KODARO

Haystack Driver Guide

November 12, 2020
This documents usage of the Haystack driver on the Niagara N4 platform.

OVERVIEW

This document serves as a guide for using of the Haystack serial driver on the Niagara N4 platform. This driver is intended to integrate the N4 tagging model directly into the Haystack tagging model. It provides a mechanism for two-way communication of tags between an N4 station and a third-party Haystack client. See *Appendix A: Application Notes* for further details concerning these options.

System Compatibility

This driver integrates directly to the Niagara N4 tagging database using the Haystack Rest API as defined by http://project-haystack.org/.

Supported Ops

- About
- Ops
- Formats
- Read
  - The read op deviates from the haystack protocol slightly by not following the haystack filter syntax. This is because Niagara does not strictly follow a haystack compliant database and rather uses NEQL as its tag querying system. The filter parameter should follow NEQL syntax for the filter parameter when using this op.
- Nav
- Watch Sub
- Watch Unsub
- Watch Poll
- Point Write
- His Read
  - Additional parameters are configurable on the driver to allow for history merging where multiple histories exist for a single point. Similar properties exist for choosing the primary history in this scenario. Merging is only allowed for a licensed version of the driver while the primary options are available in the free version.
- Invoke Action
- Query
  - The query op is a non-standard Haystack operation added for supporting Niagara's native query languages, BQL and NEQL. See Application notes for details on its usage (Using the Query Op).
- Commit
  - Allows for committing changes into the Niagara station component space. See Appendix A: Application Notes for examples on its usage (Using the Commit Op).
    - Add – Add a property onto a component
    - Update – Updates a property on a component
    - Remove – Remove a property from a component

Niagara Compatibility

This software will function on all Niagara N4 4.*n.nnn* platforms.

Workflow Summary

1. Module Installation
2. Adding the Driver Object
3. License Driver Object
4. Adding Network Objects
5. Configuring the Network
6. Adding Device Objects
7. Adding Points

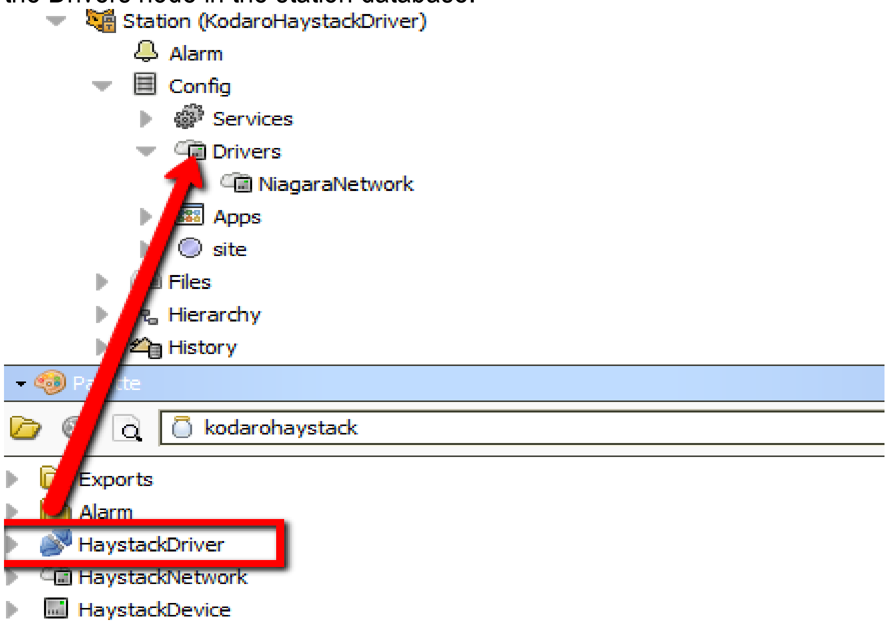
MODULE INSTALLATION

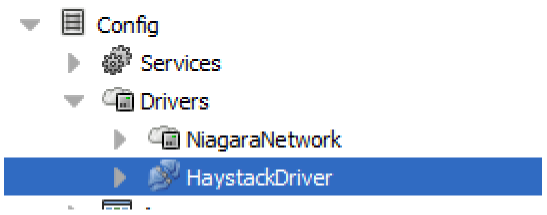
Install the driver module on the computer where Niagara N4 Workbench will be run. To install, place a copy of the file in the modules directory of your Niagara N4 installation. This is typically C:\Niagara\Niagara-4.n.nnn\modules. Restart station.

Install the module on the target station. Using a Niagara N4 Workbench where the module has already been installed, connect to the station's platform service. Go to the Software Manager and install. Restart station.

ADDING THE DRIVER OBJECT

Open the driver palette in the engineering tool. Copy and paste the HaystackDriver object under the Drivers node in the station database.





LICENSING THE DRIVER OBJECT

Many of the features of this driver are provided free of charge and do not require a license. Only some of the more advanced features require a license. Please confirm your applications needs with the following tables to determine if the functions you need require a license.

Server Features		
Feature	**Free**	**Licensed**
About Web Op	✓	✓
Formats Web Op	✓	✓
Read Web Web Op	✓	✓
Nav Read Web Op	✓	✓
Watch Sub Web Op	✓	✓
Watch Unsub Web Op	✓	✓
Watch Poll Web Op	✓	✓
Point Write Web Op	✓	✓
His Read Web Op	✓	✓
Invoke Action Web Op	✓	✓
Query Web Op	✓	✓
Commit Web Op	✗	✓

Client Features		
Feature	**Free**	**Licensed**
Convert SkySpark Sparks to Niagara Alarms	✗	✓
Export (push) Hierarchies	✗	✓
Merge Histories from Station History Imports	✗	✓

Miscellaneous Features		
Feature	**Free**	**Licensed**
Acknowledge Alarms via Invoke Action Web Op	✓	✓
Watch/Subscribe Niagara Alarms	✓	✓
Query System with BQL via Query Web Op	✓	✓
Query System with NEQL via Query and Read Web Ops	✓	✓
Integrate Directly into Niagara's Tagging System	✓	✓
Add/Remove/Update Tags via Invoke Action Web Op	✓	✓
Add New Components to Station Commit Web Op	✗	✓
Remove Components from the Station via Commit Web Op	✗	✓
Modify Properties of Station Objects via Commit Web Op	✗	✓

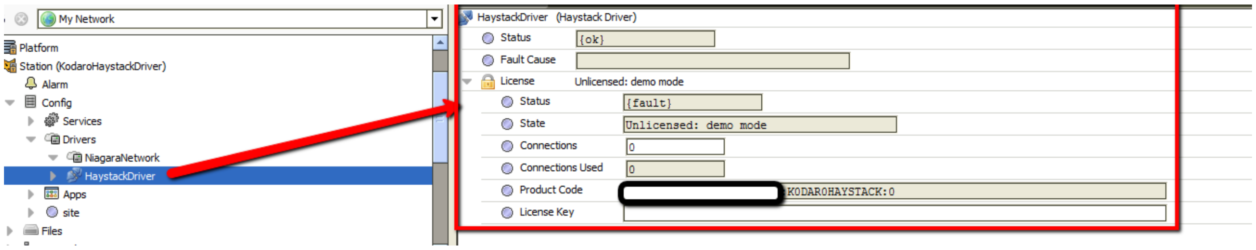
Licensing is based off connections. Each connection is defined as a connected device. This means any device added to any network will count against a connection whether that connection is active or not.

The ability of the driver to receive haystack rest ops does not rely on connections. If it is licensed for any number of connections, all web op features will operate regardless of the amount incoming connections. The majority of the web ops do not require any license to function but some do require a license be present. Please consult the licensing tables in this section to determine whether or not you need to purchase a license to unlock the web ops you will need for your application.

The demo status of the driver will last for two hours at each station start. After demo mode expires, the station must be restarted to resume operation, but it is otherwise safe to build a database.

The licensing object is located on the property sheet of the HaystackDriver. It has the following properties.

- Product Code – Text automatically generated by the driver that is needed to generate a license key.
- License Key – Where the key to validate the license must be entered.
- Connections – The number of devices under all networks to license.



HaystackDriver (Haystack Driver)
Status {ok}
Fault Cause
License Unlicensed: demo mode
Status {fault}
State Unlicensed: demo mode
Connections 0
Connections Used 0
Product Code [redacted] KODAROHAYSTACK:0
License Key

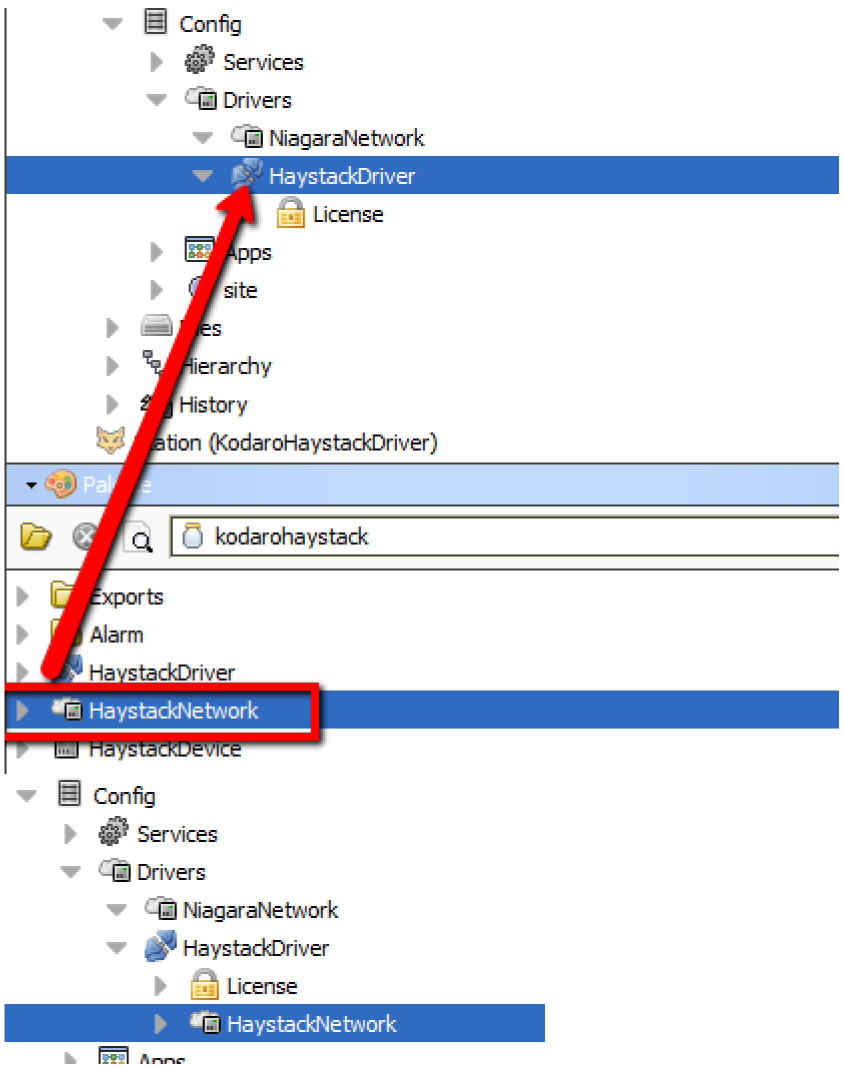
Set the number of devices you plan to use. Copy the value of the “Product Code” property that is automatically generated. You should highlight the value and copy it by pressing CTRL-C. Send the product code to your Kodaro representative. They will respond with a text string for you to enter in the “License Key” property.

You must restart the station after changing the “License Key”.

The exact text and case of the product code and license key are case sensitive and critical to licensing. Please do not send screen shots. Highlight the text, copy it and paste into an email when requesting a license.

ADDING NETWORK OBJECTS

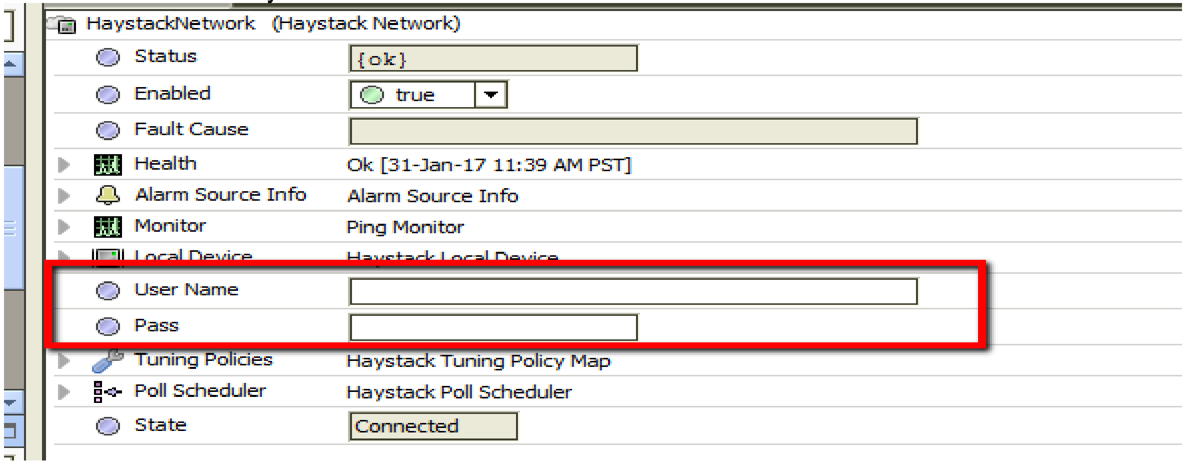
Open the driver palette in the engineering tool. Copy and paste the appropriate network object under the driver node in the station database.



CONFIGURING THE NETWORK

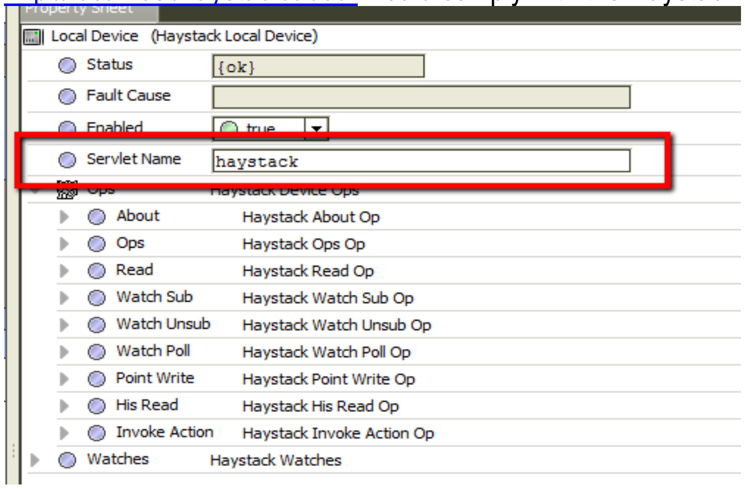
The driver uses the ping monitor to determine when to try to establish a connection. Wait for the driver to connect or manually invoke the ping action.

There are very few properties to configure to get a HaystackNetwork up and running. The only properties that may need to be configured are the User Name and Pass(password). If these are set, they will be used to authenticate all device connections. If these are not set, they can be set on each device individually and/or overridden on the device connection.



HaystackNetwork	(Haystack Network)
Status	{ok}
Enabled	true
Fault Cause	
Health	Ok [31-Jan-17 11:39 AM PST]
Alarm Source Info	Alarm Source Info
Monitor	Ping Monitor
Local Device	Haystack Local Device
User Name	
Pass	
Tuning Policies	Haystack Tuning Policy Map
Poll Scheduler	Haystack Poll Scheduler
State	Connected

The Local Device object holds information about what Haystack Operations are available for a client to communicate with as well as a way to track point subscriptions handled for the watchSub/watchPoll ops. There is only one property that may need to be changed if desired, Servlet Name. Servlet name identifies the end point for all Rest API calls, i.e. http://localhost/haystack/about would comply with the Haystack About REST op about.

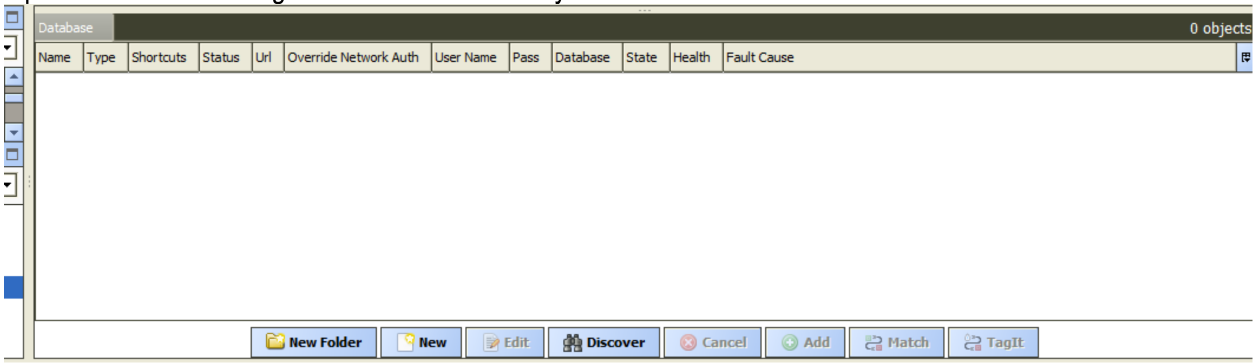


Local Device	(Haystack Local Device)
Status	{ok}
Fault Cause	
Enabled	true
Servlet Name	haystack
Ops	Haystack Device Ops
About	Haystack About Op
Ops	Haystack Ops Op
Read	Haystack Read Op
Watch Sub	Haystack Watch Sub Op
Watch Unsub	Haystack Watch Unsub Op
Watch Poll	Haystack Watch Poll Op
Point Write	Haystack Point Write Op
His Read	Haystack His Read Op
Invoke Action	Haystack Invoke Action Op
Watches	Haystack Watches

You can also configure the amount of resources(threads) to be allocated to Watches. More detail on this topic can be found in *Appendix A: Application Notes*

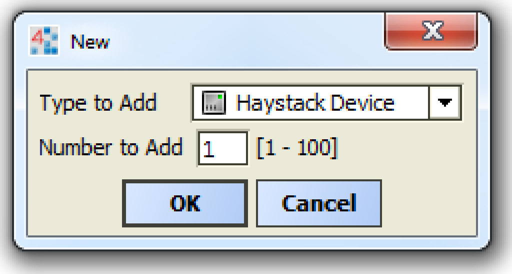
ADDING DEVICE OBJECTS

Open the *Device Manager* view of the Network you wish to add a device to.



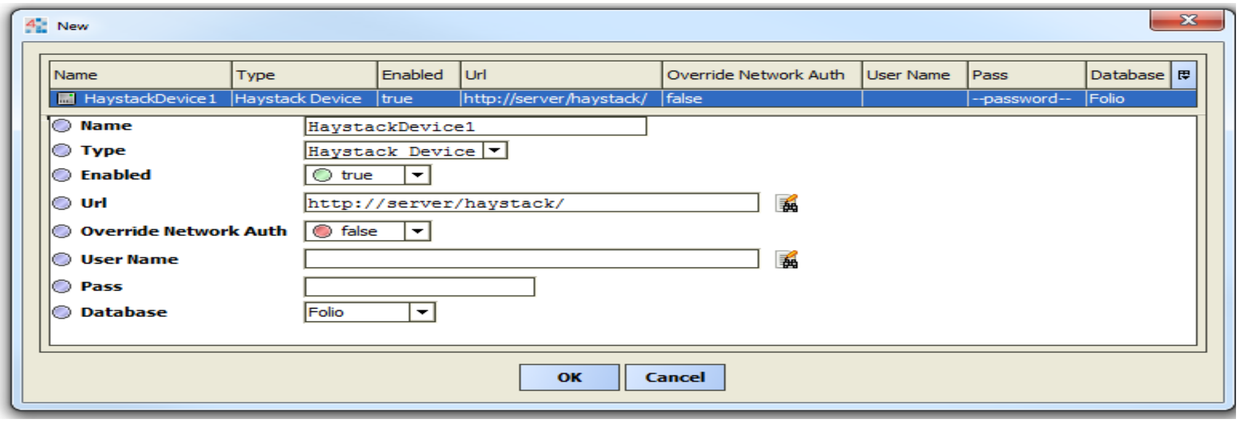
Name	Type	Shortcuts	Status	Url	Override Network Auth	User Name	Pass	Database	State	Health	Fault Cause

Use the *New* button to add a new Device. This will bring up a dialogue asking how many devices you wish to add.



Type to Add: Haystack Device
Number to Add: 1 [1 - 100]

Enter the amount you wish and select Ok. This will bring up a another window where you can configure the devices.



Name	Type	Enabled	Url	Override Network Auth	User Name	Pass	Database
HaystackDevice1	Haystack Device	true	http://server/haystack/	false		--password--	Folio

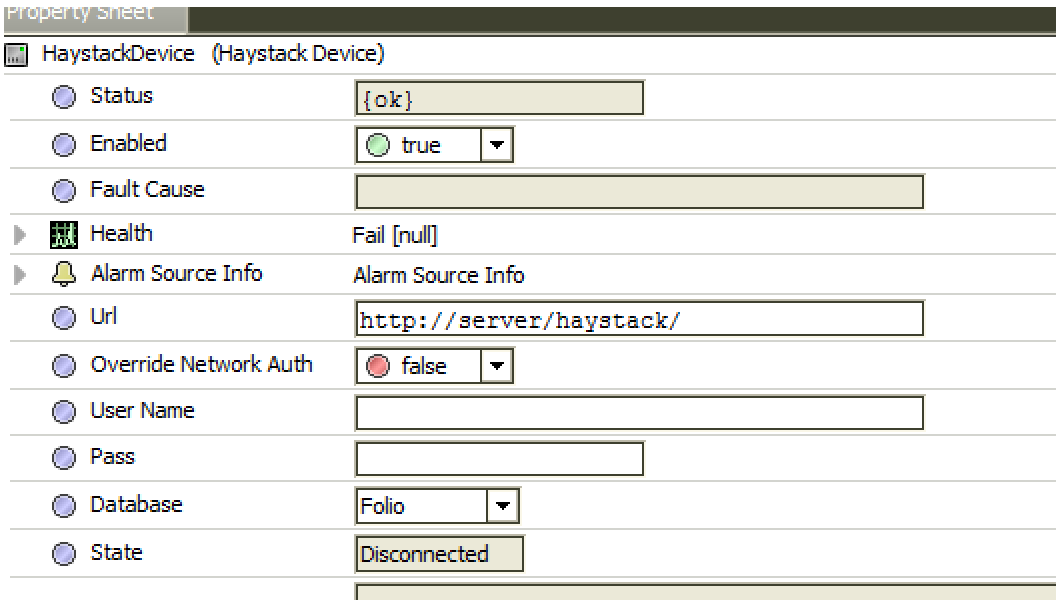
Name	HaystackDevice1
Type	Haystack Device
Enabled	true
Url	http://server/haystack/
Override Network Auth	false
User Name	
Pass	
Database	Folio

It is generally only necessary to set the Url assuming you have set authentication credentials at the network level. If network credentials have not been setup or are different from them, then it will be necessary to set the specific device authentication parameters at this time.

CONFIGURING THE DEVICE

It is generally only necessary to set the Url assuming you have set authentication credentials at the network level. If network credentials have not been setup or are different from them, then it will be necessary to set the specific device authentication parameters at this time.

a. Setting the URL – This must be a fully resolvable URL to the location of the server that accepts haystack requests. For a SkySpark(Folio) database, this format looks like: http://<ip address>/api/<project name>
b. Override Network Auth: This is false by default meaning it will use the User Name and password defined on the Network Object. If set to true, the User Name and Password defined on this object will be used instead.
c. User Name – User name to use during authentication if Override Network Auth is set to true.
d. Pass – Password to use during authentication if Override Network Auth is set to true.
e. Database – Folio and Unknown are the only two options. Folio is default as it is currently the only known haystack compliant database available.



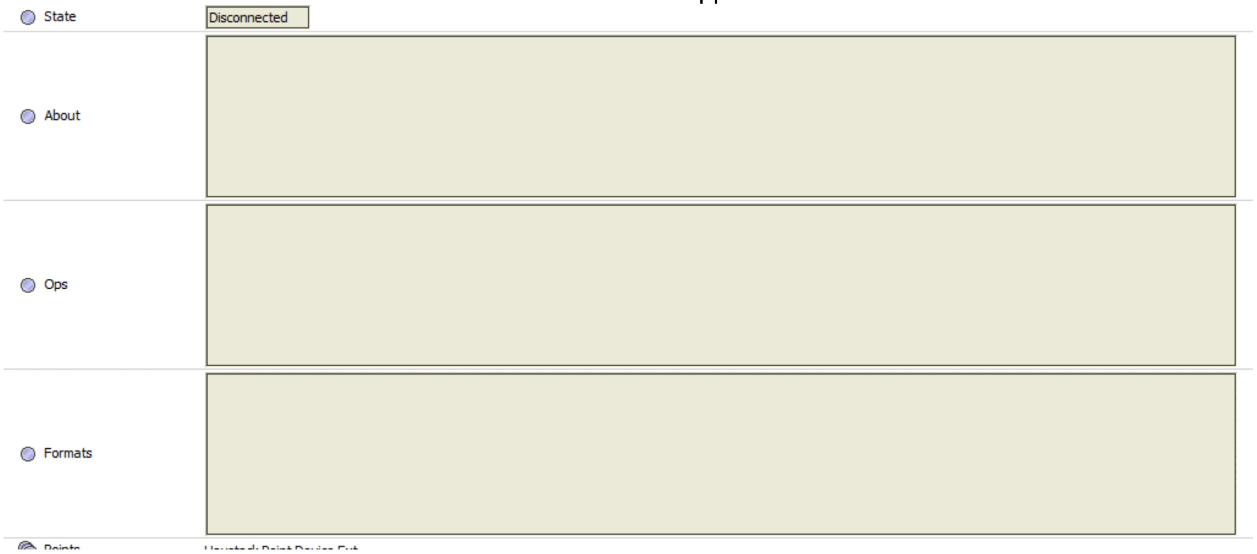
Property Sheet

HaystackDevice (Haystack Device)

Status	{ok}
Enabled	true
Fault Cause	
Health	Fail [null]
Alarm Source Info	Alarm Source Info
Url	http://server/haystack/
Override Network Auth	false
User Name	
Pass	
Database	Folio
State	Disconnected

CONNECTING TO A DEVICE

a. To make a connection, execute the connect or ping action on a device to make the connection. If this is successful, this will populate meta data about the haystack database that has been connected to.
   i. About – Shows basic data about project/database
   ii. Ops – Shows which operations are supported.
   iii. Formats – Shows which data formats are supported



State: Disconnected
About
Ops
Formats

COMPONENT REFERENCE

HaystackDriver

This is the root node of the driver. It represents a tree of Haystack networks.

Properties:

- **License** – See the section title “Licensing”.

HaystackNetwork

This represents a container for Haystack Devices. It is a network-level component in the NiagaraN4 architecture and has standard network component properties (see "Driver Architecture / Common network components" in the *NiagaraN4 User Guide* for more information).

The following properties are unique or have special importance:

- **User Name** – The user to authenticate connections with
- **Pass** – The password to authenticate connections with
- **Local Device** – Device responsible for listening to Haystack REST API calls.
  - **Servlet Name:** URL endpoint to this station to send all REST API calls. Format Example: http(s)://<ipAddress/<Servlet Name>/<op name>
  - **Ops –** Contains the Haystack REST ops supported
  - **Watches –** A new watch Object will be added each time a new watchSub is received. Every watch runs on its own thread.
    - WatchId : The reference used by the Haystack client to poll this watch.
    - Lease Time: How long to keep this watch alive in between polls.
    - Last Poll: The last time this watch was polled
    - Subscriptions: The amount of components this specific watch is currently subscribed to.

HaystackDevice

This is a device-level component in the NiagaraN4 architecture and has standard device properties (see "Driver Architecture / Common device components" in the *NiagaraN4 User Guide* for more information).

The following properties are unique or have special importance:

- **Url** – This is the fully qualified URL that points to the Haystack REST end point you wish to connect to.

- **Override Network Auth** – When set to true, this indicates to the device to use it User Name and Pass properties instead of the ones defined on the network.
- **User Name** – The user to authenticate connections with
- **Pass** – The password to authenticate connections with
- **Database** – A list of known Haystack compliant databases that may have some intricacies not specifically defined by the Haystack protocol but are handled by this driver.
  - **Folio** – The database name of SkySpark installations.
  - **Unknown** – Any Haystack database wished to be connected to but is not found in this list. No special care outside of the strict Haystack protocol will be followed when communicating with a device set as such.

HaystackProxyExt

The following properties are unique or have special importance:

- **Address** – This is the name used in a point log to poll the value.
- **Device Facets** – The facets are used to map values into Niagara. For Boolean and Enum points, it is important the trueText/falseText or enum range be set.

TROUBLESHOOTING

Problems in General

- Is the driver licensed? Has the demo mode expired?
- Turn on debug and watch the console. Anything obvious?
    - Watch the Application Director console.
- Performance issues
    - Look at the Thread Pool on the Haystack Driver property sheet
        - What are the Max Threads set to? Every network, device and export require one thread each. If there this value is less than the total of networks, devices and exports, you will need to increase it.
    - Look at the Web Threads on the Haystack Network->Local Device->Ops property sheet.
        - What are the Max Threads set to?
        - How many simultaneous web requests are being made to the haystack driver?
    - Look at the Op Timeout on the Haystack Network->Local Device->Ops property sheet. Set this to a reasonable max time each web request should need to complete before causing a timeout.
- Try to rule out the driver.
    - Is there an issue with a read web op? Run the NEQL query directly in Niagara to check for syntax issues.

Trouble Connecting

- Check license state.
- Try logging into the device connection location directly with the same credentials configured in the device.
- Login to Niagara with expected credentials and test the /about endpoint.
- Turn on debug
    - Anything printing on console?

Export Problems

- Check license state.
- Is device connected?
- Try turning on debug mode.

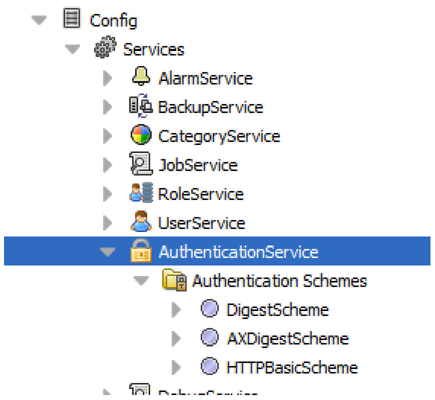
APPENDIX A: APPLICATION NOTES

Authentication Considerations

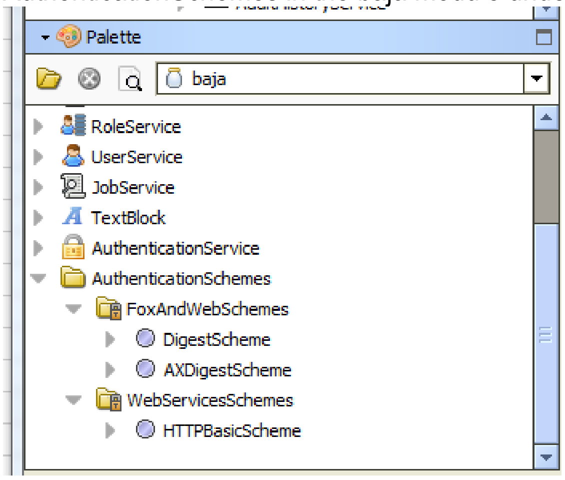
When authenticating between Niagara and SkySpark, there are a few items to consider.

1. What version of SkySpark are you running?
    a. 2.1.15 and below – This requires a Niagara user configured with the DigestScheme Authentication
    b. 3.0+
        i. For Niagara 4.2 and below, this requires a Niagara user configured with the HttpBasicAuth Authentication.
        ii. For Niagara 4.3 and above, this requires a Niagara user configured with the DigestScheme.
2. Do I have the correct authentication scheme in my station?

    After determining which authentication scheme you need, you can check if you have it available by going to your AuthenticationService under the services container directly under config in your station.

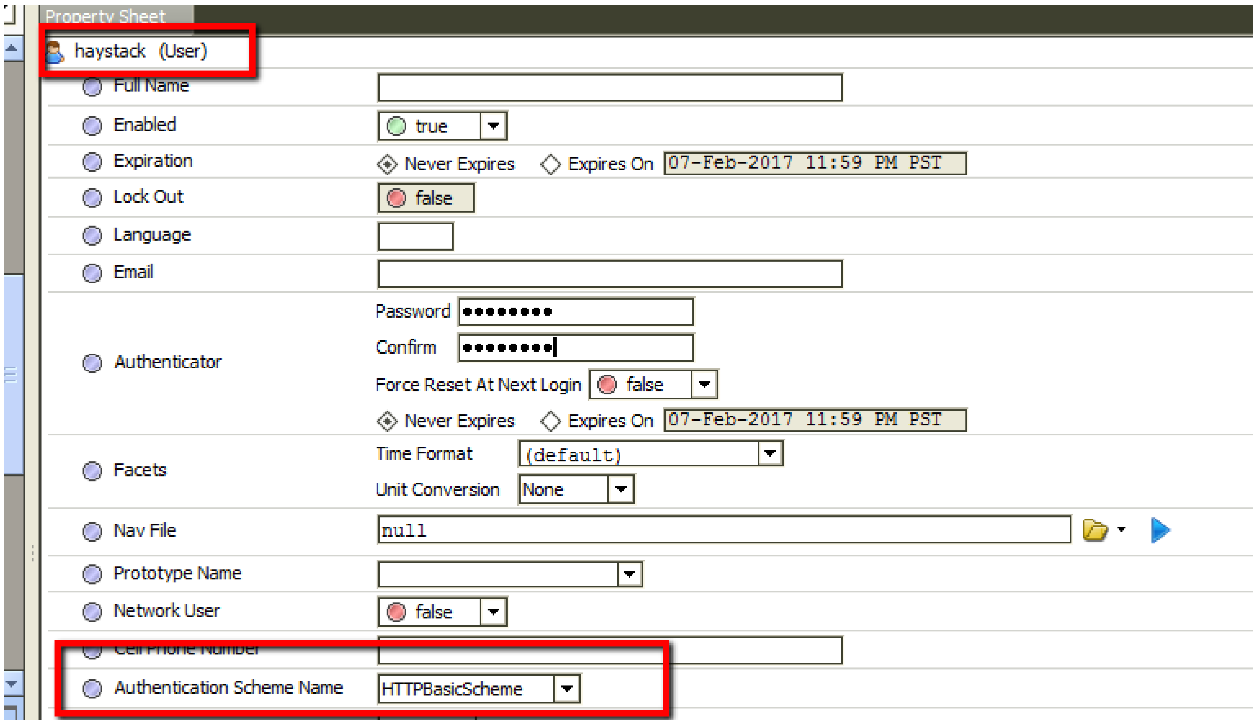


If you are missing an authentication scheme you need, you can find the built-in Niagara AuthenticationSchemes in the baja module under AuthenticationSchemes in the palette.



3. How do I configure a user to allow a connection into the station for haystack?

Assuming you have now installed the correct authentication schemes you need for the version of SkySpark you are using, you need to create a new user or configure an existing one to have this authentication scheme available for use. In this example, I have a user called haystack and assuming SkySpark 3.0+, I need to set the AuthenticationScheme to HttpBasicAuth. This can be done directly on the property sheet of the user.

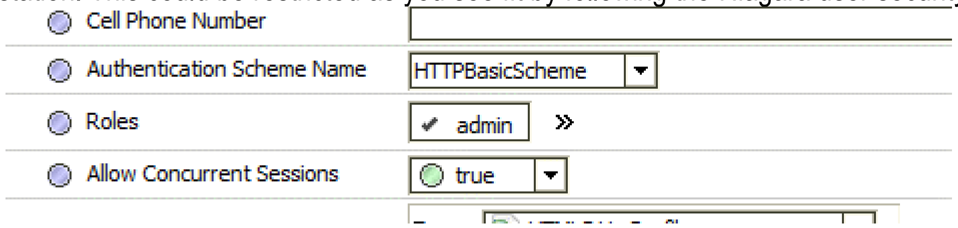

Property Sheet

haystack (User)

Full Name	
Enabled	true
Expiration	Never Expires / Expires On 07-Feb-2017 11:59 PM PST
Lock Out	false
Language	
Email	
Authenticator	Password ●●●●●●●●● Confirm ●●●●●●●●● Force Reset At Next Login false; Never Expires / Expires On 07-Feb-2017 11:59 PM PST
Facets	Time Format (default); Unit Conversion None
Nav File	null
Prototype Name	
Network User	false
Cell Phone Number	
Authentication Scheme Name	HTTPBasicScheme

4. What about access restrictions?

The Haystack read and query op both respect the Niagara roles model. This means that even if I set my haystack user up with the correct Authentication Scheme but failed to give it role access, it would never allow logging in, or it might allow logging in but not able to discover any points. By giving the user admin role access in this example, I'm assuming anybody that has this user info should have full access to the station. This could be restricted as you see fit by following the Niagara user security and role model.

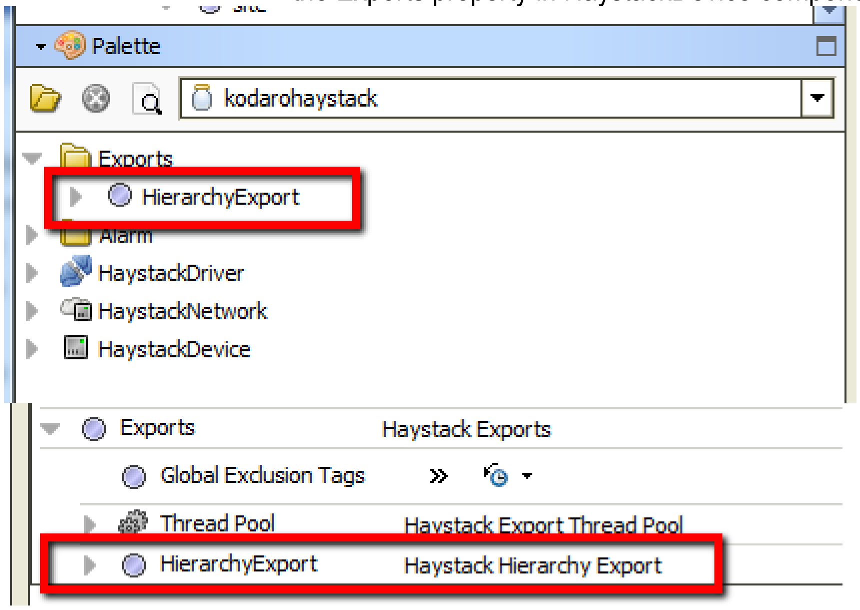


Cell Phone Number	
Authentication Scheme Name	HTTPBasicScheme
Roles	✓ admin
Allow Concurrent Sessions	true

Exporting the Niagara Database into a Haystack Database

The export method of the Haystack Driver is designed to be unidirectional and push the data from Niagara into the Haystack Database. This is designed if you wish to keep all configurations of Haystack tagging done at the station.

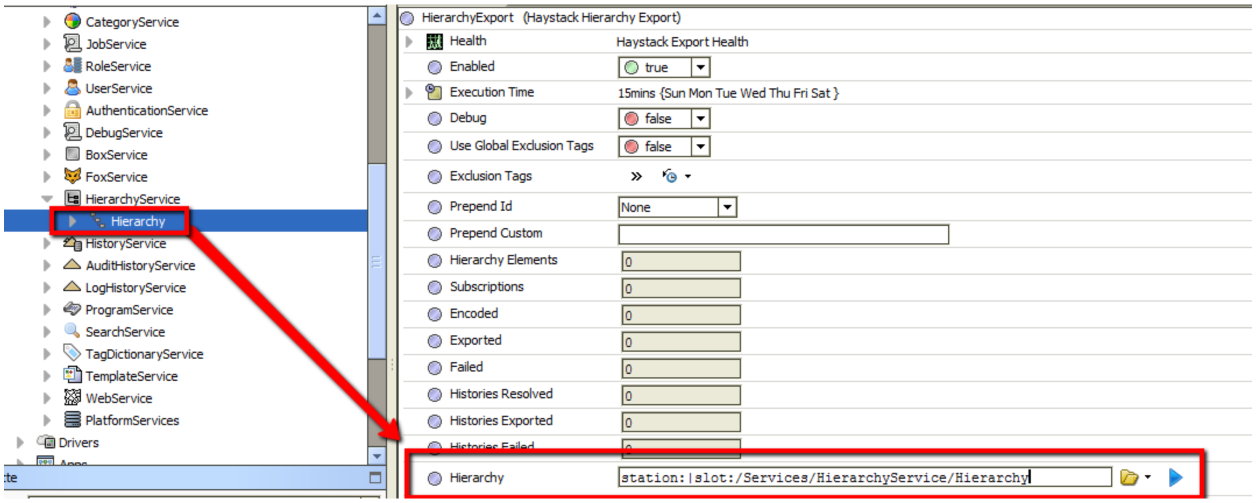
1. Data is exported via a constructed hierarchy in the N4 station. Construction of these are beyond the scope of this document but a few restrictions to using these exist:
    a. Exporting with the use of anything other than QueryLevelDef and RelationLevelDef components when creating the hierarchy are not supported.
    b. When using a RelationLevelDev, Inbound must be true and RepeatRelation must be false.
    c. Open the kodarohaystack palette and drag a HierarchyExport component under the Exports property in HaystackDevice component added in step 5.



Exports	Haystack Exports
Global Exclusion Tags	
Thread Pool	Haystack Export Thread Pool
HierarchyExport	Haystack Hierarchy Export

d.Configuring the HierarchyExport

1. Reference the Hierarchy from the HierarchyService that you wish to export



HierarchyExport (Haystack Hierarchy Export)

Health	Haystack Export Health
Enabled	true
Execution Time	15mins {Sun Mon Tue Wed Thu Fri Sat }
Debug	false
Use Global Exclusion Tags	false
Exclusion Tags	
Prepend Id	None
Prepend Custom	
Hierarchy Elements	0
Subscriptions	0
Encoded	0
Exported	0
Failed	0
Histories Resolved	0
Histories Exported	0
Histories Failed	0
Hierarchy	station:|slot:/Services/HierarchyService/Hierarchy

2.Enable Subscriptions – Defaulted to false. When set to true, will open a subscription to all components resolved in the hierarchy defined. When subscribed, points that have value changes will be pushed immediately in between exports to maintain live status values.

3.Inherit Relations – Defaulted to true. Inherits relationships from the hierarchy “parent” relationships. This is needed since N4 does not hold full relationship tree from every parent. For example, a point referencing (related to) a piece of equipment will have no idea what site that equipment is for but SkySpark/Folio needs to have this defined. By setting this to true, you will create an export of data that contains this level of information.

4.Export Histories – Defaulted to true. Define whether to export histories found on the components defined in the hierarchy.

5.Export Histories Only – Defaulted to false. When set to true **and** *Export Histories”* property is true, tags will not be exported, except for on initial export. Any missing ID determined to exist in the receiving end of the export will be added with all available tags. Subsequent exports will not push any tags if the record already exists. Exceptions exist to this behavior, see *“Include Status Tags With Histories”.*

6.Include Status Tags With Histories – Defaulted to true. When this is set to true, status tags will also be pushed instead of the entire tag suite.

7.Use Time Zone – Defaulted to False. Allows a complete override of the local station time zone for all time zone instances for historical data during the export.

8.Execution Time – Standard Niagara trigger that can be configured for Interval, Daily or Manul exports.

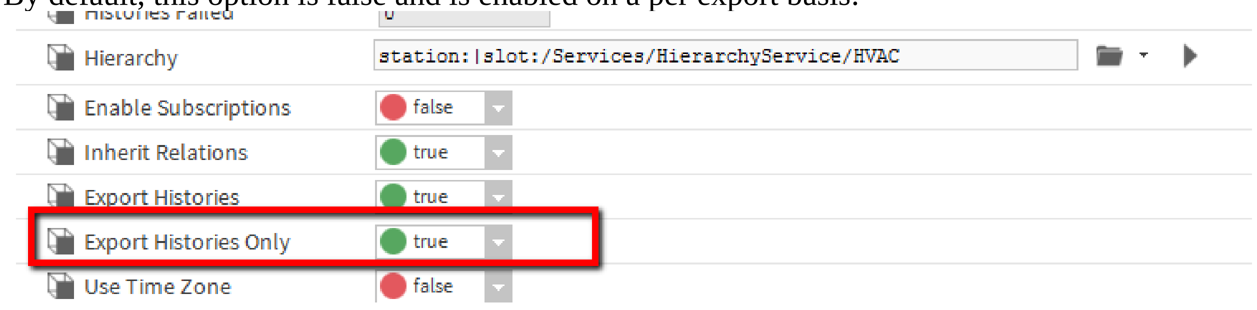
9. History Options – Complete usage examples of these properties can be found in the Application notes of this guide.

- Merge – Allow merging multiple histories resolved for the same point
- Include Deleted – This will allow histoires that were created by a history extension that has been deleted to be resolved.
- Primary History – Determines which history will be primary for time stampe conflicts when multiple histories are resolved for the same point. Local will indicate any history created on the station where the driver is currently running. Remote is any history that was generated on another station, i.e. a Niagara Network history import.
- Algorithm – This refers to the method by which to allow multiple histories. If all is selected, this will allow multiple local and multiple remote histories to be resolved. If mostRecent is selected, this will only use the history that has the most recent last record from available local and remote histories meaning that at most, only 2 histories could be used in the final merge, one local, and one remote if they existed.

History Only Export

The previous section discusses a full export of a hierarchy and its configuration. This comes with the caveat of making Niagara the primary tag database and each subsequent export will try to enforce its tags over the receiving end. As of version 1.0.22, an export only option has been implemented when only historical data is desired to be updated and you would like to pass tag management to the receiving end.

By default, this option is false and is enabled on a per export basis:



Histories Failed	0
Hierarchy	station:|slot:/Services/HierarchyService/HVAC
Enable Subscriptions	false
Inherit Relations	true
Export Histories	true
Export Histories Only	true
Use Time Zone	false

When this is option is enabled, the normal export routine is unaffected except for updating of tags. This means the life cycle of an export remains largely the same:

Export Life Cycle without Export Histories Only:

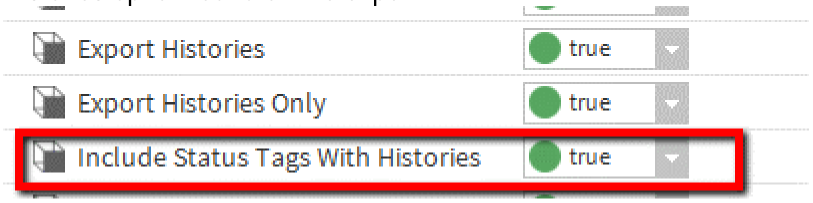
1. Resolve and encode the hierarchy
2. Determine which components need to be updated and which need to be added

3. Determine which tags need to be removed
4. Update existing records **(skipped with history only export option)**
5. Add missing records

By skipping step 4 in the process, this means a brand-new export will create all the records with the Niagara tags that are present during the export as well as its historical data. Without this record creation, Niagara has nowhere to attach and push its historical data to so it must create an initial record in the system it is exporting to, so this is required for any new records.

All other exports that follow will now only push historical data because it will have no records that do not exist in the other system. This of course can change if components are added on Niagara's end or deleted from the receiving end of the export, but it leaves the onus of the tag management now outside of Niagara to whatever system is on the other side of this export.

In addition to only histories being exported, you can also enable additional paroperties to keep data points up to date with their current values and statuses. This is achieved by enabling the Include Status Tags with Histories option found on the export.



Export Histories	true
Export Histories Only	true
Include Status Tags With Histories	true

With this option enabled, step 4 in this process is not 100% ignored anymore but modified. Instead of skipping any update process of tags, the tags that are updated are limited to only tags that have been identified as valid status flags. These contain the following:

cur, curErr, curVal, curStatus, slotPath, niagaraStatus, writeLevel, writeStatus, writeErr, writeVal

Some status values during export are adjusted to deal with their associated database. When you set the device connection database to *Folio*, this maps to the "write" values as both systems tend to want control of these for their own internal systems. This setting adjusts the above raw status values to the following values:

cur, curErr, curVal, curStatus, slotPath, niagaraStatus, niagaraWriteLevel, niagaraWriteStatus, niagaraWriteErr, niagaraWriteVal

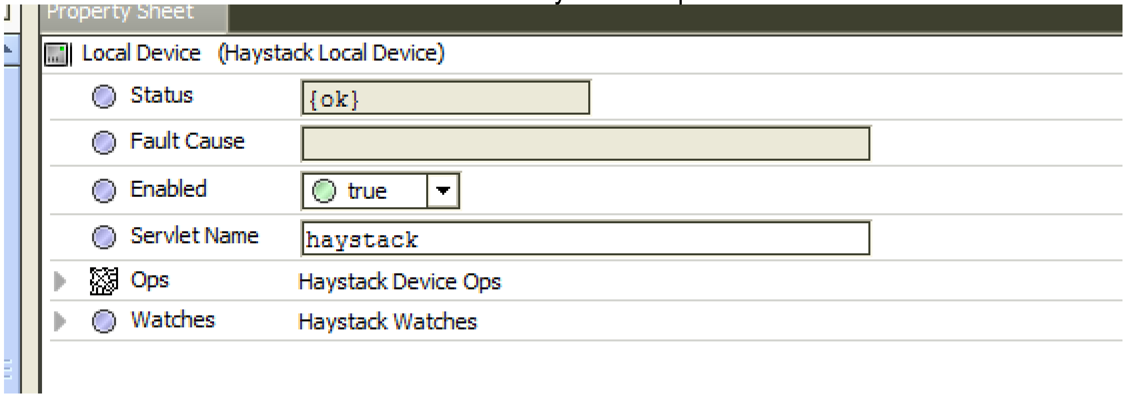
Which is mapping as follows:
writeLevel -> niagaraWriteLevel
writeStatus -> niagaraWriteStatus
writeErr -> niagaraWriteErr
writeVal -> niagaraWriteVal

Using SkySpark to Integrate to a Niagara Station

The Haystack Driver can act as a Haystack server which can be used by any third-party Haystack client to integrate to. The following section describes this process as it pertains to the SkySpark platform.

For this method to work, you must make sure that the LocalDevice found on the HaystackNetwork is enabled and in an “ok” state. Take note of the servletName, this will be where you point the SkySpark Haystack connector. The example below assumes the default value of “haystack” and that the SkySpark instance and Niagara station are running on the same system and uses LocalHost in lieu of a specified IP address.

1. Make sure that the LocalDevice is ready. This requires a valid servletName and the Status is ok.

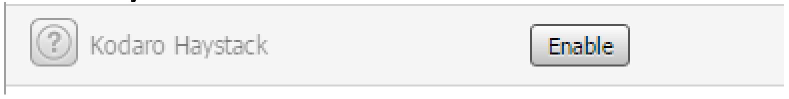


Local Device (Haystack Local Device)

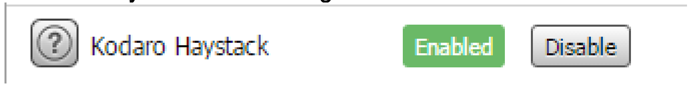
Status	{ok}
Fault Cause	
Enabled	true
Servlet Name	haystack
Ops	Haystack Device Ops
Watches	Haystack Watches

2. Install the kodaroHaystackExt.pod into SkySpark. This is included in the driver download. This is an extension pod that has some pre-built axon methods that will help to get a Niagara Station quickly integrated into a SkySpark installation. These methods serve as a guide to show how to get data from Niagara into SkySpark and how to modify data from SkySpark back to Niagara. These functions do not attempt to satisfy all possibilities but rather give a basic understanding on the process of using the Haystack Driver (Niagara) and Haystack Connector (SkySpark) to integrate a Niagara database into SkySpark. Under many circumstances these functions should be modified to meet the unique needs of your project. The extension pod does not need to be installed for the Kodaro Haystack Driver to work.
3. Once you have installed the kodaroHaystackExt.pod, you will find a Kodaro Haystack extension now available in the Extensions list in SkySpark. By default, this is disabled. Once enabled, this will make the example axon code available for use.

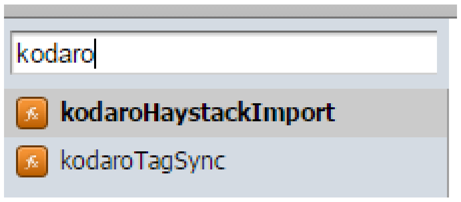
   Kodaro Haystack when first installed:



   Kodaro Haystack after being enabled:



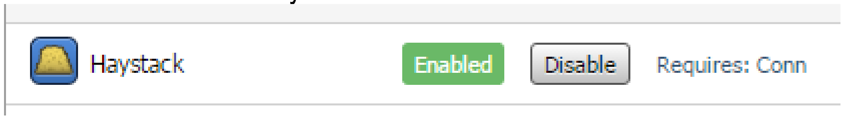
4. Once the Kodaro Haystack Extensions is enabled, you can verify the example scripts are available in the Func App. By typing kodaro in the search bar, you will get the list of kodaro scripts from this extension pod as they both start with kodaro. This search is performed from the Docs tab.



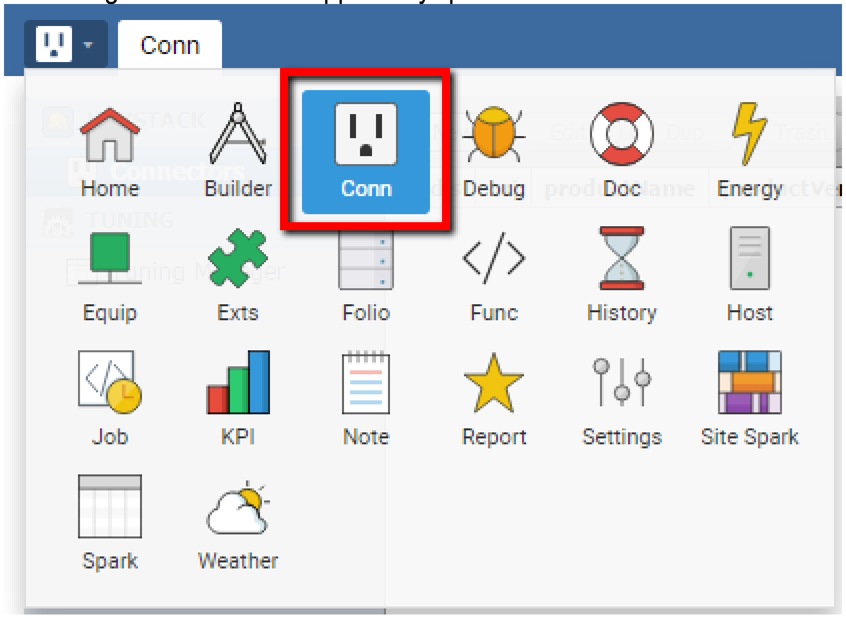
5. By clicking on any of these axon scripts, you will receive the full source code as well as a description of it's function. The latest information and source code is maintained in the scripts themselves so should always be referred to for the latest information rather than in this document.

   At this point it is assumed you have installed the kodaroHaystackExt.pod into SkySpark, enabled it and have a running Niagara N4 station with the Kodaro Haystack Driver with a Local Device ready to receive Haystack Rest Ops.

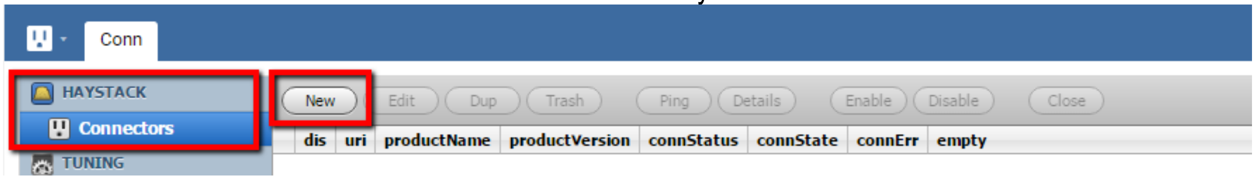
6. With everything installed on SkySpark, it is time to setup the Haystack Connector. See https://skyfoundry.com/doc/docTraining/HaystackConn for full details on this.
   a. Make sure that the Haystack Extension is enabled



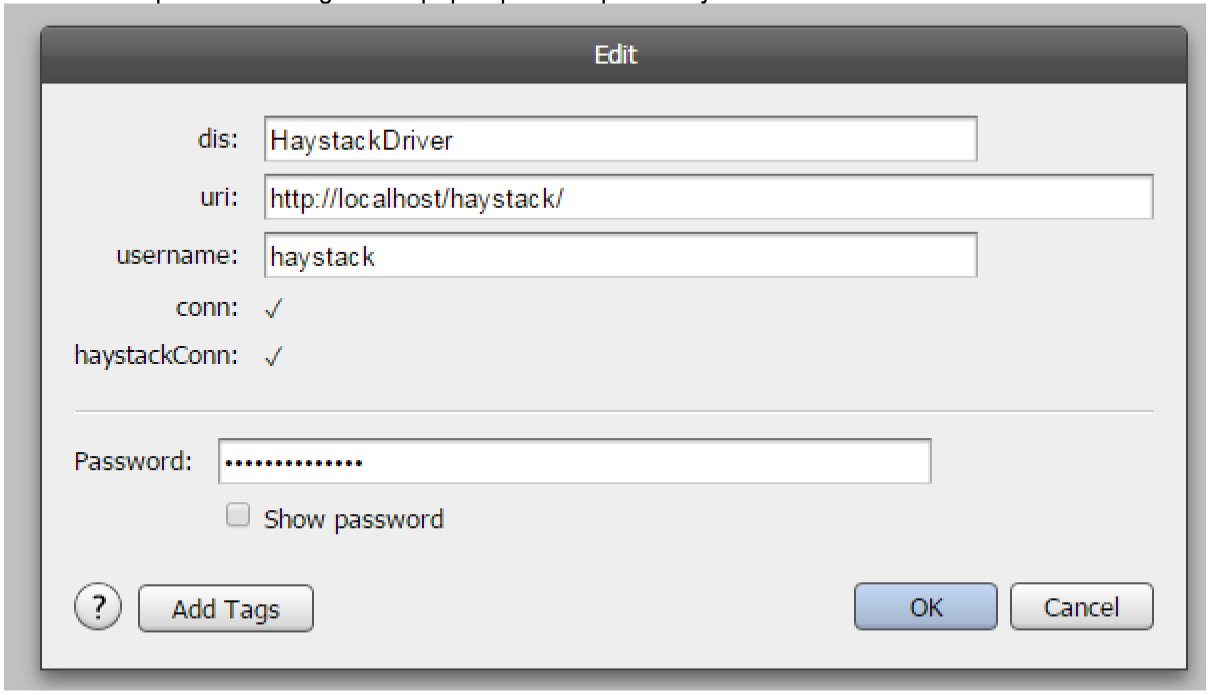
   b. Navigate to the Conn App in SkySpark



c. Select the "New" button to create a new Haystack Connector.



d. Complete the dialogue that pops up to setup the Haystack Connector.



dis: HaystackDriver

uri: http://localhost/haystack/

username: haystack

conn: ✓

haystackConn: ✓

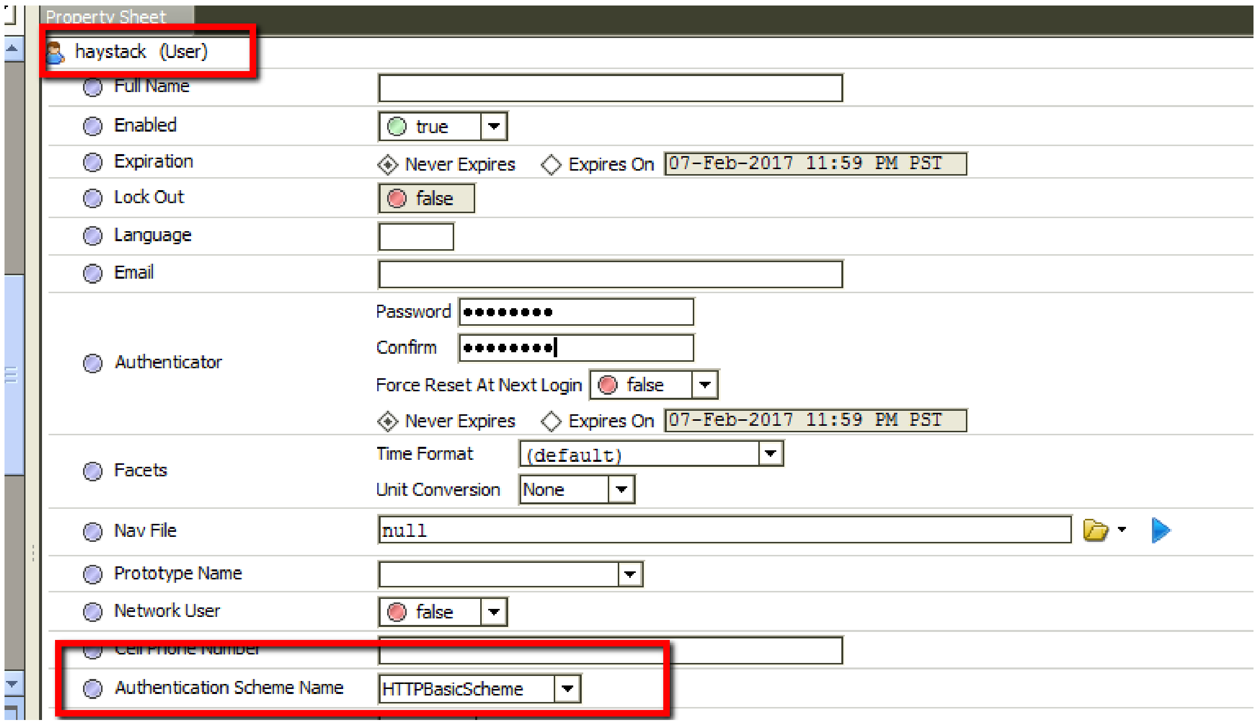
Password: •••••••••••••••

Show password

i. dis – The display name of this connector. Multiple connectors can be created so naming this may be necessary when handling a system requiring more than one.

ii. uri - In this example, as stated before, is running on the same computer that is running the Niagara driver so we set the uri to point to http://localhost/haystack, haystack being the servletName configured in the LocalDevice of the driver and localhost referencing itself as the IP.

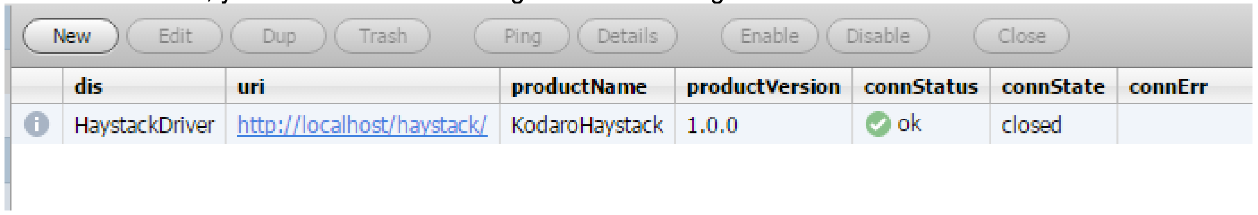
iii. username – This must be a user that exists in the Niagara N4 station that has been configured for HttpBasicScheme (for SkySpark 3.0.X) or DigestScheme (for SkySpark 2.1.X) in it's Authentication SchemeName.



haystack (User)	
Full Name	
Enabled	true
Expiration	◈ Never Expires ◇ Expires On 07-Feb-2017 11:59 PM PST
Lock Out	false
Language	
Email	
Authenticator	Password ●●●●●●●●● Confirm ●●●●●●●●● Force Reset At Next Login false ◈ Never Expires ◇ Expires On 07-Feb-2017 11:59 PM PST
Facets	Time Format (default) Unit Conversion None
Nav File	null
Prototype Name	
Network User	false
Cell Phone Number	
Authentication Scheme Name	HTTPBasicScheme

iv. Password – Password for the Niagara user.

e. Once the Haystack Connector has been created you should see this in the connector table. You can manually invoke the ping method if it doesn't connect automatically. Once it connects, you should see something like the following:

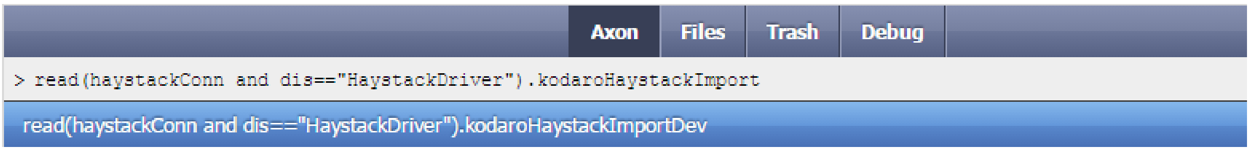


dis	uri	productName	productVersion	connStatus	connState	connErr
HaystackDriver	http://localhost/haystack/	KodaroHaystack	1.0.0	✓ ok	closed	

7. Now that we have connected SkySpark to Niagara, it's time to get data into SkySpark. We can accomplish this by examining the kodaroHaystackImport axon script found in the kodaroHaystackExt.pod. Since the full contents of the axon scripts are documented in the scripts themselves, the intention of this section is to describe its usefulness in an application context rather than trying to explain how it functions.
   a. What does the script do?
      The script can be broken down into 3 steps.
      1. Queries for all components in the Niagara station with the *hs:site* tag. All sites that are found are added into SkySpark.
      2. Queries for all components in the Niagara station with the *hs:equip* tag. All equipment that is found are added into SkySpark.
      3. Queries for all components in the Niagara station with the *hs:point* tag. All points that are found are added into SkySpark.

During the query process, the Haystack Read Op when handled in the Niagara Haystack Driver acts much like the Hierarchy Exporter when Inherit Relations is true. This means that if there is a relationship on a component, it is followed and continues up the relationship path to build a direct reference in the SkySpark database. This means a point with an equipRef but no siteRef would inherit a siteRef from the equipment it was pointing to, if that equipment had a siteRef.

b. How do you call the script?
   The script takes a Haystack Connector as it's one parameter. This is the connector that's used to query Haystack Ops to complete the import.
   Example:

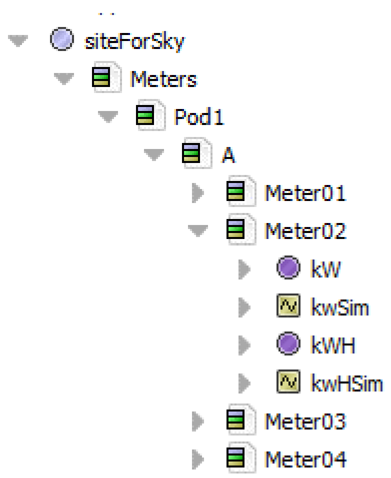


```
> read(haystackConn and dis=="HaystackDriver").kodaroHaystackImport
read(haystackConn and dis=="HaystackDriver").kodaroHaystackImportDev
```

The Axon code here first queries for Haystack Connectors (haystackConn) that are named "HaystackDriver" (dis). It then calls the kodaroHaystackImport function and passes in that result as the connector used. The result is a table of all the points added during the process. The sites and equips are not shown in the result but can be easily queried with Axon.

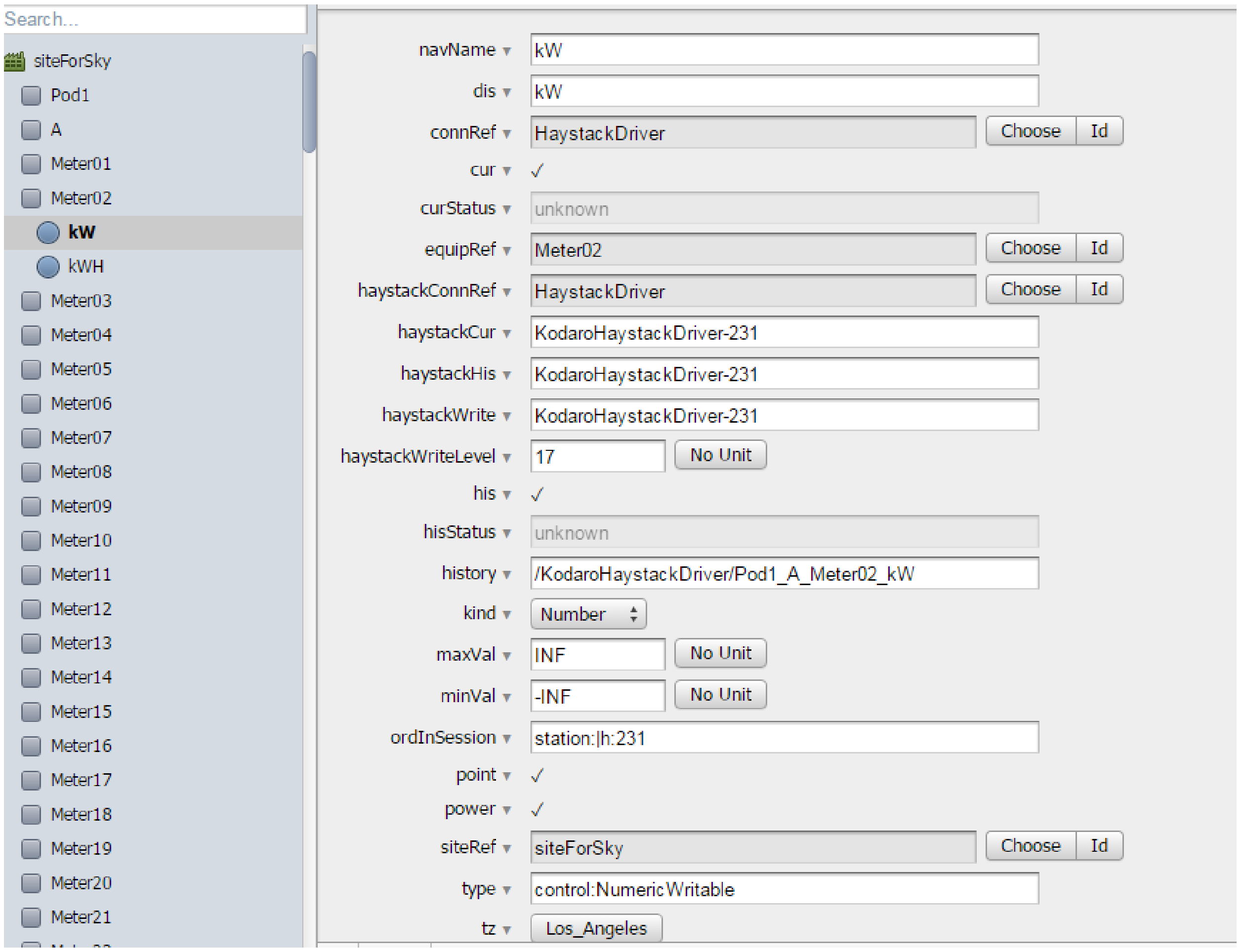
The script is designed to be only run once since it uses the add commit option. This means that if you need to run it again, the existing records will need to be deleted from SkySpark to prevent a duplicate record error. The script can be modified by the user to account for duplicates and update but is not shown in this document how to accomplish this.

c. What did we just do?
   Let's take a look at what the database looks like in the Niagara station.



The data tree here shows the relationship to all the points and folders. What you don't see here is the Niagara Relations. All the points relate to the Meter via an equipRef and every folder relates to it's parent folder with an additional equipRef. Meters is the only folder that relates to the site (siteForSky) with a siteRef.

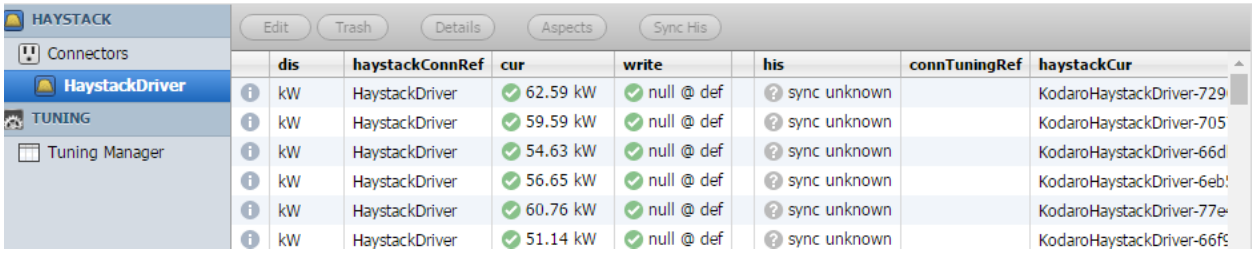
And now in SkySpark Builder App.



navName	kW
dis	kW
connRef	HaystackDriver
cur	✓
curStatus	unknown
equipRef	Meter02
haystackConnRef	HaystackDriver
haystackCur	KodaroHaystackDriver-231
haystackHis	KodaroHaystackDriver-231
haystackWrite	KodaroHaystackDriver-231
haystackWriteLevel	17
his	✓
hisStatus	unknown
history	/KodaroHaystackDriver/Pod1_A_Meter02_kW
kind	Number
maxVal	INF
minVal	-INF
ordInSession	station:|h:231
point	✓
power	✓
siteRef	siteForSky
type	control:NumericWritable
tz	Los_Angeles

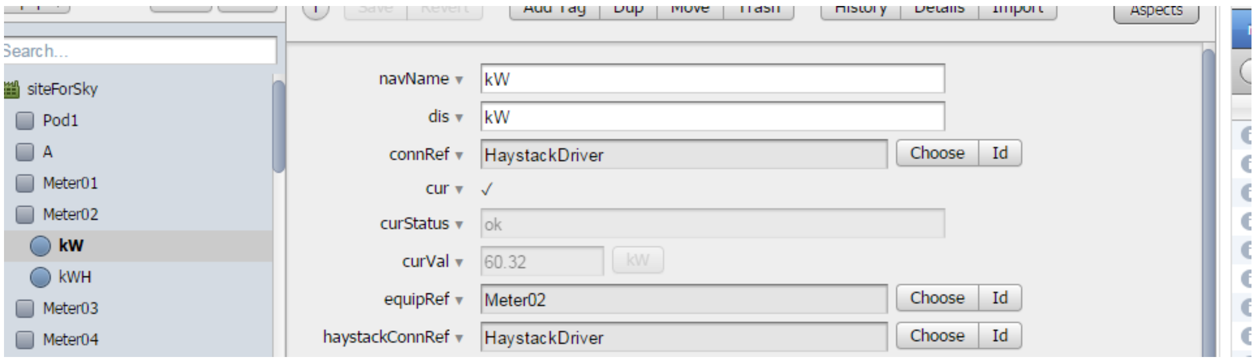
You can see the points are related to their equipment and have also inherited the relationships up the tree all the way to the siteRef even though the point in Niagara doesn't know about this.

a. What is missing? If you look closely at the screen shot above, you'll see the cur value is missing even though it is indicated to have one in Niagara. This is because a watch has not been created and polled against. This is done by setting up the tuning manager in SkySpark or by simply clicking on the Haystack Connector which will attempt to read all the points you just discovered via the import script.



dis	haystackConnRef	cur	write	his	connTuningRef	haystackCur
kW	HaystackDriver	62.59 kW	null @ def	sync unknown		KodaroHaystackDriver-729
kW	HaystackDriver	59.59 kW	null @ def	sync unknown		KodaroHaystackDriver-705
kW	HaystackDriver	54.63 kW	null @ def	sync unknown		KodaroHaystackDriver-66d
kW	HaystackDriver	56.65 kW	null @ def	sync unknown		KodaroHaystackDriver-6eb
kW	HaystackDriver	60.76 kW	null @ def	sync unknown		KodaroHaystackDriver-77e
kW	HaystackDriver	51.14 kW	null @ def	sync unknown		KodaroHaystackDriver-66f

Back in the builder you can see that the cur values have now come in by simply looking at the points in the connector.

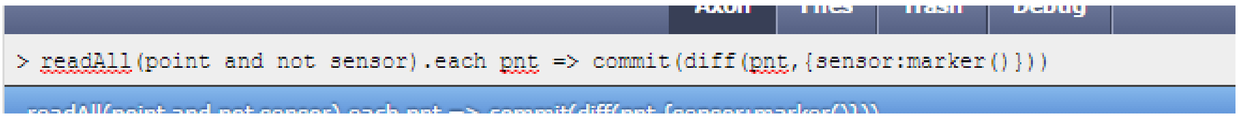


navName	kW
dis	kW
connRef	HaystackDriver
cur	✓
curStatus	ok
curVal	60.32 kW
equipRef	Meter02
haystackConnRef	HaystackDriver

As mentioned before, you will need to setup a SkySpark job to perform this or configure the tuning manager. Both of these topics are outside of the scope of this document.

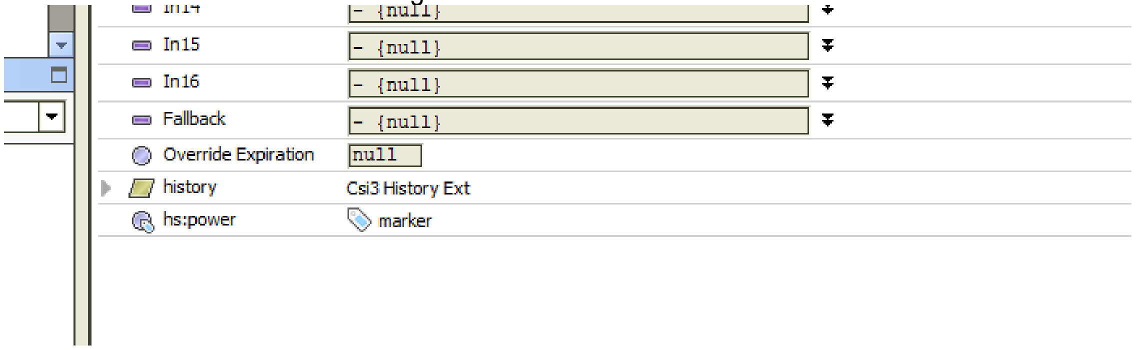
8. How do you make tag changes but keep the two databases in sync?
    i. Niagara To SkySpark – There is no prebuilt script or automatic process to identify tag changes Niagara and update SkySpark. However, the kodaroReadTags script will update SkySpark tags from the Niagara database. The script arguments include a grid of SkySpark points which the user would like to update and the Haystack Connector display name (dis).
    ii. SkySpark to Niagara – The kodaroWriteTags will push tags down to your Niagara database. In the example provided, you will find that no sensor/cmd/sp tag error exists on all the point records. This is because it is a SkySpark dependency and so a Niagara integrator may not remember to throw these on. In our example, we also know that every point we brought in is a sensor so we will now add sensor to every point and then run the kodaroWriteTags option to push these updates back down to Niagara.

        1. We first add a sensor tag to every point that doesn't have one using Axon.



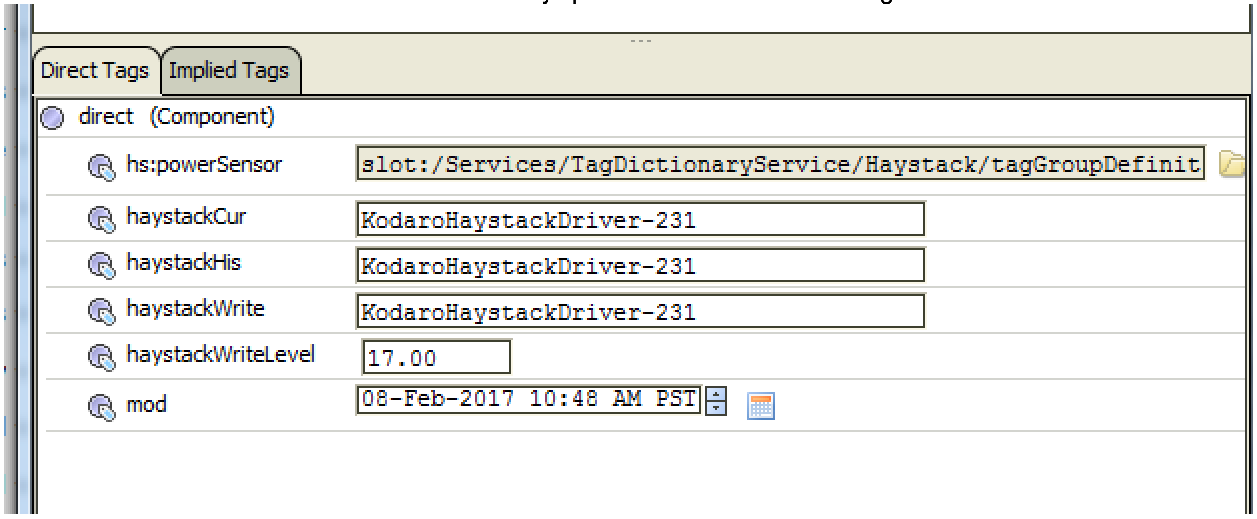
```
> readAll(point and not sensor).each pnt => commit(diff(pnt,{sensor:marker()}))
```

        2. Now we want to make sure Niagara knows about these changes, so we send all the points back down to Niagara. Right now, there is no sensor tag.



In14	- {null}
In15	- {null}
In16	- {null}
Fallback	- {null}
Override Expiration	null
history	Csi3 History Ext
hs:power	marker

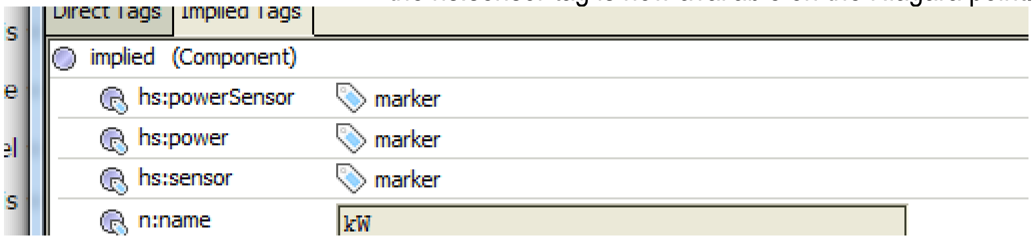
3. We'll run the kodaroWriteTags function for all points in SkySpark. After that, you see more than just sensor coming down but also any tags created in SkySpark that didn't exist in Niagara.



direct (Component)

hs:powerSensor	slot:/Services/TagDictionaryService/Haystack/tagGroupDefinit
haystackCur	KodaroHaystackDriver-231
haystackHis	KodaroHaystackDriver-231
haystackWrite	KodaroHaystackDriver-231
haystackWriteLevel	17.00
mod	08-Feb-2017 10:48 AM PST

But where is the sensor tag we added? You can see Niagara has created a tag group (hs:powerSensor). If we look at implied tags, we can see that the hs:sensor tag is now available on the Niagara point.

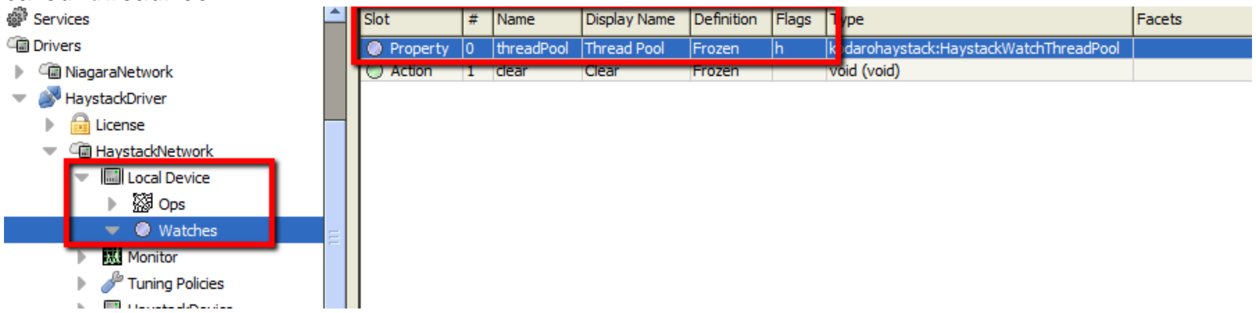


implied (Component)

hs:powerSensor	marker
hs:power	marker
hs:sensor	marker
n:name	kW

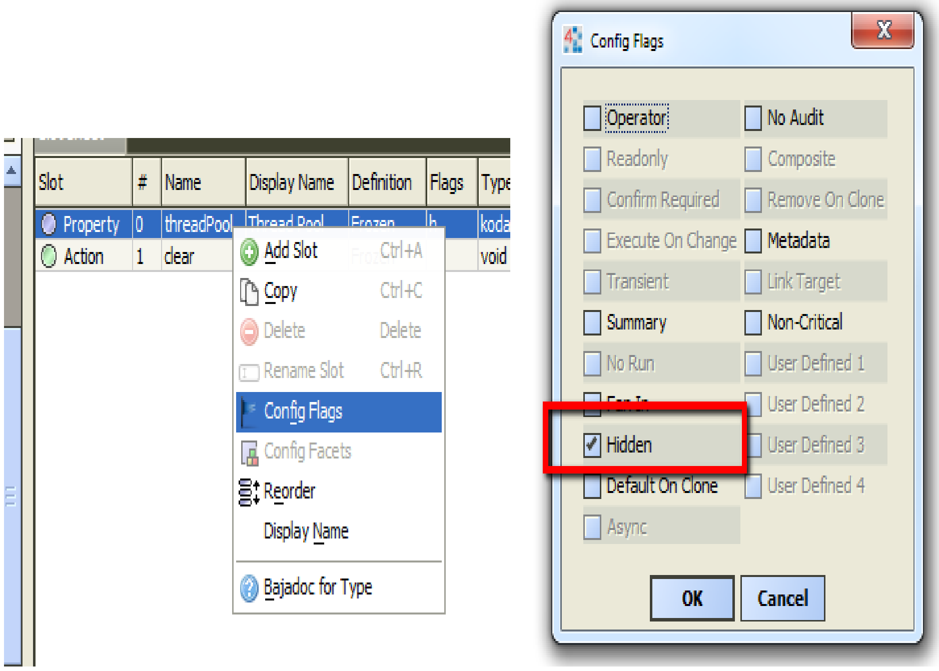
Watch Subscription Tuning

On a large system, there may be a reason you need to allocate more resources to the haystack watch section of this driver. This will only be the case if you are requiring more than 20 active watches simultaneously. By going to the slot sheet of the LocalDevice>Watches, you will see a hidden property called “threadPool”.



Slot	#	Name	Display Name	Definition	Flags	Type	Facets
Property	0	threadPool	Thread Pool	Frozen	h	kodarohaystack:HaystackWatchThreadPool	
Action	1	clear	Clear	Frozen		void (void)	

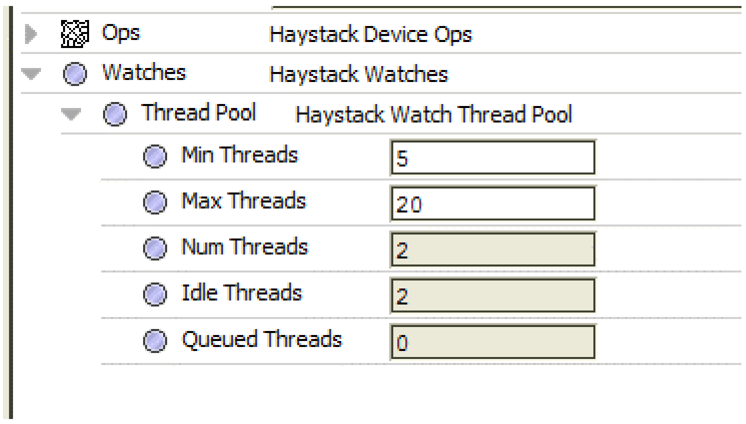
From there you want to right click and select “Config Flags” to bring up a dialogue to unhide this property by unchecking the “Hidden” flag.



Slot	#	Name	Display Name	Definition	Flags	Type
Property	0	threadPool	Thread Pool	Frozen	h	koda
Action	1	clear				void

☐ Operator	☐ No Audit
☐ Readonly	☐ Composite
☐ Confirm Required	☐ Remove On Clone
☐ Execute On Change	☐ Metadata
☐ Transient	☐ Link Target
☐ Summary	☐ Non-Critical
☐ No Run	☐ User Defined 1
☐ Fan In	☐ User Defined 2
☑ Hidden	☐ User Defined 3
☐ Default On Clone	☐ User Defined 4
☐ Async	

Once it has been unhidden, you can now view this on the property sheet of the watches object. Here you can monitor and/or configure the amount of active watches allowed.



Ops	Haystack Device Ops
Watches	Haystack Watches
Thread Pool	Haystack Watch Thread Pool
Min Threads	5
Max Threads	20
Num Threads	2
Idle Threads	2
Queued Threads	0

Every watch will spawn a new thread to subscribe to its points in the Niagara station. This means that the Max Threads property needs to be at least as large as the number of watches you intend to have active at the same time.

Release of 1.0.22, multi-threading for watchSub encoding was introduced for the initial encoding of the components in a watch. This reduces the amount of time it takes for a watchSub to respond. While each watch still requires a thread, additional threads should be added for new watches to encode all their data for fast responses.

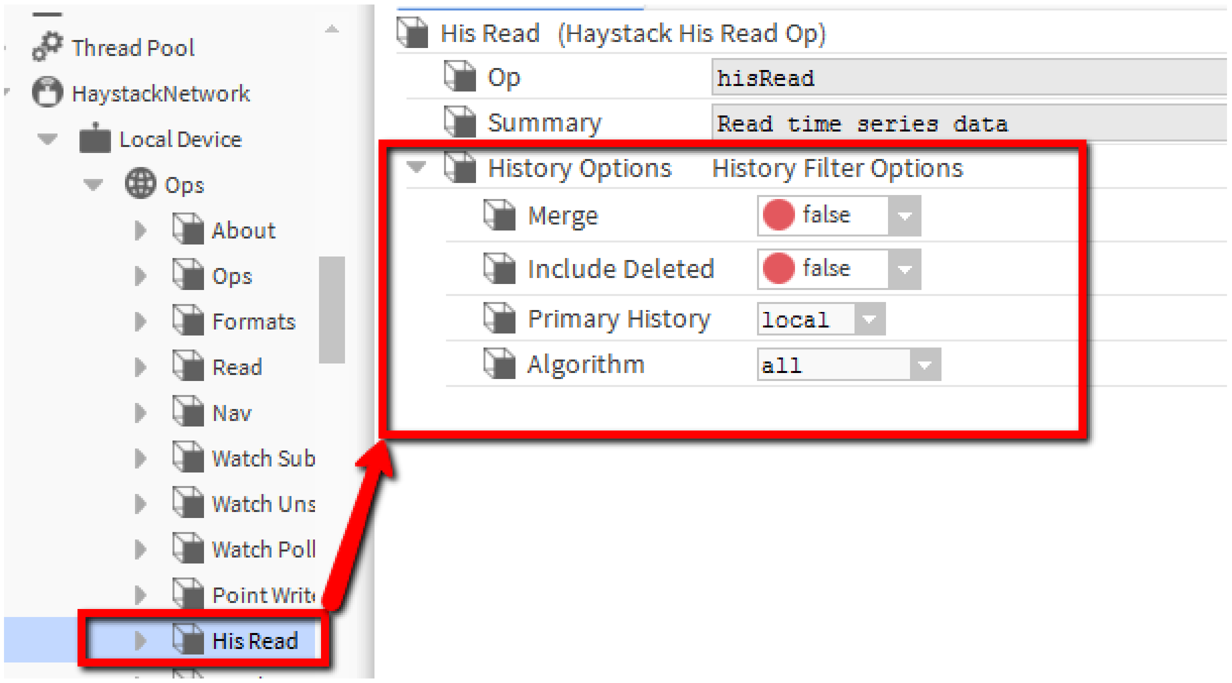
Watch encoding threads do have diminishing returns on their number somewhere around 20. This means any free threads above 20 when used for encoding will not improve performance. In a typical setup, you may have 5 different watches open from a system, which leaves 15 available free threads to be used for the initial watchSub creation. This is useful if a new watch was requested or perhaps one of the watches missed a poll and needed to be reopened, since there are still plenty of available threads to maintain a quick response.

History Resolving and Merging Options

When it comes to resolving histories for either an export or the hisRead web op, there are a few scenarios that can occur within the history database that may require additional handling. This section attempts to explain the most common scenarios with examples.

On both the HisRead Web Op and the Hierarchy Export, there is a component called “History Options”.

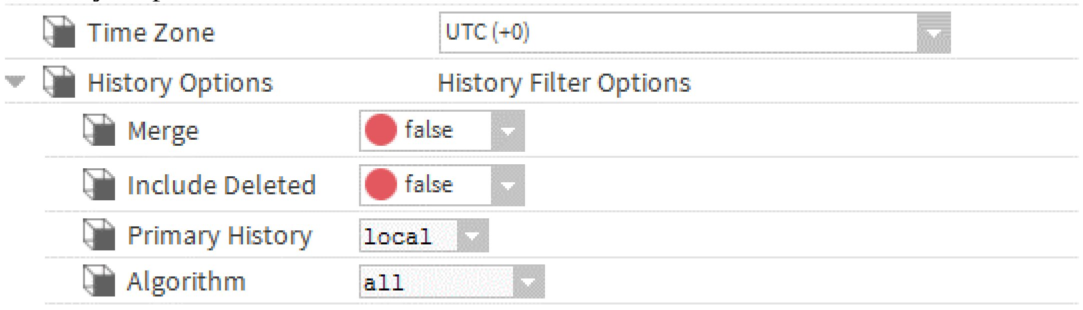
His Read:



His Read (Haystack His Read Op)

Op	hisRead
Summary	Read time series data
History Options	History Filter Options
Merge	false
Include Deleted	false
Primary History	local
Algorithm	all

Hierarchy Export:



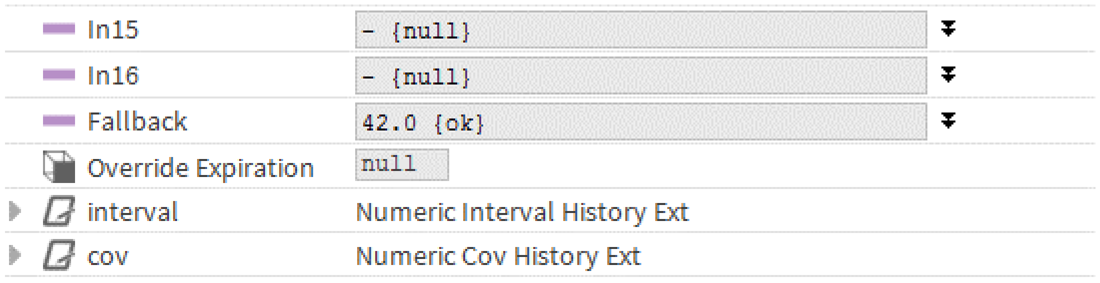
Time Zone	UTC (+0)
History Options	History Filter Options
Merge	false
Include Deleted	false
Primary History	local
Algorithm	all

The default parameters as can be seen by the two screen shots essentially tell the driver to find only the local history that still has an active (non-deleted) history extension in the station and with the most recent data. If no local history meets that criteria, resolve the remote history with the most recent data. No merging of multiple histories should be attempted. For most integrations, these will be the desired settings but there are scenarios where multiple histories exist for the same point and can cause some issues when accessing historical data.

Some Notes About the Examples:
While the following examples do not attempt to cover every single configuration, they are designed to help illustrate how the different settings effect the driver's ability to correctly resolve and merge histories for some of the more common use cases. If there are scenarios that are not clear about your integration and what settings should be used, you can always post questions to https://community.kodaro.com/.

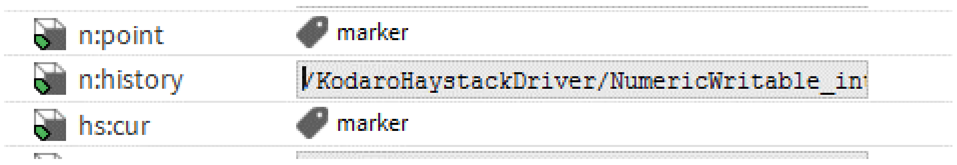
Background:
Before diving into the examples, it is a good idea to address the reason the haystack driver's history resolving algorithm exists. As a Niagara integrator, you may have noticed that there already seems to be a tag created on points called n:history which identifies the history id for that point. Why not just use that and disregard all these history options? The best way to explain that is with a simple scenario.

We will assume there are two history extensions for a single point. One is a COV and the other is an interval of 15 minutes.



In15	- {null}
In16	- {null}
Fallback	42.0 {ok}
Override Expiration	null
interval	Numeric Interval History Ext
cov	Numeric Cov History Ext

Now we know this is a completely valid configuration but now there are two histories for this one point. How does Niagara's n:history tag report this?



n:point	marker
n:history	/KodaroHaystackDriver/NumericWritable_in
hs:cur	marker

/KodaroHaystackDriver/NumericWritable_interval

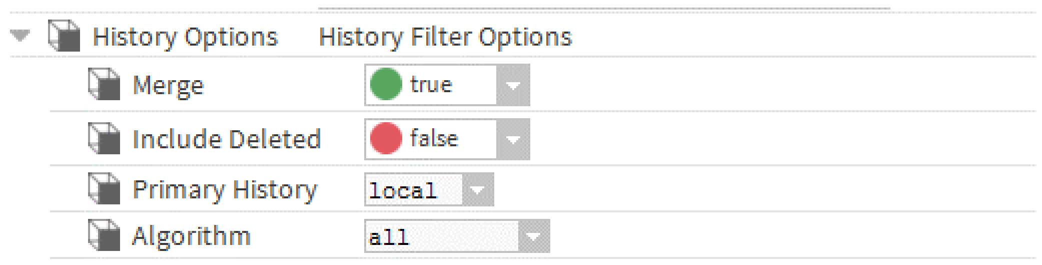
As we can see, it only reports information about the interval. Without a more complex methodology of resolving histories, in this very simple example, the driver has no idea about the COV history and thus never be able to report information about it.

Additional History Resolving Technique:
Any history defined by the n:history tag, or by the name of "history," that could exist in any tag dictionary, will be resolved as a valid history for the point where it is defined and included in the process used to determine which histories to use in the resolving process. This is a necessary technique for remote histories that may have been discovered through two stations. This will also require the use of point joins to maintain the tags across the system which are techniques beyond the scope of this document.

Multiple Local Histories
For this scenario, we will assume there are two history extensions for a single point. One is a COV and the other is an interval of 15 minutes. The default settings as previously described would only get us whichever history last recorded a record, but we are interested in both histories.

Example 1:
First, we will consider that both the histories are active, that means they have history extensions currently in the local station where the driver is running.

To accomplish this, we will turn on the merge setting.



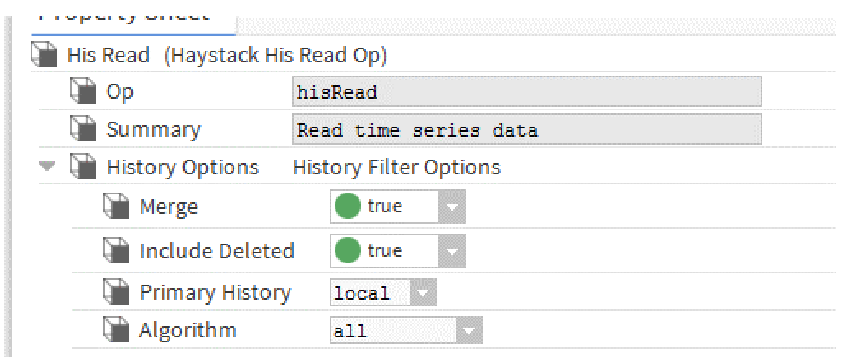
History Options	History Filter Options
Merge	true
Include Deleted	false
Primary History	local
Algorithm	all

That's all we'd have to do for this simple example. Now both histories will be resolved and returned. Since both are active histories and our algorithm says "all", this will return both histories every single time. The priority order of which they will be merged is always the most recent data first. This means that the priority history in this scenario could be different depending on the time it was requested.

i.e. if COV just occurred it would be most recent and if no change has happened for 15 or so minutes, the interval history would be most recent.

Example 2:
Now let's say some time goes by and a decision is made that we only care about COV, we don't need the interval history anymore and so that history extension is deleted. For this scenario, we do nothing. Since we have Include Deleted set to false, this will already start filtering out this history since it is now detected as a deleted history.

Let's go ahead and throw wrenches into the system. Somebody on the other side of this connection deleted all the histories they had imported/exported with the haystack driver. We now only have the COV enabled but we're going to pretend that for the first 3 months, we only had the interval history extension enabled but we still need those first 3 months of data. To resolve this, we just need to go back to the driver and flip the Include Deleted History to true.



His Read (Haystack His Read Op)

Op	hisRead
Summary	Read time series data
History Options	History Filter Options
Merge	● true
Include Deleted	● true
Primary History	local
Algorithm	all

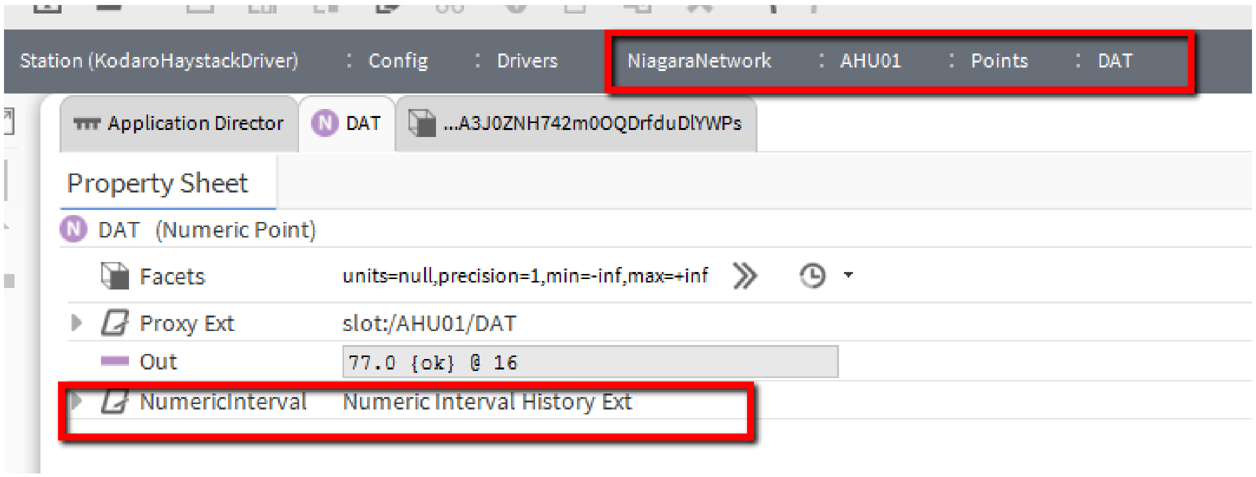
So now we have a station with two histories for one point but one of them has been deleted. That deleted history has the older data we still care about and by telling the driver to *Include Deleted*, that interval history will be merged with the COV one. But which one will take priority? Because this process always uses the history with the most recent historical data, the COV will be priority for all records that have time stamp conflicts as soon as the interval catches up to where the COV history begins.

Example 3:
Well that was a pain you might be thinking, somebody screwed up the system and I had to go back into Niagara just to turn one Boolean flag, so they could fix it? I don't want to do that again. Good news! You don't have to. If you know deleted histories have valid data in them, you can always include them. As soon as somebody requests historical data that covers the range when they were active, then you'll see that data come back and if nobody ever does, then that history will just remain in Niagara until somebody decides it can be deleted out of the history database.

Local and Remote Histories

In the following scenarios, we will consider a point that has both a local history and a remote history. A remote history is any history that was created by a station that is not the same station where the haystack driver is installed, i.e. a Niagara Network history import.

For the following examples, we will have two stations running. A JACE station called AHU01 and a supervisor station called KodaroHaystackDriver. The histories from AHU01 will be imported into the supervisor which makes them available for the haystack driver to find them when resolving remote histories.

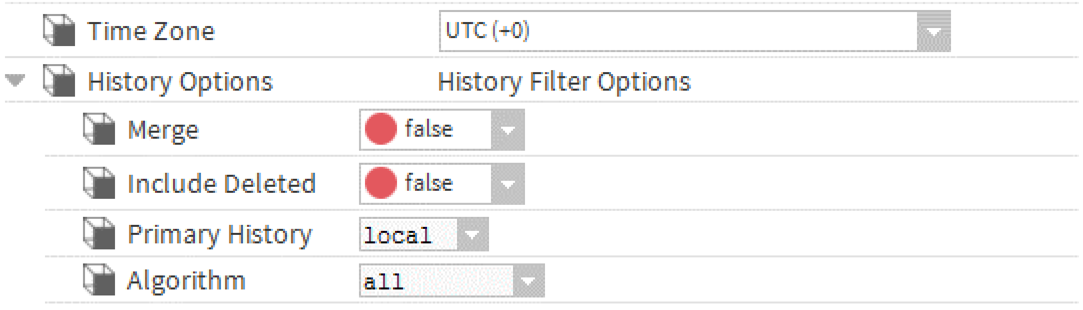

Station (KodaroHaystackDriver) : Config : Drivers
NiagaraNetwork : AHU01 : Points : DAT
Application Director
DAT
...A3J0ZNH742m0OQDrfduDlYWPs
Property Sheet

DAT (Numeric Point)

Facets	units=null,precision=1,min=-inf,max=+inf
Proxy Ext	slot:/AHU01/DAT
Out	77.0 {ok} @ 16
NumericInterval	Numeric Interval History Ext

Example1:

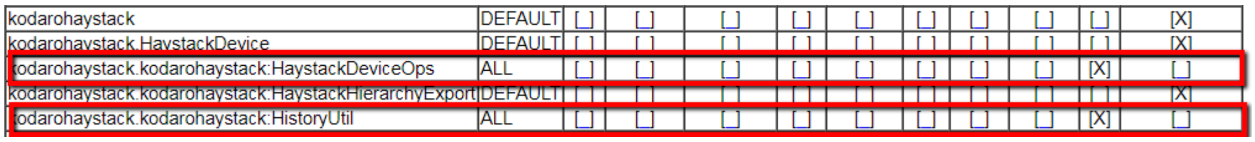
Using the default history options, this will resolve the local history only. Because merge is not enabled, it will only care about the primary history, which is defined as the local by default.



Time Zone	UTC (+0)
History Options	History Filter Options
Merge	false
Include Deleted	false
Primary History	local
Algorithm	all

If we were to turn on the logging of the driver:

kodaroHaystack.kodarohaystack:HaystackDriver Ops = All
kodaroHaystack.kodarohaystack::HistoryUtil = All



kodarohaystack	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
kodarohaystack.HaystackDevice	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
kodarohaystack.kodarohaystack:HaystackDeviceOps	ALL	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]	[ ]
kodarohaystack.kodarohaystack:HaystackHierarchyExport	DEFAULT	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]
kodarohaystack.kodarohaystack:HistoryUtil	ALL	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[X]	[ ]

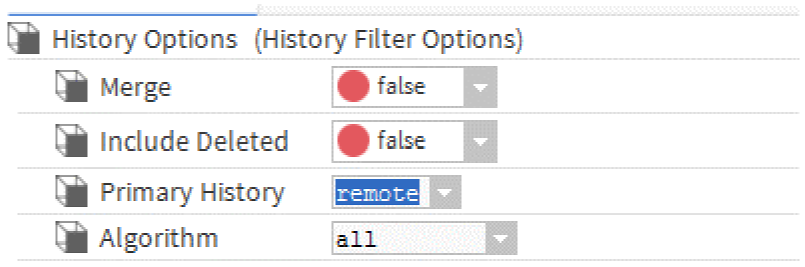
And sending a hisREad request to the station, we can see the following output:

```
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Converting Web Op to Async...
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Web Op to Async Conversion complete
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Web Op Enqueued
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Async Op Executing...
FINEST [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Uri: /haystack/hisRead
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Async Op Name: hisRead
FINEST [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Resolving Histories for:
slot:/Drivers/NiagaraNetwork/AHU01/points/DAT
INFO [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Histories Resolved: 2
FINER [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Querying records from 01-Jan-19 12:00 AM PST
to 20-Mar-19 12:00 AM PDT
FINEST [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Converting Niagara Histories for
Haystack.../KodaroHaystackDriver/DAT
FINER [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] 0/0 Successfully Converted:
/KodaroHaystackDriver/DAT
FINE [09:23:01 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Async Op Complete
```

By inspecting this output, you can see that the histories resolved count is 2 but immediately following that statement is only one query of a history, /KodaroHaystackDriver/DAT. Since history ID starts with “KodaroHaystackDriver”, this tells us that the history is local as it has the same history device name as the station name.

So now we can see that the driver knows there are two histories for this point but because we said use the primary as local without merging allowed, we don’t see the remote history in the results.

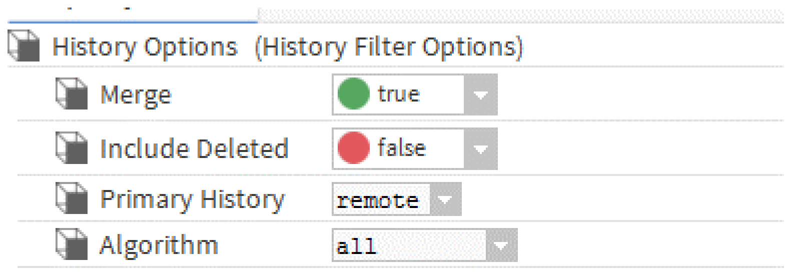
Example 2:
Not turning merge on yet, we will now switch the primary to remote and inspect the response to make sure that we are now using the remote history instead of the local.


History Options (History Filter Options)
Merge: false
Include Deleted: false
Primary History: remote
Algorithm: all

```
FINEST [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Resolving Histories for:
slot:/Drivers/NiagaraNetwork/AHU01/points/DAT
INFO [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Histories Resolved: 2
FINER [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Querying records from 01-Jan-19 12:00 AM PST
to 20-Mar-19 12:00 AM PDT
FINEST [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Converting Niagara Histories for
Haystack.../AHU01/DAT
FINER [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] 4/4 Successfully Converted: /AHU01/DAT
FINE [09:30:02 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Async Op Complete
```

We can now see that there are still 2 histories being resolved but he history read is /AHU01/DAT which tells us this is the history that came from the station AHU01, the remote history.

Example 3:
Without merge, we've seen how to select only the local or the remote history, so now we will enable history merging and again inspect the output of the logging to confirm we are now resolving both the histories.



History Options (History Filter Options)

Merge	true
Include Deleted	false
Primary History	remote
Algorithm	all

```
FINEST [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Resolving Histories for:
slot:/Drivers/NiagaraNetwork/AHU01/points/DAT
INFO [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Histories Resolved: 2
FINER [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Querying records from 01-Jan-19 12:00 AM PST
to 20-Mar-19 12:00 AM PDT
FINEST [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Converting Niagara Histories for
Haystack.../AHU01/DAT
FINER [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] 4/4 Successfully Converted: /AHU01/DAT
FINER [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Querying records from 01-Jan-19 12:00 AM PST
to 20-Mar-19 12:00 AM PDT
FINEST [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] Converting Niagara Histories for
Haystack.../KodaroHaystackDriver/DAT
FINER [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HistoryUtil] 1/1 Successfully Converted:
/KodaroHaystackDriver/DAT
FINE [09:38:04 19-Mar-19 PDT][kodarohaystack.kodarohaystack:HaystackDeviceOps] Async Op Complete
```

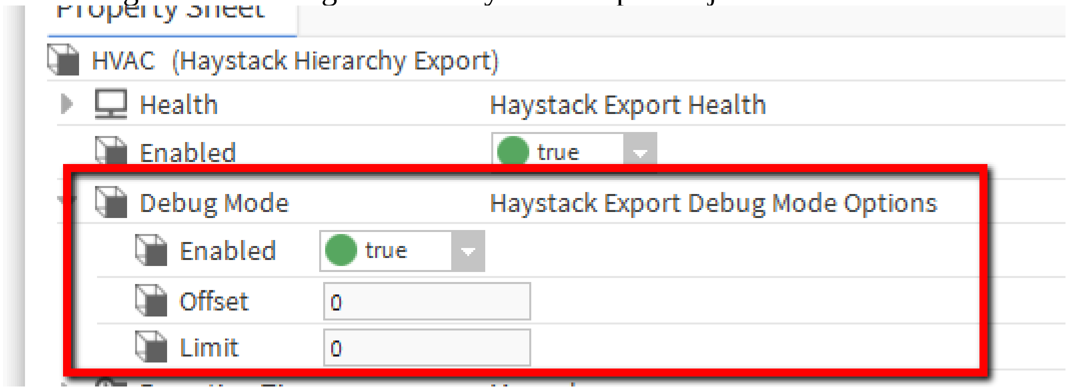
This time we see that again two histories are resolved but we are now querying them both. The order in which they are queried is also their priority order. We can see that the first one queried is the remote history /AHU01/DAT since our settings tell the driver to treat the remote one as the primary source. This could easily be switched to prefer local by just changing the primary history selection and the output would be identical except the order of which they were queried would be reversed.

At this point, you should have a good understanding of all the properties and how they change what the driver does when it comes to resolving historical data. Continuing this example path by flipping a few more switches with the algorithm, includeDeleted, etc would be showing the same results as the examples found in the Multiple Local Histories section which should be used if your use case requires the combination of these two types of database configurations.

Export Debug Mode

During a hierarchy export, every component is encoded into either an update grid or an add grid depending on whether the component needs to be updated or added to the exported database. A single bad point of data can cause the entire operation to fail and the reporting of these failures does not generally provide enough information to determine which entity caused the failure. Debug mode should only be used for determining these types of errors as it will fully abandon any network efficiency in favor of a point by point update/add operation.

The debug mode is configured directly on an export object.



HVAC (Haystack Hierarchy Export)

Health	Haystack Export Health
Enabled	true
Debug Mode	Haystack Export Debug Mode Options
Enabled	true
Offset	0
Limit	0

Enabled: Setting this to true enables debug mode. Enabling debug mode does not mean debug mode will automatically be used. Debug mode is only entered if a full add and/or update grid commit fails for any reason. Since debug mode is only necessary to determine what data in the grid caused the failure, the default is always to try the efficient large data commit before trying the single entity commit option that the debug mode utilizes.

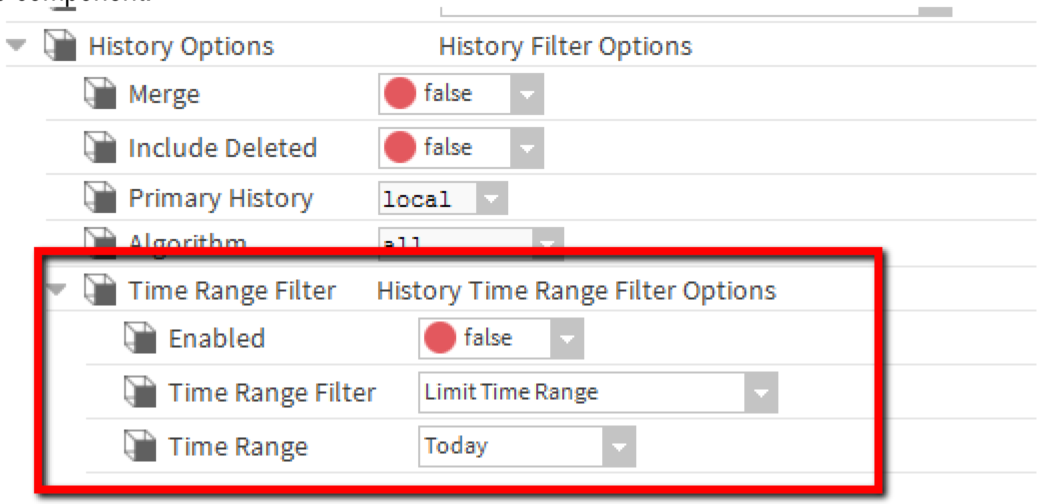
Offset: This is the row in the grid to start exporting at. If this value is greater than the number of rows, it will default to the last row in the grid.

Limit: Determines how many rows to export from the grid. If this value is zero, this means export all rows.

History Time Range Filter

When a history is exported, sometimes the data range may contain too much data for the request and timeout as a result. This is generally only a problem for new integrations where there is months to years' worth of data already stored. By allowing a time range filter override, you can restrict packet sizes. This is mostly helpful on a history export as there is no other way to restrict this range, but it is also available on the hisRead web op but is not recommended to be used as that can be controlled at a much higher granularity from the request itself.

The Time Range Filter is part of the History Options section that can be found on an export or hisRead web op component.



History Options	History Filter Options
Merge	false
Include Deleted	false
Primary History	local
Algorithm	all
Time Range Filter	History Time Range Filter Options
Enabled	false
Time Range Filter	Limit Time Range
Time Range	Today

Enabled: By setting this to true, the time range for history exports/reads will follow the parameters defined on this object. When false, this uses the entire range for exports and the range provided for hisRead's.

Time Range Filter: This determines how to apply the Time Range option

- Limit Time Range – This restricts to the time range only inclusive. Anything outside of this time range is not included.
- Limit Time Range Start Only – This will restrict from the start of the time range and allow anything after this time range.
- Limit Time Range End Only – This will restrict from anything before the end of the time of the range.

Time Range: Selectable time range to use to restrict history access.

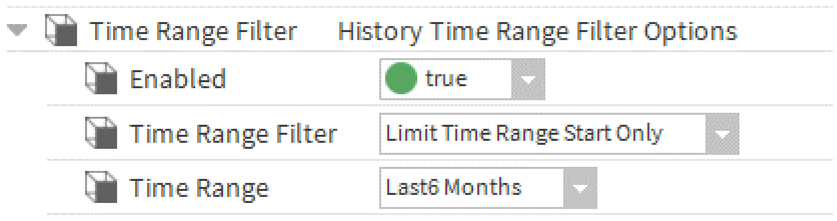
Examples

Use the following assumptions for the provided examples:

1. The date is August 28, 2019 at 8:00AM PST
2. A Niagara Station has been running for two years collecting historical data

Example 1: Attempting to export the entire two years' worth of data, the data packets for 2 years of data is just too big and continues to timeout. It's concluded that we only really care about analyzing the previous 6 months of data

To achieve this goal, we would set the time range filter as follows:



Time Range Filter	History Time Range Filter Options
Enabled	true
Time Range Filter	Limit Time Range Start Only
Time Range	Last6 Months

Enable it so that the time range filter is used. Set the Time Range Filter to Limit Time Range Start only. This will restrict how far back the time range can go but not limit new data past it, which is what we want. Setting the Time Range to Last 6 Months restricts the data to only the most recent 6 months of data. The effective time range on this configuration will be:

2019-02-01T00:00-07:00 – null

This can be seen with logging enabled and will show up as:

```
FINER [09:48:33 28-Aug-19 PDT][kodarohaystack.kodarohaystack:HaystackHierarchyExport] Querying records from 31-Jan-19 11:00 PM PST to null
```

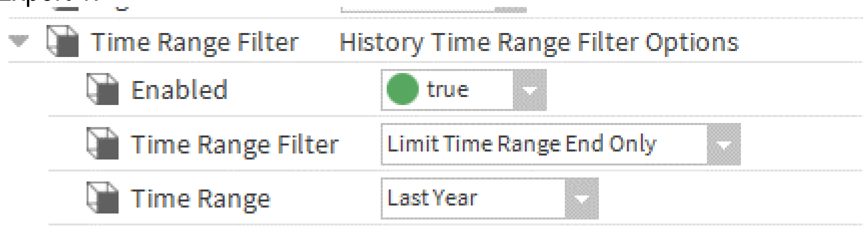
Why is logging different than mentioned above? This is because Niagara normalizes BAbsTime objects to localized time zone based on it's own format. So where it shows up as PST, even though the time is currently PDT, the actual time stamp is identical because 31-Jan-19 11:00 PM PST (-08:00) does in fact equal 2019-02-01T00:00-07:00.

What does the end time of null mean in this context? If we look at the documentation for querying the Niagara history database:

The result is inclusive of the end points. If either endpoint is `null`, then the interval will be considered open on that end.

Example 2: Attempting to export the entire two years' worth of data, the data packets for 2 years of data is just too big and continues to timeout. In example 1, only the last 6 months of data were used but in this example all the data is needed. To do this, the packets need to be broken up for 2 years.

Export 1:



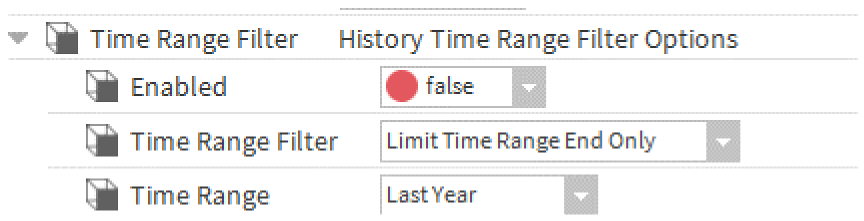
Time Range Filter	History Time Range Filter Options
Enabled	● true
Time Range Filter	Limit Time Range End Only
Time Range	Last Year

This will query over the time range:

null - 2018-12-31T23:59:59.999999999-08:00

This will restrict the export to all the histories prior to 2019. Assuming this is successful, we will now export the remaining data.

Export 2: Since the export by default determines the time range based on most recent exported data, you would only have to disable the time range filter.



Time Range Filter	History Time Range Filter Options
Enabled	● false
Time Range Filter	Limit Time Range End Only
Time Range	Last Year

This will cause each history to dynamically search its oldest data not already exported up until it's most current data. This change also allows for all future data to stay updated even if there is a long period of connection down time without having to re-adjust the time range to account for such a scenario.

APPENDIX B: Additional Ops

This section will discuss the web ops implemented on this driver that are not defined by the Haystack specification that have been added to enhance the driver's ability to integrate within the Niagara framework.

Using the Query Op

The non-standard haystack "query" op is designed to expand on the standard haystack read op but targeted to utilize the built-in query languages of Niagara (NEQL and BQL).

The query op follows the same format as the read op (http://project-haystack.org/doc/Ops#read) with a few additional parameters and modifications.

1. The filter parameter as defined by the read op in haystack is no longer haystack compliant but will need to conform to a proper BQL or NEQL query.
2. Additional Parameters to this query are "base" and "root".
   a. base – The base query language, this is either neql or bql.
   b. root – The root ord to originate the query from.

Example1: Considering the supporting SkySpark pod's documentation, you will find most of the examples using the read op. This is because it is designed to work as an example for any station. Here we will look at a more targeted query example knowing something about the station.

Let's assume our station has a very basic structure of:

Config

-HVAC

-Power

And I want SkySpark to run analytics on power only. This means that I only care about querying data out of the Power folder. Now I can try to do this with proper tag filtering and reading the whole database with the read op but what If I don't know what the tagging is yet and if it's even correct? To solve this, you would use the query op with a root ord filter. So, what this looks like is:

read(haystackConn).haystackCall("query",{root:"station:|slot:/Power",base:"neql",filter:"hs:point"})

Breaking down the axon call.

1. read(haystackConn) – This gives me a HaystackConnector to pass into the haystackCall function (https://skyfoundry.com/doc/ext-haystack/funcs#haystackCall)
2. The haystackCall then asks for the op name, in this case it's "query".

3. Lastly, it wants a grid of information which is the parameter list
    a. root – Niagara ord and/or slotPath to where to start the station, in this case it's station:|slot:/Power
    b. base – This tells the query what the query language is, in this case it is NEQL.
    c. filter – This is the formatted NEQL or BQL query to use.

The order of the arguments is not important to function properly.

To better show the difference between this and read, this could have been written as:

read(haystackConn).haystackCall("read",{filter:"hs:point and hs:power"})

The difference here being it always assumes NEQL and there is no site level ord filter. This also assumes every point you want has been properly tagged with hs:power.

Using the Commit Op

The non-standard haystack "commit" op is designed to expand on the standard haystack ops to enable integration with the Niagara Component space which is not strictly haystack compliant in nature. Due to the complexities of working with the Niagara component space, it is recommended that you have some domain knowledge before proceeding with this section.

Usage Overview:
Every commit request is performed by posting a grid with the meta defining the operation. Depending on the ability to handle the request, you will receive back a grid defining each operation in the grid as a success or information about the error that occurred when attempting to handle the requested operation.

The grid provided can either be a single row with each column containing a new object or multiple rows with an ID for every row. The ID can be unique to each row or each row can refer to the same ID and a different object. This flexibility allows the user options for whichever may be easiest to construct for their application.

Zinc Example (meta in bold):

```
ver:"3.0" commit:"add"
id,addPoint
@KodaroHaystackDriver-36bfc,{type:"StatusNumeric" module:"baja" args:[{class:"double"
args:"42"},{type:"Status" module:"baja" args:{class:"int" args:"0"}}]}
```

JSON Example (meta inbold);

```
{
  "meta": {
    "ver": "3",
    "commit": "add"
  },
"cols":[{"name":"id"},{"name":"addPoint"},{"name":"suiteUser"}],"rows":[{"id":"r:Kodar
oHaystackDriver-
36bfc","addPoint":{"type":"StatusNumeric","module":"baja","args":[{"class":"double","a
rgs":"42"},{"type":"Status","module":"baja","args":{"class":"int","args":"0"}}]}}]}
```

Both examples above define the "commit" action as add which tells the Haystack Driver to pass the grid to the add function. We will now break down the examples above in the context of an add.

Add Example:

Request:
Using the above grids shown in the usage overview, we will break down what each part means and discuss how these values were determined.

Each row defines an id, which correlates to the Niagara handle of the object you wish to add something to. The id is in the format:

Zinc - @<stationName>-<Niagara Handle>
JSON – r:<stationName>-<Niagara Handle>

```
id,addPoint
@KodaroHaystackDriver-36bfc,{type:"StatusNumeric" module:"baja" args:[{class:"double"
args:"42"},{type:"Status" module:"baja" args:{class:"int" args:"0"}}]}
```

Each column other than the id is expected to the name of the property/object to add to the parent id defined by id.

```
id,addPoint
@KodaroHaystackDriver-36bfc,{type:"StatusNumeric" module:"baja" args:[{class:"double"
args:"42"},{type:"Status" module:"baja" args:{class:"int" args:"0"}}]}
```

For this example, we are telling the haystack driver to add a new property/object to the BComponent defined by the Niagara handle h:36bfc in the Station KodaroHaystackDriver and name it addPoint".

If you inspect the grid further, the content of the addPoint column has an encoded packet of a BStatusNumeric point with the value of 42 and a BStatus of ok. To understand how this encoding was determined, see the section "Encoding Niagara Component Space in Haystack".

Success Response:
If the request is successful, you will receive a message indicating which objects were successful.

Zinc:
ver:"3.0"
id,addPoint
@KodaroHaystackDriver-36bfc,"success"

JSON:
{"meta":{"ver":"3"},"cols":[{"name":"id"},{"name":"addPoint"}],"rows":[{"id":"r:KodaroHaystackDriver-36bfc","addPoint":"success"}]}

Each Row and Column that was successful will return the string message of "success".

Failure Response:
Each Row and Column that was a failure will return a Dict with two values:

- errMsg – Short message of the error
- errTrace – Full java stack trace of the error

Zinc:

```
id,addPoint
@KodaroHaystackDriver-36bfc,{errMsg:"DuplicateSlotException:Slot \"addPoint\" already exists."
errTrace:"javax.baja.sys.DuplicateSlotException: Slot \"addPoint\" already exists.\r\n\tat
com.tridium.sys.schema.ComponentSlotMap.add(ComponentSlotMap.java:2489)\r\n\tat
javax.baja.sys.BComponent.add(BComponent.java:890)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.add(BHaystackCommitOp.java:116)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.add(BHaystackCommitOp.java:149)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:184)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:460)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:24)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:5
74)\r\n\tat javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n"}
```

JSON:

{"meta":{"ver":"3"},"cols":[{"name":"id"},{"name":"addPoint"}],"rows":[{"id":"r:KodaroHaystackDriver-36bfc","addPoint":{"errMsg":"s:DuplicateSlotException:Slot \"addPoint\" already exists.","errTrace":"s:javax.baja.sys.DuplicateSlotException: Slot \"addPoint\" already exists.\r\n\tat com.tridium.sys.schema.ComponentSlotMap.add(ComponentSlotMap.java:2489)\r\n\tat javax.baja.sys.BComponent.add(BComponent.java:890)\r\n\tat com.kodaro.haystack.device.ops.BHaystackCommitOp.add(BHaystackCommitOp.java:116)\r\n\tat com.kodaro.haystack.device.ops.BHaystackCommitOp.add(BHaystackCommitOp.java:149)\r\n\tat com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:184)\r\n\tat com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:460)\r\n\tat com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:24)\r\n\tat com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:574)\r\n\tat javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n"}}]}

Update Example:

The update command is used to set any property on any component in the Niagara station.

Example Request 1:

Each row defines an id, which correlates to the Niagara handle of the object you wish to modify a property The id is in the format:

Zinc - @<stationName>-<Niagara Handle>
JSON – r:<stationName>-<Niagara Handle>

Each column other than the id is expected to the name of the property/object to be modified on the component defined by the id.

This request will attempt to update the fallback slot of a BNumericWritable with the value of "50" with a status of "ok". To understand how this encoding was determined, see the section "Encoding Niagara Component Space in Haystack".

```
ver:"3.0" commit:"update"
id,fallback
@KodaroHaystackDriver-36bfc,[{class:"double" args:"50"},{type:"Status" module:"baja" args:{class:"int"
args:"0"}}]
```

Success Response 1:
If the request is successful, you will receive a message indicating which objects were successful.

Zinc:
```
ver:"3.0"
id,fallback
@KodaroHaystackDriver-36bfc,"success"
```

JSON:
```
{"meta":{"ver":"3"},"cols":[{"name":"id"},{"name":"fallback"}],"rows":[{"id":"r:KodaroHaystackDriver-
36bfc","fallback":"success"}]}
```

Each Row and Column that was successful will return the string message of "success".

Failure Response:
Let's assume in the last response we misspelled the slot name as "falback" and then ran the same command.

Each Row and Column that was a failure will return a Dict with two values:

- errMsg – Short message of the error
- errTrace – Full java stack trace of the error

Zinc:
```
ver:"3.0"
falback,id
{errMsg:"Exception:No property named 'falback' could be resolved" errTrace:"java.lang.Exception: No
property named 'falback' could be resolved\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.update(BHaystackCommitOp.java:249)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.update(BHaystackCommitOp.java:226)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:188)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:460)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:24)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:5
74)\r\n\tat
javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n"},@KodaroHaystack
Driver-36bfc
```

JSON:
```
{"meta":{"ver":"3"},"cols":[{"name":"falback"},{"name":"id"}],"rows":[{"falback":{"errMsg":"s:Exception:No
property named 'falback' could be resolved","errTrace":"s:java.lang.Exception: No property named 'falback'
could be resolved\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.update(BHaystackCommitOp.java:249)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.update(BHaystackCommitOp.java:226)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:188)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:460)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:24)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:5
74)\r\n\tat
javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n"},"id":"r:KodaroHayst
ackDriver-36bfc"}]}
```

Remove Example:

The remove command is used to remove any component from its parent in the Niagara station.

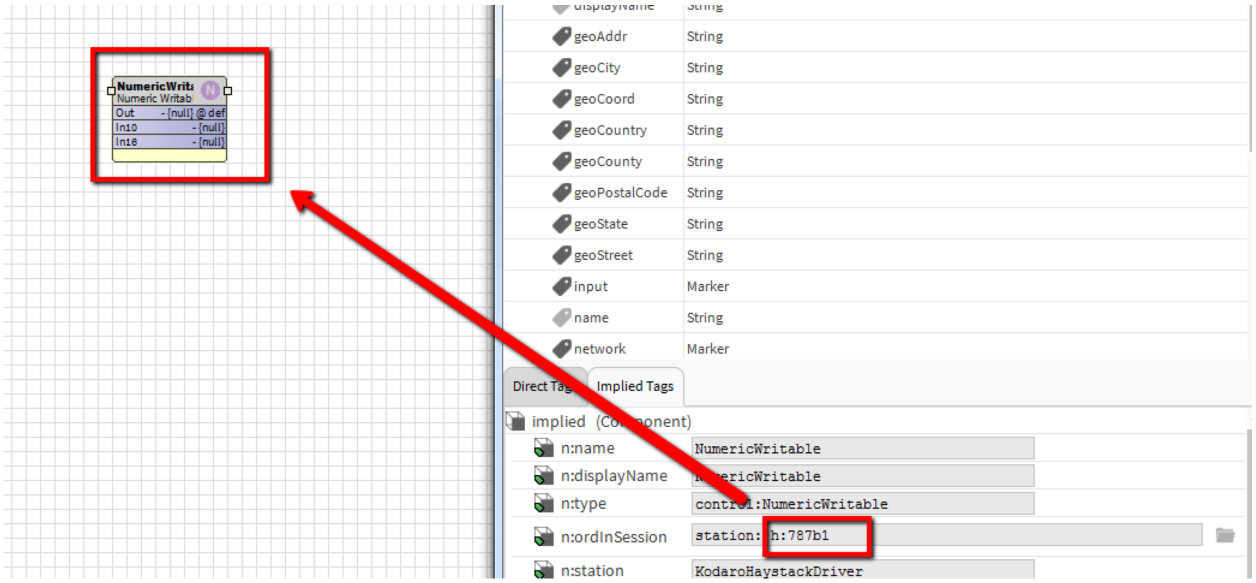
Example Request 1:

Each row defines an id, which correlates to the Niagara handle of the object you wish to remove from the station: The id is in the format:

Zinc - @<stationName>-<Niagara Handle>
JSON – r:<stationName>-<Niagara Handle>

The following example will tell the station to remove the Component defined by the id (handle) of h:787b1.

```
ver:"3.0" commit:"remove"
id
@KodaroHaystackDriver-787b1
```



displayName	String
geoAddr	String
geoCity	String
geoCoord	String
geoCountry	String
geoCounty	String
geoPostalCode	String
geoState	String
geoStreet	String
input	Marker
name	String
network	Marker

implied (Component)

n:name	NumericWritable
n:displayName	NumericWritable
n:type	control:NumericWritable
n:ordInSession	station:|h:787b1
n:station	KodaroHaystackDriver

Example Response 1:

If the request is successful, you will receive a message indicating which objects were successful. Each row will echo the id and a marker tag for success if it was successful.

```
ver:"3.0"
id,success
@KodaroHaystackDriver-787b1,M
```

Failure Response:
If we were to resubmit the same request after it had already been removed, we would see the following error response.

Each Row and Column that was a failure will return a Dict with two values:

- errMsg – Short message of the error
- errTrace – Full java stack trace of the error

Zinc:

```
ver:"3.0"
errMsg,errTrace,id
"UnresolvedException:787b1","javax.baja.naming.UnresolvedException: 787b1\r\n\tat
javax.baja.space.BComponentSpace.resolveByHandle(BComponentSpace.java:575)\r\n\tat
javax.baja.space.BHandleScheme.resolve(BHandleScheme.java:73)\r\n\tat
javax.baja.naming.BOrdScheme.resolve(BOrdScheme.java:107)\r\n\tat javax.baja.naming.BOrd.resolve(BOrd.java:274)\r\n\tat
javax.baja.naming.BOrd.resolve(BOrd.java:250)\r\n\tat javax.baja.naming.BOrd.resolve(BOrd.java:230)\r\n\tat
javax.baja.naming.BOrd.get(BOrd.java:206)\r\n\tat com.kodaro.haystack.util.ComponentUtil.resolveComponent(ComponentUtil.java:159)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.remove(BHaystackCommitOp.java:226)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:186)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:513)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:27)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:652)\r\n\tat
javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n",@KodaroHaystackDriver-787b1
```

JSON:

```
{"meta":{"ver":"3.0"},"cols":[{"name":"errMsg"},{"name":"errTrace"},{"name":"id"}],"rows":[{"errMsg":"s:UnresolvedException:787b1","errTrace":"s:
javax.baja.naming.UnresolvedException:787b1\r\n\tat
javax.baja.space.BComponentSpace.resolveByHandle(BComponentSpace.java:575)\r\n\tat
javax.baja.space.BHandleScheme.resolve(BHandleScheme.java:73)\r\n\tat
javax.baja.naming.BOrdScheme.resolve(BOrdScheme.java:107)\r\n\tat javax.baja.naming.BOrd.resolve(BOrd.java:274)\r\n\tat
javax.baja.naming.BOrd.resolve(BOrd.java:250)\r\n\tat javax.baja.naming.BOrd.resolve(BOrd.java:230)\r\n\tat
javax.baja.naming.BOrd.get(BOrd.java:206)\r\n\tat com.kodaro.haystack.util.ComponentUtil.resolveComponent(ComponentUtil.java:159)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.remove(BHaystackCommitOp.java:227)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.commit(BHaystackCommitOp.java:187)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackCommitOp.op(BHaystackCommitOp.java:87)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.asyncOp(BHaystackDeviceOps.java:513)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps.access$000(BHaystackDeviceOps.java:27)\r\n\tat
com.kodaro.haystack.device.ops.BHaystackDeviceOps$HaystackWebOp.run(BHaystackDeviceOps.java:652)\r\n\tat
javax.baja.util.ThreadPoolWorker$WorkerThread.run(ThreadPoolWorker.java:279)\r\n","id":"r:KodaroHaystackDriver-78f79"}]}
```

Using the Invoke Action Op

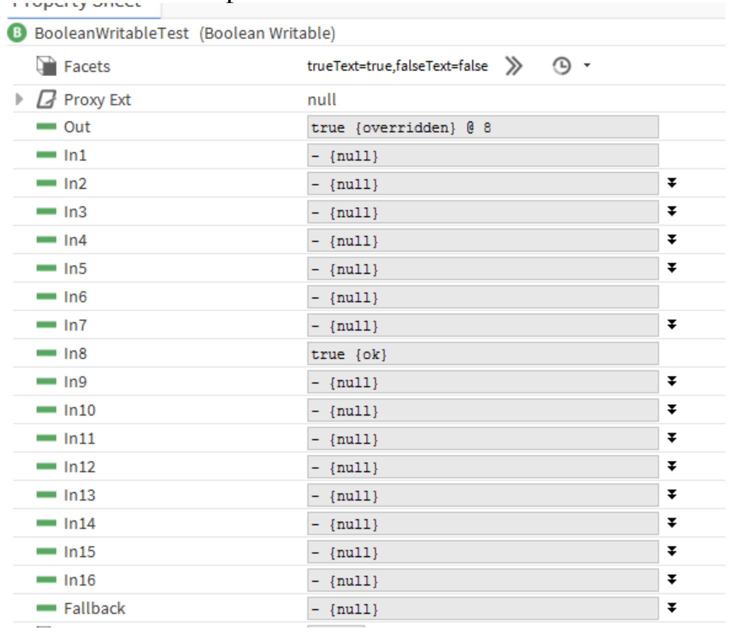
The Invoke Action Op follows the specification defined at https://project-haystack.org/doc/Ops#invokeAction. Since this leaves a lot to the implementer to decide on, this sections purpose is to discuss the specifics of this action as it pertains to the Niagara implementation.

This section will also document the non-Niagara component level actions that have been added to enhance the Haystack Driver for full integration beyond the original haystack scope. These actions include “syncTags” and “ackAlarm”.

1. **Niagara Component Invoke Action -**
   For Niagara components that have actions, the invoke process is very straightforward. To invoke a Niagara alarm, you need to provide the BValue as the argument if it requires any argument to invoke the action that can’t be null. This is provided via the first and only row in the grid provide for an invoke action. The contents of this need to be able to be resolved into a BValue. The process by encoding this type of Niagara data into Haystack format can be found in the section *Encoding Niagara Component Space in Haystack.*

   Example 1: Invoke an action with no arguments
   The simplest example of an invoke action is to tell a component to invoke one of its actions that requires no input other than telling it to execute the action. For this example, we will tell a BooleanPoint to come out of an override state.

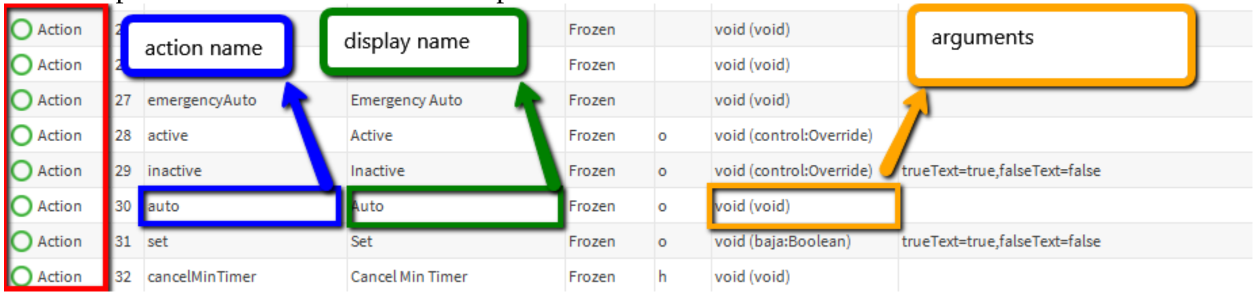
   We start with the point in an override state.



BooleanWritableTest (Boolean Writable)

Facets	trueText=true,falseText=false
Proxy Ext	null
Out	true {overridden} @ 8
In1	- {null}
In2	- {null}
In3	- {null}
In4	- {null}
In5	- {null}
In6	- {null}
In7	- {null}
In8	true {ok}
In9	- {null}
In10	- {null}
In11	- {null}
In12	- {null}
In13	- {null}
In14	- {null}
In15	- {null}
In16	- {null}
Fallback	- {null}

If we inspect the slot sheet of this component:



Action	2			Frozen		void (void)	
Action	2			Frozen		void (void)	
Action	27	emergencyAuto	Emergency Auto	Frozen		void (void)	
Action	28	active	Active	Frozen	o	void (control:Override)	
Action	29	inactive	Inactive	Frozen	o	void (control:Override)	trueText=true,falseText=false
Action	30	auto	Auto	Frozen	o	void (void)	
Action	31	set	Set	Frozen	o	void (baja:Boolean)	trueText=true,falseText=false
Action	32	cancelMinTimer	Cancel Min Timer	Frozen	h	void (void)	

We can extract the action name we need to send is “auto” and the arguments indicating “void” means there are no arguments.

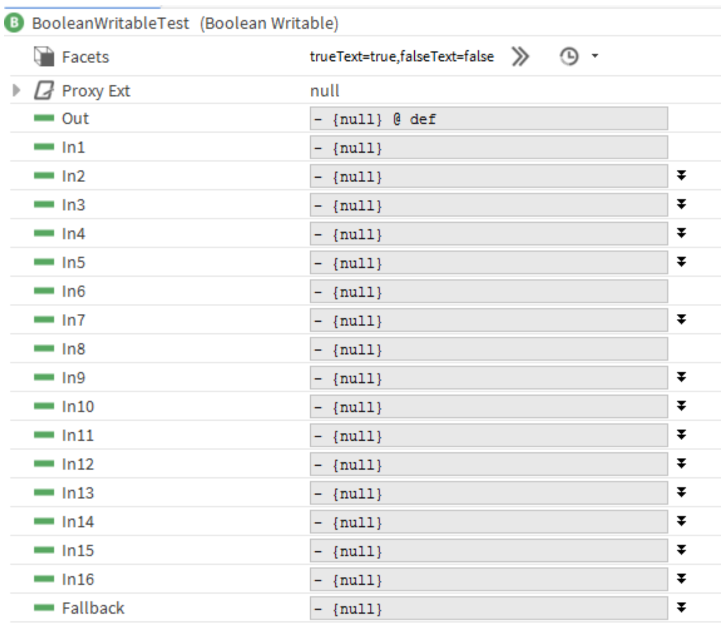
We send the following grid to the haystack invokeAction op to invoke the “auto” action on this component.

```
ver:"3.0" action:"auto" id:@KodaroHaystackDriver-36bfe
empty
```

This will return an empty grid as this action has no return value.

```
ver:"3.0"
empty
```

And take the point out of override:



BooleanWritableTest (Boolean Writable)

Facets	trueText=true,falseText=false
Proxy Ext	null
Out	- {null} @ def
In1	- {null}
In2	- {null}
In3	- {null}
In4	- {null}
In5	- {null}
In6	- {null}
In7	- {null}
In8	- {null}
In9	- {null}
In10	- {null}
In11	- {null}
In12	- {null}
In13	- {null}
In14	- {null}
In15	- {null}
In16	- {null}
Fallback	- {null}

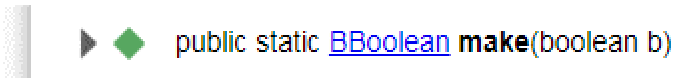
Example 2: Invoke an action with an argument
Looking at the same point from example 1, we will now attempt to call the "set" function to give it a default or fallback value so that it is not null. To do this we must pass in a baja:Boolean argument as we can extract from the slot sheet as described in Example 1 of this section.

To encode a baja:Boolean, we can use the *Encoding Niagara Component Space in Haystack* as a resource to figuring this out. Following that section, we already know we need the module and type, which are given to use from the slot sheet, baja and Boolean respectively.

The top level of this component will be:

```
{
  "module":"baja",
  "type":"Boolean",
  "args":{}
}
```

We don't have any arguments for this object yet, but we need them. We refer to the BBoolean Niagara API to determine what we need to pass in:



public static BBoolean **make**(boolean b)

This tells us we need to construct a primitive boolean value to pass in.

```
{
    "class":"boolean",
    "args":"true"
}
```

Our full argument then becomes:

```
{
  "module":"baja",
  "type":"Boolean",
  "args":{
    "class":"boolean",
    "args":"true"
  }
}
```

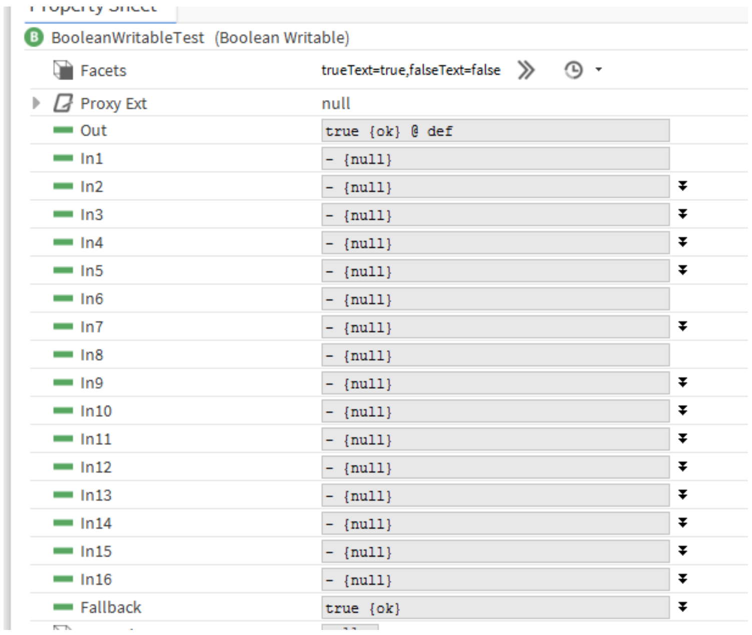
We encode this into the grid for submitting to the invokeAction op.

```
ver:"3.0" action:"set" id:@KodaroHaystackDriver-36bfe
module,type,args
"baja","Boolean",{class:"boolean" args:"true"}
```

This will return an empty grid as this action has no return value.

```
ver:"3.0"
empty
```

And set the point to true as it's fallback/default value.



BooleanWritableTest (Boolean Writable)

Facets	trueText=true,falseText=false
Proxy Ext	null
Out	true {ok} @ def
In1	- {null}
In2	- {null}
In3	- {null}
In4	- {null}
In5	- {null}
In6	- {null}
In7	- {null}
In8	- {null}
In9	- {null}
In10	- {null}
In11	- {null}
In12	- {null}
In13	- {null}
In14	- {null}
In15	- {null}
In16	- {null}
Fallback	true {ok}

2. **Ack Alarm – Acknowledges an alarm in the Niagara alarm database by its UUID**

   When submitting a request to the ackAlarm action, there are no arguments to pass so the grid you pass will be empty. The meta data of the grid should contain the action name "ackAlarm" and the ID should be the UUID of the alarm you wish to acknowledge.

   Submitting a request of:

```
ver:"3.0" action:"ackAlarm" id:@4d2a7ea1-4000-4ab4-904e-2d10c7a629d3
empty
```

   Will return an echo of the UUID committed on success:

```
ver:"3.0" success
uuid
"4d2a7ea1-4000-4ab4-904e-2d10c7a629d3"
```

3. **Sync Tags – Synchronize tags submitted in the Niagara tag database.**

   The non-standard haystack "syncTags" op is designed to expand on the standard haystack ops to enable integration with the Niagara Tag space which is not strictly haystack compliant in nature. Due to the complexities of working with the Niagara tag space, it is recommended that you have some domain knowledge before proceeding with this section.

   It is important to note before proceeding in this section that the tag space is not the same as the component space. If you wish to set different properties on a component, then you should move to the *Using the Commit OP* section of this document. This method is only intended to update, add and remove tags from components and not directly modify the components themselves.

When submitting a request to the syncTags action, you must pass a grid with a single row containing all the tags to sync including the id of the component you wish to sync them on. The meta data of the grid should contain the action name "syncTags" and the ID can be anything that resolves to a haystack Ref. This must be present for invokeAction to maintain haystack compliance, but it is not used in resolving of the component, that is done at the row level.

Sync Behavior:

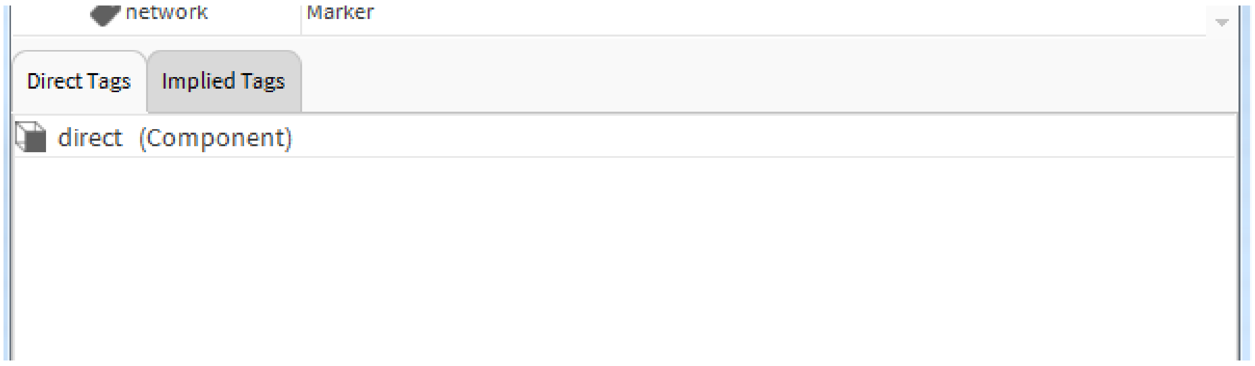
Adding Tags: Any tag present in the row submitted for the Component identified by its ID will be added if it doesn't already exist on the component.

Updating Tags: Any tag present in the row submitted for the Component identified by its ID will be updated to the provided value assuming it is of the correct data type. If it is an incompatible data type, it will fail to be updated.

Removing Tags: Any tag that contains the Remove marker as its value will be deleted from the component.

Example 1: Adding Tags
In this example we'll show how to add a tag using the tag sync.

We start by examining a Component in Niagara and determine what tags are needed. In this example, we'll assume this is a sensor point but we do not have a sensor tag.

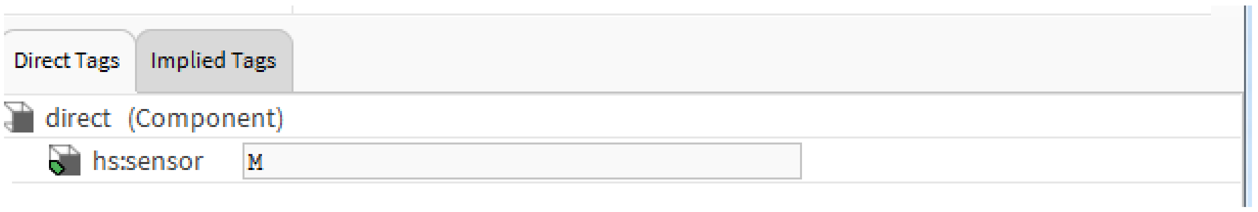


To add a sensor as a marker tag, we will build out a Dict that will contain the sensor tag since we can see that there are no direct tags of sensor currently added to this point.

By submitting the following grid:

```
ver:"3.0" action:"syncTags" id:@null
id,sensor
@KodaroHaystackDriver-36bfe,"M"
```

We can inspect the point again and see that the hs:sensor tag has now been added.



direct (Component)
hs:sensor M

But where did the "hs" come from? Niagara's tag service has an additional component of a name space defined by the tag dictionary. When syncing tags, you do not need to specify this as this is not a haystack compliant construct. The driver will attempt to resolve the name space in the order the tag dictionaries exist in the system. This means that if you had a custom tag dictionary with a sensor tag you wanted to use that was not from the haystack dictionary (hs), then you would need to re-order that dictionary above the haystack in the tag dictionary service in your station.

As you can see, all we had to do was specify which tags we wanted to sync. We did not need to know about any other tags that were present on the point. In a similar fashion, you can send tags you wish to update and/or remove and the tagSync action will only attempt to operate on those tags submitted and leave all othere tags in their current state unmodified.

Encoding Niagara Component Space in Haystack

Create Niagara components using haystack encoding requires intimate knowledge of the Niagara domain, specifically the API's around constructing instances of BComponents and/or BValues which will be generically referred to as Niagara objects from here on.

For the haystack driver to create a Niagara object for either updating or adding, it must receive very specific information about the object it is working with. In a general sense it needs the following information:

For primitive data types, you need to provide:

1. The class
2. The value as a string

For more information on Java primitive types see
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

For complex data types, you need to provide:

1. The module that defined the object
2. Type name or full class path:
    a. The type name of the object as defined by the module o
    b. The full java class path to the object
3. The arguments that will be passed into the Java constructor. Since not all Java classes have public constructors that take arguments, the driver will also attempt other methods that are known common constructor methods that exist on most Niagara Components.
    a. If there is no constructor with the given arguments, it will attempt to find any method called "make" that have the same argument signature.
    b. If there is no constructor or method called make, it will attempt to find a method called "decodeFromString".

Example 1: Primitive Data Type 1
We'll start with a simple example of creating a primitive object.
For both Zinc and JSON the encoding for an object for a primitive is the same and is as follows:

```
{
  "class": "String",
  "args": "updated value2"
}
```

We only need to create a very simple JSON object that contains the class and the args field. This tells the haystack driver to resolve the java String class and find a constructor that has a string argument. You can see by the Java API docs all the valid constructors to confirm the arguments you are passing in (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html).

This would be the same in Java code as:
new String("updated value2")

Example 2: Primitive Data Type 2
Using the information from Primitive Data Type example 1, we'll now remove the args argument.

If we now go ahead and remove the data from the args argument:
```
 {
  "class": "String"
}
```

This tells the haystack driver to resolve the String constructor that takes no arguments. This would be the same in Java code as:
```
new String()
```

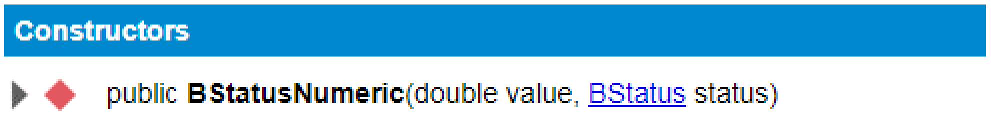
Example 3: Complex Data Type

While primitives are useful for setting some basic meta data on an object, there are a lot of complex data types in Niagara. Even when we look at something that seems simple like updating a Numeric Point in Niagara, this is a complex data type since it has both a value and a status.

We'll use encoding from the Add Example section:

```
{
 "type": "StatusNumeric",
 "module": "baja",
 "args": [
   {
     "class": "double",
     "args": "42"
   },
   {
     "type": "Status",
     "module": "baja",
     "args": {
       "class": "int",
       "args": "0"
     }
   }
 ]
}
```

The top-level object is defined as a StatusNumeric from the baja module. There are two arguments to create this object and are provided as a JSON Array or haystack list format. As you can see from the Niagara API, the arguments are defined as a double and a BStatus.



Constructors

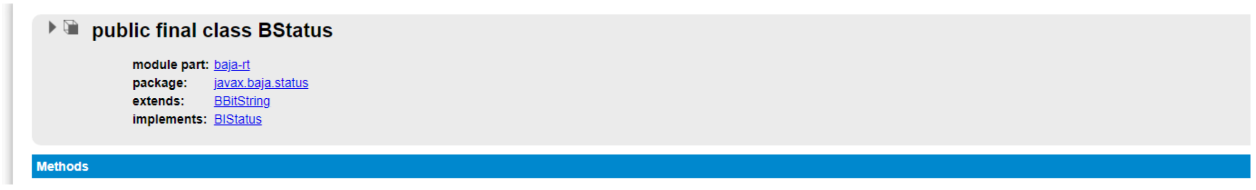
public **BStatusNumeric**(double value, BStatus status)

Argument 1: Defined as a primitive class double of 42

Argument 2: Defines a complex data type of Status from the baja module. There is 1 argument to create this object.

> Sub Argument 1: Defines a primitive class of int and 0. In the case of this, we are referring to a BStatus which can be created by defining the bitmask directly as one of its make methods. A value of 0 indicates a status of "ok".

Since BStatus is also a complex data type, we need to now nest a BStatus constructor into the object. By inspection of the BStatus Niagara API, you can see there are no Constructors as there is for a BStatusNumeric. So what do we do then?

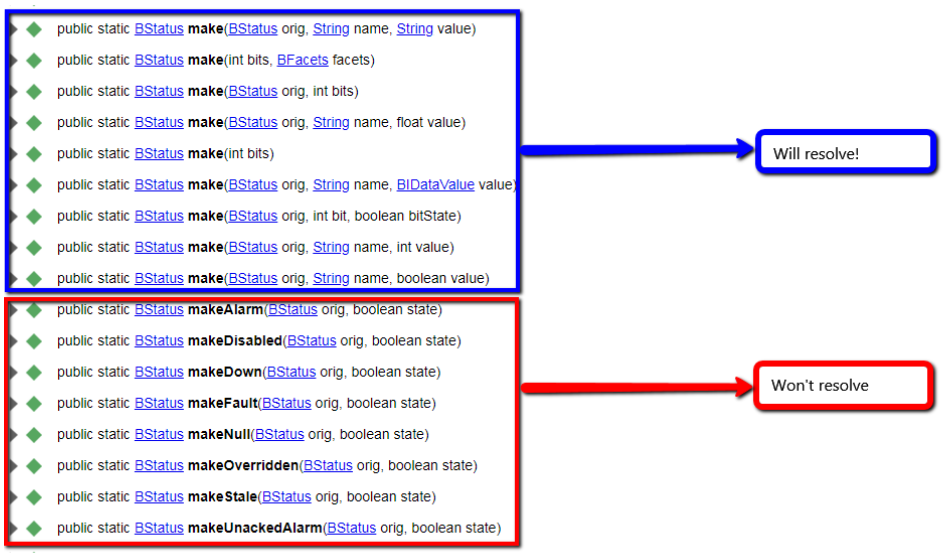


public final class BStatus

module part:	baja-rt
package:	javax.baja.status
extends:	BBitString
implements:	BIStatus

Methods

If there are no constructors available, we need to find if there's a static method called "make" that can create a new instance of the object we want. By inspecting the Niagara API for this class, we can see that there are a lot of methods that will return a BStatus that begin with make. Only methods that are named make will be used. Convenience methods such as makeAlarm, makeDisabled, etc are too specific to a class and are not resolvable in this context.



public static BStatus **make**(BStatus orig, String name, String value)
public static BStatus **make**(int bits, BFacets facets)
public static BStatus **make**(BStatus orig, int bits)
public static BStatus **make**(BStatus orig, String name, float value)
public static BStatus **make**(int bits)
public static BStatus **make**(BStatus orig, String name, BIDataValue value)
public static BStatus **make**(BStatus orig, int bit, boolean bitState)
public static BStatus **make**(BStatus orig, String name, int value)
public static BStatus **make**(BStatus orig, String name, boolean value)
public static BStatus **makeAlarm**(BStatus orig, boolean state)
public static BStatus **makeDisabled**(BStatus orig, boolean state)
public static BStatus **makeDown**(BStatus orig, boolean state)
public static BStatus **makeFault**(BStatus orig, boolean state)
public static BStatus **makeNull**(BStatus orig, boolean state)
public static BStatus **makeOverridden**(BStatus orig, boolean state)
public static BStatus **makeStale**(BStatus orig, boolean state)
public static BStatus **makeUnackedAlarm**(BStatus orig, boolean state)

For our example, we will use the simplest version of these which is

BStatus.make(int bits)

The "bits" argument is not defined by this document and will require understanding of the BStatus bitmask which is beyond the scope of this document. A bitmask of 0 indicates an "ok" status in Niagara so that is why we pass we use the arg:

```
{
  "class": "int",
  "args": "0"
}
```

This is a very simple way to tell the haystack driver to resolve any constructor that takes in an integer of value 0 and if it can't resolve, we will look at a method called make that takes in an integer value and pass in a value of 0.

If we now look at the entirety of what we just told the haystack driver to do as it relates to java code, it would be the same as doing:

```
  new BStatusNumeric(42,BStatus.make(0))
```

And now we can break down a 1:1 relationship between the object encoding and the Niagara object resolving.

BStatusNumeric: - Outer Object (in bold)
  type: StatusNumeric - Note the dropping of the 'B'. This is common in all type resolving.
  module: baja – Name of the module where the class is defined.
  args: Array or List of all the arguments required to construct the object defined by type and module
    Argument 1: double value of 42
    Argument 2: BStatus Value

This resolves the outer object to use the constructor **BStatusNumeric(**double, BStatus**)**

Argument 1:
Relates directly to the 42 in the constructor: BStatusNumeric(**42,**BStatus.make(0))

```
{
  "class": "double",
  "args": "42"
}
```

Argument 2:
Relates directly to the BStatus.make(0) in the constructor: BStatusNumeric(42,**BStatus.make(0))**

```
{
  "type": "Status",
  "module": "baja",
  "args": {
    "class": "int",
    "args": "0"
  }
}
```

HISTORY

November 11, 2020: 1.1.1 (Release)

- Added – Support for newer SkySpark rule engine to generate alarms from sparks. Defaults to newer engine so existing installs using legacy spark engine will experience a breaking change on initial update and will need to update configuration to use the correct engine.

October 15, 2020: 1.0.31 (Release)

- Fix – Update hisEnd precision discrepancy to stop history duplication on exports to SkySpark

October 15, 2020: 1.0.30

- Update – Normalize enum tag encodings

September 30, 2020: 1.0.29

- Added – overrideExpiration added to extended Niagara tags

July 13, 2020: 1.0.28 (Release)

- Fixed – Make lease optional in meta of watchSub op

July 6, 2020: 1.0.27

- Added – TagSyncConfig object to define a default missing tag behavior during syncTags action
- Added – Extended Niagara tags included in export subscription messages
- Added – Option status values option to be exported with history only export and subscription updates
- Update – Add additional mappings of implicit tags to facets during syncTags action
    - Map minVal tag to min facets value
    - Map maxVal tag to max facets value
- Fixed – Account for existing watchId's by adding components to existing watches for watchSub operation as per haystack specification https://project-haystack.org/doc/Ops#watchSub
- Fixed – Account for refresh marker in watchPoll as per haystack specification https://project-haystack.org/doc/Ops#watchPoll
- Fixed – Resolve proper SkySpark version 3 ID's for updating subscriptions on exports

June 10, 2020: 1.0.26

- Update – Finalize action encodings for release

March 24, 2020: 1.0.25

- Update – Default grid data type from text/plain to text/zinc
- Add – Preliminary action encodings added

February 07, 2020: 1.0.24

- Fix – Return empty dict instead of error data when missing rec is resolved in watchSub operation. Revert change made in 1.0.22

February 02, 2020: 1.0.23 (Release)

- Fix – Handle bad/missing record(s) in watchSub request causing SkySpark connection reset

January 27, 2020: 1.0.22

- Update – Implement mult-threading for encoding on watchSub
- Update – Add history only export option.

August 28, 2019: 1.0.21 (Release)

- Update – Detect relationship loops during hierarchy export resolving. Log issue and gracefully exit infinite loops when detected.
- Update – Add new debug mode for exports
- Update – Add additional logging to history export errors.
- Update – Add time zone to histories when useTimeZone option is enabled in exports.
- Update – Enhance history time zone resolving from Java to Haystack.
- Update – Add optional time range filters for both hisRead's and exports
- Update – Implement read by id option
- Update – Change default web op timeout from 10 seconds to 1 minute.

March 19, 2019: 1.0.20 (Release)

- Update – Implement new history merging and resolving algorithm in exports (1.0.19.6.3)
- Removed – Deprecated exporter history merger object in favor of new merging algorithm (1.0.19.6.3)
- Update – Use pointId from Niagara Network proxy extensions when resolving remote histories (1.0.19.6.2)
- Update – Add resolved history from history tag to available merge list on hisRead (1.0.19.16.2)
- Update – Enhanced history resolving methods for hisRead (1.0.19.6.1)
- Fix - Set the root of the BQL query during source data resolution to account for duplicate relative source paths (1.0.19.5.4)
- Update - Enhance synchronization of alarm watches (1.0.19.5.3)
- Fix - Account for escaped characters not being able to be queried in BQL (1.0.19.5.2)
- Fix - Unescape names during encoding (1.0.19.5.1)
- Update - Misc logging updates (1.0.19.3)

November 07, 2018: 1.0.19.2 (Release)

- Added read timeout to device
- Added connection timeout to device
- Fix logging index error on hierarchy export

November 02, 2018: 1.0.19.1

- Remove option in commit op changed to handle multiple ID's.
- Remove option examples added to documentation

- Updated licensing documentation

November 01, 2018: 1.0.19

- Added Nav Op
- Added Formats Op
- text/csv response format implemented
- plain/text response format implemented
- Implemented remove option in commit op

October 29, 2018: 1.0.18.12

- AlarmWatch encoding process optimization

October 26, 2018: 1.0.18.11

- Additional debug messaging added to history export process

October 26, 2018: 1.0.18.10

- Invoke Action on Components Implemented

October 26, 2018: 1.0.18.9

- Fix watchSub bug not returning unresolved ids as blank rows

October 25, 2018: 1.0.18.8

- Applied Niagara permissions model to both add and update options in commit op.

October 24, 2018: 1.0.18.7

- Dict to Niagara component space format changed
- Implemented add option in commit op

October 24, 2018: 1.0.18.6

- Commit op added
    - Update – implemented
    - Add – not implemented
    - Remove – not implemented
- JSON response format implemented

October 18, 2018: 1.0.18.5

- Replaced unsorted lists with sorted lists for more consistent tracking.

October 18, 2018: 1.0.18.4

- Adds additional encoding methods to handle non-haystack standard names for object/property names

October 09, 2018: 1.0.18.3

- Add imported Niagara alarm sources resolving to alarm encoding method

October 03, 2018: 1.0.18.2

- Debug statement cleanup.

September 11, 2018: 1.0.18.1

- Alarm encoding method will now encode all tags on the component that the equipRef points to. equipRef must be resolved for additional data to be added
- Added ackAlarm action for invokeAction haystack end point

August 15, 2018: 1.0.18 (Release)

- Handle 4.6 curVal and writeVal data type changes

May 17, 2018: 1.0.17.7

- Added BFormat resolving to alarmData when alarms resolved via a BQL Query

January 22, 2018: 1.0.17.5 (Release)

- Misc. updates for N4.3+ security enhancements.

November 09, 2017: 1.0.17.4 (Release)

- Updated to latest haystack protocol to include ver:"3.0" in responses instead of ver:"2.0".

November 09, 2017: 1.0.17.3

- Licensing model changed to allow web ops on LocalDevice object to always operate even without a license key.

October 30, 2017: 1.0.17.2 (Release)

- Fix Web Op "Ops". Prior to this release was only returning the last Op in the list instead of a full grid of all the ops available.

October 2, 2017: 1.0.17.1 (Release)

- Add Network Permissions to allow outgoing connections with security enforced with Niagara 4.3.
- Re-compiled to work with latest javax.servlet api 3.1.0.

October 2, 2017: 1.0.17

- Update legacy logging from AX to N4. Fixes security issue now enforced with Niagara 4.3 releases.

September 27, 2017: 1.0.16.1 (Release)

- Clean up miscellaneous error messaging during station start concerning missing classes.

September 26, 2017: 1.0.16 (Release)

- Fixed - Relationships not correctly being resolved when created outside of the Haystack dictionary. This also fixes instances where no ID would be generated if the Haystack dictionary was not installed.

September 06, 2017: 1.0.15 (Release)

- Fix the O (oh) vs zero issue in license product code

August 22, 2017: 1.0.14

- Better error handling during Alarm Resolving

August 21, 2017: 1.0.13.1

- Added Unit Mapping tr-hr in Niagara To tonrefhr in Skyspark

August 21, 2017: 1.0.13

- Addition of the HaystackAlarmWatch object

July 06, 2017: 1.0.12

- Encode Niagara facets into a Haystack Dictionary for more haystack compliant encoding
- Relation names added during tag encoding process

June 06, 2017: 1.0.11 (Release)

- Add cancel method to exports
- Fix concurrent update issue when running multiple exports simultaneously. Only an issue with Skyspark 3 due to ID pattern change

May 22, 2017: 1.0.10

- Merge encoding methods for exports and web ops

May 02, 2017: 1.0.9 (Release)

- Initial Release

February 08, 2017: 1.0.0 (Release)

- Initial Beta release