will only need to add your client ID and secret manually. (However you can revise the imported data if needed as well.)

Property Name	Description
Import from URL	URL to the OpenID Provider Configuration document. Typically, if the issuer is **"https://example.org/foo"** then the metadata URL would be **"https://example.org/foo/.well-known/openid-configuration"**
Import From File	File must be a JSON document with the properties described in section 3 (OpenID Provider Metadata) of the OpenID Connect Discovery 1.0 specification.

Configuring the Provider

Most OpenID Providers will require registering Ignition as a client. After the registration process is complete, the provider will generate a client ID and secret for Ignition, which is required below. This gives Ignition the ability to communicate securely with the provider. Most providers will also require a set of redirect URIs. An example redirect URI would look like:
`http://hostname:port/data/federate/callback/oidc`

Property Name	Description
Client ID	The client identifier registered within the Identity Provider. This value is provided the Identity Provider.
Client Secret	The client identifier registered within the Identity Provider. This value is provided by the Identity Provider.
Authorization URL	URL of the OP's OAuth 2.0 Authorization Endpoint.
Token URL	URL of the OP's OAuth 2.0 Token Endpoint.
Logout URL	Optional URL at the OP to which an RP can perform a redirect to request that the end user be logged out at the OP.
JSON Web Keys URL	URL of the OP's JSON Web Key Set document.
Use Json Web Keys URI	If checked, then identity provider public keys will be automatically downloaded from given JSON Web Keys URL. New keys will be automatically fetched when the Identity Provider generates new keys. If unchecked, then the static set of JSON Web Keys (configured below) are used, so when the Identity Provider rotates keys, they must be manually added to this configuration.
User Info URL	Optional URL to retrieve UserInfo claims from the provider. Resulting claims are typically determined by the scopes listed under the Scope setting.
User Info HTTP Request Method	The HTTP method used for sending User Info API requests to the User Info URL. Available options are POST and GET.
Authorization Response Mode	Requests the authorization server to use the form_post mechanism for returning authorization response parameters from the authorization endpoint.
Issuer	Entity that issues a set of claims.
Supported ID Token Signing Algorithm Values	A list of the JSON Web Signature (JWS) signing algorithms supported by the OP for the ID Token to encode the claims in a JWT.

Property Name	Description
Scope	A list of scopes which will be sent for each auth request to the OP. Commonly used scopes would be email and profile but check your Identity Provider's documentation for more information.
JSON Web Key Config	A list of signing key(s) the RP uses to validate signatures from the OP.

JSON Web Key Configuration

Property Name	Description
Key Type	The cryptographic algorithm family used with the key. Options are EC, RSA or oct.
Public Key Use	The intended use of the public key. Options are sig or eng.
Key Operations	The operation(s) for which the key is intended to be used.
Algorithm	The algorithm intended for use with the key.
Key ID	Used to match a specific key.
X.509 URL	A URI that refers to a resource for an X.509 public key certificate or certificate chain. The identified resource MUST provide a representation of the certificate or certificate chain that conforms to RFC 5280 in PEM-encoded form, with each certificate delimited as specified in Section 6.1 of RFC 4945.
X.509 Certificate Chain	The "x5c" (X.509 certificate chain) parameter contains a chain of one or more PKIX certificates. Each entry must be a base64-encoded (Section 4 of RFC4648 -- not base64url-encoded) DER PKIX certificate value.
X.509 Certificate SHA-1 Thumbprint	A base64url-encoded SHA-1 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate.
X.509 Certificate SHA-256 Thumbprint	A base64url-encoded SHA-256 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate.

There are some additional properties, that depend on which Key Type is selected.

Key Type: EC

Property Name	Description
crv (Curve)	The cryptographic curve used with the key.
x (X Coordinate)	The x coordinate for the Elliptic Curve point represented as the base64url encoding of the octet string representation of the coordinate.
y (Y Coordinate)	The y coordinate for the Elliptic Curve point represented as the base64url encoding of the octet string representation of the coordinate.
d (ECC Private Key)	The Elliptic Curve private key value represented as the base64url encoding of the octet string representation of the private key value.

Key Type: RSA

Property Name	Description
n (Modulus)	The modulus value for the RSA public key represented as a Base64urlUInt-encoded value.
e (Exponent)	The exponent value for the RSA public key represented as a Base64urlUInt-encoded value.
d (Private Exponent)	The private exponent value for the RSA public key represented as a Base64urlUInt-encoded value.
p (First Prime Factor)	The first prime factor represented as a Base64urlUInt-encoded value.
q (Second Prime Factor)	The second prime factor represented as a Base64urlUInt-encoded value.
dp (First Factor CRT Exponent)	The Chinese Remainder Theorem (CRT) exponent of the first factor represented as a Base64urlUInt-encoded value.
dq (Second Factor CRT Exponent)	The CRT exponent of the second factor represented as a Base64urlUInt-encoded value.
qi (First CRT Coefficient)	The CRT coefficient of the second factor represented as a Base64urlUInt-encoded value.
oth (Other Primes Info)	Information about any third and subsequent primes, should the exist. Each new Prime added will provide users with new Prime Factor, Factor CRT Exponent, and Factor CRT Coefficient properties, all of which are required.

Key Type: oct

Property Name	Description
k (Key Value)	The value of the symmetric (or other single-values) key represented as the base64url encoding of the octet sequence containing the key value.

Security Assertion Markup Language (SAML) Providers

The properties for Security Assertion Markup Language (SAML) are listed in the following tables. The values on many of these properties may require you to refer to information from your third party IdP. Refer to **SAML Example** page for how to configure an Identity Provider that is using the SAML protocol.

Importing Metadata from the Provider

This method is preferred because of its ease-of-use and accuracy. After importing, you will have the opportunity to revise the imported data if needed before saving.

Property Name	Description
Import from URL	URL to the SAML Identity Provider Metadata document.
Import From File	File must be an XML document which conforms to the SAML 2.0 metadata schema described in saml-metadata-2.0-os.

The SAML Service Provider (SP) metadata for an Ignition Gateway may be accessed at the following URL: `http://<ipaddress>:<port>/data/saml/metadata/sp`.

The Assertion Consumer Service (ACS) URL for this Ignition Gateway is: `http://<ipaddress>:<port>/data/federate/callback/saml`

Both of these addresses assume you know the IP Address and port of your Ignition install. For example, if you are on the computer Ignition is installed on, you could use: `http://localhost:8088/data/saml/metadata/sp` for the SP metadata.

SAML IdPs may send the Base64-encoded SAML Response in a line-wrapped form (with new line characters such as \r and \n separating each line). Ignition's 2.0 SP implementation can handle both line-wrapped and non-line-wrappedBase64-encoded SAML responses.

Configuring the Provider

Property Name	Description
IdP Entity ID	The Identity Provider's Entity ID.
SP Entity ID	The Service Provider's Entity ID. In this case, the Service Provider is the current Ignition Gateway. By default, the Identity Provider will automatically generate the SP Entity ID based on the hostname that the client uses to connect to this Gateway. You may opt out of this setting and input a different SP Entity ID for the Gateway.
Assertion Consumer Service (ACS) Binding	The expected binding used by the Identity Provider when interacting with Ignition's Assertion Consumer Service.
Name ID Format	The expected name ID format for subjects of assertions resulting from Authn Requests. Options are UNSPECIFIED, EMAIL_ADDRESS, X509_SUBJECT_NAME, WINDOWS_DOMAIN_QUALIFIED_NAME, KERBEROS_PRINCIPAL_NAME, ENTITY_IDENTIFIER, PERSISTENT_IDENTIFIER, TRANSIENT_IDENTIFIER.
Single Sign-On (SSO) Service URL	The Identity Provider's Single Sign-On (SSO) Service URL.
Single Sign-On (SSO) Service Binding	The binding Ignition will use for sending Authn Requests to the Identity Provider's Single Sign-On (SSO) Service.
Force Authn	Check this box to force complying Identity Providers to authenticate the user each time instead of relying on a previous security context. See section 3.4.1 of saml-core-2.0-os for more details.
Validate Response Signatures	Check this box to validate the signature of the response from the Identity Provider.
Validate Assertion Signatures	Check this box if it is expected that assertions will be signed. Ignition will validate the signatures of each assertion.

Property Name	Description
IdP Metadata URL	URL to the SAML Identity Provider Metadata document. If **Use IdP Metadata** URL is checked, the Identity Provider's signature verifying keys and certificates will be automatically downloaded from the given metadata URL. New keys and certificates will be automatically fetched when the Identity Provider generates them. If unchecked, then the static set of keys and certificates (configured below) are used, so when the Identity Provider rotates them, they must be manually added to this configuration.
Signature Verifying Keys	A list of signing key(s) that Ignition uses to validate signatures from the IdP.
Signature Verifying Certificates	A base64-encoded DER PKIX certificate value.

SAML Signature Verifying Key Configuration

Property Name	Description
Key Algorithm	The algorithm identifier for this signature verifying key. Options are DSA, RSA, or EC.
Key Value	A base64-encoded DER key value.

Import Provider Metadata for Redundant Backup

See the **Import Provider Metadata for Redundant Backup section** below.

Import Provider Metadata for Redundant Backup

Both OIDC and SAML provider configurations contain a section that allows you to define metadata which will be used by the backup Gateway in a redundant Ignition configuration. When a backup Gateway becomes the active node, it will interact with the Identity Provider via the backup settings. By default, the backup meta data will use the same configuration settings that the master node uses,

unless "Provider Metadata for Redundant Backup" is set to false, in which case the IdP configuration will allow you to define settings for the backup Gateway.

Troubleshooting

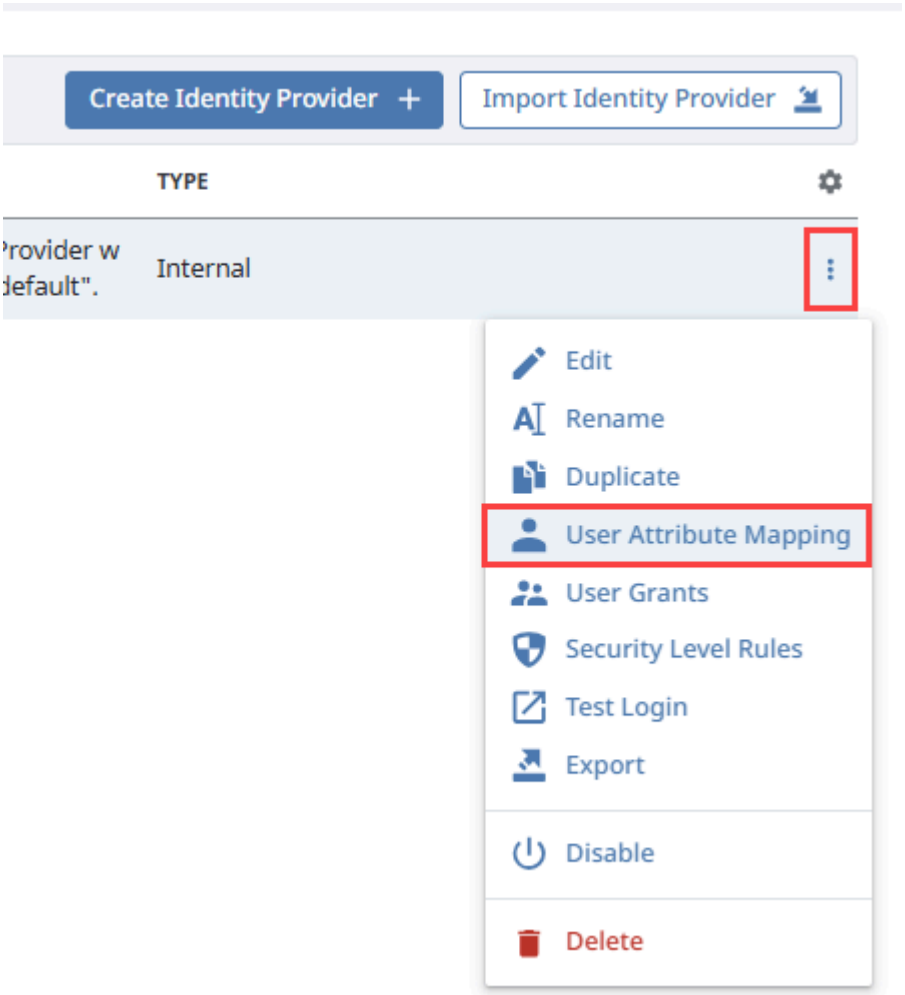
Go to **Troubleshooting Identity Providers** for helpful examples to help you diagnose and troubleshoot issues with configuring IdPs.

User Attribute Mapping

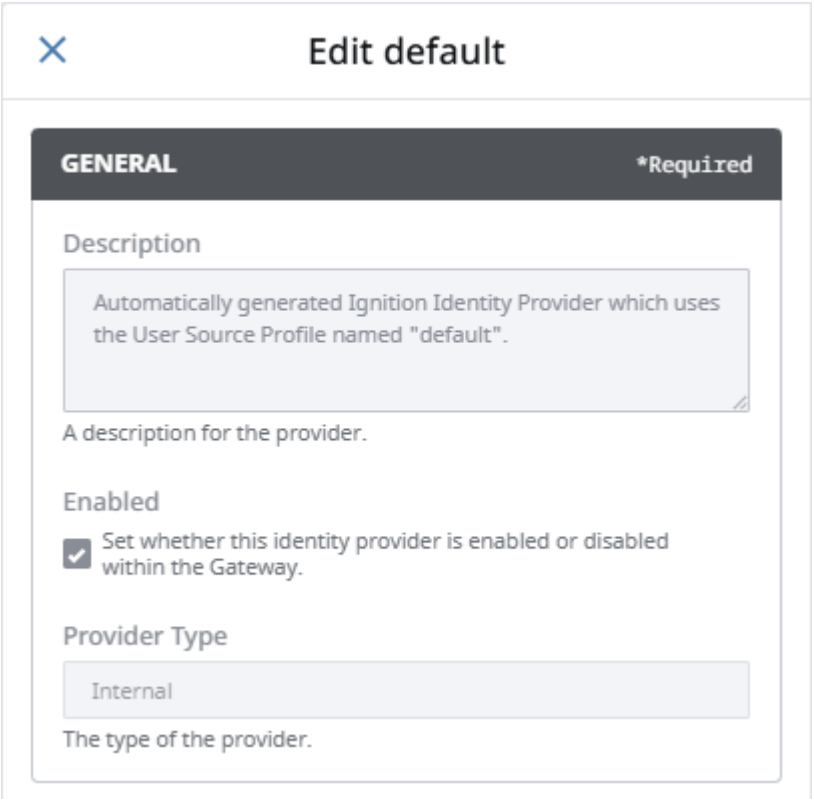
The User Attribute Mapping page allows you to map information in the Identity Providers (IdP) response document to easily understandable properties. These properties are then made available as **Session Properties** in the Perspective Session. To work, this requires that the Gateway already has a valid IdP configuration which returns a response document when attempting to login.

Configuring a User Attribute Mapping

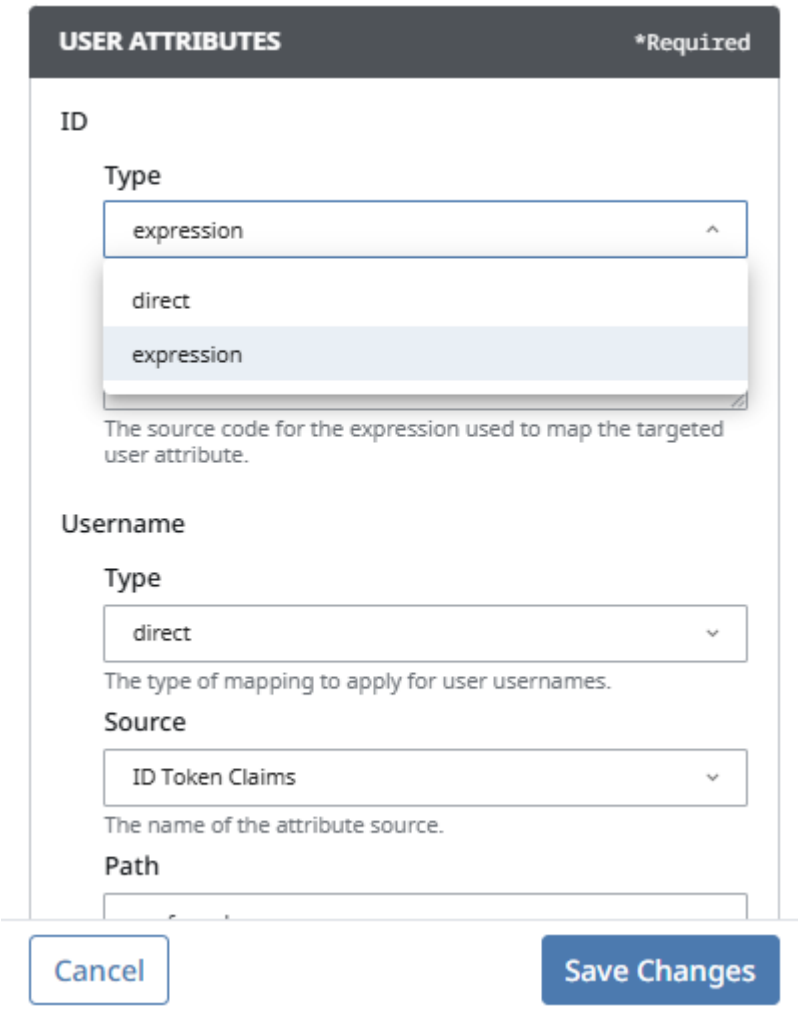
1. From the Gateway Webpage **Platform** tab, click on **Security > Identity Providers**.
2. Select the **Identity Provider** and click on **User Attribute Mapping** under the three dots menu.



3. The Name, Description, and Provider Type are not editable here, but are listed on the page to make clear which IdP the User Attribute Mapping is being configured on.



4. Under the **User Attributes** section of properties, you'll find settings for each of the mappable user attributes.



5. If you make changes to any of the mappings, be sure to click **Save Changes** when finished.

User Attribute Mapping Property Reference Table

Mappings can be configured on several user attributes.

Attribute	Description	Data Type
ID	The Type of mapping to apply for user IDs.	String
Username	The Type of mapping to apply for usernames.	String
First Name	The type of mapping to apply for first names.	String
Last Name	The type of mapping to apply for last names.	String
Email	The type of mapping to apply for email addresses.	String

Roles	The type of mapping to apply for user roles.
	
	Note: While using an Expression type attribute mapping, you can use a **runScript** call to create a list of roles through scripting. However the script must return an ArrayList, as opposed to Jython collection types:
	
	`import java.util.ArrayList as ArrayList`
	`roles = ArrayList()`
	`roles.add("Operator")`
		Java ArrayList of Strings

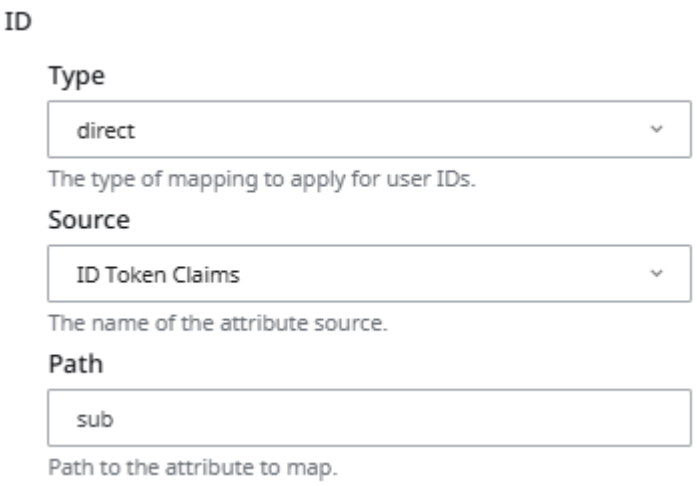
Each attribute mapping has similar properties, which are listed below

Property Name	Description
Type	The Type of mapping to apply for the attribute. Options are **direct** or **expression**.
Source	The name of the attribute source. In cases where the identity provider has several **Attribute Sources**, this property allows you to specify which source the mapping should use. (for **direct** type only)
Path	Path to the attribute map based on the selected source. Each node in the path is delimited by a slash character ("/"). (for **direct** type only)
Expression	The source code for the expression used to map the targeted user attribute. See the **Security Level Rules**. (for **expression** type only)

Direct Mapping

Direct mappings require that you enter in the path to an attribute in the response document for the given property. Values in the Identity Provider's response document are dereferenced using the configured paths and are copied to the respective user properties in Ignition. The Source dropdown allows a particular mapping to target a specific object in the response document. The Path property then determines what object inside of source should be.

In the image below, the ID of the user, as represented on the Ignition Gateway, will be determined by the **sub** attribute, located in the **ID Token Claims** source in the response document.



Attribute Sources

Each Identity Provider has a different list of possible attribute sources. Below are several tables that list each possible source, as well as the expression path name for the source (used by `{attribute-source:X:Y}` and `{multi-attribute-source:X:Y}` expression paths. See **Security Level Rules**.)

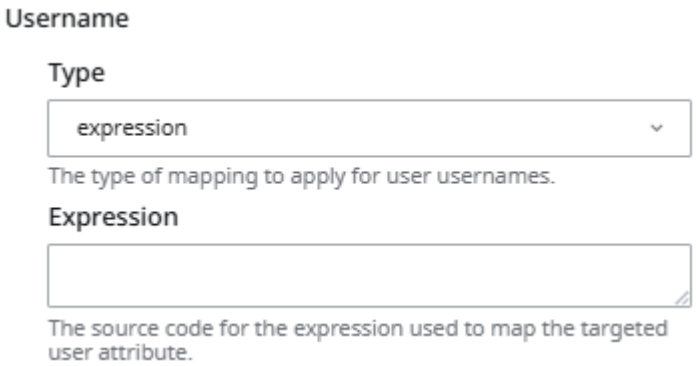
IdP Type	Attribute Source(s)
Ignition	ID Token Claims (default)
Ignition	Token Endpoint Response
OIDC	ID Token Claims (default)
OIDC	Token Endpoint Response
OIDC	User Info Claims
SAML	Authentication Response

For Ignition and OIDC IdPs, the correct source to use will vary with the referenced attribute, and is included in the response document on the **Test Login** page.

> **NOTE**
>
> The user info claims attribute source is present only when the user info URL is configured and Ignition receives a valid user info response from the IdP.

Expression Mapping

The Expression type allows you to use the **expression language** to derive the mapping from contextual data, such as the IdP response document or tags. To assist with parsing the response document in an expression, there are several built-in objects.



Built-in Expression Objects

In IdP contexts, it is possible to reference IdP response document elements using a three-part format:

```
{<Attribute Type>:<Attribute Source>:<Attribute Path>}
```

In this format, `"<Attribute Type>"` is the attribute type we're attempting to reference, `"<Attribute Source>"` is the name of the top-level attribute source, and `"<Attribute Path>"` is a path to the property inside the attribute source. Here, we'll discuss the possible values for each part of this expression, and when they should be used.

Attribute Type

Type	Expression Value	Description
Single-Valued Object	`attribute-source`	Provides the ability to reference an attribute (such as the username) returned from an IdP attribute source where the attribute's value is expected to be a single atomic value, usually a string.
Multi-Valued Object	`multi-attribute-source`	Provides the ability to reference an attribute (such as the user's roles) returned from an IdP attribute source where the attribute's value is expected to be a collection of single atomic values, usually a collection of strings.

Attribute Source

Possible values for the attribute source will vary by IdP type:

IdP Type	Attribute Source(s)	Expression Path Name
Ignition	ID Token Claims (default)	`idTokenClaims`
Ignition	Token Endpoint Response	`tokenEndpointResponse`
OIDC	ID Token Claims (default)	`idTokenClaims`
OIDC	Token Endpoint Response	`tokenEndpointResponse`
OIDC	User Info Claims	`userInfo`
SAML	Authentication Response	`authnResponse`

For Ignition and OIDC IdPs, the correct source to use will vary with the referenced attribute, and is included in the response document on the **Test Login** page.

The authnResponse must be explicitly stated when attempting to access the Authentication Response source with expression paths. See **Expression Mapping**.

Attribute Path

The final element in an attribute reference is a path to the attribute within the attribute source. Here, the **Test Login** page is helpful for accessing the attribute source's structure in the response document.

If the attribute source is a JSON document (as is the case with Ignition and OIDC IdPs), then a JSON Path may be used to reference the target attribute.

If the attribute source is an XML document (as is the case with SAML IdPs), then an **XPath 1.0 expression** may be used to reference the target attribute. See the **Multi-Valued Object Example** further down.

Single-Valued Object Example

This example makes use of the Single-Valued Object on an JSON response document from an OpenID Connect IdP. The document used by this example is listed in the code block below.

▼ Sample OIDC Response Document

```
{
  "idTokenClaims": {
    "sub": "00u4zl66kjWSxpKyH357",
    "name": "Paul Person",
    "email": "person@company.com",
    "ver": 1,
    "iss": "https://573779.okta.com",
    "aud": "0oa2uqi4odPcNHRpS357",
    "iat": 1617831866,
    "exp": 1617835466,
    "jti": "ID.itbv0TrKsyxqJpEsqwkVHaVa07xBD_haRQMnGUQ5tz4",
    "amr": [
      "pwd"
    ],
    "idp": "00ob8o86nJ2kN48V4356",
    "nonce": "a2Zq9QA1xL7t_apS58YIPaegbGD9w6FN60qSgP0i7fg",
    "preferred_username": "person@company.com",
    "auth_time": 1617831865,
    "at_hash": "TEbuSUf8aMhWGYDuHsC0jQ"
  },
  "tokenEndpointResponse": {
    "token_type": "Bearer",
    "expires_in": 3600,
    "access_token": "eyJraWQiOiJCVlowd3JweUdiRnB5cG5EQ05HbUpnMnhtbW5sc0h3TlBRdVp3
4eioSzs0CrVLk3j09vQIEKn0jdlKqY0i9_9HBiEGUrVOXuiJ587ocRTjvxeNCpr_B7DBczBbtTkOFNzw3
    "scope": "openid email profile",
    "id_token":
"eyJraWQiOiJUNEhmZDQ0LXVVQ09sMGJpTl9fWnprQkJYWVdSSXlQYXBIZi1QeFRFTHYwIiwiYWxnIjoi
55PyBX2kQ-ggyqXVrWlywmOWFNn4kSjViuzMIt2gXeW8yW-OBtxBTZr7QPDF6XKwZk6V183K0hT3ZbrzH
  },
  "userInfo": {
    "sub": "00u4zl66kjWSxpKyH357",
    "name": "Paul Person",
    "locale": "en-US",
    "email": "person@company.com",
    "preferred_username": "person@company.com",
    "given_name": "Paul",
    "family_name": "Person",
    "zoneinfo": "America/Los_Angeles",
    "updated_at": 1596495564,
    "email_verified": true
  }
}
```

If, for example, we wanted to configure a mapping on the First Name, which would allow us to retrieve the name of the user provided by the response document. The **given_name** under **userInfo** appears to have the value we're looking for.

We can use the attribute-source object to directly reference the node.

```
{attribute-source:userInfo:given_name}
```

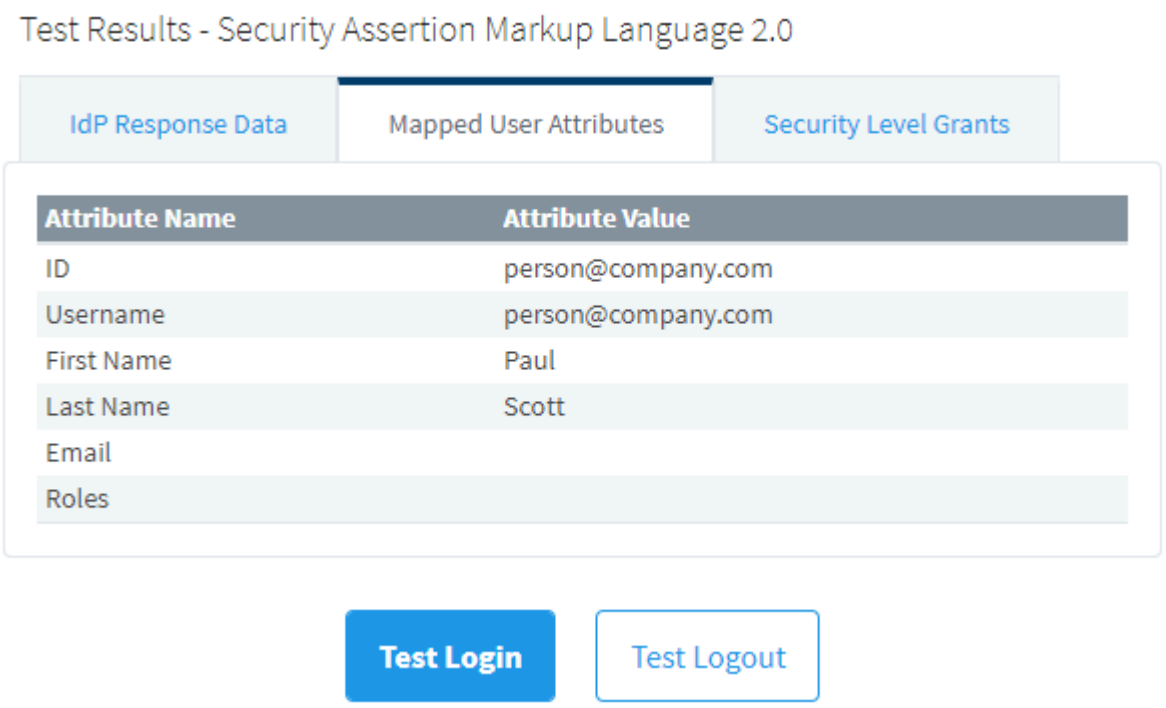
This is equivalent to using a direct mapping type with Source set to "ID Token Claims" and Path set to "name". However, since we're using the expression language, we have access to more tools, which is useful in cases where the attribute value we're looking for isn't as readily accessible.

For example, say the response document did not contain a node that represented just the users first name. However, it does contain a name under idTokenClaims (line 4) to that has the user's full name. We could access the full name, and then use the expression language's **split** function to split the first and last names.

```
split({attribute-source:idTokenClaims:name}, ' ')[0,0]
```

Multi-Valued Object Example

This example makes use of the Multi-Valued Object on an XML response document from a SAML IdP. Here we can use an **XPath expression** to examine the document and reference a specific node. We'll configure an expression type mapping that will retrieve attribute values that we want mapped to a user's Roles. Below we see the results of a **test login**. Our user does not appear to have any roles mapped.



If we click on the **IdP Response Data** tab, we see the results listed in the code block below.

▾ Sample SAML Response Document

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<saml2p:Response xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:xs="ht
Destination="http://localhost:8088/data/federate/callback/saml" ID="id5084171520
tiacipjeAZM4e4xHRRgFkV9KrOx5Mv8tHrdowuL7zi0" IssueInstant="2021-04-06T21:42:33.77
    <saml2:Issuer xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion" Format="urn
format:entity">http://www.okta.com/exkj0b0vtttYvPzH10h7</saml2:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-e
            <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more
            <ds:Reference URI="#id5084171520554334776221 24">
                <ds:Transforms>
                    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#en
                    <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c1
                        <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/
                    </ds:Transform>
                </ds:Transforms>
                <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha2
                <ds:DigestValue>UYW4pdcsWK19aRbSl3NlgRc+RZ1jIyCK6hsXnSu7VYs=</ds:
            </ds:Reference>
        </ds:SignedInfo>
    </ds:Signature>
    <saml2p:Status>
        <saml2p:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
    </saml2p:Status>
    <saml2:Assertion xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion" ID="id50
IssueInstant="2021-04-06T21:42:33.776Z" Version="2.0">
        <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:entity">http://www.okta.com/exkj0b0vtttYvPzH10h7</saml2:Issuer>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/x
                <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-
                <ds:Reference URI="#id5084171521655562565869 17">
                    <ds:Transforms>
                        <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsi
                        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-ex
                            <ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2
                        </ds:Transform>
                    </ds:Transforms>
                    <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#
                    <ds:DigestValue>xNlFDTmRTPXYqSLpruOZoPSq2Lz6q1rPrGiROR6MXdM=<
                </ds:Reference>
            </ds:SignedInfo>
        </ds:Signature>
        <saml2:Subject>
            <saml2:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-
```

```
format:unspecified">person@company.com</saml2:NameID>
            <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bea
                <saml2:SubjectConfirmationData InResponseTo="A-tiacipjeAZM4e4xHRF
NotOnOrAfter="2021-04-06T21:47:33.776Z" Recipient="http://localhost:8088/data/fec
            </saml2:SubjectConfirmation>
        </saml2:Subject>
        <saml2:Conditions NotBefore="2021-04-06T21:37:33.776Z" NotOnOrAfter="2021
            <saml2:AudienceRestriction>
                <saml2:Audience>http://localhost:8088</saml2:Audience>
            </saml2:AudienceRestriction>
        </saml2:Conditions>
        <saml2:AuthnStatement AuthnInstant="2021-04-06T21:26:07.499Z" SessionInde
tiacipjeAZM4e4xHRRgFkV9KrOx5Mv8tHrdowuL7zi0">
            <saml2:AuthnContext>

<saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtec
            </saml2:AuthnContext>
        </saml2:AuthnStatement>
        <saml2:AttributeStatement>
            <saml2:Attribute Name="FirstName" NameFormat="urn:oasis:names:tc:SAML
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
xsi:type="xs:string">Paul</saml2:AttributeValue>
            </saml2:Attribute>
            <saml2:Attribute Name="LastName" NameFormat="urn:oasis:names:tc:SAML:
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
xsi:type="xs:string">Scott</saml2:AttributeValue>
            </saml2:Attribute>
            <saml2:Attribute Name="Email" NameFormat="urn:oasis:names:tc:SAML:2.0
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
xsi:type="xs:string">person@company.com</saml2:AttributeValue>
            </saml2:Attribute>
            <saml2:Attribute Name="Role" NameFormat="urn:oasis:names:tc:SAML:2.0:
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
xsi:type="xs:string"/>Operator</saml2:AttributeValue>
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
xsi:type="xs:string"/>Supervisor</saml2:AttributeValue>
            </saml2:Attribute>
            <saml2:Attribute Name="Groups" NameFormat="urn:oasis:names:tc:SAML:2.
                <saml2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema
            </saml2:Attribute>
        </saml2:AttributeStatement>
    </saml2:Assertion>
</saml2p:Response>
```

In the example document, lines 68 and 69 contain saml2:AttribueValue elements, which contain the roles for our user. Starting from the roles, we can work our way back towards the top of the document,

but making the following observations.

- We see both elements are under a `saml2:Attribute` element (line 67).
- The `saml2:Attribute` element is under a `saml2:AttributeStatement` element (line 57).
- The `saml2:AttribueStatement` element is under a `saml2:Assertion` element (line 23).
- Finally, the `saml2:Assertion` element is under the `saml2p:Response element`, which seems to be the root for the entire path (line 2).

Thus, we're looking for all text values of all saml2:AttributeValue elements that are under a specific saml2:Attribute element. We see that the saml2SAttribute element we're looking for also has name attribute set to "Role", which we can use to help focus our path. Based on our observations, the path looks like the following:

Note that we start at the root of the saml2p:Reponse element (on the left), and build the rest of the path in reverse order of our observations.

We specified that the saml2:Attribute must contain an attribute by the name of "Name", with a value of "Role", with which filters out the other saml2:Attribute elements in the document (such as the elements with FirstName and LastName).

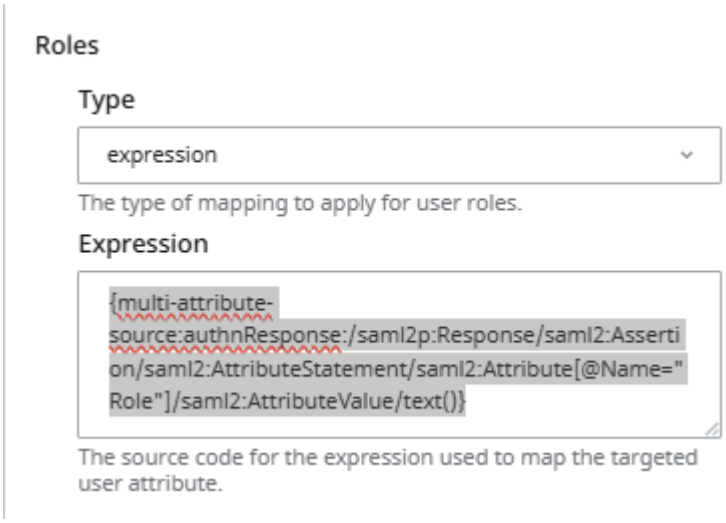
In addition, we ended the path with text(), which access the text values of the nodes.

```
/saml2p:Response/saml2:Assertion/saml2:AttributeStatement/saml2:Attribute[@Name="Role
```

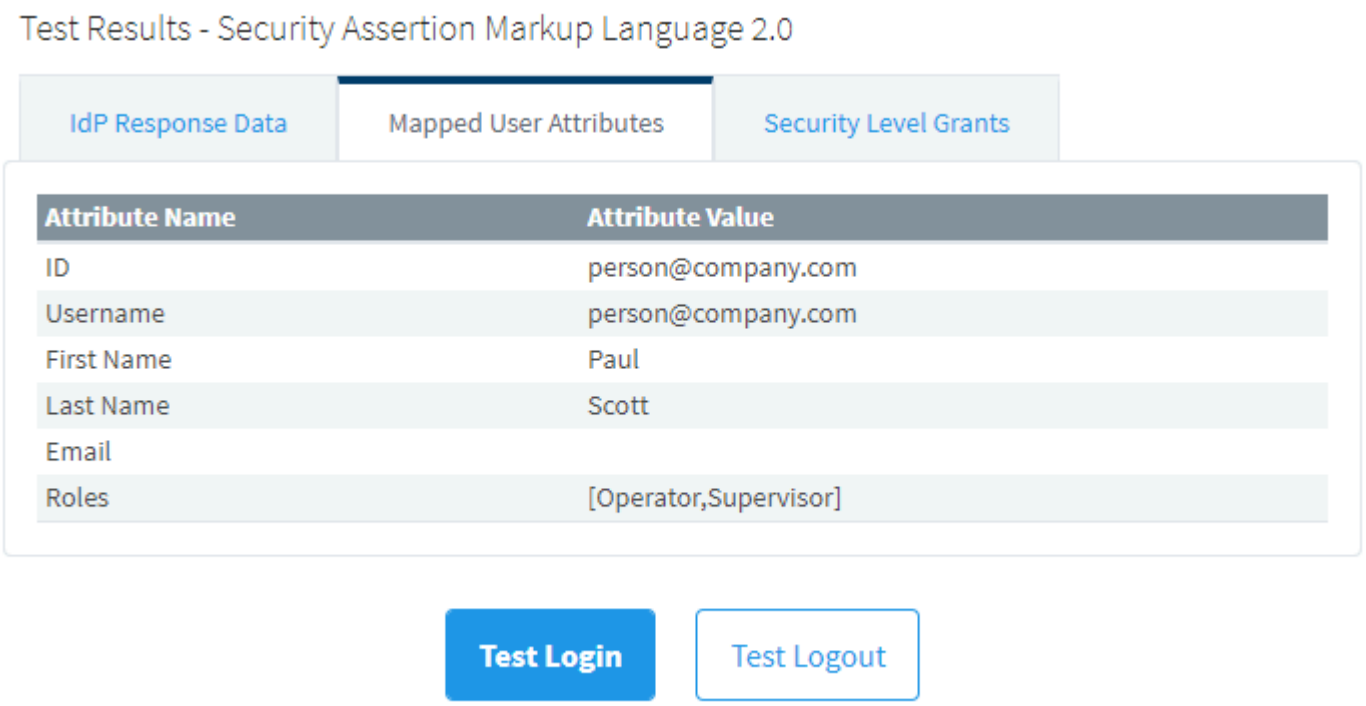
Writing the Path

Now that we know what our path looks like, we can write an expression. We'll type the following into the "Expression" field for the Roles setting:

```
{multi-attribute-
source:authnResponse:/saml2p:Response/saml2:Assertion/saml2:AttributeStatement/saml2:
```



If we then **Save Changes**, we can attempt a **test login**, and check the mapped user attributes, where we see our roles from the response document now appear mapped to our user.



Tag Path Example

In addition, tag values can be referenced by mappings. Note that the values are only checked when the user logs in. So if the value of any tags used in a mapping change after the user logs in, the result of the mapping won't retroactively change.

```
{[default]A_Folder/A_Tag.value}
```

In the example above, the expression would attempt to return the value of the tag at path "A_Folder/A_Tag", in the Tag Provider named "default".

OpenID Connect 1.0 Example

This section provides an example of how to connect an Identity Provider that is using the OpenID Connect 1.0 protocol. This example uses the Okta IdP service. Your IdP vendor may differ and the specific links will differ.

Prerequisites

Before you begin configuring Ignition there are some preliminary requirements that need to be done outside of Ignition:

- A configured remote IdP (Okta in this example)
- The metadata file specific to your IdP
- The scope data specific to your IdP
- Login credentials to use as a test

Configured IdP

An IT department is usually the one to set up and configure a remote IdP. You need a configured remote IdP that is compatible with OpenID Connect 1.0. protocol. At minimum there needs to be an account set up with the IdP, users added to the IdP account, and applications added to the IdP.

Metadata File

You will need the metadata file specific to your IdP. This document defines how to communicate with the IdP. It is usually a web page that allows the metadata file to be exported to a JSON file. Often it is a URL that ends with ".well-known/openid-configuration".

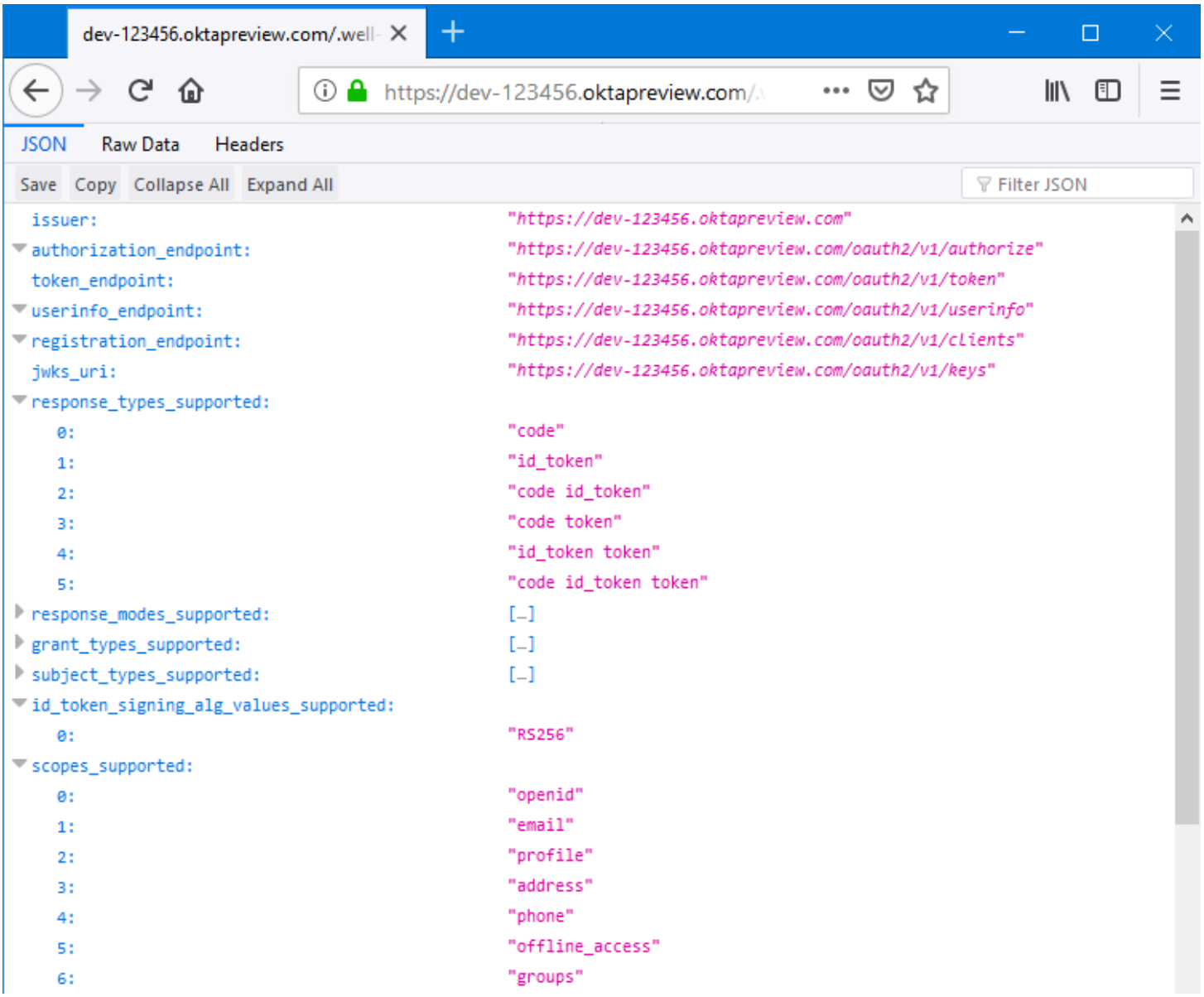
You will need the URL link to this page or a JSON export of this page. For example, if your IdP user login URL is something like this:

- **https://dev-123456.oktapreview.com/login/login.htm**

Then the metadata import URL may look like something like this:

- **https://dev-123456.oktapreview.com/.well-known/openid-configuration**

Here is an example of part of a metadata file for Okta. Notice that the URL link has "/.well-known/openid-configuration" at the end and is very similar to the login URL. It is recommended to use the URL specific to your IdP. Your IT department may choose to export this JSON file of this page and provide it to you. Either option will work.



> **NOTE**
>
> It is recommended to use the URL specific to your IdP. Your IT dept. may choose to export this JSON file of this page and provide it to you. Either option will work.

Scope Data

When a user is verified by the IdP a lot of the user specific data is not returned in the response file by default (i.e., username, email, firstname, lastname, etc). This user specific data is called the scope, and it can be returned if the **Scope** section of the Ignition Gateway is configured. The list of available scope definitions may be available from your IT deptartment or available from the developer documentation of the IdP you are using.

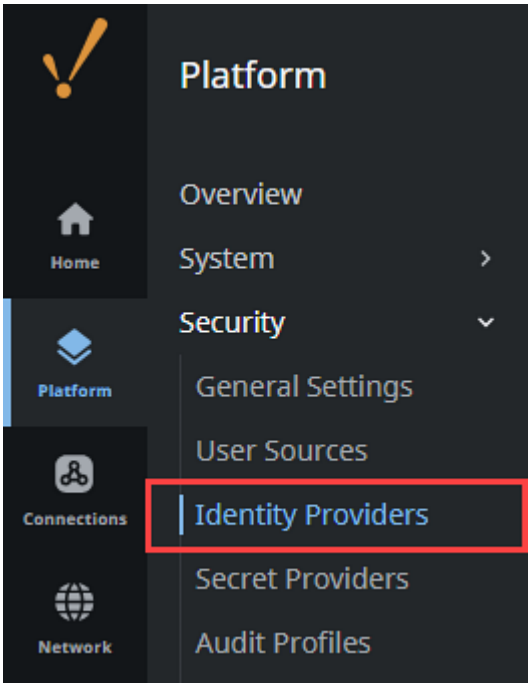
For this Okta example, the scope data is in the developer notes at https://developer.okta.com/docs/api/resources/oidc#scope-dependent-claims-not-always-returned

Test Login Credentials

You need an account specific to the IdP for testing purposes (Okta in this example). To test and verify the IdP account, login to your IdP. For our example, the Okta login page is shown here:

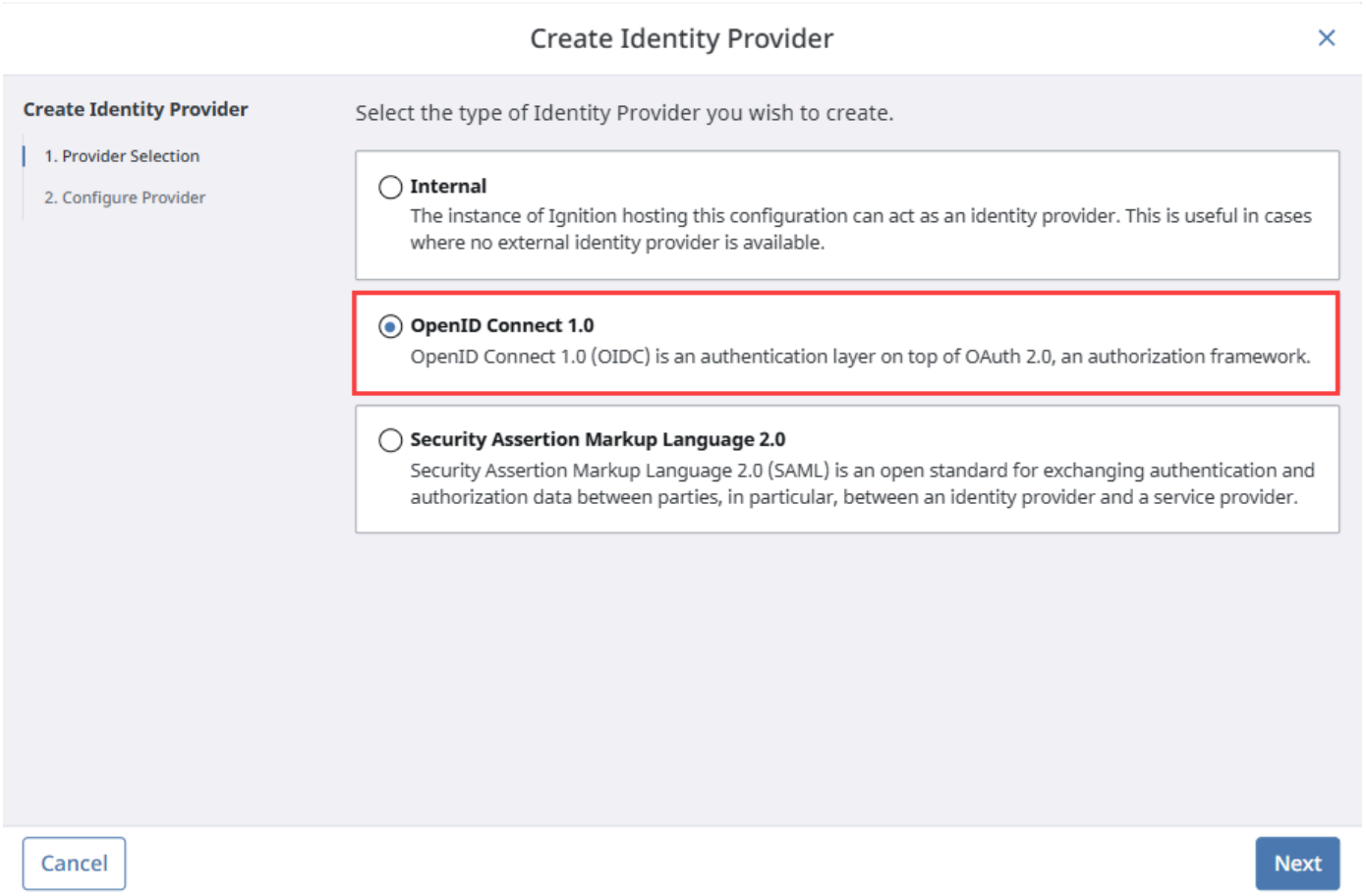
Configure Ignition Gateway

1. On the Gateway Webpage, click on the **Platform** tab. You will need to log in if you aren't already.
2. Under the Security section, click on **Identity Providers**. The Identity Providers screen is displayed. This screen will list all IdPs that have been configured. You can filter by name or adjust the number of IdPs displayed per page in the view.

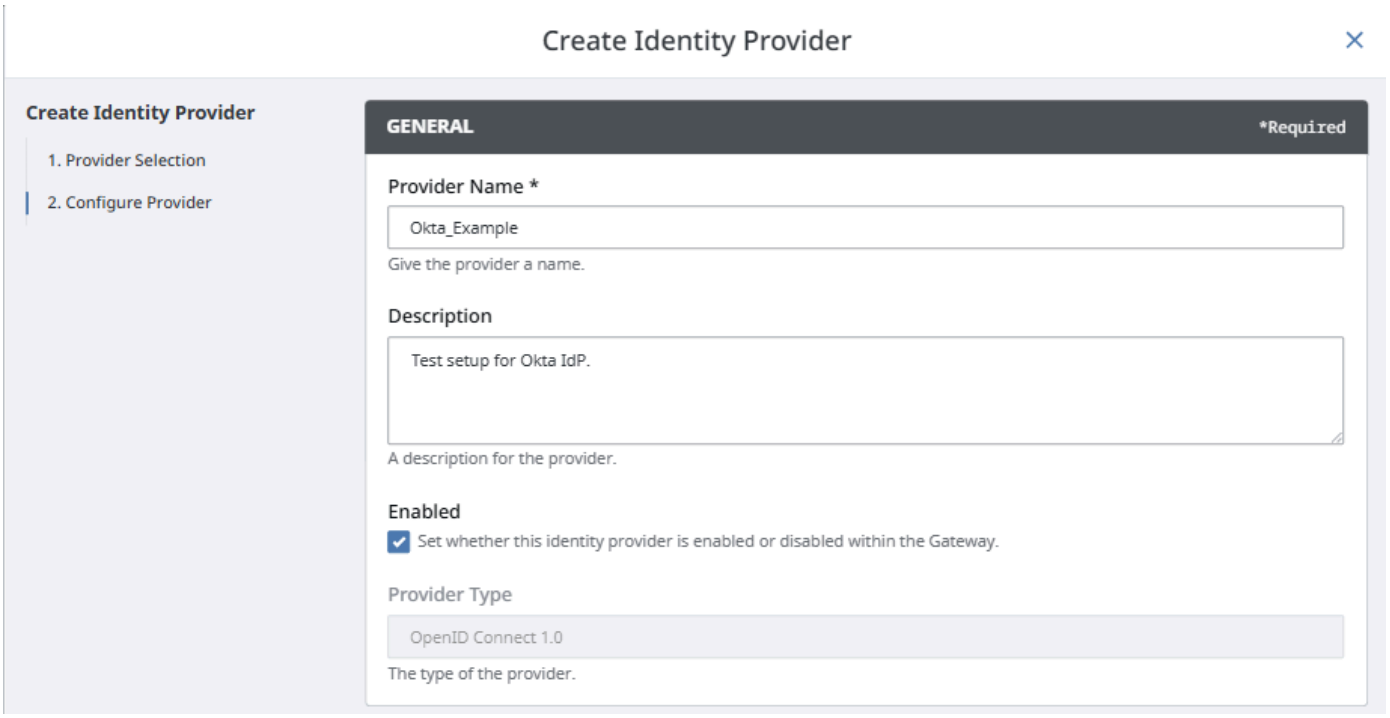


3. Click **Create Identity Provider +**.

4. Select the **OpenID Connect 1.0** option and click **Next**.

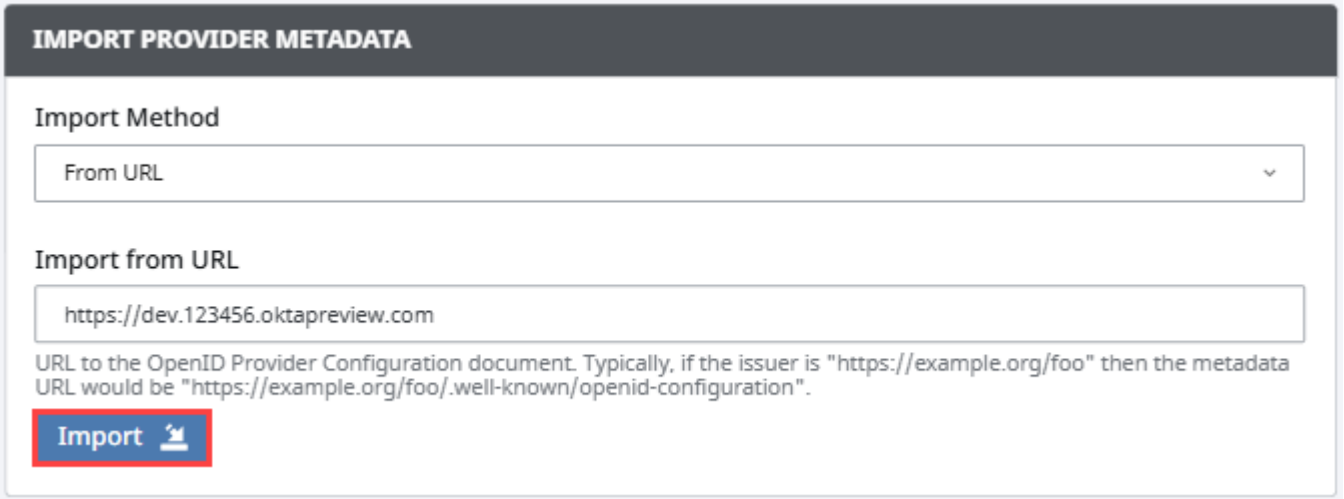


5. On the Basic Details screen, provide an **Provider Name**. You can also add an **Provider Description** if desired. The Provider Type field will fill in automatically from the previous screen.



6. The next section is **Import Provider Metadata**. In the **Import from URL** section, enter in the URL from earlier that shows the ".well-known/openid-configuration" link specific to your IdP. You can also import a file below if it was provided by your IT department.

7. Click on the **Import** button.



8. Ignition will now generate a URI redirect address for your Ignition server. It is listed just below the “Import Provider Metadata” area of the configuration form. In our example it is `http://localhost:8088/data/federate/callback/oidc`. You need to provide this URI to your IdP (usually this means giving it to your IT department).

> **NOTE**
>
> The URI should be a web address that is accessible from the end user's web browser.

Most OpenID Providers will require registering Ignition as a client. After the registration process is complete, the provider will generate a client ID and secret for Ignition, which is required below. This gives Ignition the ability to communicate securely with the provider. Most providers will also require a set of redirect URIs. This Ignition Gateway's Redundancy Role is set to **Independent**. The redirect URI for this Ignition Gateway is: **http://localhost:8088/data/federate/callback/oidc**

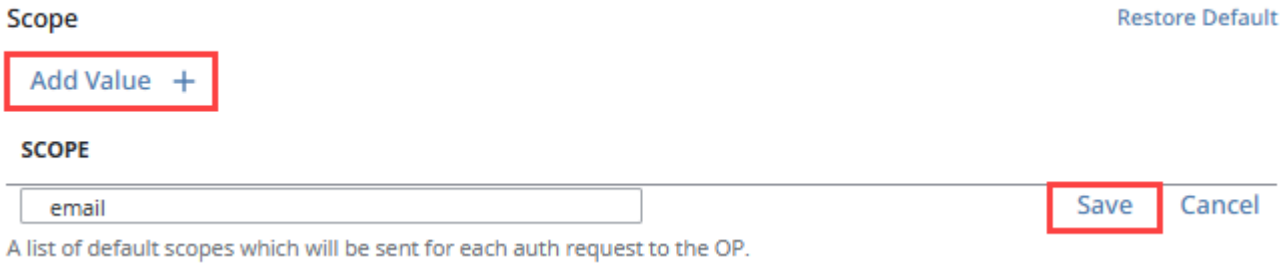
9. Once you've given your IT department the redirect address, they can add your Ignition server as an application to the IdP.
   Once they have done this, they can provide you with a **Client ID** and **Client Secret**. This is needed for the Ignition Gateway to properly communicate with the IdP.

> **NOTE**
>
> The IdP can use the same redirect address for the Login, Logout, and Initiate Login.

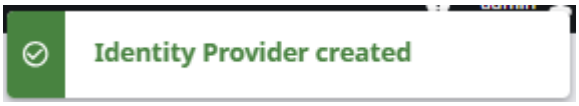
10. The next section is Provider Configuration. Most of the fields were filled in when you imported the IdP Metadata. Fill in the **Client ID** with the information obtained from your IdP (or IT department) and choose a password option for the **Client Secret**. If you don't have this information yet, you can put in temporary values and edit them later once the correct values are provided to you.

11. Providing scope is optional. These fields are specific to your IdP, and you may need to find the developer documentation specific to your IdP.

Select **Add Value +** to create a new scope field. Enter **email** and click **Save**. Repeat for each scope you want returned. For our example, the list of available scopes is in the Okta developer documentation: **https://developer.okta.com/docs/api/resources/oidc#scope-dependent-claims-not-always-returned**.

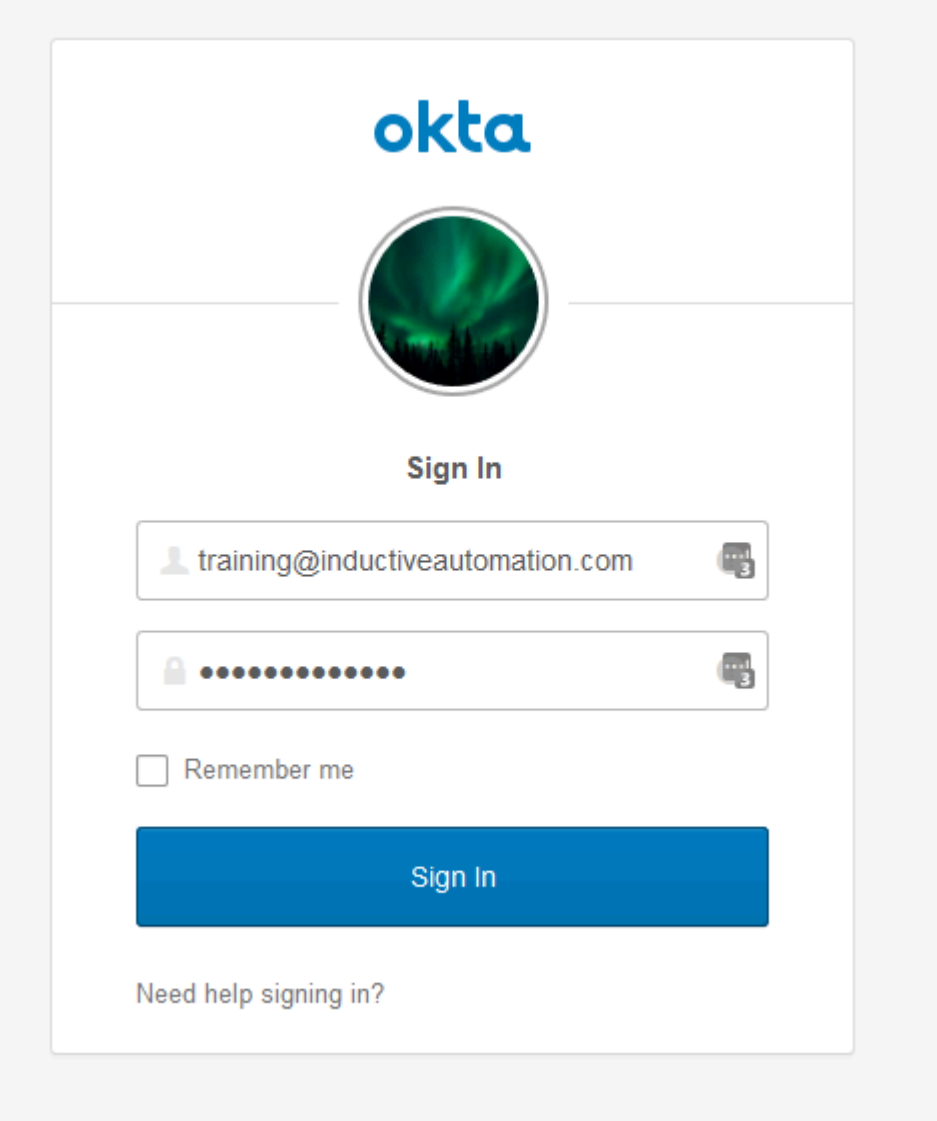


12. Press **Create Identity Provider** at the bottom right of the form. You'll see a confirmation message.



13. The next step is to perform a test login. From the Identity Providers screen, expand the three dots menu and select **Test Login**.

14. You will be re-directed to the Okta login. Enter in your test login credentials and click the **Sign In** button.



15. If the login is successful, you will be returned to the Identity Provider Test Login screen. The returned results will be displayed in the Results section.

SAML Example

This section provides an example of how to connect an Identity Provider that is using the SAML protocol. This example uses the Okta IdP service. Your IdP vendor may differ and the specific links will differ.

Prerequisites

Before you begin configuring Ignition there are some preliminary requirements that need to be done outside of Ignition:

- A configured remote IdP (Okta in this example)
- The metadata file specific to your IdP
- The scope data specific to your IdP
- Login credentials to use as a test

Configured IdP

An IT department is usually the one to set up and configure a remote IdP. You need a configured remote IdP that is compatible with SAML protocol.

At minimum there needs to be an account set up with the IdP, users added to the IdP account, and applications added to the IdP.

Metadata File

You will need the metadata file specific to your IdP. This document defines how to communicate with the IdP. It is usually a web page that allows the metadata file to be exported to an XML file. You will need the URL link to this page or an XML export of this page. For example, the metadata import URL may look like something like this:

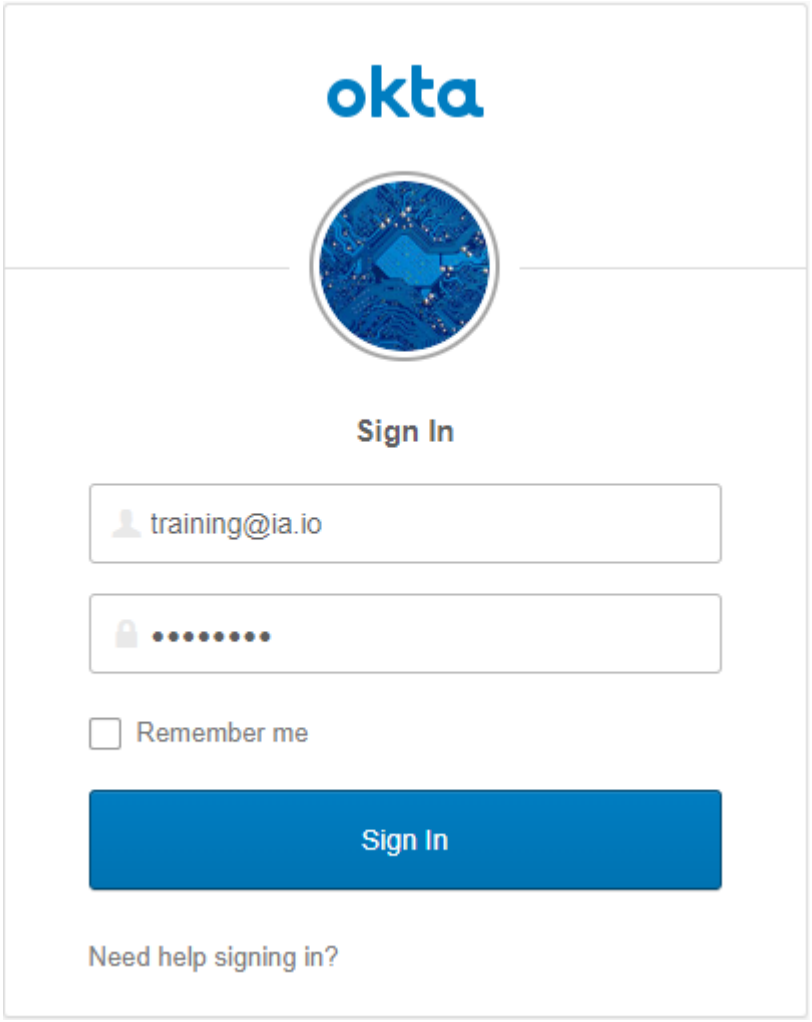
```
https://dev-123456.okta.com/app/esdfsdf7886sd6723hjkdf/sso/saml/metadata
```

Here is an example of part of a metadata file for Okta. Notice that file is XML format. You can use the file or the URL to automatically import the configuration into Ignition. Otherwise it will need to be manually typed in.

```
<?xml version="1.0" encoding="UTF-8"?>
- <md:EntityDescriptor entityID="http://www.okta.com/exkinzujwtqNw0fzi356" xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">
  - <md:IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol" WantAuthnRequestsSigned="false">
    - <md:KeyDescriptor use="signing">
      - <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        - <ds:X509Data>
            <ds:X509Certificate>MIIDpDCCAoygAwIBAgIGAWppx7LZMA0GCSqGSIb3DQEBCwUAMIGSMQswCQYDVQQGEwJVUzETMBEG
              A1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwNU2FuIEZyYW5jaXNjbzENMAsGA1UECgwET2t0YTEU
              MBIGA1UECwwLU1NPUHJvdmlkZXIxEzARBgNVBAMMCmRldi01MzAzMDIxHDAaBgkqhkiG9w0BCQEW
              DWluZm9Ab2t0YS5jb20wHhcNMTkwNDI5MTU0NjM2WhcNMjkwNDI5MTU0NzM2WjCBkjELMAkGA1UE
              BhMCVVMxEzARBgNVBAgMCkNhbGlmb3JuaWExFjAUBgNVBAcMDVNhbiBGcmFuY2lzY28xDTALBgNV
              BAoMBE9rdGExFDASBgNVBAsMC1NTT1Byb3ZpZGVyMRMwEQYDVQQDDApkZXYtNTMwMzAyMRwwGgYJ
              KoZIhvcNAQkBFg1pbmZvQG9rdGEuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
              26cLu1q2eNnUp4VBhnsJ5mucEbpSReDzaG8/CLNj73mp5dCicLZ/zrw/6SO+VblV2drShOJDQztX
              sBfPOCm7cn2QV8CARW8Ci/oPPnqrXp/8yetELp4tea1g7WQbkZxOOuStjRlOrW6ISKgKrWEo1LlI
              6iUM7kdDbU7GIlUUhY6KpSuMpT3iQq0MiQV7v+FVFx03VpGIaVFGIPY4hbDc0PwURN3MBw4cnhCK
              kTsTAqwSwH0N+9wsjwYWhEEMDYq/Eo0pbA2sGkChkz/vYJSUXjWMKnD432T3tYNYxEm95HzAIJn3
              ixdn8IPTbiERH60LyYWQtn55TAf5oCCq4UxGOwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQA4PWYO
              QBdfPnkwX2TqacsmBnliPvQaslzjek/9+jL3PKvR/GQIQMFaFis2fC+ucB426d0u3EbWSB8e9v4
              deSVWytwpMEHQijKHV8zjaX2AcWWRgiv/1VR4Klp2cahX3h2tU+n87A3a3su2O1VuJ+inSIxquTu
```


Test Login Credentials

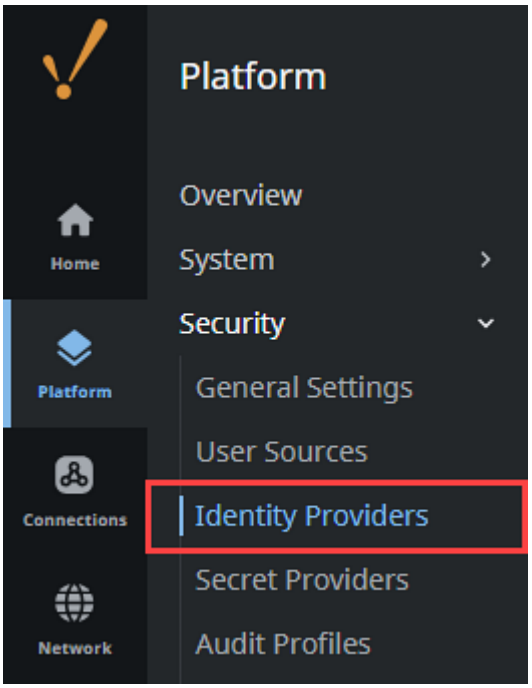
You need an account specific to the IdP for testing purposes (Okta in this example). To test and verify the IdP account, login to your IdP. For our example, the Okta login page is shown here:



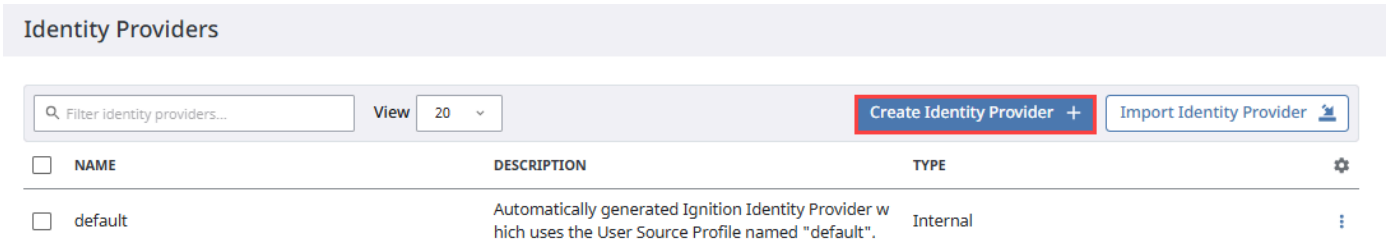
Configure Ignition Gateway

1. On the Gateway Webpage, click on the **Platform** tab. You will need to log in if you aren't already.

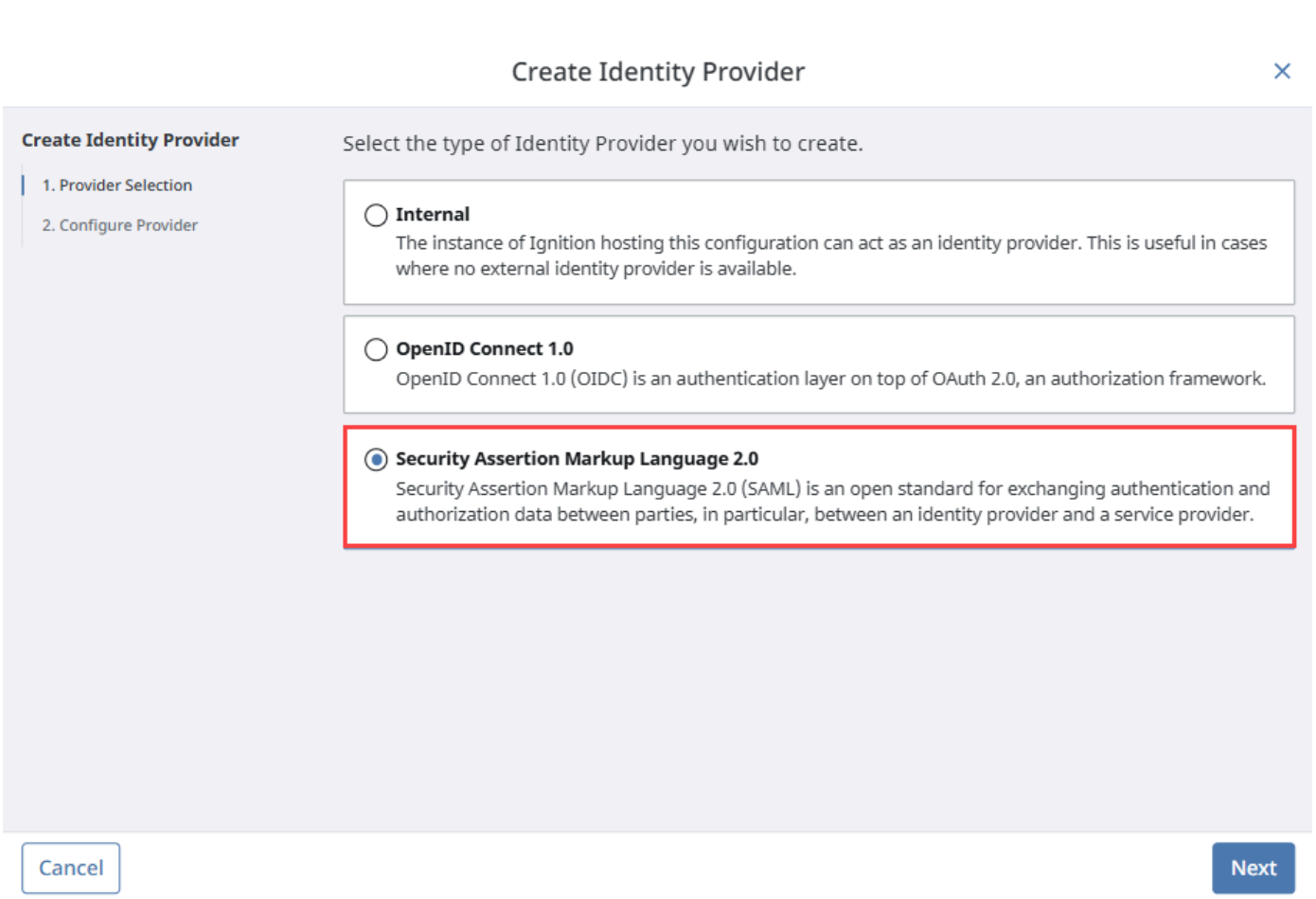
2. Under the Security section, click on **Identity Providers**. The Identity Providers screen is displayed. This screen will list all IdPs that have been configured. You can filter by name or adjust the number of IdPs displayed per page in the view.



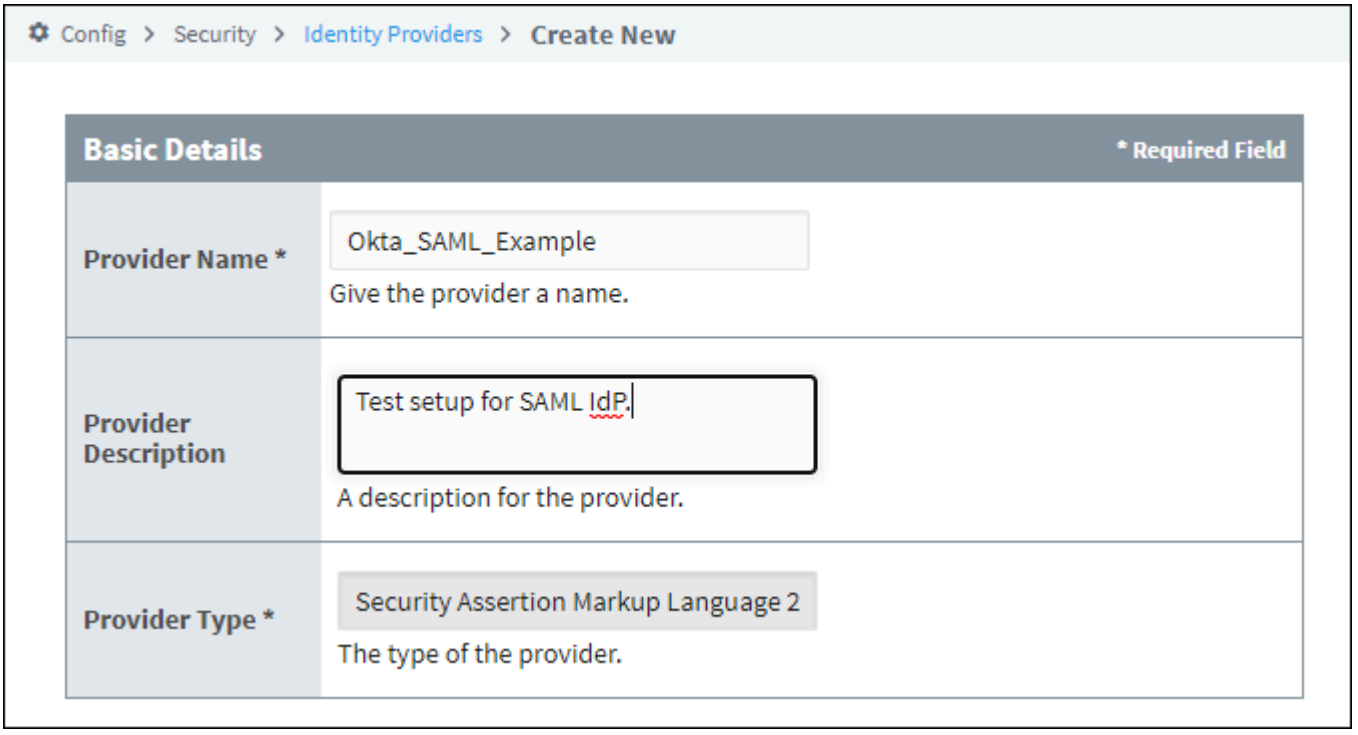
3. Click on the **Create Identity Provider +** link.



4. Select the **Security Assertion Markup Language 2.0** option and click **Next**.

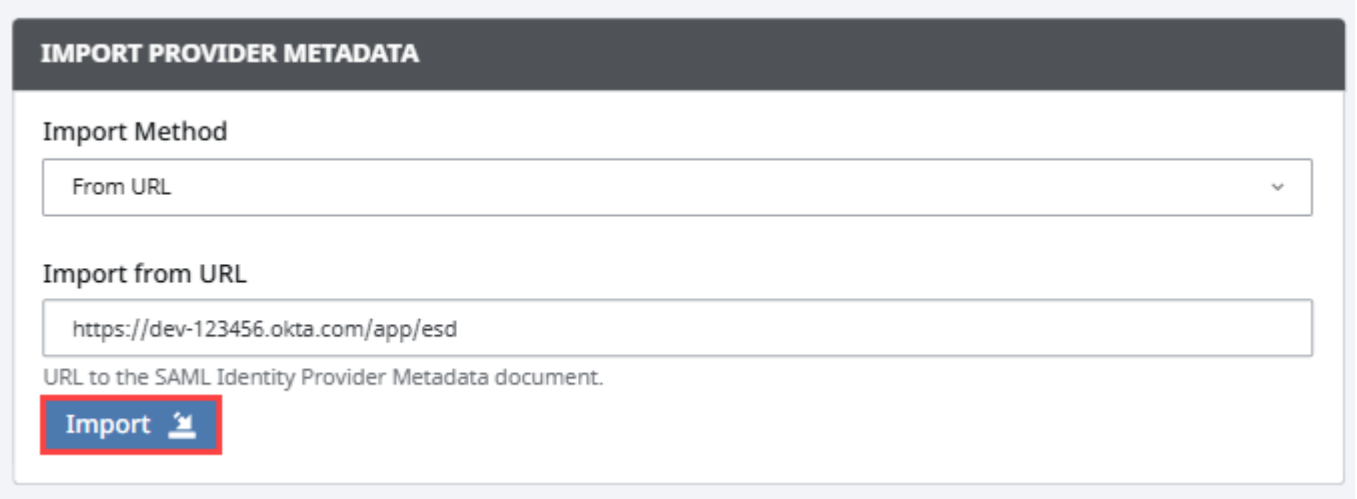


5. On the Basic Details screen, provide an **Provider Name**. You can also add an **Provider Description** if desired. The Provider Type field will fill in automatically from the previous screen.



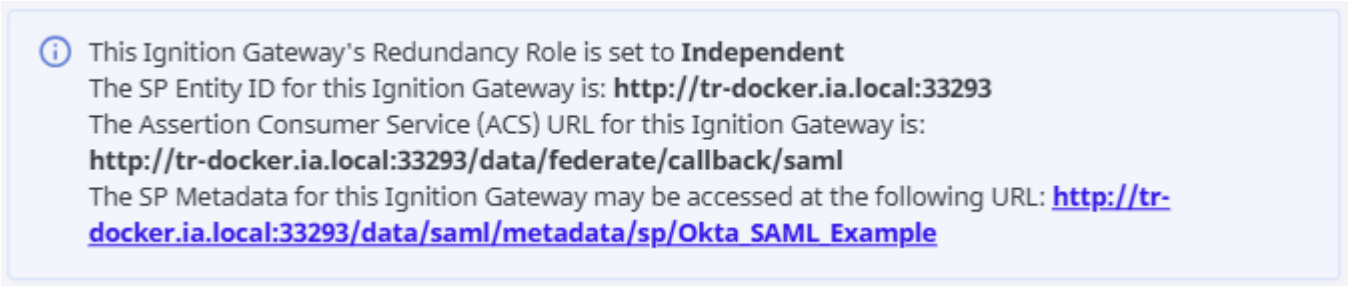
6. The next section is **Import Provider Metadata**. In the **Import from URL** section, enter in the URL from earlier specific to your IdP. You can also import a file below if it was provided by your IT department.

7. Click on the **Import** button.



8. Ignition will now generate a URI redirect address for your Ignition server. It is listed just below the “Import Provider Metadata” area of the configuration form. In our example it is `http://10.10.110.86:8088/data/federate/callback/saml`. You need to provide this URI to your IdP (usually this means giving it to your IT department).

> **NOTE**
>
> The URI should be a web address that is accessible from the IdP server.

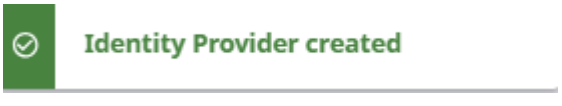


9. Once you have given your IT department the redirect address, they can add your Ignition server as an application to the IdP.

> **NOTE**
>
> The IdP can use the same redirect address for the Login, Logout, and Initiate Login.

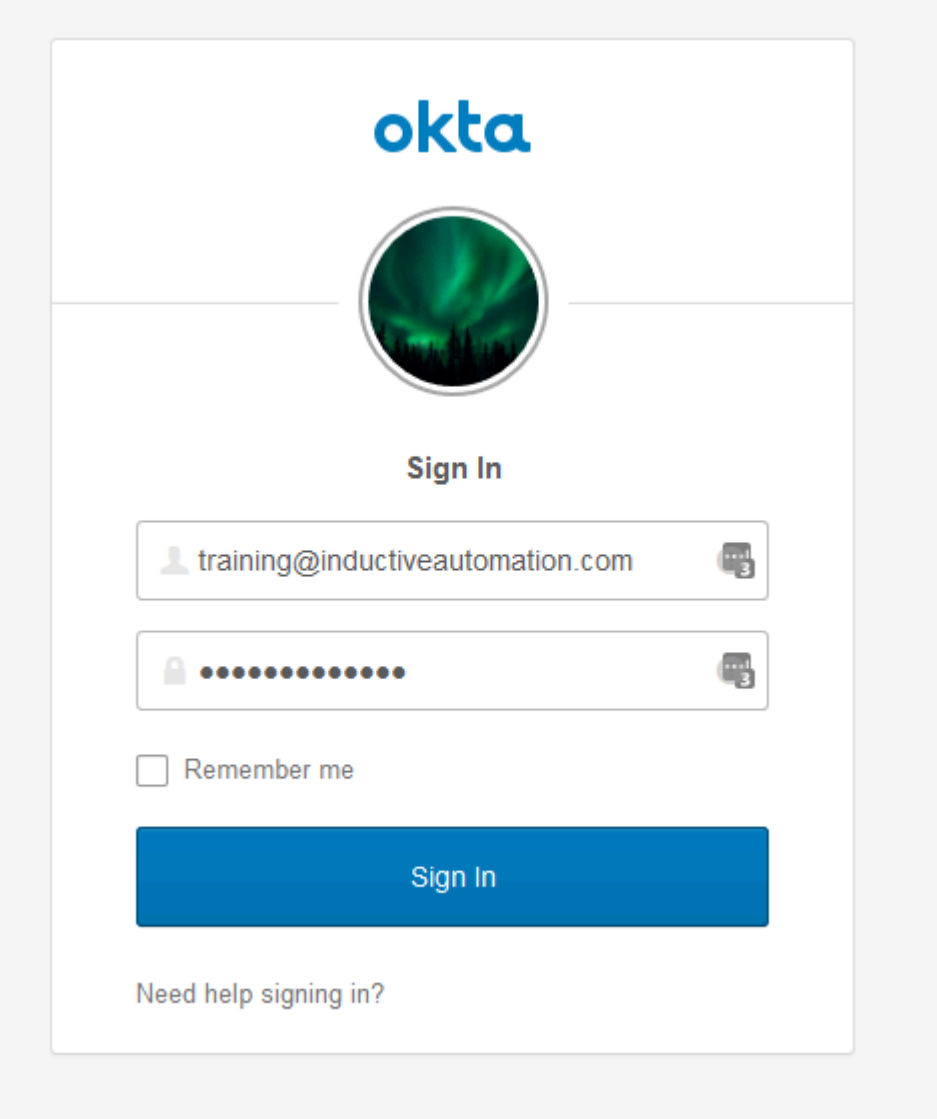
10. The next section is Provider Configuration. Most of the fields should now be filled in when you imported the IdP Metadata.

11. Click Create Identity Provider at the bottom right of the form. You'll see a confirmation message.



12. The next step is to perform a test login. From the Identity Providers screen, expand the three dots menu and select **Test Login**.

13. You will be re-directed to the Okta login. Enter in your test login credentials and click the **Sign In** button.



14. If the login is successful, you will be returned to the Identity Provider Test Login screen. The returned results will be displayed in the Results section.

User Grants

A User Grant is a way to directly assign a user to a **Security Level**, even if they do not meet the requirements of the **Security Level Rules**. User Grants essentially act as an override to the original rules of the Security Level.

User Grants are accessed from the Gateway Webpage **Platform** section in **Security > Identity Providers**. Users can be added and edited so that Security Levels can then be granted to them.

> **NOTE**
>
> When adding, editing, and deleting users in User Grants, you are only modifying the User Grant (whether the user is granted permission that overrides the Security Rules). The user is not changed in the Identity Provider.

Users are identified by either their username or their ID from the provider response document. Once you have identified a user, you can assign them any number of grants to Security Levels. Selecting a level will automatically select all security levels above it. The User Grants can only be applied to a user after they authenticate with the Identity Provider, though the grants do not have to be for levels within the Authenticated branch.

> **NOTE**
>
> The system can't validate any user created here against actual users in the Identity Provider (IdP). Instead, the username or ID needs to be entered **exactly**, including case-sensitivity, and when a user logs in, the system will check to see if they match any of the configured usernames/IDs to give User Grants to.

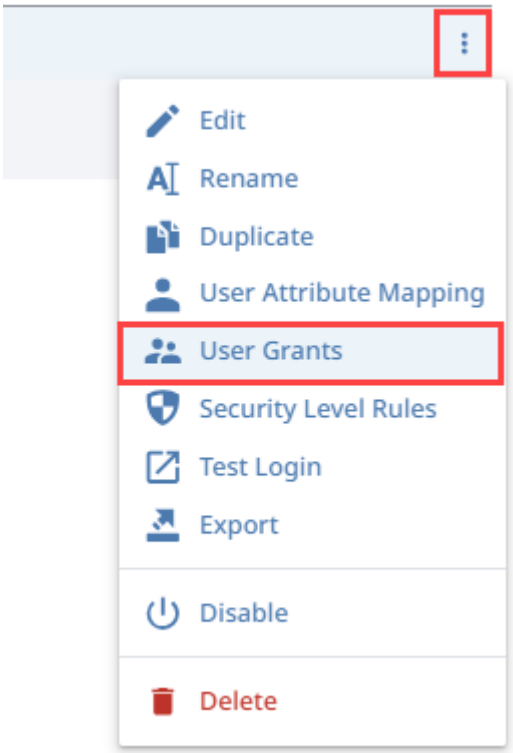
Configuring a User Grant

There are two parts to configuring a User Grant: Adding a user and applying User Grants.

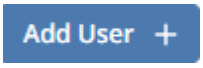
1. From the Gateway Webpage **Platform** tab, click on **Security > Identity Providers**. The screen will refresh and you will see a list of all your IdPs.

2. Choose the IdP and expand the three dots menu to see the actions in the dropdown list.

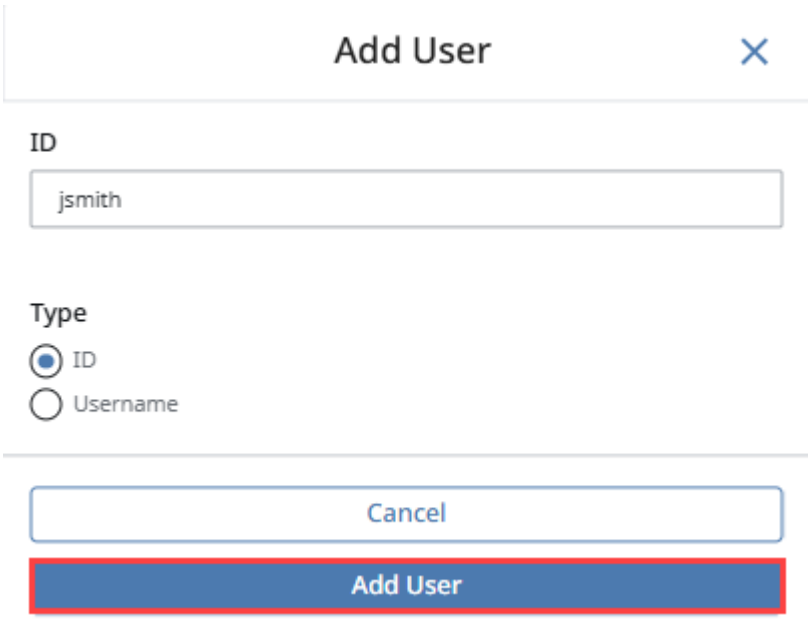
3. Select **User Grants**.



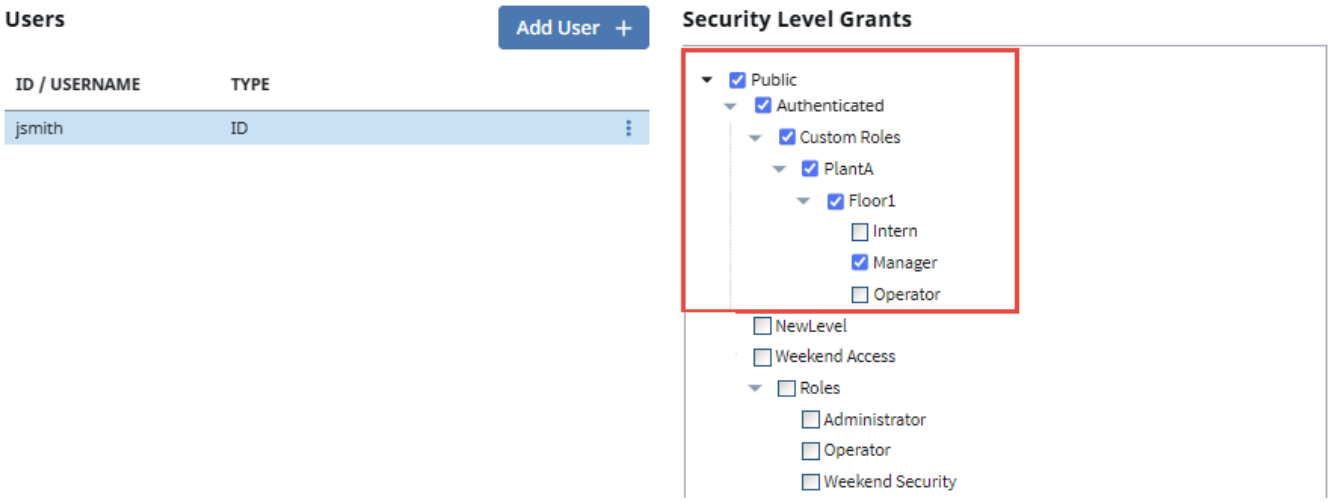
4. To add a new user, click **Add User +**



5. Choose how you will identify the user; either with a username or an ID. Click **Add User** to save the changes.



6. With the user created and highlighted in the Users table, select Security Levels to grant them when they Authenticate with this Identity Provider.



7. Click **Save Changes**.

Now you can test this user through the **Test Login and Logout** screen to verify the new roles have been assigned.

Test Login and Logout

On the Identity Providers screen you can test a username and password combination against an Identity Provider (IdP).

When you select the Test Login option for your IdP, it will confirm the IdP name and Type that you are testing against. It gives you a way to test your attribute mapping configuration and your security level rules / direct user grants configuration.

Clicking the Test Login action will redirect you to the IdP where you can login. Upon successful authentication with the IdP, the page navigates back to Ignition where a Test Results panel displays the response data. These results can vary between IdPs, so it can be useful to test out a login to see what your IdP returns.

You can then use the Test Logout option to log out of the ID you were testing.

Ignition's IdP returns an 'amr' attribute that indicates how the user was authenticated.

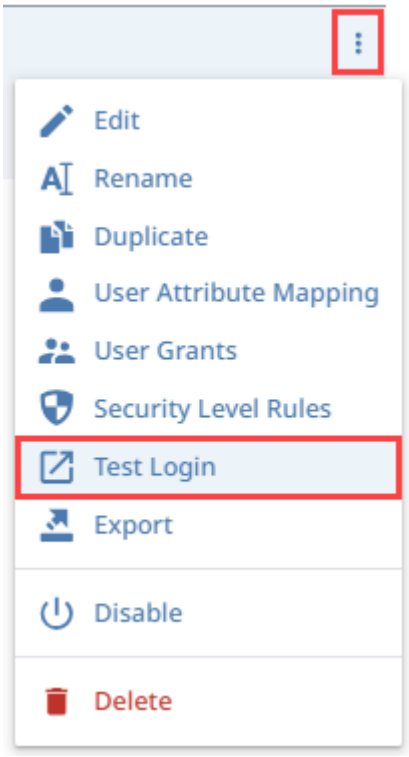
- If the user was authenticated with a username and password challenge, the amr returns: ["uname", "pwd"]
- If the user was authenticated with a badge challenge, the amr returns: ["badge"]
- If the user was authenticated with a badge and password challenge, the amr returns: ["badge", "pwd"]

With this feature, you can enable different security levels based on how the user authenticated. In a security level rule, you could enter:

```
containsAll ({idp-attributes:amr}, 'uname', 'pwd')
```

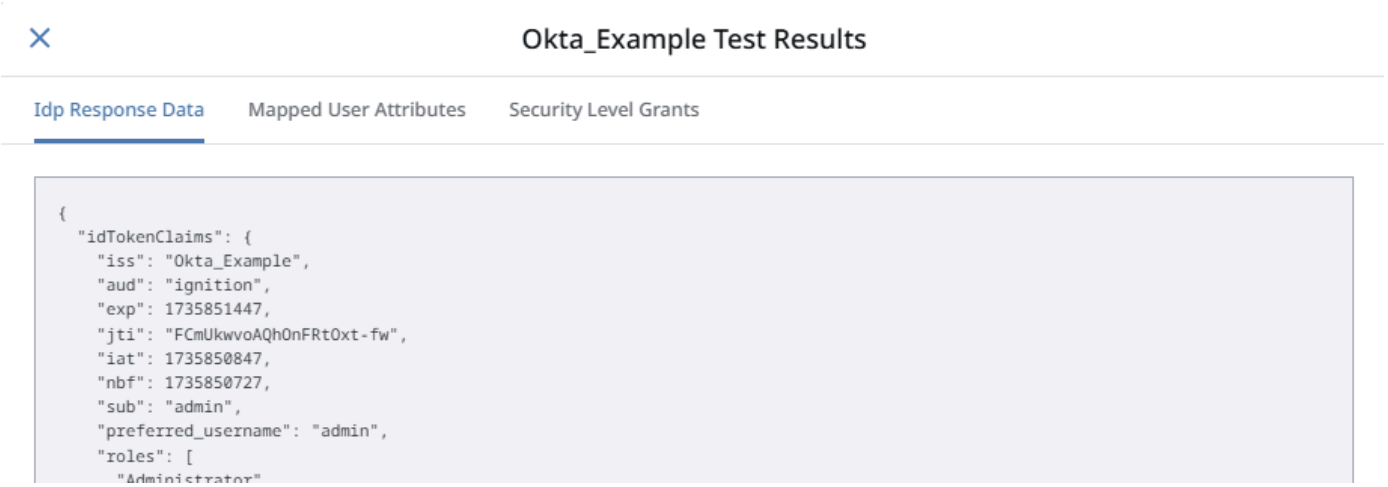
Test a Login

1. From the Gateway Webpage **Platform** tab, go to **Security > Identity Providers**. The window will refresh and your list of Identity Providers will be displayed.
2. Choose the Identity Provider and expand the three dots menu to see the actions in the dropdown list. Select **Test Login**.

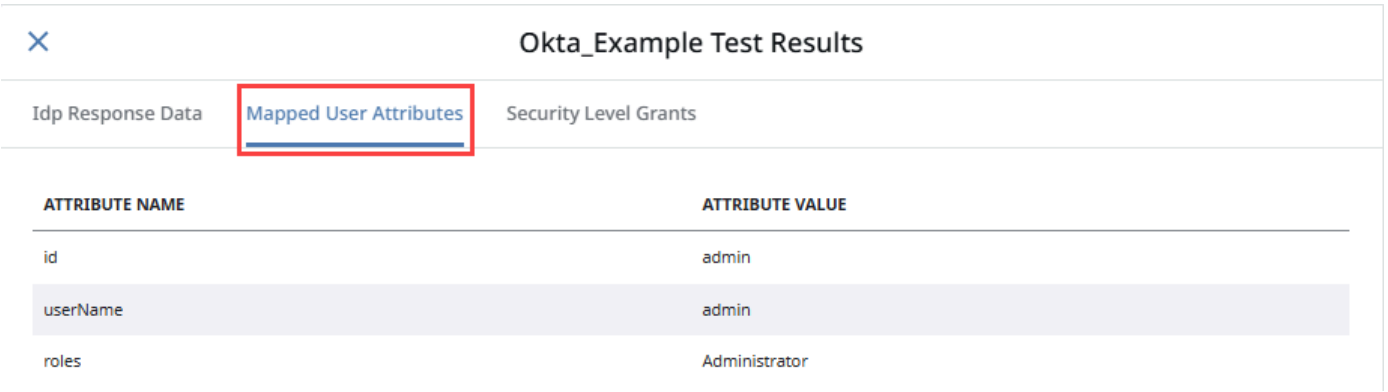


3. Log in at your IdP's login screen.

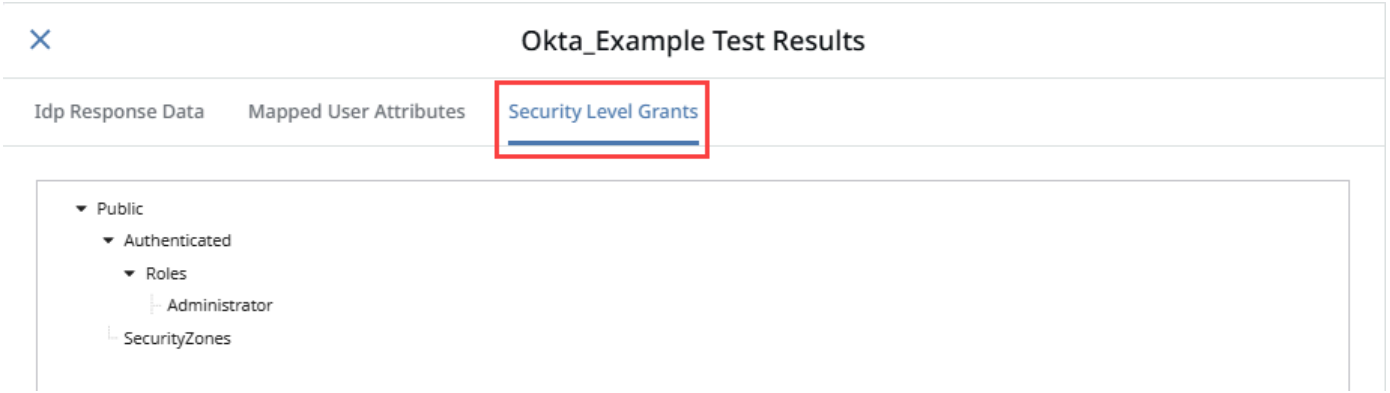
4. If the login is successful, you will be returned to the Identity Provider Test Results panel. The returned results will be displayed under the IdP Response Data tab.



5. Click on the Mapped User Attributes tab to view the user attributes for the currently logged in user.



6. Click on the Security Level Grants tab to view the Security Levels for the roles of the currently logged in user.



Test a Logout

After testing a User ID, you do not want to stay logged in as the user. You can use the Test Logout function to log out. For the Ignition IdP, this function also logs you out of the IdP. For an OpenID Connect IdP, this function will also log you out of the IdP if you have a **Logout URL**.

1. To log out of the ID you were testing, click **Test Log Out** on the Test Results panel.

2. You will get a confirmation message of a successful logout.

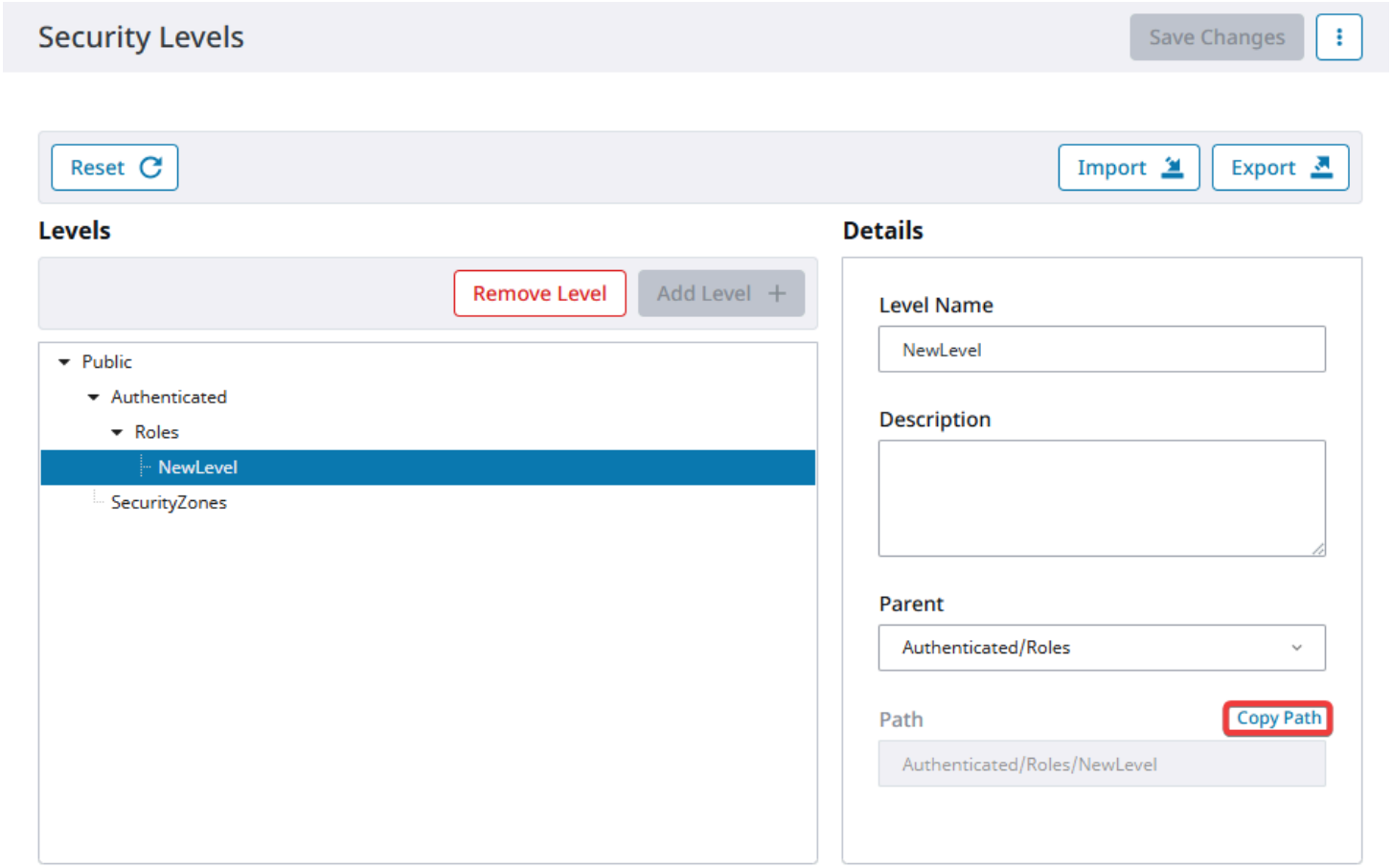
Security Levels

With Security Levels, you define a hierarchy for access inside a Perspective Session or Vision Client. This authorization system provides a way for you to map roles from an Identity Provider (IdP) to Ignition roles. Any IdP can be used to provide roles, and security levels are independent of the type of IdP being used. Any role from the IdP is automatically granted to the user as a role, but only roles in your Security Levels are available to the security screens in the Designer. You can also use the **User Grants** option to grant additional access for each user.

Security Levels are defined at the Gateway and they are arranged in a tree structure. Each child (nested) level of the tree inherits the security of its parent levels. There are four reserved Security Levels in the platform:

- Public
- Authenticated
- Authenticated/Roles
- Security Zones

To access Security Levels, go to the Gateway Webpage under the **Platform** tab, and choose **Security > Security Levels**. You can copy the selected Security Level Path to your clipboard by clicking **Copy Path** in the Security Level Details panel.



NOTE

Adding or modifying a role in the Security Levels section of the Gateway will not impact or change roles located in the **Users, Roles** section, as they are separate entities within Ignition. You will need to update the roles in the Users, Roles section separately from the roles in Security Levels.

Reserved Security Levels

The reserved security levels are mostly created for you, and have special rules that determine when a user is granted that level. They can't be renamed or deleted.

Public

All users are always granted the Public security level, even if they are not authenticated (logged in). Public security level indicates open access and the least amount of security. A session that only has the Public security level is not authenticated. This is similar to being a guest or anonymous. Unless another security level is required, guest access will be allowed. The Public security level is the ancestor of all other security levels in the hierarchy.

Authenticated

The Authenticated Security Level is a child of the Public Security Level. If a session has authenticated against the configured IdP successfully, the Authenticated Security Level is granted. Users are required to be logged in in order to have access to this level.

Authenticated/Roles

The Roles level is a special level which itself has no special rules, but it acts as a parent placeholder for potential roles returned from the IdP. This particular level is not configurable; however there can be levels added underneath the Roles level as children. You can only add one level of children to the Roles Security Level, but Custom Roles can be nested as deeply as you want. These levels should correspond to the names of roles that would be expected from the IdP. If the IdP provides role information, these roles are automatically mapped to the child security levels underneath Authenticated/Roles. The names of the roles must match exactly for them to be correctly mapped to.

> **NOTE**
>
> Upon installing a fresh Gateway, an Administrator role will be listed under the Authenticated/Roles section and then assigned to an Admin user. This helps prevent users from having to manually create this role.

For example, if you authenticate against the Ignition IdP configured to delegate to the Internal user source, and your user was granted the roles **A**, and **B**, you would have (at a minimum) the following security levels granted to you:

- Public
  - Authenticated
    - Roles
      - A
      - B

> **NOTE**
>
> Only roles under the special Authenticated/Roles level can be used with the Classic Authentication Strategy.

SecurityZones

The SecurityZones level is another special placeholder level that itself has no rules but is a parent for all of the Security Zones on the Gateway. Security Zones are automatically pulled in from the

Gateway. A **Security Zone** is a list of Gateways, Computers, or IP addresses that are defined and grouped together. This group is a zone on the **Gateway Network**, which can have additional policies and restrictions placed on it. Security Zones provide a way to bridge the IdP method of permissions with location-based permission modeling. You cannot add, edit, or remove the SecurityZones node or any node in the SecurityZones sub-tree.

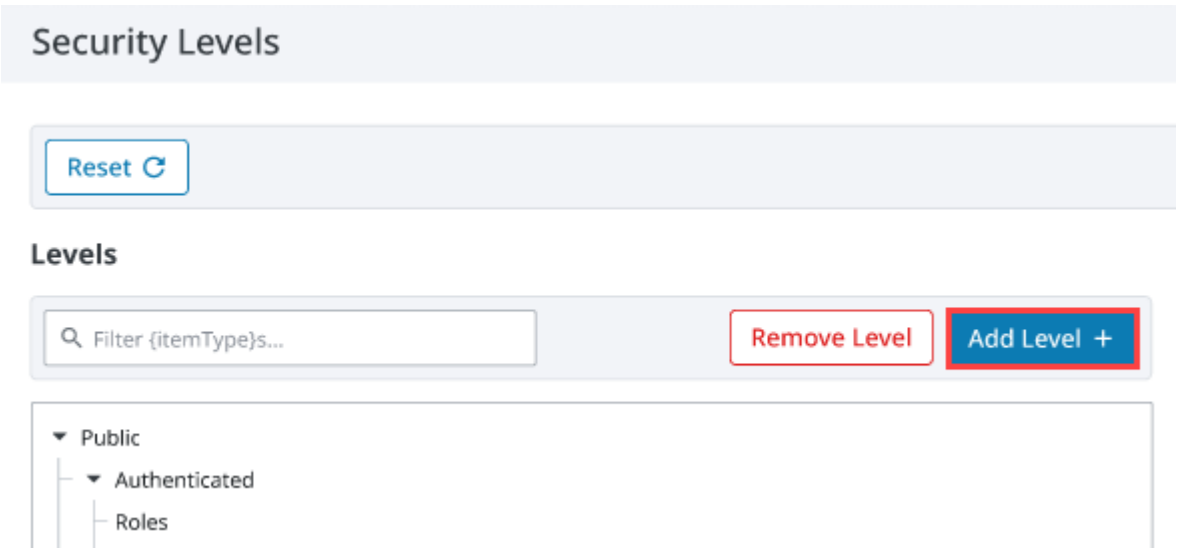
Custom Security Levels

Custom Security Levels can be added to almost anywhere within the tree. When these levels are granted to a user is determined by the **Security Level Rules**, which can pull information from the IdP, Security Zones, and even tags. The placement of custom Security Levels can affect when they may be potentially granted to a user. Any custom levels set under the Public level, but not within Authenticated, do not need to have a user authenticate against the IdP to be granted to a user. However, custom levels within Authenticated do need to have the user authenticate to be granted to the user, even if the rule for that level does not use any of the IdP attributes.

> **NOTE**
>
> Custom Security Levels do not work with the Classic Authentication Strategy.

Add a New Security Level

1. From the Gateway Webpage **Platform** tab, click on **Security > Security Levels**.
2. In the Security Level tree, select the level that will be a parent for the new level.
3. Click the **Add Level +** button.

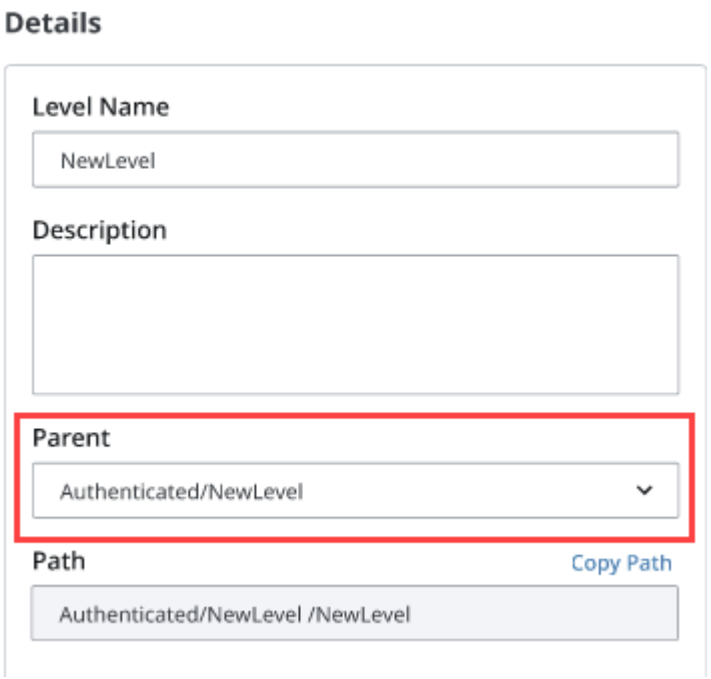


4. In the **Details** screen area, enter the **Level Name**.

> **NOTE**
>
> Security Level names within the same parent must be unique.

5. Add a **Description** for the new level (optional).

6. The path for the parent is filled in automatically. Use the **Parent** dropdown list if you want to change the parent for this new level.



7. If you make changes to Security Levels, but decide not to save them, you can use the **Reset** button to return the tree view to its currently saved configuration.

8. To save the changes, click **Save Changes**.

Edit a Security Level

1. From the Gateway Webpage **Platform** section, click on **Security > Security Levels**.

2. In the Security Level tree, select the level that you want to edit.

3. Make the desired changes in the **Details** area.

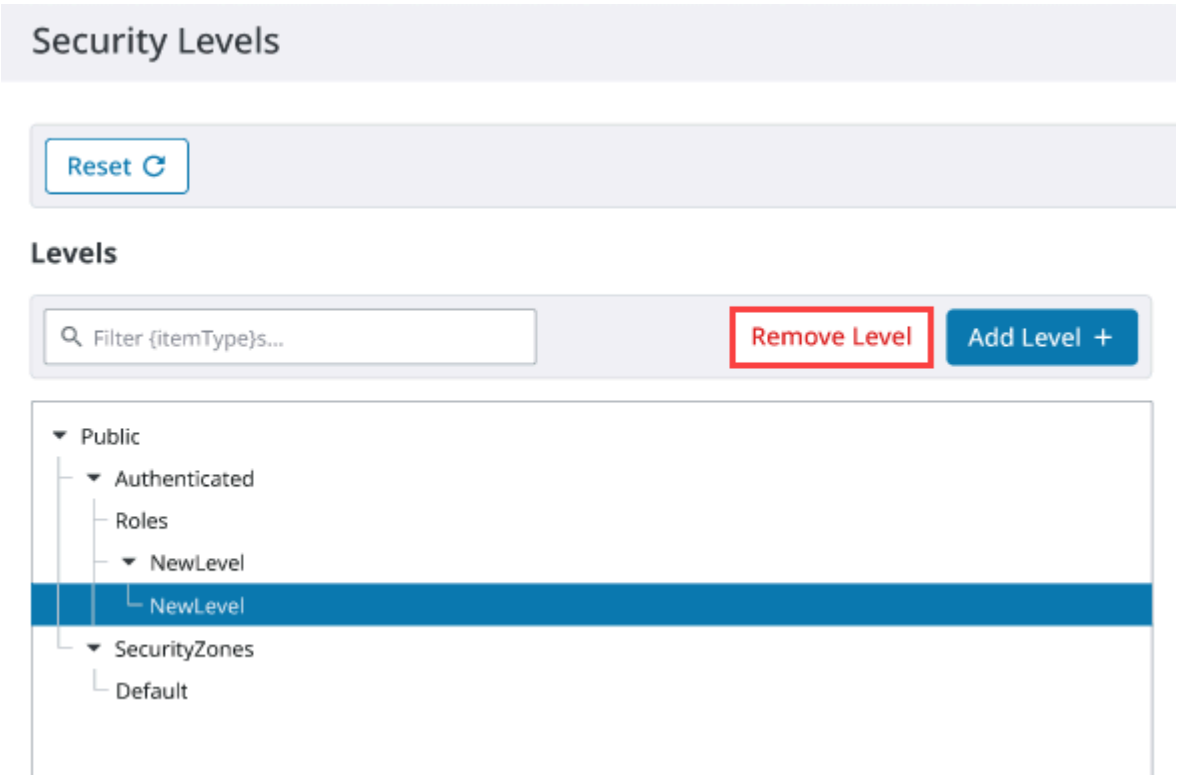
4. If you make changes to Security Levels, but decide not to save them, you can use the **Reset** button to return the tree view to its currently saved configuration.

5. To save the changes, click **Save Changes**.

Delete a Security Level

> **CAUTION**
>
> When you delete a Security Level, all children under that level will also be deleted.

1. From the Gateway Webpage **Platform** tab, click on **Security > Security Levels**.
2. In the Security Level tree, select the level that you want to delete.
3. Click **Remove Level**.



4. In the confirmation box, click **Remove** to confirm the delete.
5. To save the changes, click **Save Changes**.

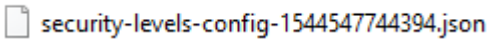
> **NOTE**
>
> Deleted security levels will need to be unchecked in the Designer permissions configurations with new settings saved to avoid permission errors. Warning indicators for selected security levels that no longer exist will be visible on the Project Properties, Event Configuration, Edit Permissions, and Tag Editor windows when applicable.

Import a Security Levels Configuration

1. From the Gateway webpage **Platform** tab, click on **Security > Security Levels**.
2. Click **Import**.
3. Choose a security levels configuration file to import.
4. Click **Open**.

Export a Security Levels Configuration

1. From the Gateway Webpage **Platform** tab, choose **Security > Security Levels**.
2. Click **Export**.
3. The security levels configuration will be saved as a .json file with a unique number, for example:

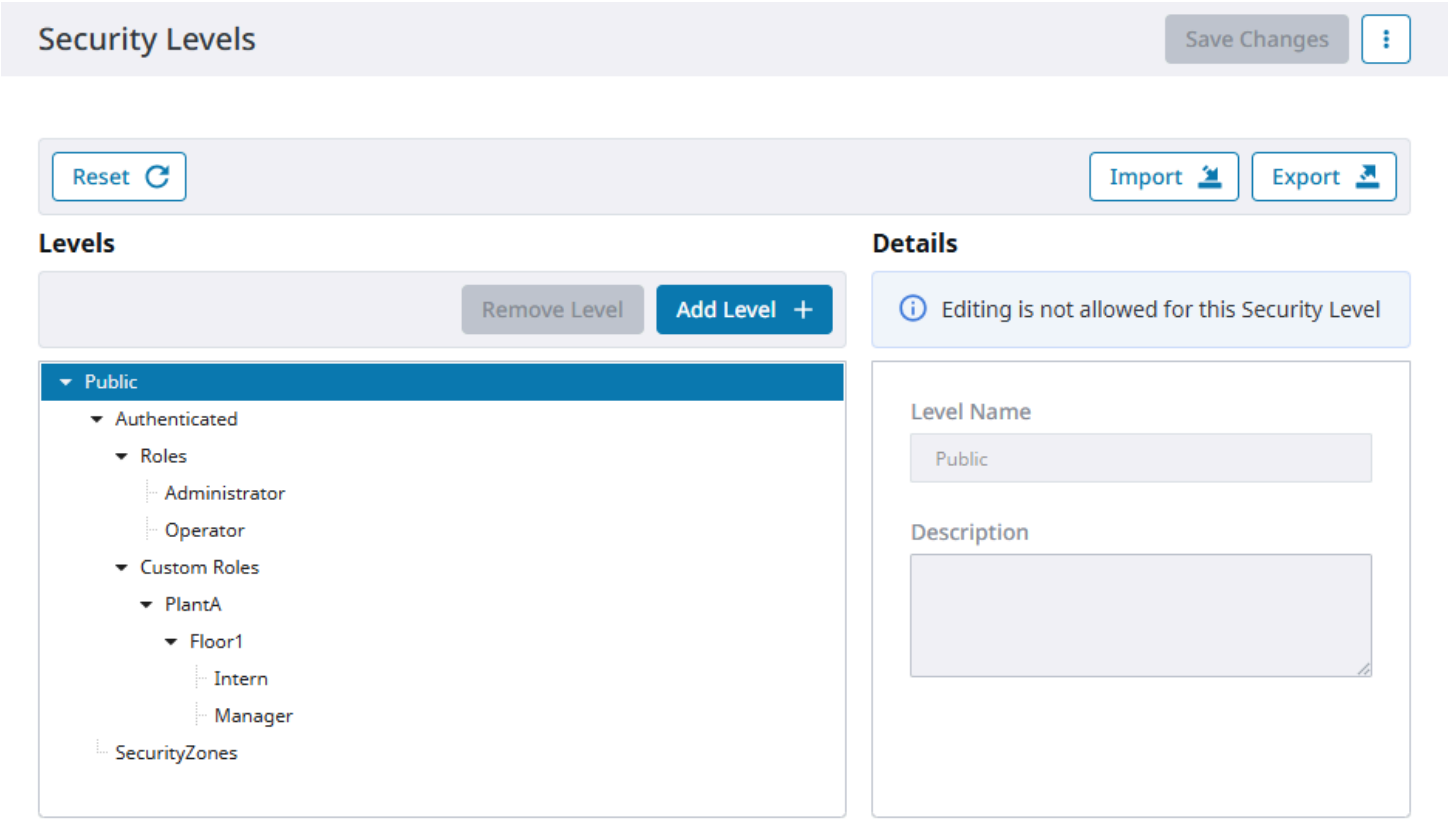


Security Level Rules

When a user accesses a project, they can fall into one or more of many **Security Levels** that you can set up. The Security Level Rules determine if the user has that Security Level or not. The rules come in the form of expressions that can access any of the Expression Language's functions, tag values, Security Zone information, or Identity Provider attributes. Security Level Rules are accessible from the Gateway Webpage **Platform** tab in **Security > Identity Providers**.

Predefined Rulesets

One thing you may notice right away is that on the Security Level Rules page, many of the built in Security Levels are missing. Some Security Levels don't allow you to create an expression that defines their rules. These particular Security Levels already have a set of rules that govern how a user gets them. The Security Levels corresponding to the different Security Zones are automatically given to users depending on which zones they fall into. Similarly, the Security Levels that correspond to the Roles a user gets while authenticated are automatically given to users depending on what roles we receive from the Identity Provider. These Security Levels are removed from the tree here because rules can't be defined on them. The "Public" and "Authenticated" Security Levels also can't have Security Level Rules defined on them, but are present in the tree because they can potentially be a parent to custom nodes which can have Security Level Rules. The Public level is granted to every user when they open the project, and the Authenticated level is granted when the user authenticates against an Identity Provider, regardless of what roles they may have.



Defining Security Level Rules

For the Security Levels that can have rules defined, the rules are defined in the form of an expression which should return either True or False, the results of which determine whether a user falls into that level or not. The rules can take advantage of everything the **expression language** has to offer, including the built in expression functions and any tag values. The expressions here also have the unique ability to access attributes from the Identity Providers response document for the authenticated user, as well as what Security Zone the user falls into.

Special Object Reference

These special objects can be used to reference information gathered from the IdP response document, mapped user attributes, or the Security Zone that the user falls under.

Object Type	Reference	Description
Response Attributes	Varies, see **Expression Mappings** for syntax info.	References Response Document Attributes directly.

User Attributes	`{user:id}`
	`{user:userName}`
	`{user:firstName}`
	`{user:lastName}`
	`{user:email}`
	`{user:roles}`
		References mapped User Attributes. Relies on mappings configured on the **User Attribute Mappings** page. The `roles` attribute can be handled using the **containsAll** or **containsAny**.

Security Zones	`{security-zones}`	References the collection of security zones that the user currently has. The collection can be handled using one of the unique functions: **containsAll** or **containsAny**.
Security Level Name	`{security-level-name}`	References the name of the security level being configured. For example, if the Security Level is "Authenticated/PlantA/Administrator", then the security level name is "Administrator"
Security Level Path	`{security-level-path}`	References the full path to the security level being configured, beginning with the first node below **Public**. For example, "Authenticated/PlantA/Administrator".

> **NOTE**
>
> Response and User Attributes are only available to Security Levels that fall within the Authenticated Security Level. See below under Special Considerations for Rules.

Special Function Reference

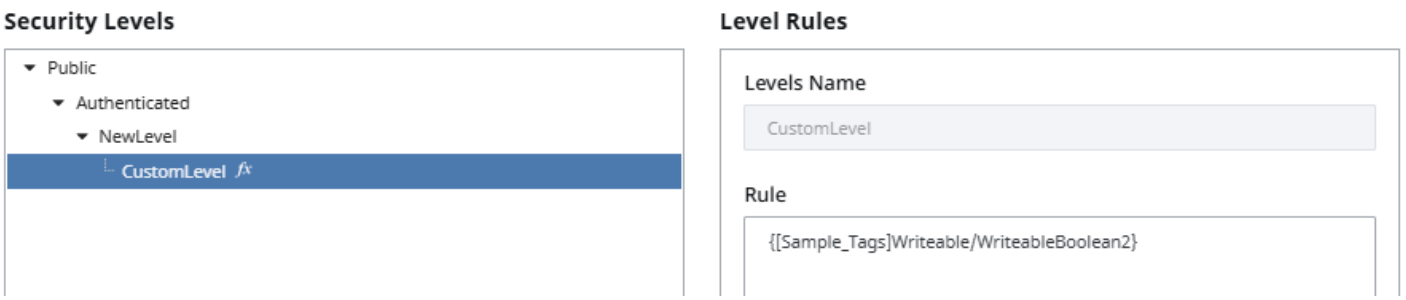
When writing an expression to determine Security Level Rules, there are functions available that are not a part of the normal set available to Expression Bindings. These additional functions are:

Function Name	Description	Example
containsAll(collection, element 0, ..., element N)	Checks to see if all of the listed elements are present in the collection object. The function requires at least two arguments, a `collection` and an `element`.	`containsAll( {security-zones}, 'PlantA', 'Floor1', 'Press Room' )`
containsAny(collection, element 0, ..., element N)	Checks to see if any of the listed elements are present in the collection object. The function requires at least two arguments, a `collection` and an `element`.	`containsAny( {attribute-source:idTokenClaims:roles}, 'Manager', 'Operator' )`

Evaluating Tag Values

Tag values can be accessed in the Security Level Rules expression area by encasing the Tag Path (including the Tag Provider) in braces, `{ }`

```
{[tagProvider]path/to/tag}
```



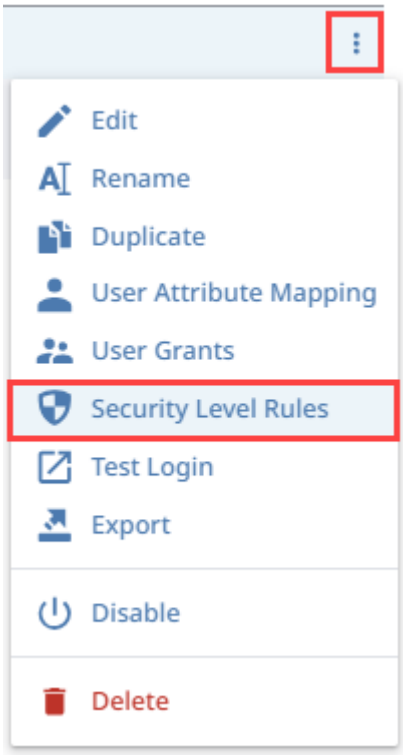
> **NOTE**
>
> Security Levels are determined on initial login for each session, so if a Security Level is using an expression that references a tag value, changing the value while the session is running won't change the Security Levels applied to the users already logged in.

Special Considerations for Rules

When defining rules for a Security Level, it is important to notice where in the Security Level tree you are. If you want to access information out of the Identity Provider such as the username, you will need to ensure that the Security Level is located in the Authenticated branch. User information is only captured once a user logs in, so that information will only fall under Security Levels that come from the user being Authenticated. If a Security Level lies outside of the Authenticated branch, then the level will only have access to information such as tag values and Security Zones.

Configuring Security Level Rules

1. From the Gateway Webpage under the **Platform** tab, go to **Security > Identity Providers**.
2. A list of the Identity Providers will be displayed. Click the three dots menu for the Identity Provider you wand to edit, and select **Security Level Rules**.



3. Select the **Security Level Name** and, if a rule is defined, it will appear in the Rule field. If not, you can create one. We copied the expression `containsAny ({security-zones}, 'PlantA', 'Floor1', 'Press Room')` from the example above for the following example.

4. After your enter your rule, click **Save Changes**.

Troubleshooting Identity Providers

This Troubleshooting section has a compilation of examples to help you diagnose and troubleshoot issues with configuring IdPs.

The Save Changes Button Is not Selectable

All required fields must be entered on the Identity Providers screen before Save Changes can be selected. Required fields have an asterisk (*) next to their name.

> **NOTE**
> If you are waiting for values for the ClientID and Client Secret fields, you can enter fake values and return when you have the correct value.

After Importing Metadata from a File, Values Did Not Auto Populate

1. Verify that import file is JSON format.
2. Verify that the import URL is valid.
3. Re-import the file.

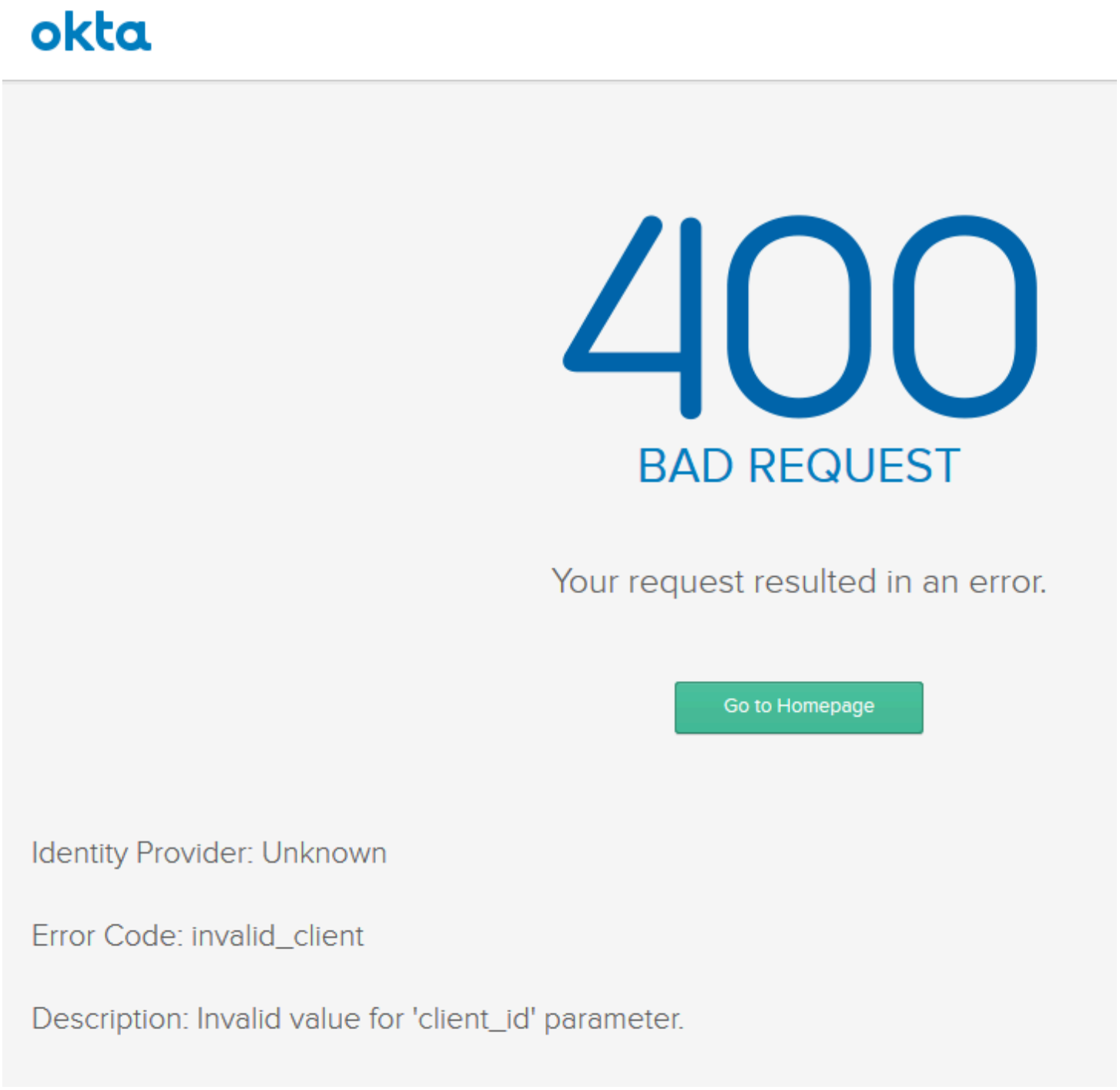
Login Testing

IdP Login Page Does not Appear

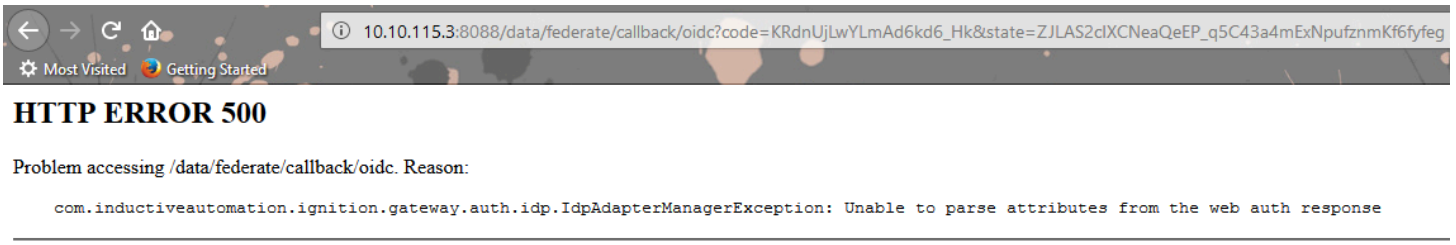
1. Confirm that the values in **Client ID** and **Client Secret** fields are correct. These come from the IdP when your Ignition Gateway is added as an application.
2. Check all other configuration settings.

The IdP Login is Displayed but the Login Attempt Fails

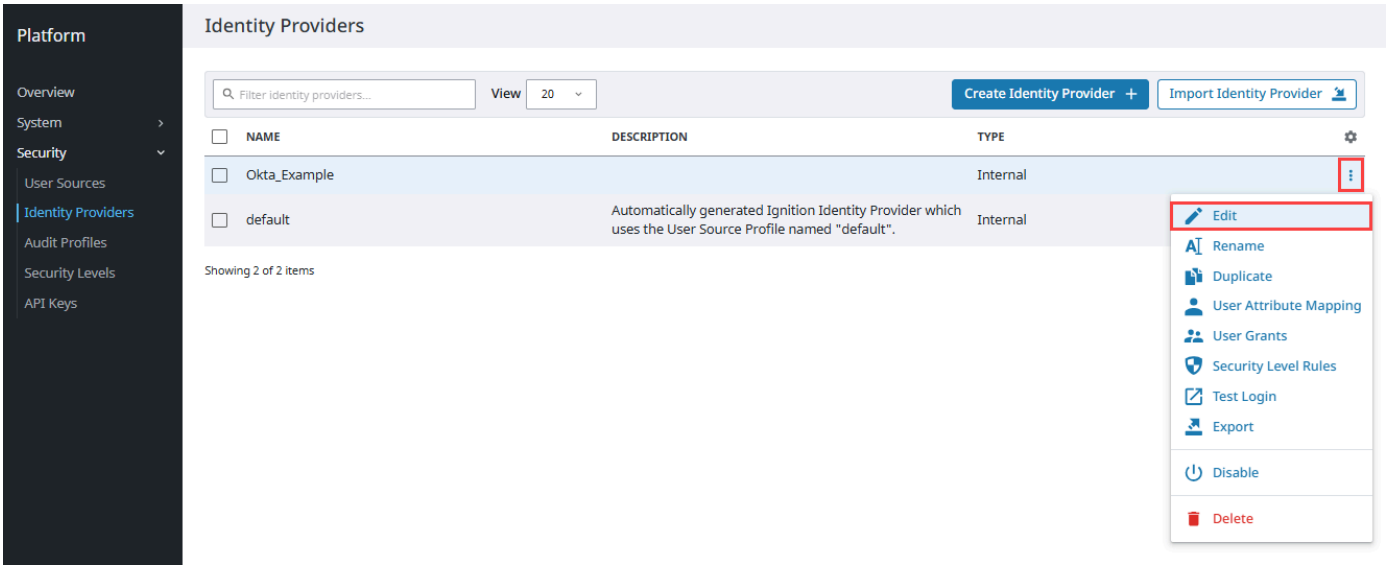
This issue is outside of Ignition. Check with your IT department and verify the login credentials (username and password) for your IdP.



The IdP Login Accepts the User but the IdP Redirect Fails (HTTP ERROR 500 for OpenID Connect Providers)



1. Go to Edit panel for you IdP. The Edit option is available under the IdP three dots menu.

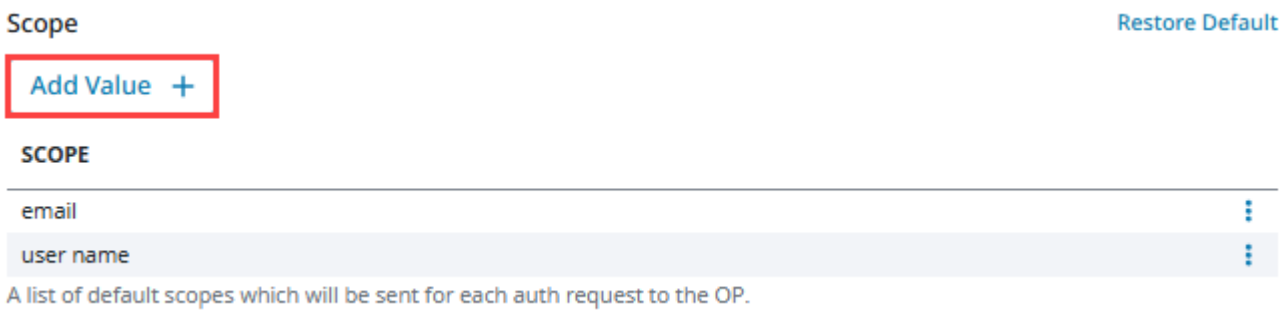


2. Verify the Provider Configuration setting for **Supported ID Token Signing Algorithm Values**.
3. If the URL for the IdP's metadata is available, try re-importing it.
4. Verify and re-enter the Client Id and Client Secret.
5. Verify the Token URL. Then re-import the IdP's metadata.
6. Verify the JSON web keys URL. (Default is to leave the check box checked.)
7. Re-import the IdP's metadata.
8. Verify the Issuer URL, then re-import the IdP's metadata.

The Test is Successful, but Results Do Not Show Useful Data (i.e., user name, email)

1. Go to Edit panel for you IdP. The Edit option is available under the IdP three dots menu.

2. Add the desired fields to the Scope section. You may have to reference the developers documentation Scope document.



3. Click Save Changes.

4. Repeat the Login test.

Revised User Attributes Are Not Shown in the Results of a Successful Test

1. Go to Edit panel for you IdP. The Edit option is available under the IdP three dots menu.

2. Add the desired fields to the Scope section.

3. Click Save Changes.

4. Repeat the Login test.

You Are Not Re-directed Back to Ignition after a Successful IdP Login

1. Verify the Authorization URL under Provider Configuration.

2. Re-import the IdP's metadata.

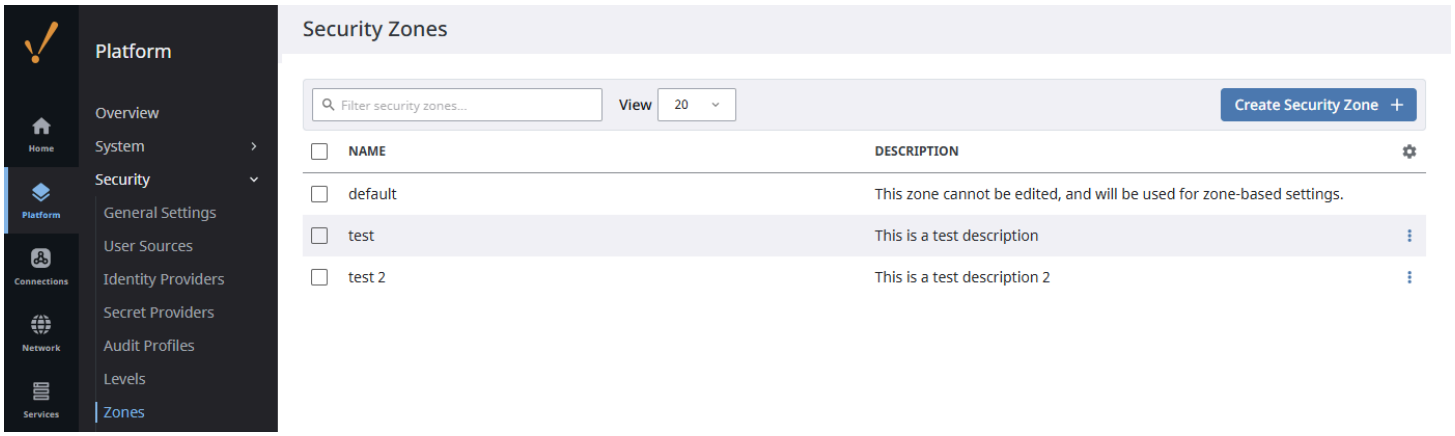
Security Zones

INDUCTIVE UNIVERSITY

Security Zones and Service Security

Watch the video

A Security Zone is a list of Gateways, Computers, or IP addresses that are defined and grouped together. This group now becomes a zone on the **Gateway Network**, which can have additional policies and restrictions placed on it. While Users and Roles restrict access to specific functions within the Gateway like making certain controls read-only for certain users and read/write for others, Security Zones provide this functionality to the Gateway Network, limiting locations instead of people to be read-only for specific actions. This allows for greater control over the type of information that is passing over the network, improving security and helping to keep different areas of the business separate, while still allowing them to interconnect.



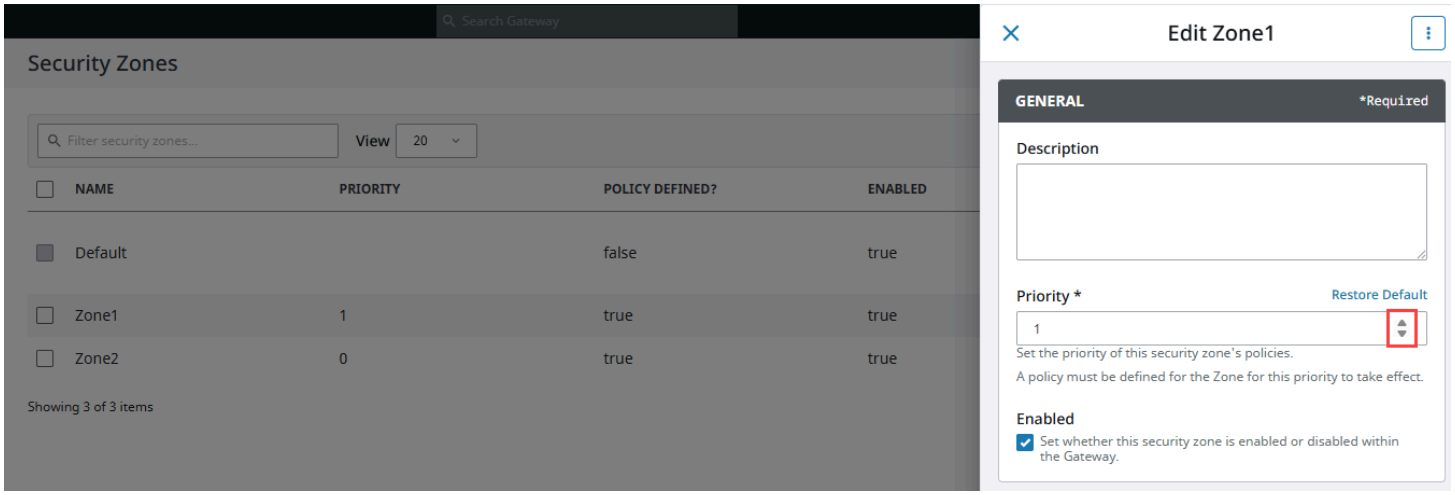
Using Security Zones

Sometimes, in addition to knowing who the user is, it is important to know their location. An operator may have permissions to turn on a machine from an HMI, but if the operator is logged into a project on a different Gateway in the network that has remote access to those tags, it might not be a good idea to let the operator write to those tags from a remote location. The operator can't see if the physical machine is clear to run.

This is where Security Zones come in. While Security Zones themselves don't define the security, they instead define an area of the Gateway Network, breaking up Gateways and network locations into manageable zones that can then have a policy set on them. Once there are zones defined, a policy can be defined for each zone, and a priority of zones can be set in the event that more than one zone

applies in a given situation. The priority of a Security Zone is set when it is created, but can be updated in the **Edit** panel. The policy from the highest priority Security Zone will be used.

For example, say there is a Zone1 and Zone2 configured. Zone1 includes specific Gateways, one of which is also contained in Zone2, that will have query and storage history access and read/write/edit access to tags. Zone2 dictates all requests coming from a range of IP addresses have query only history access, and read/write access to tags. We would want to make sure Zone1 is above Zone2 in priority, so that the Gateway in both Zone1 and Zone2 gets the full access rights afforded to it by the policy of Zone1, instead of getting the limited access rights from Zone2.



> **CAUTION**
>
> When using zone-based security in a project, the project stores the name of the Security Zone as a string. This means that if you were to modify the name of the zone in the Gateway, the zone-based security in your project will not update to reflect the new name, and instead will try searching for a zone with the original name. Be very careful when modifying the names of Security Zones.

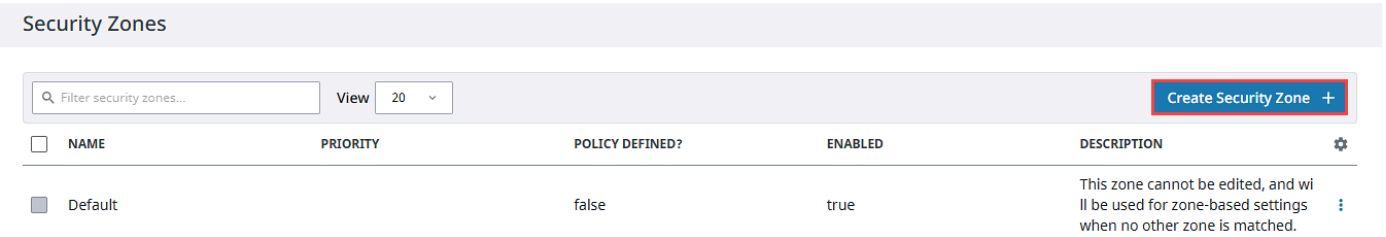
A connection must pass all of the qualifier checks before being accepted into a Security Zone. So if Require Secure Connection was checked, and Allow Client Scope was not, any requests coming from Clients would be rejected even if they are secure, and the same goes for any non-secure connections coming from sources other than a Client.

Requests can be a part of more than one zone, depending on how the zones are set up. This can be useful for making a whole section of IP addresses read only, but a specific Gateway in that IP address range may be listed specifically in another zone, which can be given read/write access. Any connection which does not fall into one of the zones will be placed in the Default Security Zone.

Defining a Security Zone

When setting up a new Security Zone, it is a good idea to set up a **Gateway Network** first if you haven't already. While Security Zones can be defined and used without a connected Gateway, they work best when used in conjunction with other Gateways on a Gateway Network. There is a special zone called Default. It is always present and can't be modified, and will be used if an incoming connection does not match any of the other defined zones. Identifiers are how incoming connections are distinguished between different zones. While there are different options to define the incoming connection, it only needs to match one of the Identifiers to match a zone. After first being identified as part of a particular Security Zone, the connection then checks the Qualifiers. With the Qualifiers, the incoming connection needs to fit in with all of the properties before it is fully placed into the Security Zone.

1. Under the **Platform** section of the Gateway Webpage, go to **Security > Security Zones**.
2. Select **Create Security Zone +**.



3. Enter a name and description for the new zone.
4. If there will be multiple Security Zones, you may want to adjust the **Priority** value. Increase the value for a higher priority, and decrease the value for a lower priority.
5. Under **Identifier**, enter an **IP Address** and a **Gateway Name**.
6. Under **Qualifiers**, select the **Require Secure Connection** option. Leave the remaining options at their default settings.
7. Click **Create New Security Zone**. The page will refresh and you will see a green banner stating that your new Security Zone was successfully created.

Settings

General

Setting	Description
Name	The name of the Security Zone.
Description	The description for the Security Zone.
Priority	The priority of the Security Zone's policies. A **policy** must be defined for the zone for this priority to take effect.
Enabled	Sets whether the Security Zone is enabled or disabled within the Gateway.

Identifier

Setting	Description
IP Addresses	This defines an IP address that the connection is coming from. This can be a list of IP addresses by using commas to separate them. It can also make use of the (*) wildcard like '192.168.100.*', or use a range such as '100.100.1-100.0-255'. With IP addresses, virtually all connections can be listed. Use 127.0.0.1 for the local connection.
Host Names	The host name refers to the system name of the machine generating the request such as Joe_Workstation. This can be a list of names separated by commas, and it can also use the (*) wildcard like '*_Workstation'.
Gateway Names	A list of Gateway system names that qualify for this zone. **Note**: When identifying a Gateway through a proxy Gateway, the IP Address should be using the IP of the proxy, but the Gateway name should use the name of the Gateway we are trying to identify.

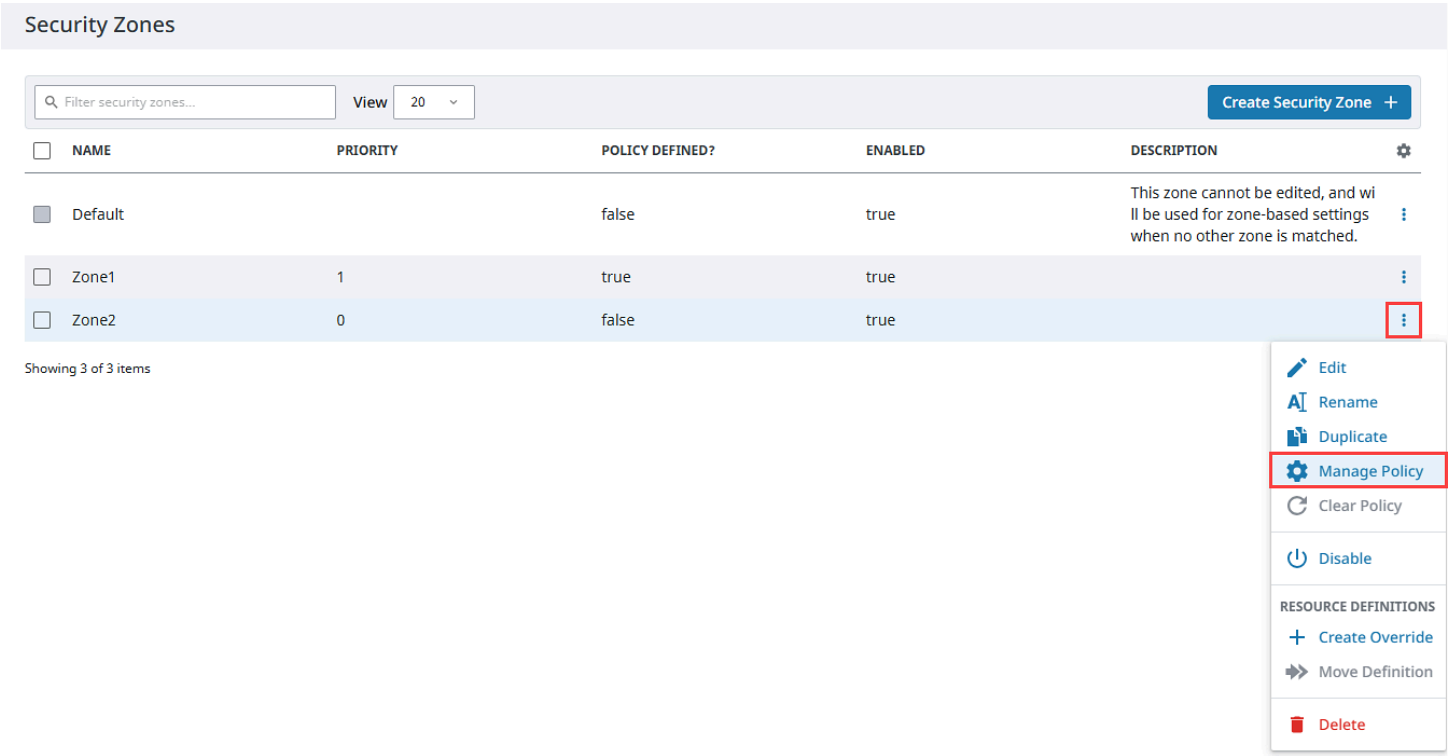
Qualifiers

Setting	Description
Require Secure Connection	If this is true, only connections that are made over a secure channel will be accepted.
Direct Connection Required	If this is true, only connections that come from a direct connection will be accepted. The Gateway Network allows you to connect three Gateways in a 1-2-3 configuration, where Gateway 1 can see Gateway 3 through the proxy Gateway 2.
Allow Client Scope	If this is false, any client scoped requests will not be accepted.
Allow Designer Scope	If this is false, any Designer scoped requests will not be accepted.
Allow Gateway Scope	If this is false, any Gateway scoped requests will not be accepted.

Service Security Policies

Policies can be defined for each Security Zone. **Security Zones** and their policies can be found within the Platform section of the Gateway under **Security** > **Zones**.

Before applying a policy, a Security Zone must exist. Once created, the Policy Defined column for the Security Zone will show as false until changes to a policy have been saved. Note that the **Default Security Zone** cannot be edited or have a policy applied. Expand the three dots menu for a Security Zone and select **Manage Policy** to define its policy.



The following services are available to define and work together to determine how the Gateway gives access to incoming Gateway connections:

- Alarm Journal Access
- Alarm Notification
- Alarm Status
- Audit Log Access
- Diagnostic Services
- EAM License Management
- History Provider Access
- Secret Provider Access
- Tag Access

> **NOTE**
>
> If you have a single Gateway, limiting access of certain Clients to certain tags must be done by configuring the individual **tag security** properties.

Policy Properties

All services have the ability to completely block access with the Service Access setting. Setting this to Deny will prevent zone access to that particular information, regardless of how the rest of the options are set.

Alarm Journal Access

Properties	Description
Service Access	Enables Alarm Journal access when set to Allow.
Default Profile Access	The default access rights for the Alarm Journal service.
Access Level	By default this setting will be **Inherited**, which will cause this specific Alarm Journal to inherit the access rights set in the Default Provider Access Level. It can also be set to **No Access** to block query and storage to this specific Alarm Journal. Setting it to **Query Only** will allow users to only query data from this Alarm Journal without any storing capability. The **Query and Storage** option allows users to store and query data from this Alarm Journal. It is important to note that every time a new Alarm Journal is created in the local Gateway, a new setting for this journal will be added to this Service Policy.

Alarm Notification

Property	Description
Service Access	Enables Alarm Notification access when set to Allow.
Access Pipeline Filter	A list of Pipelines in the current Gateway that other connections can use for alarm notification. Pipelines must be entered in the format "project_name/pipeline_name". The list is a comma separated list, and it can make use of the (*) wildcard. This setting is an inclusionary list not an exclusionary list, meaning that if there are no pipelines listed here, then all of them will be available.

Alarm Status

Property	Description
Service Access	Enables Alarm Status access when set to Allow.
Allow Acknowledge	Allows Gateways that fall within the zone to acknowledge alarms on the local Gateway.
Allow Shelving	Allow Gateways that fall within the zone to shelve alarms on the local Gateway. For this Gateway to shelve alarms on other Gateways, this must be checked on every remote Gateway.

Audit Log Access

Property	Description
Service Access	Enables Audit Log access when set to Allow.
Default Profile Access	The default access rights for the Audit Profile service.
Access Level	By default this setting will be set to **Inherited**, which will cause this specific Audit Profile to inherit the access rights set in the Default Provider Access Level. It can also be set to **No Access** to block query and storage to this specific Audit Profile. Setting it to **Query Only** will allow users to only query data from this Audit Profile without any storage ability. The **Query and Storage** option allows users to store and query data from this Audit Profile. Just like with Alarm Journals, it is important to note that every time a new Audit Profile is created in the local Gateway, a new setting for this profile.

Diagnostic Services

Property	Description
Service Access	Enables Diagnostic Services access when set to Allow.
Gateway Network Crawler	If checked, the Gateway will allow a remote Gateway to "crawl" the Gateway Network routes and return diagnostic information about the Gateways along the routes. Returned information includes platform version and edition, and connectivity statistics (such as ping time).

EAM License Management

Property	Description
Service Access	Enables EAM License access when set to Allow.
Agent Leased License Management	Applies to EAM Controller only. If checked, an EAM Agent can forward leased license activations and renewals to this Gateway. This Gateway will then contact the licensing server and send the results back to the agent Gateway.

History Provider Access

Property	Description
Service Access	Enables History Provider access when set to Allow.
Default Profile Access	The default access rights for Tag History. It is not recommended to set the Default Access Profile to Inherited since all new History Providers will automatically get added to the Service Policy with an Access Level setting of inherited. Instead, it may be beneficial to set the Default Profile Access to be either Read Only or No Access so that an added History Provider does not accidentally get storage rights.
Access Level	By default this setting will be set to **Inherited**, which will inherit the Default Profile Access rights. If set to **Query and Storage**, connections in the current zone can both run queries and store against the Tag History Provider. **Query Only** will only allow the zone to query out history data, but not store it. **No Access** will completely block access to the History Provider.

Secret Provider Access

New in 8.3.3

Property	Description
Service Access	Enables Secret Provider access when set to Allow.
Default Provider Access Level	The default access rights for Secret Providers.
Access Level	This setting will appear for each Secret Provider configured, listed by name. By default, it is set to **Inherited**, which will inherit the Default Provider Access Level value.

Tag Access

Property	Description
Service Access	Enables Tag Provider access when set to Allow.
Default Profile Access Level	Sets the default access rights for realtime Tag Providers. It is not recommended to set the Default Access Profile to Inherited.
Trust Remote Security Levels	Allows users to opt into trusting the Security Levels of remote Gateway users when remote Gateways read, write, and subscribe to local tags. If checked, Security Levels passed from the remote Gateway will be used for determining access to tags on the local Gateway. If unchecked, the remote Gateway's Security Zones and the impersonation role will be used as the Security Levels.
Impersonation Role Name	Allows you to specify a role name to use when writing to a tag from an incoming Gateway Network connection (from the selected zone).
Access Level	An Access Level setting will exist for each Tag Provider configured in the local Gateway, as well as an additional one for System tags. By default the level is set to **Inherited**, which will inherit the access rights set in the Default Provider Access Level. Other level settings include **ReadWriteEdit**, which will allow connections in the current zone to read, write to, and edit the tags in that provider; **ReadWrite**, which allows the zone to read and write to tags; and **ReadOnly**, which allows the zone to only read the tags. It also can be set to **None**, which will prevent the zone from interacting with the Tag Provider altogether.

Default Security Zone

While the Default Security Zone may not have a custom policy defined, it is configured to allow Alarm Acknowledgment, query only for History Provider Access, read only for Tag Access, and not to include Notification Pipelines. This means that if a remote Tag Provider is set up on a remote Gateway, and the local Gateway has not changed the default service settings, the remote Gateway will have read only access to the Tag History Provider. This can be changed by overriding the Default Security Zone or by creating new Security Zones with custom policies. Once a policy has been defined on a zone, it will automatically take precedence over the default policy.

Project Security in the Designer

When several users are all working on the same project, managing changes to the project can become cumbersome. By default, all users with Designer access can modify, delete, and save all resources available in the Designer. In some situations, it is desirable to limit what each user can do in the Designer. Ignition has several built-in Designer restriction methods to help in these scenarios.

Designer Project Permissions

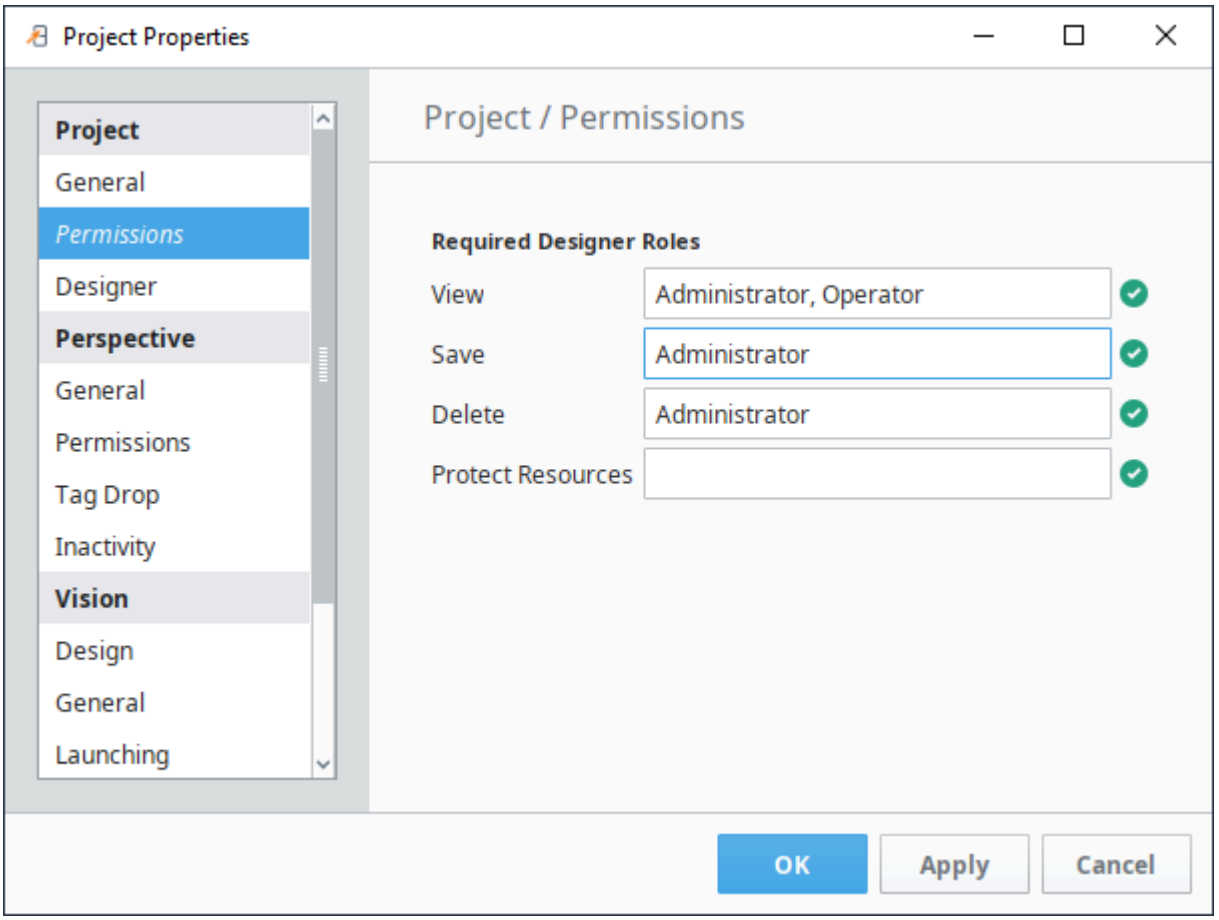
Actions such as, viewing, saving, deleting, and editing of project resources are restricted to users who have sufficient roles to do so. Editing of the these required roles is done in the permissions section of the Project Properties dialog in the Designer. If required roles are not set for an action, then all users with Designer access can perform the action.

The Designer does not poll for role changes, so if a user who is currently logged into the Designer has their roles changed, they will need to re-launch the Designer for the new role(s) to take effect.

Controlling Project Edits by Role

You can control who gets to login to a project by assigning roles and giving permissions to those roles in the Required Designer Roles property which you set up in the Designer.

1. In the Designer, from the menubar, choose **Project > Properties**.
2. Go to the **Project > Permissions** area.
3. Under the Required Designer Roles, enter the appropriate roles next to each project-level restriction, as required. You can enter a comma-separated list of role names that are required to access the project. As you start typing, matching role names will pop up.



4. Click **OK** to save the changes.

   The following table describes each of these four options.

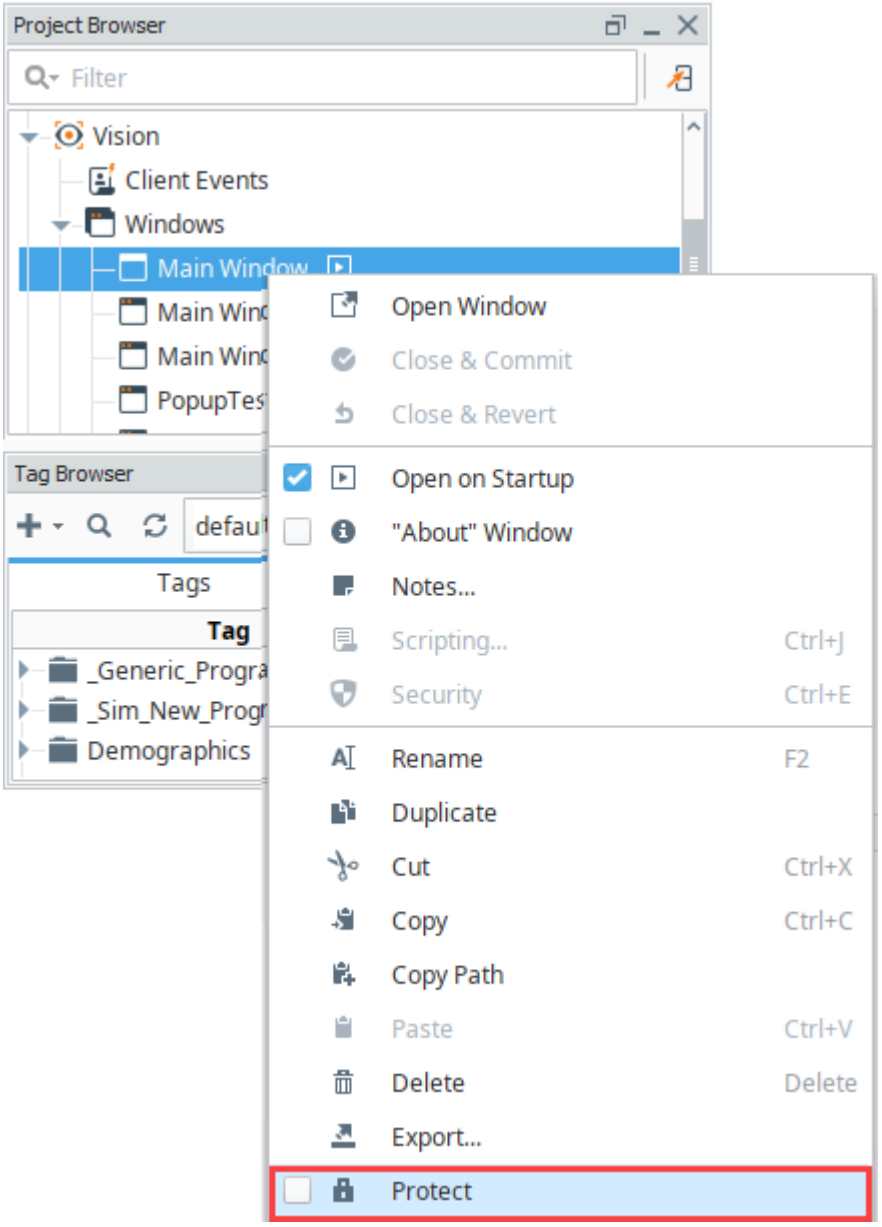
Option	Result
View	User must have at least one of these roles to view the project in the Designer.
Save	User must have at least one of these roles to save the project.
Delete	User must have at least one of these roles to delete the project.
Protect Resources	User must have at least one of these roles to access protected resources.

Protecting Project Resources

Locking Project Resources

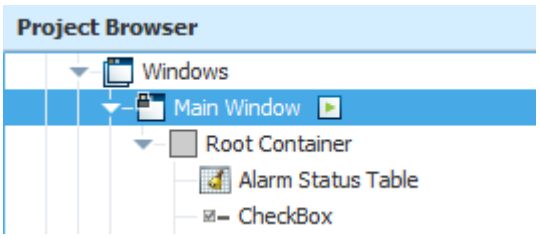
Watch the video

You can lock individual project resources from inside Designer by opening the Project Browser, and right-clicking on any of the objects that you want to lock in. Select the **Protect** option to protect it. Once it's protected, it cannot be changed except by someone who has the permission to unprotect and modify it.

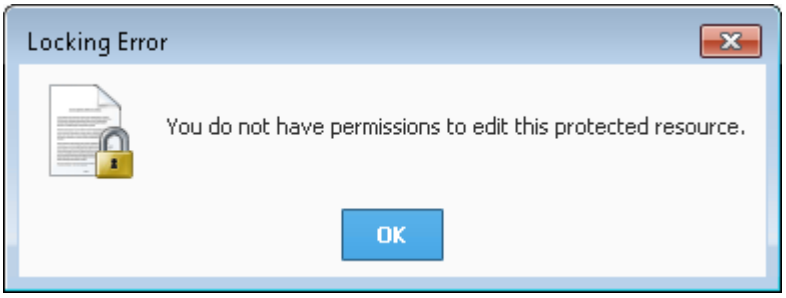


Protected resources are global or project resources that can only be edited by select users with the required roles. These roles protect resources from being edited in the Designer by other users who have Designer access, and do not apply to the Clients. It is often used in scenarios where development work is finished on a window or object, and no further changes should be made to it. Other objects like **Vision Templates** or **Alarm Pipelines** are often protected so they may be used, but not modified.

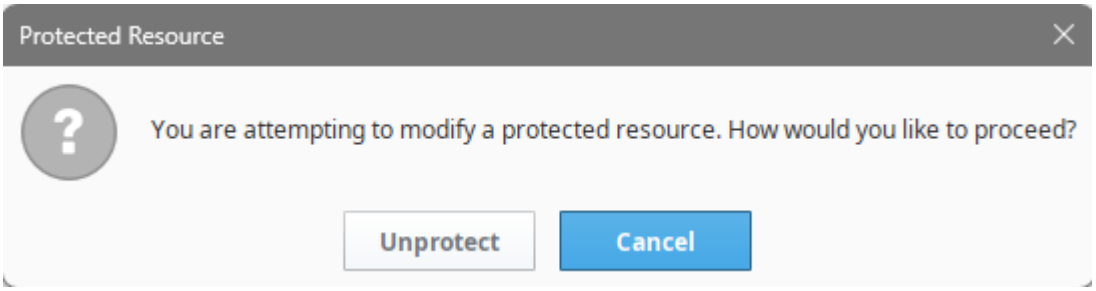
A Lock icon will appear on the resource in the Project Browser informing users that it is protected.



Users without a required role will see a Locking Error message in the Designer when attempting to open a protected resource.



Users with a required role are allowed to modify the resource, but a message will appear informing them that the resource is protected, and will be asked how to proceed.



Select **Unprotect** to remove the protection, or right-click the object in the Project Browser at any time and select the **Protect** option to unprotect it.

Security Certificates

A security certificate, also known as a digital certificate, is used to provide trust between connections. Trusted certificates establish the identity, authenticity and reliability of incoming and outgoing traffic.

The Ignition platform uses Secure Sockets Layer/Transport Layer Security (SSL/TLS), which requires certificates in multiple features. The purpose and type of certificate determine how the certificate is installed and where it is stored within Ignition. It is important to know where certificates are needed and what their purpose is to make sure all requirements are met. For example, the Gateway Network and OPC UA security both impact client and server connections, but because the Gateway Network connections are between local and remote gateways and OPC UA connections are between devices, the process for adding and trusting certificates is different.

The following is a list of locations where certificates are required, and a link pointing to a page containing the general security purpose, settings, and certificate management properties.

- **Launchers and Clients**
- Gateway:
  - **Acting as a server**
  - Acting as a client, see **Adding Security Certificates into KeyStores** below.
- **GAN Security**
- **OPC UA Security**

Types of Certificates

It may be helpful to understand the different types of certificate Ignition can use if you are new to certificates.

SSL Certificates

SSL certificates allow systems to verify identity and establish an encrypted network connection to another system using SSL/TLS protocols. There are two types of signed SSL Certificates, self-signed certificates and trusted certificate authority (CA) certificates.

- Self signed certificates are generated internally for free.

- Trusted CA certificates are signed by a trusted certificate authority
    - Ignition supports CA certificates from your organization's internal CA or any publicly trusted certificate authority.

Both signed certificates offer encryption, but without the signature of a trusted certificate authority, warning messages will appear for self-signed certificates that are not trusted.

Since SSL/TLS requires the installation of a security certificate, both the Gateway Network and the Web Server can use self-signed certificates if CA certificates are not yet available or needed, such as during testing. It is important to note that although the functionality of certificates installed on the Gateway Network and Gateway Web Server are similar, they must be treated separately because settings made on one page on the Gateway do not apply to the other, even the case of shared ports.

OPC UA Certificates

UA security contains authentication and authorization as well as encryption and data integrity by signing. Security is integral to UA and OPC UA protocols are a hybrid variant of TLS, using binary encoding and HTTPS for transport. The Ignition platform inherently offers OPC UA client functionality and the Gateway can connect to any compliant OPC UA server.

Adding Security Certificates into KeyStores

In some cases when the Gateway is acting as a client, you may need to provide supplemental security certificates so the Gateway can communicate with other systems, such as databases or devices elsewhere on the network. These supplemental certificates can be added to a Gateway by simply placing them in the following directory on the Gateway's file system:

```
%gateway installation directory%data/certificates/supplemental
```

Once added, you will need to restart the Gateway before the certificates will be used.

Supported formats are DER encoded binary X.509, and Base-64 encoded X.509 (PEM-encoded ASCII).

OAuth 2.0 Clients

What is OAuth 2.0?

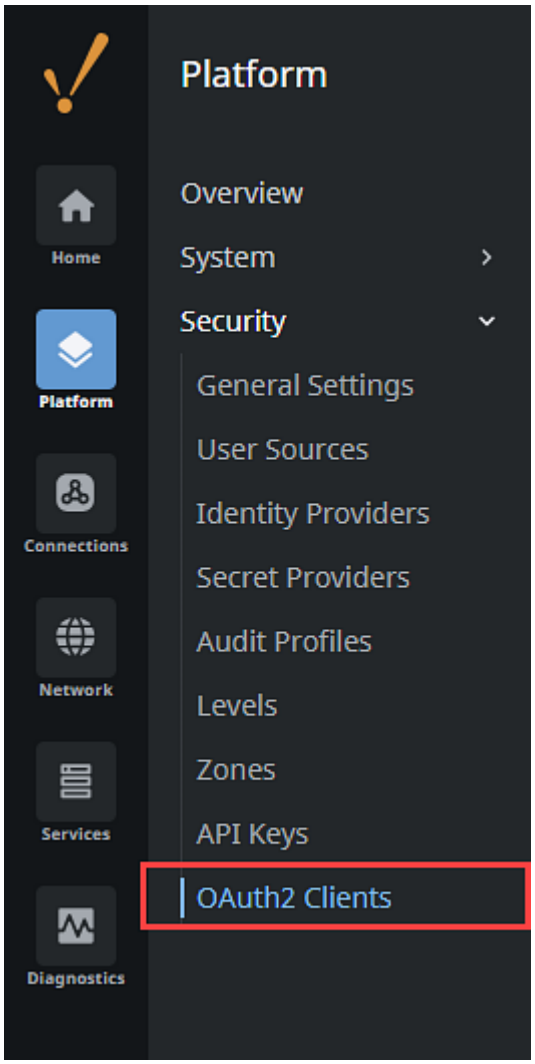
OAuth 2.0 is an authorization standard that is developed by **IETF OAuth Working Group**. Unlike basic authorization which requires a user to input a **fixed password** to log in, OAuth 2.0 requires users to input an **access token** instead. This makes using OAuth 2.0 more secure than basic authorization, since the access token is:

- Unique to the client and the requested scope of access
- Difficult to guess
- Constantly changing

Some well-known authorization protocols, such as OpenID Connect, are built on the OAuth 2.0 standard. See the **official OAuth 2.0 documentation** for more information.

Configuring OAuth2 Clients

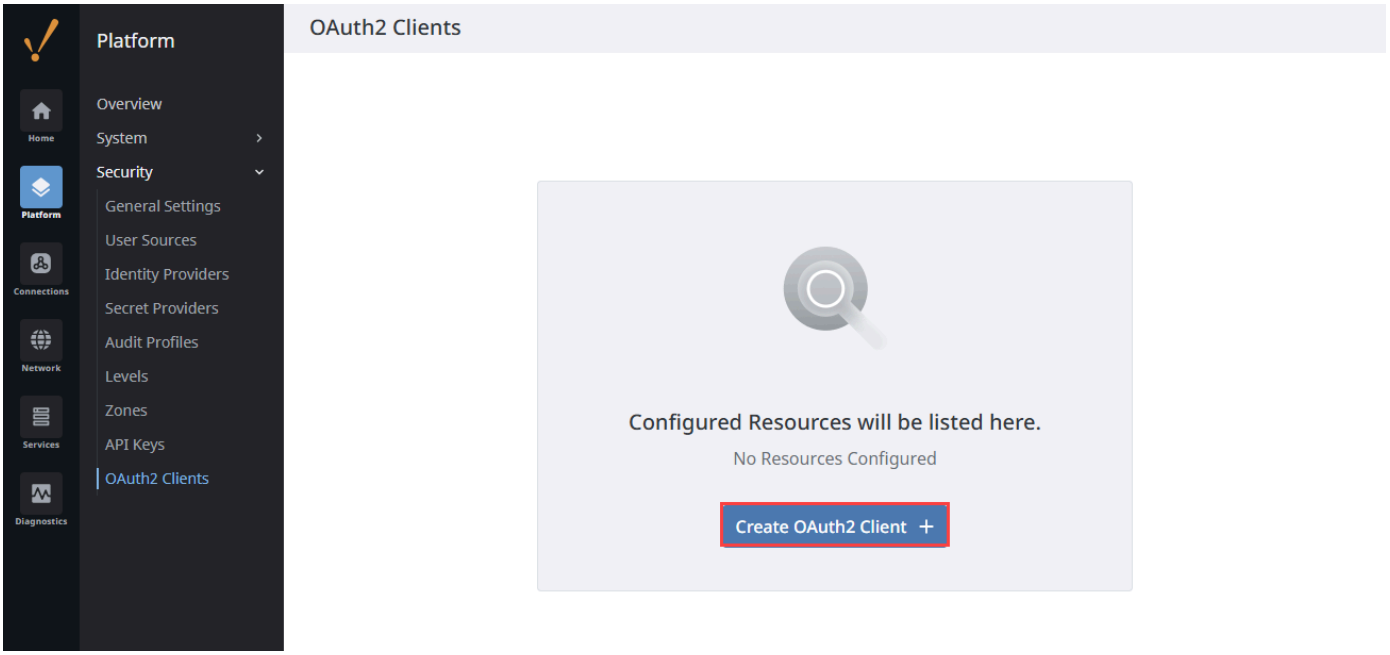
OAuth2 Clients are used by OAuth2 Email Profiles. Additionally, multiple OAuth2 Email Profiles can use the same OAuth2 Client. When configuring an **OAuth2 SMTP Email Profile**, you can reference the name of your OAuth2 Client to use those properties. To set up or modify OAuth2 Clients, navigate to the OAuth2 Clients section on the Gateway. You can find this section by going to **Platform > Security > OAuth2 Clients**.



Creating a new OAuth2 Client

The OAuth2 Clients page is where you can select an existing OAuth2 Client to modify, or create a new OAuth2 Client.

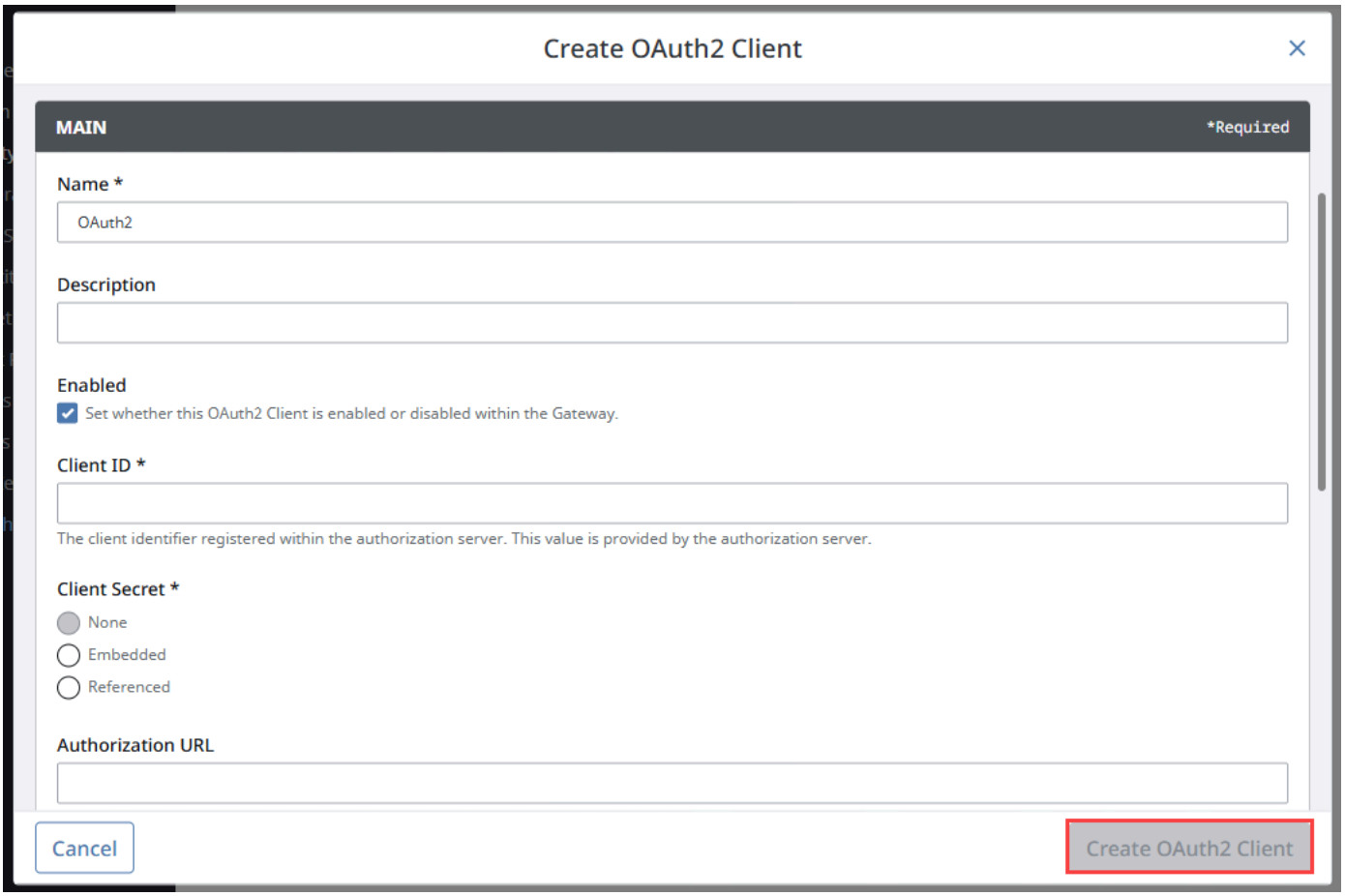
1. Select **Create OAuth2 Client +**.



2. From here, you will be able to configure your OAuth2 Clients by specifying various properties. Fill out the fields on this page.

> **CLIENT ID AND CLIENT SECRET VALUES**
>
> A Ignition generated URI redirect address for your Ignition server is listed in the information box at the top of the page. In our example it is `<http://localhost:8088/data/oauth2/client/{name}/authorize/callback>`.
>
> > Most OAuth 2.0 Authorization Servers will require registering Ignition as a client. After the registration process is complete, the authorization server will generate a client ID and secret for Ignition, which is required below. This gives Ignition the ability to communicate securely with the authorization server. Most authorization servers will also require a set of redirect URIs. The redirect URI for this Ignition Gateway is: **http://localhost:8088/data/oauth2/client/{name}/authorize/callback** (where **{name}** is the name of the OAuth2 client)
>
> You need to provide this URI to your IdP (usually this means giving it to your IT department) to add your Ignition server as an application to the IdP. Once done, they can provide you with a **Client ID** and **Client Secret**. The Client ID and Client Secret fields are needed for the Ignition Gateway to properly communicate with the IdP.
>
> If you don't know them yet, you can put in bogus values for now and edit them later once the correct values are provided to you.

3. Click **Create OAuth2 Client** once all required fields are filled.



Your OAuth 2.0 Client is now set up. You can now verify your token, authorization, or begin using the OAuth 2.0 Client.

OAuth2 Client Properties

The tables below list the available properties to configure and edit for OAuth2 Clients.

Main

Property Name	Property Description
Name	The name of your OAuth2 Client. This field is important, as redirect URIs will use the name of the OAuth2 Client.
Description	A description of your OAuth2 Client. (Optional)
Enabled	Sets whether this OAuth2 Client is enabled or disabled within the Gateway. Default is enabled.
Client ID	The client identifier registered within the authorization server. This value is provided by the authorization server.
Client Secret	The client secret registered within the authorization server. This value is provided by the authorization server. You can add it directly as an Embedded or select it from the Referenced dropdown if you previously stored it in a Secret Provider.
Authorization URL	URL of the authorization server's OAuth 2.0 Authorization Endpoint. (Optional unless using interactive mode or authorization code flow)
Token URL	URL of the authorization server's OAuth 2.0 Token Endpoint.

HTTP Client Settings

Property Name	Property Description
HTTP Version	The maximum HTTP version supported by the authorization server's token endpoint. The Gateway's HTTP client will use this version when making HTTP requests to the authorization server. Choose between `HTTP/1.1` or `HTTP/2`. Default is `HTTP/2`.
HTTP Connect Timeout	In the case where the Gateway's HTTP client needs to establish a new HTTP connection with the authorization server, if the connection cannot be established within the given amount of time in milliseconds, then the connection attempt and any pending request requiring the connection will time out. There will be no timeout if the value is less than or equal to zero, so the Gateway will wait forever for the connection to be made. Default is 30,000.
HTTP Request Timeout	The timeout in milliseconds for each HTTP request sent from the Gateway's HTTP client to the authorization server. If an HTTP response is not received within the specified timeout after sending an HTTP request then the HTTP request will time out. There will be no timeout if the value is less than or equal to zero, so the Gateway will wait forever for a response. Default is 120,000.

OAuth2 Client Options

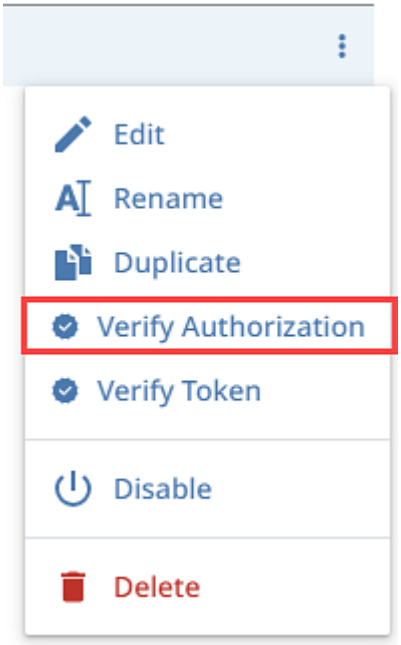
After you set up your OAuth2 Client, you will have the option to edit your properties, disable or delete the client, verify authorization, or verify your token.

Verify Authorization

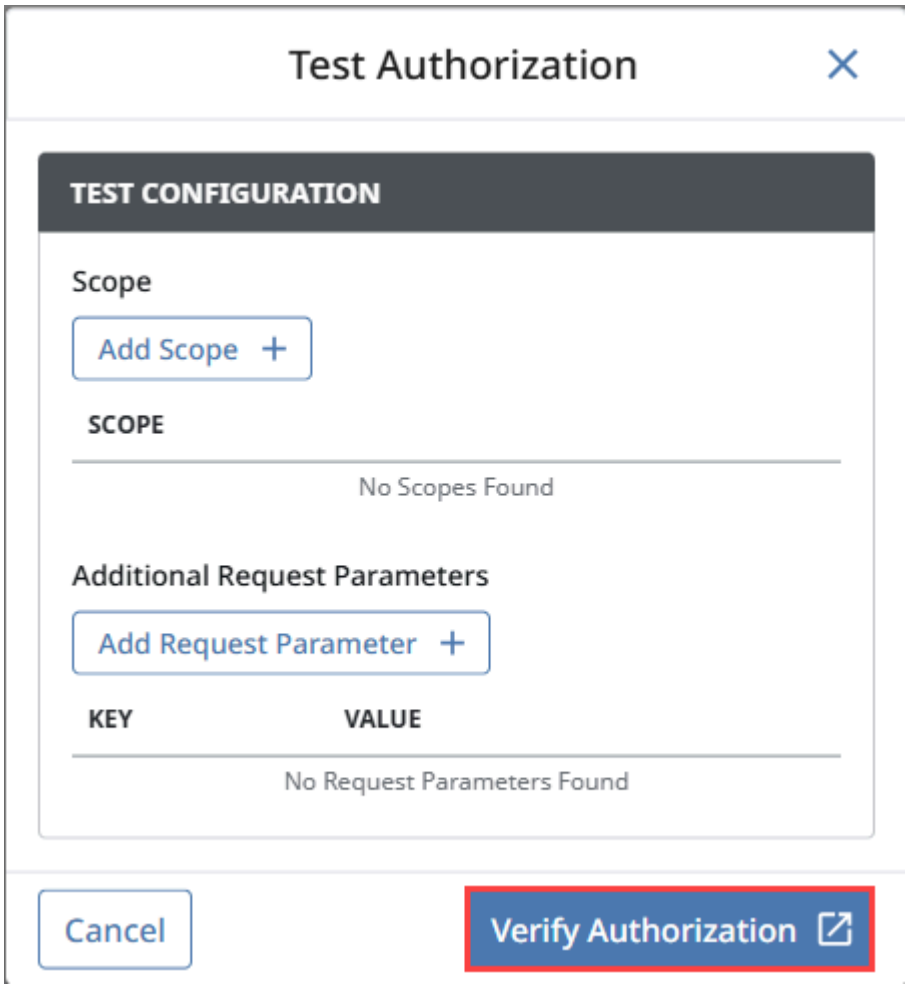
> **NOTE**
>
> The **Verify Authorization** option will be unavailable if the optional **Authorization URL** setting is blank on the OAuth2 Client's configuration settings.

The **Test Authorization** form allows you to test the integration between the OAuth2 Client configured in Ignition with the OAuth2 Authorization Server. More specifically, it allows you to walk through a test run of the **Authorization Code Grant** flow. During this process, the authorization server will validate

the end user and may prompt the end user to give the Client permissions to perform actions on their behalf, such as sending emails. Once these permissions have been granted, the authorization server will generate an authorization code, which the Client can use to obtain an access token. To see an example of this process work alongside **verifying tokens**, go to the **Verify Authorization and Token Example**.



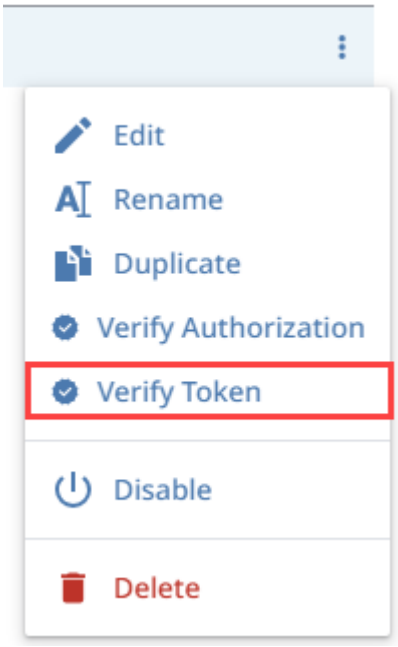
When verifying authorization, you will need to add any **scopes** you want to test with, along with any additional **Request Parameters** you want to try. You can find Request Parameters in your authorization server's documentation. Once those fields are filled out, click on **Verify Authorization**.



Verify Token

The **Verify Token** option is useful in cases where you want to test the **authorization code** to make sure it successfully gives the Client an access token. It is also useful if you want to test an access token request using a **Client Credentials** grant, or refreshing an access token using a **refresh token**. During this process, the Gateway will make an access token request against the authorization server using one of three methods and get a response back. To see an example of this process work alongside **verifying authorization**, see the **Verify Authorization and Token Example**.

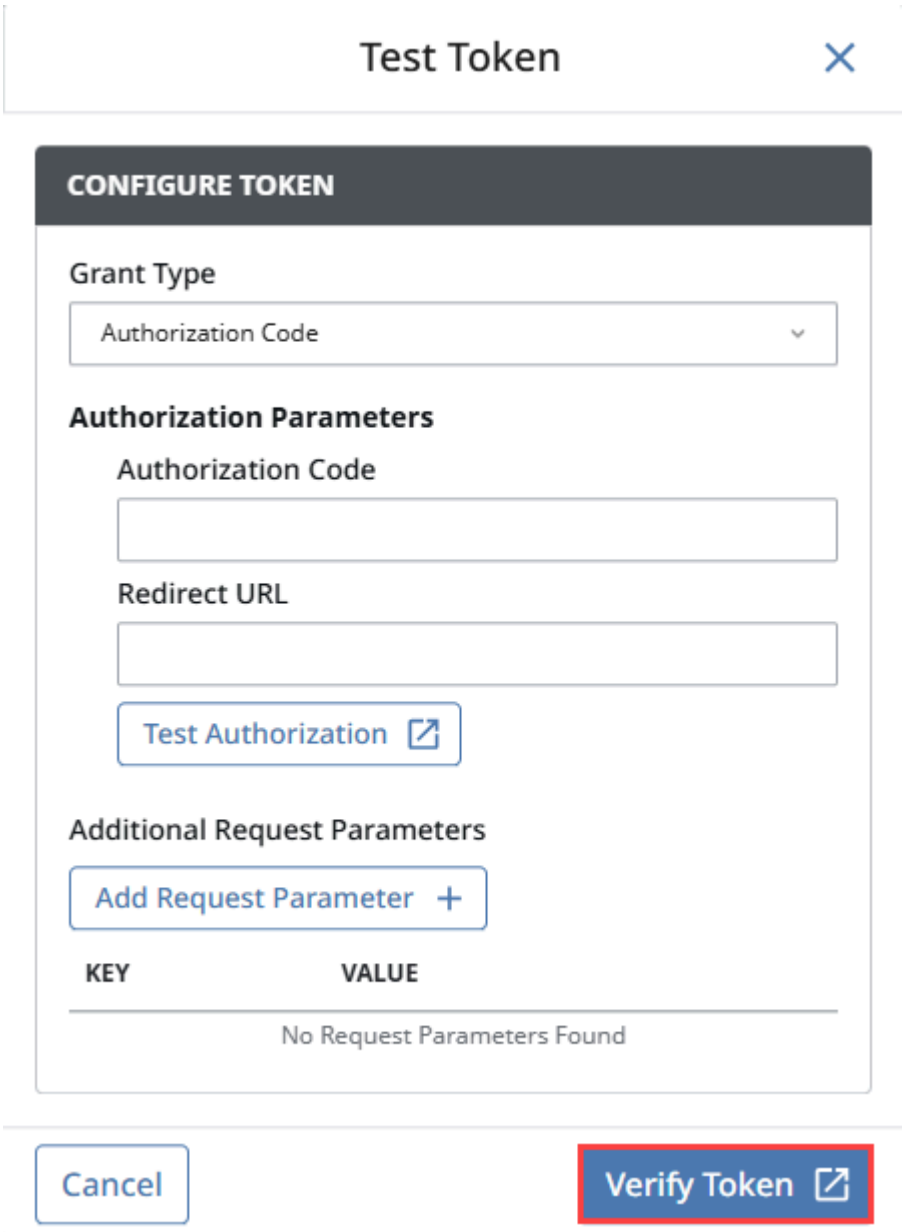
You can find the **Verify Token** option on the OAuth2 Clients page by expanding the three dots menu on the far right of the client list.



The section below details the **Authorization Code**, **Client Credentials**, and **Refresh Token** grant types. Use the Grant Type dropdown to select between the types and see the different required information for each test.

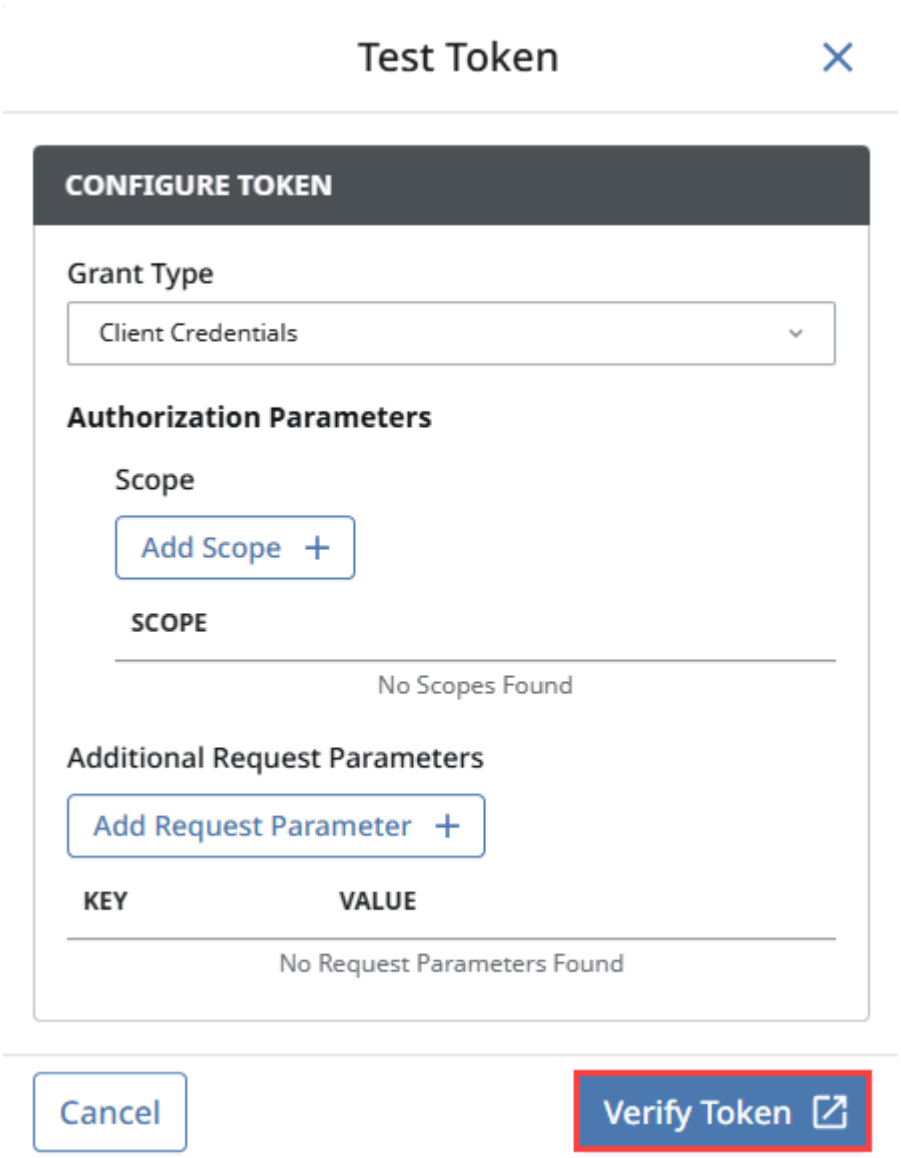
Authorization Code

If you use an Authorization Code, you will need to enter an **Authorization Code**, **Redirect URL**, and any additional **Request Parameters**. Once you input that information, click **Verify Token**.



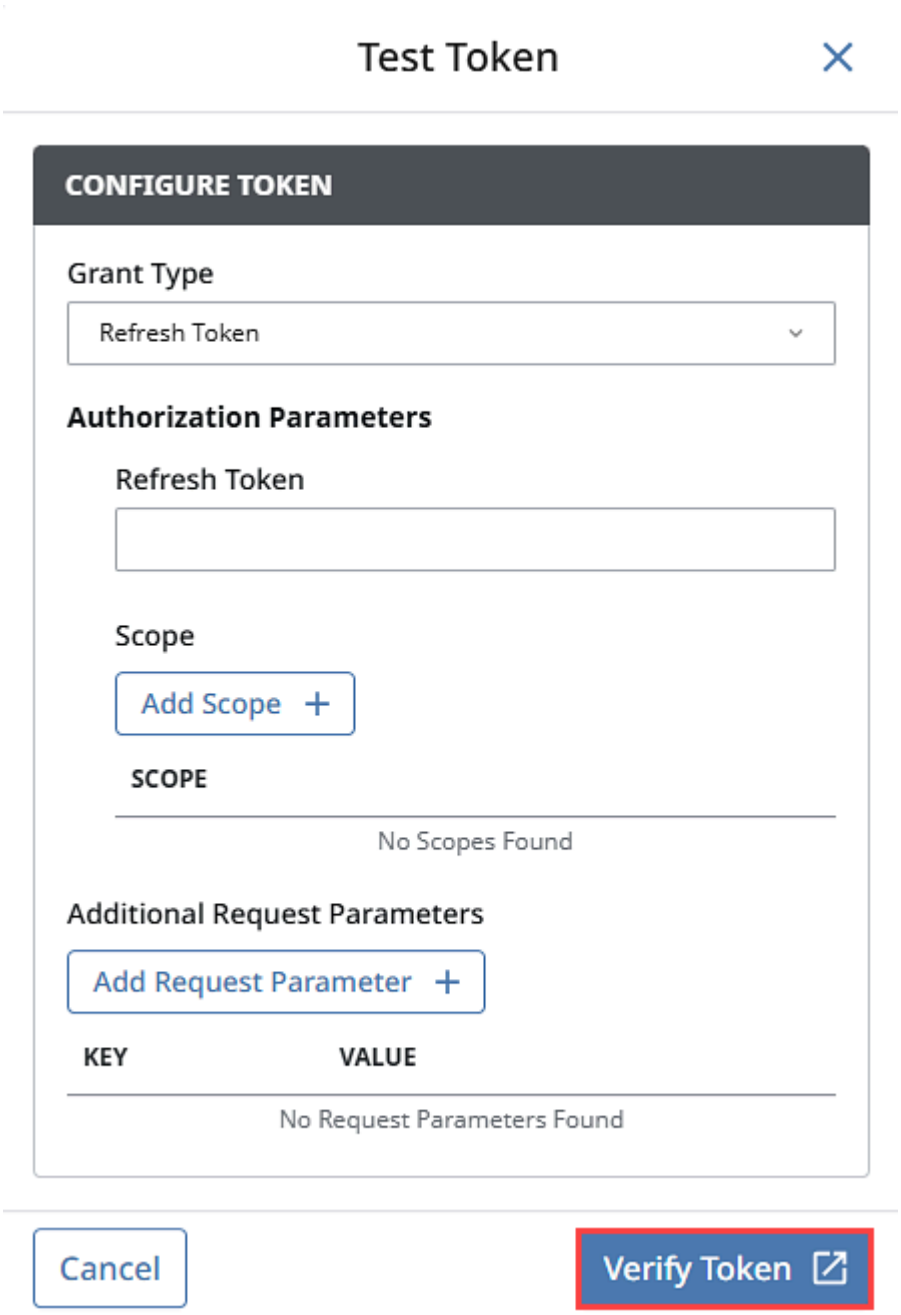
Client Credentials

If you use Client Credentials, you will need to enter a **Scope** and any additional **Request Parameters**. Once you input that information, click **Verify Token**.



Refresh Token

If you use a Refresh Token, you will need to enter the **Refresh Token**, **Scope**, and any **Additional Request Parameters**. Once you input that information, click **Verify Token**.



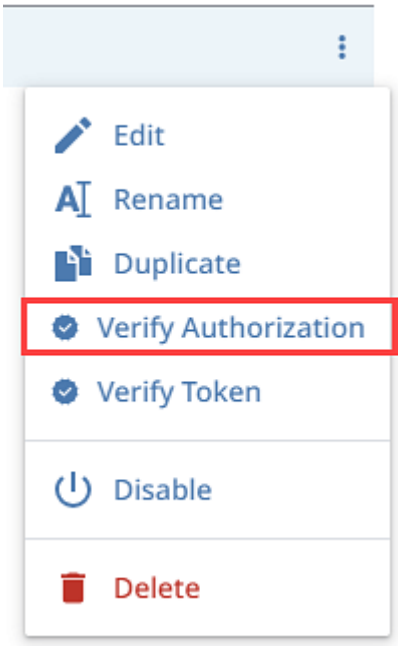
Verify Authorization and Token Example (Google Identity OAuth2)

This example walks you through a verify authorization and verify token workflow for an OAuth2 Client configured against **Google's OAuth2 Authorization Server**.

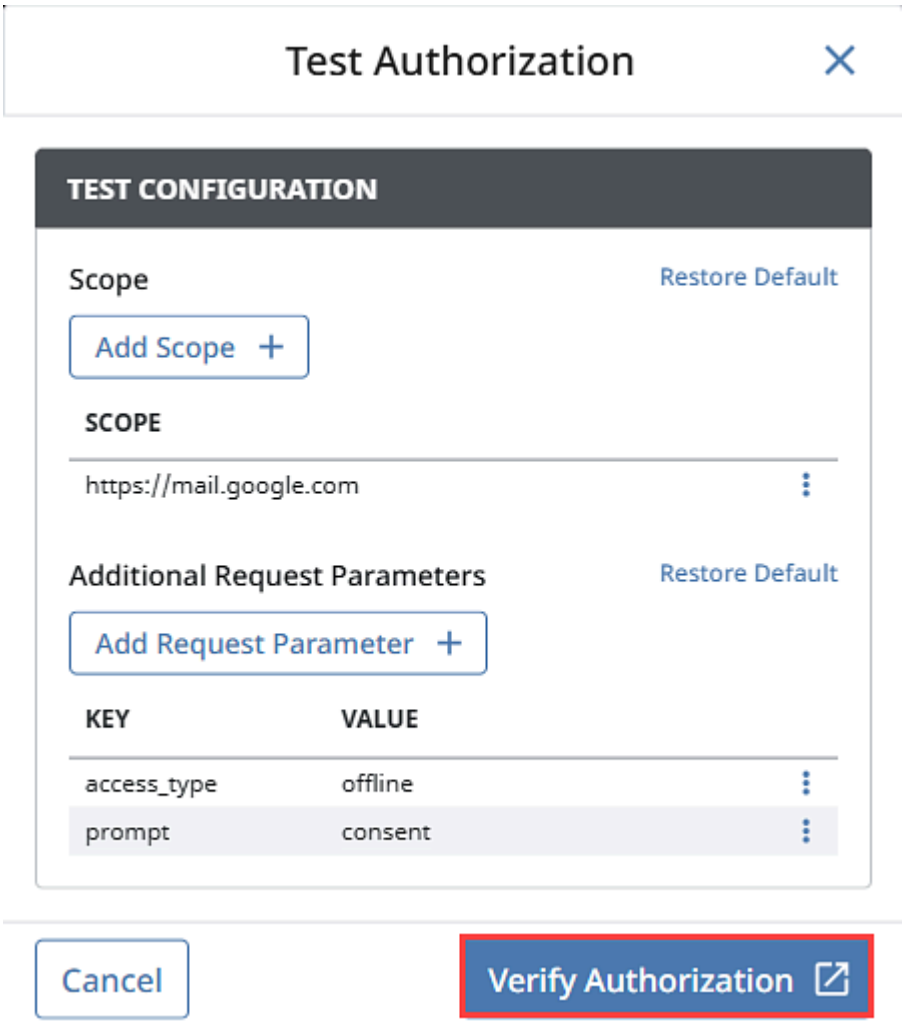
1. Go to your OAuth2 Clients page on the Gateway. This is located under **Platform > Security > OAuth2 Clients**.

2. Set up your OAuth2 Client if you haven't already. For this example, we are using the OAuth2 Client we set up earlier called **OAuth2**.

3. Expand the three dots menu and select **Verify Authorization**. This will bring up the Test Authorization form.



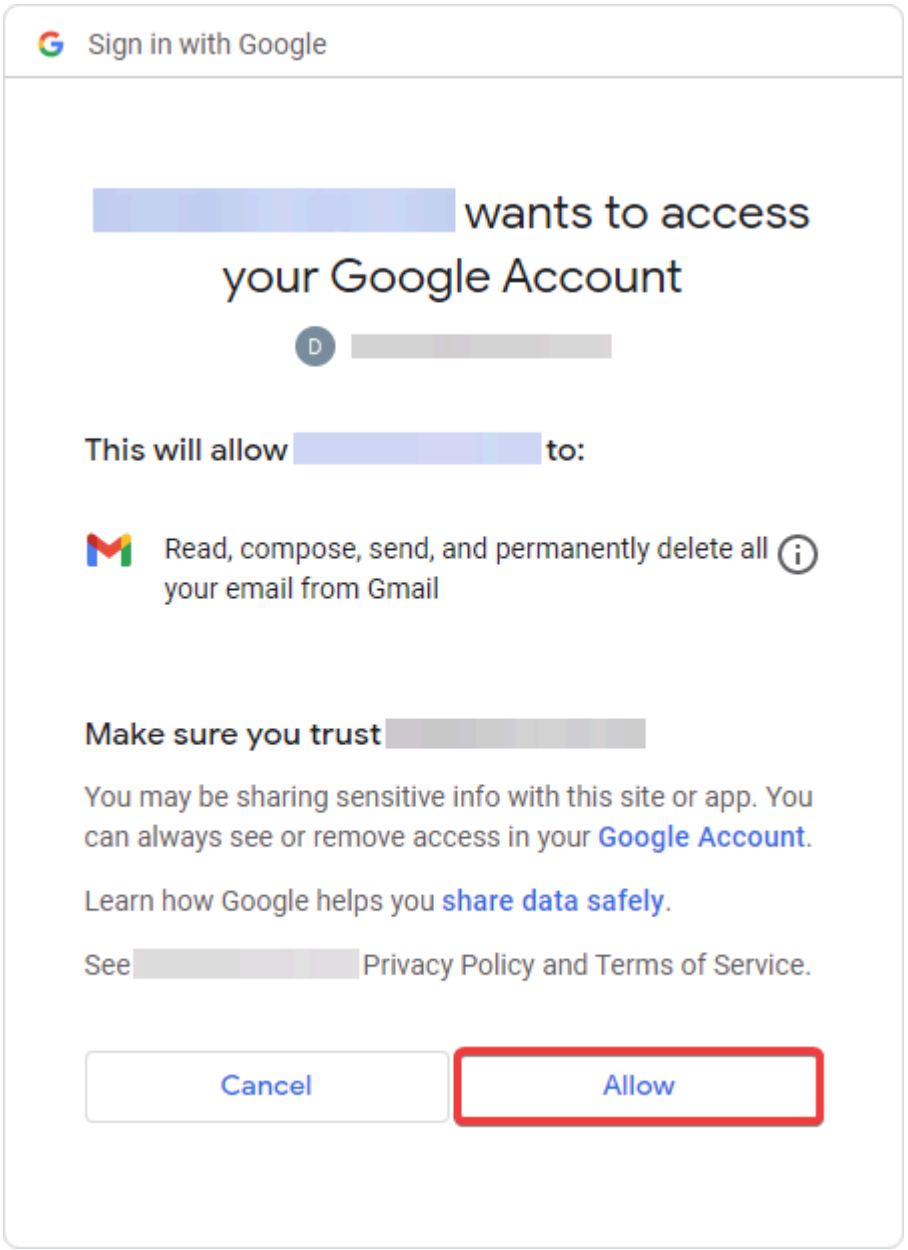
4. Add any Scopes and Additional Request Parameters for your OAuth2 Client, and then click **Verify Authorization**.



> **NOTE**
>
> The Additional Request Parameters in the example are specific to Google's Authorization Server. They do two things:

- access_type: Setting this parameter to `offline` will allow your Client to refresh access tokens while the user is not present at the browser.
- prompt: Setting this parameter to `consent` will force the Client to prompt the user for consent.

5. Since we are using Google Identity, we will be redirected, and a Google permission prompt will appear. Click **Allow**.



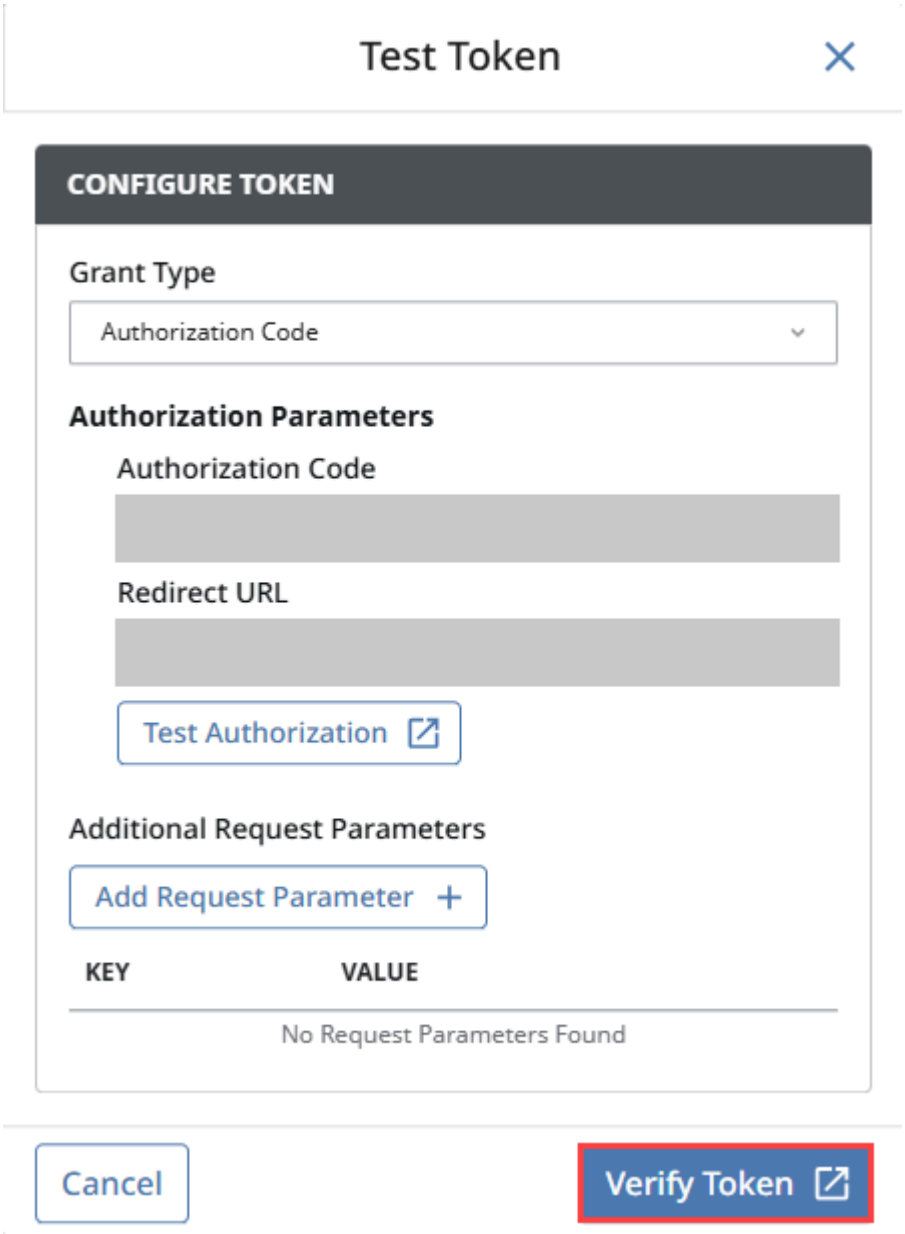
6. Once permissions are granted to the OAuth2 Client, you will be redirected to the Gateway. Information about the authentication, including the authorization code, timestamps, results, and more will be displayed.

> **NOTE**
>
> Besides the main OAuth2 Client page, you can use the **Verify Token** option on the Authentication Response dialog box to test access tokens. The benefit of testing your token

> from the Authentication Response dialog box is that some information, such as the Authentication Code and Redirect URL will be filled out.

7. Click **Verify Token** on the Authentication Response dialog box. This will bring you to the Test Token form, where some fields will already be filled in.

8. Click **Verify Token** to see what the token test results will be.



9. A Token Response dialog box will appear, showing the results of the token test, as well as timestamps, the response, and more.

Secrets Management

Secrets Management is the practice of securely storing and managing secrets on behalf of a user or system who owns and controls them. Common secrets include passwords, private keys, credentials, and access tokens. Ignition's Secrets Management system encrypts these secrets through built-in implementation. For a more robust setup, you can further customize your Ignition Secrets Management system from the default and create Secret Providers to store confidential secrets.

The process built into the Ignition platform is referred to as **embedded secrets**, while referenced integration is available through **Secret Providers**.

New in 8.3.1
Secret Providers now support the use of system functions to encrypt and decrypt data, access stored secret data, browse configured Secret Providers, and retrieve a list of secrets by provider. Refer to the **system.secrets** page for more information on using each function.

Establishing a Secrets Management System

Fresh Ignition installations are packaged with a default encryption key that is used to encrypt all secrets on the Gateway and provide the foundation for a more complex Secrets Management system. Although the default encryption key configuration is all that's required to set **Gateway System Secrets**, it is recommended to customize both your Root Key and create Encryption Key Sets, as the single default key matches every Ignition installation.

> **ACKNOWLEDGEMENT REQUIRED**
> You must opt in to the Secrets Management system complete functionality to customize the Secrets Management system and secure features, like customizing your keys, creating key sets, or otherwise enhancing your system with additional security layers. This is because it will be the user's responsibility to maintain the system once it has been customized. Refer to the **Secrets Management Key CLI Tool** section for more information about opting in, customizing your keys, and performing other Secrets Management tasks.

If you choose to establish a Secrets Management system, you are provided new keys and files that make up the layers of the Secrets Management system, beginning with the default Root Key and Encryption Key set. As described for the default setup, Encryption Keys Sets will still serve as the first step used to encrypt all secrets on the Gateway. However, instead of a single shared Encryption Key, you'll now have three keys that are all unique to you. The reason they are provided as a set is so that

they can be rotated for an additional security measure. The Encryption Key Set is stored securely as a `kek.json` file, and requires the Root Key to decrypt. Similarly, the Root Key is encrypted and stored as a `root.json` file. The final component you need to establish for your Secrets Management system is a unique environment password. This is delivered in a file and is used to decrypt the `root.json` file so the Root Key is usable.

To maintain security, the environment password is not saved within the system by default. Access to the environment password is possible through the use of environment variables, which make sure Ignition has the ability to decrypt your secrets.

Variable	Description
`IGNITION_ROOT_KEY_PASSWORD_FILE`	The path to a file in a folder with the root key password.
`IGNITION_ROOT_KEY_PASSWORD`	The custom key to start the Gateway.

It is essential to keep track of your environment password because without it nothing can be decrypted and your system will fail when attempting to verify the the Secrets Management system, such as on startup. If the Gateway cannot find the environment password, or if the password is invalid, there is an option on the Gateway or in command line utility to attempt to unseal the custom Root Key with the environment password manually. This option is available in situations where the password is held by company personnel instead of being stored digitally.

> **MAINTAINING ENCRYPTION KEYS ON UPGRADE**
>
> If you are upgrading from a version without a customized Secrets Management system configured, it will be treated as a fresh install. If you are upgrading with a Secrets Management system configured, it will be detected and maintained through the upgrade process.

> **WORKING WITH REDUNDANCY**
>
> Make sure that the `root` and `KEK` JSON files are applied to both the Master and Backup nodes in a redundant configuration. If not, the embedded secrets and Internal Provider secrets encrypted on one will not decrypt on the other.

Gateway Startup Example

The Secrets Management system is verified on each Gateway startup, regardless of how many, if any, secrets are configured. This is to ensure an encryption key set is available for both decrypting existing

secrets and for encrypting any new or changed secrets during the lifetime of a running Gateway. The following steps describe this process in simplified terms:

1. The `/data/config/ignition/keys` directory must be present with both the `root.json` and `kek.json` files.
2. The environment password is located and loaded through the environment variables, and is able to decrypt the `root.json` file.
3. The Root Key is able to decrypt the `kek.json` file.

Once the system confirms the Encryption Key Set can be decrypted and used to decrypt the Gateway secrets, the Secrets Management system is confirmed and the Gateway startup process moves on to the next task.

> **GENERATED ROOT KEYS AND ENCRYPTION KEY SETS**
>
> If the `/data/config/ignition/keys` directory doesn't exist, but the `IGNITION_ROOT_KEY_PASSWORD` or `IGNITION_ROOT_KEY_PASSWORD_FILE` environment variables are set, a new Root Key and custom Encryption Key set will be generated on startup so the startup process can continue.

Secrets Management Key CLI Tool

A Secrets Management Key CLI Tool is shipped with Ignition for users to be able to directly manage secret keys and run scripts from the command line, just like with the `gwcmd` and `ignition.sh` files. The Secrets Management Key CLI Tool exposes basic commands like generating, resetting, removing, and rotating keys.

- On Windows, use `ignition-secrets-tool.bat`.
- On *NIX, use `ignition-secrets-tool.sh`.

See the **Secrets Management Key CLI Tool** page for more information on the available commands.

Configuring Gateway System Secrets

The Secrets Management system replaces standard password and re-type password fields on the Gateway to offer the following three options when creating or editing devices, profiles, and system configurations that require security:

- **None**: Confirms no password is needed for the corresponding Gateway system. Although always visible, this option will be grayed out if a secret is required.

- **Embedded**: Displays a field to enter a secret. By default the characters will be hidden, but they can be made visible using the SHOW option. An Update Password option will also appear in case the password needs to be changed. This option replaces standard password and re-type password fields. Although secrets are encrypted at rest and hidden by the Gateway UI, the Gateway decrypts all secrets at runtime and privileged Ignition users can obtain access to the decrypted values.

- **Referenced**: Displays two new fields for choosing a Secret Provider and secret name for the corresponding Gateway system to reference. Although always visible, this option is only available for configuration when a **Secret Provider** exists. Like embedded secrets, referenced secrets are encrypted at rest and hidden on the Gateway. They are also decrypted at runtime and can be obtained by priviledged Ignition users.

Embedded Secrets

Secrets that are stored and managed internally in Ignition are considered to be embedded, as they don't depend on any external secrets management solution. Instead, embedded secrets use Ignition's encryption system.

Once an embedded secret is entered and saved, it is automatically encrypted. These secrets can only be exposed using an encryption key. Upon initial installation, you'll be given default encryption keys. For increased security, it's recommended to customize these encryption keys.

Secret Provider Types

Secret Providers offer a way for users to utilize Ignition as a Secrets Management service through referenced secrets. While the use of Secret Providers is not equivalent to incorporating a dedicated Secrets Management system, they can offer a solution to manage many secrets if no other option is otherwise configured.

Internal

With an Internal Secret Provider, secrets are managed by Ignition using an embedded secrets management strategy. All secrets within this provider are stored as files on the Gateway and referenced by an assigned name. This means secrets shared by multiple configs can reference the same name to facilitate re-use.

It is recommended to use different secrets for each system or connection requiring one.

Provider Properties

Properties	Description
Name	The name for the Internal Secret Provider.
Description	A description of the Secret Provider. [optional]
Enabled	Determines whether the Secret Provider is enabled or disabled within the Gateway.

Remote

New in 8.3.3

A Remote Secret Provider creates a link to a Secret Provider on a remote Gateway over the Gateway Network. Once connected, Ignition can list and read secrets. Remote Secret Providers service settings default to deny access and must be manually updated to allow access. The Secret Provider Access settings can be updated within the corresponding Security Zone's Manage Policy panel **Secret Provider Access** section.

Provider Properties

General

Properties	Description
Name	The name for the Remote Secret Provider.
Description	A description of the Secret Provider. [optional]
Enabled	Determines whether the Secret Provider is enabled or disabled within the Gateway.

Remote Gateway

Properties	Description
Gateway Name	The remote Ignition Gateway server from which to fetch secrets.
Secret Provider	The Secret Provider on the remote Ignition Gateway server from which to fetch secrets.

File

New in 8.3.5

The File Secret Provider is able to read secrets from files on disk. Once the provider is created, the file path is used to locate and add a secret. Files containing **secrets** can be formatted as either cleartext or as ciphertext.

A cleartext secret is comprised of all the bytes in the file, so do not include any new lines or carriage returns. Cleartext secrets are powerful on systems like Kubernetes, where secrets from external providers such as HashiCorp Vault, Azure, and AWS Secrets Manager can be mounted on disk and accessed by the File Secret Provider.

A ciphertext secret is a flat JSON Web Encryption structure generated by Ignition's system encryption service and uses the same structure as embedded secrets. A ciphertext secret can be created using the **system.secrets.encrypt** function, or by calling the REST API at `/data/api/v1/encryption/encrypt`. The REST API can be found by searching for `encryption` within the **OpenAPI docs**. If ciphertext secrets are shared between systems, ensure that each system has the encryption key used to originally encrypt the secret.

Provider Properties

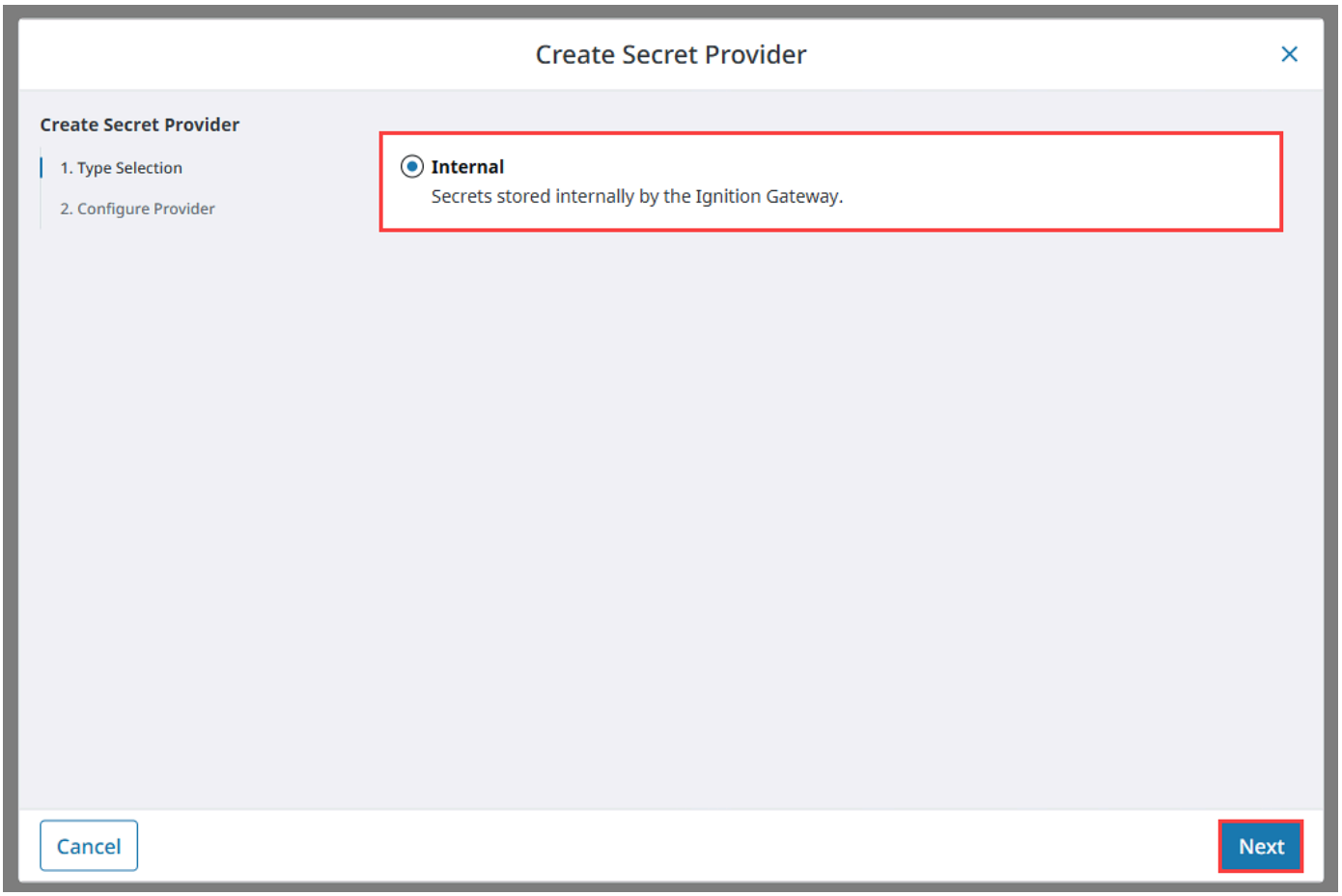
General

Properties	Description
Name	The name for the File Secret Provider.
Description	A description of the Secret Provider. [optional]
Enabled	Determines whether the Secret Provider is enabled or disabled within the Gateway.

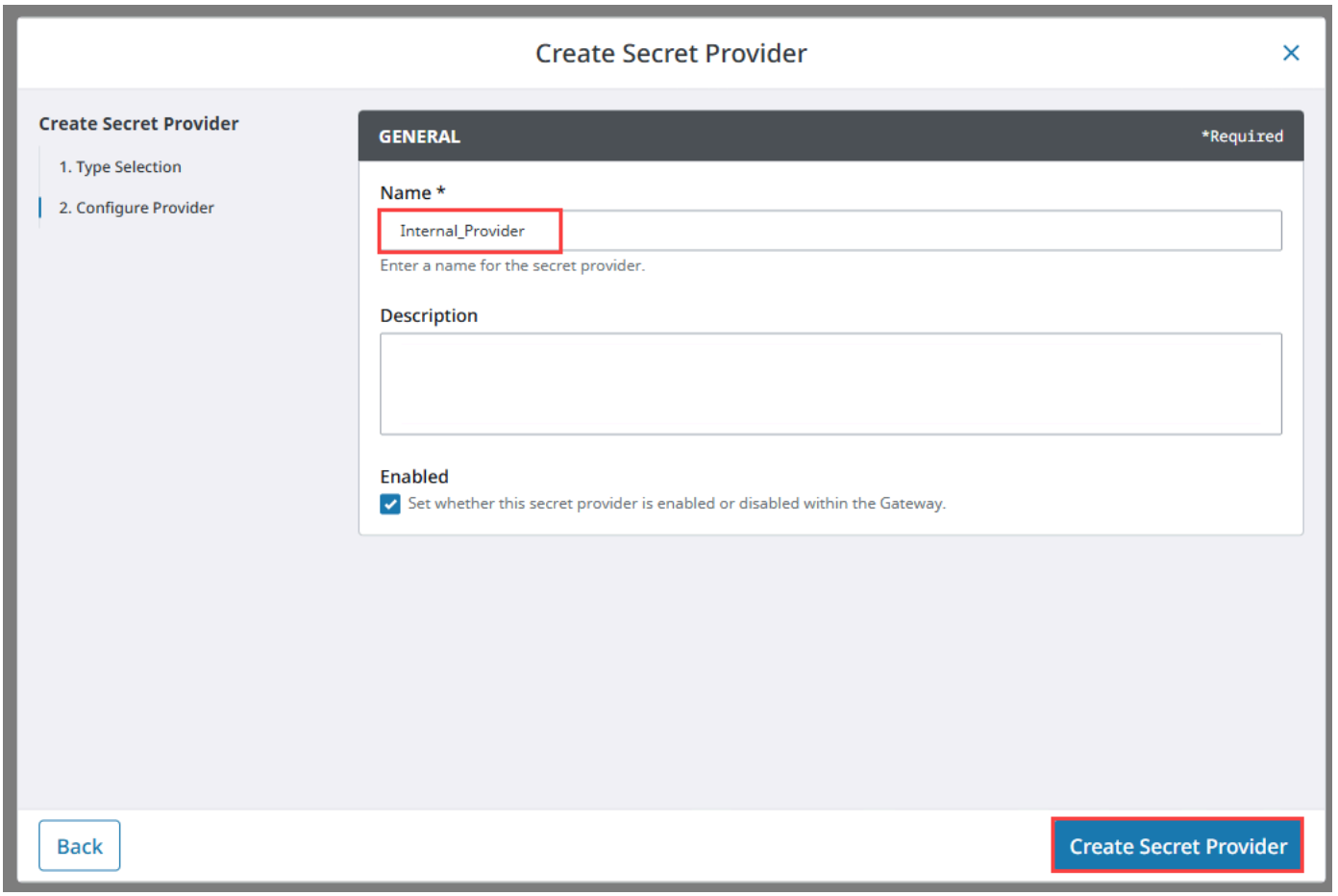
Creating a Secret Provider

To create a Secret Provider, complete the following steps. This example creates an Internal Secret Provider.

1. Access your Gateway and navigate to **Platform** > **Security** > **Secret Provider**.
2. Select **Create Secret Provider +**. The Create Secret Provider form will appear.
3. The Create Secret Provider form will appear with **Internal** selected, click **Next**.



4. Enter a name for your Secret Provider, and a description as desired. The Secret Provider will be enabled by default. The provider was named Internal_Provider for this example.



5. Click **Create Secret Provider**.

Your Secret Provider will now be displayed on the Secret Providers page where you can expand the three dots menu to make any further configurations relevant to your provider.

> **ADDING A SECRET PROVIDER TO AN EXISTING SETUP**
>
> If you already have embedded passwords configured for various systems and decide to set up a Secret Provider later, you can simply update the password selection for that secret from **embedded** to **referenced** in the Edit panel on the Gateway page.

Watch a Demo

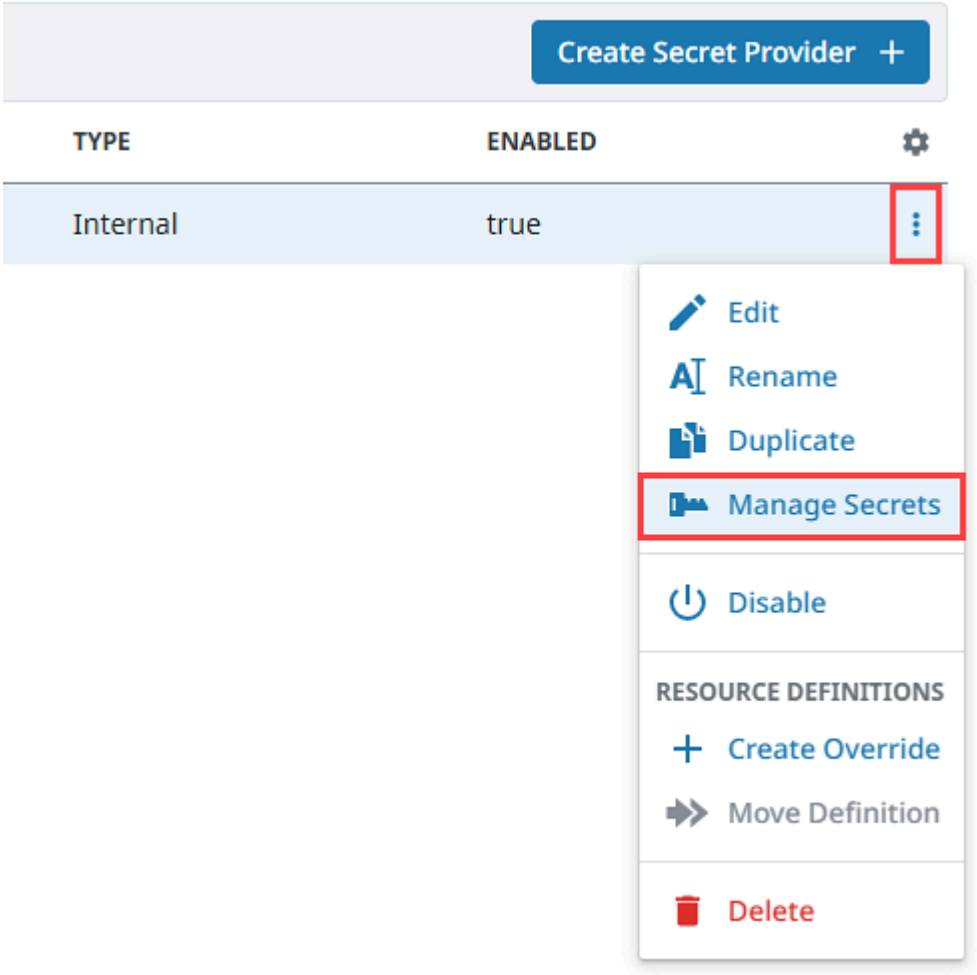
Open Demo

Try it Yourself: Create an Internal Secret Provider

Open Training Exercise

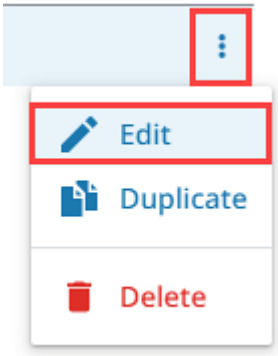
Managing Secrets

Secrets within a Secret Provider are added and updated on the Manage Secrets panel. Click the three dots menu for the Secret Provider and select **Manage Secrets** to access the Manage Secrets panel. Note that the Manage Secrets option is only available for Internal Secret Providers as secrets cannot be created with Remote Secret Providers.



From here, you can select **Create Secret +** to access the Create Secret form. Now, simply enter a name for the secret and the secret value. The Description field is optional. Once added, an encrypted secret value is created. This value cannot be read from the Gateway.

If you need to update an existing secret value, click the three dots menu for the secret and select **Edit** where you'll see the **Reset Secret Value** option. While you enter a new secret value, you'll have visibility to the value. However, once the new value is saved, it will once again be encrypted and hidden.



Internal Secret Properties

General

Properties	Description
Secret Name	The name of the secret.
Description	The description of the secret. [optional]

Secret Value

Properties	Description
Secret Value	The value to store for this secret. Upon creation, the secret value will be encrypted and cannot be read again within the Gateway.

File Secret Properties

General

Properties	Description
Name	The name of the secret.
Description	The description of the secret. [optional]

File Settings

Properties	Description
File Path	The file path on disk for where the secret resides.
File Type	Whether the file type for the secret is `CLEARTEXT` or `CIPHERTEXT`.

Secrets Management Key CLI Tool

Built-in Scripts

The built-in scripts included on this page are available with the Secrets Management Key CLI tool that is included with each Ignition installation to help manage your Secrets Management system once it's configured. Each script begins with the following command:

- On Windows, use `ignition-secrets-tool.bat`.
- On *NIX, use `ignition-secrets-tool.sh`.

The scripts are described with the commands required to run. Any of the options for each command are accepted, and can be chosen based on preference. The following commands are available for each built-in script to be used as needed.

Command Options	Description
`-f`, `--force`	Force an overwrite of the target file(s) if it already exists. Ignored if the output destination is not a file or if the same value is used for both input and output argument values.
`-h`, `--help`	Show the help message and exit.
`-V`, `--version`	Print version information and exit.

> **WORKING WITH REDUNDANCY**
>
> When making changes, make sure that the `root` and `KEK` JSON files are applied to both the Master and Backup nodes in a redundant configuration. If not, the embedded secrets and Internal Provider secrets encrypted on one will not decrypt on the other.

Generate a New Root Key

This script generates a new Root Key and encodes it as a JSON Web Key (JWK). The Root Key is encrypted using the given password or passphrase.

Usage

```
ignition-secrets-tool root generate [-fhV] [-p[=<pass>]] [-r=<value>]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.

Generate a New Key Encryption Key

This script generates a new Key Encryption Key (KEK) Set:

1. A new KEK is generated and encoded as a JSON Web Key (JWK).
2. A new KEK Set is generated and encoded as a JWK Set.
3. The new KEK is wrapped inside the new KEK Set as its only element.
4. The KEK Set is encrypted using the given Root Key.

Usage

```
ignition-secrets-tool kek generate [-fhV] [-p[=<pass>]] [-k=<value>] [-r=<value>]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-k`, `--kek=<value>`	KEK Set input source or output destination, as applicable. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<value>`.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.

List the Key Encryption Keys

This script lists Key Encryption Keys (KEKs):

1. A password or passphrase is first loaded to decrypt the Root Key.
2. The Root Key is loaded next to decrypt the KEK Set.
3. The KEK Set is finally loaded last, and its contents are printed to `stdout` with sensitive key material omitted.

Usage

```
ignition-secrets-tool kek list [-hV] [-p[=<pass>]] [-k=<value>] [-r=<value>]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-k`, `--kek=<value>`	KEK Set input source or output destination, as applicable. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<value>`.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.

Rotate the Root Key

This script rotates the Root Key:

1. A password or passphrase is first loaded to decrypt the old Root Key.
    - Optionally, a second distinct password or passphrase is loaded next to encrypt the new Root Key.
2. The old Root Key is loaded next to decrypt the KEK Set.
3. The KEK Set is loaded last. It is decrypted using the old Root Key.
4. A new Root Key is generated and encoded as a JSON Web Key (JWK).
5. The new Root Key is encrypted using the given password or passphrase.
6. The new Root Key is written to its target output destination.
7. The KEK Set is encrypted using the new Root Key.
8. The encrypted KEK Set is written to its target output destination.

Usage

```
ignition-secrets-tool root rotate [-fhV] [-p[=<shared>] | [[--pwd:in [=<in>]] [--pwd:out[=<out>]]]] [-r=<shared> | [[--root:in=<in>] [--root:out=<out>]]] [-k=<shared> | [[--kek:in=<in>] [--kek:out=<out>]]]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<shared>]`	Password or passphrase to decrypt the old Root Key and encrypt the new Root Key. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<shared>]`.
`--pwd:in`, `--pass:in`, `--password:in`, `--passphrase:in[=<in>]`	Password or passphrase to decrypt the old Root Key. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<in>]`.
`--pwd:out`, `--pass:out`, `--password:out`, `--passphrase:out[=<out>]`	Password or passphrase to encrypt the new Root Key. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<out>]`.
`-r`, `--root=<shared>`	Root Key input source and output destination. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<shared>`.
`--root:in=<in>`	Old Root Key input source. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<in>`.
`--root:out=<out>`	New Root Key output destination. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<out>`.
`-k`, `--kek=<shared>`	KEK Set input source and output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<shared>`.
`--kek:in=<in>`	KEK Set input source. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<in>`.
`--kek:out=<out>`	KEK Set output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<out>`.

Rotate the Key Encryption Key

This script rotates the Key Encryption Key (KEK):

1. A password or passphrase is first loaded to decrypt the Root Key.
2. The Root Key is loaded next to decrypt the KEK Set.
3. The KEK Set is loaded last.
4. A new KEK is generated and encoded as a JSON Web Key (JWK).
5. The new KEK is added to the end of the KEK Set, which makes it the new primary key.
6. The KEK Set is encrypted using the given Root Key.
7. The encrypted KEK Set is written to its target output destination.

Usage

```
ignition-secrets-tool kek rotate [-fhV] [-p[=<pass>]] [-r=<value>] [-k=<shared> | [[--kek:in=<in>] [--kek:out=<out>]]]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.
`-k`, `--kek=<shared>`	KEK Set input source and output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<shared>`.
`--kek:in=<in>`	KEK Set input source. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<in>`.
`--kek:out=<out>`	KEK Set output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<out>`.

Reset the Primary Key Encryption Key

This script sets the primary Key Encryption Key (KEK) to the key with a given ID. The primary KEK is the key in the last position in the KEK Set. The primary KEK is the key used for encrypting new secrets.

1. A password or passphrase is first loaded to decrypt the Root Key.
2. The Root Key is loaded next to decrypt the KEK Set.
3. The KEK Set is loaded last, and its primary key is set to the key with the given ID.
4. The KEK Set is encrypted using the given Root Key.
5. The encrypted KEK Set is written to its target output destination.

Usage

```
ignition-secrets-tool kek primary [-fhV] [-p[=<pass>]] -i=<value> [-r=<value>] [-k=<shared> | [[--kek:in=<in>] [--kek:out=<out>]]]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-i`, `--id`, `--kid`, `--key-id=<value>`	The ID of the key to set as primary or to remove.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.
`-k`, `--kek=<shared>`	KEK Set input source and output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<shared>`.
`--kek:in=<in>`	KEK Set input source. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<in>`.
`--kek:out=<out>`	KEK Set output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<out>`.

Remove a Key Encryption Key

This script removes the Key Encryption Key (KEK) with a given ID:

1. A password or passphrase is first loaded to decrypt the Root Key.
2. The Root Key is loaded next to decrypt the KEK Set.
3. The KEK Set is loaded last, and the key with the given ID is removed from the set.
4. The KEK Set is encrypted using the given Root Key.
5. The encrypted KEK Set is written to its target output destination.

Usage

```
ignition-secrets-tool kek remove [-fhV] [-p[=<pass>]] -i=<value> [-r=<value>] [-k=<shared> | [[--kek:in=<in>] [--kek:out=<out>]]]
```

Command Options	Description
`-p`, `--pwd`, `--pass`, `--password`, `--passphrase[=<pass>]`	Password or passphrase to encrypt/decrypt the Root Key, as applicable. See the **Password** section below for how to format the argument value that follows this command, or replaces `[=<pass>]`.
`-i`, `--id`, `--kid`, `--key-id=<value>`	The ID of the key to set as primary or to remove.
`-r`, `--root=<value>`	Root Key input source or output destination, as applicable. See the **Root Key** section below for how to format the argument value that follows this command, or replaces `<value>`.
`-k`, `--kek=<shared>`	KEK Set input source and output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<shared>`.
`--kek:in=<in>`	KEK Set input source. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<in>`.
`--kek:out=<out>`	KEK Set output destination. See the **KEK Set** section below for how to format the argument value that follows this command, or replaces `<out>`.

Argument Value Formatting

Password

The password or passphrase argument value can be formatted as follows:

- `pass:password`: The actual password or passphrase is password. Since the password or passphrase is visible to utilities (like 'ps' under Unix) this form should only be used where security is not important.

- `env:var`: Obtain the password or passphrase from the environment variable var. Since the environment of other processes is visible on certain platforms, this option should be used with caution.
- `file:pathname`: Reads the password or passphrase from the specified file pathname, which can be a regular file, device, or named pipe. Only the first line, up to the newline character, is read from the stream.
- `stdin`: Reads the password or passphrase from standard input. The same line handling as described for file applies to passwords read from standard input.

If no password or passphrase argument value is given then the user is prompted to enter one. This will typically be read from the current terminal with echoing turned off. If this option is omitted entirely, then the value is obtained from the environment variable IGNITION_ROOT_KEY_PASSWORD and if this environment variable is not set, then the user is prompted to enter one.

Root Key

The Root Key argument value can be formatted as follows:

- `file:pathname`: Reads or writes the Root Key from/to the specified file pathname, which can be a regular file, device, or named pipe. Only the first JSON element is read from the stream, which is expected to be a JSON object representing the flattened JSON Web Encryption (JWE) serialization of the encrypted Root Key.
- `stdout`: Reads or writes the Root Key from/to standard output.

If this option is omitted entirely, then the value defaults to `file:$CWD/data/config/ignition/keys/root.json` where `$CWD` is the current working directory.

KEK Set

The KEK Set argument value can be formatted as follows:

- `file:pathname`: Reads or writes the KEK Set from/to the specified file pathname, which can be a regular file, device, or named pipe. Only the first JSON element is read from the stream, which is expected to be a JSON object representing the flattened JSON Web Encryption (JWE) serialization of the encrypted Root Key.
- `stdout`: Reads or writes the KEK Set to standard output. The same handling as described for file applies to KEK Sets read from standard input.

If this option is omitted entirely, then the value defaults to `file:$CWD/data/config/ignition/keys/kek.json` where `$CWD` is the current working directory.

API Keys

Using API Keys

An API Key is a unique identifier used to authenticate and authorize access to a system's resources without requiring a username and password for each request. In Ignition, API Keys are used to securely authenticate API requests made to the Gateway. They allow external applications, scripts, and services to interact with Ignition without needing interactive user authentication.

When an API Key is created, the Gateway hashes the key and stores it securely. The key itself is only visible to the user at the time of creation. API Keys are then used to authenticate HTTP requests by including them in the request header. The system verifies the provided key, checks associated security levels, and grants access accordingly.

> **⚠ CAUTION**
>
> API Keys grant full access to the Gateway's HTTP API routes, including the ability to modify configuration, tags, and projects. Improper use of API Keys may result in data loss, security breaches, or performance issues. Only issue keys to trusted users or systems, and ensure appropriate security levels are applied. To track configuration changes made through API requests, enable auditing. Mutative API calls (such as POST, PUT, and DELETE) are recorded in audit logs with the user, IP address, and API key. **GET requests are not recorded in audit logs.**

API Keys are required to access newly introduced resource-based configuration routes, including endpoints under `/data/api/v1/resources`, `/data/api/v1/sync`, and `/data/api/v1/modes`. These routes are documented in the Gateway's **API Documentation**.

Example: Using an API Key in an HTTP Request

To authenticate API requests, include the API Key in the `X-Ignition-API-Token` header using the following format:

```
X-Ignition-API-Token: <your-api-token>
```

Example

```
```

```
GET /api/resource HTTP/1.1
Host: your.gateway.address
X-Ignition-API-Token: <your-api-token>
```

This method is required for all routes documented in the Gateway's **API Documentation** interface (`/openapi`).

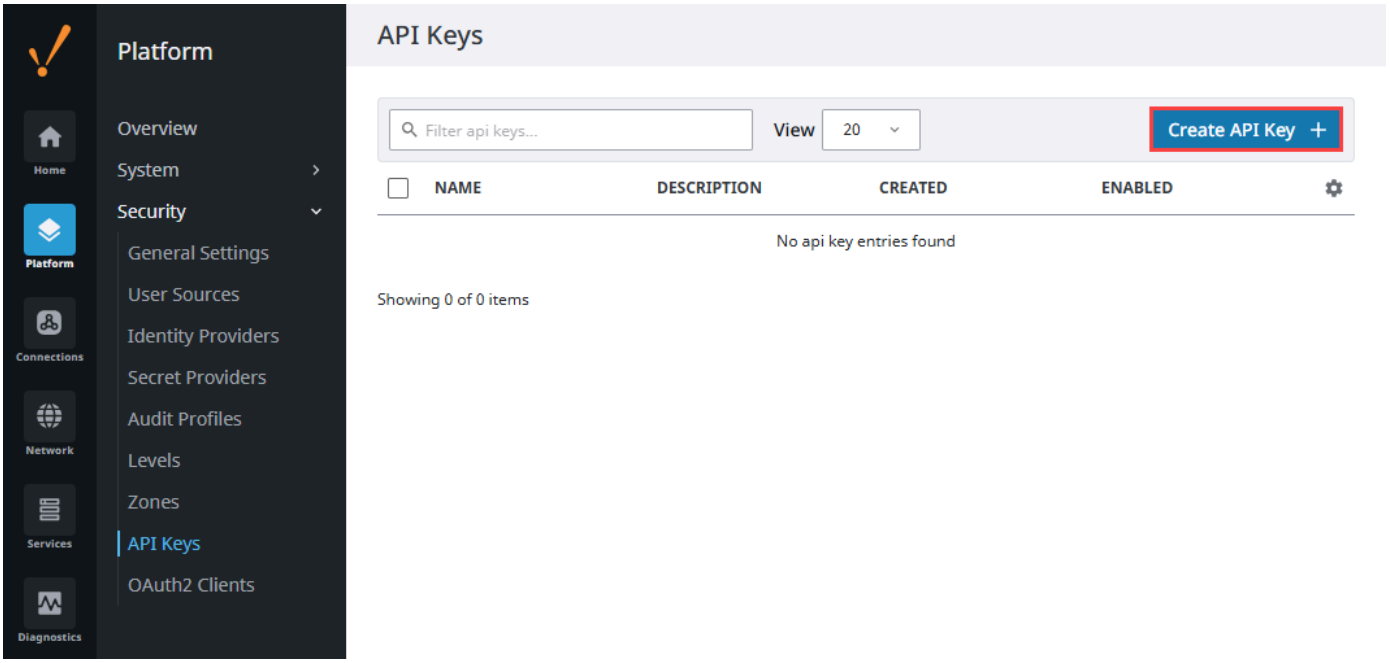
Creating an API Key

To create an API Key, follow the steps below.

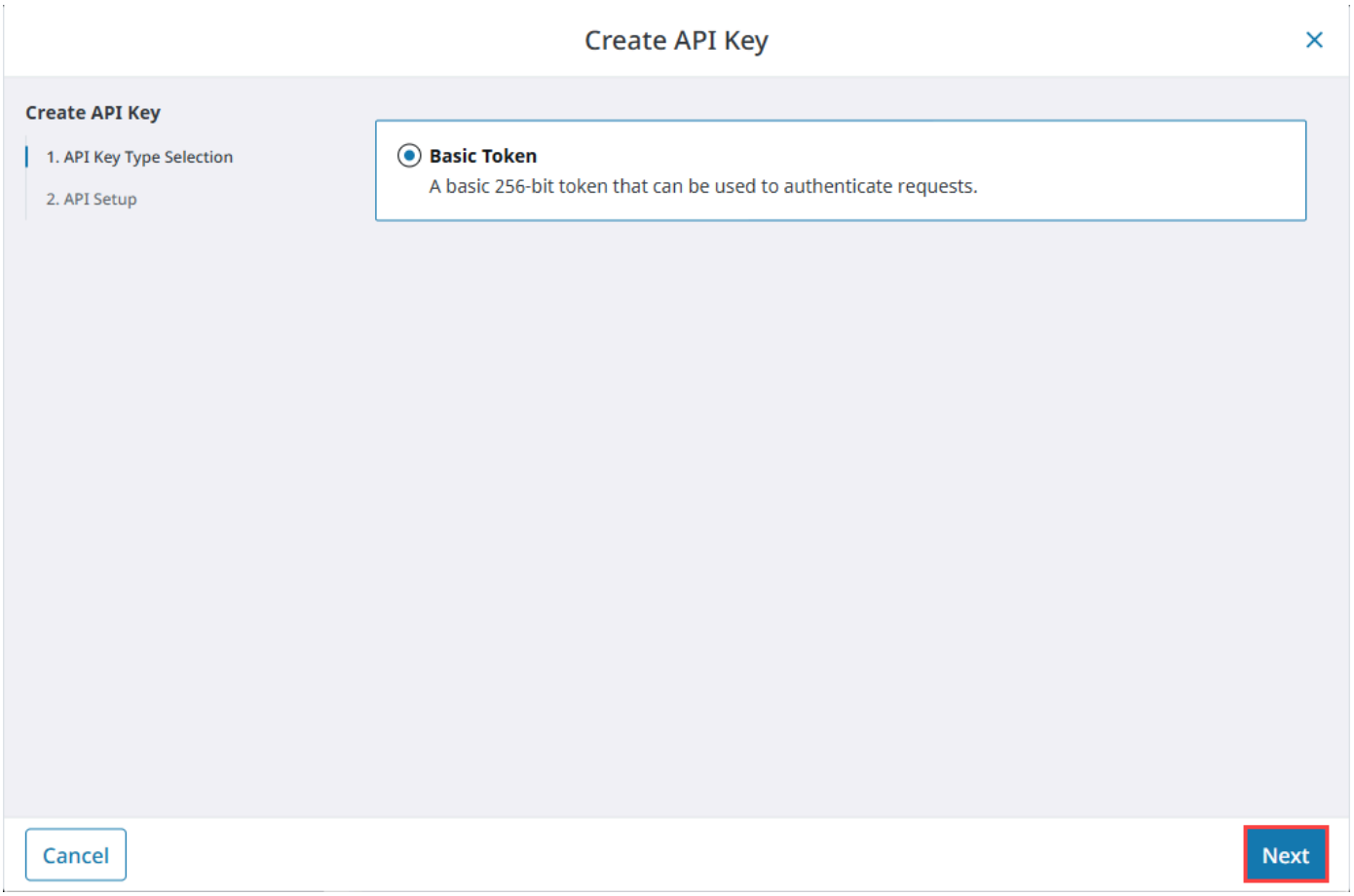
> **NOTE**
>
> Users must have proper write permissions defined on the Gateway's **Security > General Settings** page to create API Keys.

1. Access your Gateway and navigate to **Platform** > **Security** > **API Keys**.

2. Select **Create API Key +**. The Create API Key form will appear.

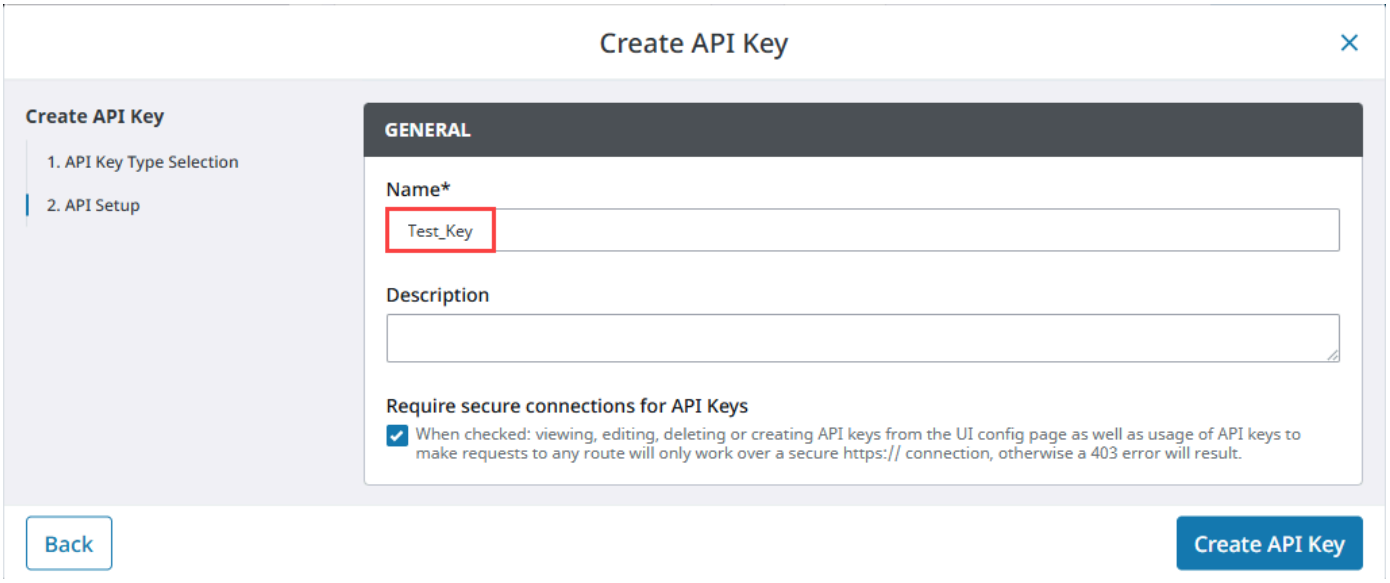


3. Currently, **Basic Token** is the only type available. Click **Next**.



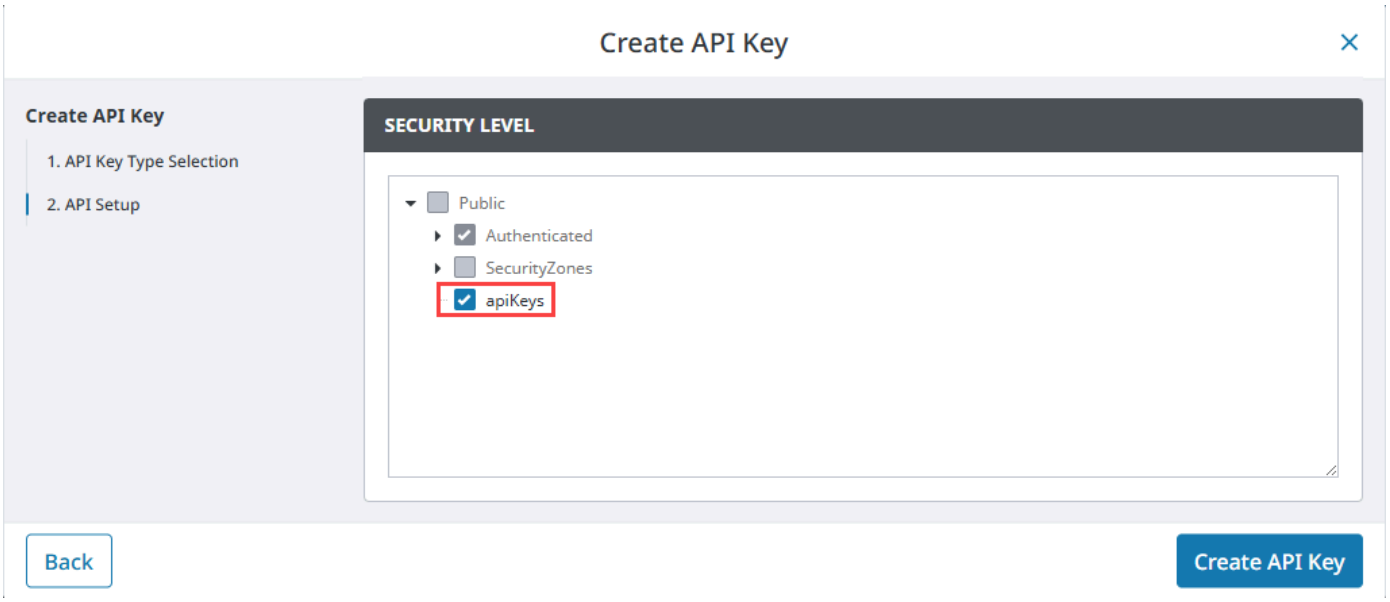
4. Enter a name for your API Key, and optional description. By default, **Require secure connections for API Keys** is enabled.

   In this example, the API Key is named **Test_Key**.



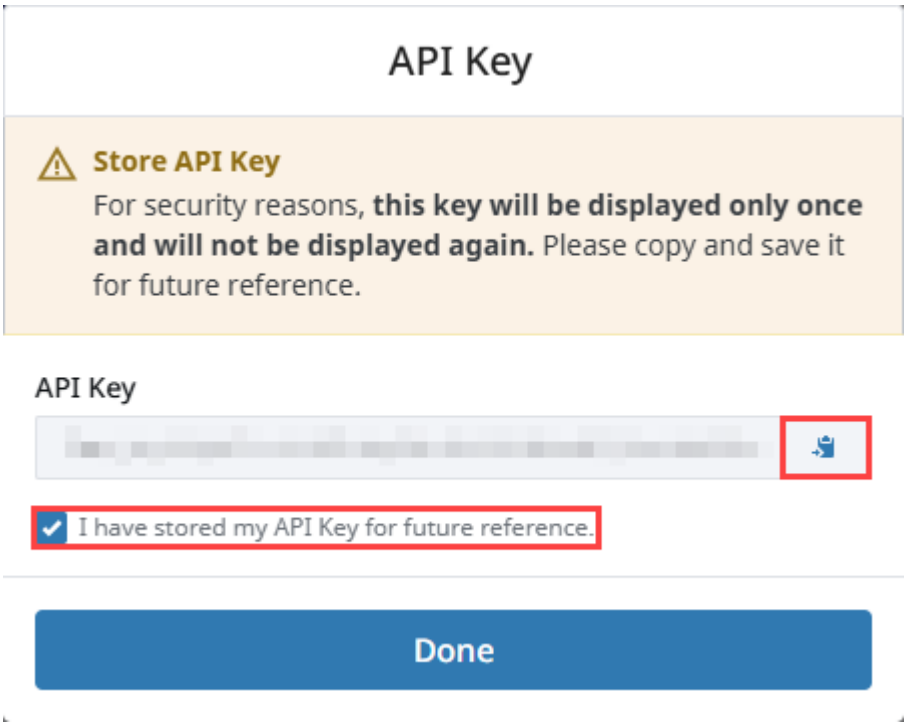
5. Select security levels for the key. By default, the **Authenticated** level is selected and cannot be changed. If you need to add more options, they can be created on the Security > **Levels** page.

   In this example, a new security level named **apiKeys** was created and selected.



6. Click **Create API Key**.

7. A warning message will appear with your API Key shown. Copy the key and store it securely, then select **I have stored my API Key for future reference.**



> **IMPORTANT**
>
> This is the **only** opportunity to copy the key and store it securely (i.e., with a reputable password manager or secure vault solution), as it **cannot** be retrieved again.

8. Click **Done**.

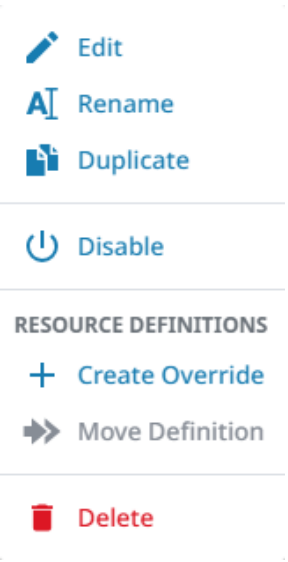
Managing API Keys

The API Keys page lists all created keys in a table with the following columns:

- **Name**: The key's name.
- **Description**: An optional description provided at creation.
- **Created**: The date and time the key was created.
- **Enabled**: Indicates whether the key is currently active or disabled.

Editing, Renaming, Duplicating, or Deleting

Open the three dots menu next to a key to choose an action:

Menu	Action Descriptions
	• **Edit**: Modify the key's settings. • **Rename**: Change the key's display name. • **Duplicate**: Copy the key's settings to create a new key. • **Disable**: Temporarily revoke access without deleting the key. • **Create Override/Move Definition**: Manage resource definitions for deployment modes. For details on overriding named resources, see the **Resource Definitions** section on the Gateway Deployment Modes page. • **Delete**: Permanently remove the key. Deleting an API Key **immediately revokes access** for any services or applications using it. This action **cannot be undone**.

When duplicating, the Duplicate API Key form copies all current settings. Enter a new unique name and click **Duplicate API Key**.

Duplicate API Key

GENERAL

Name*

Unique_Test_Key

Description

Require secure connections for API Keys

When checked: viewing, editing, deleting or creating API keys from the UI config page as well as usage of API keys to make requests to any route will only work over a secure https:// connection, otherwise a 403 error will result.

SECURITY LEVEL

- Public
  - Authenticated
  - SecurityZones
  - apiKeys

Cancel

Duplicate API Key

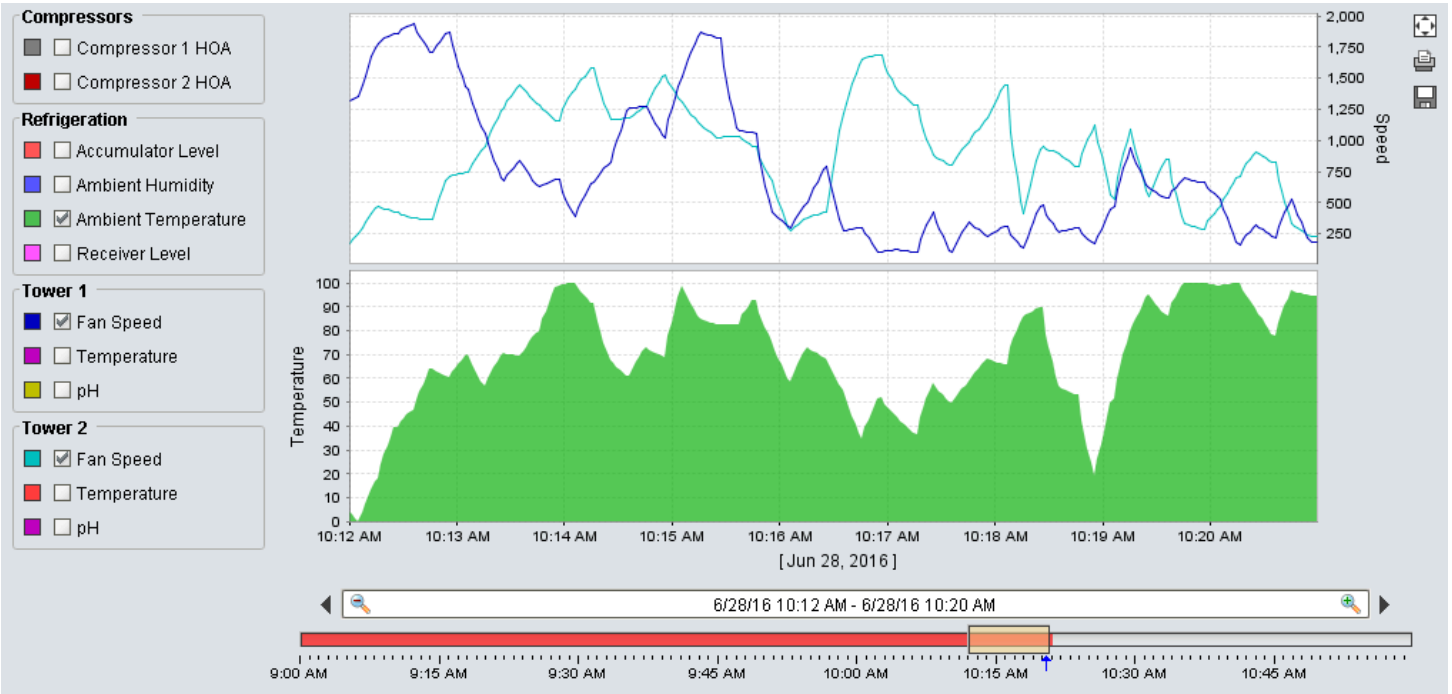
SQL in Ignition

Leveraging Databases

In addition to all the normal HMI functionality you would expect, Ignition has the ability to connect to databases, and this can greatly increase the functionality available to you! You can use databases to store history, create easy to search lists and configurations, and retrieve data from ERP or other systems. When you start using SQL, you can expand your project from a simple HMI to a project that brings your whole process together. The best part is that Ignition connects to as many databases as you want, all from one central location. Because the database lives outside of Ignition, we don't maintain any control or rules over what you can do with your data.

Displaying Data

You can easily display information from your databases on a window along with anything else in Ignition. You can show parts lists, step sequences, **realtime or historical charts**, add the ability to search for inventory, or anything else you can think of.



What is SQL?

SQL stands for "Structured Query Language" and is the backbone of most modern relational databases. It's often referred to as "S.Q.L." or "Sequel," but both are correct and widely recognized. This language (different than the **Python Scripting Language**) allows you to write out requests or

"queries" against the existing data to **view, add, edit, or remove** the information you want in a simple format.

Everything in a relational database is based around tables. Tables store the basic information for any system and can be combined together to make very efficient queries to retrieve your data.

id	name	description	location	qty
576	Mold 1	Mold for part # 1523	Warehouse A	23
577	Mold 2	Mold for part # 8825	Warehouse A	12
578	Bin 1	Finished Goods Bin 1	Warehouse B	9
579	Bin 2	Scrap Bin 1	Warehouse B	14

```
SELECT *
FROM inventory
JOIN location on location.name = inventory.location;
```

SQL Queries

SQL queries are crucial to Ignition's database-centric model. Queries can **show what is available** or **alter data in the databases**, and some companies have positions just dedicated to running databases and creating queries. Anywhere Ignition is fetching data, you can choose to use your own custom queries to get exactly what you want out of the database. You can make your queries as simple or complex as you like. If your database is large, you might have a whole team dedicated to creating these queries for you and Ignition will happily execute them.

SQL - Select Statement - Select Data from a Table

```
SELECT * FROM mytable
```

SQL - Inner Join Statement - Selects Records that have Matching Values

```
SELECT users.id, users.firstname, users.lastname, roles.name as 'rolename'
FROM users
INNER JOIN mapping ON users.id = mapping.userid
INNER JOIN roles ON mapping.roleid = roles.id
WHERE roles.name = 'Administrator'
```

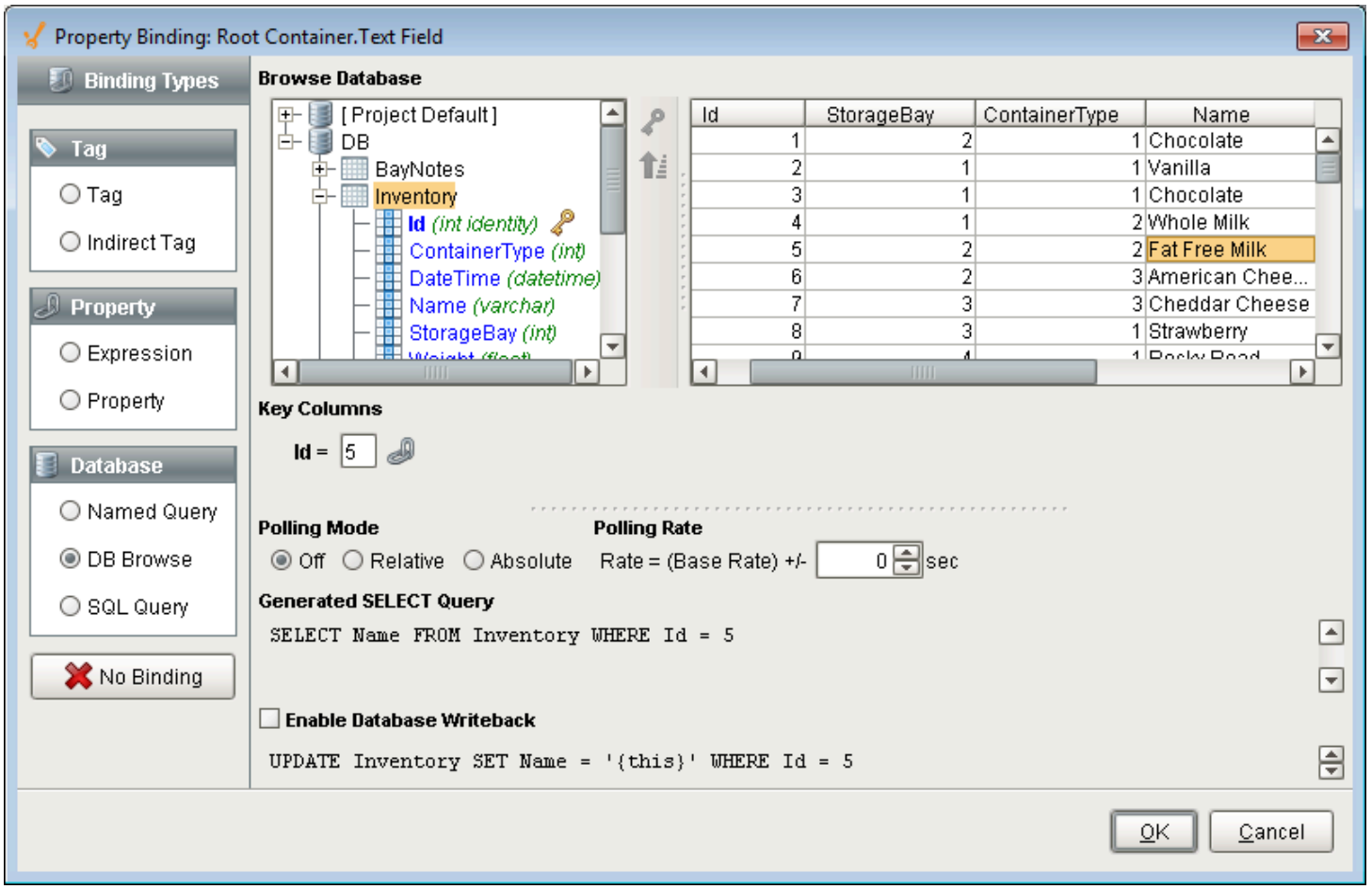
Database Connections

Any SQL query you use needs a **Database connection**, but Ignition simplifies all that by creating database connections in the Gateway instead of in the clients. This means from one central location you can manage all your database connections, and you don't have to worry about planning around adding clients in the future. Any special rules or connection restrictions are taken care of in the Gateway.

Using SQL in Ignition

There are many types of queries, and many ways to use them in Ignition. Some provide an easy to use builder to automatically store or fetch data, and some allow you to completely customize your queries.

Queries in Bindings Ignition's binding system offers a lot of flexibility in how database data can be used in a binding. The **Named Query** binding allows you to select one of the Named Queries that were previously built for that project, offering a very secure method of pulling data from the database. The **DB Browse** builder provides an interface that will build the query based on the data in the table that was selected. This allows even users with little SQL knowledge to pull data from the database. Finally, the **SQL Query** option will accept a straight query, so that a query specific to that binding can be written directly in the binding. When binding to a basic (non-dataset) data type, you can use the Writeback feature directly to send any changes back to the database.



Queries in Scripting

Ignition offers a number of built-in scripting functions for you to use to query your databases. This makes it very simple to view data, create dynamic scripts that use real data, and more. You can pull individual pieces of information, return whole tables of data, or update your database directly. Depending on the type of query and the sort of results you want, you will use different functions. The following functions are the ones you will use most, and all of them can use a special placeholder (?) to allow for dynamic query building.

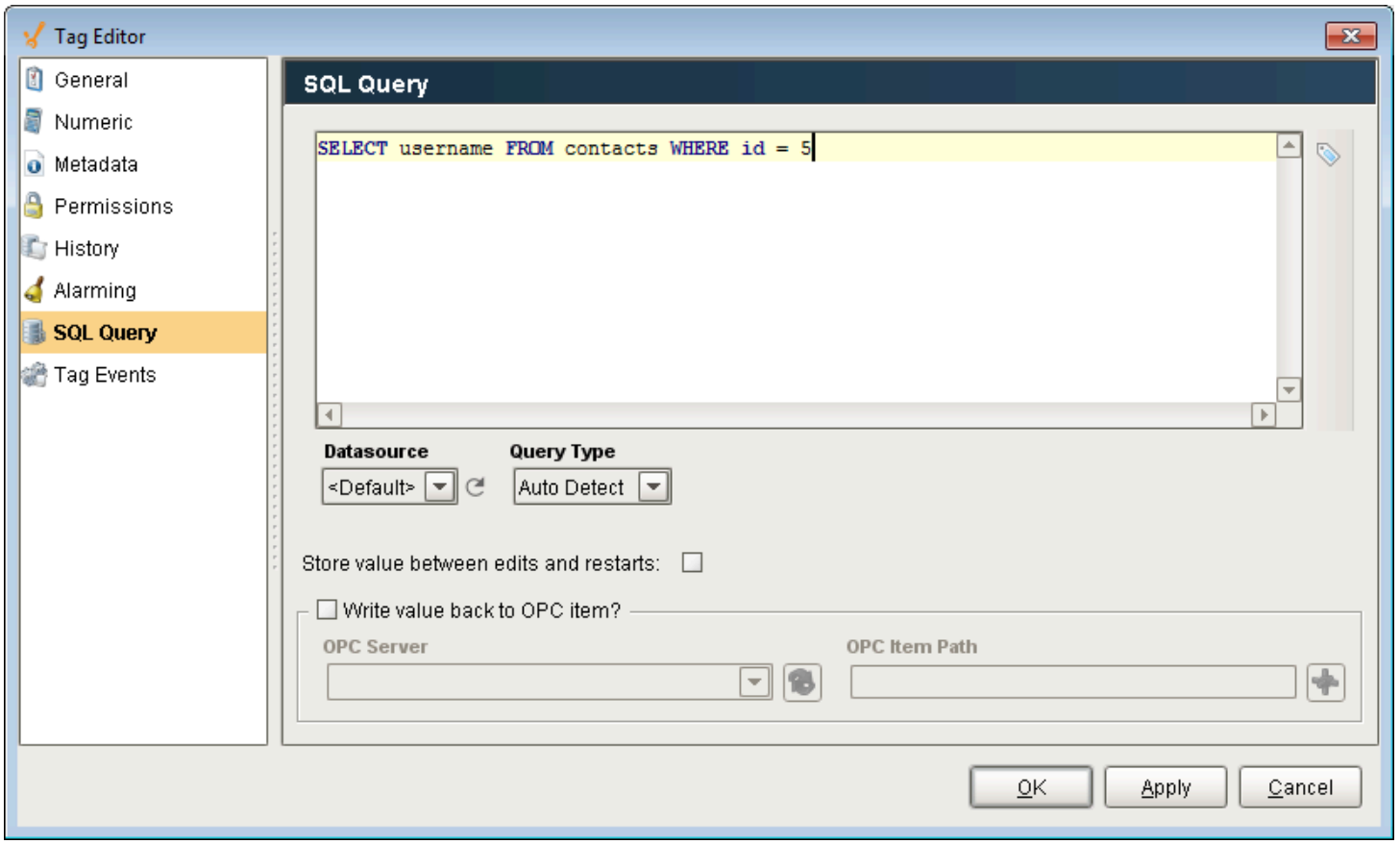
Scripting Function	Description
system.db.execQuery()	Used to execute a Select Query from a previously setup **Named Query** within a script.
system.db.execUpdate()	Used to execute an Update Query from a previously setup **Named Query** within a script.
system.db.runPrepQuery()	Used to run basic SELECT queries to fetch whole datasets. This can be used to populate tables, or to sift through the data to do your own calculations.
system.db.runPrepUpdate()	Used to run queries that change the data in the database. Usually used on input form windows to update your database records.

Each of the different functions takes in different arguments (values) and provides slightly different options and functionality. For example, the runPrepUpdate() can return the auto-generated key from insert queries. This can be extremely helpful and eliminate the need to hit the database multiple times if you are using linked tables.

You can find examples of each of these and all the other database functions in the **system.db** section of the appendix.

Queries in Tags

Ignition offers Query Tags, which can run queries and return the result as a Tag value, giving all of the projects in the Gateway access to the same Database values.



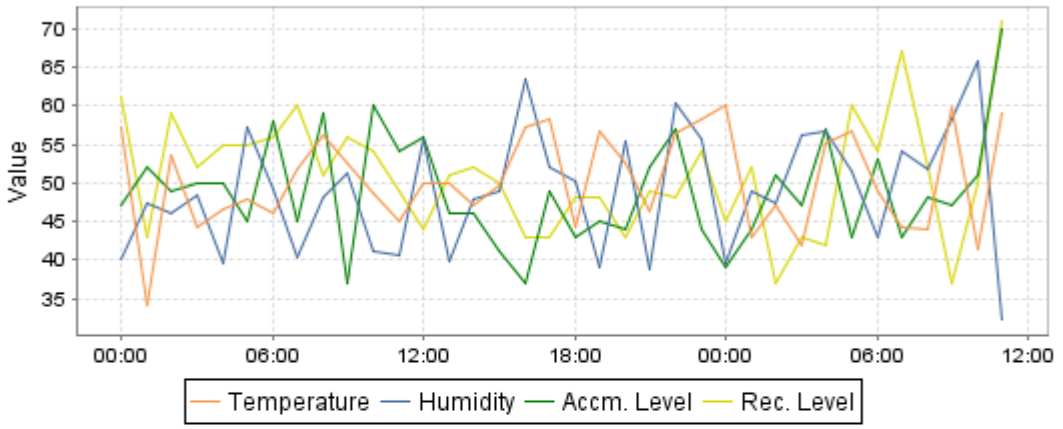
Queries in Reports

You can leverage queries to access data from all database connections to **create rich reports**, from simple data logging to complex reports with grouped charts and datasets.

Tag History Report

A report that queries tag history and displays a graph and table

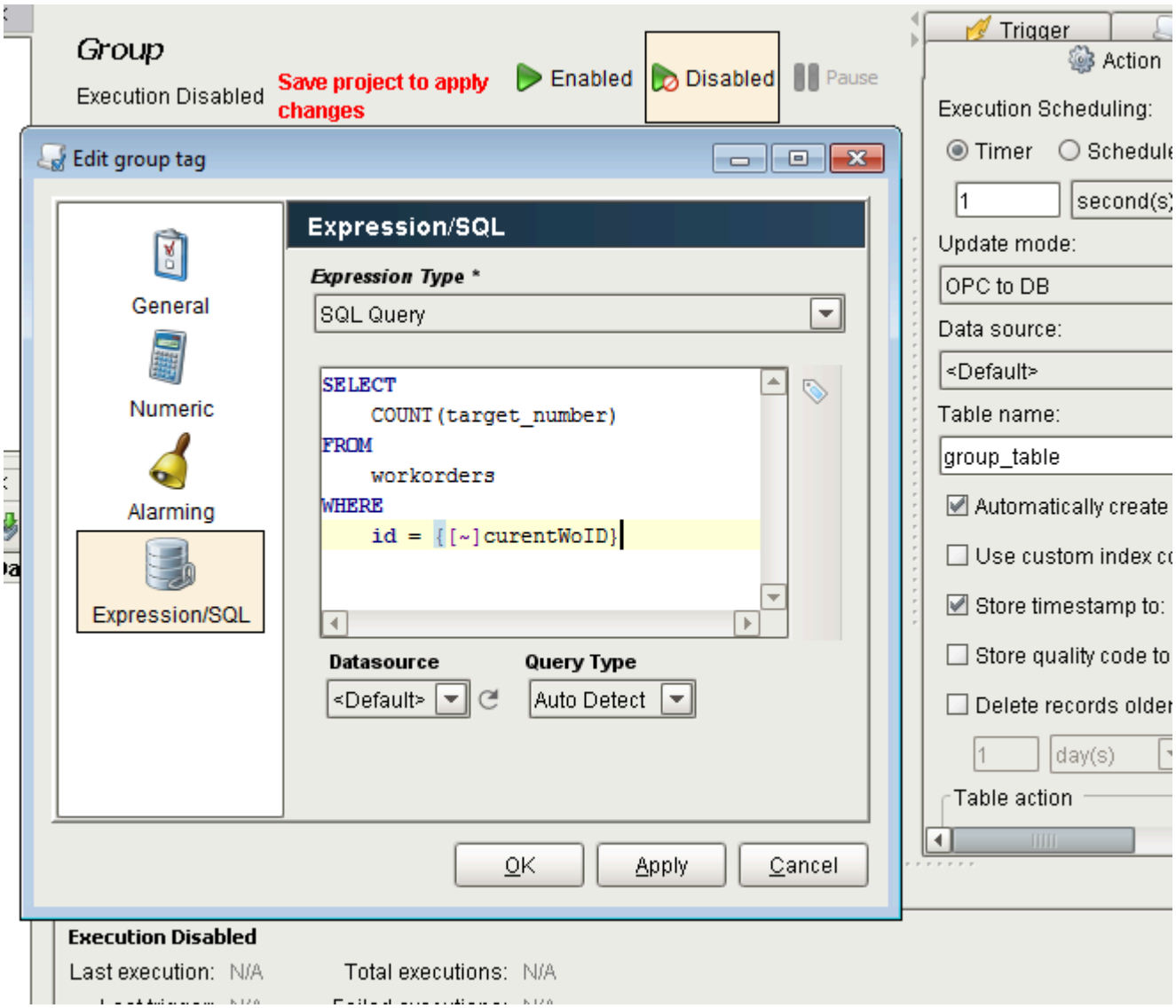
Report Start: 6/27/2016 12:00 AM
Report End: 6/28/2016 11:00 AM



Date / Time	Temperature	Humidity	Accm. Level	Rec. Level
6/27/2016 12:00:00 AM	57.34	40.14	47	61
6/27/2016 1:00:00 AM	33.98	47.33	52	43
6/27/2016 2:00:00 AM	53.61	46.15	49	59
6/27/2016 3:00:00 AM	44.3	48.5	50	52
6/27/2016 4:00:00 AM	46.57	39.59	50	55
6/27/2016 5:00:00 AM	47.9	57.16	45	55
6/27/2016 6:00:00 AM	46.04	49.16	58	56
6/27/2016 7:00:00 AM	51.87	40.42	45	60
6/27/2016 8:00:00 AM	56.24	48.07	59	51
6/27/2016 9:00:00 AM	52.42	51.17	37	56
6/27/2016 10:00:00 AM	48.74	41.01	60	54
6/27/2016 11:00:00 AM	44.89	40.67	54	49
6/27/2016 12:00:00 PM	49.85	55.73	56	44
6/27/2016 1:00:00 PM	49.82	39.84	46	51

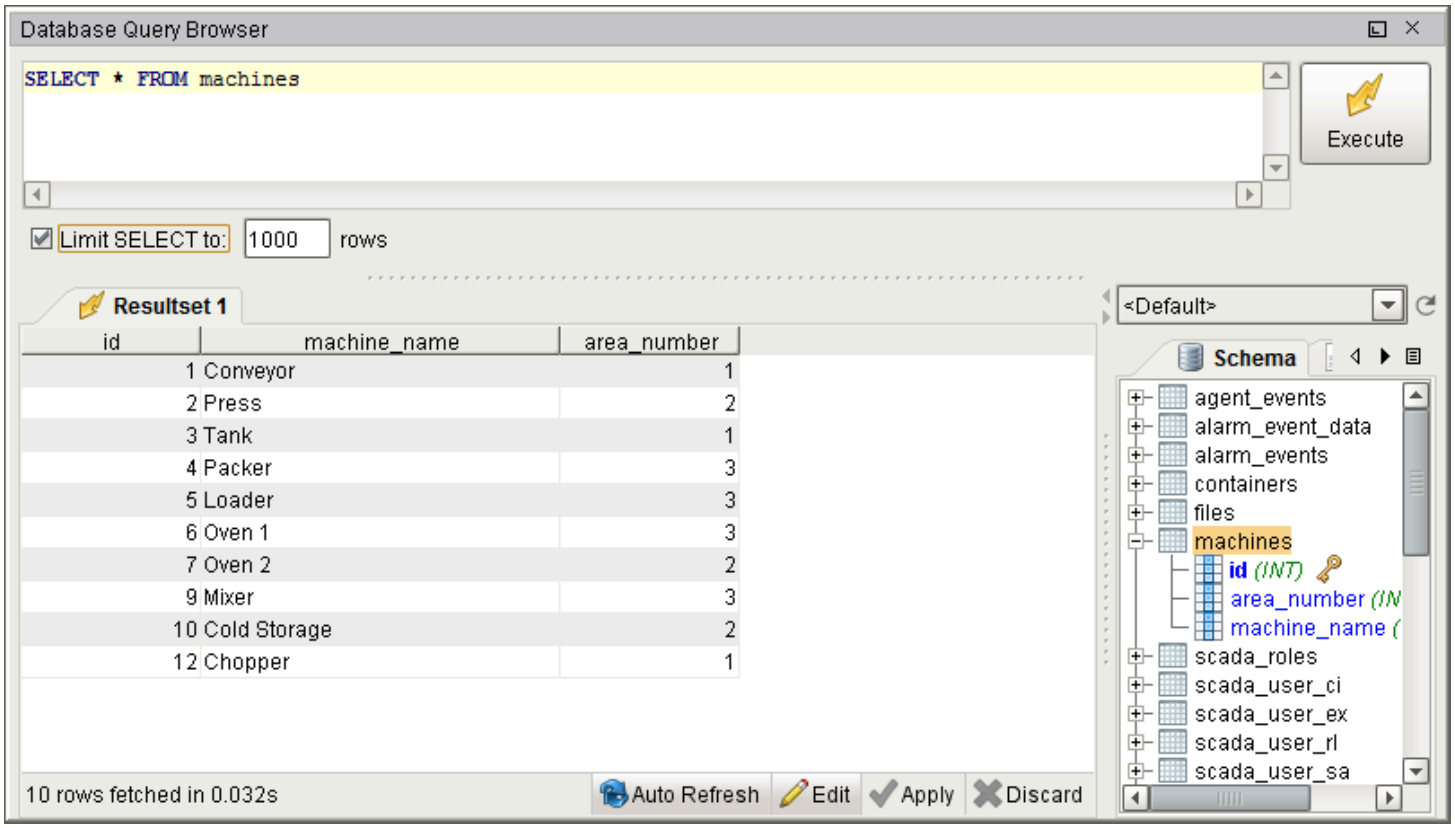
Queries in Transaction Groups

While **Transaction Groups** are great at storing Tag data to a Database automatically, the built-in Expression Items can execute a SQL Query within the Transaction Group.



Database Query Browser

The **Database Query Browser** offers an easy to use environment to run queries in for testing. Here, queries can be tested to figure out what values get returned, or data can be updated through raw queries, or the Database Query Browsers easy to use GUI editor.



Auto Generated Queries

Many systems within Ignition utilize a database connection, but the queries that are executed are constructed automatically by the system and do not require you to build the queries manually. These systems such as the **Tag Historian**, the **Alarm Journal**, or the **Database User Source** are very easy to setup and use since each system will automatically generate the necessary tables in the database, insert the relevant data, and even has prebuilt tools to extract the data. However, it is important to note that while these systems can automatically generate queries for you using the various components, these systems are simply storing data in a database which you can manually query out by building your own SQL queries.

Writing SQL Queries

SQL Tables

> **INFO**
>
> For information about databases and how to get connected, see the Database Connections section.

The foundation of every database system is a table. Every database consists of one or more tables, which store the database's data/information. Each table is identified by a name (for example `Customers` or `Orders`), and consists of column definitions and rows of data.

The database table columns have their own unique names and have pre-defined data types. Table columns can have various attributes defining the column functionality (such as the primary key, index, default value, and so on).

While table columns describe the data types, the table rows contain the actual data for the columns.

ID	Name	Address
1	Safeway	123 Safeway Lane
2	Costco	456 Costco Way
3	Target	789 Target Ave

Current common databases are almost all Relational Databases. This means that their tables can relate to each other by including an ID in one table that matches the key of another. These are called foreign keys.

Primary Key

A primary key is a way to uniquely identify each row in a table. While it is possible to create a database table without a primary key, it is highly recommended to configure one for each table. A primary key is comprised of either a single column, or set of columns. When multiple columns are specified as a primary key, this is known as a **composite primary key**. No two distinct rows in a table can have the same value (or combination of values) in those columns.

While Primary Keys can be configured in several ways, they *typically* meet the following criteria:

- **Integer Data Type**: The data type of the key column is typically an integer, and not a varchar. The primary key is only an identifier to a specific row in a table, so an integer data type can easily be

used. Some databases support a UID or a UUID (Universally Unique IDentifier) that looks like a character string, but is something specially made for primary keys.

- **Automatically Incrementing**: The value of the primary key increments as rows are added. The key is usually configured to automatically increment in the database so that external applications (such as Ignition) don't have to figure out the next available value when inserting a new row.
- **Statically Defined**: Any row that is inserted must fill in these value(s) without creating duplicates. Configuring the primary key as automatically incrementing means that the database will **automatically** handle this criteria.
- **Non-NULL**: NULL (empty) values should not be present in the primary key. This column (or columns) will usually not allow NULL values.

Index

Indexes speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book's index helps you find information quickly within that book. Indexes are extremely important when querying large sets of data. You should create an index for the set of columns you use commonly in a `WHERE` clause. For example, you should add an index on the timestamp column of a historical table when querying the table by a start and end date. Ignition does this automatically when it creates tables for Tag History or Transaction Groups.

Foreign Key

A Foreign Key is a referential constraint between two tables. The foreign key identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table. The columns in the referencing table must be the primary key in the referenced table. For example, you might have a list of suppliers with an integer ID column. Then your invoices would use the supplier's ID instead of the name. These linked tables can save a lot of space because you don't have to include everything about the supplier in each invoice.

Example

Supplier (**SupplierNumber**, Name, Address, Type)
Invoices (InvoiceNumber, **SupplierNumber**, Text, Cost)

Null Value

NULL is a special marker used in SQL to indicate that a data value does not exist in the database. This way it is clear that there is no data, instead of guessing if a value of 0 is correct or just missing data. By definition, NULL is not equal to anything, even other NULL values. Every database has a function to identify if a value is NULL, usually called isNULL() or something similar.

Comments

Comments can be added to any SQL query just like with scripting. Single line comments are done with two dashes and a space: `--`

SQL - Single Line Comment

```
-- This is a single line comment in a SQL query.
SELECT * FROM my_table
```

You can also do multi line comments by wrapping text within a forward slash and an asterisk: `/* text */`

SQL - Multi Line Comment

```
/* A multi line comment
can span multiple lines. The
comment will stop when it is closed
with another asterisk and forward slash.*/
```

SQL Queries

SQL queries (or statements) are used to create, maintain, and view relational databases like MySQL, SQLServer, Oracle, etc. They follow a specific format and use just key words to determine the structure of the query. Unlike most coding languages, SQL does not rely on newlines or start/end markers for it's format, each query is a single line of code. You will often see SQL queries split over several lines, but that is just to make them easier to read.

> **INFO**
>
> You might notice a lot of CAPITALIZED words in SQL queries. While these key words are not case sensitive, it is still common practice for people to capitalize them in a query. Things like SELECT, FROM, WHERE (and a few others) are almost always capitalized because they have a special meaning. Try not to have table or column names that use any of these special words. You will see this capitalization format in examples throughout this user manual and other online references.

Select Command

The SELECT statement is used to select data from a database. The result is returned as a data set, called the result set. This is true even if there is only one value returned. The syntax for a SELECT statement is as follows:

SQL - The Select Command

```
SELECT myColumn FROM myTable
```

Select Examples

The "*" character can be used to specify all columns from a database table. While this is the easiest way to retrieve results from a table, this is not the recommended approach.

SQL - Selecting All Columns

```
SELECT * FROM Customers
```

The recommended approach is to instead specify only the columns that are required for a query. There are several reasons for this, but performance would be the main one: less columns in a statement means less work for the database, and the resulting data set in Ignition will use less memory.

SQL - Selecting a Single Column

```
SELECT Name FROM Customers
```

SQL - Selecting Multiple Columns

```
SELECT Name, Address FROM Customers
```

Where Clause

The WHERE clause is used in conjunction with other commands to extract only those records that fulfill a specified criterion. The WHERE clause usually goes near the end of the query followed by a condition that the values must meet to be returned.

Pseudocode - Where Clause

```
SELECT myColumn FROM myTable WHERE condition
```

The WHERE clause can use various operators for its condition, with the basic operators being:

Operator	Description
`=`	Equal to.
`<>`	Not equal to.
`>`	Greater than.
`<`	Less than.
`>=`	Greater than or equal to.
`<=`	Less than or equal to.

Where Examples

Only return customers from CA.

SQL - Simple Where Clause

```
SELECT * FROM Customers WHERE State = 'CA'
```

Only return users over a specified age.

SQL - Select Users over 25

```
SELECT * FROM Users WHERE Age > 25
```

Select all things from a table where the date column has a value within a certain range.

SQL - Select Column Between Two Dates

```

```

```
SELECT * FROM events WHERE date_column BETWEEN '2023-01-01' AND '2024-01-01'
```

Additional examples can be found on the **SQL Where Clauses** page.

Insert Into Command

The INSERT INTO statement is used to insert a new row in a table. If any columns have default values or are auto-incrementing, they can be omitted from the INSERT query.

Pseudocode - The Insert Command

```
INSERT INTO myTable (column1, column2)
VALUES ('Value1', 'Value2')
```

Insert Examples

SQL - Insert Values into Columns

```
INSERT INTO Customers (Name, Address, City, State, Zip, Country, Phone)
VALUES ('Inductive Automation', '90 Blue Ravine', 'Folsom', 'CA', '95630', United States, '1-800-266-7798')
```

If inserting a value into every column of the table, the columns do not need to be listed on the INSERT INTO statement. The values just need to be listed in the same order as the columns in the table. The table in the query below has four columns: id, first name, last name, and title.

SQL - Inserting to all Columns

```
INSERT INTO Users
VALUES (5628, 'Bob', 'Smith', 'Project Manager')
```

Update Command

The UPDATE statement is used to update existing records in a table. If a WHERE clause is not used, **all rows in the table will be updated**. As a result, the UPDATE statement should be used in conjunction with a WHERE clause in most cases. Many official management tools like SQLServer's Management studio will not allow UPDATE commands without a WHERE clause.

Pseudocode - The Update Command

```
UPDATE myTable SET myColumn = 'myValue'
```

Update Examples

SQL - Updating All Rows in a Single Column

```
UPDATE Customers SET Name = 'Inductive Automation'
```

SQL - Updating a Single Column for a Single Row

```
UPDATE Customers SET Address = '2110 21st Street' WHERE ID = 1
```

Delete Command

The DELETE statement is used to delete records in a table. **NEVER** run a delete command without a WHERE clause. It will delete **ALL** records from that table. Many official management tools like SQLServer's Management studio will not allow DELETE commands without a WHERE clause.

Pseudocode - The Delete Command

```
DELETE FROM myTable WHERE myColumn = value
```

Delete Examples

SQL - Deleting Based on Column Value

```
DELETE FROM Customers WHERE Name = 'Inductive Automation'
```

SQL - Deleting Rows Based on an ID Column

```
DELETE FROM Customers WHERE id < 538
```

SQL Select Statements

While the SELECT command in its basic form can be very simple to use, the SELECT statement can be used with other statements or in certain ways that allow you to bring in exactly the data you need.

Selecting Static Values

Static values can be inserted into a result set returned from a SELECT query as another column. Simply use the static value as a column to select, and the query will return a column where the name of the column is the static value, and every row in that column will return that same static value.

Pseudocode - Selecting Two Columns From a Table, and a Third Column with a Value of 10 for every row

```
SELECT column1, column2, 10 FROM table
```

Select Distinct Command

The SELECT DISTINCT statement works much like a SELECT statement works, in that it selects data from a database. However, SELECT DISTINCT will only return distinct or different values, not duplicates.

Pseudocode - The Select Distinct Command

```
SELECT DISTINCT column FROM table
```

Examples

This can be useful for getting a better idea of the range of values in a particular column.

SQL - Select Distinct Countries

```
SELECT DISTINCT country FROM Customers
```

Order By Clause

The ORDER BY keyword is used to sort the result-set by a specified column set of column. The ORDER BY keyword sorts the records in ascending (ASC) order by default. If you want to sort the records in a descending order, you can use the DESC keyword.

Pseudocode - Order By Clause

```
SELECT column1, column2 FROM table ORDER BY column2 DESC
```

Examples

SQL - Ordering by One Column

```
SELECT * FROM Customers ORDER BY Name ASC
```

You can use multiple columns to sort, this will sort by state first, and for each state the rows will be sorted by name.

SQL - Ordering by Multiple Columns

```
SELECT * FROM Customers ORDER BY State ASC, Name DESC
```

Limiting Rows Returned

SELECT commands can have the number of rows that the query returns limited using a special keyword. The keyword differs between database providers but the effect is the same, limiting the number of rows returned to a value that you specify.

Database	Keyword	Example

MS SQL Server/ MS Access	SELECT TOP value/percent	Note that the SELECT TOP command is unique in that you c specify a percentage value instead of an exact number of rec
		`SELECT TOP 200 column1, column2 FROM table`
		`SELECT TOP 10 PERCENT column1, column2 FROM table`

MySQL	Limit value	`SELECT column1, column2 FROM table LIMIT 200`
Oracle DB	ROWNUM <= value	Note that the ROWNUM command is unique in that you can to identify the row number for any reason, not just limiting the number of rows returned. `SELECT column1, column2 FROM table WHERE ROWNUM <=`

Aliases

In a SQL query, aliases are used to give columns or even tables a temporary name for that query. Simply place the keyword AS after a column or table, followed by the alias name. If the alias is two words, it needs to be encapsulated in single quotes.

Pseudocode - Aliasing Columns and the Table

```
SELECT column1 AS a, column2 AS 'b c' FROM table AS t
```

Examples

This can be really useful when the table has complex column names.

SQL - Aliasing Complex Column Names

```
SELECT id AS 'Badge Number', name AS 'Employee Name', dob AS Birthday FROM
employees
```

This can also be useful when using multiple tables in a query, such as with a JOIN.

SQL - Joining Columns from Two Tables with Aliases

```
SELECT * FROM Contacts AS co JOIN Customers AS cu ON cu.ID = co.CustomerID
```

Union Command

The UNION operator is used to combine the results of two different SELECT statements. This differs from a JOIN in that there does not have to be a relationship between columns. However, both SELECT statements need to select the same number of columns with similar data types in a similar order. So if my first statement selects an int column and then a string column, the second statement needs to do the same. The name of the columns in the resultset will take the name of the columns from the first SELECT in the UNION.

Pseudocode - Union Two Tables

```
SELECT stringCol, intCol FROM table1
UNION
SELECT stringCol, intCol FROM table2
```

Examples

By default, the UNION operator will only select distinct values between the two tables.

SQL - Union of Users and Customers

```
SELECT username FROM users
UNION
SELECT name FROM customers
```

To select all values from both tables, we can use UNION ALL instead.

SQL - Union All

```
```

```
SELECT jobTitle FROM jobs
UNION ALL
SELECT position FROM employees
```

Static values can be used in a UNION to help differentiate the rows from each table.

SQL - Differentiating Between Users and Customers

```
SELECT 'User' AS Type, username FROM users
UNION
SELECT 'Customer', name FROM customers
```

SQL Where Clauses

Overview

The SQL WHERE clause is utilized to restrict the number of rows impacted by a query. A WHERE clause is commonly utilized in two scenarios:

- In conjunction with a SELECT statement to filter the number of rows returned
- As part of an UPDATE or DELETE statements to restrict which rows are manipulated by the query.

In either scenario, the syntax of a WHERE clause is used the same, and can have multiple predicates:

Pseudocode - Where Clause Syntax

```
SELECT * FROM table WHERE column = value
```

Where-Clauses and Data Manipulation

In most cases, statements that modify the content of a database table via UPDATE or DELETE **should** include a WHERE clause: otherwise the manipulation will be applied to every row. To demonstrate, you typically want to avoid queries that look like the following:

Pseudocode - Never do this!

```
UPDATE table SET column = 'This was a horrible mistake'
```

When manually modifying a database table, a good habit to develop involves first writing a SELECT statement. If you can successfully write a WHERE clause that only returns the results you need to modify, then you can simply change the rest of your query to manipulate the table. Thus, we could write a query like the following:

Pseudocode - Select-Statement with a Where Clause

```
SELECT * FROM table WHERE id in (100,101,150,174)
```

If we receive only the results we need to modify in the query above, we can make a simple modification to our query to now delete just the rows we want.

Pseudocode - Delete-Statement with a Where Clause

```
DELETE FROM table WHERE id in (100,101,150,174)
```

Another common use of the WHERE clause is is to search through records and return the ones during a particular time frame. Timestamp columns can use the < and > operators to compare to each other, and certain string formats can be compared to timestamps like this query:

Pseudocode - Where Clause with a Timestamp

```
SELECT * FROM table WHERE t_stamp > '1984-01-25 16:35:55'
```

And Operator

The AND operator allows you to specify two or more conditions in a WHERE clause, where each condition must be true for the row to be returned.

Pseudocode - Using And to Specify Multiple Conditions

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND column2 < value AND column3 = value
```

Example

This helps to narrow down the result set even further by adding in additional conditions that must be met. This will only return rows for customers from Germany who are also over 20 years old.

SQL - Customers from Germany Over 20

```
SELECT * FROM customers WHERE country = 'Germany' AND age > 20
```

Or Operator

The OR operator allows you to specify two or more conditions in a WHERE clause, but only one of the conditions need to be true for the row to be returned.

Pseudocode - Using Or to Specify Multiple Conditions

```
SELECT column1, column2, column3 FROM table WHERE column1 > value OR column2 < value OR column3 = value
```

Example

The OR operator can help pull in data from two different subsets in the table. This will only return rows for customers from Germany or customers who are over 20 years old.

SQL - Customers from Germany or Customers Over 20

```
SELECT * FROM customers WHERE country = 'Germany' OR age > 20
```

Not Condition

The NOT condition allows you to specify a condition that must not be met for the row to be returned.

Pseudocode - Using Not to Specify a Condition that Shouldn't be met

```
SELECT column1, column2, column3 FROM table WHERE NOT column1 = value
```

Example

This can be useful for finding all data other than a certain subset. This will return all customers who are not from Germany.

SQL - Customers that are not from Germany

```
SELECT * FROM customers WHERE NOT country = 'Germany'
```

Between Operator

The BETWEEN condition allows you to specify a range of values separated by an AND that the value must be in for a condition to be true. The value can be numbers, text or dates and is inclusive of the first and last values in the range.

Pseudocode - Using Between to Specify a Range of Values

```
SELECT column1, column2, column3 FROM table WHERE column1 BETWEEN value1 AND value2
```

Examples

SQL - Customers that are Between the Ages of 20 and 40

```
SELECT * FROM customers WHERE age BETWEEN 20 AND 40
```

Note that the BETWEEN operator would work similarly to using a greater than or equal condition and a less than or equal condition.

SQL - Customers that are Between the Ages of 20 and 40 with no Between Operator

```
SELECT * FROM customers WHERE age >= 20 AND age <= 40
```

Timestamps can also use the BETWEEN operator to check for a given start time and end time.

SQL - Customers that are Between the Ages of 20 and 40 with no Between Operator

```
SELECT * FROM customers WHERE start_time BETWEEN '1984-01-25 00:00:00' AND '1984-01-25 16:35:55'
```

Like Condition

The LIKE condition allows you to specify a condition that must meet a certain pattern. Typically used to compare to string values, the pattern can be built using a combination of characters and the two wildcard values.

- **%** - Used to specify any number of any characters including zero characters.

- _ - Used to specify exactly one character.

Pattern Examples

Pattern	Meaning	Possible Matches
'%a%'	Values that have an 'a' in them.	'a', 'Inductive Automation', 'almost', 'create'
'_a_'	Values that have an 'a' with exactly one character before and after the 'a'.	'bat', 'cat', 'can'
'_a%'	Values that have an 'a' as the second character.	'da', 'saw', 'catcher'
'a%t'	Values that start with 'a' and end with 't'.	'about', 'at'
'%a%_%_%_'	Values that contain an 'a' with at least 3 other characters after it.	'trains', 'airplane', 'canteen'
'%a%a%'	Values that contain at least two 'a' characters in them.	'Inductive Automation', 'separate', 'apart'

Once the pattern has been constructed, it can be used with the LIKE operator to find values that match the specified pattern.

Pseudocode - Using Like to Specify a Pattern of Values

```
SELECT column1, column2, column3 FROM table WHERE column1 LIKE '%a%'
```

Example

The LIKE operator can be used to find all values that match a criteria, such as all countries with 'land' in that name

SQL - Customers that are not from Countries with 'land' in the Name

```
SELECT * FROM customers WHERE country LIKE '%land%'
```

In Condition

The IN operator allows you to specify a subset of values, with the condition that the return match at least one of them. Using an IN operator is similar to using multiple OR operators for the same column.

Pseudocode - Using IN to Specify Multiple Values

```
SELECT column1, column2, column3 FROM table WHERE column1 IN (value1, value2, value3)
```

Examples

The IN can be used as a shorthand way of writing out multiple conditions for the same column separated by OR operators. This would select all values where the country is either Germany, France, or USA.

SQL - Customers from Germany or France or USA

```
SELECT * FROM customers WHERE country IN ('Germany', 'France', 'USA')
```

This would be similar to doing something like this.

SQL - Customers from Germany or France or USA

```
SELECT * FROM customers WHERE country = 'Germany' OR country = 'France' OR country = 'USA'
```

The real power of the IN operator is that instead of specifying static values, an entirely new query can be run to compare values against.

SQL - Customers from Countries that users are also in

```
SELECT * FROM customers WHERE country IN (SELECT country FROM users)
```

Combining Multiple Operators

Multiple AND and OR operators can be combined to specify multiple different conditions that need to be met in order for a particular row to be returned. Additionally, each condition can be simple using the

mathematical operators or complex using the conditions listed above. When using AND and OR operators in a WHERE clause, the AND will take precedence, evaluating first before the OR. In the pseudocode below, the row will be returned if either both the first and second conditions are met, or the third condition is met.

Pseudocode - Using Multiple Where Clause Operators

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND column2 < value OR column3 = value
```

However, the order at which the operators get evaluated can change by placing parentheses around the conditions which should be evaluated first. In the pseudocode below, the row will be returned if both the first condition is met, and either the second or third condition is met.

Pseudocode - Using Multiple Where Clause Operators with Parentheses

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND (column2 < value OR column3 = value)
```

Examples

We can use complex conditions with different operators to find all customers who are over the age of 50 in a country that has 'land' in the name, or any customers in Germany or France.

SQL - Multiple Complex Conditions

```
SELECT * FROM customers WHERE country LIKE '%land%' AND age > 50 OR country IN ('Germany', 'France')
```

Using parentheses in the same query can drastically change what valid return conditions are. Here, the customer must both be from a country with 'land' in the name, as well as either over 50 or from Germany or France.

SQL - Multiple Complex Conditions

```
SELECT * FROM customers WHERE country LIKE '%land%' AND (age > 50 OR country IN ('Germany', 'France'))
```

SQL Table Joins

Overview

The SQL JOIN allows you to run a single SELECT statement that references multiple tables. This can be used for more advanced filtering, as well as combining data from multiple tables in a single result set. The process of joining two tables involves stating both tables in the query, and then specifying that a column from one table relates to another in some way.

Joins may look imposing at first, but they are simply SELECT statements that utilize columns from multiple tables.

The JOIN keyword works in conjunction with a SELECT statement. However, there are some key concepts that must be addressed when attempting to use the JOIN keyword.

Specifying Each Column

When listing column in a statement that uses the JOIN keyword, you must denote which table each column is being retrieved from. You can do this by using an **Alias**, or with the fully qualified column name. This prevents ambiguous columns in the context of the query, and makes it easier for you to use other keywords in the statement: i.e., adding WHERE clauses that apply to multiple tables.

Pseudocode - Fully Qualified Column Name

```
table_name.column_name
```

Declare the Relation

After the JOIN keyword, you must state which columns from each table relate to each other. This is accomplished by stating the name of the table, using the ON keyword, and then stating that a column on the first table is equal to a column on the second table. The columns specified are typically **primary keys** for their respective tables.

In the example below, we're stating that the values in **some_column** on **tableA** should be associated with matching values in **some_other_column** on **tableB**.

Pseudocode - Using the Join Keyword

```

```

```
FROM
    TableA
JOIN tableB ON tableA.some_column = tableB.some_other_column
```

The order you present the columns **after** the ON clause does not matter: A = B is equivalent to saying B = A, so we could switch the columns listed with no distinguishable impact on the resulting query

Combining the Concepts

Altogether, a basic JOIN looks like the following:

Pseudocode - Join Syntax

```
SELECT
    tableA.column,
    tableB.column
FROM
    tableA
JOIN tableB ON tableA.identity_column = tableB.identity_column
```

Joins with Three or More Tables

Joins can even be done between three or more tables. The syntax is similar, with each new table relation declared below the first.

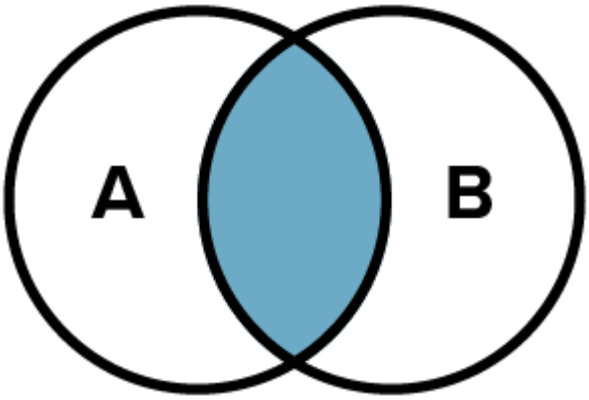
Pseudocode - Join Syntax

```
SELECT
    tableA.column,
    tableB.column,
    tableC.column
FROM
    tableA
JOIN tableB ON tableA.identity_column = tableB.identity_column
JOIN tableC ON tableA.identity_column = tableC.identity_column
```

Join

The standard JOIN, also referred to as INNER JOIN, will only return rows where the joined columns contain matching values. If one of the joined columns contains a value that is not present in the in the

other, then the row is not represented in the result set. You would use JOIN when you only want results that are represented in both tables.



This section will demonstrate the various uses of the JOIN keyword. For the sake of clarity, the queries will run against tables that look like the following:

Products Table

id	product_name
1	Apples
2	Oranges
3	Grapes
4	Plums

Inventory Table

id	product_id	quantity	product_vendor
1	1	15	Apple Corp
2	2	25	Orange Ya-Glad
3	3	56	Grape Escape
4	5	45	Banana Solutions

Join in Action

In this demonstration, only rows that pertain to Apples, Oranges, and Grapes are being returned. We're using a JOIN between **products.id** and **inventory.product_id**, so our results will only contain rows that have matching values from both of those columns. Our result set does not contain any information on products with **products.id** values of 4 or **inventory.product_id** values of 5, because those values are not present in **both** of the joined columns.

SQL - Joining Products and Inventory

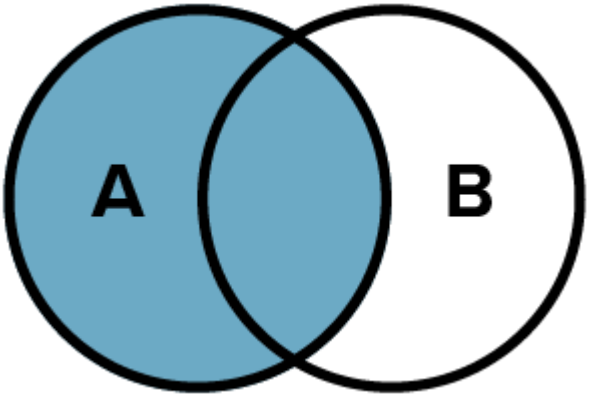
```sql
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56

Left Join

Return all rows from the left-most table (**table A** in the diagram), even if there are no matches on the right-most table (**table B**). If there isn't a matching record in the right table, then NULL values are returned.



Left Join in Action

Here we see all rows returned from our products table (since it is the left-most table in our query). In row 4, columns that are being populated via the inventory table (**product_id**, **product_vendor**, and **quantity**) contain NULL values, because there isn't a row on the inventory table that matches with a **product_id** value of 4. The query must return something in this case, so it returns NULL for these columns.

SQL - Left Joining Products and Inventory

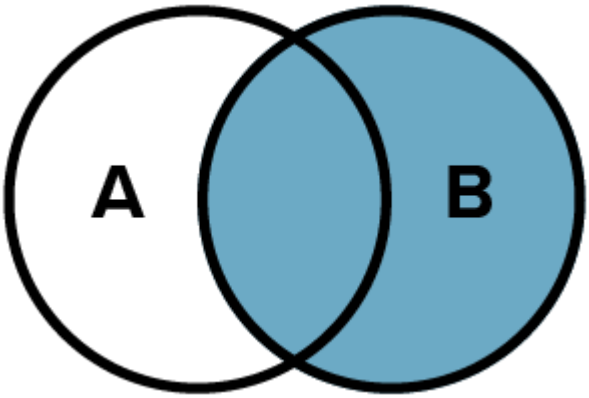
```sql
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
LEFT JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
4	NULL	Plums	NULL	NULL

Right Join

Return all rows from the right-most table (**table B**), even if there are no matches on the left-most table (**table A**). If there isn't a matching record on the left table, then NULL values are returned.



Right Join in Action

When using a RIGHT JOIN, all rows will be returned from the inventory table. The products table does not have a row that contains an **id** value of 5, so the **id** and **product_name** columns will show NULL values in our result set.

SQL - Right Joining Products and Inventory

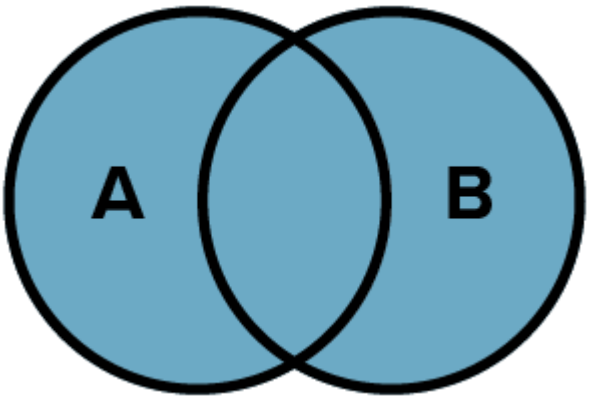
```
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
RIGHT JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
NULL	5	NULL	Banana Solutions	45

Full Join

The FULL JOIN returns all rows from both tables, regardless if there are matching values in the joined columns. You would use a FULL JOIN in cases where you want to show all applicable records from both tables, and synchronize the data across both tables via the joining columns where possible.



Full Join in Action

Note that we're using the same query as the standard JOIN, but we've prepended "FULL" to the last line of the query. Note the NULL values returned in cases where our product ID is not fully represented on both tables.

SQL - Full Joining Products and Inventory

```
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
FULL JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
4	NULL	Plums	NULL	NULL
NULL	5	NULL	Banana Solutions	45

Full Joins in MySQL

MySQL does not have an equivalent FULL JOIN. However, you can emulate one by utilizing a LEFT JOIN, RIGHT JOIN, and the UNION keyword. For the sake of simplicity, we will return all columns in the following example, but you would still want to specify individual columns in both SELECT query.

SQL - Full Join in MySQL using Left Join, Right Join, and Union Keyword

```
SELECT * FROM products
LEFT JOIN inventory ON products.id = inventory.product_id

UNION ALL

SELECT * FROM products
RIGHT JOIN inventory ON products.id = inventory.product_id
WHERE products.id IS NULL
```

SQL Common Functions

Functions are available in most SQL databases, and can provide some helpful utility to any queries you may be calling in Ignition.

This page contains some commonly used SQL functions that some databases contain. The exact functions available and usage depends on the database, so always check your database's documentation for a more complete list of available functions.

Using Column Values

It is important to understand that when calling these functions, you generally use a column name instead of a static number. For the sake of simplicity, the tables below demonstrate how to use the functions with static values, but they usually are switched for column names, ie:

Pseudocode - Passing a Column to a Function

```
SELECT SUM(downtime_duration) FROM downtime_events
```

Example Table

Some of the functions on this table are better demonstrated when used in conjunction with a table (i.e., using the AVG() function with a single value isn't too interesting). Thus, the following table contains sample data that the functions on this page will utilize if necessary.

Products Table

id	product_quantity	product_name	date_added	date_updated	or
1	100	apples	Mon Jan 29 00:00:00 PST 2018	Mon Jan 29 12:00:00 PST 2018	Ca
2	24	oranges	Mon Feb 13 00:00:00 PST 2017	Mon Feb 13 09:00:00 PST 2017	Fl
3	56	grapes	Mon Mar 07 00:00:00 PST 2016	Mon Mar 07 05:00:00 PST 2016	Ca

Numeric Functions

Function	Description	Example	Output
ABS(value)	Returns the absolute value of the passed number or column.	`SELECT ABS(-3.5)`	3.5
AVG(value)	Takes the values of a single numeric column, and returns an average. A WHERE clause may be used in the same statement to filter out some of the rows on the table.	`SELECT AVG(product_quantity) FROM products`	60
CEILING(value)	Returns the next greatest integer value based on the argument provided. Thus, CEILING(10.1) would return 11.	`SELECT CEILING(10.1)`	11
COUNT(value)	Returns a row count. Typically takes either a single column, *, or 1. Regardless of which row is passed, the function will return the number of rows on the table that meet the criteria of any WHERE clauses.	`SELECT COUNT(*) FROM products`	3
FLOOR(value)	Returns the next smallest integer value based on the argument provided. Thus, FLOOR(10.9) would return 10.	`SELECT FLOOR(10.9)`	10

Function	Description	Example	Output
MAX(value)	Returns the largest value from the specified column.	`SELECT MAX(product_quantity) FROM products`	100
MIN(value)	Returns the smallest value from the specified column.	`SELECT MIN(product_quantity) FROM products`	24
ROUND(value, decimal_places)	Returns a number rounded to a certain number of decimal places. Takes two parameters. The first is the number to round to, and the second is the number of decimal places to round to.	`SELECT ROUND(1.234, 1)`	1.2
SUM(value)	Takes the value of a single numeric column, and returns the sum. A WHERE clause may be used in the same statement to filter out some of the rows on the table.	`SELECT SUM(product_quantity) FROM products`	180

String Functions

Function	Description	Example
CONCAT(value1, value 2,....valueN)	Concatenates multiple strings or values. Some databases may require you to convert each value to a string before concatenating.	`SELECT CONCAT(product_name, ':', product FROM products`
LOWER(value)	Converts a string to lowercase.	`SELECT LOWER('MAKE Me smALL')`
LTRIM(value)	Removes leading space from a string.	`SELECTCT LTRIM(' Take a little off the le`
REPLACE(orignal_string, target_string, replacement_string)	Searchings a string for a substring (target_string), and replaces the substring with the replacement_string.	`SELECT REPLACE('Who is awesome', 'Who i`
RTRIM(value)	Removes leading space from a string	`SELECT RTRIM('Take a little off the rig`
SUBSTRING(orignal_string, character_index, [length])	Extracts a substring from another string based on character index. Takes two parameters: the original string, and the character index	`SELECT SUBSTRING('This is my string!' ,`

Function	Description	Example
	to start at. An optional third parameter can specify the number of characters to extract. Character index is one-based, so the first character in the string rests at index 1.	
TRIM(value)	Removes both leading and trailing space from a string.	`SELECT TRIM(' Trim Both Sides ')`
UPPER(value)	Converts a string to uppercase.	`SELECT UPPER('super size me')`

Date Functions

There are many date and time functions for each database (MySQL, MSSQL, Oracle, etc), but they all vary wildly. These examples work in most databases:

Function	Description	Example	C
CURRENT_TIMESTAMP()	Returns the current date and time, as reported by the database.	`SELECT CURRENT_TIMESTAMP()`	R th c ti
TIMEDIFF(date1, date2)	Returns a difference between two dates. Assumes thate date1 is the most recent datetime.	`SELECT TIMEDIFF(date_updated, date_added) FROM products WHERE id = 1`	T 0 1 F 1

Logic Functions

Function	Description	Example	Output
COALESCE(value1, value2,...valueN)	Returns the first non-null expression.	`SELECT COALESCE(NULL, 'Pick me!')`	Pick me!

ISNULL(expression)	Returns true if an expression is NULL.	`SELECT ISNULL(NULL)`
		
		`SELECT ISNULL(14)`
			True
			
			False

NULLIF(expression1, expression2)	Compares two expressions. If they are equal to each other, then the function returns a NULL. If the two expressions are not equal, the first expression passed to NULLIF() is returned.	`SELECT NULLIF(100, 100)`
		
		`SELECT NULLIF(100, 3)`
			NULL
			
			100

Group By Clause

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. This way you can find the MIN, MAX, Average, COUNT, etc., for each group of rows. Grouping can also be done for multiple columns, with precedence going in the order that they are listed.

Pseudocode - Passing a Column to a Function

```
SELECT SUM(column1) FROM table GROUP BY column2
```

Example

Here, we are grabbing the sum of product quantity for each origin state.

SQL - Simple Group By

```
SELECT SUM(product_quantity), origin_state, FROM products_table GROUP BY
origin_state
```

SQL Stored Procedures

For the uninitiated, Stored Procedures are a series of predefined SQL statements that are configured and stored in a database. They can be easily called from another application, such as Ignition, with a single statement. Conceptually, they are very similar to a scripting function as parameters may be passed to them, they can utilize looping and conditional logic, and they can return a value.

Stored Procedure Syntax

Stored Procedures are created and maintained in your database. As a result, the commands used to create a Stored Procedure vary per database. In regards to calling a Stored Procedure, the syntax can also differ.

For instance, if you have a Postgres database connection, calls have to get translated to the native SQL by the JDBC driver. This is done through the `escapeSyntaxCallMode` configuration property that treats this syntax like a Procedure call rather than a Function call, and translates it to SQL accordingly. However, if `escapeSyntaxCallMode=callIfNoReturn` isn't provided to the JDBC connection string, the Stored Procedure execution will fail when using the `system.db.execSProcCall` scripting function or a **Stored Procedure Transaction Group**. Alternatively, you can execute via a manually written query using CALL as **shown below**, which will work as expected.

SQL Server

The EXEC command is utilized to execute a Stored Procedure.

Pseudocode - Executing a Stored Procedure in SQL Server

```
EXEC dbo.myStoredProcedure
```

Parameters may be passed to the Stored Procedure. SQL Server's documentation has more details on utilizing parameters with Stored Procedures.

Pseudocode - Executing a Stored Procedure in SQL Server with Parameters

```
EXEC dbo.myStoredProcedure @myParam = 100, @AnotherParameters = 'Hello'
```

MySQL

MySQL uses the CALL command to execute a Stored Procedure. Note the parentheses characters at the end, as they must be present even when the Stored Procedure is not utilizing parameters.

Pseudocode - Executing a Stored Procedure in MySQL

```
CALL myStoredProcedure()
```

If parameters are defined, they can be passed via the parenthesis, similar to how functions work in Python.

Pseudocode - Executing a Stored Procedure in MySQL with Multiple Parameters

```
CALL myStoredProcedure(100, 'Hello')
```

For information on the creation of a Stored Procedure, as well as proper SQL syntax to call a Stored Procedure, reference your database's documentation. Alternatively, if you have a database administrator, they can typically help with the creation and execution of a Stored Procedure.

Calling Stored Procedures in Ignition

There are several locations in Ignition where Stored Procedures may be utilized.

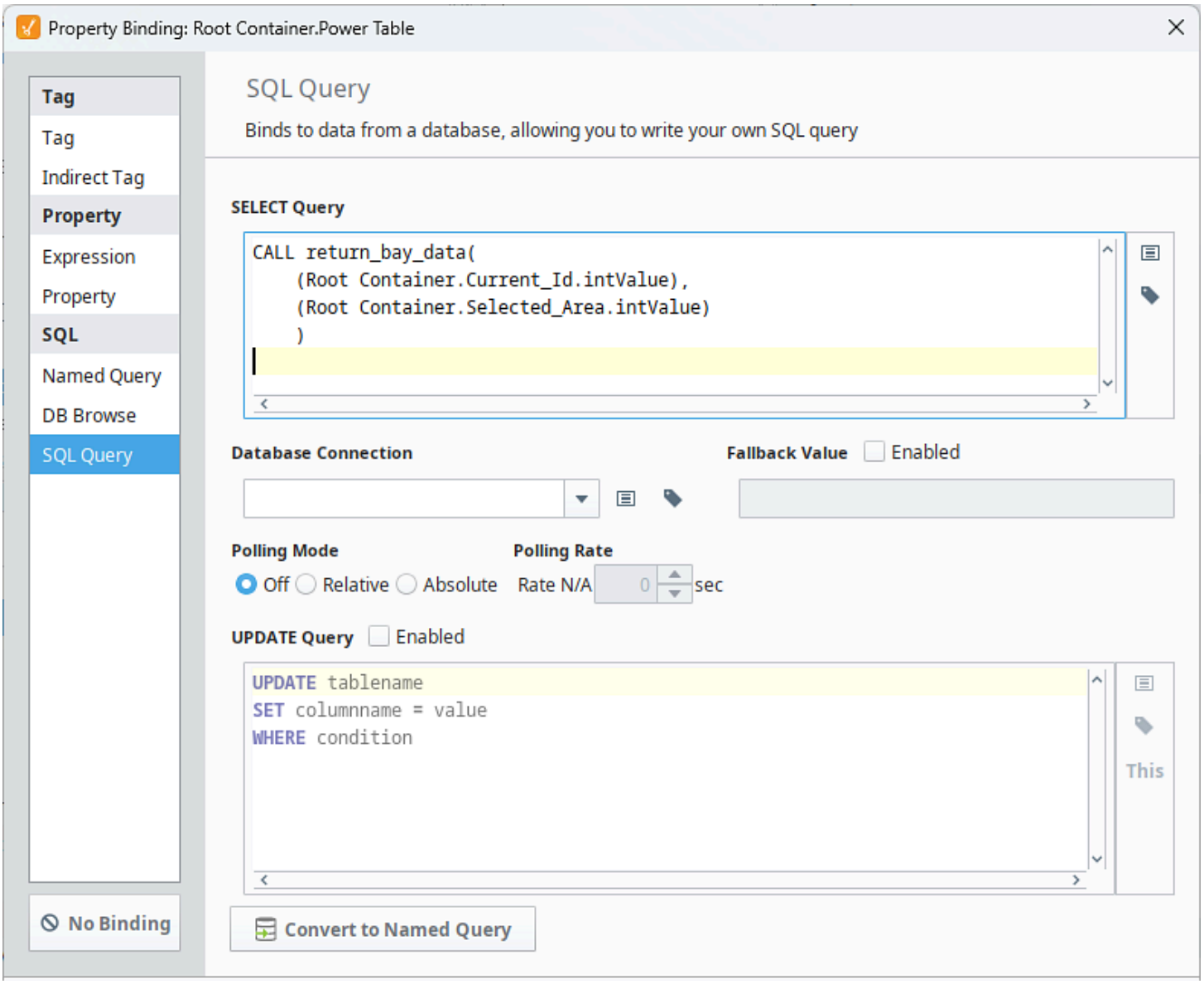
SQL Query Bindings

Instead of typing a query directly into a **SQL Query binding**, a Stored Procedure may be executed instead. Assuming a MySQL database contains a Stored Procedure named 'return_all_bays', we can call the procedure on a binding with the following:

SQL - Calling a Stored Procedure in MySQL

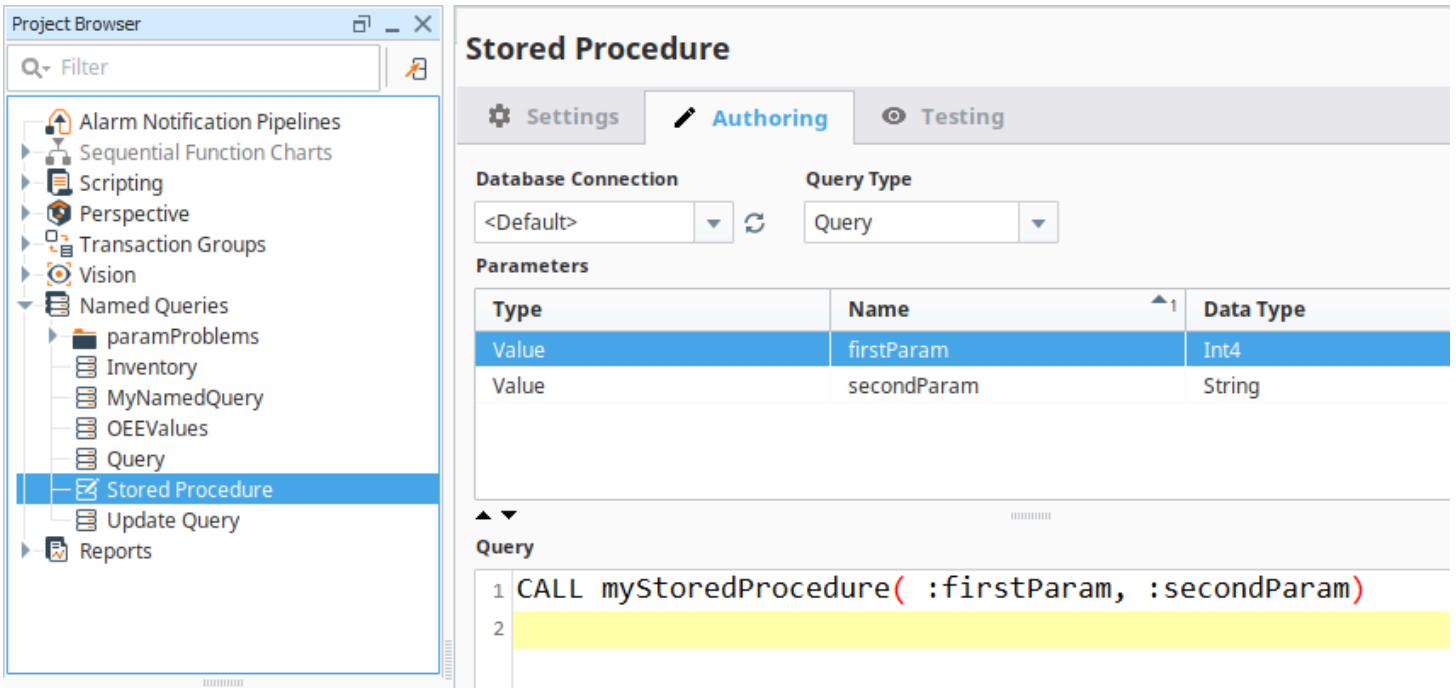
```
CALL return_all_bays()
```

As with all bindings, Tag and Property references may be utilized by clicking the appropriate icons in the binding window.



Named Queries

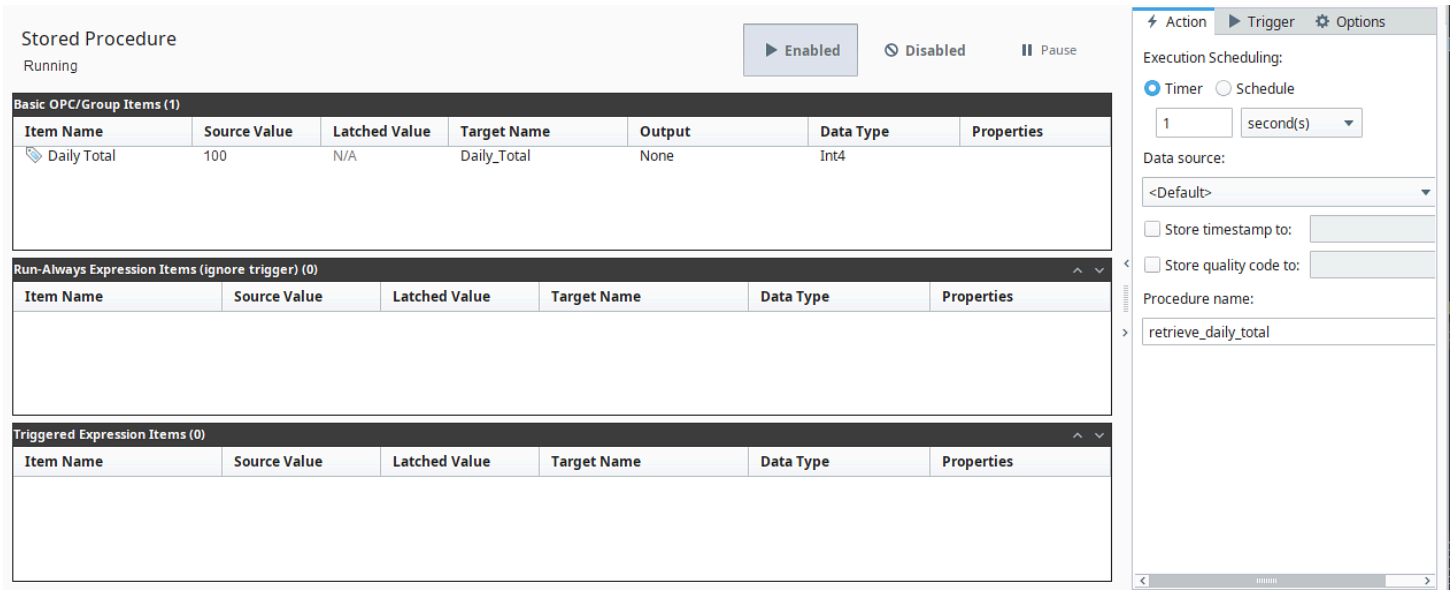
Named Queries support Stored Procedure calls. As mentioned under **Stored Procedure Syntax**, the syntax is based entirely on how your database expects a Stored Procedure to be called. Be mindful of the **Query Type setting**, as it has to match what the Stored Procedure is doing. If it is returning a result set, leave it set to Query. If it is modifying a record in the database, then set the type to Insert Query.



Stored Procedure Groups

One of the easiest ways to utilize tags with a Stored Procedure is to use the **Stored Procedure Group**. Parameters can be assigned to each item in the group, and utilize all of the features of a **Transaction Group**, such as scheduled execution and triggers.

Each item in the group is linked to a specific parameter in the Stored Procedure. Any IN or INOUT parameters can write directly to the tags, while new values can be fed into OUT and INOUT parameters allowing you to easily move data from tags into the database with the Stored Procedure.



Stored Procedures in Python Scripts

There are a few ways to call a Stored Procedure from a script in Ignition.

Using createSProcCall and execSProcCall

The recommended approach to calling a Stored Procedure from a Python script in Ignition typically involves two main steps:

1. Calling **system.db.createSProcCall** to create a call context, or object that effectively represents the impending Stored Procedure call. This object can be used to specify parameter values that will be passed to the Stored Procedure.
2. Using **system.db.execSProcCall** to execute the Stored Procedure.

Once the Stored Procedure has been executed, the call context generated in step 1 can be used to reference any values that were returned.

> **OTHER SYSTEM FUNCTIONS**
>
> Although, most other system functions in the `db` library, like **system.db.runPrepQuery**, can be used to call a Stored Procedure, we generally recommend against this. More functionality is built into **system.db.createSProcCall** and **system.db.execSProcCall** making them better suited to work with Stored Procedures.

Column Alias Extra Connection Property

If you create a stored procedure that is supposed to return a dataset with aliased column names, Ignition may have an issue and will instead use the original column names. An example of this would be executing a procedure call and reading in the result set, like so:

```
call = system.db.createSProcCall("procedure_name")
system.db.execSProcCall(call)
result_set = call.getResultSet()
```

To make sure the aliased column names are used, navigate to **Connections** > **Databases** > **Connections** on the Gateway. Expand the three dots menu for the MySQL database connection that is having issues. Scroll down to **Extra Connection Properties** and add the following parameter to any existing parameters:

`useOldAliasMetadataBehavior=true`

Click **Save Changes** at the bottom of the Edit panel to apply the change.

Query Builder

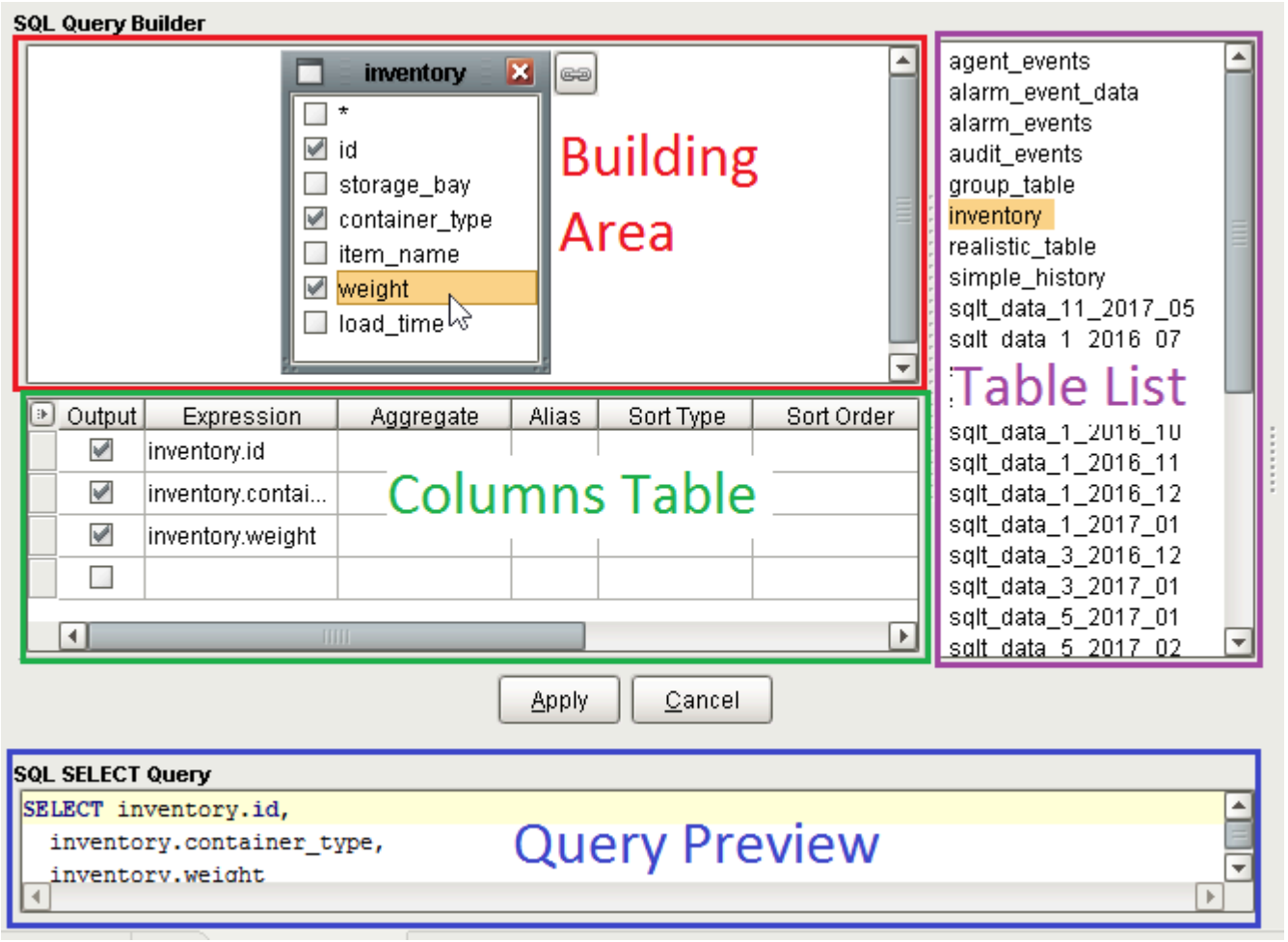
Crafting Queries with the Query Builder

Many places in Ignition that allow for SQL queries have a link to the SQL Query Builder tool. The Query Builder is a powerful Drag-and-Drop query building GUI that allows you to make complex queries from your connected databases. While a basic understanding of SQL helps make the most of the Query Builder tool, most people will have no problem creating effective queries after a brief tutorial. Additionally, the Query Builder does go over many advanced features of SQL that may be unfamiliar. We suggest looking up how these work in your favorite SQL resource guide as this covers how to use them, not what they do.

Builder Anatomy

Once opened, the Query Builder has the following items:

- **Building Area**: Visualizes the table relationships of all tables included in the query. Tables are typically dragged in from the **Table List**.
- **Columns Table**: Shows which columns from all tables are referenced in the query.
- **Table List**: Shows all database tables in the selected Database Connection.
- **Query Preview**: Shows a preview of the query that will be created once the **Apply** button has been pressed.



Opening the Query Builder

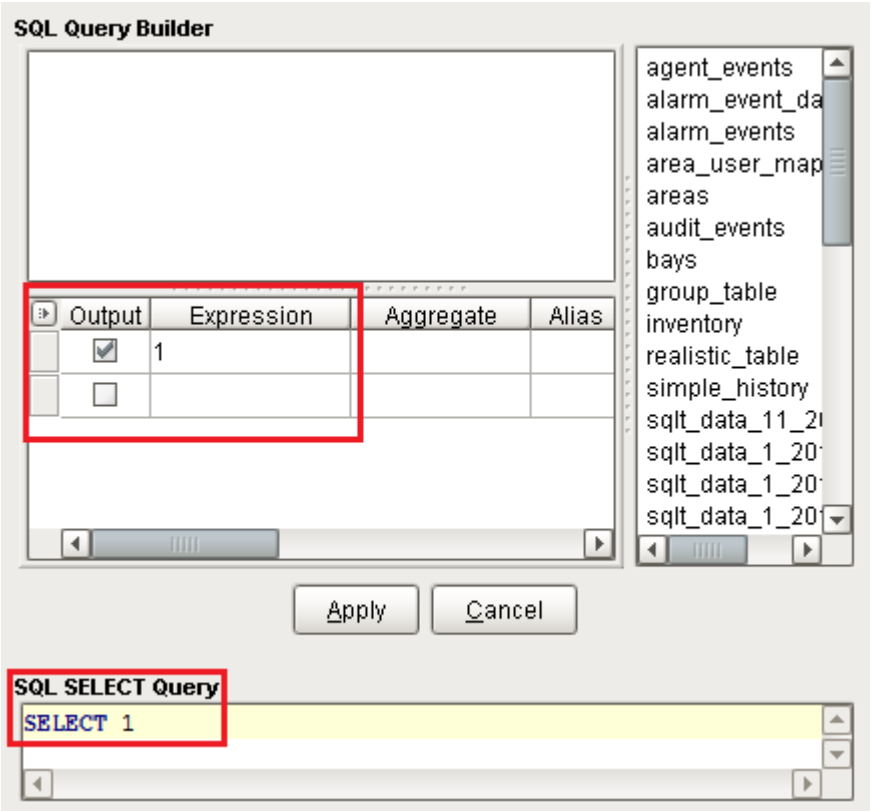
The Builder can be accessed from several different resources in Ignition, notably when using **Named Queries** and setting up **Report Data**. To use the Builder, click the **Show Builder** button.

Clicking this button will open the Builder workspace with an empty query. If a query was already written before the button was pressed, then the Builder will attempt to load the query into the Builder's interface. For example, if the following query was typed before pressing the button:

SQL - Select Statement - Loading into the Builder's Workspace

```
SELECT 1
```

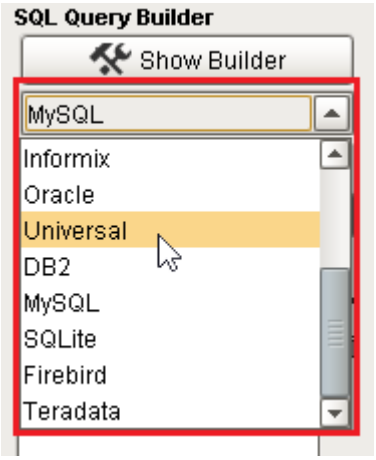
The following would be loaded into the Builder's workspace. Note that the **Building Area** is empty, because we're not querying from a table.



The Builder attempts to parse your query based on the **Syntax Parser**.

Using the Syntax Parser

When starting with a pre-existing query, the Syntax Parser tells the Builder which implementation of SQL your query's syntax is using. In most cases, this can be left with the default value of **Universal**. When set to a specific implementation of SQL, this allows the Builder to understand/accept implementation-specific keywords and syntax.



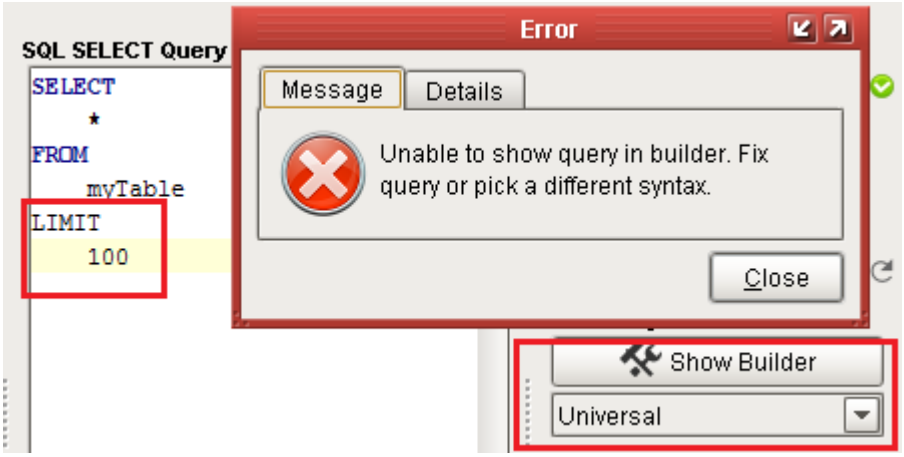
Syntax Parser in Action

In MySQL, query results may be limited with the **LIMIT** keyword:

SQL - Limit Statement

```
SELECT
    *
FROM
    myTable
LIMIT
    100
```

Attempting to open the Builder while using **LIMIT**, and the Syntax Parser is configured to a syntax that does recognize the **LIMIT** keyword, will result in an error:

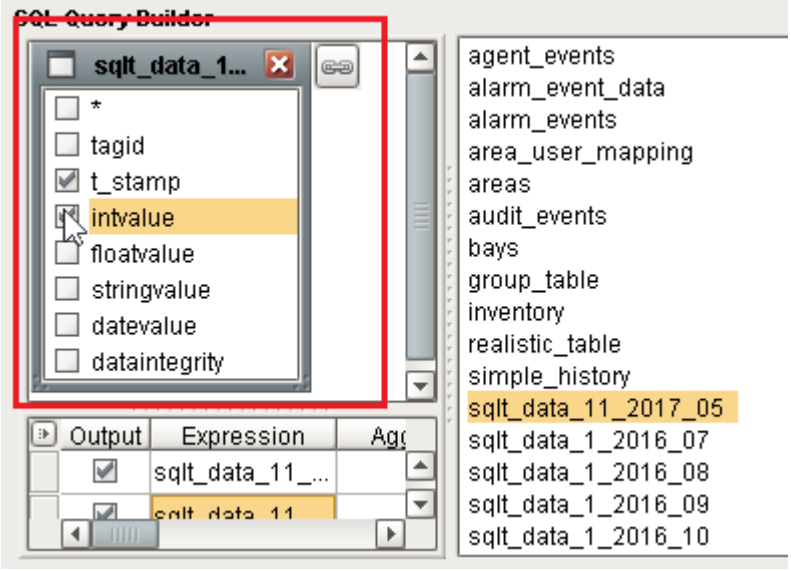


In these scenarios, you can switch to a syntax that supports the query, or remove the offending lines.

Using the Builder

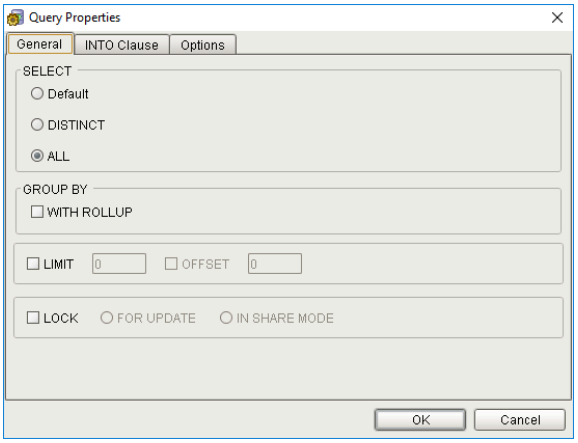
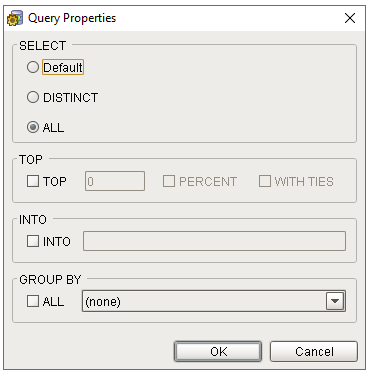
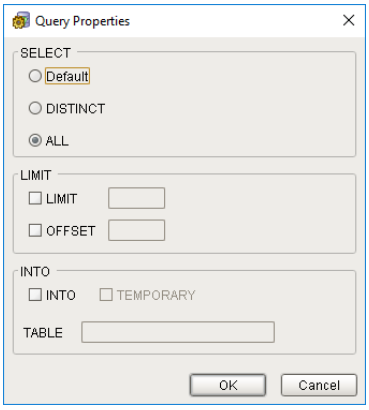
Once the Builder is open, (and if you have tables in your database connection), you can start building a query by dragging and dropping a table from the Table List into the Building Area. From here, you can select which columns on the Table Object you want to bring into the query, or drag and drop them into the Columns Table below.

The * option at the top of the table is special. If the * character is selected, then all columns from the table will be included in the results just like a "SELECT * " query.



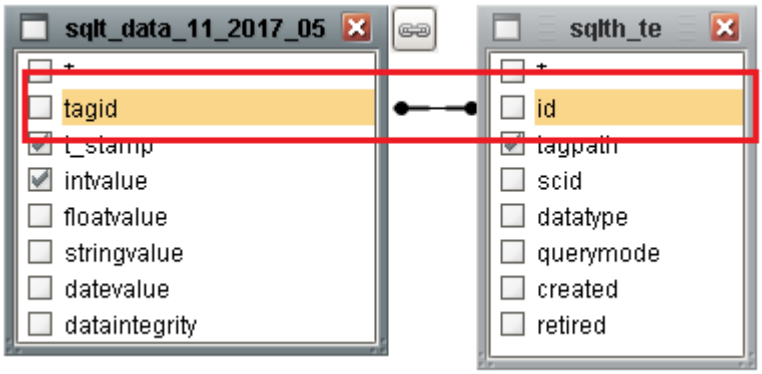
Query Properties

Right clicking on the empty space in the Building Area and selecting properties will bring up the Query Properties window that allows you to customize how the query works. How it looks and what it contains can vary, depending on what syntax parser you have selected.

MySQL Query Properties	MSSQL Query Properties	PostgreSQL Query Properties
		

Joining Tables in the Query Builder

Multiple tables may be added to the Building Area. Once there, the Builder can JOIN the two tables by dragging from one column in a table on to another column on a different table.

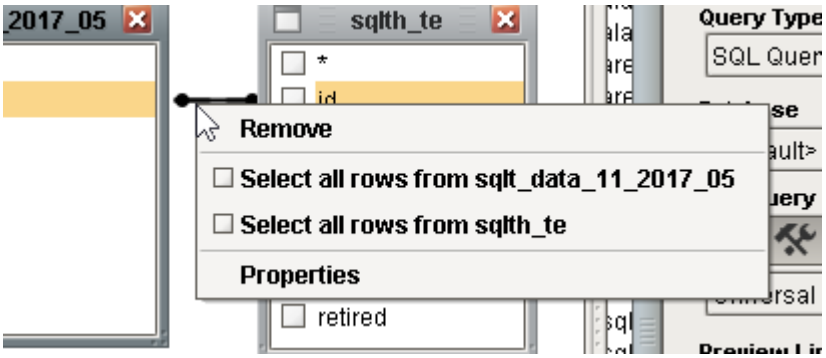


Note that "tagid" and "id" are now linked together. This means the resulting query will JOIN the two tables based on the linked columns. Much like a SQL JOIN, the joining columns do not need to be selected to be utilized in the JOIN.

Joins Right-Click Menu

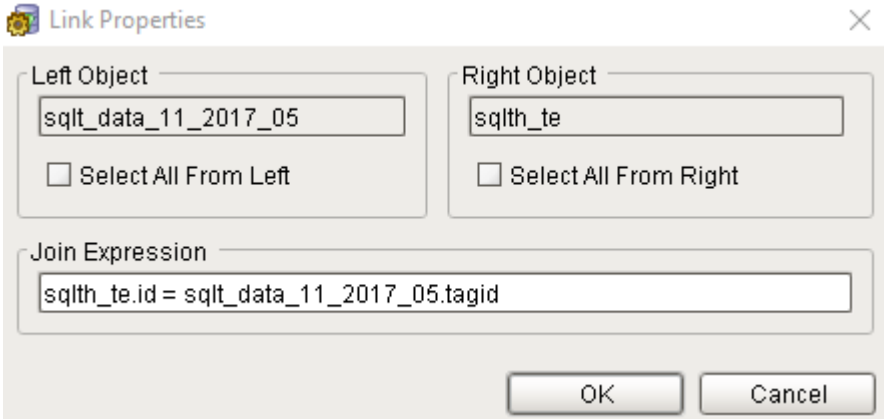
Right-clicking on the link will create a menu that allows you to remove the **JOIN**, as well as access properties of the JOIN.

Additionally, you can select all rows from one of the joining tables, which effectively creates a **RIGHT OUTER JOIN** or **LEFT OUTER JOIN**, depending on which table you select.



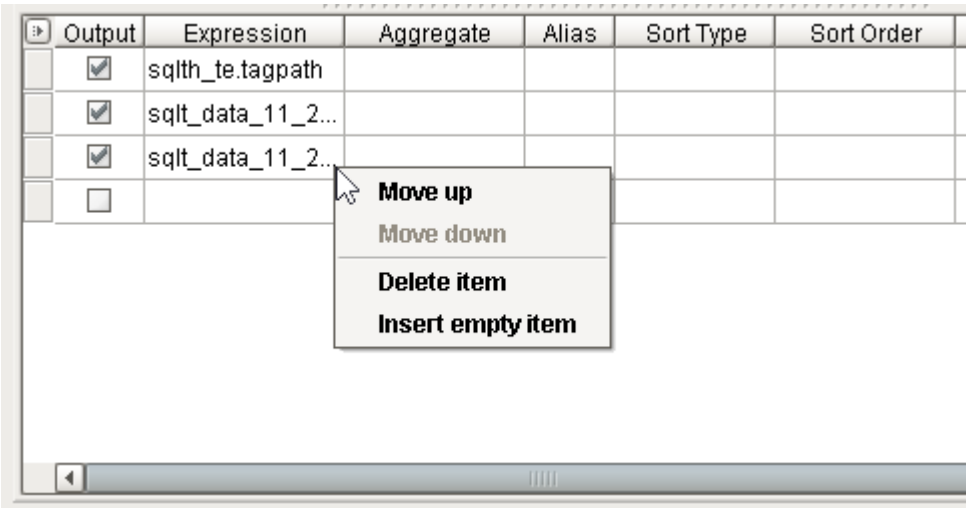
Join Properties

Clicking on **Properties** on the Right-Click Menu allows you to view the relation between each table, as well as apply **LEFT** or **RIGHT JOINs**, as mentioned above.



Columns Table

The Columns Table allows you to modify individual columns in the query. Typically, this allows you to add aggregates or aliases to each column from the Builder. Each row represents an expression, or combination of functions, columns, variables, and constants that will ultimately be a single column in the resulting query.



Right-Clicking on an Item

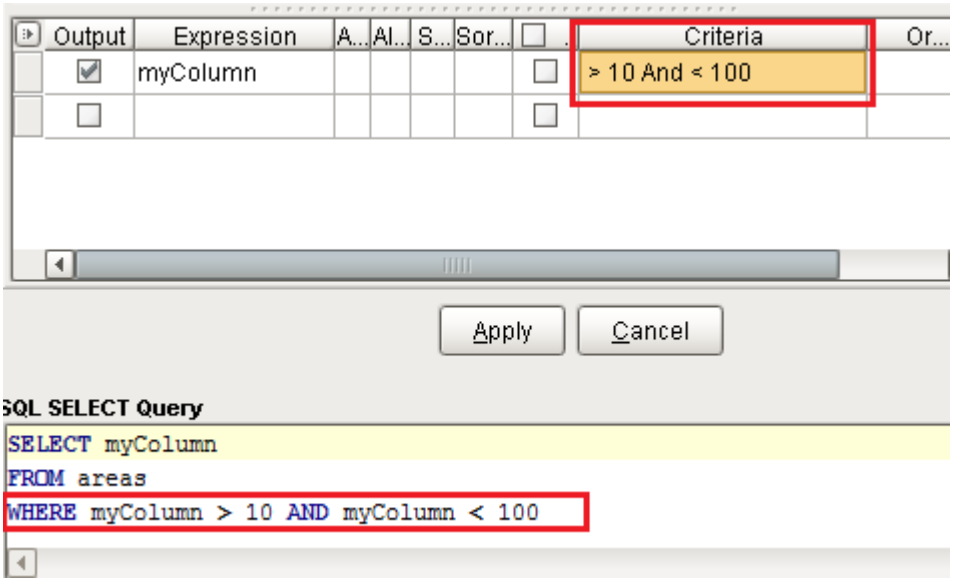
Right-clicking on a row of the Columns Table brings up a menu with the following options:

Option	Description
Move up	Moves the selected row up in the table. This means the column will appear in the query results prior to the other listed columns.
Move down	Similar to move up, but moves the row down, or towards the end of the listed columns.
Delete item	Removes the row from the Columns Table, effectively removing it from the resulting query.
Insert empty item	Adds a new row to the Columns Table with blank values.

Field Description

Each column in the Columns Table allows you to modify the resulting query in some way. The following is a description of each column:

Option	Description
Output	Specifies if the row should appear in the resulting query. Disabling a row on the table removes the column from the results.
Expression	The column from one of the tables in the Building Area that this row represents. Left-clicking on this field will create a dropdown list of possible rows.
Aggregate	Allows you to aggregate the expression. Blank by default (no aggregate), configuring this column allows you to do things like sum column, or return the average. Possible values are: • Blank (no aggregation will be performed) • Avg • Count • Max • Min • Sum You can also add the Distinct keyword to any non-blank option. For example, you could enter "Sum Distinct" or "Count Distinct".
Alias	Creates an Alias, or alternative name for the column. Effectively adds the SQL "AS" keyword to the column, which allows you to rename the resulting column.
Sort Type	Sorts the results based on this column. Enabling a Sort Type is similar to added an ORDER BY to your query. Possible values are: • Blank (no sorting on this column will be performed) • Ascending • Descending
Sort Order	When a **Sort Type** has been specified on multiple rows, this determines which row the query will sort on first.
Grouping	Allows you to group the results. Adds a GROUP BY statement to the resulting query. Enabling grouping on one of the columns makes the **Criteria for** column appear.

Option	Description
Criteria for	A dropdown list that sets how you want the Criteria column to apply to the grouping. Does not apply if both Grouping and the WHERE clause Criteria are not being used. • For Values will place the WHERE clause specified in the Criteria column first, filtering out rows that don't apply, and then applying the grouping. • For Groups will instead change the WHERE clause into a HAVING clause that is used to filter after the grouping has taken place.
Criteria	Allows you to add a WHERE clause. Supports the use of both the OR and AND keywords for multiple conditions. Example: assuming a column named "myColumn", we could limit the results of our query to rows where myColumn has a value greater than 10 AND less than 100: 
Or...	Allows you to add an additional condition to a WHERE clause. Will separate each grouped condition with "()" characters to maintain logic.

NOTE

The Query Builder is a third party tool that we brought into Ignition and while we go over how to use it here, you can also check out the **Active Query Builder's documentation** for more information on how this feature works.

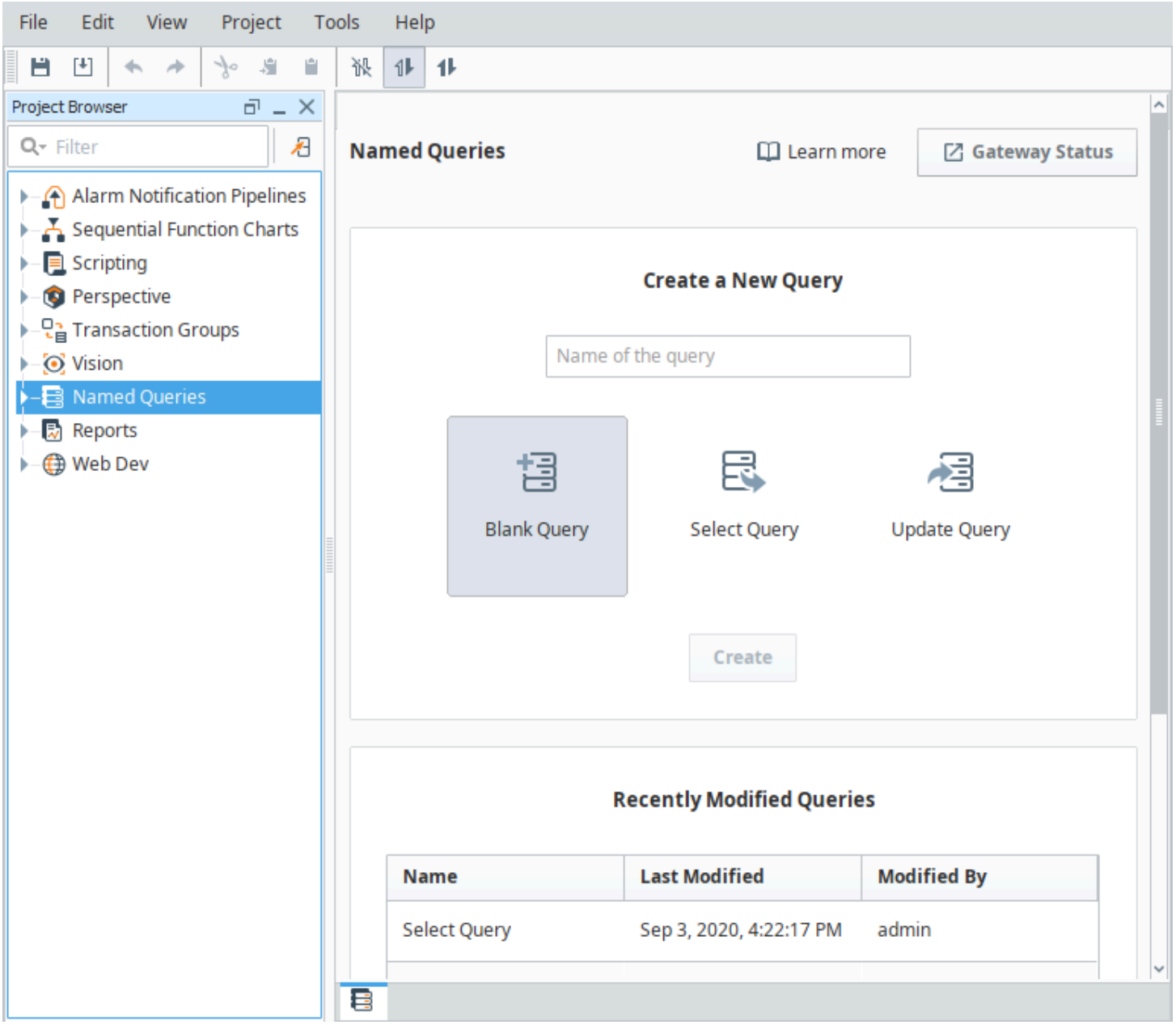
Named Queries

What are Named Queries?

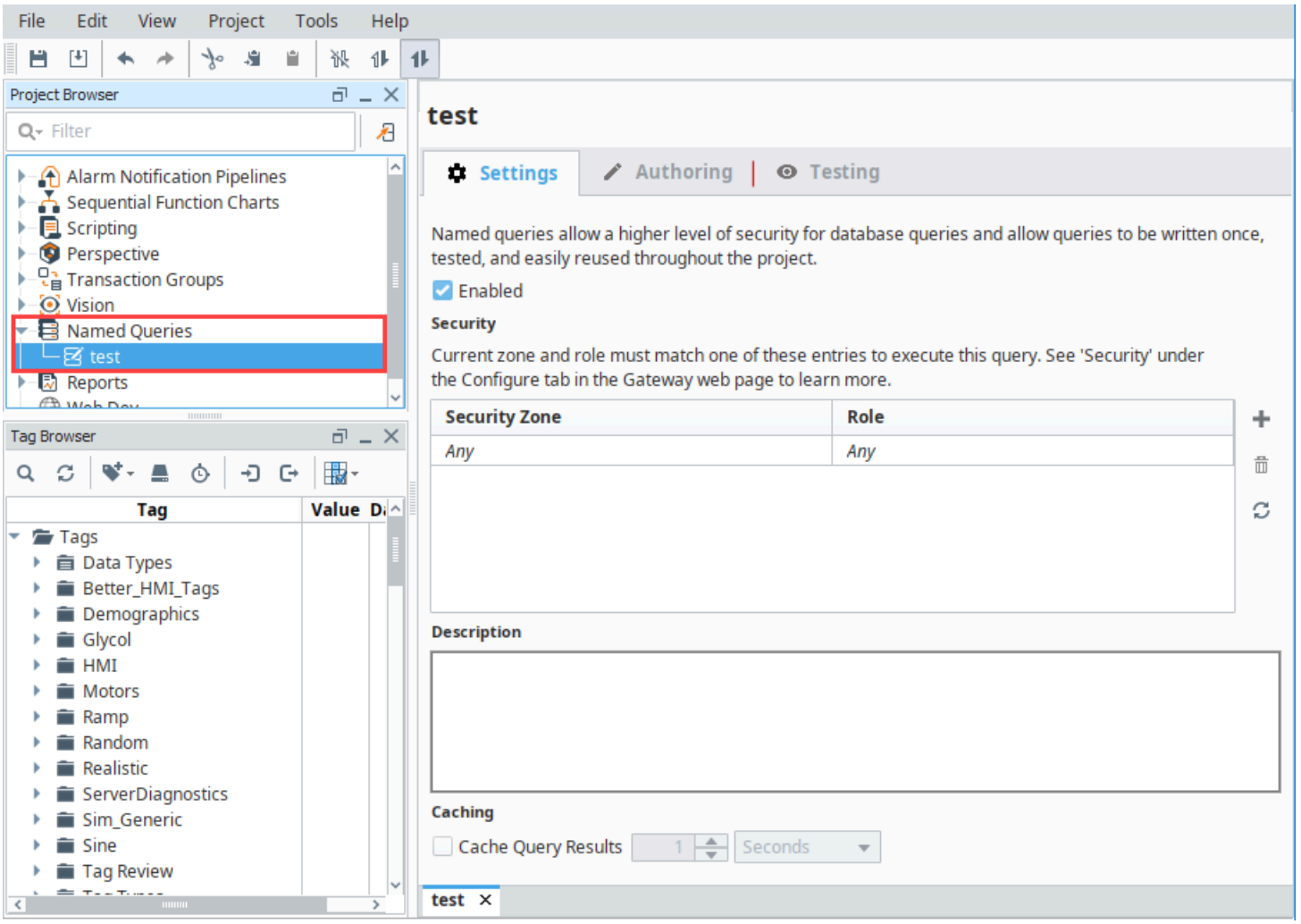
Named Queries are preconfigured queries that may be invoked elsewhere in a project. Named Queries are conceptually very similar to Project Scripts; defined in a single location, and then referenced in multiple places throughout the project. When executing a Named Query, parameters may be passed to return a dynamic result set. This way, a query may be written once, and then called from multiple locations throughout the project.

The Named Queries Welcome tab allows you to create three types of named queries. Each one of the named query types is basically a template to help you get started creating your own query. Once you select a Named Query type, enter a name, and press 'create', and the specific named query template will open. The Select Query and the Update Query will have some sample parameters and queries to help you get started. The Named Queries Welcome tab will show you any recently modified named queries along with the date it was modified and who modified it. You can even double click on a recently modified query and open it.

The Named Queries Welcome tab provides a quick way to create a new query and update existing ones.



Named Queries have their own **workspace** in the Project Browser section of the Designer.



Secure Query Execution

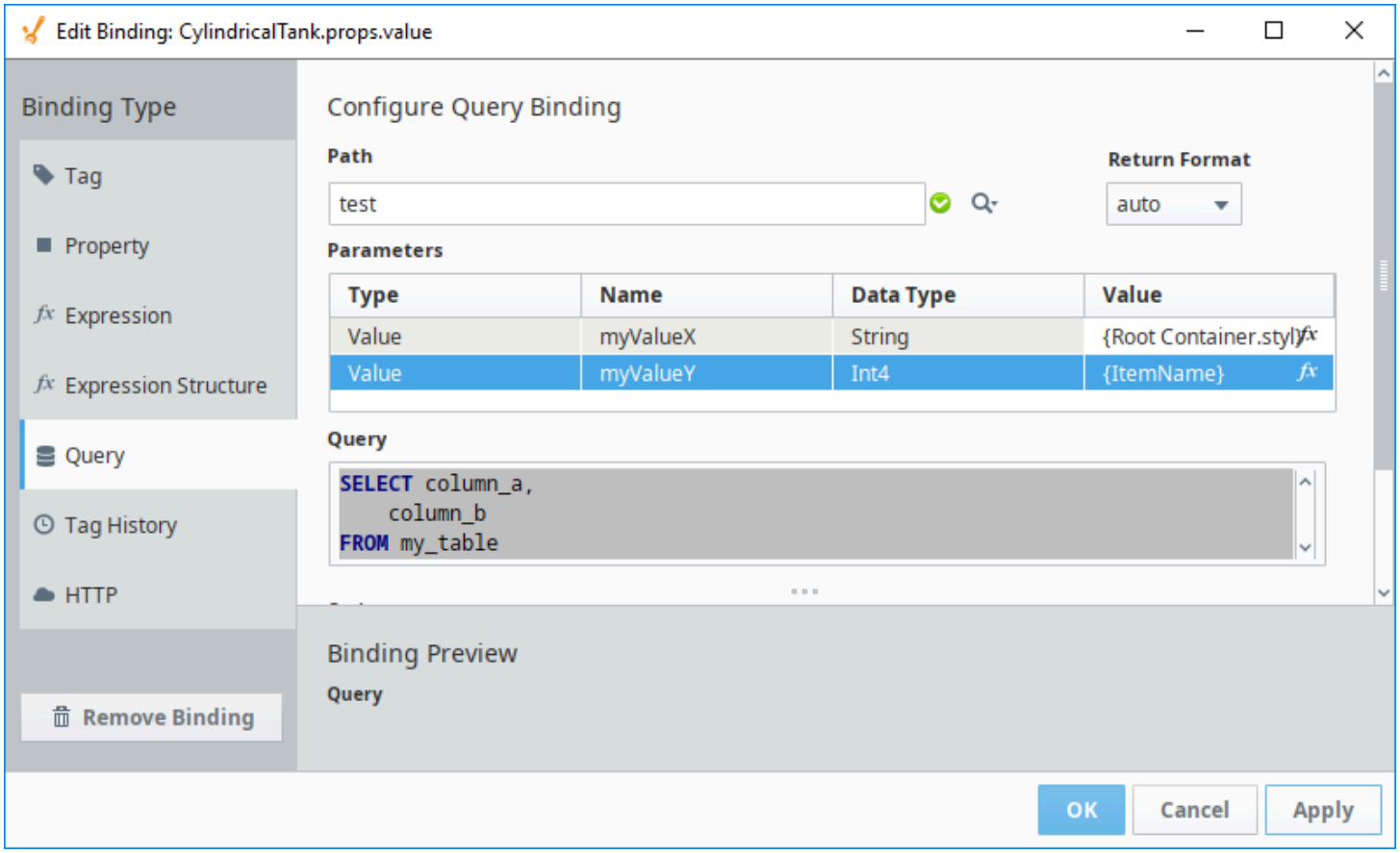
While clients may request data from a Named Query, the actual execution of the query always takes place on the Gateway. Clients simply specify which query should run, and pass parameters that the Gateway will use. Additionally, the Gateway has an opportunity to **restrict access** to the query based on Security Zone and/or User Role. This provides a single interface to restrict access to the queries, and better protect your data. Additionally, queries cannot be modified by a Client other than by passing variables into it. This creates a very secure method to control what queries are being run against your database.

Where are Named Queries Used?

Named Queries may be used in multiple locations in Ignition. They are used in all the same places that a normal SQL query would be used. Here are a few examples of locations in which named queries can be used. A more detailed example of a named query is provided **here**.

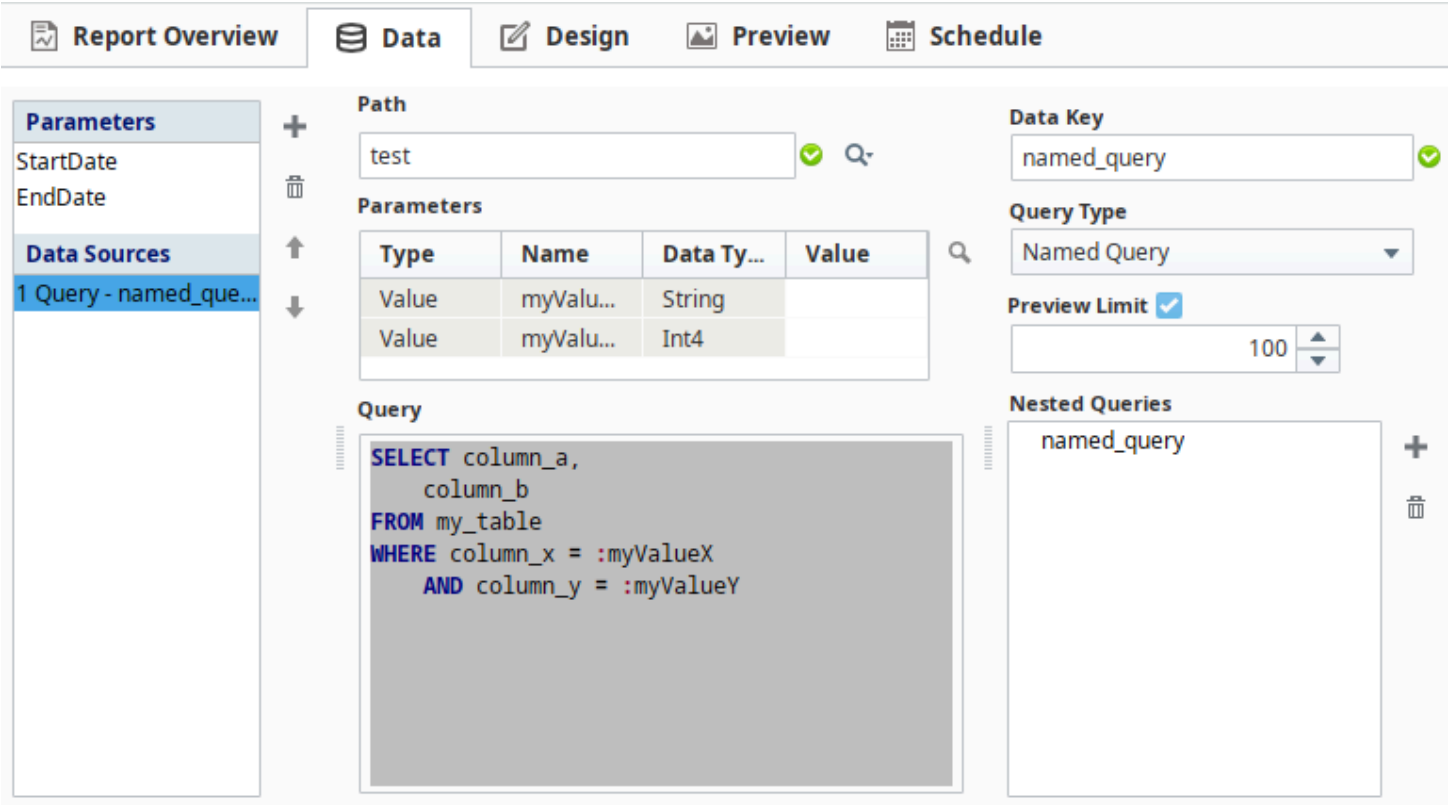
Binding

A **Query Binding** type has been added to leverage named queries for any component property bindings. **Named Query Parameters** may be bound to Tags or other properties on the same window, allowing your users to modify the resulting dataset by manipulating other components similar to the original SQL Query binding type. You can also select and update a query to simulate a bi-directional binding to the database.



Reporting

A **Named Query Data Source** has been added to **Reports**. Report Parameters may be used by the Named Query to generate dynamic reports.



Scripting

The **system.db.execUpdate** and **system.db.execQuery** functions provide secure methods to execute Update and Select Queries from a Named Query resource in any location in Ignition.

Named Query File Location

The queries and metadata used by named queries are stored in SQL and JSON files respectively on the Gateway's file system.

```
%installationDirectory%\data\projects\PROJECTNAME\ignition\named-query\QUERYNAME
```

Where PROJECTNAME is the name of the project the Named Query resides in, and QUERYNAME is the name of the Named Query.

Named Query Workspace

Named Query Workspace Overview

Named Queries have a dedicated workspace inside of the Project Section of the Designer. This workspace allows for the creation and testing of Named Queries. Once created, the Named Query may be called from another resource, such as a **data source in a report** or a component using a **Named Query binding**.

Named Queries are created by **right-clicking** the Named Query item in the Project Browser. Like other resources in the Project Browser, Named Queries can be organized in folders, which creates a unique path to the query and helps keep your queries organized.

> **NOTE**
> Named Queries are referenced by path, so renaming the Named Query or any parent folders will require you to update the path on any other resources that are using it.

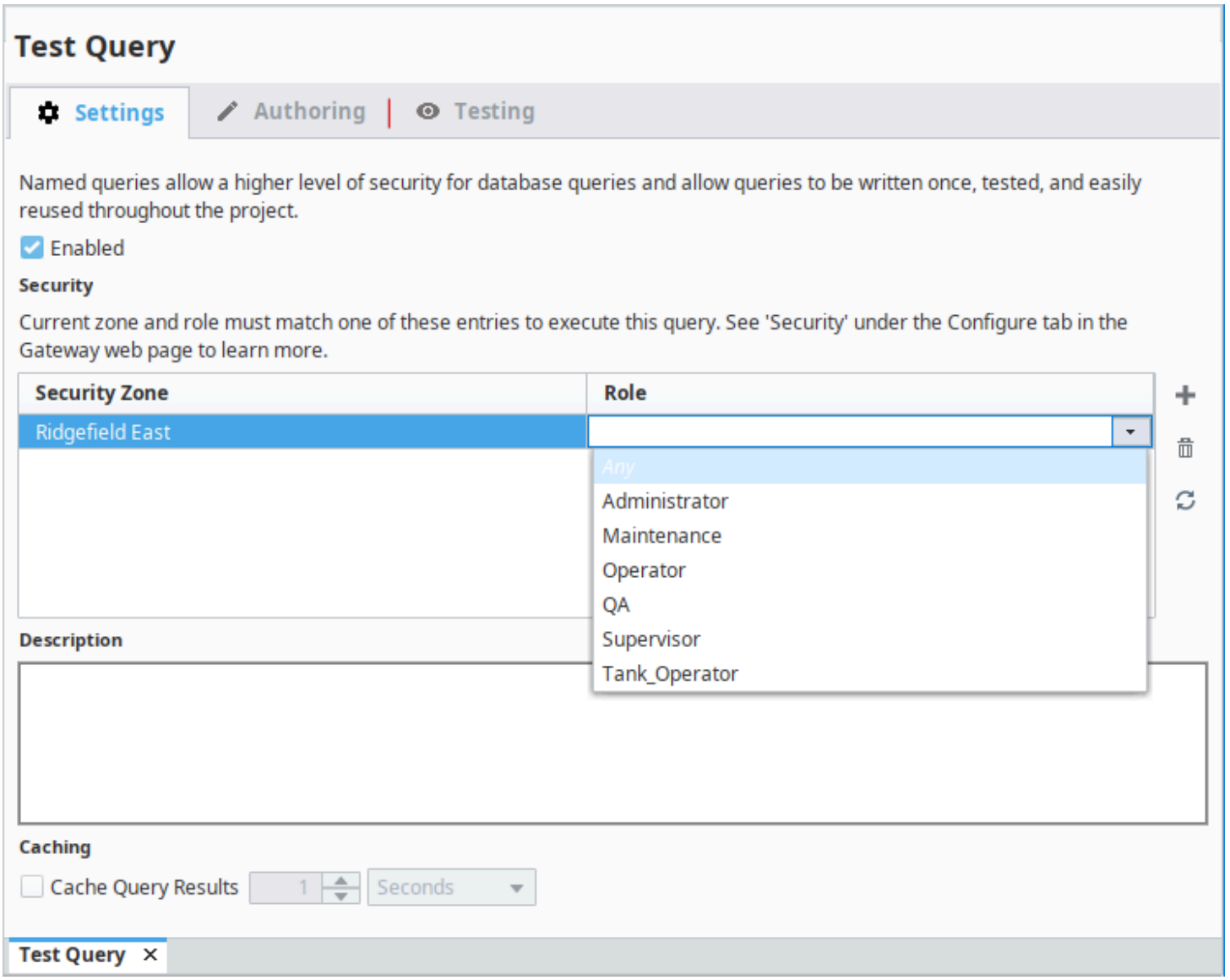
Also like other resources, multiple Named Queries may be opened in the same Designer session. Tabs at the bottom of the Designer allow for easy swapping between Named Queries.

Workspace Sections

The Named Query workspace contains three tabs: Settings, Authoring, and Testing. A description of each section follows.

Settings

This Settings tab contains configuration properties and security for the selected Named Query.

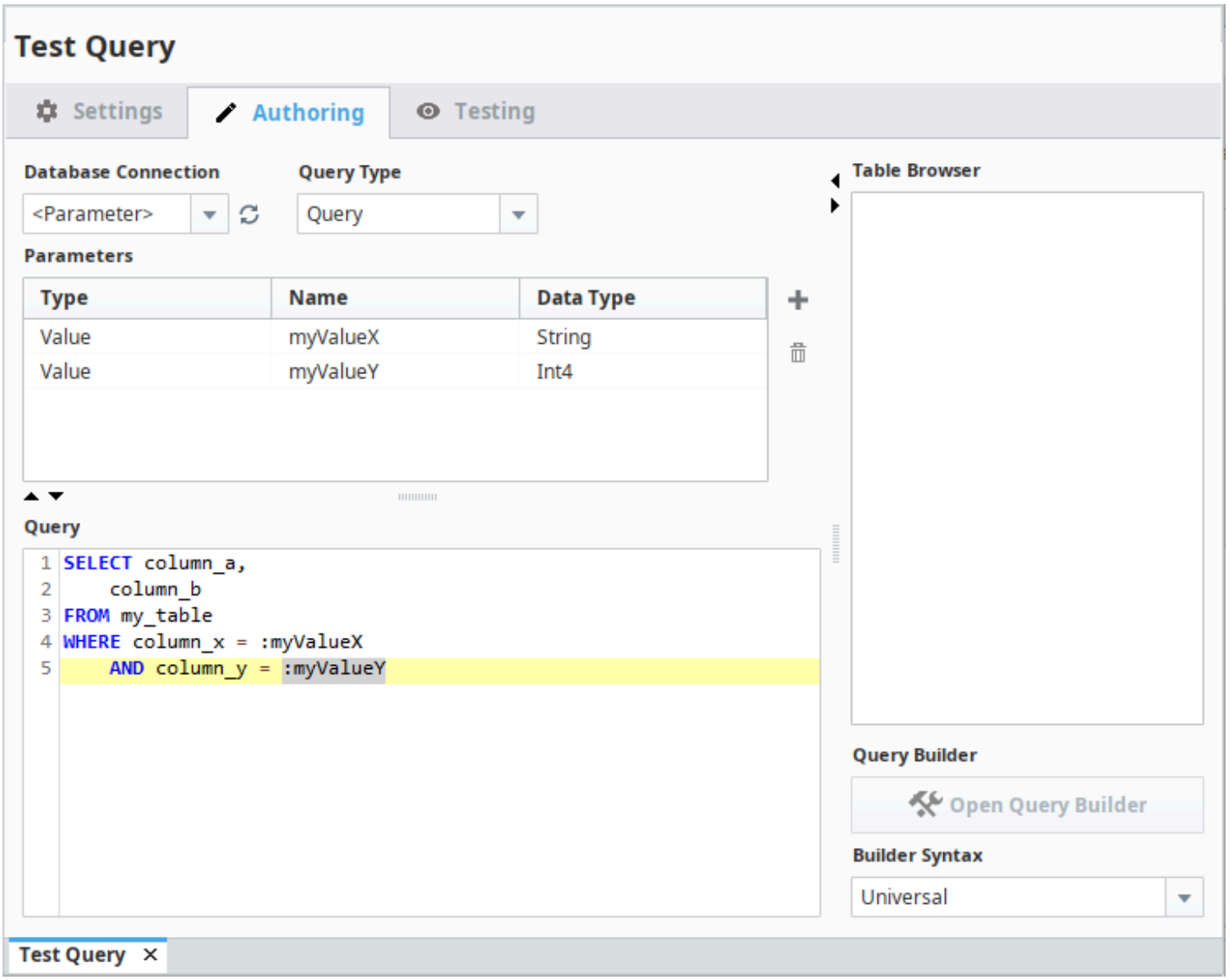


The following properties are available:

Item Name	Description
Enabled	Determines if the Named Query is enabled or disabled. A disabled Named Query may not be executed.
Security	Specifies a combination of Security Zones and Roles that may call the Named Query. Only roles in the projects user source will be available in the Role dropdown. Multiple rows may be configured to account for granular access restrictions (i.e., requests from **Administrator** roles originating from the **Office** security zone area could be allowed, while requests from same users in the **plant floor** zone could be denied). If the request does not match any of the specified zone and role combinations, then the query will not run. Additionally, if a **ScalarQuery** type has a Fallback value configured, that Fallback value will not be returned either: the query will not execute due to security settings, so there is never a chance for other errors to occur. Either the Security Zone or the Role (but not both) may be left blank. This means it is available to all objects of that type (i.e., with a blank Role and the "office" Security Zone, all roles in the "office" zone are valid).
Description	Allows you to give the Named Query a description.
Caching	Allows the Gateway to cache the results of the query. See the **Named Query Caching** page for more details.

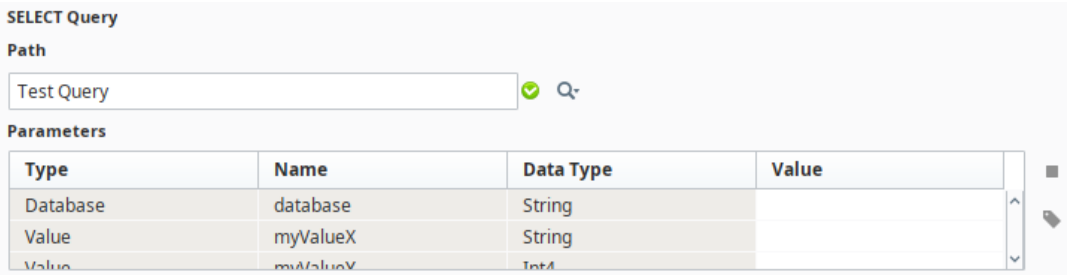
Authoring

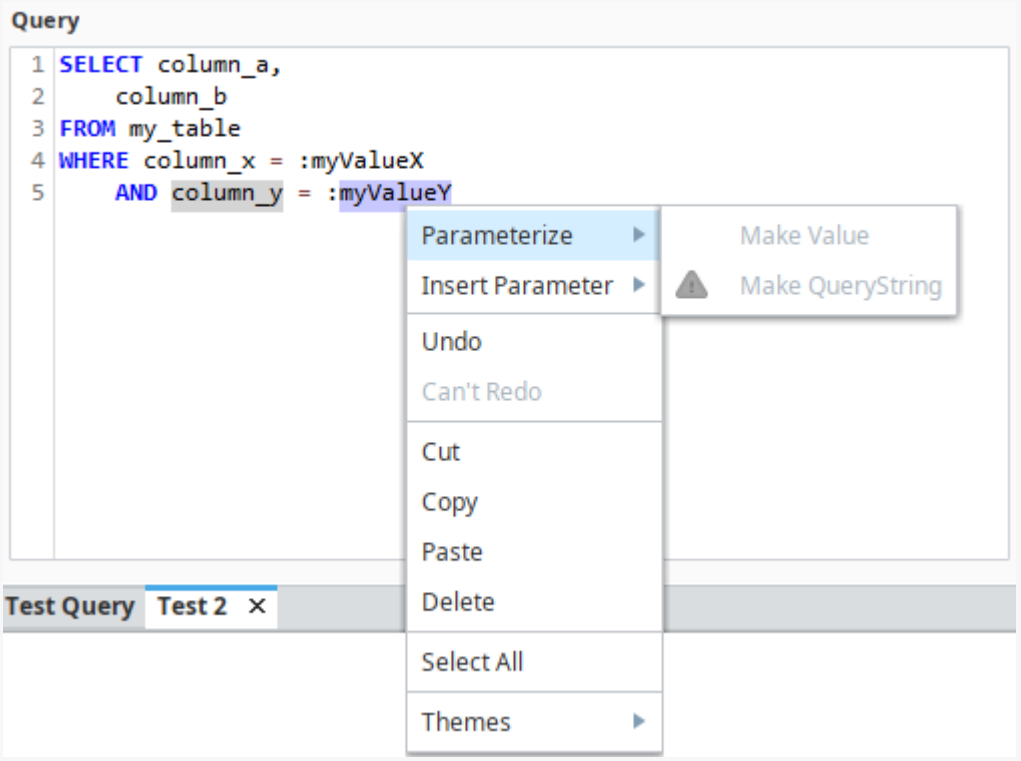
The Authoring tab is where the query and parameters are created. There is also a Table Browser and **Query Builder** that can be used to help you to create your query.

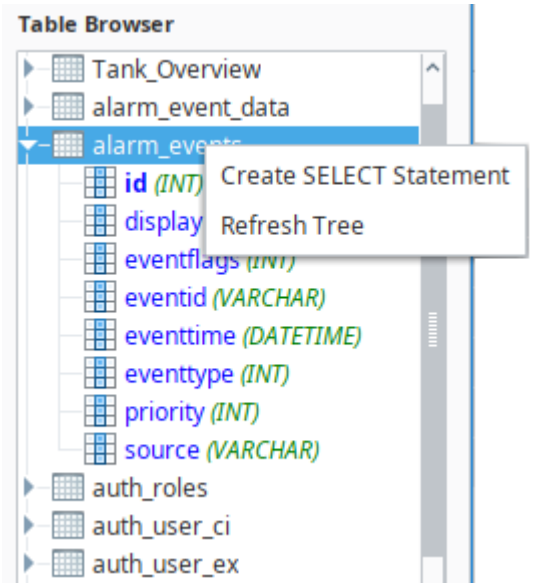


NOTE

Switching to another tab from the Authoring tab will automatically commit any changes to the Named Query's parameters.

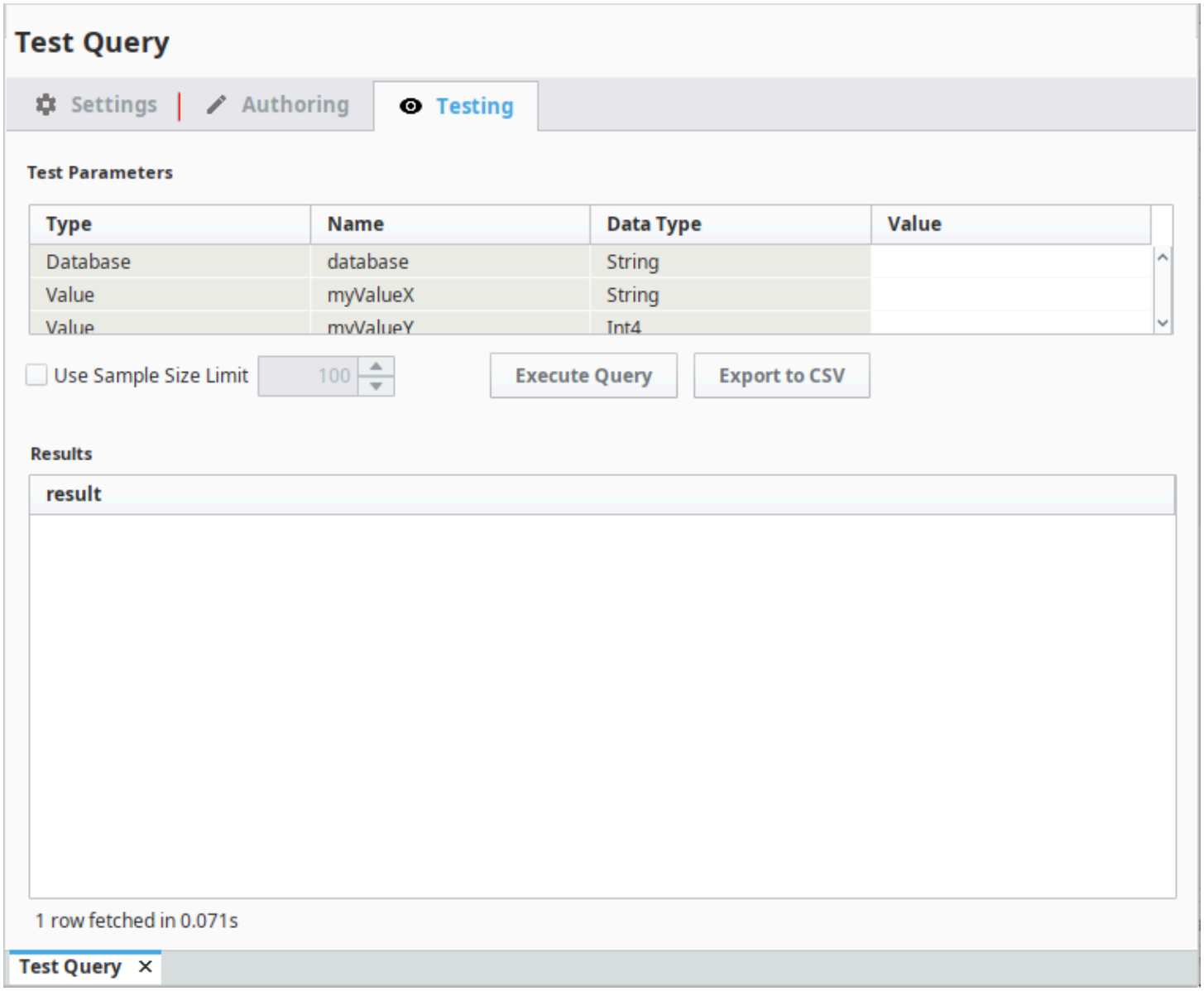
Item Name	Description
Database Connection	The database connection the Named Query should run against. In addition to a list of the **database connections** configured in the Gateway, this dropdown contains two unique values: `<Default>` and `<Parameter>`. • `<Default>`: The query will execute against the project's default database connection. • `<Parameter>`: The query expects the database connection name to be passed in as a parameter when called. This allows you to use a dynamic database connection. **Note**: A parameter for the database connection does not need to be manually created in the Parameters section. Instead, the Named Query will have a special "Database" type parameter available when called. Below is an image of a Named Query binding that is utilizing the `<Parameter>` connection type. 
Query Type	The type of query to execute. The following options are available: • **Query**: Allows SELECT Queries and returns a full dataset. This type should be selected when running a SELECT statement that returns multiple rows or columns. • **ScalarQuery**: Allows SELECT queries and returns a single value. This type should be used when running SELECT statements that only return a single value. The very first cell returned will be the only output. This type is special in that a **Fallback** value may be defined. The Fallback value will be returned if the Named Query would return an error. Note, the **Fallback** parameter will not be returned if the request does not meet the security requirements. • **UpdateQuery**: Allows all **UPDATE** types of queries (querys that mutate or otherwise modify rows on a table) and returns the number of rows affected by the query. This type should be selected when modifying the database in some way, such as when running an INSERT, UPDATE, or DELETE query.

Item Name	Description
Parameters	A table of the parameter names and types that will be used in the query. These parameters have three types, Value, QueryString, and Database. Most commonly, the **Value** type is used and can be accessed by using the **:paramName** notation. More details on this field may be found on the **Named Query Parameters** page.
Query	The query that will execute when the Named Query is called. You can type directly into this field or use the Table Browser on the right to get started. **Right-Clicking** inside this field will cause a popup menu to appear:  Most of the items on this menu are self-explanatory, but a few require special mention: **Parameterize**: Contains two sub-menu items, which are detailed below. Note, that the sub-items will be **disabled** unless you right-click on some text that does not reference a parameter, table name, or column name in the query. • **Make Value**: Turns the selected text into a value-type parameter. The new value-type parameter will appear in the **Parameter** table above the query. • **Make QueryString**: Turns the selected text into a QueryString. The new QueryString will appear in the **Parameter** table above the query. Note that QueryStrings are susceptible to SQL injection attacks. Because of this, the **Make Value** option is recommend over this option.

Item Name	Description
	Insert Parameter: Quickly creates a reference to the selected parameter. This menu is an alternative to dragging-and-dropping from the Parameters table or typing the name of the parameter.
Table Browser	Provides a list of the tables in the selected **Database Connection**. Tables may be dragged into the Query field to quickly insert the name of the table. Additionally, **right-clicking** on a table in the list will cause a popup menu to appear:  • **Create SELECT Statement** will populate the Query field with a SELECT statement targeting the selected table or selected row of the table. • **Refresh Tree** will refresh the Table Browser.
Query Builder	Opens the **Query Builder**, which provides an easy way to create SQL queries using a drag-and-drop interface. This button will be disabled if the **Database Connection** property is set to `<Parameter>`. This is the same Query Builder used in other places like the reporting data page.
Builder Syntax	Specifies the syntax the **Query Builder** should use. Contains syntax for many popular databases, and has a Universal selection that should work in most scenarios.

Testing

In the Testing tab, you can test your query without leaving the workspace. Fill in your values and click the **Execute Query** button to see your results.



> **NOTE**
>
> If your Named Query is using a DateTime parameter, you can expand the DateTime picker to easily set a DateTime value.

Item Name	Description
Test Parameters	Allows you to manually supply test values to the parameters to the queries. The table is populated by the **Parameters** field on the Authoring section.
Use Sample Size Limit	When checked, allows you to set the maximum number of rows the query will return while testing the Named Query. In addition this setting also limits the results of that named query called elsewhere in the Designer. This property is only enabled when the **Query Type** on the Authoring section is set to **Query**.
Execute Query	Runs the Named Query using the parameter values listed above. The **Results** area will display any results returned by the query.
Export to CSV	Exports the results of the query to a CSV file. The button becomes available after results are returned. Note that this will only return the values shown, check the Sample Size Limit when using this button.
Results	The results returned by the Named Query when testing. Populated by the **Execute Query** button.

Named Query Parameters

Parameters allow you to make Named Queries dynamic. They act as placeholders you can pass values into when requesting the query to execute. Other resources in Ignition can then pass arguments into the parameters. The exact implementation depends on what resource is requesting the Named Query, such as a **Named Query binding** or a **Named Query Data Source**. Check out the **Named Query Example** page for a complete example on passing parameters into a named query.

Types of Named Query Parameters

There are three types of Parameters in Named queries. Each varies in usage.

Value

The Value type should be used whenever a Named Query needs a dynamic WHERE clause. These act like values passed to a prepared statement, meaning they can never be used to parameterize column or table names. However, they are resilient to SQL injection attacks.

SQL Query - Using a Parameter

```
SELECT * FROM mytable
WHERE name = :myParam
```

QueryString

QueryStrings are more flexible than the Value type in that they can be used to parameterize column and table names. However, their values are never sanitized, which causes them to be more susceptible to SQL injection attacks. When using QueryStrings, it is best to avoid situations where the user can manually type in the value that will be passed to the Name Query. Additionally, if you are using a QueryString for a string in the where clause, you would need to provide quotation marks.

SQL Query - A Using QueryString

```
SELECT {myColumnName} FROM mytable
WHERE name = '{myName}'
```

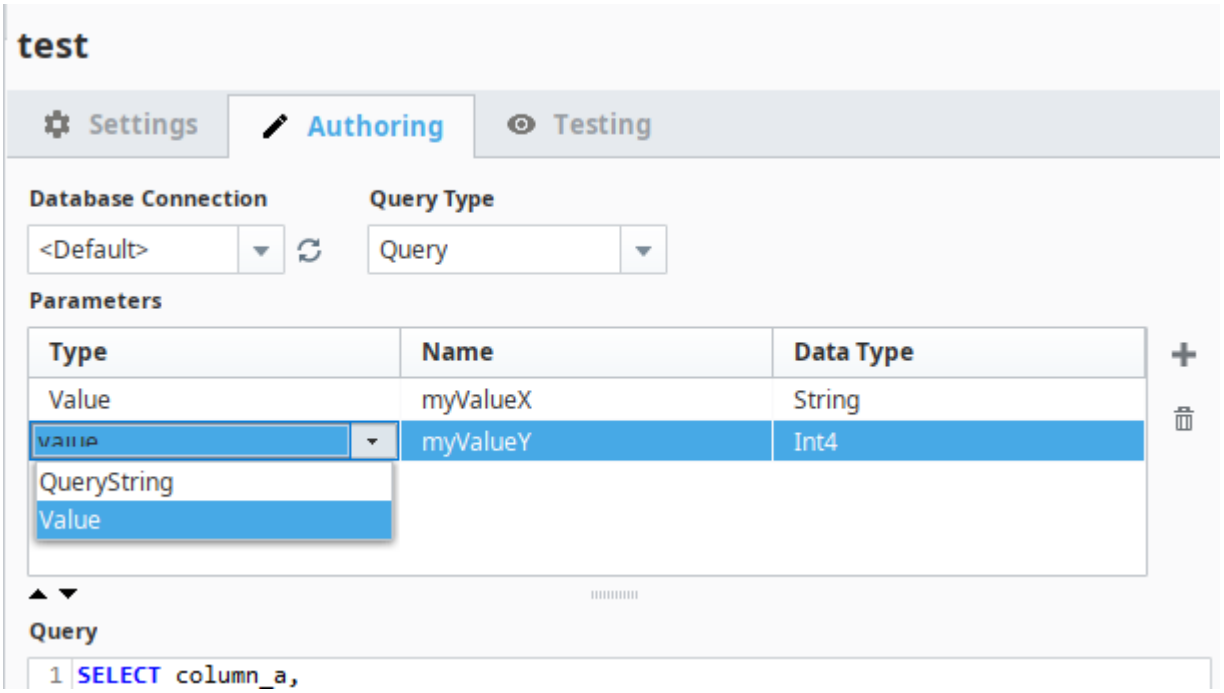
Database

Database type parameters cannot be created manually. Instead, it is automatically created when the **Database Connection** dropdown on the **Authoring** section is set to **<Parameter>**. Additionally, this parameter is not used in the body of the query. This type allows you to parameterize the database connection when the Named Query is called. This way the Named Query can run against multiple database connections specified by the resource that made the request.

Parameters while Authoring a Named Query

Creating Parameters

New parameters can be created in the Authoring section of a Named Query by clicking the Add (✚) icon next to the Parameters table.



> **NOTE**
>
> In Named Queries, Parameters are referenced by their name, so renaming the Parameter will require you to update it on any other resources that are using it.

Each parameter has three properties represented by different columns in the table. These may be edited by double-clicking on the cell you wish to modify:

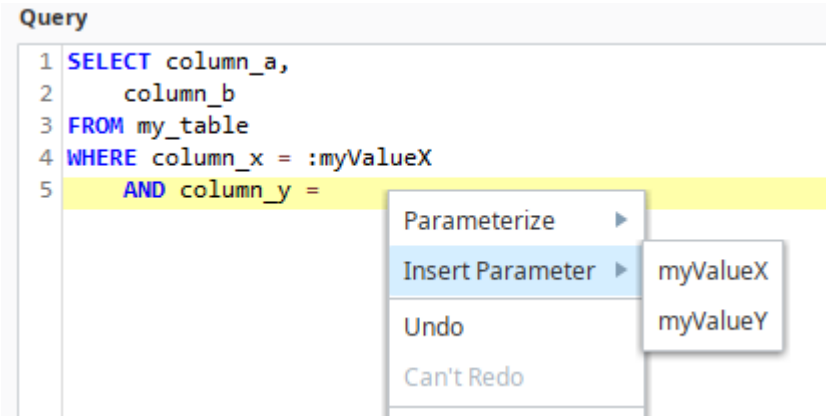
- **Type**: Changes the type between Value and QueryString.
- **Name**: Determines the name of the parameter, and how it will appear in the query. Names are **not** case-sensitive and must be unique. Additionally, they may only use letters, numbers, dashes

and underscores.

- **Data Type**: Specifies the datatype of the parameter. The Type of the parameter determines which data types are available. Note, that QueryStrings may only be configured as strings, where as Value-type parameters have more types available.

Using Parameters in the Query

Once created, parameters can be inserted into the Query field by **drag-and-drop** from the Parameter table onto the Query field, or by using the **right-click menu** in the Query field.



Additionally, the parameters may be typed in manually, but the correct syntax must be used.

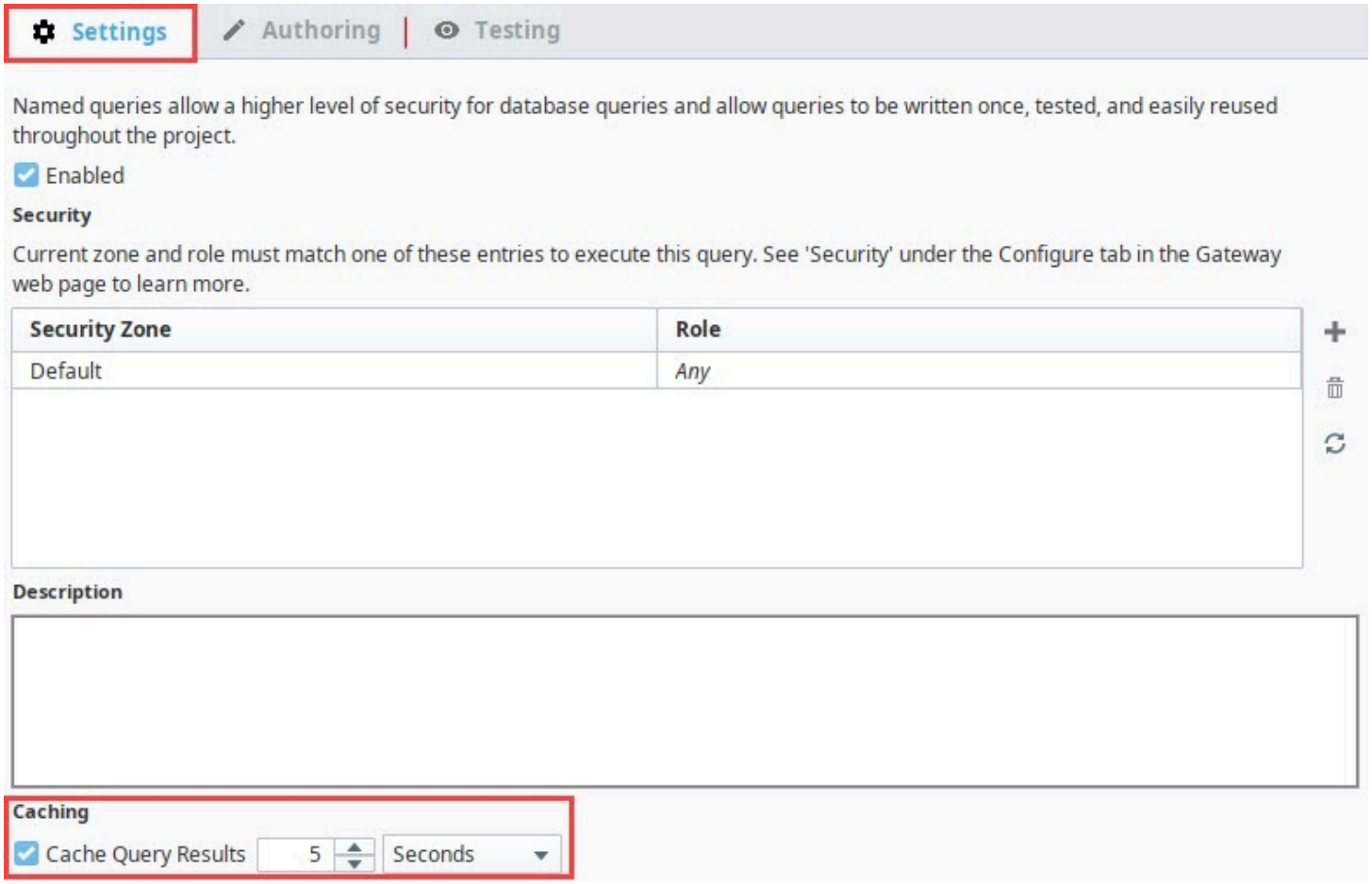
Named Query Caching

Overview

Named queries can opt-in to caching the results on the Gateway. This means if another request to the same Named Query comes in, the Gateway can return the cached result instead of having the database executing the query again. This will use more memory on the Gateway (to maintain the results), but could result in less queries running against the database.

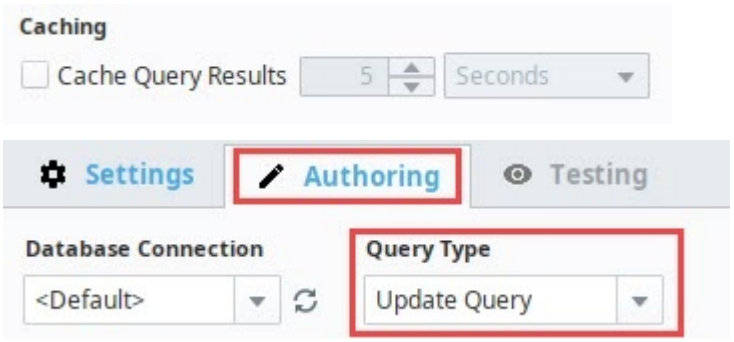
Named Query caching is disabled by default, but can be enabled on the **Settings** section of each Named Query. If caching is enabled, the spinner and dropdown fields set the **lifespan** of the cache. Once the lifespan expires, the cache is invalidated. After the cache is invalidated, the next request for the Named Query will force the Gateway to re-execute the query and build a new cache.

Caching is especially useful for tables that are not updated often like recipe or inventory tables. Tables that update often like historical storage tables are bad candidates.



When Caching is Disabled

Update queries are not allowed to cache their results. With UPDATE or DELETE statements, the work will already have been performed after the first execution, and INSERT statements typically utilize different parameters each execution. Thus if the Caching setting is disabled, then this typically means that the **Query Type** in the **Authoring** tab is set to an **Update Query**.



Scenario

Consider the following:

- A Named Query is created. This runs a query to fetch data from an arbitrary table.
- A component in a project is configured to call the Named Query on a window.
- **Client_A** navigates to the window, which triggers a request for the Named Query be executed.
- Several seconds later, **Client_B** opens the same window, and needs the same results.

In this scenario, if caching was enabled on the Named Query, then **Client_B** would not cause another query execution (assuming both **A** and **B** passed the same values to the parameters). This would result in less network traffic between the Gateway and the Database, and less work for the Database. An example with two clients isn't exciting, but the same scenario with ~50 clients would mean a huge potential performance boost, especially if those requests were polling at a 5 second rate, and the cache period was configured to 5 seconds.

Considerations

The following are considerations that should determine whether or not caching the results of a Named Query is helpful.

Gateway Memory

Each time the Named Query is called, if the arguments passed differ from those used to create the current cache, a new cache is created. This means Named Queries that are frequently called with

varying parameter values will create multiple caches. If the results are large datasets, this can result in a large amount of the Gateway Memory being tied up maintaining these caches. In this scenario, you will want to monitor the memory usage of the Gateway.

This is especially important to consider when dealing with queries that accept a timestamp parameter that uses an expression like now() that will return time to the current second. If 5 clients are opened with each one 1 second apart, they will all have different timestamps and create separate cache entries.

How Often the Database Values are Updated

Once a cache is created, the Named Query will not look for any changes made to the database table until the query next executes. This means changes, such as adding a new row to the table, will not appear in the Named Query's results until the cache is invalidated. Data that changes often may not be a good candidate for caching. In this case, the lifespan of the cache should be set to a short amount of time, depending on how often the data may get updated.

How Often the Named Query is Executed

If a Named Query is called frequently and there are not many updates to the table data, then there can be a huge performance benefit to caching the results. Fewer calls to the database result in less network traffic and better overall performance.

Cached Query Updates and Designer Values

Cached query results have some notable interactions in the Designer. Specifically:

- **Named Queries results executed in the Designer never cache**: This is because the Designer uses the live version of the query in the Designer instead of the saved Gateway version. This also prevents bad results being stored in the cache, and then appearing in the Clients.
- **Making changes to a Named Query and saving will invalidate all caches for that Named Query**: When a change is made to a Named Query and saved, the new query is pushed to the Gateway, which means all current caches are immediately outdated.

Scripting Functions to Clear Cache from a Named Query

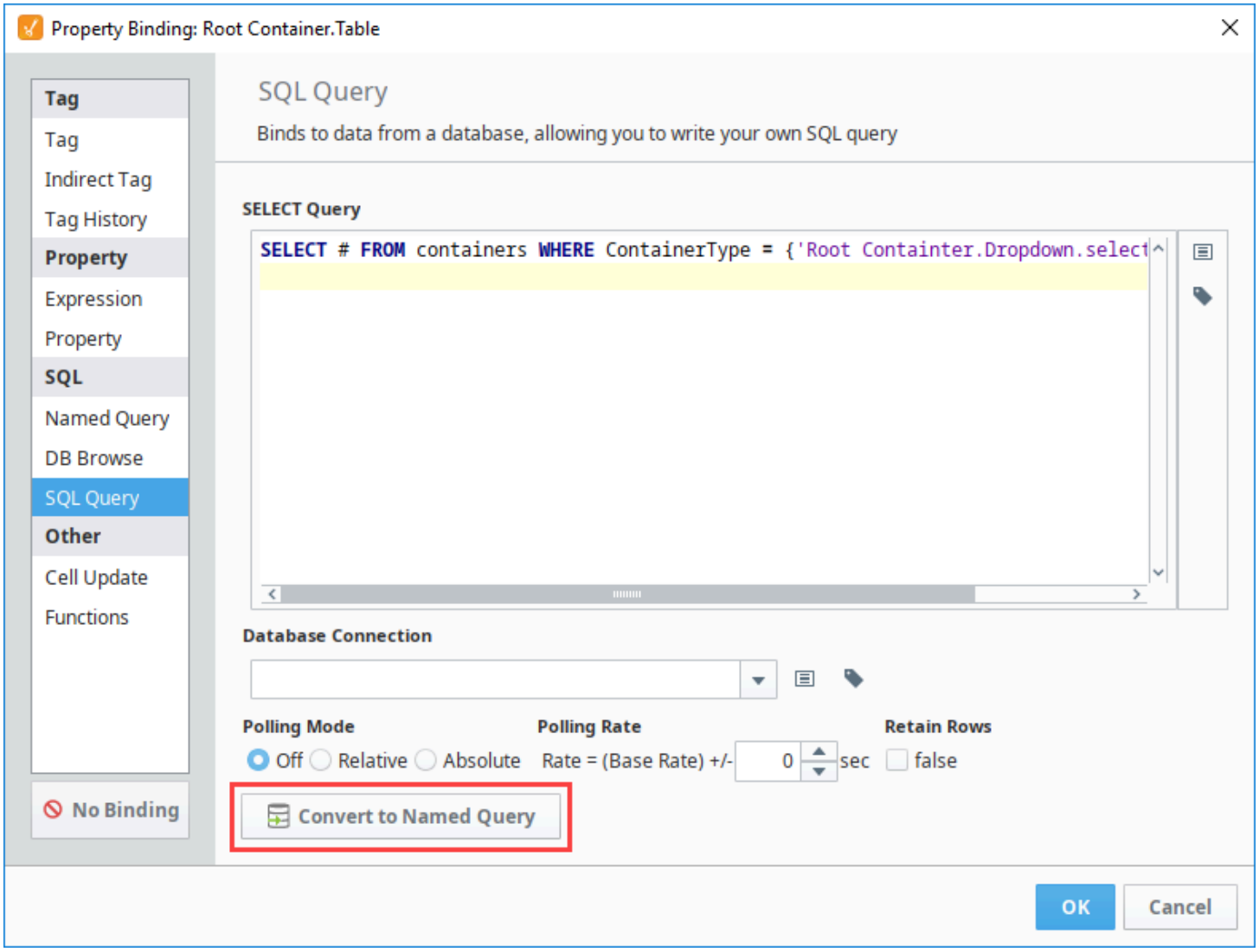
You can also clear the Named Query cache using the following scripting functions.

- **system.db.clearQueryCache** - Clears the cache of one or all Named Queries.

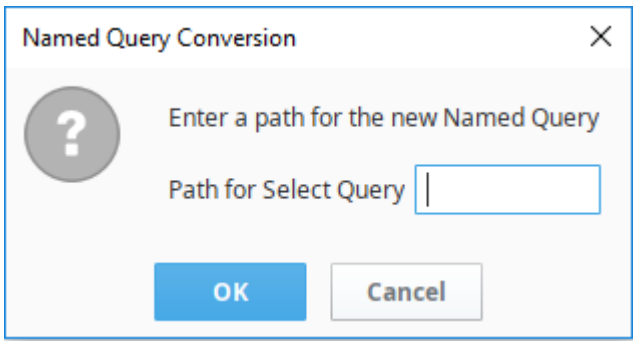
Named Query Conversions

Converting SQL Queries to Named Queries

If you have a SQL Query binding on a component, it is easy to convert it to a Named Query. Simply open up the binding and click the **Convert to Named Query** button.



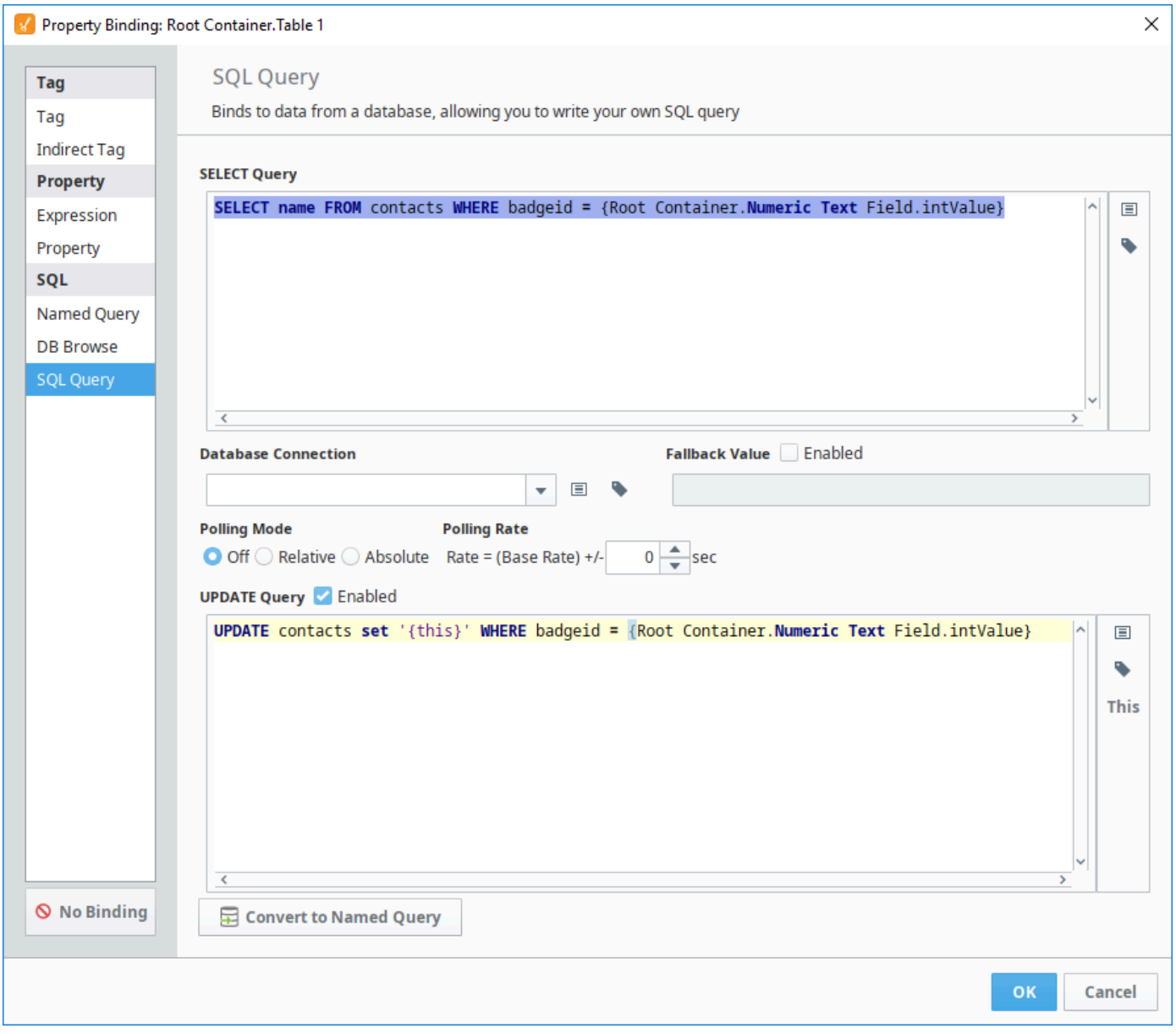
A popup appears where you can enter a path to the new Named Query.



Converting with an Update Query

If your SQL Query binding is on an component property whose value can be updated and written back to the database (for example, input components and button components), you have the option to create two Named Queries: one for the Select Query in the binding and a second for the Update Query.

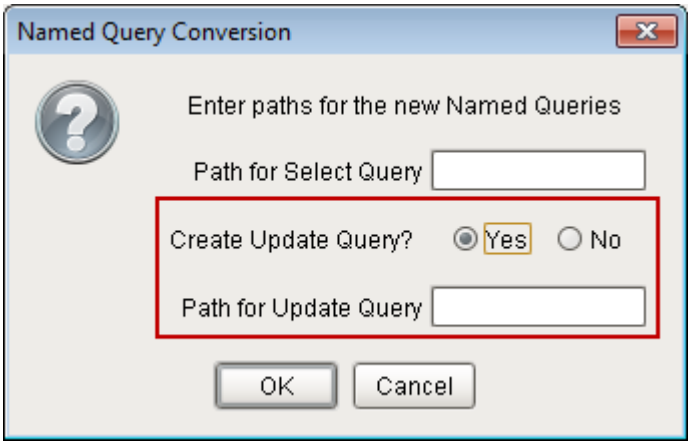
1. Enter the Select Query.
2. Click the Enabled checkbox for the Update Query.
3. Enter the Update Query.
4. Click OK.



5. In the popup, enter the path for the Select Query.
6. Click the Yes button next to Create Update Query,

7. Enter the path for the Update Query,

8. Click Save.

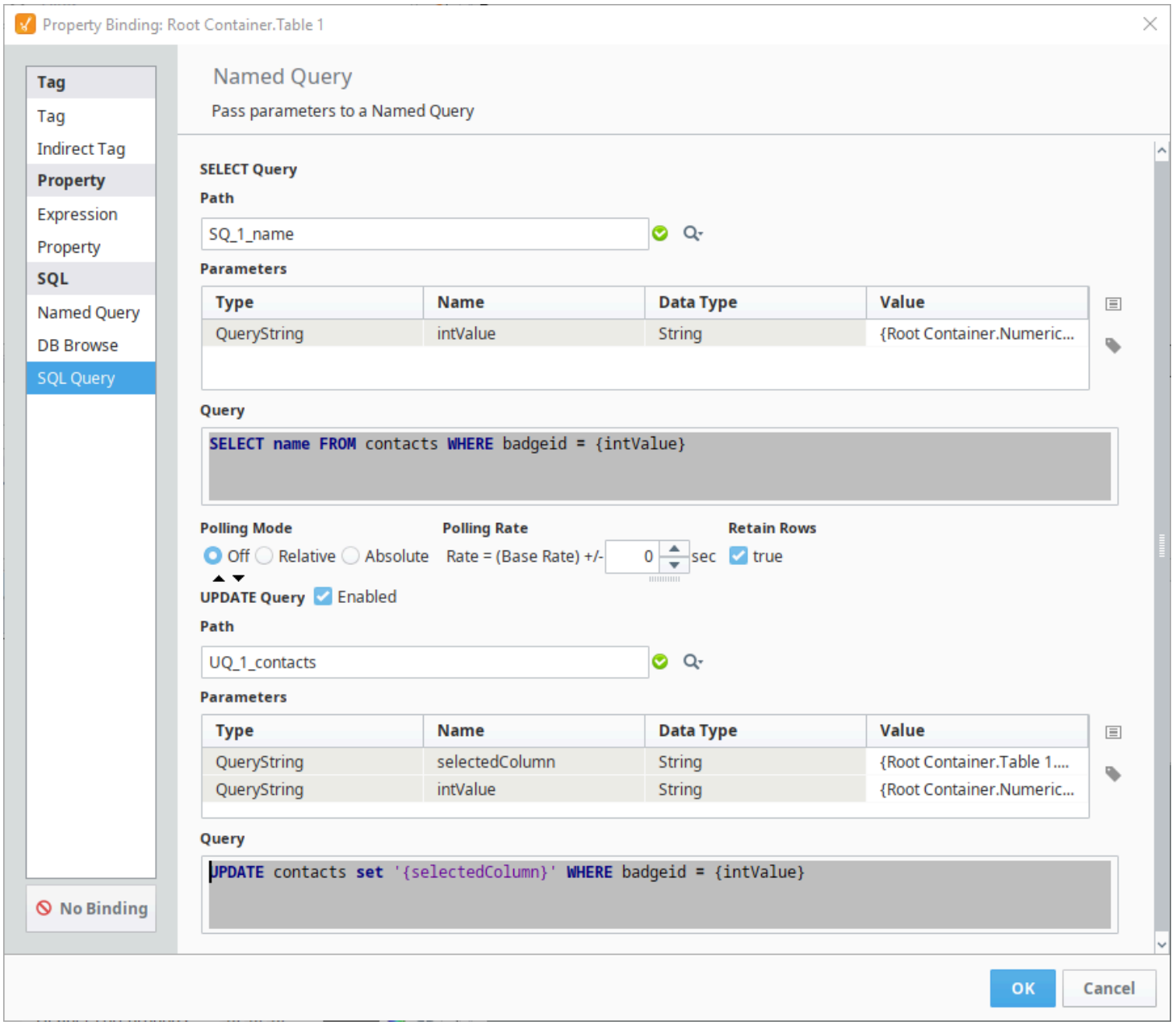


WARNING FOR 7.9.5 AND PRIOR

When converting a SQL Query Binding that contains an UPDATE Query in 7.9.5 and prior, an Update query will not be generated. Thus, it was possible to lose the Update Query upon conversion. If using these versions of Ignition, it is highly recommended to manually create the Update Named Query before pressing the Conversion button.

As of 7.9.6, this is no longer an issue, and the **Convert to Named Query** button may freely be used.

After confirming the conversion, the Window will show the query/queries. Click the **OK** button to save this change.



When converting in this manner, all parameters will be created at **QueryStrings**. It is highly recommended that you modify your new Named Queries so that these values become **Value Parameters** instead, as mentioned later on this page.

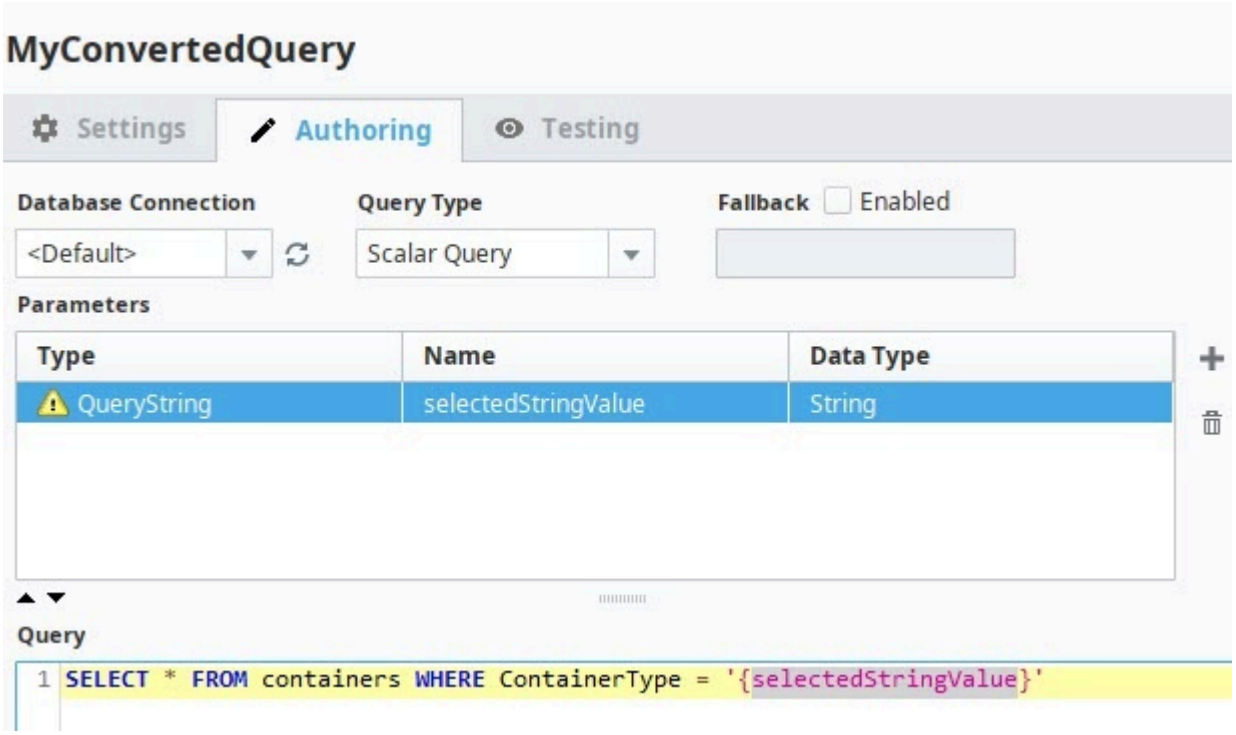
Modifying Converted Query to Use Parameters

When converting a query to a Named Query, it is **strongly** recommended that you go back into the query and convert the parameter from a **QueryString** type to a **Parameter** type. To do this, you first need to change all QueryString type parameters to the Parameter type. This is as simple as selecting Parameter from the dropdown under Type.

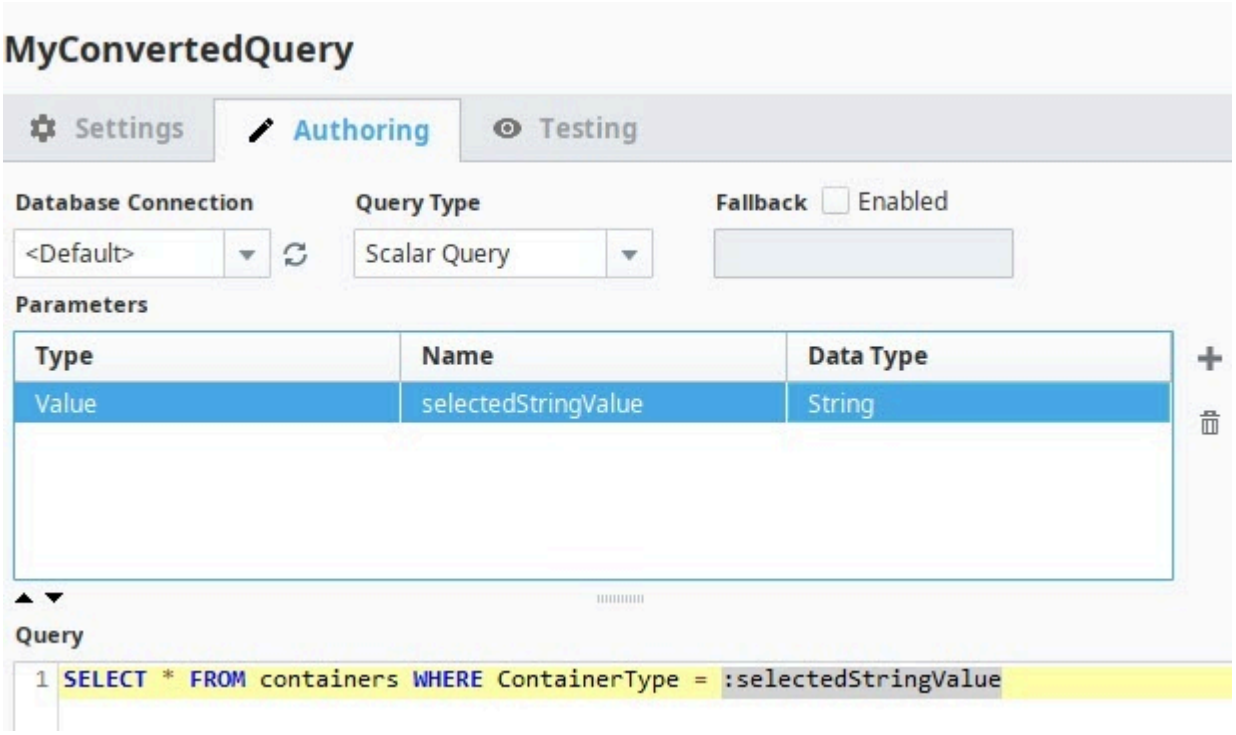
Now that your parameter(s) types have been modified, the second thing you need to do is modify syntax in the query to use the parameter(s). The Parameter and QueryString types are referenced differently in the query, so you will need to ensure that you modify how the parameter is referenced. In addition because the Parameter type works like a prepared statement, it does not need any quotation

marks around any string type parameters like a QueryString would, so all quotation marks around parameters should be removed from the query. See the images below for an example.

From this:



To this:



Once the syntax in the query has been modified, test it out in the **Testing** section at the top to make sure everything works correctly. If so, then you have successfully converted to a Named Query.

Using Named Queries - Example

A Named Query Example

Here we will go over the steps necessary to put together a basic named query. This example will create a Named Query that uses a single parameter to run a select query, and then add a Table to a window and create a binding that uses our new Named Query.

We are going to be querying a table in the database that holds information about products stored in storage bays. The database table is named "containers" and has the following structure.

id	StorageBay	ContainerType	ItemName	Weight	Time
1	1	Jug	Vanilla	25.2	2017-06-25 15:58:47
6	1	Mason Jar	Chocolate	12.3	2017-06-26 16:05:27
18	2	Tray	Swiss Cheese	88.8	2017-06-25 01:21:31
22	3	Tray	Cheddar Cheese	54.7	2017-06-25 03:52:16
23	3	Basket	Strawberry	36.8	2017-06-25 15:56:03
31	3	Jug	Whole Milk	80.1	2017-06-27 09:51:31
32	3	Jug	Fat Free Milk	76.9	2017-06-27 09:52:52

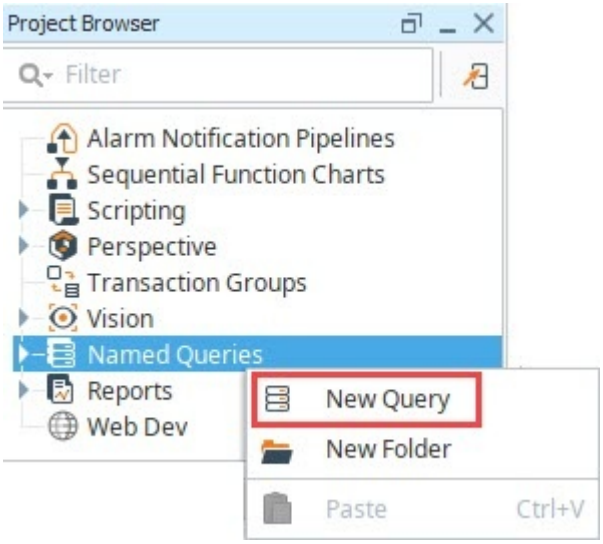
If you want to follow along with the example, feel free to make a database table that looks similar to this one and add as many rows of data as you would like, otherwise, you can use your own and

substitute in the proper column names from your database.

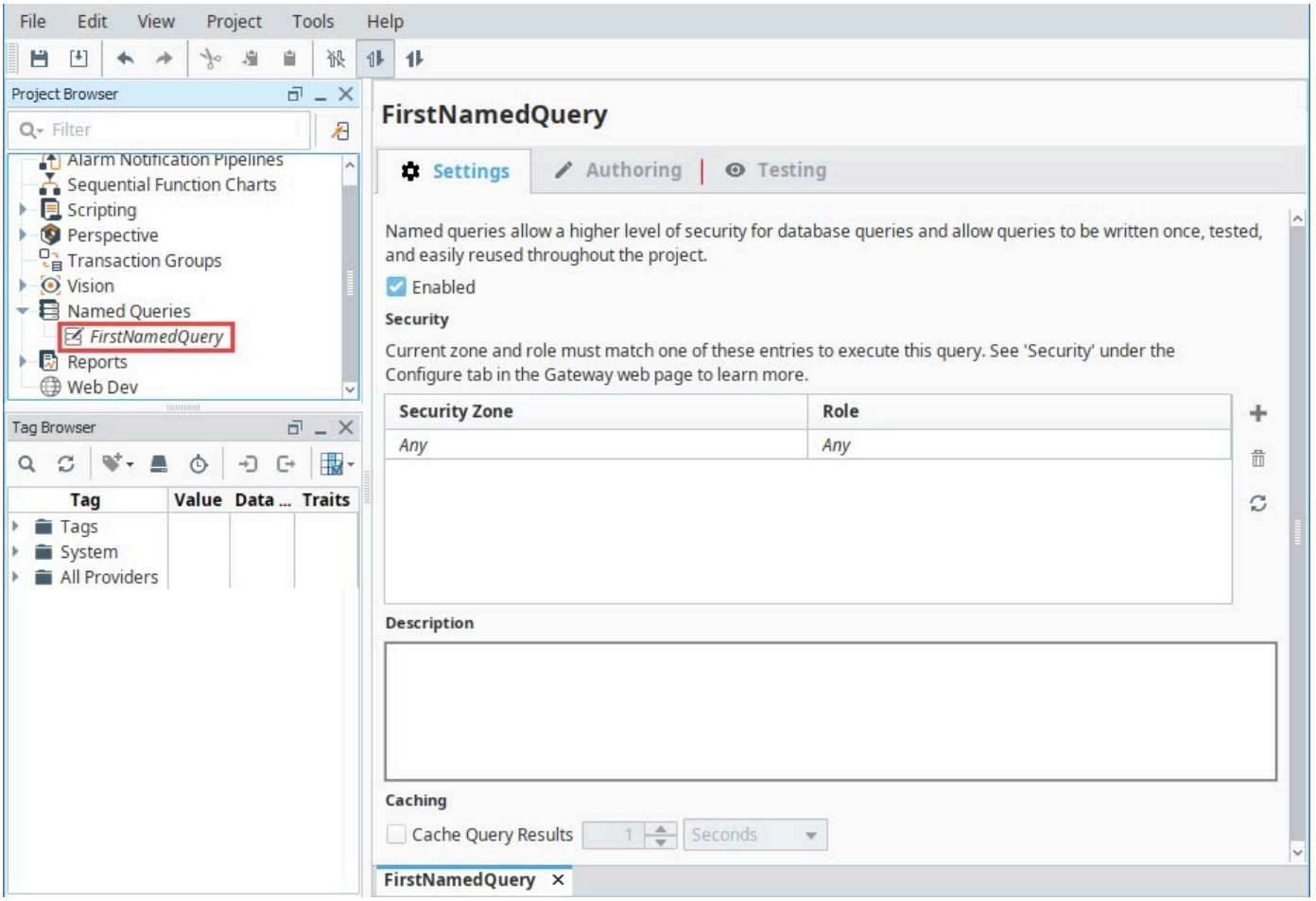
Creating a Named Query and Adding Security

1. Start by opening up the Designer and loading a project.

2. Locate the Named Query section of the project browser, right click on it, and select the **New Query** option.

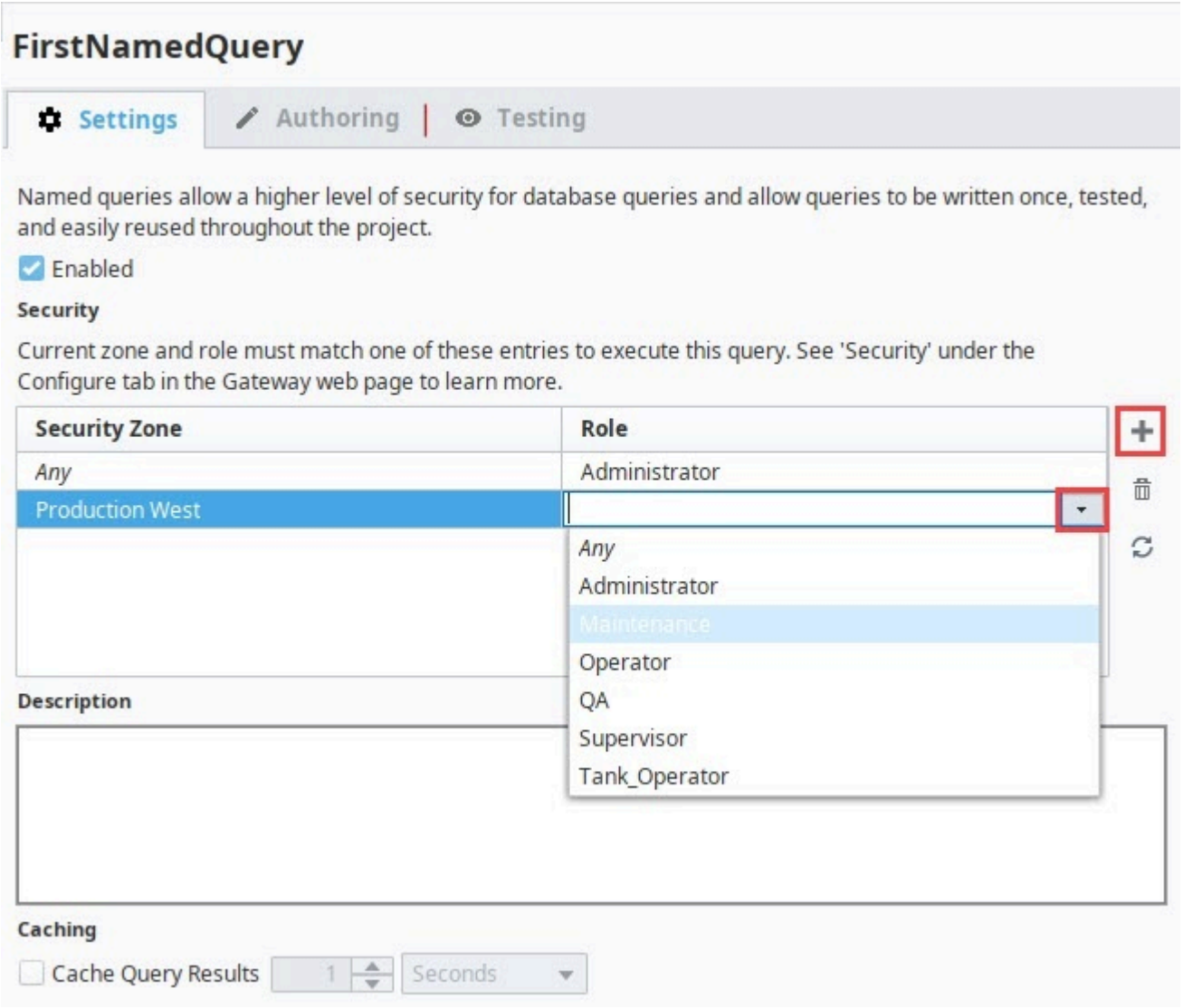


3. You should now have a fresh Named Query that you can rename whatever you want. We used **FirstNamedQuery** in the example.



4. To set up security on the Named Query, set required Security Zone and Role combinations in the Security table of the Settings tab. You can leave this blank if you don't have **roles** or **zones** set up yet.

5. If multiple security combinations are required, use the Add (✚) icon to add additional rows.



Building the Query

1. Click on the **Authoring** tab. Here is where we do most of the work.

2. Under **Database Connection**, we need to select a database connection that this named query will use. We selected **<Default>**.

3. For the **Query Type**, we can decide what type of query this will be. For this example, We am running a select query that will return a dataset, so we choose **Query**.

4. In the Parameters section, we can decide on a list of parameters that will be used in this query. This query is fairly simple, and will only use a single parameter. Click the Add (✚) icon to add a new parameter and set the following values:

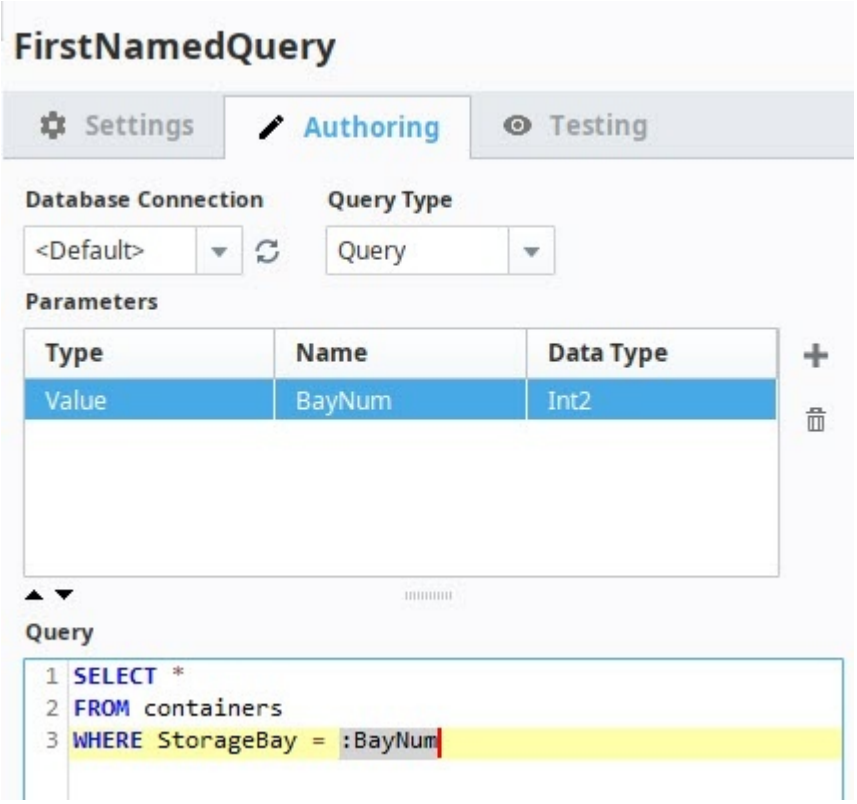
- **Type**: Parameter
- **Name**: BayNum
- **Data Type**: Integer

5. The Query section below is where we construct our query using the Table Browser.

   i. Right click on the containers table in the Table Browser on the right, and click on **Create SELECT Statement**. This will populate our query field with a basic select all statement.

   ii. Type into the **Query** field and add the following WHERE clause: **"WHERE StorageBay = "**

   iii. Now drag the **BayNum** parameter from the Parameters table to the end of the query you just typed. Notice "**:BayNum**" will be added at the end of the query.



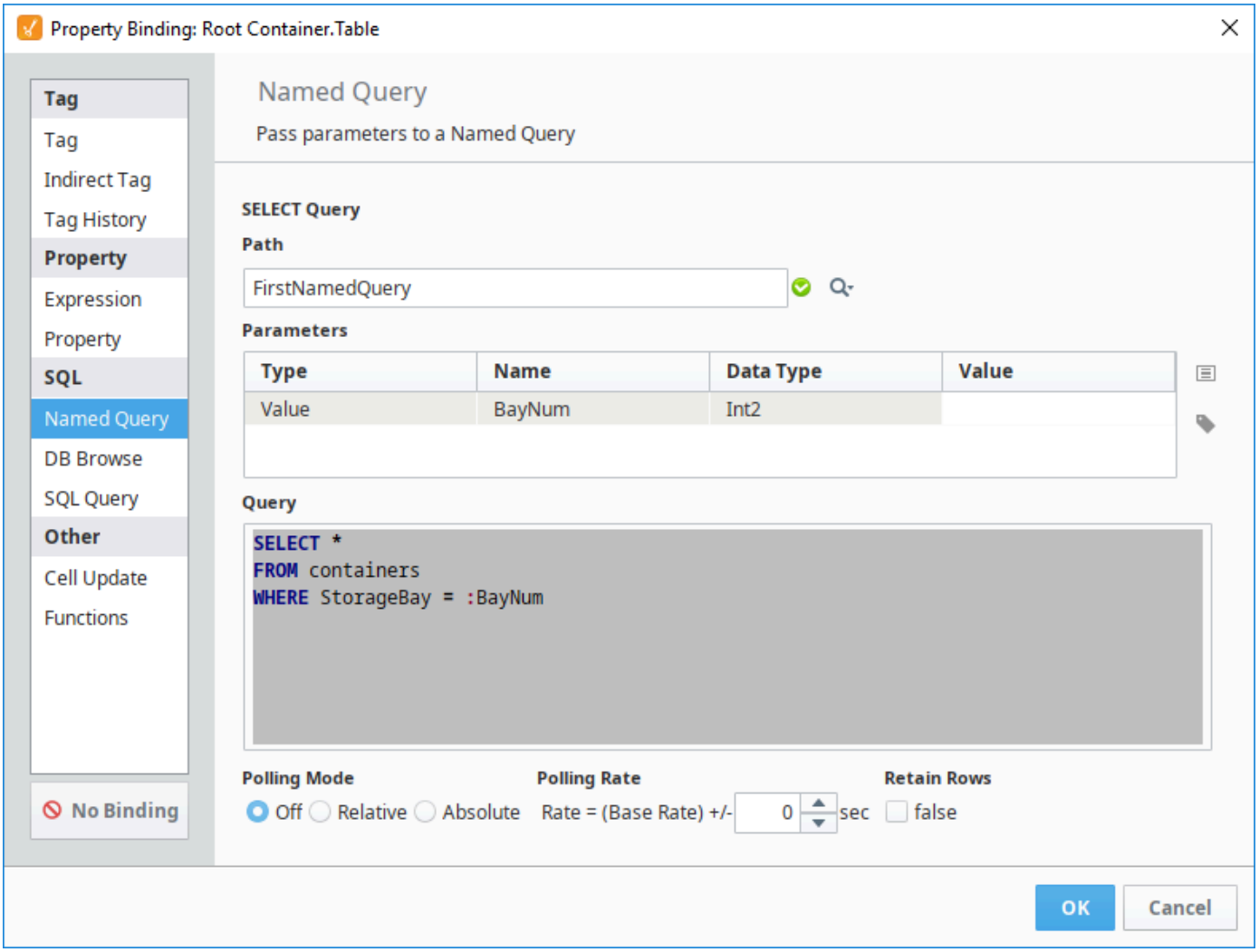
Using the Query

1. In the Project Browser, create or open a **Main Window**.

2. Drag a **Dropdown List** component onto the window.

3. The Dropdown List is where we will be able to select a Bay Number to use as our Named Query's BayNum Parameter. Use the **Dataset Viewer** to set the **Data** property of the Dropdown to look like this:

Value	Label
1	Bay 1
2	Bay 2
3	Bay 3



4. Now drag a **Table** component onto the window. We can setup a Named Query binding on the Data property.

5. Click on the **Binding** (🔗) icon for the **data** property and select the **Named Query** binding type.

6. Set the Path property by clicking on the **Select Resource Path** (🔍) icon and selecting your new Named Query from the list. For this example, it is **FirstNamedQuery**. Alternately, you can type the name in.

7. Highlight your **BayNum** Parameter (in the Parameters table) and click on the Insert Property (📝) icon. We want to select the **Selected Value** property of our Dropdown List.

8. Finally, we want to ensure the Polling Mode is set to **Off**. This means the query will not run continuously, but will only run when it changes such as when a new bay is selected from the dropdown.

9. Click the **OK** button to save your binding, and put your Designer into **Preview Mode** (▶) to test it. We can then make a selection (i.e., Bay 3) with our dropdown, and see our table populate with data.

id	StorageBay	ContainerType	ItemName	Weight	Time
22	3	Tray	Cheddar Cheese	54.7	Jun 25, 2017 3:5...
23	3	Basket	Strawberry	36.8	Jun 25, 2017 3:5...
31	3	Jug	Whole Milk	80.1	Jun 27, 2017 9:5...
32	3	Jug	Fat Free Milk	76.9	Jun 27, 2017 9:5...

Queries in Scripting

Overview

In addition to using a binding, queries can be called from a Python script. This typically involves using one of Ignition's built-in **scripting functions**. This page presents several approaches to interacting with a database from a Python Script. See the **Scripting** section for more information about using Python.

Named Queries

Named Queries may be called from a Python script using the **system.db.execUpdate** and **system.db.execQuery** functions. Named Queries can execute as a Prepared Statement when not utilizing **Query String parameters**, so they can offer a secure approach to calling a query from a script.

Additionally, Named Queries are incredibly easy to call from a script, since you only need to know their name, and don't have to bother with SQL syntax in the script. This is another way to protect your database from other users in the Designer, as you can **Protect Named Queries**.

Python - Execute an Update Query a Named Query

```
namedQuery = "Add New Order"
parameters = {"accountId":123, "productName":"Bananas"}

system.db.execUpdate(namedQuery, parameters)
```

For more information, refer to the **Named Queries** section.

Prepared Statements vs Standard Statements

A prepared statement is a feature in a database that executes a parameterized query. One of the main reasons to utilize a prepared statement is because they are resilient to SQL Injection Attacks. Because of this, we **highly recommend** you utilize Prepared Statements over Standard Statements, but Named Queries are the most secure. Especially in cases where the users can type values that will be included in the query.

Prepared Statements typically involve passing two pieces of information to the database:

- A query string that contains placeholders to represent where arguments will be passed in later. These are represented in the query as question mark characters ("?")
- A series of arguments to replace the placeholder characters in the static query.

Standard Statements vs Prepared Statements

Typical SQL insert queries look like the following:

SQL - Standard Statement

```sql
INSERT INTO orders (account_id, product_name) VALUES (123, 'Bananas')
```

A Prepared Statement instead looks like the following. Note, that the placeholders do not require quotation marks even when a string will be passed in.

SQL - Prepared Statement

```sql
INSERT INTO orders (account_id, product_name) VALUES (?, ?)
```

How do I Call a Prepared Statement?

Prepared Statements can be called from a script using specific functions. Typically, they contain "Prep" in the name such as **system.db.runPrepQuery**, or **system.db.runPrepUpdate**. When in doubt, take a look at the sub pages in the **system.db** section.

There are typically two required parameters with these functions: a string literal that represents the query, and a list of parameters.

Python - Calling an Insert Query as a Prepared Statement

```python
query = "INSERT INTO orders (account_id, product_name) VALUES (?, ?)"
args = [123, "Bananas"]
system.db.runPrepUpdate(query, args)
```

Notable Prepared Statement Functions

- **system.db.runPrepQuery**: Should be used when SELECT statements are used, as it returns a dataset containing the results. This function is effectively "read-only", as it does not manipulate

data in the database.

- **system.db.runPrepUpdate**: Should be used when manipulating the database in some way. When executing a statement that utilize INSERT, UPDATE, or DELETE, the runPrepUpdate function should be called. Note that it does return the number of rows affected, so the return value can be used to keep track of how much of an impact the query had.

Stored Procedures

If your database administrator has already configured Stored Procedures for you to use, then they can easily be called from a Python Script. Using Stored Procedures in a script typically involves two main steps:

1. An SProcCall object is created with the **system.db.createSProcCall** function. The SProcCall object contains several functions that can be used to register parameters, and access the results set returned by the Stored Procedure after it has been executed.
2. The **system.db.execSProcCall** function must be used to execute the Stored Procedure.

Python - Creating and Executing a Stored Procedure

```
# Create a SProcCall object, which will be used to configure
# parameters on the Stored Procedure, and then executed.
myCall = system.db.createSProcCall("insert_new_order")

# Register parameters on the SProcCall object.
myCall.registerInParam(1, system.db.INTEGER, 123)
myCall.registerInParam(2, system.db.VARCHAR, "Bananas")

# Execute the Stored Procedure.
system.db.execSProcCall(myCall)
```

Take a look at the **SQL Stored Procedures** page for more details.

Transactions

A SQL Transaction can also be executed from a script. For the unfamiliar, a Transaction is a batch of statements that will be executed together, and either succeed or fail as a group. Note, that the statements executed in the Transaction are not visible by other connections in the database until you commit them.

Transactions typically involve several steps:

1. Call **system.db.beginTransaction**. This returns a **transaction identifier** that can be used with other statements. Using this identifier is how you specify that a statement should be included in the transaction.
2. Start calling other statements with other functions, such as **system.db.runPrepUpdate**. The function's "tx" parameter will be passed the **transaction identifier**.
3. Commit or Rollback the transaction. Use **system.db.commitTransaction** to commit, and **system.db.rollbackTransaction** to rollback. These options are essentially the same as applying or canceling the results of the queries. Committing will make the updated results available to other database connections.
4. Close the Transaction once you're done, which can be accomplished with the **system.db.closeTransaction** function. This invalidates the **transaction identifier**.

Python - The SQL Transaction Workflow

```python
# 1) Begin the transaction. This returns a transaction identifier that we can use with other statements.
transactionId = system.db.beginTransaction(timeout = 5000)

# 2) Now we can execute statements. Because we want these to run as part of the transaction,
# we need to include our identifier.
query = "INSERT INTO orders (account_id, product_name) VALUES (?, ?)"
args = [123, "Bananas"]
system.db.runPrepUpdate(query, args, tx = transactionId)

# 3) We can continue to add statements, but in this case we'll commit them.
# We could instead rollback if there was an issue with our previous statement.
system.db.commitTransactions(transactionId)

# 4) We're done, so close the Transaction.
system.db.closeTransaction(transactionId)
```

Common SQL Tasks

This section contains examples for items we identified as "common" tasks. When first starting out with **SQL in Ignition**, many users are looking for examples in order to grasp concepts, or examples to possibly use in their projects. Additionally, this section aims to demystify some of the more complex or abstract tasks that our users may encounter when working with SQL in Ignition.

The examples in this section document several types of SQL tasks that may also touch many other areas of Ignition. While these examples are typically focused on a single concept or end result, they can easily be expanded or modified. 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 SQL tasks related to using SQL in Ignition.

Filter Data in a Table

The **Filter Data in a Table** section explains how using a simple Dropdown box and making some modifications to the SQL query can allow users to dynamically filter data coming into the table to only show a specific subset of data, or all of the data. This section also includes variations of the SQL query, thus returning different filtered results to the table.

Add Data to a Database

A common method of **adding data to a database** from within a client is using a Button component that executes an SQL query. This section provides an SQL script showing how to collect relevant properties and insert them into a database, as well as how to setup your components on a window.

Edit Data in a Database

Editing data in a database is very similar to how we add data to a database. Users can edit data within a table on the screen in realtime, and the changes are then pushed back to the database table. This section goes over the different ways to edit data in the database depending if you're using a Table component or Power Table component.

Refreshing SQL Data on a Component

Sometimes, it may be unnecessary to have a table constantly updating and requerying the database for data. By using the **system.vision.refresh** function, we can turn Polling "Off" on our query, and have a button that allows us to manually update the table with new data when the Button is pressed. Alternatively, we can add the system.vision.refresh function at the end of a script to refresh the newly entered data automatically without pressing a button.

Editing Multi-Selected Rows from Table

Tables have the ability to select and **edit multiple rows** at a time. Oftentimes, this can be used to manipulate multiple rows simultaneously, such as deleting all of the selected rows at once.

Storing PDF Files in a Database

The database is a powerful tool and can allow you to store files such as PDFs in a database table. The **Storing PDF Files in a Database** section explores how we can take a PDF file and store it in the database table, as well as how to pull it back out and display it in the **Vision - PDF Viewer** component.

Simple Database Editor

It may be beneficial to have a frontend to a database table built right into an Ignition project that allows you to control the data in the table without having to go into the database's built-in frontend, such as MySQL workbench. In the **Simple Database Editor** section, you will learn how to build a simple database table editor on a window that can add, edit, and delete data from the database.

Filter Rows in a Table

Creating the Components

Filtering table data by using a Dropdown box is possible if the **Table** component's **Data** property is bound to a SQL query. We can create a dynamic `WHERE` clause that will allow us to select all the data or a specific subset of it. To implement this solution, you must have a **Table** component and a **Dropdown List** component on a window. With a Dropdown component, we can create an option for each way of filtering the data, each with a unique value associated with it. An additional "all" entry can also be added, with its own unique value.

We can then use the value of the Dropdown to drive the query. When you select a different option from the Dropdown component the binding gets re-evaluated on the table's data binding resulting in the query executing the WHERE clause with the new parameters. You may want to turn Polling Mode off on the tables Data property binding in order to limit the periodic querying of the database.

Here we can use a value of 1 to show only entries which match that filter column value.

Pseudocode - Filtering for Area 1

```
SELECT * FROM table WHERE filterColumn = 1
```

While the following query will return everything, but will still evaluate the query's WHERE clause.

Pseudocode - Return All Rows

```
SELECT * FROM table WHERE 0=0
```

In this case 0=0 will always evaluate as true, therefore, the query will return every row.

We can combine the logic of each WHERE with an OR into a single query, and substitute in our Dropdown value. The SQL Query binding would then look something like this:

SQL - Where Clause Combined with OR Condition

```
SELECT * FROM table WHERE filterColumn = {dropdown value} OR 0 = {dropdown value}
```

This way, the WHERE clause will only be true when the filter column matches with our selected option, or it will return all rows, if we setup our "all" option to have a value of 0.

Example - Filtering Data on an Area Number

We can put together an easy example on how this might with data that stores what machines are in which areas. Our data table should look something like this:

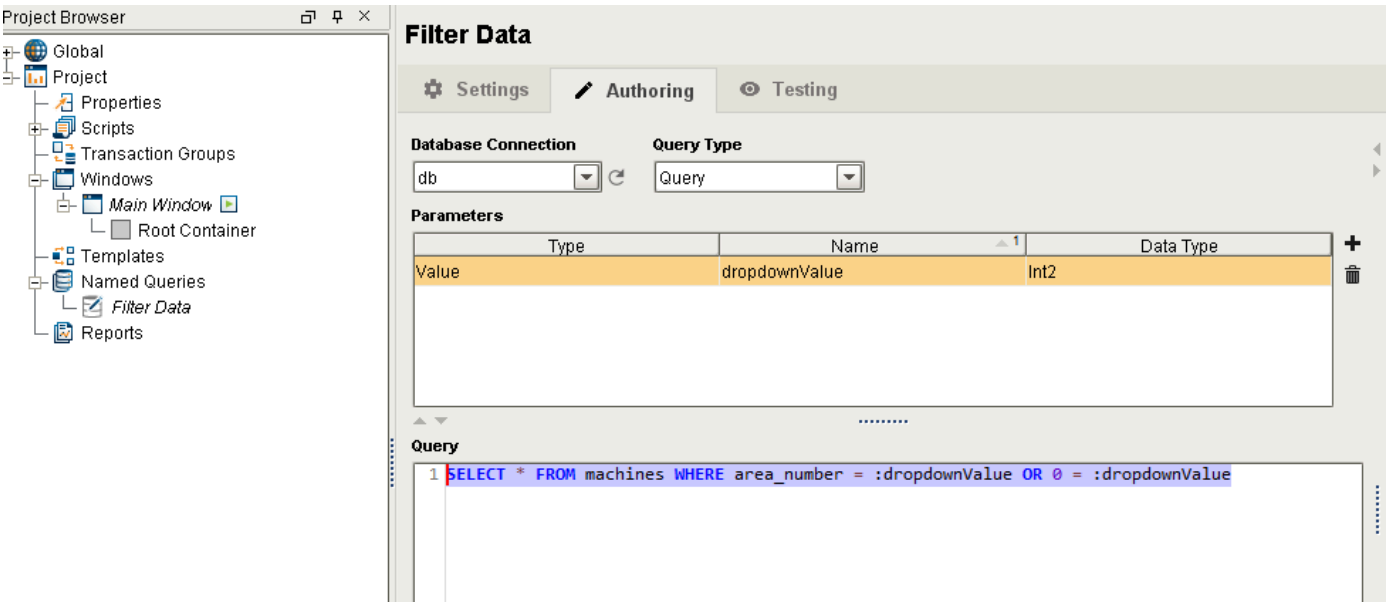
id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

We can then put together the query and components necessary to get this working:

1. Create a new Named Query. Set up security to fit your needs, and name it appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.
    i. Create a single Value type Parameter that is an Int2 data type. I called mine dropdownValue
    ii. Add in the combined query that we went over above, but use the machine table name and column names.

SQL - Selecting Values from a Table and Filtering on a Dropdown

```
SELECT * FROM machines WHERE area_number = :dropdownValue OR 0 =
:dropdownValue
```



2. Create a new Main Window and add a Power Table component and a Dropdown component to the window.

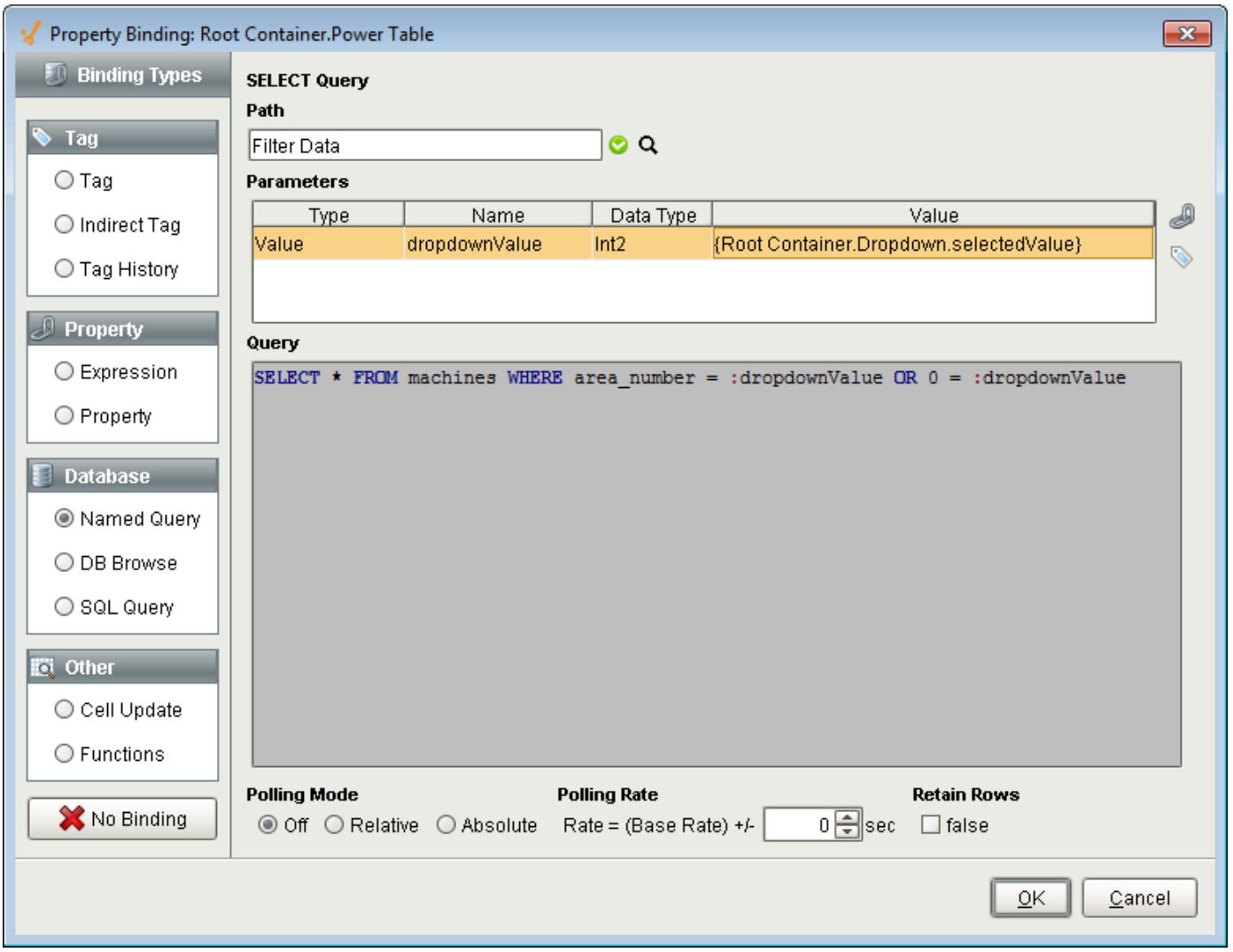
    i. On the Dropdown's **Data** property, create a dataset that looks like this:

Value	Label
1	Area 1
2	Area 2
3	Area 3
0	All Areas

3. On the Power Table component, create a Named Query Binding on the **Data** property.

    i. Select the Named Query that was created in step 1.

    ii. For the parameter, bind its value to the **Selected Value** property of the Dropdown component, and the click the **OK** button.



4. Put the Designer into **Preview** mode and try out the Dropdown Filter. You will see that selecting a different value in the dropdown filters the data coming back so that only certain rows are shown.

id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

All Areas

Inserting Data into a Database

A common way to insert information into a database is to run a script after a user presses a Button component that will collect data and execute a **Named Query** to push that data into a database table. Using a script allows values to be collected from tag values and property values as needed to insert into a table. The following syntax shows how this script can be structured.

Pseudocode - Collect Data and Insert into the Database

```
value1 = {component property reference}
value2 = {tag value}
value3 = {static value}

query = "INSERT INTO table (col1, col2, col3) VALUES (?,?,?)"
args = [value1, value2, value3]
system.db.runPrepUpdate(query, args)
```

It is important to note that if you choose to use the **runPrepUpdate** function in Vision, you'll be required to enable certain **Client Permissions**.

Inserting Values on a Button Press Example

This example demonstrates how to insert data into a database for both Vision and Perspective using the database table and Named Query created in the steps below:

1. Open the **Database Query Browser** from the top navigation bar under Tools.

2. Create a **machines** table in a database with a **machine_name** and an **area_number** column. This example uses the following statement for the Default database.

   Create Table

   ```
   CREATE TABLE machines (
     machine_name varchar(255),
     area_number int
   );
   ```

3. Open the **Named Queries** workspace from the Project Browser.

4. Enter **Insert Values** into the New Query name field.

5. Select **Update Query**.

6. Select the **Authoring** tab.

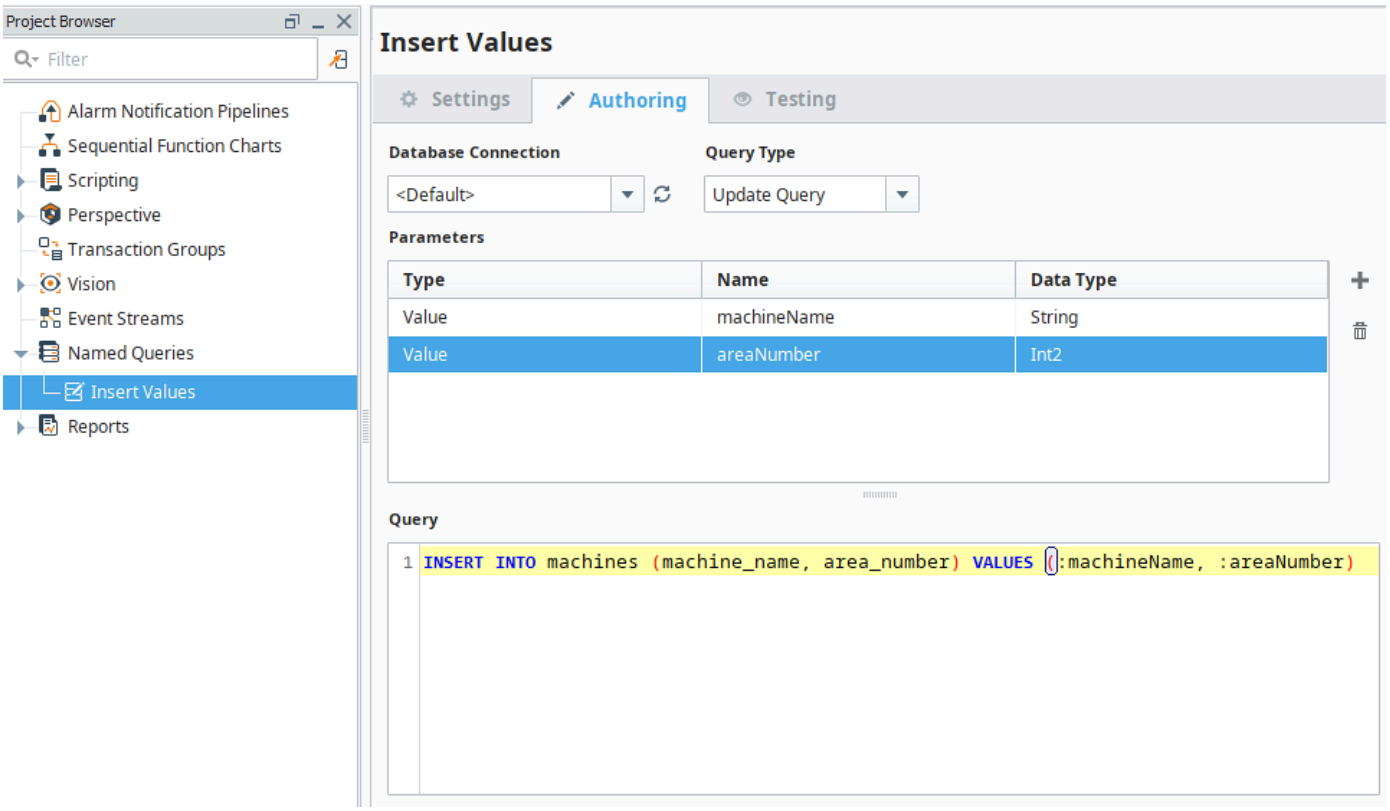
7. Update the two default Value type Parameters.

    i. Rename the first value to **machineName** and leave the data type as **String**.

    ii. Rename the second value to **areaNumber** and change the data type to **Int2**.

8. Enter the following Named Query statement:

```
SQL - Insert Values

INSERT INTO machines (machine_name, area_number) VALUES (:machineName,
:areaNumber)
```



Refer to the Vision and Perspective sections for the respective steps to build out a window or view for the rest of this example.

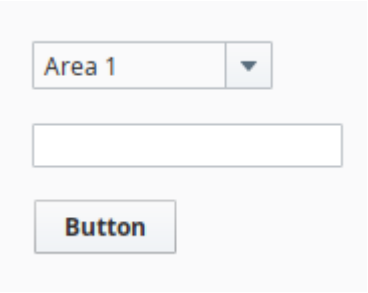
Vision

1. On a Main window, add a Button component, a Text Field component, and a Dropdown component.

2. On the Dropdown's `data` property, create a dataset that contains the following values and labels:

Value	Label
1	Area 1
2	Area 2
3	Area 3

3. Change the Button's `text` property to **Submit**.



4. Right-click on the **Button** and select **Scripting**.

5. Navigate to the **Script Editor** tab under Event Handlers > action > actionPerformed.

6. Enter the following script, which will pull in the values from the Dropdown and the Text Field, and then insert them into the table using the Insert Values Named Query created above.

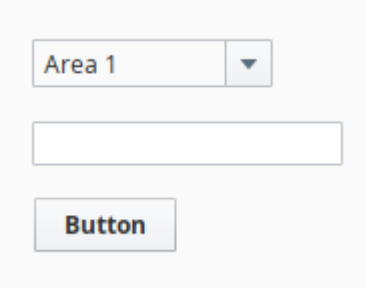
Python - Button actionPerformed Script

```
# Grab the area number and machine name from the components we added to the window.
areaNum = event.source.parent.getComponent('Dropdown').selectedValue
machineName = event.source.parent.getComponent('Text Field').text

# A call to our Named Query, inserting the two parameters using dictionary syntax.
system.db.execUpdate("Insert Values", {"machineName":machineName, "areaNumber":areaNum})
```

Verify functionality by placing the Designer in Preview mode, selecting an area, entering in a machine name, and clicking Submit. Open the Database Query Browser and execute a SELECT query to see the new row.

Perspective

1. Drag a Button component, a Text Field component, and a Dropdown component onto a view.
2. On the Dropdown's `options` property, add three array elements to enter the following value/label pairs:
    - options.0
        - value: 1
        - label: Area 1
    - options.1
        - value: 2
        - label: Area 2
    - options.2
        - value: 3
        - label: Area 3
3. Change the Button's `text` property to **Submit**.



4. Right-click on the **Button** and select **Configure Events...**.
5. Select **Component Events** > **onActionPerformed**.
6. Click the **Add** + icon and select **Script**.
7. Enter the following script, which will pull in the values from the Dropdown and the Text Field, and then insert them into the table using the Insert Values Named Query created above.

Python - Button actionPerformed Script

```
```

```
# Grab the area number and machine name from the components we added to the window.
areaNum = self.getSibling("Dropdown").props.value
machineName = self.getSibling("TextField").props.text

# A call to our Named Query, inserting the two parameters using dictionary syntax.
system.db.execUpdate("Insert Values", {"machineName":machineName, "areaNumber":areaNum})
```

Verify functionality by placing the Designer in Preview mode, selecting an area, entering in a machine name, and clicking Submit. Open the Database Query Browser and execute a SELECT query to see the new row.

Updating the Database through the Power Table

Updating Table Data

Table components can do more than show data from a database. A properly configured Table can make the data of the Table accessible to the Client and allow the user to edit the data in realtime. This page shows two examples using the Table and Power Table components editing data in a database in realtime.

Suppose your database had a Table like this:

Id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	PB	Peter	Burke	Enjoys cooking

Using a Power Table Component

Let's create a script on a Power Table component and the **onCellEdited** extension function that will write to the cell in the database.

> **DANGER**
>
> This function will not work using default Client Permissions settings. The Legacy Database Access will need to be enabled for this to work.

1. Drag a Power Table component to your window, and bind the **Data** property of the Power Table using a SQL Query Binding Type and the query below.

SQL - Selects All Records

```
SELECT * FROM users
```

2. Right-click on the **Power Table** and select **Customizers** > **Table Customizer**.

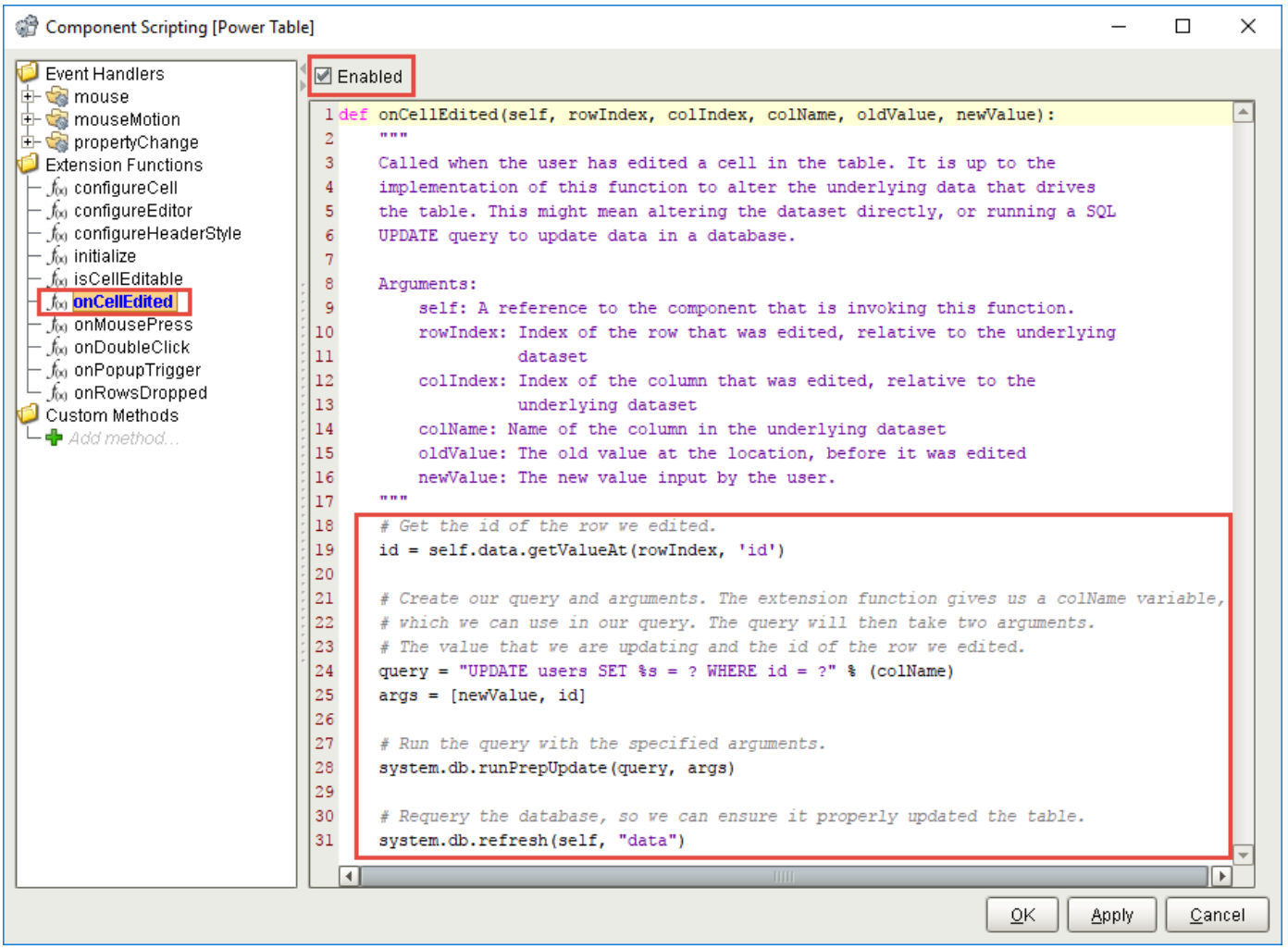
3. Select the **Editable** checkbox for each of the columns that we want to edit (i.e., same as the above example). Note, the **id** column should not be editable. Click the **OK** button.

4. Right-click the **Power Table** and select **Scripting**.

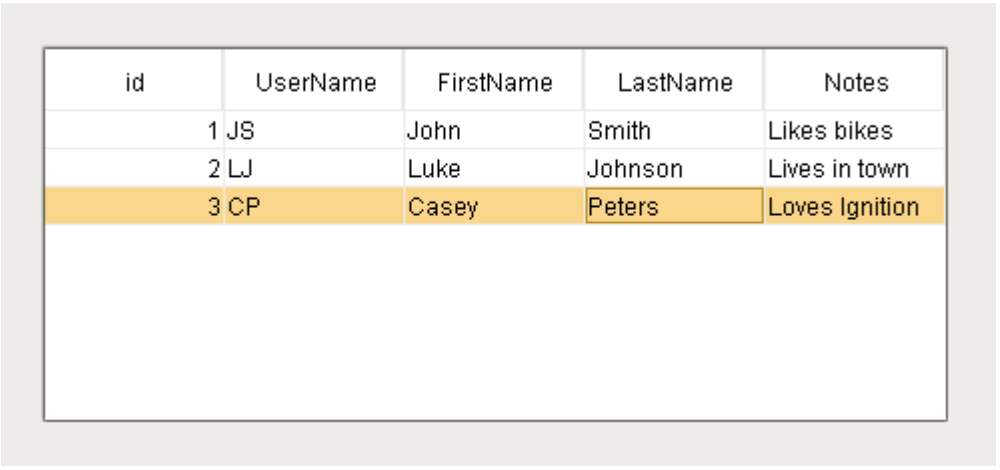
5. Under the Extension Functions folder, select the **onCellEdited** function, and check the **Enable** box. We can put a script in here that will grab the id column value of the row that we modify, and then use that and the new value that we entered to update the Database table. The extension function makes this easy, because it provides variables for the column name of the row we are editing, the row number that was edited, as well as the new value. Paste the following script in the scripting area.

Python - Writes to a Cell in the Database

```python
# Get the id of the row we edited.
id = self.data.getValueAt(rowIndex, 'id')

# Create our query and arguments. The extension function gives us a colName variable,
# which we can use in our query. The query will then take two arguments.
# The value that we are updating and the id of the row we edited.
query = "UPDATE users SET %s = ? WHERE id = ?" % (colName)
args = [newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)

# Re-query the database, so we can ensure it properly updated the table.
system.db.refresh(self, "data")
```



6. Now, let's test it out. Put the Designer in **Preview Mode**, select the cell you want to edit, hit enter to commit the change, or tab to the next cell to make additional edits.



id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	CP	Casey	Peters	Loves Ignition

Using a Table Component

Let's create another script like the example above, but this time we'll use a Table component's **cellEdited** event that will write to the cell in the database when we update it on the component. The script will be slightly different.

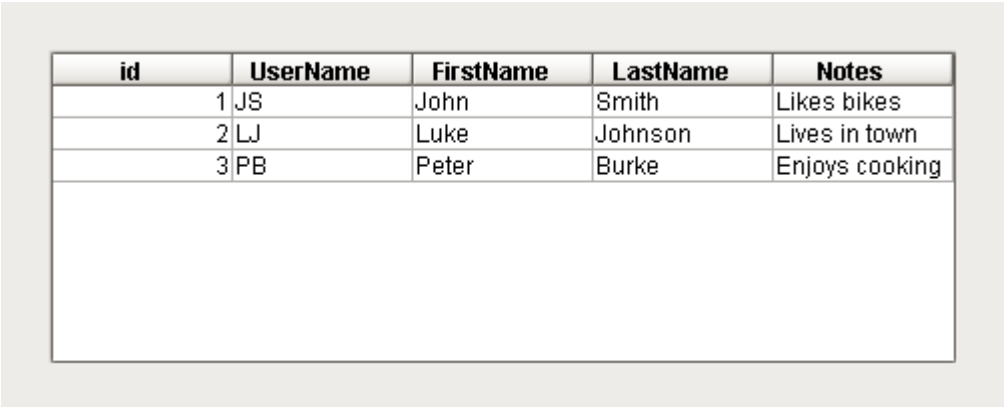
DANGER

This function will not work using default Client Permissions settings. The Legacy Database Access will need to be enabled for this to work.

1. Drag a Table component to your window. With the Table component selected, bind the **Data** property of the Table using a SQL Query Binding Type and the query below.

SQL - Select All Records

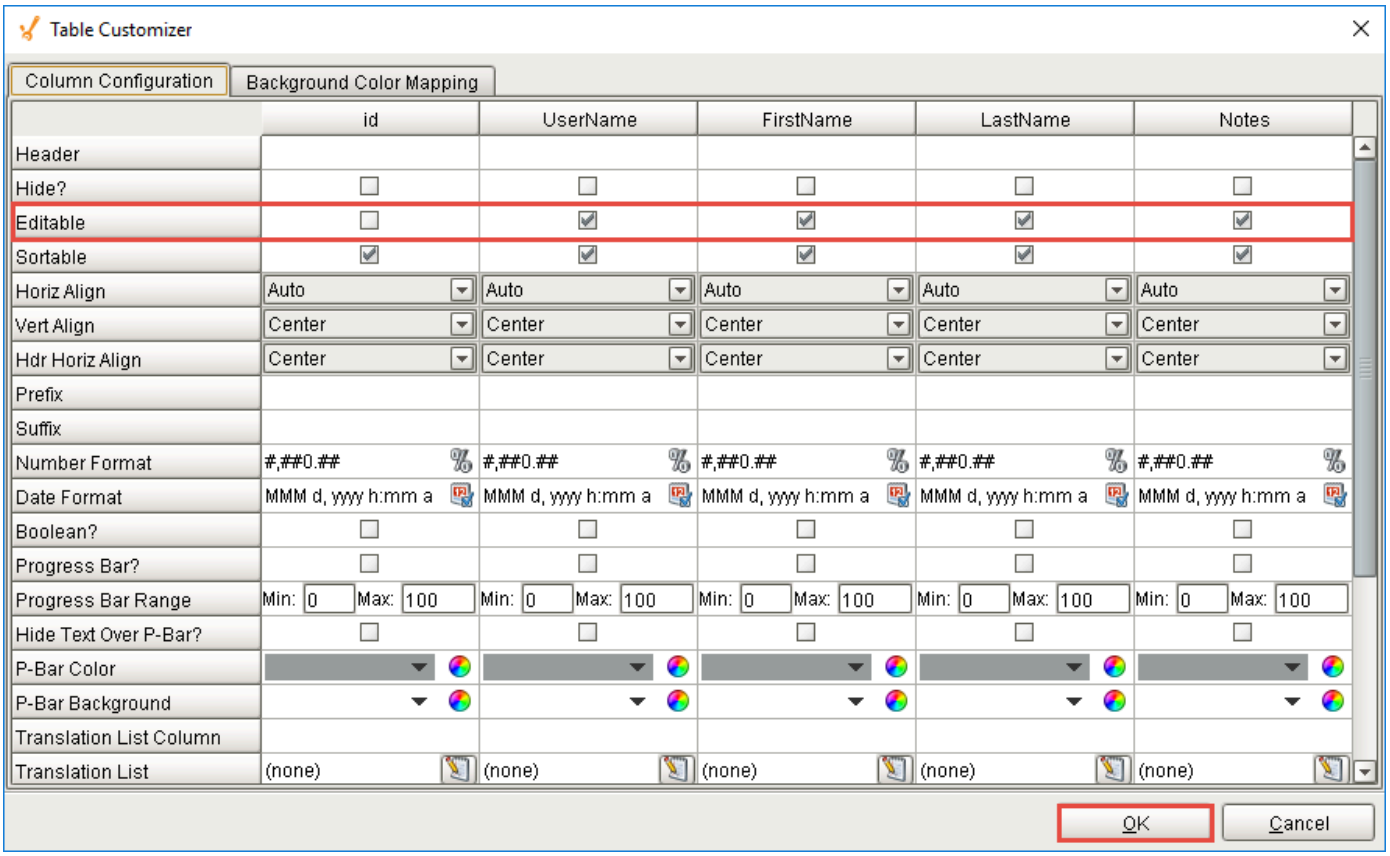
```
SELECT * FROM users
```



id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	PB	Peter	Burke	Enjoys cooking

2. Right-click on the **Table** and select **Customizers > Table Customizer**.

3. Select the **Editable** checkbox for each of the columns that we will want to edit. Note, the **id** column should not be editable. Click the **OK** button.

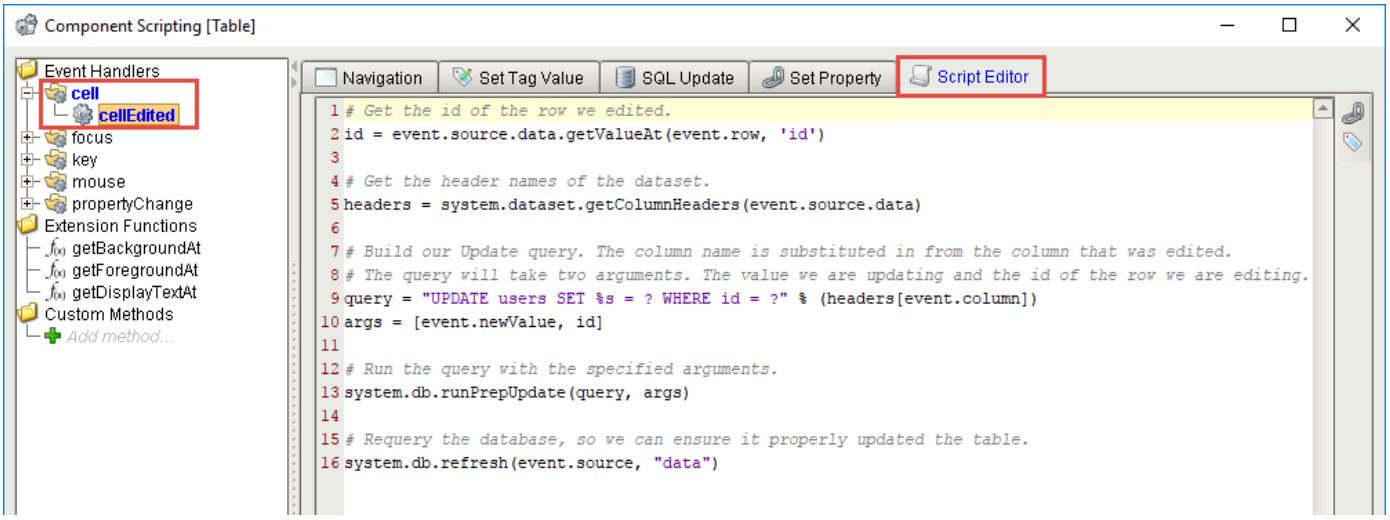


4. Right-click the **Table** and select **Scripting**.

5. Create a script in the **cell > cellEdited** event handler using the script provided below. In this script, we have variables that contain the row number that was edited, the column number that was edited, and the new value. Because we do not have the name of the column that was edited, we must first grab the list of columns using the **getColumnHeaders()** system function. We can then use the column number that was edited to find the name of the column that changed and use it in the query.

Python - Writes to a Cell in the Database When Component is Updated

```
# Get the id of the row we edited.
id = event.source.data.getValueAt(event.row, 'id')

# Get the header names of the dataset.
headers = system.dataset.getColumnHeaders(event.source.data)

# Build our Update query. The column name is substituted in from the column
that was edited.
# The query will take two arguments. The value we are updating and the id of
the row we are editing.
query = "UPDATE users SET %s = ? WHERE id = ?" % (headers[event.column])
args = [event.newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)

# Re-query the database, so we can ensure it properly updated the table.
system.db.refresh(event.source, "data")
```



6. Now, you're ready to test it out! Put the Designer in **Preview Mode**, select the cell you want to edit, hit enter to commit the change, or tab to the next cell to make additional edits.

id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	EF	Ellen	Frank	Loves Ignition

Refreshing a SQL Query

Refreshing a Query

The SQL query that populates a property on a component will refresh its data periodically if the **Polling Mode** is set to either Relative or Absolute. However, there are times when you want the data to query the database once when the window is opened, and then retain explicit control over each subsequent refresh. The Table data can be required using **system.vision.refresh**. There are two main ways of doing this: either placing the script on a button, or at the end of some code.

Refresh After Edits

If edits are being made to the Table data, or additional rows are being added, it can be a good idea to call the system.vision.refresh() function after manipulating data. This way, the data in the Table will automatically refresh with the newly entered data, saving the user the hassle of clicking a refresh button.

Pseudocode - Refreshing a Table after an Edit

```
doWork()

system.db.runPrepUpdate("INSERT INTO table")

system.vision.refresh(component, "propertyName")
```

Refresh on a Button

We can use a Button component with a script on it to refresh the component with the query binding on it. Typically the button is placed close by the query bound component, and will say something like "Refresh Data". The script is actually fairly simple, and can be placed on the **actionPerformed** event of the Button.

1. Drag a **Button** component onto a window that has a Power Table or Table component querying a Database table.

    i. Change the Button's Text property to say "Refresh."

2. On the Power Table component, open the **data** property binding and set the polling mode to **off**.

3. Right click on the Button and select **Scripting**.

    i. Select the **actionPerformed** Event Handler and navigate to the Script Editor tab.

    ii. Add in this script, which will force the Table's data property to refresh.

    Python - Refreshing the Database when the Button is Pressed

    ```
    # Will force the Power Table's Data property to run the query again.
    system.vision.refresh(event.source.parent.getComponent('Power Table'),
    "data")
    ```

4. Try it out by adding new data to the Database table and then clicking the Refresh Button.

Refreshing a Query in Perspective

The **system.vision.refresh()** function does not work in Perspective. However, an additional component method has been added to handle the more general task of refreshing a binding. The method is called refreshBinding(), and can be called on any Perspective component:

```
self.refreshBinding("props.data")
```

The function takes a string as a parameter, the path to the property to be refreshed. More info on the refreshBinding() function can be found at **Perspective Component Methods**.

Editing Multi-Selected Rows from a Table

Editing Multiple Table Rows

A common user experience is to select multiple rows of a table and then edit all of those rows from the database at once, such as deleting all of them. When a user selects more than one row in a table, there is a special function called getSelectedRows() available on both the **Table** and **Power Table** components that returns the row indexes of the table as a Python list. This list can be iterated through in order to delete the selected rows from the database.

Example - Deleting Selected Rows

We can use the list of selected rows to delete them from the database. Start with a table in the Database table that looks like this:

id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

1. Create a new Named Query that will be used to delete rows of data. **Set up Security** to fit your needs, and name it appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.

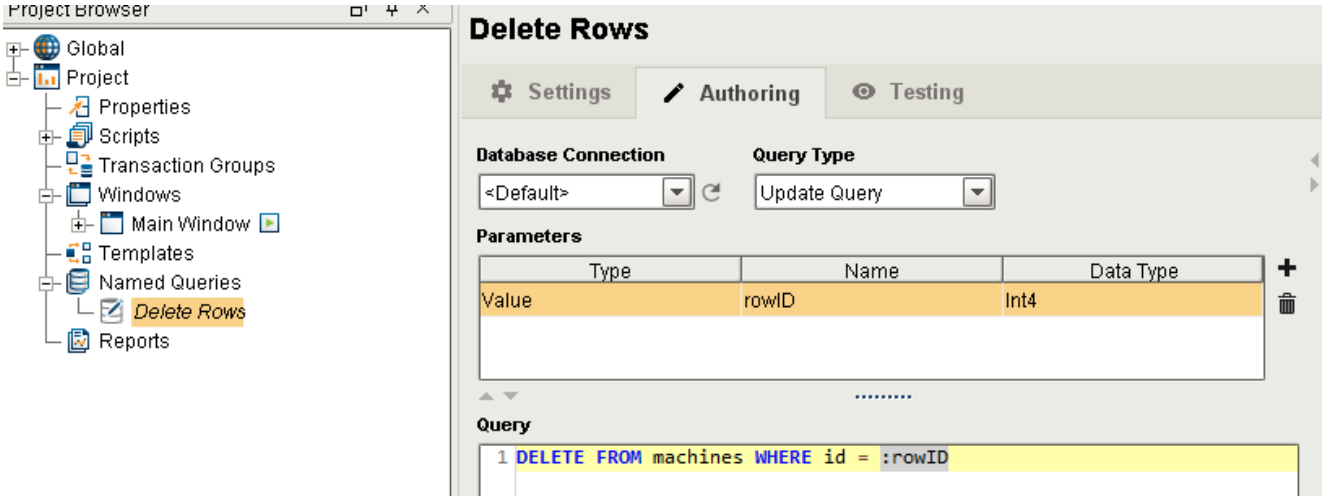
    i. Set the Query Type to **Update Query**.

    ii. Create a single Value type Parameter that is an Int4 data type. This will hold the id of the row, so we can name it rowID.

    iii. Create the Delete Query.

    SQL - Delete Row with Matching ID

    ```
    DELETE FROM machines WHERE id = :rowID
    ```



2. Create a second Named Query as in step 1, but this one will be used to select the data into a table.

    i. Set the Query Type to Query, no parameters, and add a basic select query.

    SQL - Selecting from a Table

    ```
    SELECT * FROM machine
    ```

3. On a new Main Window, add a Power Table component and a Button component.

4. On the Power Table's **Data** property, set up a Named Query binding to the Select Query that was made in step 2.

    i. Ensure that the Selection Mode property is set to **Multiple Interval**.

5. On the Button, change the Text property to say "Delete".

    i. Right click on the Button and select Scripting.

    ii. Select the actionPerformed Event Handler and navigate to the Script Editor Tab.

    iii. Here we need to call the **getSelectedRows()** function on the Power Table to determine what rows are selected, and then loop through those to grab the value of the id column in each row and delete the row based on that id.

    Python - Looping Through the Selected Rows and Deleting Them

    ```
    ```

```
# Get the data from the table and assign it to the variable called data.
data = event.source.parent.getComponent('Power Table').data

# Get the rows of the data that the user has currently highlighted.
rows = event.source.parent.getComponent('Power Table').getSelectedRows()

# Iterate through each row of the list that is associated with the rows
variable.
for row in rows:

    # Get the value associated with the current row and the the column
called "id".
    id = data.getValueAt(row, "id")

    # Run the query to delete from the database, where tableName is the
name of the database table.
    system.db.runNamedQuery("Delete Machine Rows", {"rowID":id})

# Refresh the table data.
system.db.refresh(event.source.parent.getComponent('Power Table'), "data")
```

6. You can test it out by putting the Designer into Preview Mode, selecting a few rows, and then clicking the Delete Button.

Storing Files in a Database

Storing and Displaying Files in a Database

Ignition can store different types of files into a database by storing the raw file bytes into a special database column. Ignition can also pull these file bytes out and display certain files within a Session or Client.

> **INFO**
>
> Each database has different column types that are used to store files, so it is important to check with your database documentation to see which data type the column would need to be set to for it to accept file bytes. For example, in MySQL, the datatype that accepts PDF bytes is a **LongBlob** datatype, so you will need to set the PDF Data column to the **LongBlob** datatype. MS SQL accepts the **Varbinary** datatype, so you will need to set the PDF Data column to a **Varbinary** datatype.

Perspective PDF File Example

This example shows how to store an uploaded PDF file as binary data in a database and display it in a Perspective view. It uses a database table named `files`, which includes a `fileName` text or varchar column and a `fileBytes` binary column that stores the PDF data.

Uploading PDF Files in Perspective

The File Upload component provides file contents through the `onFileReceived` event. When a user selects or drops a file, the component uploads it to the Gateway and fires this event. The event object includes the file name and file bytes, which you can write directly to the database.

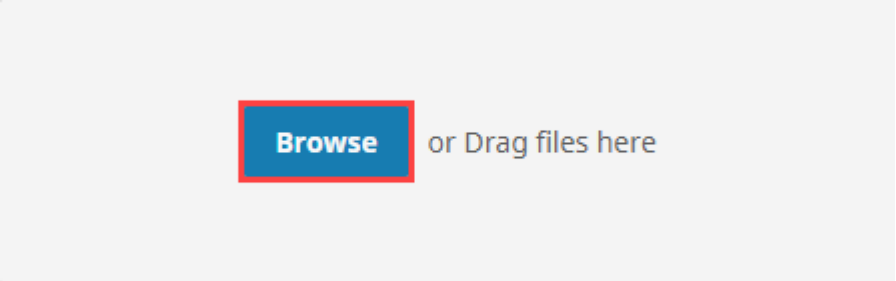
1. Add a **File Upload** component to your view.
2. Right-click the File Upload component and select **Configure Events**.
3. Select the **onFileReceived** event and add a **Script** action using the Plus ➕ icon.
4. Place the following code in the Script field and click **OK**.

Upload a File into the files Table

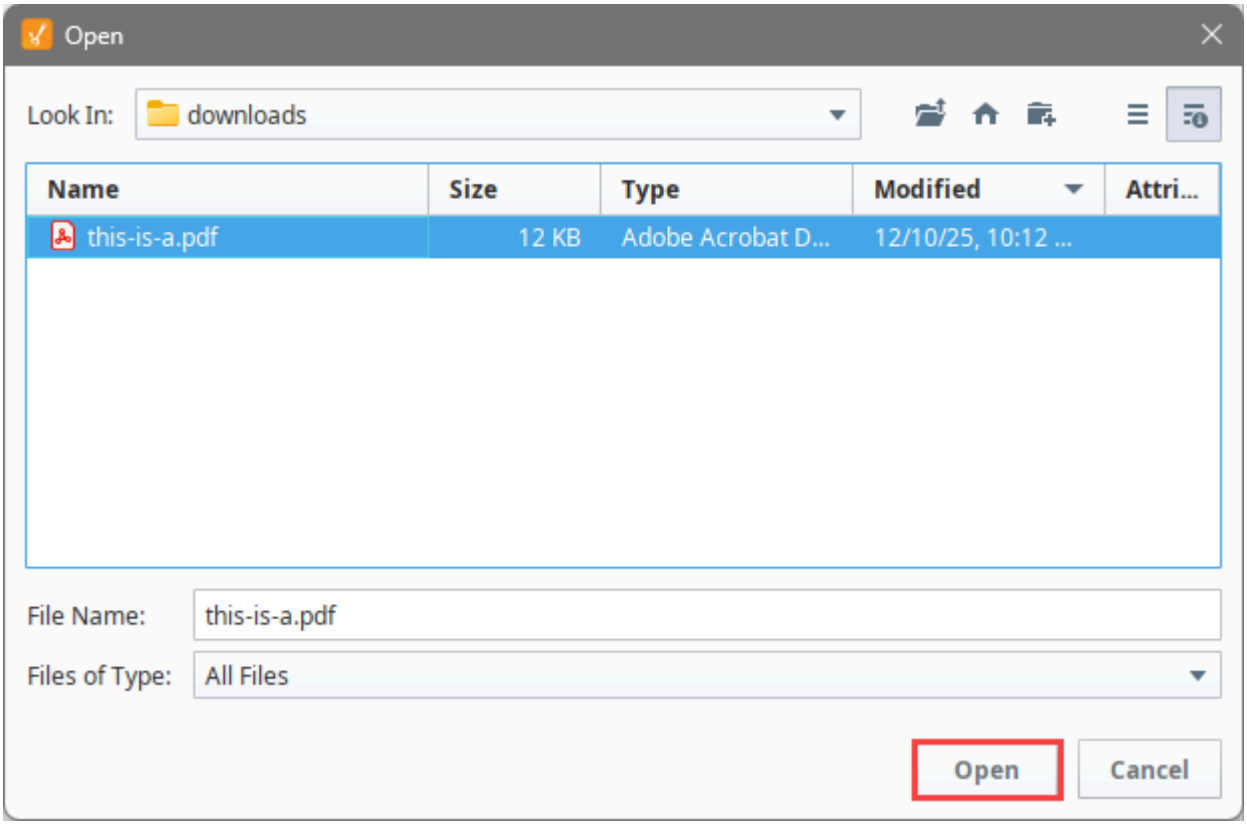
```
# Get the file name and bytes from the event
fileName = event.file.name
fileBytes = event.file.getBytes()

# Store the file in the database
system.db.runPrepUpdate(
    "INSERT INTO files (fileName, fileBytes) VALUES (?, ?)",
    [fileName, fileBytes]
)

system.perspective.print("Uploaded %s into the files table." % fileName)
```

5. On the top navigation bar in the Designer, toggle the Preview icon to enable Preview Mode.

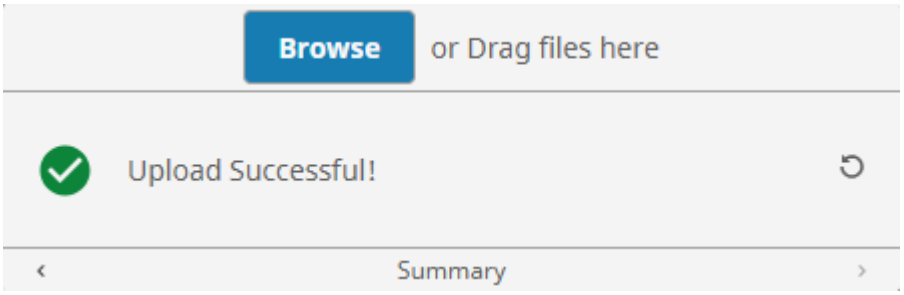
6. Click **Browse** on the File Upload component



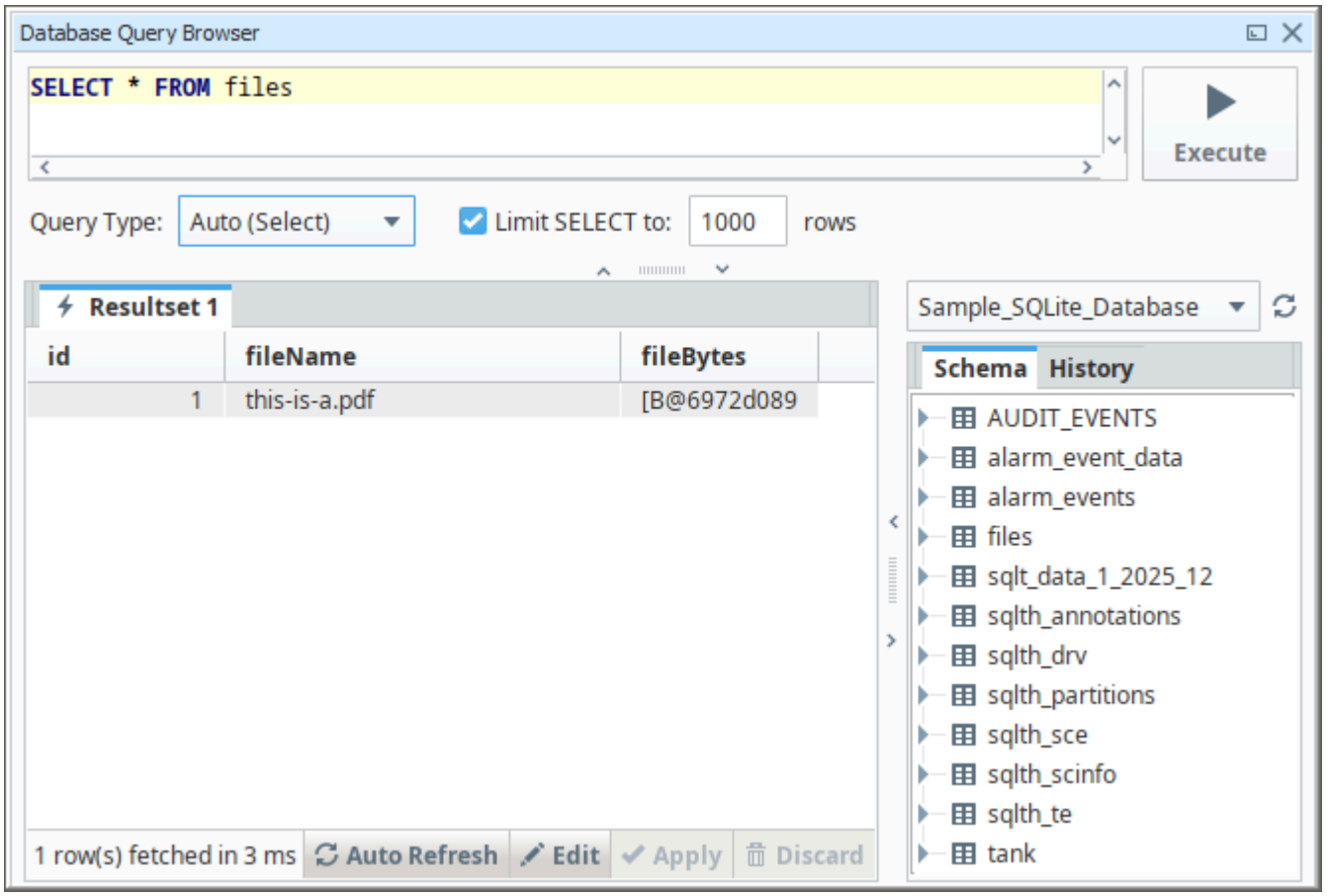
7. Select a file to upload and click **Open**.



8. A message will appear indicating a successful upload.



When a file is uploaded, the script stores the file name and bytes in the `files` table.



Displaying PDF Files in Perspective

Perspective includes a built-in PDF Viewer component that displays PDFs hosted at a URL. When a PDF is stored directly in a database as binary data, however, it does not have a web-accessible address that the browser can load. In this example, the stored file bytes are retrieved from the database, encoded as Base64, and converted into a Data URL that can be passed to a component capable of rendering the content, such as an **Inline Frame**.

> **INFO**
>
> Large PDFs may exceed browser memory limits when rendered as Base64. For large files, consider exposing the content through a WebDev endpoint instead of a Data URL.

1. Create a **Named Query** that returns the file name and bytes for a selected file:

```
SELECT fileName, fileBytes
FROM files
WHERE id = :fileID
```

2. Add a **Dropdown** component to the view and bind its **options** property to a Named Query that returns available file IDs and names. This example uses the following Named Query.

```
SELECT id, fileName
FROM files
```

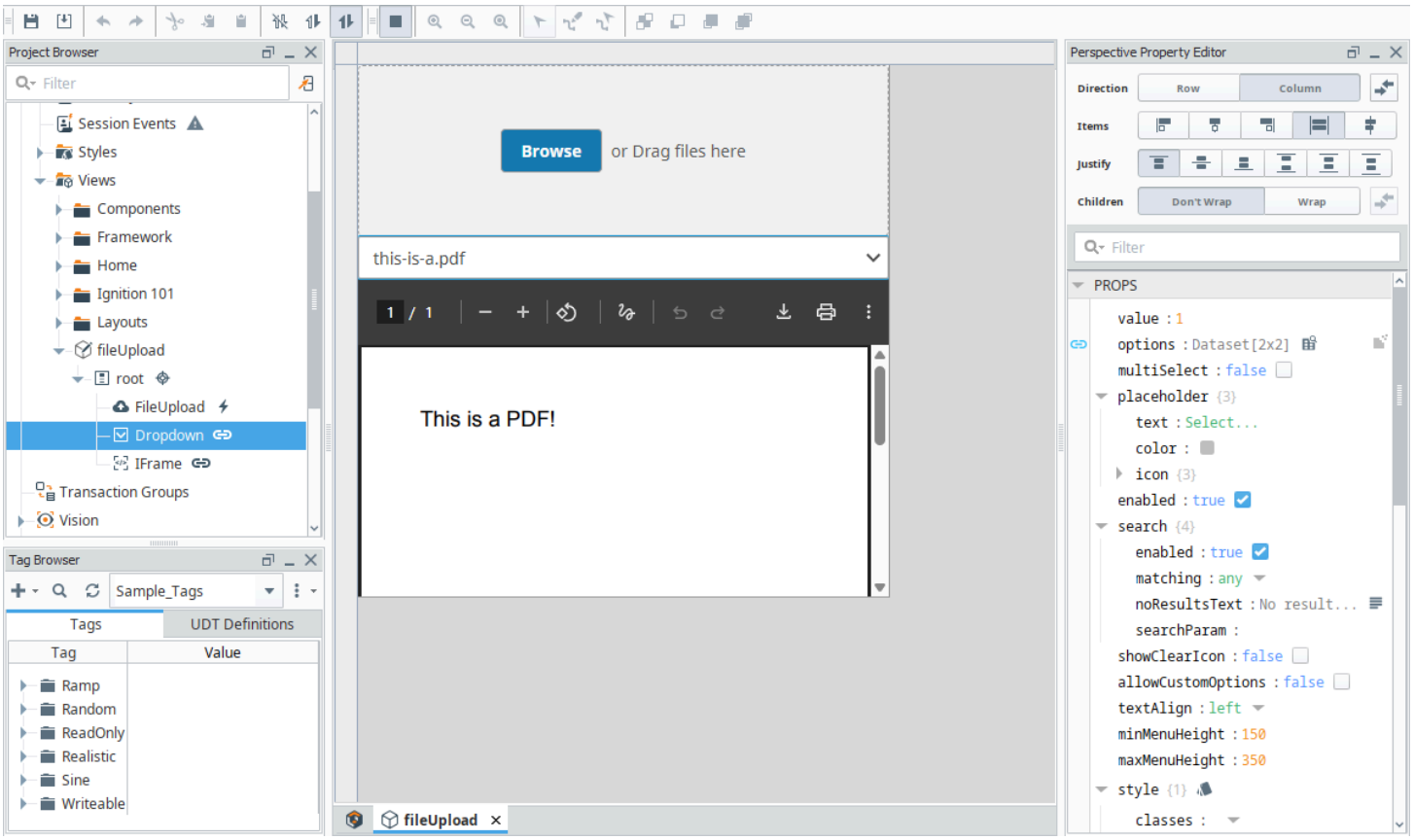
3. Add an **Inline Frame** component to the view. This component will be used to display the PDF after it is retrieved from the database and converted into a Data URL.

4. Bind the Iframe **src** property to the Named Query that returns the `fileName` and `fileBytes` columns.

5. Add a **script transform** to the binding that converts the returned file bytes into a Data URL.

```
import base64

# Expect a dataset with a "fileBytes" column
if value is None or value.getRowCount() == 0:
    return ""

fileBytes = value.getValueAt(0, "fileBytes")
encoded = base64.b64encode(fileBytes).decode("utf-8")

return "data:application/pdf; base64," + encoded
```

Now, when the user selects a file in the Dropdown, the Named Query returns the stored bytes. The transform converts the bytes into a Base64-encoded Data URL, and the Iframe renders the PDF in the browser.



Vision PDF File Example

One of the most common file types to store is a PDF. Storing each PDF in a central database allows any Vision Client to access it, instead of placing the file on a shared drive that all client machines can reach.

Uploading PDF Files in Vision

With a script on a Button component, you can store a PDF file in the database so that any user can view it later. This part does not use a Named Query, because Named Query Parameters do not have a data type that allows you to pass in raw file bytes. Instead, you can use **system.db.runPrepUpdate** to call a query from the script.

This example requires a table with a byte array column in it. For example, MySQL uses the BLOB data type and MSSQL uses the varbinary() data type.

> **DANGER**
>
> This function will not work with the default Client Permissions settings. Legacy Database Access must be enabled for this to work.

1. Add a Button component to a new Main Window.

2. Change the Text property to say **Add File**.

3. Right-click the Button and select **Scripting**.

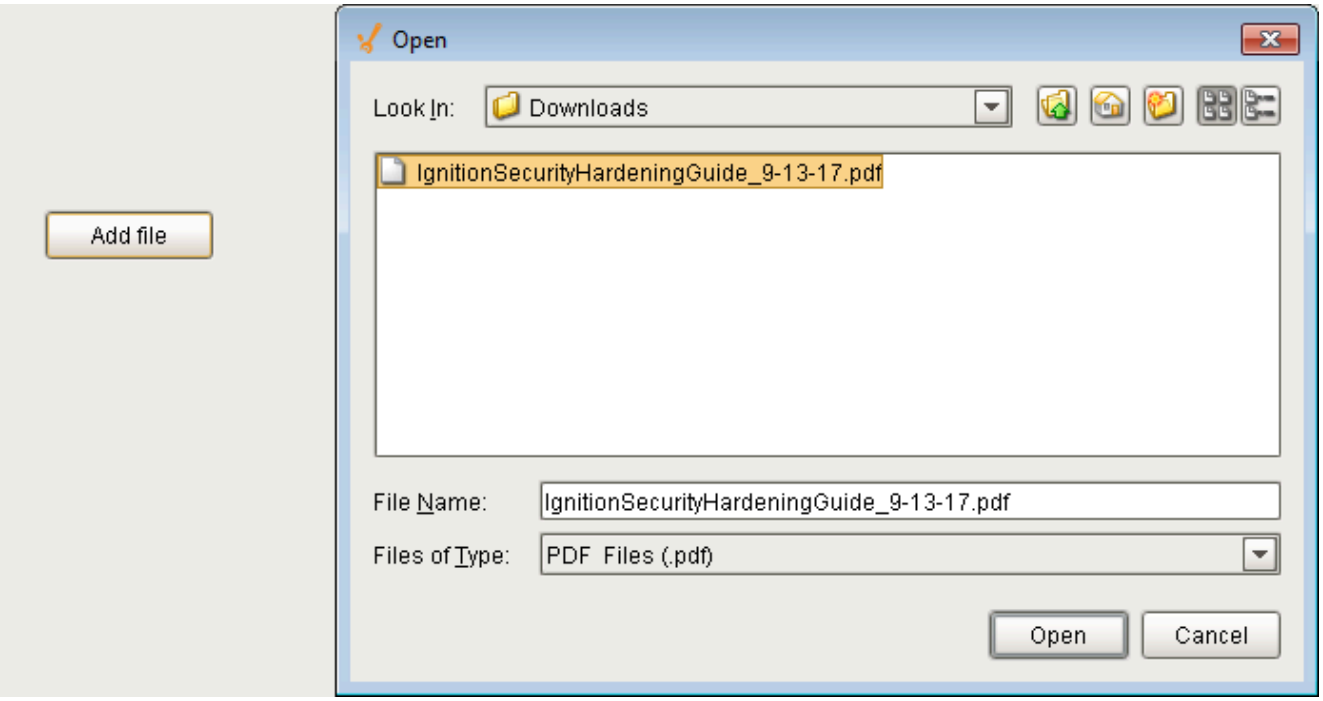
    i. Navigate to the Script Editor tab of the `actionPerformed` event handler. Here you can add a script that will grab the file bytes using the file path and the **system.file.readFileAsBytes** function, and then insert that into the database along with a user-selected file name.

    Python - Uploads PDF Files to a Database Using a Button

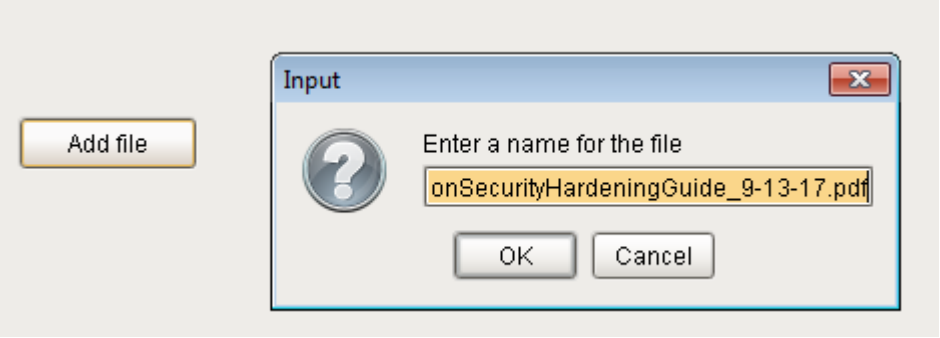
    ```
    # Find the path to the PDF file.
    path = system.file.openFile("pdf")

    # Check to ensure that the user actually selected a filepath.
    if path is not None:

        # Read the file as bytes.
        data = system.file.readFileAsBytes(path)

        # Ask the user to enter a filename.
        suggestedName = path.split('\\')[-1]
        name = system.vision.showInput("Enter a name for the file",
    suggestedName)

        # Check to ensure that the user entered a name for the file.
        if name is not None:
            # Insert the data and name into the database.
            system.db.runPrepUpdate(
                "INSERT INTO files (fileName, fileBytes) VALUES (?, ?)",
                [name, data]
            )
    ```

4. Test the script by putting the Designer into Preview Mode and clicking the Button.

    i. Select the file to load.



ii. Enter a file name.



Displaying PDF Files in Vision

Vision can render a PDF document inside the **PDF Viewer** component, which is a part of the **Reporting Module**. To view PDF files in a Client, your Ignition server must have the Reporting Module installed. Once the module is installed, you can load the bytes from the database into the PDF Viewer component.

1. This example assumes the `files` table described earlier in the section already exists and contains stored PDF records. Set up security to fit your needs, and name the Named Query appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.

    i. With no Parameters, add a query to select all the files in the `files` table.

    SQL - Selecting all Files

    ```
    SELECT id, fileName FROM files
    ```

2. Create a second Named Query, same as in step 1. This will be used to grab the file name and bytes after the user has chosen a file.

    i. Add a single Value type Parameter, `fileID`, using an Int4 data type.

    ii. Add the query to select the file name and bytes based on the selected ID.

    SQL - Selecting a File based on an ID

    ```
    SELECT fileName, fileBytes FROM files WHERE id = :fileID
    ```

3. On the window, add a Dropdown component and a PDF Viewer component.

4. On the Dropdown component, add a Named Query binding to the Data property.

    i. Set the binding to the Named Query created in step 1.

    ii. Place a small refresh Button component next to the Dropdown that will refresh this query. See **Refreshing a SQL Query** for more information on refreshing with a Button component.

5. Right-click the Dropdown component and select **Scripting**.

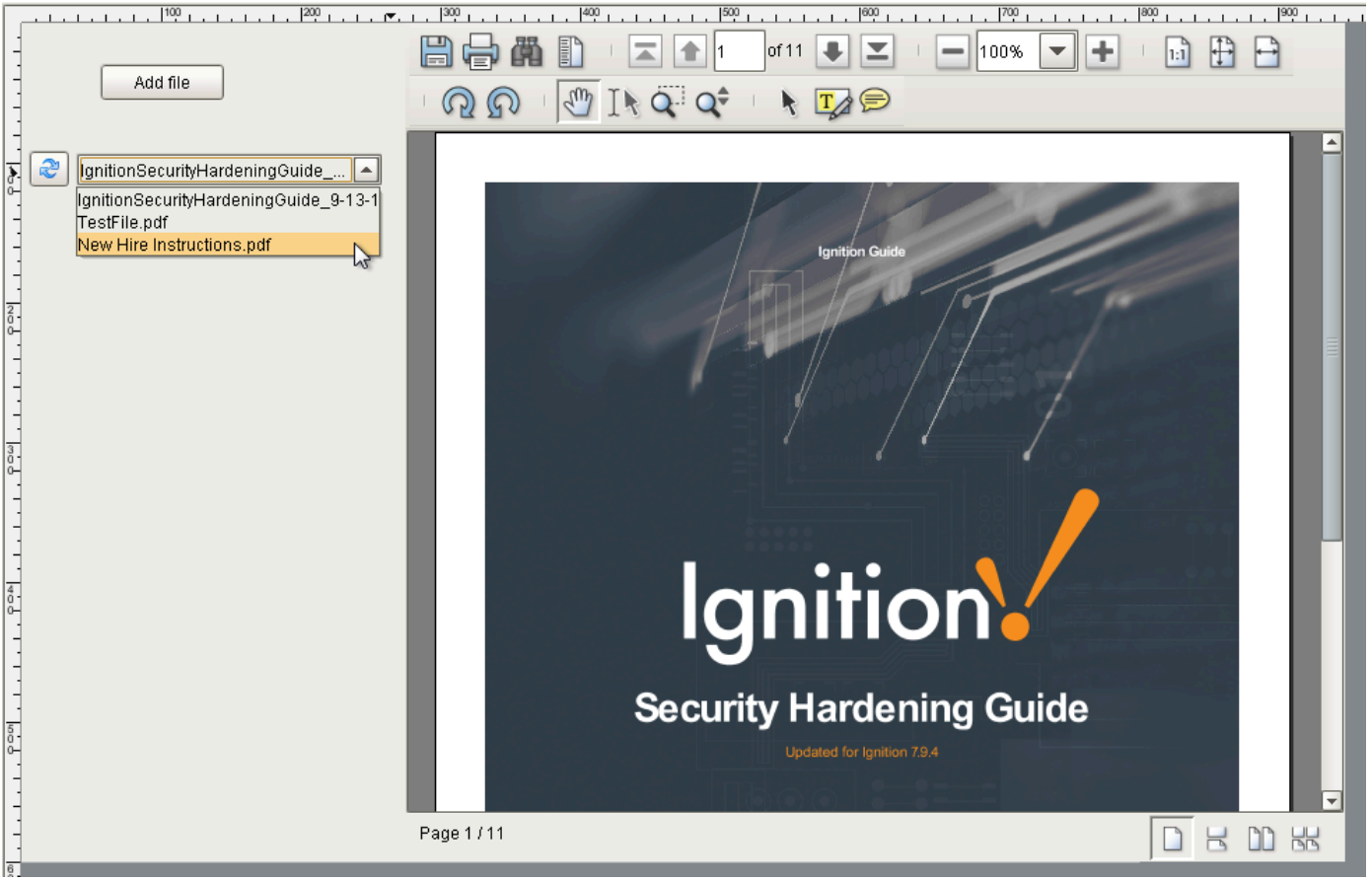
    i. Navigate to the Script Editor tab of the `propertyChange` event handler.

    ii. This script will take any new selected value and use it in the Named Query from step 2 to get the file name and bytes. It then loads the bytes into the PDF Viewer.

    Python - Displays PDF Files from a Database Using the PDF Viewer

    ```
    # Check to see if the property that changed was the Selected Value
    property.
    if event.propertyName == "selectedValue":

        # Run the query to grab the file name and bytes using the new selected
    ID value.
        data = system.db.runNamedQuery("Read File", {"fileID": event.newValue})

        # Grab the file bytes and name from the same row.
        bytes = data.getValueAt(0, "fileBytes")
        name = data.getValueAt(0, "fileName")

        # Load the bytes into the PDF Viewer component.
        event.source.parent.getComponent('PDF Viewer').loadPDFBytes(bytes,
    name)
    ```

6. Place the Designer into Preview Mode, and try selecting one of the stored files.



Simple Database Editor

Building a Database Table Editor

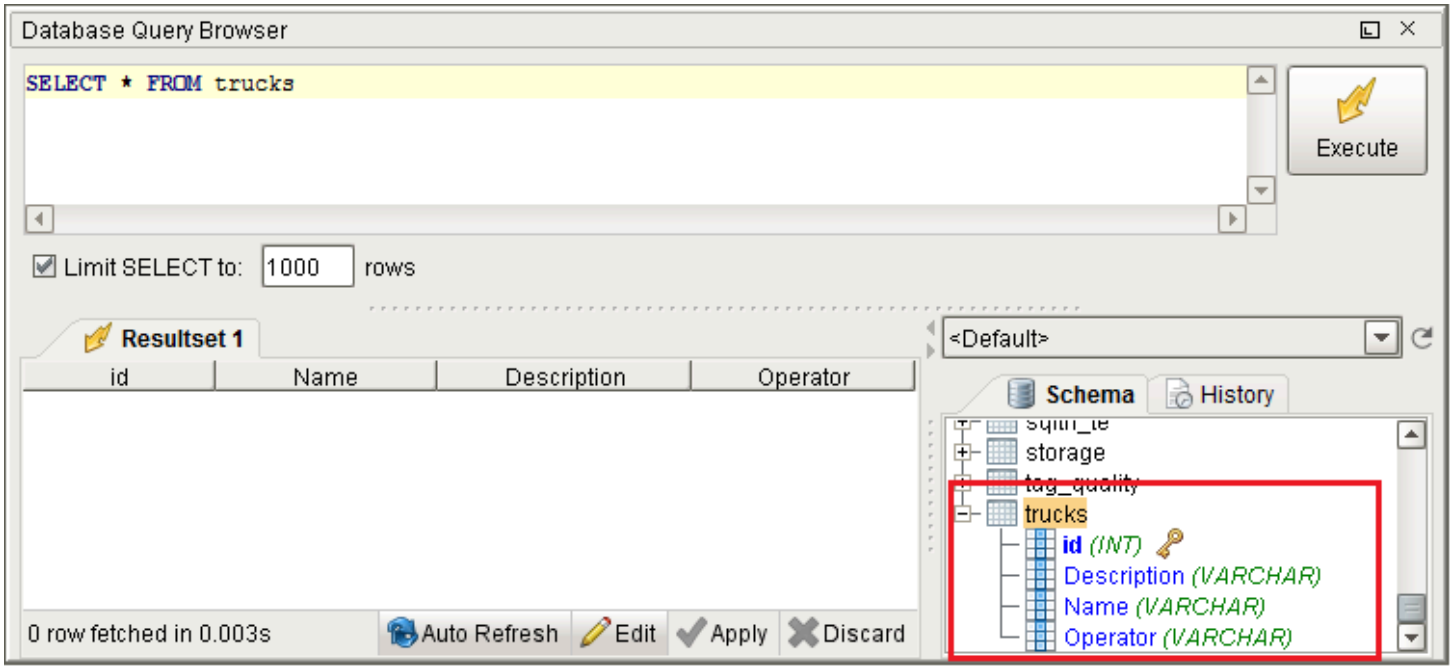
Building a Simple Database Table Editor in Ignition is actually quite easy. Using a few simple components and some scripting, we can easily make a window that will allow users to add, edit, and delete the data from a particular database table. Before we get started, there is a table that has already been created in a database that is used for this example. You can either create this table and follow along, or alter the example to fit your table.

The table is called **trucks**, and it has four columns.

id	Name	Description	Operator

- The **id** column is our primary key, it is non null, and it auto increments.
- All other columns are varchar(45)

That is it. There is no need to add any data, as we can add data when testing our tool.



Adding the Components

This example consists of a Table component that will display the data, Button components to add and delete rows from the database table, Text Fields where we can enter in new values, and an Update Details Button that will push those values to the database table.

Let's start off by going to the **Designer** and adding the necessary components to the window.

1. Add a **Power Table** component to the window.

    i. We want to query the table in the database that we made earlier, so let's add a **SQL Query Binding** on the **Data** property of the Table.

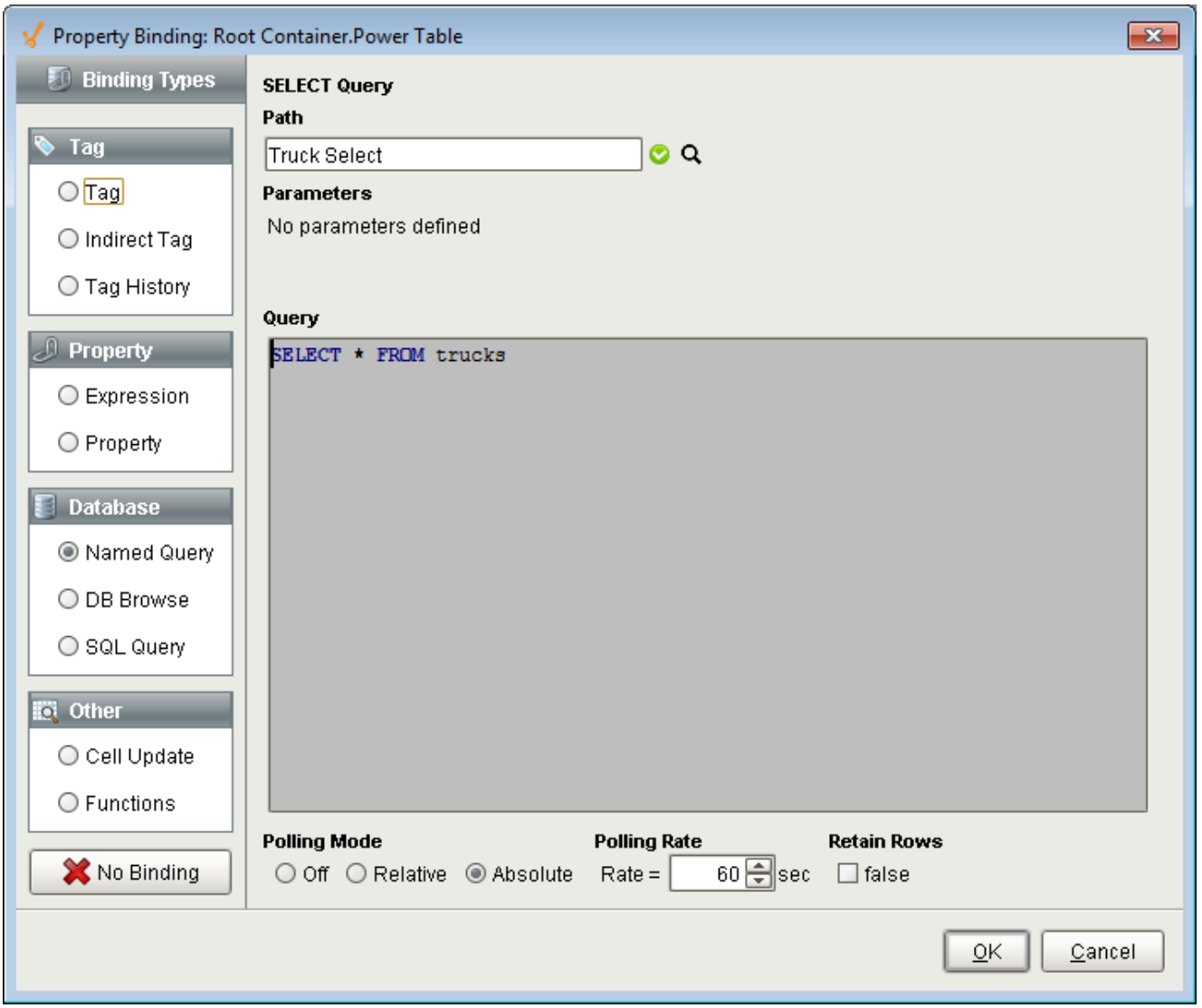
    ii. Add the following code under the **SELECT Query** area.

    SQL - Selects all Trucks

    ```
    SELECT * FROM trucks
    ```

    iii. Set the **Polling Mode** to **Absolute** and the **Polling Rate** to **60 seconds**, so that it periodically updates if left open.

    iv. Since we want to use the secure Named Query system, click the convert to Named Query button under the polling mode, and call the query "Truck Select"



v. With the Table still selected, open the **Table Customizer** by right clicking on the Power Table and going to **Customizers > Table Customizer**.

vi. Set the **Hide** property of the **id** column to **true**. We don't need to show that column to our users. Click **OK** to close the Table Customizer.

vii. In the **Property Editor**, make sure the **Selection Mode** property of the Power Table is set to **Multiple Interval**.

2. Add three **Button** components to the right of the **Table**.

i. Change the **Text** property on one Button to say "**Add**."

ii. Change the **Text** property of the second Button to say "**Delete**".

iii. Change the **Text** property of the final Button to say "**Update Details**"

iv. Optionally, you can also add an image to the **Image Path** of each button such as a green plus and a red X, and resize them as necessary.

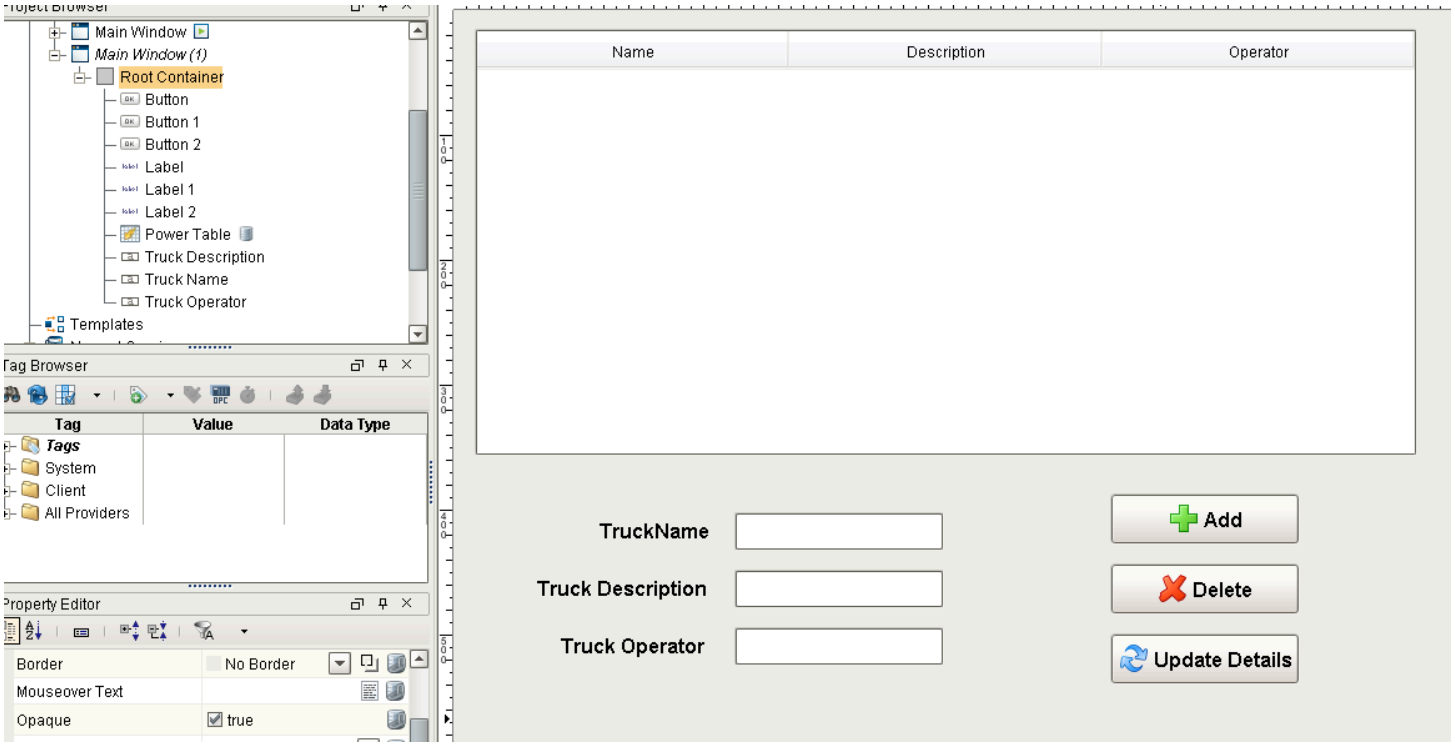
3. Add three **Label** components and three **Text Field** components under the **Table**.

i. Pair up a Label with each Text Field.

ii. Rename the Text Fields: **Truck Name, Truck Description, Truck Operator**.

iii. Change the **Text** property of each Label to match the name of their corresponding Text Fields.

Now we have our window configured with all of the components we need. Next we can start adding scripts that will alter the components.



Add a Truck

First, we can add a script to our **Add** button that will allow us to add a row to the Table. We can use the Text Fields that we added so the user can enter in values to insert into the table.

1. Create a new Named Query which we will use to insert new data. Set up **security** to fit your needs and name it appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.

    i. Set the Query Type to Update Query.

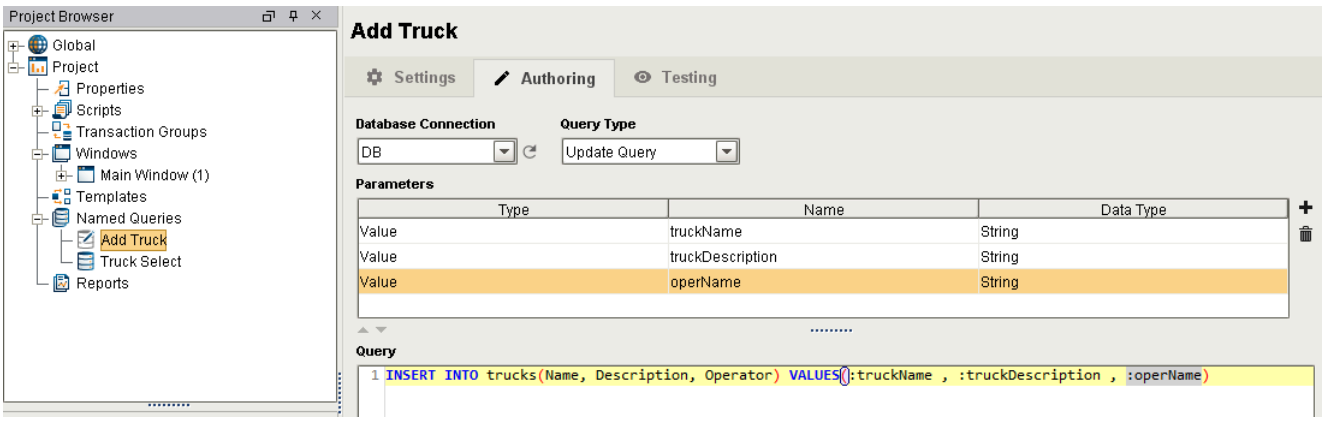
    ii. Create three Value type Parameters.

        a. The first will be a string data type and will be used for the name of the truck so it can be called "truckName".
        b. The second will be a string data type and will be used for the description of the truck so it can be called "truckDescription".
        c. The third will also be a string data type and will be used for the operators name so it can be called "operName".

    iii. Create the insert query.

SQL - Adding a Truck

```
INSERT INTO trucks(Name, Description, Operator) VALUES(:truckName ,
:truckDescription , :operName)
```



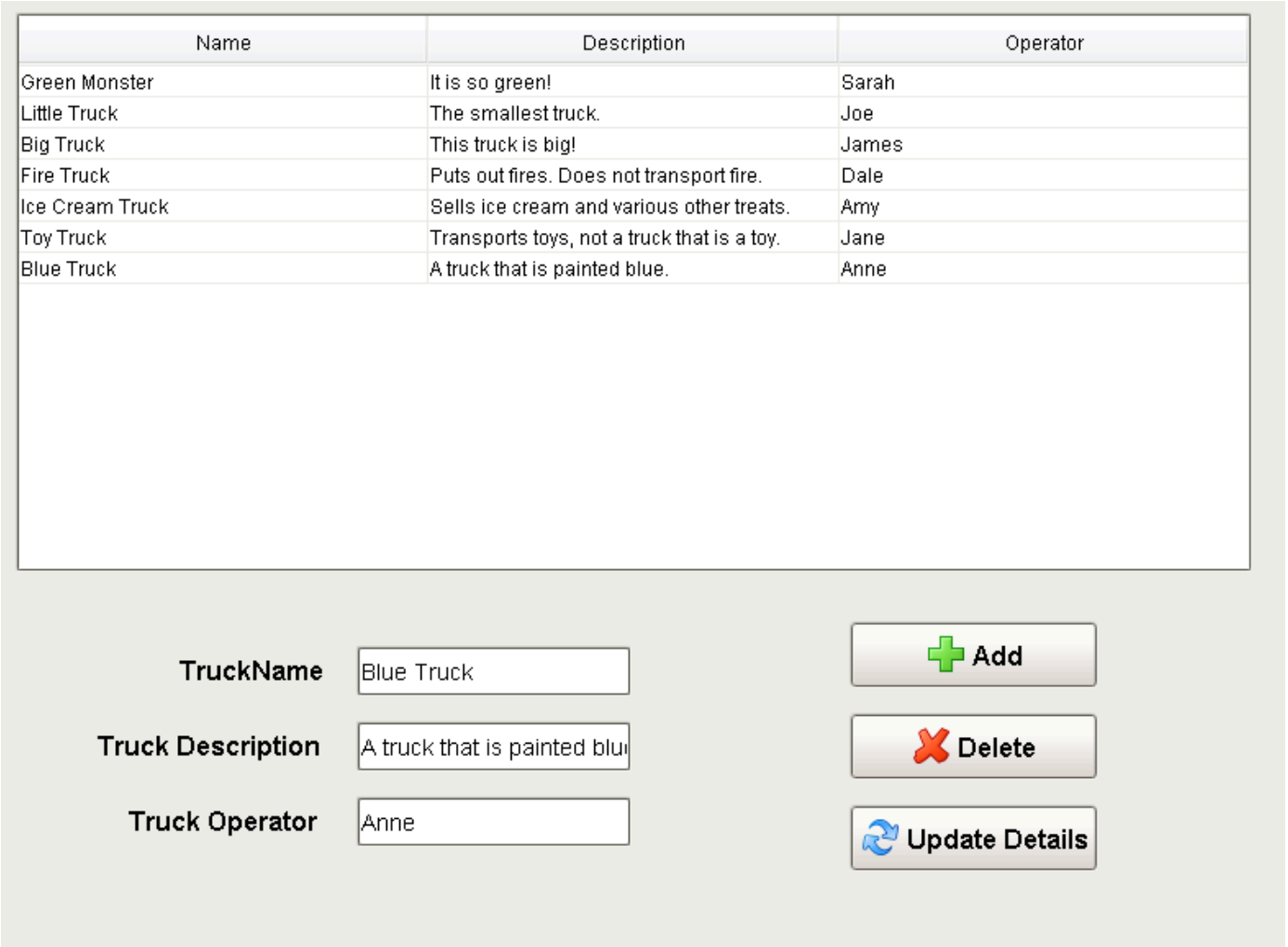
2. Back on our window with components, right click on the **Add Button** and select Scripting. Select the **actionPerformed** Event Handler and navigate to the **Script Editor** tab.

    i. Here we can add some code that will pull the values from the three Text Field components and then use those values in the Named Query we made in step 1.

Python - Adds a New Blank Row and Refreshes the Table

```
# Grab the values from the Text Fields.
name = event.source.parent.getComponent('Truck Name').text
description = event.source.parent.getComponent('Truck Description').text
operator = event.source.parent.getComponent('Truck Operator').text

# Use those values in the Add Truck Named Query.
system.db.runNamedQuery("Add Truck", {"truckName":name,
"truckDescription":description, "operName":operator})

# Refresh the table to immediately bring in the new row.
system.db.refresh(event.source.parent.getComponent("Power Table"), "data")
```

3. Now we can test out our script and query by putting the Designer into **Preview Mode**, adding some data to the **Text Fields**, and clicking the **Add Button**. You should see a row populate the table. Try adding a few rows to fill our table with data. Make sure you have the designer Communication Mode set to Read/Write to test. Go back to **Design Mode** before moving on.



Delete a Truck

Now that we can add rows, let's add a script to our **Delete button** that will delete rows. Since our users can select multiple rows, our script needs to take that into account, and delete all of the rows that the user selected. It will also have a **Message Box** popup if there are no rows selected, informing the user that they need to select at least one row. We also want to add a confirmation before the rows actually get deleted.

1. Create a new Named Query which we will use to insert new data. Set up **security** to fit your needs and name it appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.

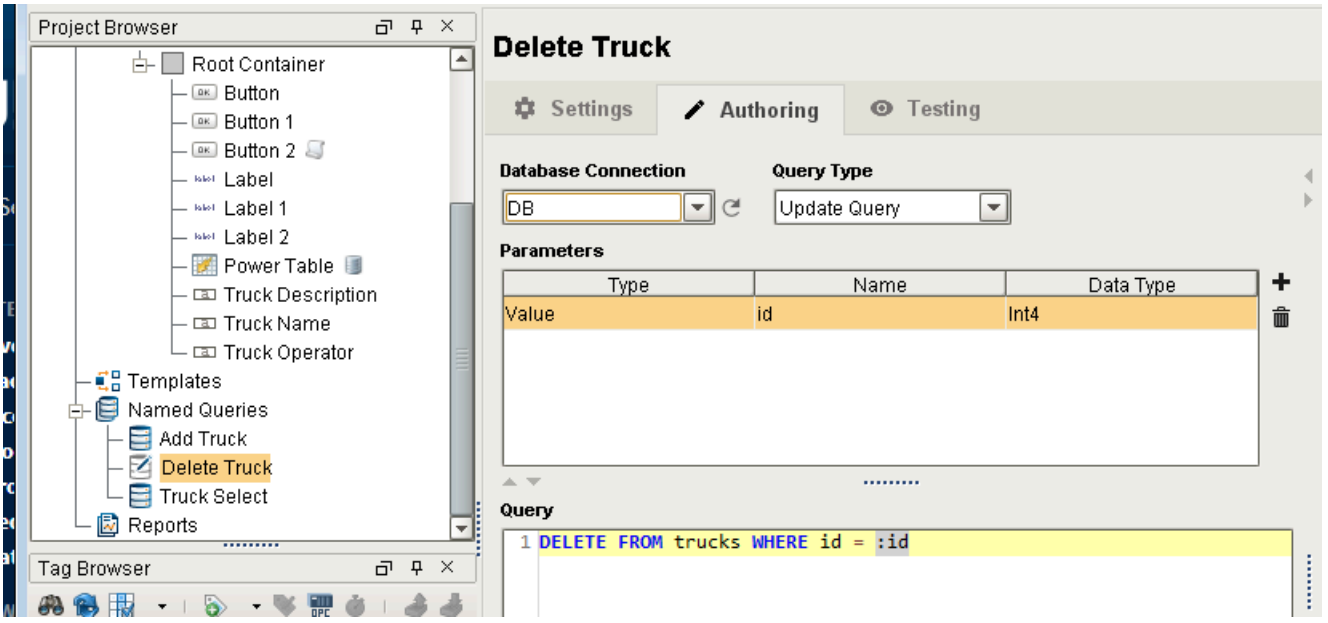
    i. Set the Query Type to Update Query.

    ii. Create one Value type Parameters.

        a. It will be a Int4 data type and will be used for the id of row so it can be called "id".

    iii. Create the delete query.

```
SQL - Deleting a Truck

DELETE FROM trucks WHERE id = :id
```



2. Back on our window with components, right click on the **Delete Button** and select Scripting. Navigate to the **Script Editor** tab on the **actionPerformed** Event Handler.

    i. Here we can add some code that will delete the user's selected rows. First we grab the Power Table's selected rows, and check to make sure rows are actually selected. If there are

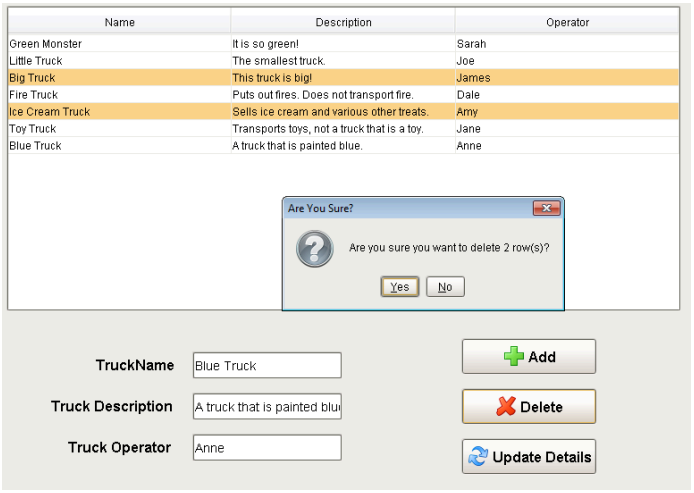
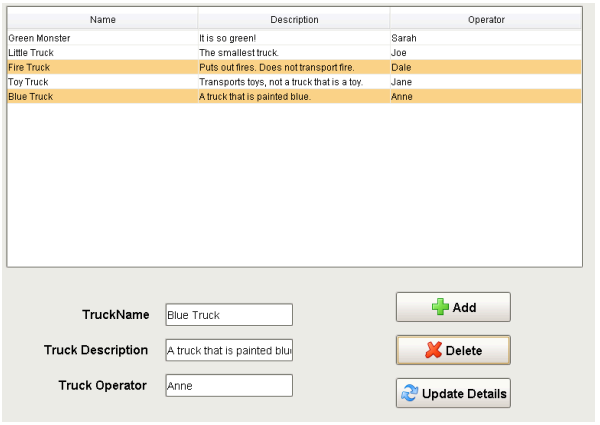
selected rows, then we have a popup that confirms the user wants to delete rows, and it shows the number of rows that will be deleted. Once the user confirms, the script will then loop through the selected rows and delete each one.

Python - Deletes the Row or Rows

```
# Get the selected row or rows.
selRows = event.source.parent.getComponent('Power Table').getSelectedRows()

# Check to see that a row is actually selected.
if len(selRows) > 0:

    # If a row is selected, ask for confirmation before deleting the rows.
    if system.vision.showConfirm("Are you sure you want to delete " +
str(len(selRows)) + " row(s)?", "Are You Sure?", 0):

        # If the user gave confirmation, we need to loop through all of
them.
        for row in selRows:
            # For each selected row, we need to get the value of the id
column in that row.
            rowId = event.source.parent.getComponent('Power
Table').data.getValueAt(row, "id")

            # Use the id of the row to delete it from the database table.
            system.db.runNamedQuery("Delete Truck", {"id": rowId})

        # Refresh the table after deleting all selected rows
        # to immediately remove the selected rows from the Power Table.

system.vision.refreshBinding(event.source.parent.getComponent('Power
Table'), "data")

    # If the user said no to the delete.
    else:
        system.vision.showMessage("User canceled the delete.", "Delete
Canceled")

# If no row is selected, have a popup message that asks the user to select
a row.
else:
    system.vision.showMessage("Please select at least one row.", "Select A
Row")
```

3. Now we can test our **Delete button** by putting the Designer into **Preview Mode** and clicking the **Delete button**. When one or more rows are selected and the **Delete button** is pressed, a popup

will confirm we want to delete the rows before deleting them. (You can select multiple rows with **Shift-Click** or **Control-Click**). When no rows are selected and the **Delete button** is pressed, a **Message Box** will pop up, informing the user to select a row first. You can deselect rows in the Table by clicking on the Table and pressing the **Escape** key. Be sure to go back to Design Mode before moving on.

Confirming Row Selection to Delete	Row Deletion Results
	

Populate Text Fields

Now that we can add and delete rows from our Table, we need a way to edit the data contained in them. However, before we can create a script on our **Update Details button**, we first need to populate the Text Fields with the current values of the selected row. We can do this by adding a simple expression to each of the text fields.

1. On the Truck Name Text Field, add an expression binding to the Text property.

    i. The expression will grab the dataset and will grab the value at the selected row of the Name column. The expression is wrapped in a try so that if no rows are selected, it will return an empty string.

    Expression - Grabbing the Selected Row's Name Value

    ```
    try({Root Container.Power Table.data}[{Root Container.Power Table.selectedRow}, "Name"], "")
    ```

2. On the Truck Description Text Field, add an expression binding to the Text property.

    i. The expression will grab the dataset and will grab the value at the selected row of the Description column. The expression is wrapped in a try so that if no rows are selected, it will

return an empty string.

Expression - Grabbing the Selected Row's Description Value

```
try({Root Container.Power Table.data}[{Root Container.Power
Table.selectedRow}, "Description"], "")
```

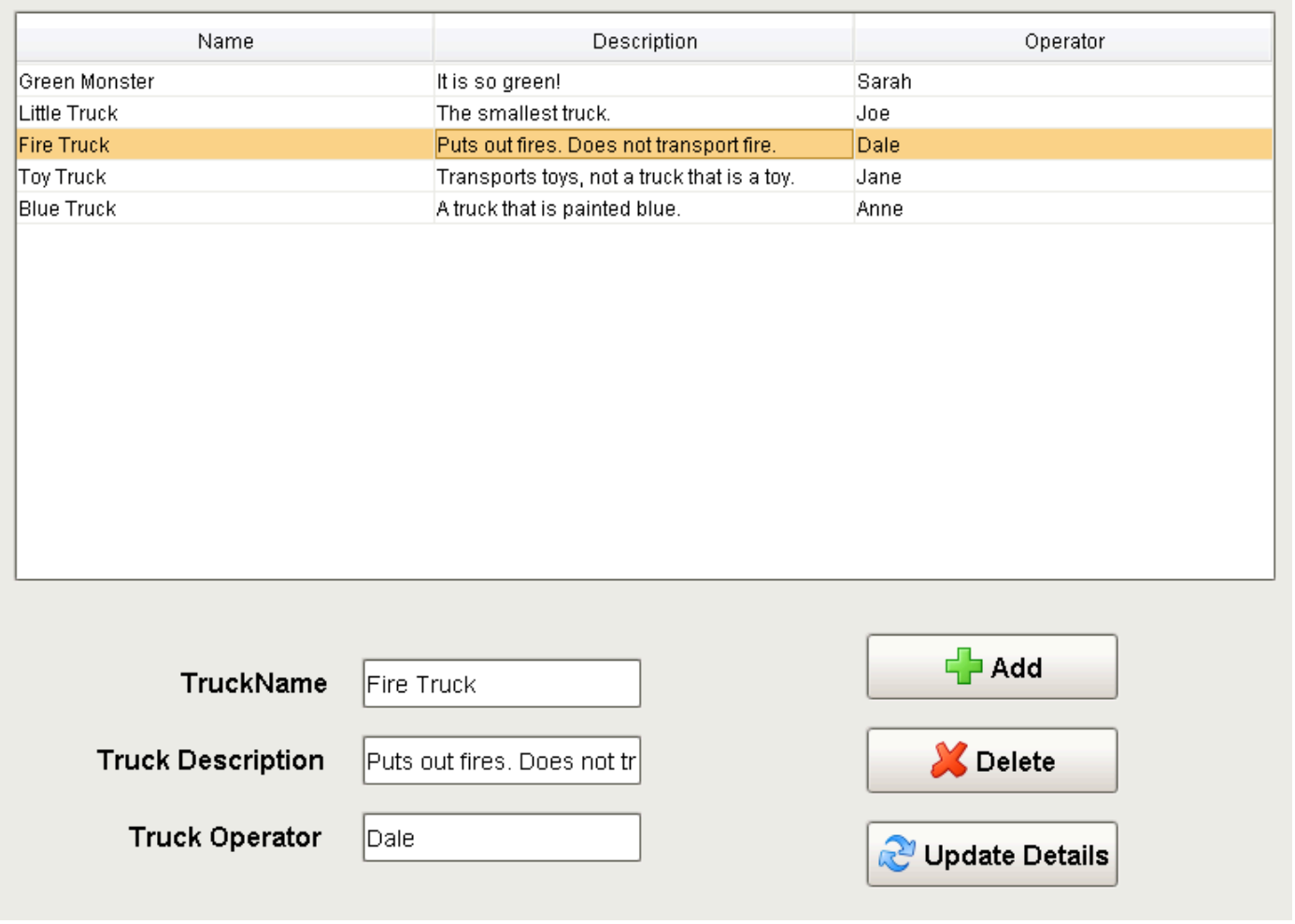
3. On the Truck Operator Text Field, add an expression binding to the Text property.

    i. The expression will grab the dataset and will grab the value at the selected row of the Operator column. The expression is wrapped in a try so that if no rows are selected, it will return an empty string.

Expression - Grabbing the Selected Row's Operator Value

```
try({Root Container.Power Table.data}[{Root Container.Power
Table.selectedRow}, "Operator"], "")
```

4. Click on a row to see the fields below populate. Press the ESC key to empty the selection.



Update Data

The last part of setting up this window is to add a script to the **Update Details button** that will pull the data from the Text Fields and insert it into the selected row of the database Table. Additionally, the script will check to ensure that a row selected, and that only one row is selected. We wouldn't want to confuse users who may have accidentally selected multiple rows at a time, since our button will only update a single row.

1. Create a new Named Query which we will use to update a row of data. Set up **security** to fit your needs and name it appropriately. For more information on creating Named Queries, see **Using Named Queries - Example**.

    i. Set the Query Type to Update Query.

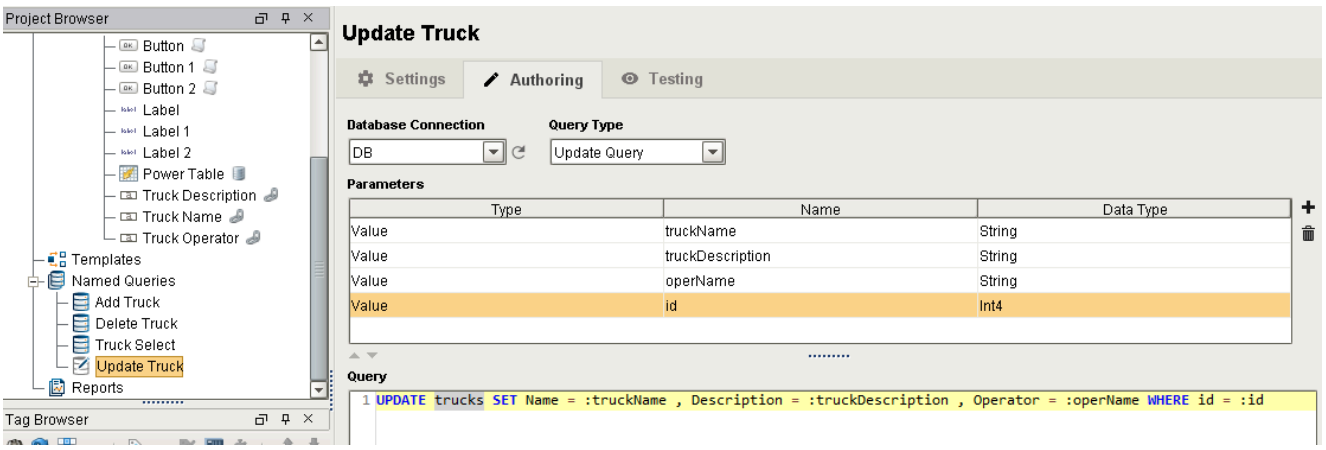
    ii. Create four Value type Parameters.

        a. The first will be a string data type and will be used for the truck name of row so it can be called "truckName".
        b. The second will be a string data type and will be used for the truck description of row so it can be called "truckDescription".
        c. The third will be a string data type and will be used for the operator name of row so it can be called "operName".
        d. The fourth will be a Int4 data type and will be used for the id of row so it can be called "id".

    iii. Create the update query.

SQL - Deleting a Truck

```
UPDATE trucks SET Name = :truckName , Description = :truckDescription ,
Operator = :operName WHERE id = :id
```



2. Back on our window with components, right click on the **Update Button** and select Scripting. Navigate to the **Script Editor** tab on the **actionPerformed** Event Handler.

    i. Here we can add some code that will update the users selected row with data pulled from the Text Fields. First we check to ensure that only one row is selected. Next the script asks them to confirm that the user wants to update the selected row. If the user confirms, the script then pulls in the values of all three Text Fields and uses those values along with the selected rows id to update the database table with new data. It will then refresh the table after updating so that the new data can be brought into the table.

    Python - Deletes the Row or Rows

```
# Ensure exactly one row is selected.
table = event.source.parent.getComponent('Power Table')
selRows = table.getSelectedRows()

if len(selRows) == 1:

    # Confirm before updating.
    if system.vision.showConfirm("Are you sure you want to update the
selected row?", "Are You Sure?", 0):

        # Grab values from input fields.
        name = event.source.parent.getComponent('Truck Name').text
        description = event.source.parent.getComponent('Truck
Description').text
        operator = event.source.parent.getComponent('Truck Operator').text

        # Select row index and row ID.
        selRow = table.selectedRow
        rowId = table.data.getValueAt(selRow, "id")

        # Update via Named Query.
        system.db.runNamedQuery(
            "Update Truck",
            {"truckName": name, "truckDescription": description,
"operName": operator, "id": rowId}
        )

        # Refresh the table binding to show the updated data.
        system.vision.refreshBinding(table, "data")

    else:
        system.vision.showMessage("User canceled the update.", "Update
Canceled")

else:
    # More than one or no rows selected.
    system.vision.showMessage("Please select one row to edit.", "Select One
Row")
```

3. Now that we added this last bit of code, we can test it by selecting a row, adding data to each **Text Field**, and clicking the **Update Details button**. You should see the data populate the row of the Table, since the table in the database has been updated. Additionally, you should notice how the Text Fields will automatically update when selecting a new row. Try adding a few rows of valid data to your Table.

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Fire Truck	Puts out fires. Does not transport fire.	Dale
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Yellow Truck	A truck that is painted yellow.	Abby

TruckName Yellow Truck

Truck Description A truck that is painted yell

Truck Operator Abby

Add

Delete

Update Details

Basic SQL Troubleshooting

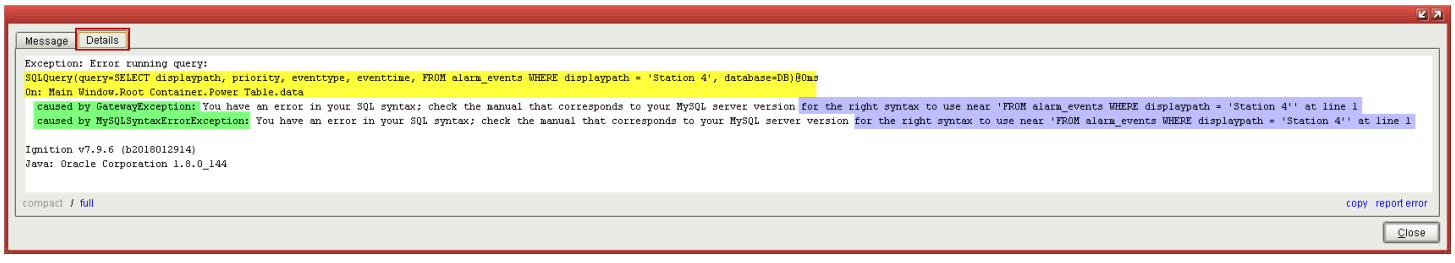
Learning when something is wrong with your database queries and learning how to resolve the problem is key to getting the best possible use out of the database connection to Ignition. Typically, the biggest problem that users face with queries is the data taking too long to load on the window, but an error in a query can also be difficult to track down. This section details what to look for when the query fails to execute, as well as what to do when facing slow queries and how to optimize them within the project to help return data to the window as quickly as possible.

Error Message Box

If a query fails to execute, chances are there was an error box that popped up. Resist the urge to close it! The Error Box usually contains a lot of useful info that can help troubleshoot why the query is failing. When looking at the error message, there are a few things to look for. First, you will want to click on the **Details** tab in the upper left corner of the Error Box. This will bring up the details of the error, which contains the information we need to track down the error.

In the image below, you can see the yellow highlighted part shows us what our query is, as well as where it is executing from. This helps to ensure we are looking at the correct error message for the query we are trying to fix. Next, you want to look for the words '**caused by**' in the text of the error details, which tell us the reason for the error. You can see in the image below, there are two of '**caused by**' messages highlighted in green.The first one is from the Gateway, and the second one is from the database (in this case, a MySQL database).

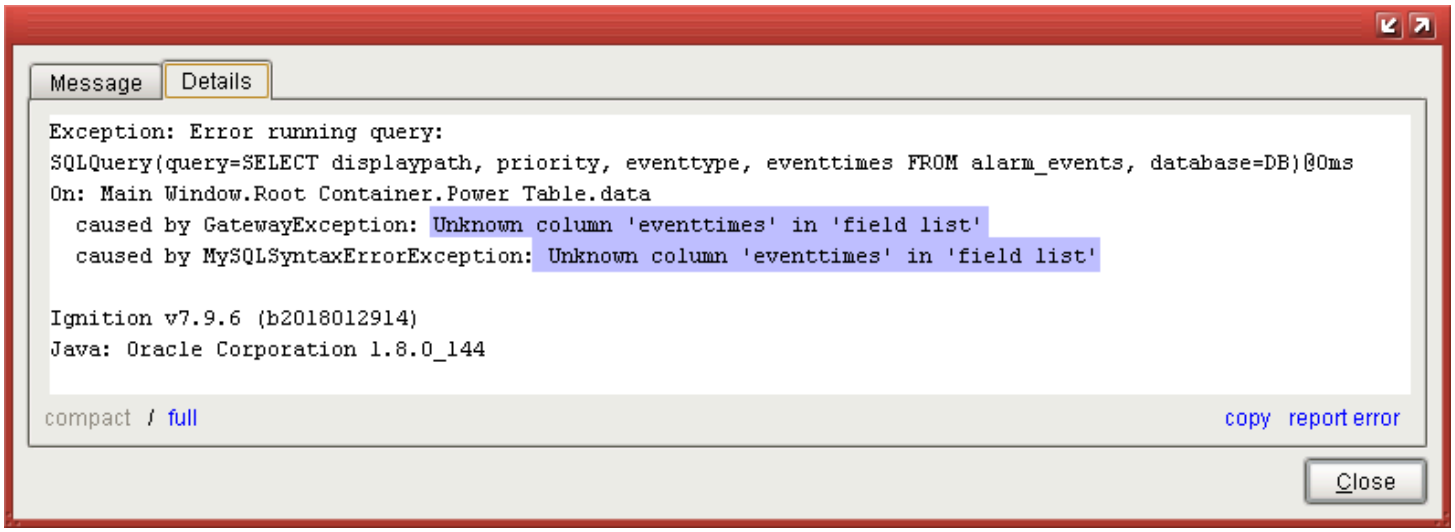
After locating the '**caused by**', the following message will help to pinpoint what the error is. In this example, we have an error with our syntax in our SQL statement. The most helpful portion of the error is at the end of the message highlighted in purple. This lets us know where the syntax error is in the SQL statement. The trick is to look immediately before the quoted query. In the image below, it starts to quote the original query with '**FROM alarm_events WHERE.....**', so in my original query, I need to look at what was right before the '**FROM**'. You can see in my original query highlighted in yellow, there is a comma right before the '**FROM**', which is incorrect, as the last column in the **SELECT** statement should not have a comma after it.



Other Common SQL Errors

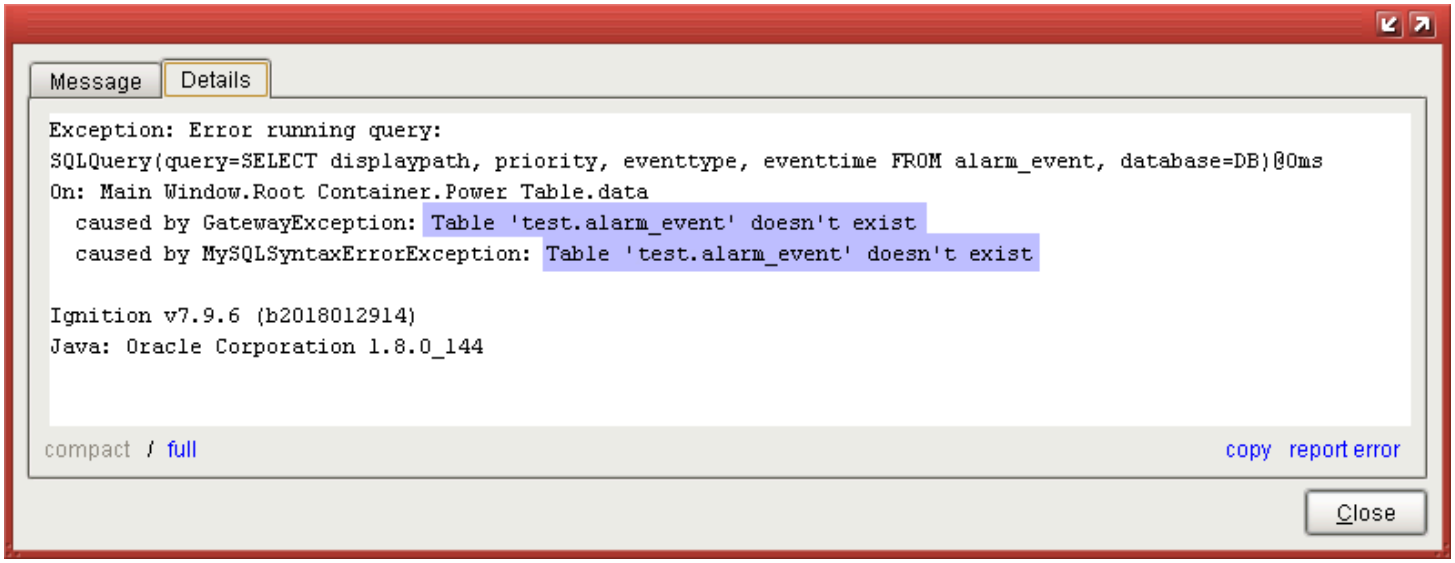
Unknown Column

Typically, the name of the column is wrong because of a misspelling. In this case, the correct spelling was 'eventtime', but there was an accidental 's' added to the name.



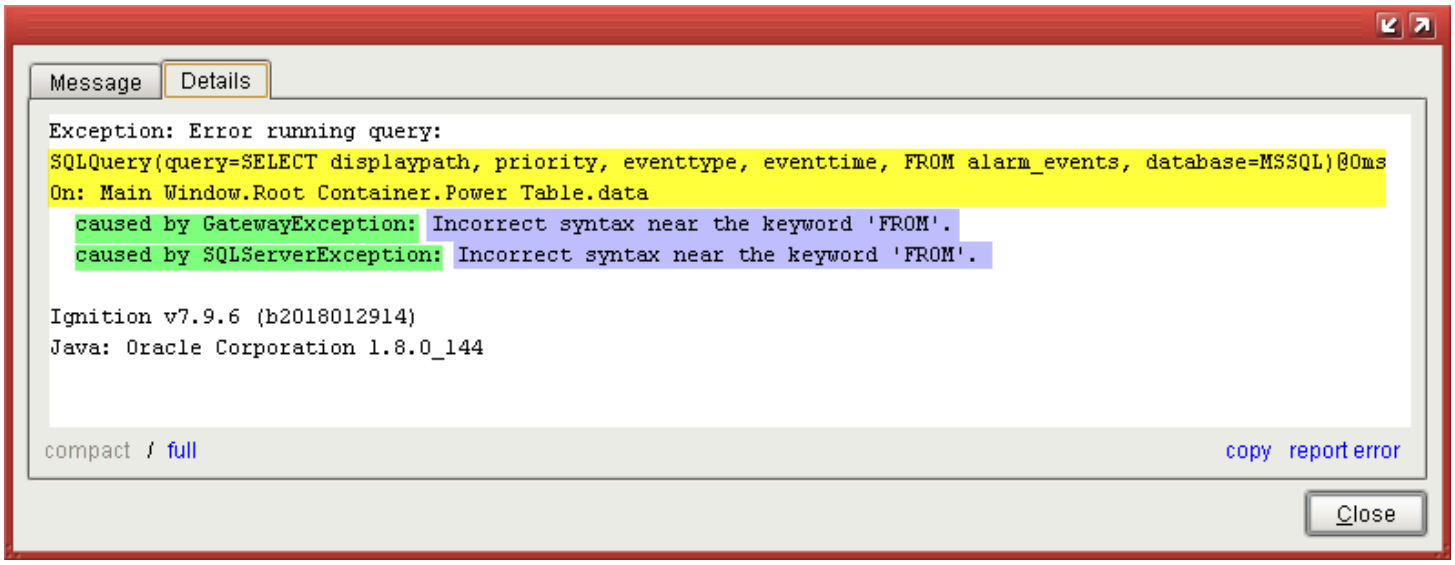
Unknown Table

If the table is not found in the database, it may also be because of a misspelling. Here, the correct table name should be 'alarm_events'.



Any Database Error

The previous error examples have all been from a MySQL database, but the same principles apply to any database. Simply locate the 'caused by', and look at the message afterwards. In this MSSQL syntax error, instead of displaying everything after the syntax error, it only displays the part immediately after. The problem in this query is an extra comma before the word 'FROM'.



Checking the Database Connection

Checking the Database Connection in the Gateway can also be useful to ensure there is a valid connection and there is nothing blocking the execution of queries. The **Connections** > **Databases** > **Connections** page of the Gateway can be used to determine if queries are taking too long, and alter the settings of the database to better handle the number of queries running from the Gateway to the database.

Testing Query Results

Using the **Database Query Browser** is a good way to test out a query before actually running it on a component. This can help you see what results will be returned so that the query can be modified to narrow down how the query should be formatted.

Checking for Slow Queries

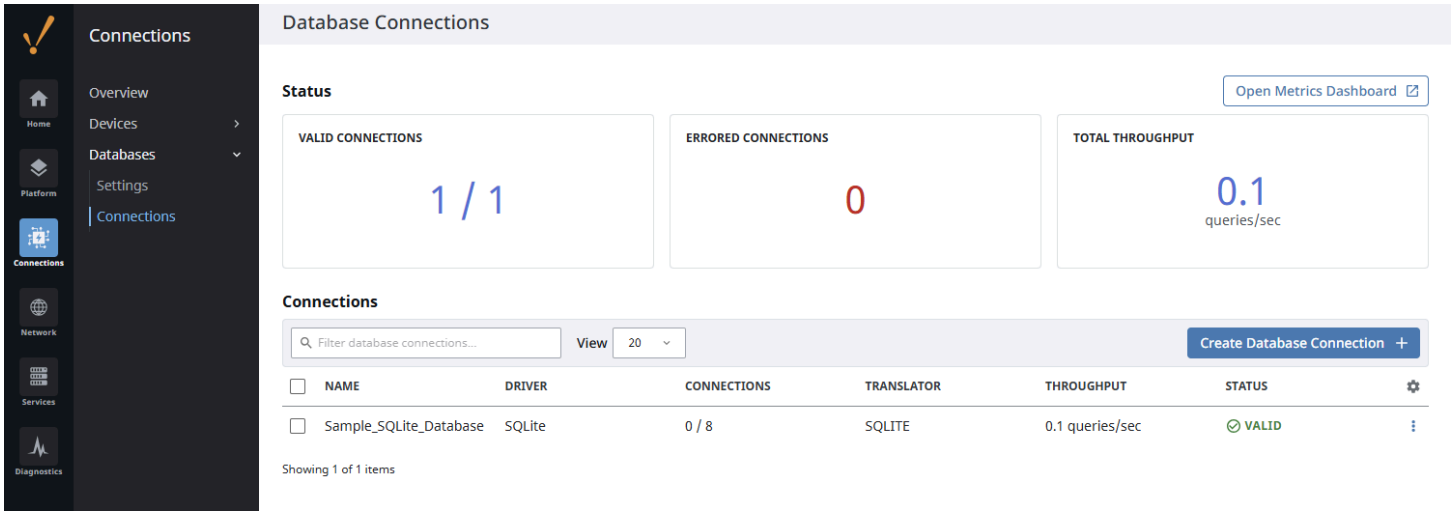
There are several ways that slow running queries can cause improper behavior. See **Slow Queries** for more details.

Slow Queries

Slow running queries can be a big problem. Not only can data take a long time to display on the screen, but it can end up slowing down the whole client or session. This page covers checks and adjustments you can make when your project has a slow running query.

Identify the Slow Query

The first step in dealing with slow queries is identifying which query is actually running slowly. A good indication of the query location is when a window is opened and a component takes time before database data is displayed. Chances are the query is on that component somewhere. However, it may also be a good idea to check the queries that are being run against that database connection in the Gateway. From the Gateway, navigate to **Connections** > **Databases** > **Connections** to see a list of all database connections. From here, you can get a good idea of any queries that may be a little slower than the others.



Execute Against the Database Management Software

The next step in identifying the type of slow query we are dealing with is to run the query directly within the database management software. By cutting out Ignition, we can determine if the query is actually running slowly, or if there is a problem within Ignition that is making the query run slow.

Fast Query in the Database Management Software

If the query runs quickly in the database management software, you can complete the checks described below to try and identify an issue within Ignition.

Check the Database Connection

First, check to make sure the database connection is valid, and there are no warnings associated with it. You may want to go into the connection settings for that database and take note of the value of the **Max Active** property, which determines the maximum number of active connections to the database, as well as the **Max Wait** property, which is the number of milliseconds to wait for a connection to come available. To get to the database connections settings, navigate to **Connections** > **Databases** > **Connections**, locate your database, and expand the three dots menu to select **Edit**. Here, you can check the database settings. Open **Advanced Properties** at the bottom of the panel, and you'll find the **Max Active** property as well as the **Max Wait** property.

Check Currently Executing Queries

On the **Database Connections** page, we can see a list of currently running queries, as well as how many of the active connections are being used. If the max number of connections is being used, it may be that there are so many queries running that each query needs to wait for an active connection to open up. If this is the case, you may want to increase the amount of active connections to the database or take steps to **reduce the query load on the database**.

Slow Query in the Database Management Software

If the query also runs slow within the database management software, then the query is just a slow query. Unfortunately, nothing within Ignition can speed up the execution of that query, so you would want to instead take a look at what the query is doing. If the query is pulling in lots of data, you can try breaking the query down into smaller queries, or writing the query in a more efficient way.

For particularly large tables, it may also be helpful to add an index to one of the table's columns. Indexes are something that the user can't see, but help the database speed up data retrieval. However, adding an index to a table will increase the amount of time that an update to the table takes, because the index also needs to be updated. For this reason, it is recommended to only make indexes on columns that are frequently searched against.

To make an index, most database management software have built-in interfaces that allow you to customize the index on each table. An index can be made for a single column, or a combination of columns in the table. Talk to your Database Administrator about adding or updating table indexes.

Alternately, you may need to take a look at the database system as a whole. As the size of the database grows, you may need to update the hardware resources available to it. If the database is installed on a server with another system like Ignition, be aware that although the two systems are now sitting next to each other, they have to share the hardware resources available to them, which may cause issues for both systems. In many cases, it is often better to have the database run on a separate server, which gives it ample room to grow.

SQL Query Volume Optimization

Overview

Leveraging a SQL Database can drastically improve the quality of a project, but improper database consideration while designing can lead to poor performance later on. This page contains some best practices and considerations when incorporating SQL queries.

Optimizing Individual Queries

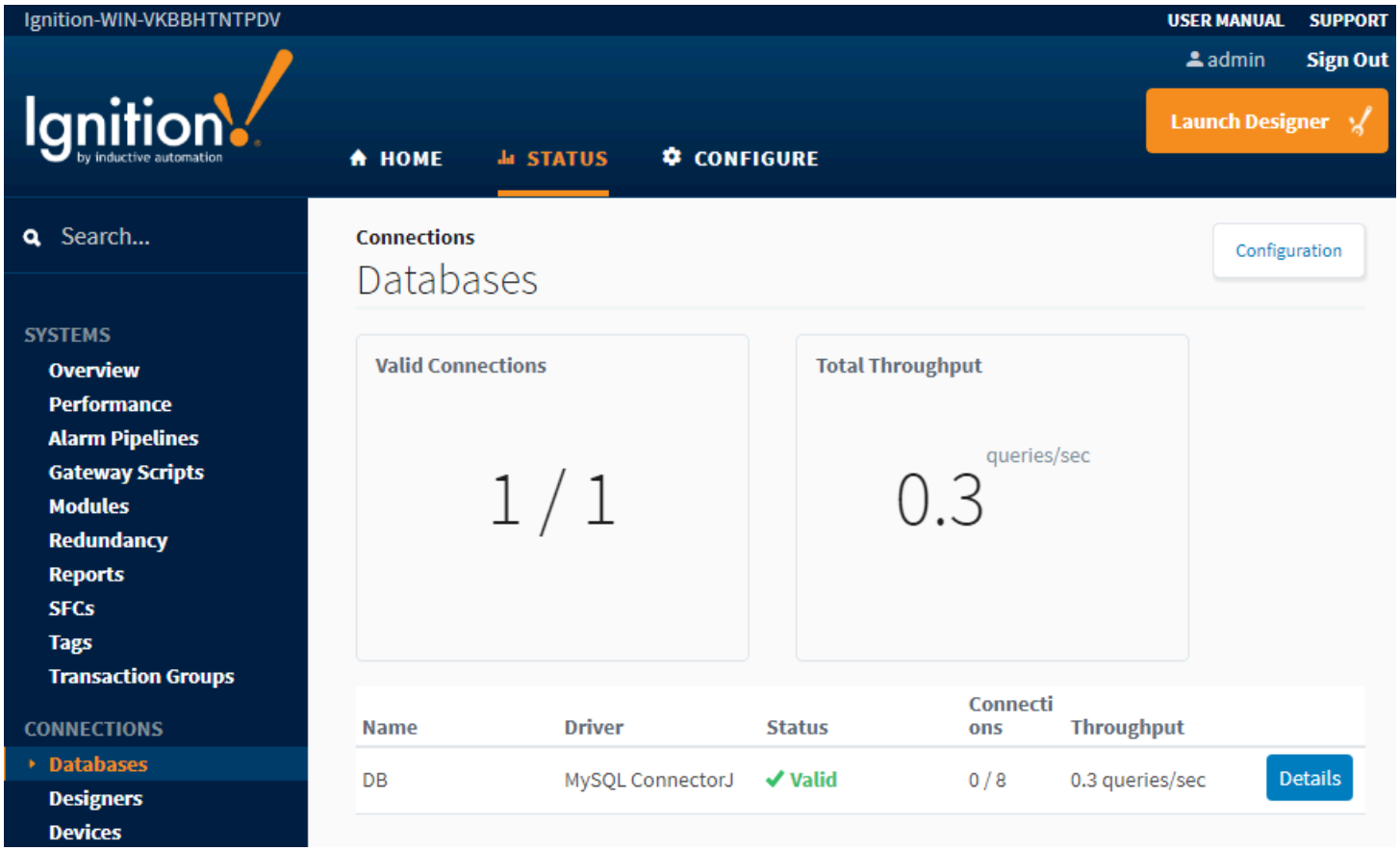
Optimizing individual queries to run faster and be more efficient is very difficult to do properly and can vary widely depending on a number of factors such as the way the tables are setup in the database and what data is being pulled out. Because each database can vary widely from another, there isn't any general way of improving the efficiency of your queries. Instead we recommend becoming more familiar with the SQL language as well as how the data is setup in your system and exactly what data you want to retrieve. This knowledge can help you build better queries.

Your company may also have a database administrator who would manage the database system for your company and would be familiar with figuring out the best way of retrieving data. They may be able to help you retrieve the data that you need.

Always Think Large

Consider how many instances of a query may be running at any given point in time. A single **SQL Query Binding** will be called for each instance of the window that is open, so if 50 clients are all looking at the same window, 50 separate queries will be called. If the binding is configured to poll, then 50 queries will poll at the rate specified for this single binding. This is already a fair amount of work without factoring in other systems, such as Tag Historian.

To provide context, you can always check the **Connections > Databases** section of the Gateway Webpage to see current throughput of each database connection.

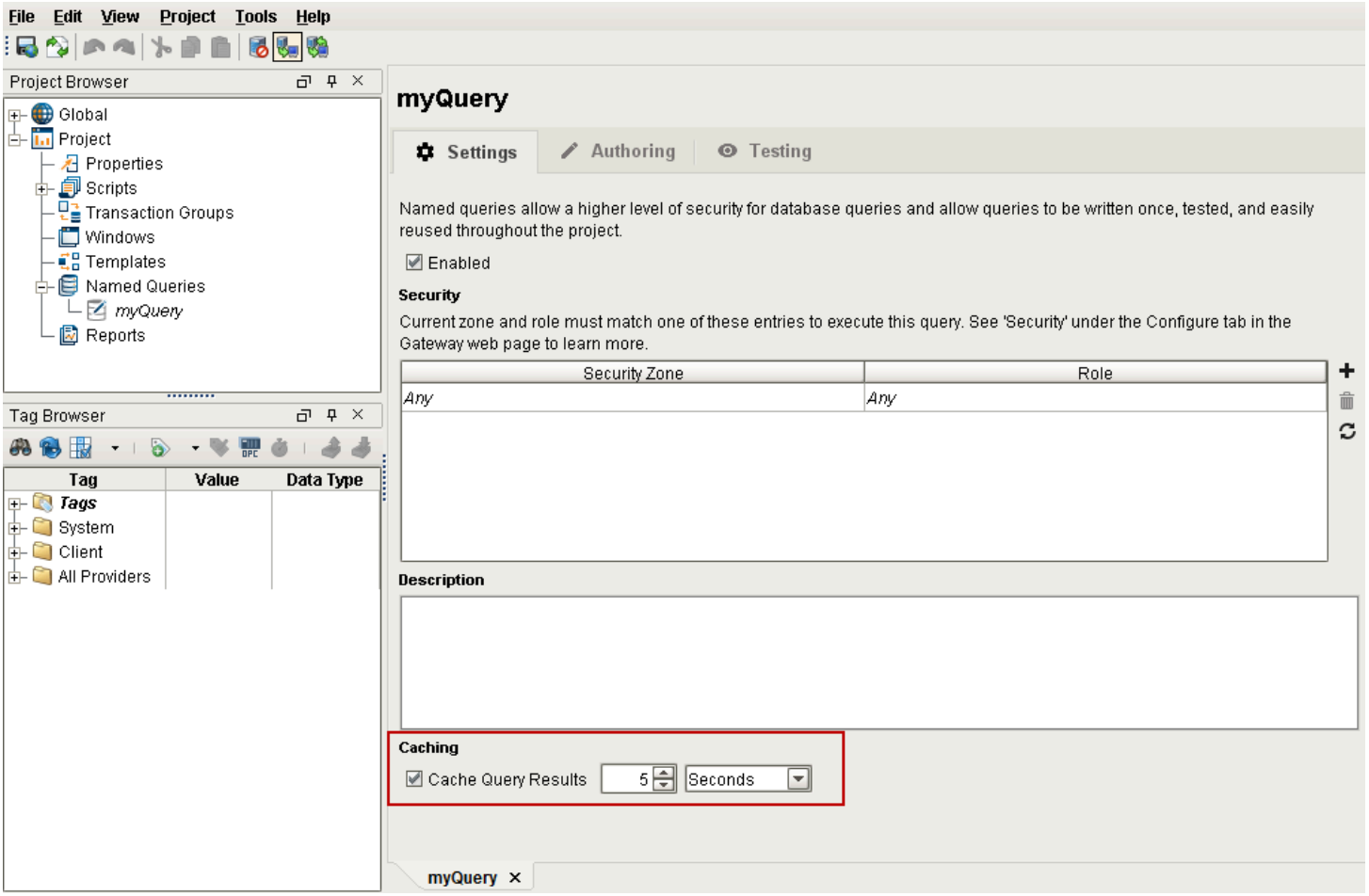


Use Cached Named Queries When Possible

Most resources in Ignition that can request a query will not cache the results for use by other resources: a SQL Query Binding that returns a result set will only do so for the one component, and can't be utilized by other resources in the same project. Thus, if two clients have the same window, the same query must be fired twice for both bindings to receive information from the database which is wasteful.

Named Queries are the exception. They can **cache the resulting dataset** for use by other resources in the project, as well as other instances of the same project. In the previously mentioned scenario, one client would trigger the Named Query to execute, and the other client would simply utilize the cached result set, reducing the number of queries running against the database.

If your project contains queries that poll slowly, or results sets that aren't frequently modified, then a Named Query with caching enabled is an efficient alternative to a SQL Query or DB Browse binding.



Use the Expression Language to Consolidate Multiple Queries

If multiple resources (such as multiple Tags, or multiple components) need separate values from the same database table, or a window contains multiple components that are all querying data from the same table, such as multiple Numeric Labels, it may be more efficient to have a single query run and fetch the large portions of the table, and then distribute the individual values to each component. This typically involves having some property or Tag query for several rows of data from a database table, and then using expression bindings or Expression Tags to retrieve individual values from the query.

Individual values may be retrieved from a dataset via the Expression Language: either an Expression binding on a property, or an Expression Tag. Here are two commonly used approaches to extracting a single value from a dataset in the Expression Language:

- The **Expression Language's Dataset Access** syntax. You may want to use the **try()** function in case the dataset is empty:

  Pseudocode - Expression Language Dataset Access Syntax

  ```
  {dataset}[rowIndex,columnIndex]
  ```

- The **lookup** function:

```
Pseudocode - Expression Language Lookup Function
```

```
lookup({dataset}, lookupValue, noMatchValue)
```

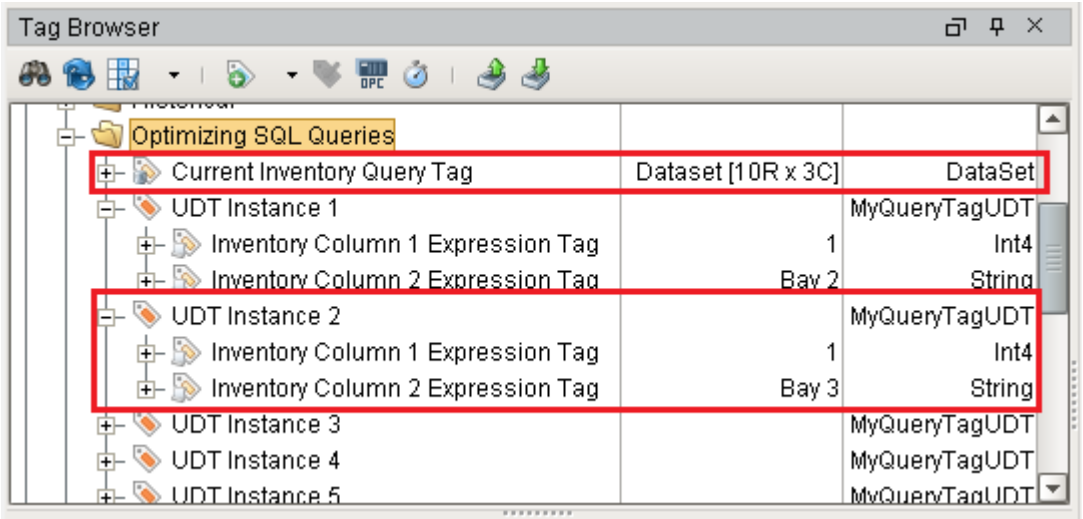
Restrict the Number of Query Tags in a UDT

Each Query Tag in a UDT will run a separate query per instance of the UDT. Assuming a scan class of one second, if a UDT definition contains 5 Query Tags, and there are 5 instances of that UDT, then there will be 25 queries executing every second.

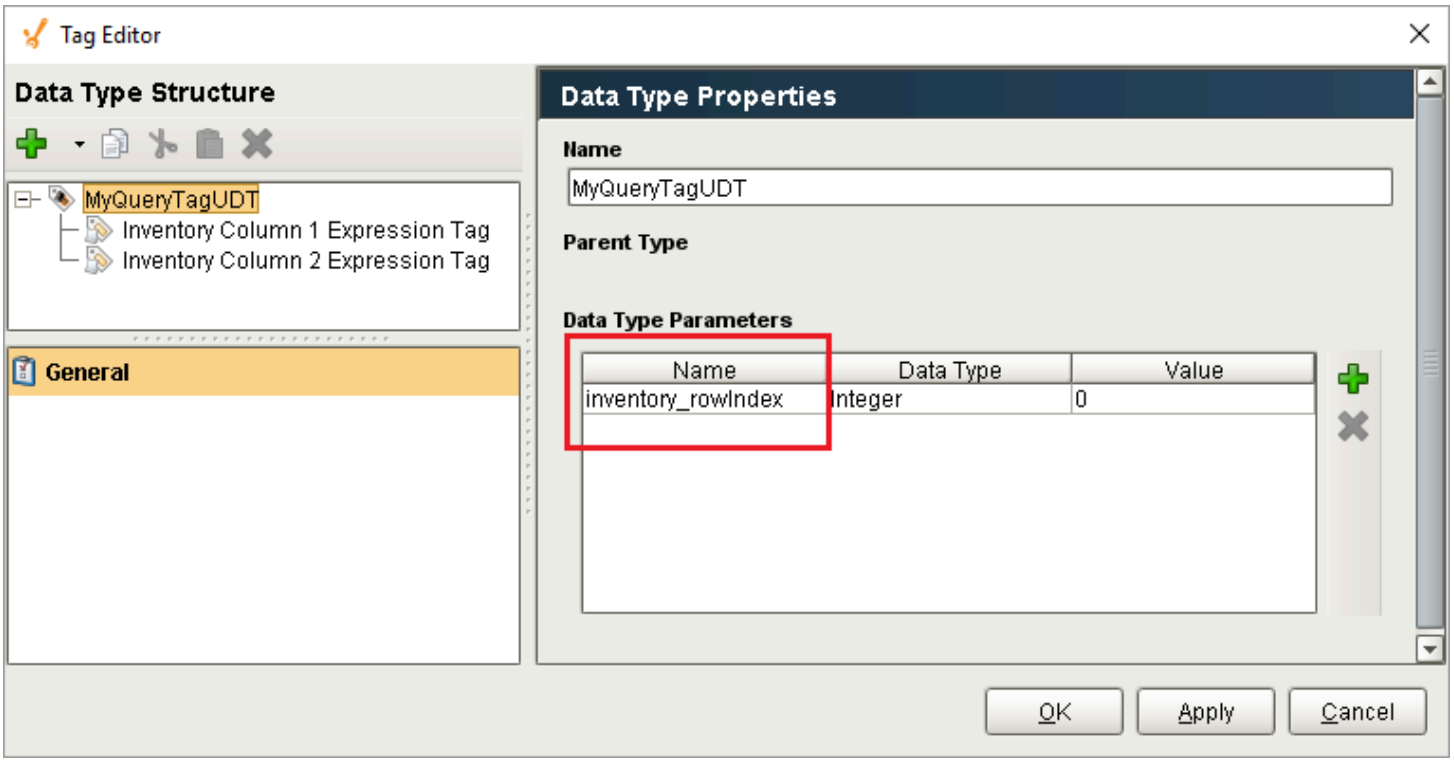
As **mentioned on this page**, the Expression Language can be used to reduce load on the database if multiple Query Tags are retrieving data from the same database table. Furthermore, UDT parameters can be utilized in the Expression Tags, so new UDT Instances can easily be configured to look up the appropriate values.

Single Database Tables

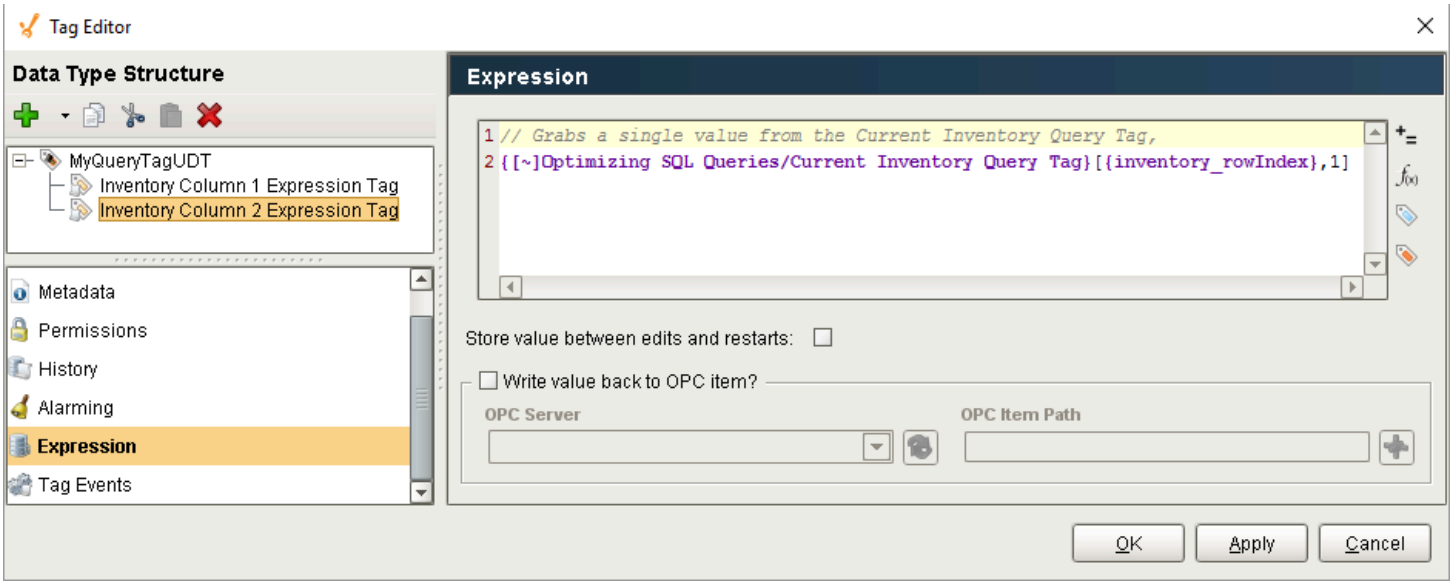
Below, the Tag named **Current Inventory Query Tag**, as the name implies, is a Query Tag retrieving multiple rows of data from a database table. We see that the highlighted **UDT Instance 2** contains two members: **Inventory Column 1 Expression Tag** and **Inventory Column 2 Expression Tag**, which are simply Expression Tags that are referencing individual cells from the **Current Inventory Query Tag**.



The UDT definition can use a parameter to specify an individual row in the Query Tag that each instance should focus on.



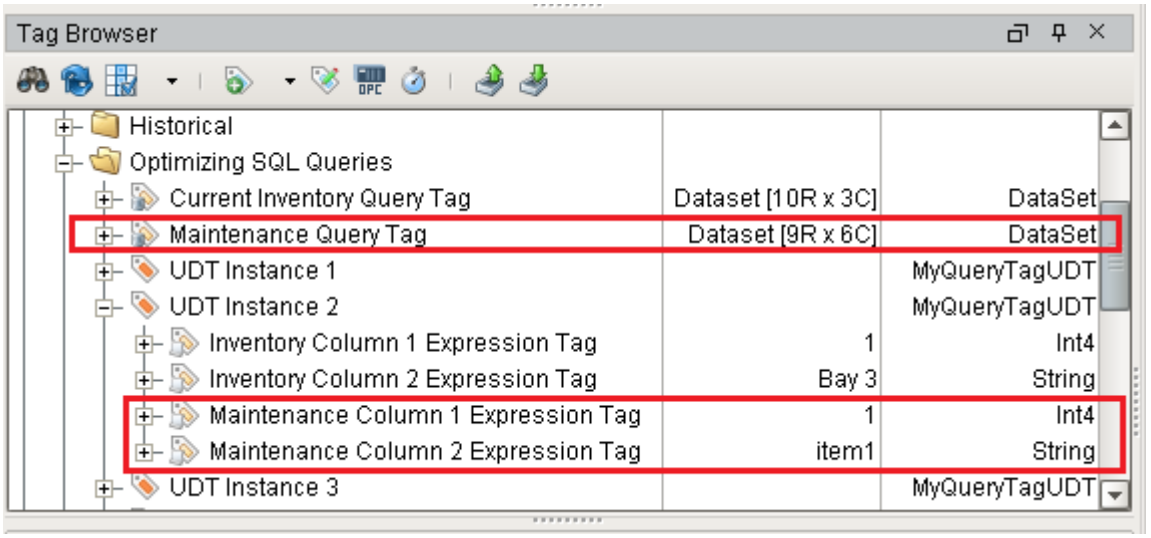
Each Expression Tag could use an expression like the following to look up individual values:



Multiple Database Tables

To add values from a separate database table, we simply need a separate Query Tag. In the image below, a new Tag named **Maintenance Query Tag** has been added, which is querying from a separate Database table. To incorporate this new data into our UDT instances, new Expression Tags have been added (**Maintenance Column 1 Expression Tag**, and **Maintenance Column 2 Expression Tag**) that simply reference specific values in the new Query Tag. Now, regardless of how many UDT instances exist in the Tag provider, we only have two Tags that are executing queries against the database.

For each separate table, we need to incorporate a single new Query Tag to collect all of the rows we want to show, add index parameters to the UDT definition, and add Expression Tags to our UDTs.



Scripting

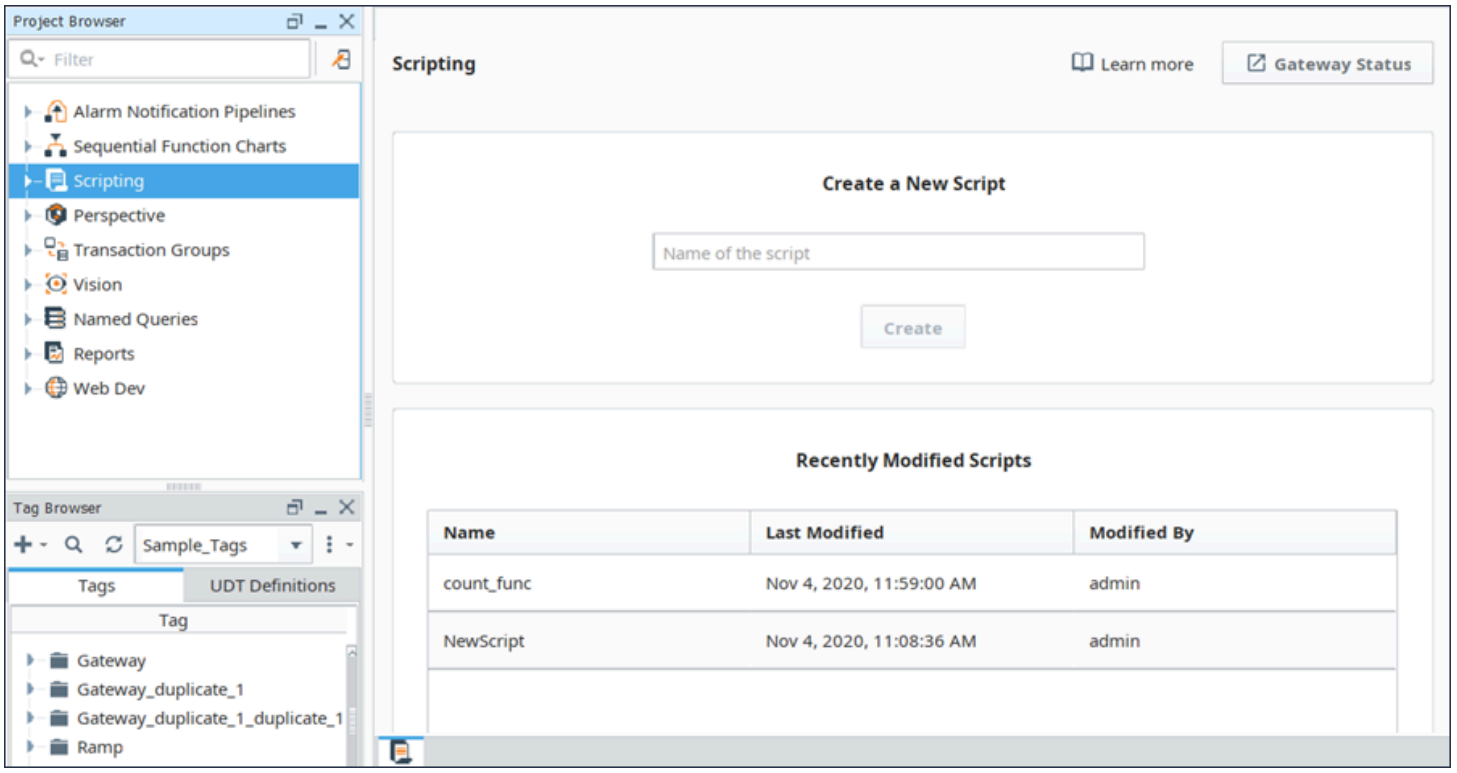
What Is Scripting?

Most of the time when we talk about "scripting" in Ignition we are talking about Python scripting, or writing code in the Python language. Python is a general purpose programming language that was developed in the early 1990s and has gained significant popularity in the 2000s. It is extremely readable, elegant, powerful, and easy to learn. As an added bonus, it gracefully interacts with Java, giving programmers an extremely powerful tool when paired with Ignition, which is written in Java.

The Scripting Welcome tab will appear when you have either the Scripting window or one of its children open. It allows you to create a script, and once you click Create, it immediately opens the window so you can start writing your script. It's a quick and efficient way to get right to your scripting tasks. At a glance, the Scripting Welcome tab will also show you any recently modified scripts along with the date it was modified and who modified it. You can even double click on a recently modified script and open it.

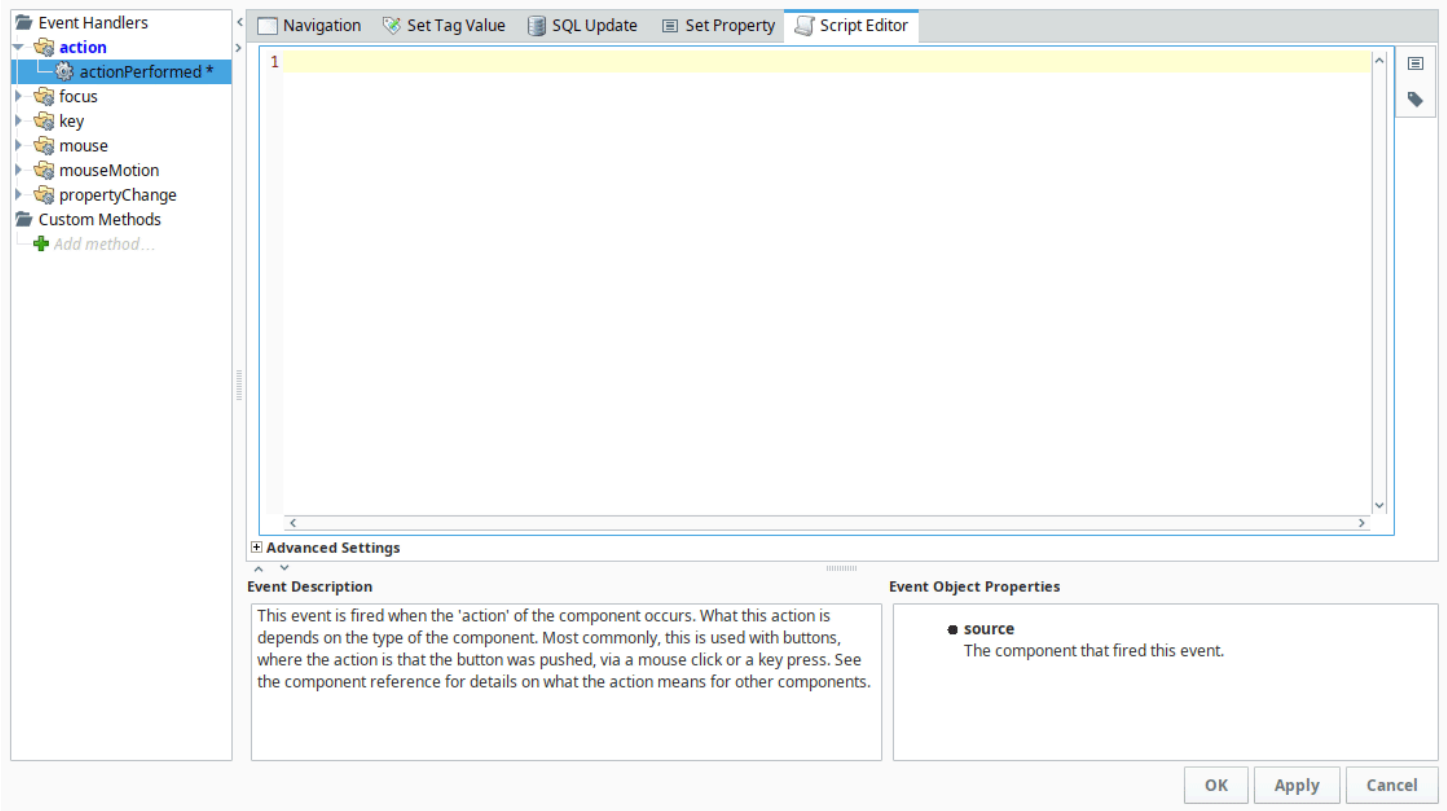
The Scripting Welcome tab provides a quick way to create a new script and update existing ones. The **Welcome Tab** should appear when the user has either Scripting, or one of its children items selected.

In Ignition, you will be mixing the core Python language with references to other components and a variety of our built-in **system functions**.



Is It Easy to Use?

Luckily, Python is a simple language to get started with. Using it in an event-driven system takes away a lot of the extra code that normally makes programming time consuming. For those that are already familiar with scripting (and those of you that are learning), we have a huge list of functions inside Ignition to do some of the common tasks in a single line of code. These **System Functions** are available while typing. Just start with **"system"** and press `ctrl-space` to see a list of available functions. The list will filter itself as you continue typing.



Where Is Scripting Used?

Python is used in many places in Ignition. Each location has its own events that trigger your scripts to run, and add functionality to your projects in different ways.

- **Components** - Add actions to components like buttons, customize the look and feel of charts and tables, and even set up a custom navigation schema.
- **Tags** - Create a script that runs on a Tag change and when an alarm goes active!
- **Reports** - Use scripting to create a customized data source, or create your own unique action to use with the scheduling system.
- **Alarm Notification** - Create custom rosters using scripting to dynamically change who gets notified with each new alarm event.
- **Client, Session, and Gateway** - Add a script that will run when certain events happen, such as when the Client or Gateway starts up, or on certain keystrokes.

Users that are new to Ignition focus mainly on the Component Binding system in **Vision** or **Perspective**, and for good reason. It's simple, flexible, and generally easy to understand without much of a computer background. However, Ignition has a complete scripting system built into every place you can think of. Using it is not a requirement, but it can add a significant degree of flexibility and customization to your projects. It allows you to create exactly what you need, giving you total control where pre-canned options fall short.

The majority of your scripting will be done in Event Handlers inside of components. This system makes it very easy to get started scripting with little to no experience. With the script builders, basic scripting like Navigation and setting Tag values takes just a few clicks. See also **Script Builders in Vision** and **Component Events and Actions.**

Python Scripting

About Python

While it is entirely possible to create a complete and powerful project in Ignition without writing a line of script, many designers will find that in order to complete projects with specific requirements, they need to learn at least a little Python. In our experience, most industrial projects involve lots of very complex and specific requirements.

This section is a short tutorial specifically for python, which should help get you started. It goes over all of the core concepts you will need for scripting in Ignition but the next section (**Scripting in Ignition**) goes over using Python directly inside Ignition.

- **Python Variables, Datatypes, and Operators**: Learn what a variable is and how to create it, what the various datatypes are, and what operators you can use on them.
- **Conditions and Loops**: Learn the common if/else type of statements, as well as loops.
- Functions: Learn about the **built-in functions** that can do complex work with a simple command, as well as **user defined functions**, which help compartmentalize the code.
- **Libraries**: Learn about our built-in system functions, as well as pulling in outside libraries.

Python or Jython?

You'll often hear Python referred to as "Jython" by advanced users of Ignition. Python is the language, Jython is the implementation of the language that we use. Most users of Python use the implementation called "CPython" - they just don't realize it. See **http://en.wikipedia.org/wiki/Python_(programming_language)#Implementations**.

One of the powerful things about using Jython is that your script has access to the entire Java standard library. For more information, see **Accessing Java**.

Many scripting users are blown away by their script's speed. We can't take credit for this - the Jython engine compiles the code when it is run. your Jython code is converted to Java bytecode, which means it runs natively in the JVM, which in turn can compile it to machine code. It's fast.

Which Version of Python is Used?

Ignition uses Jython 2.7. Jython is the Python programming language implemented over the Java Virtual Machine. When looking at outside documentation, such as on **www.python.org**, verify that you

are looking at the correct version.

Jython 2.7 allows us to use the standard functions and tools in Python 2.7, so if you want to look up something in the Python docs, make sure to use version 2.7 (**https://docs.python.org/2/**).

Scripting Basics

Python is very easy to learn, and with some understanding of its basic syntax, you can get started making your own scripts.

Hello World

Let's get right to everyone's favorite example, "Hello World." The following script will print out "Hello World" to the **output console**.

The `print` keyword is a handy tool in Python, allowing you to write text to the Output Console. This is useful for debugging your scripts. You can print multiple things by separating them with commas.

Variables

Variables are created by simply assigning a value to them. Variables do not need to be declared, because Python has a dynamic type system. That means Python figures out the type of the variable on the fly when the script is executed.

The following script would print out: 15

Python - Using Variables

```
x=5
y=3
print x*y
```

Strings

Strings are defined in Python with a matching pair of 1 or 3 single or double quotes. There are few times when the type of quotation mark you use matters - but one common reason to choose one or the other is for 'escaping' other quotes inside your content. Some of the rules are shown here:

Python - Using Single and Double Quotes

```
print "This is my text"     # Using double quotation marks
print 'This is my text'     # Using single quotation marks
print "This is my text'     # This will not work because python does not allow
mixing the single and double quotation marks
print "My name is 'David'"  # This will print: My name is 'David'
print 'My name is "David"'  # This will print: My name is "David"
print 'My name is Seamus O\'Malley' # This will print: My name is Seamus O'Malley
```

Triple quotes (single or double) can be used to make 'escaping' both single and double quotes inside your string easier, or to write multi-line comments:

Python - Multiple Lines of Comments Using a Triple Quote

```
'''
This is a lot of text
that you want to show as multiple lines of
comments.
Script written by Professor X.
Jan 5, 1990
'''
print 'Hello world'
```

Strings can also be prefixed with certain characters to change how they are interpreted - for instance, a leading u character marks a string as Unicode, allowing for characters outside of the ASCII range to be used.

Python - Unicode Prefix on String

```
print u"äöü"
```

Whitespace

Perhaps Python's most unique feature is logical blocks which are defined by an indentation in Python. A colon (:) starts a new block, and the next line must be indented (typically using a tab or 4 spaces). The block ends when the indentation level returns to the previous level. For example, the following will print out "5 4 3 2 1 Blast-off" with each value on a new line. The final print is not part of the while loop because it isn't indented.

Python - Logical Blocks / Indentation

```
countdown = 5
while countdown > 0:
    print countdown
    countdown = countdown - 1
print "Blast-off!"
```

Comments

Comments are a way to document your Python script. There are several ways to use comments, but the best advice we can give is to use them as much as possible! There are a few ways to make a comment in Python.

Individual Lines

You can start a line with a pound/hash (#) sign, or put one anywhere in a normal line of code.

Python - Document Scripts Using Comments

```
# this is a comment
print 'Hello world' # this is also a valid comment
```

Blocks of Lines

While Python doesn't explicitly have a way to block comment (comment out multiple lines), multi-line strings are functionally similar, and a common convention.

Comment Many Lines with a Keyboard Shortcut

In Ignition, you can use the Ctrl-/ keyboard shortcut to comment several lines of code at once. Just highlight one or more lines of code and hold the Ctrl key and press "/". This will prepend all of the selected lines of code with the pound/hash (#) sign. Press Ctrl-/ again to remove the pound/hash sign.

Control Flow

Control Flow statements, that is the ifs and loops, make the language do things differently based on the various conditions. Python has all of the basic control flow statements that you'd expect from a programming language.

If and Else

An if statement allows you to check if a condition is true or not true. Depending on the condition, you can either execute a block of code, or do something else. Many of these can be chained together to determine under what conditions should certain code execute.

Pseudocode - If Statement

```
if condition == True:
    print value1
```

For and While

Looping can be done with a for, which executes a block of code a set number of times, or a while, which executes a block of code until a certain condition is met. Both can be incredibly useful.

Pseudocode - For Statement

```
for item in myList:
    print item
```

Variables, Datatypes, and Objects

Without incorporating any Ignition-specific objects, it is important to understand the basics of Python. This section seeks to introduce three main Python principles: Variables, Data Types, and Operators. Additionally, this page and the sub-pages herein will provide plenty of examples to get you started.

This section of the manual attempts to introduce these principles so that they may later be used in conjunction with Ignition. The information in this section is far from comprehensive, as **Python's official documentation** is a better reference for all things Python. However, this page serves as a great way to jump right into the action.

Variables

Variables are created by simply assigning a value to them. Variables do not need to be declared, because Python has a dynamic type system. That means, Python figures out the type of the variable on the fly when the script is executed.

The following script would print out: 15

Python - Declaring and Assigning Variables

```
x=5
y=3
print x*y
```

A space may be included on either side of the assignment operator ('='), but is not required. Thus, the following example would be functionality identical as the example above.

Python - Declaring and Assigning Variables with Spaces

```
x = 5
y = 3
print x * y
```

Built-in Datatypes

Python features several built-in data types. Below is an overview and links to appropriate pages where applicable.

None

There is a special value in Python called `None` (with a capital N). This is simply a special value that means no value. This value is equivalent to Java's `null` value. None can be used to initialize a variable, but is best when checking to see if something exists before doing extra work:

Python - None

```
# If some value is not equal to None...
if something != None:
    # ...then do some work
    doWork()
```

Booleans

Python has two built-in values to represent true and false values: `True` and `False`, respectively (note the capital letters). These can be used when testing for truth, and are implicitly returned when the comparison operator is used:

Python - Booleans

```
# Prints True
print 1 == 1

# Prints False
print 1 == 0
```

When using Booleans, suchs as predicates in an if-statement, you typically don't have to use True or False directly. Instead, many other values are considered True or False. The following values are also considered False:

- None
- Numeric values of 0, such as 0
- Empty Sequences and Dictionaries, like `[]` or `{}`

Values aside from the ones mentioned above are considered true, so you can easily utilize the existence of a non-zero value as a True. Note that for integers both positive and negative numbers are True, only a value of 0 is False.

Strings

Literal **strings** can be typed in using either double quotes or single quotes. This can be handy when your string contains one quote or the other. You can also use the backslash character to escape special characters including these quotes. See the **Strings** page for more information.

Numeric Types

Numbers can just be typed in normally, like 42 or 3.14159 . Adding a decimal point differentiates an Integer from a Float. More information on Numeric types can be found on the **Numeric Types** page.

Colors

Working with colors in Python is remarkably easy. You can simply use any tuple of 3 or 4 integers to represent a color in RGB or RGBA. For example, to set a label's text color to red, you can simple do something like this:

Python - Tuple as Color

```
label = event.source
label.foreground = (255,0,0) #(red,green,blue)
```

Additionally, the **system.vision.color** function allows you to pick a color in a similar fashion:

Python - System Function as Color

```
newColor = system.vision.color(255,0,0)
```

Lists and Tuples

Python offers a variety of sequence types: most notably **Lists and Tuples**. These are ordered collections, meaning they are indexed and the sorted order is maintained. More information on these types can be found on the **Lists and Tuples** page.

Python - System Function as Color

```
newList = [1,2]
newTuple = (2,5,7)
```

Dictionaries

A **Dictionary** is a very useful type that holds a set of key-value pairs. Unlike sequences, they are not ordered, so there is no need to sort them. Instead you give each Value in the dictionary a Key, which handles as a reference to Value. You may have used these in other languages and know them as hashmaps, maps, associative memories, or associative arrays. More information on Dictionaries can be found on the **Dictionaries** page.

Python - System Function as Color

```
newDictionary = {"itemName":5}
```

JSON

JSON stands for JavaScript Object Notation. While it is not a data type, it is a way of defining data in a human readable format, and is commonly used in many applications. It comes from the Javascript programming language, but it is language independent. Each JSON object contains lists and objects. A **list** is a series of ordered values separated by commas that is commonly used within Python. The object works like a **dictionary**, using any number of name/value pairs, where each value could be any basic data type, a list, or even another object with its own name/value pairs. Because JSON is just a way of defining data, it can be used in many different programming languages, including Python. This makes it a useful tool for defining data in a way that humans can easily read.

Ignition also has two scripting functions that allow you to convert between a JSON string and a native Python object: **system.util.jsonEncode** and **system.util.jsonDecode**.

Official Documentation

For more information on JSON, see **http://www.json.org**.

Datasets

A **Dataset** is a multidimensional collection of values, stored in a manner similar to how values on a spreadsheet appear. Python does **not** natively have a Dataset type. Instead these datasets were created for use inside of Ignition. There are two types of datasets:

- Dataset: Sometimes called a "Standard Dataset", this type is commonly used on many Vision Components, such as the **Power Table** or **Chart**, to display multiple values simultaneously.
- PyDataset: Short for "Python Dataset", these datasets act in a manner very similar to a Python Sequence when it comes to accessing specific values, or iteration (see Lists and Tuples). Ignition's built-in system functions that interact with the database typically return a PyDataset.

While there are two types of datasets, you can easily convert one type of dataset to the other. Additionally, you can easily create a dataset from a script.

Python - Creating a Dataset

```
header = ["The Only Column"]
rows = [[1],[2],[3]]

myDataset = system.dataset.toDataset(header, rows)
```

Dates

Dates and times can be created in Python with the `datetime` and `time` libraries. However, the Ignition's built-in system.date functions can also be used instead without having to import either library.

Python - Show the Current Datetime

```
currentTime = system.date.now()

print "The current time is: %s" % currentTime
```

Using the In-Keyword

Python's `in` keyword can be used to check the contents of something for a specific instance of another object. One use is to check the contents of a string for a certain substring:

Python - Using In Keyword to Look for a Substring

```
```

```
myString = "Hello World"
subString = "World"

# Here we are using an if-statement to look for the substring. If the substring
exists inside of myString, the expression returns True.
if subString in myString:
    # This will print out, because the substring "World" exists in the string
"Hello World"
    print "The word 'World' appears in myString"
```

The `in` keyword can be used to look for a certain object inside of a sequence:

Python - Check for the Presence of an Object

```
myList = [1,2,3,4]

# If the integer 3 exists in myList, then this will print True, otherwise it will
print False.
print 3 in myList
```

Python - Check for the Presence of an Object

```
# Check for the Administrator role.
if "Administrator" in system.security.getRoles():
    print "Administrator found"
```

Additionally, the `in` keyword can be used in an expression on a `while` loop:

Python - Check for the Presence of a Keyword in an Expression

```
myList = [1,2,3,4]

# As long as the integer 4 is in myList, the while loop will continue to iterate.
while 4 in myList:
    # pop(0) will remove the first element in the list
    myList.pop(0)
else:
    print "all done!"
```

Basic Operators

Python has many common operators as you would expect, or are at least familiar with if you've worked with other scripting languages.

These are just the basics. There are other operators, like bit shift operators and more. Read about them at: **http://docs.python.org/library/stdtypes.html**

Operator Reference

Arithmetic Operators

> **ARITHMETIC OPERATORS**
>
> In regards to Arithmetic Operators, the precision of the returned value depends on the data types of the arguments: including at least one **float** will return a **float**, otherwise, an **integer** is returned.

Operator	Meaning	Example	Output
+	Addition. Note that the data type of the returned object depends on the datatype of the arguments being used:	`print 5 + 9` `print 5.0 + 9`	`14` `14.0`
-	Subtraction	`print 5 - 9`	`-4`
*	Multiplication	`print 5 * 9`	`45`
/	Division	`print 10 / 4.5`	`2.2222222222222223`
//	Floored Division: will divide and return just the nearest integer value, even if dividing floats.	`print 10 // 4.5`	`2.0`
%	Modulo: returns just the remainder of the dividend (the left argument) after being divided by the right argument. In the example on the right, the 4.5 divides evenly twice, leaving 1.0 as a remainder.	`print 10 % 4.5`	`1.0`
**	Power: raise the the number on the left to the power of the number on the right.	`print 5 ** 3`	`125`

Boolean Operators

Operator	Meaning	Example	Output

or	Returns True if either argument is True. If both are False, then returns False.	`x = False`
		`y = True`
		`print x or y`
			`True`

and	Returns True only if both arguments are True. Otherwise, returns False.	`x = False`
		`y = True`
		`print x and y`
			`False`

not	Returns a boolean value that represents that opposite value of the trailing expression: False becomes True, and True becomes False.	`x = True` `print not x`	`False`

Comparison Operators

Operator	Meaning	Example	Output
<	Less than	`print 10 < 5` `print 10 < 10`	`False` `False`
<=	Less than or equal to	`print 10 <= 5` `print 10 <= 10`	`False` `True`
>	Greater than	`print 10 > 5` `print 10 > 10`	`True` `False`
>=	Greater than or equal to	`print 10 >= 5` `print 10 >= 10`	`True` `True`
==	Equal	`print 10 == 10` `print 10 == 0`	`True` `False`
!=	Not equal	`print 10 != 10` `print 10 != 0`	`False` `True`

is	Returns True if both arguments are referring to the same object, otherwise, returns False.	`x = 10`
		`y = 5`
		`print x is y`
			`False`

is not	Returns True if both arguments are referring to different objects.	`x = 10`
		`y = 5`
		`print x is not y`
			`True`

Order of Operations

Note that there is an order of operations for Arithmetic and Boolean operators that can be modified with parenthesis. For arithmetic they are grouped in tiers and evaluated left-to-right within a given tier:

- Parenthesis
- Exponents
- Multiplication, Division, Modulo
- Addition, Subtraction

Python - System Function as Color

```
6+10*2   # produces 6+(10*2) = 26
6*10+2   # produces (6*10)+2 = 62

10*6%2   # produces (10*6)%2 = 0
10%6*2   # produces (10%6)*2 = 8
```

For Boolean operators:

- Not
- And
- Or

Numeric Types

The Differences Between the Types

Within Python, there are a few numeric types that we will often use within the scope of Ignition: integers, floats and booleans. **Integers** have at least 32 bits of precision and consist of non decimal values (1, 2, 50, 246). When working with larger numbers, Python also supports a **long** type that has unlimited precision. **Floats** consist of numbers with decimal values (1.0, 2.4, 50.7, 246.8734). **Booleans** are a unique subset of integers. They are either **True** or **False**, but can also be represented using a 1 or 0, respectively.

Numbers in Arithmetic

When using numeric values in expressions or equations, Python is very flexible in how it handles multiple types in the same operation. When using two of the same type in an expression, the outcome will always be of that type. Adding two integers together yields an integer, while multiplying two floats will yield a float. With division, using integers will always result in a floor division so that the answer remains an integer. Floor division will simply remove the decimal, and will not round up at any point.

When using different types, the more precise type will be used for the outcome. Doing an operation with one integer and one float will produce an outcome that is a float. Adding a float and an integer together will yield a float.

Below, we have some examples that show off some of the different ways that Python handles expressions. You can try these out in the interactive interpreter of the **Script Console**.

Type of Arthimetic	Description	Example	Output

Addition and Subtraction	Adding/Subtracting integers will return an Integer.
	If at least one of the numbers is a float, then the resulting number will also be a float.
	Otherwise the result will be an integer
		`4 + 5 # int + int = int`
		`4.0 + 5 # float + int = float`
		`4.0 + 5.3 # float + float = float`
			`9`
			`9.0`
			`9.3`

Multiplication	Similar to Addition and Subtraction, multiplication will only return a float if at least one of the numbers is a float	`4 * 5 # int * int = int` `4 * 5.0 # int * float = float`	`20` `20.`
Division	Dividing an integer by another integer results in floor division: meaning the remainder of division is never returned. Note that this is not the same as rounding to the nearest integer. To include the remainder, at least one of the numbers needs to be a float	`4 / 5 # int / int = int` `4 / 5.0 # int / float = float`	`0` `0.8`

Operators and Functions

Python offers a wide range of operators that can be used to perform calculations on numeric values. Additionally, there are many functions that can perform specific operations to numeric values. The tables below contain some of the most common operations and functions, but this is not an exhaustive list.

Operators

Operator	Description	Example	Types that use the Operator
+	Used to find the sum of two numeric values. Can also be used to concatenate string values.	`5 + 1 = 6` `"Hello " + " World"`	Numeric Types, Strings
-	Used to find the difference of two values	`5 - 1 = 4`	Numeric Types
*	Used to find the product of two values	`5 * 2 = 10`	Numeric Types
/	Used to find the quotient of two values	`4 / 2 = 2`	Numeric Types
%	Modulus operator. When used with numeric types, returns the remainder of the quotient of two values The operator is also seen in **string formatting**.	`5 % 2 = 1`	Numeric Types, Strings
**	Used to find a value to the power of another value	`5 ** 2 = 25`	Numeric Types

Functions

Function	Description	Example
int(x)	Converts x to an integer.	`int(4.8) = 4`
long(x)	Converts x to a long.	`long(5.7) = 5`
float(x)	Converts x to a float.	`float(4) = 4.0`
abs(x)	Absolute value of x.	`abs(-4.6)=4.6`
round(x[, n])	Rounds x to n digits. If n is not specified, default is 0.	`round(5.674)=6.0` `round(5.674, 2)=5.67`

Strings

What are Strings?

In Python, as well as most other programming languages, a string is a grouping or string of characters that are used literally instead of as names of variables or other objects. In Python, they can be enclosed in either single quotes or double quotes:

Python - Print Statements

```
# These two print statements will both print out a sentence that looks the same as the other.
print 'These will both print the same!'

print "These will both print the same!"
```

Strings are also considered sequences, which means they can be **iterated through** much like a list.

String Escape Character

Strings use a special character called an "escape" character to denote when something different should be happening within the string. In Python, the escape character is the backslash (\ ). The escape character is mainly used in a few different ways:

The Escape Character and Quotation Marks - "" and ''

Comments	Example	Output
Here, we use the escape character to let Python know we want to ignore the second single quote and directly print it. This works with either single or double quotes.	`print 'doesn't'`	doesn't
Alternately, we can enclose the string in the opposite quotes that we are using in the string. Because we enclose the string in double quotes here, the single quote prints normally.	`print "doesn't"`	doesn't
This also works in reverse, where we can enclose the string in single quotes and use double quotes within.	`print '"Test", he said.'`	"Test", he said.
Of course, we can also use the escape character with double quotes enclosing the string.	`print ""Test", he said."`	"Test", he said.

New Lines and Tab Spacing

Comments	Example	Output
A new line may be specified with the escape character and "n". Alternatively, a **multi-line string** may be used to add line breaks to a string literal.	`print "First line\nSecond line"`	First line Second line
Tab-spacing may be added to a string literal with the escape character and "t"	`print "Hello \tWorld"`	Hello World

Specifying a Backslash

Comments	Example	Output
If you need to place a backslash inside of a string literal, simply use the escape character twice.	`print "Folder\file.txt"`	Folder\file.txt

Multi-Line Strings

If a string literal should contain multiple lines, you can use a multi-line string, which is created by using three quotation marks.

Example	Output
`print '''This is on multiple lines'''`	This is on multiple lines

This is useful when using the **system.db** functions, as full SQL queries are typically passed to these functions, so using a multi-line string allows you to write the query in an easy to read format.

Python - Multi-line String

```
query = ''' SELECT * FROM myTable
            WHERE id < 10'''

system.db.runPrepQuery(query,[])
```

Raw Strings

Sometimes, it is necessary to print the raw string without allowing escaped characters. This is done by placing the letter "r" in front of the string.

Python - Raw String

```
print r'This string \n will print out directly as written.'
```

This is especially useful in cases where a file path needs to be hard coded as a string literal:

Python - File Path as a Raw String

```
# Specifying a Windows file path
myPath = r"C:\Folder\Another Folder\file.txt"
print myPath

# Specifying a Linux file path
myPath = r"/home/Directory/Another Directory/file.txt"
print myPath
```

Unicode Strings

Strings that contain characters beyond 7-bit ASCII, such as é or ? need to be marked as unicode strings by placing the letter **u** in front of the string. Implementing unicode can be additionally useful when working with system functions as some functions will fail if tags paths include special characters.

Python - Unicode String

```
print u'été'
```

Combining Strings

Two different string type variables can actually be combined or concatenated using the plus (+) sign. It is important to understand that they are concatenated exactly.

Python - Concatenated String Type Variables

```

```

```
a = "this"
b = "that"


# Will print 'thisthat' because there was no space at the end of a or the
beginning of b.
print a + b

# Will print 'this that' because we added a space between them.
print a + " " + b
```

String Indexing

Strings in Python are actually indexed, with the first character having an index of 0. To grab a value at a specific index, you place a value within square brackets ([ ]) after the string variable.

Python - Indexed Strings

```
a = "Ignition"

# Will print out 'I', since it is in the zero position.
print a[0]

# Will print out 't', since it is in the fourth position.
print a[4]
```

You can also use a negative index, which will start from the right side of the string with the last character having a index value of -1. This is useful for getting the last character when you aren't sure how long the string is.

Python - Negative Indexed String

```
a = "Ignition"

# Will print out 'o', since it is the second to last character.
print a[-2]
```

Slicing Strings

Using the string index values, we can actually grab slices or parts of the string. Similar to grabbing an individual character, we place two values separated by a colon within square brackets. You can think of these numbers as the slices between characters, with 0 before the first character, 1 before the second, and so on. It should look like this: string[4:7] The first value is the start location, while the second value is the end location.

Python - Sliced Strings

```
a = "Inductive Automation"

# Will print out 'ctive Auto'. Note how the space is counted as a character,
# and the 14th location is before the m, which is not included.
print a[4:14]
```

If left blank, the first value will default to 0, while the second value will default to the length of the string. Additionally, negative values can be used as well, just like with the index.

Python - Negative Value Sliced Strings

```
a = "Inductive Automation"

# From index 0 to index 12.
print a[:12] # Will print out: Inductive Au

# From index 12 to the end of the string.
print a[12:] # Will print out: tomation

# From the 4th character from the end to the 5th index. Note this prints nothing
because the start character is after the end character.
print a[-4:5] # Will print out:

# From the 8th character from the end to the 2nd character from the end.
print a[-8:-2] # Will print out: tomati

# From index 5 to the 5th character from the end.
print a[5:-5] # Will print out: tive Autom
```

You can use all of these together (and with integer variables or functions) to have a very flexible way to format strings.

String Formatting

Also known as string substitution, it allows you to enter in values into a string, similar to having a variable inside of the string. This is useful when doing a database query where you can build the query as a string, and then add in the query parameters when executing the query.

Doing this requires the use of the percent (%) sign followed by the parameter type within the string. After the string, you then use the percent sign followed by the parameter values within parentheses in order.

Python - String Substitution

```
a = "I have %i apples."

# Will print exactly as written, because we are not substituting any values in.
print a

# Will print with an 8 substituted in for the %i. I have 8 apples.
print a % (8)
```

Instead of using literal values, you can also use variables as parameter values instead. This is useful when pulling the value from a function. Each string can also have multiple values substituted, as long as the parameters are in the order that they are in the string.

Python - String Substitution using Variables as Parameters

```
apples = 10
oranges = 14
peaches = 5

a = "I have %i apples, %i oranges, and %i peaches."

# Will print with the variable values substituted in for the %i. I have 10 apples,
14 oranges, and 5 peaches.
print a % (apples, oranges, peaches)
```

There are a few different types used with string formatting, the most common of which are listed here. The character values are what go after the percent sign.

Type	Character
Signed Integer	i
Floating Point Decimal Format	f
Floating Point Exponential Format	e
String	s

String Search Example

You can bring all this together for a simple way to print out just the last word in a string.

Python - Find the Last Word of a String

```
# Find just the last word of a string.
myString = "Inductive Automation"

lastSpaceIndex = myString.rfind(" ")

# Add one to not include the space.
print myString[lastSpaceIndex+1:] # Will print out: Automation
```

String Functions

Python strings have many functions available that can manipulate the string or give information on the string. The most common are in the table below.

Function	Description	Example
len(string)	Returns the length of the **string**.	`a = "Inductive Automation"` `print len(a)`

x in string	Will return **True** if **x** is within the string, **False** if not. Can also be used to iterate through the string.	`string = "Inductive Automati`
		
		`f 'd' in string:`
		`  print "There is a d in the`
		`for letter in string:`
		`  print letter`

string.find(x, [, start[, end]])	Returns the first location of the substring **x** from the string. Returns -1 if the substring is not found.	`string = "Inductive Automati` `print string.find(" ")`
string.rfind(x, [, start[, end]])	Similar to find, but returns the last location of the substring **x**. Returns -1 if not found.	`string = "Inductive Automati`
string.upper()	Returns a copy of the string with all characters uppercase. This is useful when comparing user input to a string value, as the user may use a different case for certain letters.	`userInput = "administration"`
string.lower()	Returns a copy of the string with all characters lowercase. This is useful when comparing user input to a string value, as	`serInput = "ADminIStration"`

Function	Description	Example
	the user may use a different case for certain letters.	
string.capitalize()	Returns a copy of the string with the first character capitalized and all other characters lowercase.	`string = "here is my sentenc`
string.title()	Returns a copy of the string with the first letter of each word capitalized and all other characters lowercase.	`string = "here is my sentenc`

string.strip([x])	Returns a copy of the string with leading and trailing characters removed, where **x** is the string of characters.. If **x** is omitted, then removes whitespace from the leading and trailing edges of the string. `lstrip()` and `rstrip()` may instead be used to strip characters from the leading or trailing edge	`string = "    This string ha`
		`print string`
		`print string.strip()`
		`fencedString = "|||||___My S`
		`print fencedString.strip("|_`
		`print fencedString.lstrip("`
		`print fencedString.rstrip("`

string.count(x[, start[, end]])	Returns the number of occurrences of **x** in the string. A **start** and **end** can be specified that will limit the count to that area.	`string ="Inductive Automatic`
		`print string.count('i')`
		`print string.count('i', 4, 9`

Function	Description	Example

string.split([delimiter[,maxsplit]])	Returns a list of the words in the string. Optionally, specifying a **delimiter** will split the string on the delimiter string. Specifying a **maxsplit** will split the string a maximum number of times, with the remainder of the string as the final list object. The number of items in the list will not be more than maxsplit + 1	`sentence = "This is an examp`
		`print sentence.split()`
		`dashedSentence = "Yet-anothe`
		`print dashedSentence.split('`

string.rsplit([delimiter[,maxsplit]])	Similar to split, but splitting is performed from right to left. Thus if **maxsplit** is smaller than the total number of **delimiter** in the string, only the rightmost words will be split off as separate items.	`sentence = "This is an examp`
		`print sentence.rsplit()`
		`dashedSentence = "Yet-anothe`
		`print dashedSentence.rsplit(`

string.join(x)	Returns a copy of the string that is the concatenation of the strings in the iterable **x**. The string calling the join is what will separate the values of the iterable.	`a = '-'`
		`b = 'abcdef'`
		`print a.join(b)`

string.replace(old, new[, count])	Returns a copy of the string where occurrences of the **old** substring are replaced with the **new** substring. Optionally, if **count** is specified, will	`string ="Ignition is good!"` `print string.replace("good",`

Function	Description	Example
	only replace the first count number of occurrences.	

Lists and Tuples

Sequences

Like **Strings**, Python has two other common sequence types: Lists and Tuples. While lists and tuples are similar to strings in that they share many of the same functionality, they are unique in that they are used to group other values together. Both lists and tuples define a number of values separated by commas, but lists are enclosed in square brackets **[ ]** and are mutable, meaning their contents can change, while tuples are enclosed in parentheses **( )** and are immutable, meaning their contents can't change.

Lists

As stated above, lists are groups of comma separated values enclosed in square brackets, but can utilize many of the features available to other sequences like strings.

Python - Lists

```
# Lists are very simple to create.
myList = [1, 2, 3]

print myList # Will print out: [1, 2, 3]

# Empty lists can also be created, to have items added to them later.
myList = []

# Lists are not confined to hold values of a single data type either.
myList = [1, "hello", 3.3]

# Lists can even hold other lists! In this case, myList would actually hold 5
elements:
# 1, a list, 4, 'that', and another list. The last list also contains a list as
well.
myList = [1, ["this", 3.3], 4, 'that', [6, [7.7, 'other'], 9]]
```

List Concatenation

Like strings, lists also support concatenation and can be combined to form a single larger list.

Python - List Concatenation

```
a = [1, 2, 3]
b = ['four', 'five', 'six']

print a + b # Will print out [1, 2, 3, 'four', 'five', 'six']
```

List Indexing

Like strings, lists are also indexed, which allows you to grab single values or splice them to get ranges of values. Just like everything else in Python, lists start with 0.

Python - List Indexing

```
myList = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

print myList[3] # Will print out: d

print myList[-2] # Will print out: f

print myList[2:5] # Will print out: ['c', 'd', 'e']
```

When trying to index a nested list, we simply need to add a second index after the first.

Python - Index a Nested List

```
myList = [1, 2, [3, [4, 5, 6, 7], 8], 9, 10]

print myList[2][1][3]  # Will print out: 7
```

Appending to Lists

You can add values directly to the end of a list with the append() function. You can add anything that is normally allowed in a list.

Python - Appending to a List

```
myList = []

myList.append('Hello')
myList.append('World')

print myList # Will print out ['Hello', 'World]
```

List Functions

Below is a list of common list functions. Some of them are similar to other sequences like strings, while others are unique to lists because lists are mutable.

Function	Description	Example	Output
len(list)	Returns the length of the **list**.	`list = [1, 2, 3, 4, 5]` `print len(list)`	`5`

x in list	Will return **True** if **x** is within the list, **False** if not. Can also be used to iterate through the list.	`list = [1, 2, 3, 4, 5]`
		
		`if 4 in list:`
		`  print "There is a 4 in the list"`
		
		`for value in list:`
		`  print value`
			`There is`
			`1`
			`2`
			`3`
			`4`
			`5`

list.index(x)	Will return the index number of item **x**. Throws an error if item is not found.	`list = [1, 2, 3, 4, 5]`
		
		`print list.index(3)`
			`2`

min(list)	Returns the smallest item of **list**.	`list = [1, 2, 3, 4, 5]`
		
		`print min(list)`
			`1`

max(list)	Returns the largest item of **list**.	`list = [1, 2, 3, 4, 5]`
		
		`print max(list)`
			`5`

list.count(x)	Will return the number of times **x**	`list = [1, 2, 3, 4, 5, 4]`
		
		`print list.count(4)`
			`2`

Function	Description	Example	Output
	appears in the list.		

list.append(x)	Will add **x** to the end of the list.	`list = [1, 2, 3, 4, 5]`
		`list.append(6)`
		`print list`
			`[1, 2, 3`

list.insert(i, x)	Will insert **x** at position **i**.	`list = [1, 2, 3, 4, 5]`
		`list.insert(1, 5)`
		`print list`
			`[1, 5, 2`

list.remove(x)	Will remove the first **x** from the list.	`list = [1, 2, 3, 2, 4, 5]`
		`list.remove(2)`
		`print list`
			`[1, 3, 2`

list.pop([i])	Will remove the item at index **i**, and return it. If no index is specified, it will remove and return the last item in the list.	`list = [1, 2, 3, 4, 5]`
		`list.pop(2)`
		`print list`
		`list.pop()`
		`print list`
			`3`
			`[1, 2, 4`
			`5`
			`[1, 2, 4`

Function	Description	Example	Output

list.reverse()	Will reverse the items in the list.	`list = [1, 2, 3, 4, 5]`
		`list.reverse()`
		`print list`
			`[5, 4, 3`

Tuples

Tuples look similar to lists in that they are defined as a group of comma separated values, but they are enclosed by parenthesis and they are immutable like strings, meaning they can't be altered. Besides that, they are sequences, like lists and strings. This means that tuples have the functionality that other sequences have, such as concatenation, indexing and slicing, and even nesting.

Python - Tuples

```
# Tuples are very simple to create.
myTuple = (1, 2, 3)

print myTuple # Will print out: (1, 2, 3)

# Empty tuples can also be created, to have items added to them later.
myTuple = ()

# Like lists, tuples are not confined to hold values of a single data type either.
myTuple = (1, "two", 3.3)

# Tuples can even hold other tuples!
myTuple = (1, ("two", 3.3), 4, 'five', (6, (7.7, 'eight'), 9))

a = (1, 2, 3)
b = (4, 5, 6)

# Combine two tuples to make a new tuple
print a + b # Will print out: (1, 2, 3, 4, 5, 6)

myTuple = ('a', 'b', 'c', 'd', 'e', 'f', 'g')

print myTuple[2:5] # Will print out: ('c', 'd', 'e')
```

> **NOTE**
>
> Lists and tuples can also be nested within each other!

Tuple Functions

All of the functions that work on tuples work on lists, so these should look familiar. However, because they are immutable, not all of the list functions work on tuples. Check out the common tuple functions below.

Function	Description	Example	Output

len(tuple)	Returns the length of the **tuple**.	`tuple = [1, 2, 3, 4, 5]`
		
		`print len(tuple)`
			`5`

x in tuple	Will return **True** if **x** is within the tuple, **False** if not.
	
	Can also be used to iterate through the tuple.
		`tuple = [1, 2, 3, 4, 5]`
		
		`if 4 in tuple:`
		`  print "There is a 4 in the tuple"`
		
		`for value in tuple:`
		`  print value`
			`There is`
			`1`
			`2`
			`3`
			`4`
			`5`

min(tuple)	Returns the smallest item of **tuple**.	`tuple = [1, 2, 3, 4, 5]`
		
		`print min(tuple)`
			`1`

max(tuple)	Returns the largest item of **tuple**.	`tuple = [1, 2, 3, 4, 5]`
		
		`print max(tuple)`
			`5`

Dictionaries

Mapping Type

A Dictionary is a mapping object. Where sequences are indexed with a numeric index value, dictionaries are indexed using keys. These keys then have a matching value pair that is associated with a particular key. For example, with a list we can extract the object at index 0, which we may have decided is the name, whereas with dictionaries, I can instead extract the a value using a key "name". Because of how they work, dictionaries are sometimes known as associative arrays in other programming languages.

Dictionaries are created using braces `{ }`, where each key/value pair is separated by a comma (,) and keys are separated from their values using a colon (:). In the example below, I created a dictionary with two keys: name, id.

Python - Creating a Dictionary

```
# In this dictionary, I associated the name John Smith to the key "name",
# and the id number 12345 to the key "id".
myDictionary = {"name":"John Smith", "id":12345}
```

Using a Dictionary

The keys in a dictionary can be numbers, strings, or tuples, but typically a string is used to make a key that best describes the value. Any given key may only appear once in a dictionary, so trying to set another value for a key that already exists will overwrite the previous value for that key. Alternately, attempting to access the value of a key that does not exist will throw an error, while setting a value to a key that does not exist will create a new key/value pair within the dictionary.

To access a value in a dictionary works much like accessing a value in a list; simply place brackets containing the key after the dictionary object.

Python - Accessing Values in a Dictionary

```
# Creates a dictionary with three key/value pairs.
myDictionary = {'Bob': 89.9, 'Joe': 188.72, 'Sally': 21.44}

print myDictionary['Joe'] # Will print out: 188.72

# Adds a key for 'Amir', and alters the value associated with the key 'Sally'.
myDictionary['Amir'] = 45.89
myDictionary['Sally'] = 146.23

print myDictionary # Will print out the whole dictionary: {'Joe': 188.72, 'Amir':
45.89, 'Bob': 89.9, 'Sally': 146.23}
```

It is also easy to loop through all of the values of a dictionary using the keys() function. For example:

Python - Keys Function

```
# The keys() function provides us with a list of keys, which we can iterate
through and print out in addition to using in the value lookup.
for key in myDict.keys():
   print key, myDict[key]
```

There are many use cases for dictionaries, but they are commonly used in Ignition when passing values into a Message Handler or **creating a dynamic roster** for alarms.

Dictionary Functions

Dictionaries have a few functions that allow for greater control over the dictionary object and the values contained within.

Function	Description	Example

len(dictionary)	Returns the number of items in the dictionary.	`myDictionary = {"name":"John Smith", "id":12345}`
		
		`print len (myDictionary)`

del dictionary[key]	Will remove the named key.	`myDictionary = {"name":"John Smith", "id":12345}`
		
		`del myDictionary["id"]`
		
		`print myDictionary`

key in dictionary	Will return True if the dictionary has that key.
	
	Can also use "key not in dictionary"
		`myDictionary = {"name":"John Smith", "id":12345}`
		
		`if "name" in myDictionary:`
		`  print myDictionary["name"]`

dictionary.clear()	Remove all of the items in the dictionary.	`myDictionary = {"name":"John Smith", "id":12345}`
		
		`myDictionary.clear()`
		
		`print myDictionary`

dictionary.keys()	Returns a list of the dictionary's keys.	`myDictionary = {"name":"John Smith", "id":12345}`
		
		`print myDictionary.keys()`

Function	Description	Example
dictionary.values()	Returns a list of the dictionary's values.	`myDictionary = {"name":"John Smith", "id":12345}` `print myDictionary.values()`

Datasets

Overview

Datasets are used to manage tabular data and can be accessed like native Python objects. They support features such as indexing, iteration, and dataset-specific methods, making them simple to use in scripting. Datasets in Ignition can now be accessed directly for both reading and writing operations, eliminating the need for manual conversions.

Creating Datasets

Datasets can be created programmatically in Python using the **system.dataset.toDataset** function. This function requires two inputs: a list of headers and a list of data rows. Each data row must have the same number of elements as there are headers.

Creating a Dataset

```python
# First, create a list that contains the headers.
headers = ["City", "Population", "Timezone", "GMTOffset"]

# Create a list to house the data rows.
data = [
    ["New York", 8363710, "EST", -5],
    ["Los Angeles", 3833995, "PST", -8],
    ["Chicago", 2853114, "CST", -6],
    ["Houston", 2242193, "CST", -6],
    ["Phoenix", 1567924, "MST", -7]
]

# Use the headers and data to create a Dataset.
cities = system.dataset.toDataset(headers, data)
```

> **NOTE**
>
> All code snippets on this page will reference the cities dataset we created above, so place that code at the beginning of every code snippet.

Accessing Data in a Dataset

Datasets can be accessed using either dataset-specific methods or Python-style indexing and iteration. Below are examples of common ways to interact with dataset data.

The table below lists common methods available for interacting with datasets.

Function	Description	Example	Output
getColumnAsList(colIndex)	Returns a column as a list.	`cities.getColumnAsList(0)`	[New York, Los Angeles, Chicago, Houston, Phoenix]
getColumnCount()	Returns the number of columns.	`cities.getColumnCount()`	4
getColumnIndex(colName)	Returns the index of a column by name.	`cities.getColumnIndex("Timezone")`	2
getColumnName(colIndex)	Returns the name of a column by its index.	`cities.getColumnName(1)`	Population
getRowCount()	Returns the number of rows.	`cities.getRowCount()`	5
getValueAt(rowIndex, colIndex)	Gets the value at a specific row and column.	`cities.getValueAt(1, 2)`	PST

With Python-style access, datasets can be iterated over or indexed like lists, allowing for intuitive interaction with rows and columns.

Accessing Data

```
# Access a specific value directly using row and column indices.
print cities[1][2]  # Outputs "PST"

# You can also access a column by name if you iterate through rows.
for row in cities:
    print row["City"], row["Population"]
```

Iterating Through a Dataset

```
# Loop through all rows and columns in the dataset.
for row in cities:
    for value in row:
        print value
```

Altering a Dataset

Datasets in Ignition are immutable, meaning you cannot modify them directly. Instead, you create a new dataset by applying changes to the original dataset using functions like **system.dataset.addRow**, **system.dataset.deleteRow**, or **system.dataset.setValue**. The original dataset remains unchanged.

Modifying a Dataset

```
# Create a new dataset with the updated value.
newData = system.dataset.setValue(cities, 1, "Population", 5000000)

# The original dataset remains unchanged. Iterate over both datasets to compare.
print "Original Dataset:"
for row in cities:
    print row

print "\nUpdated Dataset:"
for row in newData:
    print row
```

Backward Compatibility

Scripts using the legacy **system.dataset.toPyDataSet** function will continue to function as expected. However, as of 8.3, datasets no longer require explicit conversions. You can directly access datasets

using indexing, iteration, and dataset methods.

For example, the following is no longer necessary:

Legacy approach

```
pyData = system.dataset.toPyDataSet(cities)
```

Instead, you can work directly with the dataset:

Modern approach

```
for row in cities:
    print row["City"], row["Population"]
```

Dates

Overview

Dates can normally be tricky since they generally require very specific formats. Furthermore, some functions/objects require a date object instead of a string. Fortunately, there are several ways to create and alter date objects with scripting in Ignition.

Python has some **built-in libraries** to create and manipulate dates and times. However, most users find both Ignition's built-in **system functions** and even Java's Calendar class easier to use. Regardless, this section will demonstrate some examples from each approach.

Ignition's System Functions

Ignition's **system.date** library has a large number of functions that provide easy access to datetime creation and manipulation. This page has just a few simple examples. Additional examples and functions can be found in the **scripting appendix**.

Creating Dates

New datetimes can be created by using either the **system.date.now** or **system.date.getDate**. The system.date.getDate function returns a datetime, but the time is set to midnight. However, we can use **system.date.setTime** to change the time.

Python - System Functions - Creating Dates

```
# Get the current datetime.
print system.date.now()

# Create a date. The time will be set to midnight.
newDate = system.date.getDate(2018, 10, 28)
print newDate

# Change the time on the new date to 11:30 am.
print system.date.setTime(newDate, 11, 30, 0)
```

Formatting Dates

When printed, datetimes default to a format like the following: **Sun Jan 1 00:00:00 TZ 2018**. However, this can be manipulated by using special characters in the **system.date.format** function:

Python - System Functions - Date Formatting

```
rightNow = system.date.now()

# Demonstrating the standard format.
print rightNow

# Demonstrating the modified format.
print system.date.format(rightNow, "yyyy-MM-dd HH:mm:ss")
```

Date Arithmetic

The **system.date.add*** functions can be used to add to or subtract some amount of time from a date. See the **system.date.add*** functions for more information.

Python - System Functions - Date Arithmetic

```
# Get the current datetime.
newDate = system.date.now()

# Change the time on the new date to 30 minutes ago.
print system.date.addMinutes(newDate, -30)
```

Date Formatting Characters

The following is a reference of date formatting characters that can be used by **system.date.format** or Java's DateFormat class. Additionally, there are many other non-scripting uses in Ignition (such as the **Vision - Calendar** component's Format String property) that can utilize this reference.

Symbol	Description	Presentation	Example	Other Notes
G	Era designator	**Text**	G=AD	
y	Year	**Year**	yyyy=1996; yy=96	Lowercase y is the most commonly used year symbol
Y	Week year	**Year**	YYYY=2009; YY=09	Capital Y gives the year based on weeks (ie. changes to the new year up to a week early)
M	Month in year	**Month**	MMMM=July; MMM=Jul; MM=07	
w	Week in year	**Number**	27	If Dec31 is mid-week, it will be in week 1 of the next year
W	Week in month	**Number**	2	
D	Day in year	**Number**	189	
d	Day in month	**Number**	10	
F	Day of week in month	**Number**	2	2nd Sunday of the month
E	Day name in week	**Text**	EEEE=Tuesday; E=Tue	
u	Day number of week	**Number**	1	(1 = Monday, ..., 7 = Sunday)

Symbol	Description	Presentation	Example	Other Notes
a	Am/Pm marker	**Text**	PM	
H	Hour in day (0-23)	**Number**	0	
h	Hour in am/pm (1-12)	**Number**	12	
k	Hour in day (1-24)	**Number**	24	
K	Hour in am/pm (0-11)	**Number**	0	
m	Minute in hour	**Number**	30	
s	Second in minute	**Number**	55	
S	Millisecond	**Number**	978	
z	Time zone	**General time zone**	zzzz=Pacific Standard Time; z=PST	
Z	Time zone	**RFC 822 time zone**	Z=-0800	
X	Time zone	**ISO 8601 time zone**	X=-08; XX=-0800; XXX=-08:00	

Java's Calendar Class

While Java's Calendar class if useful, in many cases Ignition's built-in **system.date** functions are simpler to use. Furthermore, the system.date functions typically use the Calendar class to retrieve the current time, so you are not losing any functionality by using the system functions.

> **IGNITION'S SYSTEM FUNCTIONS VS JAVA'S CALENDAR CLASS**
>
> It is highly advisable to use **Ignition's system functions** to generate and manipulate dates. The information on this page pertaining to the Calendar class is maintained in the interest for legacy installations.

Creating Dates

To create an arbitrary date, you can use the `java.util.Calendar` class. It has various functions to alter the calendar fields, like Calendar.HOUR, Calendar.MONTH, and so on. After you're done manipulating the Calendar, you can use its `getTime()` function to retrieve the Date represented by the calendar. It also has a handy `set()` function that takes the common parameters of a Date. The one major "gotcha" here is that January is month zero, not month one. For example:

Python - Calendar Class - Creating Dates

```python
from java.util import Calendar
cal = Calendar.getInstance()

# set year, month, day, hour, minute, second in one call
# This sets it to Feb 25th, 1:05:00 PM, 2010
cal.set(2010, 1, 25, 13, 5, 0)
myDate = cal.getTime()
```

Date Arithmetic

Often you'll have a Date object from a component like the **Popup Calendar** and want to alter it programmatically. Say, subtracting 8 hours from it, or something like that. The `java.util.Calendar` class is used for this as well. Following the example above, this code would subtract 8 hours from the variable `myDate`.

Python - Calendar Class - Date Arithmetic

```python

```

```
from java.util import Calendar
cal = Calendar.getInstance()
cal.setTime(myDate)
cal.add(Calendar.HOUR, -8)
myNewDate = cal.getTime()
```

Python's Time and Datetime Libraries

Many components in Ignition that contain a Date property actually expect a Java calendar object. Creating a datetime object using Python's built-in libraries and passing them to a date property on a component will result in an exception.

> **IGNITION'S SYSTEM FUNCTIONS VS PYTHON'S LIBRARIES**
>
> It is highly recommended to use **Ignition's built-in system.date** functions.

Creating Dates - Python's Time Library

The time library can be use to return dates as well as time. Times are created as a tuple of integers. The integers represent the following values: year, month, day of the month, hour, minute, second, weekday, day of the year, daylight savings time)

Check out **Python's time library documentation** for more information.

Python - Python Library - time

```
import time

# Finds the current local time. The time is returned as a tuple of integers.
myTime = time.localtime()

# Print the time into a 24-character string with the following format: Sun Nov 20
12:00:00 2017
print time.asctime(myTime)

# Alternatively, we can reformat the time in a custom manner, then print it
print time.strftime('%H:%M:%S %b %d %Y', myTime)
```

Creating Dates - Python's Datetime Library

Python's datetime library offers a bit more flexibility since arithmetic can easily be applied. Additional information on the datetime library can be found in **Python's official documentation**.

Notice the double use of `datetime` in the example below. This is because the `datetime` library has a class named `datetime`.

Python - Python Library - datetime

```
import datetime

# Returns the current datetime.
print datetime.datetime.now()
```

However, we can clean up the above by importing in the datetime class from the library:

Python - Python Library - datetime

```
# Imports the class named 'datetime' from the 'datetime' library, so we don't have to state it twice.
from datetime import datetime

# Returns the current datetime.
print datetime.now()
```

If you need to create a specific datetime, instead of just using the current, you can pass in the values directly when creating an instance of datetime:

Python - Python Library - Creating a New Time

```
from datetime import datetime

# Prints out the following datetime: 2018-01-02 03:04:05.000006
print datetime(2018,1,2,3,4,5,6)
```

Finding the difference between two datetime objects can easily be accomplished by using the '-' character

Python - Python Library - Date Difference

```
```

```
from datetime import datetime

rightNow = datetime.now()
someTime = datetime(2018,1,1,1,1,1,1)

# Find the difference between the two dates.
print someTime - rightNow
```

Date Arithmetic

With Python's built-in libraries, the timedelta class provides the simplest way to perform arithmetic on a date: It simply creates an object that effectively represents a duration. The duration can then be applied to a datetime.

Python - Python Library - Date Arithmetic

```
# We're including the timedelta class here
from datetime import datetime, timedelta

rightNow = datetime.now()

# Creating a timedelta object, and setting the hours to 8
offset = timedelta(hours = 8)

# Print the current time, and then print the time minus the offset.
print rightNow
print rightNow - offset
```

Conditions and Loops

If-Statements

The **if** statement should be familiar to anyone with a passing knowledge of programming. The idea of an **if** is that you want your script to execute a block of statements only when a certain condition is true. Python's **if** is simple to use, and has some additional keywords to provide more flexibility.

Simple If-Statement Example

The syntax for **if** is as follows:

Pseudocode - If Statement

```
# Note that 'if' uses lowercase characters.
# Additionally, a colon is placed after the expression.
if expression:

    # The statements that should execute when the expression is true
    # MUST be indented.
    statement
```

Example	Output

`x = 5`
`z = 15`
`if x < 10:`
`    # Since the condition "x < 10" is true,`
`    # the following line will execute`
`    print "'x' is less than 10"`
`if z < 10:`
`    # This condition "z < 10" is false,`
`    # so the following line will not execute`
`    print "this will never show"`
	'x' is less than 10

If and Else

You can use the **if...else** form of an **if** statement to do one thing if a condition is true, and something else if the condition is false.

Example	Output

`x = 15`
`if x < 10:`
`   print "x is less than 10"`
`else:`
`   print "x is not less than 10"`
	x is not less than 10

Elif (Else If)

Lastly, you can use the **if...elif** form. This form combines multiple condition checks. `elif` stands for `"else if"`. This form can optionally have a catch-all `else` clause at the end. For example, this script will print out three:

Example	Output

`x = 3`
`if x == 1:`
`  print "one"`
`elif x == 2:`
`  print "two"`
`elif x == 3:`
`  print "three"`
`else:`
`  print "not 1-3"`
	three

You can use as many elif items as you want, and the else is not required at the end.

For-Loops and While-Loops

For-Loop

Python's **for** loop may be a bit different than what you're used to if you've programmed any C. The **for** loop is specialized to iterate over the elements of any sequence, like a list. A **for** loop uses an iterator variable to reference each item as it steps through the sequence. This means it's very simple to write a loop!

Note that the syntax of the for-loop requires use of the in-keyword.

Pseudocode - For Loop

```
# In this example, "item" is a variable created specifically by the "for" loop to act as an iterator.
# The name "item" is not a keyword, and a different variable name may be used.
# Additionally, note that "for" and "in" are lowercase, and a colon is present at the end of the line.
for item in sequence:
    # All statements that should execute each iteration must be indented after the "for" statement
    statement
```

Example	Output

`listOfFruit = ['Apples', 'Oranges', 'Bananas']`
`for fruit in listOfFruit:`
`  print fruit`
	Apples
	Oranges
	Bananas

You don't need to manually create a sequence to repeat a task several times in a for loop. Instead, the built-in function **range()** function can generate a variable-size list of integers starting at zero. For example, calling **range(4)** will return the list [0, 1, 2, 3].

Example	Output

`# Even though this example isn't using the value of "x",`
`# the print statement will still be executed once for each item`
`# in the list returned by range().`
`for x in range(4):`
`  print "this will print 4 times"`
	this will print 4 times
	this will print 4 times
	this will print 4 times
	this will print 4 times

While-Loop

A while loop will repeat a block of statements as long as a condition is true. This code will print out the contents of the items in the list.

Pseudocode - While Loop

```
# A while loop simply needs the keyword "while", the condition that
# determines when we should stop iterating, and a colon at the end of the line.
while condition:
    # All statements that should be repeated each iteration must be indented after
the "while" statement
    statement.
```

This code uses a function called **len()**, which is a built-in function that returns the length of a sequence.

Example	Output

`listOfFruit = ['Apples', 'Oranges', 'Bananas']`
`x = 0`
`while x < len(listOfFruit):`
`  print listOfFruit[x]`
`  x = x + 1`
	Apples
	Oranges
	Bananas

The Break and Continue Statements

You can stop a loop from repeating in its tracks by using the **break** statement. This code will print out " Loop " exactly two times, and then print " Finished ".

Example	Output

`for x in range(10):`
`  if x >= 2:`
`  break`
`  print "Loop"`
`print "Finished"`
	Loop
	Loop
	Finished

You can use the **continue** statement to make a loop stop executing its current iteration and skip to the beginning of the next iteration. The following code will print out the numbers 0-9, skipping 4.

Example	Output

`for x in range(10):`
`  if x == 4:`
`  continue`
`  print x`
	0
	1
	2
	3
	5
	6
	7
	8
	9

Infinite Loops

It is incredibly easy to create an infinite loop when using a **while** statement. Depending where the infinite loop was created, it could cause you to lose your work in the Designer, or create a large amount of overhead on the Gateway.

Python - Infinite Loop Created by While Statement

```
x = 0
while x < 10:
    x += 1 # Forgetting to add a way to increment "x" will cause an infinite loop
    print x
```

In many cases, a **for** loop could be used instead of a **while**, but this is not always possible. When using **while**, the best way to avoid an infinite loop is to make sure you always have a way to exit the loop: a simple approach involves using a counter that can eventually trigger a **break** statement, or add the counter as an additional condition to the **while**.

Python - Preventing Infinite Loops Using the Break Keyword

```
###Example 1: using the break keyword
# The counter variable will be used as a guaranteed way out of the While.
counter = 0

# Normally, using True as a condition in a While would be a quick
# way to generate an infinite loop, but the counter helps prevent that
while (True):

    # Increase the counter
    counter += 1

    # Check the value of the counter. If it's at the point where we can assume
we're going to be looping indefinitely...
    if counter >= 1000:

        # Break out of the loop
        break
```

Python - Preventing Infinite Loops Using an Additional Condition

```
###Example 2: using an additional condition
# Again, the counter variable will be used as a guaranteed way out of the While.
counter = 0

# Instead of using nested logic, we can simply add counter's value as an
additional condition with "and"
while (True and counter < 1000):

    # Increase the counter. Once counter >= 1000, the while loop will be forced to
end.
    counter += 1
```

The Pass Keyword

When using conditional statements and loops, the pass keyword can be especially useful when writing a new script. When called, the pass keyword does nothing, which may seem useless. However it is great when you find yourself in a situation where you need a line of code to meet a syntax requirement, but don't want the code to do any additional work.

Python - Pass Keyword

```
```

```
# Read the tag value using readBlocking. Returns a list, so take [0].value.
myVar = system.tag.readBlocking([tagPath])[0].value

if myVar == 0:
    firstFunction()
elif myVar == 1:
    secondFunction()
elif myVar == 2:
    # I haven't implemented the thirdFunction() yet, so I can use pass here as a
placeholder
    pass
```

Error Handling

The concept of error handling is recognizing when an error might occur in a block of code, and instead of throwing the error, handling it gracefully. This can involve providing a personalized error message, letting a user know that their attempt to run the code failed, or even undoing something that the code set it motion so that it can be started again.

Understanding Try and Except Structure

Within Python, `try` blocks can be used to handle errors. The structure of a `try` block requires the inclusion of `except` blocks, and can be further expanded with the use of `else`, `pass`, or `finally` keywords. The **Error Handling Example** will gradually build out these elements to explain each usage.

But first, we'll demonstrate the basic structure of these blocks, where `try:` is followed by the code we want to test and `except:` is followed by the code to run if there is an error. The code will then execute in the following order:

1. The `try` block is run, and the code within is executed line by line.
    - If an error occurs, the code in the `try` block will immediately stop and the `except` block will begin to execute.
    - If no error occurs after all code in the `try` block has been executed, the `try` block will be finished and the code in the `except` block will be skipped.
2. After either outcome, any code after the `except` will then execute.

Pseudocode - Error Handling

```
# With try, we can attempt any amount of code.
try:
    some code
    more code
    even more code

# If any of lines in the try block were to throw an error, then we move down and
run the block under except.
# The except statement is NOT optional.
# You must define what your code should do in the event an exception occurs.
except:
    failover code
```

This execution order makes these blocks useful in situations that require user input, where something may have been incorrectly entered and resulted in an error.

Error Handling Example

We will use a division example to demonstrate how error handling works since we can easily cause errors. To begin, we will start with code that will print a value of 50, with no error in the division and the `try` block finishing successfully.

Python - Error Handling Division

```
# We start with a value, which could represent some user input.
x = 2

# We use that value in our try block, and have a simple print statement if there
is an error.
try:
    value = 100/x
    print value
except:
    print "An error occurred!"
```

The `except` block can be tested by changing the value of x to 0. In this case, the statement "An error occurred!" would print.

Although each `try` block must be followed by at least one `except` block, it is not limited to only one `except` block. Multiple `except` blocks make it possible to handle different types of errors separately. This is done by listing the **error object** after the `except` keyword and before the colon. The specific classes listed in your `except` clause are checked against the thrown exception, in declaration order, to see if they are a direct or superclass match.

For example, making an exception to describe dividing by zero as a specific error would look like the following:

Python - Exception-Specific Failover

```
# We start with a value, which could represent some user input.
x = 0

# Use the user input in division in our try block.
try:
    value = 100/x
    print value

# If the exception that would occur is of type ZeroDivisionError, we can run a specific block of code
except ZeroDivisionError:
    print "Dividing by zero is not allowed. Please stop trying to divide by zero"
# We can then have a second exception without a specific error. This except acts as a catch-all;
# if the user caused an exception we didn't account for above, we can failover to the block of code below.
except:
    print "An error occurred!"
```

The `except` blocks in the code above cover all possible errors as represented in the table below. Now, we have a more tailored approach to how we handle errors while still catching all that may occur.

Inputs (x value)	Result	Output
2	Successful Result	50
0	Specified Zero Division Error	Dividing by zero is not allowed. Please stop trying to divide by zero
'a'	Unspecified Error	An error occurred!

In addition to each `try` block having more than one `except` block, each `except` block can also name multiple types of errors. Keep in mind that an error occuring within a `try` block will only ever trigger a single `except` block.

Pseudocode - Try Block with Except Blocks

```
try:
    some code
except (ZeroDivisionError, RuntimeError, TypeError):
    failover code
```

Determining the Error Object

To determine the name of the error object that will be thrown from certain errors, we can take a look at the error. We already know dividing by zero gives a `ZeroDivisionError`, but we can force another error to display a different error object. Dividing by a string of `a` without including any error handling would instead show the following error:

Traceback (most recent call last):
File "buffer", line 3, in module
TypeError: unsupported operand type(s) for /: 'int' and 'str'

The last line of the error includes the name `TypeError` followed by the cause of that error, which makes sense as the string `a` is the wrong type to be using in division.

Not all errors are so straightforward, and may require a bit more effort to discover. For a list of Python error names see the **Built-in Exceptions** page in the Python docs. Additionally, because Ignition is written in Java, many methods, including built-in system library functions, may throw **Java exceptions**. In these cases, Oracle's documentation on the base **Throwable class** is more useful.

Additional Keywords

Pass

The `pass` keyword is unique in that it does nothing but fill in a spot where code is required. This is useful if we want to handle the exception so that it doesn't throw an error message, or any other type of response. In this case, we can use pass to tell the script to continue with no further action.

Pseudocode - Pass Keyword

```
try:
    some code
    more code
except:
    # The error will bring the code to the exception, and then the exception will
simply do nothing.
    pass
```

Else

The `else` keyword allows you execute code when there was no error.

Pseudocode - Else Keyword

```
try:
    some code
    more code
except:
    print "An error occurred!"
else:
    print "All good here!"
```

Finally

The `finally` keyword offers a way to wrap up the code execution cleanly, regardless of if an error existed or not. In this example, whether or not "An error occurred!" is printed, "the Finally Keyword code example is complete." will also print out, confirming that the `try` block has completed all execution steps. This is most useful in situations where you need to do some "cleanup" task, whether your code succeeded or threw an error.

Pseudocode - Finally Keyword

```
try:
    some code
    more code
except:
    print "An error occurred!"
finally:
    print "The Finally Keyword code example is complete."
```

Java Exceptions

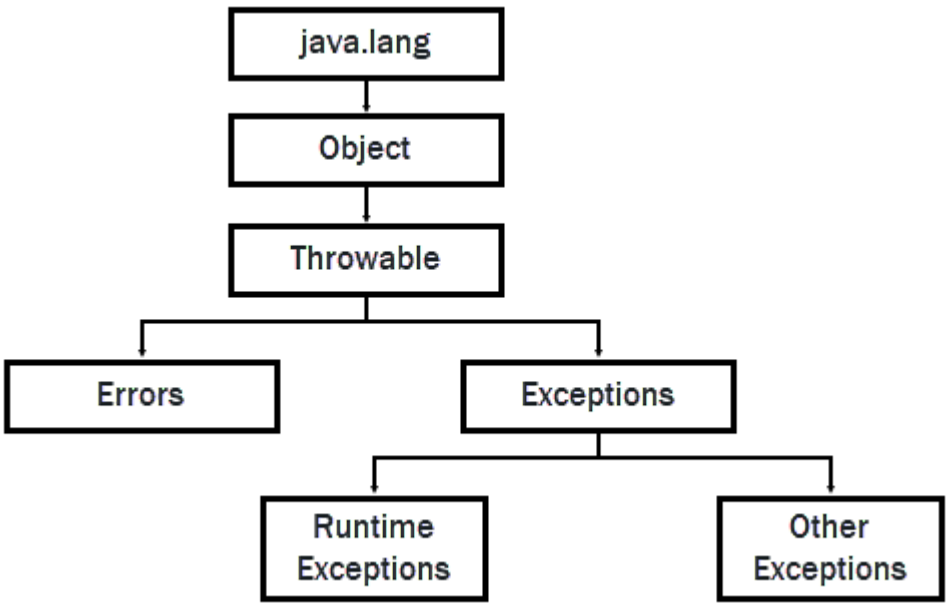
Although scripting in Ignition is done in Python, system library functions may also throw Java exceptions, since Ignition is written in Java. The Python scripts demonstrated above will not catch these, because Jython models Java exceptions as an entirely separate exception hierarchy. Importing Java's Exception class with `java.lang.Exception` will catch most exceptions thrown in Java code, excluding only "unrecoverable" errors such as `OutOfMemoryError`.

Pseudocode - Exception Class

```
```

```
# Import the Exception class
import java.lang.Exception

# With try, we can attempt any amount of code
try:
    code you want to test
    more code
    code that will throw an exception

# If any of lines in the try block were to throw an error, then we move down and
run the block under except.
# The except statement is NOT optional
# You must define what your code should do in the event an exception occurs.
except java.lang.Exception as e:
    failover code
```

Ultimately, all Java exceptions and errors inherit from the Throwable class, so you can choose to intercept even normally unrecoverable errors such as `StackOverflowError` or `OutOfMemoryError` by widening your `except` clause to `Throwable`.



Pseudocode - Throwable Class

```
from java.lang import Throwable

try:
    code you want to test
    more code
    code that will throw an exception
except Throwable as t:
    failover code
except Exception as e:
    failover code
```

Common Exceptions

Some common exceptions are listed below:

- **ArrayIndexOutOfBoundsException**: This typically occurs when a line of code attempts to access an index in a collection, but the index specified doesn't exist.

  ```
  myDataset = system.dataset.toDataSet(["colName"],[[0]])

  # This will fail because the dataset only has a single row,
  # so trying to read the value at row index 5 means we're trying to
  # read something that doesn't exist.
  print myDataset.getValueAt(0,5)
  ```

- **AttributeError**: Typically seen when a script tries to access a property that doesn't exist. For example, when a script tries to get the value of a property on a component, but the property name is misspelled.

  ```
  myVar = 10

  # integers do not natively have a name variable, so this will fail with an
  AttributeError.
  # there isn't an 'attribute' on an integer by the name of 'name'
  print myVar.name
  ```

- **IndexError**: Similar to ArrayIndexOutOfBoundsException above, but occurs with a Python specific object, such as a list.

```
myList = [1,2,3]

# There isn't an element in the list at index 4, so this will fail.
print myList[4]
```

- **NameError**: Occurs when the local or global name is not found. Typically happens when referencing a variable that hasn't been assigned a value.

```
# We haven't given a value to the variable myValue, so it will fail.
print "The value of the variable is: ", myValue
```

- **TypeError**: A TypeError occurs when a function or similar operation is applied to another object of an incorrect type.

```
myList = [1,2,3]

# The first argument for pop() expects an integer, so passing a string value
is innappropriate.
# Passing a string to pop() will return a TypeError.
print myList.pop("0")</pre>
```

- **ValueError**: A ValueError is returned when the value of an argument is inappropriate. Typically the exact issue is returned in the returned exception.

```
myVar = "Hello"

# Strings can be passed to the int() function, but the value needs to be
something that
# can be coerced into an integer, like "0".
# Because "Hello" can't be easily converted, the line below will fail.
print int(myVar)
```

Example

In the complete example below, you can see how all these possible variants of Python's `try` and `except` mechanism can be used in combination with Ignition's system functions to offer robust error handling for your code.

Pseudocode - Java Exceptions Example

```
from java.lang import Throwable

__log = system.util.getLogger("myTransactionUtil")

def doSomethingWithTheDatabase(args):
	tx = system.db.beginTransaction("myDatabase")

	try:
		return system.db.runPrepUpdate("query", [args], tx=tx)
	except Exception as e:
		# roll back the transaction
		system.db.rollbackTransaction(tx)
		__log.warnf("Update failed due to Jython Exception: %s", e)
	except Throwable as t:
		# roll back the transaction
		system.db.rollbackTransaction(tx)
		# full Java exception stacktrace is captured in logs
		__log.warn("Update failed due to Java Throwable", t)
	else:
		# no exceptions were thrown - commit the transaction
		system.db.commitTransaction(tx)
	finally:
		# whether an exception was thrown or not, we need to clean up and
close the transaction
		system.db.closeTransaction(tx)
```

Built-In Functions

Python Built-In Functions

Functions are code that can be called repeatedly from other places. Functions can have parameters passed into them and may return a resulting value. Some functions, like len(), are built-in. Some functions, like system.vision.showMessage(), are part of the **scripting libraries** provided by Ignition. Some functions, like math.sqrt(), are provided by the Python Standard Library.

Functions are invoked by using their name followed by an argument list surrounded in parentheses. If there are no arguments, you still need an open and close parenthesis.

This section details several useful Built-in Functions, along with some simple examples. See the **official docs** for more information.

Type Casting Functions

Python has many functions to convert between data types. Some common type casting functions are listed below.

Function	Notes	Example	Ou

bool()	When casting a numeric value to a boolean, a zero value is false, while all non-zero numbers are True.
	
	When casting a String or Unicode value to a boolean, an empty string is False, any other string is True.
		`# Results in False`
		`print bool("")`
		
		`# Results in True`
		`print bool("Test")`
			Fal
			Tru

int() and long()	When casting a float, rounding will not occur automatically to the decimal value. Instead, the round() function should be called.
	
	When casting a String or Unicode value, the string literal needs to be a
		`# Float to Integer`
		`print int(123.8)`
		
		`# Float to Long`
		`print long(321.8)`
		
		`# String to Integer`
		`print int("400")`
		
		`# ValueError: the value is not base 10`
		`print int("400.5")`
			123
			321
			400
			Val

Function	Notes	Example	Ou
	valid integer or long: decimal values contained in the string will result in a ValueError. Integers have at least 32 bits of precision, while Longs have unlimited precision.		

float()	When casting a string literal as a float, non-numeric characters in the string will result in an exception, except for a decimal point (".").	`# Integer to Float`
		`print float(123)`
		
		`# String to Float`
		`print float("400.5")`
			123
			400

str() and unicode()	Most objects can be cast as a string representation of some sort, including sequences.	`print "First line:" + str(80)`
		
		`# Even sequences can be cast as a string,`
		`# making for easy concatenation`
		`myList = [1,2,3]`
		`print str(myList)`
			80
			[1,

Checking an Object's Type

Checking the data type of an object can easily be done with both the `type()` and `isinstance()` functions.

Function	Notes	Example	Output

type(object)	When passed a single parameter, this function returns the type of the `object`.	`var = 10`
		`print type(var)`
		`print type(str(var))`
			`<type 'i`
			`<type 'st`

isinstance(object, classinfo)	Returns True if the `object` is an instance or subclass of `classinfo`, otherwise, returns false.
	
	If checking for a string or unicode type, a `classinfo` of "basestring", which is the base class for both strings and unicode types, would return True.
		`var = 10`
		`print isinstance(var,int)`
		
		`strVar = "string"`
		`print isinstance(strVar,basestring)`
			True
			True

Python - Type Validation: type vs isinstance

```
## type() Example
# This example attempts to validate the type of a variable. As written, this will
evaluate as True,
# and thus the print statement would execute.
var = "My String"
if type(var) == type(""):
    print "Variable 'var' is a string"

## isinstance() Example
# The isinstance() function can offer the same functionality as above.
var = "My String"
if isinstance(var, str):
# Note the lack of quotation marks around the classinfo parameter.
# We want to reference the class str, not the string "str".
    print "Variable 'var' is a string"
```

Generating a Range of Values

In some cases, it is useful to generate a range of integers for iteration. Python's `range()` function will return a list of integers.

Function	Notes	Example	Output

range([start,] stop[, step])	Returns a list of progressively greater integers.
	
	`start` - Integer value denoting the initial value in the list. If omitted, defaults to 0. This parameter is inclusive.
	
	`stop` - Integer value, determines when to cease generating integers. This parameter is exclusive.
	
	`step` - Integer value to increment each new integer by. If omitted, step defaults to 1.
	
	If `step` is positive, integers will
		`print range(5)`
		`print range(1, 5)`
		`print range(1, 10, 3)`
		`print range(15, 0, -3)`
			`[0, 1, 2, 3, 4]`
			`[1, 2, 3, 4]`
			`[1, 4, 7]`
			`[15, 12, 9, 6, 3]`

Function	Notes	Example	Output
	be generated as long as (`start + i * step < stop`) is true. If `step` is negative, integers will be generated as long as (`start + i * step > stop`) is true.		

Assume we need to read from five separate Tags with a nearly identical Tag path in a single script:

Pseudocode - Tag Path

```
[Provider]Folder/Sub_Folder_1/Tag
[Provider]Folder/Sub_Folder_2/Tag
[Provider]Folder/Sub_Folder_3/Tag
[Provider]Folder/Sub_Folder_4/Tag
[Provider]Folder/Sub_Folder_5/Tag
```

Instead of manually typing each path, we could use `range()` in a for loop that would write the paths automatically.

Python - Range in a For Loop

```

```

```
# Initialize an empty list that will ultimately hold all the Tag paths.
tagPaths = []

# Use range to repeatedly append 5 tag paths to the tagPaths list: starting with a
value of 1,
# and ending with a value of 5.
for num in range(1, 6):
    # Use String Formatting to create a Tag path with the iterator's (num) value.
    tagPaths.append("[Provider]Folder/Sub_Folder_%i/Tag" % num)

# Now that tagPaths contains all our tag paths, we can use the list to interact
with the tags
# by reading their values simultaneously.
results = system.tag.readBlocking(tagPaths)
tagValues = [qv.value for qv in results]
```

Rounding Numbers

You can round numbers inside Python with a few simple functions.

Function	Notes	Example	Output

round(number[, digits])	When passed a single parameter, this function returns a rounded integer value of a number. If the decimal is greater than or equal to .5, the it rounds up, less than .5 rounds down.
	
	If the optional digits argument is used, then it rounds to that many decimal places.
		`var = 10.236981`
		`print round(var)`
		`print round(var,3)`
			10
			10.237

math.floor(number)	Returns a truncated integer from the number given. The largest integer value less than or equal to *number*.
	
	Note that the example needs to **import that math library** before being able to call floor().
		`import math`
		
		`var = 100.938`
		`print math.floor(var)`
			100.0

math.ceil(number)	Returns the ceiling integer from the number given. The smallest integer value greater than or equal to *number*.
	
	Note that the example needs to **import that math library** before being able to call ceil().
		`import math`
		
		`var = 100.138`
		`print math.ceil(var)`
			101.0

Python - Simple Casting

```
stringVar = "40"

# Without using int(), this line would cause an exception. However int() is able to cast the
# type of stringVar's value to an integer.
print 20 + int(stringVar)

# Type casting is also useful in cases where a string parameter is required,
# but a numerical value should be given, such as the message parameter in system.vision.showMessage().
intVar = 60
# Note that this could also be accomplished with String Formatting instead of using str().
system.vision.showMessage(str(intVar))
```

Libraries

The System Library

Ignition comes with a group of system functions, called the System Library. Using a system function is simple. For example, the following code will access the value of a Tag.

Pseudocode - Reading a Tag Value

```
# system.tag.readBlocking returns a list of QualifiedValue objects.
# Take the first element [0] to get the value.
value = system.tag.readBlocking(["tagPath"])[0].value
```

The **scripting appendix** is full of built-in functions such as this.

Python Libraries

Python Libraries are packages of extra functions that expand the functionality of the code and can be imported into a script. We do this by using the import keyword:

Pseudocode - Import a Library

```
# This pseudo code will import a library, and then call a function of that
library.
import myLibrary

myLibrary.specialFunction()
```

The import keyword imports that entire library and allows you to use all of the functions inside of it by calling them off of the imported library. You can also import a piece of a library:

Pseudocode - Import a Function of a Libray

```
# This pseudo code will import a function from a library, and then call that function.
from myLibrary import specialFunction

specialFunction()
```

Note, that since we are directly importing in the function, we can directly call it instead of having to call it off of the library.

Python Standard Library

Python has an extensive standard library that provides a host of new functionality to the scripting language. The python documentation goes over all of the libraries in its standard library as well as how to use them here: **https://docs.python.org/2/library/index.html**

Let's take a look at an example of using a common library:

Python - Accessing Files in a Python Standard Library

```
# The csv library provides an easy way to read csv files, regardless of how they are formatted.
import csv

# We first grab our filepath, and feed it into the open function, which opens the file.
filepath = "C:\\test.csv"
csvFile = open(filepath, 'r')

# We then pass our opened csv file object into the csv.reader function, which will read the file.
# This can be looped through in a for loop to print every row of the csv.
reader = csv.reader(csvFile)
for row in reader:
    print row
```

Importing 3rd Party Libraries

In addition to the standard libraries, 3rd party libraries can also be imported into Ignition's scripting environment. A Python Library or individual module file will consist of a python file (.py) that contains the code that implements the functions of the library. You can often find python libraries built by other

users on the web, or can even create your own. These files can then be placed into a folder within your Ignition server.

Windows MacOS Linux

`C:\Program Files\Inductive Automation\Ignition\user-lib\pylib`

Once the python file is in that folder, you can then import the library into a script just like any of the standard libraries.

> **CAUTION**
>
> Ignition uses Python version 2.7. This means that any imported libraries must be compatible with Python 2.7.

Accessing Java

Scripting in Ignition executes in the java based implementation of Python called Jython. (See **Python or Jython?**). While this doesn't have any great effect on the Python language itself, one of the great side benefits is that your Python code can seamlessly interact with Java code as if it were Python code. This means that your Python code has access to the entire Java standard library.

To use Java classes, you simply import them as if they were Python modules. For example, the following code will print out all of the files in the user's home directory. This code uses the Java classes java.lang.System and java.io.File to look up the user's home directory and to list the files. Notice that we can even use the Python-style for loop to iterate over a Java sequence.

Python - Accessing Java

```python
# Importing the appropriate java libraries.
from java.lang import System
from java.io import File

# Used to look up the files in the users home directory.
homePath = System.getProperty("user.home")
homeDir = File(homePath)

# Loops through the list of files and prints them.
for filename in homeDir.list():
   print filename
```

You can find the reference documentation for the Java standard class library (also known as, the "JavaDocs") at: **http://docs.oracle.com/javase/8/docs/api/**

Subclassing Java

You can also create Python classes that implement Java interfaces. You do need some understanding of Java and object-oriented programming concepts, which are outside the scope of this manual. To create a Python class that implements a Java interface, you simply use the interface as a superclass for your Python class. For example, we could augment the example above to use the overload **java.io.File.list(FilenameFilter)**. To do this, we'll need to create a `FilenameFilter`, which is an interface in Java that defines a single function:

`boolean accept(File dir, String name)`

To implement this interface, we create a Python class that has `java.io.FilenameFilter` as its superclass, and implements that Java-style function in a Python-esque way.

Python - Implementing Java Interfaces

```
# Importing the appropriate java libraries.
from java.lang import System
from java.io import *

# This sets up an extension filter that can check the file extension. Txt is the
default.
class ExtensionFilter(FilenameFilter):
   def __init__(self, extension=".txt"):
      self.extension=extension.lower()

   def accept(self, directory, name):
      # make sure that the filename ends in the right extension
      return name.lower().endswith(self.extension)

# Used to look up the files in the users home directory.
homePath = System.getProperty("user.home")
homeDir = File(homePath)

# Prints out all .txt files. Txt is provided if nothing is specified.
for filename in homeDir.list(ExtensionFilter()):
   print filename

# Prints out all .pdf files.
for filename in homeDir.list(ExtensionFilter(".pdf")):
   print filename
```

User Defined Functions

Functions

A function is code that can be called repeatedly from other places. Functions can have parameters passed into them, and may return a resulting value. Some functions, like `len`, are built-in. Some functions, like `system.vision.showMessage()`, are part of the **scripting libraries** provided by Ignition. Some functions, like `math.sqrt()`, are provided by the **Python standard library**. However, functions can also be defined in a script that can be used later on in the script. In these user defined functions, you give the function a name and some code that will run when the function is called. Then later on in the script, you can call the function by its name and it will run the code specified in the function. This is useful, because it allows you to run a segment of code many times without having to repeat it within the script.

Functions are invoked by using their name followed by an argument list surrounded in parentheses. If there are no arguments, you still need an open and close parenthesis.

Defining Functions

Functions are defined using the def keyword. A function needs a name and a list of the arguments that it can be passed. For example, this code defines a function that prints "Hello World!".

Python - Defining a Function

```
# First we define our function.
def printHW():
   print "Hello World!"

# We can then call our function.
printHW()
```

Functions Arguments

When a function accepts arguments, the names of those arguments become variables in the function's namespace. Whatever value was passed to the function when it was invoked becomes the value of those variables. Arguments can have default values, which makes them optional. If an argument is omitted, then its default value will be used. The following code defines a function called **cap**, which will check if a number is within an upper and lower limit. The default lower limit is 0, and the default upper limit is 100.

Python - Defining Arugments

```
# We first define our function. Notice that we have 3 different arguments.
# x, min, and max. The min and the max are set to equal 0 and 100 respectively.
def cap(x, min=0, max=100):

    # Check if x is less than the min, return the min if true.
    if x < min:
        return min

    # Check if x is greater than the max, return the max if true.
    elif x > max:
        return max

    # Return the value if it is within the bounds.
    else:
        return x

# We can then see the outcome by running our function with a few different
parameters.
# This will print out a 40, since it is within the bounds.
print cap(40)

# This will print out "0", since it is less than the min of 0.
print cap(-1)

# This will print out "100", since it is greater than the max of 100.
print cap(150)

# This will print out "150", because it uses a max of 200 instead of the default
100.
print cap(150, 0, 200)
```

Keyword Arguments

In Ignition, some complicated script functions are designed to take keyword arguments instead of normal parameters. In the description for those functions, you may see the following info box in this User Manual:

> **TIP**
>
> This function accepts keyword arguments.

Arguments can also be specified by keyword instead of by position. In the example above, the only way someone would know that the 200 in the last call to cap specified the max is by its position. This can lead to hard-to-read function invocations for functions with lots of optional arguments. You can use keyword-style invocation to improve readability. The following code is equivalent to the last line above, using 200 for the max and the default for the `min`.

Python - Using Keyword Arguments

```
print cap(150, max=200)
```

Because we used a keyword to specify that 200 was the max, we were able to omit the min argument altogether, using its default. However, using a keyword argument before a non-keyword or positional argument is not allowed.

Python - Non-keyword Argument

```
# This would fail, because the function isn't sure what 150 is being used for.
print cap(max=200, 150)
```

Functions Are Objects

Perhaps one of the most foreign concepts for new Python users is that in Python, functions are first-class objects. This means that functions can be passed around to other functions (this concept is similar to the idea of function pointers in C or C++).

Suppose we wanted a general way to filter a list. Maybe sometimes we want the odd entries, while other times we want even ones. We can define a function called extract that takes a list and another function, and returns only entries that "pass" through the other function.

Python - Functions Passed to Other Functions

```python
# We define a function that checks if the value passed in is odd.
def isOdd(num):
    return num % 2 == 1

# We define a function that checks if the value passed in is even.
def isEven(num):
    return num % 2 == 0

# We define a function that inserts our list into the appropriate function and
returns valid values.
def extract(filterFunction, list):
    newList = []
    for entry in list:
        if filterFunction(entry):
            newList.append(entry)
    return newList

# Prints out [0, 2, 4, 6, 8]
# Notice that isEven as not invoked, but passed to the filter function.
print extract(isEven, range(10))
```

Where Can Functions Be Defined

User Defined Functions can be defined anywhere that a script is used. As stated before, they are useful to run segments of code multiple times without having to repeat it. They are also used extensively in **project scripts** where multiple functions can be defined in a single script module. Finally, some special Ignition System functions like **system.vision.createPopupMenu** or the **runScript** Expression function use functions as arguments.

Function Scope

The concept of scope is very important in all programming, and Python is no exception. Scope defines what names are directly accessible without any qualifiers. Another way to put this is that the scope determines what variables are defined. In Python, a variable is defined at the time that it is assigned. What scope it belongs to is also defined by where the assignment occurs.

Pseudocode - Defining a Function for Scope

```
```

```
# On this line, there is no variable 'x' in scope.
doSomeWork()

# Now 'x' is defined in our scope, because we've assigned a value to it
x = 5

# This will work because x is in scope.
print x
```

When you define a function, that function gets its own scope. Variables that are assigned within that function body will not be available outside of the function.

Python - Variables Defined within a Function Not Available Outside Scope

```
# x is local to myFunction() because this is where it is defined.
def myFunction():
   x = 15
   print x

# This will fail, because x is not available in the outer scope
y = x + 10
```

Scripting in Ignition

Where Is Scripting Used?

Python is used in many places in Ignition. Each location has its own events that trigger your scripts to run, and add functionality to your projects in different ways. The most apparent place is in **event handlers** on components and other objects in Vision Clients and Perspective Sessions.

Script Scope

One important thing to keep in mind before scripting in Ignition, is to understand the concept of scope.

Within Ignition, there are different scopes:

- Gateway Scope - The script runs on the Gateway. Scripts running in this scope cannot interact with components in the other two scopes.
- Perspective Session Scope - The script runs as a part of a Perspective Session. Note that scripts in Perspective execute on the Gateway, not in the browser, but this scope is still distinct from the Gateway scope.
- Vision Client Scope - The script runs inside of an instance of a Vision Client.

Where a script was written determines which scope it executes in. For example, tags are in the Gateway scope, so **Tag Event Scripts** execute in the Gateway scope.

This means that the script will not be able to access any client level resources such as windows or components that you may have open in the Client. Additionally, some of the system functions like **system.vision.showError** only work in the Client scope, so you will not be able to use them in the script on the tag.

System Functions, Hints, and Autocomplete

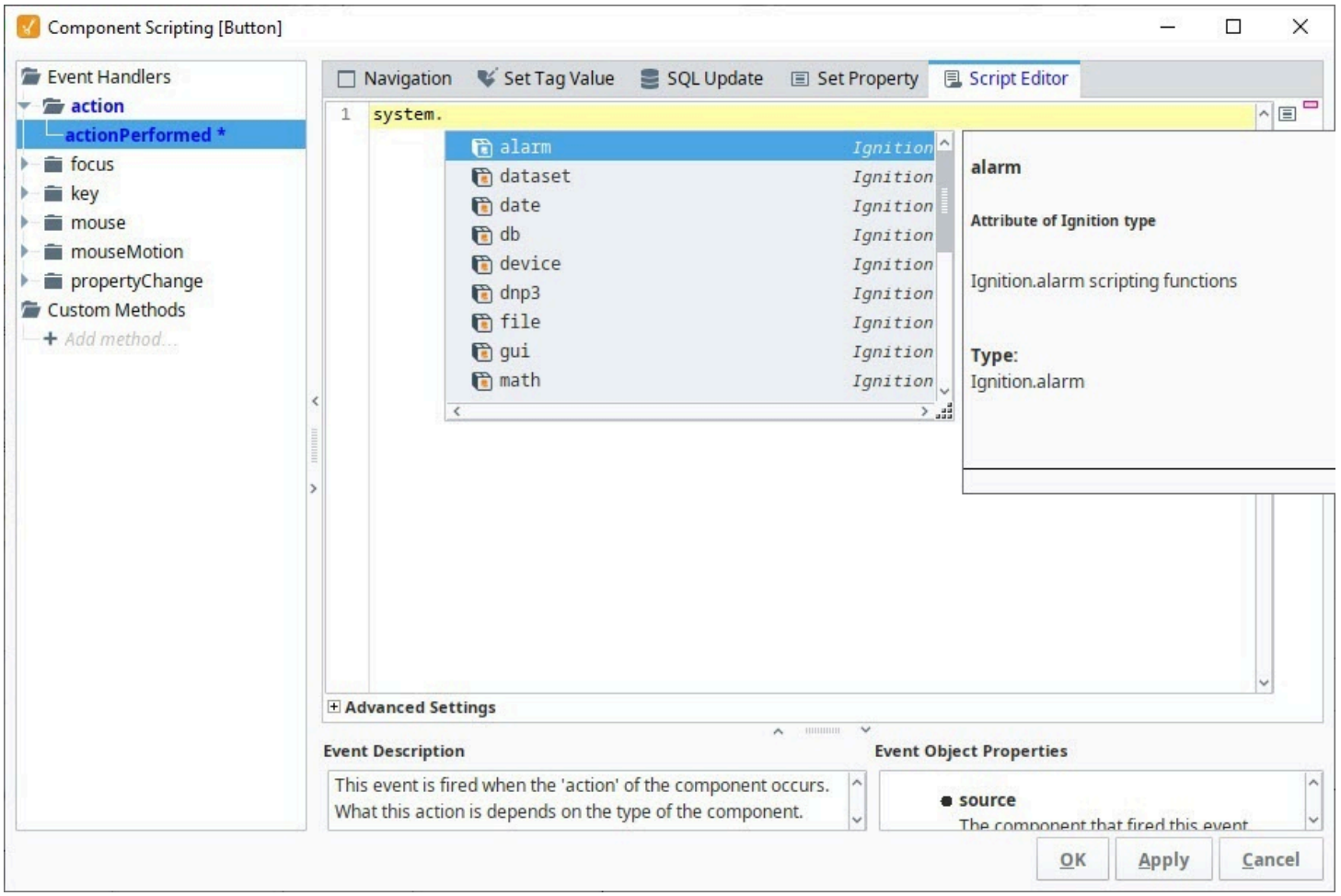
Ignition comes with a group of system functions, which are built-in functions that interact with Ignition features.

Python - Simple Script Using a System Function

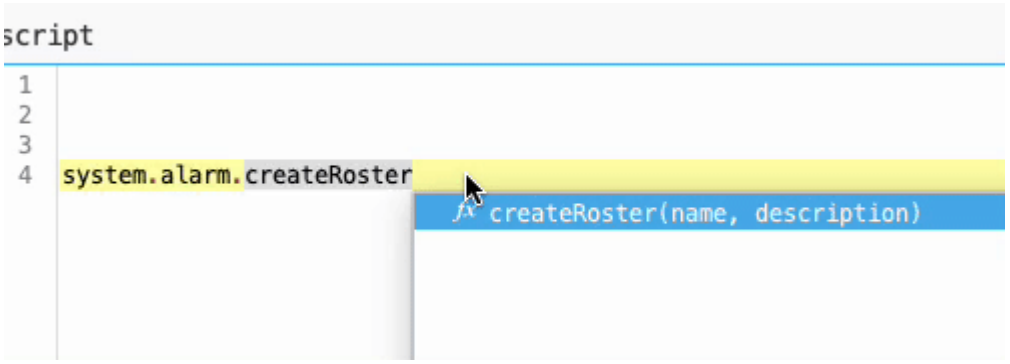
```python
value = system.tag.readBlocking(["tagPath"]).value
```

A complete list of these functions (with their definitions) is available from the autocomplete popup. The autocomplete popup is enabled by default and will appear wherever you can add a script after typing `system.`. You can manually open the popup with **Ctrl+Space** after you've at least typed `system`. As you keep typing, the list will automatically narrow down. Additionally, the **System Functions** page in the appendix contains complete documentation for the built-in system functions.

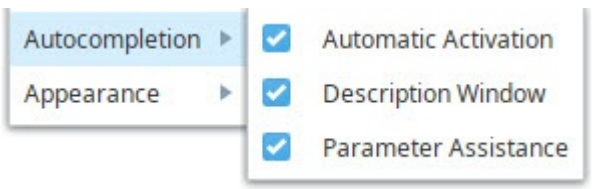
> **NOTE**
>
> The autocomplete popup shows all system functions scoped to the current script. If a system function does not appear in the list, that means the function is not available in the current scope.



Additionally, the editor offers parameter completion assistance. If you auto-complete a method with multiple required parameters, you'll automatically enter a Parameter Assistance mode, where you can tab through the parameters and enter them one at a time:

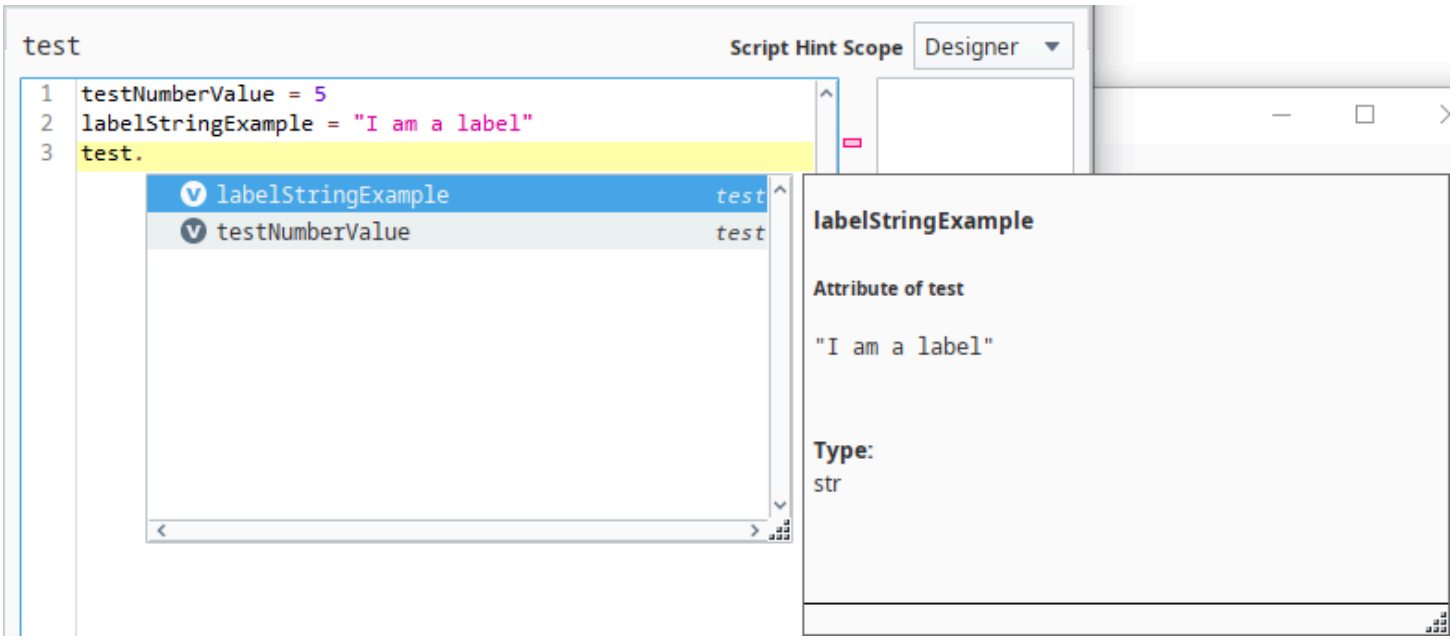


To disable these features, right-click anywhere within the Script Editor window and deselect **Automatic Activation** and/or **Parameter Assistance**.



Autocomplete hints are also displayed for code other than Ignition's system functions. If Ignition detects a function or project script, a popup will automatically appear, from which you can select which

function or project script you are trying to reference. You can also bring up this popup by pressing **Ctrl+Space**.

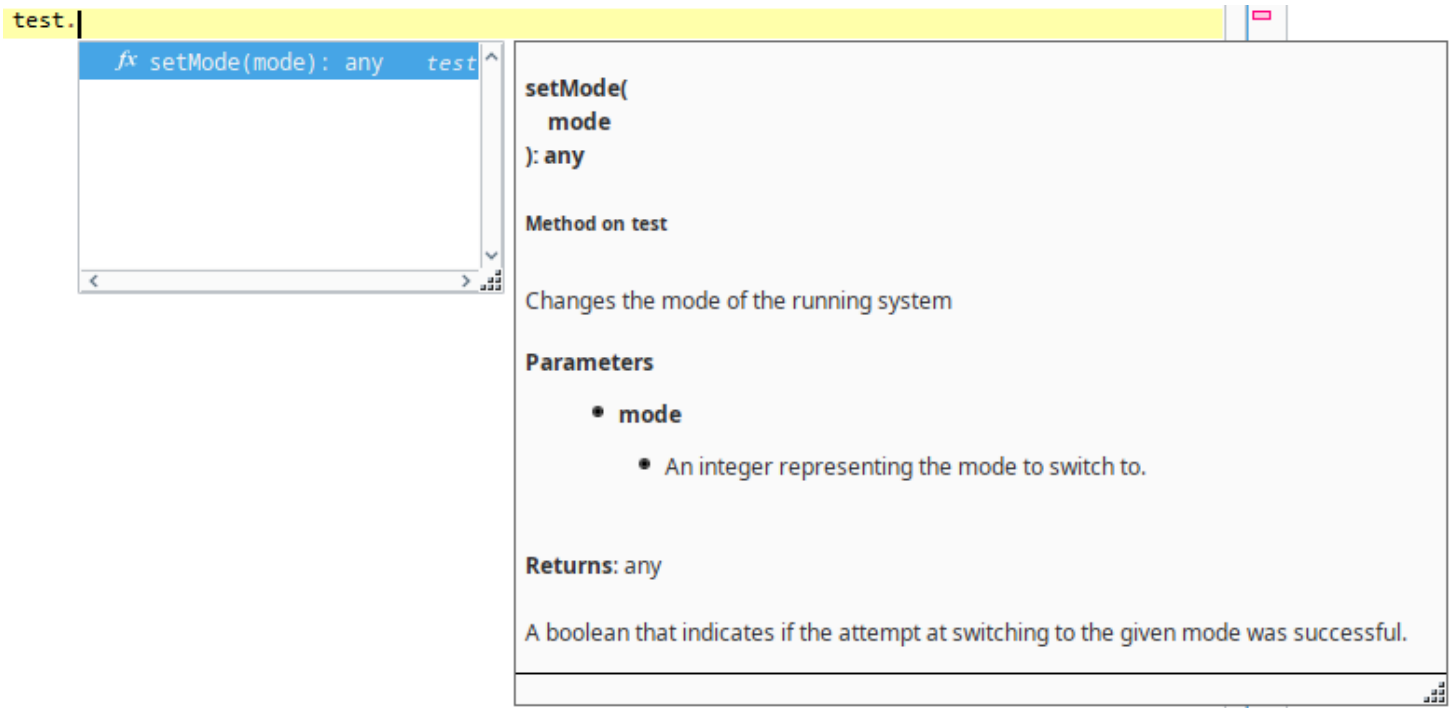


Autocomplete hints will extract method parameters, return information, and limited type awareness for project script functions and class docstrings written in **Google Python Style Guide's** docstring format. An example of the expected format is as follows:

Code Snippet

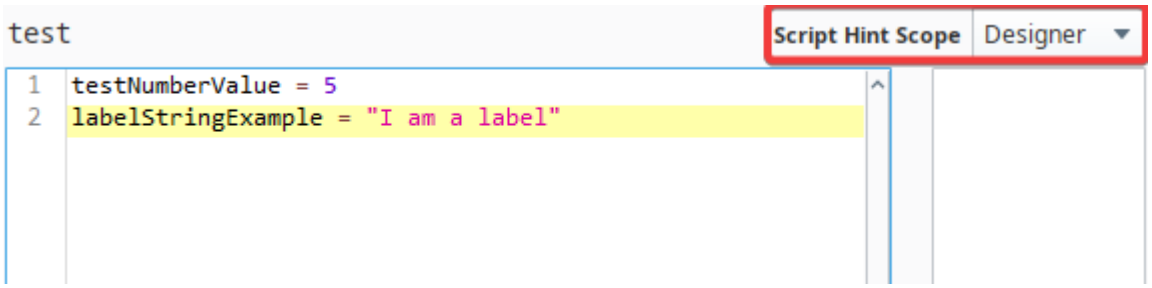
```
def setMode(mode):
    """Changes the mode of the running system

    Args:
        mode: An integer representing the mode to switch to.

    Returns:
        A boolean that indicates if the attempt at switching to the given mode was
successful.

    Raises:
        Error: If communications are down, an exception will be thrown.
    """
    # Function code goes here
```

The code above will display the description, parameter, and return information when autocomplete hints render the docstring:



Script Hint Scope

You can choose the scope of your scripting hints and suggestions come from by selecting the dropdown menu in your Project Library's scripts:

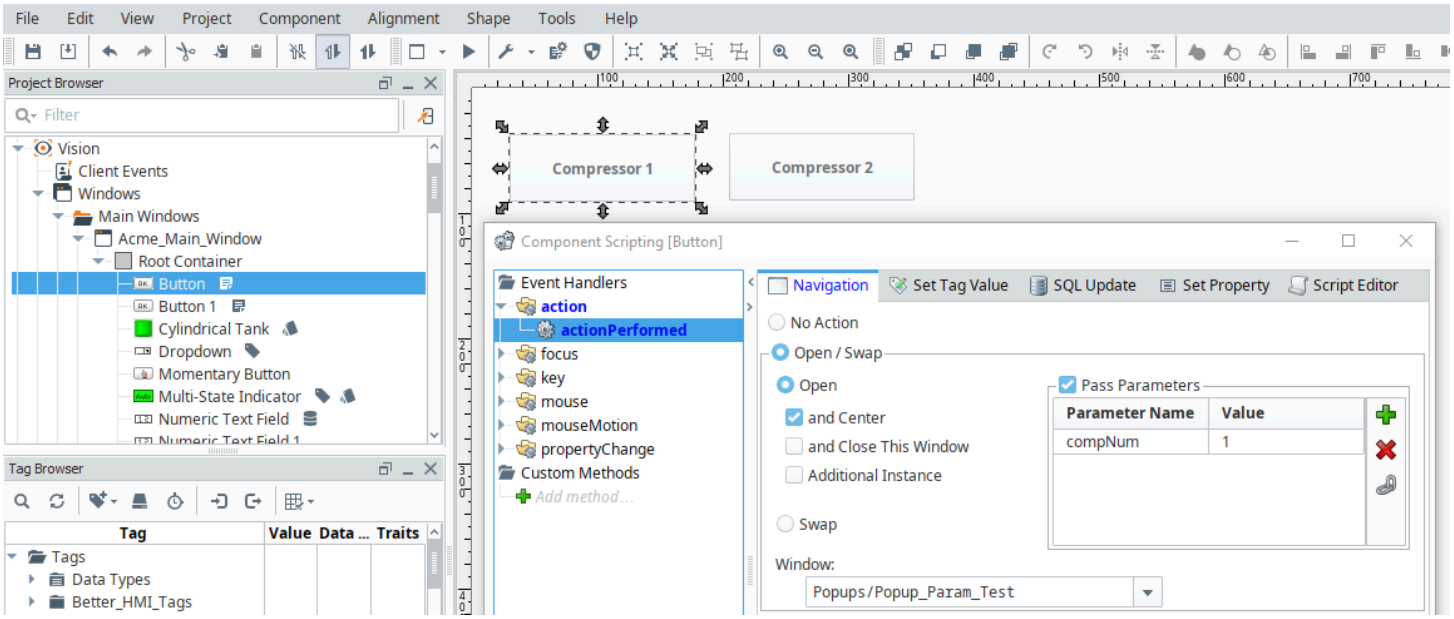


The table below lists possible values you can choose from in the dropdown menu:

Value	Description
None	The project library will not use the Designer nor the Gateway to populate scripting hints.
Designer	The project library will use the Designer scope to populate scripting hints.
Gateway	The project library will use the Gateway scope to populate scripting hints.
All	The project library will use both the Designer and Gateway scope to populate scripting hints.

Components

Both Perspective and Vision offer component based scripting triggers, providing a means to execute a script under a number of different situations, such as a user interacting with a component or a component property value changing. For more information on how both module handle component based scripts, take a look at the **Scripting in Perspective** and **Scripting in Vision** sections.



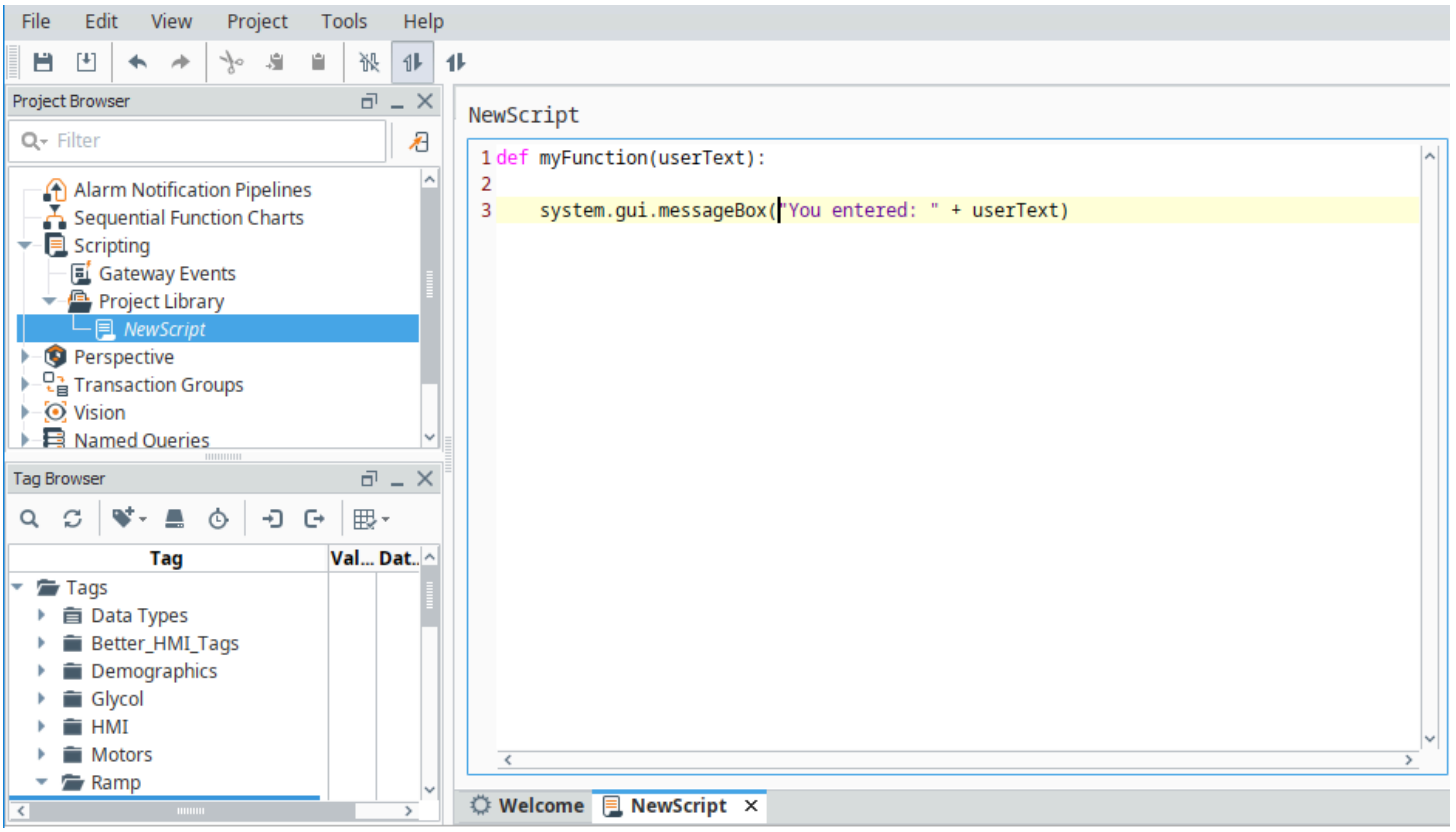
Client, Gateway, and Session Event Scripts

Scripts can be set to activate on specific events that occur during runtime. For example, you can trigger a script to run when a Vision client starts, or on certain time intervals.

Each of these Event Script types are considered project resources and are stored as readable PY files. More information on these events can be found on the **Client Event Scripts**, **Gateway Event Scripts**, and **Perspective Session Event Scripts** pages.

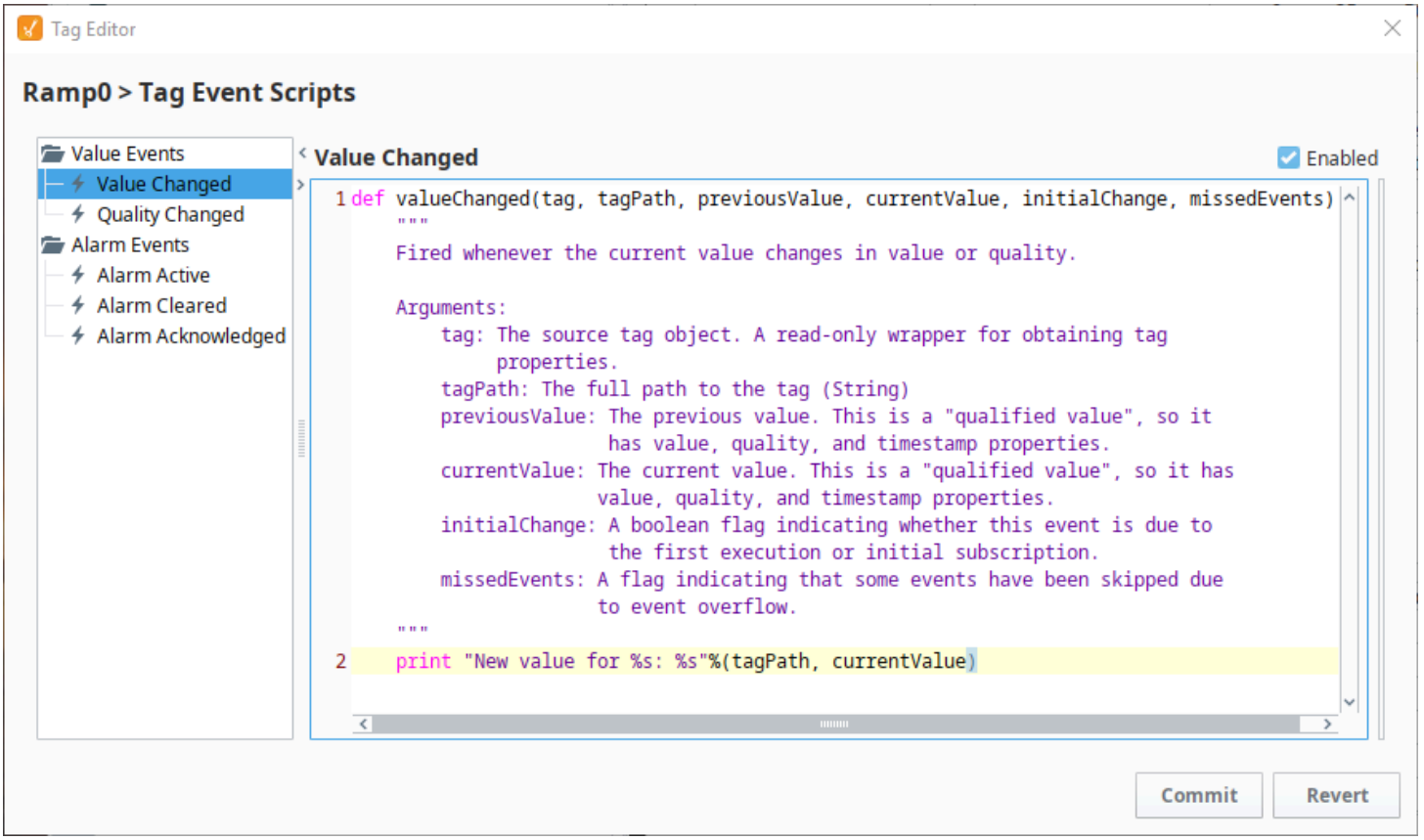
Project Scripts

You can create your own reusable blocks of code in the **Project Library**. Once configured, these functions can be called from anywhere in a project, just like our **system.*** functions.



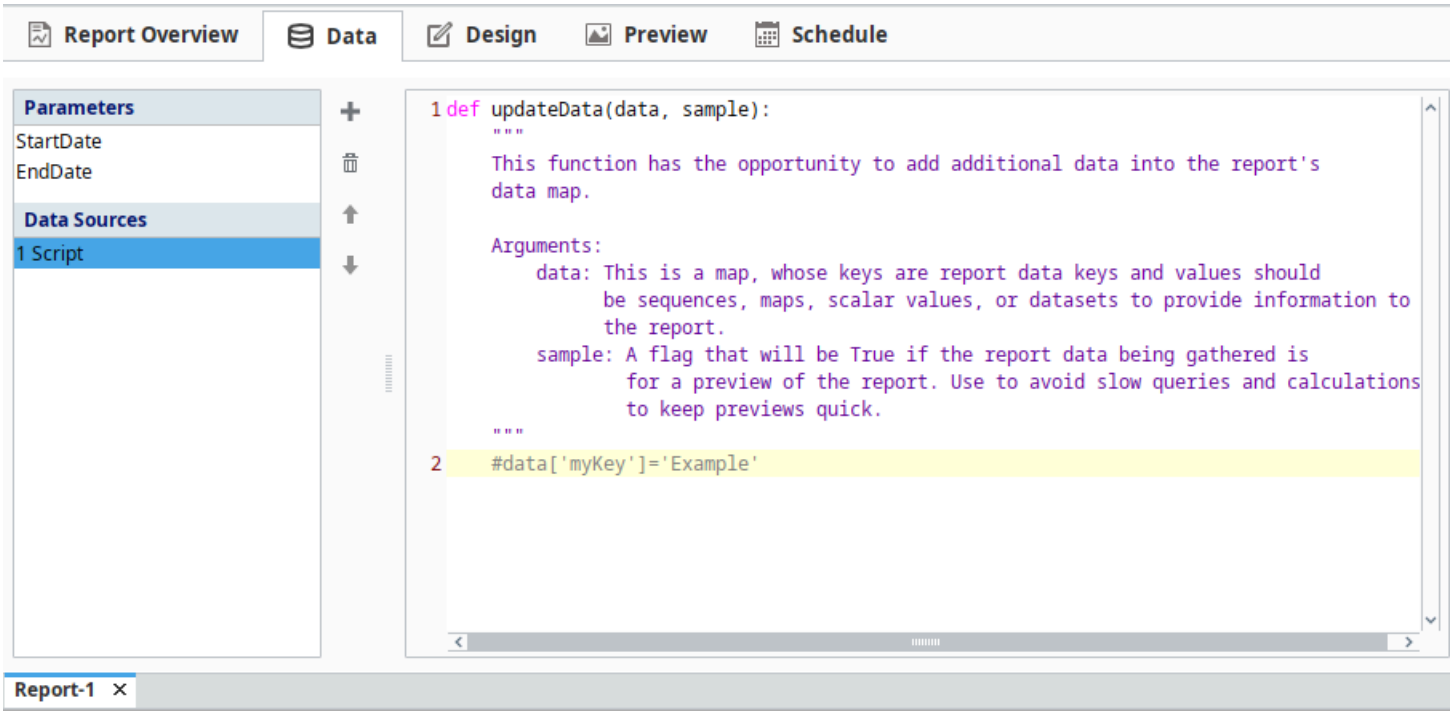
Tag Scripts

Once Enabled, these scripts are fired whenever a **tag value changes** or an alarm event happens. You can use them for additional diagnostics, to set additional tags, or to react to an alarm event. Because these events are on tags, they are Gateway-scoped.



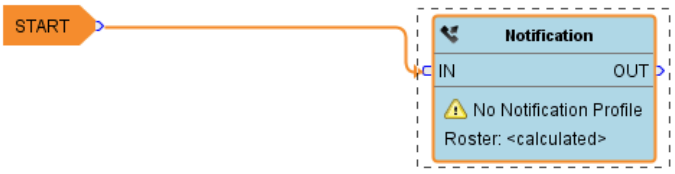
Reporting

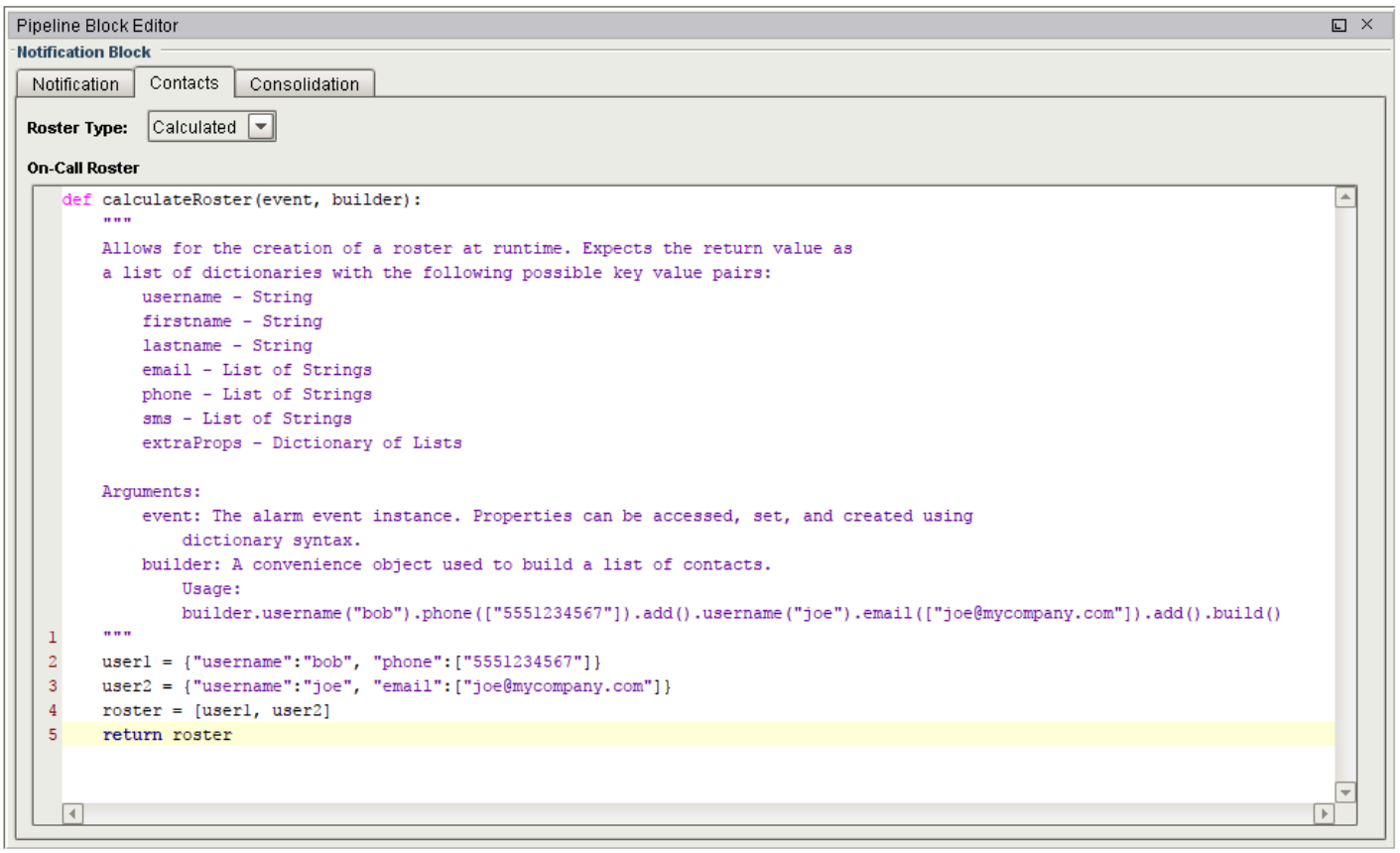
Reporting uses scripting in many different ways to help increase the effectiveness of the report. Scripting in Reports is used to **create and modify data sources**, **manipulate charts**, and set up a script as a **scheduled report action**.



Alarming

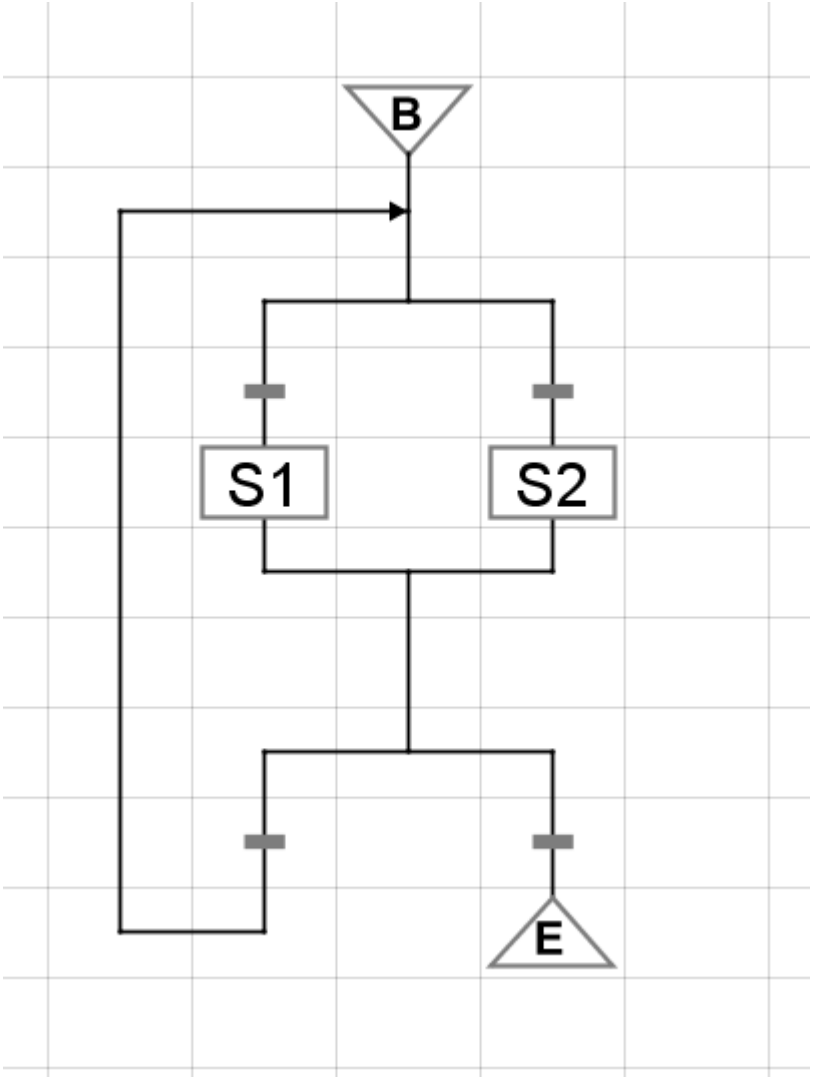
The Alarm Notification system can also use scripting to great effect. A **script block** allows a script to be run within the pipeline, allowing data to be manipulated as the alarm event travels through the pipeline. Additionally, scripting can be used to generate a **custom roster** of users at runtime, giving full customization to who gets notified by the alarm event.





Sequential Function Charts

Sequential Function Charts (SFCs) are a flowchart of blocks that run scripts. They are executed in a specific sequential order along with some logic to potentially loop or call other charts. The scripts here can interact with the Gateway, and provide greater control when each step needs to complete before the next one can begin in multi-step processes.


B
S1
S2
E

Getting Started with Scripting in Ignition

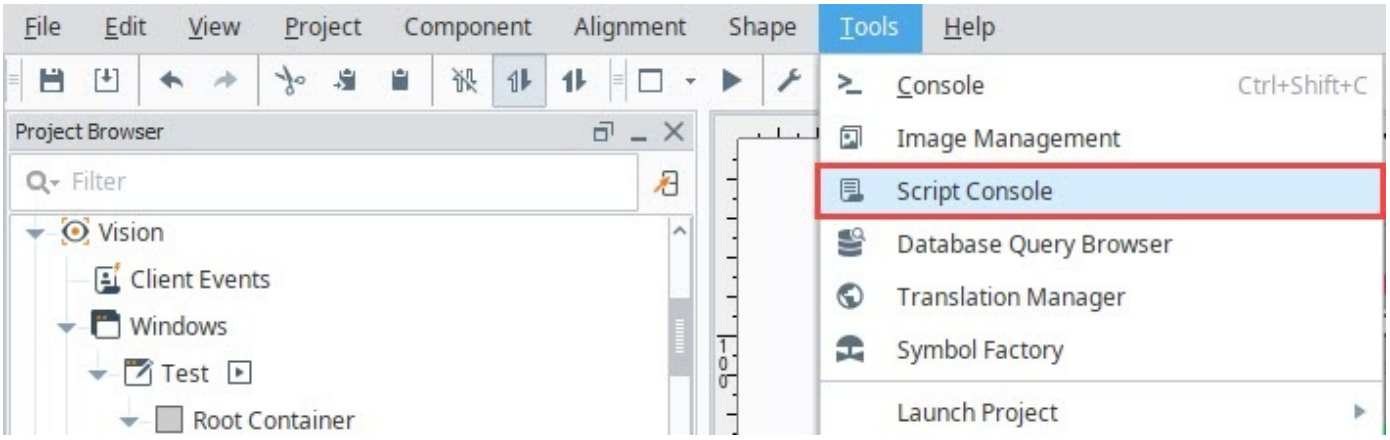
Overview

The best way to get started with scripting in Ignition is to write some very simple scripts and print them within Ignition. Let's take the phrase, "Hello World," and print it within Ignition. To show how flexibile Ignition is, there are actually a few ways to easily do this. One way is to use the Scripting Console, and another way is to use a Button component. Both examples are shown below.

Example - "Hello World" Using the Script Console

When testing or writing a new script, the **Scripting Console** is very useful since you can immediately get some feedback on the results of the script. When learning Python, it is a great place to start since you don't have to create a window or component before you begin writing your code.

1. In the **Designer**, open the Script Console from the Menu Bar: **Tools** > **Script Console**.

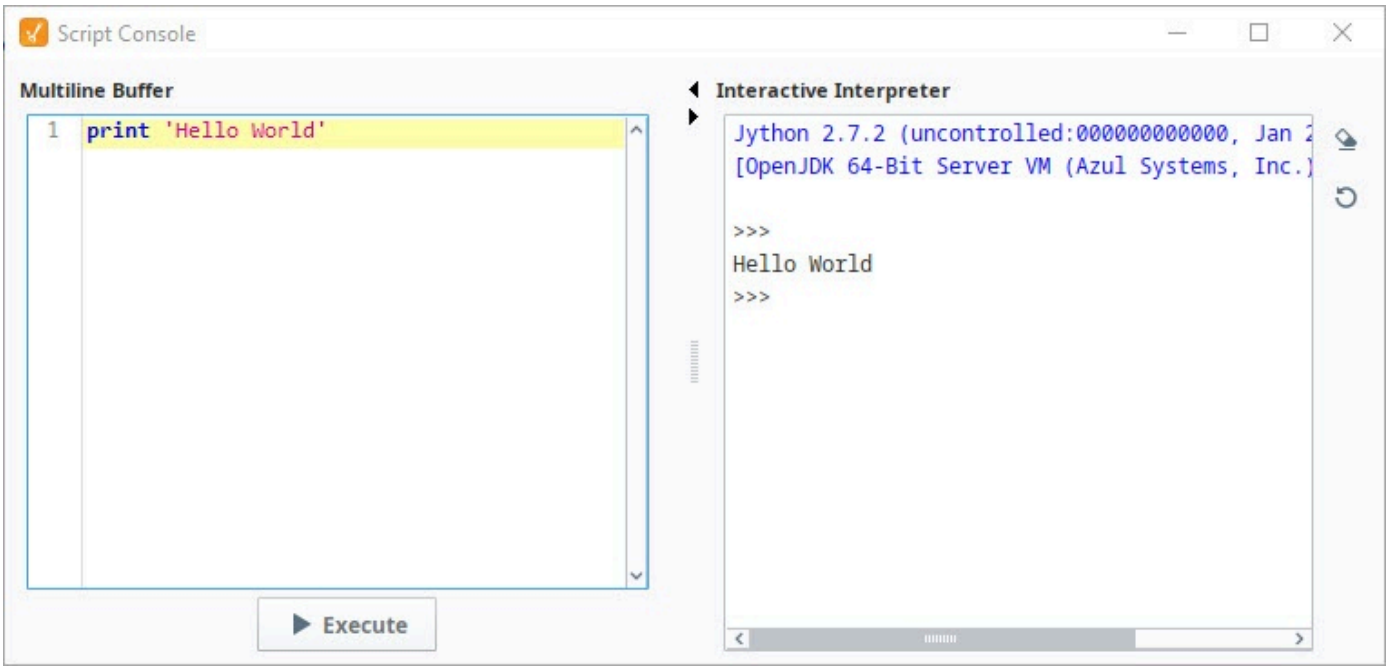


2. The Script Console will appear. Type the following code, or simply copy and paste it into the text area in the Multiline Buffer on the left side of the Script Console:

Python - Simple Print

```
print 'Hello World'
```

3. Click the **Execute** button at the bottom of the Script Console. You should see the message "Hello World" appear in the Interactive Interpreter on the right side of the Script Console.

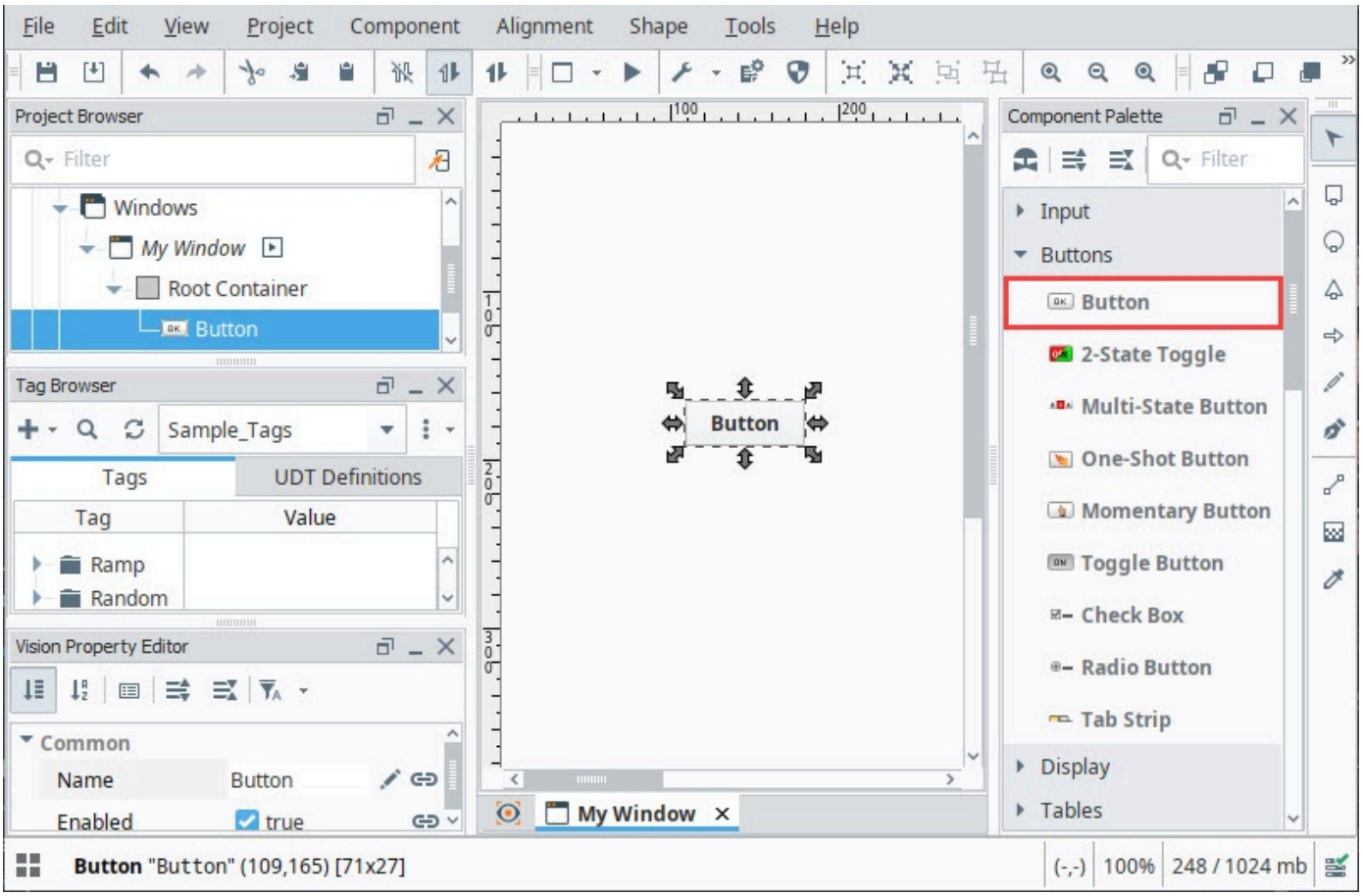


Example - "Hello World" on a Button Component

Scripts are commonly located on components in a window. In this example, let's add a script on a Button component, and print out "Hello World" when the Button is pressed.

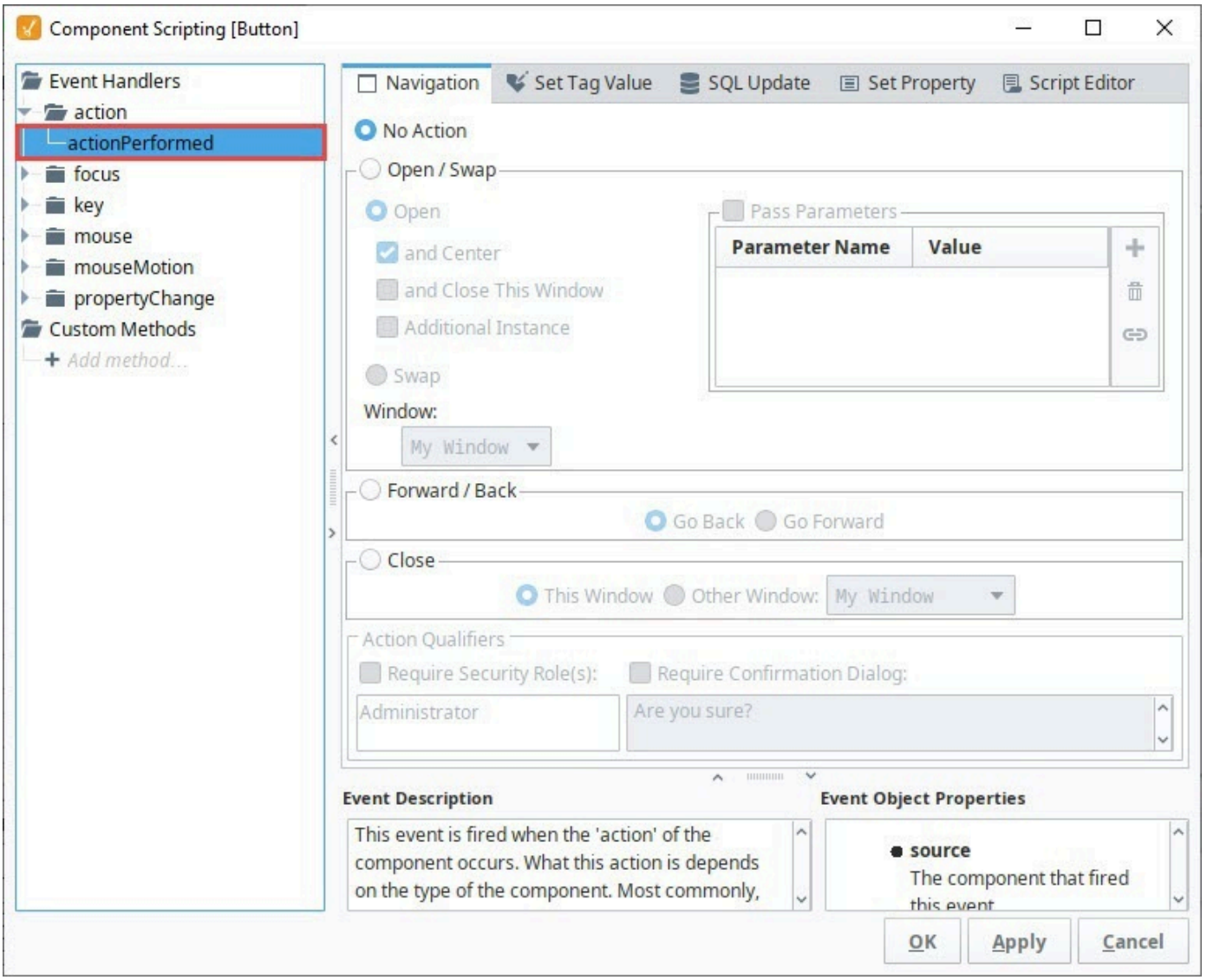
1. In the **Designer**, create a new **Main Window**. Give it a meaningful name if you like. (We won't need to reference the name of the window in any of these examples).

2. Drag a standard **Button** component to your window.



3. Let's add a script that triggers when the Button is pressed. With the Button component selected, right click on **Scripting** to open the **Component Scripting** window.

4. Under Event Handlers, select the **action > actionPerformed** option, and click the **Script Editor** tab. Make sure the **actionPerformed** event is highlighted. If an event is not selected, all of the remaining features in this window will be disabled, so we will not be able to write a script. Additionally, if the wrong event is selected, then our script will not trigger when we expect it to.

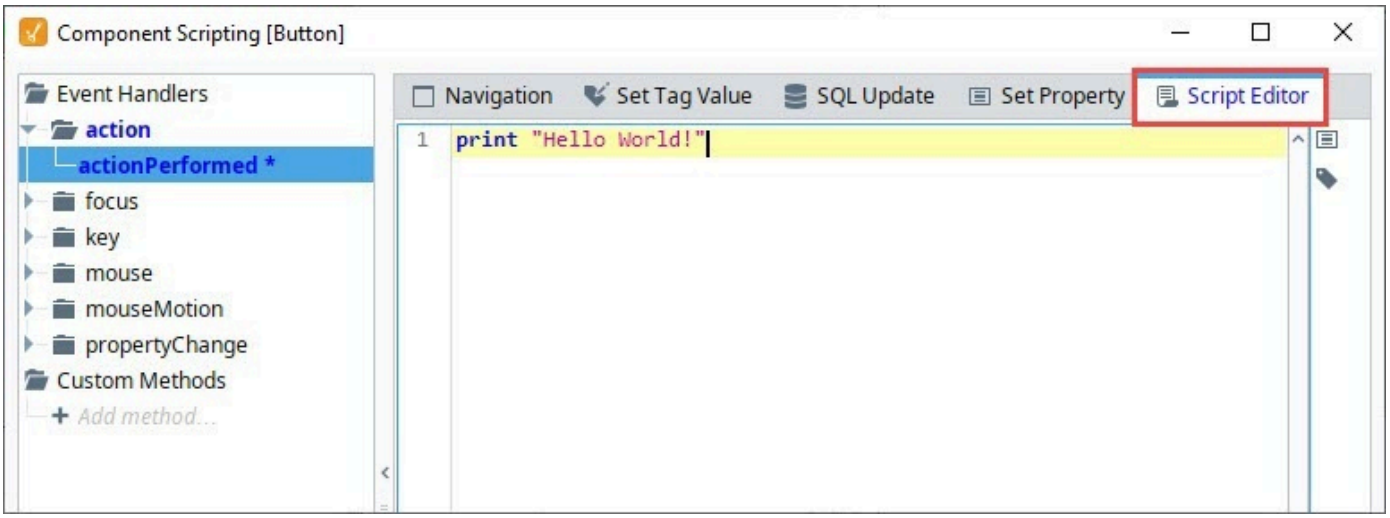


5. In the Script Editor, you will see a large editable Text Area. This is where we will type our script.

6. Let's generate a message that shows "Hello World!" Use the following code to get started. Make sure the word "print" lines up exactly to the left edge. Indention in Python means something, so we need to avoid starting our lines with a space or tab unless we're denoting a block of code under something like an **if-statement** or **function definition**. Click **OK** to close the **Component Scripting** window.

Python - Simple Print

```
print "Hello World!"
```

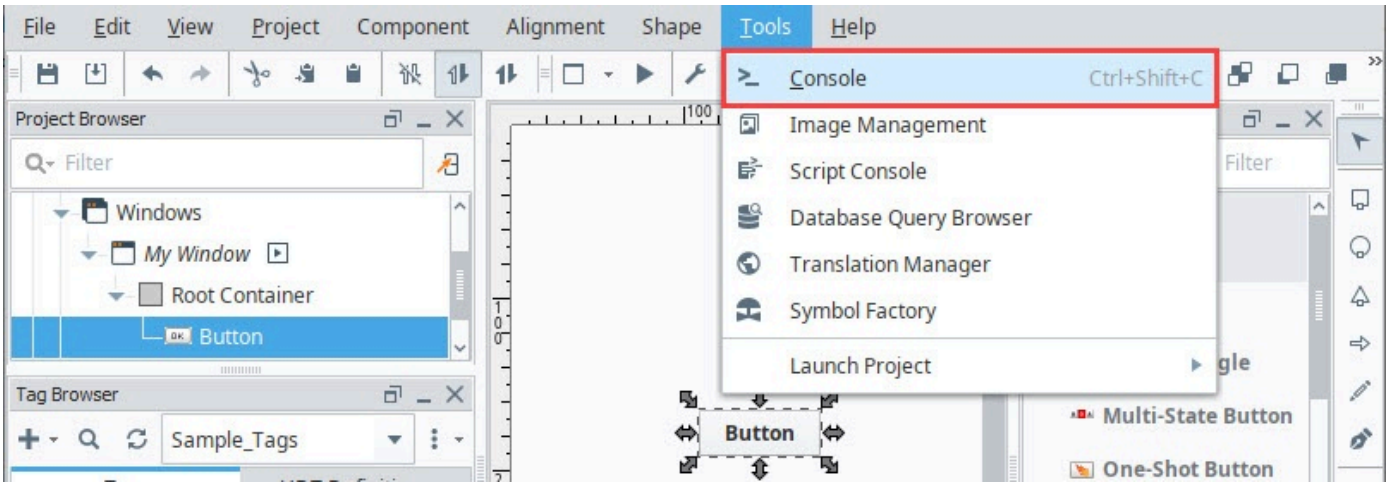


Notice the **actionPerformed** event is blue and bold. This means there is a script on this event. This is useful to know in situations where a component has scripts on multiple events.

Additionally, an asterisk character (*) is next to the event. This means you have not applied/saved the changes to the script. The asterisk will disappear when you press either **OK** or **Apply**, and reappear whenever you make a change to the script. If you see this, then it means you may want to save any changes you made.

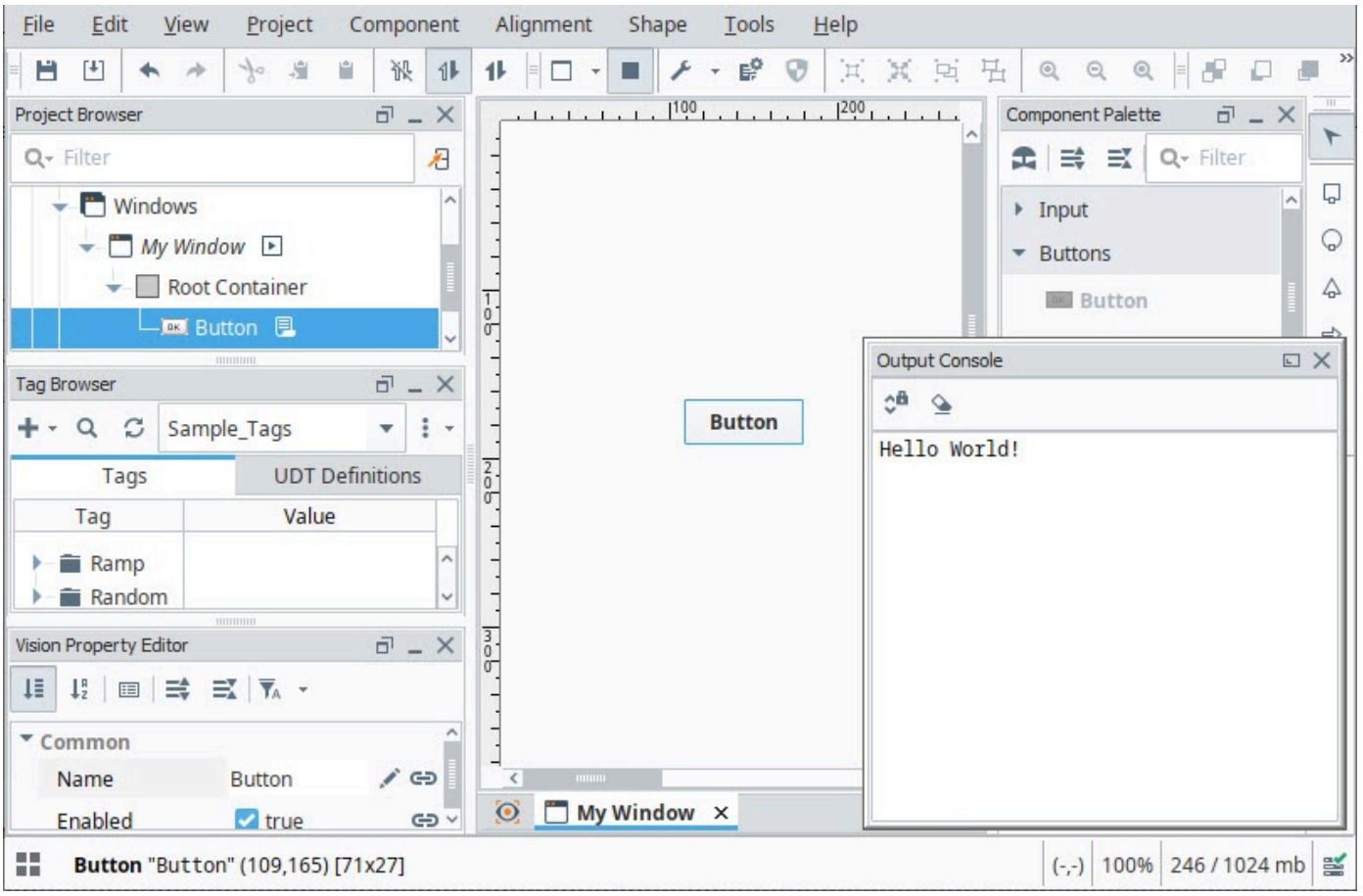
The **Script Editor** tab also has a blue color, denoting where the script is. An event will only ever have a script located on a single tab at any time. If a new tab is selected and configured, it will wipe out the work on the prior tab, for example, writing a script on **Script Editor** and then configuring the **Navigation** tab will erase the script on the **Script Editor** tab.

7. Now we can test the script. Place the **Designer** into Preview Mode, and press the Button. If everything is working as intended, then it should appear as if nothing happened. This is because we used print in our script, which always outputs to a console, as opposed to popping up at the user. This means we need to open the Designer's **Console** to see the results of our script. At the Designer's menu bar, select **Tools** > **Console**. This will make the console appear.



8. There may be a large amount of text in the Output Console. The Designer logs many different types of activities and events here, including polling events from components on other windows

that are currently open in the Designer. However, the most recent events should be towards the bottom. As a quick tip, you can also click the eraser icon to clear out the console, and then press your **Button** again to generate a new entry like we did in this example.



We can now see where print statements go when called from a component, but this isn't too useful, as we don't want our users to open the console to see messages. Print statements are very useful when troubleshooting a problematic script, or even when testing a new script. Keep this in mind as you start to delve into scripting more.

Example - Using a Message Box

Now that we've seen how to print to the Output Console, lets make our message appear when we call it. This time, we will modify the script on the Button component to bring up a window that the user will see.

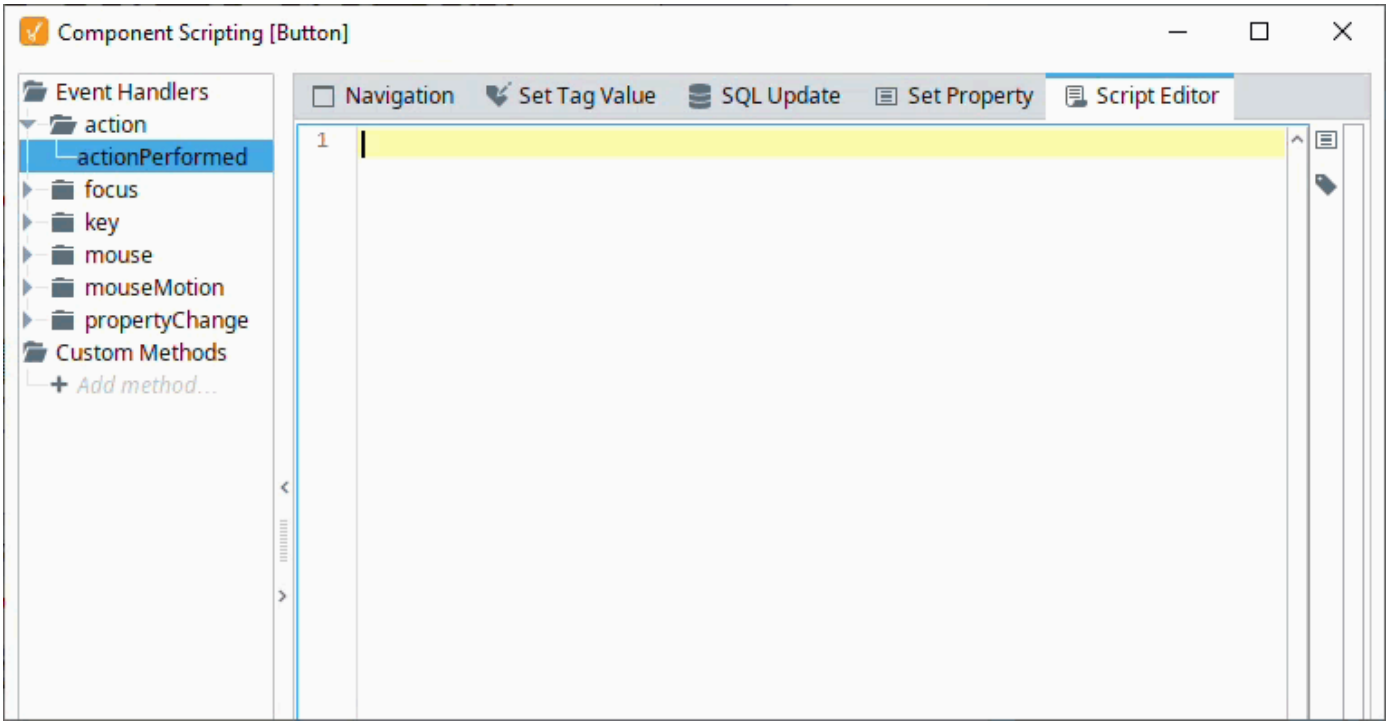
1. Open the **Component Scripting** window again, and find the script on the Button.

2. We will use one of Ignition's built-in functions, called **system.vision.showMessage**, to display the message. This will make a message box appear, which is a modal window that we can pass a string to. Remove your old code on the Button, as we will be replacing it. Start by typing the following:

Pseudocode - Built-in Functions Begin with system.

```
system.
```

3. The Autocompletion popup appears by default after typing ".", which lists all of the available system functions. Start by clicking on **vision**, and then **showMessage**. Note, that you can still type while the popup is open to filter the results in the list.



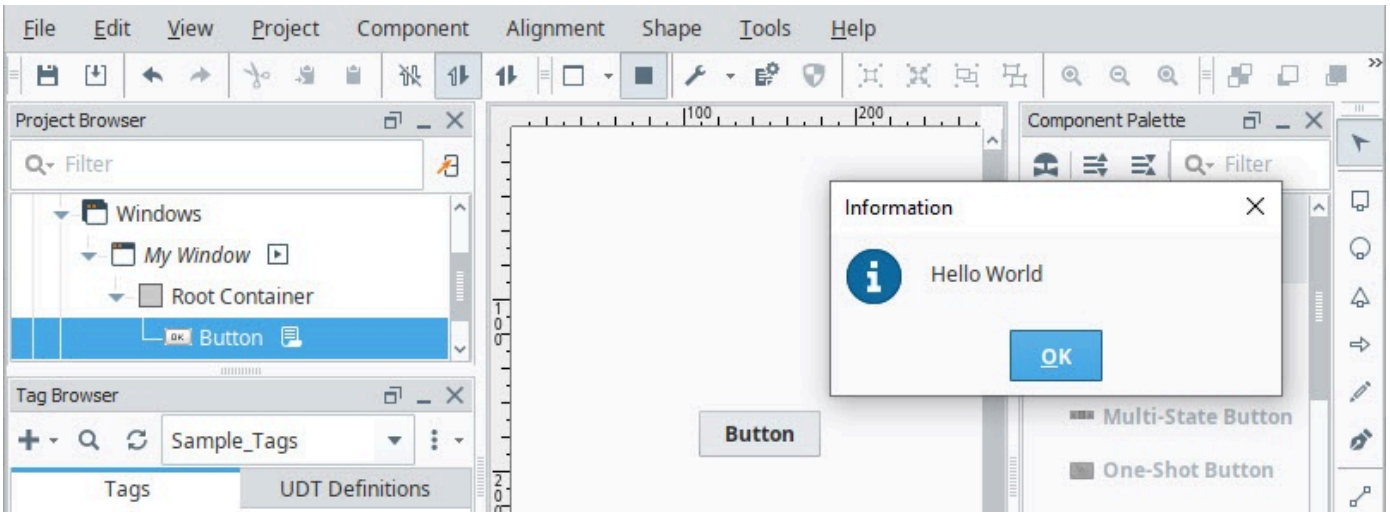
4. Once you've selected **showMessage**, click on it, and it should start some of the code for you automatically.

5. Complete the code by placing some parentheses and a message as a string. Alternatively, you can copy the example below.

Python - Using a System Function to Write to a Message Box

```
system.vision.showMessage('Hello World')
```

6. Place the **Designer** into **Preview Mode**, and click your Button. You will see a Message Box appear displaying the message we passed to the function.


File
Edit
View
Project
Component
Alignment
Shape
Tools
Help
Project Browser
Filter
Windows
My Window
Root Container
Button
Tag Browser
Sample_Tags
Tags
UDT Definitions
100
200
Component Palette
Filter
Information
Hello World
OK
Button
Multi-State Button
One-Shot Button

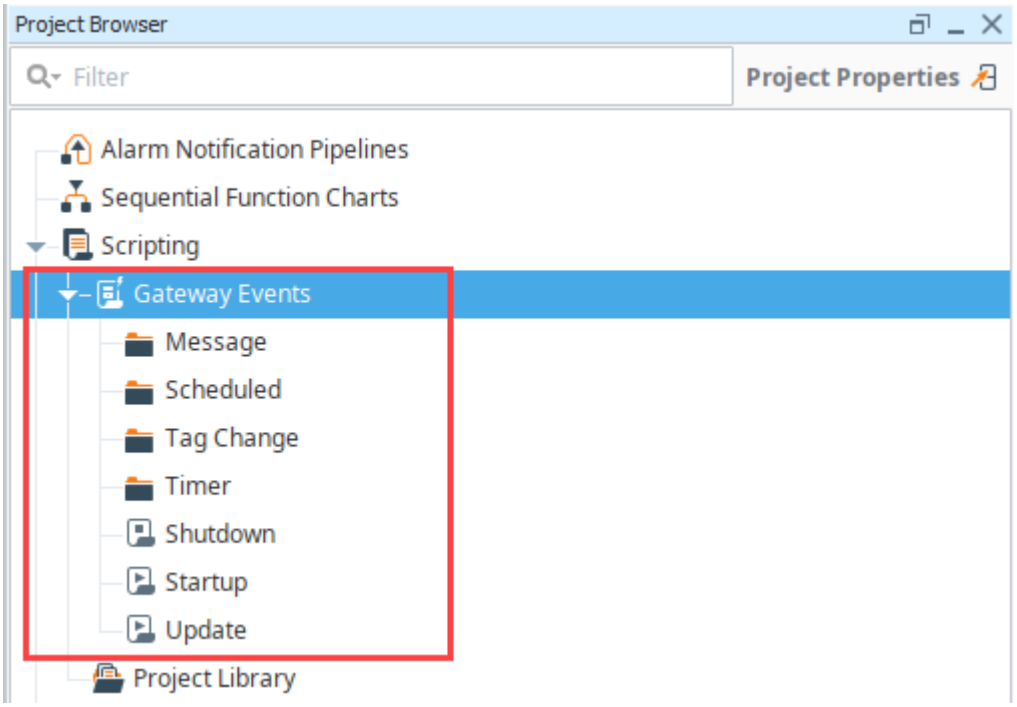
Gateway Event Scripts

Gateway Event Scripts Overview

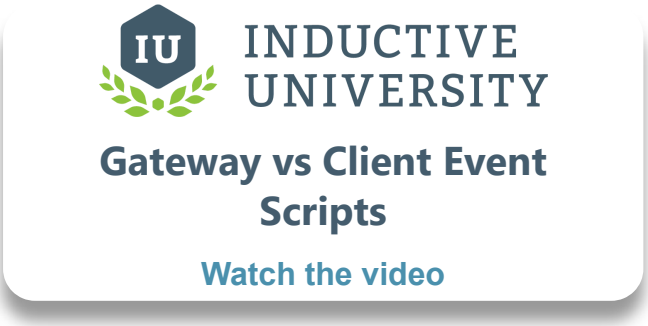
Gateway Event Scripts are scripts that run directly on the Gateway. They are useful because they always run, regardless if any Sessions or Clients are open. They also provide a guaranteed way to make sure only a single execution of a particular script occurs at a time, as opposed to placing a script in a window where there could be multiple instances of the window open at any given point.

Note that even though Gateway Event Scripts run on the Gateway, they're still considered a project resource and are stored as readable PY files. Project backups will include any Gateway Event Scripts.

The Gateway Event Script workspace is located in the Scripting menu of the Designer or in the Project Browser under **Scripting** > **Gateway Events**.



Other Event Scripts

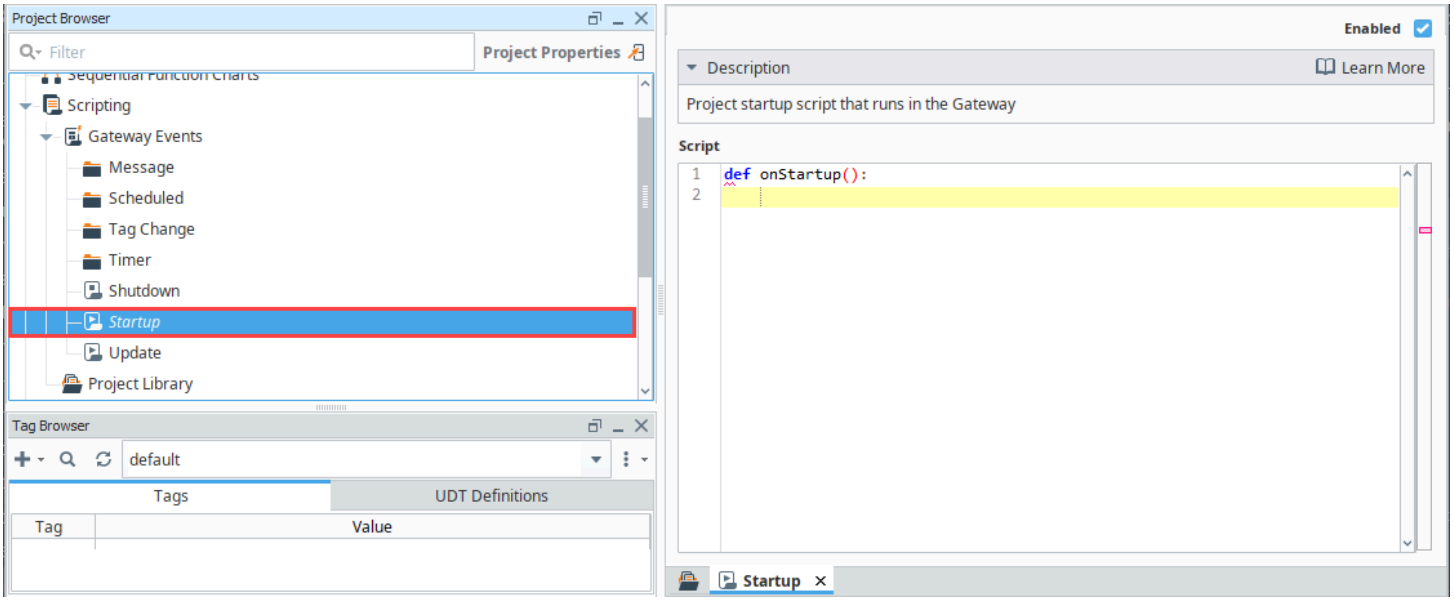


The content on this page will focus primarily on Gateway Event Scripts. However, there is some overlap with other Event Scripts, as they have similar events. More information can be found on the **Client Event Scripts** and **Perspective Session Event Scripts** pages.

> **NOTE**
>
> System functions are available for all types of event scripts, but some system functions are specific to certain ones. When you're writing event scripts, it's important to remember the scope of where you're writing the script. You can check the **System Functions** in the Appendix to see their descriptions and what scope they run in.

Startup Script

The Startup Script event runs at the startup of the Gateway. Additionally, if the project is restarted in some way, such as by making a change to a Gateway Event Script and saving, then the Startup Script will be called. This means that while editing scripts frequently in the Designer, the startup and shutdown events may happen frequently.



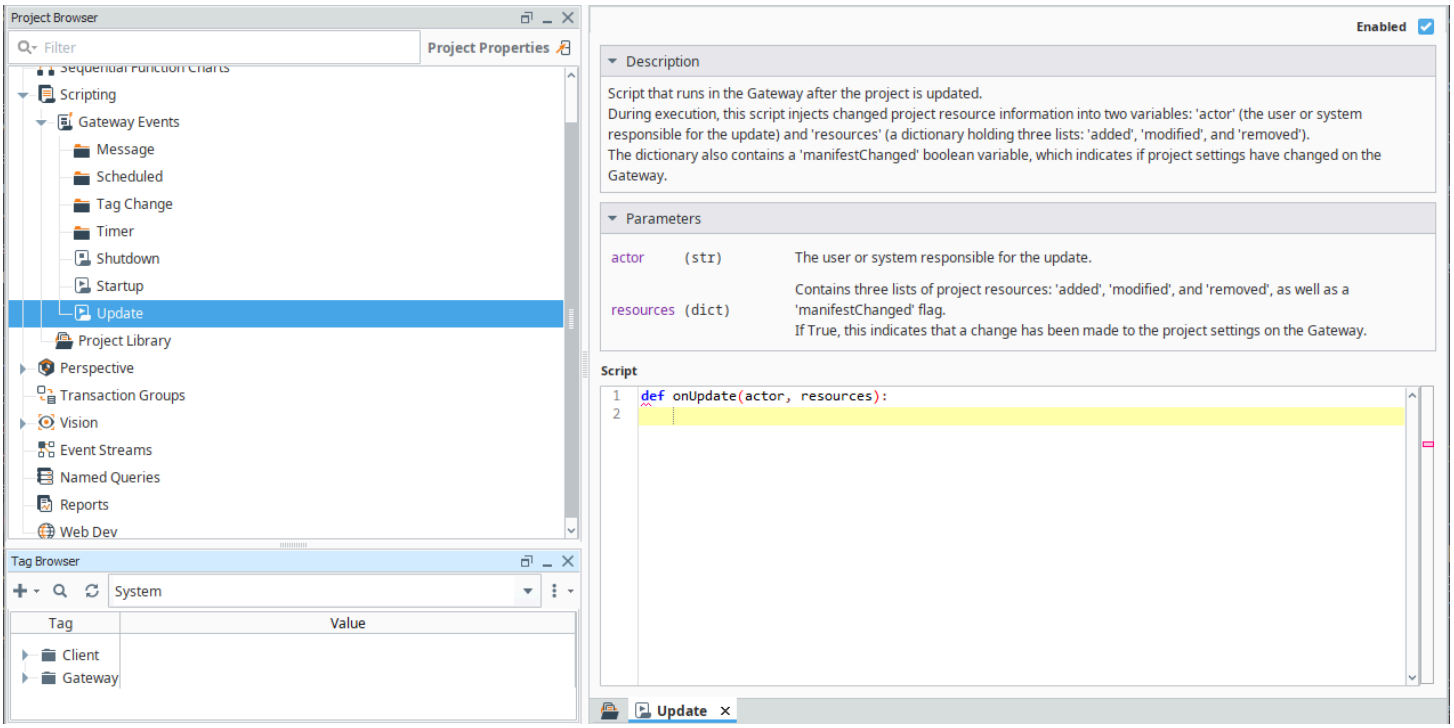
Gateway Startup Behavior

There is a specific order to when the various Startup Scripts are run. When troubleshooting your Gateway startup times, consider the following:

1. **Gateway starts**: The Gateway will start as an OS service, and start the context. No startup scripts can run before this is complete.
2. **Projects are started**: This includes all of the Gateway-scoped items in the projects such as Transaction Groups, SFCs, etc. This does not refer to launching clients, and no clients can be automatically launched at this time. All Gateway Startup Scripts are run at this time for each project. Note that if you copied a project, always check for Gateway-scoped events such as these. You generally don't want a Gateway Startup Script to run twice because it is in two projects.

Update Script

The Update Script event runs after a project is saved or updated on the Gateway. This enables you to insert a script that will run every time a project is saved.



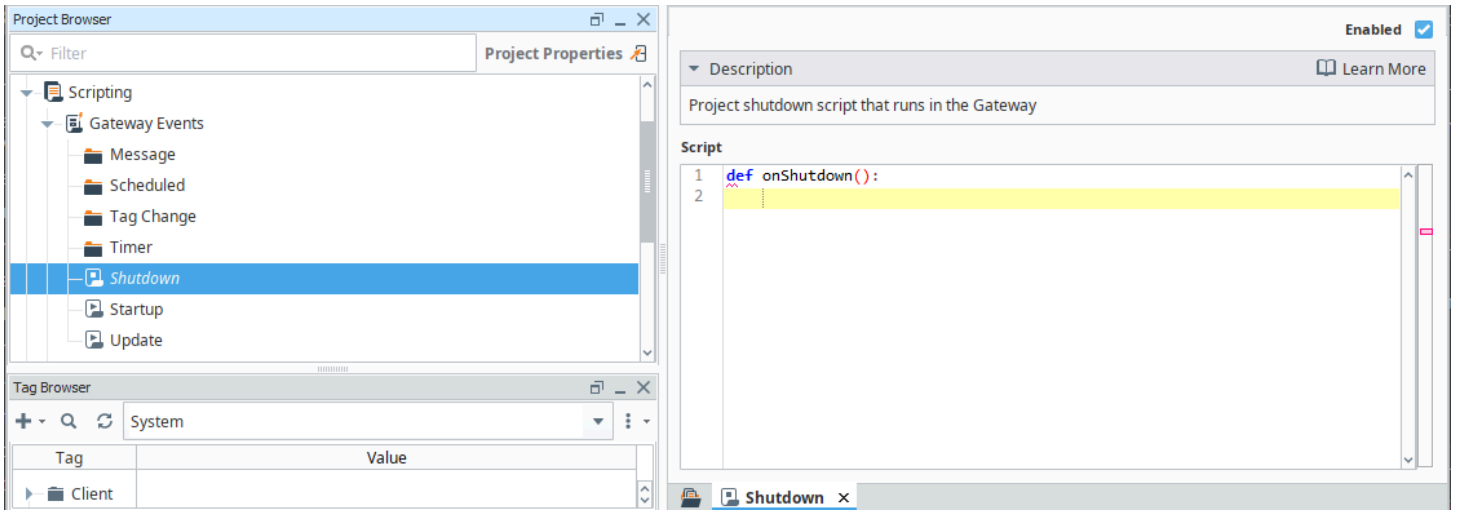
This script reports which resources were modified during a project save and includes the following parameters.

Parameter	Description	Type
actor	The user or system responsible for the project update.	String
resources	A dictionary that holds the following keys: • **added**: List of dictionaries containing information about resources that were added to the project. • **removed**: List of dictionaries containing information about resources that were removed from the project. • **modified**: List of dictionaries containing information about resources that were modified. • **manifestChanged**: A boolean variable indicating whether or not a change has been made to the project settings on the Gateway.	Dictionary

Shutdown Script

The Shutdown Script event runs at the shutdown of the project, which means it can be used as a way to trigger a script when the Gateway has to be restarted. It allows you to run a piece of code as the shutdown is occurring. After the script completes, the shutdown will finish.

> **NOTE**
>
> The Shutdown Script event only gets called if the Gateway is requested to shut down. If power is lost abruptly (power outage, hard restart, etc.) this shutdown script will not run.

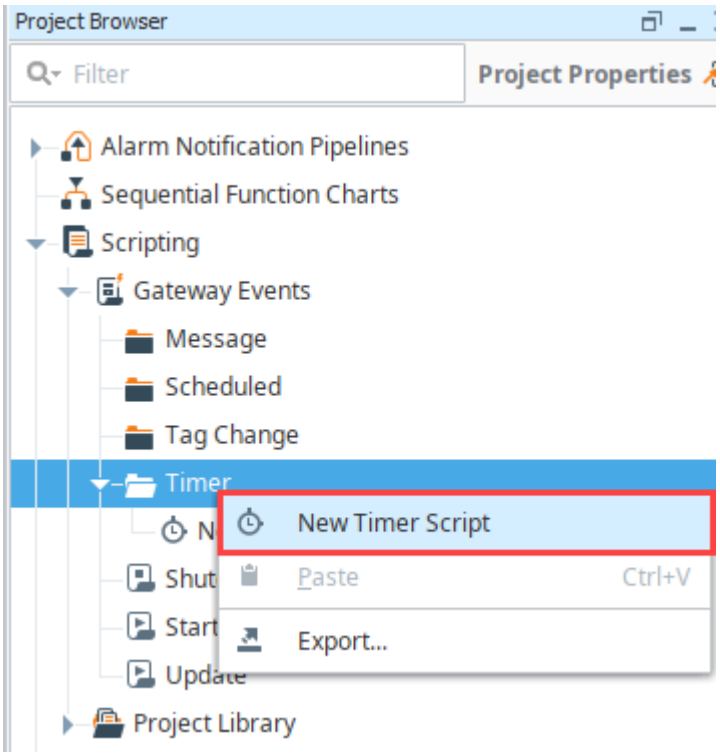


Gateway Shutdown Behavior

Similar to how a **Startup Script** behaves, "Shutdown" in the context of this event means "project shutdown", so shutting down the Gateway would trigger this event, as well as disabling the project containing a script on this event. Additionally, making a change to a Gateway Event Script in the Designer, and then saving the project will cause the project to restart, which means this event can get called by simply making changes in the Designer and saving.

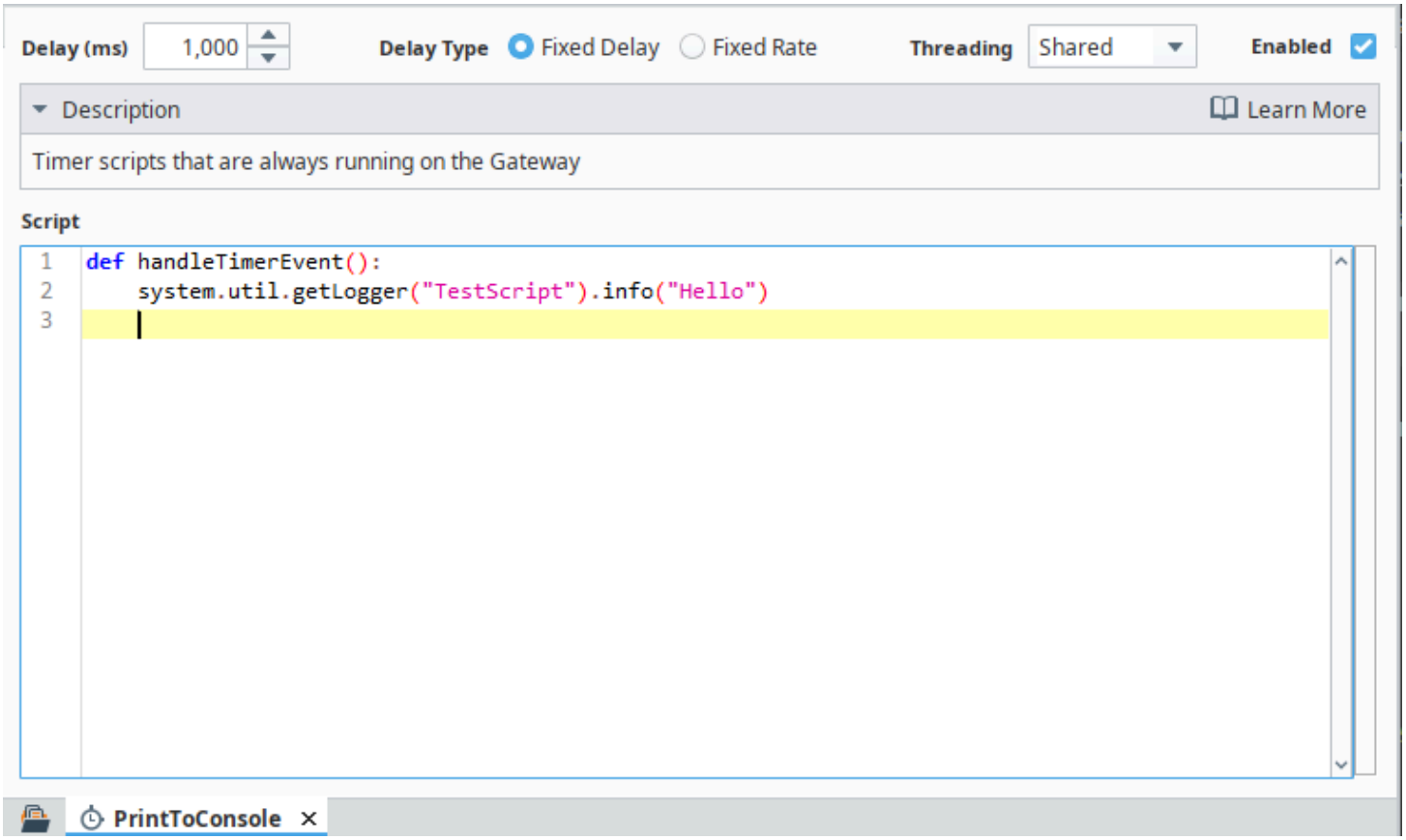
Timer Script

The Timer Scripts execute periodically on a timer at a fixed delay or rate. This allows you to set up a sort of heartbeat that can run on the Gateway. This is the ideal event to use if you need the Gateway to periodically perform some scripting task.



Since multiple Timer Scripts can be added, the Timer folder has a right-click menu to add new scripts. Once created, use each timer's right-click menu to copy, rename, or delete the script. Selecting a script will open it so that it can be viewed and modified.

Timer Script Settings



Timer Script names are entered when creating the script. All other properties can be found on the top bar of the Script Editor. Below is an overview of the settings for a Timer Script:

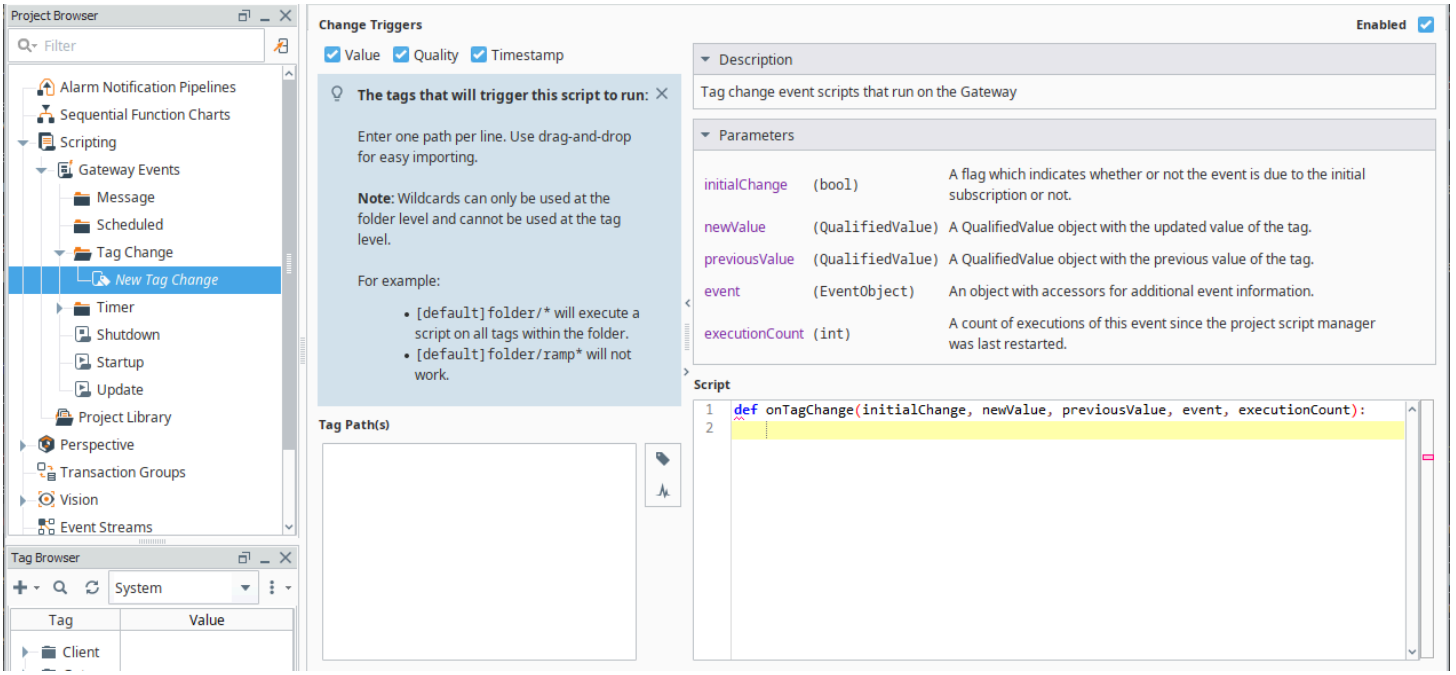
- **Name**: The name of the Timer Script. Names must be unique per project, so two Timer Scripts in the same project cannot have the same name.
- **Delay**: The delay period in milliseconds. The meaning of this setting is dependent on the **Delay Type** setting.
- **Delay Type**: Determines how the Delay setting is utilized.
  - A **Fixed Delay** Timer Script waits for the given **Delay** between each script invocation. This means that the script's rate will actually be the delay plus the amount of time it takes to execute the script. This is the default and safest option, since it prevents a script from mistakenly running continuously because it takes longer to execute the script than the delay.
  - **Fixed Rate** scripts attempt to run the script at a fixed rate relative to the first execution. If the script takes too long, or there is too much background process, this may not be possible. See the documentation for **java.util.Timer.scheduleAtFixedRate()** for more details.
- **Threading**: Determines which thread this script should run in. In other words, this setting allows you to specify if you want this Timer Script to share execution resources or not. The rule of thumb here is that quick-running tasks should run in the shared thread, and long-running tasks should get their own dedicated thread.
  - The **Shared** setting means that all Timer Scripts will share a thread. This is usually desirable, as it prevents creating lots of unnecessary threads: threads have some overhead, so a small amount of resources are used per thread. However, if your script takes a long time to run, it will block other timer tasks on the shared thread.
  - The **Dedicated** setting will create a separate thread specifically for the Timer Script to use. This setting is desirable when your scripts executions must be as consistent as possible, as other Timer Scripts can't slowdown or otherwise impact the execution of a script in a separate thread.
- **Enabled**: Allows you disable the Timer Script when set to false.

Tag Change Scripts

The Tag Change Script event allows you to specify any number of tags, and trigger a script when one of them change. Since these execute based on a tag changing value, Tag Change Scripts are ideal when you need a script to run based on some signal from a PLC.

Having the Tag Change Scripts run in the Gateway scope means that the scripts are active as long as the Gateway is running. Thus, you do not need a Client or Session to be open for a Gateway Tag Change Script to execute. When executing, each Tag Change Script runs in a separate thread. This prevents long running scripts from blocking the execution of other Tag Change Scripts.

> **NOTE**
>
> Due to their nature, Client Tags will not trigger Gateway Tag Change Scripts. However, **Client Event Scripts** enable you to trigger a script based on a Client Tag changing value.



Tag Change Script Interface

Select **New Tag Change Script** from the Tag Change folder right-click menu to name and add a script. Once created, use each Tag Change Script right-click menu to copy, rename, or delete the script. Selecting a script will open it so that it can be viewed and modified.

The following settings can be configured with a Tag Change Script:

- **Change Triggers**: When the tag changes, the script can trigger based on the Value, Quality, and/or Timestamp. Note that regardless of how many triggers changed, the script only executes once per tag, so leaving all triggers enabled will not trigger three executions each time the tag changes.

- **Enabled**: Determines if the script is active or not. Set to false to disable the script.

- **Tag Paths**: A list of tag paths to monitor. When any of the tags listed in this area change, the script will trigger. Note that the list is not comma separated. New paths are specified on each line. The following items are included and accepted to assist with creating a tag path list:

- The **Tag Browser** icon opens a Tag Browser window, allowing you to quickly lookup and add tag paths to the Tag Change Script.
- The **Path Diagnostic** icon verifies the paths specified under the tag paths list. This is useful for checking for typos in the list.
- Wildcards can be used at the end of configured tag paths. Using a wildcard will execute the script for all tags within the same folder. In the below example, the script "New Script" will run when the value of any tag in the "folder" folder changes.

> **USING WILDCARDS IN VERSIONS 8.3.4 AND LATER**
>
> **Changed in 8.3.4** You can now use wildcards at both the folder level to help specify a folder path, and at the tag level to help specify a tag path. Prior to 8.3.4, wildcards could only be used at the folder level. See the **Wildcards in Tag Paths** page for more example usage.

- **Script**: Where the Python script associated with this event will be placed. The Tag Change Objects section describes the parameters included in this script.

Tag Change Objects

Tag Change Scripts contain several built-in objects that are useful for inspecting the event, such as seeing what value the tag changed to. These objects are listed below.

The initialChange Value

The boolean initialChange variable indicates if an event is due to an initial subscription. The initialChange value is true only when the script runs for the first time after the project updates, which happens when the executionCount is 0. The initialChange variable can be used to filter out an event preventing a script from running when the values haven't actually changed.

> **NOTE**
>
> Whenever you make changes to your project that affect scripts—like updating your project library or modifying event scripts—all the event scripts, including timers, scheduled scripts, and tag change scripts, are restarted. Once restarted, your executionCount will be reset to 0.

```
if not initialChange:

    # Do something useful here
```

The executionCount Value

An integer representing a number of event executions since Gateway scripts were restarted (typically by applying a change to a script and saving the project).

```
if executionCount <= 1:
    pass
```

The newValue Object

A **QualifiedValue** object that represents the current value on the tag.

Method	Returns	Description	Usage Example
getValue()	Varies	Returns the new value on the tag.	newValue.getValue()
getQuality()	QualityCode	Returns the new quality on the tag.	newValue.getQuality()
getTimestamp()	Date	Returns the new timestamp value.	newValue.getTimestamp()

The previousValue Object

A **QualifiedValue** object that represents the previous value of the tag.

Method	Returns	Description	Usage Example
getValue()	Varies	Returns the previous value on the tag.	previousValue.getValue()
getQuality()	QualityCode	Returns the previous quality on the tag.	previousValue.getQuality()
getTimestamp()	Date	Returns the previous timestamp value.	previousValue.getTimestamp()

The Event Object

This object offers some additional utility, such as accessing the previous values on the tag.

Method/Attribute	Returns	Description	Usage Example
getCurrentValue()	QualifiedValue	Returns a **QualifiedValue** object (similar to the newValue object), representing the current value on the tag.	event.getCurrentValue().getValue()
getPreviousValue()	QualifiedValue	Returns a **QualifiedValue** object, representing the values on the tag before the change.	event.getPreviousValue().getValue()
getTagPath()	TagPath	Returns a TagPath object, that can be further examined for details about the path on the tag that changed. See the TagPath Object table below. Additionally, the TagPath object can easily be turned into a string, providing quick access to the path of the tag that changed value.	event.getTagPath()

Method/Attribute	Returns	Description	Usage Example
getValue()	QualifiedValue	Returns a **QualifiedValue** object (similar to the **newValue** object), representing the current value on the tag. This method is functionally identical to getCurrentValue(), and maintained mostly for backwards compatibility reasons.	event.getValue().getValue()

changes	TagChangeType	A RegularEnumSet typed attribute that describes what changed: the value, quality, or timestamp. Values in the set are **TagChangeType** objects, which can be converted to string with either Python's built-in str() or Java's toString().	`for i in event.changes:`
			`  if i.toString() == "Value`
			`  foo()`

tagPath		An attribute that describes the tag path on the tag	event.tagPath.getItemName() event.tagPath.getParentPath()

Method/Attribute	Returns	Description	Usage Example
		that changed. See **tagPath**.	

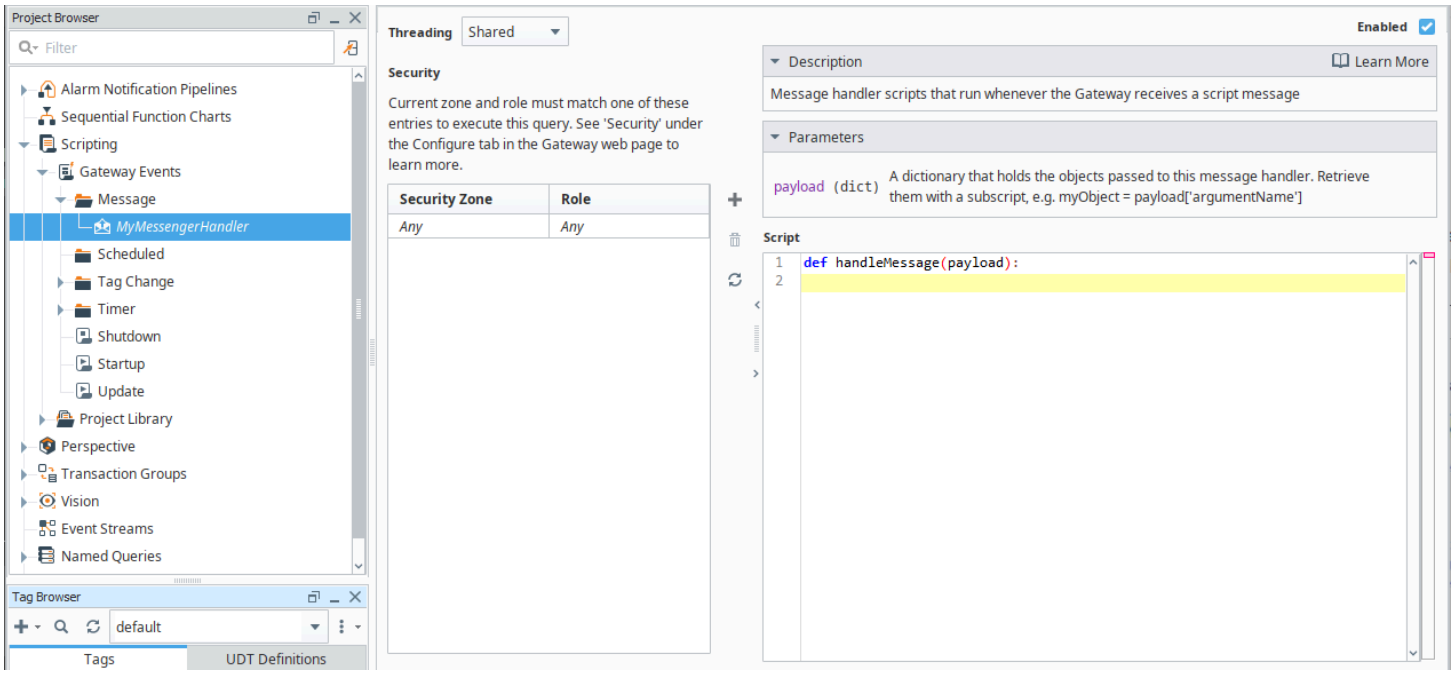
`tagPath`

Method/Attribute	Returns	Description
getItemName()	string	Returns the name of the item at the end of the tag path, which can be used to get the name of the tag that changed.
getParentPath()	BasicTagPath	Returns a BasicTagPath of the tag's parent folder. Python's str() method can be used to convert the path to a string.

Message Scripts

Message Handlers allow you to write a script that will run in the scope they are located in, but they can be invoked by making a call from other projects or even other Gateways. They can be called using three different scripting functions:

- **system.util.sendMessage**
- **system.util.sendRequest**
- **system.util.sendRequestAsync**



Select **New Message Script** from the Message folder right-click menu to name and add a script. Keep in mind that each message handler must have a unique name per project. Once created, use the right-click menu for each script to copy, rename, or delete the script. Selecting a script will open it so that it can be viewed and modified.

Gateway Message Handler Settings

When adding or modifying a message handler, the following settings are available:

- **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.
- **Security**: Allows you to specify security zone and role combinations that are allow to request this message handler.

The Payload

Inside the message handler is your script. The script will have a single object available to it, the **payload**. The payload is a dictionary containing the objects that were passed into it. In essence, the payload is the mechanism that allows you to pass the message handler values.

The payload is simply a python dictionary, so extracting values involves specifying the key:

Pseudocode - Payload Values

```
value1 = payload["MyFirstValue"] # "MyFirstValue" is the key that is associated with a value. We are taking the value associated with MyFirstValue, and assigning it to value1.
value2 = payload["MySecondValue"] # Similarly, we are taking the value associated with MySecondValue and assigning it to value2.
```

Calling Message Handlers

Once you have your message handlers created, you can then call them from a script using one of three scripting functions:

- **system.util.sendMessage**
- **system.util.sendRequest**
- **system.util.sendRequestAsync**.

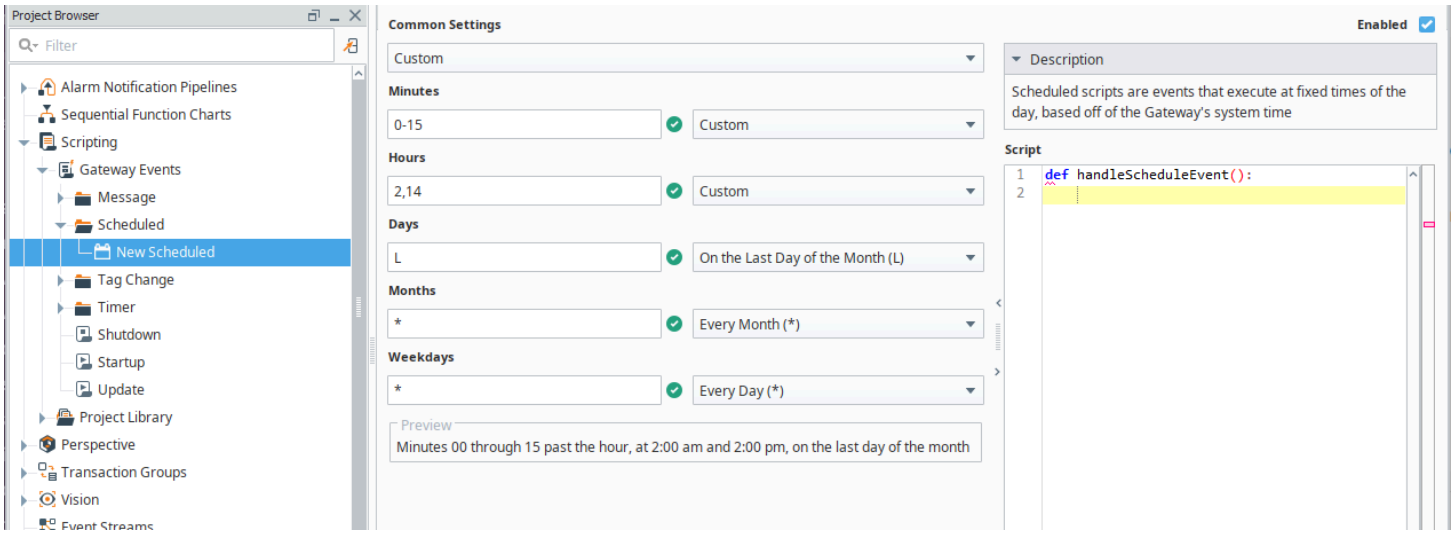
These functions allow you to call a message handler in any project, even if the project that the message handler resides on is different from the one you are calling it from. The message handler will then execute in the scope in which it was created, and will use any parameters that you pass in through the payload.

Pseudocode - Calling a Message Handler

```
project="test"
messageHandler="My Message Handler"
myDict = {'MyFirstValue': "Hello", 'MySecondValue': "World"}
results=system.util.sendMessage(project, messageHandler, myDict)
```

Gateway Scheduled Scripts

Scheduled scripts are events that execute at fixed times of the day, based off of the Gateway's system time. Configuration for the event is split between two sections: **Settings** and **Script**.



Settings

The Settings selections allow you to give the event a unique name and determine how often it executes. Schedules are driven by cron job scheduling.

The Common Settings dropdown contains several common selections, which can be further modified with the fields and dropdowns under each unit of time.

Each unit of time consists of a field, and corresponding dropdown. The dropdown is filled with suggestions and simple options, but more complex values can be provided in the field on the left. See **Crontab Formatting Reference**.

Script

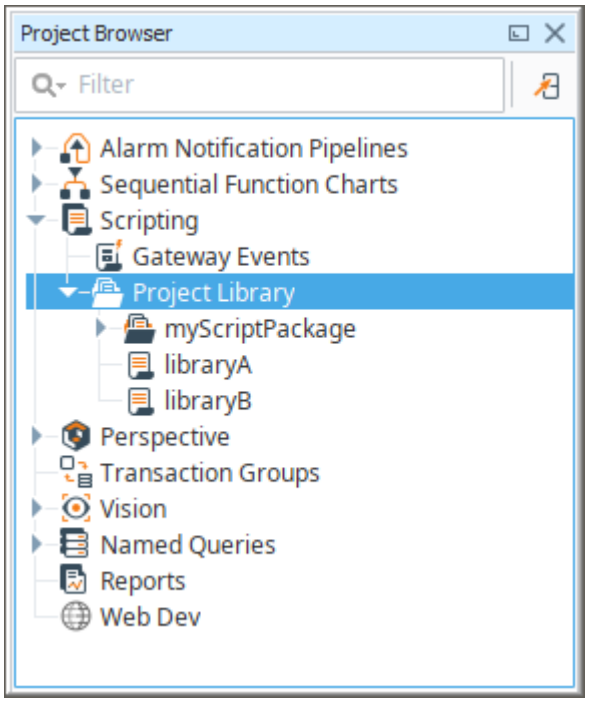
The Script field houses the Python script that will execute when the scheduled event executes.

Troubleshooting Gateway Scripts

While they are technically project resources, remember that Gateway Event Scripts run on the Gateway. Therefore, the **Diagnostics section of the Gateway** is useful for diagnosing issues with Gateway Event Scripts.

Project Library

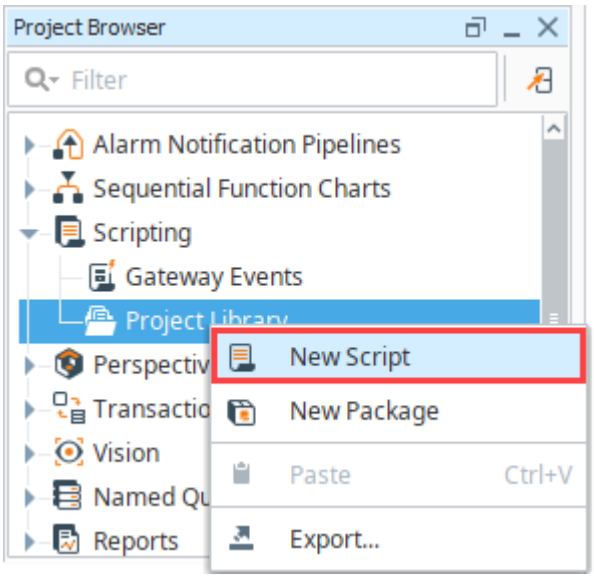
Scripts under the Project Library are a project-based resource that allows user created Python scripts to be configured. Objects and functions created in a Project Library script can be called from anywhere in the project. Project Library scripts are accessible from the Project Browser, under the **Scripting** item.



Additionally, a single project can be designated as the **Gateway Scripting Project**, meaning that scripts defined in the stated project can be called from the Gateway scope.

Add a Script

To add a a Project Library script, right click the **Project Library** item and click the **New Script** option.



Scripts and Packages

There are two main types of resources under the Project Library.

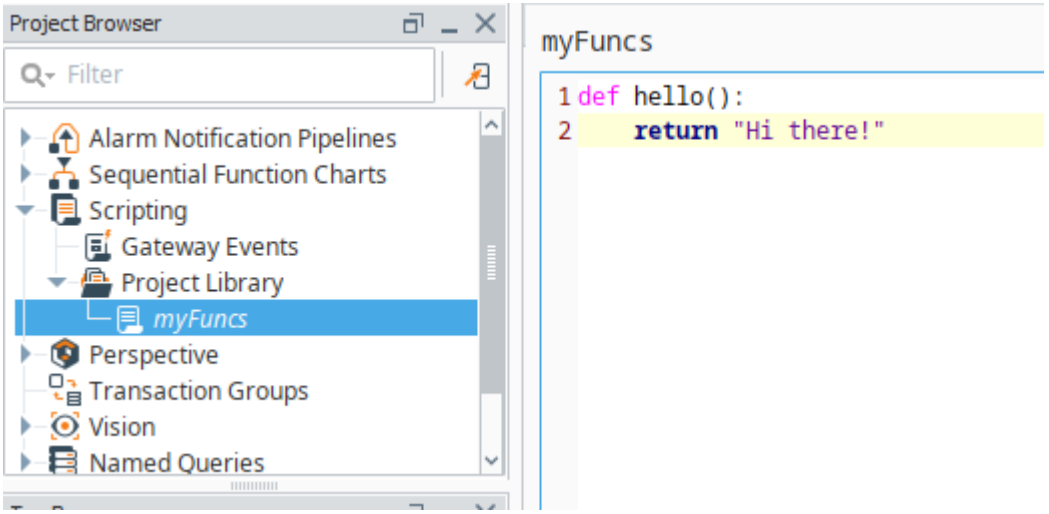
- **Scripts** - Each script resource can contain many **functions** and objects.
- **Packages** - Each package effectively acts as a folder, allowing you to better organize each script resource.

Usage Example

For example, let's suppose you added the following script module named **myFuncs**, whose body is shown below.

Python

```
def hello():
    return "Hi there!"
```



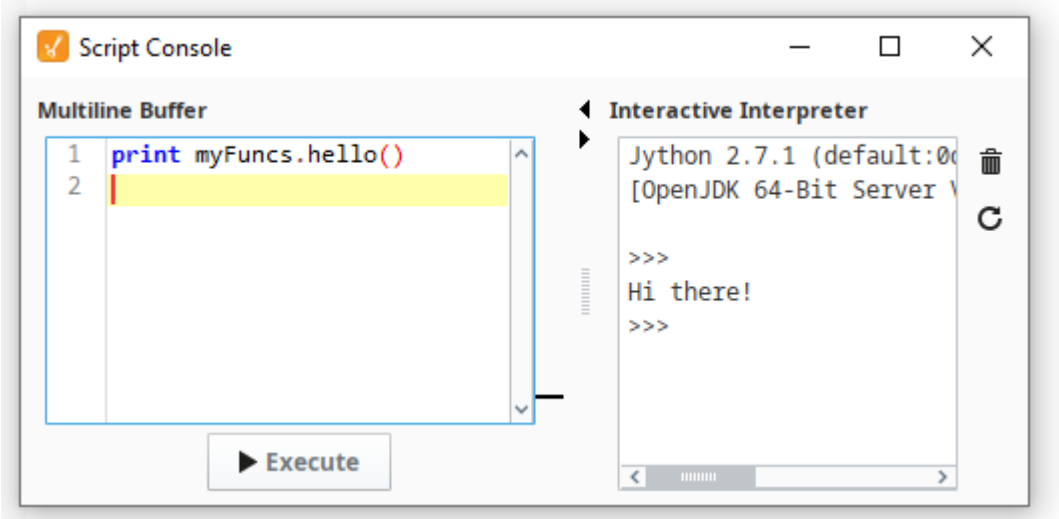
Once we **save** our project, we can now call this function from anywhere within the project using the following syntax.

Python - Calling the Project Script

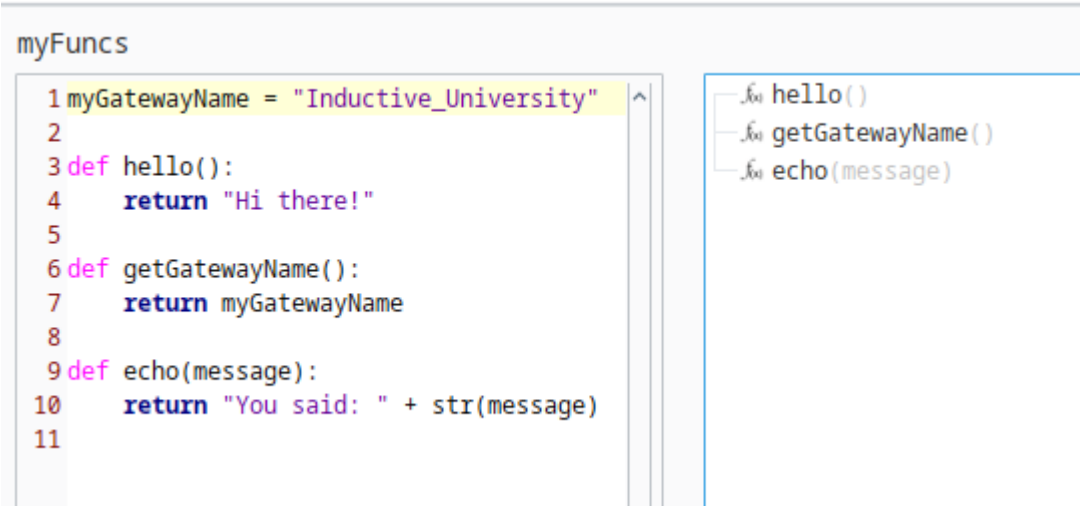
```
myFuncs.hello()
```

> **NOTE**
>
> Project Library scripts are not accessible to the other resources until the project is saved.

For example, we could open the Script Console (Tools menu > Script Console), write the following, and execute the script.



Each script resource can contain multiple functions and objects. As you add new function definitions, the list on the right will populate giving you a quick way to navigate through long scripts.



Project Libraries and Execution

Because Python is a dynamic language, any code inside of a project library must be run to build the function and class definitions. This is a common behavior across interpreters. Within Ignition, these project libraries will run under certain conditions. For example, such as when the Script Console in the Designer starts up, if changes are made to third-party libraries inside of the gateway's installation directory, when saving changes to these project libraries, and several other conditions.

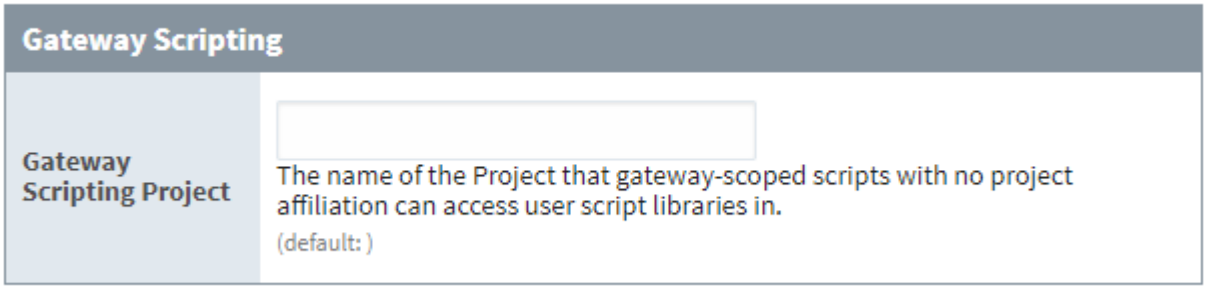
When this occurs, code inside of a project library is executed. Meaning classes and functions are defined, and any code that is not contained within either a class or function will execute.

Because of this process, it's generally recommended that all code within a project library is wrapped inside of a function or class definition.

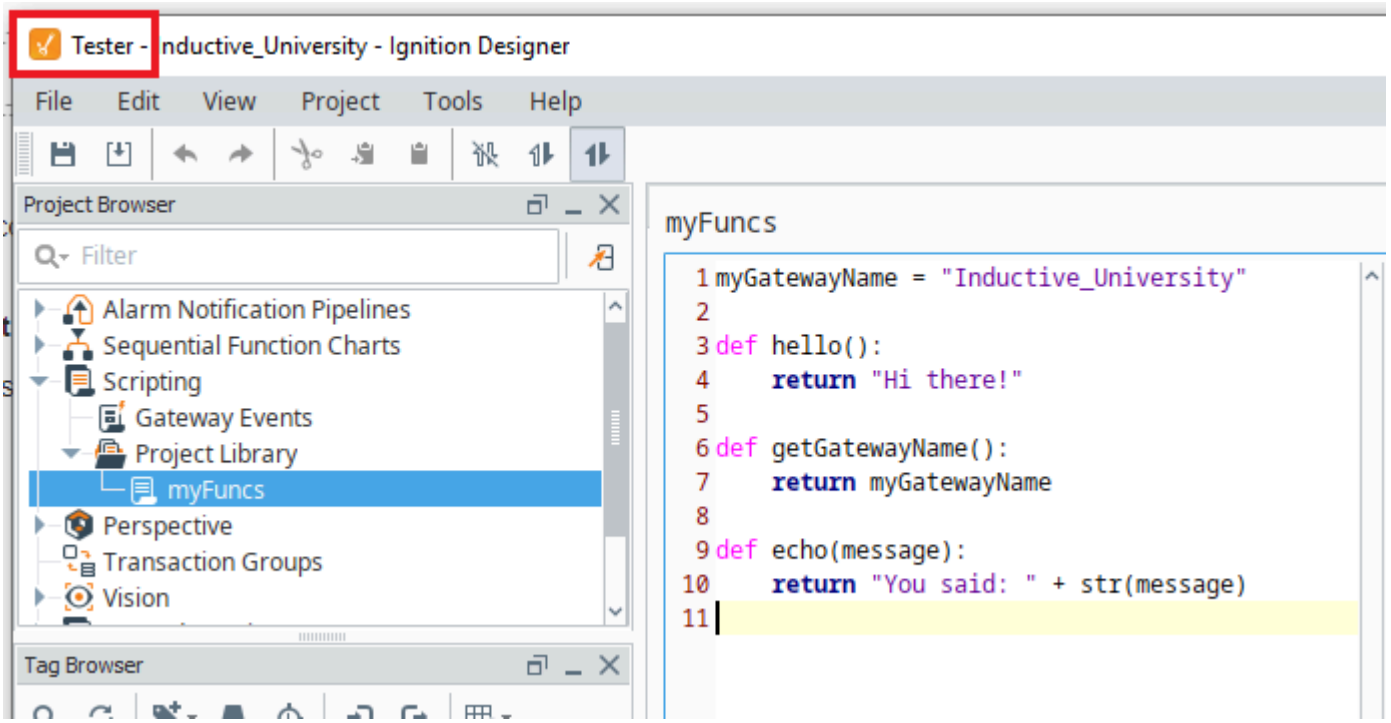
Gateway Scripting Project

Project Library scripts are normally only accessible from the project they were defined in. Thus objects that exist in other scopes, such as Tags that exist in the Gateway scope, are unable to call Project Library scripts. Attempting to do so will result in Gateway log errors stating that "global name 'yourScript' is not defined".

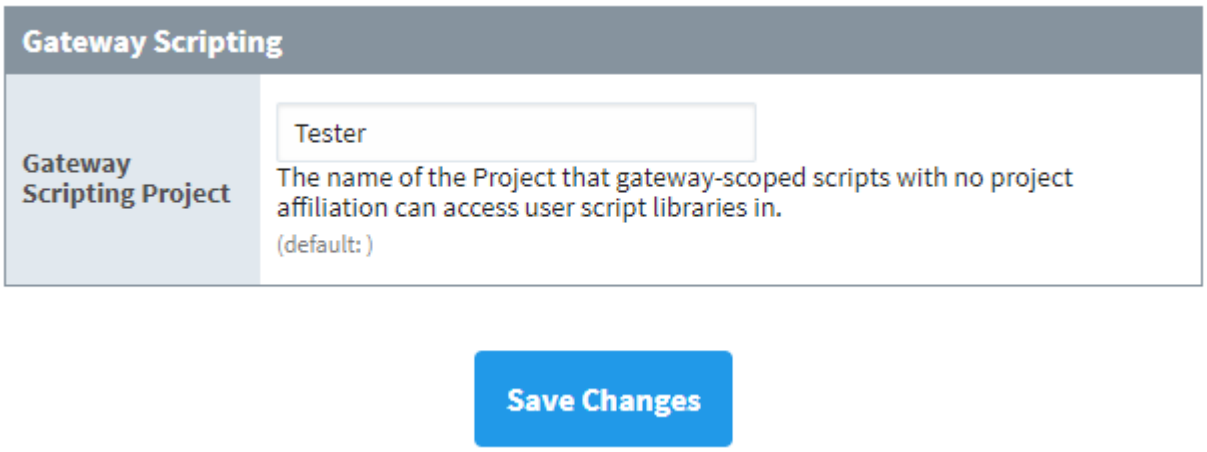
The exception to this rule is the Gateway Scripting Project. This project is specified by the **Gateway Scripting Project** property, which is set in the Config section of the Gateway Webpage under **Gateway Settings**. Entering the name of a project under this property allows the Gateway access to Project Library scripts configured in the specified project.



Thus, if the **myFuncs** library in the prior section was configured in a project named Tester, we would enter Tester in the Gateway Scripting Project field.



Once the name of the project is entered, select **Save Changes**.



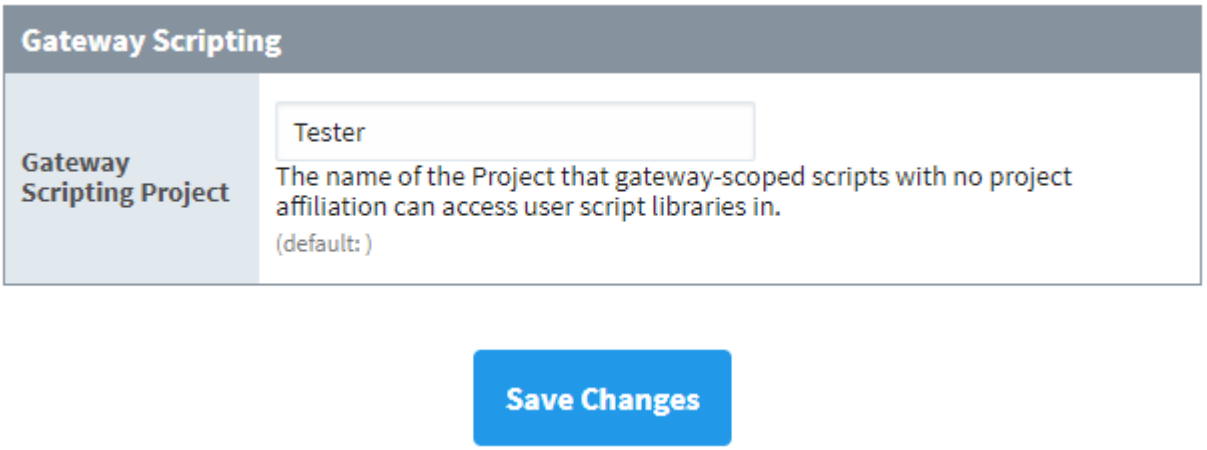
Now, you can start calling any of the global scripting events, such as tag event scripts or runScript calls, from the Tester Project Library.

> **NOTE**
>
> The Gateway Scripting Project has no impact to Gateway event scripts defined in the Designer (startup, shutdown, update, message handler, etc.) as those are Project Resources and thus only function for the project they run under.

Example

This example demonstrates how to call scripts in your Gateway Scripting Project from a tag.

1. Set up your Gateway Scripting Project under Config > Gateway Settings:



2. Open your Gateway Scripting Project and add a library called **myFuncs**.

3. Define a new function in this library called **thankYou:**

```
def thankYou(user):
    system.vision.showMessage("Thanks for noticing me, " + user)
```

4. Save your Gateway Scripting Project and exit.

5. Open a new project.

6. Choose a tag and configure a **Tag Event Script** on the Alarm Acknowledged event.

7. You can call any function within the **myFuncs** library in your Gateway Scripting Project:

```
myFuncs.thankYou(user = ackedBy)
```

> **NOTE**
>
> You do not need to specify the name of your Gateway Scripting Project when calling any of its functions.

Script Logging and Print Statement Output Reference

Just as each script has its own events based on location that trigger the script to run, output logging is also dependent on where the script is executed. This page details where to find your print statements and logs to assist with script testing and debug. Refer to the following list for how to find the output logging locations:

- **Script Console**: Accessed using the top navigation bar in the Designer, selecting Tools > Script Console.
- **Output Console**: Accessed using the top navigation bar in the Designer, selecting Tools > Output.
- **System logs**: Located on the Gateway Status page under Diagnostics > Logs.
- **Wrapper logs**: Located inside the logs folder in the Ignition installation directory.

Designer Execution

When executing a script anywhere other than Perspective and the Script Console in the Designer, `print` and `system.util.getLogger` will log to the Output Console.

Script Console

- `print` will log directly to the Multiline Buffer on the right-hand side.
- `system.util.getLogger()` or `system.util.invokeAsynchronous` will log to the Output Console, not the Script Console Multiline Buffer.

Perspective

Even when scripting in Perspective in the Designer, your script is being run on the Gateway. For this reason, it's recommended to use `system.perspective.print` instead of the bare `print` statement.

- `system.perspective.print` will log to the Output Console in the Designer.
- `print` will show on the wrapper.log text files.
- `system.util.getLogger()` will display on the Gateway system logs and the wrapper.log text files.

> **PERSPECTIVE SCRIPTING ERRORS**
>
> Certain errors from scripting are deliberately redirected to the Designer and don't log on the Gateway, so keep in mind that some errors from a design session will appear to be "Designer local" when they are not.

Gateway Execution

Gateway Event Scripts, Tag Event Scripts, SFCs, alarm pipeline scripts, and reporting scripts are all examples of scripts executing on the Gateway. In these scripts:

- `print` will show on the wrapper.log text files.
- `system.util.getLogger` will show on the Gateway system logs and the wrapper.log text files.
- `system.perspective.print` will not work (unless a Perspective session ID has been specified).

Perspective Session

- `system.perspective.print`
  - If the scope is set to **client**, it will log in the browser console wherever the session is running.
  - If the scope is set to **gateway**, the output is logged to the Gateway system logs.
  - If the scope is not specified, it will be passed as **client** when running in the Designer and **gateway** when running in a session.
- `print` will show on the wrapper.log text files.
- `system.util.getLogger` will show on the Gateway system logs and the wrapper.log text files.

Client Session

From a Client, the Help > Diagnostics popup window will display `print` statements in the Console tab.

Web Services, SUDS, and REST

Web Services Overview

Web services are software solutions that allow for interacting with machines residing on a network. In short, web services are nothing more than web pages for machines. They provide a standard way for a third party to request and receive data from a piece of hardware on the network without having to know anything about how that machine works.

Protocols

There are two common approaches to Web Services in Ignition: making a HTTP (HyperText Transfer Protocol) method call, or making a SOAP (Simple Object Access Protocol) call.

- HTTP methods, such as GET and POST, are called using the built-in system function **system.net.httpClient**. More information on HTTP calls can be found on the **HTTP Methods page**.
- SOAP is XML based, and typically requires a third party Python library. There are **many third party Python libraries that utilize SOAP**, however, documenting them goes outside the scope of this manual (as do all third party libraries).

The approach you choose typically depends on the server you're trying to make calls to: more specifically, the protocol(s) it supports.

> **WHAT ABOUT THE SUDS LIBRARY?**
>
> The SUDS library, a library that used to come included with the Python Standard Library, offered SOAP based functionality. However, SUDS development has been halted, and is no longer included in the standard library.
>
> In the interest of posterity, the legacy SUDS documentation has been condensed and can be found on the **SUDS - Library Overview page**. Note that the legacy documentation should be considered deprecated.

Can Ignition Make RESTful Calls?

Yes it can! However, it is important to understand that REST is an architecture, NOT a protocol. Instead, REST utilizes and describes how a protocol should be used. Thus, a RESTful architecture could use both of the protocols mentioned on this page, although HTTP is far more common.

Common Web Services Workflow

While all Web Services follow the same standards, they all do different things. They wouldn't be worth anything if you didn't get the information you need, or if they contained a lot of excess data. If you are unfamiliar with a particular Web Service, there are a few things that you can do to figure out what data is available and how to get it.

1. Identify a Web Service that you will be using. Usually the Web Service has an API somewhere documenting how requests should be made.
2. Write a script to pull some information from the Web Service. If using HTTP, this could mean starting with a GET call, where as SOAP would involve retrieving the WSDL (Web Services Description Language). In both cases, you may need to find a way to authenticate against the server (usually with some user credentials or an auth token, the API for the service would have more details).
3. Once you have the results from the GET/WSDL, identify the information or functions you want to use.
4. Write a script to use that function and return your values.
5. Parse the results and use them. This can be for display, saving to a database, or anything else you need.

> **NOTE**
>
> Web Services sometimes take a lot of time to return results, especially the first time they are called. If you put your Web Services script in a button, the client will freeze until the call is complete (this is because the event handlers are run on the GUI thread). It's a good idea to use **system.util.invokeAsynchronous()** or add a waiting image to your screen to let the user know Ignition is working as expected.

HTTP Methods

Overview

Web services calls typically require some protocol to make requests. HTTP is an incredibly common protocol, so this page will introduce how to incorporate these calls in a python script. Note that all of the examples on this page can be easily called with the Script Console, but can be utilized through some other means (the actionPerformed event on a Button).

Finding an Endpoint

Ignition doesn't natively expose an endpoint for web services calls, so we'll have to utilize a service of some sort for the examples on this page. Fortunately, there are many public services we can utilize, such as **OpenWeather**. From here, we can generate an endpoint to use. At the time of this writing, we're using the following endpoint URL template:

Endpoint URL Template

```
https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API key}
```

The following table describes the various parameters in the endpoint URL:

Parameter Key	Description
lat, lon	Geographical coordinates (latitude, lognitude).
appid	Your unique API key. **Note:** The code snippets on this page use "Your API key" as a placeholder for unique API keys. You will need to substitute the placeholder with your own API key for the code snippets to function as expected.

The following code snippets are using the geographical coordinates for the Inductive Automation corporate headquarters located in Folsom, CA.

Making the Call

To retrieve the results of this information in Ignition, we can use **system.net.httpClient()** to fetch the results of this call. We can try the following script in the **Scripting Console**:

Python - Creates a variable to Store the Endpoint and Retrieves Results

```
#Set the endpoint URL
url = "https://api.openweathermap.org/data/2.5/weather"

#Declare a variable for system function we are using
myClient = system.net.httpClient()

#Declare a variable and set the parameters for the endpoint URL
#Instead of specifying the API call parameters for the endpoint URL in the
beginning
response = myClient.get(url, {"lat":38.652330, "lon":-121.189773, "appid":"Your
API key"})

#Print the output
print response.getText()
```

Printing this results in the following Python string:

Python - Results

```
{"coord":{"lon":-121.1898,"lat":38.6523},"weather":[{"id":800,"main":"Clear","descrip
sky","icon":"01d"}],"base":"stations","main":
{"temp":310.34,"feels_like":310.11,"temp_min":308.88,"temp_max":312.74,"pressure":101
{"speed":4.12,"deg":290},"clouds":{"all":0},"dt":1654893775,"sys":
{"type":2,"id":2006213,"country":"US","sunrise":1654864794,"sunset":1654918128},"time
```

Parsing the Results

If we wanted to extract a single value ouf of the results, we have a number of approaches. One useful approach would be to turn this JSON string into a Python Dictionary. This way we can single out a key instead of using regex or looking for substrings (both valid approaches in their own regard).

When presented with a JSON string, we can call **system.util.jsonDecode()** to turn a JSON string into a native Python Object. Thus, we can modify our code to the following:

Python - Parsing the Results Code

```
#Set the endpoint URL
url = "https://api.openweathermap.org/data/2.5/weather"

#Declare a variable for system function we are using
myClient = system.net.httpClient()

#Declare a variable and set the parameters for the endpoint URL
response = myClient.get(url, {"lat":38.652330, "lon":-121.189773, "appid":"Your
API key"})

#Set a variable for the contents of the API call
results = response.getText()

# Convert the JSON string into a Python object. In this case, it results in a
Dictionary.
decodedDict = system.util.jsonDecode(results)

# Now we can treat the results like a nested dictionary, thus we can specify the
"weather" key,
# and then the nested "description" key to return a description of the current
weather conditions.
# "[0]" is needed to specify the index of the list before specifying the
"description" key.
print decodedDict.get("weather")[0].get("description")
```

Now we can easily retrieve a single value by specifying key names on the results. Printing this results in the following Python string (at the current time):

Python - Results

```
clear sky
```

Make the Results Human Readable

Now that we know how to extract the results, we should clean up the output of the GET call. The JSON string returned by the endpoint could potentially be long and cumbersome to read through for a human, but we can use Python's built-in **pprint** library to pretty print the results.

Python - Now with Pretty Print

```
```

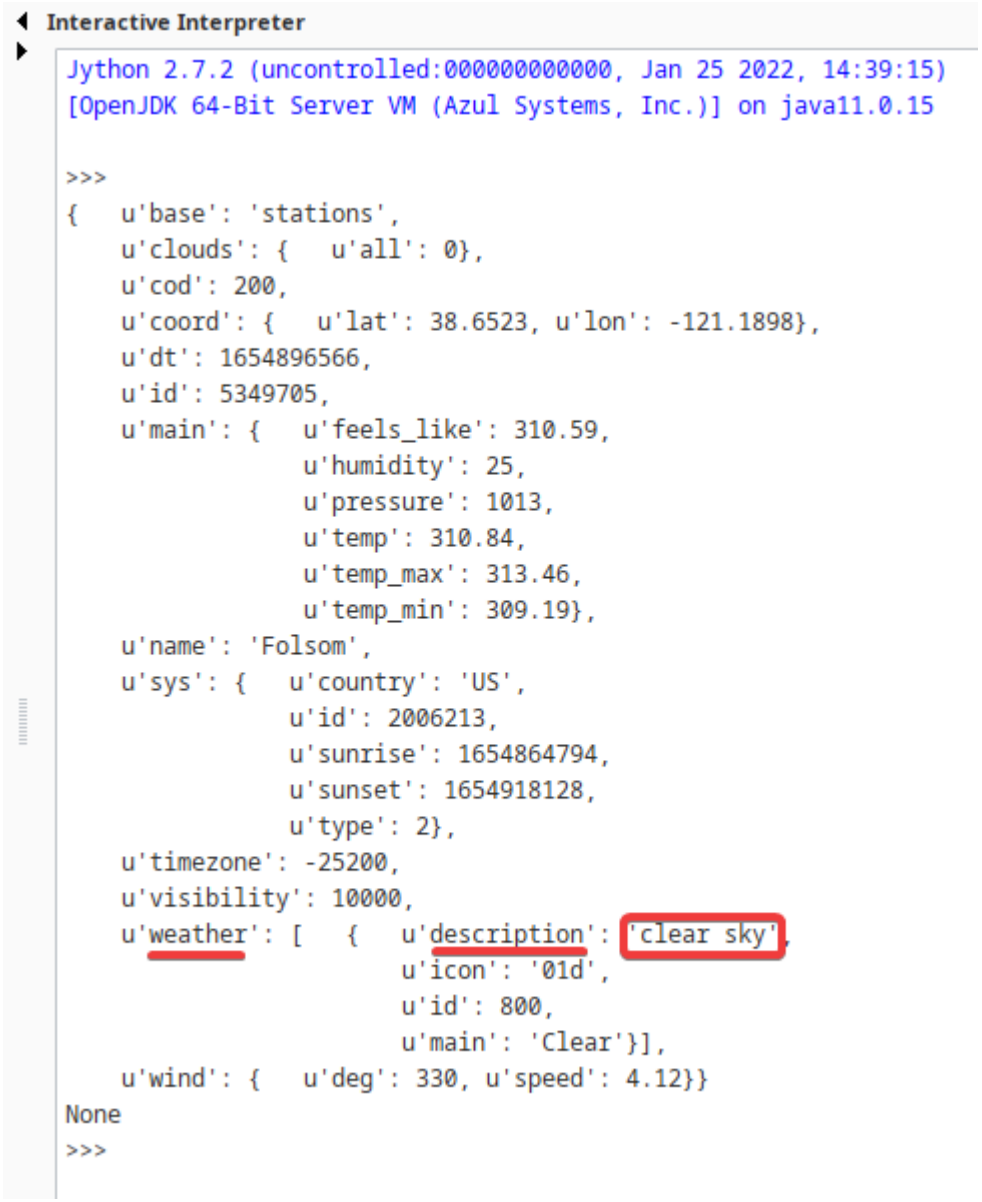
```
# Import the pprint library
import pprint

# We'll instantiate an instance of PrettyPrinter, and store it in a variable named
pp.
pp = pprint.PrettyPrinter(indent=4)

#Set the endpoint URL
url = "https://api.openweathermap.org/data/2.5/weather"

#Declare a variable for system function we are using
myClient = system.net.httpClient()

#Declare a variable and set the parameters for the endpoint URL
response = myClient.get(url, {"lat":38.652330, "lon":-121.189773, "appid":"Your
API key"})

# Convert the JSON string into a Python object. In this case, it results in a
Dictionary.
decodedDict = response.getJson()

# Print out the dictionary in an easy to read format.
pp.pprint(decodedDict)
```

The resulting output, which is much easier to read, looks like the following:

Python - Results

```
{   u'base': 'stations',
    u'clouds': {   u'all': 0},
    u'cod': 200,
    u'coord': {   u'lat': 38.6523, u'lon': -121.1898},
    u'dt': 1654896566,
    u'id': 5349705,
    u'main': {   u'feels_like': 310.59,
                 u'humidity': 25,
                 u'pressure': 1013,
                 u'temp': 310.84,
                 u'temp_max': 313.46,
                 u'temp_min': 309.19},
    u'name': 'Folsom',
    u'sys': {   u'country': 'US',
                u'id': 2006213,
                u'sunrise': 1654864794,
                u'sunset': 1654918128,
                u'type': 2},
    u'timezone': -25200,
    u'visibility': 10000,
    u'weather': [   {   u'description': 'clear sky',
                        u'icon': '01d',
                        u'id': 800,
                        u'main': 'Clear'}],
    u'wind': {   u'deg': 330, u'speed': 4.12}}
None
```

From here we can see all of the keys that lead to our final value:



To get to the value for the description key, we simply need to address each key along the way:

Python - To Get Value for the Sunset Key

```
# "[0]" is needed to specify the index of the list before specifying the
"description" key.
print decodedDict.get("weather")[0].get("description")
```

Troubleshooting HTTP Methods

When making HTTP calls, it is helpful to be familiar with the status codes that are returned by errors. To demonstrate, we could modify an earlier example:

Python - Returns Error Status Codes

```
#Set the endpoint URL
url = "https://api.openweathermap.org/data/2.5/weather"

#Declare a variable for system function we are using
myClient = system.net.httpClient()

### Note that instead of replacing the "appid" parameter with a unique API key, we
will leave it as the "Your API key" placeholder.
#Declare a variable and set the parameters for the endpoint URL
response = myClient.get(url, {"lat":38.652330, "lon":-121.189773, "appid":"Your
API key"})

#Print the output
print response.getText()
```

This will return an error, which looks like the following:

Python - IOError: Response Code 400

```
{"cod":401, "message": "Invalid API key. Please see
http://openweathermap.org/faq#error401 for more info."}
```

Note that HTTP response code **401**, which means **unauthorized**, was referenced. This error code is correct because we intentionally used an incorrect API key.

HTTP Response Codes

The World Wide Web Consortium has **a page dedicated to HTTP response codes**, which details all possible error codes. However, several common codes are listed below:

Response Code	Description
400	**Bad Request** - The server could not understand the request due to malformed syntax.
401	**Unauthorized** - The request requires some sort of authentication. Potentially some user credentials or an auth token of some kind.
403	**Forbidden** - The server understood what you requested, but is intentionally refusing the request. In some cases, the error message may include a reason why the request was not fulfilled (but not always). Typically, if the server doesn't include a reason, they'll use a 404 error code instead.
404	**Not Found** - Your syntax was correct, but the server could not find the resource you were asking for. This could mean a typo, or missing portion of the URL you are using. In this case, double check the address you're specifying. Depending on the configuration, this could also mean that the server does actually have the resource you requested, but doesn't want to confirm its existence (typically due to a security policy. See error code 403 above).

SUDS - Library Overview

The SUDS Library

The SUDS library is a SOAP-based web services client developed for Python. It is extremely simple to use and practically eliminates the need for the user to understand or even view the WSDL of a web service.

> ⚠ **DISCLAIMER**
>
> The SUDS library used to be included in the Python Standard Library. However it has since been removed, meaning you may not have access to it when performing a fresh install of Ignition. Additionally, development on the library has mostly ceased, so any copies you find online may be drastically outdated.
>
> The information on this page will be maintained for legacy users that need to be familiar with the old SUDS library. As a result, this page and its contents should be considered deprecated.

The SUDS library interprets the WSDL file for you and through a couple simple function calls allows you to get a list of the available methods provided to you by the web service. You can then invoke these methods in Ignition through event scripting to send and receive data in your projects. You will have to familiarize yourself with the SUDS library in order to make use of it.

A Simple Example

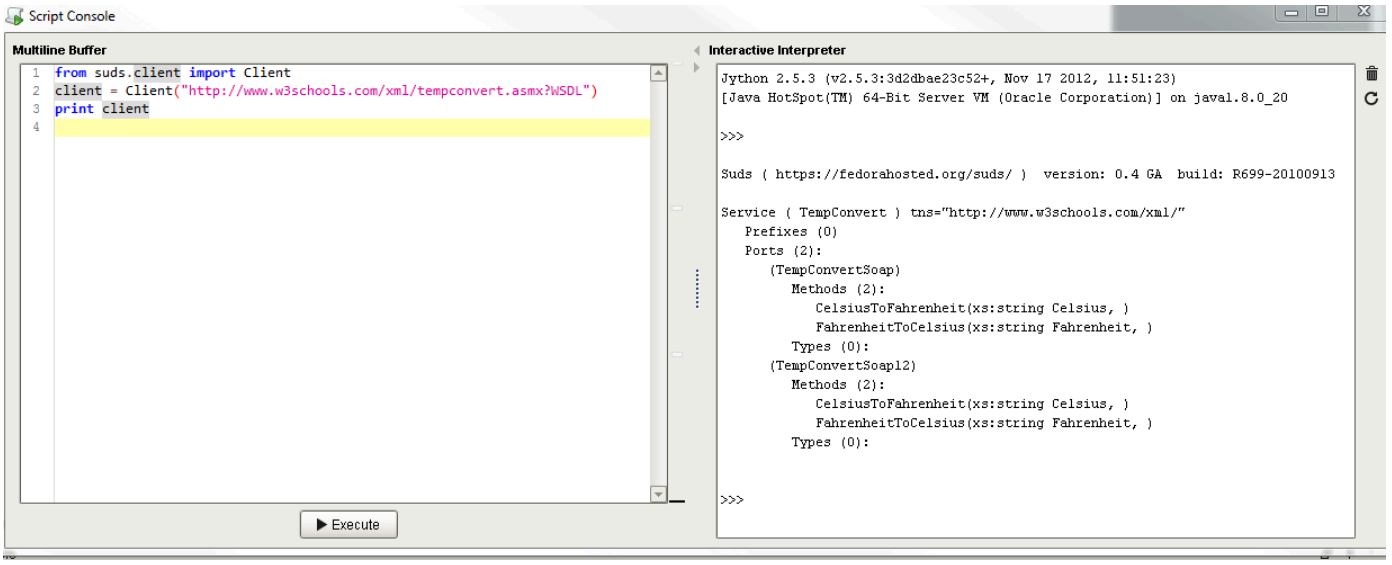
If you read through the SUDS documentation, you'll see that the Client object is the primary interface for most users. It is extremely simple using this object and a few print statements to view a list of available methods provided by the web service you are connecting to. This example will illustrate how to make an initial connect to a web service, print out the list of available methods, and then call one of these methods and display the resulting value in a label on an Ignition window at the push of a button. The below example uses a public web service and is available for anyone to test against.

1. First, we can use the script playground to test out some scripting calls and see the output. The below example shows how to get a reference to a client object. By printing this client object to the console we get an output of all the available methods, types, and other information about the web service as defined in the WSDL file.

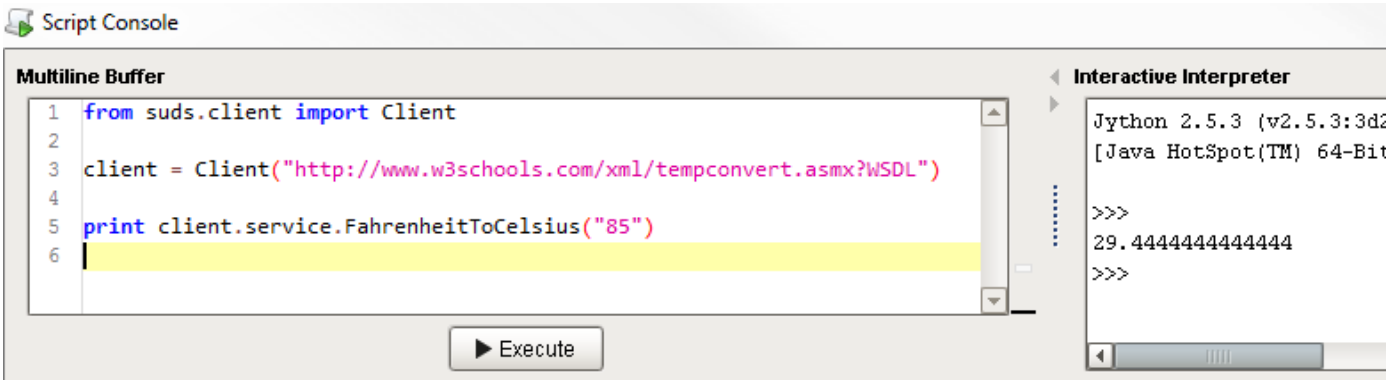
   Python - W3Schools WSDL

   ```
   ```

```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")
print client
```



This WSDL defines two functions: `CelsiusToFahrenheit(string tempCelsius)` and `FahrenheitToCelsius(string tempFahrenheit)`. These are the functions that this web service makes available to you. Don't worry about the fact that the methods are listed twice. This is just because the WSDL has two definitions of the functions that are formatted for different SOAP version standards. To call these functions in Ignition scripting, you have to make use of the "service" member of the client object. You can see printing the returned results shows the conversion:



2. To make a simple conversion window in an Ignition project you can add a button, a numeric textbox, and a label to a window. Then on the button to calculate a Fahrenheit to Celsius calculation, you would place something like the following:

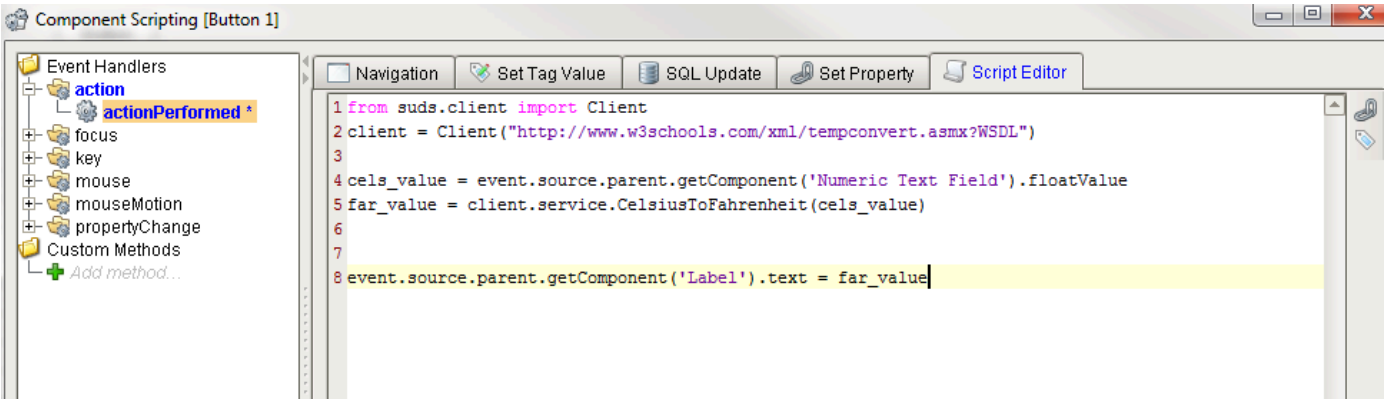
Python - Fahrenheit to Celsius

```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")

far_value = event.source.parent.getComponent('Numeric Text Field').floatValue
cels_value = client.service.FahrenheitToCelsius(far_value)

event.source.parent.getComponent('Label').text = cels_value
```

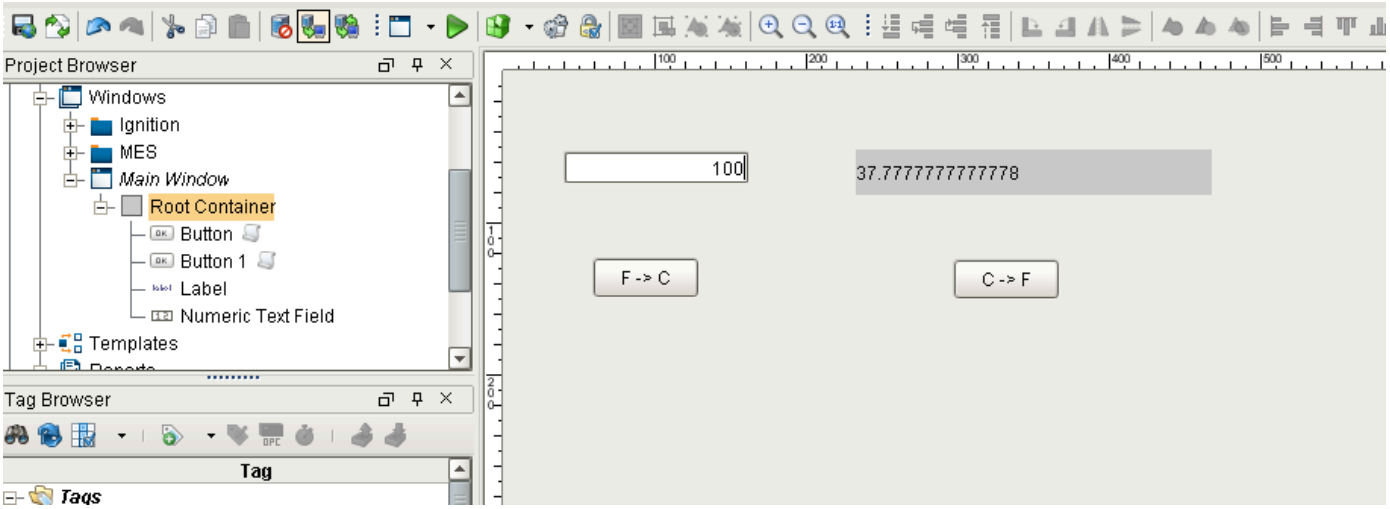
3. Then you can make a second button to do the opposite: calculate Celsius to Fahrenheit.

Python - Celsius to Fahrenheit

```
from suds.client import Client
client = Client("http://www.w3schools.com/xml/tempconvert.asmx?WSDL")

cels_value = event.source.parent.getComponent('Numeric Text Field').floatValue
far_value = client.service.CelsiusToFahrenheit(cels_value)

event.source.parent.getComponent('Label').text = far_value
```



4. With your scripts in place your window should now function as a simple temperature conversion tool!



Beyond the Example

While the example is relatively simple, it can easily be expanded upon. However, always keep the general workflow in mind when using the SUDS library:

Pseudocode - WSDL Workflow

```
#Import the SUDS Client object
from suds.client import Client

#Instantiate a new Client Object
client = Client("url_to_your_wsdl")

#Call the desired method using the service instance variable
client.service.MyMethod(myArgument)
```

Complex Arguments

In the overview, the methods provided by the web service were very simple and took simple argument types. Sometimes however, the web service will describe complex types and allow you create instances of these types that can then be added to the system/machine that the web service is providing an interface for.

A simple, hypothetical example of this would be a system that stores contact information of clients and can be used as an address book of sorts by other systems on the network. It may provide not only a way to pull contact information for a certain individual out, but also a way to insert new contacts. We'll keep the example simple and say that contacts have only a name and a phone number.

> **NOTE**

> This example is completely hypothetical. It is intended to give insight into complex types. It does not make use of an an actual functional web service.

For example, say we create and print the client object we associated with our web service in the following manner:

Pseudocode - Client Object

```
from suds.client import Client
url = 'http://localhost:7575/webservices/hypothetical_webservice?wsdl'
client = Client(url)
print client
```

And the resulting output is the following:

Python - Results

```
Suds ( https://fedorahosted.org/suds/ )  version: 0.4 GA  build: R699-20100913

Service (hypothetical_webservice)
   Prefixes (0):
   Ports (1):
     (Soap)
       Methods:
         addContact(Contact contact, )
         getContactList(xs:string str, xs:int length, )
         getContactByName(Name name, )
   Types (3):
     Contact
     Name
     Phone
```

Here you can see that, while not too complicated, the web service defines more than just methods that take simple type arguments and return the same. Under the Types section, you can see there are three "complex" types. These are basically just objects like one creates in an object oriented programming language like java. The SUDS Client object has an instance variable called "factory" that allows you to create these complex types so you can use them to invoke methods defined by your web service that take complex arguments.

If we wanted to add a contact using the addContact() method, we have to create a contact object first:

Pseudocode - Using a Method

```
contact = client.factory.create('Contact')
print contact
```

The create function creates a new contact object that knows its own structure. We can view this structure by calling print on this new object and see that it prints the following:

Python - Structure

```
(Contact)=
  {
    phone = []
    age = NONE
    name(Name) =
        {
            last = NONE
            first = NONE
        }
   }
```

By examining the Contact type object, we can see its structure and know what we need to create in order to have a valid Contact to add to the address book. We could then do the following to supply the necessary information for the Contact object and then call our addContact function.

Pseudocode - Adding a Contact

```
contact = client.factory.create('Contact')

phone= client.factory.create('Phone')
phone.areacode = '916'
phone.number = '5557777'

name = client.factory.create('Name')
name.first = 'John'
name.last = 'Doe'

contact.name = name
contact.phone = phone
contact.age = 30

client.service.addContact(contact)
```

After execution a new contact will have been added via the web service.

Steps to remember when using complex types:

Pseudocode - Complex Type Reminders

```
#Create a new type object using the factory instance variable of the Client object
my_type = client.factory.create('MyType')

#If you don't know the structure of the newly created object then print it to the
console
print my_type
```

JSON Format

About JSON

JavaScript Object Notation (JSON) is a language-independent data format. It's a lightweight format for storing and transporting data (i.e., when data is sent from a server to a web page). JSON is easy for users to read and write and for machines to parse and generate.

JSON Rules

JSON has a few simple rules:

- Data is in name/value pairs.
- Data is separated by commas.
- Curly braces hold objects.
- Square brackets hold arrays.

How JSON Works

JSON Data - Name and a Value

JSON data is written as name/value pairs. A name/value pair consists of a field name in double quotes, followed by a colon, followed by a value.

```
{
    "companyName":"Inductive Automation"
}
```

JSON Objects

JSON objects are written inside curly braces.

> **NOTE**
>
> The properties of a JSON object have no defined order. If you need a defined order, use a dataset or an array instead.

```
{
    "firstName":"Sally",
    "lastName":"Smith"
}
```

JSON Arrays

JSON Arrays are written inside square brackets. An array can contain objects. In the following example, the object "companies" is an array and contains three objects.

```
{
   "companies":[
      {
         "companyName":"Inductive Automation",
         "cityName":"Folsom",
         "stateName":"CA"
      },
      {
         "companyName":"Hewlett Packard",
         "cityName":"Palo Alto",
         "stateName":"CA"
      },
      {
         "companyName":"Apple",
         "cityName":"Cupertino",
         "stateName":"CA"
      }
   ]
}
```

Where JSON Is Used in Ignition

Ignition uses the JSON format to store much of its data internally, including Tags and Perspective component properties.

Perspective Component Properties

Components have properties (props), which are simply named values. These properties are arranged in a tree structure following the structure and data model of the common JSON document format. Component properties are defined as a JSON structure, and are variable according to the type of component that the config object represents. All components registered in the module have a set of

default properties included. These defaults are provided to the instantiated component at runtime, and so default props are not saved. Instead, only those which have a value that differs from the default are stored, serialized and sent to the client during loading.

Component Property Example in JSON Format:

Example - Sample Data from Table Component


```
[ { "city": "Helsinki", "country": "Finland", "population": 635591 }, { "city": "Jakarta", "country": "Indonesia", "population": 10187595 }, { "city": "Madrid", "country": "Spain", "population": 3233527 }, { "city": "Prague", "country": "Czech Republic", "population": 1241664 }, { "city": "San Diego", "country": "United States", "population": 1406630 }, { "city": "Tunis", "country": "Tunisia", "population": 1056247 } ]
```


Tags

Ignition exports and imports Tag configurations to and from JSON. Tags are defined as JSON objects, which consist of properties, arrays, and sub-objects. The system.tag.configure function can take either a String document definition, or a JSON object that defines one or more Tags. Overrides for UDTs are created by simple redefinition of properties, and complex structures like Event Scripts and Alarm configurations will be merged with inherited definitions.

You can copy the JSON or one or or more Tags in the Tag Browser. This copies them into the system clipboard. In addition, pasting the JSON into a different provider/designer will create or overwrite tags.

Tag Export Examples in JSON Format:

Example 1 - Tag Export

```
{
  "name": "Tank Instance",
  "typeId": "Tank UDT",
  "tagType": "UdtInstance",
  "tags": [
    {
      "value": "80",
      "name": "Tank Level",
      "tagType": "AtomicTag"
    },
    {
      "value": 80,
      "name": "sliderValue",
      "tagType": "AtomicTag"
    }
  ]
}
```

Example 2 - Tag Export

```
{
  "tags": [
    {
      "valueSource": "memory",
      "dataType": "Boolean",
      "alarms": [
        {
          "setpointA": 1,
          "name": "Above Normal"
        }
      ],
      "name": "Boolean Tag",
      "value": false,
      "tagType": "AtomicTag"
    },
    {
      "valueSource": "memory",
      "dataType": "Boolean",
      "name": "One Shot Trigger",
      "tagGroup": "Driven One Shot",
      "value": true,
      "tagType": "AtomicTag",
      "enabled": true
    },
    {
      "valueSource": "opc",
      "opcItemPath": "ns\u003d1;s\u003d[Generic]_Meta:Random/RandomDouble1",
      "dataType": "Float8",
      "name": "Pressure3",
      "tagGroup": "Driven One Shot",
      "tagType": "AtomicTag",
      "enabled": true,
      "opcServer": "Ignition OPC UA Server"
    },
    {
      "valueSource": "opc",
      "opcItemPath": "ns\u003d1;s\u003d[Generic]_Meta:Random/RandomDouble2",
      "dataType": "Float8",
      "name": "Thickness3",
      "tagGroup": "Driven One Shot",
      "tagType": "AtomicTag",
      "opcServer": "Ignition OPC UA Server"
    }
  ]
}
```

JSON in Tag UDTs

You can also set JSON strings as properties in an Ignition Tag. Any properties of a Tag can be set to a string that represents a JSON object.

> **NOTE**
>
> In a UDT, the braces are used to denote a property reference. Make sure you don't have any properties with names that look like JSON objects so there is no overlap. Strings will appear as black text, parameter references will appear as grey and italicized.

Looping through JSON Objects with Scripting

Traversing a JSON object in scripting is simple as long as the structure is known. If there are objects within objects, you can use multiple loops to get through it all.

Let's use the JSON Array object above for a simple example. You can loop through the list to get the repeating items by name.

```
# loop through the JSON data
# fetch the data, this will change depending on where the script is in relation to
the table
json = self.items
# access the companies object (which is a list)
companies = json["companies"]

# loop through the companies list
for company in companies :
    # get each item out of the row object
    name = company["companyName"]
    city = company["cityName"]
    state= company["stateName"]
    # now do something with the data
```

Basic Python Troubleshooting

When learning how to code in Python, most Ignition users tend to place most of their learning efforts on memorizing syntax or other aspects of the language. While being comfortable with the language is useful, there are plenty of references available: countless books and websites that describe syntax and usage already exist.

In truth, the best thing you can do to make yourself a better programmer is to learn some basic troubleshooting behaviors. While syntax examples are all over the Internet, examples detailing exactly what you want your code to do, to the extent you want it to, will be difficult if not impossible to find.

This section details how to troubleshoot a script in Ignition. You won't walk away from this section having exact answers to specific problems, but rather examples and concepts that you can apply to your own scripts.

Coding Best Practices

The following are some general and helpful best practices that can help minimize the amount of time you spend troubleshooting, as they can help better direct you to a problem.

Look for Errors

When code fails mid-execution, it always generates an error message. Where the message appears depends on where the script executed:

Scope	Print Command	Print Output Location
Designer	Depends. See the output location.	The Designer reports errors in a similar manner to Clients: errors appear in the Designer's Console (**Ctrl + Shift + c**), Component Event Handlers will generate a red Error Message. Note that events some events, such as Client Startup Scripts, will not trigger in the Designer, so get in the habit of launching a Client when testing non-component scripting events. While in the Designer, and working on Perspective resources, calls to system.perspective.print will appear in the Designer's Console as well.
Gateway *Note: anything that isn't in the Client or Session scopes listed below uses this scope*	**system.util.getLogger**	Scripts on Gateway-scoped resources (Tags, Alarm Pipelines, SFCs, etc.,) will appear on the Logs page of the Status section on the Gateway. Additionally, the wrapper.log file in Ignition's installation directory will have these messages. Here are the default wrapper.log file paths for each operating system: • **Windows**: Program Files/Inductive Automation/Ignition/logs • **Linux**: /var/log/ignition • **Mac OS X**: /Users/UserName/Documents/Ignition-osx-x.x.x/logs
Perspective	**system.perspective.print**	Web browsers generally offer a way to inspect a page, which usually contains a console of some sort.
Vision	Python's `print` command, or **system.util.getLogger**	The Client Console will contain any errors generated in the client: press **Ctrl + Shift + F7** to open the console, or using the menubar in the client to go to **Help > Diagnostics** then click on the **Console Tab**.

Scope	Print Command	Print Output Location
		Additionally, a red Error Box should appear with details on the error if a Component Event Handler threw the exception: Extension Functions do not generate the Error Box. If you don't see the error box then it might be minimized, or open in the background (behind the Client).

What happens if I don't see any errors?

If there truly isn't an error message somewhere, but your code isn't doing what you expected, then ask yourself the following questions:

1. What is your script supposed to do?
2. What is it actually doing?

Question #2 is harder to answer: if you knew what it was doing, you wouldn't be stuck! The best way to answer this question is by adding `print` statements to your code.

When a script doesn't perform to expectations, it can suggest a problem with the script's workflow. Some of the **pages in this section** can offer some suggestions on what to do.

If You Find an Error, Read It!

It is common for users that are new to scripting to see an error message, immediately close it without much thought, and then stare at their code as if the problem will politely make it self known. While looking over your code line-by-line will eventually lead you to the problem, the error messages can provide you with a shortcut to the issue. **Check out some of the pages** in this section for more information on reading an error message.

Use Print Statements

When testing your scripts, the print command can help you verify that your code is behaving the way it should. This allows you to reconstruct what your code did when it executed. Don't be afraid to add helpful print statements to your code.

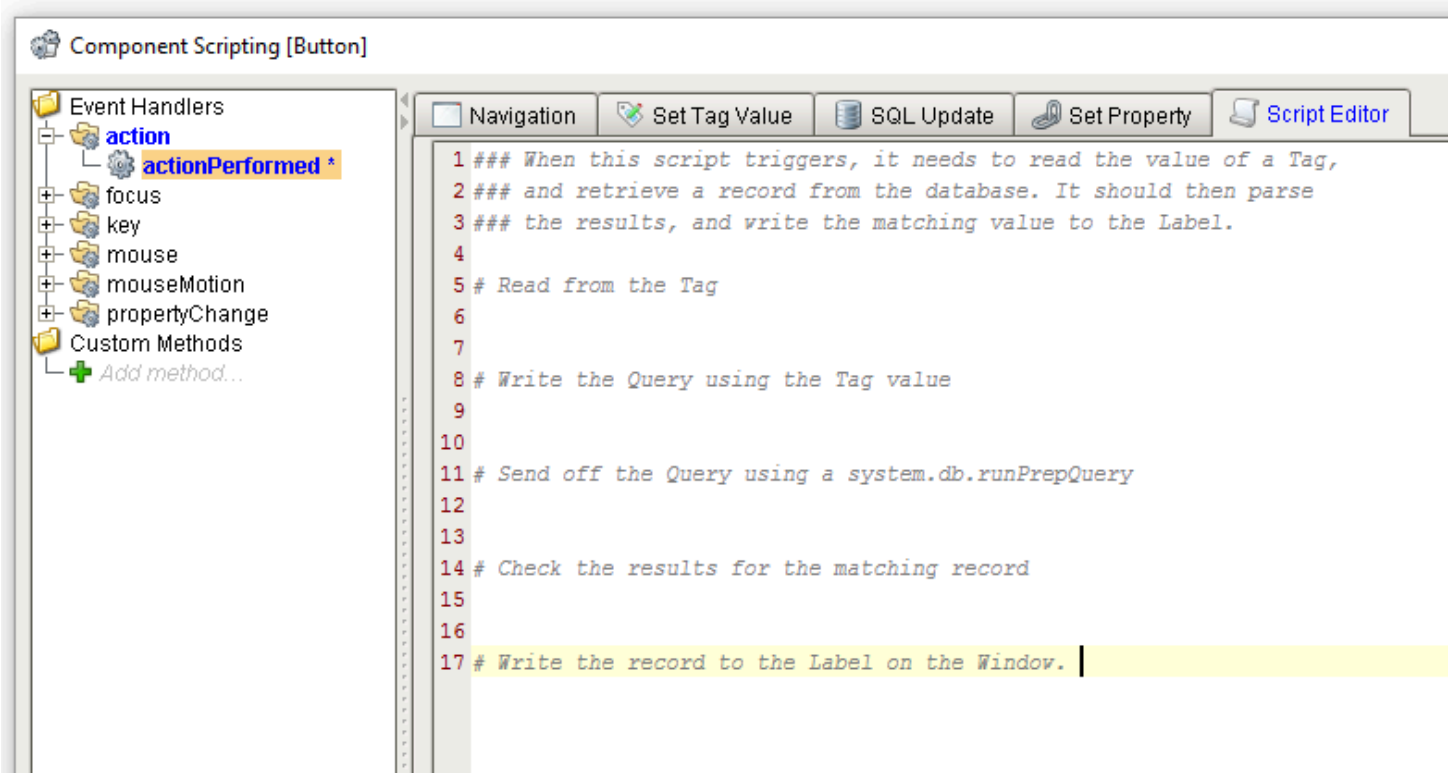
Once your code works as expected, remember to either remove or comment out the print statements, so they don't flood the console during normal use as this makes troubleshooting other issue more difficult.

Python - Using print to Troubleshoot

```
# Read returns a list of QualifiedValue objects in 8.3, so grab the value
myVar = system.tag.readBlocking(["[default]folder/tag"])[0].value

# Printing out variables you are going to use in an if-statement later
# allows you to confirm that the values are what you expect them to be.
print "myVar is set to: " + str(myVar)

# Sometimes viewing the data type of the variable can prove helpful.
print "myVar is a: " + str(type(myVar))

# Should your code have multiple if-statements, adding a print statement
# before and after can show you where the flow of the script went.
print "Starting if-statement"

if myVar > 100:
    # If you don't see this print statement, then your if-statement evaluated to
False.
    print "Inside if-statement"
    doWork()

# Printing the end of your script doesn't give you any useful troubleshooting
information,
# but if you need to trigger the script multiple times, it helps delineate each
execution.
print "Script Ended"
```

Add Comments

While comments are useful to remind you how your code works, you can also use them to plan your script before you write any code. Break down what you want the code to do into several smaller steps, and then leave comments describing those steps in order. This provides you a chance to review the script's workflow before worrying about syntax. It also provides natural points to stop and test your code, to make sure it is doing what you think it should do.



Test Early, Test Often

Unless the script is very simple, avoid writing the entire script and then testing at the end. You may have missed an important line early on, so now you have to adjust all of the code below that line to make the script run. As mentioned above, add print statements, and run your script to make sure it is doing what you think it should.

Additionally, stopping to test your code provides an excellent opportunity to **save your project**. Get into the habit of saving before you execute any new code.

Avoid Hard-Coding Arguments: Use Variables Instead

Instead of doing this:

Hard Coded Message Box

```
system.vision.showMessage("Hello you. Glad to see you")
```

Try to get into the habit of doing this:

```
message = "Hello you. Glad to see you"
system.vision.showMessage(message)
```

Simple examples like the above don't make for the best use case, but when you have a large script that references a value multiple times, it is easier to declare the value once in a variable, and then just reference the variable throughout your code. If you need to change the value later, then you can simply change it once where you initialize the variable, and you don't have to search every line of your code looking for the value.

Decide on a Naming Convention

When creating variables, try to adopt a naming convention that comes natural to you, and stick to it. If you consistently write variables using the same conventions, you will be less likely to end up with a typo when referencing that variable later in your code. Remember that code is case-sensitive, so something as simple as forgetting to capitalize a letter will cause an error.

This is especially important if you are working in a group on the same project. It's better to get together with your colleagues and agree upon some naming conventions before you write any code.

Ignition's *system. functions** use the Camel Case (camelCase) naming convention. That is, the first letter of each word is capitalized except for the very first letter. We recommend that you use it for variable names because It is easy to remember to use camelCase for both functions and variable names instead of for just one.

The Simplest Approach Really Is the Best Approach

When learning how to code, it's not uncommon to run into multiple issues that require you to find a workaround. However, these workarounds can cascade into other issues and make your script more complicated. Consider the following:

> "The goal of my script is to access A and then output B...
> Shoot, B requires C, so I'll add C...
> Wait, C requires that D exists, so I'll create that...
> Oh, D needs interfaces E, F, and G, so let's add those in...
> Hmm. E needs H and I, F needs J and K, while G needs Y, Z and...A **again?!?**"

Take a step back and ask yourself "what is this script doing"? If you can't do that in a sentence or two, you may want to rethink the script. If the scope of the script is too large, then it considerably increases the complexity of the code, which in turn could add a plethora of problems later.

If you keep adding workarounds, but your code is not getting any closer to achieving its end goal, there may be an easier way to accomplish what you're trying to do with a different approach.

Reading Error Messages

When an error occurs in the execution of a script, an error message box will pop up. The popup box appears in front of any open Designer windows, and will remain in view until you close it or click on something behind it.

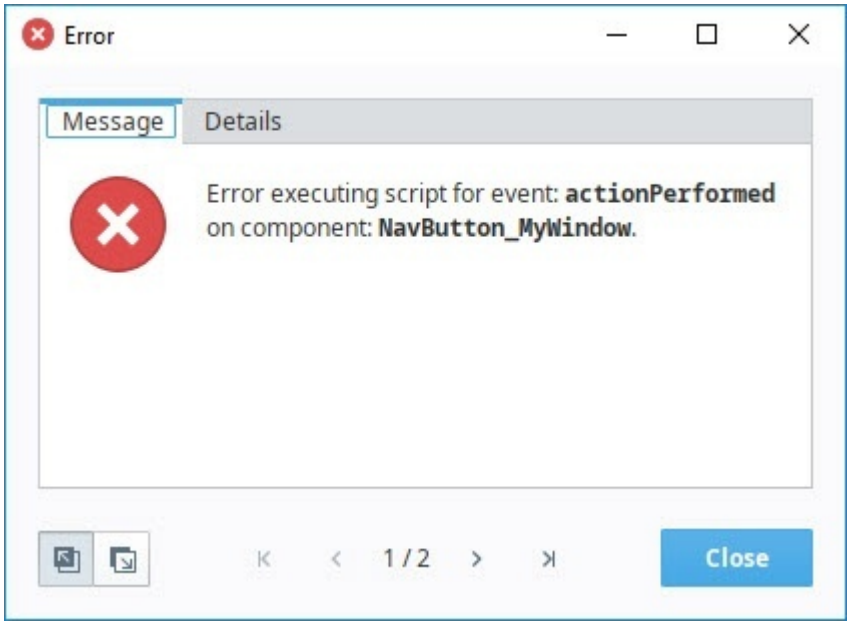
There are two modes for the Error box selectable by the **Send to Front** and **Send to Back** icons.

- **Send to Front** means additional error messages will cause the popup to reappear on top.
- **Send to Back** will cause the errors to remain hidden below the Designer.

Error Message Box Overview

Exceptions usually include a line number. Take note of the number in the Details tab, and start your search for the problem there. Be aware, however, that the line reported may not be the cause of the issue. The actual problem may be higher up in the code due to a faulty initialization, or some other issue. When troubleshooting, always start looking at the line reported, and work your way back towards the top.

When testing a script, you may come across an Error Message Box like the following. Your first inclination may be to close the error without closer examination: resist it!

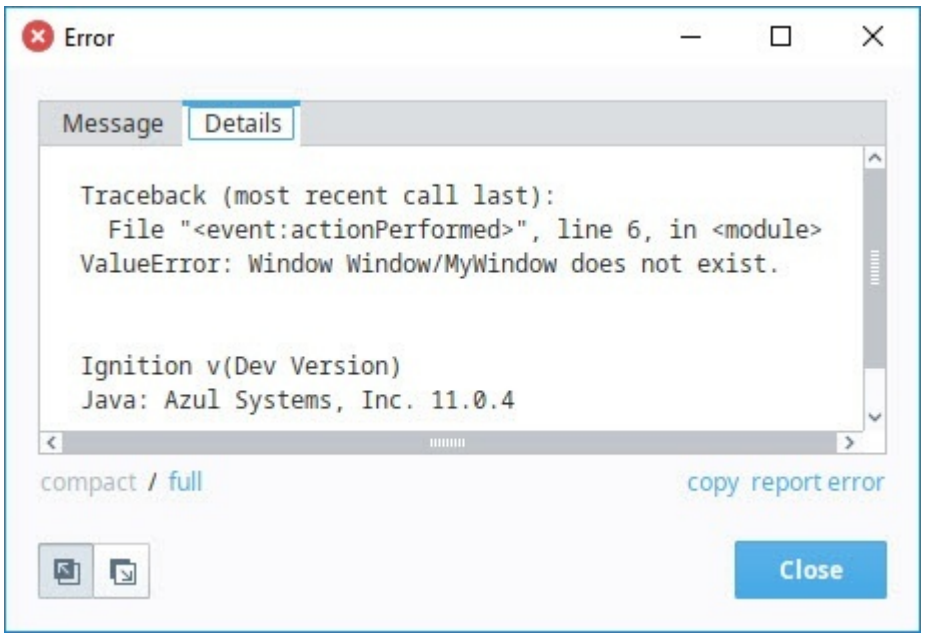


The error messages generated from failed script executions are incredibly helpful. The **Message** tab on the error describes the **Event Handler** that encountered the exception; in this case, the

actionPerformed event handler. This is important in cases where multiple Event Handlers on the same component have scripts. Without knowing which Event Handler generated the script, we could waste time trying to troubleshoot the wrong script.

Additionally, it shows the name of the component, which is "NavButton_MyWindow". Again, this is helpful if multiple scripts from multiple components are triggering in quick succession. The **Message** tab will clearly point you towards the source of the exception. This is another reason to give meaningful names to your components: doing so makes the process of tracking an error much easier.

The Details tab has even more information, specifically the line number that the exception occurred on, as well as the error message. The message here states that the window at path "MyWindow" does not exist, so we can check the Project Browser to see if we simply mistyped the name of the window.



Troubleshooting Errors Using the Error Message Box

When we don't get our expected results from our script, as in the following examples, always read both the Message and Details tabs in the Error Message Box. They are pretty good about pointing you to the root cause of your error. From there, you can easily find and fix any errors in your script.

The following examples show how to troubleshoot some of these error messages from the Error Message Box. Keep in mind that every script is unique, but at least you'll become familiar with what some of the error messages mean, and gain a little insight of the troubleshooting process.

Broken Example - Incorrect Attribute

In the following script, when a Button is pressed a Message Box is supposed to open on a window and display "Hello World".

Python - Broken: Incorrect Attribute

```
system.vision.showmessage("Hello World")
```

From the error message box, you know two important things right away:

- The script is using the actionPerformed event on the Hello Button.
- Line 1 displays the description of the error.

Check to be sure you used the correct attribute. Then check the spelling, case sensitive letters, spacing, and operators in your script. If you are familiar with Ignition's **Built-in Scripting Functions**, you can probably spot the error immediately. If not, you might want to use the **autocompletion popup feature** to retype your `showMessage` scripting function. This will automatically fix any syntax errors.

The error in this example is in the attribute name. It uses case sensitive letters (i.e., **showMessage**).

In this case, "showMessage" must be spelled with a capital "M" since we're trying to use Ignition's **system.vision.showMessage** function. We corrected the script by changing our code to the following:

Python - Corrected: Incorrect Attribute

```
system.vision.showMessage("Hello World")
```

Broken Example - Undefined

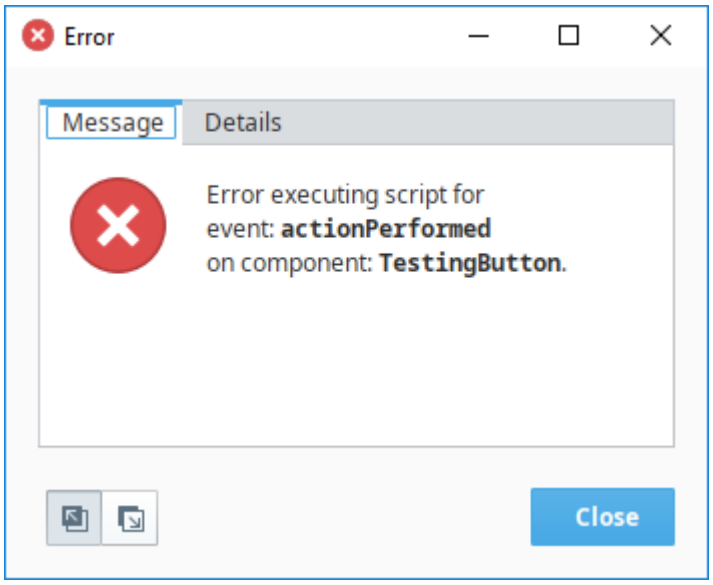
Here is a similar script. When a Button is pressed, it's supposed to open a Message Box on a window and display "Testing".

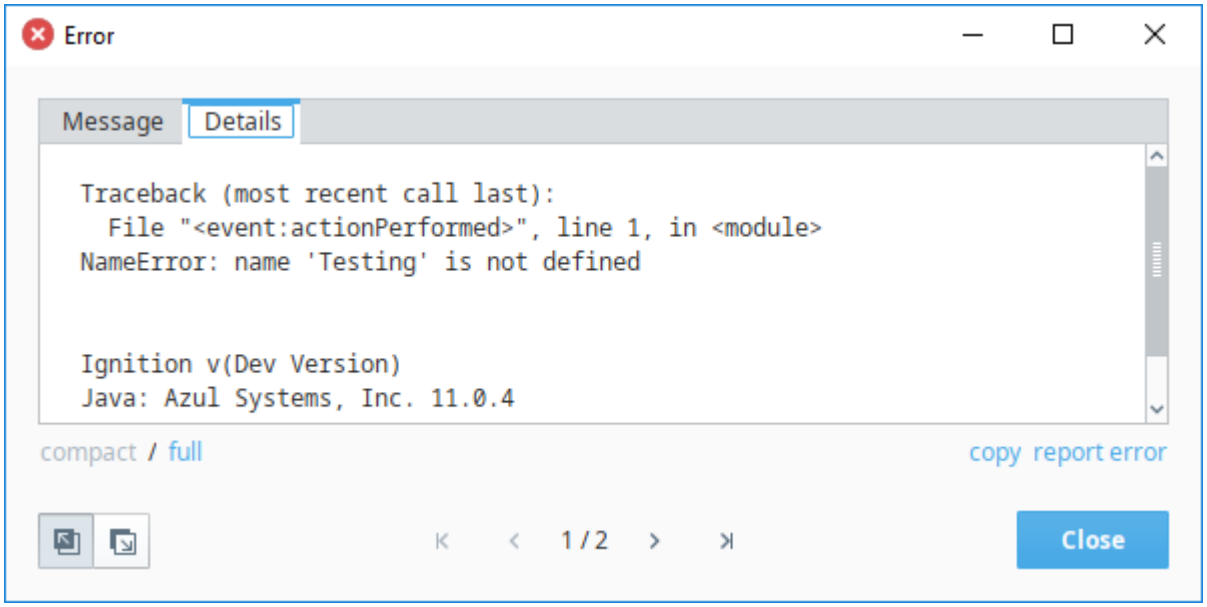
Python - Broken: Undefined

```
system.vision.showMessage(Testing)
```

The Error Message Box displays an error on the actionPerformed script on the button component in line 1 with an **undefined** name. The Details tab is simply telling us that something is not defined in the

script. You can either define a variable or create a string by putting "Testing" in quotes to correct the error.





For this example, we corrected the code by turning the argument passed to system.vision.showMessage into a string literal with quotation marks:

Python - Corrected: Undefined

```
system.vision.showMessage("Testing")
```

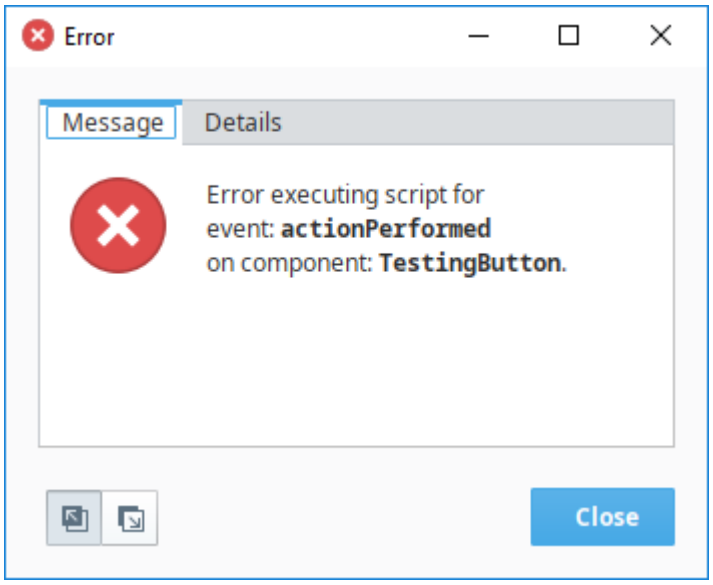
Broken Example - Type Error

Similar to the other scripts, this script opens a Message Box and is supposed to display "100" when a Button is pressed.

Python - Broken: TypeError

```
system.vision.showMessage(100)
```

Since this script failed to execute successfully, the Error Message Box popped up and displayed an error on the actionPerformed script on the button component in line 1 with a **TypeError**. This error Details tab tells us that the first parameter is expecting a string, not an integer.



For this example, we corrected the code by changing the argument passed to system.vision.showMessage into a string literal with quotation marks.

Python - Corrected: TypeError

```
system.vision.showMessage("100")
```

You can see from the examples above, that the Error Message Box provides quite a bit of information that points to the root cause of an error on an Event Handler script. As you learn from the Error Message Boxes about what these messages mean, you'll be able to quickly spot them in your script and quickly correct any errors you encounter.

Troubleshooting - Nothing Happened

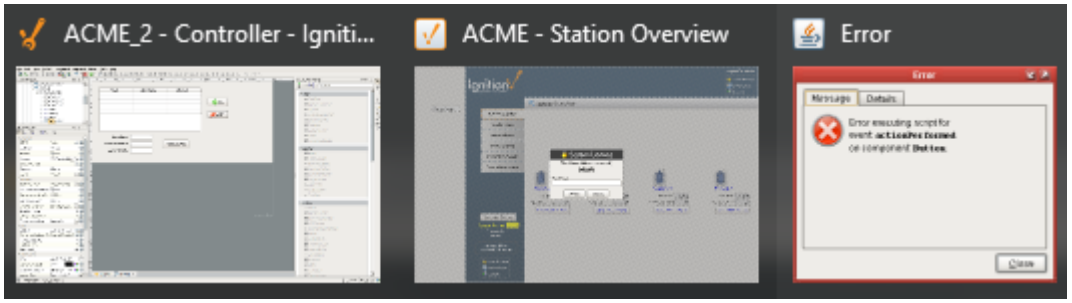
Did It Work?

When testing a script, you may find yourself in a situation where the script appears to be running, but doesn't seem to work. Furthermore, you may not see any error messages stating there is a problem.

When a script doesn't perform to expectations and doesn't throw an exception, it can suggest a problem with the script's workflow. This page describes a couple of common scenarios.

Before You Begin...

Make **absolutely** certain that there isn't an Error Message Box hiding somewhere, otherwise you'll waste time applying the troubleshooting tips outlined below, when the real the error message was hiding in the background behind another window the whole time.



If a button component on a window is running a script, the output will be displayed on the Designer/Client Output Console. You can check for any error messages by navigating to the Console. From the **Client** menu bar, go to **Help > Diagnostics** and select the **Console tab**. From the **Designer** menu bar, select **Tools > Console**.

Common Scenarios When Nothing Happens

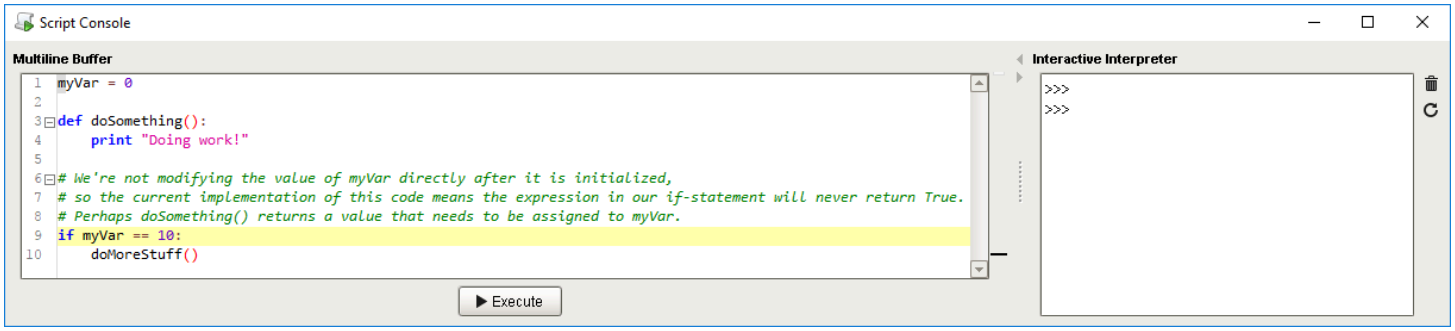
An Important Line Is Missing

Some critical part of your code is missing or commented out. It could be something simple, like your code is supposed to increment a value, but you forgot to write the line that increments the value.

Pseudocode - Missing Code

```
myVar = 0

def doSomething():
    print "Doing work!"

# We're not modifying the value of myVar directly after it is initialized,
# so the current implementation of this code means the expression in our if-statement will never return True.
# Perhaps doSomething() returns a value that needs to be assigned to myVar.
if myVar == 10:
    doMoreStuff()
```

A quick way to test your code is by running your script in the **Script Console** before attaching it to a scripting event or specific component. You can see that running the code above in the Script Console that nothing happened, thus, suggesting an error.



The Script Is Not Being Called

The script doesn't appear to be working because the mechanism that is supposed to be triggered hasn't been called. This can be caused by using the wrong event: (i.e., perhaps you placed the code on a Button's **propertyChange** event, when you meant to place it on **actionPerformed**).

Alternatively, perhaps you defined a function in your script, but you forgot to call the function.

Additionally, if you're testing the script in the Designer, make sure it is in **Preview Mode**. Event based scripts will not run in the Designer unless it is in Preview Mode.

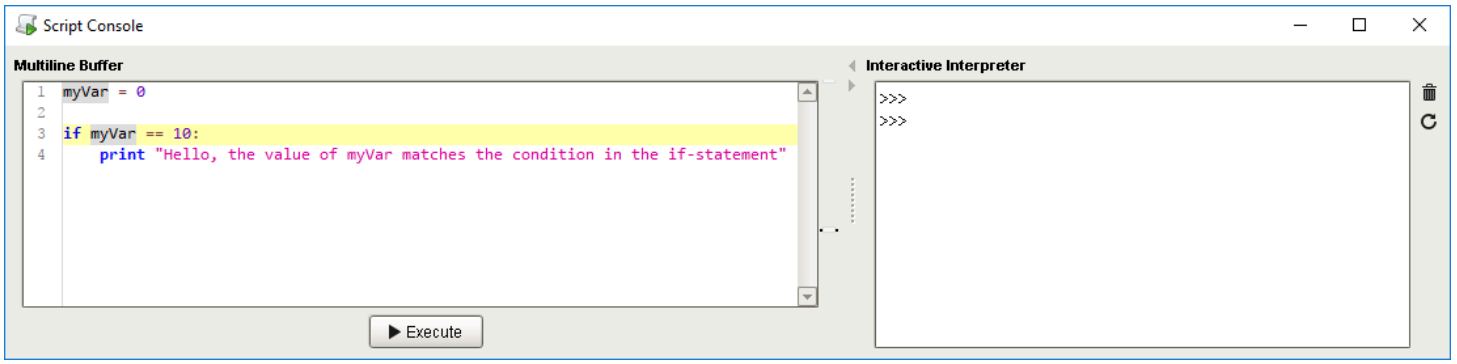
Important Lines Are Being Skipped

This can be caused by incorrect indentation, or a misconfigured condition in an if-statement's expression. For example, you used "==" when you meant to use "!=". Verify that you are using the correct operators in if-statements.

Pseudocode - Skipping Code

```
myVar = 0

if myVar == 10:
    print "Hello, the value of myVar matches the condition in the if-statement"
```

Test your code for any errors by running your script in the Script Console. You can see nothing happened by running the code above in the Script Editor, once again suggesting an error.



Determining the Cause

The easier step to take when troubleshooting your script is to start adding print statements to your code. From here, you can start piecing together what the code is doing:

- **Print** the value of variables to make sure they are coming in the way you expect.
- Place **print** statements at the start and end of your code. This allows you to determine when your script is being called, and when it finishes.
- Adding a **print** statement before and after an if-statement can show you the flow of your script went as expected, or was filtered out by the if-statement.

Troubleshooting Workflow

Troubleshooting a script is an iterative process. Since the script stops executing at the first error, we won't see if there are any other errors on later lines unless, of course, we spot them ourselves while writing the code. As a result, we will have to keep trying our script until it executes successfully.

It is important to understand that scripts always execute from top to bottom, and each line must complete before the next line may move on. In the event, our code returns an exception, we can assume that our troubleshooting process should always start at the line reported in the error, and then work up until we find the problem.

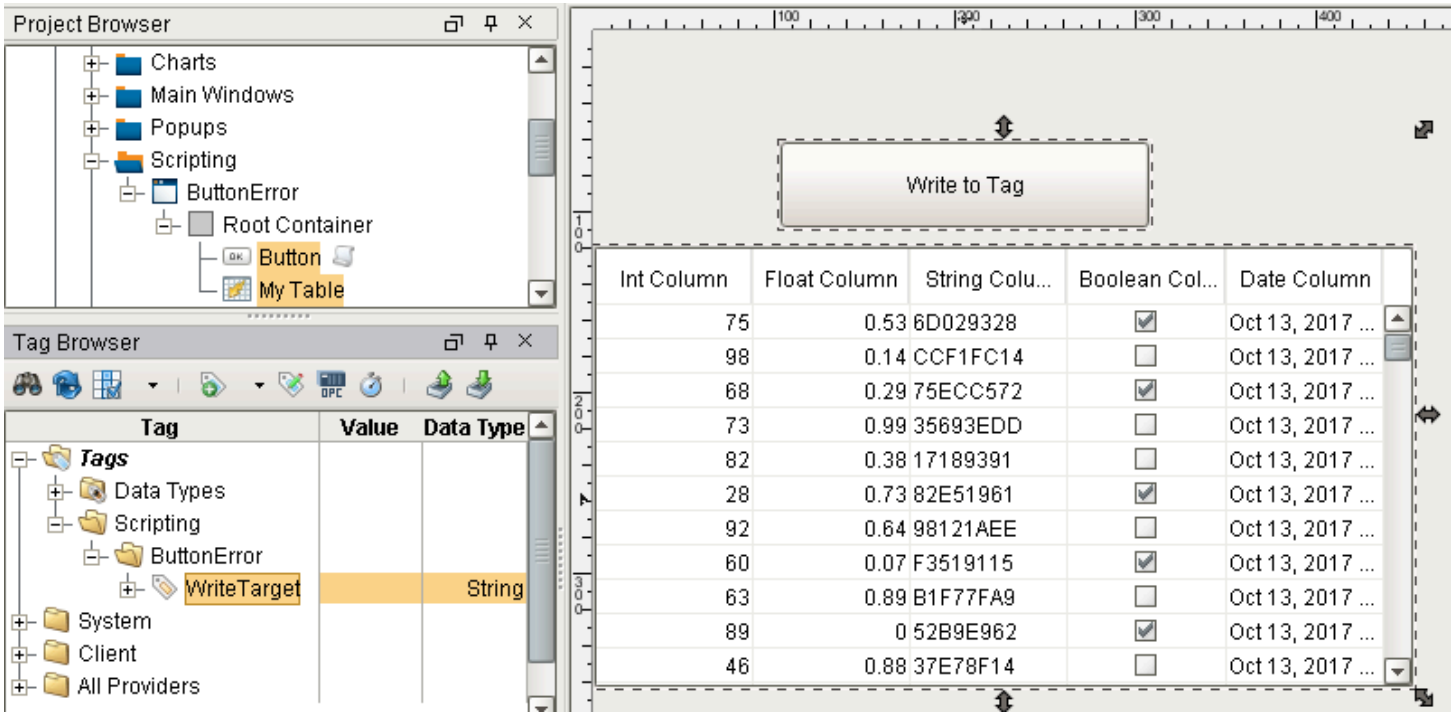
On this page, we will take an in depth look at a script with multiple problems, and work through each one. Note, that this is not a comprehensive list of all possible types of exceptions that could occur when writing a script, but instead, this page attempts to demonstrate the troubleshooting process.

To learn more about troubleshooting specific script errors, refer to the following sections:

- **Reading Error Messages**
- **Troubleshooting - Nothing Happened**

Scenario Overview

Here we have a window with two components: a **Button** and **Power Table**. The purpose of the button is to find which cell in the Power Table the user selected, and write the value to a Tag.



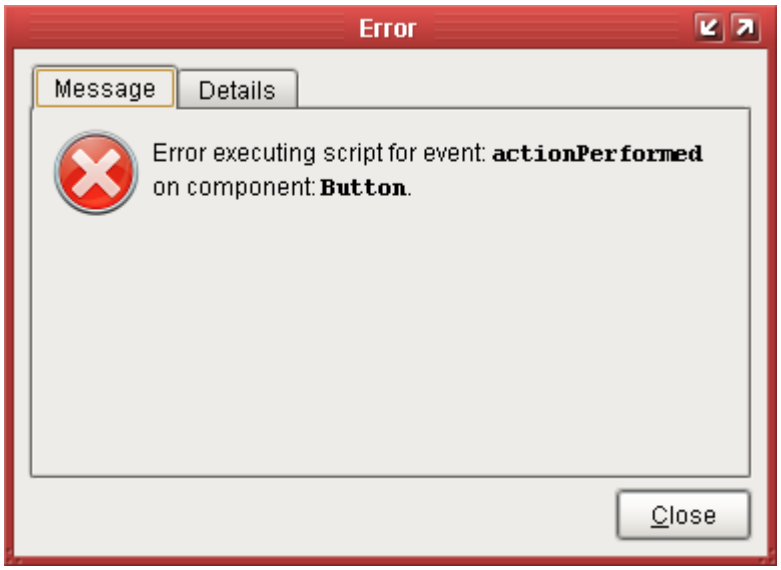
The code on the button is listed below.

Python - Sample Code Block

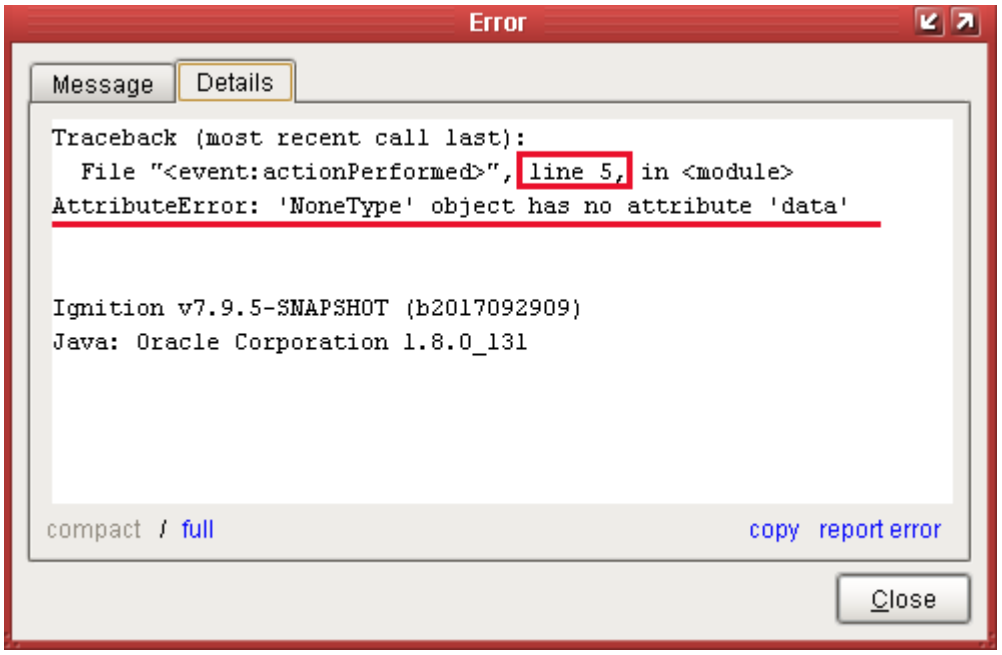
```python
# Create a variable that references the Power Table
table = event.source.parent.getComponent('Power Table')

# Find the cell the user currently has selected, and store the value in a variable
userSelectedValue = table.data.getValueAt(table.selectedRow, table.selectedColumn)

# Write the User Selected Value to a tag
system.tag.writeBlocking(["Scripting/ButtonError/WriteTarget"],
[userSelectedValue])
```

First Error - NoneType Object

When the button is pressed, we are presented with an error. The **Message** tab in the Error Box describes the **Event Handler** that encountered the exception.



The **Details** tab tells us where to start looking, specifically the line number where the exception occurred as well as the error message. Line 5 is referenced, so the troubleshooting process should start there.



Exception Error Explained

When an error message refers to a **NoneType** object, that simply means a null, or None in Python. The word '**attribute**' is used to reference a property on an object. In this case, this means the script was trying to access the '**data**' property on nothing. The message is telling us that it couldn't find a property named '**data**' on a nothing, which is correct since NoneType objects don't have any properties named 'data'.

If we were to reword this message to something a bit more straightforward, it would say the following: "I tried to access the 'data' property on 'null', but it doesn't have a property by that name."

What Should We Look For?

If your exception is referring to a NoneType object, then some line of code probably tried to reference something else (i.e., Tag value, property, another variable, etc.), but couldn't find anything at the location you specified. It is also helpful to note what attribute was mentioned in the error, which was **data**. If we look at our code, we see the following on line 5.

Python - Line 5

```
userSelectedValue  = table.data.getValueAt(table.selectedrow,
table.selectedColumn)
```

Based on this information, we should take a closer look at the table variable since our code is specifying '**data**' as an attribute of the table. The error states that it could not find the data attribute on a NoneType object, but our code only references data in regards to the table object, so maybe there is

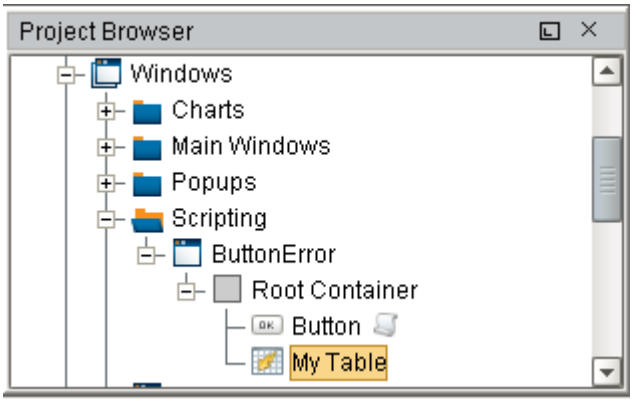
something wrong with how that table object was referenced or initialized. If we look further up in our code, we see that `table` was initialized on line 2:

Python - Line 2

```
table = event.source.parent.getComponent('Power Table')
```

Since this script was placed on the Button component's actionPerformed event, we can trace back to the source of the issue:

- event = actionPerformed event
- source = Button this script is placed on
- parent = Container that the Button was placed in. Based on the Button's position in the window, this appears to be the Root Container.
- getComponent('**Power Table**') = Should return a reference to a component named '**Power Table**' directly inside of the Root Container, assuming one exists. If we take a look at the Project Browser, we see the following:



There is no component named '**Power Table**' in the Root Container. However, there is a Power Table component named '**My Table**', so it appears someone renamed the component, which caused our script to initialize the variable table as a '**NoneType**' instead of a reference to a component.

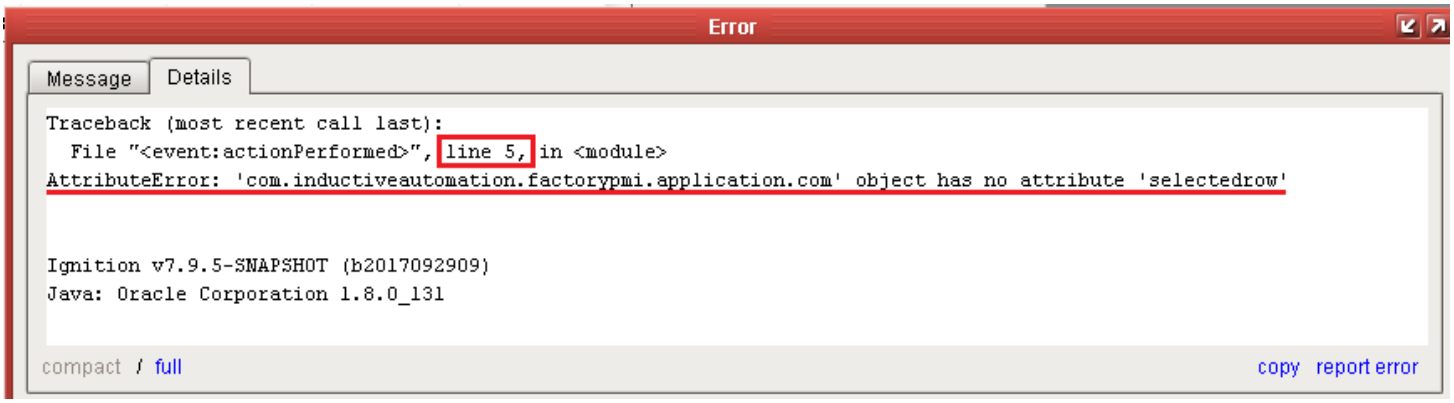
Solution

To fix this issue, we can simply update our code to use the new name of the Power Table, which is '**My Table**'. We can type this in manually, but since it is case sensitive, it must match exactly. It may be easier to find the **Name** property on the component, copy the name to the system clipboard (**Ctrl-C**), and paste it (**Ctrl-V**) into the script. Our code now looks like the following:

```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('My Table')

# Find the cell the user currently has selected, and store the value in a variable
userSelectedValue = table.data.getValueAt(table.selectedRow, table.selectedColumn)

# Write the User Selected Value to a tag
system.tag.writeBlocking(["Scripting/ButtonError/WriteTarget"],
[userSelectedValue])
```

Second Error - Checking Attributes

If we try our button again, we get the following exception error.



Exception Error Explained

Like the last exception, the problem has to do with our code trying to access an attribute (property) on an object, but the given attribute doesn't exist on the object.

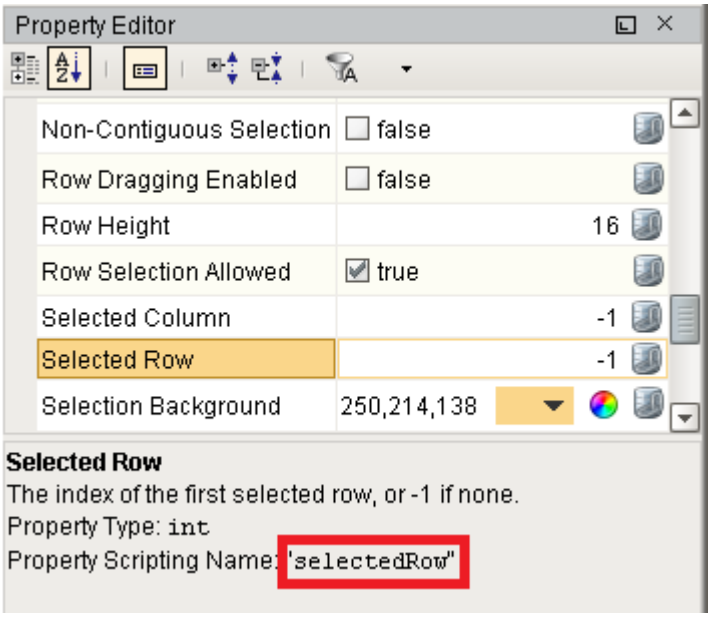
What Should We Look For?

Again, the exception refers to line 5, but note that the error mentions a different object and attribute this time. Without knowing what a `com.inductiveautomation.factorypmi.application.com` object is, we can tell that our script is trying to reference an attribute named `selectedrow`. This gives us a starting point for troubleshooting the error. If we check line 5, we see the following:

Python - Line 5

```
userSelectedValue = table.data.getValueAt(table.selectedrow, table.selectedColumn)
```

We see `table.selectedrow` as the first parameter being passed to the getValueAt() function. We checked into the `table` variable earlier, so this doesn't necessarily mean that the variable is the problem, although there is no harm in double checking, especially if you have not personally verified the variable. Assuming that the table variable is correct, let's check the attribute. We can check the Property Editor in the Designer to find a matching property, or head over to the Power Table page in the manual and check the property reference. We can see that there is a **Selected Row** property on the Power Table, but it should be spelled "`selectedRow`" with a capital "**R**".



We also know that the getValueAt() function takes a row index as the first parameter, so it would make sense that our code has a typo.

Solution

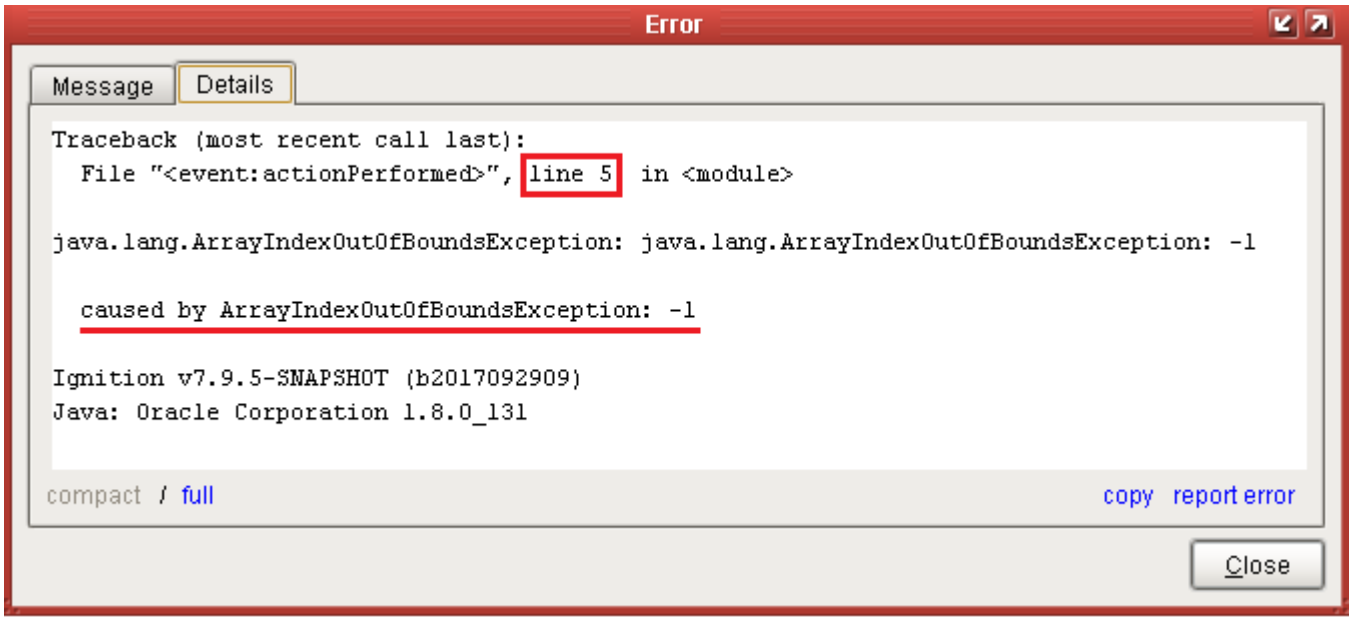
We should fix the typo in the attribute name.

Python - Line 5 Updated

```
userSelectedValue = table.data.getValueAt(table.selectedRow, table.selectedColumn)
```

Third Error - ArrayIndexOutOfBounds

After making our most recent change, the button works great! Users are able to select a cell, and write to the Tag. The one exception, is if the user does not have a cell selected before clicking the button. In this case, the following error occurs:



Exception Error Explained

Our code attempted to look up something in a collection of some sort (i.e., sequence, dataset, etc.,) by index, but was told to check index -1, which is out of bounds. Indexes in Ignition are typically zero-based, meaning they start at 0, and increment from there, so an index of -1 doesn't exist. Commonly, when a property in Ignition refers to index -1, that means nothing is selected, so our code failed because nothing was selected.

What Should We Look For?

Fortunately, the exception told us what the issue is: the user did not have anything selected before pressing the button, so we don't have to search further.

Solution

The solution to this issue is more open ended, as there are many ways to suggest to our user that they need to select a cell in the table first. The **Selected Row** and **Selected Column** properties on the Power Table will have -1 values if none of the cells are selected, so we could use an '**if**' statement checking the values of one of those properties. Our new code could look like the following:

```
# Create a variable that references the Power Table
table = event.source.parent.getComponent('My Table')

# Make sure a cell in the Power Table is selected first
if table.selectedRow != -1:
    # Find the cell the user currently has selected, and store the value in a
variable
    userSelectedValue = table.data.getValueAt(table.selectedRow,
table.selectedColumn)

    # Write the User Selected Value to a tag
    system.tag.writeBlocking(["Scripting/ButtonError/WriteTarget"],
[userSelectedValue])

# If a cell isn't selected, then let the user know
else:
    system.vision.showMessage("Please select a cell in the table first!")
```

Scripting Vs. SQL Vs. Expressions

Expression Language & SQL Queries vs Scripting

There are three major languages in Ignition, the **Expression language**, the **SQL Queries**, and **Python Scripting**. It is important to understand the differences between the three and to know where each is used. Scripting is used in the event handlers that are available all over Ignition, but Expressions and SQL are in the **Property Binding** locations shown here.

Spot the Difference - Comments

When starting out with Ignition, it can be difficult to know which syntax you should be using for a particular area of text. One little trick that may help is to be familiar with how each language handles comments, and then utilize the **Ctrl** + **/** command which automatically adds the characters that comments out a line of code in the language are you are typing in. Once you remember to use **Ctrl** + **/**, you simply need to be familiar with the characters that each language uses.

Expression Language - Comment

```
// The Expression Language uses double forward slashes.
```

SQL - Comment

```
-- Areas in Ignition that accept SQL syntax use double dashes.
```

Python - Comment

```
# Python uses the pound/number/hash character.
```

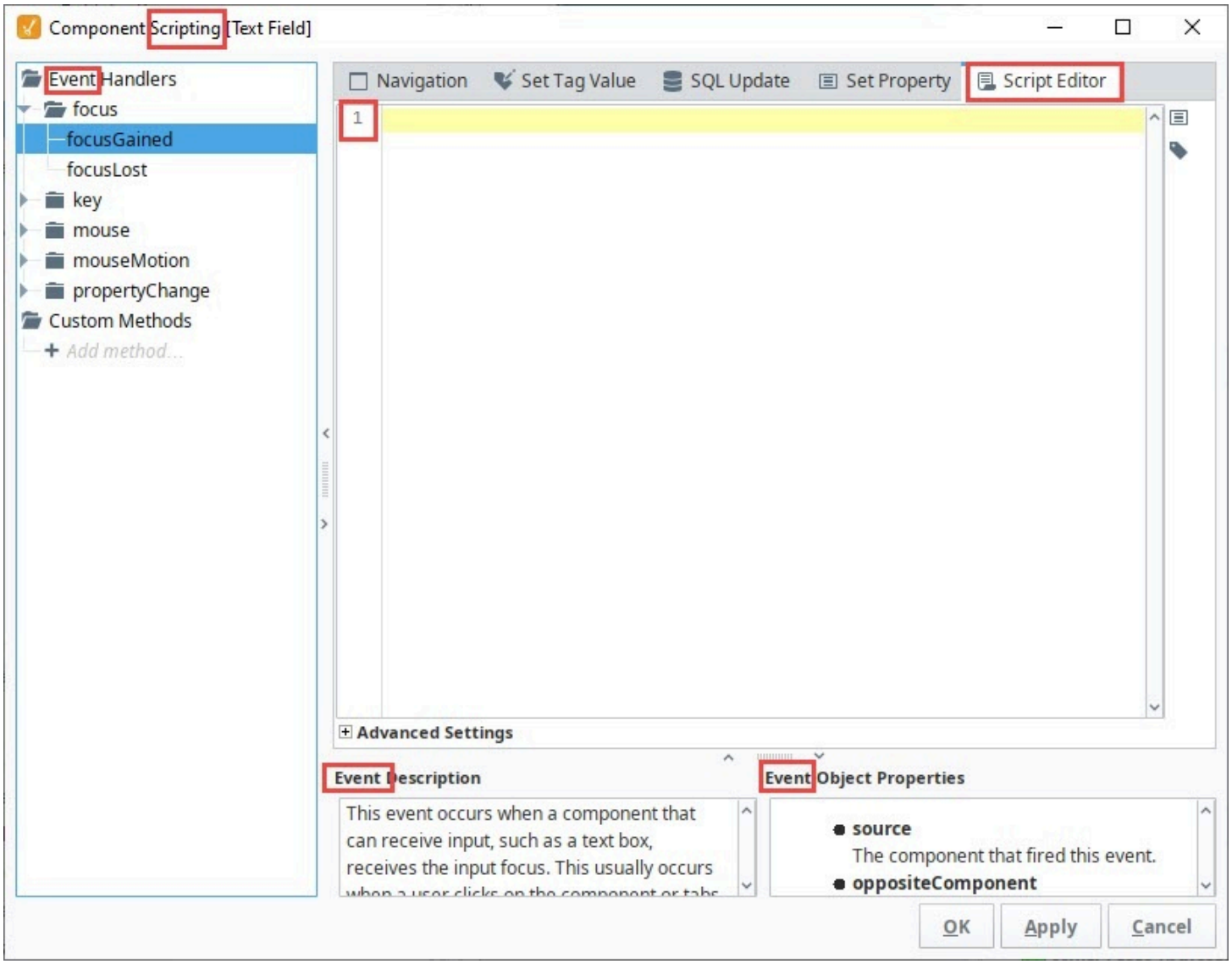
In addition to comments, interfaces for each of the languages usually contain other signs or reminders about the language. These will be covered in their respective sections on this page.

Python Scripts

Python is featured prominently throughout Ignition and many different resources can contain a Python Script.

How Can I Tell If I'm Writing a Python Script?

Python Scripts typically use the words **Script** or **Event** in the interface. Additionally, Python requires a particular Event to be selected so if you see event Handlers on the left, you know you are looking at a Python script.



Where Are Python Scripts Used?

Python Scripts are used all throughout Ignition. Some resources, such as components, even have multiple places to write scripts! Below are some common locations for scripts:

- **Scripting in Vision**
- **Extension Functions**

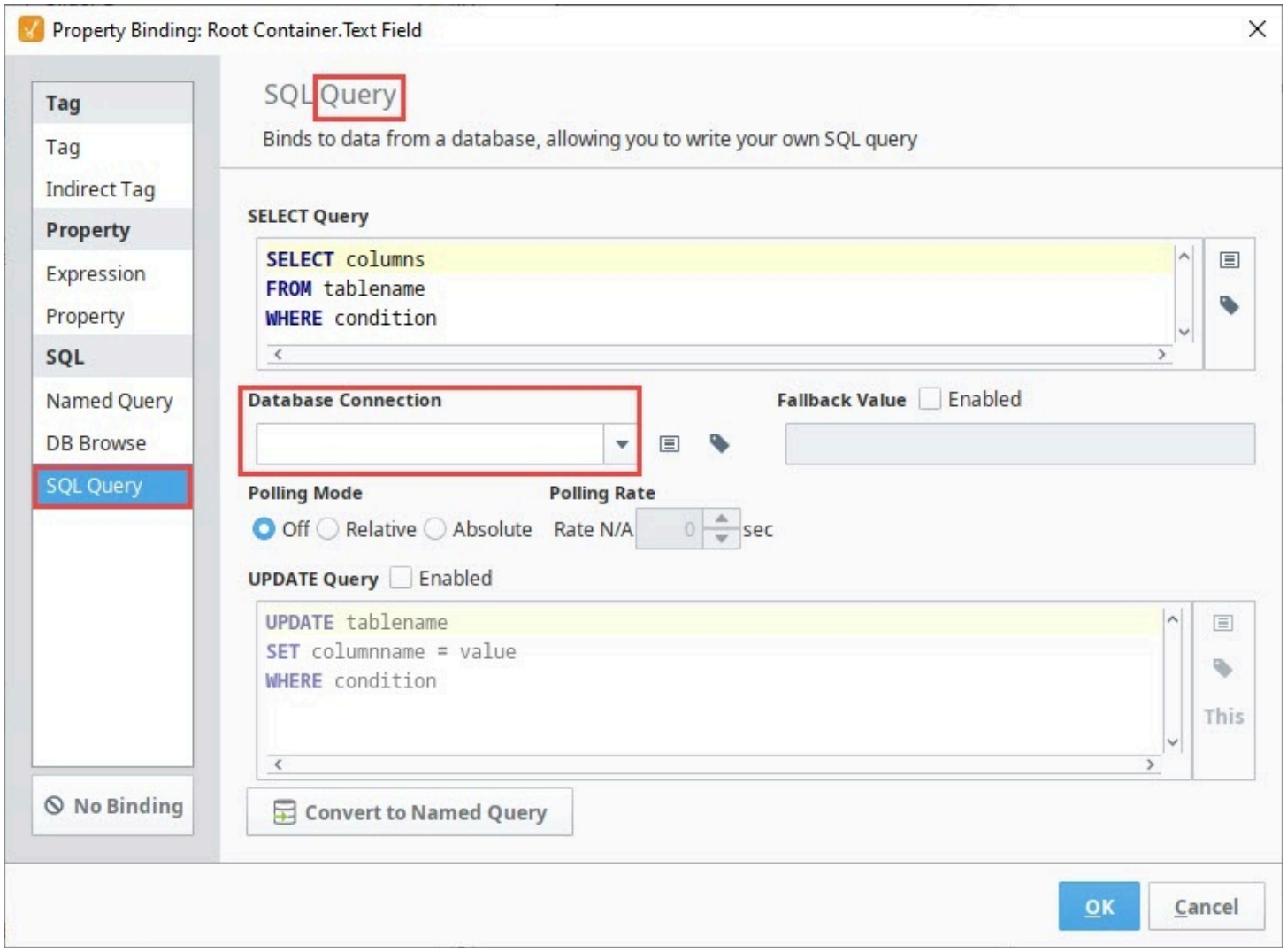
- Tag Event Scripts
- Client Event Scripts
- Gateway Event Scripts

SQL Queries

The SQL language is used for selecting information from a database. It can be used in a variety of ways, but most of them will either make modifications to the database or return a set of values (with a few notable exceptions like Stored Procedures). The majority of users will be returning large chunks of data into a Table or Report in Ignition. This means a complete dataset will be returned based on a user selection, a time range, or any combination of factors.

How Can I Tell If I Should Be Using SQL Syntax?

Typically, the words **Query** and **Database** appear in the interface somewhere. Additionally, there is usually a way to specify a **Database Connection**.



Where Is SQL Used in Ignition?

Below is a reference of the most common areas in Ignition where SQL queries may be used.

- **Vision SQL Query Bindings**
- **Named Queries**
- **Database Query Browser**
- Reporting Data Source:
  - **SQL Query Data Source**
  - **Basic SQL Query**
- **Query Tags**
- Python Scripts - See the SQL in Python header below for more details

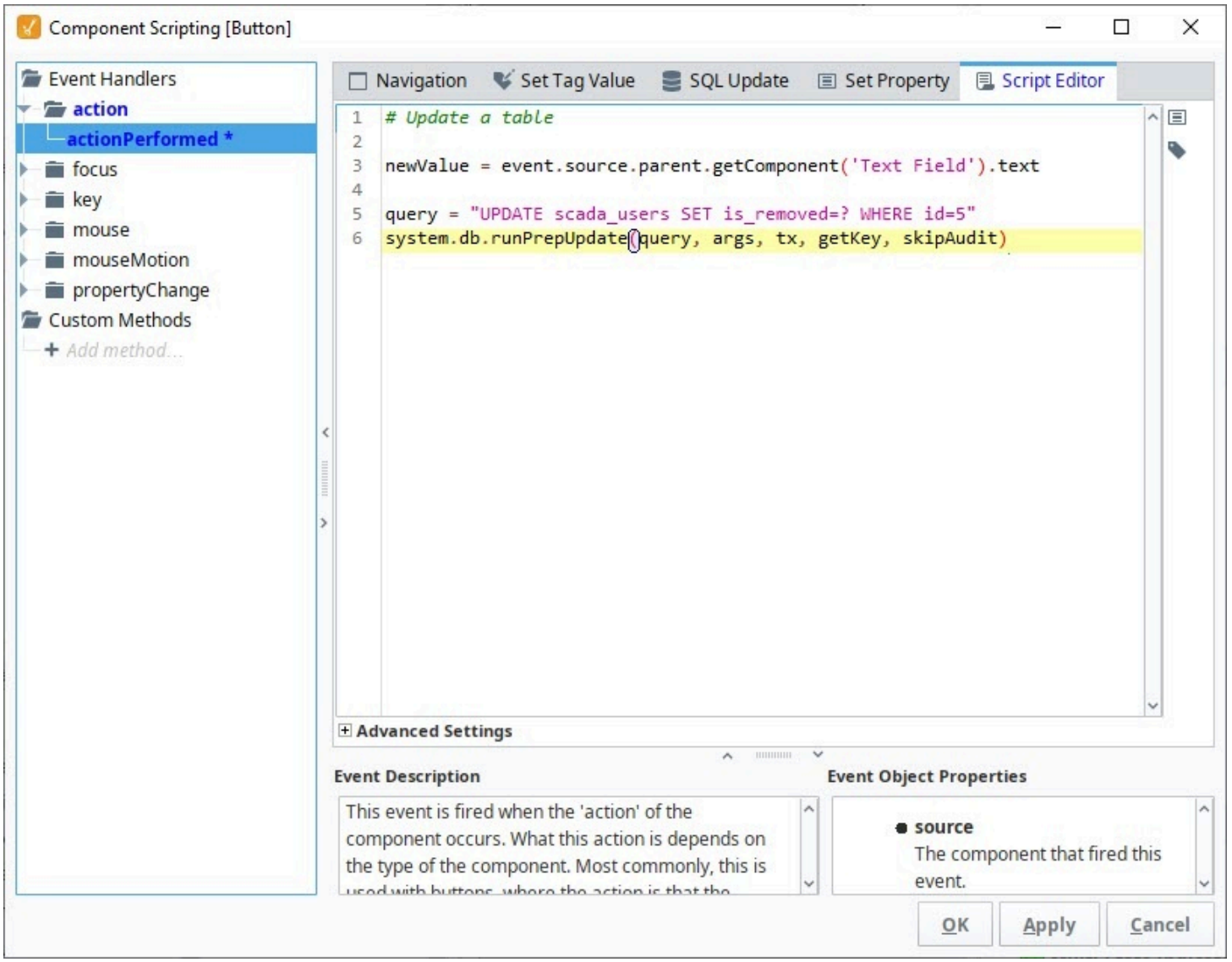
SQL in Python

SQL queries can be called from a **Python** script. There are several system functions in Ignition that allow a script to run a query against the database, such as **system.db.runPrepQuery**. This is a more advanced technique, as you need to adhere to both language's syntax. Furthermore, when typing a SQL query in a Python script interface, the syntax highlighting can not help with the SQL portions. The syntax highlighting in a Scripting Window is only looking for Python syntax, not SQL.

In cases where you plan on calling a SQL query from a Python script, it is highly recommended to write the query in the **Database Query Browser** first (substituting parameters with static values for testing purposes), and then move the query over to the script once the query executes successfully on its own. This approach can save you some time troubleshooting, as there will be less ambiguity when an error occurs since you know the query runs.

Below we see an example of calling a SQL query in a script. Line 5 creates a variable called "query", and assigns it a string consisting of a prepared statement (using SQL). The query is then executed with the **system.db.runPrepUpdate** function.

For more examples of using a query in a Python Script, check out the **system.db.*** functions.



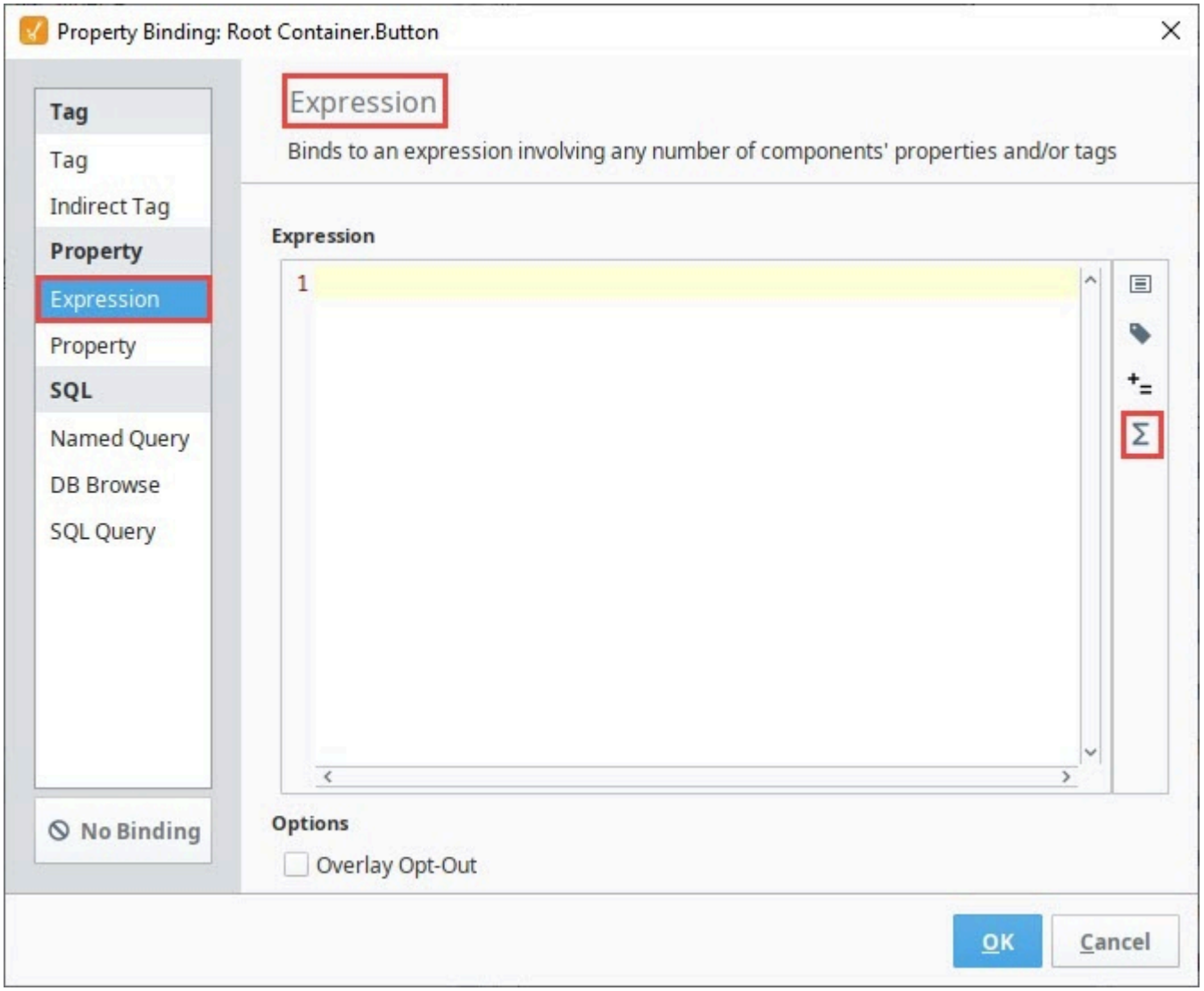
Expression Language

The Expression Language is a simple programming language that we invented (very similar to many other existing expression languages), and is different from the Python scripting you will find in Ignition. The expression language is a very simple kind of language where everything is an expression - which is a single piece of code that returns a value. This means that there are no statements, and no variables, just operators, literals, and functions.

The most common expression language that most people are familiar with is the one found in Microsoft Excel. You can have Excel calculated a cell's value dynamically by typing an expression like `=SUM(C5:C10)`. Our expression language has similar functionality, but different syntax. It is mainly used to define dynamic values for **Tags** and **component properties**. In Ignition's Expression Language, you will use component properties and functions like if(`{Root Container.type}`="Type C",True,False).

How Can I Tell If I'm Writing an Expression?

Typically interfaces that expect the Expression Language use the word **Expression**. Additionally, you'll commonly find the **Expression Function** button $f_{(x)}$ towards the right side of the text area.



Where Is The Expression Language Used?

Below is a list of resources that commonly utilize the Expression Language

- **Expression Bindings** on Property Bindings and **Alarm Bindings**
- **Expression Tags** and **Derived Tags**
- Several **Alarm Pipeline Blocks**, including the **Expression Block**, **Switch Block**, and **Notification Blocks**.
- **Expression Items** in Transaction Groups
- **Parameters on Reports**
- **SFC Transitions**

Scripting Examples

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 Ignition. 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.

Reading and Writing to Tags

There are simple interfaces in Ignition that allow you to easily write to a Tag on some Event, and Reading can be as simple as creating a Tag binding. Sometimes, however, the built-in approaches can be too simplistic or limiting. The **Reading and Writing to Tags** page details how to better interact with Tags from scripting.

Importing and Exporting a CSV

CSV files are used by many software programs to export data so that other systems may utilize the information contained within. The **Importing and Exporting a CSV** page demonstrates how to both import a CSV into Ignition, as well as export data from Ignition into a CSV.

Reading a Cell from a Table

Once data is populated into a Table component, it's useful to know how to read and extract a data from a cell in a Table, particularly if users can select a row in a Table. The **Read a Cell from a Table** page has some good examples for retrieving data from a single cell and multiple cells in a Table.

Export Tag Historian to CSV

The Tag Historian Data can be great, but it's sometimes difficult to view it outside of Ignition. The **Export Tag Historian to CSV** page details how to pull out a subset of Tag history data and export it to a CSV file.

Location Based Vision Startup Scripts

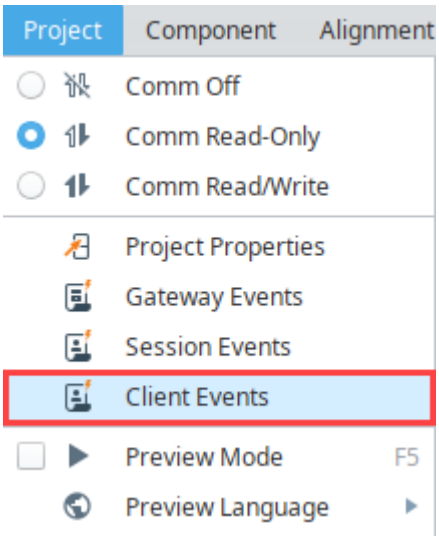
Client Startup Scripts

Security can be created on a Vision Client using a Startup Script that checks certain user information and then customizes the level of access within the project based on that information. We can pull user information from places like **System Tags** for Vision Clients or **Session Properties** for Perspective Sessions. Using that information, the script can then customize which windows or views the user sees, provide some additional information to the user, or even prevent the user from accessing the project completely. An additional example is located **here**.

Location Based Vision Client Restrictions

Each Client has access to distinct hostname and IP addresses. These can be used in Client startup scripts that evaluate the Client's information and compares it to a list of acceptable host names or IP addresses. This information can come from a database or a set of Tags. In this example, we will prevent access to the client if the user is logging in from an incorrect location.

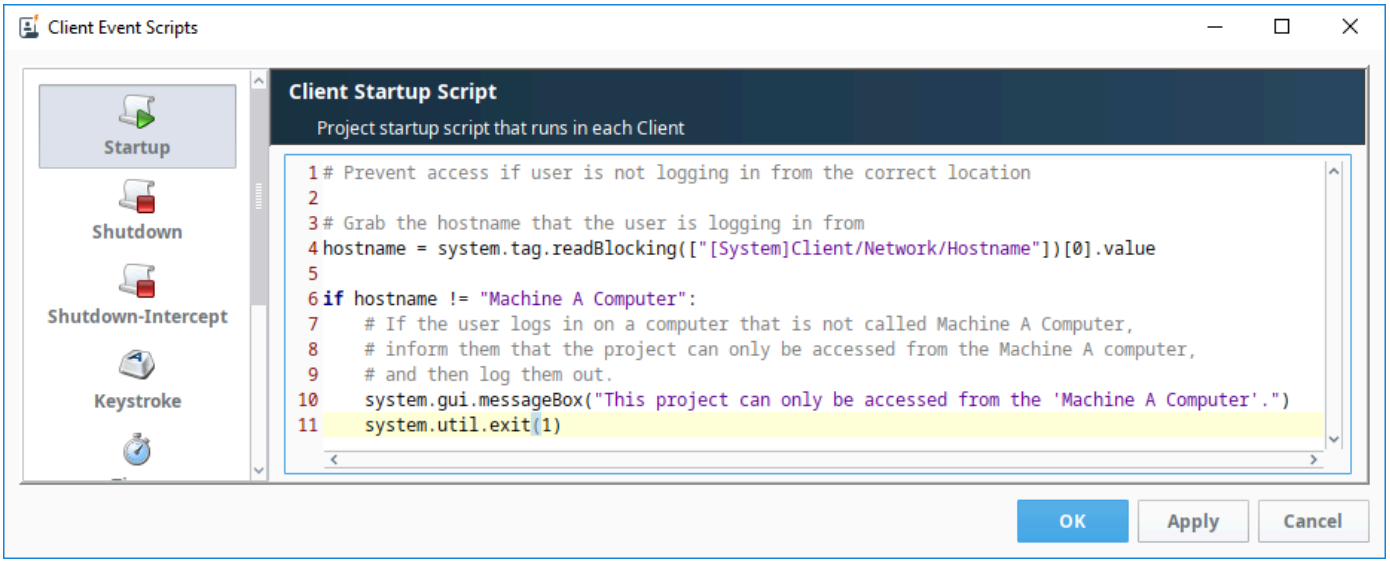
1. On the Project tab, choose Client Events.



2. Select the Startup icon.

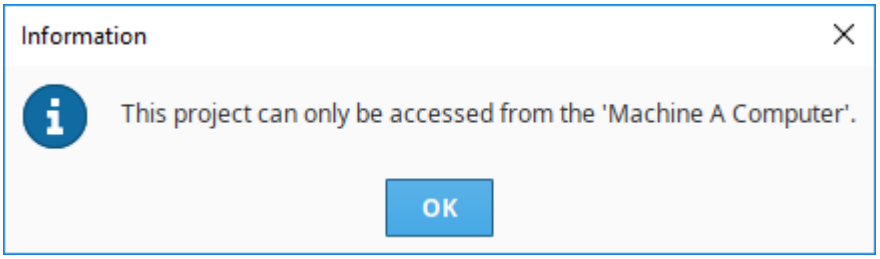
3. Add the following script:

```
# Prevent access if user is not logging in from the correct location

# Grab the hostname that the user is logging in from
hostname = system.tag.readBlocking(["[System]Client/Network/Hostname"])
[0].value

if hostname != "Machine A Computer":
    # If the user logs in on a computer that is not called Machine A Computer,
    # inform them that the project can only be accessed from the Machine A
computer,
    # and then log them out.
    system.vision.showMessage("This project can only be accessed from the
'Machine A Computer'.")
    system.vision.exit()
```



4. Click **OK** to save the script.

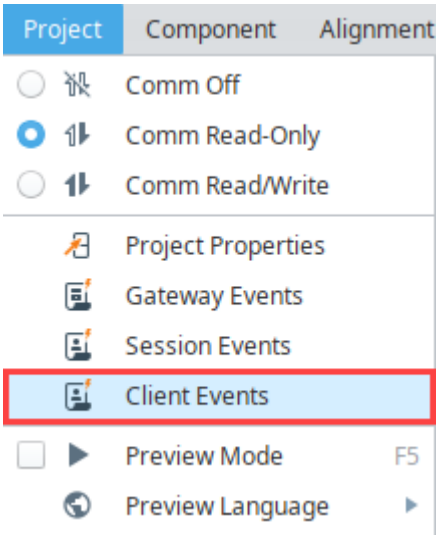
5. Save your project and launch the client. You will get a popup that informs you that the project must be launched using the Machine A Computer.



Location Based Startup Display

Another common use for startup scripts on Vision Clients is to open a specific window depending on the location of the log in. In this example, if the user logs in on Machine A Computer, then the Machine A Details window will be displayed. If they log in on a different computer, the Overview window is displayed.

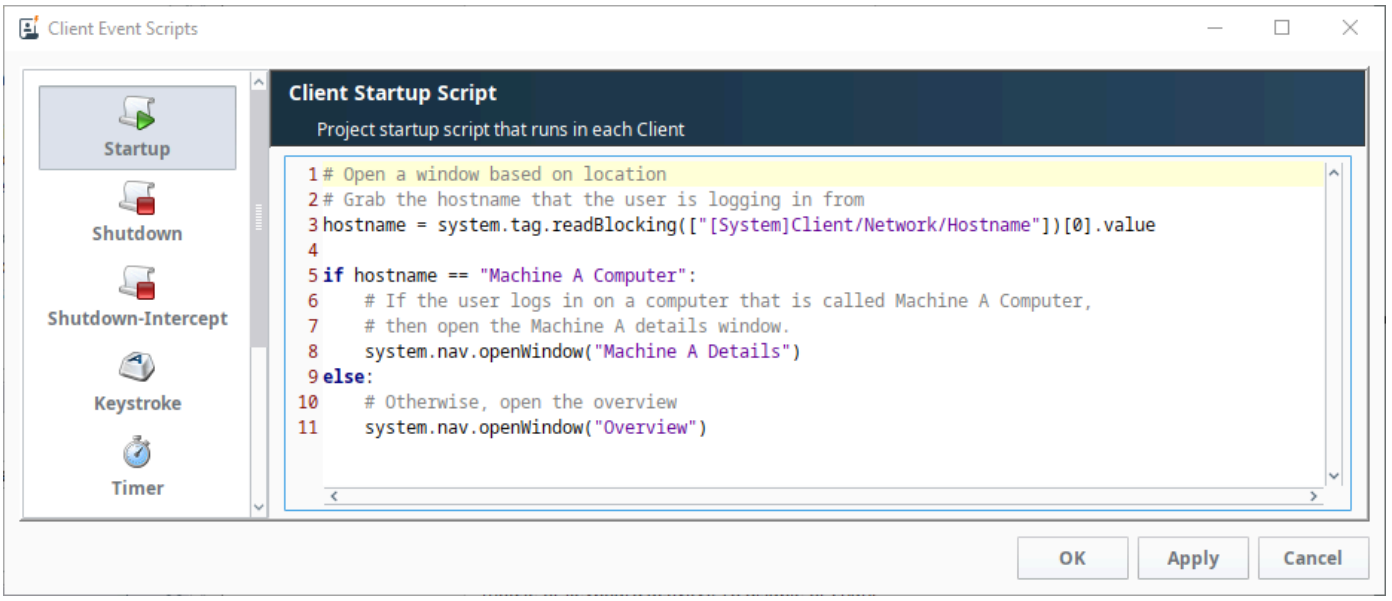
1. On the Project tab, choose Client Events.



2. Select the Startup icon.

3. Add the following script:

```
# Open a window based on location

# Grab the hostname that the user is logging in from
hostname = system.tag.readBlocking(["[System]Client/Network/Hostname"])
[0].value

if hostname == "Machine A Computer":
    # If the user logs in on a computer that is called Machine A Computer,
    # then open the Machine A details window.
    system.nav.openWindow("Machine A Details")
else:
    # Otherwise, open the overview
    system.nav.openWindow("Overview")
```



4. Click **OK** to save the script.

> **NOTE**
>
> Any windows that are set to Open on Startup will still open in addition to the window specified in the startup script. You should disable any main windows from opening on startup if using this method.

5. Save your project.

Reading and Writing to Tags

In many cases, the binding system is the most appropriate way to display a Tag value on the screen. However, you may wish to access a Tag's value in a script. Using the system functions, you can read from a Tag and write to a Tag in Ignition.

Script Builder

If you simply need to write to value to a Tag from a script, you can use the Set Tag Value tab of the Script Editor. To learn more, refer to the sections on **Script Builders** for more information. However, if you need to do more than just send a single write, see the section below.

Know Your Scope

Whenever a script from a Shared resource (such an Alarm Pipeline) attempts to interact with a Tag, the script must specify the Tag Provider, otherwise, the script could return an exception. Tag Providers are always included at the start of the Tag Path and look like the following: `[tagProvider]`

When interacting with a Tag from the Project scope (such as a script on a component) you may optionally include the Tag provider. If omitted, the Project default Tag provider will be used.

Pseudocode - Tag Read Scope

```
# Reading a Tag without the Tag Provider. You would not want to use this from the Shared scope.
system.tag.readBlocking(["My/Tag/Path"])

# Reading from the same Tag, but specifying the Tag Provider. This format may be used safely in either scope.
system.tag.readBlocking(["[default]My/Tag/Path"])
```

Manual Tag Reads

Reading from a Single Tag

Reading a Tag from a script is accomplished with the **system.tag.readBlocking()** function, which requires the Tag path you wish to read from. This function returns a 'Qualified Value'; this object is more than just the value. A Qualified Value is the Tag value that has three attributes; Value, Quality, and TimeStamp.

The function normally expects a list of Tag paths, but can be used with a single Tag: simply provide a list of only a single Tag path.

When handling the results of the read, the results will be returned in a list of qualified values, so you'll need to specify which qualified value you're interested in via Python slicing, even if there is only a single qualified value in the list.

Pseudocode - Reading Tag Attributes

```
# get the Tag value
value = system.tag.readBlocking(["tagPath"])[0].value

# get all three attributes
tag = system.tag.readBlocking(["tagPath"])[0]
value = tag.value
quality = tag.quality
timestamp= tag.timestamp
```

Reading from Multiple Tags

The **system.tag.readBlocking()** function can easily be used to read multiple Tags in a single call:

Python - Reading Multiple Tags and Printing the Values

```
# Create a List of Tag Paths to read
paths = ["Scripting/Tags/Alarm_1", "Scripting/Tags/Alarm_2",
"Scripting/Tags/Alarm_3"]

# Read the Tags, and store the complex results in a variable
values = system.tag.readBlocking(paths)

# For each Tag Path, iterate through our results...
for index in range(len(paths)):

    # ...and do something with the individual values
    print values[index].value
```

Relative Tag Paths

Like elsewhere in Ignition, relative paths may be used from within a script. This is especially useful when writing a Tag Event script inside of a UDT, as you can specify relative members in the same UDT with "[.]".

Pseudocode - Reading with Relative Tag Paths

```
# Assuming a UDT with two sub-members, a script from one can read from one member
to the other using the following code:
system.tag.readBlocking(["[.]otherMember"]).value
```

Manual Tag Writes

Writing to a Single Tag

Much like reading, there is a function you can use to write to Tags: **system.tag.writeBlocking**. It requires a list of Tag paths, as well as a list of values to write to those Tags.

Pseudocode - Tag Write

```
system.tag.writeBlocking(["tagPath"],["Hello World"])
```

Writing to Multiple Tags

The **system.tag.writeBlocking** function can also write to multiple Tags, again by providing multiple paths and values.

Python - Multiple Tag Writes

```python
# Create a List of Tag Paths to write to
paths = ["Scripting/Tags/Alarm_Setpoint_1", "Scripting/Tags/Alarm_Setpoint_2"]

# Create a List of values to write
values = [72, 72]

# Send of the write requests
system.tag.writeBlocking(paths, values)
```

Exporting and Importing a CSV

A CSV (comma separated values) file, is one of the most simple structured formats used for exporting and importing datasets. It is a convenient and flexible way to edit and share data across applications. Ignition has a built-in function to convert a dataset to CSV data called **system.dataset.toCSV**. You can even convert the contents of a CSV in a script and move it to an Ignition component, such as a Table.

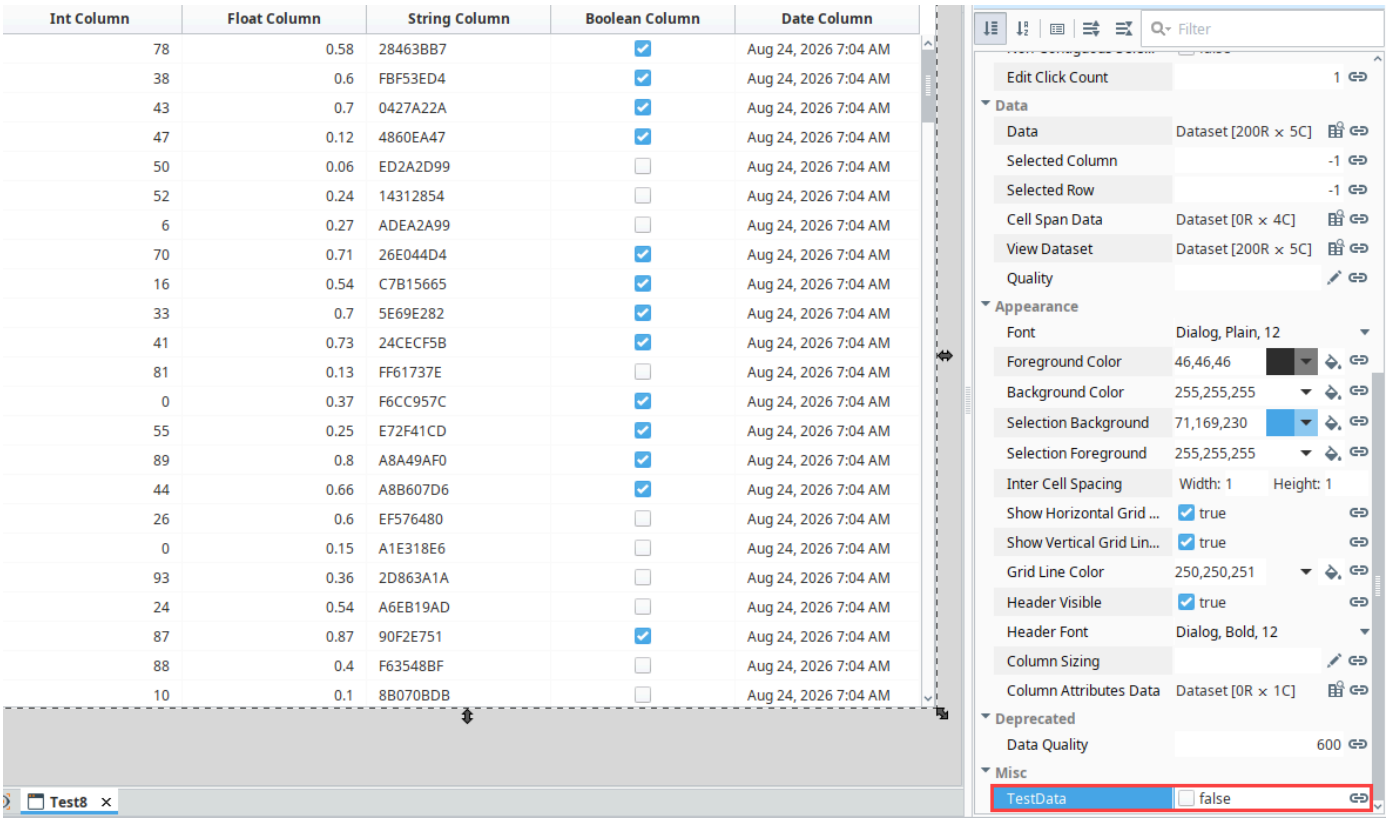
This section contains examples for exporting and importing data to a CSV, as well as converting the contents of a CSV file.

Exporting Data to a CSV

You can export a dataset from a query or table to a CSV file. In the following examples, we will generate data on a Power Table and Table component in Vision and Perspective, respectively, and then export that data.

Vision Example

1. Drag a **Power Table** component on to your window and toggle its **TestData** property to generate some data.



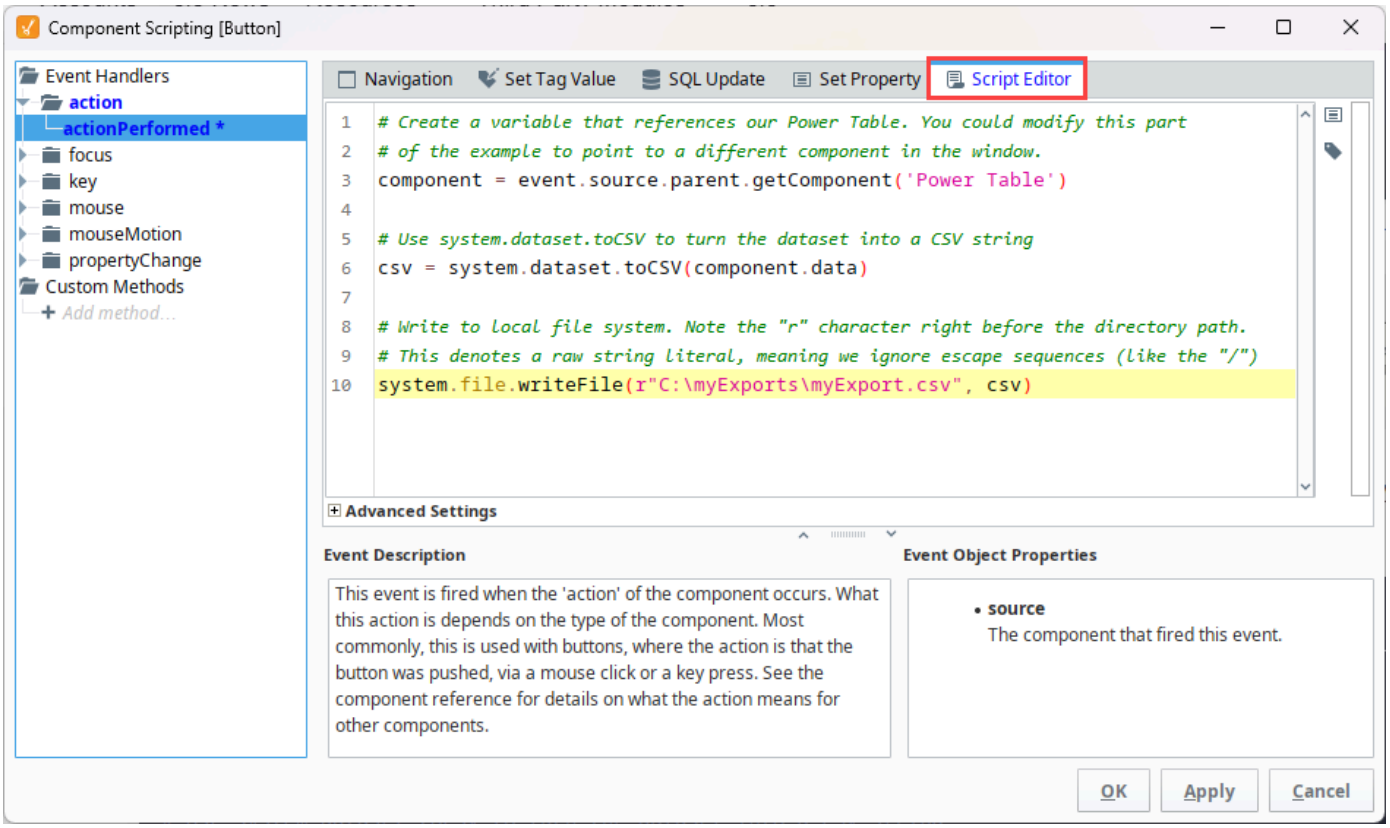
2. Drag a **Button** component on the window, and select **Scripting** from the Button's right-click menu to open the Component Scripting popup.

3. Select the **action** > **actionPerformed** event, and click on the **Script Editor** tab.

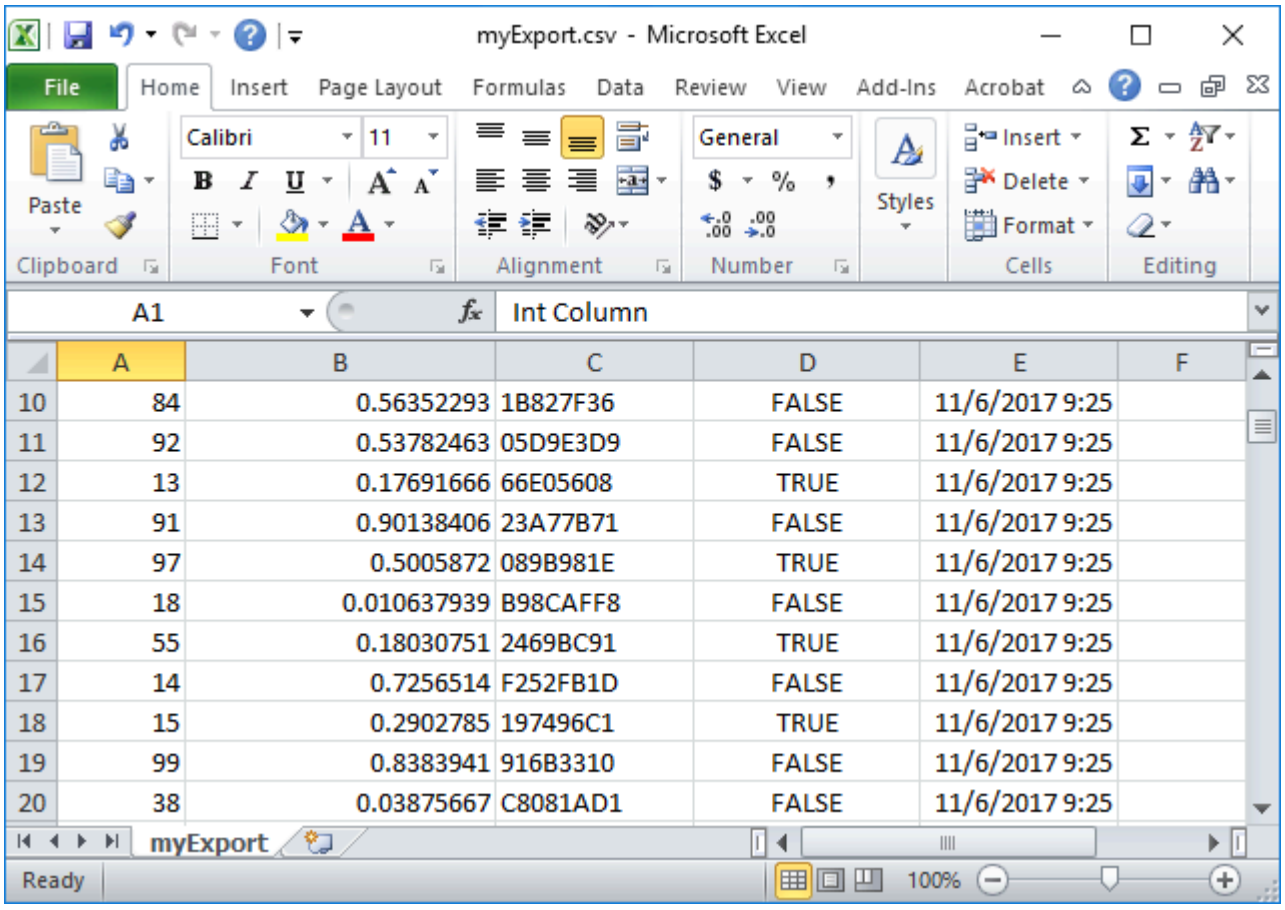
4. Copy the example script below, and paste the contents to the Script Editor.

Hard Coded Filepath

```
# Create a variable that references our Power Table. You could modify this part
# of the example to point to a different component in the window.
component = event.source.parent.getComponent('Power Table')

# Use system.dataset.toCSV to turn the dataset into a CSV string
csv = system.dataset.toCSV(component.data)

# Write to local file system. Note the "r" character right before the directory path.
# This denotes a raw string literal, meaning we ignore escape sequences (like the "/")
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```



To test your script, put the Designer in Preview Mode, and click the Button component. Then, open your myExport.csv file and check your data.



SELECTING A PATH DIRECTORY

Instead of hardcoding the path as we did in the above example, we could ask the user to select a directory on the local system with **system.vision.saveFile**:

User Selected Directory

```
# Create a variable that references our Power Table. You could modify this
part
# of the example to point to a different component in the window.
component = event.source.parent.getComponent('Power Table')

# Use system.dataset.toCSV to turn the dataset into a CSV string.
csv = system.dataset.toCSV(component.data)

# Use system.vision.saveFile to have the user find a directory to write to.
filePath = system.vision.saveFile("myExport.csv", "csv", "Comma Separated
Values")

# We can check the value of filePath to make sure the user picked a path
before
# attempting to write.
if filePath:
    system.file.writeFile(filePath, csv)
```

Perspective Example

This example will demonstrate how to generate a CSV file from a Table component that contains a dataset. Note that you can run the script in this example using the default data for a Table component, but since it is a JSON array structure and not a dataset, the CSV file format will result in a single row of data.

1. Drag a **Table** component to your view.
2. Right-click on the `data` property and select **Change to** > **Value**.

```
{
"$": [
    "ds",
    192,
    1787772403105
],
"$columns": [
    {
    "name": "Col 1",
    "type": "Integer",
    "data": [
        44,
        86,
        0,
        78,
        20,
        21
    ]
    },
    {
    "name": "Col 2",
    "type": "String",
    "data": [
        "Test Row 2",
        "Test Row 3",
        "Test Row 8",
        "Test Row 9",
        "Test Row 10",
        "Test Row 13"
    ]
    },
    {
    "name": "Col 3",
    "type": "Double",
    "data": [
        1.8713151369491254,
        97.4913421614675,
        20.39722542161364,
        34.57127071614745,
        76.41114659745085,
        13.880548366871926
    ]
    }
]
}
```

Col 1	Col 2	Col 3
44	Test Row 2	1.87
86	Test Row 3	97.49
0	Test Row 8	20.4
78	Test Row 9	34.57
20	Test Row 10	76.41
21	Test Row 13	13.88

3. Drag a **Button** component to the view, and select **Configure Events** from the Button's right-click menu to open the Event Configuration popup.

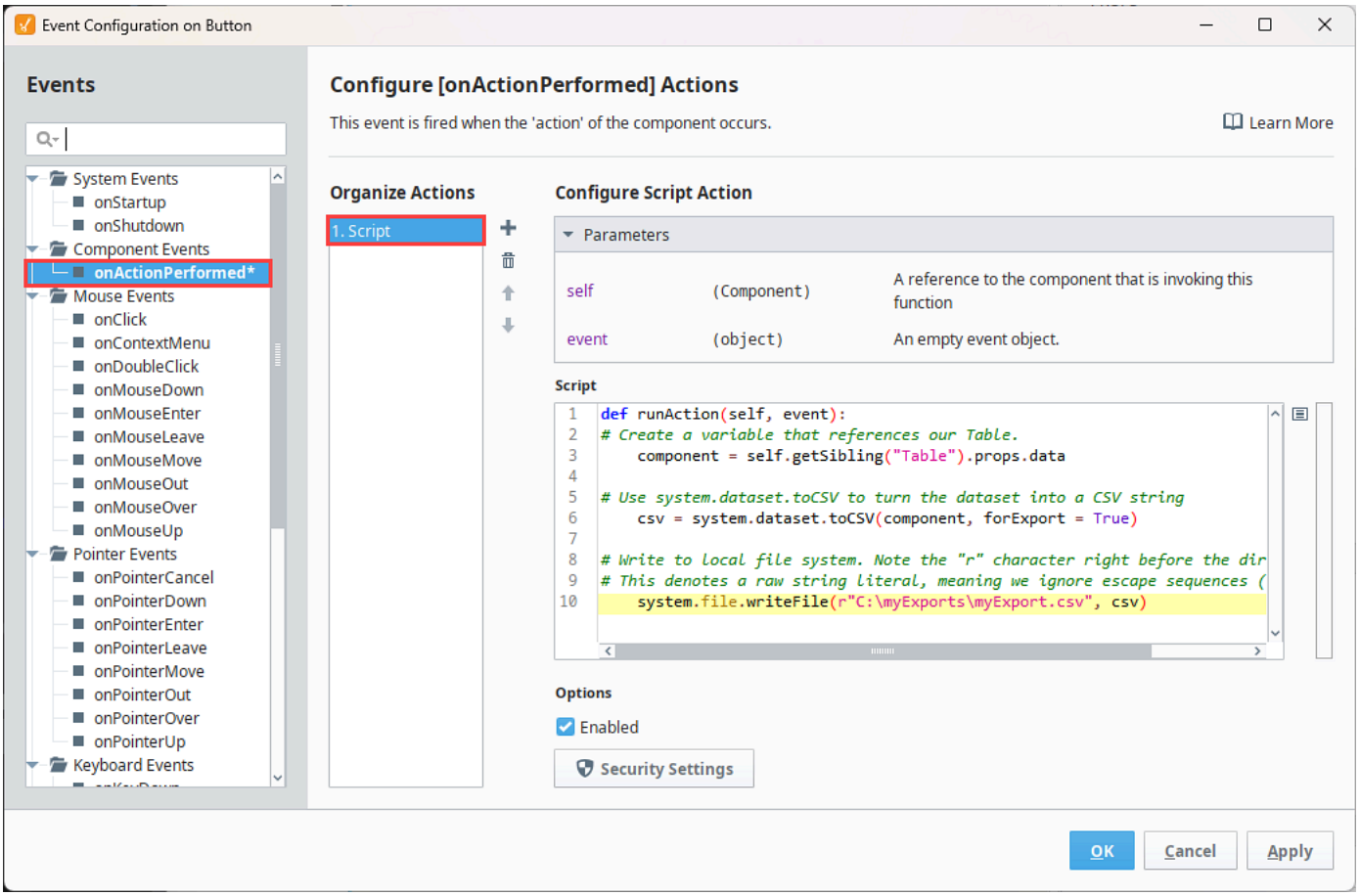
4. Select **Component Events** > **onActionPerformed**.

5. Click the Add icon and select **Script**.

6. Copy and paste the script example below into the Script field.

Exporting Data

```
# Create a variable that references our Table.
    component = self.getSibling("Table").props.data

# Use system.dataset.toCSV to turn the dataset into a CSV string
    csv = system.dataset.toCSV(component)

# Write to local file system. Note the "r" character right before the
directory path.
# This denotes a raw string literal, meaning we ignore escape sequences (like
the "/")
    system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```



To test your script, put the Designer in Preview Mode, and click the Button component. Then, open your myExport.csv file and check your data.

> **SELECTING A PATH DIRECTORY**
>
> Instead of hardcoding the path as we did in the above example, we could ask the user to select a directory on the local system with [system.perspective.download]:
>
> User Selected Directory
>
> ```
> # Create a variable that references our Table.
>     component = self.getSibling("Table").props.data
>
> # Use system.dataset.toCSV to turn the dataset into a CSV string
>     csv = system.dataset.toCSV(component)
>
> # Have the user find a directory to write to.
>     system.perspective.download("myExport.csv",csv)
> ```

Importing Data from a CSV

There are several ways to import data from a CSV file. First, we could use **system.file.readFileAsString** to read the entire file as a string. Note, that this will read the file as is, meaning `\n` can be used to denote new lines.

Python - Using system.file.readFileAsString()

```
# Specify the CSV path in the local file system.
file = "C:\\my_dataset.csv"

# Use readFileAsString to read the contents of the file as a string.
# This string will be the parameter we pass to fromCSV below.
stringData = system.file.readFileAsString(path)

# Split stringData into a List of strings, delimited by the new line character
stringData = stringData.split("\n")

# Iterate through the list, and do something with each line
for i in range(len(stringData)):

    # We're printing the row here, but you could do something more useful with the
data.
    print stringData[i]
```

Alternatively, Python's CSV Library could be use to read in the contents of a CSV. In some cases, this is the easier approach, as the reader object is ready to be iterated over. Note, that this approach does read in each row as a List of strings:

Python - Using csv.reader()

```
# Import Python's built-in csv library
import csv

# Specify the CSV path in the local file system.
file = "C:\\my_dataset.csv"

# Create a reader object that will iterate over the lines of a CSV.
# We're using Python's built-in open() function to open the file.
file = open(path)
csvData = csv.reader(file)

# Iterate through the reader object, and do something with each line
for row in csvData:

    # We're printing the row here, but you could do something more useful with the
data.
    print row

# Close the file once we're done with it
file.close()
```

Converting the Data into a Dataset

If you want to move CSV data into a component, it will need to be converted into a dataset so that it fits into the component's **data** property. There are a couple of functions that can be used to accomplish this depending on the format of your CSV file.

Calling the system.dataset.fromCSV Function

The **system.dataset.fromCSV** function can take a string and convert it to a dataset. Note, that the function expects a very specific format:

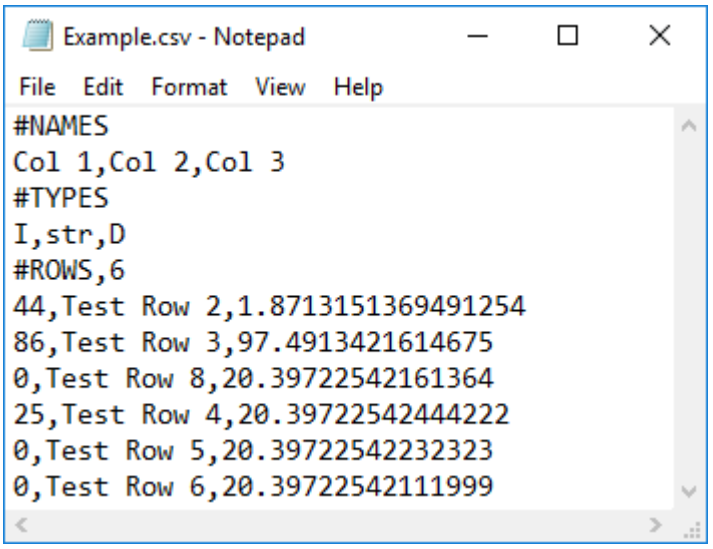
CSV File Content

```
```

```
#NAMES
Col 1,Col 2,Col 3
#TYPES
I,str,D
#ROWS,6
44,Test Row 2,1.8713151369491254
86,Test Row 3,97.4913421614675
0,Test Row 8,20.39722542161364
25,Test Row 4,20.39722542444222
33,Test Row 5,20.39722542232323
62,Test Row 6,20.39722542111999
```

Vision Example

1. Create a text file on your local system named **example.csv** containing the CSV file Content data.



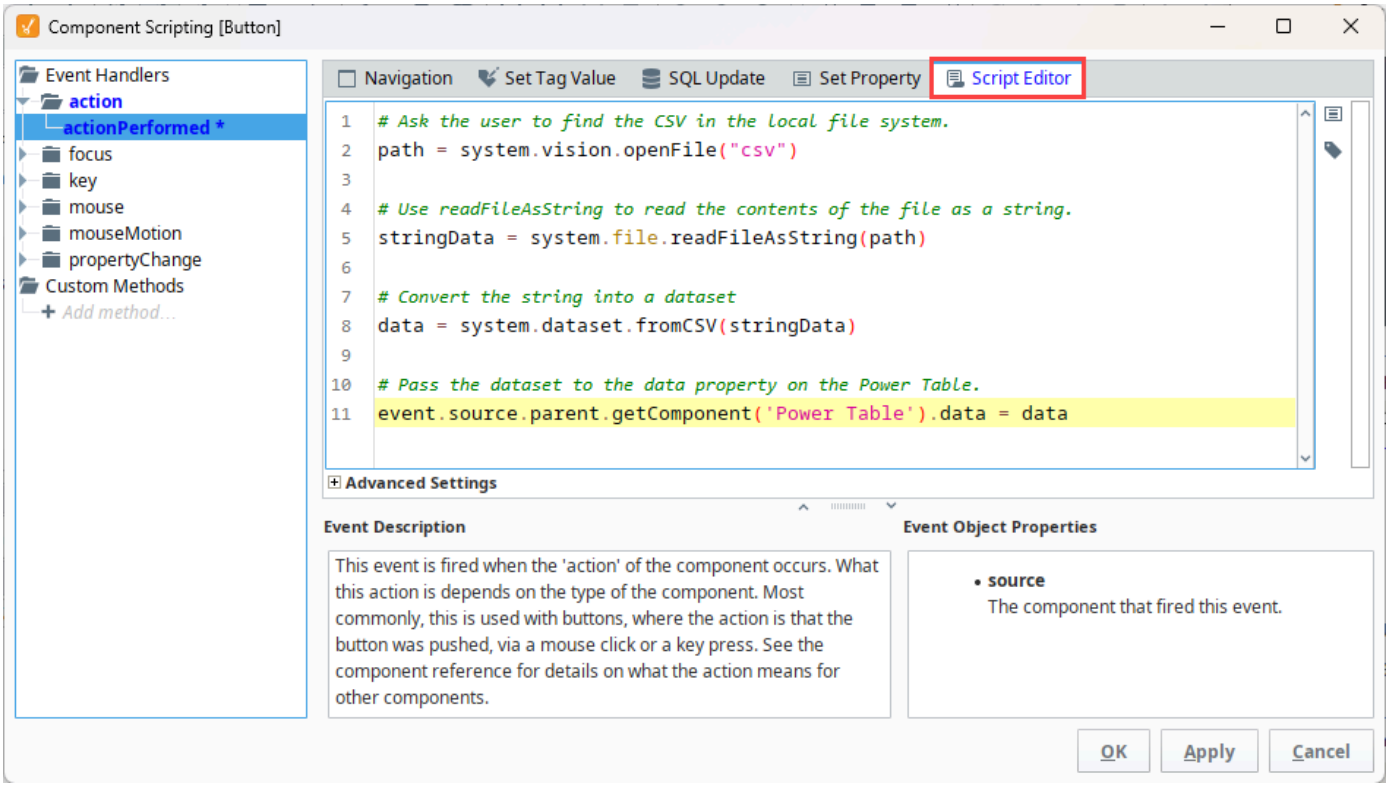
2. Add a **Power Table** and a **Button** component to your window.

3. Select **Scripting** from the right-click menu for the Button component, and paste the following code into the **Script Editor** of the actionPerformed event:

Python - Using system.dataset.fromCSV()

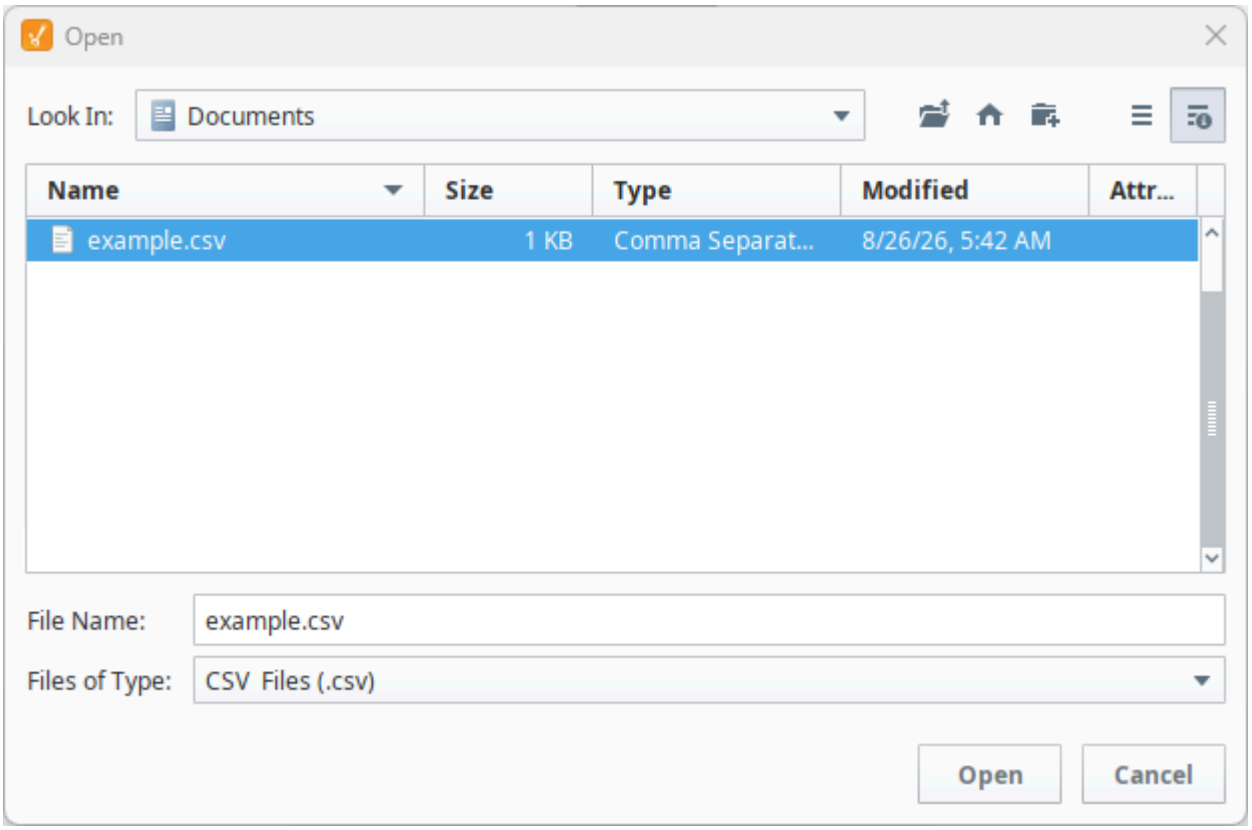
```
```

```
# Ask the user to find the CSV in the local file system.
path = system.vision.openFile("csv")

# Use readFileAsString to read the contents of the file as a string.
stringData = system.file.readFileAsString(path)

# Convert the string into a dataset
data = system.dataset.fromCSV(stringData)

# Pass the dataset to the data property on the Power Table.
event.source.parent.getComponent('Power Table').data = data
```



4. Put the Designer into Preview Mode, and click the **Button**.

5. A window will open for you for you to navigate and choose your CSV file, then click Open.



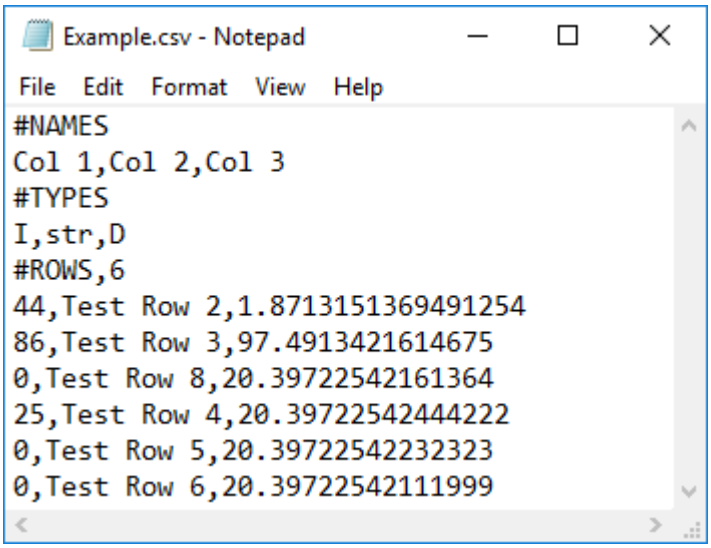
Your data will be displayed in the Power Table as shown below.

Col 1	Col 2	Col 3
44	Test Row 2	1.87
86	Test Row 3	97.49
0	Test Row 8	20.4
25	Test Row 4	20.4
33	Test Row 5	20.4
62	Test Row 6	20.4

Button

Perspective Example

1. Create a text file on your local system named **example.csv** containing the CSV file Content data.



2. Add **Table** component and a **Button** component to your view.

3. Select **Configure Events** from the Button's right-click menu to open the Event Configuration popup.

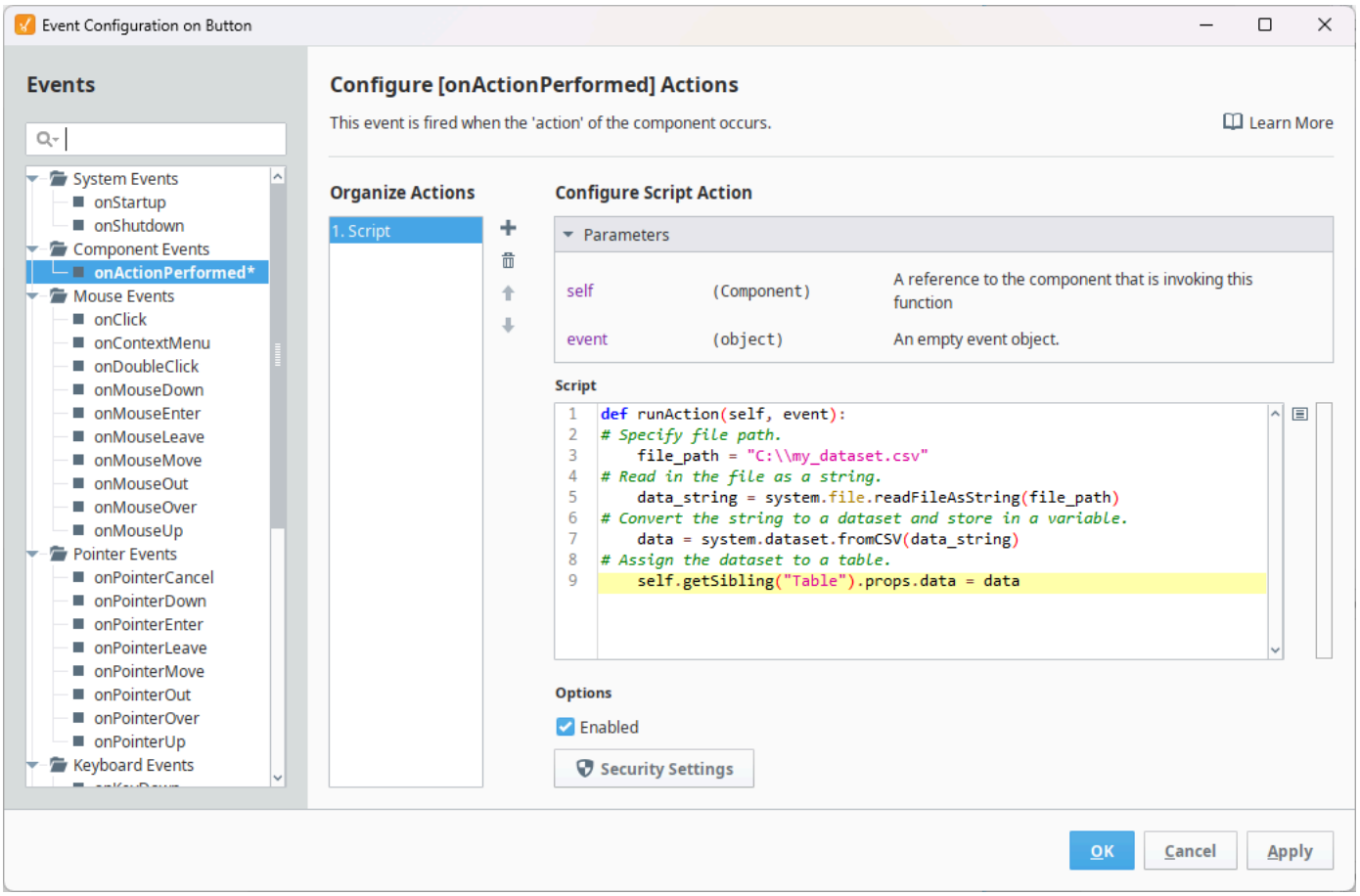
4. Select **Component Events** > **onActionPerformed**.

5. Click the Add icon and select **Script**.

6. Paste the following code into the **Script Editor** of the actionPerformed event:

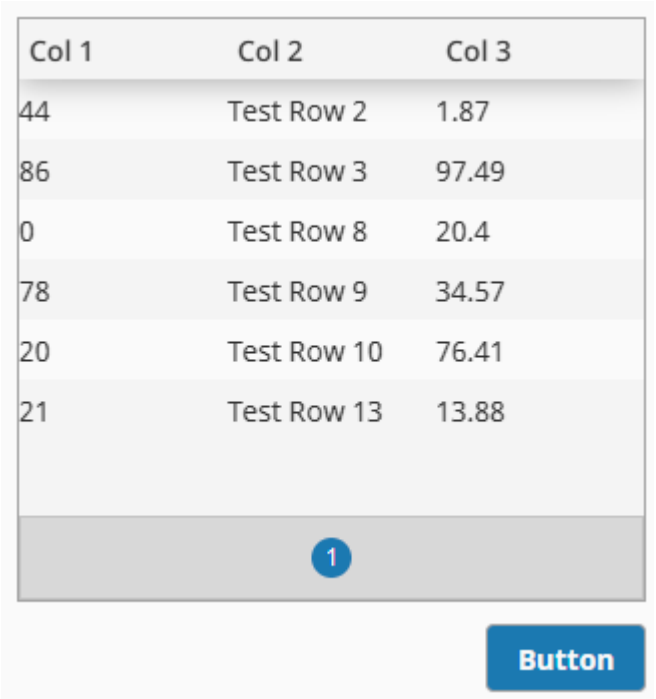
Python - Using system.dataset.fromCSV()

```
# Specify file path.
        file_path = "C:\\my_dataset.csv"
# Read in the file as a string.
data_string = system.file.readFileAsString(file_path)
# Convert the string to a dataset and store in a variable.
        data = system.dataset.fromCSV(data_string)
# Assign the dataset to a table.
        self.getSibling("Table").props.data = data
```



7. Put the Designer into Preview Mode, and click the **Button**.

Your data will be displayed in the Table as shown below. Notice that the default array structure has been changed to value.



Calling the csv.reader Function

As mentioned, `system.dataset.fromCSV()` requires a specific format, which may not match the format of your file. In this case, we can use Python's CSV Library to parse the file and convert it to a dataset.

CSV File Content

```
Col 1,Col 2,Col 3
44,Test Row 2,1.8713151369491254
86,Test Row 3,97.4913421614675
0,Test Row 8,20.39722542161364
25,Test Row 4,20.39722542444222
33,Test Row 5,20.39722542232323
62,Test Row 6,20.39722542111999
```

Below is Vision code example to import the Python CSV data. Specifying the file path directly and replacing `event.source.parent.getComponent('Power Table').data = dataset` with `self.getSibling("Table").props.data = dataset` will enable this script to work for a Perspective Table component.

Python - Using Python's CSV Library

```python
# Import Python's built-in CSV library.
import csv

# Ask the user to find the CSV in the local file system.
path = system.vision.openFile("csv")

# Create a reader object that will iterate over the lines of a CSV.
# We're using Python's built-in open() function to open the file.
file = open(path)
csvData = csv.reader(file)

# Create a List of strings to use as a header for the dataset. Note that the
number
# of headers must match the number of columns in the CSV, otherwise an error will
occur.
# The simplest approach would be to use next() to read the first line in the file,
and
# store that at the header.
header = csvData.next()

# Create a dataset with the header and the rest of our CSV.
dataset = system.dataset.toDataset(header, list(csvData))

# Store it into the table.
event.source.parent.getComponent('Power Table').data = dataset

# Close the file
file.close()
```

Adding a Delay to a Script

Overview

In some cases, having a script execute after a delay is preferable. A common use case is waiting for some event elsewhere in the system to finish: a tag change script executes that needs to wait for a new value from a separate tag. One approach to this is to trigger our script, and then hold or wait until the other event occurs. On this page, we'll take a look at a couple of different approaches to this problem.

It is important to note that pausing a script can cause your client to lock. It is often preferred to look for another event to trigger the script you need. For the example above where we are waiting on a tag to change, you might be able to use the tag change to fire your script instead of waiting in the original script. It is up to you to determine the best trigger based on what exactly your script does.

Using the system.vision.invokeLater Function

The **system.vision.invokeLater** function is a great way to add a delay mid-way through the script. Simply create a function that represents all of the work that should occur after the delay, and pass the function to invokeLater, along with a delay period.

The example below calls two Message Boxes: once initially when pressed, and the other after a three second delay.

Python - Two Message Boxes

```
message = "All Done!"

# Create a function that will be called by invokeLater.
def runThisLater():
    system.vision.showMessage(message)

# Call invokeLater with a 3000ms delay. Note that our function will not run
immediately
# because invokeLater always executes once the rest of this script is complete.
system.vision.invokeLater(runThisLater, 3000)

# Bring up another Message Box. This will appear before the "All Done!" message,
because of invokeLater.
system.vision.showMessage("Waiting...")
```

One of the main limitations with invokeLater is that you can not pass parameters to the function that will be called. Parameters need to be initialized and determined elsewhere in your script, usually in the function definition.

Using a Timer from Python's Threading Library

The Threading Library has a Timer function that works in a very similar fashion to invokeLater. The main difference is that you can pass parameters to the function parameter when calling the Timer. Take a look at **Python's official documentation** for more details on the Timer object.

The example below will again call two Message Boxes with a three second delay between them. However, the text that is defined in the second Message Box is specified when starting the Timer.

Python - Python Threading Timer

```
from threading import Timer

# Create a function that will be called by the Timer.
def runThisLater(param):
    system.vision.showMessage(str(param))

# Constructs a Timer object that runs a function after a specified interval of
time has passed.
# Note the start() at the end: this is required to start the Timer. Don't forget
this part!
Timer(3.0, runThisLater, ["Stop Waiting, I'm done"]).start()

# Bring up another Message Box. This will appear before the "All Done!" message,
because of the Timer
system.vision.showMessage("Waiting")
```

Another benefit to using the Timer, is you can cancel the execution of the Timer's action using the cancel() function, but it only works if the Timer is still in its waiting stage.

Calling time.sleep from Python's Time Library

The simplest approach to pausing a script is to use the sleep() function in Python's Time Library:

Python - Sleeping the Code

```
```

```
from time import sleep

# This will pause execution of the script for 3 seconds. After that time, the
script will continue.
sleep(3)

print "I'm awake!"
```

Using a While Loop

The **While Loop** is another simple approach to adding a delay: simply keep looping until the other event occurs. As always, you will want to take steps to ensure that an infinite loop never occurs: easiest by initializing a counter variable before iterating in the loop, and breaking out if the counter reaches a certain value.

Python - While Loop Safeguard

```
counter = 0

while not otherEvent:

    checkOtherEvent()
    counter += 1

    # Use this to break out if the event takes too long. You'll need the rest of
your script to account for this possibility.
    if  counter >= 10000:
        print "Took too long. Leaving the While Loop"
        break
```

Approaches to Avoid - Locking the Client

While the sleep() and while loop functions are simple to use, they can cause problems by locking up the client and because of this, we generally recommend avoiding them if possible. Typically, **system.vision.invokeLater** on a delay can accomplish the same task.

Note, that there are use cases for both approaches outlined below especially so in regard to the While Loop. However, neither function should be used to force a delay in a Client.

DANGER

> Using either of these methods to pause or delay a script can lock up the entire client, which may be very dangerous.

Reasons to Avoid time.sleep and While Loops

If used on a component, these approaches could lock up the Client or Designer. Scripts called from a component run in the same thread that the Client and Designer use to refresh the screen. Whenever a script on a component triggers, the screen is unable to refresh until the script finishes. In most cases, this is so fast that no one notices. If your script is intentionally calling sleep() or using a long running While Loop, then the Client will lock up for the duration.

Additionally, these approaches run an extended period of time, and block other component based scripts from executing. They do not yield to other scripts while waiting.

Because of these two reasons, the sleep() and While Loop functions can cause your clients to appear unresponsive, when really they are just running a script that prevents the screen from refreshing.

Recommended Alternative

As mentioned earlier, the **system.vision.invokeLater** function (**also mentioned on this page**) can provide the same functionality. If a sleep() or While Loop function must be used, then they should be called from **system.util.invokeAsynchronous** to prevent blocking any other scripts.

Export Tag Historian to CSV

Obtaining Historian Data

Sometimes, it may be useful to export the values from the Tag Historian. There are many ways that this can be done, but the easiest way is to use the built in **system.historian.queryRawPoints** function. This allows you to specify a list of tag paths to query history for.

Python - Grabbing History Data

```
# First we start by creating a start and end time that we will be query history for.
end = system.date.now()
start = system.date.addMinutes(endTime, -30)

# Next we call our queryRawPoints function, using the start and end dates as well as a tag path.
# The other parameters listed for this function can be altered to fit your need.
data = system.historian.queryRawPoints(paths=["[default]myTag"],
startTime=start, endTime=end, returnSize=10, returnFormat='Wide',
includeBounds=True)
```

This simple script will query tag history for a tag called myTag. It will query the last 30 mins of data, return 10 records, includes bounds, and the return format will be wide. For more information on the function parameters and the different options available, see the **queryRawPoints** function page. The data returned is then being stored in a variable **data**, and we can use it later in the script in anyway we want.

Exporting to CSV

Now that we have our data, it is really easy to export it to a CSV file. Ignition has a built-in function to convert a dataset to CSV data called **system.dataset.toCSV**. This CSV data can then be written to an actual file using **system.file.writeFile**.

Python - Export History to CSV

```
# First we start by creating a start and end time that we will be query history
for.
end = system.date.now()
start = system.date.addMinutes(endTime, -30)

# Next we call our queryRawPoints function, using the start and end dates as well
as a tag path.
# The other parameters listed for this function can be altered to fit your need.
data = system.historian.queryRawPoints(paths=["[default]myTag"],
startTime=start, endTime=end, returnSize=10, returnFormat='Wide',
includeBounds=True)

# Turn that history data into a CSV.
csv = system.dataset.toCSV(data)

# Export that CSV to a specific file path. The r forces it to use the raw path and
not require double backslashes.
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```

History Tag Search and Export

If you have a lot of tags storing history, it can be difficult to list them all out in the function above. This example makes things easier by specifying a folder of tags instead of individual tags. The script first grabs all of the historical tags from a particular folder and gets the tag paths using the **system.historian.browse** function, runs those through the **system.historian.queryRawPoints** function, and then exports the results to a CSV file. This script is great, because it can be run from the **scripting console**, or from a button on the Client. However, everything in the script is hardcoded to work with a specific system, though that can be easily changed to fit any system. Below are the list of parts that can be changed with what each of them is used for.

Part of Script	Description
`path='histprov:myDB:/sys:myIgnitionGateway/prov:myTagProvider:/tag:myFolderOfTags'`	The path to the parent folder that we are searching through for history tags. Each part of the string corresponds to a different part of your system. • histprov:myDB:/ - This is the name of your Historical Tag Provider that you are searching in. Replace **myDB** with the name of your Historical Tag Provider. • sys:myIgnitionGateway/prov:myTagProvider - The first thing is the name of your Ignition Gateway. Replace **myIgnitionGateway** with the name of your Ignition Gateway, which can be found by going to the Gateway Configure Webpage and navigating to **System > Gateway Settings**. It is the **System Name**. The second thing is the name of the Realtime Tag Provider where the tags reside. Replace **myTagProvider** with the name of your Tag Provider. • tag:myFolderOfTags - This is the name of th folder you want to search for historical tags. Replace **myFolderOfTags** with the name of a folder in your Tag Browser to search through. **Note:** Paths to the **Internal Historian (Legacy)** (such as those used by Edge Gateways) do not need to include the Gateway name.
`system.historian.queryRawPoints(...)`	This is what will actually query for historical data. While the path parameter should be kept the same, the other parameters can be altered to fit your needs. Currently, the function is querying th last 30 mins of history, has a return size of 10, includes bounds, and has a return format of wide Of course, all of this can be changed. See **system.historian.queryRawPoints** for more

Part of Script	Description
	information on the parameters that can be used here.
`r"C:\myExports\myExport.csv"`	This is the path that the CSV file will save at. This can be changed to be any location you want. You can even use **system.vision.saveFile** to first ask the user for a save location and filename, and then passing the path returned from that function into the writeFile function. The r at the beginning of the string is important as it allows us to not have to use double backslashes.

CAUTION

This script can be dangerous! It recursively looks through a folder to find all historical tags, and then uses all of those tags to look up tag history. Looking through a folder that has too many history tags, or querying history for too large a period of time can potentially result in locking up your system. Use caution when using this function, and never search through too large a set of tags or query too much history.

Python - Recursively Browse for History and Export to CSV

```
# Our browse function that will browse for historical tags.
# By setting this up as a function, it allows us to recursively call it to dig down
through the specified folder.
# Pass in an empty list that we can add historical paths to, and the path to the
top level folder.
def browse(t, path):

    # Loop through the results of the historical tag browse, and append the path to
the empty list.
    for result in system.historian.browse(path).getResults():
        t.append(result.getPath())

        # If the result is a folder, run it through the browse function as well.
        # This will continue until we are as deep as possible.
        if result.hasChildren():
            browse(t, result.getPath())

# Start with an empty list to store our historical paths in.
historyPaths = []

# Call the browse function, passing in an empty list, and the folder that we want
to browse for historical tags.
# This path is a placeholder. It should be replace with your valid path.
browse(historyPaths,
path='histprov:myDB:/sys:myIgnitionGateway/prov:myTagProvider:/tag:myFolderOfTags')

# Create another empty list to store our tag paths that we will pull out of the
historical paths.
tagPaths = []

# Loop through the list of historical tag paths, split out just the tag path part,
# and push it into our tag path list.
for tag in historyPaths:
    tagPaths.append("[myTagProvider]" + str(tag).split("tag:")[1])

# Now that we have a list of tag paths, we need to grab the historical data from
them.
# Start by creating a start and end time.
end = system.date.now()
start = system.date.addMinutes(endTime, -30)

# Then we can make our query to tag history, specifying the various parameters.
# The parameters listed for this function can be altered to fit your need.
data = system.historian.queryRawPoints(paths=tagPaths, startTime=start,
endTime=end, returnSize=10, returnFormat='Wide', includeBounds=True)

# Turn that history data into a CSV.
csv = system.dataset.toCSV(data)
```

```
# Export that CSV to a specific file path. The r forces it to use the raw path with backslashes.
system.file.writeFile(r"C:\myExports\myExport.csv", csv)
```

Parsing XML with Java Libraries

XML DOCUMENT SECURITY

Parsing XML can be potentially dangerous without hardening the parser implementation. Best practices on how to prevent attackers from exploiting vulnerabilities can be found here.

NOTE

You will need to place your script in the current working directory where the script will be running. For example, if your script affects the Designer Launcher, put your script in the directory your Designer Launcher is installed.

What is the DOM Parser?

The Document Object Model (DOM) parser provides a powerful way to parse and manipulate XML documents. It's commonly used due to its ease of use and comprehensive functionality. The DOM parser breaks down XML into accessible elements, each representing a node in the XML tree structure. For more information on interfacing with the DOM parser, refer to the **Java XML DOM Parser Documentation**.

Using the DOM Parser

There are several ways to import XML data using the DOM parser, depending on how it's stored. It can retrieve data from an XML file using the file path or from a string. Regardless of the method, it provides a root object representing the XML document.

Jython - Reading a File

```
from javax.xml.parsers import DocumentBuilderFactory
from java.io import File

# Define your XML file path
xmlFilePath = "file.xml"  # Replace with your actual XML file path

# Create a DOM document builder
builderFactory = DocumentBuilderFactory.newInstance()
builder = builderFactory.newDocumentBuilder()

# Parse the XML file
file = File(xmlFilePath)
document = builder.parse(file)

# Access the root element
root = document.getDocumentElement()
```

Jython - Reading from a String

```
from javax.xml.parsers import DocumentBuilderFactory
from java.io import ByteArrayInputStream

# Define your XML string
xmlString = """
<employee id="1234">
    <name>John Smith</name>
    <start_date>2010-11-26</start_date>
    <department>IT</department>
    <title>Tech Support</title>
</employee>
"""  # Replace with your actual XML string

# Create a DOM document builder
builderFactory = DocumentBuilderFactory.newInstance()
builder = builderFactory.newDocumentBuilder()

# Parse the XML string
stream = ByteArrayInputStream(xmlString.encode('utf-8'))
document = builder.parse(stream)

# Access the root element
root = document.getDocumentElement()
```

Each tag is considered an element object. For instance, in the given example, the root element would be the `employee` tag. Elements can have attributes contained within the tag itself. In the example

above, the `employee` element has an `id` attribute with a value of `1234`. Additionally, elements can have additional data, typically between the start and end tags. This data can be accessed using the Element object's built-in functionality.

Function	Description	Example	Output
Element.tagName	Returns the name of the element's tag.	`print root.tagName`	`employee`
Element.attributes	Returns a dictionary of the element's attributes.	`print root.attributes.item(0)`	`id: "1234"`

Element.textContent	Returns the additional data of the element.	`print root.textContent`	`John Smith`
			`2010-11-26`
			`IT`
			`Tech Support`

Element.item(index)	Allows direct reference to an element's children by index.	`print root.item(5).tagName`	`department`

A Simple Employee Example

Using the functions above, let's parse through a sample XML string and extract employee data. We'll demonstrate how to access different elements and attributes and display them. Let's iterate through the XML elements and print out the following employee details:

Code Output

```
Employee ID: 1
Name: John Doe
Department: Engineering

Employee ID: 2
Name: Jane Smith
Department: Marketing
```

Code Snippet - Extracting Employee Details

```
from javax.xml.parsers import DocumentBuilderFactory
from java.io import ByteArrayInputStream

# Define your XML string
xmlString = """
<employees>
  <employee id="1">
      <name>John Doe</name>
      <department>Engineering</department>
  </employee>
  <employee id="2">
      <name>Jane Smith</name>
      <department>Marketing</department>
  </employee>
</employees>
"""  # Replace with your actual XML string

# Create a DOM document builder
builderFactory = DocumentBuilderFactory.newInstance()
builder = builderFactory.newDocumentBuilder()

# Parse the XML string
document = builder.parse(ByteArrayInputStream(xmlString.encode()))

# Access the root element
root = document.getDocumentElement()

# Iterate through employees
employees = root.getElementsByTagName("employee")
for i in range(employees.getLength()):
  employee = employees.item(i)
  # Convert the id attribute to an integer
  id = int(employee.getAttribute("id"))
  print "Employee ID:", id
  print "Name:", employee.getElementsByTagName("name").item(0).textContent
  print "Department:",
employee.getElementsByTagName("department").item(0).textContent
  print
```

What is the SAX Parser?

The Simple API for XML (SAX) parser, available through Java libraries, provides an event-driven approach to parse XML documents. It's widely used for its efficiency, especially when handling large XML files. SAX parses XML sequentially and triggers events as it encounters elements, attributes, and

other components in the XML document. For more detailed information about the SAX parser, refer to the **Java XML SAX Parser Documentation**.

Using the SAX Parser

The SAX parser doesn't build a tree structure like the DOM parser. Instead, it parses the XML document sequentially and triggers events that the developer can handle. Here's an example of using the SAX parser to parse an XML file:

Java - Reading a String

```
from javax.xml.parsers import SAXParserFactory
from org.xml.sax.helpers import DefaultHandler
from java.io import ByteArrayInputStream

# Define your XML string
xmlString = """
<employees>
    <employee id="1">
        <name>John Doe</name>
        <department>Engineering</department>
    </employee>
    <employee id="2">
        <name>Jane Smith</name>
        <department>Marketing</department>
    </employee>
</employees>
"""  # Replace with your actual XML string

# Define a custom ContentHandler
class MyContentHandler(DefaultHandler):
    def startElement(self, uri, localName, qName, attributes):
        print("Start Element:", qName)
        for i in range(attributes.getLength()):
            print("Attribute:", attributes.getQName(i), "=",
attributes.getValue(i))

    def endElement(self, uri, localName, qName):
        print("End Element:", qName)

    def characters(self, ch, start, length):
        print("Character Data:", ch[start:start+length])

# Create a SAX parser
saxParserFactory = SAXParserFactory.newInstance()
saxParser = saxParserFactory.newSAXParser()

# Parse the XML string
stream = ByteArrayInputStream(xmlString.encode('utf-8'))
saxParser.parse(stream, MyContentHandler())
```

In the above example, we define a custom `ContentHandler` class that extends `DefaultHandler`. This class overrides methods to handle idfferent events encountered during XML parsing, such as starting and ending elements, and character data.

What is the StAX Parser?

The Streaming API for XML (StAX) parser, available through Java libraries, offers a cursor-based approach to parse XML documents. It provides an efficient way to read and process XML sequentially without loading the entire document into memory. StAX parsers allow developers to iterate through XML elements, attributes, and other components as they are encountered in the XML stream. For more detailed information about the StAX parser, refer to the **Java XML StAX Parser Documentation**.

Using the StAX Parser

The StAX parser operates in a streaming fashion, allowing developers to read XML content sequentially without the need to build a complete in-memory representation of the XML document. Here's an example of how to create an XML input factory and a stream reader to parse the XML content. We iterate through the XML stream and handle different events such as starting and ending elements, as well as character data.

Code Snippet - Extracting Employee Details

```
from javax.xml.stream import XMLInputFactory, XMLStreamReader
from java.io import ByteArrayInputStream

# Create an XML input factory
inputFactory = XMLInputFactory.newInstance()

# Create an XML stream reader
streamReader =
inputFactory.createXMLStreamReader(ByteArrayInputStream(xmlString.encode()))

# Iterate through the XML stream
while streamReader.hasNext():
    event = streamReader.next()
    if event == XMLStreamReader.START_ELEMENT:
        print("Start Element:", streamReader.getLocalName())
        # Print attributes, if any
        for i in range(streamReader.getAttributeCount()):
            print("Attribute:", streamReader.getAttributeLocalName(i), "=",
streamReader.getAttributeValue(i))
    elif event == XMLStreamReader.END_ELEMENT:
        print("End Element:", streamReader.getLocalName())
    elif event == XMLStreamReader.CHARACTERS:
        print("Character Data:", streamReader.getText())
```

Audit Log and Profiles

Ignition's built-in auditing system automatically records certain actions that occur in the system, such as a Tag writes or User Source authentication, into a SQL database table. Utilizing the system involves creating an Audit Profile, followed by enabling auditing in a project. Once both prerequisites have been met, the Gateway will automatically create a database table named AUDIT_EVENTS, and use the table to start tracking user actions.

The Remote Audit Log configuration option allows audit events to be automatically sent to a remote Gateway's Audit Log. The remote Gateway you plan to connect to must have a Audit Profile created. To learn more about sending audit events to a remote Gateway, refer to section **Creating a Remote Gateway Audit Profile** on this page.

> **NOTE**
>
> You can use Audit Profiles for Gateway events and project events. See the **Enabling Auditing for Gateway-Scoped Actions** or the **Enabling Auditing in a Project** sections for more details.

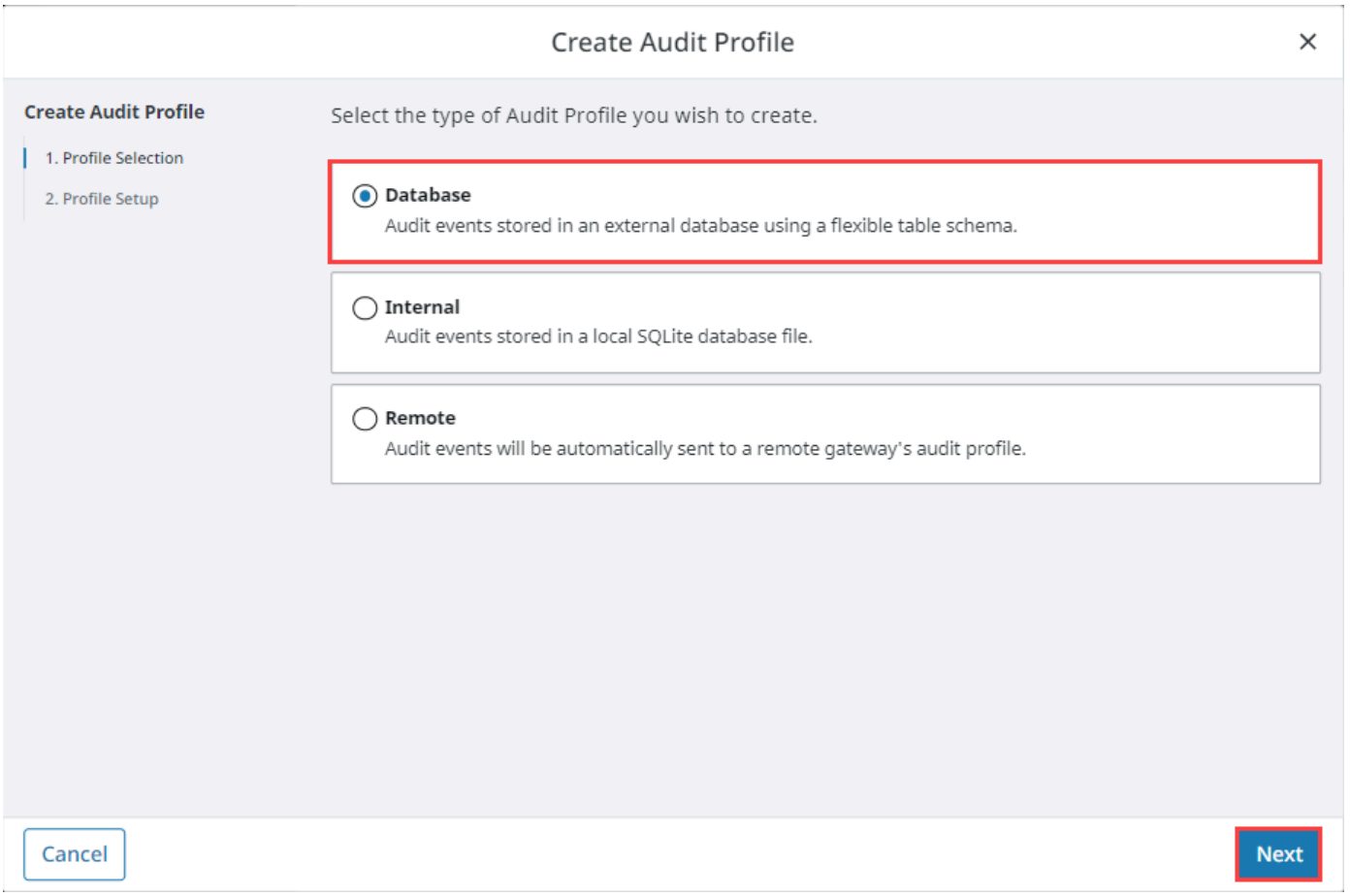
Auditing Actions

For a list of actions that are recorded by an audit profile, see the **Auditing Actions Reference** page.

Create a Database Audit Profile

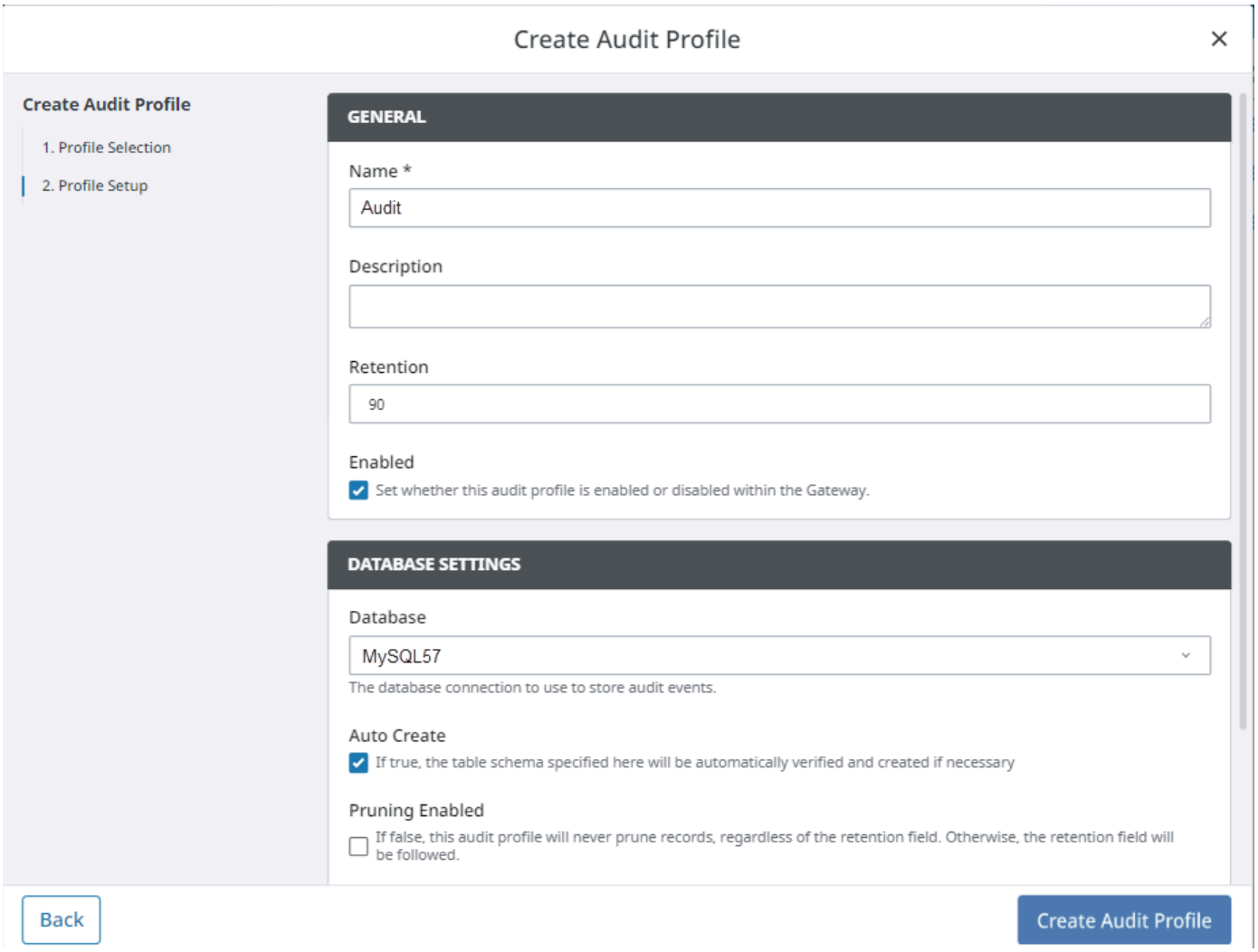
1. Go to the **Platform** section of the Gateway Webpage.
2. Select **Security > Audit Profiles** from the menu on the left. The Audit Profile Logs page is displayed.
3. Click **Create Audit Profile +**.

4. Select **Database** and then **Next**.



5. Enter the **Name** of the Audit Log and **Description** if desired.
6. In the Retention field, set a value in days for how long you want audit records kept. The default is 90 days.
7. Under the **Database Settings**, select the **Database** where the table will be stored, leave the **Auto Create** box checked.
8. Enter a desired title im the **Table Name** field.

9. Click **Create Audit Profile**.



Once some changes have been made to a tag or a Database table, Ignition will begin recording.

Database Audit Profile Properties Table

Main

Name	Description
Name	The default name, is the name of the Audit Profile.
Description	Description of the audit profile. Optional.
Retention	How long (in days) should audit records be kept? Values less than or equal to 0 will disable pruning. Default is 90 days.
Enabled	Enables/Disables the Audit Profile within the Gateway. Profiles are enabled by default.

Database Settings

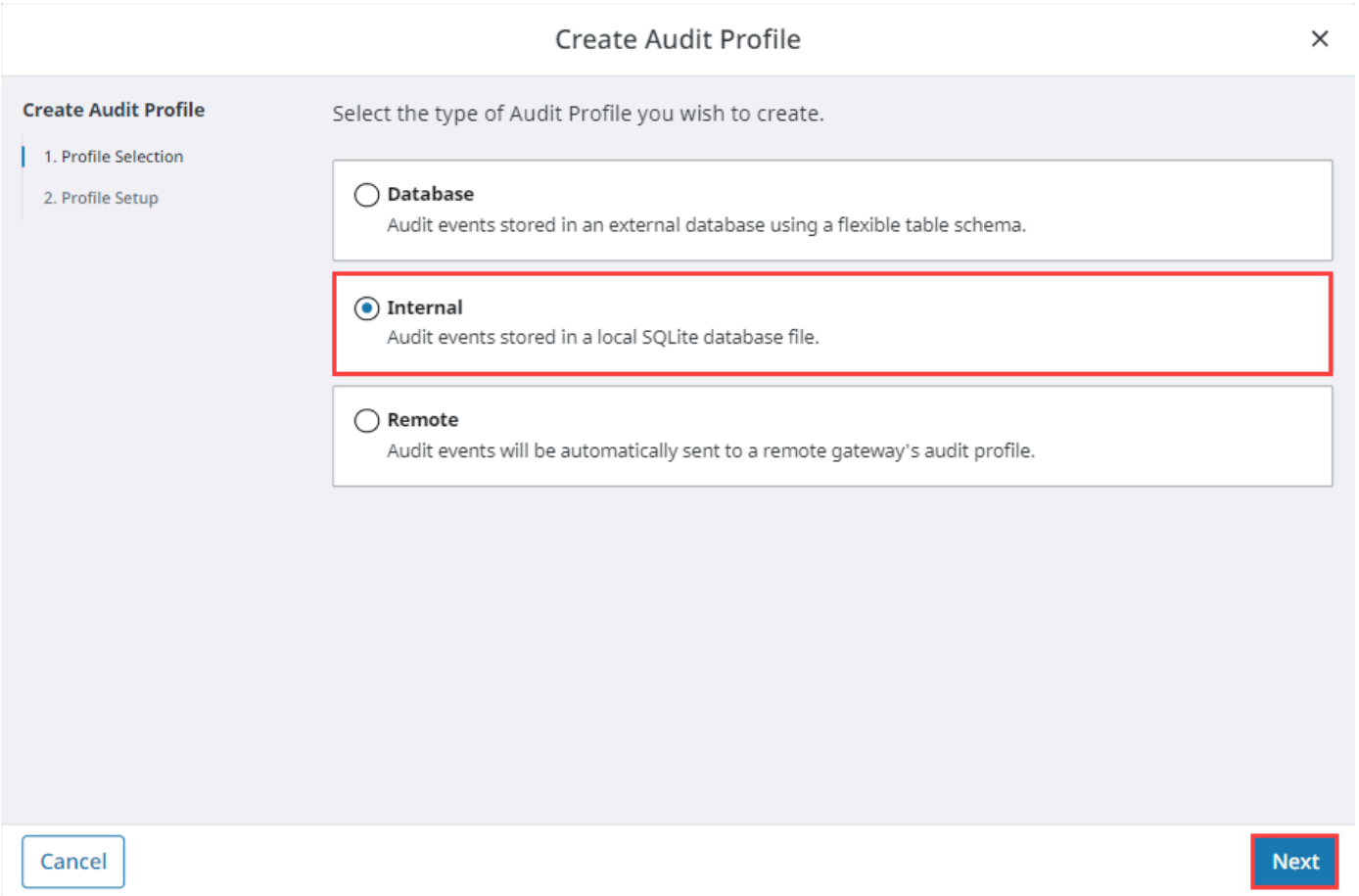
Name	Description
Database	The database connection to use to store audit events.
Auto Create	If true (selected), the table schema specified here will be automatically verified and created if necessary. Default is true.
Pruning Enabled	If false, this audit profile will never prune records, regardless of the retention field. Otherwise, the retention field will be followed. Default is false.
Table Name	The name of the table to store audit events. Default is AUDIT_EVENTS.

Create an Internal Audit Profile

The Internal Audit Profile option allows an Ignition Gateway to store audit records without an external SQL database.

1. Go to the **Platform** section of the Gateway Webpage.
2. Scroll down to the **Security > Audit Profiles** from the menu on the left. The Audit Profile Logs page is displayed.
3. Click **Create Audit Profile +**.

4. Select **Internal**.



5. Enter a name for the Audit Log.
6. In the Retention field, set a value in days for how long you want audit records kept. The default is 90 days.
7. Click **Create Audit Profile**.

Internal Audit Profile Properties Table

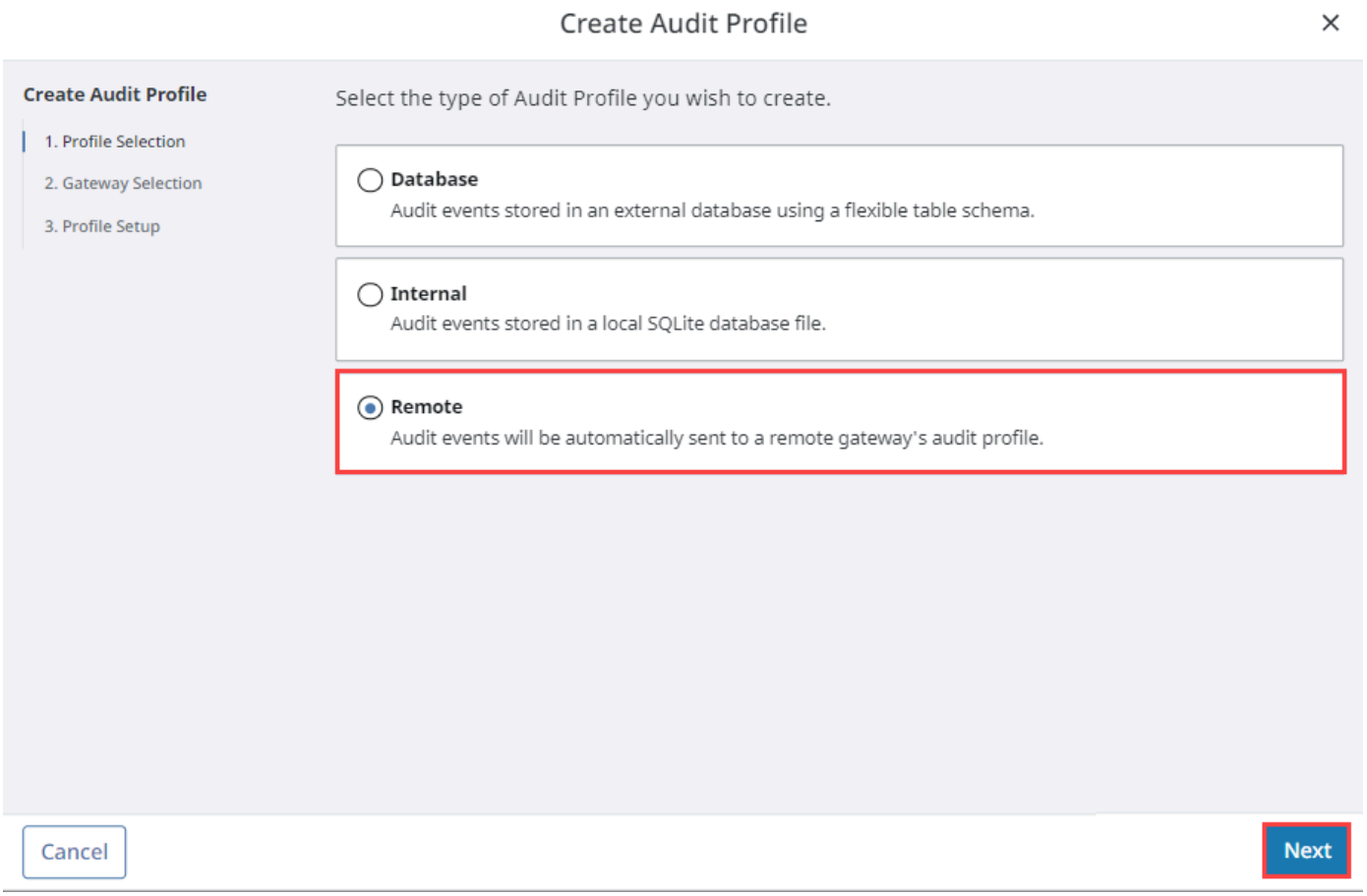
Name	Description
Name	The default name, is the name of the Audit Profile.
Description	Description of the audit profile. Optional.
Retention	Value in days for how long you want audit records kept. Default is 90 days.
Enabled	Enables/Disables the Audit Profile within the Gateway. Profiles are enabled by default.

Creating a Remote Gateway Audit Profile

Just like configuring audit events to be logged into an external database, it is done from the Gateway Webpage, **Platform > Security > Audit Profiles**.

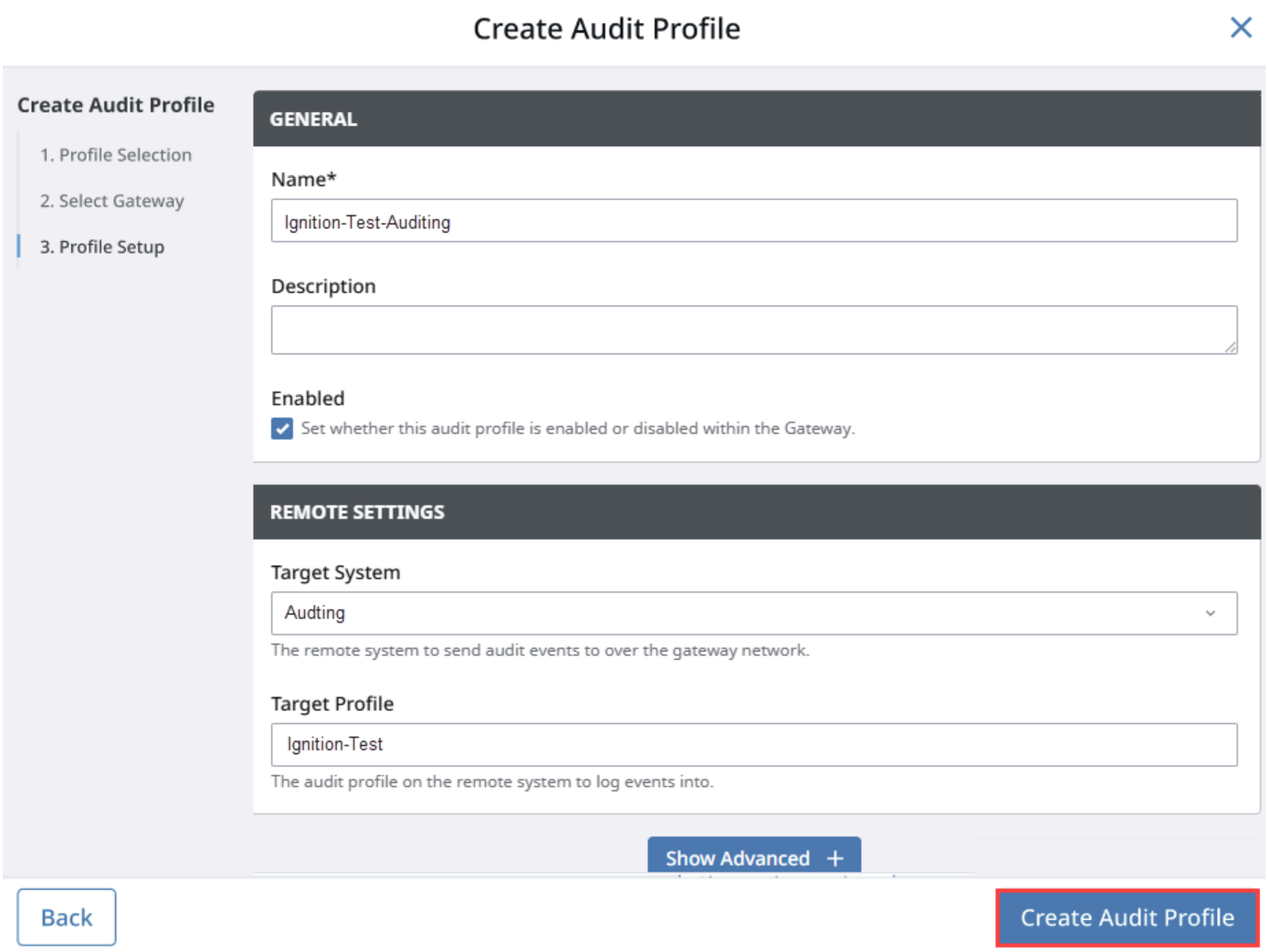
> **CAUTION**
>
> Due to serialization updates, 8.3 Gateways will not be able to store Audit Log data to a remote 8.1 Gateway. It is recommended to first upgrade the central server hosting data to 8.3 before upgrading any remote 8.1 Gateways. See the 8.3 Upgrade Guide for more information.

1. To have your audit events automatically sent to a remote Gateway's audit profile, select **Remote**, and click **Next**.



2. A list of known Gateways will be displayed. If you don't see a Gateway that you expected to see, check your Gateway Network settings to verify that the connections are valid. You also have the option to specify a Gateway manually. Once you've selected a valid Gateway, click **Next**.

3. If an Audit profile exists, the fields will auto-populate. The name of the Gateway will appear in the **Name** field prefaced with the Audit Profile name (i.e., Ignition_Test_Auditing), as shown in the following example. Click **Create Audit Profile**.



You will now receive a successful message stating your new Audit Profile was created.

Remote Gateway Audit Profile Properties Table

Main

Name	Description
Name	The default name, is the name of the Remote Gateway and Audit Profile.
Description	Description of the audit profile. Optional.
Enabled	By default, the journal profile is enabled.

Remote Settings

Name	Description
Target System	The remote system to send audit events to over the Gateway network.
Target Profile	The audit profile on the remote system to log events into.

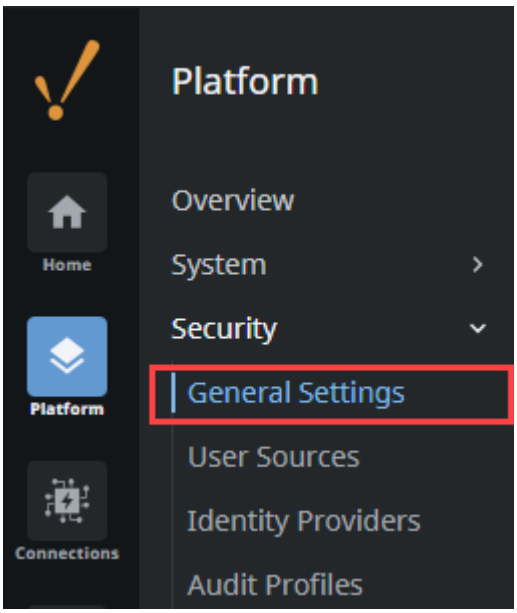
Advanced

Name	Description
Use Store and Forward	If enabled, audit events will be stored through the Store and Forward system. If not enabled, they will be stored directly against the remote Gateway. Default is true.
Max Group Size	The maximum number of data points that can be sent per request. This value is used in conjunction with the Store and Forward setting to dictate how much data is sent at once. The default value of 0 means unlimited data points per request.

Enabling Auditing for Gateway-Scoped Actions

After setting up an Audit Profile, you can have the Audit Log record Gateway events.

1. Navigate to the **Gateway General Security Settings** page. This is located under **Platform > Security > General Settings**.



2. Set the **Gateway Audit Profile** setting to the audit profile you want to record with, then save your settings.

A confirmation banner will now appear at the top of the page and your Gateway will begin recording events to the specified audit profile.

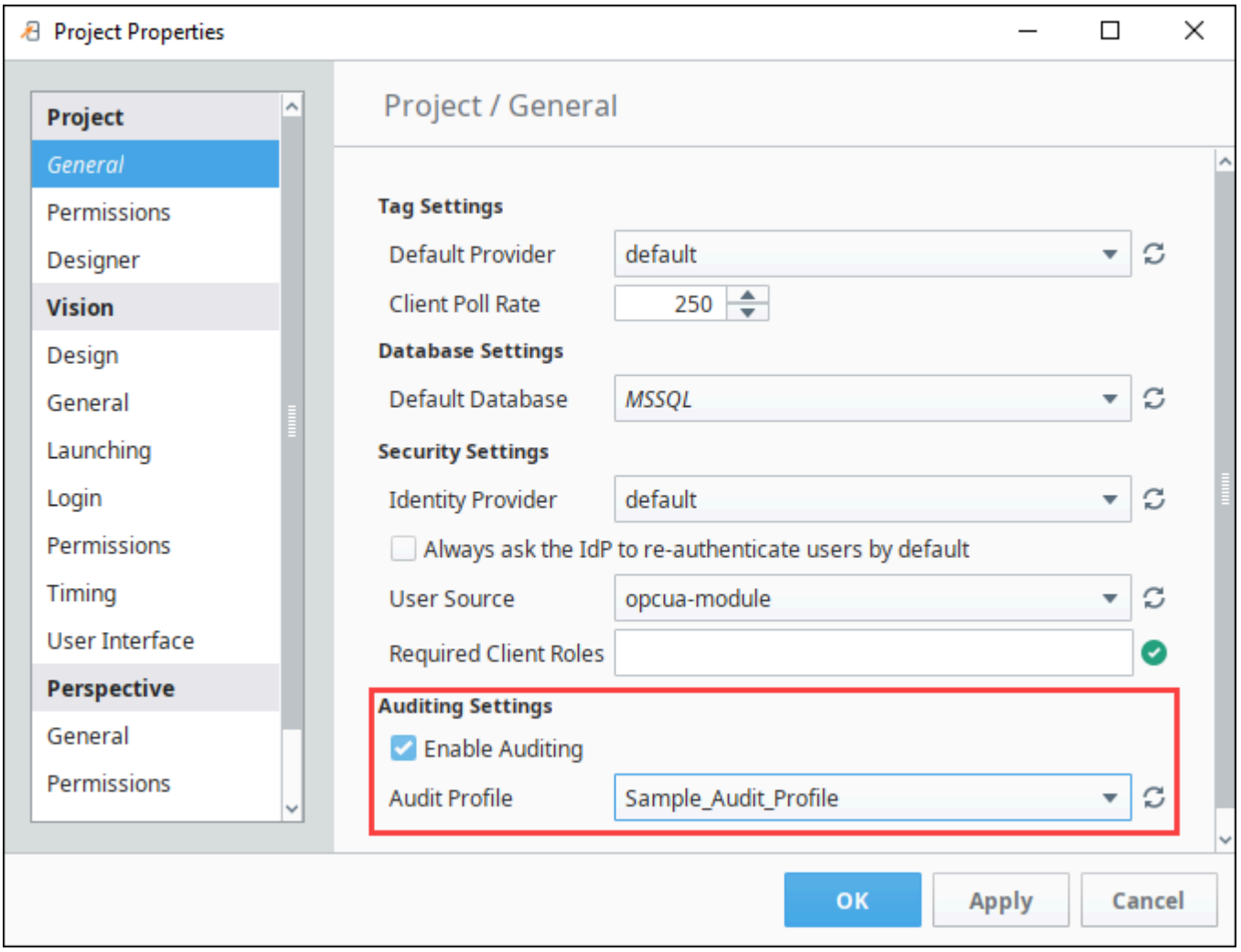
Enabling Auditing in a Project

1. Go to the Designer, open the project that you want to enable auditing on, then go to **Project > Properties**.

2. Go to the General section, select the **Enable Auditing** check box, and select your **Audit Profile** from the drop-down menu. The audit profile is used to record audit actions for your project. If the new audit profile does not show up, click **Refresh**.

3. Click **OK**.

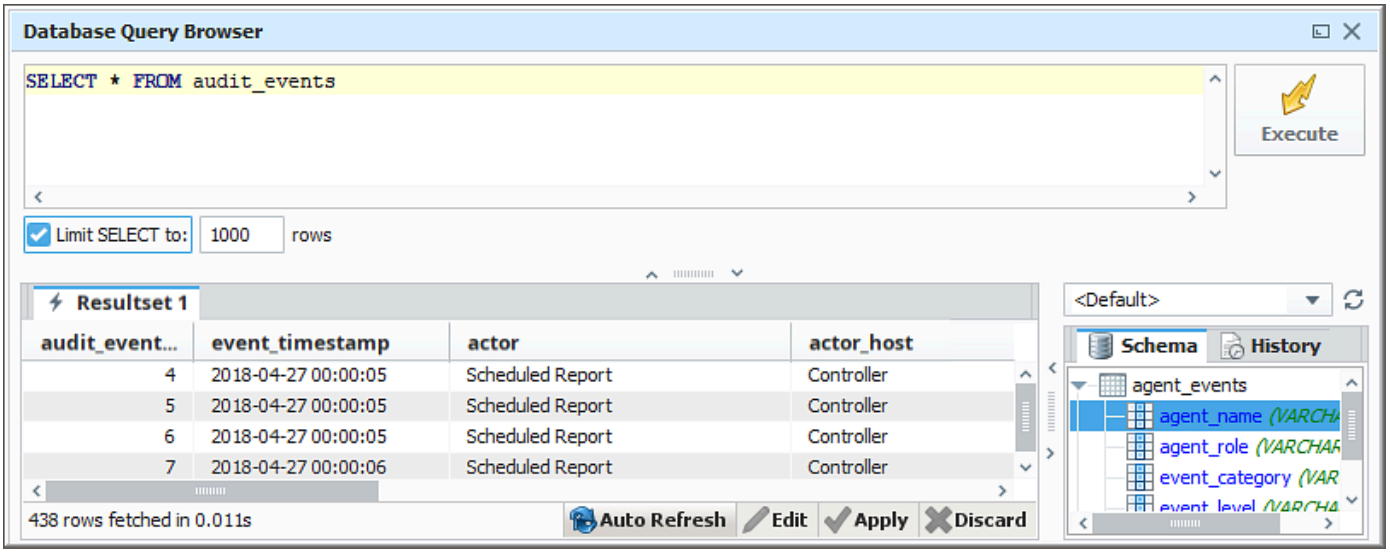
4. **Save** your project.



Viewing Information in an Audit Log

There are a few ways to **view audit information**: using a Table component, interface on the Gateway, or the Database Query Browser. Here is one example of viewing an Audit Log using the Database Query Browser.

1. In the Designer, go to **Tools > Database Query Browser**.
2. Under the **Schema** area, double click on a **table**, and it will expand the query in the Database Query Browser area.
3. Click **Execute**. All the Audit Log data will be displayed in the Resultset1 area.



Audit Log Table Descriptions

A description of the columns used by the Audit Log's database table can be found on the **Ignition Database Table Reference** page.

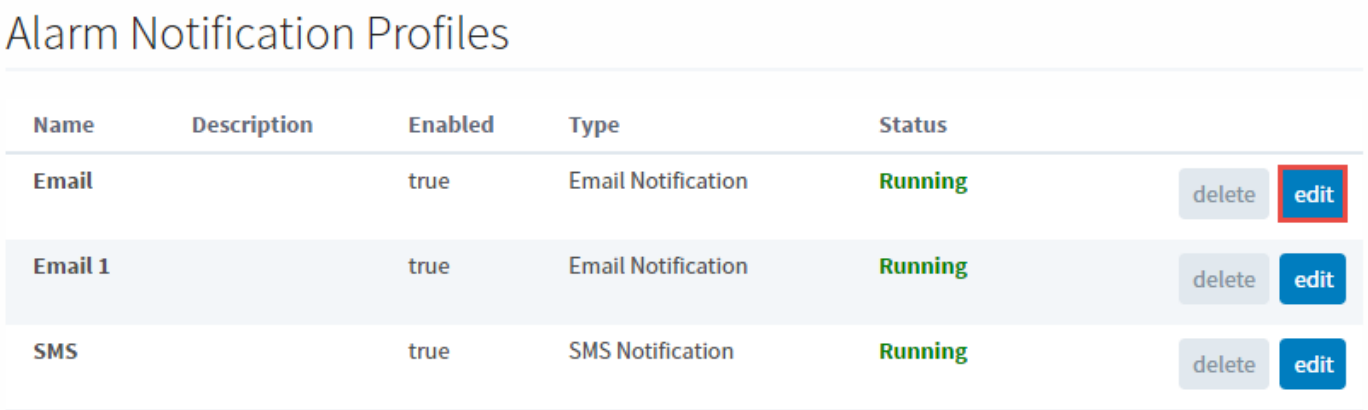
Alarm Notification Auditing

Alarm Notification profiles can be set to store information in an Audit Profile.

How to Store Alarm Notifications in an Audit Profile

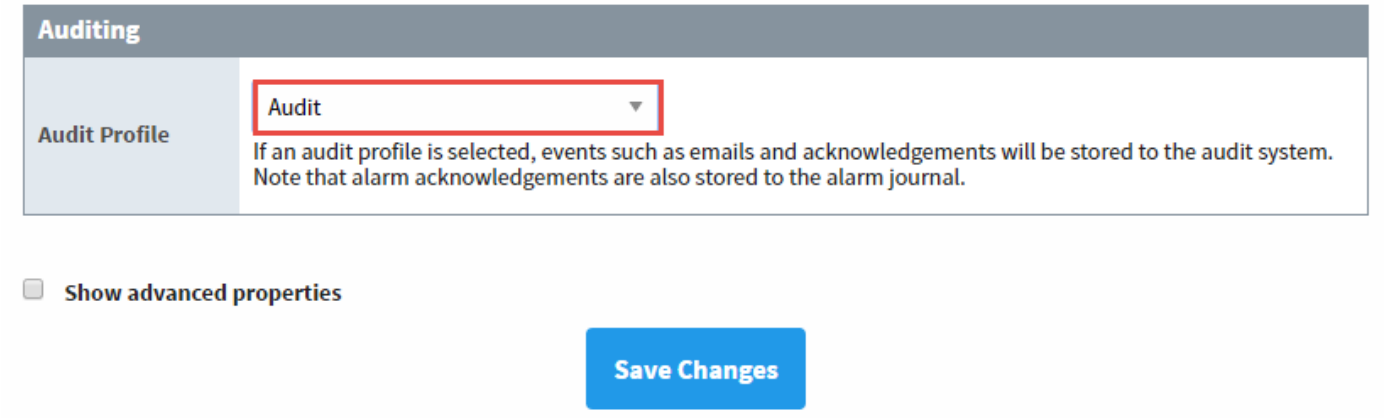
Once you have an **audit profile** created in Ignition, you can configure your **Alarm Notification Profile** to start using it.

1. Go to the **Config** section of the Gateway.
2. Choose **Alarming > Notification** from the menu on the left. The Alarm Notification Profiles page is displayed.
3. Edit the appropriate notification profile by clicking the **edit** link.



4. The Edit Alarm Notification Profile page is displayed. Scroll down to the **Auditing** section, and select an **Audit Profile** from the drop-down menu. Click **Save Changes**, and Ignition will

automatically begin storing information.



Now that the Alarm Notification Profile is storing data into the Audit system, you have a complete log of all alarm emails and acknowledgements that you can review.

See **Audit Log Display** for more info on how to retrieve information from the Audit Log.

Audit Log Displays

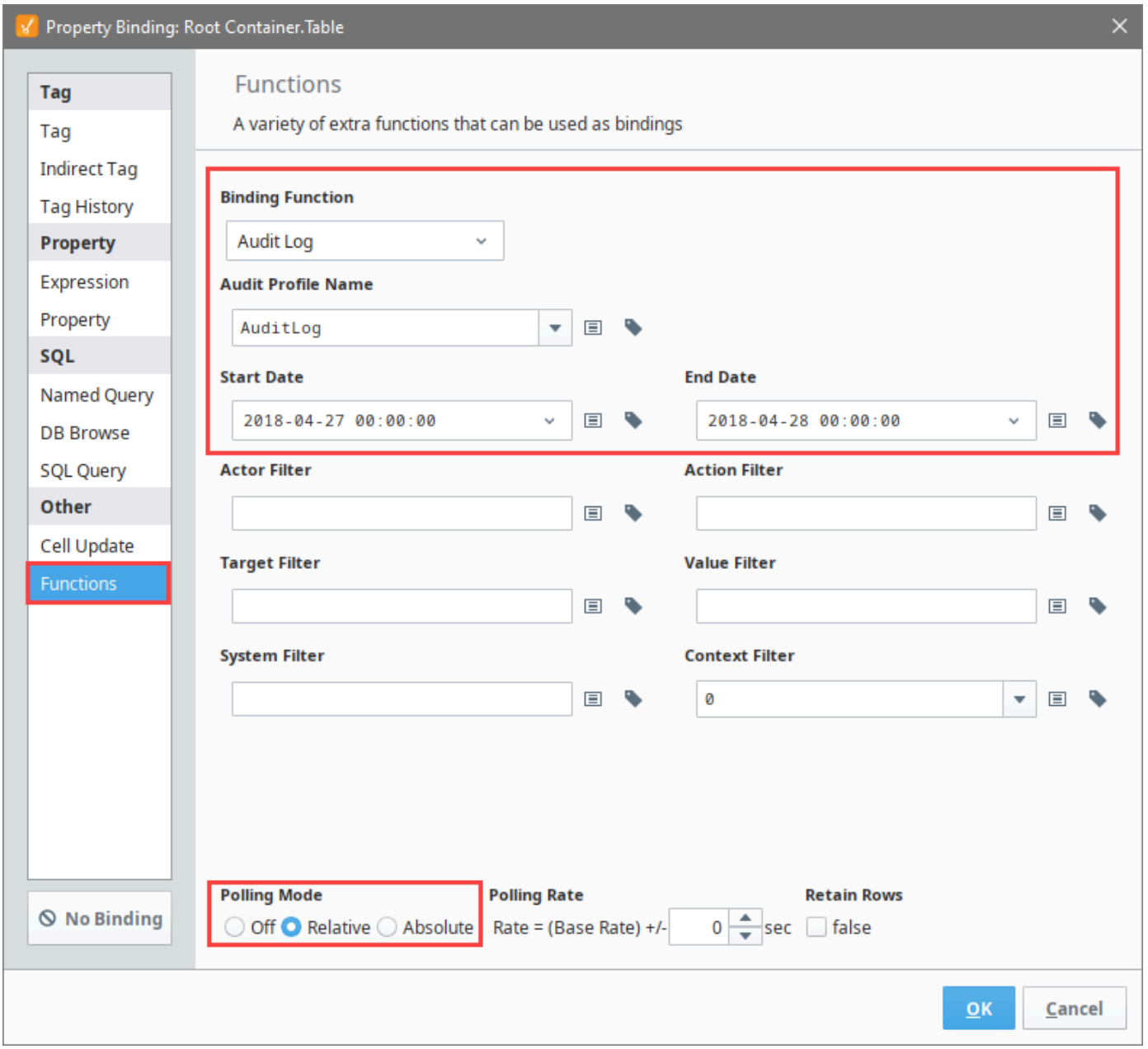
This page documents several ways to view results from the Audit Log system. For more information on how the Audit Log works, see the **Audit Log and Profiles** page.

Access the Audit Log with Table Functions

Vision

In Vision, you can easily access and display the Audit Log using table functions.

1. Ensure you have an **Audit Log set up and attached to a project**.
2. Open the Designer and navigate to your Vision project.
3. Drag a **Table** component onto a window.
4. In the **Property Editor**, click the binding icon for the table's **Data** property. The Property Binding window will appear.
5. Select the **Functions** binding type and choose **Audit Log** from the **Binding Function** drop-down menu.
6. Configure the binding options:
    - **Audit Profile Name**: Select your Audit Profile.
    - **Start Date**: Enter the appropriate start date.
    - **End Date**: Enter the appropriate end date.
    - **Polling Mode**: Select **Relative**.



7. Click **OK** to apply the binding. The Table will populate with the Audit Log data based on the selected parameters.

8. Use the **Table Customizer** to configure the Table's appearance by reorganizing and hiding columns, making columns sortable, and assigning meaningful headers.

Timestamp	Actor	Action	Action Target	Action Value	Result...	Result	System	Context ...	Context
Apr 27, 2018 11:39 AM	admin	login			0	AuditStatus...	project=Parameters	4	Client
Apr 27, 2018 3:48 PM	admin	logout			0	AuditStatus...	project=Parameters	4	Client
Apr 27, 2018 3:48 PM	admin	project save	Parameters		0	AuditStatus...	project=Parameters	2	Designer
Apr 27, 2018 3:49 PM	admin	project save	Parameters		0	AuditStatus...	project=Parameters	2	Designer
Apr 27, 2018 3:49 PM	admin	logout			0	AuditStatus...	project=Parameters	2	Designer
Apr 28, 2018 12:00 AM	Scheduled Report	report executed	WorkOrder Report	Failed Action -...	-2,147,4...	AuditStatus...	project=Parameters	1	Gateway
Apr 28, 2018 12:00 AM	Scheduled Report	report executed	Test Report	Failed Action -...	-2,147,4...	AuditStatus...	project=Parameters	1	Gateway
Apr 28, 2018 12:00 AM	Scheduled Report	report executed	Veggie Report 2		0	AuditStatus...	project=Parameters	1	Gateway
Apr 28, 2018 12:00 AM	Scheduled Report	report executed	Car Inventory Report	Failed Action -...	-2,147,4...	AuditStatus...	project=Parameters	1	Gateway

Perspective

Currently, Perspective does not have an automatic way to query the Audit Log. Instead, you can use a Named Query or the `system.util.queryAuditLog` function to display Audit Log data.

Using a Named Query

To display Audit Log data using a Named Query:

1. Create the Named Query:

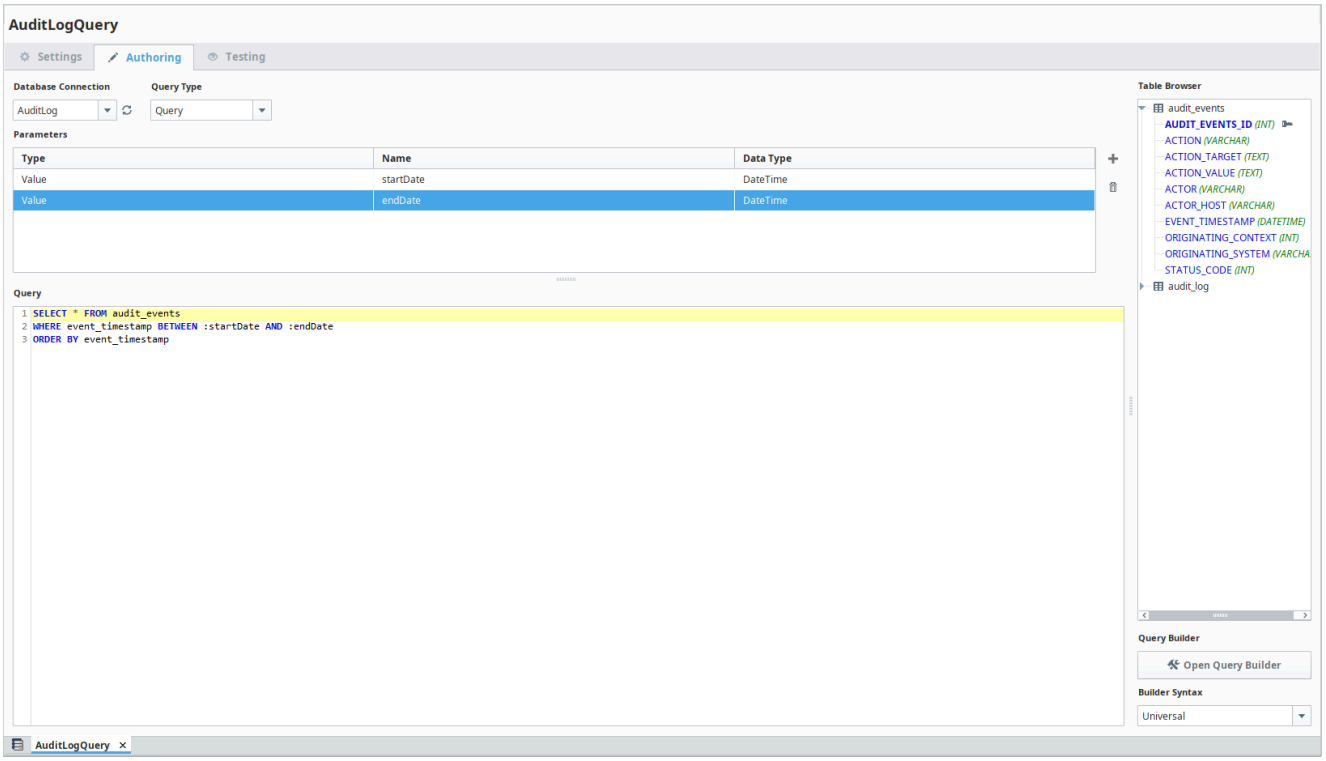
    i. In the Designer, navigate to **Project** > **Named Queries**.

    ii. Create a new query named `AuditLogQuery`.

    iii. Select the Database Connection that contains the Audit Log data and enter the following SQL query:

    ```
    SELECT * FROM audit_events
    WHERE event_timestamp BETWEEN :startDate AND :endDate
    ORDER BY event_timestamp DESC
    ```

    iv. Add parameters `startDate` and `endDate` with the Data Type **DateTime**.

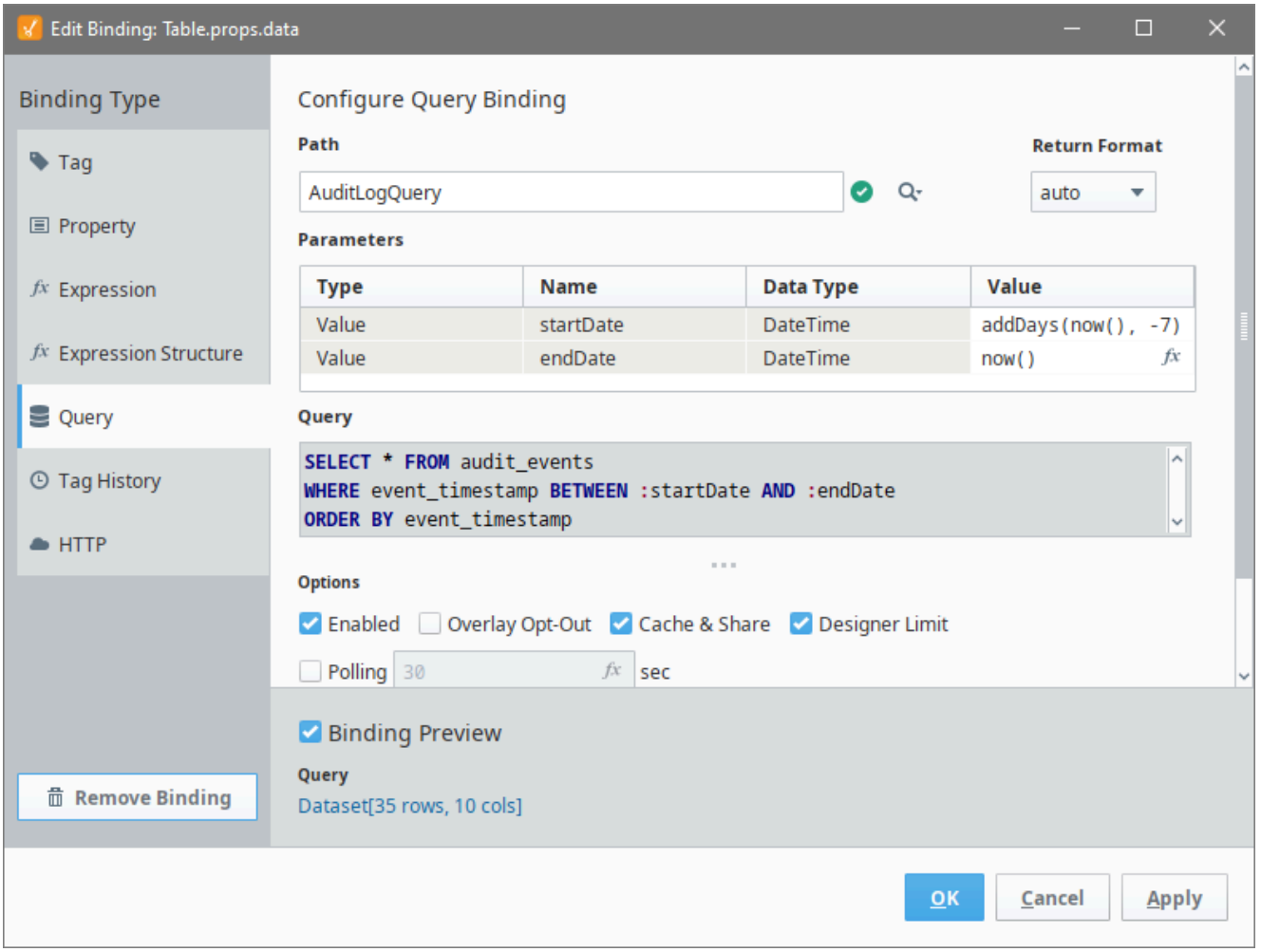
2. Bind the Named Query to a Table:

    i. In a Perspective View, drag a **Table** component onto the canvas.

ii. Select the Table component and go to the **Props** section.

iii. Bind the **data** property to the Named Query you created.

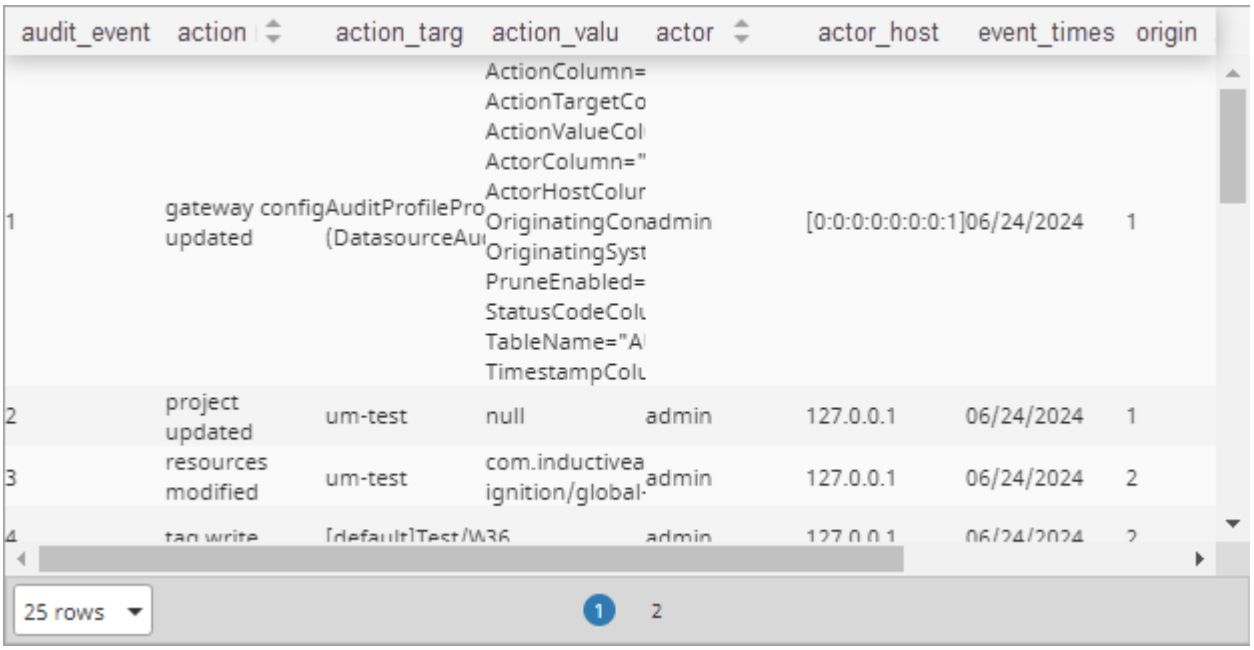
iv. Configure the parameters `startDate` and `endDate` to specify the date range for the query.



3. Customize the Table columns to display the data correctly. This may include formatting dates and adjusting column headers.

4. Save your changes and launch the Perspective session.

The Table component will now display the Audit Log data for the specified date range.



audit_event	action	action_targ	action_valu	actor	actor_host	event_times	origin
1	gateway config updated	AuditProfilePro (DatasourceAu	ActionColumn= ActionTargetCo ActionValueCol ActorColumn=" ActorHostColur OriginatingCon OriginatingSyst PruneEnabled= StatusCodeCol TableName="A TimestampCol	admin	[0:0:0:0:0:0:0:1]	06/24/2024	1
2	project updated	um-test	null	admin	127.0.0.1	06/24/2024	1
3	resources modified	um-test	com.inductivea ignition/global-	admin	127.0.0.1	06/24/2024	2
4	tag write	[default]Test/W36		admin	127.0.0.1	06/24/2024	2

25 rows 1 2

For more details, see **Named Queries**.

Using `system.util.queryAuditLog`

To display Audit Log data using the `system.util.queryAuditLog` function:

1. In the Designer, drag a **Table** component onto a new View. Name it **auditLogTable** for the script example below to work.
2. Right-click on the root container of the View and select **Configure Events...**.
3. In the editor, select the appropriate event (e.g., **onStartup**).
4. Click the ✚ and select **Script**.

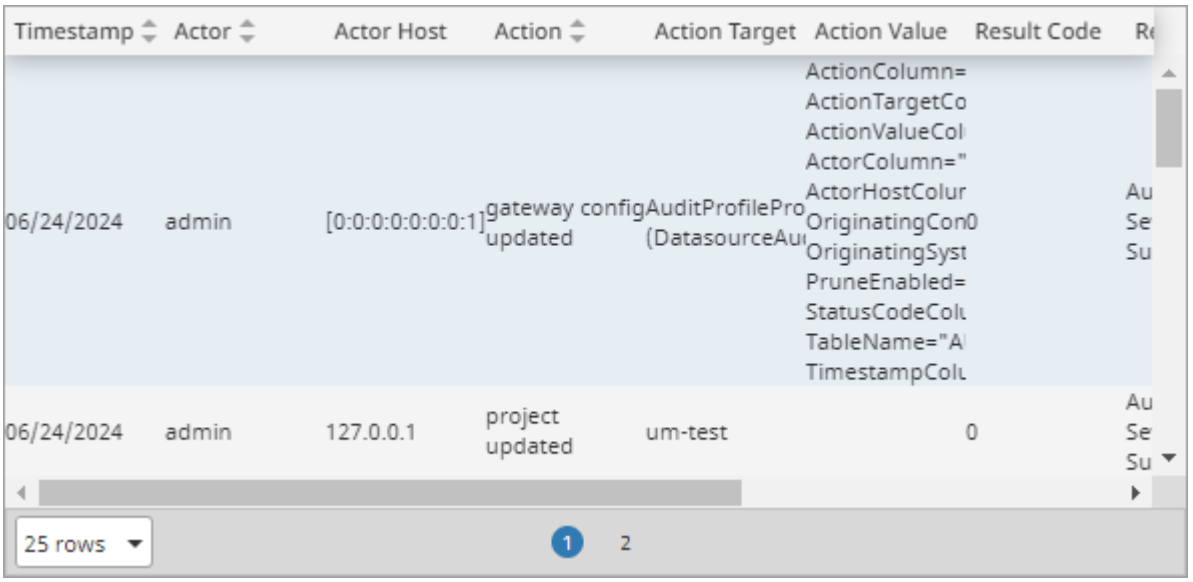
Example Script

```
# Define the start date and end date for the query
today = system.date.now()
startDate = system.date.addDays(today, -7)  # Example: last 7 days
endDate = today

# Query the Audit Log using the specified date range
results = system.util.queryAuditLog("AuditLog", startDate, endDate)

# Set the data property of the Table component with the results
self.getChild("auditLogTable").props.data = results
```

5. The Table component can take the raw output of the `system.util.queryAuditLog` function. However, you will need to configure the columns in the Table's **props** to display the data correctly. For instance, the date will show as seconds since the Unix epoch by default, so you may need to format it appropriately.

6. Save your changes in the Designer.

7. Open the Perspective session in a browser and you'll see the **auditLogTable** automatically populates with the Audit Log data on startup.



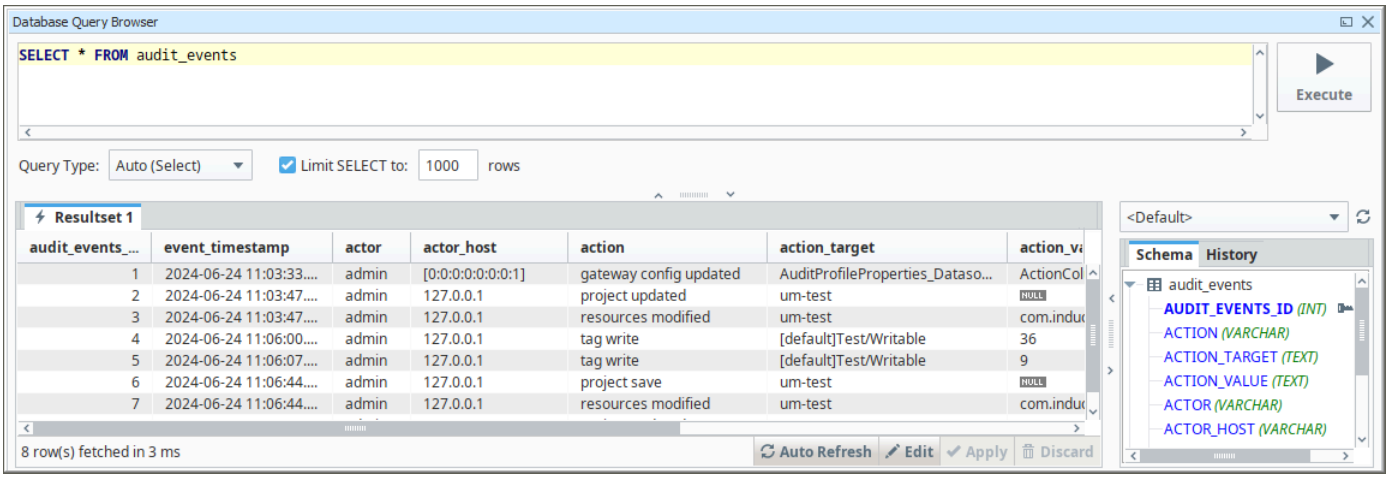
For more details, see **system.util.queryAuditLog**.

Access the Audit Log Using the Database Query Browser

The Database Query Browser allows you to search your database tables to view audit information.

1. In the Designer, go to the **Tools** menu and select **Database Query Browser**. The Database Query Browser window will open.

2. On the right side under **Schema**, double-click the **audit_events** table, then click **Execute**. The data from the audit_events table will appear in the Resultset tab.



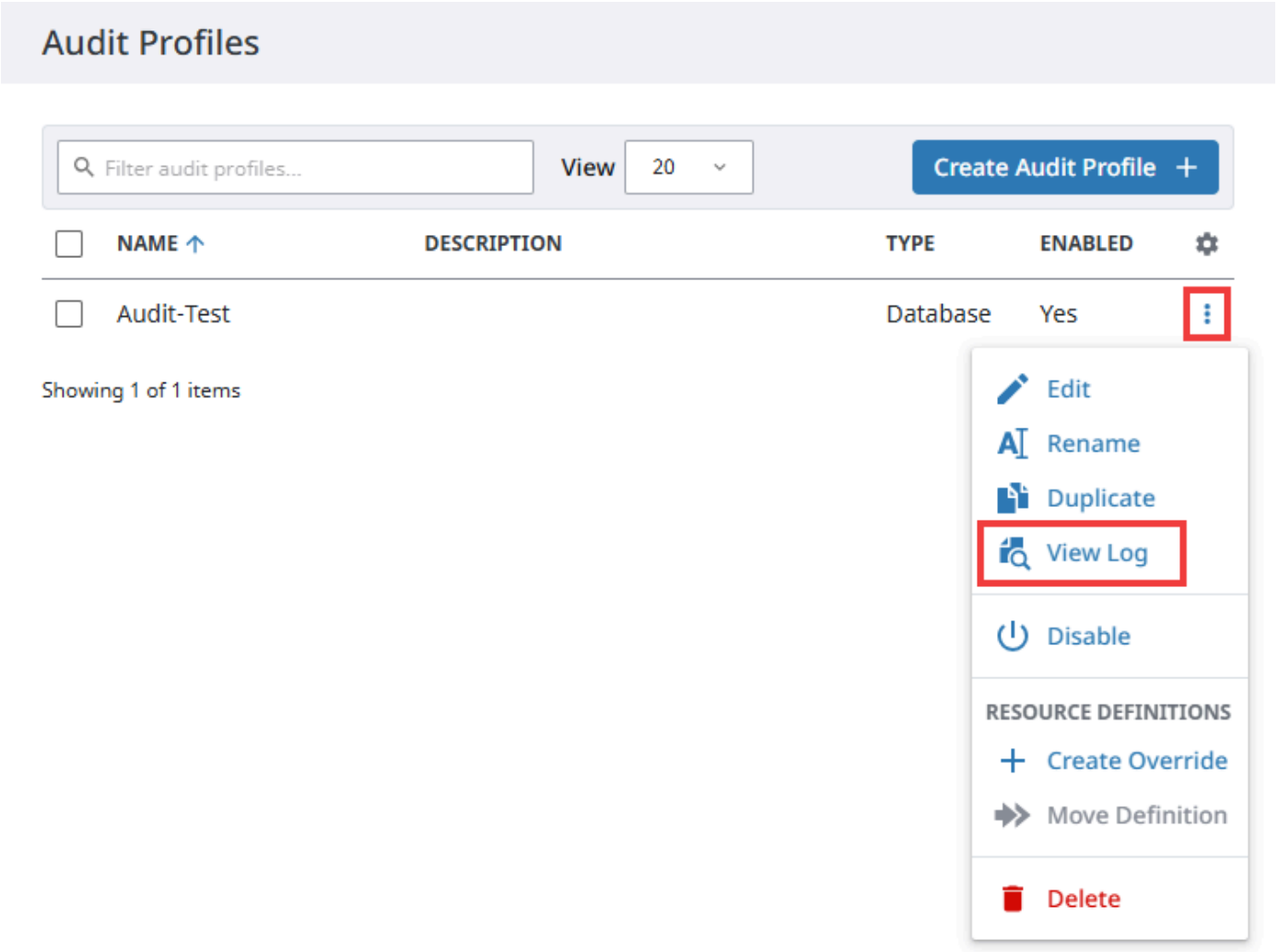
For more details, see **Database Query Browser**.

Access the Audit Log on the Gateway

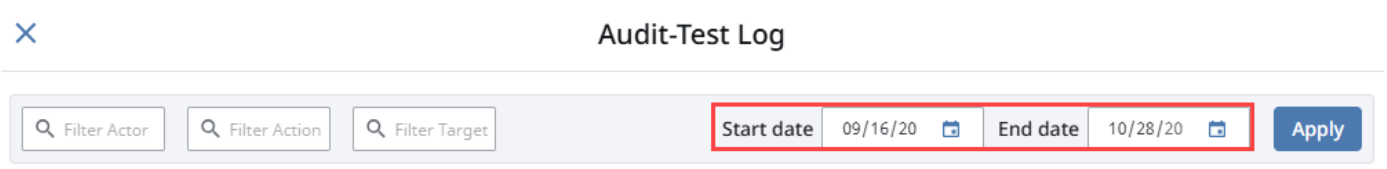
Ignition provides a simple interface to view Audit Logs directly on the Gateway:

1. On the Gateway Webpage, navigate to the **Platform** section and scroll down to **Security > Audit Profiles**.

2. The Audit Profile Logs page will display. Select the Audit Profile where your information is stored and click the three dot menu and select **View Log**.



3. Enter specific parameter text if you're looking for something particular, otherwise, enter a **Start Date** and **End Date**, then click **Apply**.



Auditing Actions Reference

The auditing system in Ignition records actions originating from the Gateway, Perspective, and Vision projects. This page describes the actions logged by the auditing system.

You can **view and search logged actions** in the Designer or on the Gateway. Note that if you're using the Action search feature on the Gateway, you must enter action terms exactly as they are shown in the Action column. Although exact terms are required, the search is not case sensitive. If you don't have a current reference for the action you want to search on your Audit Log page, the following list includes possible action terms:

List of Actions

- answer confirmed
- Authentication Lockout
- call acknowledged
- call ignored
- call completed
- call started
- email sent
- gateway config added
- gateway config updated
- gateway restore initiated
- gateway shutdown
- gateway startup
- IdP login attempt
- license unactivate
- license updated
- license activated
- login
- login attempt
- Login Request
- Login Response
- logout
- Logout Request

- Logout Response
- project created
- project deleted
- project save
- project updated
- query
- Reload Trusted JSON Web Keys
- Reload Trusted Certificates and Keys
- report executed
- resources modified
- SMS acknowledgement received
- SMS sent
- stored procedure called
- tag edit
- tag delete
- tag move
- tag rename
- tag write
- token login attempt
- trusted certificates deleted
- trusted certificates uploaded
- Web Auth Status Change

Gateway Audit Actions

The following actions are recorded in an Audit Log when the Gateway has a **Gateway Audit Profile** is configured.

Project System

The following project-based actions are tracked by the auditing system.

- Project Property changes made from the Designer.
- Project setting changes made from the Gateway's web interface.
- Creating or deleting a project.

- Saving a project.

Gateway Systems

In addition, project files on the Gateway's file system are closely monitored. If a user or third-party system modifies any of the project files, an entry will be recorded in the auditing system. The following Gateway-level actions are recorded in an Audit Log when the Gateway has a **Gateway Audit Profile** configured and selected on the Gateway's **Security General Settings** page.

Modules

- Installing modules on the Gateway.
- Disabling a module.
- Enabling a module.
- Deleting a module.

Gateway - General

- Gateway startup
- Gateway shutdown (assuming the Gateway was requested to shutdown: unintended shutdowns from power failures and such will not be recorded).
- Gateway Login
- Gateway Logout

Gateway - Restore

- Restoring the Gateway from a Gateway backup. Specifically, the Gateway will log that it was asked to before a restore, then perform the restoration.

Licensing Changes

- Activating a license.
- Unactivating a license.
- Updating a license.

Redundancy

- Saving after making any changes to the Redundancy Settings page.

Web Server Page

- Installing or removing a security certificate.
- Making changes to the Web Server Settings page.

Gateway Network

- Saving changes to Gateway Network General Settings .
- Creating, editing, or deleting outgoing connections.
- Approving incoming connections.

Email Settings

- Creating, editing, or deleting an SMTP Profile.

Audit Profile

- Creating, editing, or deleting an Audit Profile.

User Sources

- Creating, editing, or deleting a User Source.
- Creating, editing, or deleting a user.
- Creating, editing, or deleting a role.
- User Lockout Events will also be recorded. Note that the Audit Log will record only the initial lockout event, rather than each failed authentication attempt.

Service Security

- Editing and saving a policy.

Identity Providers

- Creating, editing, or deleting an Identity Provider configuration.
- Making changes to a User Attribute Mapping.
- Creating, editing, or deleting a User Grant.
- Saving changes on a Security Level Rule.
- Receiving new verification keys or certificates.

Security Levels

- Creating, editing, and deleting security zones

Security Zones

- Creating, editing, and deleting security zones

Database - Connections

- Creating, editing or deleting a database connection.

Database - Drivers

- Creating, editing or deleting a JDBC driver
- Creating, editing or deleting a Translator

Store and Forward

- Creating, editing or deleting a Store and Forward engine.

Alarming - General

- Saving changes on the Alarming General settings page.

Alarming - Alarm Journal

- Creating, editing or deleting an Alarm Journal Profile

Alarming - Notification

- Creating, editing, or deleting an Alarm Notification Profile.

Schedules

- Creating, editing or deleting a schedule.
- Creating, editing or deleting a holiday.

Tags - Realtime

- Creating, editing or deleting a Realtime Tag Provider

Tags - Historical

- Creating, editing or deleting a Historical Tag Provider.

OPC Client Connections

- Creating, editing or deleting an OPC connection.

OPC UA - Device Connections

- Creating, editing, or deleting a device connection (editing/saving a device connection configuration without making any changes will be recorded as an edit).
- Editing a Modbus address mapping via Gateway interface on Modbus device connections.
- Editing DNP3 Aliased Points via Gateway interface on DNP3 device connections.
- Editing tags via Gateway interface on Omron NJ device connections.
- Adding FINS tags via Gateway interface on Omron FIN device connections.

OPC UA - Server Settings

- Editing the OPC UA Settings page.

Enterprise Administration

- Configuring a Gateway to be either an Agent or Controller.

Enterprise Administration - Event Thresholds

- Changes made to Event Thresholds.

Enterprise Administration - Controller Settings

- Making changes to the Controller Settings page, including uninstalling the controller.

Enterprise Administration - Agent Settings

- Making changes to the Agent Settings page, including uninstalling the agent.

Enterprise Administration - Agent Management

- Creating, editing, deleting an Agent Group.

Enterprise Administration - License Management

- Adding or removing a license from the License Management page.

Enterprise Administration - Agent Tasks

- Creating, editing, or deleting an agent task
- Separate records are taken each time a task executes.

Sequential Function Charts

- Changes made to the SFC Settings page.

Add a Record Manually

- You can also add a record into the audit profile using the function **system.util.audit**.

Remote Gateway Tag Writes

- Actions on tags from remote servers are recorded in the Audit Log. The 'System' column shows the originating Gateway name. Note that 'Host', and 'Context' will appear unknown for these records, but auditing events will include the actor.

Perspective Auditing Actions

Perspective Sessions generate entries in an assigned audit profile. The following actions are recorded in the Audit Profile:

- Tag changes from a component binding.
- Authentication level changes (a user's security level changes).
- Login Request - Indicates a user is requesting to log into an Identity Provider (IdP). The user should have been redirected to the IdP with a login request and Ignition is awaiting the IdP's login response. Note that the user is not logged in until the IdP redirects the user back to Ignition with a login response and Ignition validates the login response.
- Login Response - Records when a login response is received from the IdP. It's possible that a login response will never be received for a login request. For example: if the user bails out of the login flow by closing their web browser before completing the login, Ignition will never receive the login response and will time out the request.
- Logout Request - Indicates a user is requesting to log out of an IdP. The user *may* be redirected to the IdP to log out of their IdP session. Regardless the user will be redirected back to the Perspective Session in a logged out state.
- Logout Response - Records when a logout response is received from the IdP after a user logged out of their IdP session. This event will not occur if the IdP does not support logout or if Ignition is not configured to redirect the user to the IdP for logging out.

- Tag changes from a Perspective script. Specifically:
  - Writes, such as, but not limited to, those from **system.tag.writeBlocking**
  - Edits and renaming, such as those caused by **system.tag.configure**
  - Deletions, such as those caused by **system.tag.deleteTags**
  - Moves, such as those caused by **system.tag.move**

Vision Auditing Actions

The Vision project needs an audit profile configured and auditing enabled. Vision Clients will then log records to an assigned audit profile. Here is a list of audit actions that will be tracked in the Ignition auditing system:

Tags

The following Tag related actions generate entries in the Audit Log. Note that the functions below must originate from the Tag Browser, the Designer's Scripting Console, or Vision component-based scripts.

- Tag Creation - Including tags created with the Tag Editor and the **system.tag.configure** function.
- Tag Deletion - Including those deleted from the Tag Browser's UI and the **system.tag.deleteTags** function.
- Tag Edits - Including edits made to tags from the Tag Editor and the **system.tag.configure** function.
- Moving Tags - Including moves made by drag-and-drop in the Tag Browser or by calling the **system.tag.move** function.
- Tag Renames - Renaming a tag generates an entry.

Vision Tag Writes

Write requests sent from a tag either through a standard Tag Binding, Indirect Tag Binding, or manual entry from the Tag Browser.

Vision Component Database Writes

The system explicitly captures modifications made to database tables through the following methods:

- **SQL Query Bindings** - modifications from the UPDATE Query will be recorded.
- **DB Browse Binding** - modifications made with the Enable Database Writeback area will be recorded.

Vision User Login/Logout

- Logging into a Vision Client will generate an entry in the auditing system, as will logging out of the client.
- Closing the client while logged in is treated as a logout. Note that the entry is only recorded if the client is aware that it is closing, which excludes cases where the client closed unexpectedly.

Database Query Browser

- If the project opened in the Designer has an assigned Audit Profile, then changes made to database tables using the database query browser are automatically recorded to the Audit Log. "Changes" in this case refer to UPDATE, DELETE, or INSERT statements manually typed and executed from the database query browser.
- Enabling edit mode and applying changes, including typing in new values, adding rows, removing rows, and clearing out fields, are recorded as queries called from the project.

Vision Scripting

The following functions generate entries in the Audit Log if called from Vision component-based scripts, or from the Designer's Scripting Console.

- **system.db.execSProcCall**
- **system.db.runPrepUpdate**
- **system.tag.writeBlocking**
- **system.tag.writeAsync**
- **system.report.executeAndDistribute**
- **system.report.executeReport**

Designer

Designer Login and Closing

- Opening a project in the Designer that has auditing enabled will also generate a login entry in the auditing system. Note that this occurs when the user opens the project, not when they log in using the Designer's login screen: auditing is project-based, so the user has to select a project that is being edited first.
- Closing the Designer effectively counts as logging off, and will generate a "logout" entry. Similar to vision, should the designer close unexpectedly, then an entry will not be recorded.

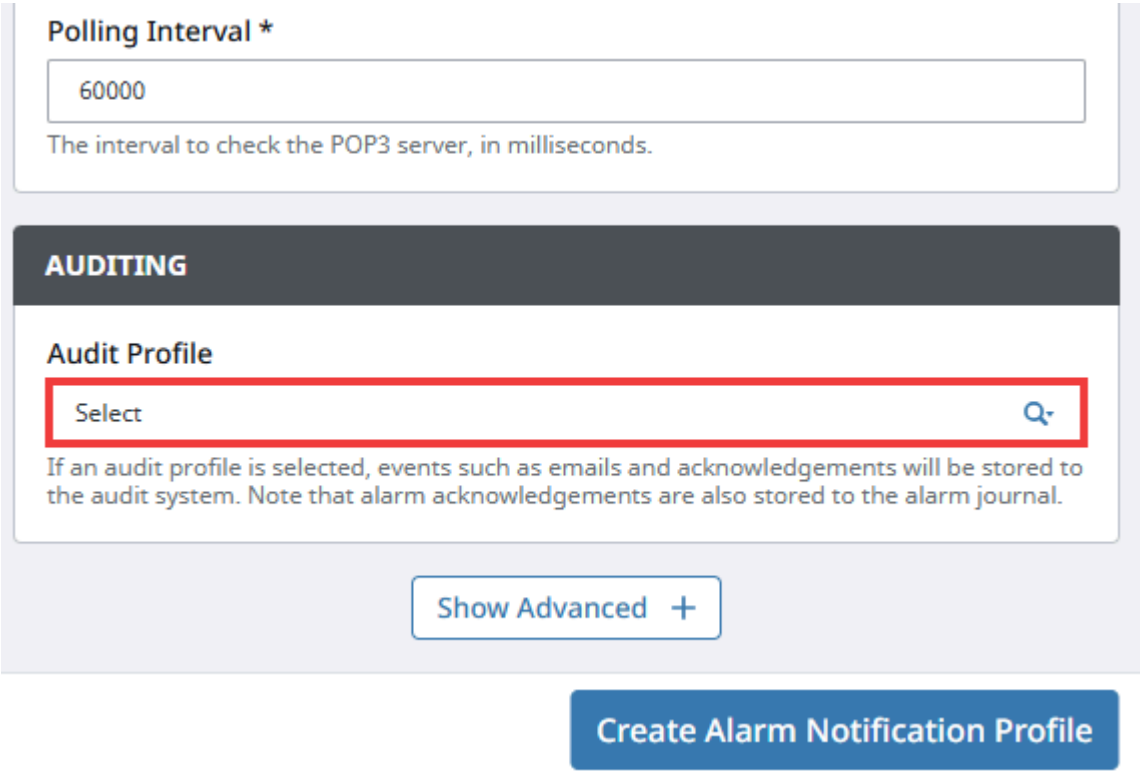
Database Query Browser

If the project opened in the Designer has an assigned Audit Profile, then changes made to database tables using the database query browser are automatically recorded to the Audit Log. "Changes" in this case refer to UPDATE, DELETE, or INSERT statements manually typed and executed from the database query browser.

Enabling edit mode and applying changes, including typing in new values, adding rows, removing rows, and clearing out fields, are recorded as queries called from the project.

Alarm Notification

Alarm notifications do not use the Gateway or project Audit Profiles. To audit alarm notification actions, you must create an individual Audit Profile, and then select the desired Audit Profile within the Alarm Notification Profile's configuration form.



Alarm Notification Attempts

Attempts to send out alarm notifications are recorded in the auditing system. Specifically, the Gateway will record when it attempted to send out a notification, as well as if the attempt failed (such as the SMTP server refusing the request). It is important to note that the auditing system can not report failures that occur outside of the Gateway. Thus, if a voice notification fails to send due to some error in the VOIP system, it's possible that the Gateway won't report the VOIP error, but the Audit Log will have an entry stating that the Gateway attempted to send the notification.

Reporting Module

Report Execution

Reporting module reports generate an entry in the auditing system when a report is executed. Thus:

- Reports running on a schedule will generate an entry.
  - These entries are recorded to the selected Audit Profile in the Auditing Settings of your Project Properties.
- Report schedules executed on demand will generate an entry.
  - Reports executed on demand using scripting functions, such as **system.report.executeReport** and **system.report.executeAndDistribute**, will have their audit entries recorded to the Audit Profile for the project used in the `project` parameter. The selected profile can be found under the Auditing Settings of the Project Properties.
- Navigating to a Vision window (in either the Designer or a Vision Client) will trigger a report execution, generating an entry in the auditing system.

Advanced Deployments

Several advanced deployment approaches are available for Ignition beyond those described within the **Installing and Upgrading** section. For instance, Ignition has an official Docker image available on **Docker Hub** that follows the Open Container Initiative (OCI) Runtime and Image specifications. Additionally, users looking to run large-scale deployments can take advantage of the official Helm charts for Kubernetes orchestration.

The following pages cover best practices for deploying Ignition with these third party platforms. If you want information about setting up dev, staging, or production environments for your Gateway, refer to the **Gateway Deployment Modes** page.

An important aspect to be aware of for these deployment methods, regardless of selection, is that they depend on **leased licensing** for use. Leased licenses, also called eight-character licenses, differ from a standard six-character license as they require an activation token and must frequently check in with an activation service to remain valid. Regular monitoring and thorough understanding of your leased license status is necessary to avoid Gateway activation issues.

Docker Image

Kubernetes

Helm Chart for Kubernetes Deployments

Docker Image

The Ignition Platform offers a container image for use on popular container runtimes such as Docker. You can find the image on Docker Hub at **https://hub.docker.com/r/inductiveautomation/ignition**. Our official image is a Linux container, based on **Ubuntu Linux**.

Ignition's Docker Image build is multi-architecture and supports linux/amd64, linux/arm64, and linux/arm/v7. Users can `docker pull inductiveautomation/ignition` and get an image that automatically matches their system architecture.

Getting Started

To get started with Ignition running as a container, you'll need a container runtime. **Docker Desktop**, available for Windows and macOS, includes both the **Docker Engine** container runtime as well as container tooling and supporting tools such as **Docker Compose**. If you are running Linux, you'll need to download and install the container runtime and tools such as Docker Compose individually. Guidance for the installation and configuration of Docker (on all platforms) are available here: **Get Docker**

Configuration

Various aspects of the Ignition Gateway can be configured through the use of either command-line arguments to the container or environment variables. This section will detail the available configuration options and how best to use them.

Environment Variables

The available environment variables can either accept explicit values or load the target values from a file. Specifying any of the environment variables below with a `_FILE` suffix will direct the system to take that environment variable value as a path to a file containing the value of interest. This is useful for secrets management systems when you want to keep the values out of the process environment, where individuals with access to `docker inspect <container>` might be able to see the values.

> **NOTE**
>
> Some of the variable descriptions below make use of a `#` character. In these cases, you can specify the same variables multiple times by replacing the `#` character with a new number for

each use.

```
GATEWAY_NETWORK_0_HOST=1.2.3.4
GATEWAY_NETWORK_0_PORT=8088
GATEWAY_NETWORK_0_DESCRIPTION="For tag server"

GATEWAY_NETWORK_1_HOST=5.6.7.8
GATEWAY_NETWORK_1_PORT=8089
GATEWAY_NETWORK_1_DESCRIPTION="For edge server"
```

The table below describes the environment variables exclusive to Docker. For a full list of environment variables supported by the Ignition platform, see the **Platform Environment Variables** reference page.

Environment Variable	Description
`GATEWAY_RESTORE_DISABLED`	Set to `true` to have the restored gwbk disabled on startup.
`IGNITION_UID`	Numeric user ID for the target user. Passing this variable and `IGNITION_GID` allows Ignition to run within a container as an `ignition` user rather than as the `root` user. When set, the entrypoint will automatically update file ownerships for the Ignition installation on startup to match the target user prior to stepping down from the `root` user to launch the Gateway.
`IGNITION_GID`	Numeric group ID for the target user. Passing this variable and `IGNITION_UID` allows Ignition to run within a container as an `ignition` user rather than as the `root` user. When set, the entrypoint will automatically update file ownerships for the Ignition installation on startup to match the target user prior to stepping down from the root user to launch the Gateway.

Module Identifiers Table

Module identifiers are used by Ignition to search and find modules. The module identifiers listed below represent the modules that are built-in with the container image. These modules are saved in the `/usr/local/bin/ignition/user-lib/modules` folder. Any third party modules installed through the Gateway will be saved in the `/usr/local/bin/ignition/data/local/modl` folder. This third party folder location is defined by the **ignition.gateway.externalModulesFolder** system property.

LEGACY MODULE IDENTIFIERS

Ignition will still include the **module identifiers listed in version 8.1** when attempting a search, but it is recommended to use the full module identifiers listed in the table below.

Module Identifier	Module Filename
com.inductiveautomation.alarm-notification	Alarm Notification-module.modl
com.inductiveautomation.opcua.drivers.ablegacy	Allen-Bradley Drivers-module.modl
com.inductiveautomation.opcua.drivers.bacnet	BACnet Driver-module.modl
com.inductiveautomation.opcua.drivers.dnp3	DNP3-Driver.modl
com.inductiveautomation.opcua.drivers.dnp3v2	DNP3-Driver-v2.modl
com.inductiveautomation.eam	Enterprise Administration-module.modl
com.inductiveautomation.eventstream	EventStream-module.modl
com.inductiveautomation.historian	Historian-module.modl
com.inductiveautomation.opcua.drivers.iec61850	IEC 61850 Driver-module.modl
com.inductiveautomation.connectors.kafka	Kafka Connector-module.modl
com.inductiveautomation.opcua.drivers.logix	Logix Driver-module.modl
com.inductiveautomation.opcua.drivers.micro800	Micro800 Driver-module.modl
com.inductiveautomation.opcua.drivers.mitsubishi	Mitsubishi-Driver.modl
com.inductiveautomation.opcua.drivers.modbus	Modbus Driver v2-module.modl
com.inductiveautomation.connectors.mongodb	MongoDB Connector-module.modl
com.inductiveautomation.opcua.drivers.omron	Omron-Driver.modl
com.inductiveautomation.opcua	OPC-UA-module.modl
com.inductiveautomation.perspective	Perspective-module.modl
com.inductiveautomation.reporting	Reporting-module.modl

Module Identifier	Module Filename
com.inductiveautomation.sfc	SFC-module.modl
com.inductiveautomation.opcua.drivers.siemens	Siemens Drivers-module.modl
com.inductiveautomation.opcua.drivers.siemens-symbolic	Siemens Enhanced Driver-module.modl
com.inductiveautomation.sms-notification	SMS Notification-module.modl
com.inductiveautomation.sqlbridge	SQL Bridge-module.modl
com.inductiveautomation.historian.sql	SQL Historian-module.modl
com.inductiveautomation.symbol-factory	Symbol Factory-module.modl
com.inductiveautomation.opcua.drivers.tcpudp	UDP and TCP Drivers-module.modl
com.inductiveautomation.vision	Vision-module.modl
com.inductiveautomation.phone-notification	Voice Notification-module.modl
com.inductiveautomation.webdev	Web Developer Module.modl
com.inductiveautomation.jdbc.postgresql	PostgreSQL JDBC Driver Module.modl
com.inductiveautomation.jdbc.mariadb	MariaDB JDBC Driver Module.modl
com.inductiveautomation.jdbc.mssql	MSSQL JDBC Driver Module.modl

If you are using Solution Suites, the corresponding identifiers are listed below.

Suite Name	Suite Identifier
Application Building Suite	com.inductiveautomation.suite.application
Industrial Historian Suite	com.inductiveautomation.suite.historian
DataOps Suite	com.inductiveautomation.suite.dataops
Enterprise Integration Suite	com.inductiveautomation.suite.enterprise
Alarm Management Suite	com.inductiveautomation.suite.alarms

Runtime Arguments

The following options and flags are accepted by the container as runtime arguments. Options should be followed by a value, while flags are stand-alone. Note that these arguments are passed after the image name in a Docker Run statement.

Options

Option	Arguments	Descriptions
`-n`	String	Gateway name to be applied to the Ignition Gateway on startup. **Note**: Do not use the `-n` rename flag with a backup Gateway. The backup Gateway will take the name of the master Gateway, and this flag may override or conflict with the set backup node name, leading to mismatched names.
`-a`	String	Public web address*
`-h`	Integer	Public HTTP port*
`-s`	Integer	Public HTTPS port*
`-m`	Integer	Max memory for JVM
`-r`	String	Path to Gateway Backup for Automated Restore
`--restore-local`	String	**New in 8.3.9** Alternative option to using `-r` that additionally restores the local-only configuration files within the .gwbk file. Use this option to restore files intended for use on a specific Gateway, such as hostname-specific certificates. Note that if an 8.1 .gwbk file is supplied, the restore local functionality is ignored.

* *Specify Public address/HTTP/HTTPS ports together, all three must be specified*

Flags

Flag	Description
`-d`	Debug mode, applicable to development and attaches port 8000 for remote JVM debugging. Note that port 8000 may need to be published on the container.

Supplemental JVM and Wrapper Arguments

You can also specify JVM and Java Service Wrapper arguments to your Docker containers runtime arguments by adding a double-hyphen to delimit these arguments from the other runtime arguments listed above. Wrapper arguments (starting with `wrapper.*`) will be merged and JVM arguments will be added to those specified in the `ignition.conf` file. Refer to **Customizing JVM/Wrapper/Gateway Arguments on Container Launch** for an example.

> **LEASED LICENSING TERMINATION SETTINGS**
>
> A couple parameters to be aware of are the **Leased Licensing Parameters**. Since leased activation license session termination on shutdown is necessary to ensure the session is returned to the activation server, these allow you to modify your session termination settings.

Gateway Arguments

The image also supports the use of Gateway arguments, allowing you to modify the gateway.xml file when launching a container. Like JVM and wrapper arguments, Gateway arguments must be specified after a double-hyphen. Only entries in the file that follow the pattern of `gateway.#` can be modified in this way.

Logging Settings

Unlike the non-containerized version of Ignition, the Ignition container image is designed to emit logs to stdout (“standard out”) so they can be leveraged by various logging drivers in the container engine. This is done by directing what would normally be stored in the `logs/wrapper.log` file to stdout. As a result, Ignition itself doesn’t control rotating the wrapper.log based on size, which can result in container logs growing unbounded. You can configure the default logging driver to constrain maximum log sizes. Reference the **Docker logging configuration** page to specify logger size limits.

Licensing

The Inductive Automation software contained in this Docker container image is licensed under the terms of the **Inductive Automation Software License Agreement**.

By passing the environment variable value `ACCEPT_IGNITION_EULA=Y`, you agree that your use of Inductive Automation software running in this Docker container image is governed by the terms of the **Inductive Automation Software License Agreement**.

All third party software (whether open or closed) running in this Docker container image is licensed as indicated in this Docker Hub “License” section or in the container itself, usually as a `NOTICE` or `LICENSE` file. You agree to comply with any relevant licenses for third party software contained in this Docker container image.

Examples

Refer to the **Docker Image Examples** page to learn more about how to run and configure the Ignition Docker image, as well as some standard best practices.

Docker Image Examples

This page contains example run configurations for the Ignition Docker image to demonstrate best practices for specified scenarios.

Deploying an Ephemeral Gateway for Testing

The run statement below will launch a container in detached mode `-d`, publishing port `9088` from the host to port `8088` in the container. The container will be named `ignition-test` and use the `8.3.0` image. Finally, the runtime arguments provided will set the Gateway name `-n` to `docker-test` with public address of `localhost` on HTTP port `9088` and HTTPS port `9043`.

```
docker run -d -p 9088:8088 --name ignition-test inductiveautomation/ignition:8.3.0
    -n docker-test -a localhost -h 9088 -s 9043
```

> **MULTI-LINE COMMANDS**
>
> For Linux/macOS/WSL, you can use the backslash with multi-line commands for better readability. You can substitute the backtick ` for Powershell and caret ^ for Windows Command prompt, these characters technically "escape" the newline character, so make sure they are the last character on a given line.

Preserving Gateway State

When using containers, it is common for stateful applications (such as the Ignition Gateway) to leverage volumes to persist that state across image updates, container remove/create cycles, etc. This can be done by applying a named volume to the `/usr/local/bin/ignition/data` path within the container. This is especially important if you need to make a change to your container configuration. Since container configurations are immutable, the only way to change it is to stop/remove the old container and start a new one (with a new configuration).

The sample run statement will be similar to the one above, but this time with a named volume gw-data attached to `/usr/local/bin/ignition/data` within the container. If the volume doesn't already exist, it will be created automatically. Note the additional `--pull` option to make sure that the `latest` tag is always pulled before running. Without this, the `latest` tag is only pulled if one isn't already present on your system.

```
docker run -d -p 9088:8088 --name ignition-test \
    -v gw-data:/usr/local/bin/ignition/data \
    --pull always inductiveautomation/ignition:latest \
    -n docker-test -a localhost -h 9088 -s 9043
```

By using named volumes, you're now able to stop and remove the `ignition-test` container and create a new one with a different configuration. As long as you attach your volume, your Gateway will start up with all of your tags and projects in their previous state.

Automating the Restore of a Gateway Backup

You can automate the restore of a Gateway backup on first-launch of your Gateway container. This allows for having a new Ignition Gateway restore to a known initial state automatically, without waiting for the commissioning steps. To leverage this feature, bind-mount a Gateway backup into the container and then use the `-r` runtime argument to specify the location and command the restore. Additionally, supply the `ACCEPT_IGNITION_EULA=Y` environment variable to accept the Ignition EULA (see the **Licensing** section) and bypass that Gateway commissioning step.

```
docker run -d -p 9088:8088 --name ignition-test \
    -v gw-data:/usr/local/bin/ignition/data \
    -v /path/to/gateway.gwbk:/restore.gwbk \
    -e ACCEPT_IGNITION_EULA=Y \
    inductiveautomation/ignition:8.3.0 \
    -n docker-test -a localhost -h 9088 -s 9043 \
    -r /restore.gwbk
```

> **WHAT IF I RESTART THE CONTAINER**
>
> The Gateway restore will only apply on a fresh Gateway launch. Subsequent restarts of the container will not restore the indicated Gateway backup.

Automate the Commissioning of a Fresh Gateway

You can automate the commissioning steps that normally require manual user interaction on the very first launch of the Ignition Gateway. By supplying specific **environment variables** , you can seed the commissioning steps with the required information to start the rest of the Gateway automatically.

```
docker run -d -p 9088:8088 --name ignition-test \
    -e ACCEPT_IGNITION_EULA=Y \
    -e GATEWAY_ADMIN_PASSWORD=password \
    -e IGNITION_EDITION=standard\
    inductiveautomation/ignition:8.3.0 \
    -n docker-test
```

READING ENVIRONMENT VARIABLES FROM A FILE

You can also specify environment variables in the form of `<env var>_FILE=/path/to/file` in order to have the environment variable value resolved by reading it from a file. This is helpful for secrets management systems within container orchestrators, such as Docker Swarm and Kubernetes.

Customizing JVM/Wrapper/Gateway Arguments on Container Launch

This example shows how to leverage the supplemental JVM/wrapper args feature.

```
docker run -d -p 9088:8088 --name args-test \
    -e ACCEPT_IGNITION_EULA=Y \
    -e GATEWAY_ADMIN_PASSWORD=changeme \
    -e IGNITION_EDITION=standard\
    inductiveautomation/ignition:8.3.0 \
    -n args-test \
    -- wrapper.java.initmemory=256 -Dignition.allowunsignedmodules=true \
    -Dignition.license.leased-activation-terminate-sessions-on-shutdown=true \
    -XX:MaxGCPauseMillis=200
```

The following demonstrates how to enable the Gateway's Resolve Host Names and Use Proxy Forwarded Headers settings by modifying the entries in the `gateway.xml`:

```
docker run -d -p 9088:8088 --name ignition-test inductiveautomation/ignition:8.3.0
-n docker-test -a localhost -h 9088 -s 9043 \
   -- gateway.resolveHostNames=true gateway.useProxyForwardedHeader=true
```

Using Docker Compose

A common practice is to leverage **Docker Compose** to start your container and bundle the configuration with other services, such as databases and MQTT brokers. Below is an example `docker-compose.yml` file that you can create inside of a new, empty folder. The example below incorporates many of the configurability features mentioned in the various sections above.

Compose Example

```yaml
# Docker Compose Example for inductiveautomation/ignition
# Compose Spec: https://github.com/compose-spec/compose-spec/blob/master/spec.md
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    ports:
      - 9088:8088
      - 9043:8043
    volumes:
      - gw-data:/usr/local/bin/ignition/data
    # env_file: ignition.env  # optionally specify variables in a file, or using `environment:` below
    environment:
      - ACCEPT_IGNITION_EULA=Y
      - GATEWAY_ADMIN_USERNAME=admin
      - GATEWAY_ADMIN_PASSWORD_FILE=/run/secrets/gateway-admin-password
      - IGNITION_EDITION=standard
      - TZ=America/Chicago  # see https://en.wikipedia.org/wiki/List_of_tz_database_time_zones#List
    secrets:
      - gateway-admin-password
    command: >
      -n docker-test
      -m 1024
      --
      wrapper.java.initmemory=512
      -Dignition.allowunsignedmodules=true

secrets:
  gateway-admin-password:
    file: secrets/GATEWAY_ADMIN_PASSWORD

volumes:
  gw-data:
```

Once defined, you can bring up the solution using `docker compose` commands.

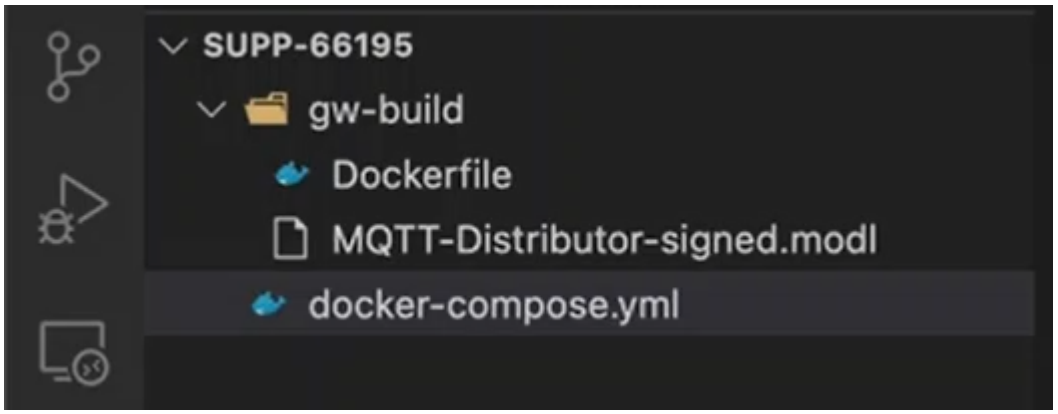
Integrate Third Party Modules After an Upgrade

Third party modules installed through the web interface are automatically routed to the **external modules folder** within the data volume. This means no special consideration is required for these modules when upgrading Ignition to a newer image version.

However, if you wish to integrate third party modules such that they cannot be uninstalled by the user, use the following example of how to build a derived image with your modules co-located with the built-in modules under `/usr/local/bin/ignition/user-lib/modules`.

> **NOTE**
>
> If a module is added to the built-in modules location through this method with the same module identifier, that module will route to the external modules folder and effectively "shadow" the built-in module file. For instance, instead of the built-in module loading, the one in the external modules folder with the same module identifier will take precedence.

To start, a module subfolder with the modules we are attempting to integrate will be labeled gw-build. The subfolder will also include a Dockerfile. The `docker-compose.yml` file needed for this process is located outside of this subfolder. We will be using a `docker-compose.yml` file similar to the previous example, but instead of using an image tag, we will be using a build tag.



docker-compose.yml

```yaml
services:
  gateway:
    build:
      context: gw-build
      ## Specify the upstream version to derive from for the build argument in the Docker file.
      args:
        IGNITION_VERSION: 8.3.0
    ports:
      - 8088:8088
    volumes:
      - gateway-data:/usr/local/bin/ignition/data
    environment:
      ACCEPT_IGNITION_EULA: "Y"
      GATEWAY_ADMIN_PASSWORD: password
      IGNITION_EDITION: standard
    command: >
      -n Ignition-supp-66195

volumes:
  gateway-data:
```

The Dockerfile within the gw-build folder will define the derived image by containing a build argument for the Ignition version. Notice this does not give a default version. This is because the version we want to build is specified and passed through our docker-compose definition. Next, we will add a command to copy the module files from this folder to the user-lib module folder inside the new Ignition image.

Dockerfile

```
ARG IGNITION_VERSION
FROM inductiveautomation/ignition:${IGNITION_VERSION}

COPY *.modl /usr/local/bin/ignition/user-lib/modules/
```

With these files defined, we can now run `docker compose build` to pull the image, perform all derived image build steps, and create the new image.

> **NOTE**
>
> Since `docker compose build` needs to be run each time a new module is added, you can alternatively add `pull_policy: build` in the yml file to automatically build each time.

User Migration - Docker Compose

The following example uses Docker Compose to demonstrate how to migrate the container user and file ownership from the `root` user to a standard, non-elevated `ignition` user. If you are instead using the command line, see the **next example** for migrating users. Your environment may vary, depending on how your container is set up.

User Migration Example (Docker Compose)

```
# User Migration Example (Docker Compose)
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    volumes:
      - ignition-data:/usr/local/bin/ignition/data
    ports:
      - 8088:8088
    command: >
      -n docker-test

volumes:
  ignition-data:
```

1. Upgrade the container to the latest version by changing the image tag to the version you want to upgrade to:

User Migration Example (Docker Compose)

```
# User Migration Example (Docker Compose)
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    volumes:
      - ignition-data:/usr/local/bin/ignition/data
    ports:
      - 8088:8088
    command: >
      -n docker-test

volumes:
  ignition-data:
```

2. Add the `IGNITION_UID=2003` and `IGNITION_GID=2003` environment variables to update the container permissions:

User Migration Example (Docker Compose)

```
# User Migration Example (Docker Compose)
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    volumes:
      - ignition-data:/usr/local/bin/ignition/data
    ports:
      - 8088:8088
    environment:
      - IGNITION_UID=2003
      - IGNITION_GID=2003
    command: >
      -n docker-test

volumes:
  ignition-data:
```

3. Declare the root user (`user: "0:0"`):

User Migration Example (Docker Compose)

```yaml
# User Migration Example (Docker Compose)
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    volumes:
      - ignition-data:/usr/local/bin/ignition/data
    ports:
      - 8088:8088
    environment:
      - IGNITION_UID=2003
      - IGNITION_GID=2003
    user: "0:0"
    command: >
      -n docker-test

volumes:
  ignition-data:
```

4. Save the Docker Compose file, then run `docker compose up -d`.

> **NOTE**
>
> By default, we are using the `-d` option in our Docker command. While you can also run the command without `-d`, the option runs the container in detached mode, allowing the container to run in the background even after closing the terminal.

5. The container will now use the `root` user when it starts up, but will launch Ignition as a standard `ignition` user. All of the files in the ignition-data volume have the proper ownership. We can now remove the `IGNITION_UID` and `IGNITION_GID` environment variables and the `user:"0:0"` override:

User Migration Example (Docker Compose)

```yaml
# User Migration Example (Docker Compose)
---
services:
  # Ignition Gateway
  gateway:
    image: inductiveautomation/ignition:8.3.0
    volumes:
      - ignition-data:/usr/local/bin/ignition/data
    ports:
      - 8088:8088
    command: >
      -n docker-test

volumes:
  ignition-data:
```

6. Finally, we can rerun the `docker compose up -d` command to recreate the container. Since we removed the environment variables and updated container permissions, the container will launch with the default user being a standard `ignition` user.
7. To check if the user successfully migrated over, verify the process listing and who the user is for each process using the command `docker compose exec gateway ps aux` in the terminal. Additionally, you can use the command `docker compose exec gateway whoami` to see who the default user for the container is:

```
$ docker compose exec gateway ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
ignition     1  0.0  0.0   2500   584 ?        Ss   21:25   0:00 tini -g -- ./ig
ignition    31  0.0  0.0  16348  4084 ?        Sl   21:25   0:00 ./ignition-gate
ignition    49  0.8  1.0 5907440 176192 ?      Sl   21:25   0:07 /usr/local/bin/
ignition  1312  0.0  0.0   8896  3252 pts/0    Rs+  21:40   0:00 ps aux

$ docker compose exec gateway whoami
ignition
```

User Migration - Command Line

The following example uses the command line to demonstrate how to migrate the container user and file ownership from the `root` user to a standard, non-elevated `ignition` user. If you are instead using Docker Compose, see the **preceding example**. Your environment may vary, depending on how your container is set up. The following container is called `example-gw` and is running Ignition version 8.3.0:

User Migration Example (Command Line)

```
docker run --name example-gw -v example-gw-data:/usr/local/bin/ignition/data \
    -e IGNITION_EDITION=standard -e ACCEPT_IGNITION_EULA=Y -e
GATEWAY_ADMIN_PASSWORD=password \
    -p 8088:8088 \
    inductiveautomation/ignition:8.3.0 \
    -n example-gw
```

1. Stop and remove your Docker container using the following commands. Keep in mind that you will need to replace `example-gw` with your container name:

   User Migration Example (Command Line)

   ```
   # Stop and remove container
   docker stop example-gw
   docker rm example-gw
   ```

2. Modify the container run configuration to declare the root user using `--user 0:0` (The UID and GID will both be 0):

   User Migration Example (Command Line)

   ```
   # Declare the root user
   docker run --name example-gw --rm -v example-gw-
   data:/usr/local/bin/ignition/data \
       -p 8088:8088 \
       --user 0:0 \
       inductiveautomation/ignition:8.3.0 \
       -n example-gw
   ```

3. Update the container permissions by adding the `IGNITION_UID` and `IGNITION_GID` environment variables. You can do this by adding `-e IGNITION_UID=2003 -e IGNITION_GID=2003` after `--user 0:0`:

   User Migration Example (Command Line)

   ```
   ```

```
# Update the container permissions
docker run --name example-gw --rm -v example-gw-
data:/usr/local/bin/ignition/data \
    -p 8088:8088 \
    --user 0:0 -e IGNITION_UID=2003 -e IGNITION_GID=2003 \
    inductiveautomation/ignition:8.3.0 \
    -n example-gw
```

4. The container's Gateway should now be running under a standard `ignition` user, using a UID of 2003 and a GID of 2003. You can run the following command to count the number of processes running as a standard `ignition` user:

User Migration Example (Command Line)

```
# Count the number of running processes under the ignition user
docker exec example-gw pgrep -u 2003 -c
```

> **NOTE**
>
> There should be 3 processes:
>
> - the entrypoint
> - the Java wrapper
> - the Java process

5. File ownership should now also belong to the standard `ignition` user. You can use the following command to verify that there are no files owned by any other user aside from the standard `ignition` user:

User Migration Example (Command Line)

```
# Verify file ownership
docker exec example-gw bash -c "find /usr/local/bin/ignition ! -user 2003 | wc
-l"
```

6. Once you have verified file ownership and the user the processes are running under, you can now recreate the container against the new image with the default `ignition` user (UID = 2003) and update your Gateway to the latest version:

User Migration Example (Command Line)

```
# Recreate the container
docker stop example-gw
docker rm example-gw
docker run --name example-gw --rm -v example-gw-
data:/usr/local/bin/ignition/data \
    -p 8088:8088 \
    inductiveautomation/ignition:8.3.<new> \
    -n example-gw
```

Kubernetes

Helm Chart for Kubernetes Deployments

Users managing large-scale deployments may be interested in using the Ignition Container Image on Kubernetes. Kubernetes, or K8S, is an open source system for managing containerized applications. Helm, the package manager for Kubernetes, allows users to define, install, and upgrade full application stacks with ease and repeatability. Ignition offers an official Helm Chart to provide a standardized way to deploy an Ignition Gateway on a Kubernetes cluster. It details best practices, default value configurations, and system requirements.

The Ignition Helm Chart is available for cloud-native and containerized Ignition deployments, so Ignition can be deployed into Kubernetes clusters using a versioned, declarative Helm chart that serves as a fully functional starting point. The chart can be used to install Ignition directly or as a building block for deciding how you want to configure your architecture.

Refer to the official Ignition Helm Chart at **https://charts.ia.io/** for a ready-to-use deployment template that will get you up and running quickly while still allowing the flexibility to customize and expand the configuration to suit your specific infrastructure needs.