Technical Document

Niagara Analytics Framework Guide

February 21, 2023

niagara4

Niagara Analytics Framework Guide

Tridium, Inc.
3951 Westerre Parkway, Suite 350
Richmond, Virginia 23233
U.S.A.

Confidentiality

Trademark notice

Copyright and patent notice

Contents

About this guide

This guide contains important information about how to install and configure the Niagara Analytics Framework running on a Supervisor or remote host station.

Audience

The information in this guide is for Systems Integrators and Facility Managers who are responsible for configuring the tools used to manage complex building systems.

Document Content

This guide provides procedures for configuring each aspect of the Niagara Analytics Framework, and concludes with a troubleshooting chapter for resolving common problems and answering questions. An index is provided to help you find the specific information you are looking for.

Product Documentation

This document is part of the Niagara technical documentation library. Released versions of Niagara software include a complete collection of technical information that is provided in both online help and PDF formats.

Document change log

This topic provides a brief listing of changes made to the document.

February 21, 2023

Added practical examples.

Corrected errors throughout.

Clarified and rewrote explanations that may have caused confusion. For example, expanded what a data definition is how to use one.

Reworked procedures and updated screen captures.

December 2, 2022

Extensive changes made following review.

November 3, 2022

Reorganized several chapters, including disbursing the *Examples* chapter topics near related information.

Added many concept topics with examples for blocks.

October 19, 2022

Added to "Features" topic in *How the framework works*.

Changed the title of chapter 2 to *Getting Started*.

Emphasized tagging data in "Configuration overview" (*Getting Started*).

Added the *Examples* chapter.

Added tagging information to "Tags, hierarchies and relationships" in the chapter of the same name.

Added new topics, "Tags: direct and implied," "Applying a direct tag" and "Setting up implied tag rules" to the *Tags, hierarchies and relationships* chapter.

Significantly rewrote "Acquiring tags from remote stations" in the *Tags, hierarchies and relationships* chapter.

Added information to "Missing data configuration" in the *Missing data management* chapter.

Added "Creating a missing data strategy for a data set" to the *Missing data management* chapter.

In addition, updated screen captures and edited text.

October 23, 2020

- Added topic regarding `Privileged Action Exception` error
- Corrected information in Prerequisites topic about editing `nre.properties` file for increasing station memory.

March 10, 2020

Added best practice to refresh the UX chart when loading the report seems sluggish.

June 14, 2019

- Added Raw Data Filter topics to the *Outlier handling* chapter.

April 30, 2019

- Added the *Outlier handling* chapter.
- Added new scenario in *Troubleshooting* chapter.

September 5, 2018

Initial 2.1 release.

Related documentation

Several documents provide additional information about the *Niagara Analytics Framework*.

- *Niagara Analytics Framework Reference* documents each component and view.
- *Niagara Analytics Framework Web API Guide* documents the code you can use to extend this product.
- *Niagara Hierarchies Guide* provides information about setting up logical hierarchies.
- *Niagara Tagging Guide* provides information about adding metadata to objects.
- *Niagara Relations Guide* explains how to configure relationships in a hierarchy.
- *Niagara Graphics Guide* provides general information about how to create Px graphics.

Chapter 1 Getting started

Topics covered in this chapter

- ♦ Configuration overview
- ♦ Prerequisites
- ♦ Setting up a station
- ♦ Installing on a remote host
- ♦ About licensing
- ♦ Setting up user authentication
- ♦ Features
- ♦ How Analytics works
- ♦ The framework in the Nav tree
- ♦ Analytics library

The Analytics framework processes data in the local station: JACE, Edge device or Supervisor station. It does not support using virtual Ords to resolve data in remote stations.

To get started, you need a station with a tag dictionary that defines tags, tag groups and relations) and a hierarchical tree structure for devices and points.

Applying dictionary tags to the components in the station's hierarchy creates a data model.

Applying tags and configuring properties answer the significant questions: what? where? how far back? and which?

- What? The **Property Sheet** for the lowest child node in the tree structure sets up the request (query) by defining input sources and output values. An input source may be a history or the value generated by a real-time driver point in a local or remote station.
- Where? A unique **Data Definition** component associated with each point may define, among other things, which **`History`** file or real-time value (identified by a **`Value Ord`**) to use. If multiple values are involved, the Px properties identify which node to search for multiple instances of a given data source.
- How far back? The **`Time Range`** property defines how far back in time to go when collecting data. Properties define how to represent output (the roll up of history using an interval or the aggregation of multiple current and historical values), and how to calculate values (sum, average, etc.)
- Which? Tags filter and define the source data to include in output values and charts. Formulas, diagnostics and Px widgets can use tags.

Once all properties and nodes are configured, request results are available as needed in Px views and through the use of Niagara Analytics Explorer.

Hierarchical data model tree structure

Parent and child nodes arranged in a hierarchal tree comprise the primary organizational unit of the data model. Nodes can contain other nodes. They may represent geographical locations, groups of buildings, individual buildings, types of systems, types of tenants, devices, and so forth. Generally, they represent tangible things in the data model.

Proxy nodes in a Supervisor station imitate nodes in a remote station. Their input values mirror the input device values contained in the remote station.

NOTE: You do not have to completely rebuild your **Drivers** folder. You can start working with a few nodes and build your data model with formulas and diagnostics a little at a time.

You assign tags to nodes. Tags identify types of nodes. For example, your application may include many buildings, each with a unique name. When you tag each structure with the "building" tag, formulas or diagnostics can easily find all of them.

Configuration overview

Configuring the a data model for the first time should begin with a planning phase in which you decide on the information to analyze.

The Niagara Framework® maintains a hierarchy of components, devices, and points that reflect the physical network to which each object belongs. While information from each device and point is useful for tracking real-time values and raising alarms, Niagara 4 provides separate hierarchies, tags and relations with which to set up more meaningful relationships that may have nothing to do with the physical arrangement of devices on a network. The Niagara Analytics Framework (referred to as the framework in this documentation) builds on these standard Niagara 4 features to collect and analyze real-time and historical data in a variety of ways.

For example, your campus may include many buildings. Each building has its own AHU unit for which the data model monitors the values generated by three points: Cool Setpoint, Heat Setpoint and Supply Temp.

The following list summarizes the tasks involved in configuring the data model in this environment:

- If you are using a Supervisor computer, you need to add the framework components to the station. A Supervisor computer provides the necessary resources to process large quantities of data.
- You add the `a` tag to each device point that will be part of the data model.

  Tagging data is the key to setting up the model. The easiest way to tag data is by using a tag dictionary with a tagging rule.
- Each tag may be accompanied by a data definition, which identifies the type of data (cooling capacity, temperature, voltage, etc.) the tag represents. You can view the associated data definitions at **AnalyticService→Definitions**.
- After tagging each point, you mayset up an optional hierarchy by geographical location or, perhaps, by the person responsible for maintaining the AHU unit(s).
- Next, you use a pre-defined algorithm (formula) or create your own algorithm to perform calculations. These calculations define how to combine historical trend data, and maximum and minimum acceptable values.
- Each algorithm includes a **Data Source Block**. The **`Data`** property for this component contains the same tag as that used to tag the device points. Data collection happens by virtue of the assigned tags and hierarchy without requiring complicated programming.
- To visualize the collected data, you bind an algorithm to a Px or Web chart that can take the form of a dashboard or report. To run an algorithm you open the chart that references the algorithm or set up a poller to run the algorithm at regular intervals.
- Alerts use algorithms that yield a binary result (`true` or `false`). A `true` result can generate an alarm, which appears on the standard alarm console.
- A control point with an Analytic Proxy Extension stores the result of processing an algorithm and can serve as an input to standard components for the purpose of controlling device performance based on logic.

NOTE: You can start by working with a few tags, hierarchies, and algorithms, learning how to visualize and manage the results a little at a time.

Prerequisites

To use Niagara Analytics Framework, several conditions must be met.

Niagara certification

As the systems integrator or data model designer, you have completed Niagara certification training and are familiar with the Workbench interface.

Supported hosts

Niagara Analytics Framework runs on a Niagara 4 Supervisor, JACE-8000 orNiagara Edge 10 host.

Memory requirement

The number of points, algorithms and history records a host (PC or remote host) can process is limited by available memory. Compared to running in a JACE or Edge device, more memory resources are available when running a Supervisor station on a PC.

A lack of adequate memory (heap memory) to run the framework can cause a Supervisor station to fail. The amount of available heap memory on a PC is determined by the **Xmx** property as configured in the `nre.properties` (Niagara Runtime Environment) file. This text file is located in the `etc` folder under the `daemon user home` directory. Your PC must have enough physical memory resources to accommodate any change you make to the **Xmx** property.

NOTE: You can choose the `daemon user home` location during Workbench installation. Assuming you install on a "C" drive, the default location is `C:\ProgramData\Niagara4.x\<brandname>`. See the *Selecting the Daemon User Home location* topic for related information about `daemon user home`.

License requirement and limitations

Niagara Analytics Framework must be licensed for your host (Supervisor PC, JACE or Edge device). You add the feature to an engineering license using the standard Niagara licensing model.

The license limits the number of points the framework can use. An `a:a` tag on a point marks it as being used by the framework. The **AutoTagAnalyticPoint** property on the **AnalyticService** controls the automatic tagging of points used by the framework. When this property is set to `true`, the framework applies the `a:a` tag to any point referenced by an analytic request.

Modules required

These modules run under all framework versions.

- `analytics-lib-ux.jar`
- `analytics-rt.jar` is required by both stations and engineering platforms running tools (Workbench)
- `analytics-ux.jar`
- `analytics-wb.jar` is the user interface. This module is required to run the engineering tool (Workbench).

Core software and modules required

The framework requires the latest version of Niagara. The specific modules the framework requires include:

- `alarm.jar`
- `baja.jar`
- `bajaui.jar`
- `bql.jar`
- `control.jar`
- `driver.jar`
- `email.jar`
- `file.jar`
- `fox.jar`
- `gx.jar`
- `history.jar`
- `niagaraDriver.jar`
- `platform.jar`
- `schedule.jar`

- wbutil.jar
- web.jar
- workbench.jar

These modules reside in the modules folder.

Browser requirement

The framework's visualization tools include web charts (Ux charts) with scalable, vector graphics. These graphics require a browser that supports HTML5.

Station configuration

The procedures in this guide assume that you have configured the network with at least one remote host and station whose device drivers and points have been set up and configured for your application. It also assumes that all proxy points have been discovered and configured in any remote host station, and in your Supervisor station.

Setting up a station

To set up the framework in a station, you drag the **AnalyticService** from the **analytics** palette to the **Services** folder in the Nav tree.

Step 1 Open Workbench and connect to your Supervisor station.

Step 2 Open the **analytics** palette.

Step 3 Drag the **AnalyticService** component from the palette to the **Services** folder in the station's Nav tree.

When you add the **AnalyticService**, the framework automatically adds a standard Tag Dictionary named **analytics** to the station's **TagDictionaryService** component. This dictionary includes a single a:a marker tag under its **Tag Definitions**.

Step 4 To use algorithms from the analytics-lib palette, open the **analytics-lib** palette and expand the **Tag_Dictionary** folder.

Many of the algorithms provided in the **analytics-lib** palette (**Algorithm→English** or **Algorithm→Metric** palette sub folders) use tag definitions that are not found in the default Niagara or Haystack tag dictionaries. Those additional tags are defined in a tag dictionary named Analytics (subtle difference with a capital A). This tag dictionary is in the **analytics-lib** palette under the **TagDictionary** sub folder.

Step 5 Delete the existing analytics tag dictionary from the station's **TagDictionaryService**.

Step 6 Drag (or copy and paste) the **Analytics** container from the palette to the **TagDictionaryService**.

Step 7 As a best practice, save the station (right-click the station in the Nav tree and click **Save Station** from the drop-down menu).

Installing on a remote host

This is the preferred installation method for installing the framework on a remote host.

Step 1 Open Workbench on your Supervisor computer.

Step 2 Open a platform connection to the remote controller, and connect to the station.

Step 3 Open the **Software Manager** view in the remote station and scroll down to the analytics module in the list of software installed on the Supervisor computer.

Step 4 Select analytics and click **Install** (at the bottom of the view).

The installation software checks the versions of all other installed modules on the host, displays a list of any that are out-of-date (compared to the modules installed locally), and pre-selects an install option.

Step 5 De-select the install option for any modules other than the framework modules.

CAUTION: Do not bring any modules other than the framework modules up to date.

Step 6 To install the module files on the remote host, click **Commit**.

The **Software Manager** may need to automatically stop any running station. It displays a confirmation window.

Step 7 To confirm the station stop, click **OK**.

The **Software Manager** stops the station and continues with the module installation process. When finished, the **Software Manager** displays that module files are `Up to Date`.

About licensing

For a point to be used by the Niagara Analytics Framework, it must be tagged with the `a:a` (analytics) tag. The system automatically compares the total number of points thus tagged with the allowed points on your license.

Be aware of these licensing-related factors:

- When set to `true`, the **AutoTagAnalyticPoint** property on the **`AnalyticService`** causes the framework to automatically tag each new point with the `a:a` tag as required for use with the service. During initial station configuration, setting this property to `true` saves time. However, once the framework is configured and running in your Supervisor station, you should set this property to `false`.
- If the **analytics** tag dictionary and `a:a` tag definition are missing from the **TagDictionaryService**, the **`AnalyticService`** tries to add them to the **TagDictionaryService**.
- You can use a BQL scaler function in a platform's Program service to query points with a specific tag (such as the `a:a` tag) and bulk edit them.
- If you remove the `a:a` tag from a point, you must refresh cache to decrement the `a:a` tag counter. To refresh cache, right-click the **`AnalyticService`** and click **Actions→Refresh Cache Full**.

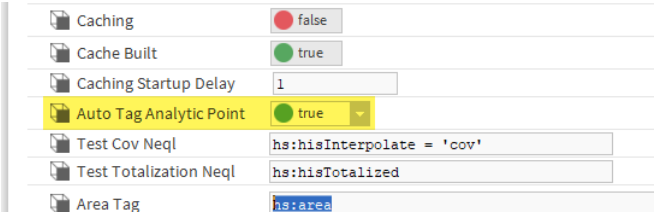
Configuring the service for licensing

A Niagara Analytics Framework license is based on the number of points used in algorithms. As you identify the points to use in algorithms and alerts, the system automatically tags each point with the `a:a` tag. You can view the number of points configured. An **`AnalyticService`** property turns the automatic tagging on and off. This procedure explains how to configure this property.

Prerequisites: You are working in Workbench and are connected to a station.

Step 1 Right-click the **`AnalyticService`** and click **View→Property Sheet**.

The **`AnalyticService`** property sheet opens.



Step 2 To enable automatic tagging with the `a:a` tag, set **`Auto Tag Analytic Point`** to `true` and click **Save**.

You enable this property before adding algorithms and alerts. Once your framework is configured you should disable this feature.

CAUTION: If you leave this feature enabled, and, at some future time, exceed the number of licensed points, the framework will stop working.

Determining the number of points used

This procedure explains how to determine the number of points used. This number must not exceed the number of points allowed by the license. If it does, the framework stops working.

Prerequisites: You are working in Workbench and are connected to a station.

Step 1 Right-click the **AnalyticService** in the Nav tree and click **View→Property Sheet**.

The **AnalyticService** property sheet opens.

Step 2 Check the **Point Count** property.

The system counts each point that has the `a:a` tag associated with it. The system automatically adds this tag as you configure algorithms and alerts. If you change your mind about using a particular point in a calculation, you can remove the `a:a` tag from the point.

If you remove the `a:a` tag, you must rebuild memory cache for the deletion to take effect.

Step 3 To rebuild cache, right-click **AnalyticService** and click **Actions→Rebuild Cache**.

Rebuilding cache re-calculates the **Point Count**.

Confirming that the AnalyticsService component is licensed

A component is licensed if its **Status** property reads `{Ok}`.

Prerequisites: You are working in Workbench and are connected to a station.

Step 1 Right-click the AnalyticService and click either **Views→Ax Property Sheet** or **Views→Property Sheet**.

The property sheet opens.

Step 2 Confirm that the **Status** property reads `{Ok}`.

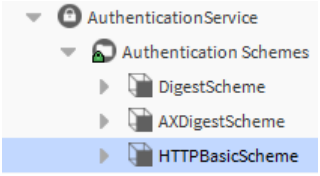
You are ready to begin analyzing data and spotting trends.

Setting up user authentication

Access to a station database requires user authentication, which is managed by the station's **AuthenticationService**. As with physical access, programmatic access requires authentication using the HTTPBasicScheme (HTTP Basic Authentication Scheme). Consider using a separate user for each type of access (physical access, programmatic access, etc.). This practice provides additional security as each user requires only the minimum number of access rights necessary to accomplish a specific task. Using roles and tagged categories allows for highly-configurable permissions for accessing various station components.

Prerequisites: You have administrative rights. The station is open in Workbench.

Step 1 Open the **baja** palette.



Step 2 Add the **HTTPBasicScheme** under the **Services→Authentication Service→Authentication Schemes** node in the nav tree.

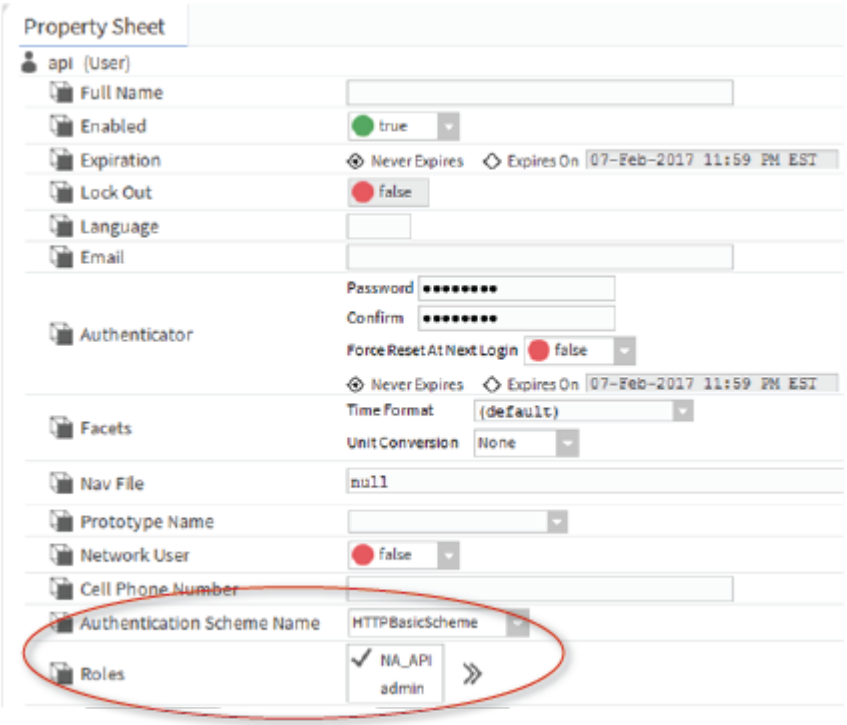
Step 3 Create a role to assign to users based on the type of access.

For example, users who are permitted to create and view web charts may be assigned the "NA_charts" role. Permissions for this role might allow a user to read from the database but not invoke actions or write records to it.

Users who are permitted to query the station database with API calls may be assigned a "NA_API" role. Permissions for this role might allow a user to read from the database and invoke actions, but not write records to it.

Step 4 Expand **Services→UserService** in the nav tree and double-click the user name you intend to use to access the station database.

The **Edit User** window opens.



Step 5 Select `HTTPBasicScheme` from the **`Authentication Scheme Name`** drop-down list, assign the role you created, and click **OK**.

Features

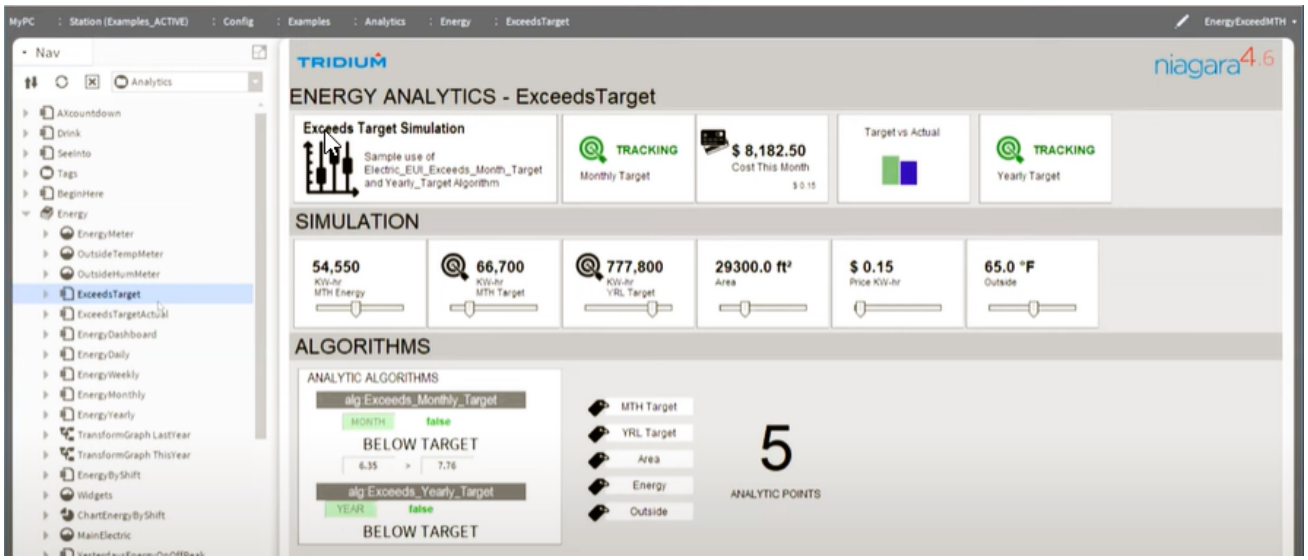
The heart of the framework is an advanced high-performance calculation engine. With this engine, real-time data can be combined with historical data using a set of wire and property sheets. This visual programming interface defines algorithms (formulas) that analyze the real-time and trend data collected from components, devices, and points. The output from this analysis can be visualized in charts and used as input to standard Niagara logic.

When applied to historical and real-time data, framework algorithms (formulas) can help you gain insight to better manage your operations. The product includes these features:

- An open and extensible analytical environment that you can customize to meet your needs
- Analytic tools that apply to any industry, including manufacturing, as well as building management
- The ability to set up complex analysis without custom programming
- Support for third-party API visualization and other complementary applications

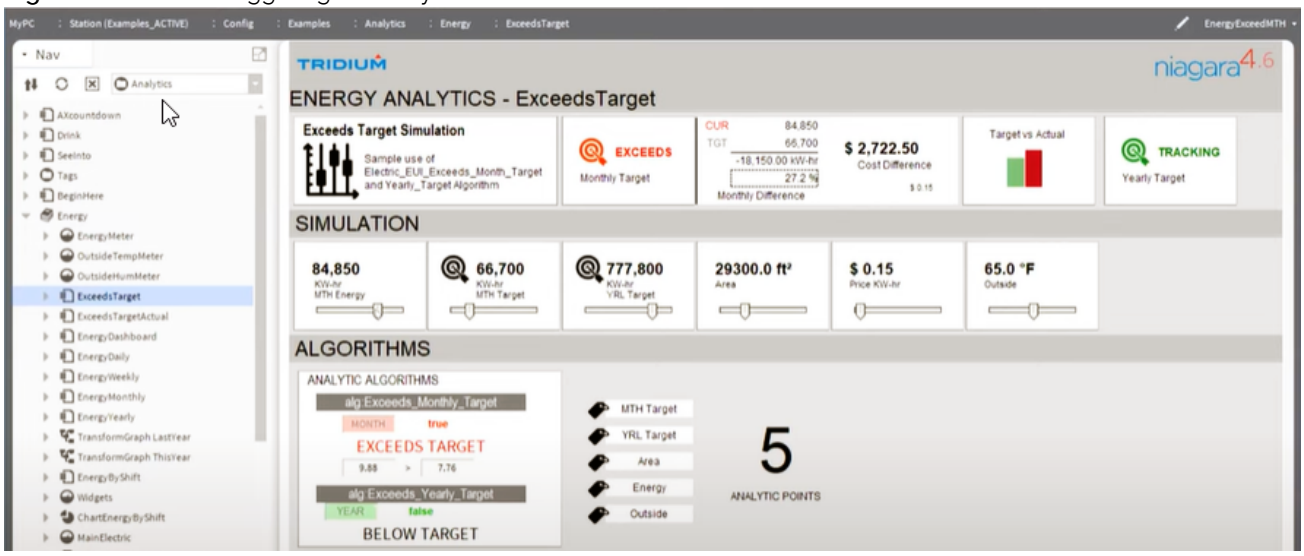
With these tools you can compare monthly and yearly energy expenditures against targets.

Figure 1 Example of energy analytics



You can scroll through histories to take a look at statistics.

Figure 2 Result of triggering the analytic model



In this example, you can compare your energy for the month to your budget and observe the result. This is evaluating the data in the station by a given time period, every 15 seconds in this example.

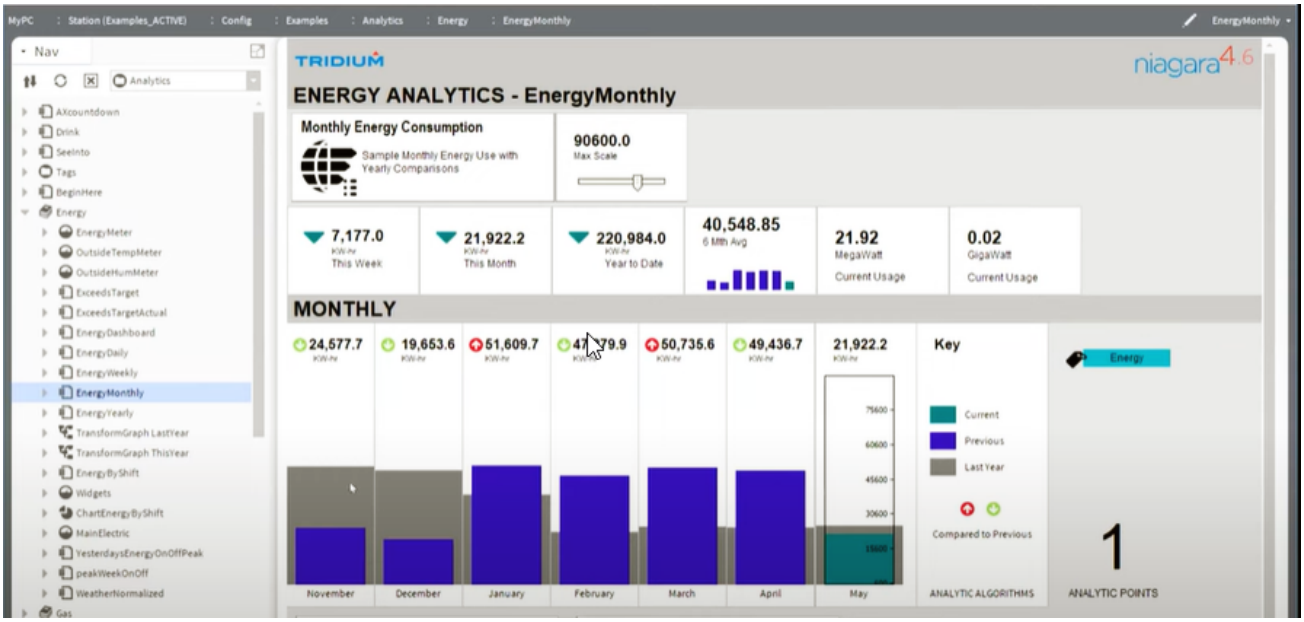
The framework can:

- Look for something and let you know it found it.
- Perform a calculation on a set of inputs to give you the result.
- Set up a graphic to visualize data for a particular need.
- Look for faults in systems if you know how a system fails.

The custom rules you create are very powerful.

Often people want to compare a series of histories with a baseline.

Figure 3 Comparison to a baseline



The gray in the background shows last year's monthly performance compared to this year's blue bars.

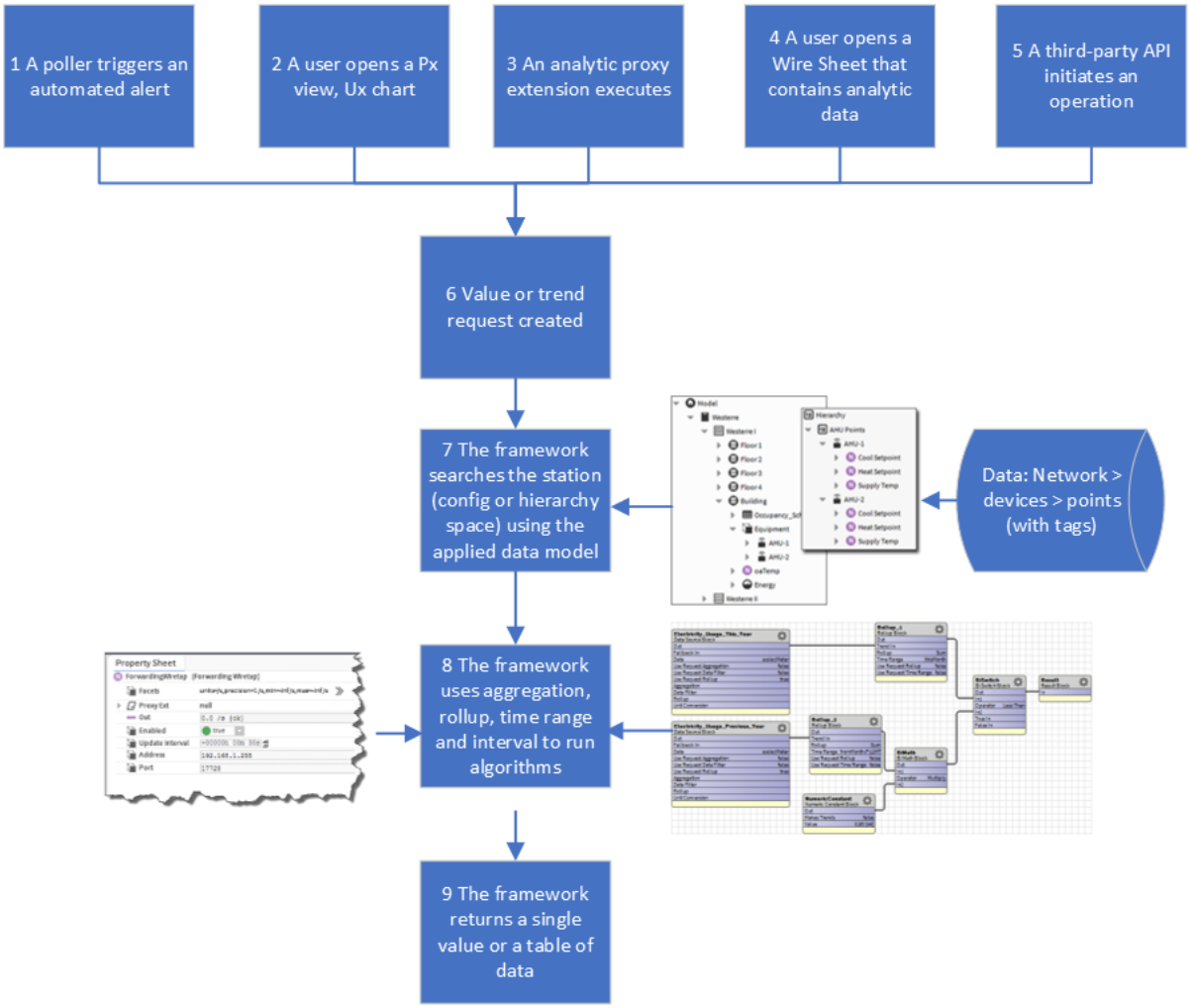
Algorithms designed to work with a specific data type, such as electrical consumption (KWH), could easily be duplicated and modified to instead work with water consumption (gallons) or gas consumption (ccf). This is possible because inputs to algorithms are defined based on tags, such as `hs:energy` to identify a source of electrical consumption instead of being bound to specific control points (end points) in the station.

How Analytics works

The structure of Niagara 4 makes it possible for the framework to render charts, generate alerts and automate device management with minimal configuration effort.

At the risk of oversimplifying framework workflow, the following flowchart shows how a request for data initiates processing, which results in human-readable analysis and action.

Figure 4 Flowchart



In the standard Niagara Framework, device points update on change of value (CoV). By contrast,Niagara Analytics Framework points update upon request. An analytic request is inherent in the structure of the data model, formulas and Px views. There are two types of requests:

The framework processes an analytic request when:

1. A poller triggers an analytic alert
2. A user opens a Px view, which causes a Px widget (analytic chart, analytic table, bound label with analytic binding, or any widget with an analytic binding) to execute
3. An analytic proxy extension executes.
4. A user opens a Wire Sheet that contains analytic data.
5. The framework receives an analytic web API request.
6. An analytic request may just be for a type of data, such as `hs:zoneAirTempSensor` and not specifically an algorithm like `alg:HighTemp`.
7. The request uses tags to pull data from the station database.
8. In addition to the **Property Sheet**, the framework uses the familiar **Wire Sheet** as its canvas on which to collect source data. An algorithm then analyzes the data you tagged by executing the blocks on the **Wire Sheet**.

Alerts, Algorithms, Data Definitions, Analytic Proxy Extensions, Data Source Blocks, and Request Overrides use aggregation, rollup, totalize and data filters to configure calculations.

9. A value request returns a value. A trend request returns a table, which contains one or more records including at least a timestamp (BAbsTime) and value (Boolean, Numeric, Enum or String).

There is no way to directly invoke an action based on the result of an analytic query, but the results of an algorithm, resolved through a control point with an analytic proxy extension, could be linked into some custom component that might invoke an action.

The framework in the Nav tree

To begin your understanding of how the framework data model works symbiotically with Niagara, this topic points out where in a typical Nav tree framework configuration appears. These are the areas you use most frequently as you set up the framework.

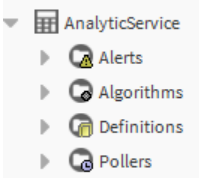
TIP: Consider opening each container in a separate tab so you can quickly navigate among them without having to scroll the Nav tree.

Drivers container

Your **Drivers** folder models your network and devices. This is where all your points reside. When tagging points, you can use this space or the Hierarchy space (once you create one or more hierarchies).

The AnalyticService

Figure 5 AnalyticService under the Services folder



You use the **AnalyticService** in the **Services** container to configure framework properties. This folder also contains four sub-folders for configuring framework features:

- The **Alerts** folder contains alerts. These components use algorithms to evaluate conditions and may or may not generate an alarm.
- The **Algorithms** folder contains the formulas used by alerts, Px Views and Web Charts.
- The **Definitions** folder contains data definitions.

  A **Data Definition** is an optional component the framework uses to configure default properties for a specific data item defined in a station. Data Definition components simplify configuration by defining defaults that apply across the framework. These default properties may be overridden for a specific analytic request in an Alert, Analytic Proxy Ext or Binding.

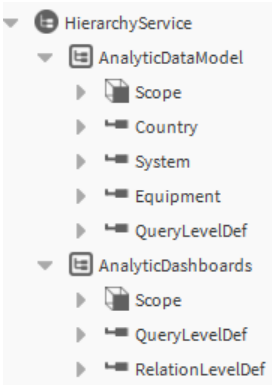
  An **Analytic Data Folder** under the **AnalyticService** groups **Data Definition** components. The **Analytic Data Manager** serves as the primary view for this folder.

  You may have a data definition for each type of data to be used in formulas and charts. Pre-defined data definitions are located in the data model's **Dictionary**. You can modify and add to these definitions.

  A formula (algorithm) may serve as a data definition. A graphic widget also provides a data definition property.
- The **Pollers** folder contains components to manage data sampling frequency.

HierarchyService

Figure 6 An example of a HierarchyService

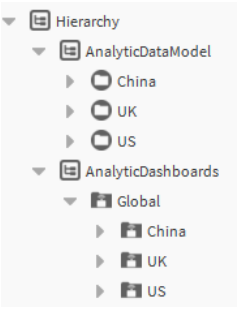


You use the **HierarchyService** to define a logical data model that is independent of your drivers-network-device model.

While this service is a feature of Niagara 4, and not specifically a feature of the framework, it is indispensable to the framework. The **HierarchyService** provides the foundation for the alternative, meaningful data model that appears in the **Hierarchy** space.

Hierarchy folder

Figure 7 Example of a Hierarchy node

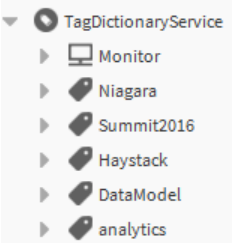


When you save a hierarchy, the **HierarchyService** automatically creates a **Hierarchy** structure at the same level as the **Config** container. You can use this hierarchy space to navigate from point to point.

The example above defines two hierarchies. The **AnalyticDataModel** organizes building equipment (AHUs, etc.) by geographic location. the **AnalyticDashboards** hierarchy contains all dashboard representations for the geographical locations.

TagDictionaryService

Figure 8 An example of a TagDictionaryService



You use the **TagDictionaryService** to set up tags. When assigned to individual points within a hierarchy, the framework uses tags to identify data source values.

While the tag dictionary is a feature of Niagara 4, and not specifically a feature of the framework, the Niagara Analytics Framework provides one of the most compelling reasons to use tags. This is where you can create your own tag dictionary.

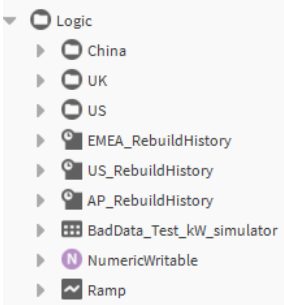
You define inputs (data sources) and outputs by tagging data and configuring properties. Several tag dictionaries provide the tags to apply to points including:

- Haystack tag dictionary
- Niagara tag dictionary

You may create your own tags using an **Analytics dictionary** component under the **TagDictionaryService**.

Folder to contain Analytics' logic

Figure 9 An example of a logic folder

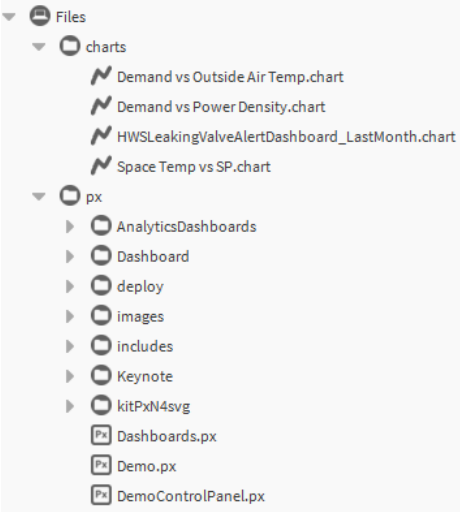


At the same level as the **Services** and **Drivers** containers, you can create a folder to contain framework-specific components, such as Px views, schedules, special points, time triggers and proxy extensions.

The example screen capture above is from a demonstration station. Your logic folder may be very different.

Files folder

Figure 10 Example of chart files in the Files folder



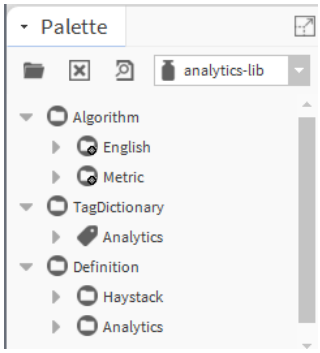
The **Files** folder contains framework chart files.

In the example, two folders separate the UxCharts (Web Charts) from the Px Views.

Analytics library

The **analytics-lib** palette provides the pre-defined algorithms for a variety of common calculations. You can modify them or use them as examples when creating your own algorithms. To use an algorithms drag it to a **Wire Sheet** and supply at least a data source. The *Algorithm Library* chapter of the *Niagara Analytics Framework Reference* documents these pre-defined formulas.

Figure 11 analytics-lib palette



The **analytics-lib** palette consists of three folders:

- The **Algorithm** folder contains two sets of pre-defined algorithms: one set that supports English and the other that supports Metric units of measure. Each set contains instances of all algorithms including those that do not require specific units.
- The **TagDictionary** folder contains a tag dictionary named Analytics with pre-configured tag and tag group definitions. Those tag and tag group definitions provide algorithms, which you assign to applicable points in the station.
- The **Definition** folder contains English and Metric configurations. A definition is a set of properties that associates specific processing information with each tag, and consequently with each object (point) to which the tag is assigned. You can remove, modify, and add new definitions. Often the best course of action is to copy an existing definition, rename it with a unique name, then modify it to meet your needs.

Chapter 2 Tags, hierarchies and relationships

Topics covered in this chapter

♦ Tags: direct and implied
♦ Hierarchy setup
♦ Relationships

A station's **Config→Drivers** tree structure organizes a physical group of devices by network protocol, device, and point. This structure does not, for example, identify which building, region or tenant a point belongs to. Using the powerful hierarchy, tag, and relations features of Niagara 4, the framework can set up data source structures for analysis without requiring you to completely configure a separate data model.

A tag is a piece of information added to objects to make them more accessible and flexible for search and analysis. Tags support the use of hierarchies to organize objects in a station.

A hierarchy is a logical navigation tree for a system. Rather than define each element of an organization in a navigation (Nav) file, the **HierarchyService** defines a navigation tree based on a set of level definition rules.

A relation connects components to one another for the purpose of building a hierarchy.

Tags are contained in dictionaries. You can use a standard tag dictionary, such as Haystack, or your own tag dictionary.

Tags, hierarchies and relations are features of Niagara 4. More information about working with them can be found in the *Niagara Tagging Guide*, *Niagara Hierarchies Guide* and *Niagara Relations Guide*.

Tags: direct and implied

Tagging identifies, in a consistent way, the data to include in an analytic request. This may be the most important set up task and is the last step before running a query.

There are two ways to tag points so that the framework can find the data to analyze:

- with a direct tag
- with an implied tag

Setting up points with direct tags can be a very time-consuming process, especially if you have hundreds of points to analyze.

Using implied tags is the best practice. Implied tags rely on rules you set up to assign tags to points. These rules depend on your point naming convention. The best way to tag data is with a tag rule.

Three standard (default) dictionaries are available with the framework.

- The Niagara dictionary contains commonly-used tags.
- The Haystack dictionary is an open source dictionary created by Project Haystack to "streamline working with data from the Internet of Things." (project-haystack.org).

  The Niagara Analytics Framework comes with a third pre-configured Analytics Tag Dictionary.

The framework automatically created an Analytics tag dictionary for you when you added the **AnalyticService** to the station. A pre-configured Analytics dictionary in the **`analytics-lib`** palette includes the additional tag and tag group definitions used with the provided algorithms. Before you create your own dictionary, view the tags provided by these tag dictionaries. They may meet your needs.

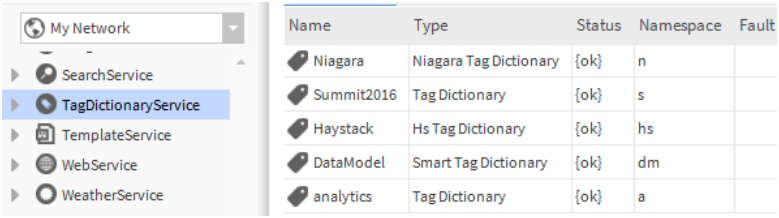
Creating a tag dictionary

You would create your own tag dictionary to apply your own unique tags.

Prerequisites: The station is connected. You are working in Workbench

Step 1 To confirm the presence of tag dictionaries, double-click the **TagDictionaryService** in the Nav tree.

The **Tag Dictionary Manager** opens.



Step 2 To create the Analytics tag dictionary in a station, open the **`analytics-lib`** palette, expand the **`Tag_Dictionary`** folder in the palette, and drag (or copy and paste) the **`Analytics`** tag dictionary to the **Config→Services→TagDictionaryService** in the Nav tree.

Step 3 To view the tags contained in any tag dictionary, expand the **TagDictionaryService**, expand the specific tag dictionary name, and expand or double-click the **Tag Definitions** node.

Step 4 To view the **Property Sheet** for an individual tag, double-click the tag in the list.

Applying a direct tag

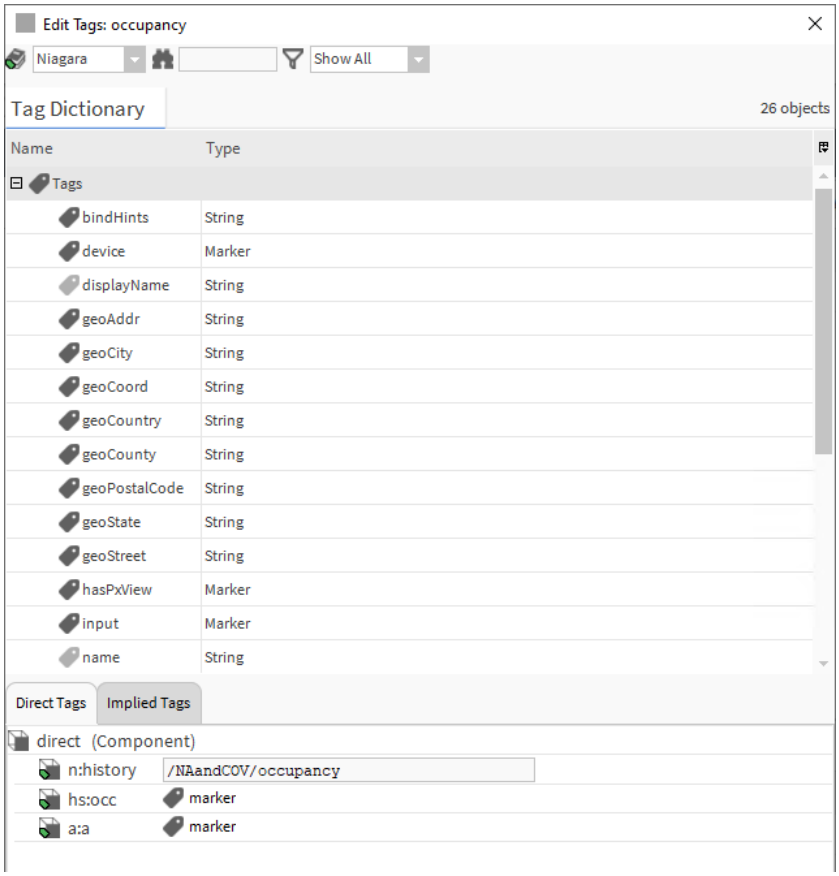
Direct tags add metadata to points.

Prerequisites: You are logged in to the station with the points you intend to tag. The tag dictionary(ies) you want to use are available.

Step 1 Locate the point to tag.

Step 2 Right-click the point and click **Edit Tags**.

The **Edit Tags** window opens.



You may have several dictionaries open. The Niagara dictionary is open by default.

Step 3 Use the Tags or Tag Groups trees in the upper pane to locate and select the desired tag or tag group, then click the **Add Tag** button to assign the tag or tag group to the lower pane (direct tags).

You may directly associate more than one tag with a point.

Step 4 After adding all relevant tags, click **Save**.

Setting up implied tag rules

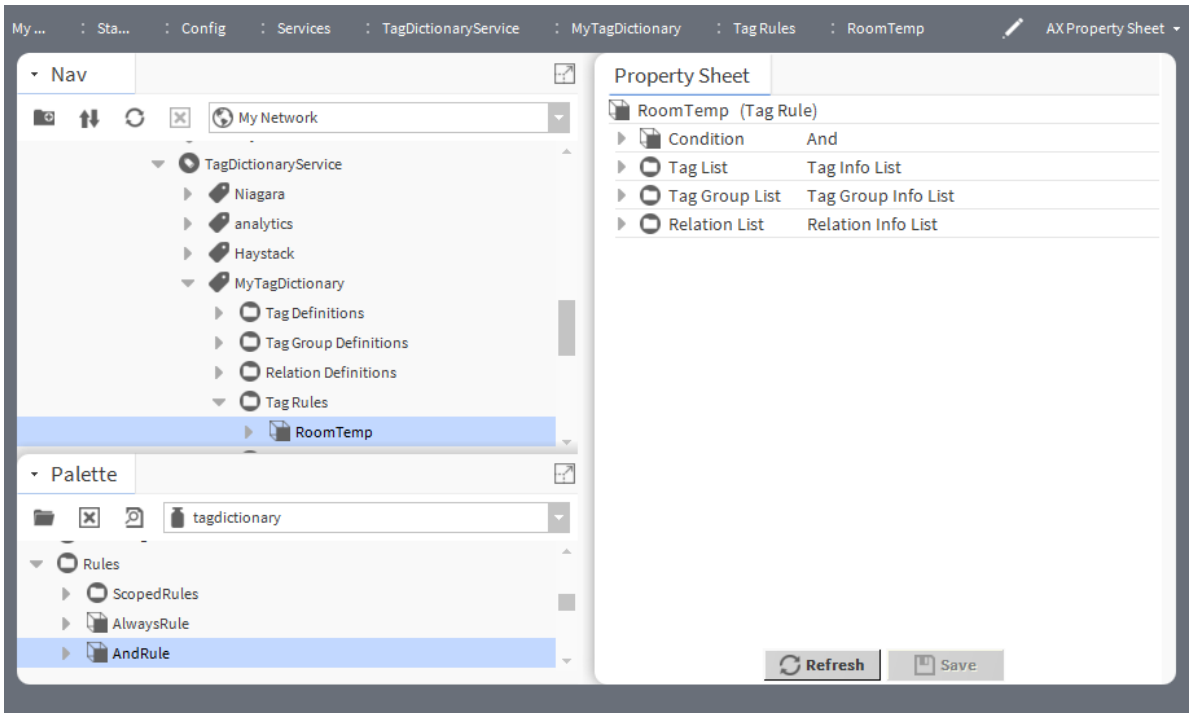
To automatically assign tags to points, you set up tag rules in your smart tag dictionary that are based on your component naming convention, component type or assigned tags. Building tag rules in the Supervisor station assign implied tags to **NiagaraNetwork** proxy points or other drivers' control points (BACnet, Modbus TCP/IP, OPC, etc.).

Prerequisites: You have a component naming convention and have created your own smart tag dictionary under the **TagDictionaryService**. The **`tagdictionary`** palette is open.

Tag rules reside under the **TagRules** (TagRuleList) component of a smart tag dictionary. The Niagara and Haystack dictionaries have pre-defined tag rules, which apply the tags or tag groups defined in the applicable dictionary as implied tags to components in the station. These dictionaries are frozen dictionaries. You cannot add to, delete from or edit their tag rules. Your goal in this procedure is to create your own rule in your own smart tag dictionary. The framework can then assign the tags defined by your rule automatically to the points and other components in your station.

Step 1 Expand your smart tag dictionary node under **Services→TagDictionaryService**.

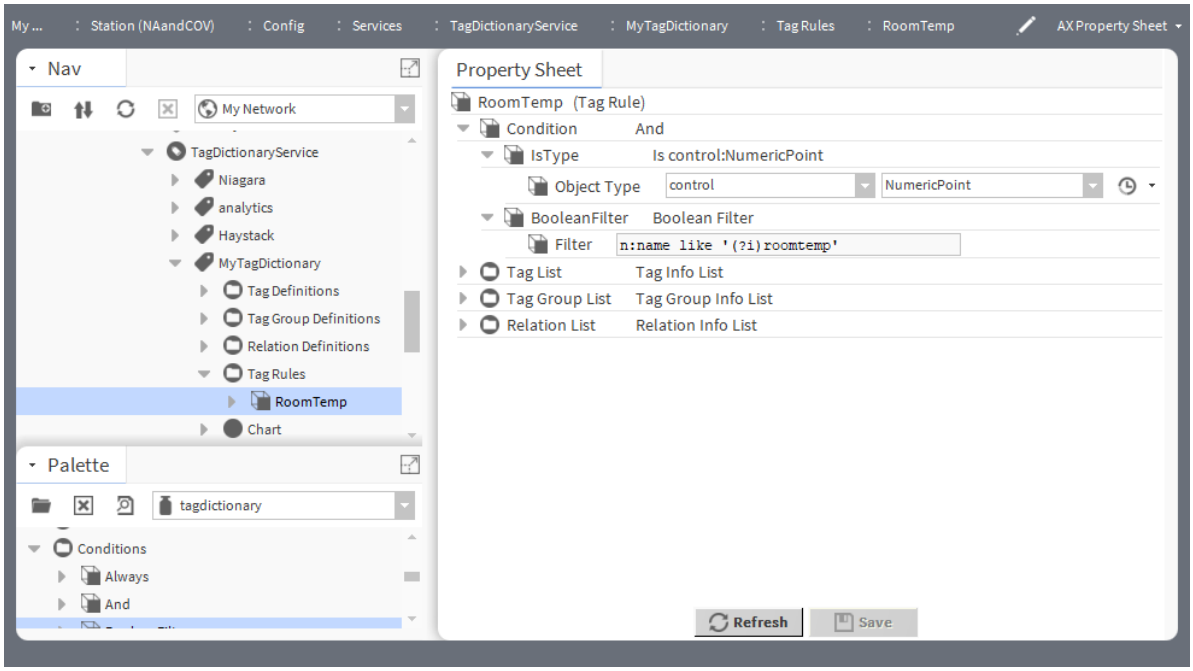
Step 2 Expand the **Tag Rules** folder in the palette and drag a rule, such as the **AndRule** from the **`tagdictionary`** palette to the **Tag Rules** folder under your smart tag dictionary.



In the example, the smart tag dictionary is called **MyTagDictionary**.

The palette provides several types of rules including: **AlwaysRule**, **AndRule**, **BooleanFilterRule**, etc. This procedure demonstrates the **AndRule**.

Step 3 To open the rule's **Property Sheet**, double-click the rule, expand its **Condition** node, expand the **Conditions** node in the palette and drag one or more conditions to the and tag rule.



In the example, **RoomTemp** is an **AndRule**, which checks for two things:

- The point must be a numeric control point (**IsType**).

- The name of the point must contain some form of the words “roomtemp” (**BooleanFilter**).

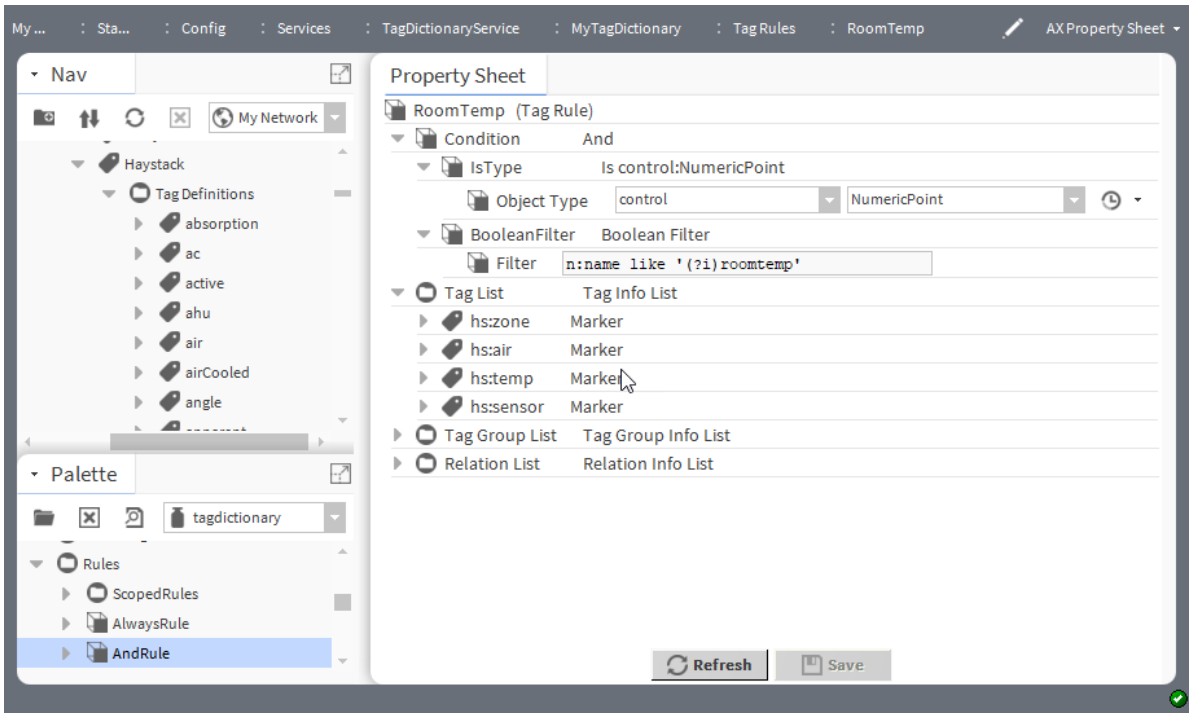
The idea here is that the tag rule might be applied to points named RoomTemp, roomtemp, SpaceTemp, ZoneTemp, etc. All are different naming derivations for a point that represents a zone air temperature sensor reading.

Step 4 Configure the tag rule conditions.

`n:name like '(?i)roomtemp'` is a NEQL predicate, which configures the rule to apply the listed tags to any point that contains “roomtemp” in its name.

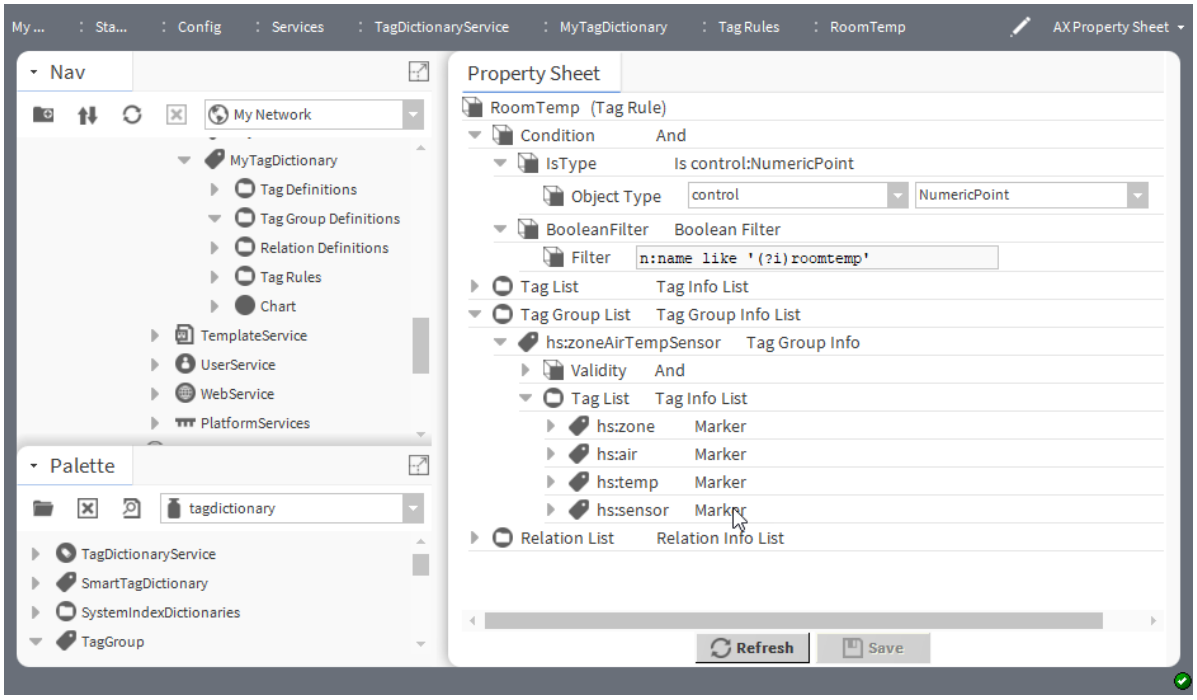
The `like` key word in the NEQL predicate leverages regular expressions (regex) that specify a search pattern in text. The `'(?i)'` syntax performs a case insensitive comparison of the following text. In this case, the pattern matches actual names of numeric points in the station, such as RoomTemp, roomtemp, ROOMtemp, ROOMTEMP, roomTemp and other capitalization variations in the name.

Step 5 To populate the **Tag List** under the tag rule, copy and paste tags from one of the standard tag dictionaries, such as the Haystack dictionary or the **Tags** sub folder from the **`tagdictionary`** palette.



The tags to be assigned are `hs:zone`, `hs:air`, `hs:temp` and `hs:sensor`.

Step 6 If you are using a rule in your smart tag dictionary to apply a tag definition or tag group definition from another dictionary, expand the **Tag Group List** and specify the fully-qualified names, such as `hs:zone` or `hs:zoneAirTempSensor`.



Without any other action, the framework applies the configured tags and tag group definitions in this tag rule as implied tags to any component in the station whose name matches the conditions defined by the rule.

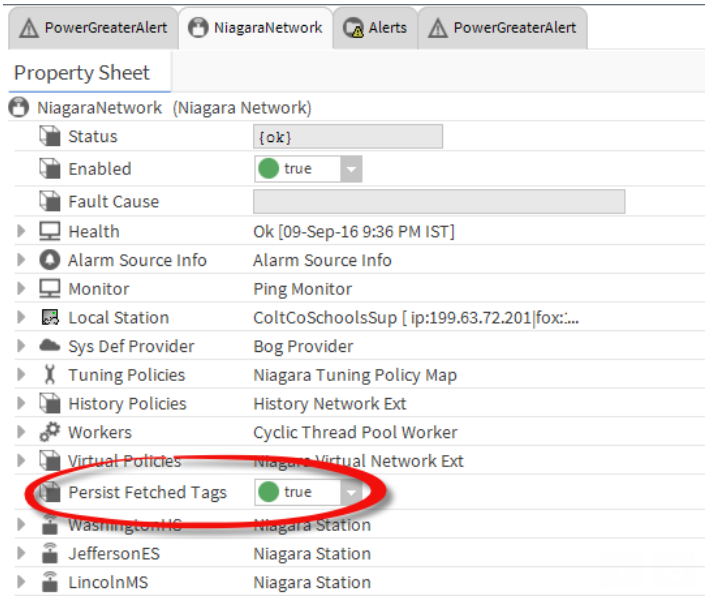
Tagging proxy points with n:history

To render data on charts and in tables, each point requires an `n:history` tag. As newly-discovered proxy points coming in from remote stations do not have history extensions with an implied `n:history` tag, use this procedure to automatically add the `n:history` tag to each newly-discovered point.

Prerequisites: Points have been discovered.

Step 1 Right-click the **Config→Drivers→NiagaraNetwork** node in the Nav tree and click **Views→AX Property Sheet**.

The **NiagaraNetwork AX Property Sheet** opens.



Step 2 Enable **`Persist Fetched Tags`** (change its value from `false` to `true`).

Step 3 To finish preparing the points for analysis, right-click again and click **Actions→Force Update Niagara Proxy Points**.

This adds a direct `n:history` tag to the network control point with a tag value matching the `n:history` tag of the remote control point. The framework adds the direct `n:history` tag to the network control points where the control points in the remote station have an `n:history` tag, there is a history import descriptor configured to import the applicable remote history to the Supervisor's history database, and the remote history has been successfully archived to the Supervisor's history database.

NOTE: It may take a few seconds to add the `n:history` tag to all newly-discovered points. Be patient and wait for the procedure to complete.

BACnet Network points with n:history

If BACnet control points in the station have standard history extensions, they already have the implied `n:history` tag assigned.

If a BACnet device supports trend log objects and is collecting trend data in the BACnet device, the station's BACnet driver may be configured to import the BACnet trend log objects from the remote BACnet device. This is a similar situation to importing histories from remote stations instead of adding a history extension to a network control point.

The BACnet protocol does not support the concept of reading an `n:history` tag from a remote BACnet device. To map from the control point to the imported BACnet trend data—if you are using BACnet history imports—you must manually assign `n:history` tags to the BACnet control points. It may be more efficient to use a program object or robot editor to add the `n:history` tag and dynamically configure the tag value with the history ID of the applicable imported BACnet trend.

Associating definitions with tags

The **Definitions** container under the **AnalyticService** contains information types that identify the characteristics of the data to be analyzed. A set of pre-defined definitions are in the **`analytics-lib`** palette. You can create your own definitions. This topic explains how to modify a definition, how to copy another, and customize it to create a new definition. This work is done in Workbench running on a Supervisor platform.

Prerequisites: Workbench is connected to the station, the AnalyticService component is in the Config container, and the TagDictionaryService includes the Haystack set of tags.

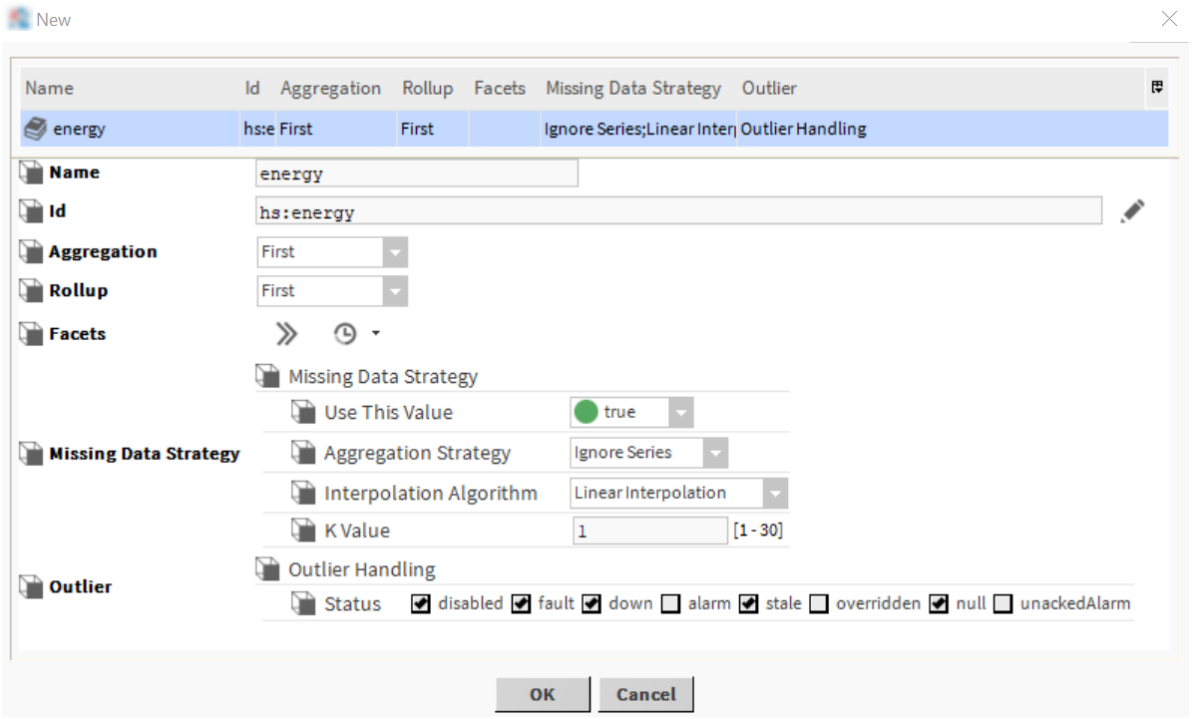
Step 1 To access the definitions, double-click **AnalyticService→Definitions**.

The **Analytic Data Manager** opens.

Step 2 Do one of the following:

- To create a new definition, click **New** at the bottom of the window.
- To edit an existing definition, double-click it in the Nav tree or **Analytic Data Manager**, or select it in the manager and click **Edit**.

The definition **Property Sheet** or **Edit** window opens.



The example shows definition properties as they appear in the **Property Sheet**.

Step 3 Configure the properties as needed.

The `Id` property associates the definition with a tag.

Changing the default behavior of a tag

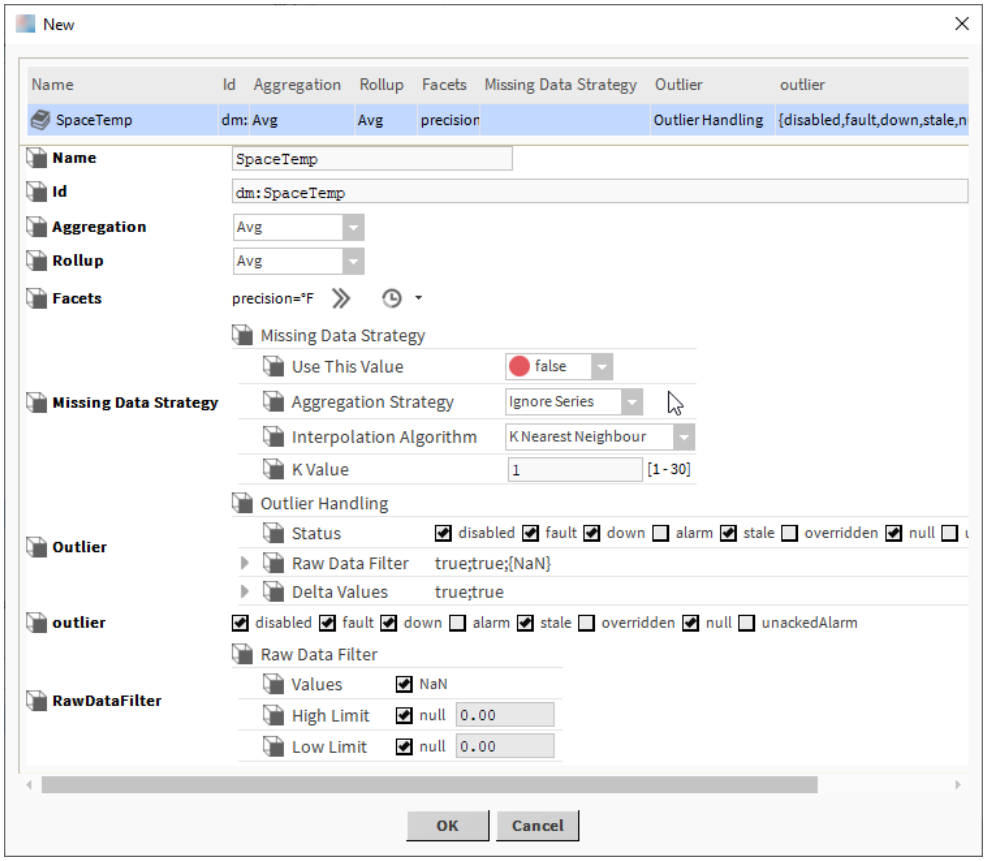
Using Data Definition components can simplify setup by configuring the typical properties (aggregation function, rollup function, unit and precision facets, missing data strategy and outlier handling) in one place, which you then apply as defaults through the application.

Step 1 Expand the **Config→Services→AnalyticService** container and double-click **Definitions**.

The **Analytic Data Manager** opens.

Step 2 To create a definition, click **New** at the bottom of the window or double-click a definition row to edit an existing definition.

The Edit window opens.



Step 3 Change the properties associated with the tag identified by the **Id** property.

You can set facets in the definition. This is important to ensure that all aggregated data use the same units and precision.

As with other such **Edit** windows, you can change the properties for more than one tag at the same time.

Tag inheritance and the a:a tag

The `a:a` marker tag identifies each point used by the framework. It is an origin entity. Entities that declare a tag are an instance of that tag. Think of this entity as having an "is a" relationship with the tag, while descendants of this entity have an "in a" relationship. This is a useful concept for algorithms.

When you add the `a:a` tag to a device or point in a Supervisor station, that device or point becomes an ancestor to similar points in remote stations. You can use these ancestors to find certain tags, such as `hs:zoneAirTempSensor` and potentially aggregate or access the information from the remote station.

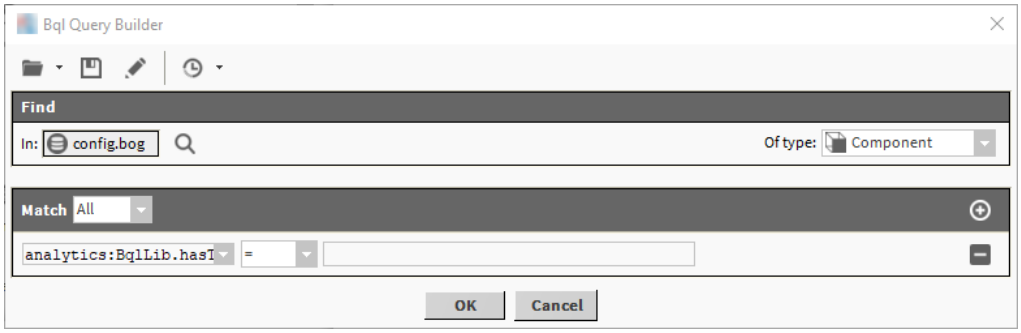
Hierarchies group tags hierarchically. Any device or point to which you assign a tag as part of a hierarchy can serve as an ancestor tag.

Removing all a:a tags

You can use the **ProgramService**'s **Batch Editor** to remove the `a:a` tag from all points used in an alert or algorithm.

Step 1 Open the **ProgramService** and click **Find Objects**.

The **Bql Query Builder** opens.



Step 2 To add a filter, on the **Match** row click the plus icon ( ) at the right end of the row.

A new row opens.

Step 3 Paste this syntax into the first field editor (left most) of the filter: `analytics:BqlLib.hasTag ('a:a').`

Step 4 Configure the equality operator (middle field editor) as equals (=) and click **OK**.

Step 5 Right-click **AnalyticService** and click **Actions→Rebuild Cache**.

Hierarchy setup

A hierarchy is a tree of level definitions, which identify the tags and NEQL queries to use when searching for data. Hierarchies provide meaningful relationships among data points. The same data accessed in different hierarchies can yield different analytical results.

Setting up a hierarchy is documented in the *Niagara Hierarchies Guide*.

Relationships

The relations feature provides the mechanism for structuring relationships among points in a hierarchy.

A relationship exists between two components. A parent component has one or more child relationships. The collection of data flows from one point to another based on the relationship between points. There are two types of relations:

- Direct relations are those that you apply (using relation markers) directly to a point. These relations are defined in a tag dictionary. The query configured by a relation definition on a hierarchy causes the system to return data.
- Implied relations are defined by tag rules in a Smart Tag Dictionary and applied automatically by the system. The query configured by a relation definition on a hierarchy causes the Smart Tag Dictionary to interpret the tag rules against the given point and return a list of implied relations.

You establish relationships in two places:

- By placing relations markers on points. These identify parent and child points.
- By adding relation level definitions to hierarchies set up where the system searches for data beyond the individual point.

Relations themselves may be tagged. The *Niagara Relations Guide* describes more fully how to set up relationships.

Chapter 3 Algorithms, alerts and alarms

Topics covered in this chapter

- ♦ DataSourceBlocks and calculations
- ♦ Creating an algorithm
- ♦ Creating an alert
- ♦ Viewing an alert in the alarm console
- ♦ Real-time request configuration
- ♦ Trend Interval defined in a binding
- ♦ Trend Interval defined in a proxy extension
- ♦ COV histories
- ♦ Best practices

An algorithm performs a calculation on real-time or historical (trend) data to generate a result. The result can trigger an alert or an alarm, be displayed on a chart, or can become an input to another calculation.

A single algorithm can run against data collected from an entire building. For example, assume your building has 100 air handling units and an algorithm to monitor their performance. When you add 50 more units, and tag each unit appropriately, without any additional effort on your part the original algorithm applies to all 150 units.

Using a poller, an alert runs an algorithm to monitor point performance. Alerts may trigger alarms, which appear on the normal alarm console. If the alert that triggered an alarm continues to exist, the alarm persists on the alarm console even after it has been acknowledged or force cleared.

The framework provides two uses for algorithms. You can use them to:

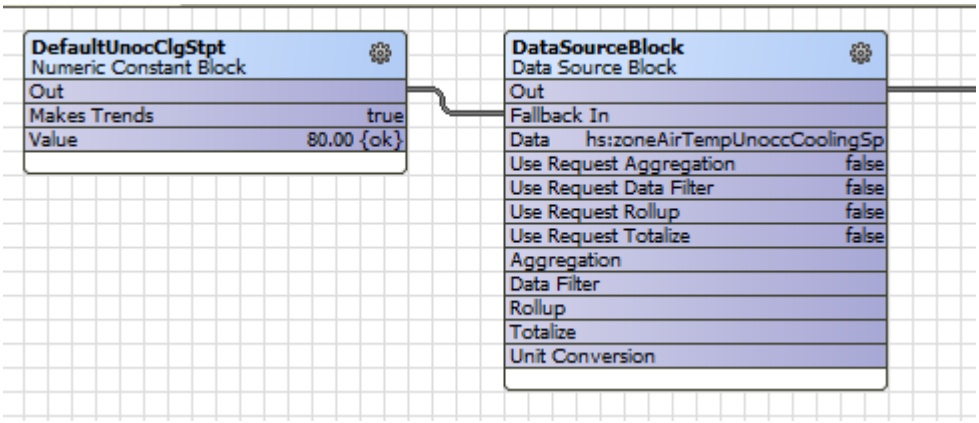
- Test for an individual condition. The alert running the algorithm can then automatically sound an alarm or trigger a remedy.
- Look back at the history of a point and report any events that meet defined criteria. For example, an algorithm can answer the question: Has there been any time in the past when a hot water valve was open more than 90% with a room temperature of three degrees below the setpoint for an hour? Depending on the amount of data (that is, how far back in the past the stored data exist), you may be able to identify a consistent pattern (a trend) that points to a condition requiring attention.

  NOTE: Although algorithms run on any platform, if you intend to process large volumes of historical data, consider running algorithms on a Supervisor platform.

DataSourceBlocks and calculations

The **DataSourceBlock** is the primary means to supply data to an algorithm. The block supports both value and trend requests. Logic blocks, of which there are many, provide the algorithm calculations. Each algorithm's **Wire Sheet** contains these blocks.

Figure 12 Example of a DataSourceBlock



The **Data** property in the block does not reference specific end points in the station by slot path or handle ORD, rather it references a type of data available in the station based on metadata, which are implied or based on direct tags or tag groups, or another algorithm.

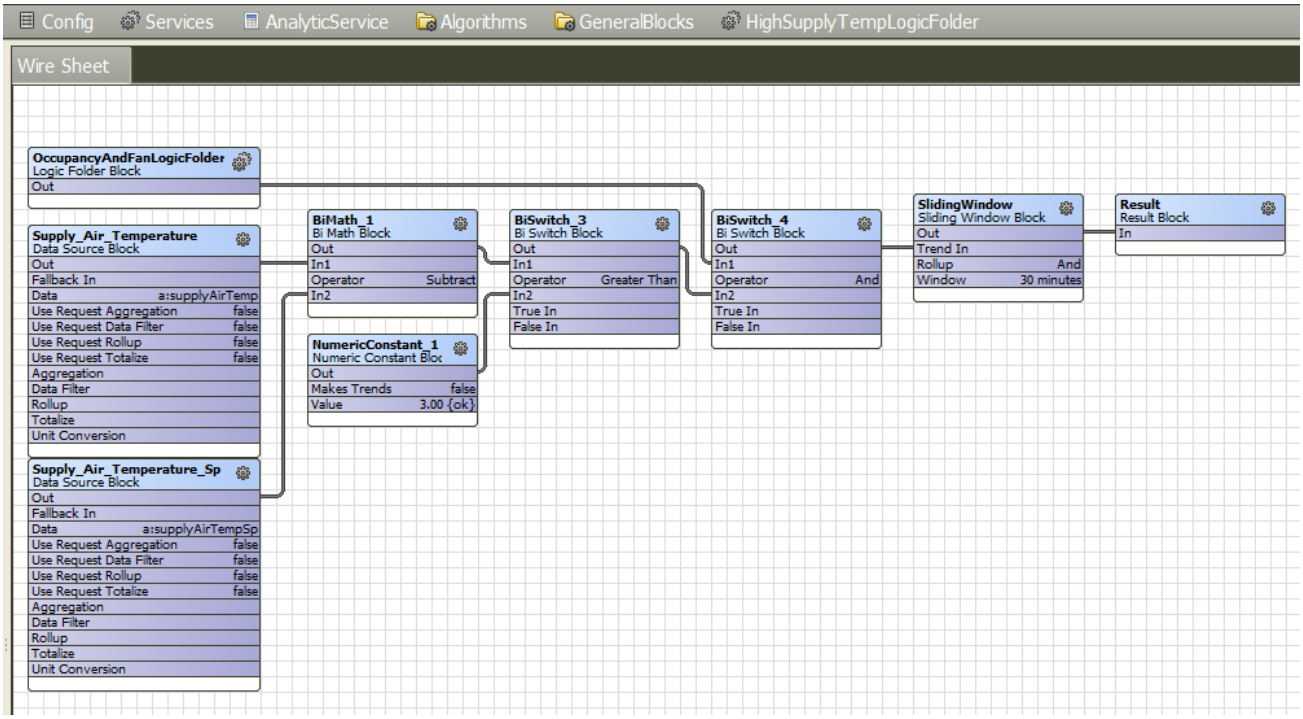
An edit icon on the right side of the **Data** property launches a **Select Data** field editor. You use this editor to configure the desired tag or algorithm for the block to use. You may use metadata tags, such as `hs:power` or `hs:zoneAirTempSensor`, to reference specific data in the station, or you may reference another algorithm from the station using the `alg:` prefix, such as `alg:CostCalculation`.

If the configured data are not available for the analytic request to process, the request uses the value from an object linked into the **Fallback In** slot as an alternate data source. This makes it possible to design an algorithm to use against components in the station that function the same but my not have identical data sources. For example, not all zones might have an unoccupied cooling setpoint (`hs:zoneAirTempUnoccCoolingSp`) so the **DataSourceBlock** may need to have a numeric constant block linked to the **Fallback In** slot. You may link the **Fallback In** slot from the output of other logic blocks, constant blocks, or another **DataSourceBlock**.

The framework processes the data in the **DataSourceBlock** using these properties, which you can configure:

- **Aggregation** combines the values from multiple data sources into a single value.
- **Rollup** combines records from a single data source into less granular records. This typically only applies to trend requests, but may also apply to value requests where the algorithm contains a block like Runtime or Sliding Window, which processes a trend request.
- **Totalize** applies to requests where the Data Source resolves to a history with ever increasing values tagged with the hs:hisTotalized tag.
- A **Data Filter** is an optional NEQL predicate used to configure data sources in the sub-tree of the node specified by the request. When a node meets the predicate, its sub-tree stops searching for additional values.

Figure 13 Logic Folder used to organize logic blocks



You may use a **Logic Folder** within the **Algorithms** component to organize these calculations. A logic folder is like an algorithm. It does not do anything itself, meaning that it contains no special code to perform calculations or evaluations, rather it executes all of the child blocks in its **Wire Sheet**, which must terminate in a final link to the **Logic Folder**'s **Result** block.

When you nest a **Logic Folder** in an algorithm, you may think of it as a **DataSourceBlock** and link its **Out** slot to other blocks in the algorithm.

Aggregation configuration

The **Aggregation** and **Use Request Aggregation** properties on the **DataSourceBlock** work together to configure the aggregation function and apply it to the data source. The aggregation function applies when the data source resolves to multiple components in the station and determines how to combine the multiple values.

When **Use Request Aggregation** is `false` and the **Aggregation** property is not configured on the **DataSourceBlock** (set to its default value of `First`), the **DataSourceBlock** uses the aggregation function as defined in the applicable **Data Definition**, unless there is no **Data Definition**, in which case the block uses the default `First` function. When **Use Request Aggregation** is `false` and the **Aggregation** property is configured, the block uses the **Aggregation** property's value.

When **Use Request Aggregation** is `true`, the block uses the **Aggregation** property defined in the request (analytic proxy extension, analytic binding, etc.), unless the request does not specify the **Aggregation** property, in which case the block uses the algorithm's **Aggregation** property value.

The following table demonstrates how the actual aggregation function used by an analytic model may vary depending on how you configure the **Use Request Aggregation** property on the **DataSourceBlock** along with the other **Aggregation** properties that are available for configuration.

Table 1 Aggregation configuration

DataSourceBlock		Data Definition	Request	Algorithm	Actual
Use Request Aggregation	**Aggregation**	**Aggregation**	**Aggregation**	**Aggregation**	aggregation function used by the algorithm
false	not configured	not configured	Sum	Max	First (default)
false	**Last**	not configured	Sum	Max	Last
false	Last	**Avg**	Sum	Max	Avg
true	Last	Avg	**Sum**	Max	Sum
true	Last	Avg	not configured	**Max**	Max

The bold options in the table highlight which option takes the priority based on the configuration choices in this example.

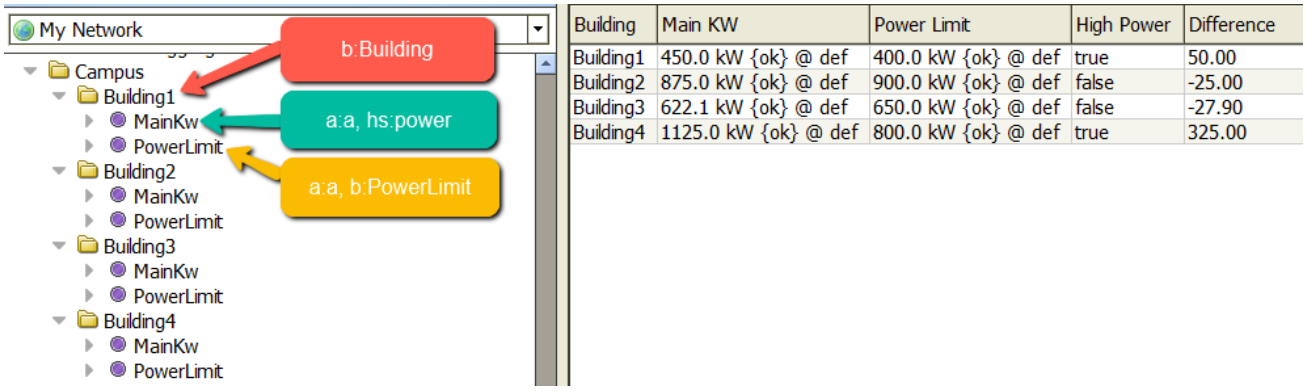
Example: data aggregation

One of the main benefits of using the framework is that you can create a single algorithm, which you can use many times in a station: in alerts, in control points with analytic proxy extensions and in Px views (widgets with various analytic bindings). This results in fewer components in the station (less heap memory usage) and minimizes redundant engineering (having to replicate the same **Wire Sheet** logic in many places in the same station).

In most cases, you design an algorithm that runs against a specific piece of equipment (AHU, VAV, FCU, boiler, chiller, electric meter, etc.), in which case, you are likely to have a single matching component for each data source in the algorithm. You may run that same algorithm against ancestor nodes in the station to show analytic results at a less granular level, such as for all equipment on a floor, all equipment in a building or all VAV zones associated with a specific AHU. When you run the algorithm against those ancestor nodes you must understand how the analytics engine aggregates the data. Typically, the default is to aggregate the data sources (inputs) to the algorithm and process the algorithm one time; however, this might not return the desired results.

Consider a tree of components in a folder named BuildingX (Building1 in this example):

Figure 14 Building example with tags identified



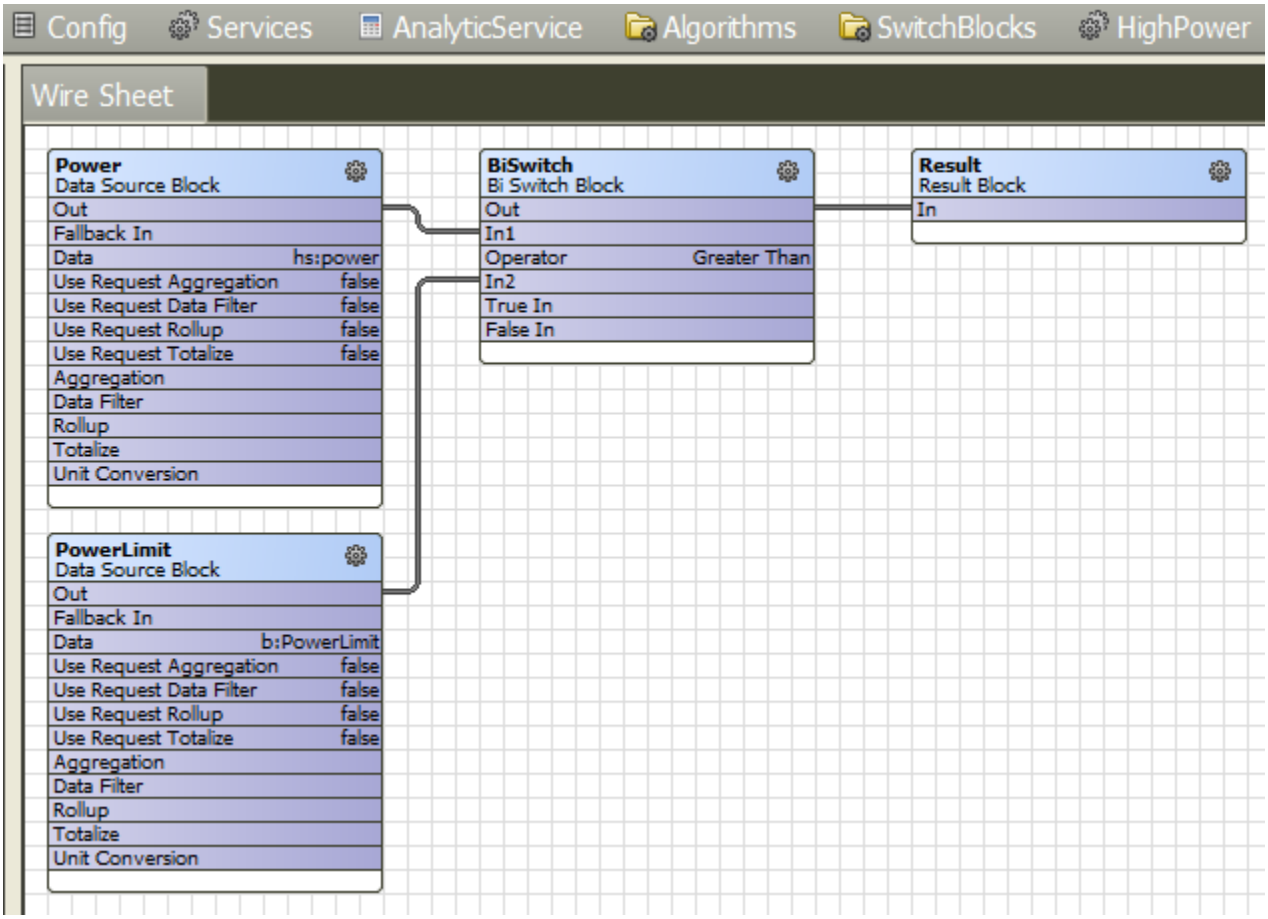
Building1 has the `b:Building` marker tag, and two numeric points:

- Point **MainKw** has `a:a` and `hs:power` marker tags.
- Point **PowerLimit** has `a:a` and `b:PowerLimit` marker tags.

The current values of the points are shown in the table to the right in the screen capture above.

The **Wire Sheet** for Building 1's algorithm is called **HighPower**:

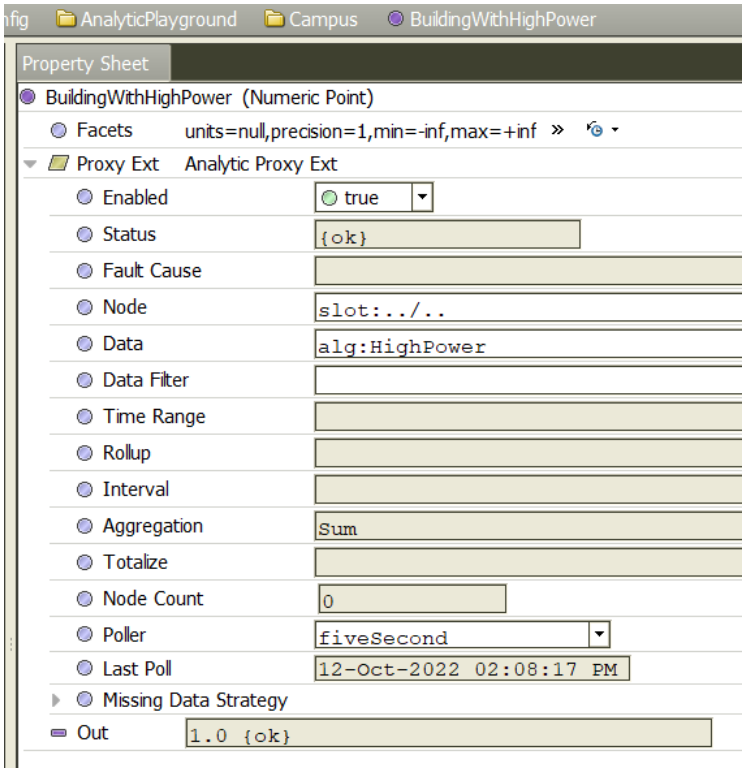
Figure 15 Building 1's HighPower algorithm



This algorithm has **DataSourceBlock**s for two inputs based on two tags associated with the points: `hs:power` and `b:PowerLimit`. The algorithm result returns `true` (value = 1) if **MainKw** > **PowerLimit**, otherwise it returns `false` (value = 0).

Extending this example, the customer might want to know in how many of their campus buildings they can currently detect a high-power fault? To answer this question, you would use a Numeric Point on an **Analytic Proxy Ext** to run the `alg:HighPower` algorithm with the **`Aggregation`** property configured as `Sum`. You would configure the property sheet for the extension's Numeric Point as follows:

Figure 16 Algorithm with Aggregation = Sum

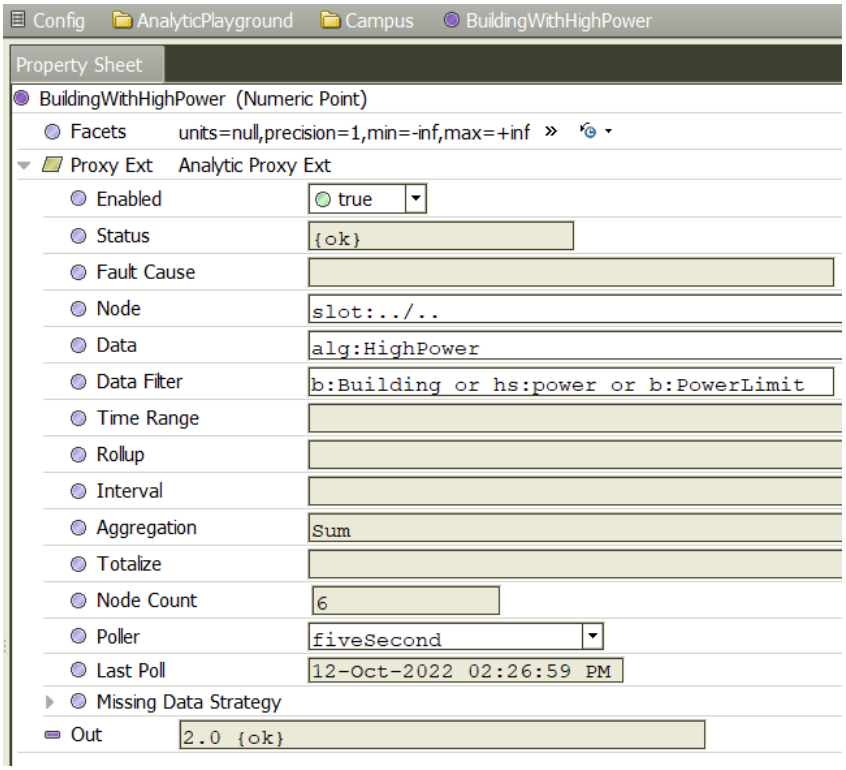


In this case, the result (**`Out`**) is `1.0` at the campus level because the **DataSourceBlock**s in the algorithm have their **`Use Request Aggregation`** properties set to `false` and their **`Aggregation`** properties set to null. With these settings, the algorithm defaults to the `First` aggregation function. Building1's **MainKw** point reports 450, which is greater than the building's **PowerLimit** of 400, so the result is true (a value of 1). This is not the desired result since two buildings currently have power values greater than the power limit.

The **DataSourceBlock**s in the algorithm could be configured with **`Aggregation`** = `Sum`, but that would sum all of the **MainKw** point values for a single power value and all of the **PowerLimit** values for a single setpoint and then process the algorithm one time. It is very unlikely that summing all of the point values and running the algorithm once will return the expected answer.

The BuildingX folders have a `b:Building` marker tag. To control where the algorithm is executed, the **`Data Filter`** property on the **Analytic Proxy Ext** can be configured with a NEQL predicate. Then, instead of aggregating all of the data sources (inputs) at the campus level and running the algorithm one time, each BuildingX folder can run the algorithm and the results of the algorithm (output true=1 or false=0) for each BuildingX folder can be summed together.

Figure 17 Algorithm with Data Filter configured



In this situation, the **Data Filter** NEQL predicate needs to be an 'or' statement that includes a tag to identify each data source required in the algorithm and a tag to identify the base components (nodes that are children, grandchildren, etc.). This gives the root node specified by the **Analytc Proxy Ext**'s **Node** property something to run the algorithm against. In this example, the algorithm runs against each of the four BuildingX folders and returns a value of `2.0` because Building1 and Building4 currently have high power levels. Widgets with analytic bindings also have a **Data Filter** property, which you can configure in the same fashion.

Rollup configuration

The **Rollup** and **Use Request Rollup** properties on the **DataSourceBlock** configure the rollup function and apply it when a trend request specifies an interval other than the interval used to collect the history data. Together they determine how to combine multiple sequential records in the history data into less granular intervals. For example, if the history **Interval** is `15 minutes` and the trend request **Interval** is set to `hour`, the rollup function combines the four 15-minute records from each hour into a single hourly result.

When the **Use Request Rollup** value on the **DataSourceBlock** is `false` and the **Rollup** property on the block is not configured, the **DataSourceBlock** uses the rollup function defined in the applicable **Data Definition**, unless there is no **Data Definition**, in which case the block uses the default `First` function. When the **Use Request Rollup** value is `false` and the **Rollup** property is configured, the block uses the **Rollup** property value.

When the **Use Request Rollup** value is `true`, the block uses the rollup function defined in the request (analytic proxy extension, analytic binding, etc.), unless the request does not specify the rollup function, in which case the block uses the algorithm's **Rollup** property value.

The following table is an example of how the actual rollup function used can vary depending on how you configure the **Rollup** properties.

Table 2 Rollup configuration

DataSourceBlock		Data Definition	Request	Algorithm	Actual
Use Request Rollup	**Rollup**	**Rollup**	**Rollup**	**Rollup**	rollup function used by the algorithm
false	not configured	not configured	Sum	Max	First (default)
false	**Last**	not configured	Sum	Max	Last
false	Last	**Avg**	Sum	Max	Avg
true	Last	Avg	**Sum**	Max	Sum
true	Last	Avg	not configured	**Max**	Max

Data filter configuration

A data filter is an optional NEQL predicate used to identify data sources in the sub-tree of the node specified in the request. When a node meets the predicate, its sub-tree no longer searches for additional values.

When **Use Request Data Filter** on a **DataSourceBlock** is `false` and the **Data Filter** property is configured, the block uses the value of the **Data Filter** property as defined in the **DataSourceBlock**.

When **Use Request Data Filter** is `true`, the block uses the value of the **Data Filter** property as defined in the request.

If **Data Filter** is not defined, the **DataSourceBlock** filters no data.

Totalize configuration

The **Totalize** property on the **DataSourceBlock** applies to requests where the data source resolves to a history with totalized (ever increasing) values and where the `hs:hisTotalized` tag is present. In general, when the **Totalize** property is `false`, the result returns delta values. When the **Totalize** property is `true` the result returns totalized values. There are some subtleties to be aware of.

When **Use Request Totalize** is `false` on the **DataSourceBlock**, and the block's **Totalize** property is configured, the block uses the value of the **Totalize** property. When the **Use Request Totalize** is `true`, the block uses the **Totalize** property as defined in the request. If the **Totalize** property is not defined, the block uses the default value of `true`.

The table demonstrates how the totalize function may or may not apply based on the configuration options.

Table 3 Totalize configuration

DataSourceBlock		Request	Actual
Use Request Totalize	**Totalize**	**Totalize**	totalize function
false	not configured	false	true (default)
false	**false**	true	false
true	true	**false**	false

Unit conversion

A data source supplied using a unit of measure that is not compatible with the result block's unit of measure requires a units conversion.

For example, a Numeric Point ZoneTemp with `hs:zoneAirTempSensor` and `a:a` marker tags is configured with units facets of temperature °F and precision 1. If its **DataSourceBlock Unit Conversion** property is

configured with temperature °C, the block converts a source value of 62.8 °F to an out value of 17.1 °C. The configured units must be convertible, otherwise the algorithm does not process.

Creating an algorithm

Algorithms are formulas created on a wire sheet that enable calculations, where the framework can combine real-time and historical data to produce values and analyze trends (time series).

Prerequisites: All objects are tagged, hierarchies created, and relationships established.

Step 1 In the station's Nav tree, expand **Config→Services→AnalyticService**, and double-click **Algorithms**.

The **Algorithm Manager** opens to the **Database** view.

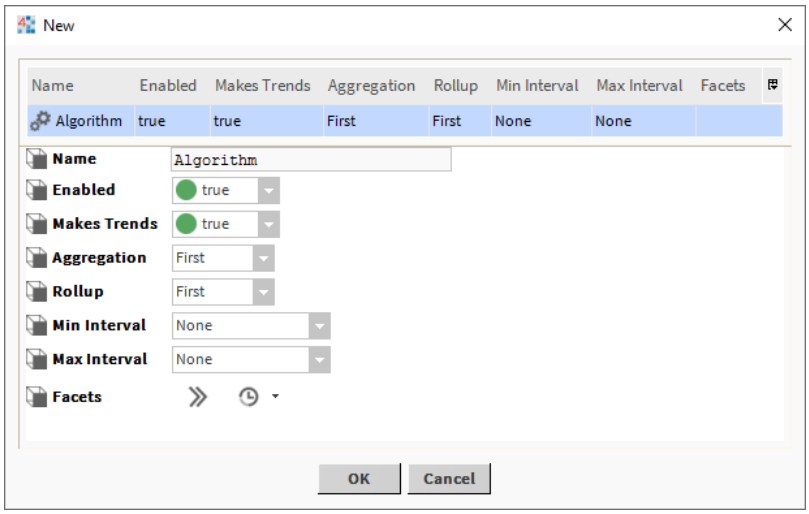
Step 2 To group algorithms, create a folder by clicking **New Folder** at the bottom of the manager and supplying a descriptive name.

Step 3 Do one of the following

a. To create a new algorithm, click **New**, select the number of algorithms to add, and click **OK**.

b. To edit an existing algorithm, select it in the **Database** table and click **Edit** (or double-click the row in the table).

The **New** or **Edit** window opens.



Step 4 Give the algorithm a name, make any changes to the default properties and click **OK**.

The framework requires data policies for these algorithm properties:

- **Aggregation** – defines how to combine current values from multiple data sources.
- **Rollup** – defines how to combine the multiple values contained in a single interval of a trend.
- **Min** and **Max** interval – some data may have limitations that establish the acceptable minimum and maximum period to use for rollups. For example, heat and cooling degree days should not have an interval of less than one day.

The algorithm and properties display as a row in the **Database** table.

Defining the data source

Tags and tag groups identify the data sources that feed an algorithm's calculations. Tags and tag groups replace the ORDs that in standard wire sheets (not algorithm wire sheets link blocks. A **Data Source Block** defines the tag that supplies the data for processing and charting.

Prerequisites: You are connected to a station.

A single algorithm can run against data collected from an entire building. For example, assume your building has 100 air handling units and an algorithm to monitor their performance. When you add 50 more units, and tag each unit appropriately, without any additional effort on your part the original algorithm applies to all 150 units.

Step 1 To open the **Algorithm Manager**, expand the **AnalyticService** and double-click **Algorithms**.

The **Algorithm Manager** opens.

Step 2 Do one of the following:

a. To start with a pre-configured algorithm from the library, open the **`analytics-lib`** palette, expand the **Algorithm** node and version (**English** or **Metric**), and drag an algorithm to the **Algorithm Manager**.

b. To build an algorithm from scratch, click the **New** button, click the **OK** button in the New window, in the subsequent window configure the **`Name`** and any other desired properties, then click the **OK** button.

Step 3 Right click the algorithm in the **Algorithm Manager** view and select **Views→AX Wire Sheet**, or double click the wire sheet icon ( ) for the algorithm in the Views column.

The **Wire Sheet** for the new algorithm opens. A new algorithm contains a single result block.

Step 4 For a new algorithm, open the **`analytics`** palette, drag a **DataSourceBlock** to the algorithm's **Wire Sheet**, and click **OK**. To open the data source property sheet, double-click the block and, if necessary, expand the window to see all properties.

You do not need to name a **Data Source Block** but it may provide context to other users when reviewing the algorithm.

Step 5 Go to the Property Sheet of the algorithm and click the **`Facets`** property (double-chevron icon in the field editor) and assign the appropriate unit of measure to use for output values.

No unit conversion from a **Data Source Block** occurs unless the algorithm defines its unit of measure. If algorithms are chained together, the next algorithm in the chain defaults to the previous algorithm's units. If no units are defined in the previous algorithm, the next algorithm defaults to the units defined in its **Data Source Block**.

Step 6 Configure any other properties and click **Save**.

Filtering algorithm input data

An algorithm has one or more **Data Source Block**s that have several properties (**`Data`**, **`Use Request Data Filter`**, **`Data Filter`**). The framework uses these properties to evaluate which components in the station have the required data available to process the algorithm. The **`Data`** property defines the required input using a tag or tag group. The **`Data Filter`** is an optional NEQL predicate used to identify data sources in the sub-tree of the node against which the algorithm is processing. This topic provides steps for configuring an algorithm's **Data Source Block** properties.

Prerequisites: All objects are appropriately tagged. Algorithms exist.

Step 1 To locate the desired algorithm, expand the **AnalyticsService** folder and double-click **Algorithms**.

Step 2 To open the algorithm **Wire Sheet**, right click the algorithm in the **Algorithm Manager** view and select **Views→AX Wire Sheet**, or double click the wire sheet icon ( ) for the algorithm in the Views column.

Step 3 To open the properties for the data source, double-click a **Data Source Block** logic block.

Step 4 To filter source input, configure one (or more) of the following **DataSourceBlock** properties:

a. Edit the **`Data`** property to identify a different source tag.

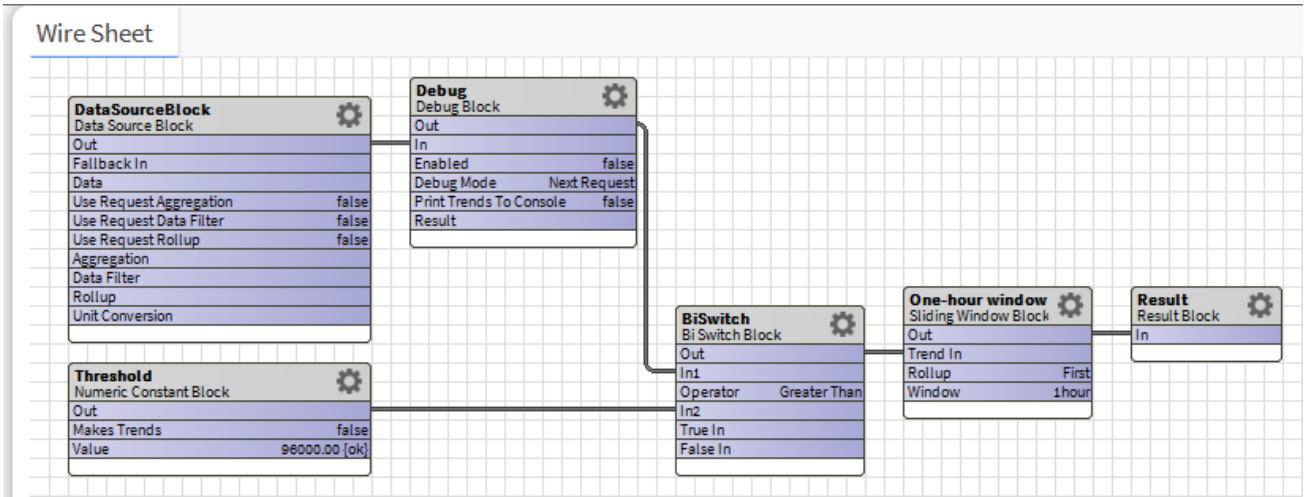
b. Edit the **`Data Filter`** property.

NOTE: Data filtering applies at the time an algorithm begins processing. The ability to filter is also available in Px bindings, however, if the DataSourceBlock Use Request Data Filter property is `false`, the framework ignores the **`Data Filter`** property in the binding.

Logic

You add logic to an algorithm by dragging or copying constants and blocks from the **`analytics`** palette to the **Wire Sheet**, configuring properties and linking the blocks.

Figure 18 Example algorithm



Each algorithm, typically, has at least one **DataSourceBlock**, which is linked to other algorithm blocks. Blocks are chained together with links from the output of one block to an input on the next block. It is easy to mis-connect an **`Out`** to an **`In`**. Make sure you connect from the **`Out`** slot, and not from another slot.

To open a block's **Property Sheet**, double-click the block. The*Niagara Analytics Framework Reference* and online Help system document the properties for each specific block. If you edit a property, click **Save**.

The **`UniMath`** and **`BiMath`** objects indicate the number of inputs. Something similar is true for the switches. You must connect all inputs required for each math block. For example, do not use a bi block if you have only one input.

To add comparison logic, drag a **`BiSwitch`** from the palette's **`Switches`** folder. Using this block you define the **`Operator`** (`Greater Than`, `Less Than`, etc.), then connect the appropriate **`Out`** slot(s) to it.

Some blocks provide actions. Right-click the block and select the action. For example, right-clicking a constant block assumes you want to set its value.

Using algorithm results in standard logic

The output from an algorithm can feed back into a station to optimize energy, reduce faults, or perform any other task based on historical and current conditions. For example, if an algorithm determines that a hot water valve is open while room temperature is high for an hour or more, standard **Wire Sheet** logic can close the hot water valve. This type of closed-loop analytics uses a control point with an Analytic Proxy Extension from the **Points** sub folder of the **`analytics`** palette..

Prerequisites: Standard logic already exists.

Step 1 Open the **`analytics`** palette and expand the **`Points`** folder.

Step 2 Go to the **Wire Sheet** view of a component in the station where you plan to configure the logic sequence.

Step 3 Drag an analytics **NumericPoint** to the **Wire Sheet**.

This action drags an object from the **analytics** palette to a non-analytics wire sheet, a powerful feature of the software.

Step 4 Right click the numeric pointand select **Views→AX Property Sheet** or **Views→Property Sheet**, then expand the proxy extension property.

A new Analytic Proxy Extension status is in fault because the **Data** property is not configured. The **Pollers** component under the **AnalyticService** includes a frozen property named **Default**, which is a cyclic poller configured for 5 minutes. A new Analytic Proxy Extension **Poller** property's default value maps to this default poller and may need to be configured to a dynamic, user-added poller.

Step 5 Configure the **Data** property to reference the desired tag, tag group or algorithm.

Step 6 Configure the **Node** property to reference the component in the station (config or hierarchy space).

The framework uses the value of the **Node** property as the base from which to search for the required data

Step 7 Configure the **Poller** property to specify how often the framework automatically processes the configured **Data** property and updates the control point's Out slot value.

When the **Poll** action is invoked either manually or automatically by the configured **Poller**, the framework updates the Analytic Proxy Extension's **Last Poll** property with the current station time.

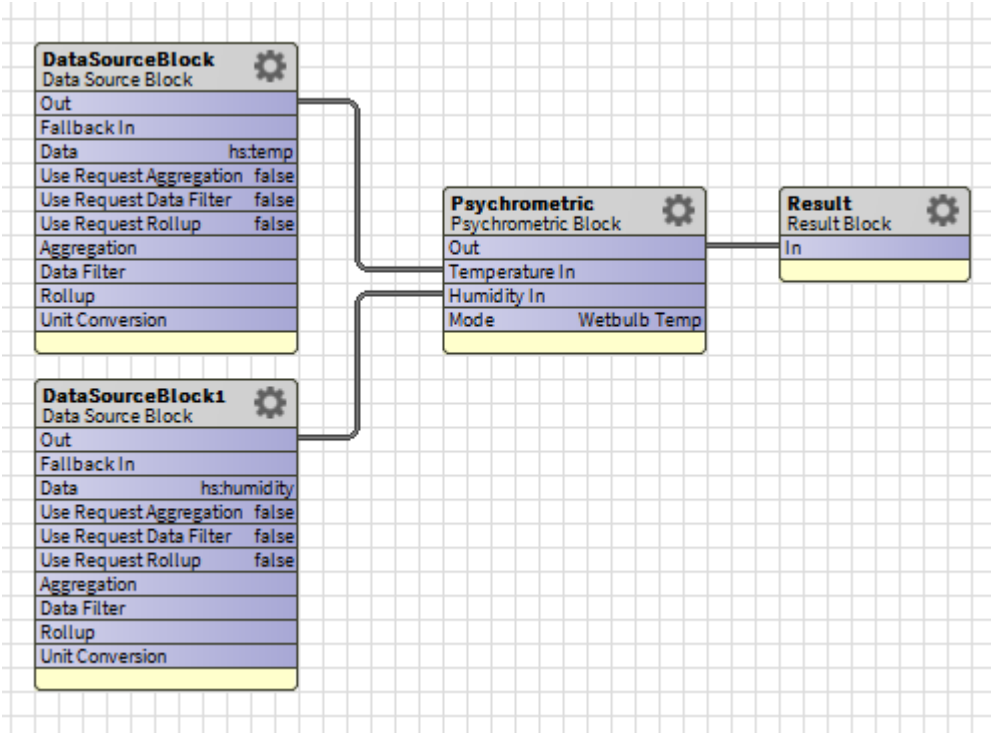
Example: Monitoring temperature and humidity

This task creates an algorithm to monitor temperature and humidity, including defining appropriate units, Metric or English.

Step 1 Create the algorithm, give it a name, and configure its properties.

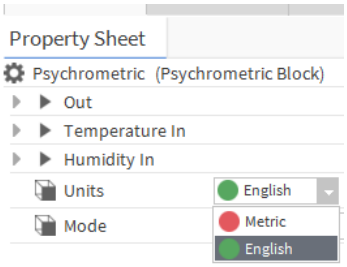
Step 2 With the algorithm in the Algorithm Manager, select the **Wire Sheet** view, open the **analytics** palette, and add logic, including two **DataSourceBlocks** and a **Psychrometric** block.

The algorithm should look something like this:



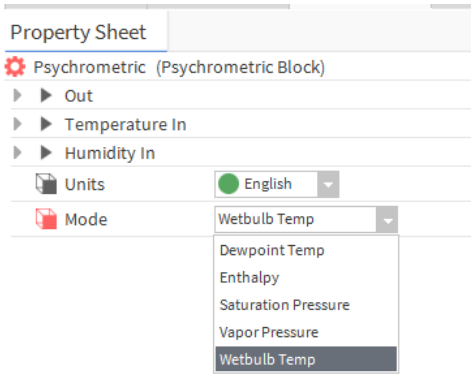
Step 3 Define the **Data** name when the Poll action is invoked either manually or automatically by the configured Poller for the first **DataSourceBlock** as `hs:temp`, and the **Data** property for the second DataSourceBlock as `hs:humidity`.

Step 4 Add a Psychometric block (under General Blocks) and select the units to use.



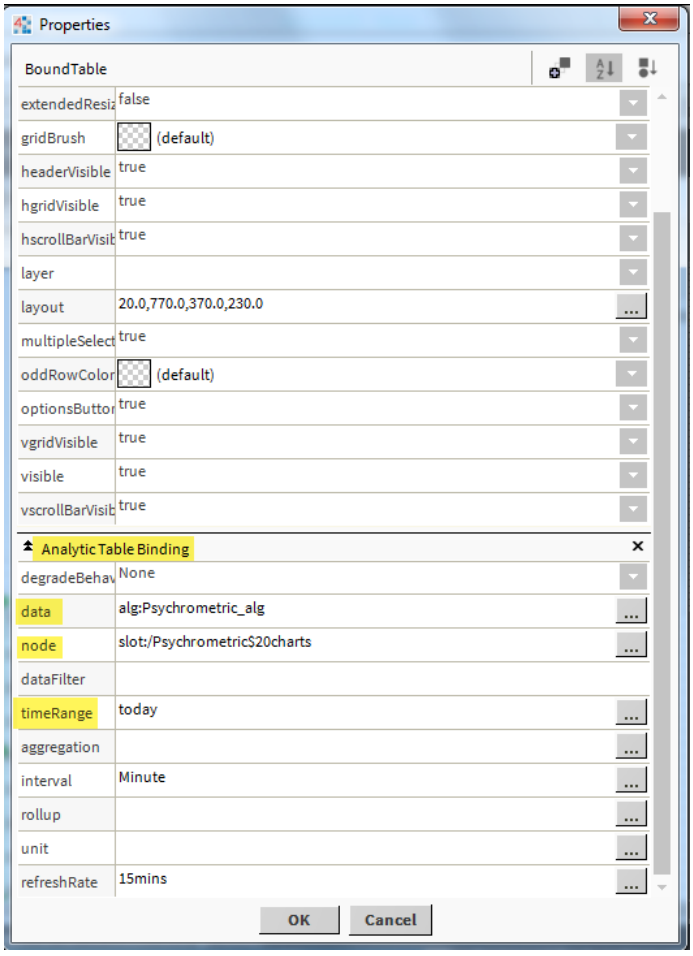
The system converts the raw data to the units you require. For information about Metric and English temperature values refer to the *Niagara Analytics Framework Reference* manual.

Step 5 Configure the **Psychrometric** block's **`Units`** property to the desired `English` or `Metric` value.



Step 6 Link the output and input slots, refresh cache, and save the station.

Step 7 To visualize the results, add a Bound Table with an Analytic Table Binding to a Px view.



Step 8 Configure the Analytic Table Binding with **`data`** = `alg:Psychrometric_alg`, set **`node`** to reference a component in the station with the required `hs:temp` and `hs:humidity` data available, and set **`timeRange`** = `today`.

The data display in a list.

Timestamp	Value	Status
30-Aug-17 11:03:00 AM IST	16.29	{ok}
30-Aug-17 11:04:00 AM IST	16.29	{ok}
30-Aug-17 11:05:00 AM IST	16.29	{ok}
30-Aug-17 11:06:00 AM IST	16.29	{ok}
30-Aug-17 11:07:00 AM IST	16.29	{ok}
30-Aug-17 11:08:00 AM IST	16.29	{ok}
30-Aug-17 11:09:00 AM IST	16.29	{ok}
30-Aug-17 11:10:00 AM IST	16.29	{ok}
30-Aug-17 11:11:00 AM IST	16.29	{ok}

Example: Removing unwanted data

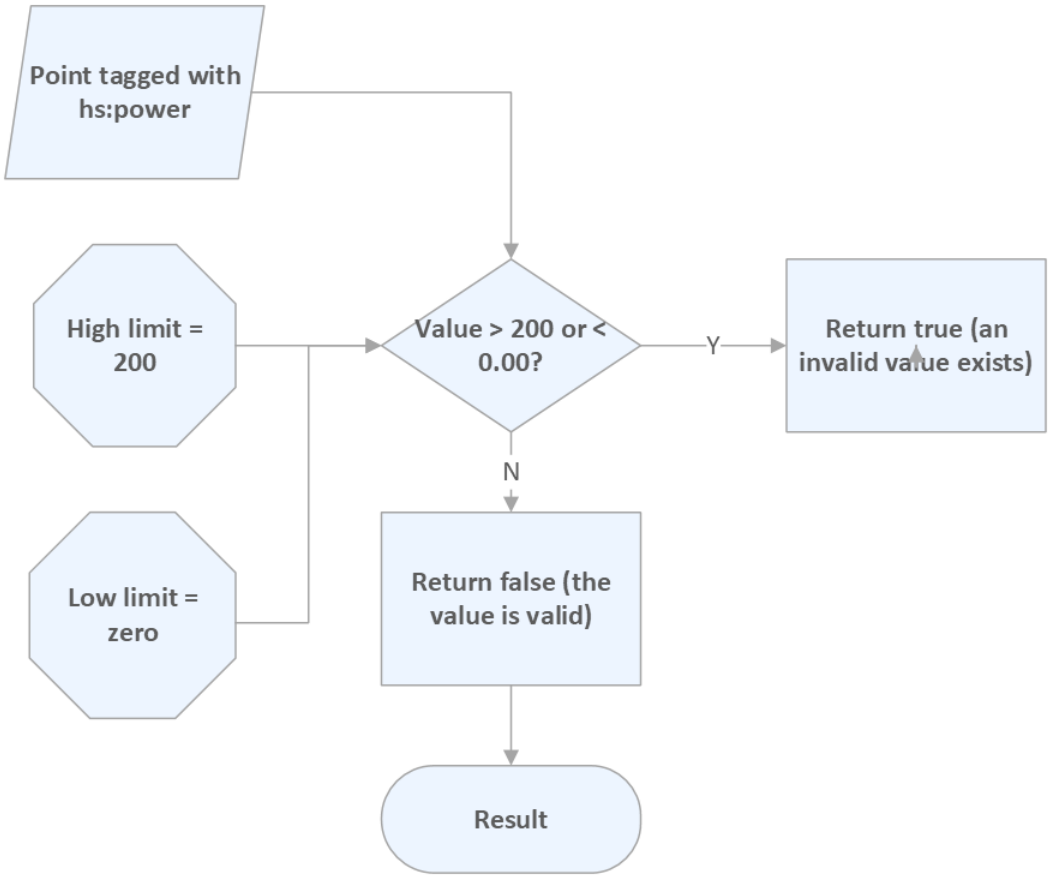
From time to time, a point does not report a plausible realistic value. For example, a value may get corrupted coming over the network. This erroneous value could be a very high or a negative number. Such data can render Px views and charts meaningless, especially if you are using aggregation or rollup, or if you are passing the data to another function. Typically, an organization may have to send the data outside of the

system to cleanse it or manually modify the histories. This topic documents two algorithms. The first filters out corrupt data. The second not only filters the data out, but also changes the data to a valid value.

Demand Range Filter Algorithm

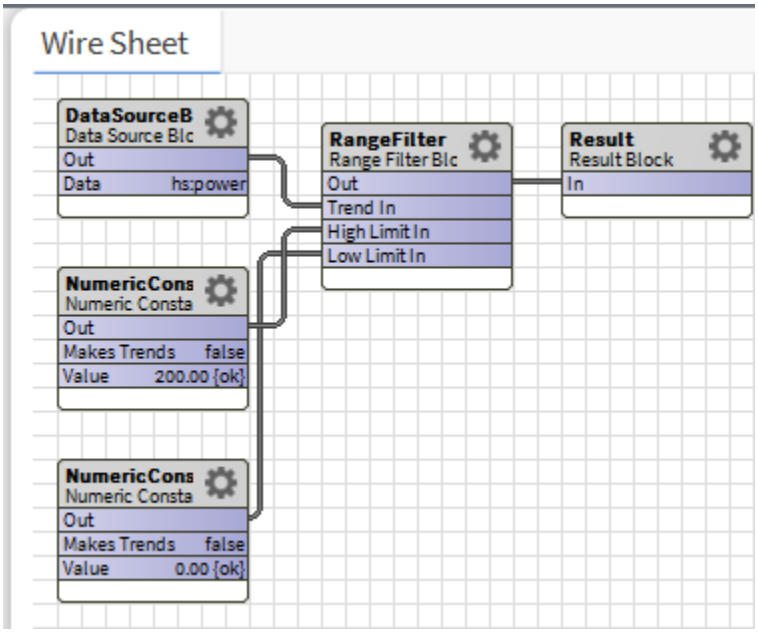
This algorithm determines if a value falls within a valid range.

Figure 19 Demand range filter algorithm flowchart



Wire sheet view

Figure 20 Demand range filter used to cleanse erroneous data

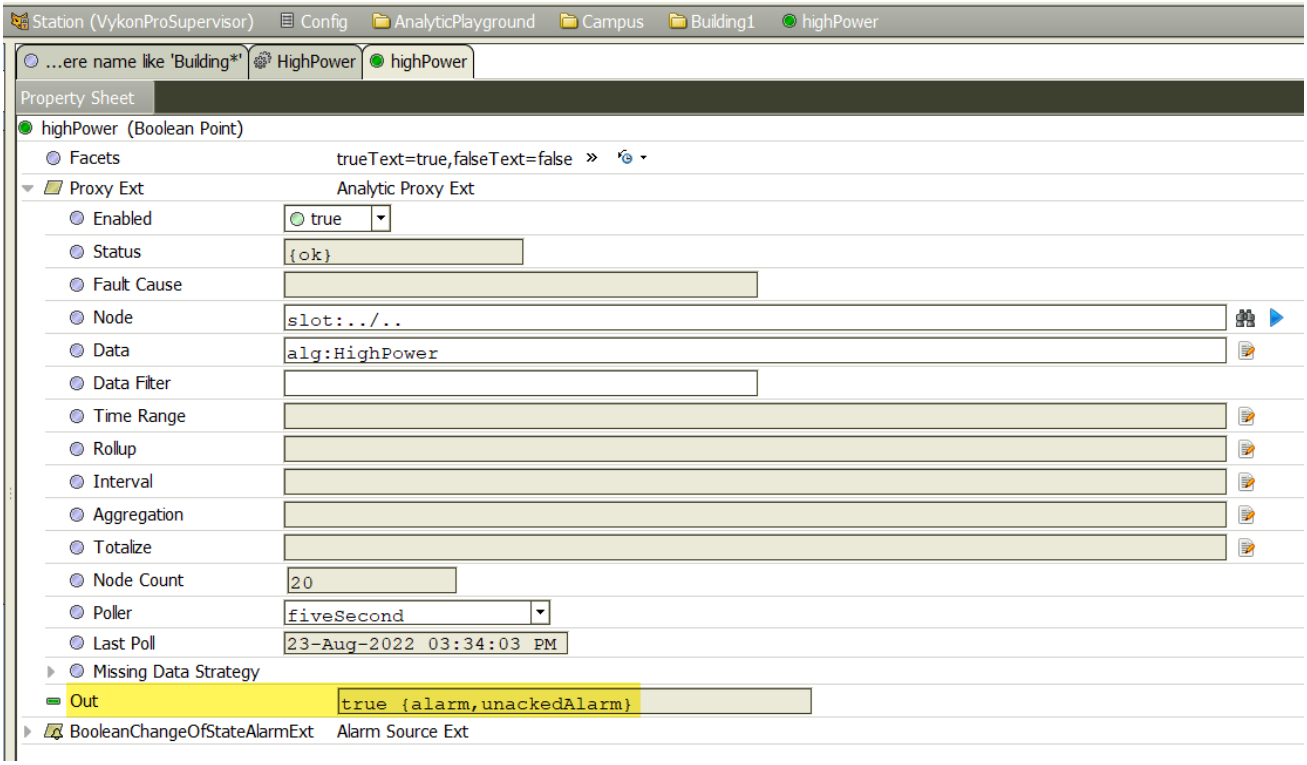


Example: Fault detection

This example demonstrates how to set up fault detection logic in a single source algorithm instead of having to replicate the fault detection logic in the **Wire Sheet** of every building.

Consider a BooleanPoint set up under each BuildingX folder with an analytic proxy extension to run the `alg:HighPower` algorithm at some frequency.

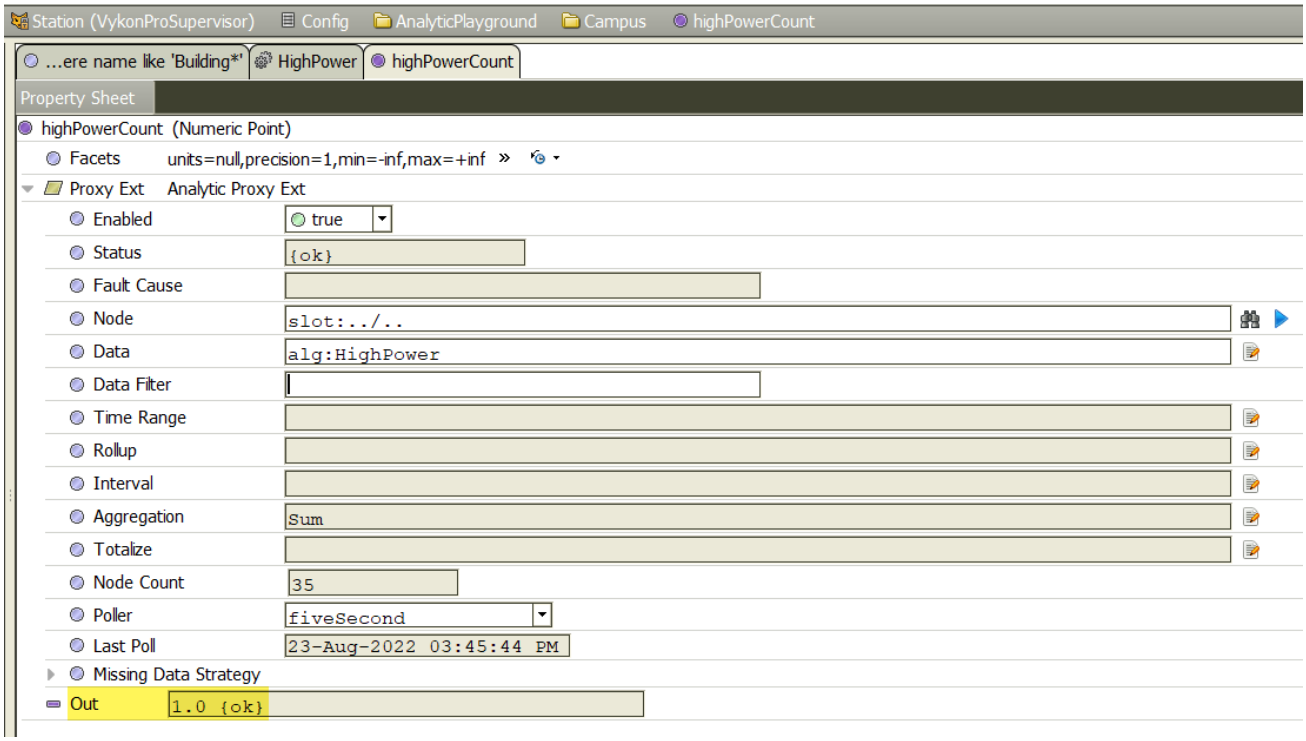
Figure 21 Proxy Extension set up to return an alarm



This configuration returns `true` in the **`Out`** slot, which indicates that an unacknowledged alarm condition for a specific Building (Bulidng 1 in this case) on the Campus..

To know how many buildings on your campus currently have a high power fault detection, you can add a **NumericPoint** to the Campus folder with an analytic proxy extension, run the same `alg:HighPower`, but this time with the **`Aggregation`** property set to `Sum`.

Figure 22 Proxy Extension configured with Aggregation set to Sum

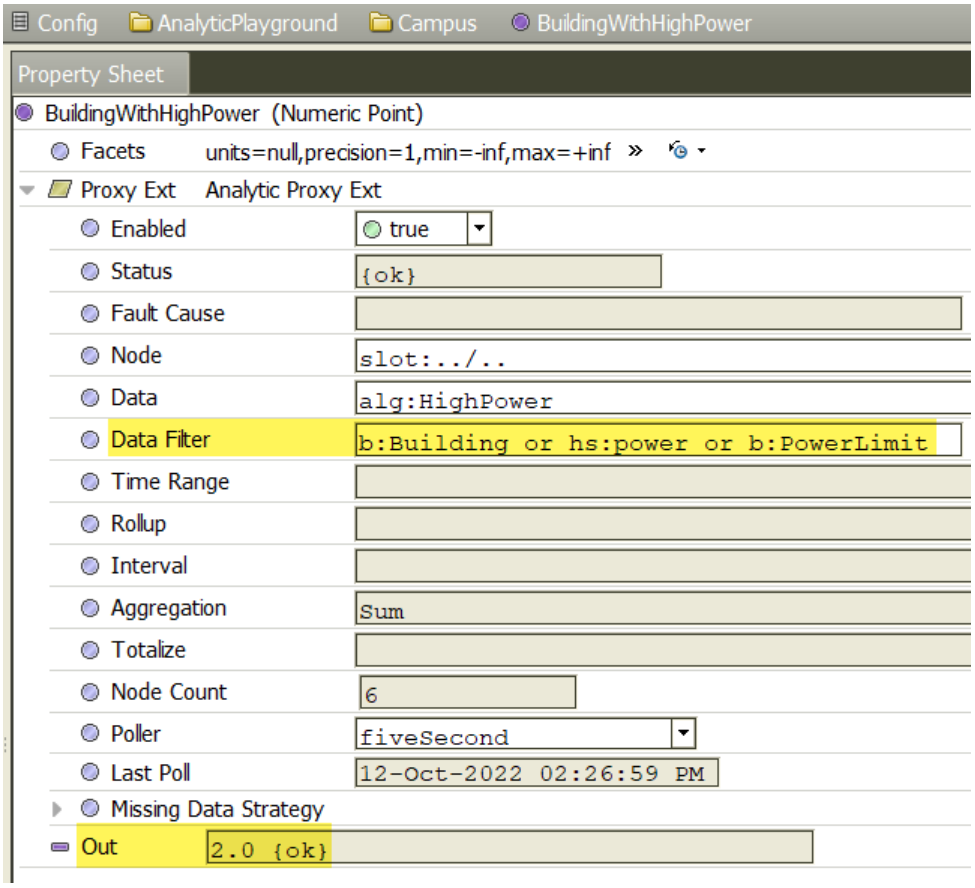


In this case, the algorithm returns a result of `1` at the campus level because the **DataSourceBlock**s in the algorithm have the **`Use Request Aggregation`** property set to `false` and the **`Aggregation`** property set to `null`. This configuration causes the algorithm to default to `First` for the aggregation function. If Building1 is in the first slot position and its MainKw value of 450.0 is greater than its PowerLimit of 400, the algorithm returns `true` (value of 1). The alarm condition exists.

The **DataSourceBlock**s in the algorithm could be configured with **`Aggregation`** = `Sum`, but that would sum all of the MainKw point values for a single power value, sum all of the PowerLimit values for a single setpoint and, then, process the algorithm one time. It is very unlikely that summing all of the point values and running the algorithm once will return the expected result.

The BuildingX folders have a `b:Building` marker tag. You can configure the **`Data Filter`** property on the analytic proxy extension with a NEQL predicate to control where the algorithm is executed. Instead of aggregating all of the data sources (inputs) at the campus level and running the algorithm one time, you can configure the algorithm to run at each BuildingX folder and sum the results of the algorithm (output true=1 or false=0) together.

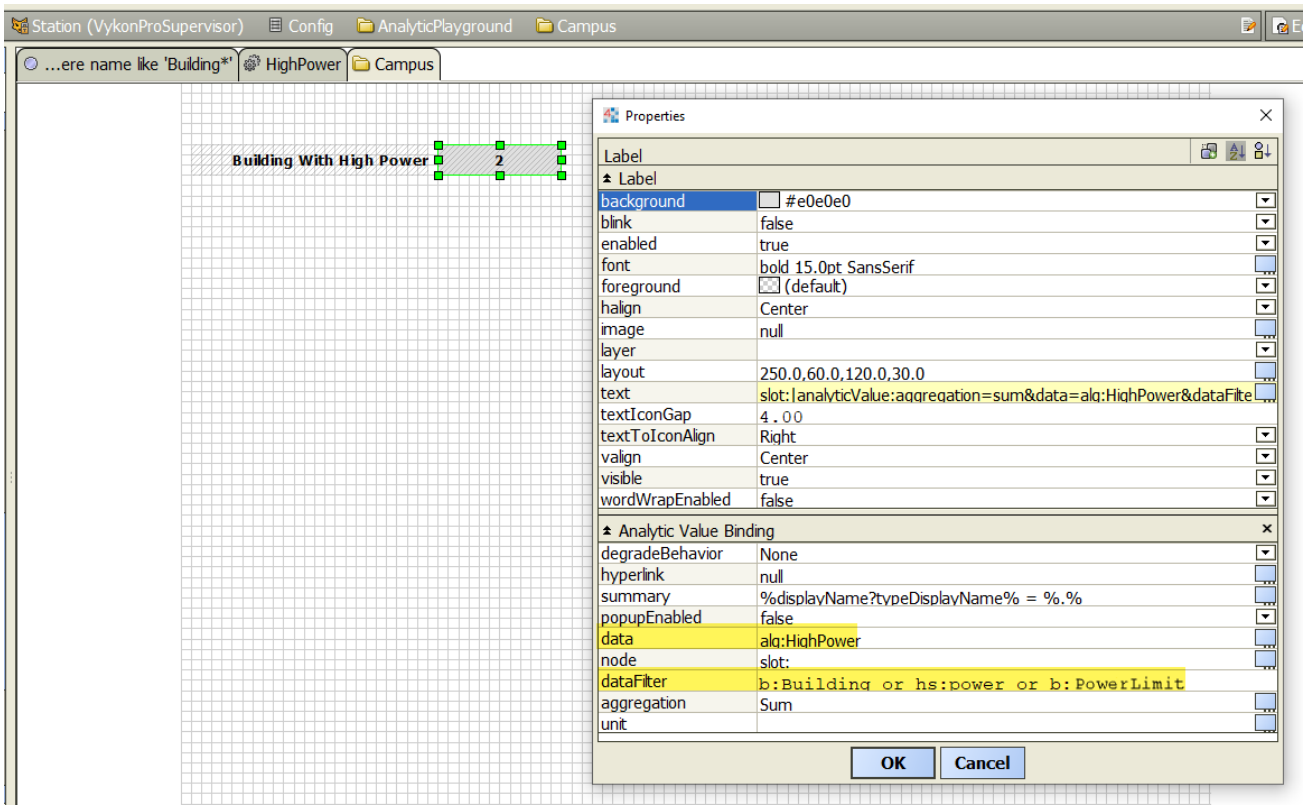
Figure 23 Use of a NEQL predicate to control algorithm execution



This **`Data Filter`** NEQL predicate needs to be an 'or' statement, which includes a tag that identifies each data source and a tag that identifies the base components (nodes that are children, grandchildren, etc. under the root node specified by the analytic proxy extension) to run the algorithm against. In this case, the algorithm runs against each of the four BuildingX folders and returns a value of `2` because Building1 and Building4 currently have high power levels.

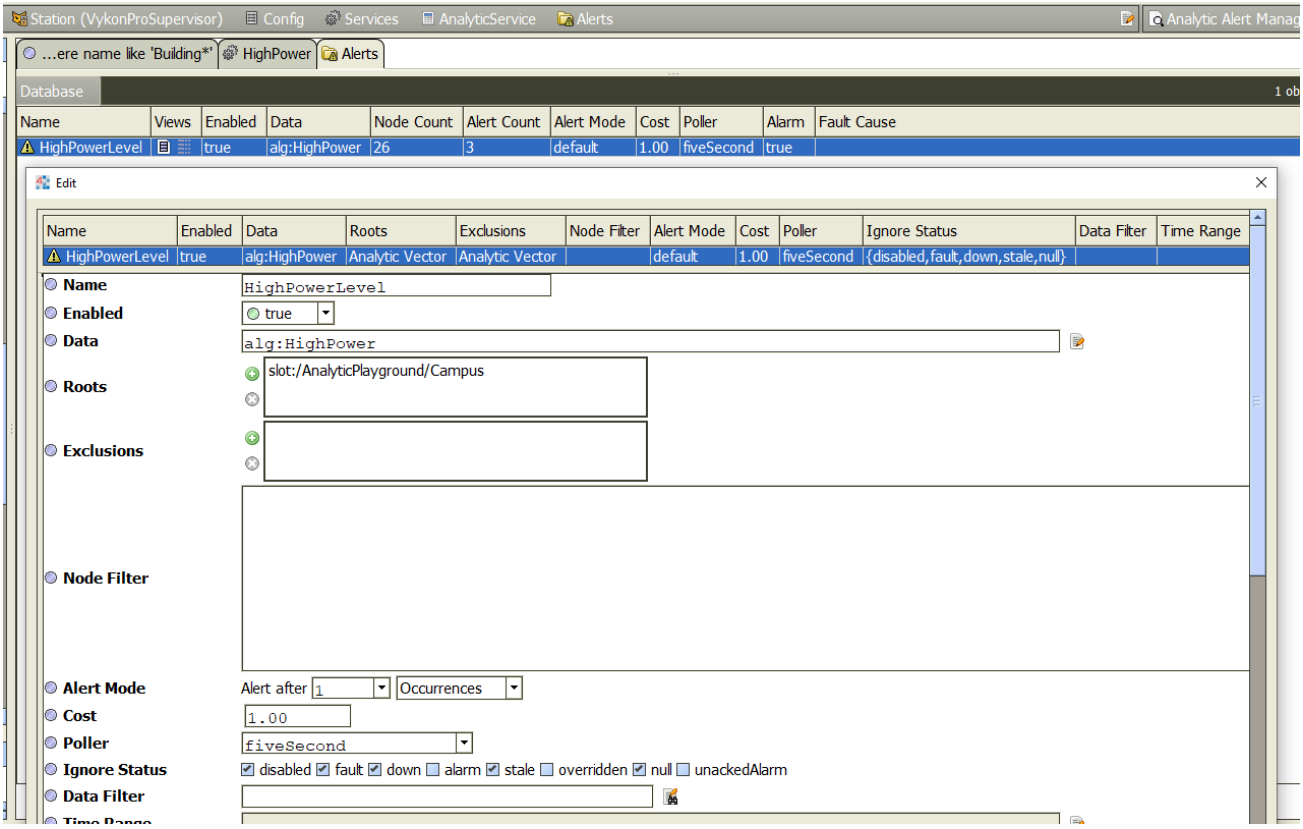
Widgets with analytic bindings also have a **`Data Filter`** property, which you can configure in the same fashion.

Figure 24 Binding with its Data Filter configured



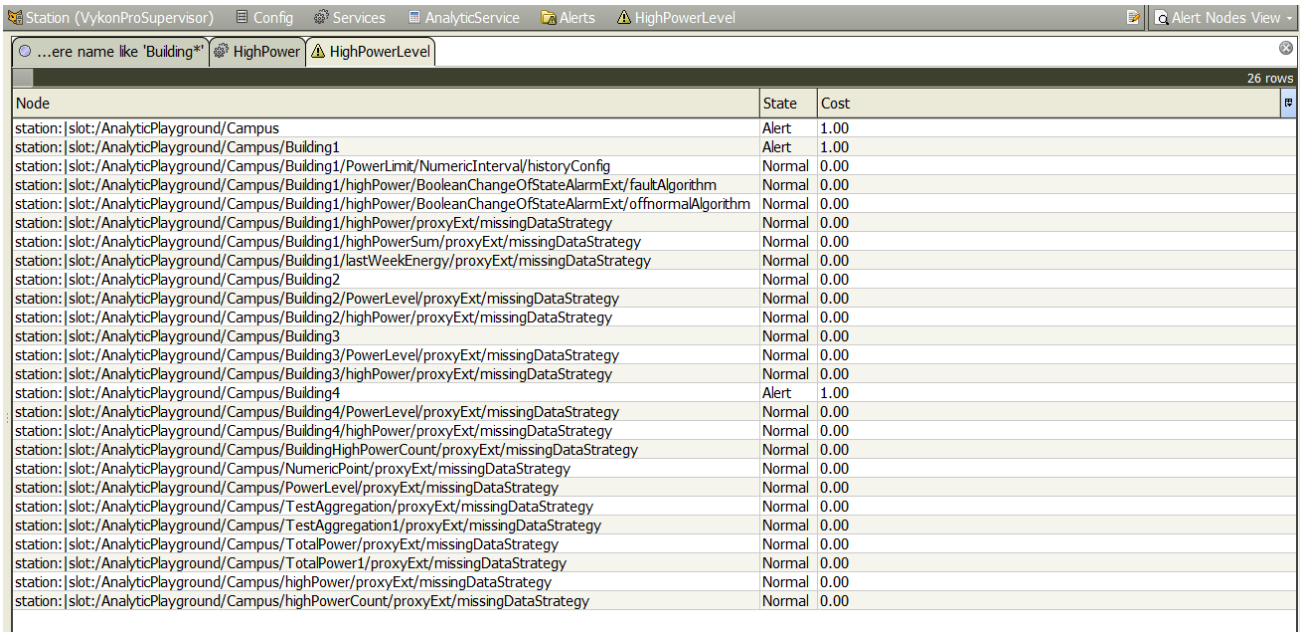
Alerts also have a **`Data Filter`** property, but it may not be used in the same way as the same property does in an analytic proxy extension and analytic binding. Consider an alert that is configured to run the same `alg:HighPower` against the Campus folder configured with the **`Roots`** property.

Figure 25 Analytic alert configured with the Roots property



This results in the following **Alerts Node View**.

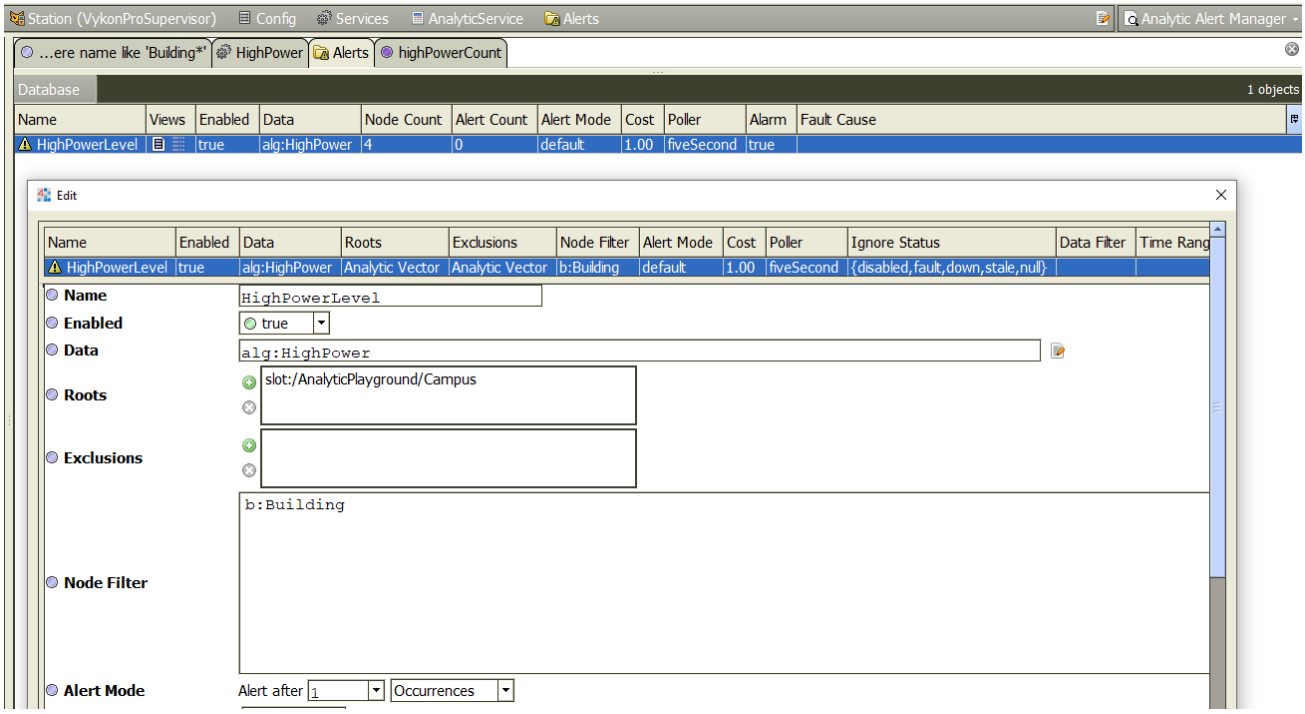
Figure 26 Alerts Node View



This view shows that the algorithm is being run against many nodes under the Campus folder. It is likely that you only want the alert to be run against the BuildingX folders.

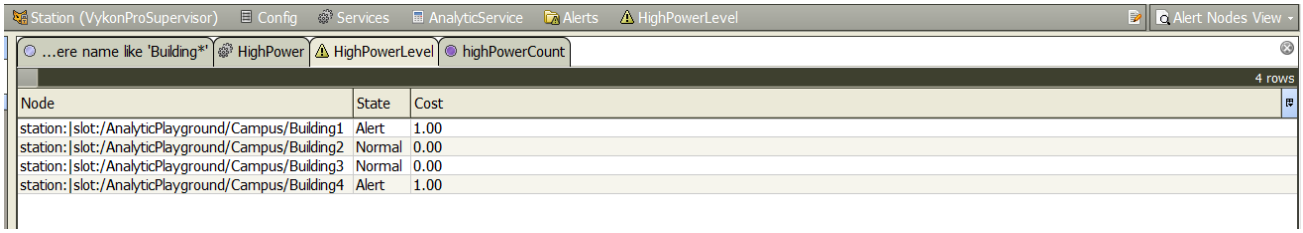
The alert has a **`Node Filter`** property, which also accepts a NEQL predicate, such as `b:Building` with which to configure the alert so that it only runs the algorithm against a node with the `b:Building` marker tag.

Figure 27 Alert configured with a Node Filter



Now the **Alert Nodes View** shows the algorithm only running against each of the BuildingX folders.

Figure 28 Alert Nodes View after running against a limited number of nodes



Creating an alert

An alert has a **`Data`** property the same as does an analytic proxy extension and analytic binding, which you can technically configure with a tag or tag group; however, the intention of an alert is to configure the **`Data`** property to reference an algorithm, which has some fault detection logic that returns a Boolean value. The Alert's **`Poller`** property configures how often the Alert automatically processes the configured algorithm. This updates the applicable properties on the Alert component. When the result of processing the algorithm returns a true value, an alert condition exists. The **Analytic Alert Manager** view on the Alerts component summarizes the current state of all alerts. The **Alert Nodes View** on an individual alert provides the details for each alert.

You may also configure an alert to generate an alarm record, Niagara, which routes through normal mechanisms (console recipients, email recipients, etc.). If actual alarm records are needed it is likely better to configure Boolean points with an analytic proxy wxtension to process the desired fault detection algorithm, and use a standard alarm extension on the Boolean point.

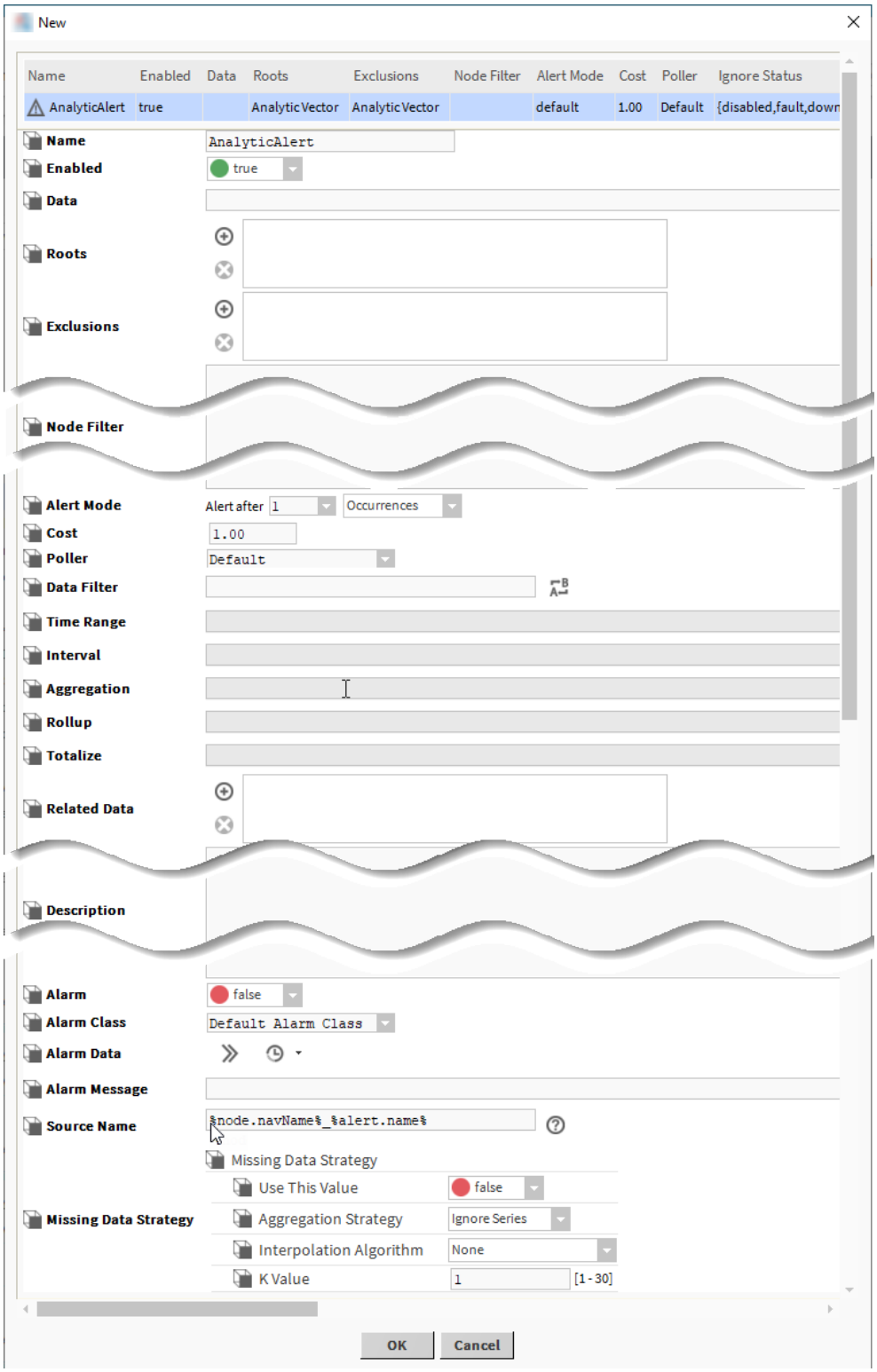
Step 1 Double-click **AnalyticService→Alerts**.

The **Analytic Alert Manager** opens.

Step 2 Do one of the following:

- To create a new alert, click **New**.
- To modify an existing alert, select the alert and click **Edit** or double-click the alert in the row.

The **New** or **Edit** window opens.



Pay special attention to these properties:

- **`Data`** defines the algorithm that returns true or false. When true, the alert exists; when false, no alert exists.

- **Roots** defines where to collecting data for the alert. This alert runs against all devices under the `Drivers/BacnetNetwork/Global` folder.
- **Node Filter** reduces the number of nodes to search. `n:device` limits alerts to devices. This is an appropriate place to use a NEQL predicate.
- **Alert Mode** provides two settings. **Alert after**, configures when to generate the alert. The second drop-down list identifies how many of what to use to generate the alert.

  `Occurrences` generates an alert after the selected number of the same event occurs.

  `Seconds`, `Minutes` and `Hours` define how much continuous time must elapse where the algorithm result is true for an alert to be generated. This value is part of the total cost calculation.
- **Cost** associates a currency value with each alert occurrence.
- **Poller** defines how frequently to run the alert.
- **Alarm** controls alarm generation. The default is `false` (no alarm generation).

If you are testing, consider creating an Analytics Smart Alarm Class and send all framework alarms to this class as an easy way to differentiate between alarms coming from a standard alarms extension and the alarms coming from the framework.

The rest of the properties are described in the *Niagara Analytics Framework Reference*.

Step 3 Configure properties and click **OK**.

Step 4 To complete the configuration, right-click the **AnalyticService** and click **Actions→Refresh Cache**.

This step is required any time you create a new or edit an existing alert.

Step 5 After creating the alert, double-click it in the Nav tree.

The **Alert Nodes View** opens.

This view displays the current state of all points monitored by the alert.

Step 6 If the view is empty, right-click the **AnalyticService** and click **Actions→Refresh Cache Full**

If you still do not see the points in the table, click **Refresh** at the bottom of the window.

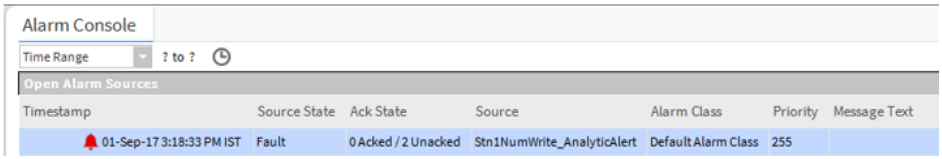
Viewing an alert in the alarm console

Alerts can generate alarms, which appear in the alarm console.

Prerequisites: An alert has been configured to display the source node.

Step 1 Access the alarm console.

The alarm console opens.

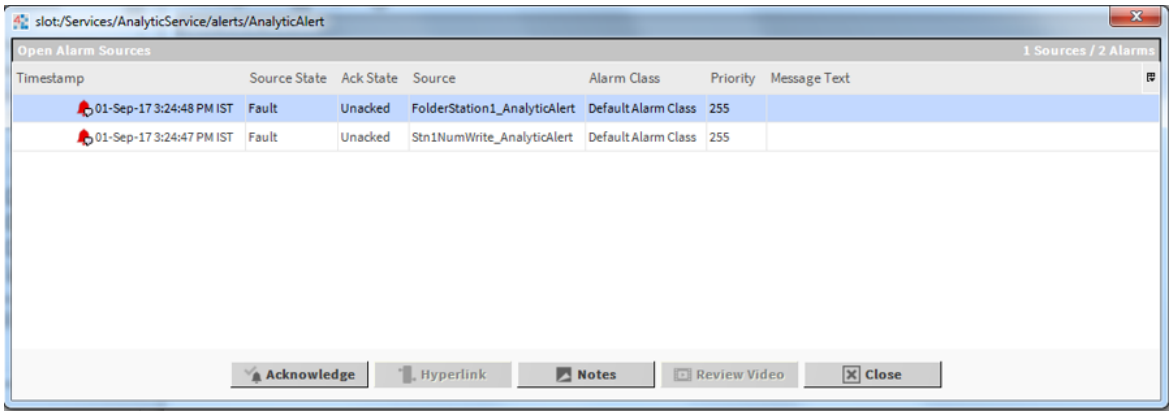


The screen capture shows an alarm that was generated by an alert. The alarm console's Source column identifies the source of the alert: `FolderStation1` as a node in the Nav tree and `AnalyticAlert` describes what happened. The Message Text column displays any notes associated with the alert.

If the alert was defined incorrectly, the default BFormat (%node.navName%_%alert.name%) displays in the Source column instead of the alert source information.

Step 2 Double-click the alarm record generated by the alert.

The **Open Alarm Sources** view opens.



This view filters the alarms to display only alarms generated by alerts.

Step 3 To display additional information or to add your own comments, select the alarm row and click the **Notes** button.

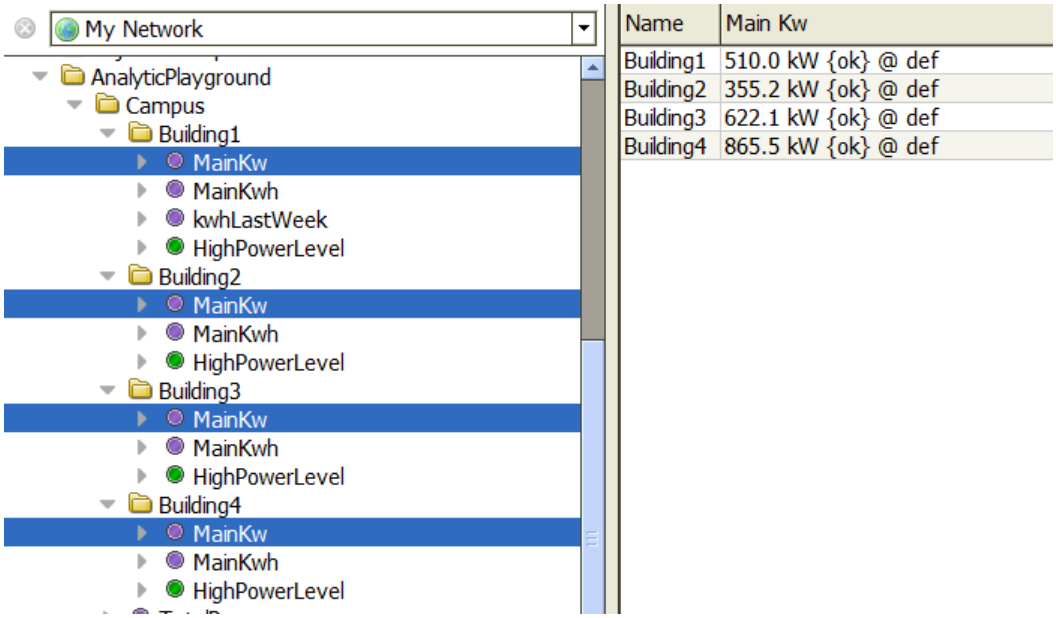
Real-time request configuration

The framework provides complex yet flexible methods to configure the application, which may sometimes give you seemingly unexpected results It is important to understand the various ways properties in an analytic request might be set or overridden. These examples set up a real-time aggregation in different ways to demonstrate how the framework evaluates settings.

Each property, including **`Time Range`**, **`Rollup`**, **`Interval`**, **`Aggregation`**, **`Totalize`** and **`Missing Data Strategy`** has a default setting, which the framework uses if you do not explicitly set the property.

Consider the following station real-time configuration where there are four BFolder components named BuildingX each with a **`NumericPoint`** named MainKw. The MainKw points have the `a:a` and `hs:power` marker tags. Each point has returned a different value.

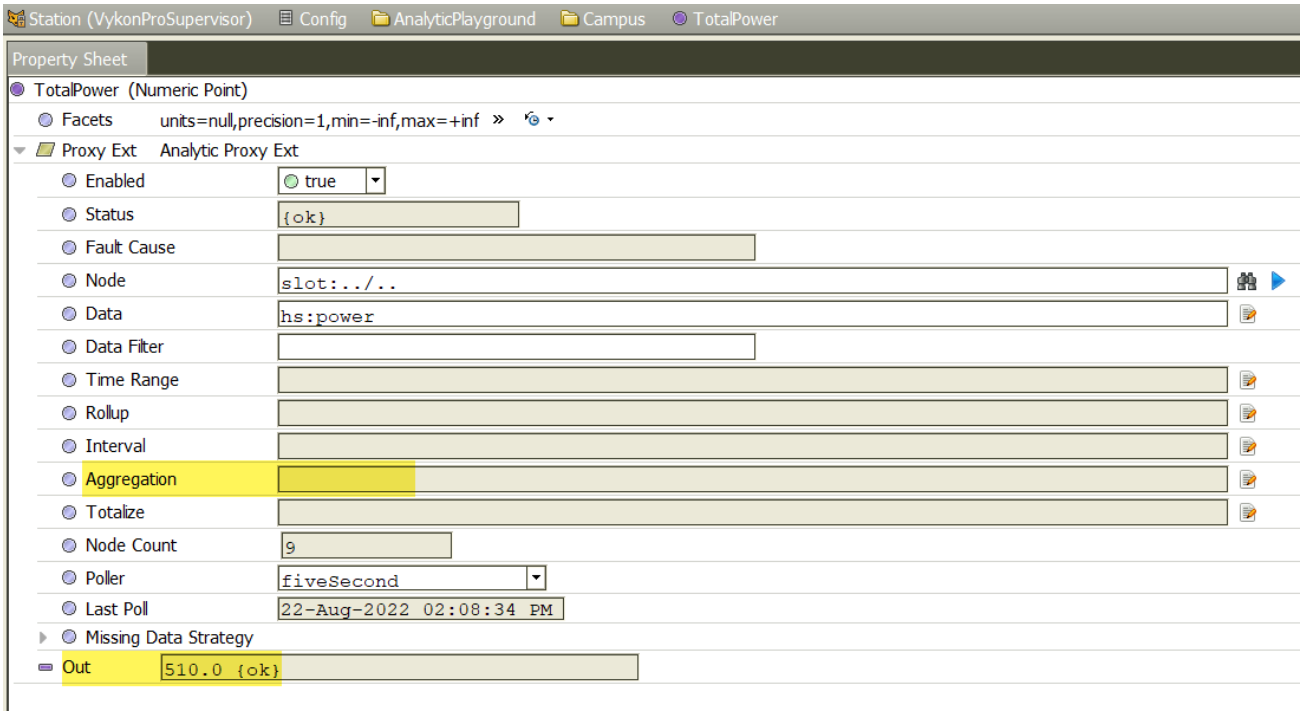
Figure 29 Example of four slots to aggregate



There are a number of ways to submit an analytic request, such as using a control point with an analytic proxy extension in the station, a label in a Px with an analytic value binding, a data definition, or an algorithm with a **Data Source Block**.

Aggregation combines multiple data sources into a single result using some function like `Sum`, `Avg` (average), `Min` (minimum), `Max` (maximum), etc. The **`Aggregation`** property on the **Proxy Ext** or binding defines the function to use.

Figure 30 Default result

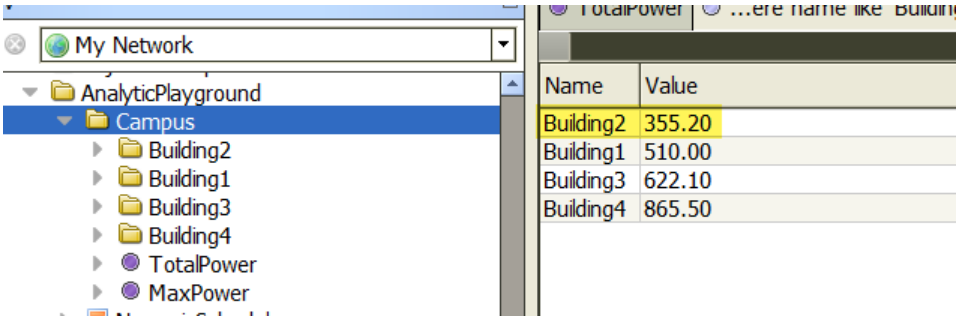


The example above does not explicitly set the **`Aggregation`** property, and it is not obvious from looking at this property sheet, that if you do not configure the **`Aggregation`** property, it defaults to `First`, which returns the first value in the combination.

When the framework resolves this example request, it returns an **`Out`** value of `510.0` kw, which is the value reported by the first data source (Building1), that is, the first component in slot sheet order.

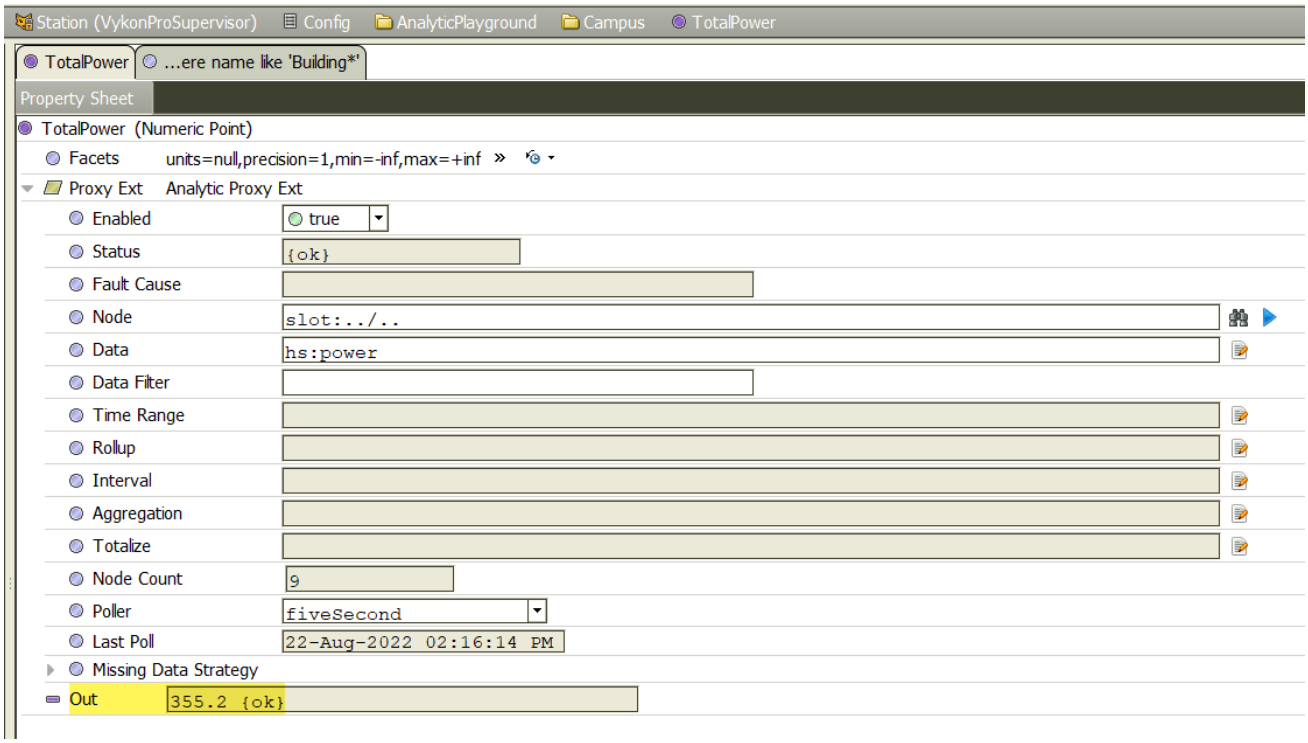
Reordering the BuildingX folders under the Campus folder so that Building2 is now before Building1 in the slot sheet changes the result.

Figure 31 Reordered slots



After invoking the refresh cache action on the **AnalyticService**, and without any change to the **`Aggregation`** property on the **Proxy Ext**, the aggregation result is now `355.2` KW because Building2's point with the `hs:power` tag is now the first slot on the Campus slot sheet.

Figure 32 Result after reordering the slots



TIP: Explicitly configuring each configuration property based on your requirements is a good idea.

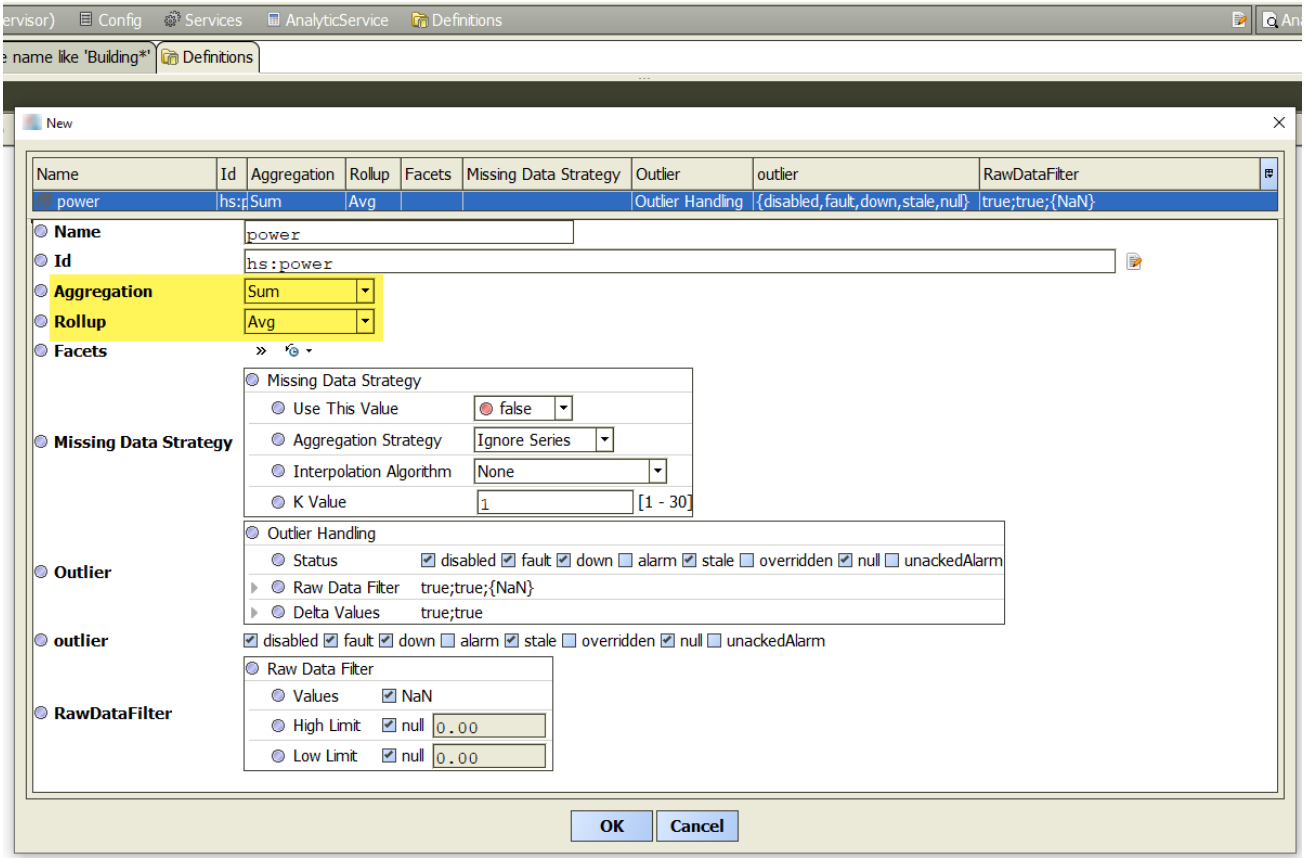
Aggregation defined by data definition or proxy extension

This topic expands on the aggregation request example using the same four BuildingX and MainKw points set up previously.

The data definition is an optional component used to configure default properties for a given piece of data in one place. Using this component eliminates or reduces the additional configuration required on analytic proxy extensions and widget (Chart, Table, Label) bindings.

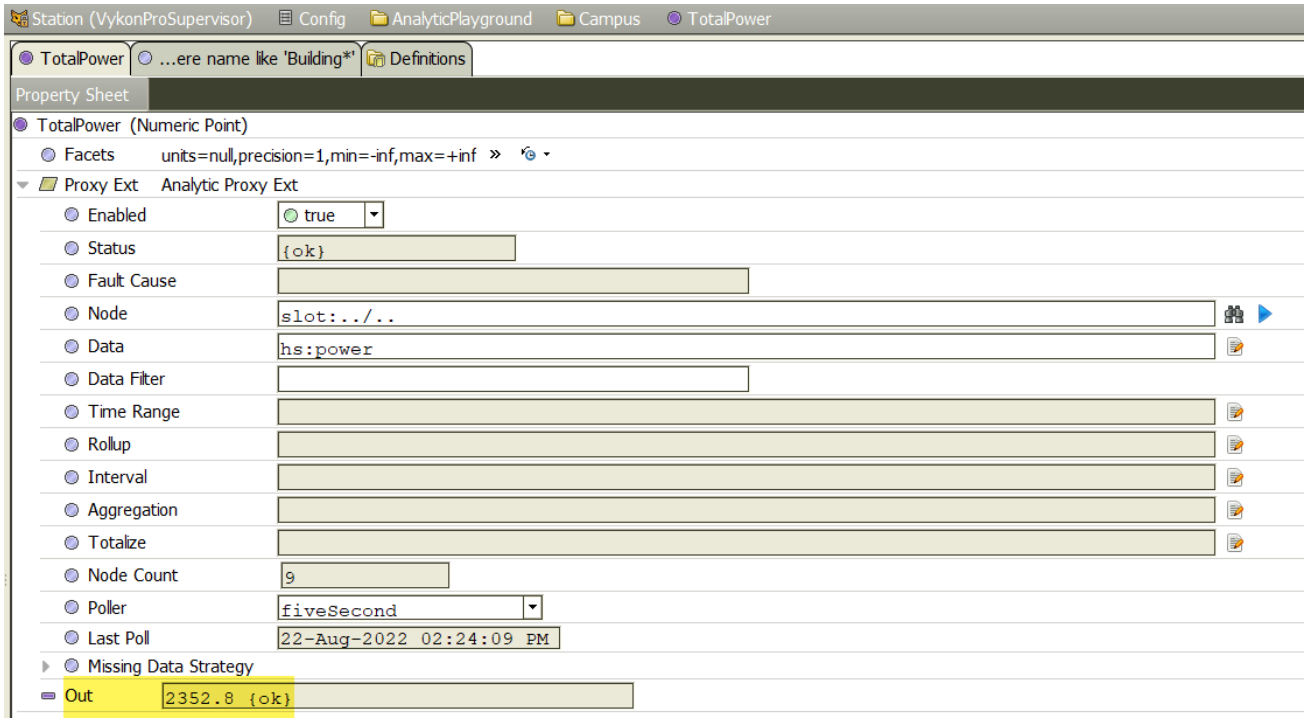
Consider a data definition for `hs:power`. In this example, you only need to configure **`Aggregation`** and **`Rollup`**.

Figure 33 Data definition for aggregation



In the screen capture, **`Aggregation`** is configured as `Sum` and **`Rollup`** is configured as `Avg`. Other properties, such as the **`Missing Data Strategy`**, **`Outlier`** and **`Raw Data Filter`** are useful but only applicable to analytic requests with trend data.

Figure 34 Result of the Sum function

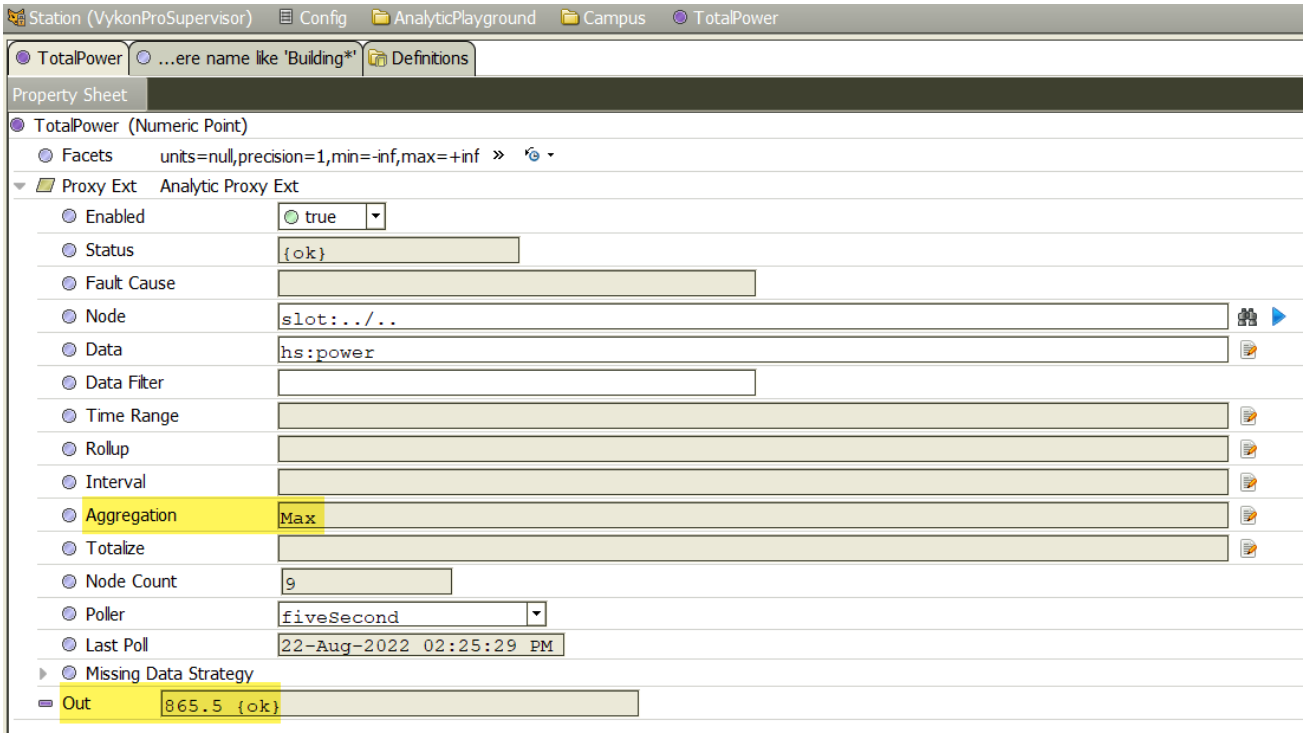


Once the framework processes the analytic request (the interval poller on the analytic proxy extension triggers every five seconds) the **Out** slot updates and displays the sum of the four MainKw point values (2352.8).

Notice that the **Aggregation** property of Sum on the data definition has overridden the default **Aggregation** function of First.

Now, instead of using a data definition to configure **Aggregation**, you can explicitly configure the **Aggregation** property on the **Proxy Ext**.

Figure 35 Aggregation property configured on the Proxy Ext

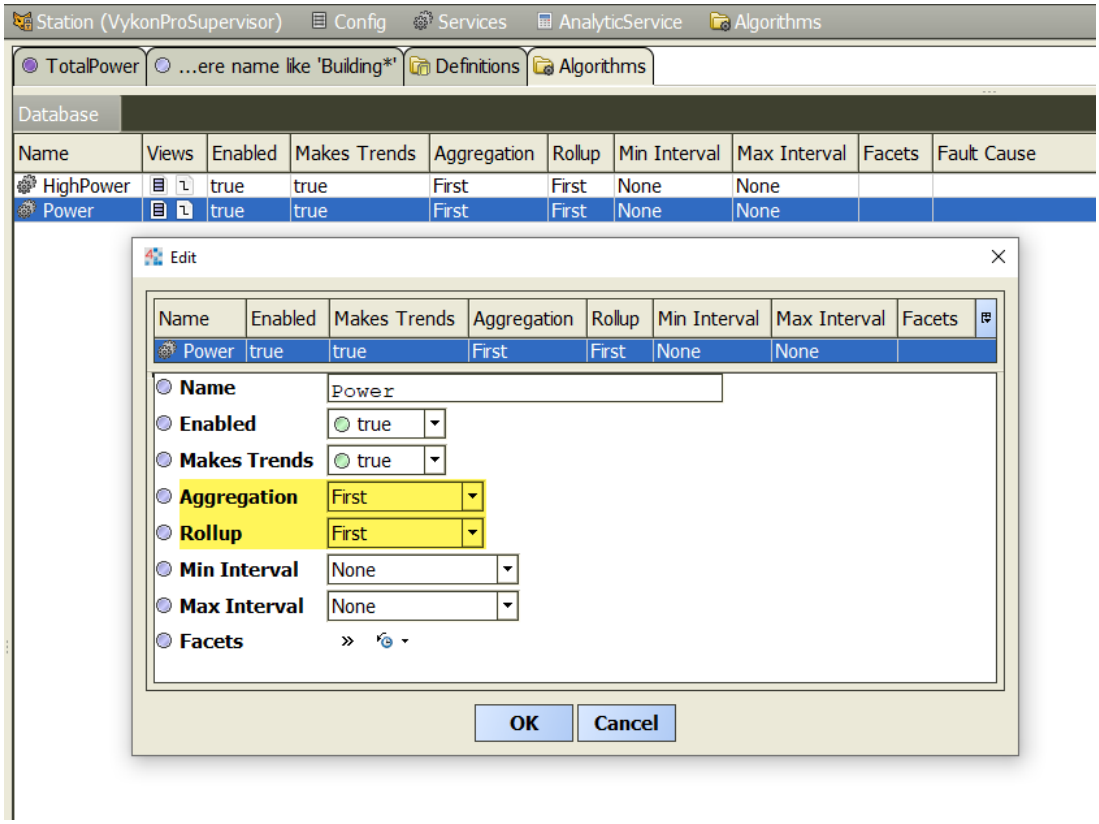


In the screen capture, **`Aggregation`** is set to `Max`, which should override the data definition **`Aggregation`** property. The result in the **`Out`** slot is now `865.5`.

Aggregation defined by an algorithm

Algorithms also have properties for **`Aggregation`** and **`Rollup`**, which default to `First`. These examples use the same four BuildingX and MainKw points set up previously.

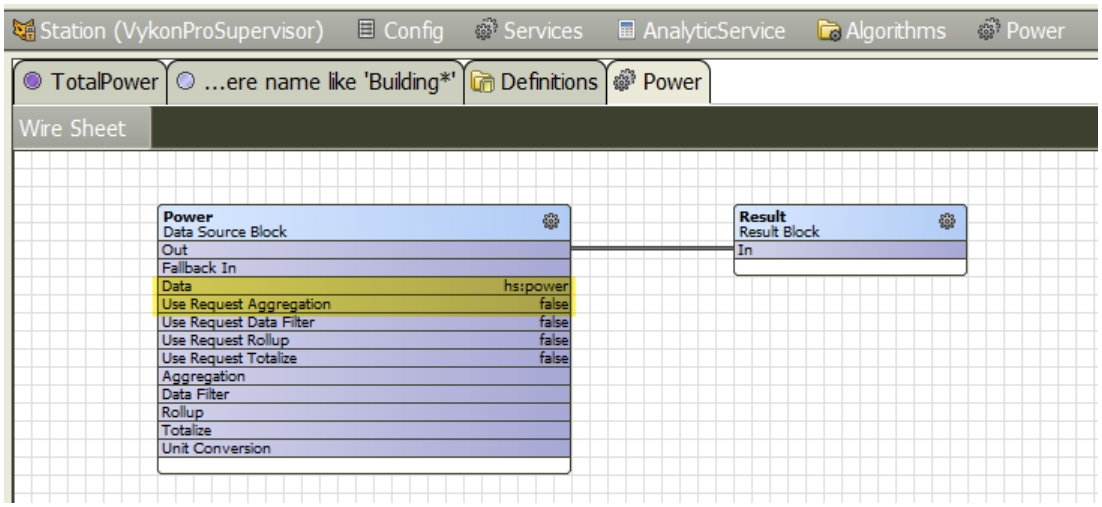
Figure 36 Aggregation algorithm



The screen capture shows the default values for **`Aggregation`** and **`Rollup`**.

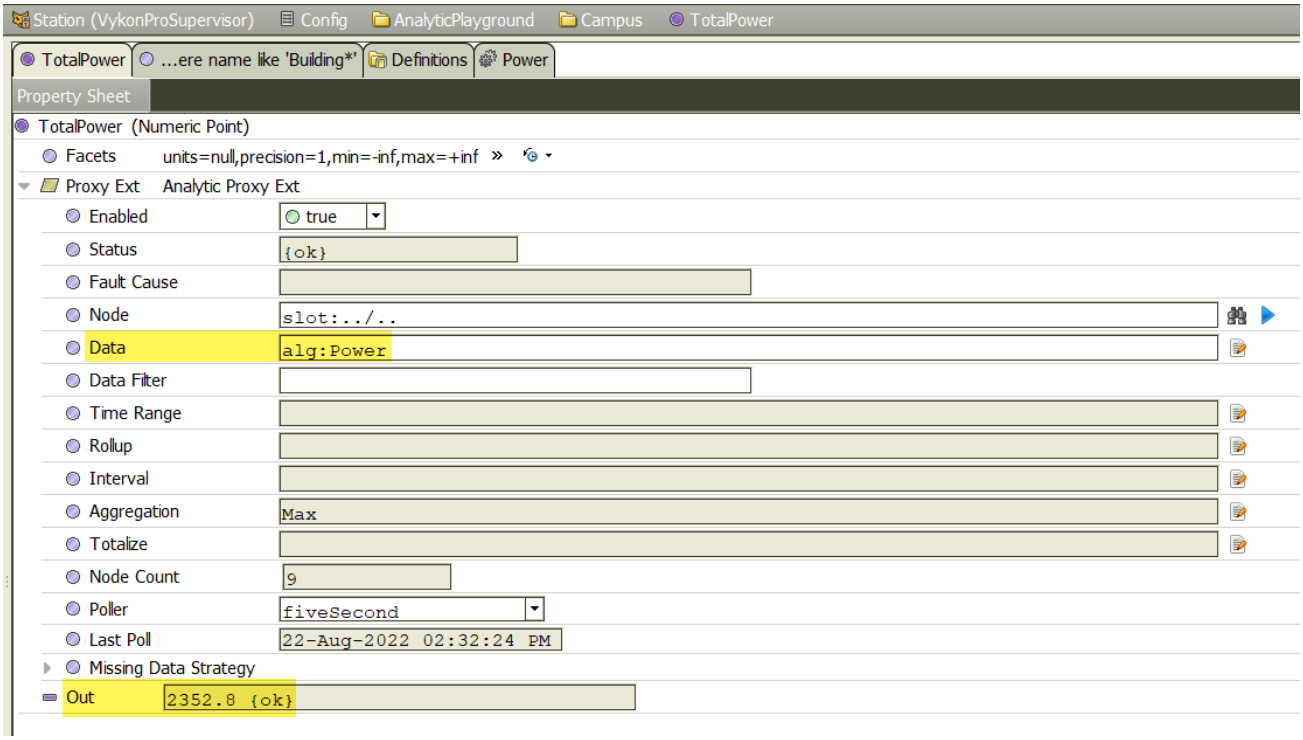
Consider this very basic algorithm, which has a single data source with the data configured to `hs:power` and linked to a result block:

Figure 37 Algorithm with a single data source



The **Data Source Block** has a property named **`Use Request Aggregation`** with a default value of `false`, and an **`Aggregation`** property with a null value (not explicitly configured).

Figure 38 Proxy extension with Data property configured

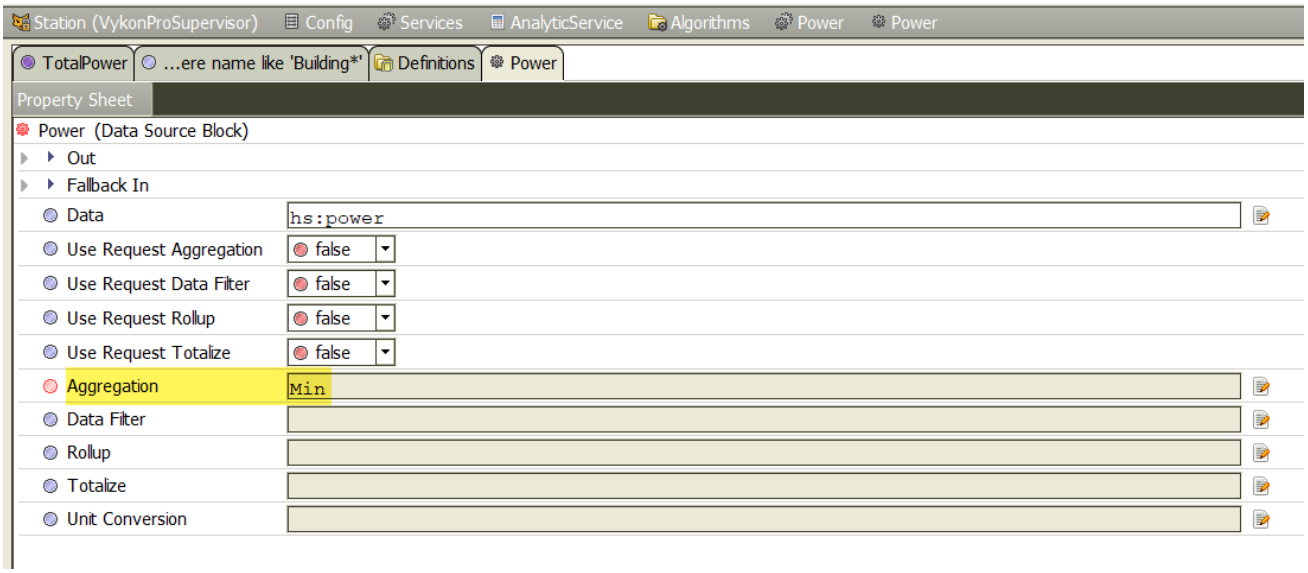


Using the same control point with the analytics proxy extension and changing the **Data** property from `hs:power` to `alg:Power` results in an **Out** value that represents the sum of the four points (`2352.8`).

This happens even though the **Aggregation** property on the proxy extension is configured as `Max`, and the **Data Source Block** in the algorithm is configured to ignore the **Aggregation** property in the request, which causes it to default to the `hs:power` data definition.

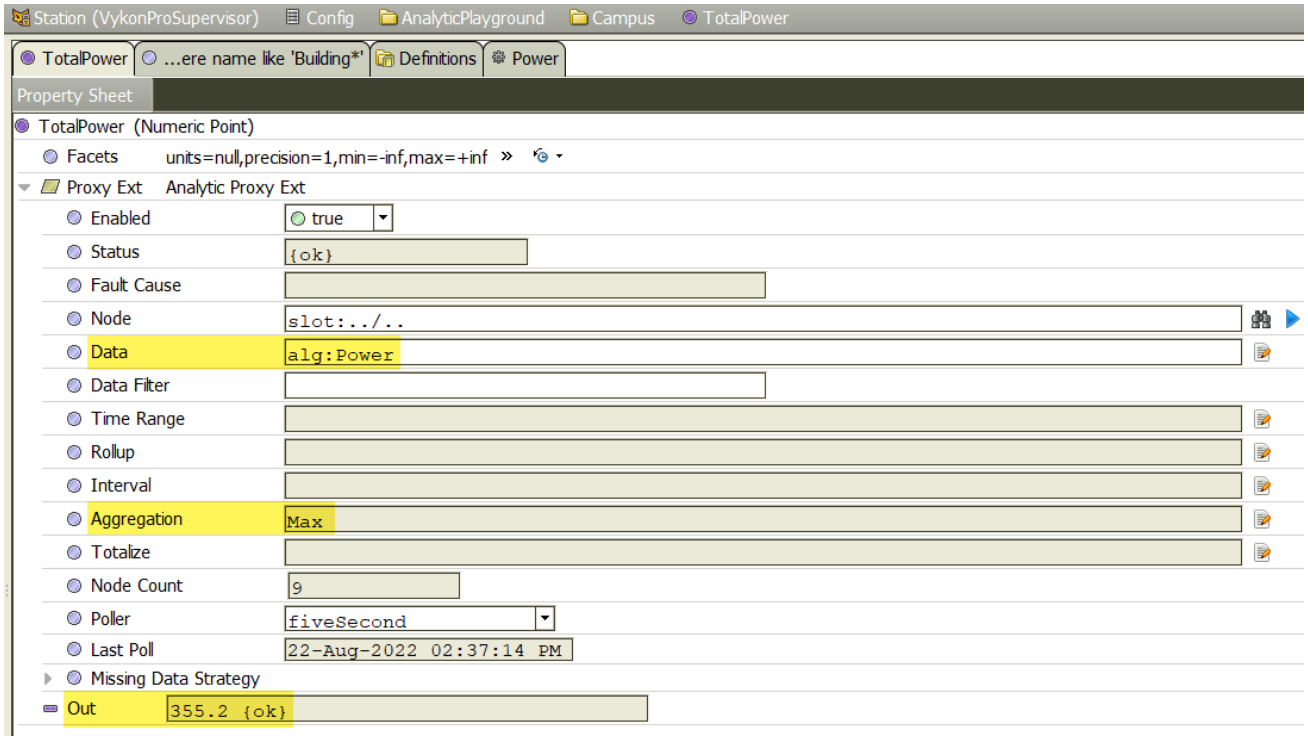
You can configure the **Aggregation** function explicitly on the **Data Source Block**.

Figure 39 Aggregation on the Data Source Block configured to Min



This explicit configuration setting **Aggregation** to `Min` on the algorithm's **Data Source Block** overrides the data definition **Aggregation**, which is set to the `Sum` function.

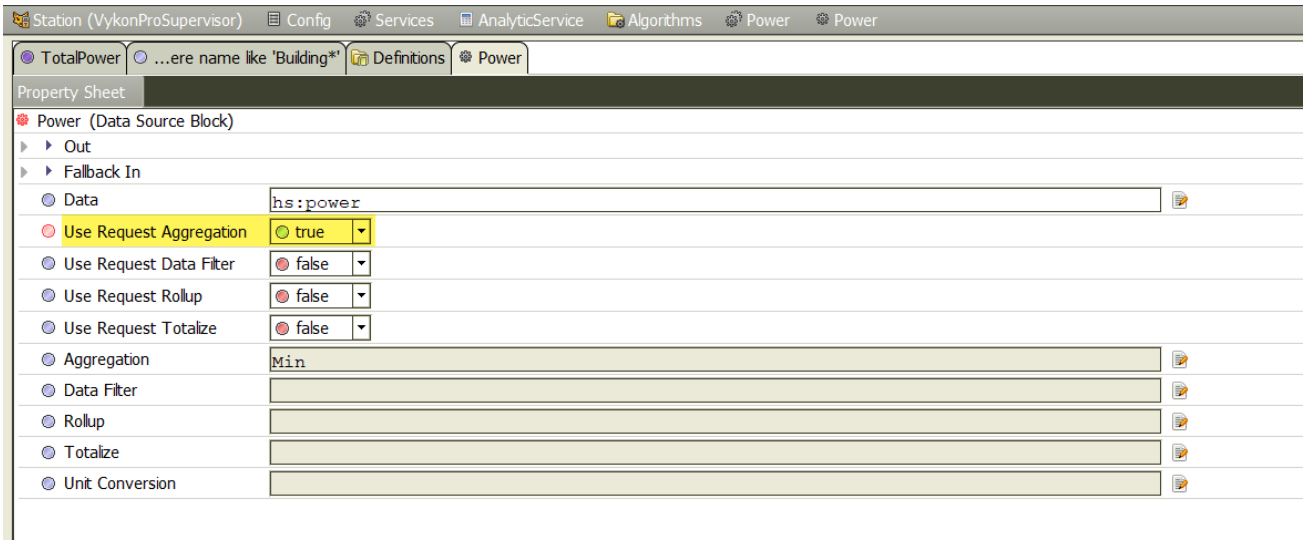
Figure 40 Result after applying the algorithm



The **Out** slot now reports the minimum value of the four points, `355.2`. This means that explicit configuration of the **Aggregation** using the Wire Sheet's **Data Source Block** has overwritten the **Data** property (`alg:Power`) and **Aggregation** (`Max`) as defined on the proxy extension.

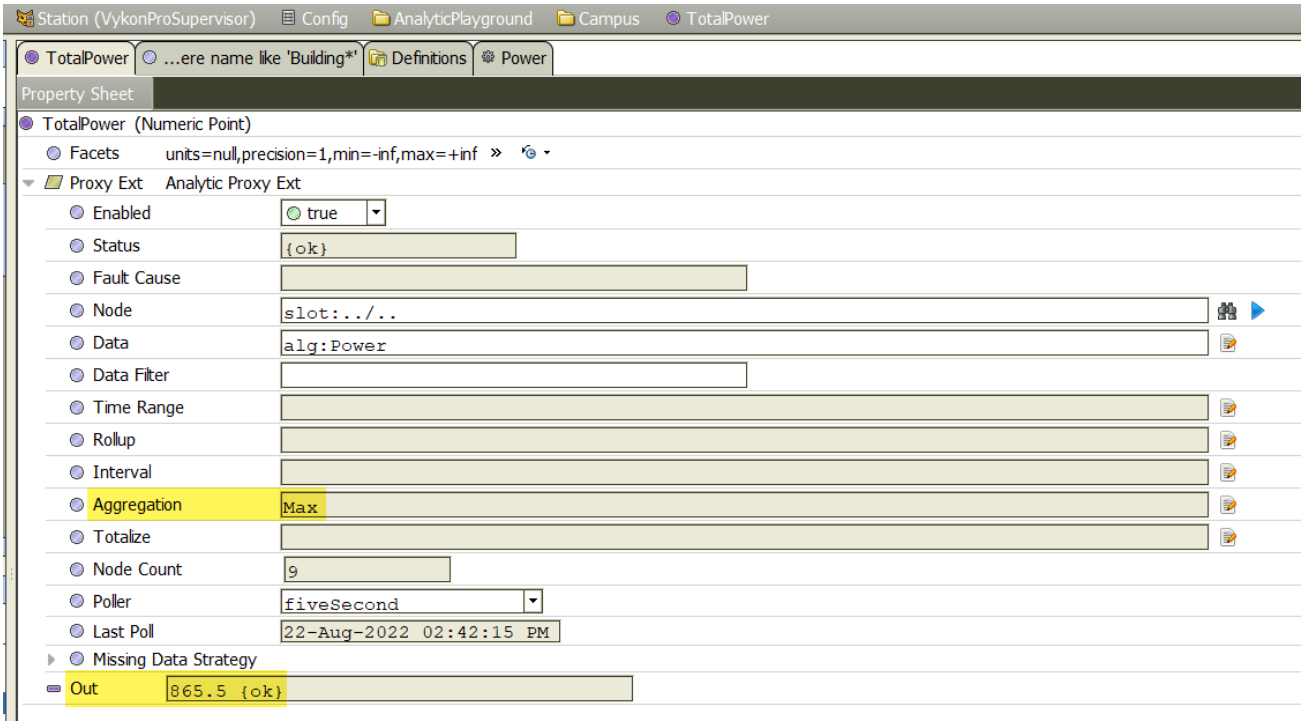
Now go back to the algorithm's **Data Source Block** and enable **Use Request Aggregation**.

Figure 41 Use Request Aggregation enabled on the Data Source Block



Changing **Use Request Aggregation** to `true` configures the data source to instead use the **Aggregation** property value as defined by the analytic request and configured on the analytic proxy extension.

Figure 42 Analytic request as configured on the analytic proxy extension.



The control point value now reports the maximum of the four points (`865.5`) because the **`Aggregation`** function configured by this analytic proxy extension property sheet is set in the analytic request to the algorithm, and the algorithm's **Data Source Block** is configured to use the **`Aggregation`** function from the request, which overrides the data definition for `hs:power`.

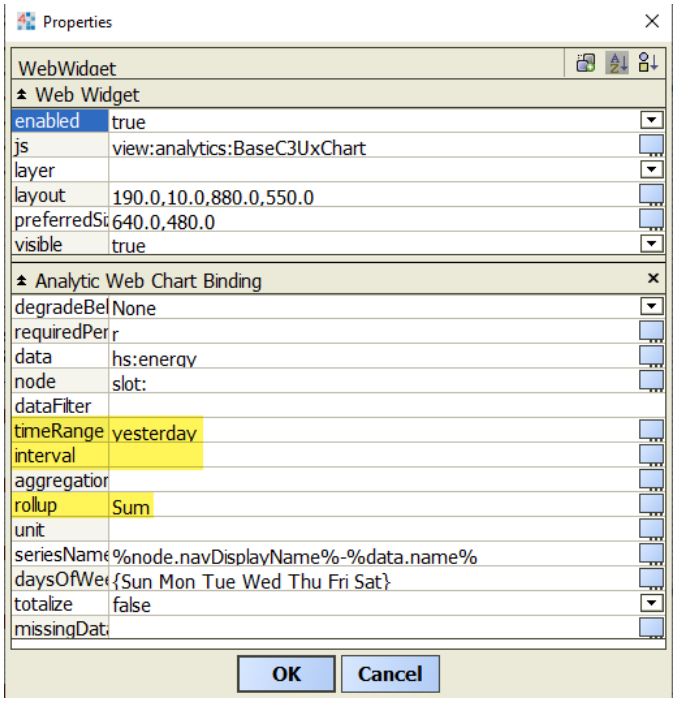
IMPORTANT: As mentioned in the introduction to these examples, each property, including **`Time Range`**, **`Rollup`**, **`Interval`**, **`Aggregation`**, **`Totalize`** and **`Missing Data Strategy`** has a default setting, which Analytics uses if you do not explicitly set the property. Each property should behave similarly to the **`Aggregation`** function as far as how the override behavior works when you configure the property on a data definition, analytic proxy extension, binding or Data Source Block.

Trend Interval defined in a binding

`Interval` is a property used in conjunction with analytic trend requests to specify the time window to combine records using the configured **`Rollup`** function. The framework handles the **`Interval`** property in special ways depending on where you configure it.

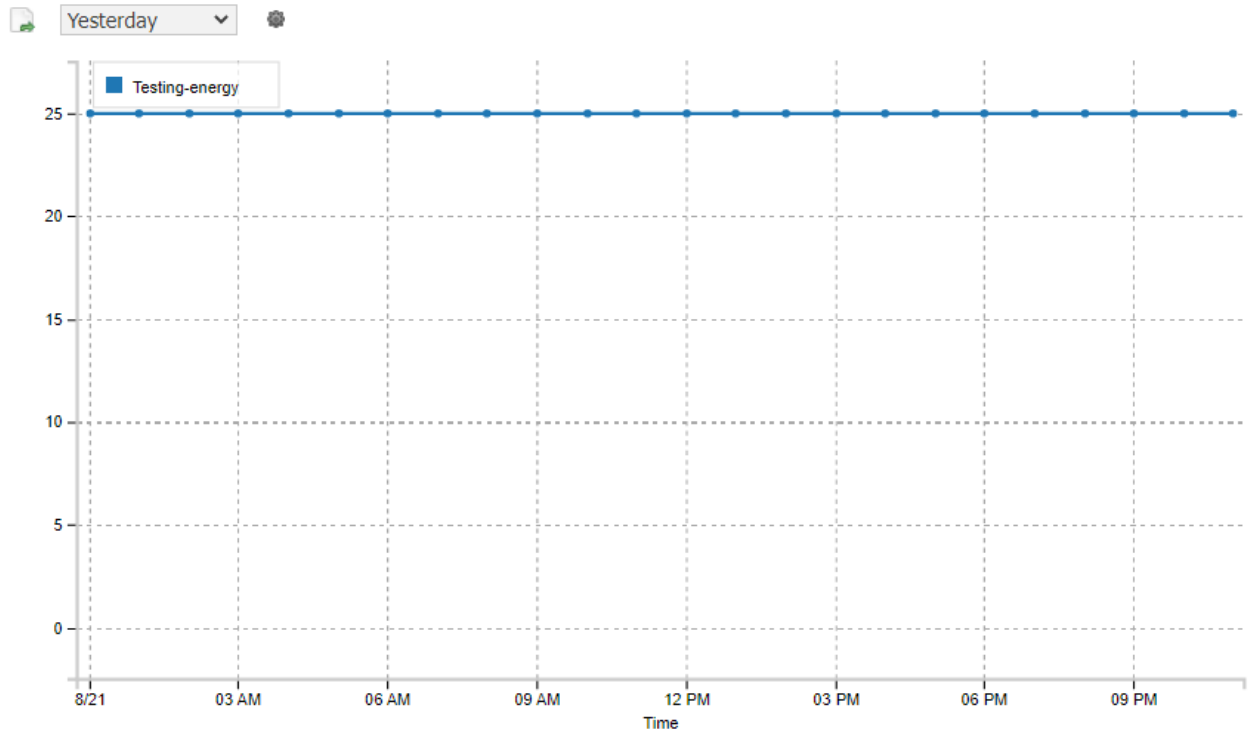
Widgets, such as a table or chart use the Analytic Web Bindings function similarly to the way standard Web Chart sampling works, where sampling type is a function like analytic **`Rollup`** and the desired period is a window of time like analytic **`Interval`**. Consider a **`NumericPoint`** with associated history data (an actual history extension or imported from a remote station that has an `n:history` tag mapping to the imported history), where the history records are coming in at one hour intervals and the value is totalized (ever increasing) by a fixed value of 25 for each record. The point has tags including `hs:hisTotalized`, `hs:energy` and `a:a` marker tags. The px view includes a web widget (analytic web chart) with an Analytic Web Chart Binding:

Figure 43 WebWidget configured for a NumericPoint



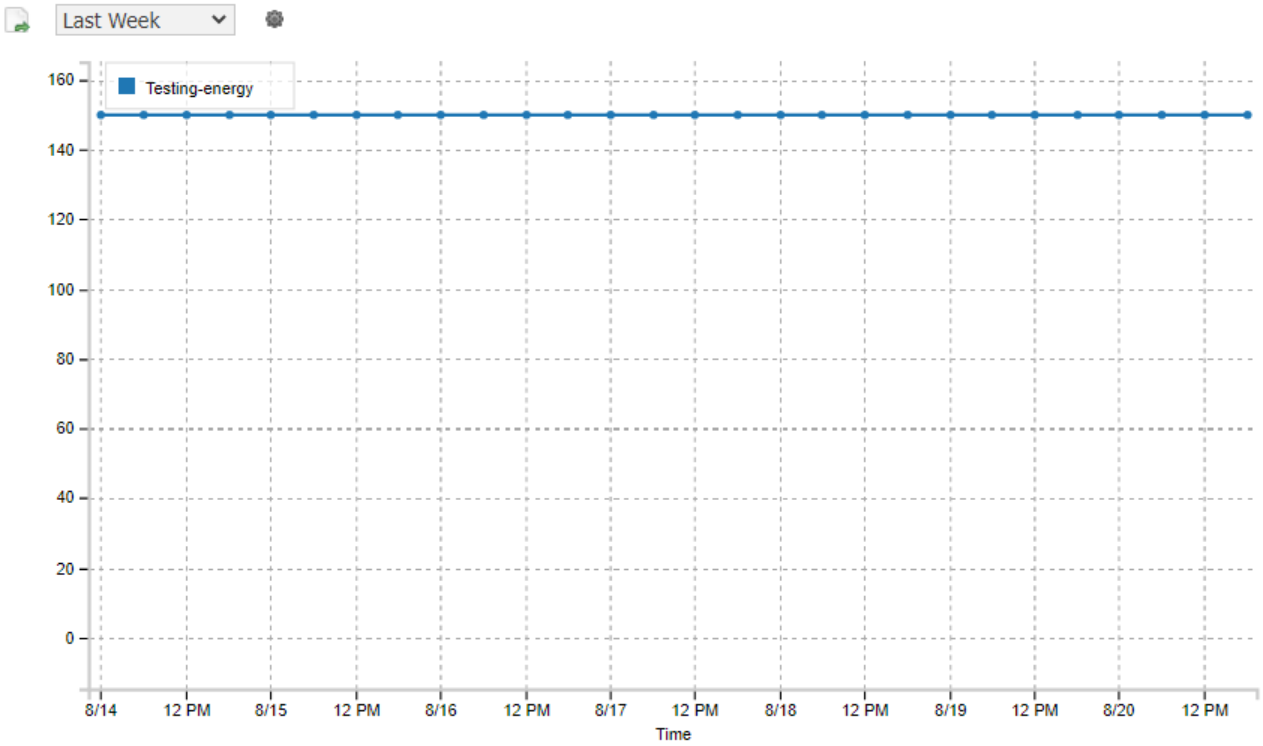
The binding is configured with a **`Rollup`** function of `Sum` and a **`Time Range`** of `Yesterday`, but does not explicitly configure the **`Interval`**. Analytics automatically picks a best fit interval based on the time range and available data records. In this case, it uses the raw collection interval of one hour. As expected, the value of each record is 25.

Figure 44 Chart produced with the default Interval of one hour



Simply changing the **Time Range** for the chart to `Last Week` instead of `Yesterday` results in a different automatic **Interval** of six hours instead of one hour.

Figure 45 Chart produced with the default Interval of six hours



In this example, the framework plots records at midnight, 6 am, noon and 6 pm for each day. Each record value is 150, which is the record value of 25 * 6 records in the interval window. This is just one example of how changing the **Time Range** likely results in different automatic intervals being applied. Explicitly configuring the **Interval** in the binding takes precedence over these default automatic interval values.

TIP: To ensure that your trend result is what you expect, always configure the **Interval** in your binding.

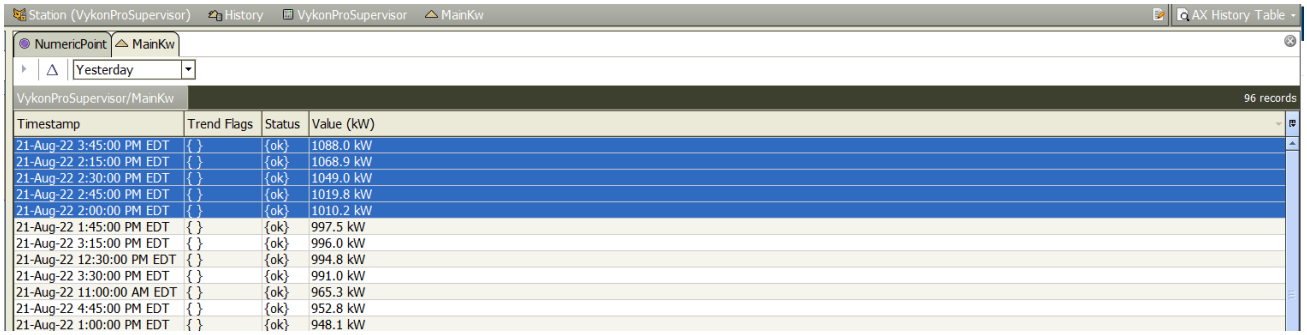
Trend Interval defined in a proxy extension

The framework handles the **Interval** property on the **AnalyticProxyExt** in a special way because, ultimately, the extension must resolve to a single answer even if it is using a **Time Range** for an analytic trend request.

Unlike a table or chart, which can display multiple values, the control point only has a single **Out** slot. Configuring the **Interval** property may impact the final result because the data (tag or algorithm) might produce different results when the framework rolls up history records into fewer records to be processed by the algorithm. In this case, the framework rolls up the historical records and runs the rolled up values through the algorithm. Then, running the algorithm many times rolls up everything into a single result.

Consider a power history for `Yesterday` where there are five records (highlighted below) with a value greater than 1,000 kW.

Figure 46 NumericPoint table reporting kW values from Yesterday

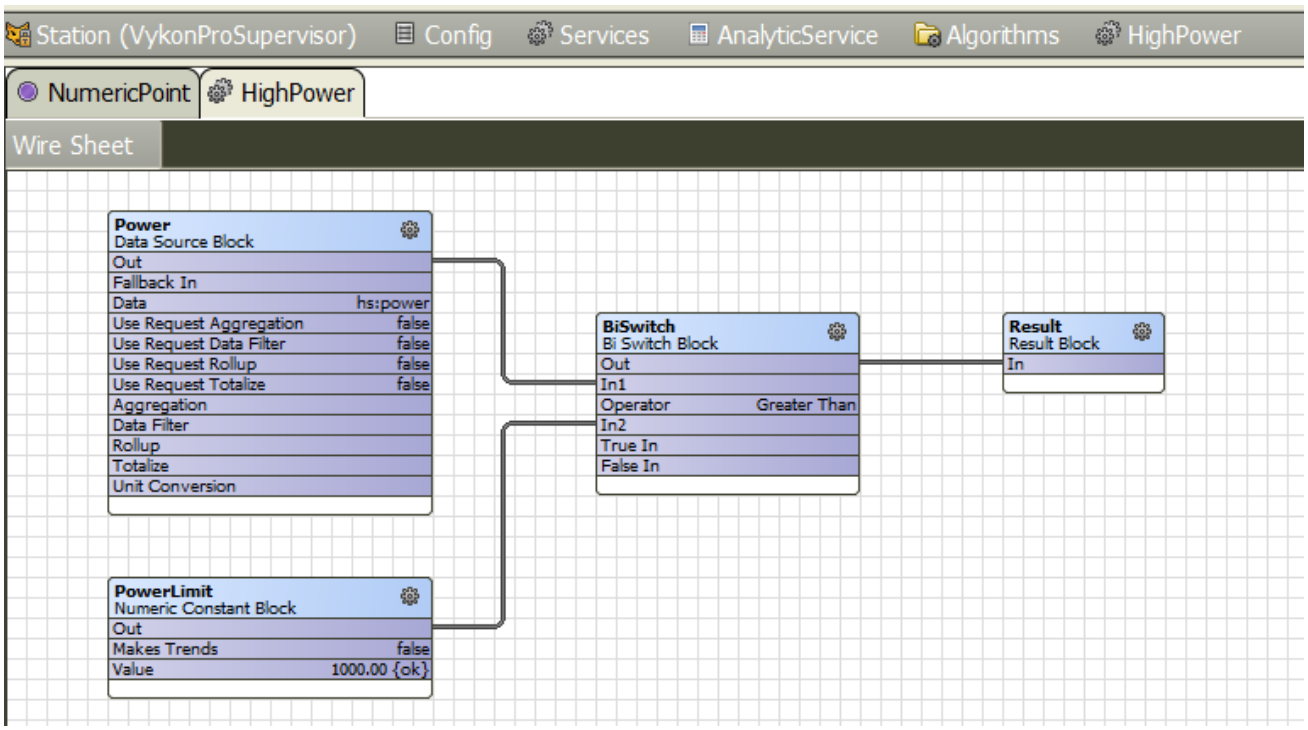


Timestamp	Trend Flags	Status	Value (kW)
21-Aug-22 3:45:00 PM EDT	{ }	{ok}	1088.0 kW
21-Aug-22 2:15:00 PM EDT	{ }	{ok}	1068.9 kW
21-Aug-22 2:30:00 PM EDT	{ }	{ok}	1049.0 kW
21-Aug-22 2:45:00 PM EDT	{ }	{ok}	1019.8 kW
21-Aug-22 2:00:00 PM EDT	{ }	{ok}	1010.2 kW
21-Aug-22 1:45:00 PM EDT	{ }	{ok}	997.5 kW
21-Aug-22 3:15:00 PM EDT	{ }	{ok}	996.0 kW
21-Aug-22 12:30:00 PM EDT	{ }	{ok}	994.8 kW
21-Aug-22 3:30:00 PM EDT	{ }	{ok}	991.0 kW
21-Aug-22 11:00:00 AM EDT	{ }	{ok}	965.3 kW
21-Aug-22 4:45:00 PM EDT	{ }	{ok}	952.8 kW
21-Aug-22 1:00:00 PM EDT	{ }	{ok}	948.1 kW

Notice that there are 96 records and the table is sorted on the Value column, not the Timestamp column.

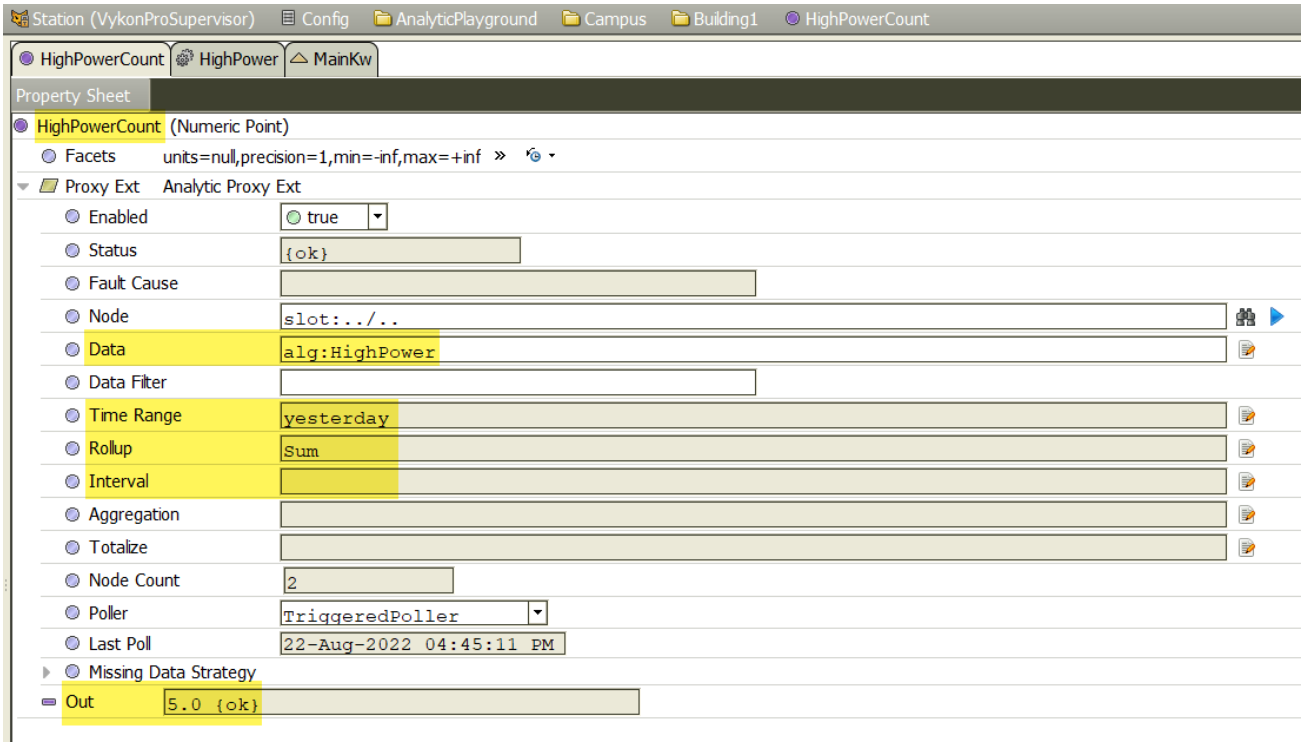
The following basic algorithm evaluates whether the power value is greater than 1,000 kW.

Figure 47 Algorithm to evaluate kW greater than 1000



In the screen capture that follows, the control point, HighPowerCount, is configured with an **AnalyticProxyExt**.

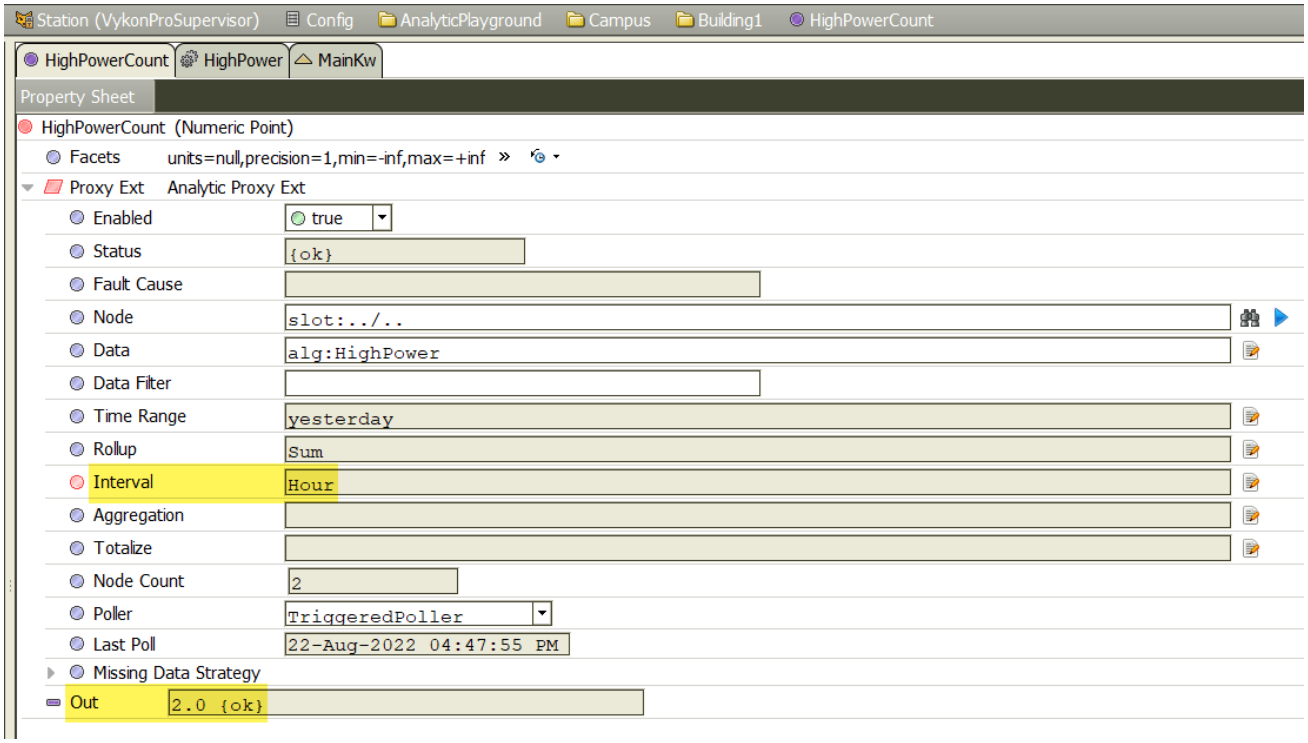
Figure 48 AnalyticProxyExt configured with Interval left at the default



This configuration queries the **HighPower** algorithm for a **`Time Range`** of `yesterday` and a **`Rollup`** of `Sum`, which sums the results (false = 0, true =1) of running the algorithm 96 times once with each 15-minute record in the **`Time Range`**. The result is the expected `5.0` based on the history table above.

If not explicitly configured, **`Interval`** defaults to `Fifteen Minutes`, which just happens to match the collection frequency of the history, so, by default each record in the example above was actually processed.

Figure 49 AnalyticProxyExt configured with Interval explicitly set



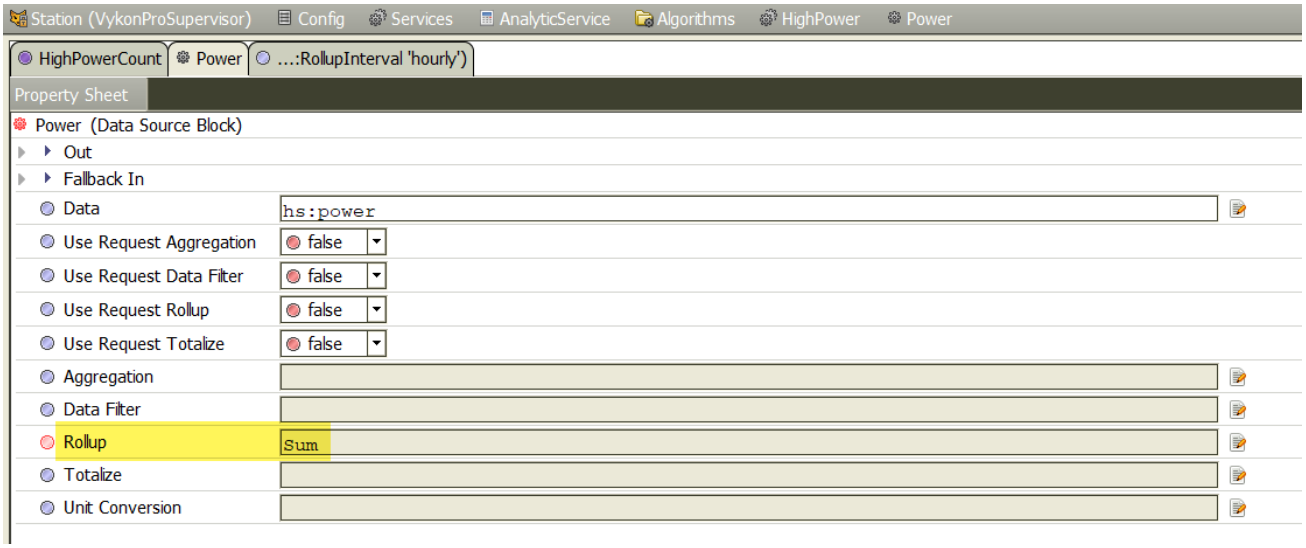
If you explicitly set the **`Interval`** to `Hour`, the framework rolls up the four 15-minute records from each hour into a single value, then processes those 24 rolled-up values once each through the algorithm. Finally, it sums the results of the 24 algorithm executions returning only `2.0`.

This result is probably confusing.

The **Data Source Block** property **`Use Request Rollup`** is `false` and the **`Rollup`** is null, so even though the analytic trend request specifies the **`Interval`** as `Hour` and **`Rollup`** as `Sum`, the **Data Source Block** uses the default **`Rollup`** function of `First` to return the result. Apparently, in this data set there were only two records in yesterday for the top of an hour (1:00 am, 2:00 am, etc.) where the value was greater than 1,000.

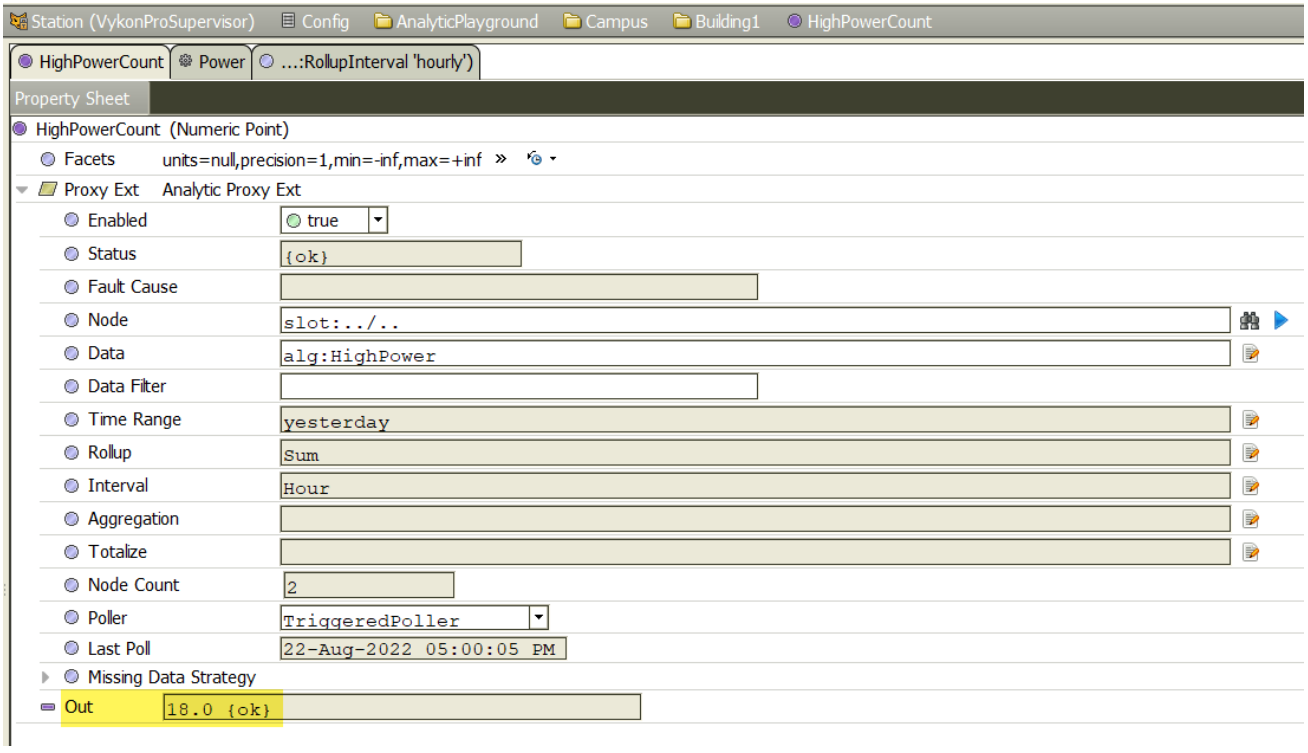
You could configure the **Data Source Block** to explicitly apply the **`Rollup`** function of `Sum` or to use the **`Rollup`** function from the analytic request.

Figure 50 Data Source Block explicitly configured with a Rollup of Sum



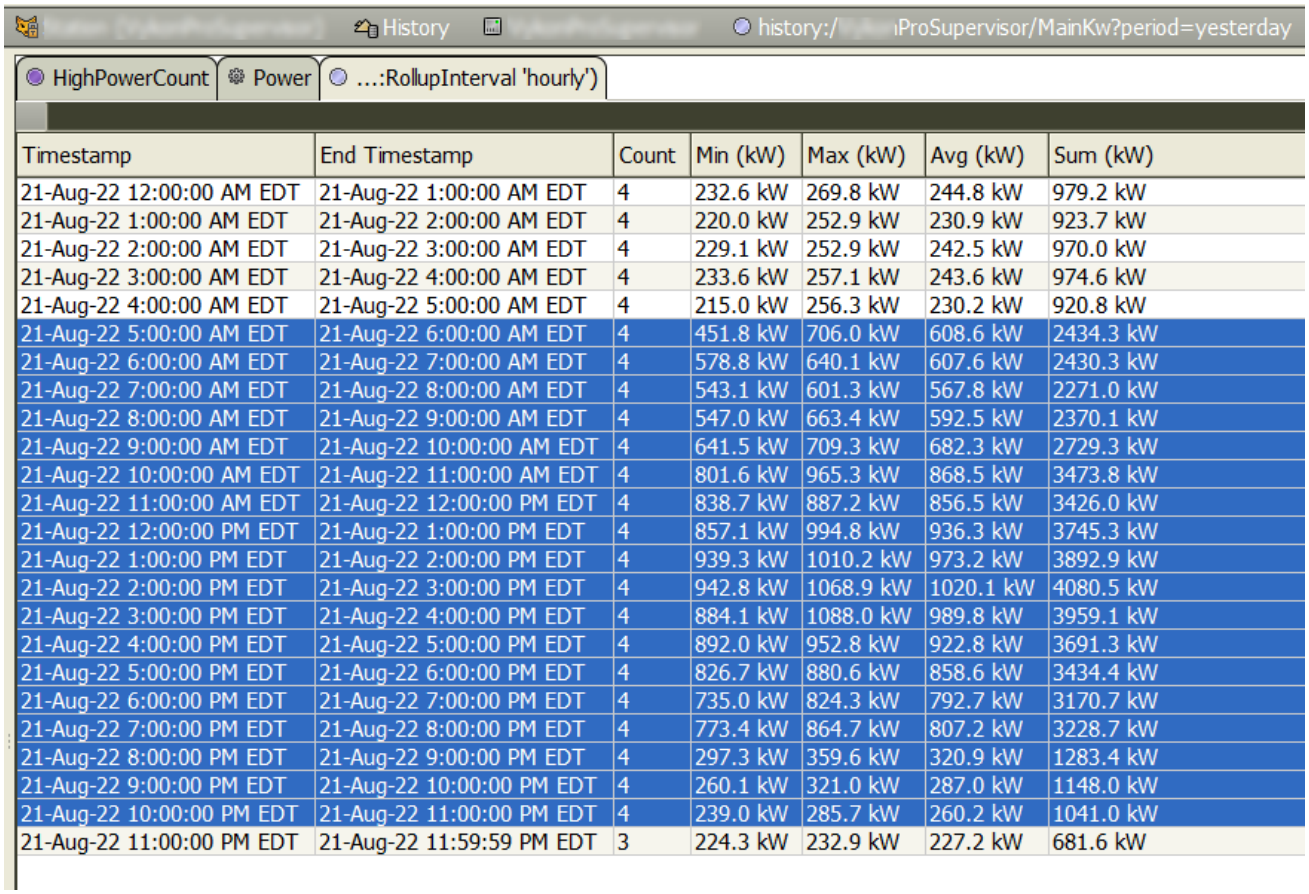
The screen capture shows the **`Rollup`** function of `Sum` explicitly configured in the **Data Source Block**. However, this configuration still does not yield the desired result because the high limit is hard coded in the algorithm and does not get rolled up like the power value does.

Figure 51 AnalyticProxyExt still with an invalid result



Using the bql query: `bql:history:HistoryRollup.rollup(history:RollupInterval 'hourly')`, the request returned 18 because there were 18 hourly intervals yesterday where the sum of the four values is greater than the hard coded high limit of 1,000 as shown below.

Figure 52 NumericPoint showing 18 instances where kW is greater than 1000



History history:/ [redacted] iProSupervisor/MainKw?period=yesterday

HighPowerCount | Power | ...:RollupInterval 'hourly')

Timestamp	End Timestamp	Count	Min (kW)	Max (kW)	Avg (kW)	Sum (kW)
21-Aug-22 12:00:00 AM EDT	21-Aug-22 1:00:00 AM EDT	4	232.6 kW	269.8 kW	244.8 kW	979.2 kW
21-Aug-22 1:00:00 AM EDT	21-Aug-22 2:00:00 AM EDT	4	220.0 kW	252.9 kW	230.9 kW	923.7 kW
21-Aug-22 2:00:00 AM EDT	21-Aug-22 3:00:00 AM EDT	4	229.1 kW	252.9 kW	242.5 kW	970.0 kW
21-Aug-22 3:00:00 AM EDT	21-Aug-22 4:00:00 AM EDT	4	233.6 kW	257.1 kW	243.6 kW	974.6 kW
21-Aug-22 4:00:00 AM EDT	21-Aug-22 5:00:00 AM EDT	4	215.0 kW	256.3 kW	230.2 kW	920.8 kW
21-Aug-22 5:00:00 AM EDT	21-Aug-22 6:00:00 AM EDT	4	451.8 kW	706.0 kW	608.6 kW	2434.3 kW
21-Aug-22 6:00:00 AM EDT	21-Aug-22 7:00:00 AM EDT	4	578.8 kW	640.1 kW	607.6 kW	2430.3 kW
21-Aug-22 7:00:00 AM EDT	21-Aug-22 8:00:00 AM EDT	4	543.1 kW	601.3 kW	567.8 kW	2271.0 kW
21-Aug-22 8:00:00 AM EDT	21-Aug-22 9:00:00 AM EDT	4	547.0 kW	663.4 kW	592.5 kW	2370.1 kW
21-Aug-22 9:00:00 AM EDT	21-Aug-22 10:00:00 AM EDT	4	641.5 kW	709.3 kW	682.3 kW	2729.3 kW
21-Aug-22 10:00:00 AM EDT	21-Aug-22 11:00:00 AM EDT	4	801.6 kW	965.3 kW	868.5 kW	3473.8 kW
21-Aug-22 11:00:00 AM EDT	21-Aug-22 12:00:00 PM EDT	4	838.7 kW	887.2 kW	856.5 kW	3426.0 kW
21-Aug-22 12:00:00 PM EDT	21-Aug-22 1:00:00 PM EDT	4	857.1 kW	994.8 kW	936.3 kW	3745.3 kW
21-Aug-22 1:00:00 PM EDT	21-Aug-22 2:00:00 PM EDT	4	939.3 kW	1010.2 kW	973.2 kW	3892.9 kW
21-Aug-22 2:00:00 PM EDT	21-Aug-22 3:00:00 PM EDT	4	942.8 kW	1068.9 kW	1020.1 kW	4080.5 kW
21-Aug-22 3:00:00 PM EDT	21-Aug-22 4:00:00 PM EDT	4	884.1 kW	1088.0 kW	989.8 kW	3959.1 kW
21-Aug-22 4:00:00 PM EDT	21-Aug-22 5:00:00 PM EDT	4	892.0 kW	952.8 kW	922.8 kW	3691.3 kW
21-Aug-22 5:00:00 PM EDT	21-Aug-22 6:00:00 PM EDT	4	826.7 kW	880.6 kW	858.6 kW	3434.4 kW
21-Aug-22 6:00:00 PM EDT	21-Aug-22 7:00:00 PM EDT	4	735.0 kW	824.3 kW	792.7 kW	3170.7 kW
21-Aug-22 7:00:00 PM EDT	21-Aug-22 8:00:00 PM EDT	4	773.4 kW	864.7 kW	807.2 kW	3228.7 kW
21-Aug-22 8:00:00 PM EDT	21-Aug-22 9:00:00 PM EDT	4	297.3 kW	359.6 kW	320.9 kW	1283.4 kW
21-Aug-22 9:00:00 PM EDT	21-Aug-22 10:00:00 PM EDT	4	260.1 kW	321.0 kW	287.0 kW	1148.0 kW
21-Aug-22 10:00:00 PM EDT	21-Aug-22 11:00:00 PM EDT	4	239.0 kW	285.7 kW	260.2 kW	1041.0 kW
21-Aug-22 11:00:00 PM EDT	21-Aug-22 11:59:59 PM EDT	3	224.3 kW	232.9 kW	227.2 kW	681.6 kW

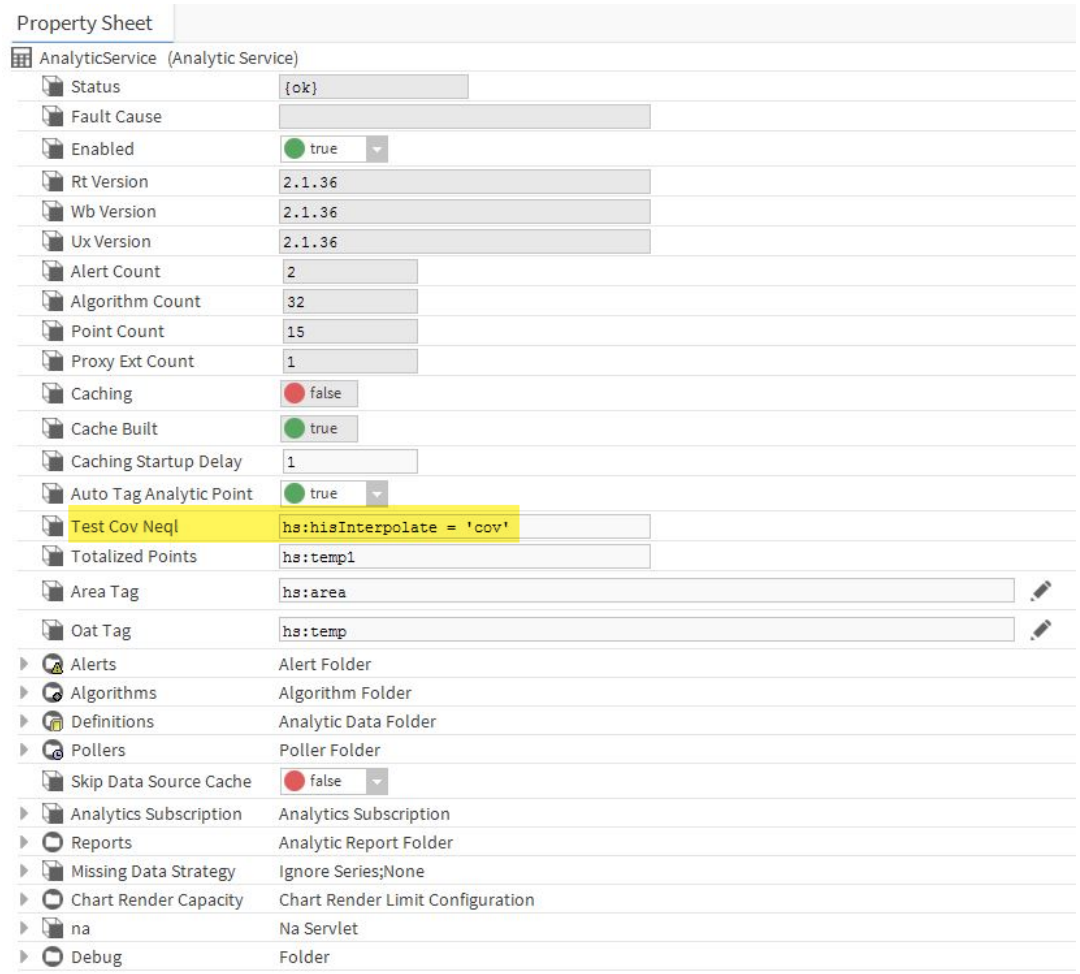
In this case, if you planned to apply an **`Interval`** that is different from the history collection **`Interval`** you would need to have a history with the high limit value. Even if you do have a history with the high limit value, comparing the rolled up (sum) power values versus the rolled up (sum) high limit values may not produce the same or expected result as processing individual history records.

COV histories

COV (Change of Value) histories are often used in place of interval histories (records created at a fixed interval regardless of value change) for control points where the value does not change frequently. Using COV histories may consume less memory for storage. The records COV histories create indicate when the value of the control point changed, whereas, interval histories provide consistent record counts at the configured intervals, which makes analysis simpler at the expense of, perhaps, not knowing exactly when the value changed within the interval.

The framework handles COV history records in a special way.

Figure 53 The AnalyticService's NEQL test for COV identity



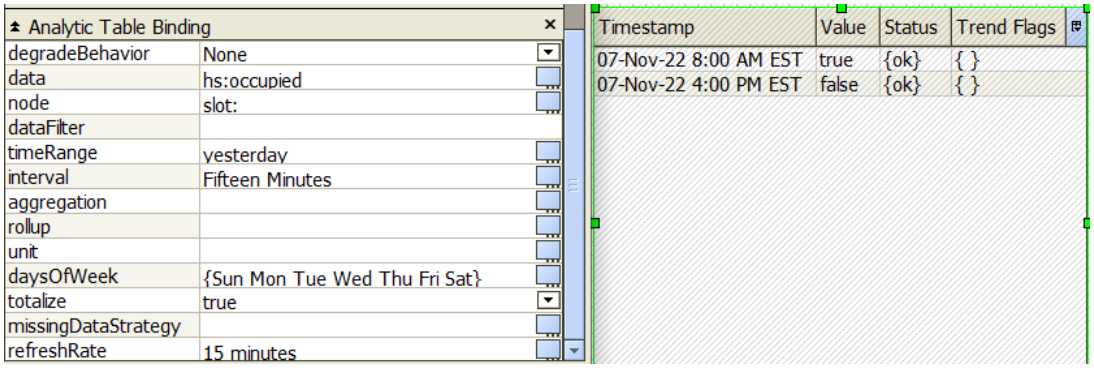
The **`Test Cov Neql`** property on the **AnalyticService** determines if a history contains COV (irregular interval) histories. This property, which defaults to `hs:hisInterpolate='cov'`, relies on the `hs:hisInterpolate` tag from the Haystack tag dictionary being applied to COV histories in the history database.

The Haystack tag dictionary is a smart tag dictionary with a tag rule that applies the `hs:hisInterpolate` tag as an implied tag to any control point with a history extension. If the history extension is a COV history (records at irregular intervals), the `hs:hisInterpolate` tag value is `cov` and if the history extension is an interval history (records at regular intervals), the `hs:hisInterpolate` tag value is `linear`.

The tag rule does not automatically apply the `hs:hisInterpolate` tag to COV histories in the station's database, which are imported, such as Niagara history imports or BACnet history imports. You must manually apply this tag as a direct tag with the `cov` tag value. This is similar to the process of applying a direct `n:history` tag to histories imported using a BACnet history import.

Consider a Boolean point named **`Occupancy`** with a COV history extension, where the value is controlled by a schedule that is active from 8:00 AM – 4:00 PM Monday through Friday. The **`Occupancy`** point has the `a:a` and `hs:occupied` marker tags applied.

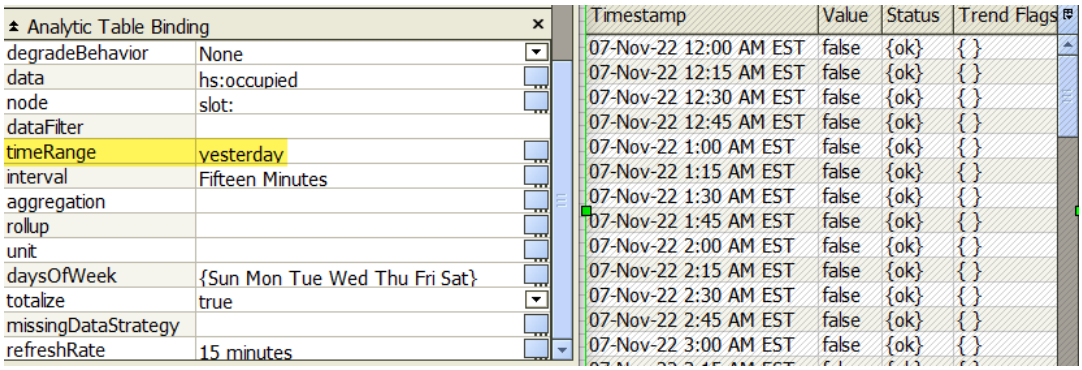
Figure 54 Bound table with missing hs:hisInterpolate='cov' tag



A bound table with an **Analytic Table Binding** may be used to display the records from yesterday. Even though the binding's **`Interval`** property is configured to create a record every `Fifteen Minutes`, the framework returns only two expected COV records: one for 8:00 AM and the other for 4:00 PM. The framework returns only the actual COV records because the control point does not have the `hs:hisInterpolate='cov'` tag applied.

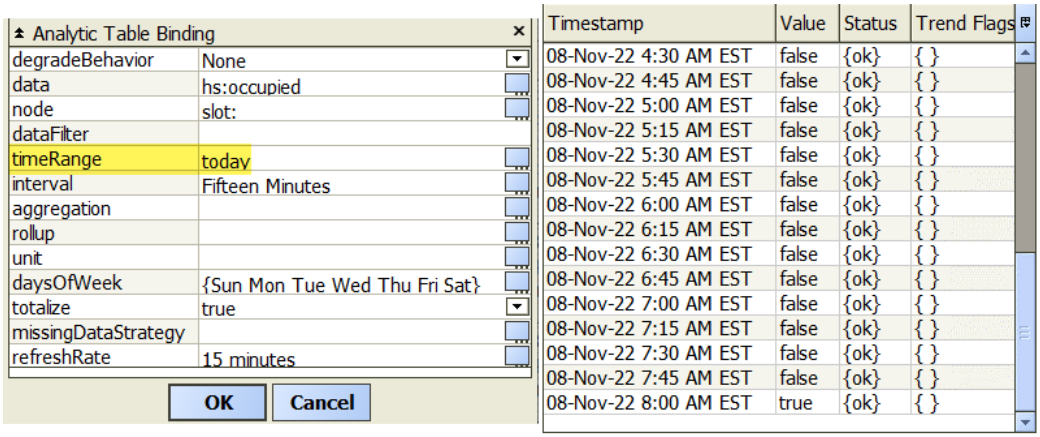
If the `hs:hisInterpoalte='cov'` tag is applied to the **`Occupancy`** control point, the same bound table displays a record for every 15 minutes by filling in the missing intervals with the best known value.

Figure 55 Bound table with hs:hisInterpolate='cov' tag applied reporting from yesterday



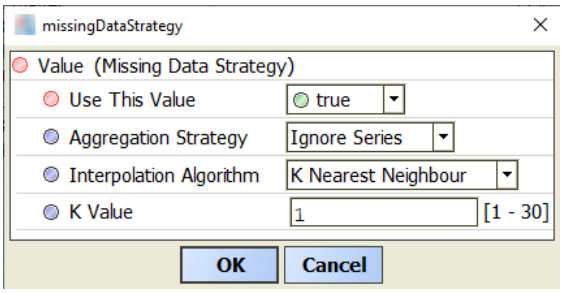
The framework uses the `hs:hisInterpolate='cov'` tag to automatically fill in missing records from yesterday for the requested interval where the requested time range has a COV record available. The framework does not fill in future records, such as a time range of today where the last record in the history is 8:00 AM and the current time is after 8:00 AM.

Figure 56 Bound table with hs:hisInterpolate='cov' tag applied reporting for today



In this example it is necessary to enable the **`Missing Data Strategy`** to use an **`Interpolation Algorithm`**, such as `K Nearest Neighbour`.

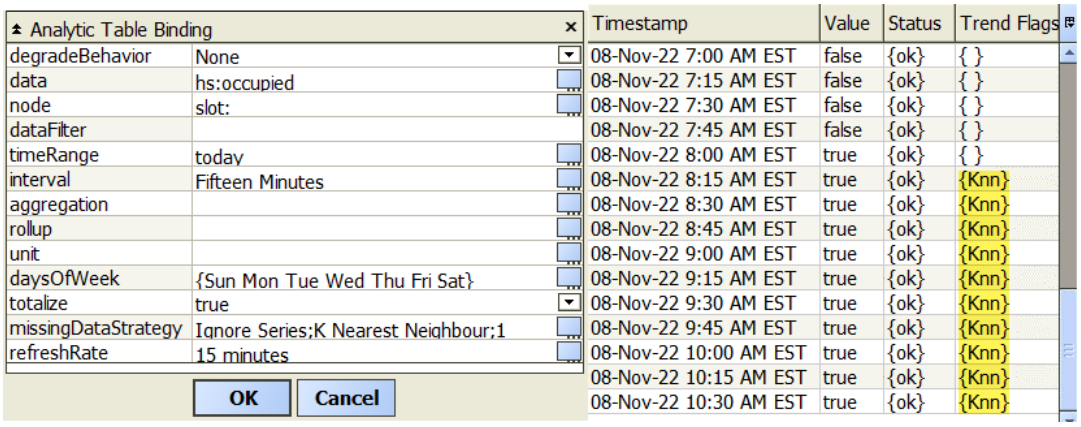
Figure 57 Missing data strategy configuration



The missing data strategy fills in the missing records at the requested interval based on the **`Interpolation Algorithm`**. The trend flags for the interpolated records provide a text indicator, such as `{Knn}`, which indicates that the framework used the K-nearest-neighbour interpolation algorithm to display the data.

Figure 58 Bound table with hs:hisInterpolate='cov' tag applied using K Nearest Neighbour

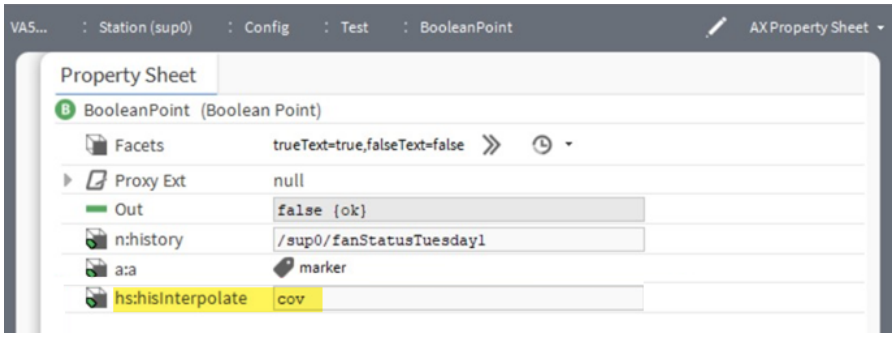
xNAFBoolea



COV configuration in a remote station

To identify the history as a COV, the associated point in the remote station must have an added COV tag.

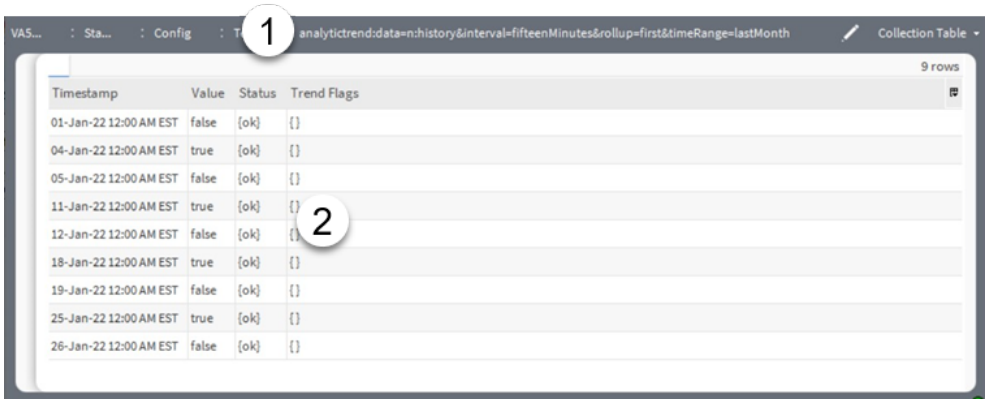
Figure 59 Point with COV tag



The example screen capture above shows a Boolean point with a COV tag (highlighted).

Consider the following analytic request and result for history data that does not include a COV tag.

Figure 60 History not identified as COV



Notice that this example uses an analytic trend ORD scheme (1) to identify the trend data (2). Analytic trend, analytic value, and analytic rollup are three different ORD schemes particular to the framework. It is important to understand how to use these ORD schemes, as they are the foundation of all framework functionality.

The ORD itself is:

`analytictrend:data=n:history&interval=fifteenMinutes&rollup=first&timeRange=lastMonth`

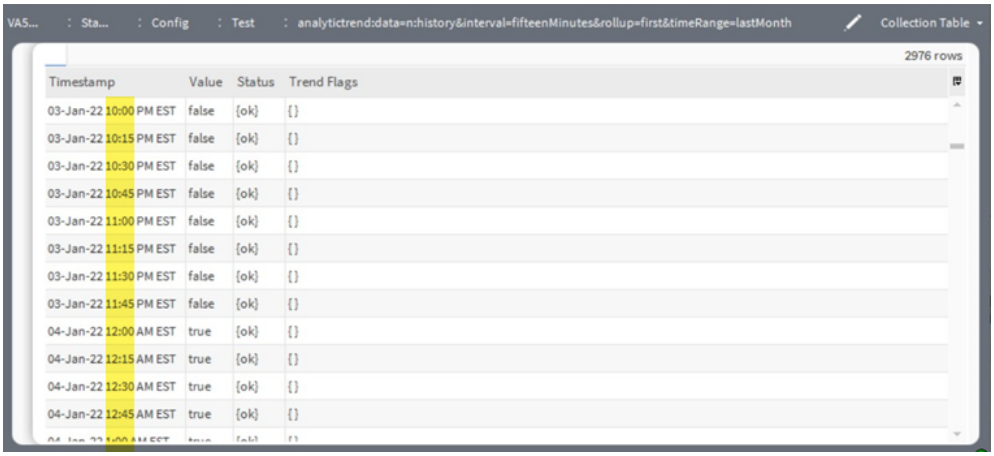
For clarity, it is asking for data identified by the `n:history` tag, using:

- an interval of fifteen minutes
- a rollup value of `first`
- and a time range of last month

Based on the `n:history` tag, the output (2) shows that for all of last month, the point went `true`, first thing Tuesday morning, at midnight, and it went `false` at the very end of the same day. No data were collected on any other day, since the value did not change. The ORD requests an interval of 15 minutes but instead of one row every 15 minutes, the request returns the same table that is visible on the COV history. This is because the framework is not aware that the history type for this point is COV.

To resolve this issue, add a tag that identifies the history type as COV. The framework applies the NEQL query defined by the **`Test Cov Neql`** property and returns `true` for the point—the point's history type is COV.

Figure 61 History identified as COV

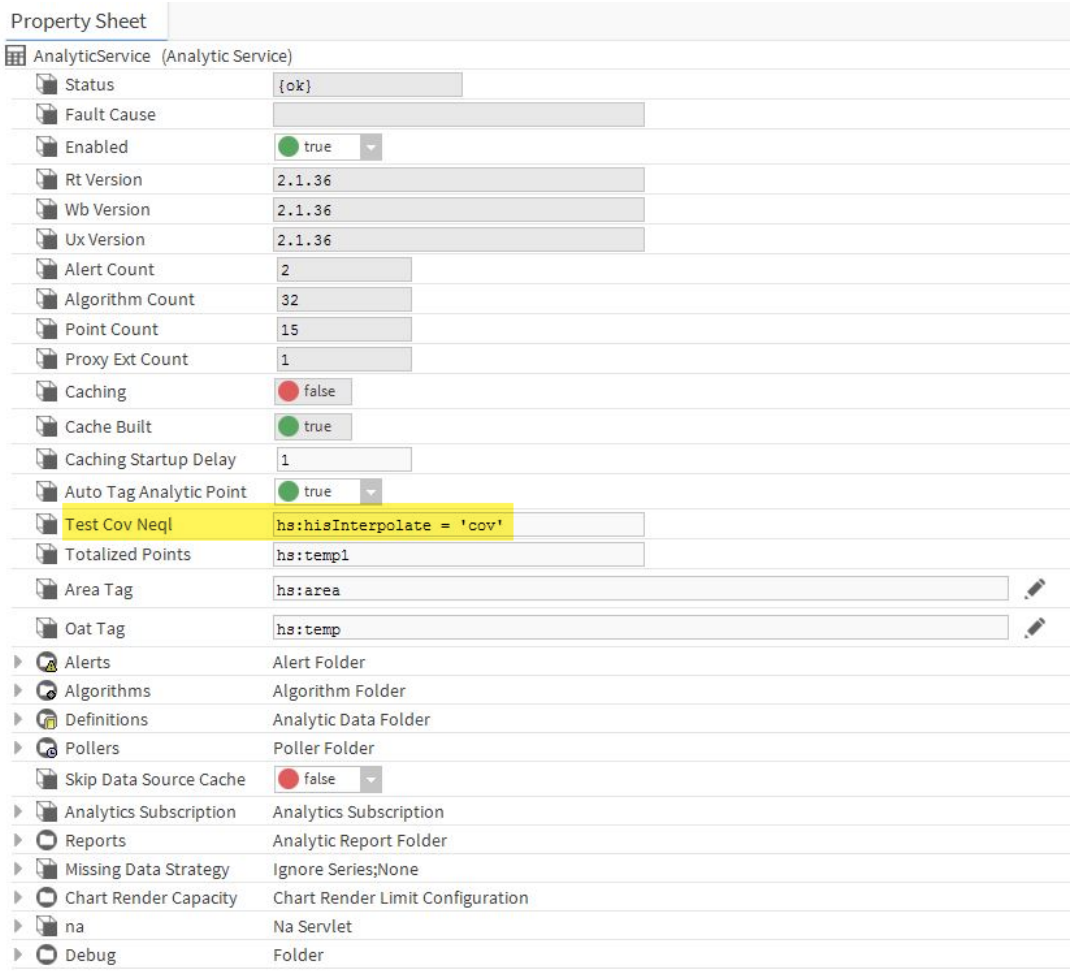


Now that the history is identified as COV, we get a 15 minute rollup (highlighted), showing the correct value for the timestamp.

All Analytic charts, tables, points, and alerts use ORD schemes. So, when something, for example, an alert that uses an algorithm, is not working the way you expect it to work, try the same algorithm using the same ORD scheme that the alert uses. This lets you see the data the alert is actually receiving as input.

Alternatively, you can use some other tag, and modify the NEQL query. You do not have to use Haystack tags for this.

Figure 62 Alternate Test Cov Neql property

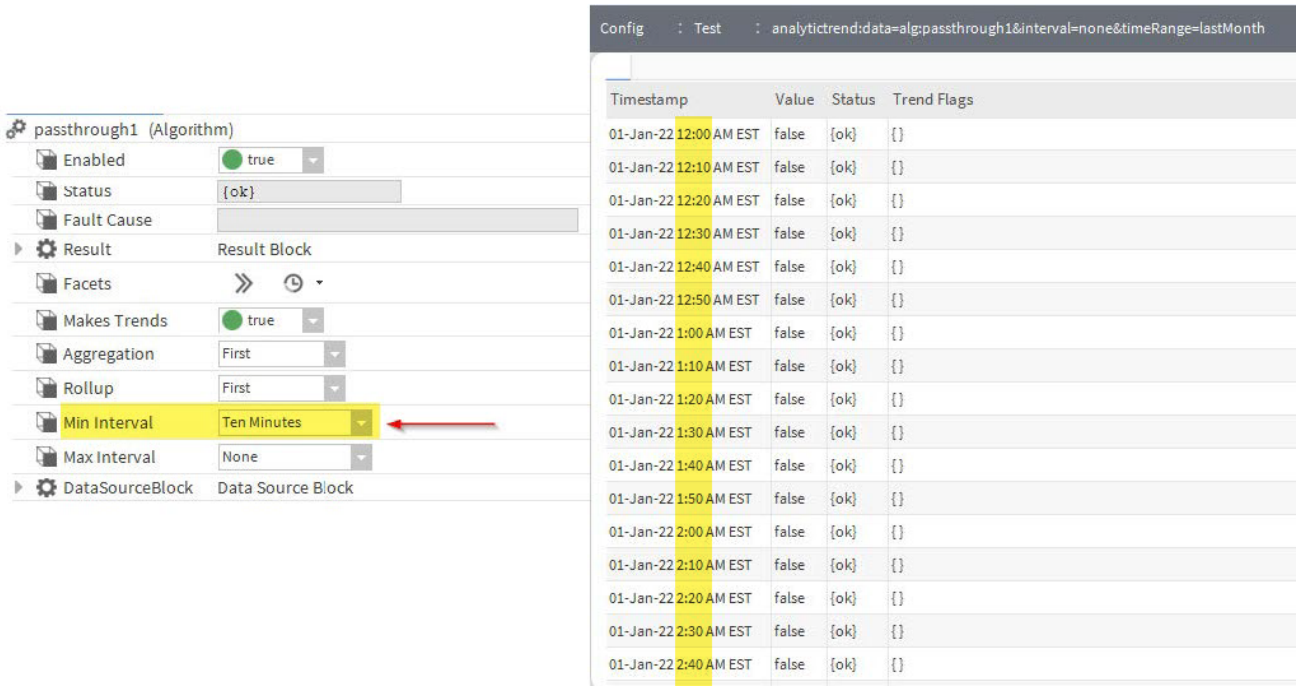


Algorithm Min and Max Intervals

It is easy to think of the algorithm object as a container for "the algorithm." But, it has its own set of properties to configure. One of those important properties is **`Interval`**.

Consider a simple algorithm that consists of a **Data Source Block**, a tag name on the **`Data`** slot, and a connection to the result block.

Figure 63 Algorithm illustrating interval properties



TIP: For debugging, it is best to use pass-through algorithms and Analytic ORD schemes to help you visualize the data that the algorithm is processing. Debug blocks do not provide the same insight, especially if you are just starting out with Niagara Analytics.

On the left side of the illustration above, the passthrough1 algorithm's **`Min Interval`** property is set to `Ten Minutes`. The right side of the illustration above shows a table with the results of resolving this ORD:

`analytictrend:data=alg:passthrough1&interval=none&timeRange=lastMonth`

The ORD is an analytic trend request processing the algorithm with an interval of `none` (special handling, which essentially means show every record) and a time range of last month. Even though the interval equals `none`, the resulting **Collection Table** shows one row every 10 minutes. This is because the **`Min Interval`** property on the algorithm object is set to **`Ten Minutes`**. This property is on the algorithm, not on any object that you see on the **Wire Sheet**.

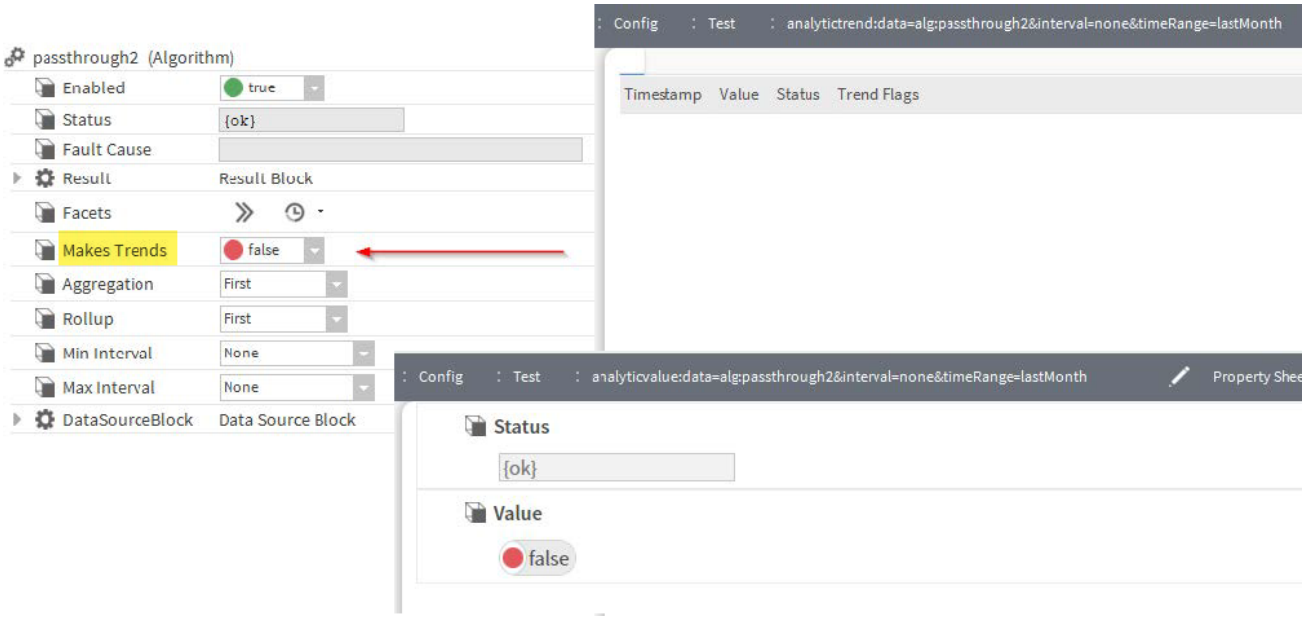
If you were to specify a **`Max Interval`** of `Hour` on the algorithm property sheet, and then specify an interval of `Day` in the Analytic Trend ORD, you would see a similar result. The **`Max Interval`** value on the algorithm would override what you specified on the ORD, and you would ultimately end up with a one hour rollup.

In summary, when you see an unexpected rollup interval, check to see that no one has set a **`Max Interval`**, or a **`Min Interval`** on the algorithm property sheet.

Algorithm Makes Trends property

The **`Make Trends`** property determines if an algorithm generates a constant value or a trend. These blocks have a **`Make Trends`** property: **Interval Count**, **Value Duration** and the four constant blocks, **Boolean**, **Enum**, **Numeric** and **String**. All default to `false`, which configures each block to return a constant value.

Figure 64 Algorithm illustrating Makes Trends property



When a **`Makes Trends`** property is `false` on the algorithm **Property Sheet**, the framework does not process any trend requests. A value of `true` allows the framework to process both value and trend requests using the algorithm.

TIP:

Since analytic requests can be configured as either value or trend requests when the **`Makes Trends`** property is set to `true`, if a trend request (chart or table) result is empty (no data displayed), verify that **`Makes Trends`** is not set to `false` on the algorithm **Property Sheet**.

Some algorithm blocks, like a **Numeric Constant**, do not map to a history in the station's database. These blocks may provide a value when the algorithm processes a trend request, but the block does not actually output a trend to pass to downstream linked blocks. If you link this type of block's output to the input of a downstream block that requires a trend, you must set the **`Makes Trends`** property `true`.

Best practices

The framework provides powerful tools for managing large and small buildings with increasingly sophisticated control features.

Running in a remote controller

When running the framework in a remote controller, such as the JACE-8000, configure your PX graphics bindings and number of points carefully. While a remote controller can handle hundreds of points and multiple PX bindings, memory has its limits. During a heavy processing period, the system may slow and report server session time out errors.

Refreshing cache memory

The framework requires memory based on the number of hierarchies, tags and points you configure for real-time and trend analysis. As you create hierarchies, tag points, build algorithms, and create alerts you should refresh cache memory frequently. To do so, right-click **Config→Services→AnalyticService** and click **Actions→Refresh Cache**.

Once your framework data model is configured and stable, you refresh cache only after changing the configuration in some way. On a system with hundreds of points and complex algorithms, a refresh can take a substantial amount of time. Plan any framework changes so that your station will have time to refresh cache.

Invoking the **Stop Caching** action interrupts a refresh. You could use this action if a **Refresh Cache** action is taking a very long time.

Algorithms

As you expand the framework's potential, keep these algorithm best practices in mind:

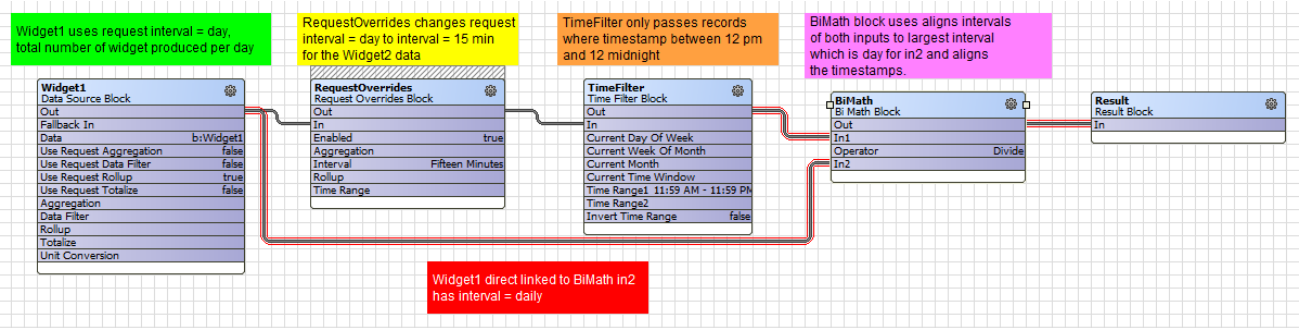
- Define data sources carefully. When you refresh cache, the system scans every node in the hierarchy to determine where to run algorithms. The data sources, as well as some configuration properties, determine which nodes to run against.
- Once an algorithm has executed, set up a poll timer/poll queue to run it again at a specified time.
- While it is physically possible to nest an algorithm inside of another algorithm, the preferred method is to configure the **`Data`** property of the **DataSourceBlock** using the `alg:otherAlgorithmName` syntax.
- To make an algorithm **Wire Sheet** that is particularly complex with lots of blocks and links easier to build and understand, you should organize your logic in **Logic Folder** blocks.

Interval alignment

If an algorithm block supports multiple trend inputs, such as BiSwitch, LogicFilter, Psychrometric, DeadbandFilter, etc., the block evaluates the trend inputs and synchronizes the intervals to match the largest interval. In addition, it aligns the timestamps.

Consider an algorithm, like the one below, where a RequestOverrides block changes the interval of the request from `Day` to `15 Minutes` for the logic branch connecting through the TimeFilter to In1 on the BiMath block.

Figure 65 Interval alignment example



In this example, the trend input to In1 on the BiMath block includes *15-minute* interval records during the time range 12 pm (noon) to 12 midnight but the logic branch connecting the **DataSourceBlock** directly to In2 on the BiMath block includes *daily* interval records.

The BiMath block needs to divide the records from the TimeFilter block (15-minute interval records) by the records from the **DataSourceBlock** (Daily-interval records). To do so, it must synchronize the intervals, otherwise the result will be meaningless. To apply the synchronization, the BiMath block changes the TimeFilter block output to **`Interval`** = `Daily` and applies the rollup function specified in the trend request. This aligns both the intervals and the timestamps to allow the BiMath block to perform the math calculation. The result from the output of the BiMath block contains daily records even though the TimeFilter block provided 15-minute interval records.

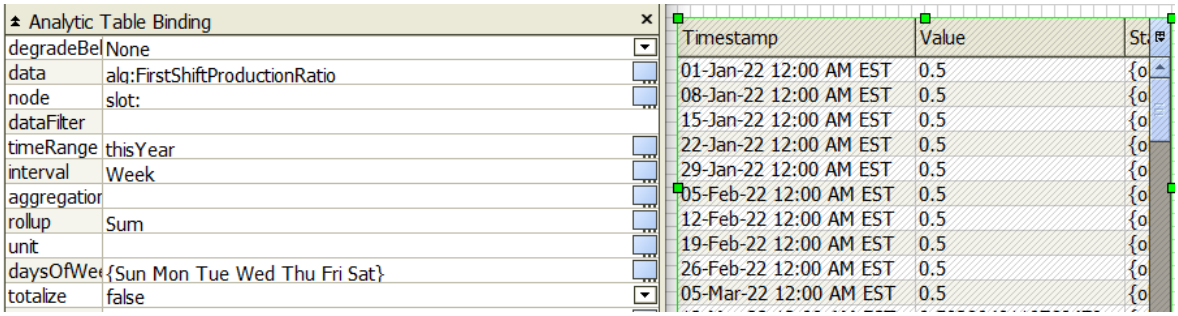
A Px view with a Bound Table and Analytic Table Binding calls the algorithm with a

`Timerange` of `thisYear`

`Interval` of `Week`

`Rollup` of `Sum`, which displays the percentage of total widget production during first shift.

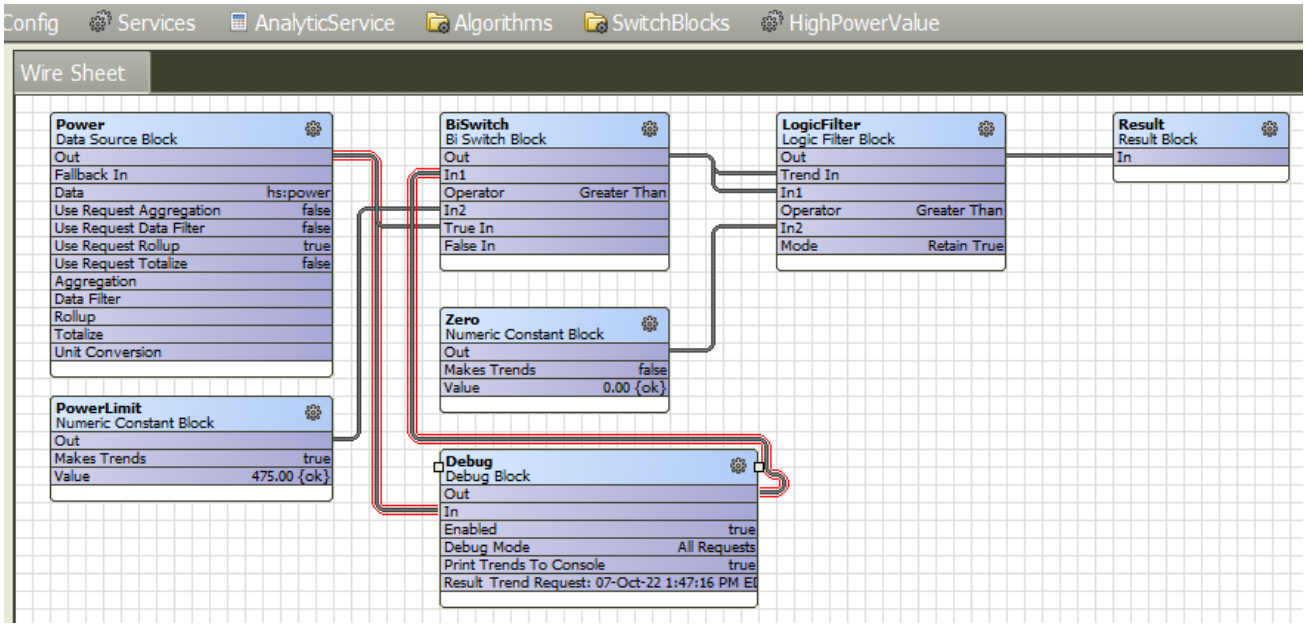
Figure 66 Analytic Table Binding using the interval alignment example



Debug block

To provide debug information for value and trend requests, you link this block in line between other blocks in an algorithm.

Figure 67 Example of a Debug block in an algorithm



When the **`Enabled`** property is `false` the **Debug Block** just passes any value or trend request through to any downstream linked blocks. When its **`Enabled`** property is `true` the **`Debug Mode`** property controls which requests are processed to update the block's **`Result`** property.

The following table documents how the **`Debug Mode`** property arrives at a **`Result`**.

Table 4 Debug mode results

Debug Mode	Description
All Requests	If **`Enabled`** is `true`, updates **`Result`** for every request.
All Trend Requests	If **`Enabled`** is `true`, updates **`Result`** with trend requests only.
All Value Requests	If **`Enabled`** is `true`, updates **`Result`** with value requests only.
Next Request	If **`Enabled`** is `true`, updates **`Result`** for the next request, then sets **`Enabled`** to `false`.

Debug Mode	Description
Next Trend Request	If **Enabled** is `true`, updates **Result** for the next trend request, then sets **Enabled** to `false`.
Next Value Request	If **Enabled** is `true`, updates **Result** for the next value request, then sets **Enabled** to `false`.

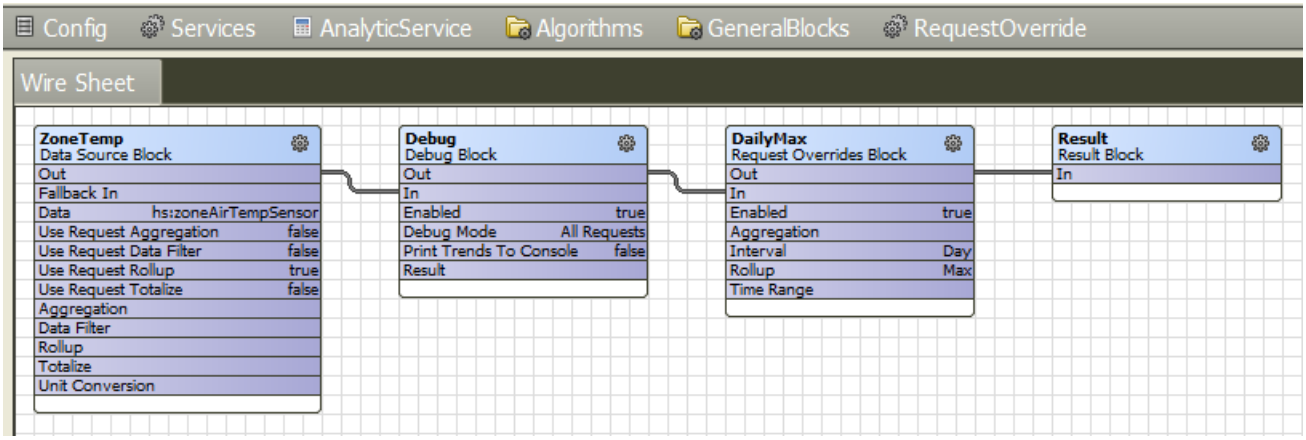
To help explain the algorithm results, the **Debug Block** updates the **Result** property with details that include: whether it was a trend or value request and what its configuration values were, such as time range, interval, missing data strategy, rollup function, aggregate function, etc. when it processes a request. The text below is an example of a **Result** property for a value request.

```
Value Request: 11-Oct-22 12:10:06 PM EDT
  aggregation = first
  data = alg:HighPowerBoolean
  dow = 7f
  interval = fifteenMinutes
  mdStrategy = Ignore Series;None
  node = local:|station:|slot:/AnalyticPlayground/HighPowerBoolean
  rollup = first
  seriesName =
  statusFilter = {ok}
  timeRange = today
  timeZone = America/New_York
  user = admin
  [Result] 11-Oct-22 12:10 PM EDT 952.8933040641983 {ok}
```

Even though this is a value request, as indicated by the first line of the result, the analytic request context contains default values for properties that were not explicitly set in the request, such as timeRange = today, interval = fifteenMinutes, rollup = first and aggregation = first.

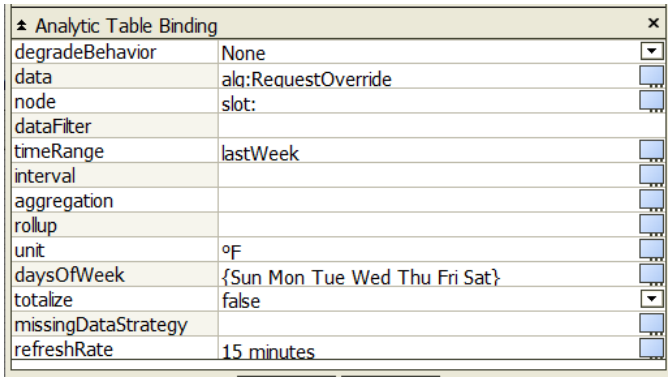
Some analytic blocks are designed to modify the incoming request properties. This might affect the **Result** property of an upstream **Debug Block**. Consider the algorithm below with a **Debug Block** linked between a **DataSourceBlock** and a **Request Overrides Block**. The **Request Overrides Block** is configured to override the **Interval** = `Day` and **Rollup** = `Max` in the upstream analytic request.

Figure 68 Example of a Debug Block in a sequence that overrides upstream property settings



A Px view with an Analytic Table Binding sends a trend request without specifying the **Interval** or **Rollup** function.

Figure 69 Table binding for a trend request



The trend request time range is `lastWeek` so the unconfigured default **`interval`** should have been `6 hours` and the unconfigured default **`rollup`** function is `first`. The **Debug Block**'s **`Result`** property shows the interval = day and rollup = max because the downstream **Request Overrides Block** is overriding those properties in the incoming request.

```
Trend Request: 11-Oct-22 4:40:56 PM EDT
  aggregation = first
  data = alg:RequestOverride
  dow = 7f
  interval = day
  mdStrategy = Ignore Series;None
  node = local:|station:|slot:/AnalyticPlayground/RequestOverride
  rollup = max
  seriesName =
  statusFilter = {ok}
  timeRange = lastWeek
  timeZone = America/New_York
  user = admin
```

Frequently-asked algorithm questions

Algorithms customize data analysis. These may be some of the questions you are asking about them.

How do the alert Time Range and Interval properties affect the calculation of an algorithm?

The **`Time Range`** defines the history records to process. The **`Interval`** works in conjunction with the **`Roll-up`** property to configure whether all individual records should be processed or the data should be rolled up into fewer records, which are then processed.

For example, if **`Time Range`** is yesterday and **`Interval`** is 15 minutes, the algorithm processes each of the 96 records. If **`Time Range`** is yesterday and **`Interval`** is one hour, the query applies the rollup function to the records causing the algorithm to process each of 24 records.

How do I set up an algorithm and alert with data sources that use a mix of COV and Interval history extensions?

Make sure the data are tagged to identify the history as a COV. Then you could implement a **Data Definition** with a missing data strategy to interpolate missing records using the K nearest neighbor. When there is no COV record matching the interval history record, the analytic engine uses the last COV record prior to the timestamp.

How do I get an alert to return to normal once the alert condition has cleared?

If the alert is assigned to a cyclic poller, the next time it executes and does not detect the fault condition it should return to normal.

What are the default settings on the alert property sheet slots for Time Range, Aggregation, Interval, Rollup, and Totalize?

- **`Time Range`** defaults to `Today`.
- If **`Aggregation`** is not enabled in the binding/settings window, the **Data Definition** defines its value for all chart bindings, reports and tables, returning the logical "and" for Boolean values. You configure **`Aggregation`** using a check box (if optional) or a drop-down list.

  For information about all the places where you can configure the aggregation function, refer to Aggregation configuration, page 35
- When a request does not specify the default **`Interval`**, the system calculates one based on the time range:
  - If the time range is >= one year, the interval is one month.
  - If the time range is >= one month, the interval is one day.
  - If the time range is >= one week, the interval is six hours.
  - If the time range is >= one day, the interval is fifteen minutes.
  - If the time range is >= twelve hours, the interval is five minutes.
  - If none of the conditions above match, the interval is one minute.
- You configure **`Rollup`** using a check box (if optional) or a drop-down list. As a drop-down list, **`Rollup`** provides these options:

  If rollup is not enabled in the binding/settings window, the rollup value configured in the Data Definition applies to all chart bindings, reports and tables.

  `And` returns the logical "and" of Boolean values.

  `Avg` returns the statistical mean, which is determined by calculating the sum of all values and dividing by the number of values.

  `Count` returns the total number or quantity of values in a combination. If you request this value on a binding in a PX view, the system counts the number of values based on the properties defined by the data source block and the algorithm's **Property Sheet**.

  `First` returns the first value in the combination.

  `Last` returns the last value in the combination.

  `Max` returns the highest value in the combination.

  `Median` returns the value in the middle of a sorted combination—the number that separates the higher half from the lower half.

  `Min` returns the lowest value in the combination.

  `Mode` returns the statistically most frequently occurring number in the combination.

  `Or` returns the logical "or" of Boolean values.

  `Range` returns the statistical difference between the largest and smallest values in the combination.

  `Sum` adds together all values in the combination resulting in a single value.

  `Std Dev` calculates the standard deviation of the values in the combination.

  `Load Factor` calculates the average divided by peak (Max) value.

  For information about all the places where you can configure the rollup function, refer to Rollup configuration, page 39
- **`Totalize`** is a Boolean property or an ORD scheme parameter. It:Turns on (`true`) and off (`false`) value accumulation.

By default, the framework totalizes (accumulates) all consumption history values in charts, tables and reports. To prevent cumulative values, disable this property (set it to `false`).

What specifically does the Totalize property do?

It tells the analytic engine whether to calculate a delta value or to totalize values. This applies to trend requests where the underlying data might be totalized or delta logged.

Haystack defines the `hisTotalized` tag as indicating values, which are a continuous stream of totalization. History data read and should be normalized by computing the delta.

You may need to experiment with the **`Totalize`** property to fully understand how it functions. But clearly it has to do with consumption values, like KWH, or gallons, and how they are logged. It is beneficial to log gallons consumed since the last record, rather than gallons since the meter was installed, because otherwise, there is a risk of "rolling over the odometer." Totalization adds up the values, so that you see how much you have consumed over a period of time, rather than calculating a rate of consumption, which is technically what you are logging in an ideal scenario.

How do I accommodate trends with offset time stamps. How does that affect an algorithms operation?

One way to do this is to apply an interval and rollup function to align records with the smallest timestamp possible.

Another option is to implement a missing data strategy and enable interpolation with the K nearest neighbor.

There is also a timestamp offset block, which you can use in an algorithm if you actually need to compare data from one timestamp to data from another history with an offset timestamp.

Chapter 4 Data visualization

Topics covered in this chapter

- Rollup and aggregation
- Changing rendering limitations
- Automatic conversion of values in tables
- Historical comparisons using baselineValue
- Configuring a baselineValue in charts and tables
- Pre-defined charts
- Reports

Charts take large volumes of detailed data and make them usable to detect patterns and solve problems. A set of pre-defined charts can work together to accomplish these goals. You can also create your own charts.

Rollup and aggregation

Rollup and aggregation are features the framework uses to combine data for meaningful analysis. A rollup combines multiple adjacent rows of historical data into a single row. Aggregation applies to any request for individual data by tag, such as `hs:power` or an algorithm where multiple data sources are found and processed. Trend requests may find multiple data sources, each with histories and the aggregation function combines those historical values as well. Rollups and aggregation control how an algorithm queries the database for input data.

A rollup uses a function (sum, average, etc.) to combine historical data. It allows you to view dissimilar histories at common intervals. For example, if one point samples data at five-minute intervals, and another samples at 10-minute intervals, you can use rollup to calculate and compare their values at, 10-minute intervals.

An aggregation uses a function (sum, average, etc.) to combine multiple data source values into a single value. If not explicitly configured, the framework uses the first function for both rollup and aggregation.

Both features share the same set of functions

The framework tends to convert primitive data types in the background if necessary. For example, using an And function for aggregation of numeric points returns a false (0) if any numeric value is <=0. The And, Or functions work on numeric and enum (ordinal) values. The Math functions also work on Boolean values using 0 for false and 1 for true.

Function	Description
And	Logical and.
Avg	Calculates the sum divided by the count.
Count	Returns the number of values.
First	Returns the initial value in the set.
Last	Returns the final value in the set.
Load Factor	Returns the average value divided by peak (Max) value.
Max	For numerics, this is the greatest value. For Booleans, false = 0 and true = 1. For enums, this returns the greatest ordinal.
Median	Returns the value in the middle of a sorted combination—the number that separates the higher half from the lower half.
Mean	Returns the arithmetic mean (average) of the values in the data source(s).

Function	Description
Min	For numerics, this is the smallest value. For Booleans, false = 0 and true = 1. For enums, this returns the smallest ordinal.
Mode	Returns the statistically most frequently occurring number in the combination.
Or	Logical or.
Std Dev	Returns the standard deviation of the values in the combination.
Sum	Adds all values together.

When the difference between two values matters

Some of the functions used by aggregation and rollup may not make sense for certain values. For example, the sum of KWh for a group of points yields the total energy consumed; however, the average of those same points yields a meaningless number. In another example, summing air temperature readings may not yield a useful number. You may be more interested in the delta (change) that occurs between the historical values. To have the system calculate this value, make sure data source is tagged with the `hs:hisTotalized` marker tag and the request totalize property is false.

Best practice

As you configure the visualization of values and trends, experiment with the rollup and aggregation properties on the binding. If you get a result you do not expect, consider the settings for these properties.

Changing rendering limitations

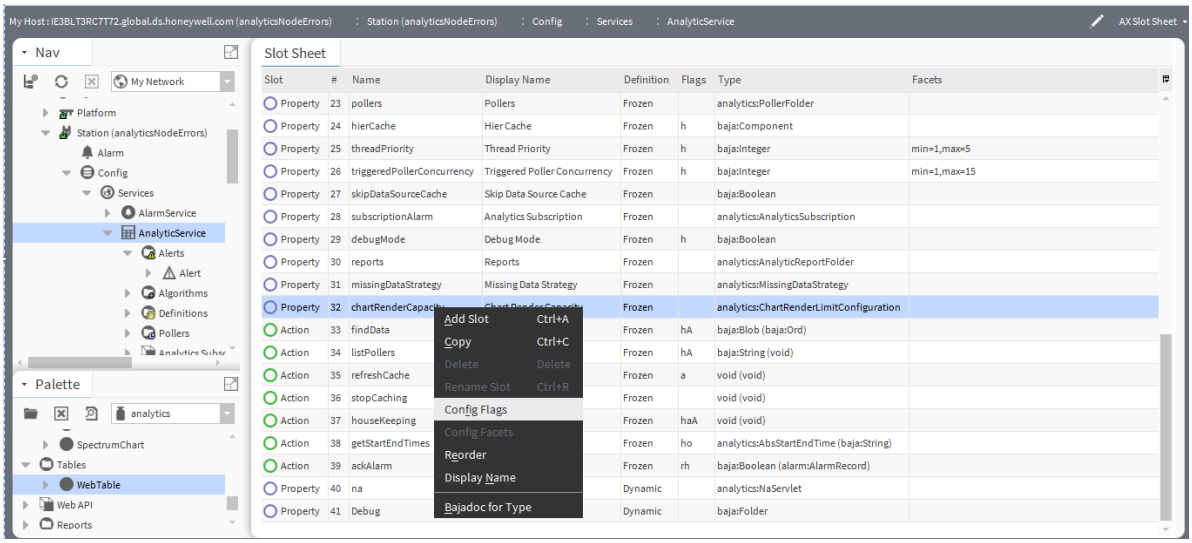
The number of records included in a chart or report relates directly to system speed and performance. By default, the framework limits this number. The default values for each chart and report were set based upon extensive testing in the laboratory. The configuration of these values is hidden so that they cannot be changed unintentionally.

Prerequisites: You understand the impact on framework performance of increasing the number of records rendered on charts and reports. You are working in Workbench.

CAUTION: Increasing the number of records rendered can slow system performance to an unacceptable level.

Step 1 Expand **Config→Serivces**, click the **AnalyticService**, and select `AX Slot Sheet` from the drop-down list in the upper right corner of the view.

The slot sheet opens.



Step 2 Right-click the **chartRenderCapacity** property, select **Config Flags** from the menu, click to de-select the **Hidden** check box, and click **OK**.

Step 3 Select **AX Property Sheet** from the drop-down list in the upper right corner of the view, modify the **chartRenderCapacity** value as desired and save.

Automatic conversion of values in tables

When the system converts one unit it automatically shortens the unit's value and adds a suffix. This improves the readability of large data and data with multiple decimal places in report tables.

This automatic rescaling of values applies to all large or small values regardless of whether they have Metric or English Standard facets applied.

For example, a value of 16100 converts in the table as 16.10k.

Figure 70 A table with converted values

Time Of Day	NumericWritable (W per degree day)
00:00:00	16.10k
06:00:00	62.45k
12:00:00	76.53k
18:00:00	26.53k

Number conversion

These abbreviations are for values that are greater than 1,000.

Suffix symbol	Name	Positive orders of 10
T	trillion	1,000,000,000,000
G	billion	1,000,000,000
M	million	1,000,000
k	thousand	1,000

Decimal number conversion

These abbreviations are for values that are less than 1.

Suffix symbol	Name	Negative orders of 10
m	thousandth	0.0001
µ	millionth	0.000 001
n	billionth	0.000 000 0001
p	trillionth	0.000 000 000 001

Historical comparisons using baselineValue

The baselineValue feature compares the two historical values of a single point. The existing Analytic Web Chart and Analytic Web Table are updated and designed in HTML5 to visualize the baselineValue and historical data in the browser.

The procedure for configuring the baselineValue feature in charts and Web Tables is an additional option for predefined analytics charts and analytics web tables. You can configure the baselineValue feature in the workbench and browser. Based on the Time Range selected in the chart, the baselineValue time range is adjusted automatically to the baselineValue time range type.

These charts and table support the baselineValue feature:

- **Analytic Web chart**: the multipurpose chart generates line, bar, and area charts. In addition, now you can enable the baselineValue option and compare the historical data of each point.
- **Analytic Average Profile Chart**: the chart represents an average of a data value over a period; with the baselineValue feature, you can compare averages for two different periods.
- **Analytic Load duration Chart**: the chart monitors the duration of peak demand. Comparing peak demand for two periods is now possible with the baselineValue feature.
- **Web Table**: you can enable the baselineValue option and compare the historical data of each point in the web table.

Configuring a baselineValue in charts and tables

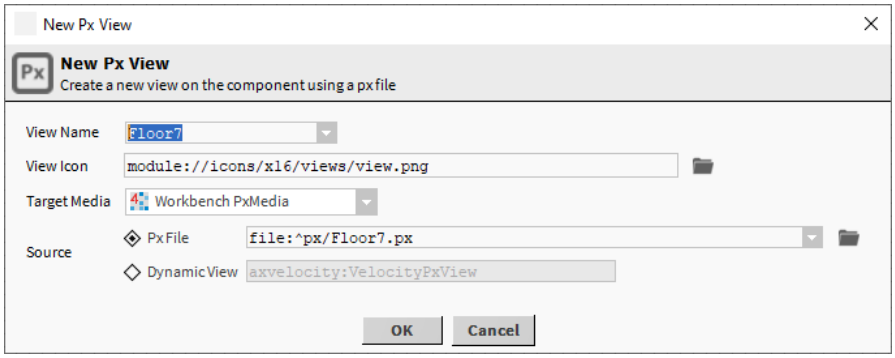
This procedure explains configuring baselineValue in the **AnalyticWebChart**, **AverageProfileChart**, and **LoadDurationChart**. You can also use a **Web table** to compare individual values of historical data.

Prerequisites: You are working in Workbench connected to a station whose database contains a substantial amount of historical data.

This procedure explains configuring the baselineValue feature by using the example of a chart. Similarly, you can compare the values using a **Web table**.

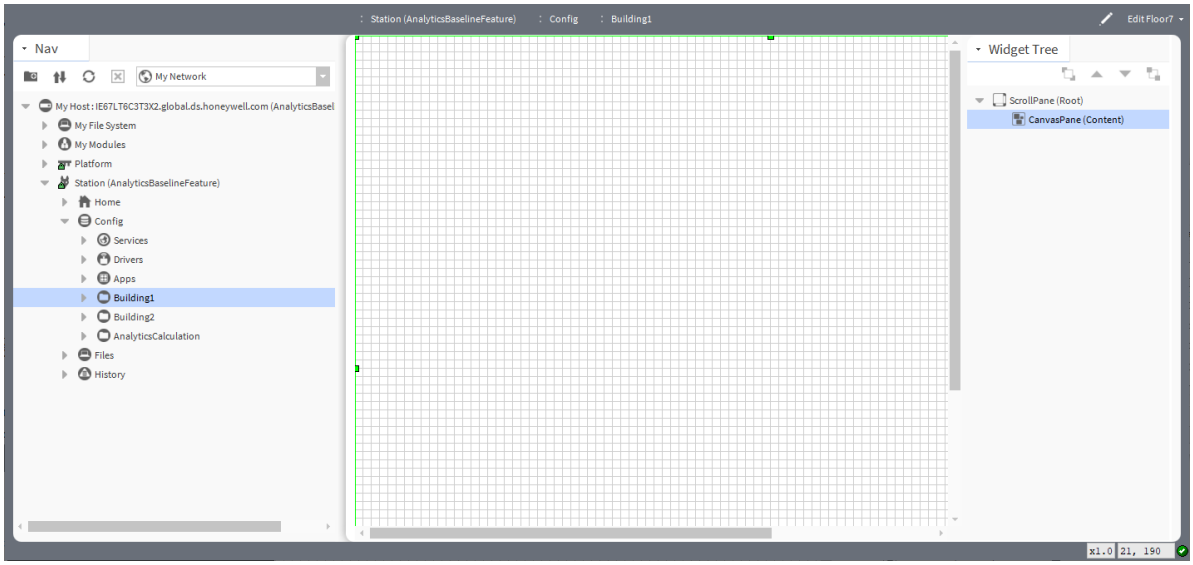
Step 1 Navigate to the logic folder in the view pane, and right-click logic **Views→New View**.

The New Px View opens



Step 2 Select the properties and click **OK**.

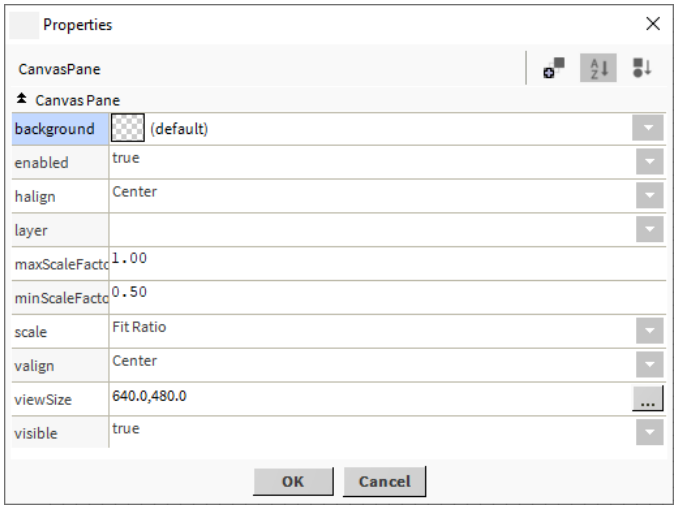
The Px editor Opens.



Step 3 To change the view size property, right-click the background canvas, and select **Edit Properties**.

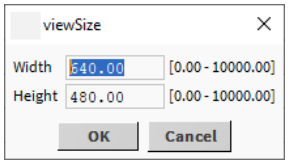
The default canvas size is 1000 by 800 pixels. You may find this size too big for a chart that compares a baseline value with another value.

The **Canvas Pane** properties window opens.



Step 4 To decrease the **viewSize** in pixels, click ....

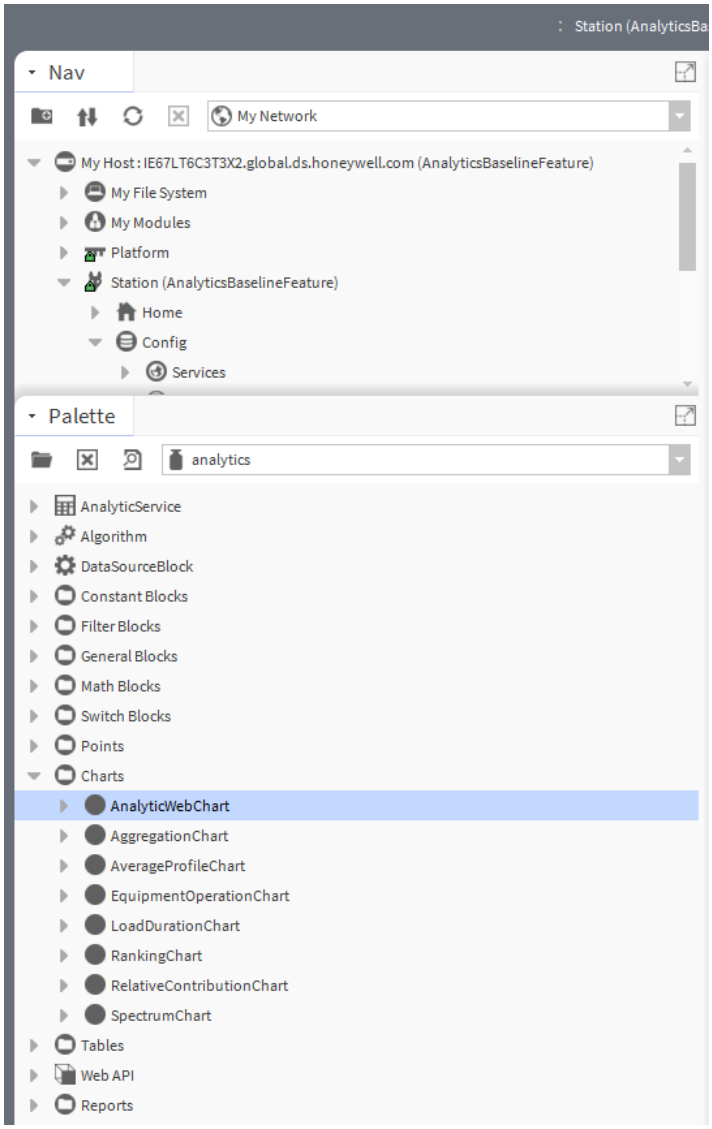
The viewSize window opens.



Change the pixel size to 640 x 480 pixels.

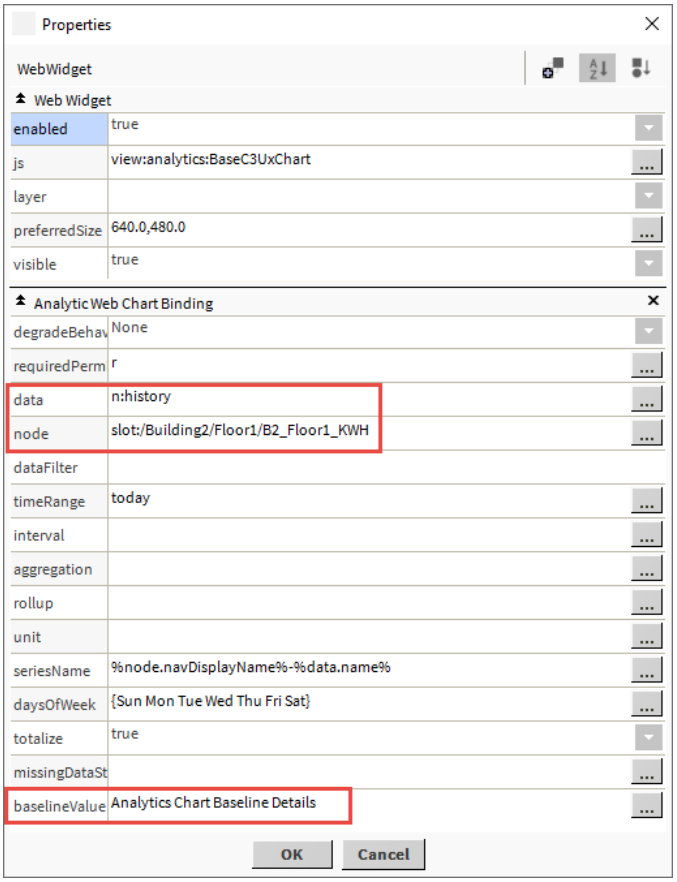
Step 5 Open the **analytics** palette.

The **analytics** palette opens in the sidebar.



Step 6 Expand the **Charts** folder and drag an **AnalyticsWebChart**, **AverageProfileChart**, or **LoadDurationChart** to the Px canvas pane.

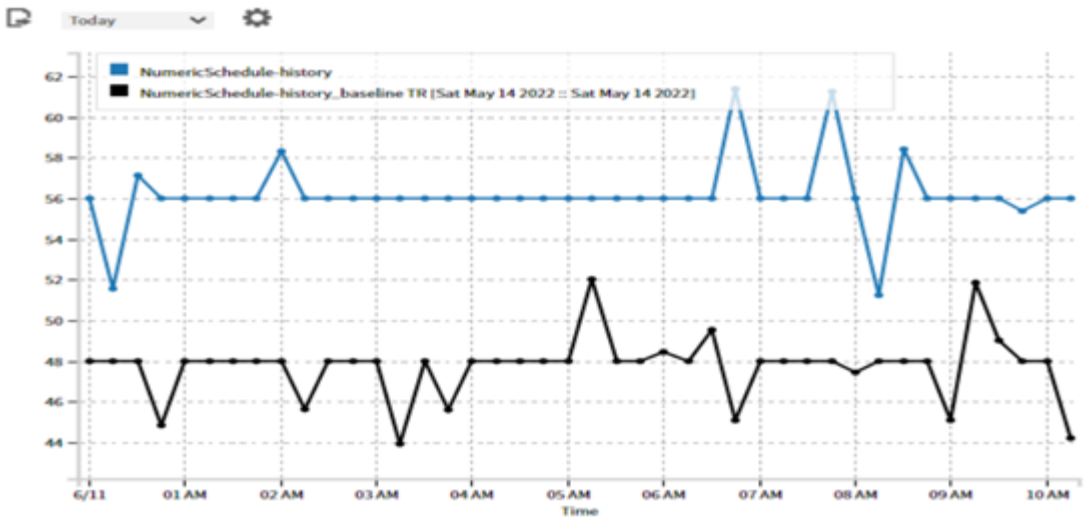
The Web Widget **Properties** window opens.



Step 7 Configure the following properties and Click **OK**.

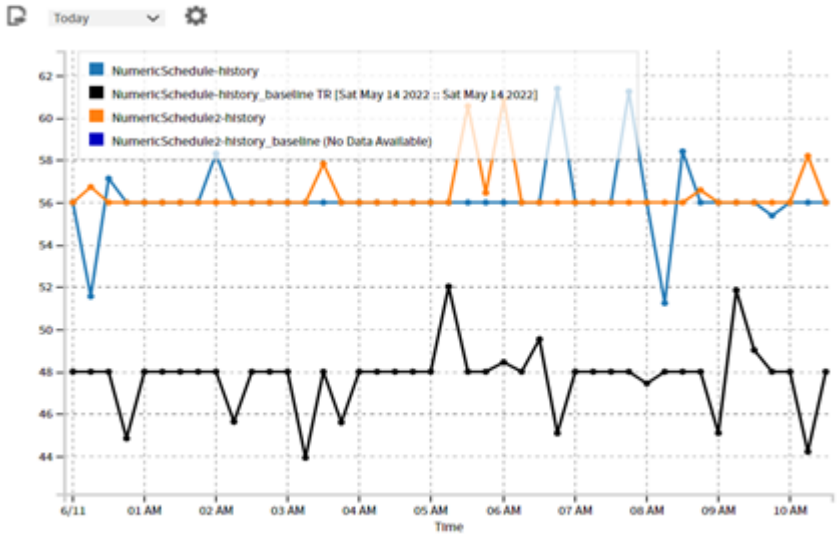
- **`Node`** selects the ORD for the desired slot.
- **`Data`** specifies the tag used to retrieve data.
- **`baselineValue`** enables the baselineValue feature.

The chart with comparing two historical values opens.



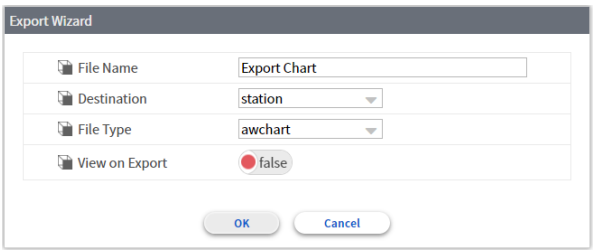
Step 8 Select the time range form the right corner drop-down to visualize the different patterns of historical data.

You can observe the chart in browser view also. You can add multiple points and observe different patterns for all the points.



Step 9 To export the chart click .

The **Export Wizard** window opens.



Step 10 Configure the properties and click **OK**.

- **`File Name`** provides the file name for the resulting file per your choice.
- **`Destination`** selects the destination of the file to save.
- **`File Type`** selects the `awchart` as file type from the drop-down list.

You can view the exported chart in Workbench or a browser.

Pre-defined charts

The pre-defined charts work both in Workbench and a browser.

Here are some needs a Facility Manager, Data Center Manager, or Utility Manager may have and which chart to use to meet each need:

The Need (what you might want to do)	The Recommended Chart to use
• Combine data from separate nodes into single figures. • View spikes in power usage at a point in time.	**AggregationChart** combines values across a selected time range showing the average for each value. For example, if you select This Week, the chart reports the average for each day of the week.
• View power spikes grouped by day. • Evaluate the average monthly temperature in a storage area for a period of a year.	**AverageProfileChart** plots a graph that shows the average of each value for all bindings grouped by time.
View equipment status (when was the power on and when was it off?)	**EquipmentOperationChart** shows when a piece of equipment is powered on and off. providing insight into equipment operating patterns.
• Compare the same value across multiple locations. • Determine the energy required when adding additional equipment.	**RankingChart** uses vertical bars to compare binding values from lowest to highest (left to right).
Determine for how long a value was at a specific level (high/low), such as to view the number of hours that a generator generated specific kilowatts of power per hour.	**LoadDurationChart** plots load versus duration.
• Drill down to the specific time when something occurred. • Observe temperature or pressure over a period of time.	**SpectrumChart** uses pattern recognition techniques and color coding to illustrate multiple aggregated values obtained from the same data source. The colors on the chart quickly identify trouble spots for further investigation.

Details about each chart and a screen capture of each are in the *Niagara Analytics Framework Reference*

Configuring a pre-defined chart

The pre-defined charts work both in Workbench Px Views and as well as in a browser (Web Charts). This topic provides basic instructions using framework examples.

Prerequisites: The **analytics** palette is open.

Step 1 Right-click your logic folder in the Nav tree and click **Views→New View**.

The **Px Editor** opens.

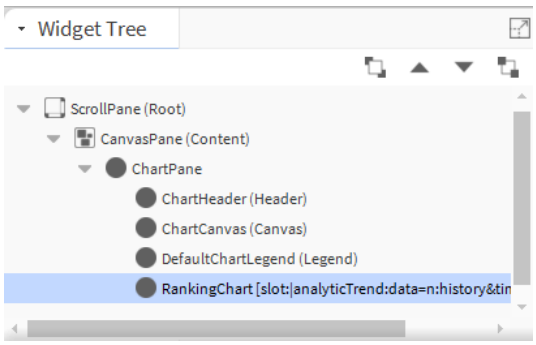
Step 2 Select the background canvas, and change the **viewSize** property to 640 x 480 pixels.

The minimum height for an Aggregation chart is 560 pixels. When set to 550 or less, the **Time Range** is not available for this chart.

Step 3 Scroll down to the bottom of the **analytics** palette, expand the **Charts** folder, and drag a chart to the **Px Editor**.

Step 4 Drag the chart to fill the canvas.

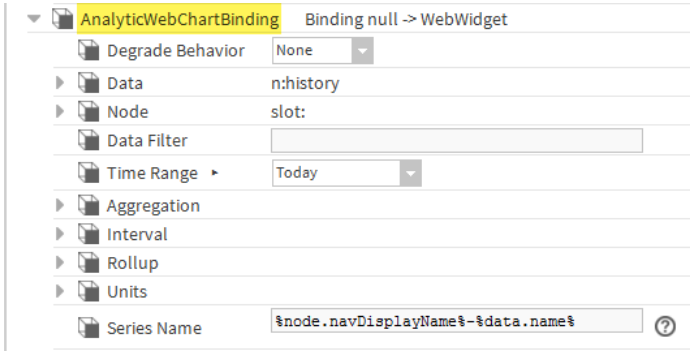
The system populates the tabs to the right of the window, one of which is the **Widget Tree**.



The screen capture is for a **Ranking Chart**.

Step 5 Double-click the widget.

The **Properties** window for the chart opens.



The properties related to the framework are in the **AnalyticChartBinding** or **AnalyticWebChartBinding** section of the window.

Step 6 For an Aggregation chart, confirm that **`Time Range`** is available.

Step 7 Configure at least the **`Data`** property by clicking the chooser button (...) and selecting a data source tag from the drop-down list.

The remaining properties default to current values.

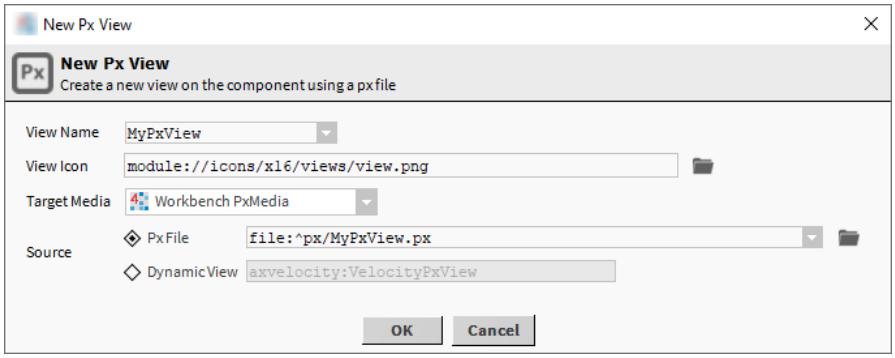
Creating a new Px view

In core Niagara, a widget (label or chart) is associated with a data source (object) using a binding. This binding defines an ORD property that identifies the location of the object that updates and animates the widget. The framework replaces the ORD with a tag, which causes the binding to collect data from all points tagged with the same tag. You set up a Px View in Workbench to visualize framework data the same way you set up a regular Niagara Px View. This topic provides basic instructions using framework examples.

Prerequisites: You are working in Workbench and are familiar with how to use the PxEditor.

Step 1 Right-click your equivalent of a **Logic** folder, click **Views→New View**, and assign a view name.

The **New Px View** window opens.



Step 2 Assign a **`View Name`** and click **OK**.

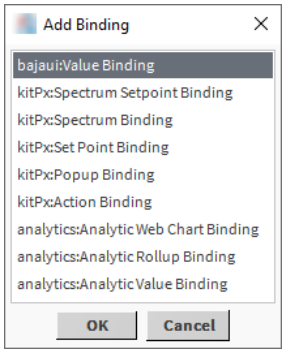
The **Edit** view of the PxEditor opens.

Step 3 Right-click the canvas pane and click **New→Label** (or duplicate a similar label you already created) and expand the size of the label.

The system creates an unbound label and populates the **Properties** tab at the bottom right corner of the **Px Editor** view and opens the **Properties** window for the **Label**.

Step 4 Click the add binding button ( ) at the top of the window.

The **Add Binding** window opens.

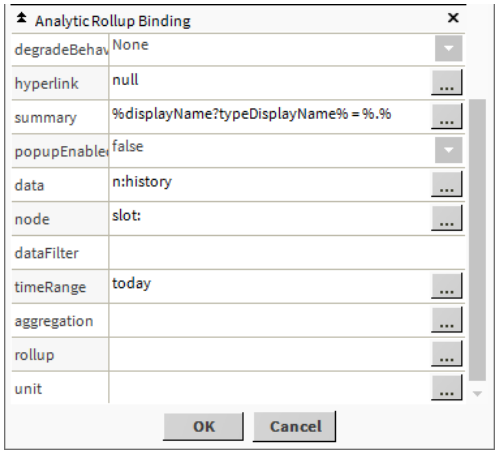


Step 5 To an unbound label, select one of the **`analytics:`** bindings and click **OK**.

- The **`analytics:Analytic Rollup Binding`** combines (rolls up) multiple instances of a value into a single value.
- The **`analytics:Analytic Value Binding`** reports the current, real-time value of the point.

The system associates the binding with the label and displays the binding properties.

Step 6 Scroll down in the **Properties** window until you see the binding properties for the **`analytics:`** binding you selected.



The example shows the **Analytic Rollup Binding** properties. The first four properties: **degradeBehavior**, **hyperlink**, **summary**, and **popupEnabled** are familiar Workbench properties. The rest are unique to the framework.

Any request (query) from the database requires you to configure at least the **data** source. The framework pulls data from one or more points with this tag.

The other values are optional depending on the binding request. A value binding always deals with current values. The system ignores any setting of the **rollup** property for a value binding. An aggregation requires the starting **node** when aggregating multiple data sources. This node is usually a container that identifies a building or geographic location.

Step 7 Use the file finder button (...) and component chooser to populate the **data** and **node** properties.

NOTE: Unlike building a **Px View** in core Niagara, you do not select a point to establish an ORD. Your tag and node selections determine the point(s) to use. These properties take the place of a traditional ORD.

Step 8 Do one or both of the following:

- If you are configuring a rollup value based on historical data, click the **rollup** property, enable **Use This Value**, select (from the drop-down list) the function (count, first, last, avg, etc.) to use to roll up the data and click **OK**.
- If you are aggregating multiple data sources, click the **aggregation** property, enable **Use This Value**, select (from the drop-down list) the function (count, first, last, avg, etc.) to use to aggregate all values into a single resulting value, and click **OK**.

Both rollup and aggregation default to their preferred settings in the data definition that is associated with each tag.

Step 9 If you started this procedure from an unbound label, scroll up in the **Properties** tab, right-click **text** and click **Animate**.

The **Animate** window opens with the default format set to `%.%`. In coreNiagara you might configure this property to read `%out.value%`. The equivalent in the Niagara Analytics Framework is `%value%`.

Step 10 Change **Format** to `%value%` and click **OK**.

The framework returns the value and not the point status.

TIP: If the chart includes a large number of bindings, and some bindings yield small quantities of data (very near each other on the chart), the labels may overlap and become unreadable in a PDF. To fix this problem, increase the size of the chart to allow space for label placement without overlaps.

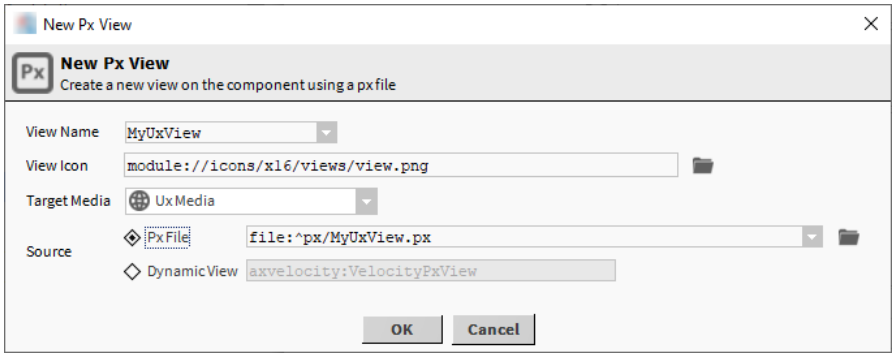
Creating a new Ux chart

Ux (User Experience) charts are web charts. Programmed using HTML5, these charts are designed to work best when viewed in a browser. They also work as Px views and they use the standard Workbench PxEditor.

Prerequisites: You are working in Workbench and are familiar with how to use the PxEditor.

Step 1 Right-click your equivalent of a **Logic** folder and click **Views→New View**.

The **New Px View** opens.



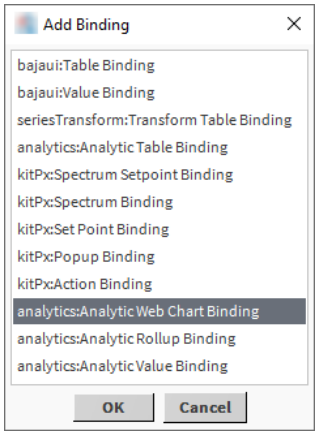
Step 2 Assign a **`View Name`**, choose `Ux Media` for **`Target Media`** and click **OK**.

The **Edit** view of the PxEditor opens.

Step 3 Open the **`webChart`** palette, drag a **`Chart`** component to the wire sheet and position it for easy viewing.

Step 4 Click the add binding button ( ) next to **`WebWidget`** in the **Properties** pane.

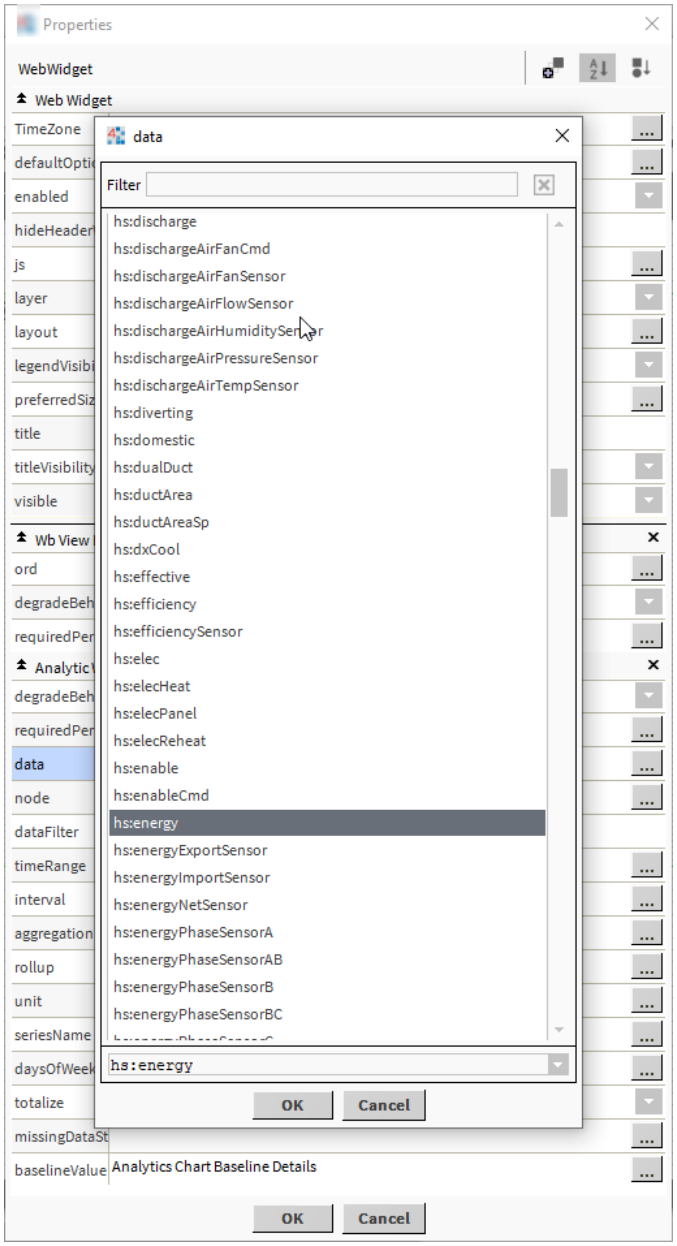
The **Add Binding** window opens.



Step 5 Select `analyticsAnalytic Web Chart Binding` and click **OK**.

Step 6 Double-click the chart component.

The **Properties** window opens



.

Step 7 Configure the **data** property to display the desired information, such as `hs:energy` to show electrical energy values while leaving other properties set to their default values.

The table displays all values plotted throughout the day (today is the default interval).

Step 8 To view the resulting chart, click the Toggle View/Edit Mode icon (

Workbench renders the chart as a Px graphic.

Step 9 To view how the chart looks in a browser, click the Toggle Browser Preview Mode icon ( ).

Workbench renders the chart as it will look in a browser.

Step 10 Using the drop-down list at the top of the chart component, change the interval to `Last Week`, `Last Month`, etc.

The data displayed in the chart change.

Step 11 To configure multiple plots on the same chart, edit the binding by changing the node to a component in the station that represents a building that has an electrical energy point with an `hs:energy` tag, add another **Analytic Web Chart Binding** to the **Web Chart** widget, configure its node for a component that represents a different building and edit its properties: **`data`** and **`timeRange`** to match the first binding.

For example, you could plot values for two different networks or two different buildings on the same graph. This would work for comparing energy usage or possibly space temperature in a couple of different rooms.

Step 12 To save the configuration, click the Save icon ( ).

When viewing the chart outside of the PxEditor, the Settings icon (*) at the top of the chart allows configuring the same properties that appear on the binding property sheet. Changing the properties with the Settings icon affects the live chart rendering but does not change the actual binding property values, which are saved in the px file. This allows an end user to dynamically change the data source, time range, rollup or aggregation functions, interval, etc. to analyze data as needed in the chart.

TIP: You may embed Web Charts into a Dashboard Pane to allow end users to persistently save properties modified using the Settings icon. The framework saves the dashboard settings persistently for each user and those changes do not affect other users when viewing the same px.

Observing patterns using the Spectrum chart

This Ux chart provides a powerful tool that converts raw data into visual patterns, which you can use to identify subtle problems before they develop into major system issues. This chart works with historical data. The procedure for configuring this chart is similar to those for configuring the other charts.

Prerequisites: The system has collected enough historical data to produce a meaningful chart. Workbench or your browser is open and connected to a station, most likely a Supervisor station.

Step 1 Open the **`analytics`** palette and expand the **`Charts`** folder.

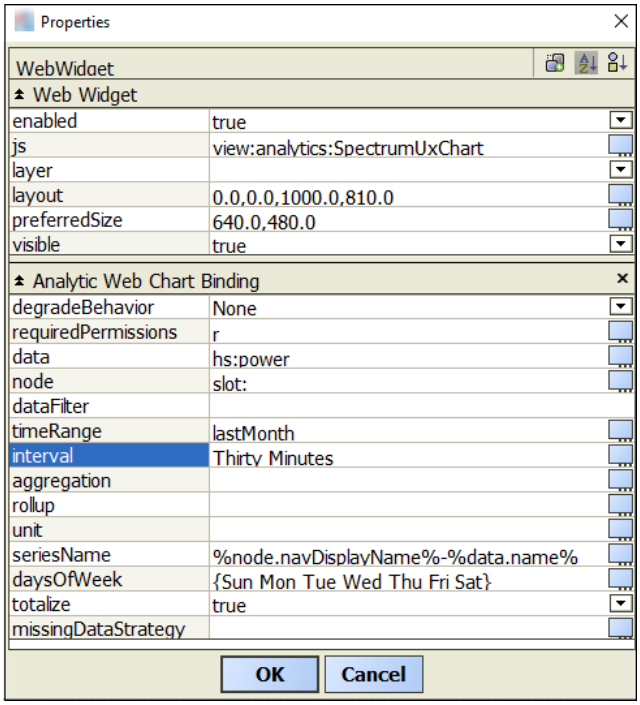
Step 2 Open a new Px view.

Step 3 Drag the **`Spectrum Chart`** from the palette to the Px view.

Step 4 Double-click the chart or right-click the chart and click `Edit Properties`.

You can change the values plotted by manipulating the chart's wire sheet properties, or by right-clicking the chart and using the properties window.

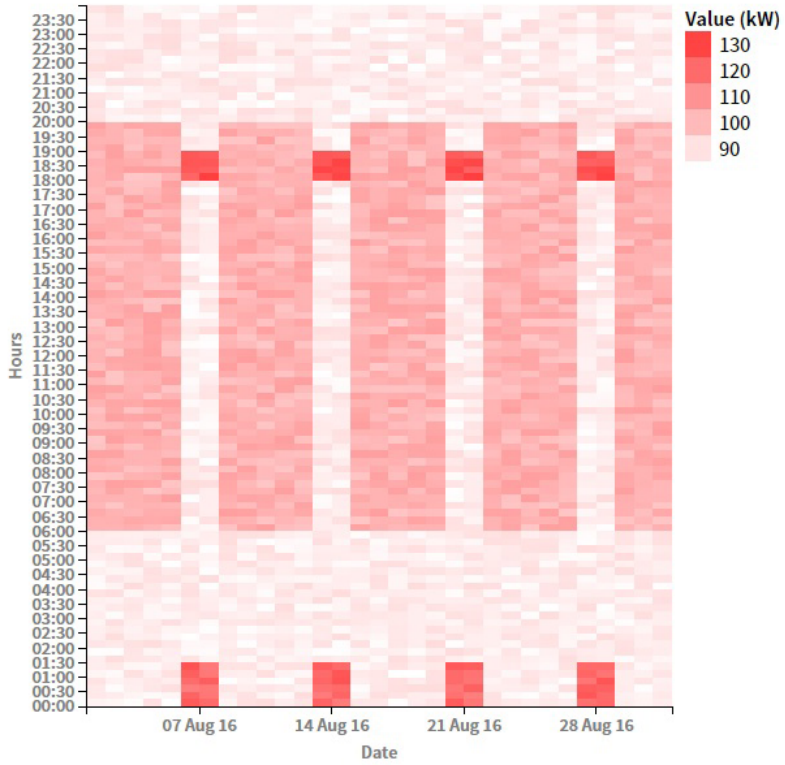
The **Properties** window for the chart opens.



Step 5 Configure properties and click **OK**.

NOTE: For best results, configure this chart with at least a height of 480 Abs or greater.

The framework renders the chart.



This chart shows electrical demand values for an entire month where colored swatches indicate the actual values for a specific time and day during the month. In this case, there are 31 colored

swatches horizontally at each timestamp representing each of the 31 days. The vertical size of the colored swatches vary based on the **`Interval`** property, which in this case is 30 minutes so there are 48 vertical colored swatches for each day. Notice that between midnight and 1 am as well as from 18:00 to 19:00 (6–7 p.m.), electrical demand was higher than expected for each Saturday and Sunday. This could be an anomaly or it could indicate a condition that requires further investigation. Such a spectrum chart allows an energy analyst to quickly spot a potential problem.

Step 6 To plot a pattern for another point, drag the point from the Nav tree to the chart.

Changing the aggregation function

The default aggregation function displays the first value that the framework finds.

Prerequisites: A Px view exists that uses aggregation to combine real-time values.

Step 1 Open an existing Px View.

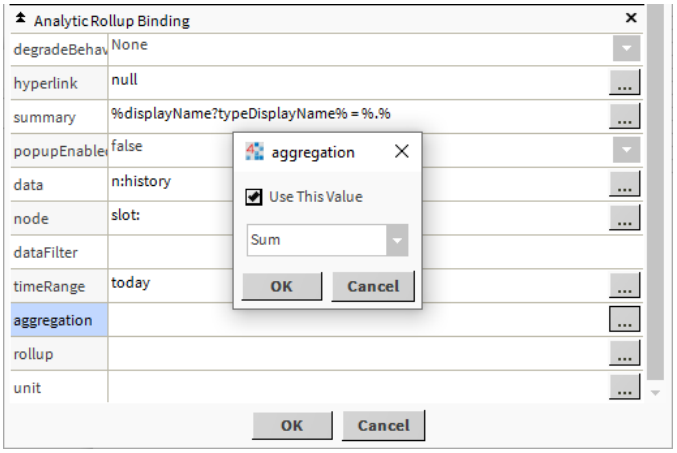
Step 2 Double-click the chart widget.

The **Properties** window opens.

Step 3 If necessary, scroll down in the window to the **Analytic** binding properties.

Step 4 Click the chooser button (...) to the right of the **`aggregation`** row.

The aggregation window opens.



Step 5 Enable **`Use This Value`**, choose a value other than `First` from the drop-down list, click **OK**, and click **OK** again to close the **Properties** window.

Setting up an analytic table binding

Tables by their nature are historical. Setting up an analytic table binding assumes that the points you are using have histories with them. Using a bound table is a good way to troubleshoot problems with bound labels because in a table you see the historical data that the framework is processing.

Prerequisites: The points you are using have history extensions.

Step 1 Open the **`bajaui`** palette, expand the **`Widgets`** folder and drag a **`BoundTable`** widget to the **Wire Sheet**.

Step 2 Double-click the widget, click the Px binding button ( ) select an `analytics Table Binding`, and click **OK**.

Step 3 Scroll down to the **Analytic Table Binding** section of the window and enter the tag for **`data`** that identifies the data source.

Step 4 Change any other properties and click **OK**.

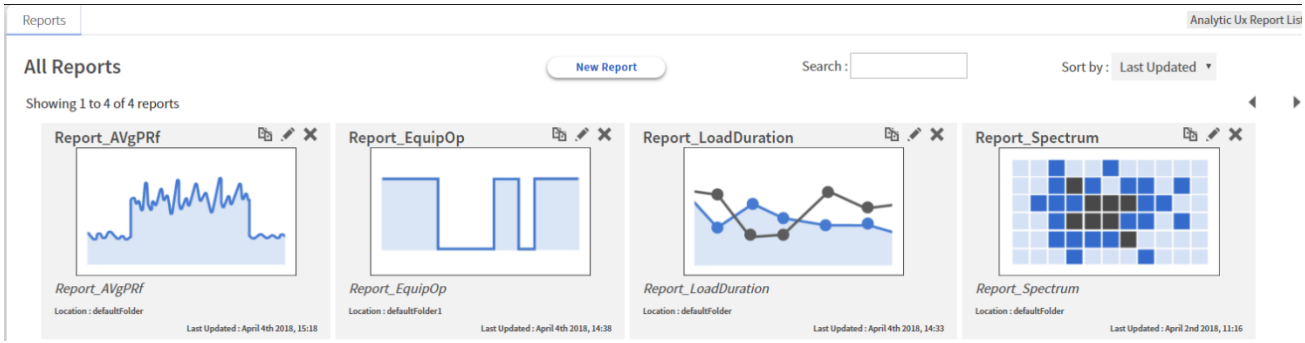
You see the result of the change in the table. Rollup does not actually impact the results until the interval changes.

Reports

The Report view provides an HTML page where end users can create ad-hoc reports and access saved reports. The available reports have names that correspond to the specialized Analytic charts including Aggregation, Average Profile, Equipment Operation, Load Duration, Ranking and Spectrum. Each report leverages the applicable specialized chart and also displays the historical data in a table below the chart. The reports also provide additional functionality allowing end users to configure the nodes, data, time range and many other properties.

To view each report click its thumbnail.

Figure 71 All Reports thumbnails



The framework intelligently selects the interval to use, if not explicitly set, for each report based on the optimal number of records for each report. You can change the interval by clicking the **Advanced** button in the report editor.

Creating Ux reports

Ux (User Experience) reports use HTML technologies. They are designed to be managed using a standard browser client, to be human-friendly and easy to use. Where Px reports require a Workbench client to configure, Ux reports share a centralized HTML home page with large visual tiles and identifiable icons. From the single centralized view you can create, edit, delete and clone reports as well as search and sort report lists. Only reports created using this view are visible in this view. Px reports are not visible here.

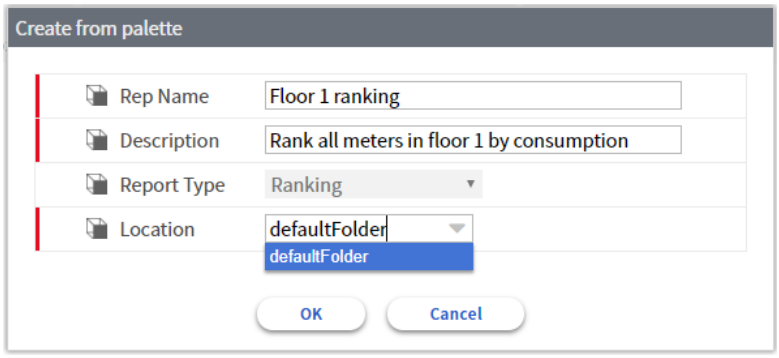
Prerequisites: You are working in Workbench or a browser client connected to the station.

Step 1 Right-click **AnalyticService→Reports** and click **Views→Analytic Ux Report List View**.

The **All Reports** view opens.

Step 2 Click the **New Report** button.

The **Create from palette** window opens.



Step 3 Give the report a name and description, select the type of report from the drop-down list, and select where to store the report file and click **OK**.

Managing Ux reports

Once a Ux report exists, you may edit its properties, clone it and delete it.

Prerequisites: The report exists.

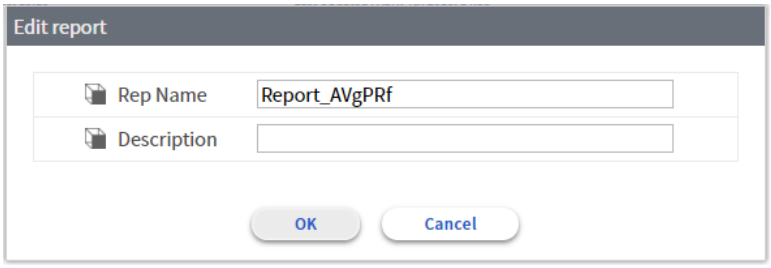
Step 1 Expand **AnalyticService** and double-click **Reports**.

The **Analytic Ux Report List View** opens.

Step 2 Scroll to the report or search for the report by name.

Step 3 To edit the report, click the edit icon ( ).

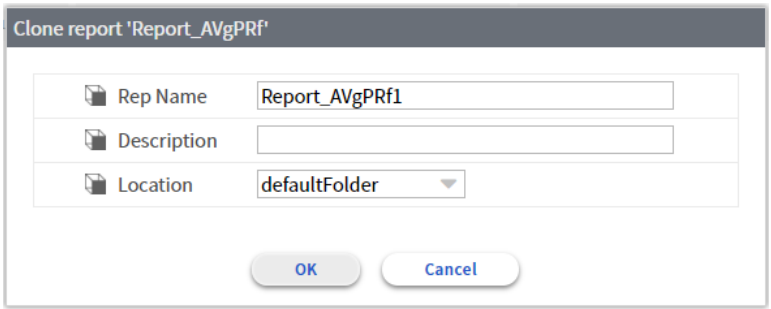
The Edit report window opens.



You can change the name and description.

Step 4 To make a new report by using an existing report (clone), find the report and click the clone icon ( ).

The Clone report window opens.



Step 5 To continue cloning the report, enter a name, description, and location for the new report.

Step 6 To delete a report, locate it, click the delete icon ( ) and respond to the message with **OK**.

Step 7 To view the report, click on it.

The report may take a few seconds to open.

Creating a dashboard

Each time a saved analytic report is accessed from the **Report** view, the report loads blank. The end user must configure the desired node, time range, data and other options. You can embed the **Analytic Report** widgets under a **Dashboard Pane** in a Px view, which causes the framework to save the report configuration. When an end user opens an **Analytic Report** in a Px dashboard, the Report loads with the last saved configuration.

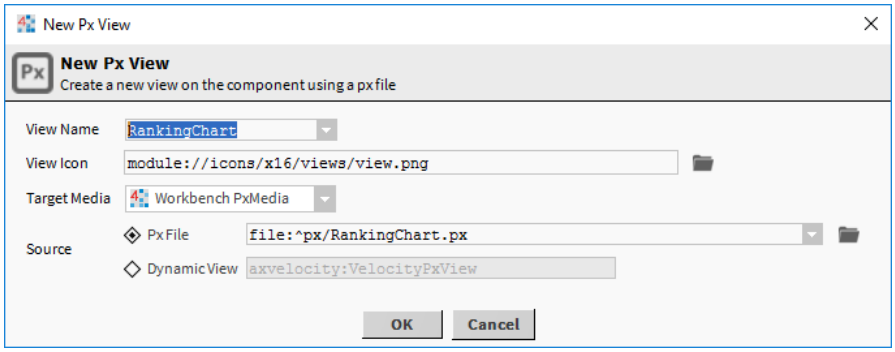
Prerequisites: The **DashboardService** exists in the station's **Services** container and the **`dashboard`** palette is open.

Step 1 Make a folder in your station to contain dashboards.

Step 2 Do one of the following:

- To create a new view, right-click the folder, click **Views→New View**, and assign a view name.
- To edit an existing view, double-click the view name.

The **New Px View** window opens.



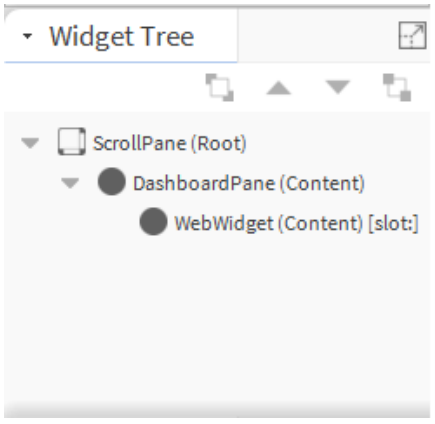
Step 3 Fill in the **View Name** for the dashboard and click **OK**.

Step 4 Expand the Canvas Pane **viewSize** to 1024 x 768 (this property is in the **`Properties`** side pane).

Step 5 Drag the **`Dashboard pane`** from the palette to the **`Widget Tree`** (side panel in the **PxEditor** view).

Step 6 Open the analytics palette, expand the reports node, and drag a report to the **Dashboard Pane** child content.

The system adds the report to the Widget Tree panel. For example:



Step 7 Save the Px view.

Configuring a report or a dashboard

Configuration involves selecting nodes, picking colors selecting the date range and other options.

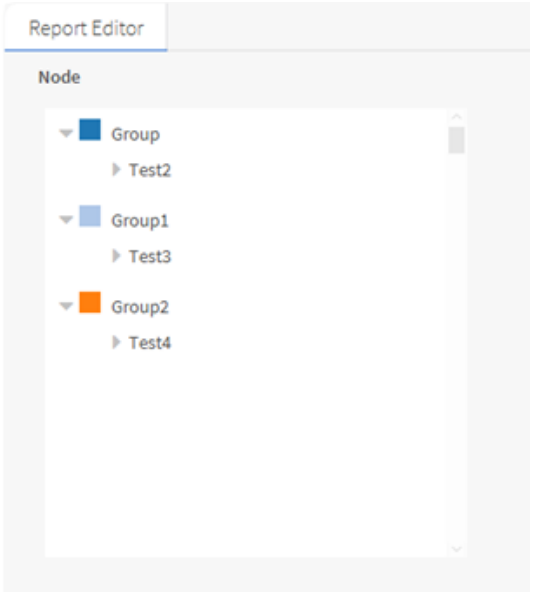
Prerequisites: You are connected to the station and the report or dashboard to configure exists.

Step 1 Expand the Nav tree to locate the points to include in the report, then expand the report or dashboard folder in the station, and double-click the report or dashboard name.

Step 2 Drag a component (folder, device, point folder, etc.) or an individual point to analyze from the Nav tree to the **Node** pane.

To group nodes in the **Node** pane, drop the node you are dragging onto the node parent group. To configure a parallel group for analysis, drop the node you are dragging outside the structure in the **Node** pane.

NOTE: The Aggregation and Spectrum reports support only one group.



The screen capture is an example of the **Node** pane with three groups.

Step 3 To change the color associated with the group, click the color swatch to the left of the group name and select the color in the **Color Picker**.

Step 4 To rename a group, right-click it, click `Rename`, enter a new name and click **OK**.

Step 5 Click the Tag Chooser and define the data type.

Step 6 Click the calendar chooser and select the day or date range.

The **Time Range** window selects a general time period or allows you to define a specific time and date to start and end data collection.

Step 7 Click to toggle the desired days of week.

When the background of the day of week button is black, the day will be included in the displayed data.

Step 8 For reports that support displaying a baseline trend, select the check box under the **Baseline** heading.

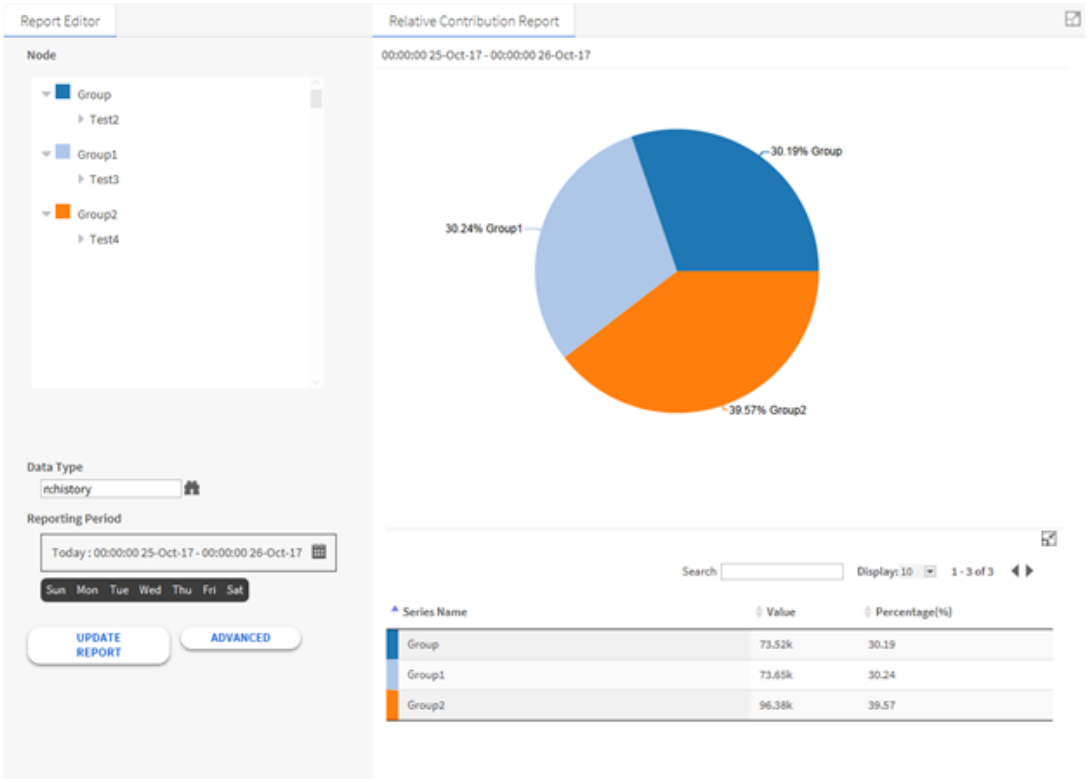
When this check box is deselected no baseline is displayed and the text beside the check box displays `No baseline`. When this check box is selected a baseline may be displayed and the text beside the check box displays the configured baseline time range.

Step 9 To edit the baseline details, click the binocular icon ( ).

Step 10 To enable either or both routines for reports that support normalization, select the check box for **`Floor Area`** or **`Degree Day`**.

Step 11 Click **Run Report**.

The report updates to display data in the chart and table.



The screen capture is an example of a **Relative Contribution Report**.

Step 12 Do one of the following:

- To expand the view to see the chart only, click the expand icon ( ) in the upper right corner of the chart area.
- To expand the view to see the table only, click the expand icon ( ) in the upper right corner of the table area.

Step 13 To return to the overall view, click the contract icon ( ).

Step 14 To apply the changes you make, click the **Update Report** button.

Step 15 To configure additional properties, click the **Advanced** button.

The additional properties vary depending on the report. See the *Reference* for a description of each property.

Typically, when a dashboard-able widget is placed under a dashboard pane, the widget adds a save button to allow end users to decide whether to persist any configuration changes they have made. When any of the analytic reports is placed under a dashboard pane the framework adds a reset button in the **Report Editor** section to the right of the advanced button. Instead of adding a save button to the **Report Editor** section, the report saves any configuration changes (nodes, time range, days of week, etc.) automatically. Clicking the reset button prompts the end user with a window indicating, "This will remove your dashboard data from the widget. Are you sure you want to

proceed?" If they click the **Yes** button, the framework resets the report back to its default, unconfigured state.

Normalizing energy consumption values based on floor area

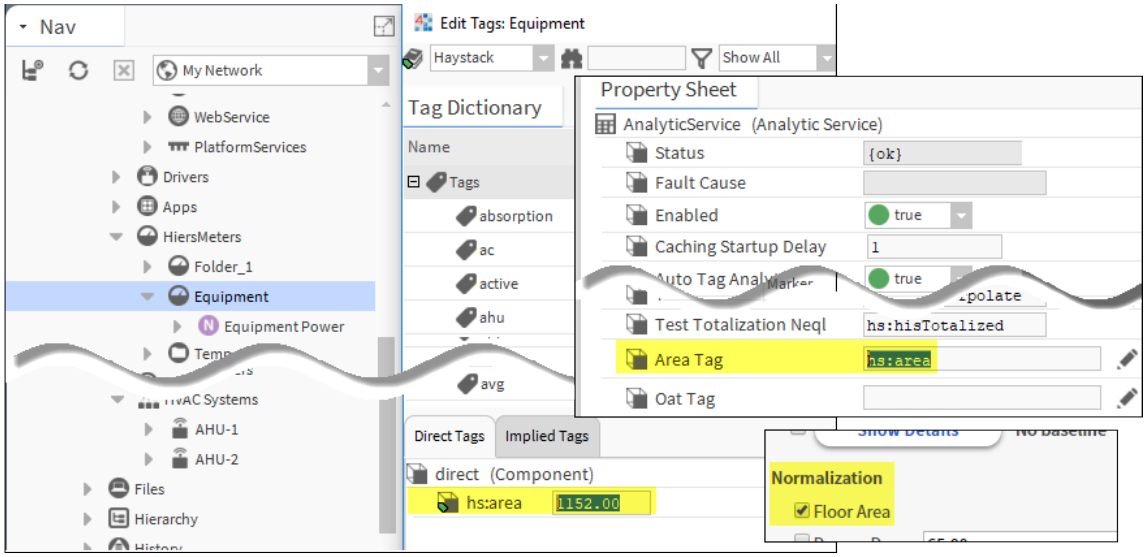
An energy procurement strategy can reduce consumption volatility. It also sets up a load profile that is more attractive to energy providers and can reduce your energy costs. To create an energy procurement strategy you need to understand current energy consumption patterns exhibited by your equipment. Normalizing the floor area of a facility evens out the energy consumption or demand differences caused by large and small spaces. This results in more useful comparisons. This topic documents how to configure the framework's Average Profile report to plot and report these patterns for a single piece of equipment.

Prerequisites: You are working in Workbench with any tag dictionary in the TagDictionary service. You have historical energy consumption data collected for the piece of equipment in the station database.

In this procedure's example the facility uses Hiers meters to monitor equipment power usage.

Step 1 Right-click the equipment and select **Edit Tags** from the popup menu.

The **Edit Tags** view opens.



Step 2 To configure **AnalyticService** properties, right-click the service in the Nav tree, click `Property Sheet`, and edit the **`Area Tag`** property with the area tag name.

If you do not configure the **`Area Tag`** property in the **AnalyticService**, the framework defaults to `hs:area`.

Step 3 Select the applicable tag dictionary (Haystack for default configuration), select the applicable tag (`hs:area` for default configuration) and click **Add Tag**.

The **Direct Tags** tab in the lower half of the window displays the **`hs:Area`** tag.

Step 4 Enter your facility's area (square feet or square meters) and click **Save**.

The area in the example is configured for 1152 square units.

Step 5 Expand the AnalyticService in the Nav tree and double-click the **Reports** node.

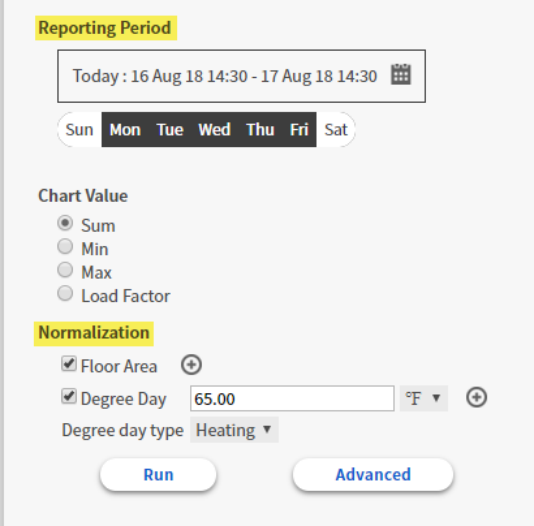
The set of available **Analytic Ux Report List View** report tiles opens.

Step 6 Do one of the following:

- To create a new report, click **New Report**; supply a **`Rep Name`** and **`Description`**; select `Average Profile` or `Load Duration Report` for **`Report Type`**, click **OK**; and click the report tile.

- To edit an existing report, click its tile.

The **Report Editor** opens.



Step 7 Drag the desired node, in this case the **Equipment** folder under the **HiersMeters**, from the Nav tree to the **Node** section in the **Report Editor** pane.

Step 8 Configure the **`Reporting Period`**; click to enable **`Floor Area`** under **`Normalization`**, and click **Run**.

The framework produces the report.

If neither the direct tag on the equipment node nor one of its children has an area value configured, or if the value is zero (0), the framework performs no normalization even if **`Floor Area`** normalization is enabled in the **Report Editor**.

Normalizing energy consumption values based on degree-day temperature

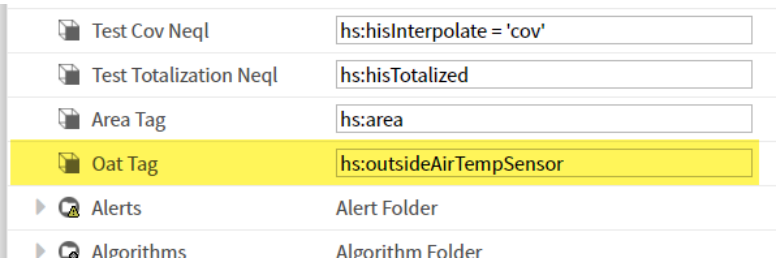
Normalizing energy usage based on degree days allows more accurate comparisons for a building when analyzing data from different times of year, such as summer versus winter. Degree day normalization may also facilitate more accurate comparisons when buildings are located in different geographic regions that experience different weather patterns.

Prerequisites: You are working in the web UI with the AnalyticService. Historical OAT (Outside Air Temperature) data are available in the station database and tagged with an identifiable OAT Tag.

Step 1 To view the Nav tree, click the Navigation Tree button ( ).

Step 2 Right-click the **Config→Services→AnalyticService** in the Nav tree and click **Views→Property Sheet**.

The **AnalyticService Property Sheet** opens.



Step 3 In the **`Oat Tag`** property, enter the tag that identifies the OAT values in the database.

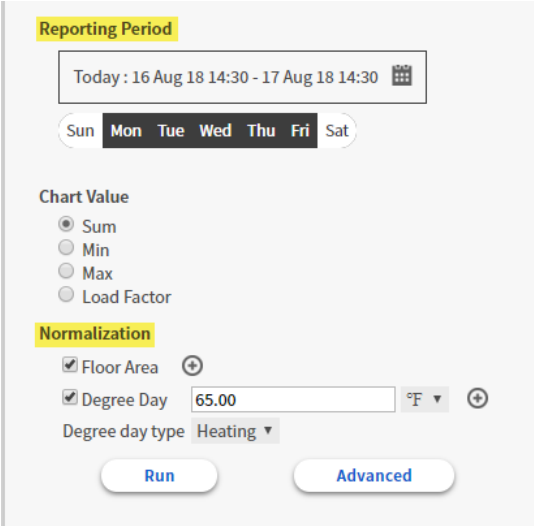
Step 4 Expand the AnalyticService in the Nav tree and double-click the **Reports** node.

The set of available **Analytic Ux Report List View** report tiles opens.

Step 5 Do one of the following:

- To create a new report, click **New Report**; supply a **Rep Name** and **Description**; select `Average Profile` for **Report Type**, click **OK**; and click the report tile.
- To edit an existing report, click its tile.

The **Report Editor** opens.



Step 6 Drag a component from the Nav tree to the **Node** window in the **Report Editor**.

The report loads blank (no node configured) so the report does not actually render data without configuring a node.

Step 7 Configure the **Reporting Period**, including the days of the week; under **Normalization**, click to enable **Degree Day**; and enter the outdoor temperature at which neither heat nor air conditioning is required to maintain a satisfactory indoor air temperature.

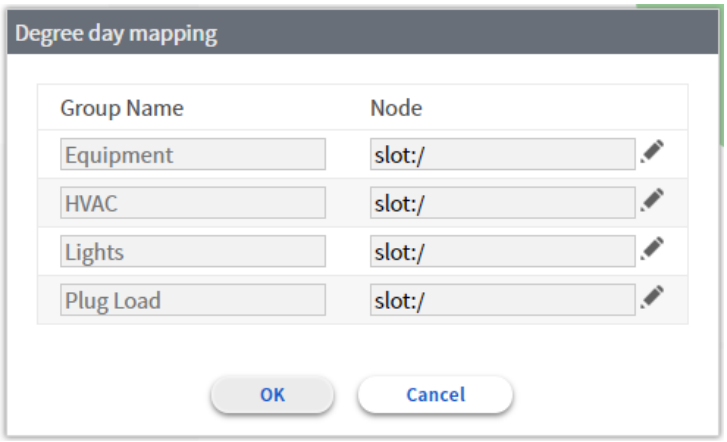
This outside temperature value defaults to 65.00, which is a Fahrenheit temperature.

Step 8 If you are using this default temperature, select `°F` from the temperature scale drop-down list, otherwise, enter a Celsius or Kelvin temperature value and select `°C` (Celsius) or `K` (Kelvin) from the drop-down list.

Step 9 To select the type of temperature supplementation, select `Heating` or `Cooling` for **Degree day type**.

Step 10 To identify the point node in the Nav tree to provide the outside air temperature values, click the plus icon to the right of the **Degree Day** properties.

The **Degree day mapping** window lists the Node groups configured in the report and allows mapping each group to a specific control point with the `hs:outsideAirTempSensor` tag.



Step 11 If necessary, edit the group and node, then click **OK** followed by clicking **Run**.

For the **`Reporting Period`** you defined, the framework calculates the degree-day value by taking the average of the differences between the base outside air temperature (default: 65.00) and each actual outside air temperature value. Then it calculates the energy consumption per degree-day by dividing the energy consumption for each interval by the average degree-day value.

Printing a report

There are two ways to create a hard copy of a report: export the report to a PDF and print the PDF from your PC, or print the report from the web UI using either Chrome or Firefox.

Prerequisites: The report exists. You are working in the web UI.

Step 1 Do one of the following:

- To prepare to print a report in Chrome, click **Ctrl** + **P**.
- To prepare to print a report in Firefox, click **Menu→Print**.

For Chrome, the print preview page opens with the **Print** properties in the left pane.

For Firefox, the print options appear along the top of the page.

Step 2 Click the tiny arrow above the report editor to minimize it so that only the report itself prints.

Step 3 After configuring properties, click **Print**.

Chapter 5 Outlier handling

Topics covered in this chapter

- ♦ Filtering data with the status filter
- ♦ Filtering data with the raw data filter
- ♦ Raw data filter example

Historical data collected from meters, sensors and other building automation devices can skew calculations if records are missing or contain invalid values. Data may contain unwanted or inaccurate values caused by a sudden electrical current surge, meter reset or device failure. Operations performed on incomplete data sets and on records that contain junk, outlier, or noise values inevitably produce inaccurate results. Beginning with a recent version of Niagara, you can select which history records to exclude from the data set based on record Status..

For example, the following table contains invalid data.

Timestamp	Value	Status	Description
15/12/21 2:00	99999099	{ok}	This is an extremely high value, which is invalid.
15/12/21 3:00	10	{ok}	This is a valid value.
15/12/21 4:00	20	{ok}	This is a valid value.
15/12/21 5:00	NaN	{ok}	This value is invalid because it is not a number. Not a Number, expressed as NaN, is an actual numeric value similar to positive infinity expressed as +inf or negative infinity expressed as -inf.
15/12/21 6:00	20	{fault}	This value is valid, but the status of the device indicates a problem.
15/12/21 7:00	20000	{ok}	This invalid high value was caused by a sudden meter reset.
15/12/21 8:00	20010	{ok}	This is another invalid high value caused by a sudden meter reset.

Based on device status, you can filter out the records that contain invalid data. This creates a data set with missing records. You then use a missing data strategy to interpolate the missing data.

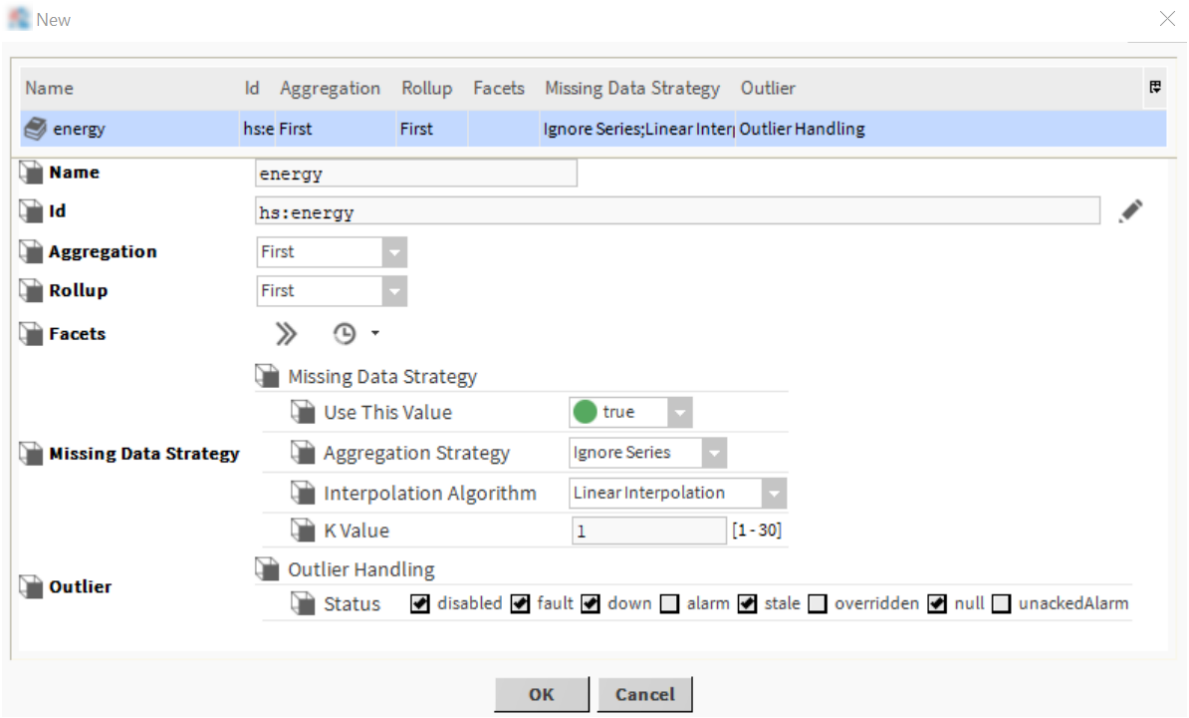
Filtering data with the status filter

This filter identifies records with invalid data. After removing the records, you can configure the framework to interpolate the missing data using a missing data strategy. Follow these steps before applying a missing data strategy.

Prerequisites: You are using Workbench in a Supervisor or controller station. The database contains history records collected from tagged devices for which the data definition exists.

Step 1 Navigate to **Config→Services→AnalyticService** and double-click the data definition associated with the tag and device.

The edit view of the data definition opens.



`Outlier` `Status` options default to `disabled`, `fault`, `down`, `stale`, and `null`. This means that if you make no changes, the framework removes from the data set history those records whose **`Status`** value is one of these.

Step 2 To filter (remove) records that have specific statuses, select one or more check boxes under **`Outlier`**.

Step 3 To retain records that the framework would otherwise remove, click to remove one or more check marks.

Step 4 To continue, click **Save**.

If you check all boxes, the framework filters out all records except those with a status of `{ok}`, which is always enabled. If you check no box, the framework filters out no records based on **`Status`**.

The framework removes all records with the selected **`Status`** values from the data set.

For example, here is a set of historical data:

Timestamp	Data value	Status
12/15/21 11:00	25	{fault}
12/15/21 12:00	25	{null}
12/15/21 13:00	45	{ok}
12/15/21 14:00	56	{overridden, alarm}

If you select the fault and null check boxes for Outlier, only these records pass through:

Timestamp	Data value	Status
12/15/21 13:00	45	{ok}
12/15/21 14:00	56	{overridden, alarm}

You would now interpolate the missing data using linear interpolation or K nearest neighbor. Following interpolation, the data would look like this:

Timestamp	Data value	Status	Trend Flags
12/15/21 11:00	45	{ok}	{li}
12/15/21 12:00	45	{ok}	{li}
12/15/21 13:00	45	{ok}	{}
12/15/21 14:00	56	{overridden, alarm}	{}

If there is no preceding record, the linear interpolation uses the first available record. If there is a valid preceding and post record, linear interpolation calculates a value between to two values. If the 10:00 value was 30, the interpolated values would be 11:00 35 and 12:00 40.

An algorithm can process these data directly to create a graph or other visual representation of the data.

Filtering data with the raw data filter

The raw data filter removes numeric-value records based on configured low and high limits. Then, a missing-data strategy, such as linear interpolation, creates new values for the filtered records. Values that are too low or too high can skew analytic results making them unusable. Use of this filter removes the outliers so that future data analysis returns more realistic results.

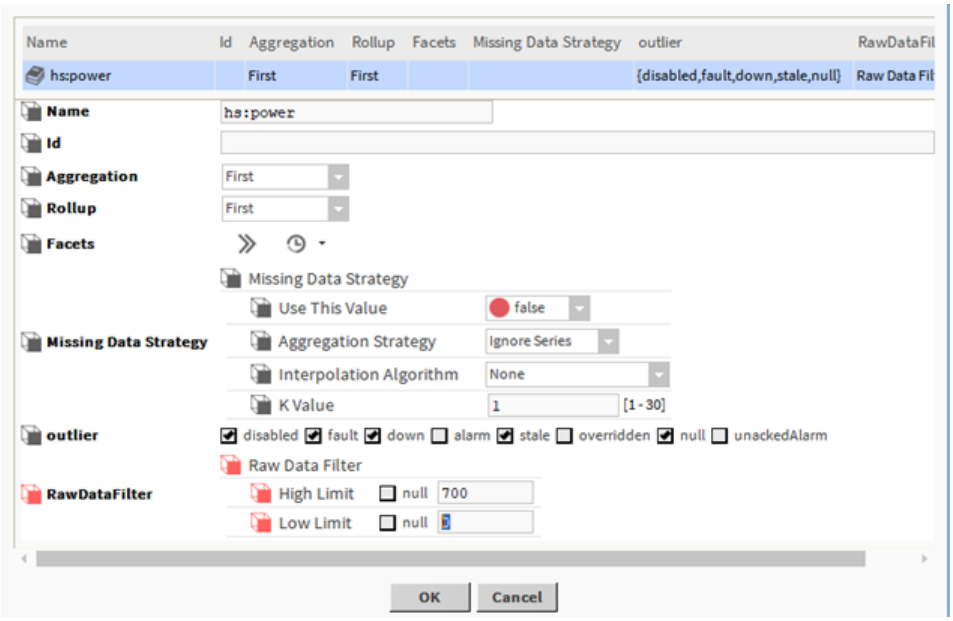
Prerequisites: Latest version of Niagara

Step 1 Expand **Config→Services→AnalyticService** and double-click **Definitions**.

The **Analytic Data Manager** view opens.

Step 2 To add a definition, click the **New** button.

The **New** window opens.



The Raw Data Filter is located below the outlier property.

Step 3 Enter a name for the definition and configure all relevant properties.

Step 4 To edit the high and low limits, click to remove the null check marks, and enter appropriate values.

A check mark in a `null` check box disables the related limit. Removing the null check mark enables the limit.

If you only set the high limit, the filter only discards records with values above the limit (exclusive).

If you only set the low limit, the filter only discards records with values below the limit (exclusive).

If you set both limits, the filter discards records with values above and below the defined limits (exclusive).

You can also configure the Raw Data Filter limits using the Data Definition Property Sheet (double-click **Config→Services→AnalyticService→Definitions**, right-click the definition, and click **Views→Property Sheet**).

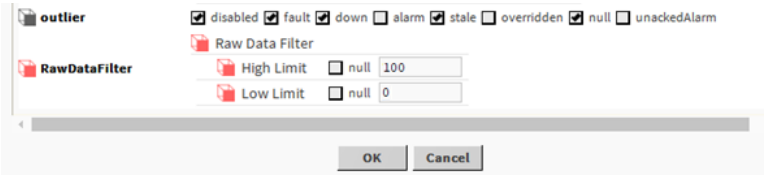
Raw data filter example

This filter prepares data for meaningful analysis. This example illustrates how the filter works and suggests what a meaningful dataset would look like.

Here is a raw dataset representing power readings taken hourly beginning at 11 am for 10 hours:

Timestamp	Value	Status
15/12/21 11:00	20	{}
15/12/21 12:00	25	{}
15/12/21 13:00	15	{}
15/12/21 14:00	-20	{}
15/12/21 15:00	85	{}
15/12/21 16:00	130	{}
15/12/21 17:00	60	{}
15/12/21 18:00	150	{}
15/12/21 19:00	-15	{}
15/12/21 20:00	90	{}

Values higher than 100 and lower than zero are outliers that will inaccurately skew the resulting analysis. A data definition is configured as follows:



The raw data filter outputs these values:

Timestamp	Value	Status
15/12/21 11:00	20	{}
15/12/21 12:00	25	{}
15/12/21 13:00	15	{}
15/12/21 14:00	removed by the filter	
15/12/21 15:00	85	{}
15/12/21 16:00	removed by the filter	

Timestamp	Value	Status
15/12/21 17:00	60	{}
15/12/21 18:00	removed by the filter	
15/12/21 19:00	removed by the filter	
15/12/21 20:00	90	{}

Using linear interpolation the resulting dataset looks like this:

Timestamp	Value	Status	Trend Flags
15/12/21 11:00	20	{}	{}
15/12/21 12:00	25	{}	{}
15/12/21 13:00	15	{}	{}
15/12/21 14:00	50	{}	{li}
15/12/21 15:00	85	{}	{}
15/12/21 16:00	72.5	{}	{li}
15/12/21 17:00	60	{}	{}
15/12/21 18:00	70	{}	{li}
15/12/21 19:00	80	{}	{li}
15/12/21 20:00	90	{}	{}

Chapter 6 Missing data management

Topics covered in this chapter

♦ Linear interpolation
♦ K-nearest neighbor (KNN)
♦ Aggregation strategies
♦ Missing data configuration
♦ Creating a missing data strategy for a data set
♦ Missing data indication

In statistics, imputation is the process of replacing missing data with substituted values. Incomplete, incorrect, inaccurate and irrelevant data are replaced, modified, or deleted. This is also known as data cleansing or data cleaning.

Where the data are missing in the series plays an important role in the calculations. Data may be missing at the beginning of the data set, interspersed among the other data, or at the end of the data set.

The framework offers multiple strategies for managing missing data:

- Linear interpolation derives an estimated value of the missing data in the series.
- K-nearest neighbor finds the nearest neighbors to a missing datum, identifies the majority value represented by the neighbors, and fills in that value for the missing datum.
- Aggregation strategies: ignore series and ignore point

Analytics applies the missing data strategy based on the method for filling in the missing data that you use. Analytics does not go back and update the history records themselves with the missing data.

Linear interpolation

This interpolation algorithm linearly interpolates the missing values based on the surrounding values in the series.

There are three locations where data can be missing:

- At the beginning of the series
- Interspersed among the series
- At the end of the series

Data missing at the beginning of the series

The system replaces missing values with the first available value in the series. For example, if M1 is a faulty meter installed in Building 1, which fails to record a daily energy reading for three days, and its replacement meter records 20 on day four, linear interpolation assigns 20 to each of the missing days.

Data missing interspersed among the series

The system replaces missing values by calculating the slope between the last and next collected values. The interpolation equation is:

```
slope = (nextValue - previousValue) / (nextTimestamp - currentTimestamp)
```

For example, meter 1 functioned accurately and logged values for three days after which a fault occurred in the meter. It took three days to identify and fix the fault. On day six the system began to log values again. The missing values in this data set occur between intervals. If the recorded value for each day at the beginning of the series is 20, and the recorded value for the sixth day is 30, the calculation for day four is:

```
20 + ((30-20)/(6-3)) = 23.33
```

And the calculation for day 5 is:

```
20 + ((30-23.33)/(6-5)) = 26.66
```

Data missing at the end of the series

The system replaces missing values with the last recorded value in the series. For example, meter 1 takes a reading for each of three days after which it goes into fault for two days, ending the series. The reading for day three is 20. It makes no difference what the readings are for days one and two. The system interpolates the value for days four and five as 20.

K-nearest neighbor (KNN)

KNN is for numeric, enum and Boolean records. For intervals, other than `none`, this strategy replaces a missing value by calculating the majority value recorded for the item's k nearest neighbors.

The number of neighbors to consider is `k`. The system selects `k` previous and next nearest neighbors to calculate the missing value. If a tie occurs between two values, the algorithm selects the lowest timestamp value.

Boolean data series example

For example, the system monitors if meter 1 is on or off reporting a value of true (on) or false (off) every 15 minutes (the interval). If a value for 2:15 is missing between time stamps 2 pm and 3 pm, and `k = 3`, the system finds the three nearest timestamps to 2:15, which are 1:45, 2:00, and 2:30 and takes the majority of these three values. The table records these values:

Timestamp	Meter 1 on state
1:30	true
1:45	false
2:00	true
2:15	false (interpolated value)
2:30	false
2:45	false

In the example, the majority value, considering the three neighbors, is "false." The system assigns this value to 2:15.

Tie example

A tie can occur with numeric, Boolean and enum data. The system handles a tie in a particular way:

1. First, it gives preference to the highest frequency of k nearest neighbors.
2. Next, when the same frequency (`k`) of nearest neighbors exists, the system gives preference to the value from the record with the timestamp nearest to the missing record.
3. Finally, when the timestamps are equidistant from the missing data, the system gives preference to the record preceding the missing data.

In Table 1, `k=4`, its preceding nearest neighbors' majority value is zero. Its succeeding nearest neighbors' majority value is 1. Using rule 2 above, the system breaks the tie by assigning the missing value to the same value as the preceding timestamp (1:45).

Table 5 K = 4, Same frequency of nearest neighbors

Timestamp	Enum data
1:30	0
1:45	0

Timestamp	Enum data
2:00	0 (interpolated value)
2:15	1
2:30	1

In Table 2, `k=1`, its preceding nearest neighbor's value is 1. Its succeeding nearest neighbor's value is 3. The system recorded both values at the same interval before and after the missing value, so it gives preference to the preceding timestamp, and sets the missing data value to: 1.

Table 6 K = 1 Timestamps equidistant from the missing data

Timestamp	Enum data
1:30	0
1:45	1
2:00	1 (interpolated value)
2:15	3

Aggregation strategies

These strategies configure the system to ignore either the aggregated sum of a series with missing data or to ignore only the missing values while calculating the aggregated sum of the records with values.

Ignore series

This strategy ignores any aggregated sum that includes missing data, even if only a single record is missing.

For example, a meter is added to a site, and four days later it starts recording energy consumption data. On a report or chart configured to aggregate the sum of all energy meters, the system ignores the aggregated sum for days 1–3 because the calculation for at least one meter contains missing data.

Day	Meter 1 energy values	Meter 2 energy values	Aggregated values (sum)
1	-	10	-
2	-	10	-
3	-	10	-
4	30	20	50
5	30	30	60

The aggregated sum ignores the fact that meter 2 recorded values of 10 for the first three days.

Ignore point

This strategy ignores only the values in the interval that are missing and accommodates the recorded values for the overall calculation.

For example, using the same meter as in the example of ignoring the series, the system aggregates the sum of all values ignoring only the missing values themselves.

Day	Meter 1 energy values	Meter 2 energy values	Aggregated values (sum)
1	-	10	10
2	-	10	10
3	-	10	10

Day	Meter 1 energy values	Meter 2 energy values	Aggregated values (sum)
4	30	20	50
5	30	30	60

The system counts Meter 2's values for days 1–2.

Missing data configuration

Missing data strategies (algorithms) apply to specific types of points.

These types of points benefit from missing data strategies:

Interpolation algorithm	Numeric Point	Boolean Point	Enum Point
K-nearest neighbor	☑	☑	☑
Linear interpolation	☑		

NOTE: Linear interpolation works on raw history and also when an interval is selected. Any interval can be applied to get interpolated records. KNN works only when an interval other than none is selected for the series.

There are several places where you can set up a missing data strategy:

- Using the **Missing Data Strategy** property under the **AnalyticService** sets up a global strategy that applies to every trend request.
- In a data definition, which applies to any trend request for the specific data set identified by the **`Id`** tag.
- In an Analytic Binding, proxy extension and alert that uses a trend request.

If no missing data strategy is defined in the Analytic Binding, proxy extension or alert, the system defaults to the missing data strategy configured on the data definition for the tagged points.

If no missing data strategy is defined on the data definition for the tagged points, the system defaults to the global strategy defined on the **AnalyticsService**.

Creating a missing data strategy for a data set

Setting up a missing data strategy for a specific data set rather than as a global strategy provides a way to fine-tune your analytic results.

Prerequisites: All points are tagged in preparation to run an analytic query.

Step 1 Expand **Config→Services→AnalyticService** and double-click **Definitions**.

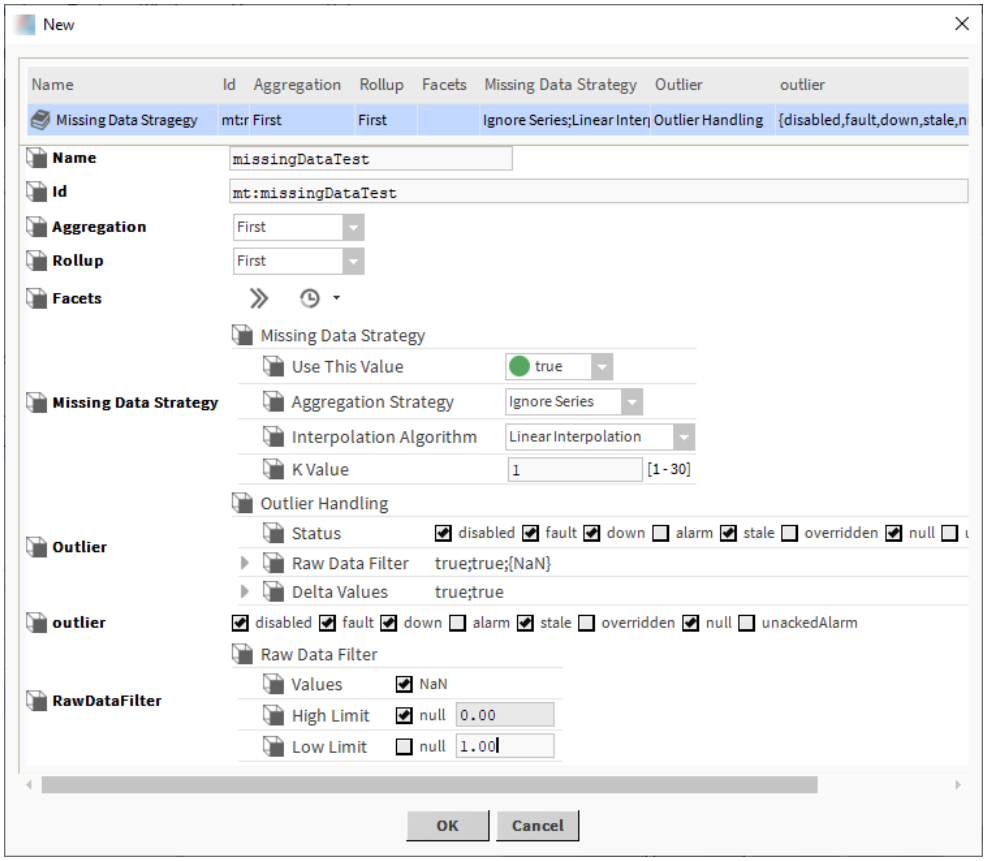
The **Analytic Data Manager** view opens.

Step 2 To create a missing-data definition, click **New**.

The **New** window opens.

Step 3 To accept the default **`Type to Add`** (`Analytic Data Definition`), click **OK**.

A second **New** window opens.



Step 4 Name the strategy, enter a tag in the **`Id`** property, enable **`Use This Value`** (set to `true`), select an **`Interpolation Algorithm`**, define the **`RawDataFilter`** properties used to evaluate each record and click **OK**.

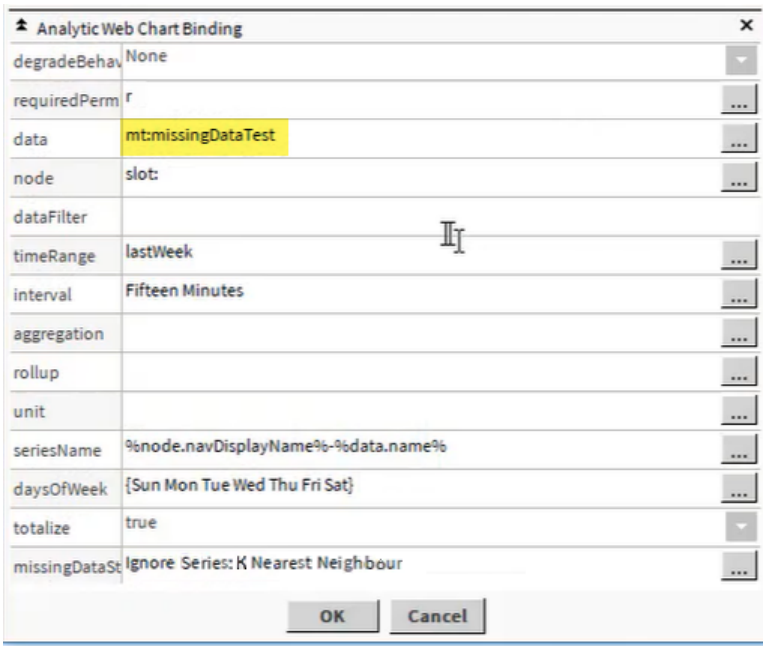
The Id tag identifies the data set to include all points tagged in this example with `mt:missingData`. Setting the **`Low Limit`** on the **`RawDataFilter`** to `1.00` configures the strategy to remove any value that is less than 1, which removes any zeros from the data set.

Step 5 Add an **AnalyticWebChart** to a PX view.

The reason to use this type of chart is because it allows you to define a tag that identifies the data to return on the chart.

Step 6 Double-click the chart to open the properties window.

The widget opens.

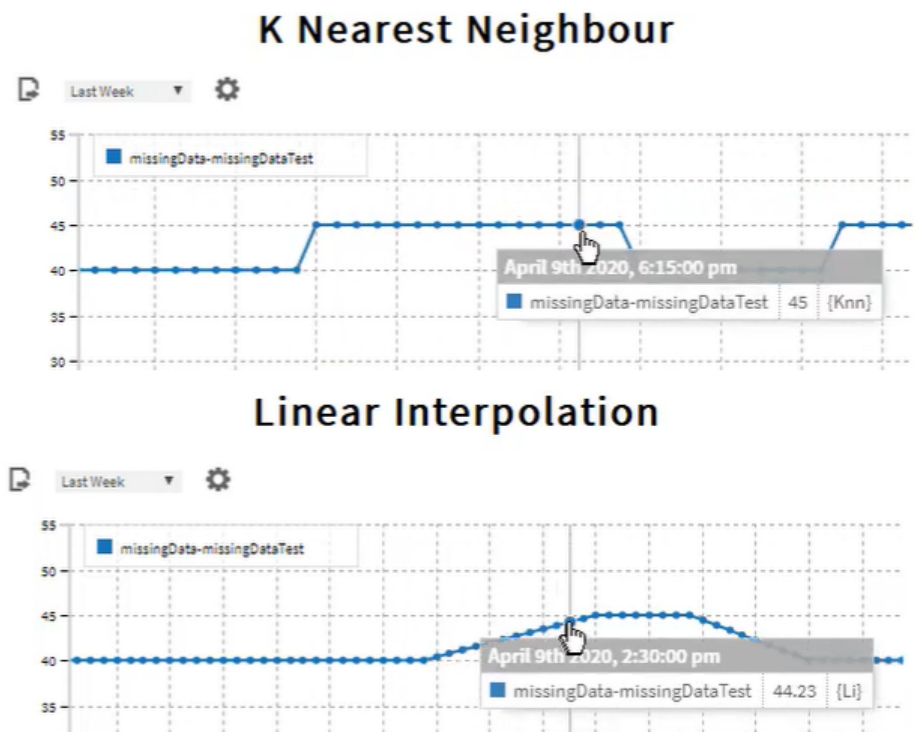


The **data** property should already be set to the tag you entered when you created the definition. This capture shows the same binding configuration but with KNN-1 enabled.

Step 7 Configure the **data** property to the tag you entered when you created the definition.

Step 8 Set any other properties you ignored when you created the definition, including **aggregation**, **rollup**, **missingDataStrategy**, or outlier handling. and click **OK**.

You should be able to identify the interpolated date points in the chart.



The tool tips identify the interpolated data.

Step 9 For another way to configure each chart, click the configuration icon ( ).

The **Settings** window opens. In this window changes to the **`Missing Data Strategy`** overrides directly in the chart. An end user could use this configuration feature.

Missing data indication

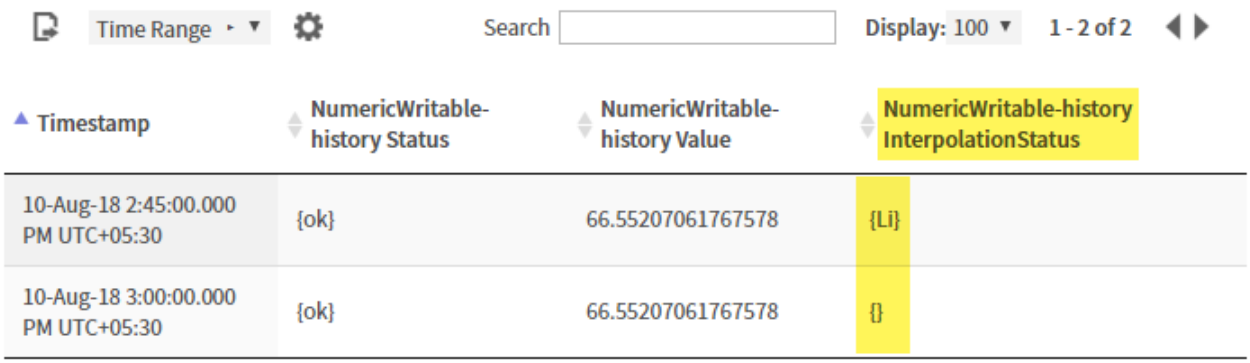
The system identifies data that were interpolated on both tabular views as well as report and chart views.

Tabular views

Across all controls, flags identify interpolated data:

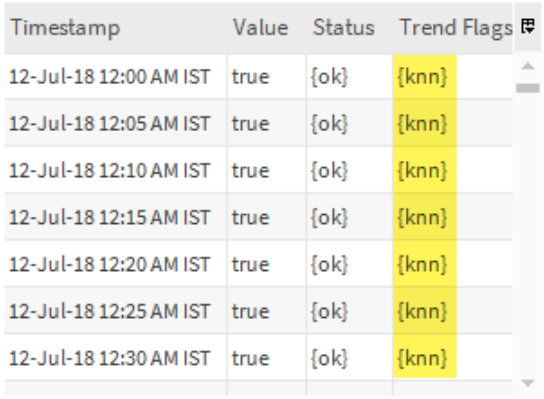
- `{Li}` indicates that linear interpolation was applied.
- `{knn}` indicates that k-nearest neighbor interpolation was applied.
- `{igp}` indicates that **`Ignore Point`** was selected for aggregation strategy.
- `{ }` indicates that no interpolation algorithm was applied.
- `{knn,igp}` indicates that k-nearest neighbor interpolation was applied with Ignore Point as aggregation strategy.

Figure 72 Example of a web table with interpolated data



The Interpolation Status column to the right indicates how the data were interpolated.

Figure 73 Example of a bound table with interpolated data



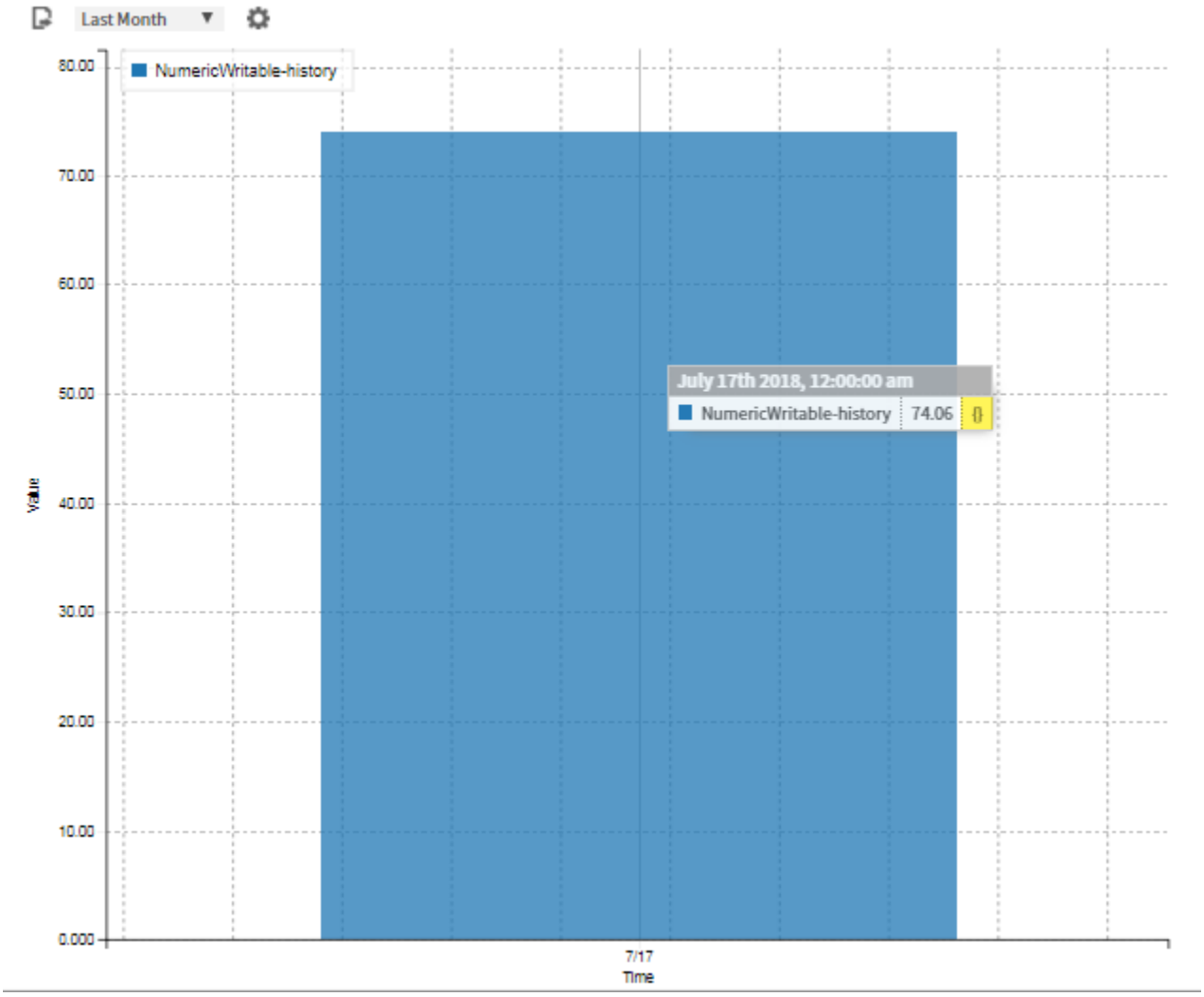
The Trend Flags column contains the interpolated data flags.

NOTE: **`Ignore Series`**never triggers a flag because this property causes the system to ignore the entire series. Interpolated data do not appear in the table, report or chart. This was the framework's default behavior when no data are available.

Charts

A tool tip on a chart indicates an interpolated or a real record.

Figure 74 Numeric writable chart

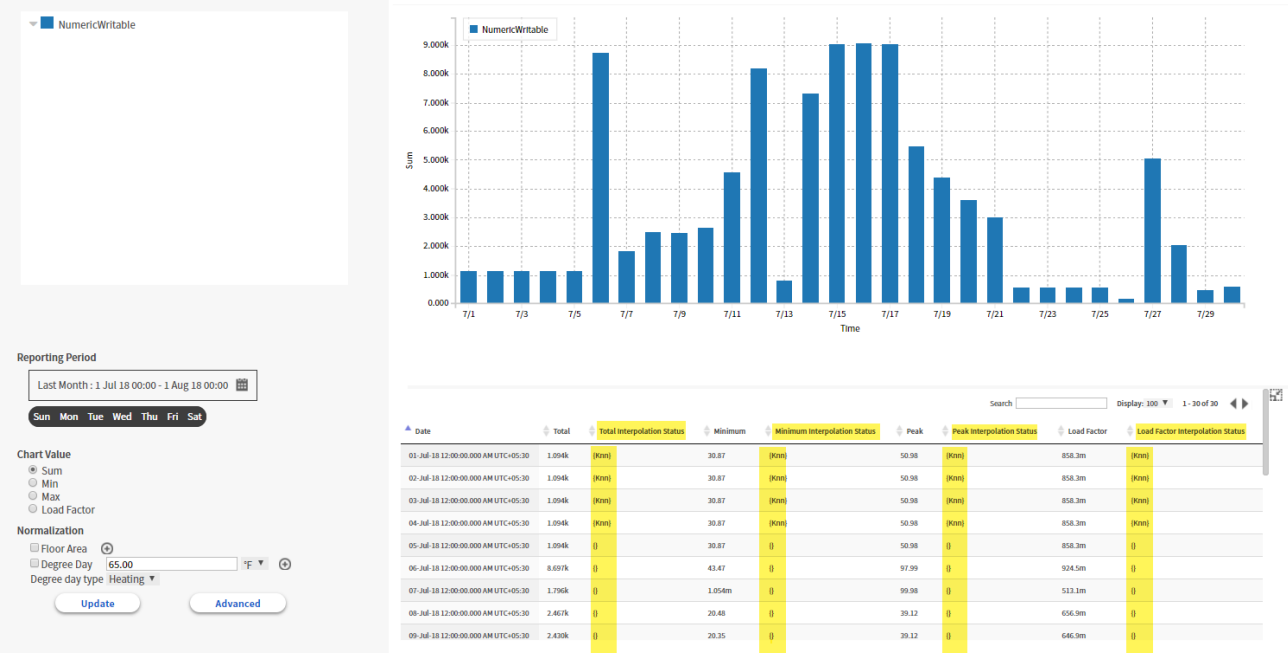


The tool tip indicates that no interpolation algorithm was applied.

Reports

Reports provide interpolated data in a separate column. To view these data, you must enable **`Show Interpolation Status`** in advanced settings.

Figure 75 A report configured to show interpolated data



The interpolated data are visible in the table.

Chapter 7 Troubleshooting

Topics covered in this chapter

- Point status
- Enabling error logging
- HTTP ERROR: 500 Privileged Action Exception
- Scenarios

Make sure all cables are correctly connected and all equipment turned on. This topic covers general issues that occur after the system is installed and configured.

Definitions

If you are having difficulty visualizing data that appears to be configured correctly, make sure that you have a definition for each tag. This is especially important if you created your own tag dictionary. When you create a definition, you associate it with a tag by entering the tag name space and name in the `Id` property. To set up definitions, expand **Services→AnalyticService→Definitions**.

Logs

System logs contain information you can use when debugging problems.

If a point matches the criteria required to generate an alert, appears in a Proxy Extension, or is included in a binding, but the point does not have an `a:a` tag, the system logs an error. The log level for this error is FINER. At the default info log level, the system does not print this error, but you can change the analytics log level using the Station spy option to view the FINER log.

Point status

Point status reports the current status flag(s). More about these flags and point status is in *Getting Started with Niagara*

The status of points, especially proxy points, may explain what appears or does not appear as expected in a bound table or on a web chart. The following summarizes the meaning of each status and what, if anything, to do about it.

Table 7 Point status

Status	Description	Remedy
alarm	The point has a value in an alarm range as defined by a property in its alarm extension.	Acknowledge the alarm, investigate and fix the condition that caused the alarm.
down	Driver communication with the parent device as configured in the extension has been lost. All proxy point children of the device report a status of "down." This status originates from a proxy point only.	Confirm that the parent device is on line and functioning correctly.
disabled	The proxy extension has been disabled. Polling stops for the point. This status originates from a proxy point only.	Enable the proxy extension.
fault	Typically, this indicates a configuration or license error. If a fault occurs following normal {ok} status, it could be a condition detected within the device, or perhaps some other fault criterion that was met.	Check the point's proxy extension's **Fault Cause** text for more information.
null	Indicates no status is available. Algorithm blocks may return a value with a null status in some cases.	Check the point configuration.
ok	Indicates that the point is functioning as expected. No status flag(s) are set.	No action required.

Status	Description	Remedy
overridden	The functioning of the point has been stopped usually by a hardware override switch.	Make a physical inspection to the device.
stale	Since the last poll update, the system has not updated the point's value within the specified **Stale Time** of its Tuning Policy. This status originates from a proxy point only.	This status clears upon receipt of the next poll value.

Enabling error logging

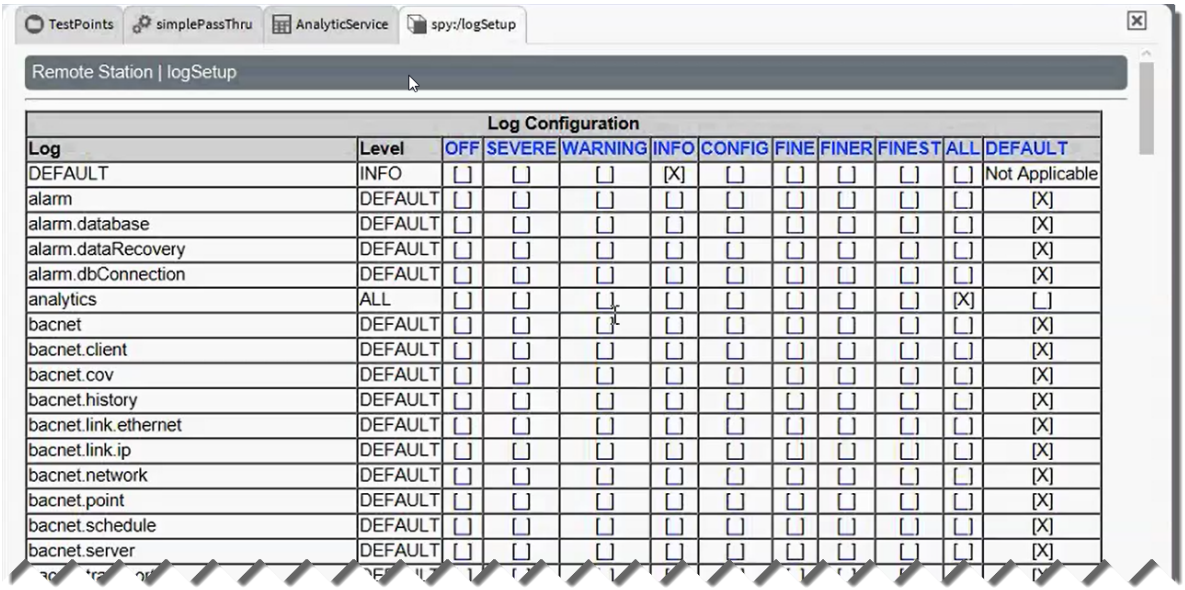
To debug, configure the module you are working with to output all error messages to the station log. This involves configuring the Spy.

NOTE:

Configure the Spy to output all messages only while debugging. This feature requires system resources and could slow operations if left on.

Step 1 Right-click the station and click **Spy**.

The Log Configuration table opens.



Log Configuration

Log	Level	OFF	SEVERE	WARNING	INFO	CONFIG	FINE	FINER	FINEST	ALL	DEFAULT
DEFAULT	INFO	[ ]	[ ]	[ ]	[X]	[ ]	[ ]	[ ]	[ ]	[ ]	Not Applicable
alarm	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
alarm.database	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
alarm.dataRecovery	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
alarm.dbConnection	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
analytics	ALL	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]	[ ]
bacnet	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.client	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.cov	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.history	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.link.ethernet	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.link.ip	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.network	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.point	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.schedule	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
bacnet.server	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]

Step 2 Configure the module level to log `ALL` messages.

Step 3 Debug the feature.

Step 4 Change the log level back to a limited number of messages.

HTTP ERROR: 500 Privileged Action Exception

If, after upgrading the Niagara software on a Windows Supervisor, you get this error when you open the station in a browser: HTTP ERROR 500, Privileged Action Exception, follow this procedure.

Prerequisites: You are connected to the Supervisor station and Workbench is open.

HTTPBasicScheme is the intended authenticator for use with the Web API protocol. Attempting to access a station using a browser URL without this authentication scheme results in the privileged access exception error.

Step 1 Open the **`baja`** palette.

Step 2 In the palette, expand **AuthenticationSchemes→WebServicesSchemes** and drag the **HTTPBasicScheme** authenticator component to the **Config→Services→AuthenticationService→AuthenticationSchemes** node in the Nav tree.

Step 3 To create a new user or ensure that an existing user is configured for HTTPBasicScheme authentication, expand **Config→Services→UserService**, double-click the user name, select **HTTPBasicScheme** for **`Authentication Scheme Name`** and click **Save**.

Step 4 Use an API platform, such as Postman to send an HTTP request to the Analytics Servlet to verify functionality. See the Niagara Analytics Web API Protocol document for more details.

Scenarios

Use this topic to read about a condition others have experienced that you may also experience, and what to do about it.

Everything was working just fine when suddenly the framework stopped working.

You may have exceeded the license point limit allowed for your system. This can happen if the **`AutoTagAnalyticPoint`** property on the **`AnalyticService`** property sheet is set to `true`. This property should be set to `false`, unless you are configuring the system for the first time.

My chart is not displaying data correctly.

Confirm that the number of bindings is correct. If more bindings are specified than are required, nothing happens. Binding support is as follows:

- Average Profile Chart supports a single binding.
- Ranking Chart supports multiple bindings.
- Equipment Operation Chart supports a single binding.
- Load Duration supports multiple bindings.
- Relative Contribution Chart supports multiple bindings.
- Spectrum Chart supports a single binding.

I configured the Time Range and Interval, but my Spectrum Chart is completely blank.

If the Spectrum Chart binding returns less than three data points (that is 0, 1 or 2 data points), the chart fails and reports an exception. To investigate, calculate the total number of expected data points by dividing **`Time Range`** by **`Interval`**. and adjust the **`Time Range`** or **`Interval`** to report three or more data points.

How do I limit the data source to average for a min/max of one day in an algorithm?

Enable **`User Request Rollup`** on the data source and set **`Rollup`** to `min` or `max` as required.

I can add an Analytic Web Chart Binding to a web widget and configure it, but I cannot seem to get it to work.

After adding the Analytic Web Chart Binding, double-click the widget to open the **Properties** view, and delete the default WbView Binding.

I tried to use the Analytic Value Binding, but where do I define the time range and rollup functions?

The Analytic Value Binding gives the current value without any rollup.

I imported histories and points under my NiagaraNetwork from my remote host and tagged them, but I cannot visualize anything.

Enable the **`Persist Fetched Tags`** on the **AX Property Sheet** of the **NiagaraNetwork** driver, right-click on the Niagara Driver and click **Actions→Force Update Niagara Proxy Points**. This should apply a direct

`n:history` tag to each **NiagaraNetwork** point where the point from the remote station has a history that has been imported to this station.

I want my algorithm to return trend results if a value was less than or equal to zero. Algorithm results show in a bound label but not on a web chart.

The bound label is likely resolving an analytic value request, whereas the web chart is resolving an analytic trend request. It it likely the control point does not have an `n:history` tag used by the framework to locate the history data. For a **NiagaraNetwork**, right-click the **Config→Drivers→NiagaraNetwork** folder in the Nav tree and click **Actions→Force Update Niagara Proxy Points**. This applies a direct `n:history` tag to any point that lacks a history extension.

For all other networks, use a Program Object in the V2 bog file to add an `n:history` tag to each point.

I'm running in a JACE-8000. My Web chart causes a server session time-out.

Check to see if you have specified a COV (change of value) point directly in the binding. A COV point that changes frequently can cause CPU spikes. As a best practice, instead of specifying the COV point directly, specify its parent in the binding. For other point types, configure a less frequent Refresh Rate in the binding to minimize CPU spikes.

I notice that, when I run PX views, my JACE-8000 slows down, and sometimes reports server session time out errors.

For best performance on the JACE-8000, limit the number of points configured in a PX view to 100–200 with no more than 200 tags, 500 history rollups and five bindings. For more complicated configurations, set up PX graphics in a Supervisor station running on a PC.

For an analytic request (binding, alert, etc.), the unit of measure output from an algorithm is not being converted correctly or it does not match the unit set in the algorithm.

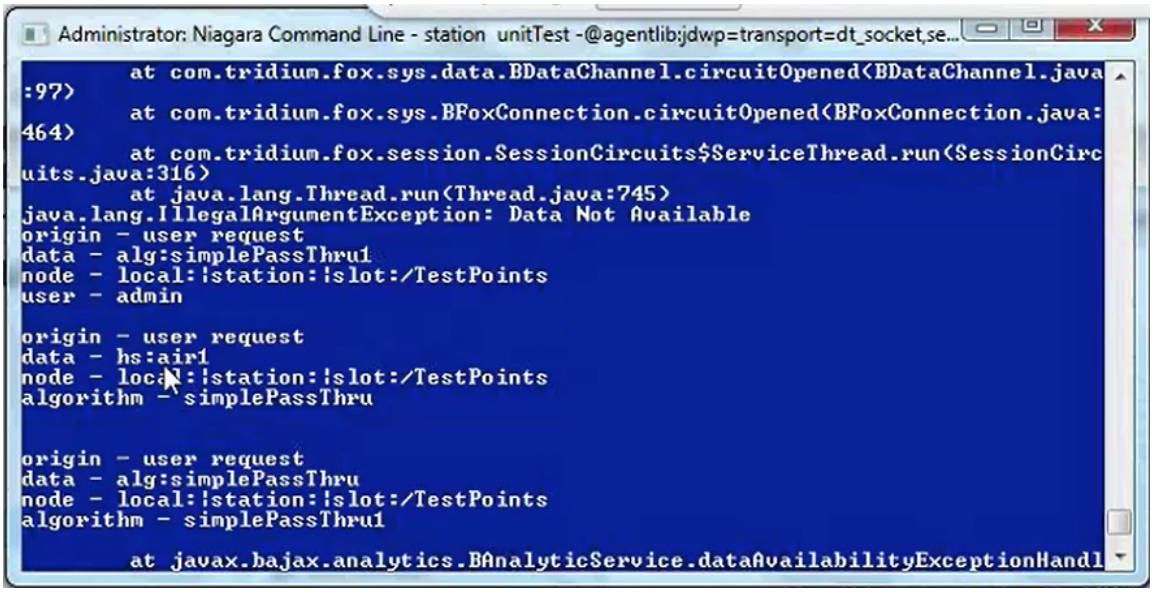
Check the algorithm to ensure that the correct unit of measure is defined (by facets on the algorithm's property sheet). Algorithms perform no unit conversion from data source to algorithm output. The unit of measure defined in the algorithm's facets is directly output with the calculated value. This makes it imperative to define the correct unit of measure on the algorithm's property sheet.

For a series of chained algorithms, for example: Algorithm 1 becomes a data source for Algorithm 2, which, in turn, becomes a data source for a Px binding, the system converts the output unit from Algorithm 1 (assuming the Algorithm 1 unit is defined in its facets) to the unit specified for the data source in Algorithm 2. If Algorithm 1 has no unit defined, and Algorithm 2 has the unit defined, Algorithm 2 applies its unit of measure to the input it receives from Algorithm 1.

I configured a data source for my web chart, but the system says that the data source is not available.

Check the station log to identify the origin of the request for data.

Figure 76 Station log



A request from a graphic is classified as a user request. This is followed by which point, node, and user are involved in this request, as well as the name of the algorithm in which the request was made. The multiple origins in the screen capture example represent nested algorithms.

If this does not help you solve the problem, open the **Property Sheet** for the **AnalyticService**, and set **`Skip Data Source Cache`** to `true`. The framework engine caches memory to improve response time. Disabling memory cache, by setting this property to `true`, causes the system to display the current error in the station log.

NOTE: When you are finished debugging, make sure you set **`Skip Data Source Cache`** to `false` again so not to impact performance.

I am trying to figure out why an alert occurred.

Check the station log. It shows which point generated the user request that triggered the alert along with the alert name.

I tagged points to be used with analytics and my AnalyticService is now in fault and all analytic requests fail. What happened?

There is more than one reason for this to happen:

- **`Auto Tag Analytic Point`** on the **AnalyticService Property Sheet** is set to `false`.
- It is a licensing issue. The number of points you can use for the framework is limited by your license. If you tag more points than you are licensed to use, the service goes into fault. Update your license to add more points or remove the extra tags.

I set up a Source Name for an alert expecting that the text and BFormat I entered would display in the alarm console. Instead, the alarm console displays the default BFormat (% node.navName%_%alert.name%) in its Source column.

You have a syntax error in your BFormat. Clicking the **Notes** button displays this message, "The BFormat value for sourceName is invalid for alert `AnalyticAlert`." where `AnalyticAlert` is replaced by the name you configured for the alert. Check the *Reference* manual for examples of BFormat syntax.

After changing an algorithm, refreshing cache can take as long as 20 minutes. What is going on?

`Refresh Cache` calculates data memory requirements again by searching all hierarchies. Depending on the hierarchical structure of your data, this could take some time.

Glossary

aggregation	The process of combining multiple pieces of the same type of data into a single value (sum, average, maximum, minimum, median, etc.) Each piece of information comes from a different data source. The request for the specific type of data starts at a specific node and travels down the data model tree aggregating all sources (points) tagged with the search argument tag.
alert	A warning regarding a condition identified by a Niagara Analytics Framework that can be routed to an alarm or used to visualize real-time and historical data.
algorithm	A formula that uses real-time values, historical trend data, and the results of calculations made by other algorithms to analyze data collected by the system.
analytics	The discovery and presentation of meaningful patterns in data. Analytics rely on the simultaneous application of statistics and computer programming to quantify performance, communicating the results on graphs and charts, as well as using results to control devices.
COV or Cov	Change-of-Value. Characterizes the option to track data based on when a value changes rather than at a consistent interval, such as every minute, 15 minutes, etc.
data definition, definition	Defines the type of information a request is looking for, such as real energy, zone temperature, air flow quantity, set point, phase A amperage, phase B amperage. Definitions are related to tags in that the definition **Id** is the tag name space and name used to search the database.
data model	A hierarchical tree structure that organizes points based on the usage or reporting of information rather than on the drivers required to manage physical devices. A typicalNiagara station is built around device drivers (lon, bacnet, modbus device, etc.). Data modeling allows you to structure information in potentially more useful ways, such as by geographic location, equipment type or responsible party.
direct tag/relation	A tag or relationship that has been manually associated with an entity.
implied tag/relation	A tag or relationship tag automatically assigned by the system to an entity.
origin entity	The object in a hierarchy from which a search begins.
request	The query for input data that seeks either a point's current or historical value.
rollup	The process of combining historical data for a single data source into one value (sum, maximum, minimum, average, etc.).
scope	In programming, the range within a program's source code within which an element name is recognized without qualification. Variable definitions are not limited to the beginning of a block of code, however, they must be declared before they can be used. In Niagara, the scope of an action applies to the selected components. The component tree is hierarchical. If you delete or move a component that contains other components, you are deleting or moving all items that are contained in that container component (its scope).

tag	A piece of semantic information (metadata) associated with a device or point (entity) for the purpose of filtering or grouping entities. Tags identify the purpose of the component or point and its relationship to other entities. For example, you may wish to view only data collected from meters located in maintenance buildings as opposed to those located in office buildings or schools. For this grouping to work, the metering device in each maintenance building includes a tag that associates the meter with all the other maintenance buildings in your system. Controllers are associated with Supervisors based on tags; searching is done based on tags. Tags are contained in tag dictionaries. Each tag dictionary is referenced by a unique namespace.
trend	The result of analyzing historical data collected by the system. A trend involves rolling up data into meaningful intervals.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W