Technical Document

Niagara JSON Toolkit Guide

December 9, 2019

niagara4

Niagara JSON Toolkit Guide

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

Contents

About this guide

The JSON Toolkit provides a way to easily extract data from a station as well as a way to input information to control connected devices.

The beginning chapters introduce data output and input. The Developer Guide chapter explains how to extend toolkit features.

Document change log

Changes to this document are listed in this topic.

December 9, 2019

Initial document release

Related documentation

These documents provide additional information about how to construct data models using the Niagara Framework.

Internal resources

- Niagara Developer Guide
- Niagara Drivers Guide
- Niagara Graphics Guide
- Bajadoc (accessed through the Workbench Help system)

External resources

- Java Platform Standard Edition 7 Documentation: https://docs.oracle.com/javase/7/docs/
- Unix time: https://en.wikipedia.org/wiki/Unix_time
- Chart.js, JavaScript charting for designers and developers: https://www.chartjs.org/
- Google Chart: https://developers.google.com/chart

Chapter 1 Introduction

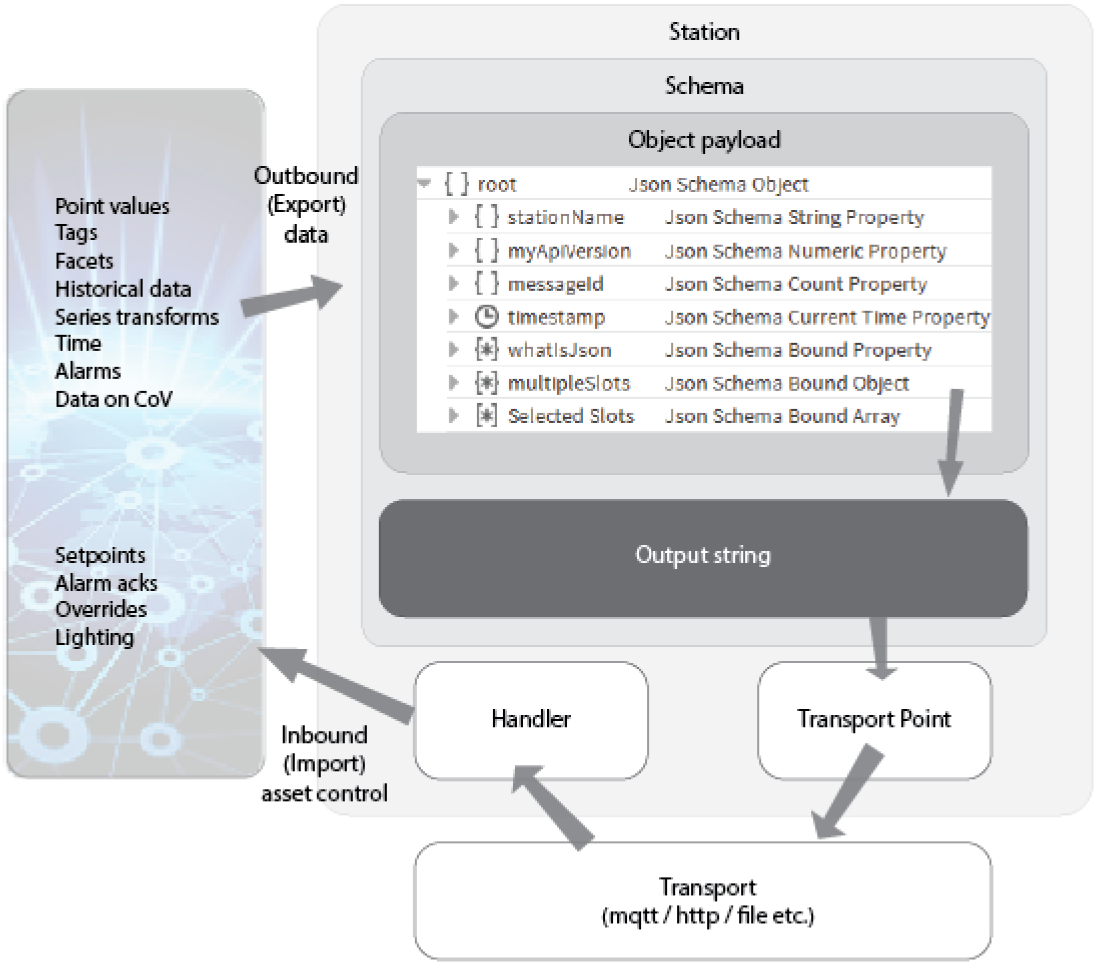
Topics covered in this chapter

- ♦ Quick JSON example
- ♦ JSON Toolkit use cases
- ♦ Transport protocols
- ♦ Feature summary
- ♦ Comparison to alternatives
- ♦ License requirements
- ♦ JSON schema service
- ♦ Supervisor

The JSON Toolkit module adds functionality to the Niagara Framework®, enabling you to export JSON data (payloads) from a station, or, when importing data, to influence the station in some way. A schema generates a payload for export, whilst a handler processes imported JSON. The Toolkit is intended to give you the power to adapt as needed.

JSON (JavaScript Object Notation) is a simple, lightweight, data encoded string. Used for data interchange since 2002 to communicate between a web browser and a server for the Javascript language, it has gained popularity and is used in many scenarios beyond those implemented in 2002. Many IoT devices can easily receive a JSON payload.

Figure 1 Logical JSON flow


Station
Schema
Object payload
root Json Schema Object
stationName Json Schema String Property
myApiVersion Json Schema Numeric Property
messageId Json Schema Count Property
timestamp Json Schema Current Time Property
whatIsJson Json Schema Bound Property
multipleSlots Json Schema Bound Object
Selected Slots Json Schema Bound Array
Output string
Point values
Tags
Facets
Historical data
Series transforms
Time
Alarms
Data on CoV
Outbound (Export) data
Setpoints
Alarm acks
Overrides
Lighting
Inbound (Import) asset control
Handler
Transport Point
Transport (mqtt / http / file etc.)

On the left is the universe of data available to a Niagara *Station*. The station database provides some data; other data can come from outside the station. The *Schema* contains configuration properties, which set up its functions. A schema updates when a CoV triggers a generation action from a bound entity or a person invokes the generate action on a schema Property Sheet. This causes any linked properties of the JSON *Object payload* to create the *Output string*, which retrieves and routes the data onward through a *Transport Point* and *Handler* to an alarm recipient, the cloud or other destination, and back to the device for asset control, such as to control the lighting in a home or acknowledge an alarm.

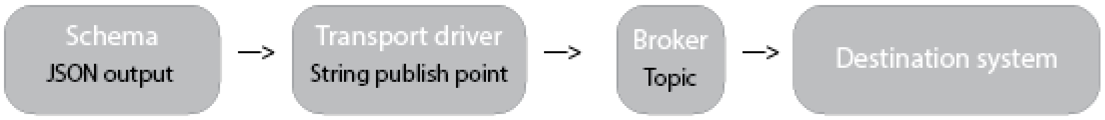
The format of the JSON output string is relatively simple, organised into a list of `key:value` pairs, with support for data types: Numeric, Boolean, Enum and String much like Niagara points. JSON messages can use any sequence of objects, arrays and key/value pairs. The JSON Toolkit is flexible. You, as a developer can extend the Toolkit or use APIs to access station data. You can drag schema elements around and change the order of the messages.

JSON supports two data structures: objects and arrays. Complexity emerges from these simple constructs mainly due to the variation in expected payload between different pieces of software, and also their expected encoding of non-primitive types, such as date and time. This is where the demand arises for a flexible solution to marshal Niagara’s rich object model to and from the JSON format.

You can extend the Toolkit or use APIs to access a station’s data. JSON can post data to APIs for data transmission. For example, using the inbound components of the JSON Toolkit, external systems can send a JSON-encoded message to a Niagara station to change a setpoint or acknowledge an alarm.

As the data manipulator, you set up data retrieval and use by creating links between JSON objects. Each schema contains a single root object, which itself contains the JSON objects that establish the links.

Figure 2 JSON workflow



From the station to the destination, you link the output string, typically via MQTT, to a string-publish point, which sends the payload to a topic in the broker that forwards (transports) it to the destination system. The schema itself is transport agnostic. Linking produces the desired result.

For example, just as an external oBIX client can poll a station for data, the JSON output can be retrieved via an HTTP GET request to a URL that exposes its contents as a web servlet. Using JSON, you can have the same rich data that oBIX provides without the pre-defined oBIX format. Using JSON, you choose the format of the data, which affords total flexibility.

MQTT brokers can link the output of a JSON schema to a cloud platform, such as Bluemix, Google Cloud, and AWS.

Quick JSON example

This is a simple example with JSON objects and arrays.

```
{
  "temperature": [
     {
        "Timestamp":"28-Jun-18 4:42 PM BST",
        "Value":21.83
     }
  ]
}
```

In this example:

- A root *object* encloses the whole *payload* with open and close braces { }. A JSON schema object is a named container that holds other schema entities. By itself an object has no properties or additional containers.
- A JSON *array* named “temperature”. An array is a named container represented by brackets [ ] that holds other schema entities.
- An object, in braces { }, is contained by the “temperature” array.
- The object contains a string “Timestamp” and a numeric “Value.” Each appears as a key/value pair inside the object.

You could construct this simple example using `StringConcat` components from the kitControl module, however, if you have many points it would take a lot of work to create this construct for every point. A JSON schema can work across many points without extra effort.

JSON Toolkit use cases

The following information includes a summary of typical JSON use cases, transport protocols and a summary of the toolkit features.

Typical use cases

Following are some possible use cases for the JSON Toolkit:

- Cloud connectivity (IoT)
- Visualization
- Device connectivity

- Machine learning
- Analytics
- Data archival

Transport protocols

The JSON Toolkit itself does not mandate the transport protocol used.

Potential transport protocols include:

- MQTT (by linking the JSON schema output to an Mqtt String Publish Point)
- HTTP(s)
- Box (bajaux widget)
- File

These options may be valid for both incoming and outgoing JSON payloads. When linking to a publish or subscribe control point, you may need to use an **`Engine Cycle Message Queue`** component to ensure that the schema outputs all messages to the linked transport.

Feature summary

The JSON Toolkit supports a significant list of features and options to aid the engineering effort.

- Customer-definable JSON payloads
- Payloads for types other than points, such as, tags and facets
- Payloads for histories, series transform, and alarms
- Data selection using either bindings (ords), bql, or the addition of markers that directly identify points
- Support for encoding alarm events via a **`JsonAlarmRecipient`**
- The ability to respond to incoming requests to change a setpoint or acknowledge an alarm by uuid
- On-demand payloads generated by a CoV
- Topic generation for a relative schema (for example, mqtt publish topic)
- Tuning policies to throttle output
- Program object-based overrides
- The ability for developers to extend the toolkit

Comparison to alternatives

JSON Toolkit alternatives include oBIX and bajaScript.

- oBIX (obix.org) provides a comprehensive connectivity option for Niagara, the JSON Toolkit differs by offering a flexible, user-defined payload, and support for publish-on-change.
- bajaScript (bajascript.com) provides a means to access the Niagara component model with convenient support for complex objects, subscription, action invocation, and querying.
- Project Haystack (project-haystack.org) offers both a common semantic model and a protocol to enable the exchange of data. These tags can be included in the payloads generated by the JSON Toolkit.

In contrast to the above options, the JSON Toolkit does not dictate the protocol or layout used to exchange data. This could be an advantage when dealing with charting libraries and cloud service providers who expect to send or receive data in a specific format.

License requirements

To use the JSON Toolkit, your host requires the DR-JSON or DR-S-JSON feature added to the host's license. Production (non-demo) licenses also require an active Software Maintenance Agreement (SMA) for the toolkit to function. Engineering or Demo licenses should have this feature added by default.

SMA Expiration Monitor

In addition to the licensed feature requirement, the toolkit requires an active SMA in order to run. The Expiration Monitor increases notifications as expiration of this agreement approaches. It runs on startup, the monitor (of the **`JsonSchemaService`**) checks every 24 hours to establish if the expiration date is within the warning period, or expired, and generates an `offNormal` or fault alarm accordingly. Although the alarms are likely the most accessible type of notification, the SMA Monitor also logs the days remaining to the station console, which, for example, could be shown on a dashboard. The station's **`UserService`** also has an **`SMA Notification`** property, which alerts users at the web login screen.

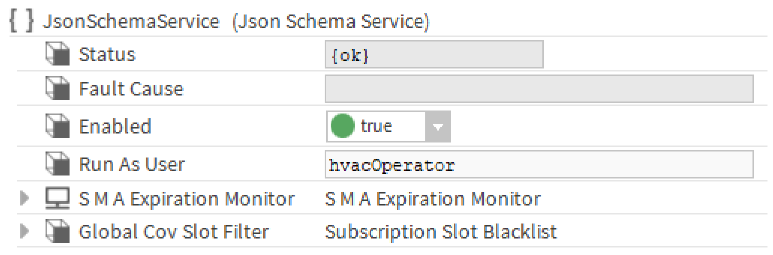
As the extension of the SMA currently requires a reboot to install the new license, once the monitor detects that the agreement has expired it performs no further checks until the station starts again.

JSON schema service

To use the toolkit, you first need to set up the **`JsonSchemaService`** by adding it to the station **Services** container.

Adding the **`JsonSchemaService`** component to the **Services** container can provide some station global filtering as well as the ability to restrict user access when handling inbound messages.

Figure 3 JsonSchemaService properties



{ } JsonSchemaService (Json Schema Service)

Status	{ok}
Fault Cause	
Enabled	● true
Run As User	hvacOperator
S M A Expiration Monitor	S M A Expiration Monitor
Global Cov Slot Filter	Subscription Slot Blacklist

The Spy page for the service maintains a registry of the export markers contained within the station. This registry might prove useful when debugging issues with relative schema used in conjunction with the export marker paradigm. In the event that setpoint changes are received, the register aids in finding the marked points.

Global Cov Slot Filter

The **`Global Cov Slot Filter`** can denote which slots to ignore when subscribing to bound values. The default list of slots includes a good example of why this function is necessary in that changes to a component's **`wsAnnotation`** property (which defines the position and size of a component glyph on the wire sheet) should generally be excluded from the changes of value reported to any upstream consumer of data.

Run As User

Another important property provided by the **`JsonSchemaService`** is **`Run As User`**. This property specifies the user account to assume in the event that a router processes an incoming change. For example, this assumption is mandatory when using the **`SetpointHandler`**, so that any changes triggered by a cloud platform are limited to areas in the station where the platform has write access. JSON schema export data also optionally use this property.

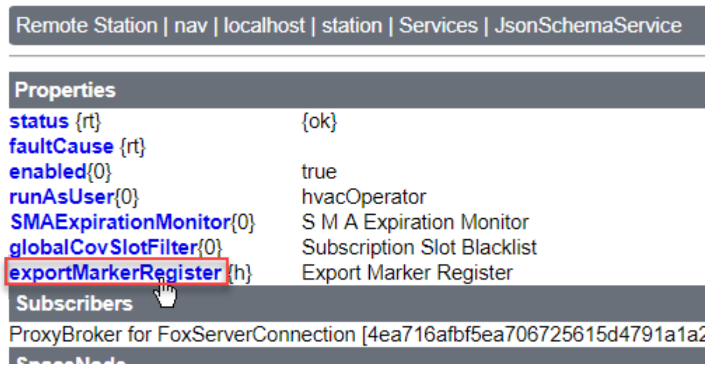
Operation	Optional?	How it works
Configuring a setpoint handler for incoming JSON	No	The set operation only succeeds if the **`Run As User`** is a real user who has operator write permission on the slot target.
Defining a schema for exporting JSON	Yes	When set, the data value of the exported slot defaults to an empty string unless the **`Run As User`** is a real user who has operator read permission on the slot.

NOTE: **`Run As User`** is important for security. This property may only be set by a super user.

Debugging with Spy page

The Spy page for the **`JsonSchemaService`** also has a registry of the export markers (refer to the *Export Marker* topic) contained within the station, which might prove useful when debugging relative schema issues that are used in conjunction with the export marker paradigm. The central register aids finding the marked points in the event that setpoint changes are received.

Figure 4 JsonSchmaService Spy page link



Remote Station | nav | localhost | station | Services | JsonSchemaService

Properties

status {rt}	{ok}
faultCause {rt}	
enabled{0}	true
runAsUser{0}	hvacOperator
SMAExpirationMonitor{0}	S M A Expiration Monitor
globalCovSlotFilter{0}	Subscription Slot Blacklist
exportMarkerRegister {h}	Export Marker Register

Subscribers

ProxyBroker for FoxServerConnection [4ea716afbf5ea706725615d4791a1a2

Supervisor

The most convenient deployment of the jsonToolkit in cloud connectivity is to connect directly from the controller schema to the remote transport. However, if a controller does not have remote connectivity, a Supervisor is required. There are a few options to consider.

NiagaraNetwork point export

You import points from subordinate controllers into a Supervisor under the NiagaraNetwork and create Supervisor schemata.

- Schema queries and bindings may target points under the **NiagaraNetwork** and subscription (change of value) will work ok.
- Some information, such as the parent name and original slot path of the points, may not be available in the schemata.
- Tag data and permissions may need to be redefined at the Supervisor level.
- Alarms and histories need to be imported to the Supervisor if you require these data for your schema.
- This approach requires the most configuration overhead and may not be desirable to import all the points to the Supervisor.

System database

Use the system database to index subordinate controllers and sys: ords for queries within a Supervisor schema.

- Example schema query: `station:|sys:|neql:n:point|bql:select name, out.value, out.status`

- Example schema base query: `station:|sys:|neql:n:point|bql:select *`
- Subscription to the remote points works so that change of value is available, as is the parent name and original slot path of the points.
- You cannot use a system database ord for a specific point binding.
- This option is not suitable for export of alarms and histories.

NiagaraNetwork schema export

The schema runs locally on each controller and is linked to a **`StringWritable`**. This writable point is then imported into the Supervisor and linked to remote transport.

- It makes sense to do the processing at the data source where full point fidelity is available.
- The framework deals with permissions locally at each station.
- You have full alarm and history support.
- Linking the imported **`StringWritable`** to a transport, such as MQTT, keeps the point subscribed.

Proxy

If you are using MQTT as your transport, you may set up an intermediary broker to proxy messages to the remote broker. This solution requires extra IT overhead and support.

Chapter 2 Exporting with a JSON schema

Topics covered in this chapter

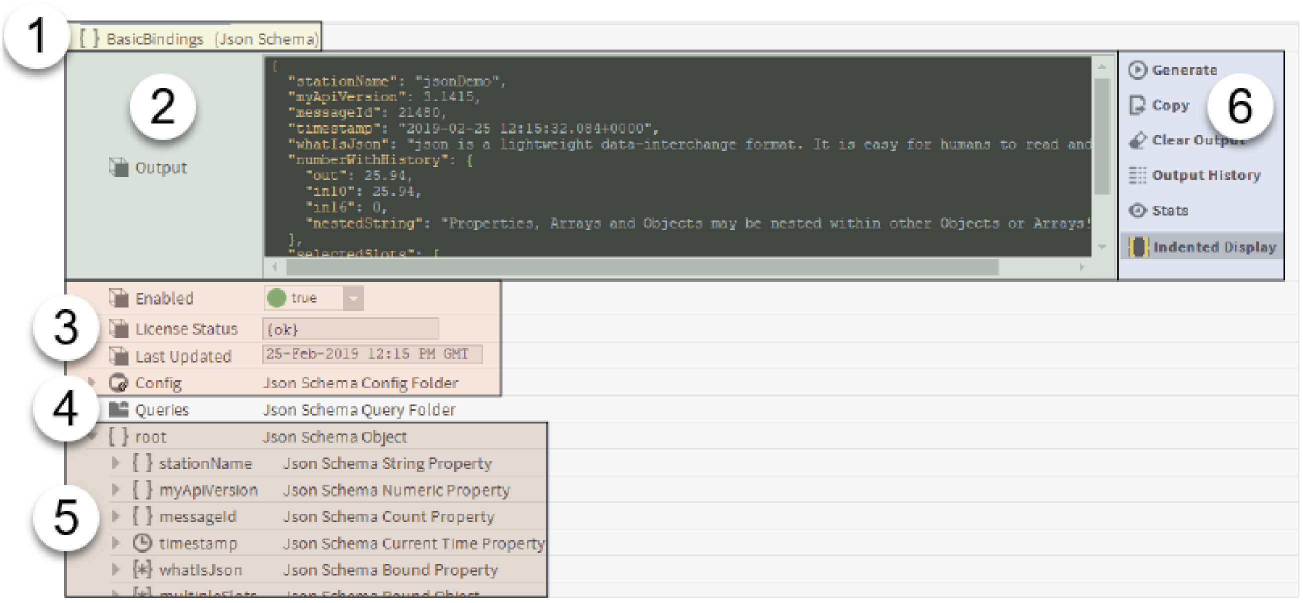
♦ Config folder
♦ Tuning policy
♦ Overrides
♦ Debugging errors (Schema History Debug)
♦ JSON schema metrics
♦ Schema construction
♦ Queries
♦ Alarms
♦ Exporting schema output (JsonExporter)
♦ Exploring the examples

Adding a `JsonSchema` component to your station allows for the construction of a JSON payload to suit the requirements of your particular application.

Overview

There are examples in the **jsonToolkit** palette, which may help with learning how to construct a schema. You can simply drag the `JsonExampleComponents` and `Schemas` folders into a running station to work with them.

Figure 5 Schema parts



```
{
  "stationName": "jsonDemo",
  "myApiVersion": 3.1415,
  "messageId": 21480,
  "timestamp": "2019-02-25 12:15:32.084+0000",
  "whatIsJson": "json is a lightweight data-interchange format. It is easy for humans to read and
  "numberWithHistory": {
    "out": 25.94,
    "in10": 25.94,
    "in16": 0,
    "nestedString": "Properties, Arrays and Objects may be nested within other Objects or Arrays!
  },
  "selectedSlots": [
```

Enabled	true
License Status	{ok}
Last Updated	25-Feb-2019 12:15 PM GMT
Config	Json Schema Config Folder
Queries	Json Schema Query Folder
root	Json Schema Object
stationName	Json Schema String Property
myApiVersion	Json Schema Numeric Property
messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property

You construct a schema by placing “entities” from the **jsonToolkit** palette below a `JsonSchema` in the station and then use configuration properties and queries to get the output you want. Use the numbers in the screen capture to learn about schema elements:

1. You can give each schema a unique name.
2. The `Output` property contains the resulting JSON payload (message or string).

NOTE: The format of this black box (with new lines and spacing) is purely for presentation. The actual string output is minified and does not contain extra spaces.

3. The **`Enabled`** property turns the generation of output, execution of queries and subscription to bound values on and off. The **`Config`** folder contains properties that configure general schema attributes.
4. The **`Queries`** folder can contain query entities to insert bql, historical or alarm database content into a payload.
5. A **`{ } root`** object or an **`[ ] array`** contains JSON entities that structure the **`Output`** message. Some entities may be simple—for example braces { } represent a simple JSON object, while other entities represent Niagara bql queries (refer to the *Niagara Developer Guide*) and, therefore, have the potential to be more complex.
6. Actions build and manage schema contents. The `Generate` action builds and updates schema output. For relative schemata, `Generate` evaluates the base query and publishes the results for each resolved base item.

What can a Schema contain?

The schema supports a nested structure of child entities. These can be Objects, Arrays, or Properties of various types. Niagara alarm, history or point data may populate these entities, which include:

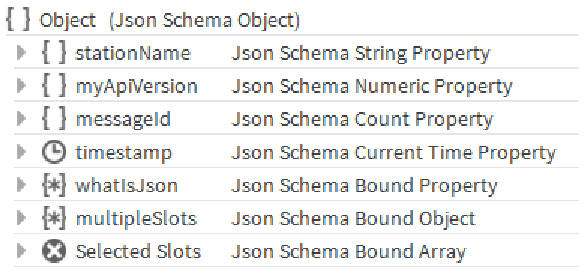
Entity Type	Output
Object	`“objectName” :{"name” : value, “name2” : value2….}`
Array	`“arrayName” :[value, value2….]`
Property	`“key”: value`
Property List	`“key”: value, “key2”: value2`

All entities (minus Property) support nested child entities. This lets you build a schema using a tree structure with entities found in the **jsonToolkit** palette.

What structure is allowed?

Every schema requires a root member that is allowed by the JSON standard: this means an object `{ }` or an array `[ ]`.

Figure 6 root Json Schema Object



{ } Object (Json Schema Object)

stationName	Json Schema String Property
myApiVersion	Json Schema Numeric Property
messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property
multipleSlots	Json Schema Bound Object
Selected Slots	Json Schema Bound Array

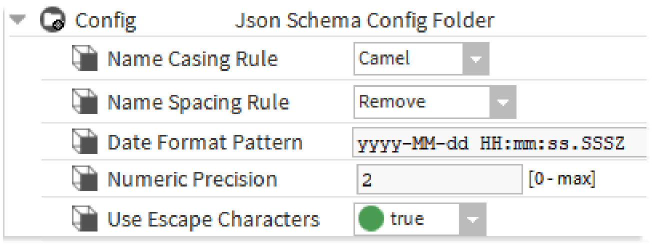
The screen capture shows how Niagara represents a JSON root object in a standard **Property Sheet** view.

Config folder

The JSON Toolkit provides several options to help you create consistent naming and formatting. The root properties in each schema’s **Config** folder provide these consistency properties.

The schema **Config** folder is separate from the station **Config** folder and applies only to the parent schema.

Figure 7 Config properties



Config	Json Schema Config Folder
Name Casing Rule	Camel
Name Spacing Rule	Remove
Date Format Pattern	yyyy-MM-dd HH:mm:ss.SSSZ
Numeric Precision	2 [0 - max]
Use Escape Characters	true

- **`Name Casing Rule`** conforms names to camel case or another style.
- **`Name Spacing Rule`** defines a character to insert between words in a name, such as a space, hyphen, underscore, etc.
- **`Date Format Pattern`** configures dates.
- **`Numeric Precision`** configures the number of decimal digits to show on exported floating point numbers, values are rounded. Point facets are not used.
- **`Use Escape Characters`** turns on and off the use of escape characters around symbols that otherwise would have special meaning. For example, when `false`, $20 becomes a space character.

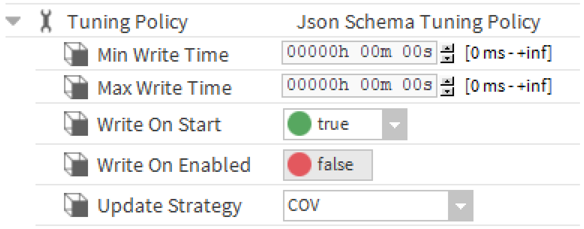
The "Components" chapter documents the options for these properties.

Tuning policy

Most tuning policies properties are explained by the *Niagara Drivers Guide*.

Tuning properties provide rules for evaluating when JSON outputs data and for indicating an update strategy. Configuring this policy can affect system performance. They are located in the Schema **Config** folder.

Figure 8 Tuning policy properties



Tuning Policy	Json Schema Tuning Policy
Min Write Time	00000h 00m 00s [0 ms - +inf]
Max Write Time	00000h 00m 00s [0 ms - +inf]
Write On Start	true
Write On Enabled	false
Update Strategy	COV

`Update Strategy` determines when JSON string generation occurs: at change-of-value or on demand.

There is a built-in **`Min Write Time`** to ensure that hundreds of concurrent CoV changes over a short time do not result in a deluge of JSON messages. For example, when set to five (5) seconds and a change-of-value occurs within five seconds of the last change of value, schema generation defers for a full five seconds. However, if this amount of time exceeds the **`Max Write Time`** setting, the system forces schema generation. In contrast, **`Max Write`** forces an update after the specified interval.

NOTE: A `Force Generate Json` action overrides all tuning policy settings.

Export markers applied to numeric points also have a **`CoV Tolerance`** property which can be used to throttle output.

The **`Write On Start`** and **`Write On Enabled`** properties provide other ways to invoke schema generation, for example, when the station starts.

Overrides

An **Overrides** folder is a standard container under the JSON **Config** folder.

This folder adds a `TypeOverride` component to the schema, should it be necessary to override how the schema converts specific datatypes to JSON. The override applies to anywhere the system encounters the data type in the entire schema. Examples might be:

- replacing Facets with a locally-understood value, such as 'degC' to 'Celsius'
- defining a different format for simple types, such as Color and RelTime
- managing expectations for +/- INF in a target platform

For further information, refer to the "Type Override Example" in the "Developer Guide" chapter of this document.

Debugging errors (Schema History Debug)

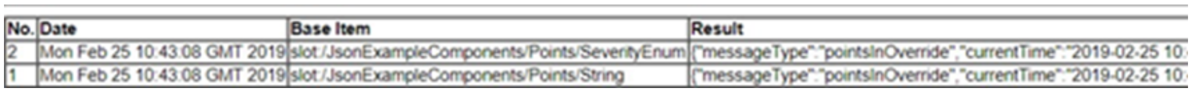
When output updates rapidly, such as when a link calls a generate JSON action in quick succession or a relative schema quickly changes output once per base item, it may be useful to view the most recent output history. This task describes how to view the output history.

Prerequisites: You are viewing the Property Sheet for the schema.

Step 1 Do one of the following:

- Click the **Output History** button to the right of the `Output` property on the schema.
- Expand the schema's **Config→Debug** folder, right-click the **Schema Output History Debug** slot, and click **Views→Spy Remote**.

The **Schema Output History Debug** view opens.



No.	Date	Base Item	Result
2	Mon Feb 25 10:43:08 GMT 2019	slot:/JsonExampleComponents/Points/SeverityEnum	{"messageType":"pointsInOverride","currentTime":"2019-02-25 10:
1	Mon Feb 25 10:43:08 GMT 2019	slot:/JsonExampleComponents/Points/String	{"messageType":"pointsInOverride","currentTime":"2019-02-25 10:

The History Size allows you to store more but be careful not to fill memory with JSON strings.

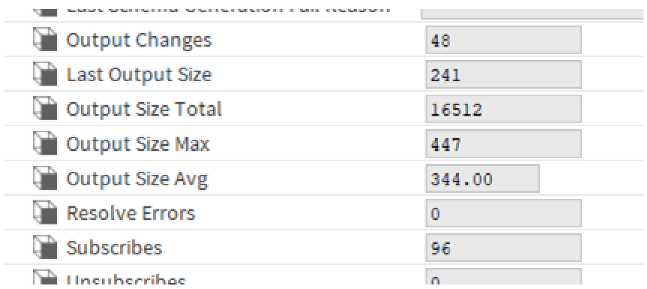
Step 2 To configure the amount of debug data stored in the station, expand the **Schema Output History Debug** folder and configure the `History Max Size` property.

It is a good idea to reduce this value once you have finished debugging.

JSON schema metrics

Metrics expose schema generation, query execution and CoV subscription data. You can log this information, link it, and use it to generate alarms.

Figure 9 Schema metrics



Output Changes	48
Last Output Size	241
Output Size Total	16512
Output Size Max	447
Output Size Avg	344.00
Resolve Errors	0
Subscribes	96
Unsubscribes	0

These help with sizing and provisioning capacity from a cloud platform by estimating the traffic a station is likely to generate with a given JSON schema. They may also assist in identifying performance problems. Debugging can be assisted by using the reset action.

The metrics provide three categories of performance information: query performance, generate performance, and subscription performance.

Queries	Generation	Subscription
Query Folder Executions	Request Schema Generations	Subscribes
Individual Query Executions	Schema Generations	Unsubscribes
Query Fails	Schema Generation Fails	Subscription Events
Last Query Fail Reason	Last Schema Generation Fail Reason	Subscription Events Ignored
Last Query Execution Millis	Output Changes	Cache Hits
Query Execution Millis Total	Last Output Size	Cache Misses
Query Execution Millis Max	Output Size Total	
Query Execution Millis Avg	Output Size Max	
	Output Size Avg	
	Resolve Errors	

Schema construction

Setting up a schema involves binding station data to JSON entities.

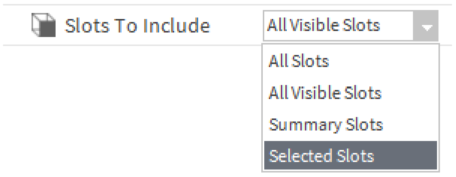
Binding configuration, about binding

Bound properties, objects and arrays are JSON entities, which can use the current values of an ord target to render their values. Fixed variants do not support binding.

Slot selection

When picking a bound object or array, you may choose which slots from the target to include in the resultant JSON container. Currently the options are:

Figure 10 Slots to include



- All slots
- All visible slots (hidden slots excluded)
- Summary slots—only those with a summary flag
- Selected slots—manually-selected slots from a list

Target types

NOTE: When choosing the bind target for a binding you could select any type of slot, from devices to control points to out slots to simple values, there are no restrictions.

Bound arrays and objects output the value of each of the selected slots (refer to Slot Selection, page 21). The default behaviour for each encountered slot type is as follows:

Selection	Output
Strings	The string value is unchanged
Booleans	A JSON Boolean

Selection	Output
Integer/Long	A JSON number
Double and float decimals	A JSON number rounded to use the schema's decimal places config
Enum value	A JSON String that represents the Enum value
AbsTime	A String representation of the date formatted as per the schema config
Control Point	A JSON String, Numeric, Boolean, or Enum to represent the out slot's value
Status Value	A JSON String, Numeric, Boolean, or Enum to represent the value
Status	A JSON string to represent the value, for example, `{ok}`
Anything else	The string representation of the value as returned from the framework. This is often the type display name.

NOTE: Bound objects and arrays do not recurse. Only direct child slots are included. These behaviours make a few assumptions about the most-expected case, for example, excluding the status string from certain types. Program overrides may override all these behaviours.

Naming

For binding results you may choose what the key is in the key/value pair:

Selection	Output
Display Name	The name of the bound property, object or array
Target Name	The name of the ord target
Target Display Name	The display name of the ord target
Target Parent Name	The name of the ord target's parent
Target Path	The absolute path of the target from the root of the component tree

TIP: You may use a **`Tag`** property with the name `n:name` to include point names. This property inserts a single tag value from the bound component in the output. If the **`SearchParents`** property is `true`, the framework searches up the hierarchy for the closest component with a matching tag id (if the tag not found on binding target.

Entities

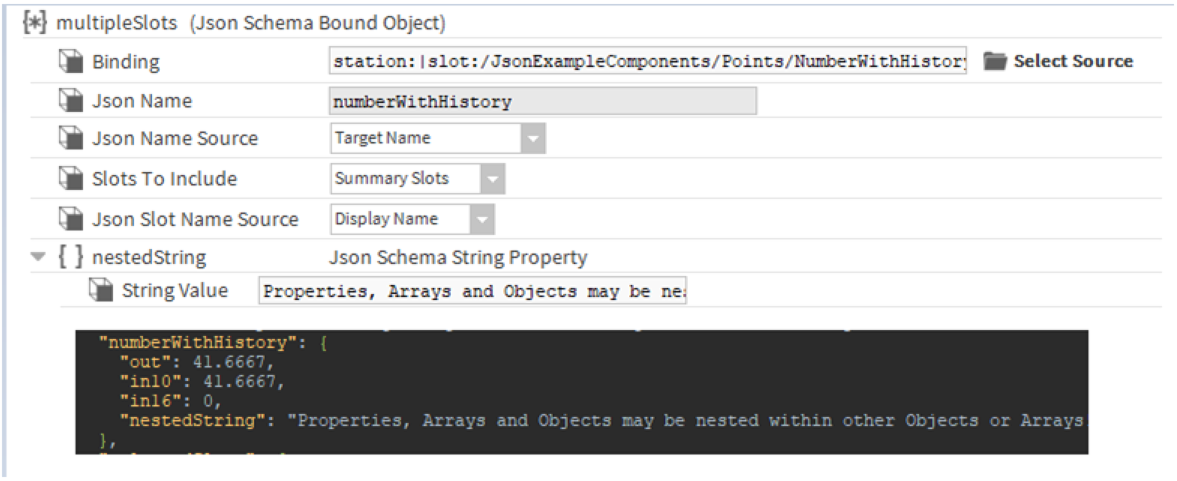
Entities are objects, arrays, properties and bound properties.

Objects

Objects are entities used to create containers in the JSON message and identify slots in a target ord.

- A JSON schema **`Object`** inserts into the schema an empty named container (**`{ }`**) for holding other schema entities.
- A **`BoundObject`** is a named JSON object whose child name and value pairs are the slots within a target ord.

Figure 11 A Json Schema Bound Object



multipleSlots (Json Schema Bound Object)

Binding	station:|slot:/JsonExampleComponents/Points/NumberWithHistor
Json Name	numberWithHistory
Json Name Source	Target Name
Slots To Include	Summary Slots
Json Slot Name Source	Display Name
nestedString	Json Schema String Property
String Value	Properties, Arrays and Objects may be ne

```
"numberWithHistory": {
  "out": 41.6667,
  "in10": 41.6667,
  "in16": 0,
  "nestedString": "Properties, Arrays and Objects may be nested within other Objects or Arrays
},
```

Arrays

Arrays contain a list of values. They do not include names.

- A JSON schema **`Array`** inserts into a schema an empty named container ([ ]) for the purpose of holding other schema entities.
- A JSON schema **`BoundArray`** is a named JSON object that renders values as a list.

Fixed properties

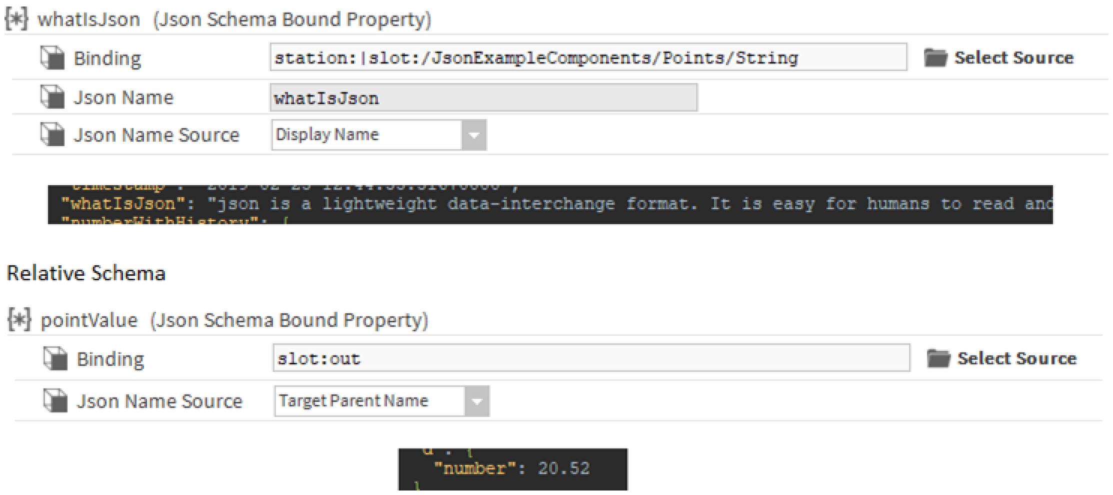
Fixed **`Properties`** are hard-coded name and value pairs, which you always want to appear as constants in the JSON string. You can link to these if the value is expected to vary. The next generation event, triggered by a CoV on a bound entity or by the invocation of the `Generate` action, includes the current value. A change in the value of any fixed property does not trigger a CoV generation event in the same way that a bound equivalent does.

- A **`FixedString`** property inserts a string value.
- A **`FixedNumeric`** property inserts a numeric value.
- A **`FixedBoolean`** property inserts a Boolean value.

Bound properties

A bound property inserts the current value of the object specified in the binding.

Figure 12 Bound properties



whatIsJson (Json Schema Bound Property)

Binding	station:|slot:/JsonExampleComponents/Points/String
Json Name	whatIsJson
Json Name Source	Display Name

```
"whatIsJson": "json is a lightweight data-interchange format. It is easy for humans to read and
```

Relative Schema

pointValue (Json Schema Bound Property)

Binding	slot:out
Json Name Source	Target Parent Name

```
  "number": 20.52
```

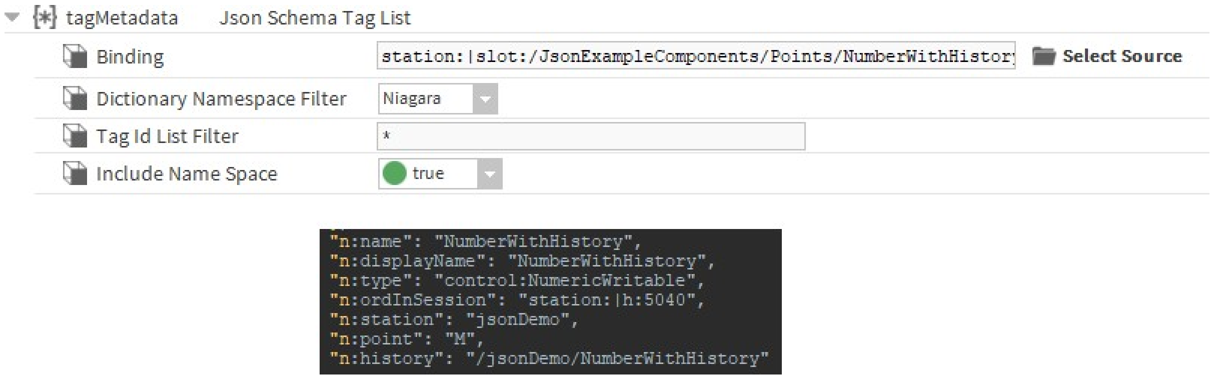
`BoundProperties` include:

- A **`BoundCSVProperty`** is a named JSON string that renders child slots as a string, comma–separated list with no surrounding **`[ ]`** or **`{ }`**.
- A **`Tag`** property is a list of name and value properties based upon selected tags found on a binding target. If the tag is not found on the binding target, and**`SearchParents`** is `true`, the framework searches up the hierarchy for the closest component with a matching tag id.
- A **`TagList`** is a list of name and value property pairs that are based upon selected tags found on a binding target. A comma-separated list specified in the **`Tag Id List Filter`** property can limit the tags to be included in the output. Example: `n:name`, `n:type` or `*` for all. If **`Include Namespace`** is `true`, the tag dictionary prefix is added to the key (for example, the hs: is added to hvac to give: `hs:hvac`).

  NOTE:

  `Facet` and **`Tag`** properties are not bound like the other bindings, in that changes of value do not prompt schema generation. The current value is retrieved from the station when the schema generates.

Figure 13 Json Schema Tag List



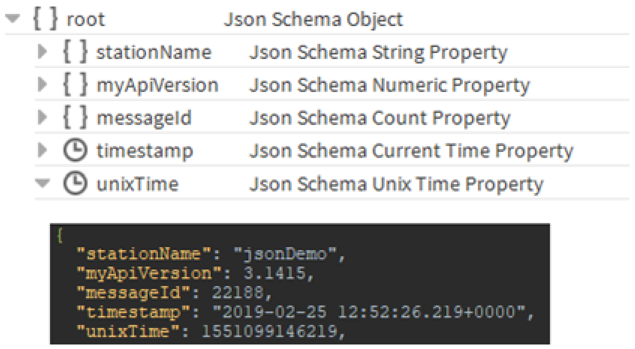
tagMetadata Json Schema Tag List

Binding	station:|slot:/JsonExampleComponents/Points/NumberWithHistor
Dictionary Namespace Filter	Niagara
Tag Id List Filter	*
Include Name Space	true

```
"n:name": "NumberWithHistory",
"n:displayName": "NumberWithHistory",
"n:type": "control:NumericWritable",
"n:ordInSession": "station:|h:5040",
"n:station": "jsonDemo",
"n:point": "M",
"n:history": "/jsonDemo/NumberWithHistory"
```

- A **`Facet`** property inserts a single facet value from a bound component into the schema output, for example, the units of the current point.
- A **`FacetList`** inserts a list of name and value facet properties based on a comma-separated list or * for all. Add facet keys as follows: `units`, `mix`, `max`
- **`Message`** properties

Figure 14 Message properties



{ } root	Json Schema Object
{ } stationName	Json Schema String Property
{ } myApiVersion	Json Schema Numeric Property
{ } messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
unixTime	Json Schema Unix Time Property

```
{
  "stationName": "jsonDemo",
  "myApiVersion": 3.1415,
  "messageId": 22188,
  "timestamp": "2019-02-25 12:52:26.219+0000",
  "unixTime": 1551099146219,
```

- A **`Count`** property is a named numeric value, which increments by 1 on each schema generation. Could be used for message IDs.
- A **`CurrentTime`** property inserts the current time as set up in the **Config** folder's **`Time Format`** property.
- **`UnixTime`** property inserts the current time in Unix time as seconds from January 1, 1970.

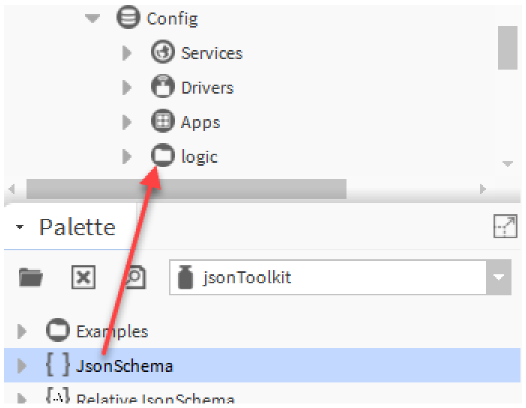
Creating a regular schema

You construct a schema by placing objects from the **`jsonToolkit`** palette in a **`JsonSchema`**.

Prerequisites: The station is running.

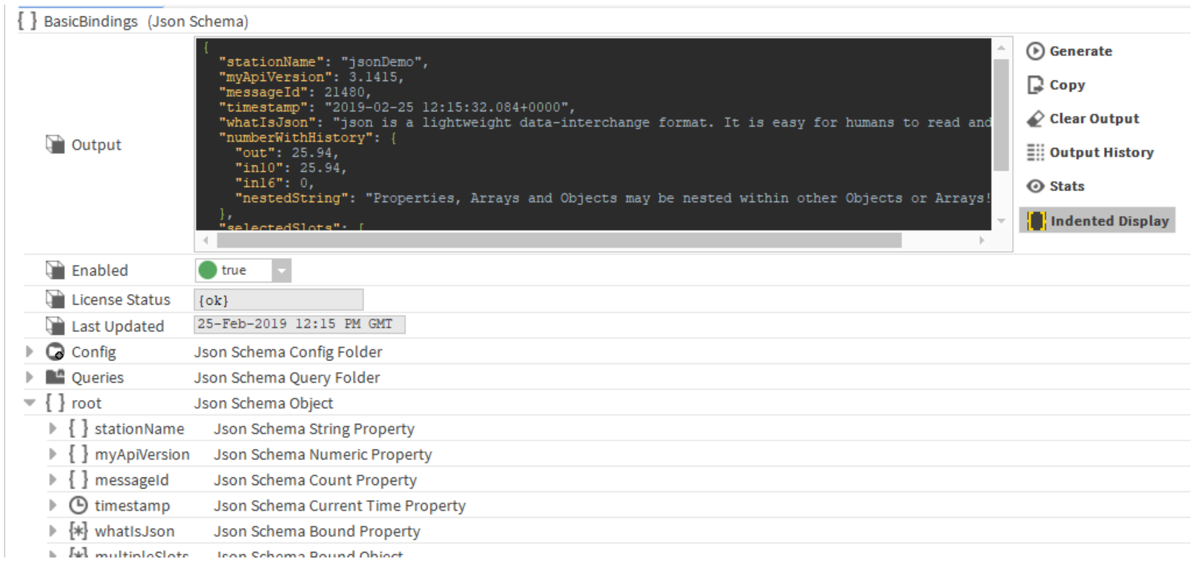
Step 1 Open the **`jsonToolkit`** palette from the **`workbench`** palette sidebar.

Step 2 Drag a **`JsonSchema`** to the **Config** node or another desired folder location and type a unique name for the schema when prompted.



Step 3 To view the schema **Property Sheet**, double-click the schema glyph in the Nav tree.

The **Property Sheet** opens.


{ } BasicBindings (Json Schema)

Output

```
{
  "stationName": "jsonDemo",
  "myApiVersion": 3.1415,
  "messageId": 21480,
  "timestamp": "2019-02-25 12:15:32.084+0000",
  "whatIsJson": "json is a lightweight data-interchange format. It is easy for humans to read and
  "numberWithHistory": {
    "out": 25.94,
    "in10": 25.94,
    "in16": 0,
    "nestedString": "Properties, Arrays and Objects may be nested within other Objects or Arrays!
  },
```

Enabled	true
License Status	{ok}
Last Updated	25-Feb-2019 12:15 PM GMT
Config	Json Schema Config Folder
Queries	Json Schema Query Folder
{ } root	Json Schema Object
{ } stationName	Json Schema String Property
{ } myApiVersion	Json Schema Numeric Property
{ } messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property

When you initially view the **Property Sheet** for a new schema, the **`Output`** property is an empty black box. JSON strings appear here when you generate output.

Step 4 In the **Property Sheet** view, ensure that the **`Enabled`** property is set to `true`.

Setting **`Enabled`** to `false` prevents the generation of output, the execution of queries and the subscription to bound values.

Step 5 To begin setting up the message, expand the **Objects** folder in the palette, drag an **Object** to the **Property Sheet** and name it, for example, `root`.

Braces { } represent this object in the **`Output`**. This single top-level object serves as the JSON parent container for other JSON objects that make up the message. Each JSON object requires a pair of braces ({ }) and arrays require brackets ([ ]).

Step 6 Drag an object, array, or property from the palette to the **Property Sheet** root container.

Some objects may be simple and other objects may yield the more complex results of Niagara bql queries. The objects that you choose to add depend on your unique requirements.

- Empty braces { } icons represent a JSON object. A bound object is a named object whose child name and value pairs are the slots within an ord target.
- Bracket [ ] icons represent an array, which is an empty named container of other schema entities. A bound array is a named object that renders values as a list.
- Other icons represent properties, which may be fixed or bound.

Step 7 To update the schema **`Output`** based on the current values retrieved from the station, click **Generate**, or right-click the schema name and click **Actions→Generate Json**.

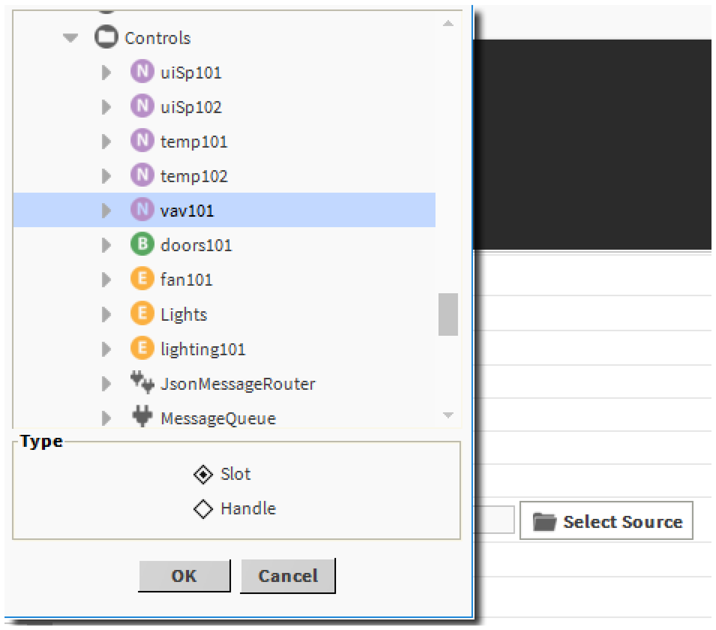
This action causes a regular schema to re-evaluate any query and populate the **`Output`** box with JSON.

Step 8 To set up some actual station data, drag in a **`BoundObject`**, name it appropriately, expand the bound object and click the **`Select Source`** finder to the right of the **`Binding`** property.

This object requires a binding similar to the way components on Px pages require bindings to actual points in a station.

The **Choose component/slot for JSON** window opens.

Step 9 Navigate to and select the source component, click **OK** and then click **Save**.



When choosing the target for a binding, you can select any type of slot, from devices to control points to out slots to simple values. There is no restriction. Due to subscription, saving the schema also generates the JSON message (output).

If your logic contains one or more points whose values change periodically, the schema generates a new JSON message every time a CoV occurs. If the schema is connected to MQTT, the schema can send each new message to the web.

Step 10 To change the **`Json Name`** (a read-only property) to the name of the bound input slot on your **Wire Sheet**, change **`Json Name Source`** property to `Target Name`, and, from the **`Slots To Include`** property, choose `Summary Slots`.

To include specific slots, use the **`Slots to Include`** properties, identify and pick individual slots for more fine-grained control.

You may link the output slot to an **`EngineCycleMessageQueue`**, if required, which buffers output sent to the onward transport. These could be MQTT or HTTP depending on the onward linked point.

Relative schema construction

A relative schema enables the scaling of JSON payload generation and much faster engineering than absolute object binding.

The type of schema discussed thus far uses only absolute ords. In situations with many points, absolute ords could limit scalability. One schema per point or device would not be an efficient way to proceed. In the same way that relative ords in graphics enable efficient engineering with the Niagara framework, a relative schema provides easier scaling for an existing station and also requires no changes to the JSON when adding new components and points.

A base query feeds base components to the schema, which the query resolves against the schema one at a time. In this manner it is possible to select, for example, all BACnet points in a station and output their name, status and present value for export to the cloud. If an engineer adds an extra device to the BACnet network in the future, the base query can automatically include it in the data exposed by the station, if the query allows.

Alone, a relative schema can select data to export or, when combined with an Export Marker, it can send only recent history or publish only when a set tolerance value is exceeded. Further still, you can move points between schema based on their status. You might have one schema that sends verbose point data and another with simple latest values once you add an export marker.

NOTE: A best practice is to limit the scope of the base query to a subset of points in the station and limit the frequency of JSON message generation. Very frequent payload generation could degrade station performance.

Base query examples

This base query would return all the overridden points beneath the **Drivers** container:

```
slot:/Drivers|bql:select * from control:ControlPoint where status.overridden = 'true'
```

This query returns all points with the Haystack marker tag, hvac:

```
slot:/|neql:n:point and hs:hvac
```

The base query's **`Publish Interval`** causes the base query to be re-executed periodically and triggers a complete publish output (of every returned component) at the interval selected.

Invoking the `Generate` action on a relative schema evaluates the base query again.

CAUTION: Do not include the schema output itself in the base query. This will quickly consume available Java heap memory!

Export markers

Export markers on points and other entities set up efficient data retrieval.

Export marker: selecting control points

You select control points to export using:

- Absolute ord bindings in a standard schema
- Bql or neql in a relative schema
- by adding an export marker to a component.

JSON export markers offer several benefits beyond just marking points to include in a relative schema. For example, you can use it to limit the export of alarm or history data related only to points with an export marker present. It can also store a unique identifier supplied by a third party platform. This can allow you to differentiate among registered points with an ID and unregistered points without an ID. An example use case is sending different payloads prior to registration including more detailed information (units, min/max, descriptive tags) than should be sent upon every change of value. When applied to a numeric point an export marker introduces a **`CovTolerance`** property to reduce unwanted updates from the station if a value changes only slightly. You can also use an export marker with incoming JSON payloads.

Here are some examples of relative schema configuration.

- Base Query: `station:|slot:/|bql:select * from jsonToolkit:JsonExportMarker`
- Example bound property binding ord: `slot:..` (References the parent of the **`JsonExportMarker`** base)

Export marker filters

Both filters below have a `Send Since` action, which allows alarms or histories since a given date to be exported. This feature might be useful following network disruption or during initial commissioning of a system.

The `Send Since` action allows you to specify a start time. The linked schema considers only records stored on or since this time for output.

Two common filter properties are:

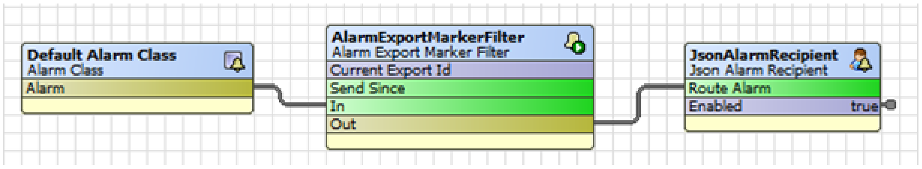
- **`Current Export Id`** includes a description of the export marker if it is linked to a fixed string in the schema.
- **`Count`** reports how many export-marked points were processed in the last invocation. It resets when the station restarts.

Alarm export marker filter

This filter selects specific alarms a station generates before the station passes them to a recipient. Typically, the recipient would be a JSON alarm recipient, but it could be SNMP, BACnet, etc. with the source alarm class linked to the In slot of the filter.

In the context of alarming the filtering occurs normally on alarms passed from the alarm class as they are generated.

Figure 15 Wire Sheet showing the use of an export marker filter



Filter mode	Outputs alarms
Marked With Id	If the source has an export marker present, with Id set
Marked	If the source has an export marker present
Pass All	All alarms
Block All	No alarms

In the context of alarming, the filtering occurs normally on alarms passed from the alarm class as they are generated.

The `Send Since` action queries the alarm database and passes existing records in to this filter (inclusive of the supplied timestamp) so that they can be checked for a suitable export marker and then passed to the receiving schema as required to create a new record for each alarm. The timestamp, being in the past, should help identify when this mode is active.

NOTE:

To prevent an accidental data deluge, `Send Since` does not function if the filter is in Pass All mode. A bql query on the alarm database could be used if this is a requirement.

History export marker filter

This filter exports history data for points with an export marker.

The filter overlaps somewhat with the relative history query, which can select history for points using many different selection criteria, or an appropriate base query may also be used to generate history for each export marked point. The **`HistoryExportMarkerFilter`** allows updating of the timestamp stored on each export marker so that only recent history records are sent to the remote system (typically, records added since the last export).

The schema nested under the filter determines the payload format. To complete the export, link the output from that schema to a target transport point.

If one does not exist already, the **`HistoryExportMarkerFilter`** adds a new query to the **`Queries`** folder of the schema. This query needs to be referenced by a **`BoundQueryResult`**.

In the event that an export-marked point has more than one history extension beneath it, the schema exports each extension in turn.

In most cases, it is likely the **`Current Export Id`** property needs to be linked into the schema output to provide identifying information, or even the query used to select data may be included if the target system could infer useful data from it.

NOTE: Because the export marker relies on being added to a local control point in the station, it is not possible to match histories imported over BACnet or NiagaraNetwork using this method. Use a relative schema instead.

Use the `Send Since Last Export` action to send only unsent history data using the timestamp stored on each export marker.

These are some important filter properties:

- **`History Export Filter`** is the schema that produces the output.
- **`Current Query`** identifies the query fed into the schema below. The first query in the **Queries** folder is linked on start, does not have to be the only query, and is output first by the schema.
- **`Columns`** sets up comma-separated values, for example, timestamp, value, status.
- **`Update Send Since Time`** determines if the schema updates most recent send time when the schema generates data and enables sending only changed records on the next run. If `true`, every time the schema exports history it updates the timestamp stored on each export marker.

Queries

Queries search the station database for the data to include in a schema.

Query folder

The **Queries** folder of a JSON schema stores queries whose results are available to be used in the schema. This allows JSON content to be generated from the results of bql or neql queries. For example, to name just a few, you can generate a report of overridden points, active alarms, or history logs for a given point.

`Query Interval` is an important property of the queries folder. It determines how often queries execute, and, therefore, how up-to-date any data exported by the schema will be when an update strategy of CoV is used.

NOTE: If multiple queries exist, the station runs each query in parallel each time the schema executes.

Queries do not execute each time a schema generates in change-of-value mode, otherwise a query could run every time a point value changes, which could have a negative impact on the performance of the control strategy running in a station. Instead, a **`BoundQueryResult`** caches the results and adds them to the schema.

Schemata in on-demand mode and relative schemata do execute each query every time a schema generates.

It is possible to manually invoke query execution using the `Execute Queries` action of the schema, which could also be linked to some appropriate logic to trigger execution when needed.

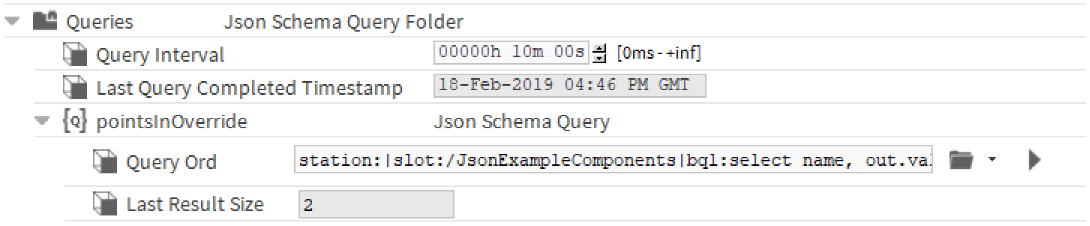
IMPORTANT:

When executing queries against your station, bear in mind the potential performance implications of running queries frequently. To reduce the scope of the query, focus the first part of the ord to the location where the data are likely to be found, or by using the stop keyword to prevent depth recursion.

Query

You add queries below the **Queries** folder found at the top level of the schema.

Figure 16 Query properties



Queries	Json Schema Query Folder
Query Interval	00000h 10m 00s [0ms-+inf]
Last Query Completed Timestamp	18-Feb-2019 04:46 PM GMT
pointsInOverride	Json Schema Query
Query Ord	station:|slot:/JsonExampleComponents|bql:select name, out.va
Last Result Size	2

A query can be any valid transform, neql or bql statement which returns a BITable.

Here are some useful examples to include in a schema:

Data to return	Query
BACnet points currently in {override} status	`slot:/Drivers/BacnetNetwork|bql:select name, out.value from control:ControlPoint where status.overridden = 'true'`
History records	`history:/Newhaven/waveHeight|bql:select timestamp, value`
Output from a series transform	`station:|transform:slot:/VelocityServlet/lineChart/TransformGraph`
Alarm database contents	`alarm:|bql:select timestamp, uuid, ackState, source as 'origin'` **NOTE:** You may rename the columns using the 'as' keyword, which the resultant JSON reflects.

Relative history query

Used in conjunction with a relative schema, the query **`Pattern Property`** pre–appends the current base item to a bql query, so that query data can be included in the payload for a given set of points or devices:

```
%baseHistoryOrd%?period=today|bql:select timestamp, value
```

You may use this in conjunction with a base query that returns a HistoryConfig or a HistoryExt (or the parent of these types):

```
station:|slot:/JsonExampleComponents|bql:select * from history:HistoryConfig
```

Consider the effect on performance that running many queries on an embedded controller may have.

BoundQueryResult

Once you define a query, use the **`BoundQueryResult`** to determine where and how to insert the results into the payload.

You can mix query results, such as bound properties or other query results with all other schema member types in the same payload. For example, if required by the target platform, you could construct a floor summary with historical data and current alarms.

The JSON Toolkit provides various output formats as the following examples demonstrate, and a developer can create new output formats.

The following examples use two columns for the sake of brevity. You may add more columns.

You can format the timestamp returned by a query using the format options in the schema's **Config** folder.

REMEMBER:
Executing a bql query does not trigger subscription of the component in question. The values used are the last values known to the station.

Example	JSON
Row array with header	"data": [["timestamp", "value"], ["2019-02-07 23:27:42.116+0000", 45], ["2019-02-07 23:28:03.157+0000", 15], ["2019-02-07 23:28:24.197+0000", 85], ["2019-02-07 23:28:45.222+0000", 55], ["2019-02-07 23:29:06.247+0000", 25]]
Row array	"data": [["2019-02-07 23:27:42.116+0000", 45], ["2019-02-07 23:28:03.157+0000", 15], ["2019-02-07 23:28:24.197+0000", 85], ["2019-02-07 23:28:45.222+0000", 55], ["2019-02-07 23:29:06.247+0000", 25]]

Objects array	"data": [
	{
	"timestamp": "2019-02-07 23:27:42.116+0000",
	"value": 45
	},
	{
	"timestamp": "2019-02-07 23:28:03.157+0000",
	"value": 15
	},
	{
	"timestamp": "2019-02-07 23:28:24.197+0000",
	"value": 85
	},
	{
	"timestamp": "2019-02-07 23:28:45.222+0000",
	"value": 55
	},
	{
	"timestamp": "2019-02-07 23:29:06.247+0000",
	"value": 25
	}
	]

Named objects (The first column is assumed to represent the object name.)	"data": [
	"2019-02-07 23:27:42.116+0000": {
	"value": 45
	},
	"2019-02-07 23:28:03.157+0000": {

Example	JSON

	`"value": 15`
	`},`
	`"2019-02-07 23:28:24.197+0000": {`
	`"value": 85`
	`},`
	`"2019-02-07 23:28:45.222+0000": {`
	
	`"value": 55`
	`},`
	`"2019-02-07 23:29:06.247+0000": {`
	`"value": 25`
	`}`
	`]`

Column array with header	`"data": [`
	`[`
	`"timestamp",`
	`"2019-02-07 23:27:42.116+0000",`
	`"2019-02-07 23:28:03.157+0000",`
	`"2019-02-07 23:28:24.197+0000",`
	`"2019-02-07 23:28:45.222+0000",`
	`"2019-02-07 23:29:06.247+0000"`
	`], [`
	`"value", 45,`
	`15,`
	`85,`
	`55,`
	`25`
	`]`
	`]`

Column array	`"data": [`
	`[`
	`"2019-02-07 23:27:42.116+0000",`
	`"2019-02-07 23:28:03.157+0000",`
	`"2019-02-07 23:28:24.197+0000",`
	`"2019-02-07 23:28:45.222+0000",`
	`"2019-02-07 23:29:06.247+0000"`
	`], [`
	`45,`
	`15,`
	`85,`
	`55,`
	`25`
	`]`
	`]`

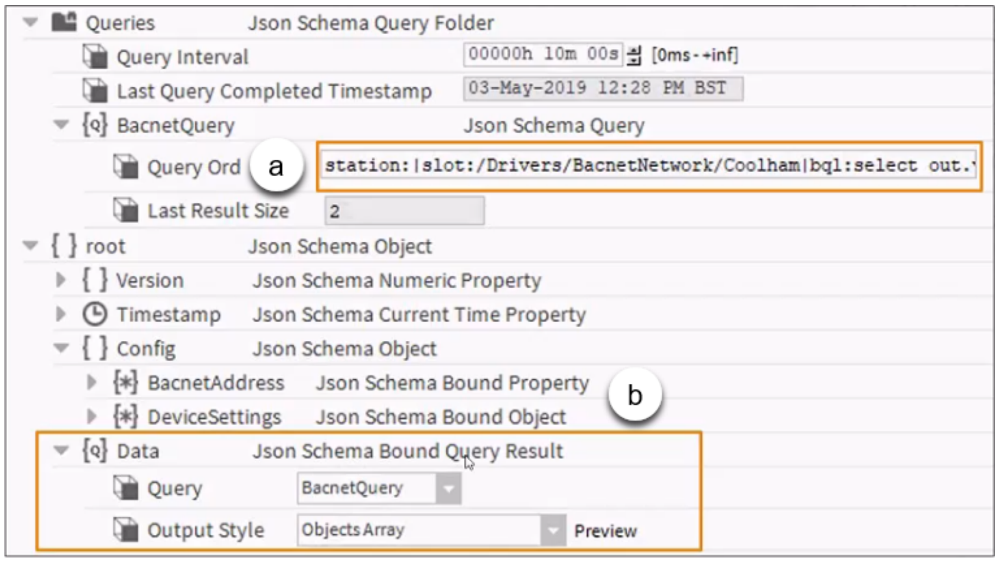
Single column array
 	
NOTE: The query used to populate the BoundQueryResult should only return one column. It would be wasteful to select data that are not expected to emerge in the payload.
	`"data": [`
	`45,`
	`15,`
	`85,`
	`55,`
	`25`
	`]`

Tuning	You may use the hidden query folder property **`queriesMaxExecutionTime`** to increase the amount of time granted to complete all the queries during each cycle. Failure to complete in this time causes schema generation to fail.

Setting up queries

In addition to the binding queries, which set up a single query bql, neql or ord, you can add additional queries to a **`Queries`** folder. The schema turns the queries in this folder into a string.

Step 1 Create a regular schema.



```
Queries                          Json Schema Query Folder
  Query Interval                 00000h 10m 00s [0ms-+inf]
  Last Query Completed Timestamp 03-May-2019 12:28 PM BST
  BacnetQuery                    Json Schema Query
    Query Ord      a   station:|slot:/Drivers/BacnetNetwork/Coolham|bql:select out.
    Last Result Size   2
root                             Json Schema Object
  Version                        Json Schema Numeric Property
  Timestamp                      Json Schema Current Time Property
  Config                         Json Schema Object
    BacnetAddress                Json Schema Bound Property     b
    DeviceSettings               Json Schema Bound Object
  Data                           Json Schema Bound Query Result
    Query                        BacnetQuery
    Output Style                 Objects Array   Preview
```

The example above uses the points of a BACnet device. This JSON configuration includes the `Queries` folder and the `root` object container for the schema.

a. Identifies the regular queries that define the source of the data for binding. In this example, the query uses bql to identify the data.

b. Identifies a query that can become a JSON string. The query result injects the query referenced from the Queries folder into the point in the schema output. You can nest these queries anywhere within your JSON message.

By default, each schema includes a **Queries** folder, which comes with two properties: **`Query Interval`** (to configure how frequently to execute the query), and **`Last Query Completed Timestamp`**.

Step 2 To configure the **`Query Interval`**, right-click the **Queries** folder, click **Views→AX Property Sheet**, configure the interval, and click **Save**.

Step 3 To add an *ad hoc* query to the schema, expand the **Query** node in the palette, drag a **Query** from the palette to the **Queries** folder in the schema, double-click the **Query**, enter the **`Query Ord`**, and click **Save**.

For simplicity, the example **`Queries`** folder contains a single query. It could contain additional queries.

A above identifies the ord for the single *ad hoc* query (BacnetQuery): `station:|slot:/Drivers/BacnetNetwork/MyName|bql:select name, proxyExt.objectId, out.value AS 'v', status from control:ControlPoint`

This query searches a particular BACnet device for the name, object ID, current value and status of all points under the device. The **`Last Result Size`** property indicates that the query finds two points.

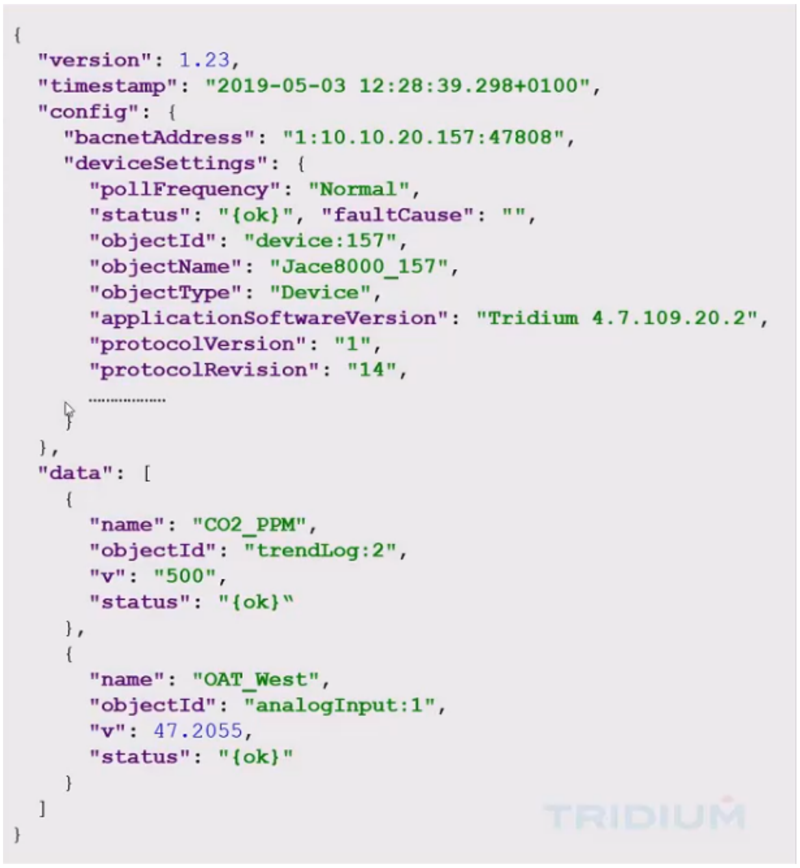
Step 4 To create a bound query result, expand the **Query** node in the palette and drag a **BoundQueryResult** from the palette to the root object in the schema.

In the example, the bound query result (identified by the second box) references the query (BacnetQuery) and defines the **`Output Style`** to render the query in.

Step 5 To update the payload message, click the **Generate** button.

The result of running the example query looks like this:

Figure 17 Device connectivity JSON payload



```
{
  "version": 1.23,
  "timestamp": "2019-05-03 12:28:39.298+0100",
  "config": {
    "bacnetAddress": "1:10.10.20.157:47808",
    "deviceSettings": {
      "pollFrequency": "Normal",
      "status": "{ok}", "faultCause": "",
      "objectId": "device:157",
      "objectName": "Jace8000_157",
      "objectType": "Device",
      "applicationSoftwareVersion": "Tridium 4.7.109.20.2",
      "protocolVersion": "1",
      "protocolRevision": "14",
      .................
    }
  },
  "data": [
    {
      "name": "CO2_PPM",
      "objectId": "trendLog:2",
      "v": "500",
      "status": "{ok}"
    },
    {
      "name": "OAT_West",
      "objectId": "analogInput:1",
      "v": 47.2055,
      "status": "{ok}"
    }
  ]
}
```

The first group of name and value pairs reports the result of the main binding query (under config). The `data` block at the bottom shows the result of the *ad hoc* query in the **Queries** folder. The data block displays as an object array identified by the square brackets. The array contains one object per BACnet point, in this case two objects, each inside a pair of braces.

This example could have used a relative schema. Which one to use depends on your requirements. Does your API need all data in a single JSON message or does it require one message per point? This procedure does not subscribe to the component model. It runs a bql query to populate the BITable and encodes that data. The power of bql to select data feeds into the input to the schema the same as you could feed a series transform into this schema, query the historical alarm data, or query history data.

This type of query configuration does not have to be done with device points. By "query" in this context, we mean anything that returns a BITable so you could use a transform ord, bql on the history space or neql on the component space. Any time you have something you can feed to the ReportService you can encode and output it with a schema.

Alarms

The **`JsonAlarmRecipient`** exports alarms using the recipient's schema.

AlarmRecipient

Linking the alarm topic of an alarm class into the route action of a **`JsonAlarmRecipient`** triggers the generation of a new payload each time the alarm class receives an alarm.

The **`JsonAlarmRecipient`** comes with a nested schema whose payload output depends on the alarms passed through from the parent recipient.

Queries, bound objects and arrays, and/or properties can include present value data from the station in the payload.

There are, however, some alarm-specific data types you can include, notably the properties from a Niagara Alarm Record: BAlarmRecord

By including the unique identifier in an outgoing message, an inbound payload can acknowledge alarms.

Alarm Record Property

Only the **`JsonAlarmRecipient`**'s schema supports these alarm-related properties. Adding each of these to the schema allows inclusion of the selected alarm property in the output.

For example, the **`sourceState`**, **`uuid`**, alarmClass etc. As with other schema properties the name is determined by renaming the property, for example **`AlarmRecordProperty`** becomes **`timestamp`**.

BFormat Property

This property defines the alarm data to be extracted from the Niagara alarm database. For example, if an engineer used the **`Metadata`** property of an AlarmExt to record the location of a point in the building, this could be fetched using alarmData.location to include in the payload.

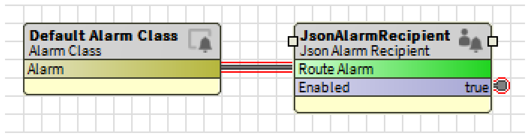
Exporting alarm records to the JsonAlarmRecipient

This component comes with a nested schema whose payload output depends on the alarms passed through from the parent recipient.

You may include queries, bound objects or arrays, and properties to return a station's present value data in the payload. You may also include some specific alarm data types, notably the properties from the alarm record: BAlarmRecord.

Step 1 Drag the JsonAlarmRecipient to the **Wire Sheet**.

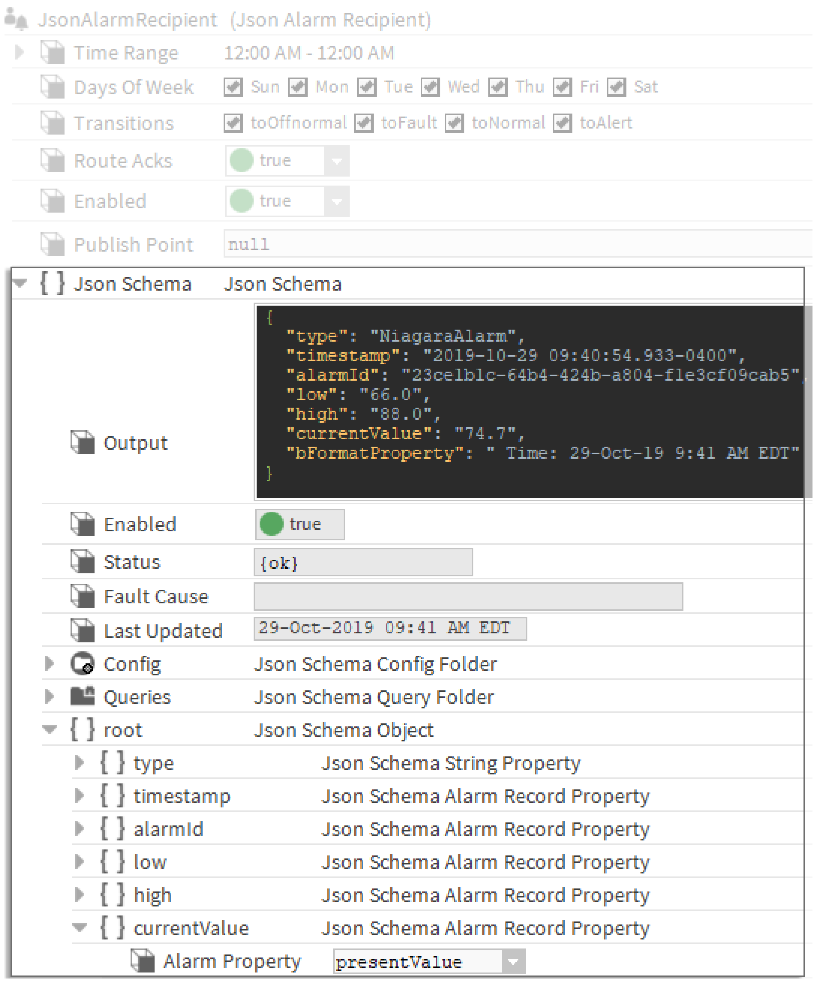
Step 2 Connect the Alarm Class to the Route Alarm action on the recipient.



Linking the alarm class to the route action of a **`JsonAlarmRecipient`** component triggers the generation of a new JSON payload each time the recipient receives an alarm from the alarm class.

Step 3 Add an **`AlarmRecordProperty`** component to the schema and select one or more properties.

]



JsonAlarmRecipient (Json Alarm Recipient)
Time Range 12:00 AM - 12:00 AM
Days Of Week ✓ Sun ✓ Mon ✓ Tue ✓ Wed ✓ Thu ✓ Fri ✓ Sat
Transitions ✓ toOffnormal ✓ toFault ✓ toNormal ✓ toAlert
Route Acks true
Enabled true
Publish Point null
{ } Json Schema Json Schema

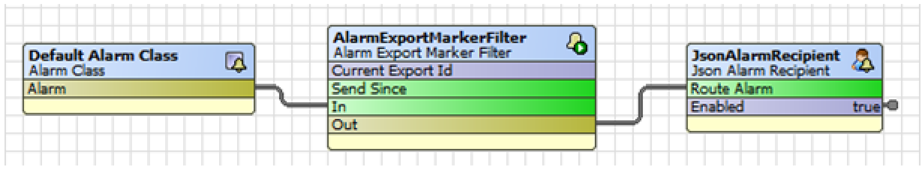
Output

```
{
  "type": "NiagaraAlarm",
  "timestamp": "2019-10-29 09:40:54.933-0400",
  "alarmId": "23ce1b1c-64b4-424b-a804-f1e3cf09cab5",
  "low": "66.0",
  "high": "88.0",
  "currentValue": "74.7",
  "bFormatProperty": " Time: 29-Oct-19 9:41 AM EDT"
}
```

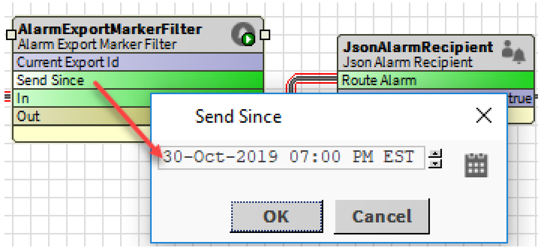
Enabled true
Status {ok}
Fault Cause
Last Updated 29-Oct-2019 09:41 AM EDT
Config Json Schema Config Folder
Queries Json Schema Query Folder
{ } root Json Schema Object
{ } type Json Schema String Property
{ } timestamp Json Schema Alarm Record Property
{ } alarmId Json Schema Alarm Record Property
{ } low Json Schema Alarm Record Property
{ } high Json Schema Alarm Record Property
{ } currentValue Json Schema Alarm Record Property
Alarm Property presentValue

Each property you add to the schema can include selected alarm data in the output, such as the sourceState, uuid, alarmClass etc. As with other JSON schema properties, you can rename the property; for example “AlarmRecordProperty” can be renamed to “current value”, as shown above.

Step 4 To filter out unwanted alarms before sending data to the alarm recipient, add the **AlarmExport-MarkerFilter** to the **Wire Sheet** and connect it as shown below.



Normal filtering occurs on alarms passed from the alarm class to the recipient as the station generates the alarms.



The **Send Since** action queries the alarm database and passes existing records to the filter (including the supplied timestamp). The system checks the records for a suitable ExportMarker, and passes them to the receiving JsonSchema to create a new record for each alarm. Since the timestamp is in the past, the filter should be able to identify when its mode was active.

NOTE:

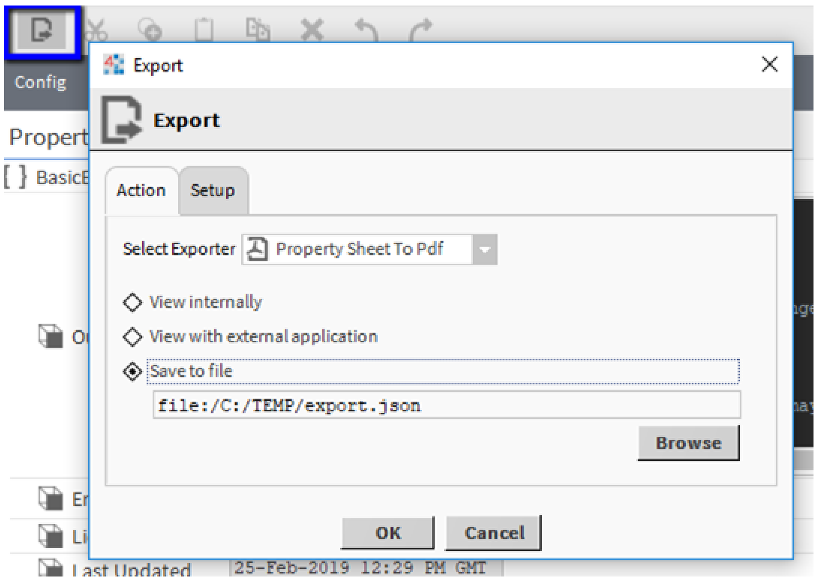
To prevent accidental data deluge, **Send Since** does not function if the filter's **Mode** is set to `Pass All`. You could use a bql query on the alarm database if this is a requirement.

Exporting schema output (JsonExporter)

The **JsonExporter** creates a file with the current output of the schema you are viewing. You could use this feature with the ReportService to export on a regular basis, perhaps via file, email, ftp or HTTP for a machine learning application or similar application.

Step 1 To export current JSON data, either click the **Export** button () or click **File→Export**

The **Export** window opens.



Select Exporter: Property Sheet To Pdf

View internally
View with external application
Save to file

file:/C:/TEMP/export.json

Last Updated 25-Feb-2019 12:29 PM GMT

Step 2 Select the exporter and where to view.

Step 3 To export to a file, you may click the **Browse** button to locate the file.

A URL like the following also allows access to the schema output via the JsonExporter:`http://127.0.0.1/ord/station:%7Cslot:/JsonSchema%7Cview:jsonToolkit:JsonExporter`

This means that using a web client you can easily query the data in a station over HTTP.

Exploring the examples

The JSON palette includes several examples you can explore to learn how schemata work.

Prerequisites: You are connected to a running station. The jsonToolkit palette is open.

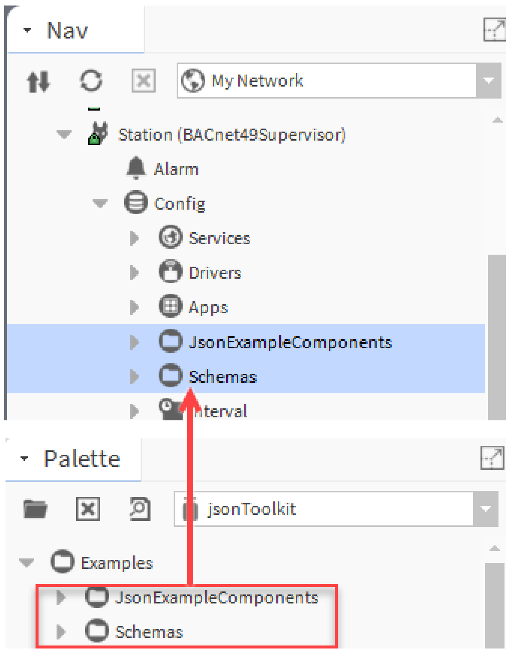
Step 1 Expand your **Station→Config** and double-click the **Config** node.

Step 2 Expand the **Examples** node in the palette.

Step 3 Select the two folders: **JsonExampleComponents** and **Schemas** and drop them into the **Config** node of your station.

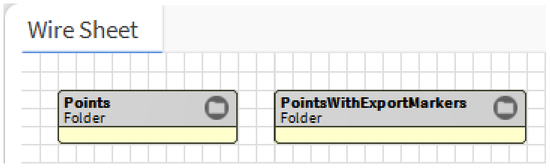
The examples folders must be at the root of the station **`Config`** component for them to work correctly.

The screen capture shows the example folders in the station **Config** folder.



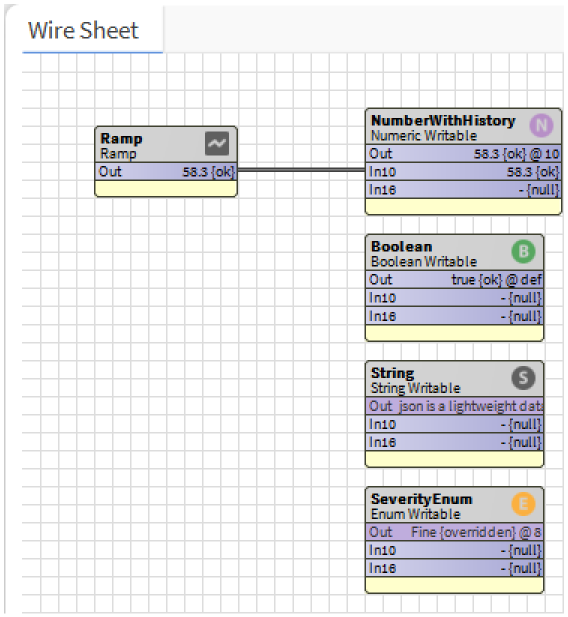
Step 4 To view the components, double-click the **JsonExamplecomponents** node.

The **Wire Sheet** opens to reveal two folders with points.



Step 5 Double-click the **Points** folder.

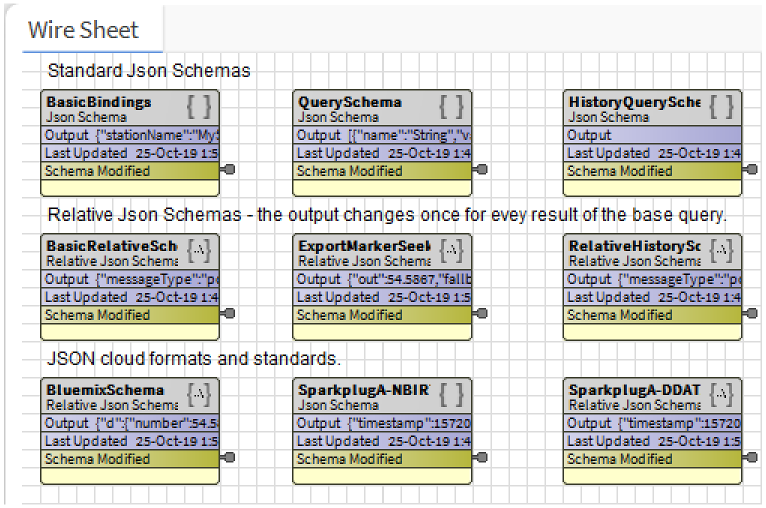
The **Wire Sheet** opens the **Points** folder.



This folder includes a Ramp that is updating.

Step 6 To view the sample schemata, double-click the **Schemas** folder in the Nav tree.

The **Wire Sheet** opens with nine example schemas.


Wire Sheet
Standard Json Schemas
BasicBindings
Json Schema
Output {"stationName":"My
Last Updated 25-Oct-19 1:5
Schema Modified
QuerySchema
Json Schema
Output [{"name":"String","v
Last Updated 25-Oct-19 1:4
Schema Modified
HistoryQuerySche
Json Schema
Output
Last Updated 25-Oct-19 1:4
Schema Modified
Relative Json Schemas - the output changes once for evey result of the base query.
BasicRelativeSch
Relative Json Schema
Output {"messageType":"p
Last Updated 25-Oct-19 1:4
Schema Modified
ExportMarkerSeek
Relative Json Schema
Output {"out":54.5867,"fallb
Last Updated 25-Oct-19 1:5
Schema Modified
RelativeHistorySc
Relative Json Schema
Output {"messageType":"p
Last Updated 25-Oct-19 1:4
Schema Modified
JSON cloud formats and standards.
BluemixSchema
Relative Json Schema
Output {"d":{"number":54.5
Last Updated 25-Oct-19 1:5
Schema Modified
SparkplugA-NBIR
Json Schema
Output {"timestamp":15720
Last Updated 25-Oct-19 1:4
Schema Modified
SparkplugA-DDAT
Relative Json Schema
Output {"timestamp":15720
Last Updated 25-Oct-19 1:5
Schema Modified

There is a basic example with bindings. Another that runs a query. There is a relative schema. Along the bottom are examples of how to apply a schema to a practical job. For example, there is are formats for communicating with an IBM cloud and the Sparkplug standard.

Connecting a device

This procedure uses an example to demonstrate how to connect a device. The example sets up a relative schema to look for all folders in the station that have a particular tag, such as "lights," "sensor," etc.

Step 1 Set up writable points in the station folders and connect them to the source points.

Step 2 Drag a relative schema to a logic **Wire Sheet**.

Step 3 Set up a base query to locate the point values.

For example: `slot:/Hue|neql:n:light` (assuming a “light” tag has been applied to the point’s parent folder).

Step 4 Specify the binding.

For example: `slot:`

The `Output` property displays the JSON message payload.

Step 5 Drag an **EngineCycleMessageQueue** to the **Wire Sheet**.

Step 6 On the **Wire Sheet**, link the **RelativeJsonSchema**’s Current Base And Output to the Enqueue slot of the queue.

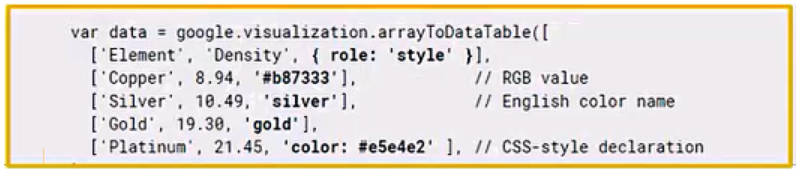
Step 7 Post the output from the program to HTTP.

Visualization

Generating the data for a graph uses the JSON queries. For example, you could use a JSON message to embed a chart in a web page.

The screen capture shows a JSON message that serves as the source for rendering a chart in a web site.

Figure 18 JSON message for Google chart data



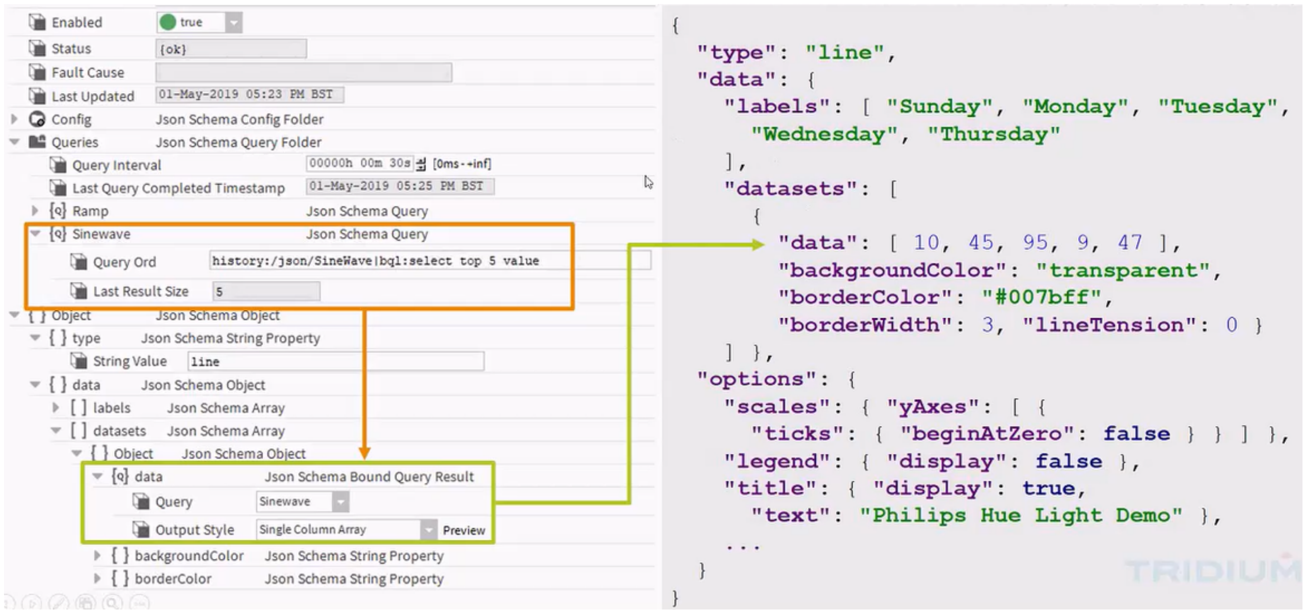
```
    var data = google.visualization.arrayToDataTable([
      ['Element', 'Density', { role: 'style' }],
      ['Copper', 8.94, '#b87333'],            // RGB value
      ['Silver', 10.49, 'silver'],            // English color name
      ['Gold', 19.30, 'gold'],
      ['Platinum', 21.45, 'color: #e5e4e2' ], // CSS-style declaration
```

This shows all the square brackets of several arrays with values. The JSON that generates this payload queries the history for a particular ramp in a station:

`history:/json/Ramp|bql:select top 5 value`

This is another (different) example of a schema and the JSON message that creates its chart.

Figure 19 JSON schema and output

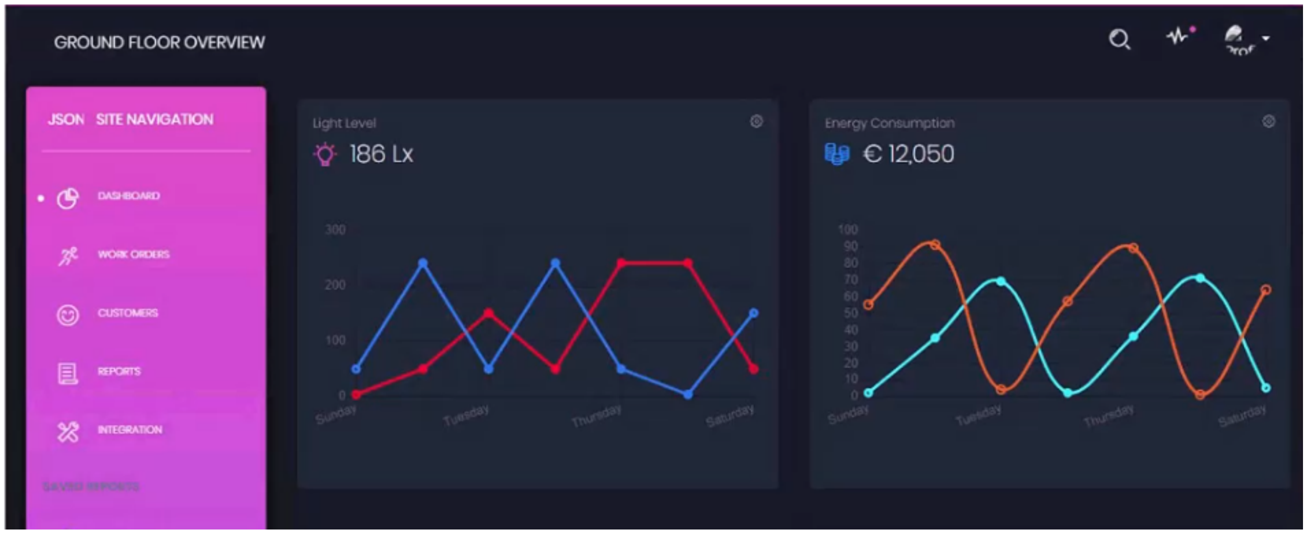


```
Enabled                         true
Status                          {ok}
Fault Cause
Last Updated                    01-May-2019 05:23 PM BST
Config                          Json Schema Config Folder
Queries                         Json Schema Query Folder
  Query Interval                00000h 00m 30s [0ms-+inf]
  Last Query Completed Timestamp 01-May-2019 05:25 PM BST
  {Q} Ramp                      Json Schema Query
  {Q} Sinewave                  Json Schema Query
    Query Ord                   history:/json/SineWave|bql:select top 5 value
    Last Result Size            5
{ } Object                      Json Schema Object
  { } type                      Json Schema String Property
    String Value                line
  { } data                      Json Schema Object
    [ ] labels                  Json Schema Array
    [ ] datasets                Json Schema Array
      { } Object                Json Schema Object
        {Q} data                Json Schema Bound Query Result
          Query                 Sinewave
          Output Style          Single Column Array   Preview
        { } backgroundColor     Json Schema String Property
        { } borderColor         Json Schema String Property
```

```
{
  "type": "line",
  "data": {
    "labels": [ "Sunday", "Monday", "Tuesday",
      "Wednesday", "Thursday"
    ],
    "datasets": [
      {
        "data": [ 10, 45, 95, 9, 47 ],
        "backgroundColor": "transparent",
        "borderColor": "#007bff",
        "borderWidth": 3, "lineTension": 0 }
    ] },
  "options": {
    "scales": { "yAxes": [ {
      "ticks": { "beginAtZero": false } } ] },
    "legend": { "display": false },
    "title": { "display": true,
      "text": "Philips Hue Light Demo" },
    ...
  }
}
```

Gold (orange) identifies the basic Query ord. The block identified by the green box and arrow (data) is the bound query result. The resulting graph looks like the following on a web page.

Figure 20 Charts created by JSON messages



Chapter 3 Importing JSON

Topics covered in this chapter

♦ Routing complete incoming messages
♦ Routing part of a message
♦ About the Json Path selector
♦ Handlers and alarm acknowledgments
♦ Setpoint handler and writing to points
♦ Export setpoint handler and export registration

Data coming into a station can be used to modify a setpoint or execute some other action. A handler processes imported JSON.

Routing complete incoming messages

A **`JsonMessageRouter`** component directs a whole incoming message (payload) to a new slot so that incoming messages may redirect the JSON to be handled by another component, such as a “handler” component type.

Step 1 Open the jsonToolkit palette, expand **Inbound→Routers** and drag a **`JsonMessageRouter`** component to a working folder in the station.

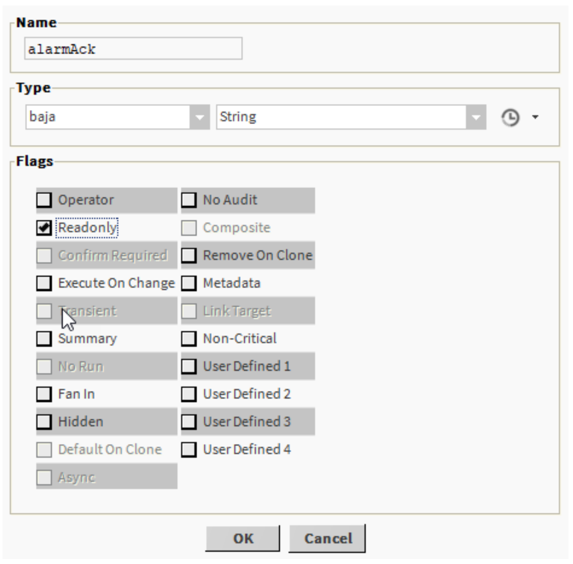
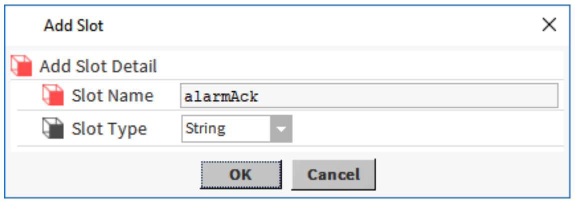
Step 2 Open the router Property Sheet by double-clicking the **`JsonMessageRouter`** component.

Step 3 Type a value in the **`Key`** property to identify the type of message (for example: `messageType`) and click **Save**.

Enabling **`Learn Mode`** adds a dynamic slot on input. This procedure documents how to add the slot manually.

Step 4 Manually add a dynamic string slot to the router component by opening the **AX Slot Sheet** view, or by simply right-clicking the sheet and clicking **Add Slot**.

An **Add Slot** window opens for either method, as shown below.

Adding a slot from the Slot Sheet View	Adding a slot using the Action menu
	

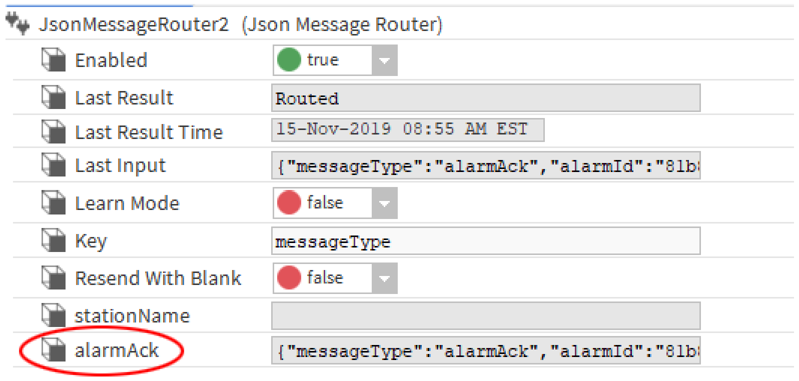
Name: alarmAck
Type: baja | String
Flags:
- [ ] Operator
- [✓] Readonly
- [ ] Confirm Required
- [ ] Execute On Change
- [ ] Transient
- [ ] Summary
- [ ] No Run
- [ ] Fan In
- [ ] Hidden
- [ ] Default On Clone
- [ ] Async
- [ ] No Audit
- [ ] Composite
- [ ] Remove On Clone
- [ ] Metadata
- [ ] Link Target
- [ ] Non-Critical
- [ ] User Defined 1
- [ ] User Defined 2
- [ ] User Defined 3
- [ ] User Defined 4

OK Cancel

Add Slot
Add Slot Detail
Slot Name: alarmAck
Slot Type: String
OK Cancel

Step 5 Give the slot a name, use the transient and read-only flags to avoid onward handlers running again at station start and click **OK**.

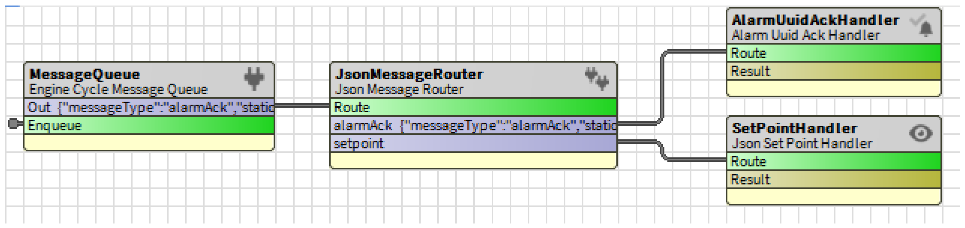
The new slot is added.


JsonMessageRouter2 (Json Message Router)

Enabled	true
Last Result	Routed
Last Result Time	15-Nov-2019 08:55 AM EST
Last Input	{"messageType":"alarmAck","alarmId":"81b
Learn Mode	false
Key	messageType
Resend With Blank	false
stationName	
alarmAck	{"messageType":"alarmAck","alarmId":"81b

Step 6 On the Wire Sheet, connect the router.

The following Wire Sheet routes the entire incoming message to the dynamic slot for onward processing:



For example, if Key = messageType, the JSON routes this message to a string slot with a name “alarmAck” and then on to connected handlers, as shown above.

```
{
  "messageType": "alarmAck",
  "user": "AJones",
  "alarmId": [ "5cf9c8b2-1542-42ba-a1fd-5f753c777bc0" ]
}
```

Routing part of a message

A **`JsonDemuxRouter`** directs a subset of an incoming message (payload) to a new slot so that links may redirect the JSON to be handled by another component. This procedure provides an example of routing part of a message.

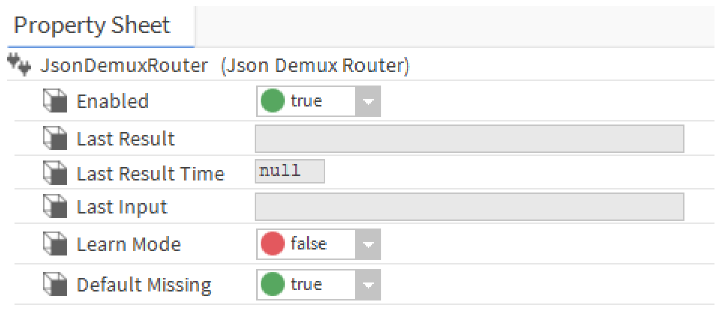
Prerequisites: The following instructions assume that you have an incoming message (payload) with the following key value pairs: “hue”, “sat”, “bri”, “on”.

```
{
  "hue": 43211,
  "sat": 254,
  "bri": 254,
  "on": true
}
```

Step 1 Open the palette, expand **Inbound→Routers** and drag a **`JsonDemuxRouter`** component to a desired location in the station.

Step 2 Open the **`JsonDemuxRouter`**’s **Property Sheet** by double-clicking the router.

The property sheet view displays.



Property Sheet

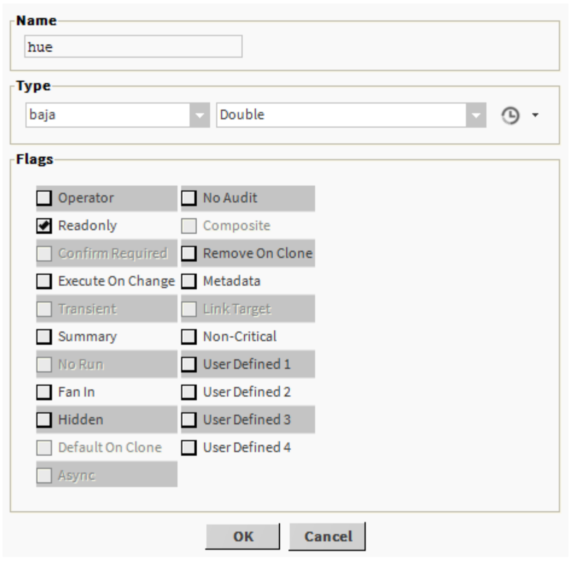
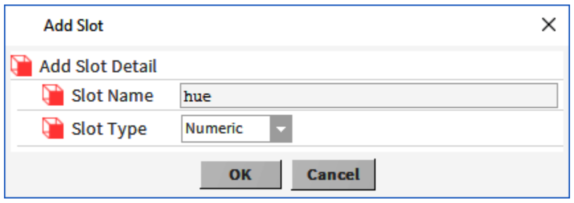
JsonDemuxRouter (Json Demux Router)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Learn Mode	● false
Default Missing	● true

NOTE: Enabling **Learn Mode** adds a dynamic slot on input. This procedure documents how to add the slot manually.

Step 3 Manually add a baja:double slot by opening the **AX Slot Sheet** view, or by simply right-clicking the sheet and clicking **Add Slot**.

An **Add Slot** window opens for either method, as shown below.

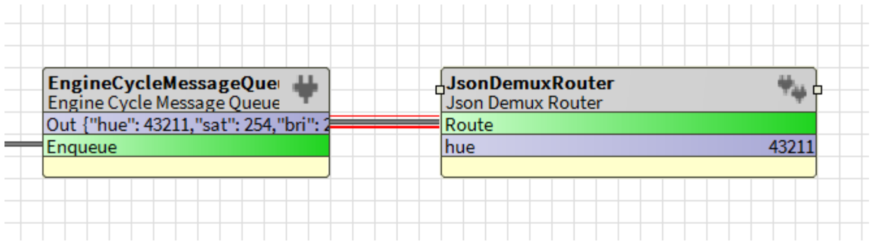
Adding a slot from the Slot Sheet View	Adding a slot using the Action menu
 Name: hue; Type: baja, Double; Flags: Readonly checked	 Add Slot; Add Slot Detail; Slot Name: hue; Slot Type: Numeric

Step 4 To add the slot to the `JsonDemuxRouter` component, give the slot a name ("hue" for this example), choose **Type**: `baja:Double` and click **OK**.

The new slot is added.

Step 5 In the **Wire Sheet** view, connect the schema output to the `JsonDemuxRouter` component's Route slot.

The following image shows a **Wire Sheet** view of components routing part of an incoming message to the slot for onward processing. The slot that you add must match the key name, to select that key, and should be either Boolean, Numeric or String to match the JSON value.



Once the **`JsonDemuxRouter`** component has a slot of type `baja:Double` named "hue", it passes the hue to expose the value "43211" for use in the station.

NOTE: To extract nested JSON objects, add a string with an appropriate name, for example, a demuxed string named 'data' could contain this entire nested object:

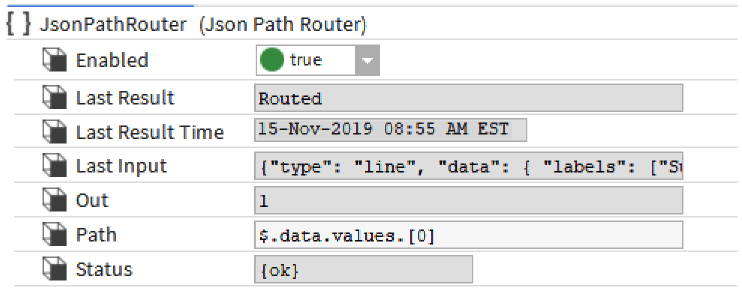
```
{
 "type" : "line",
  "data" :
  {
    "labels" : ["Sunday", "Monday"],
    "values" : [ 1, 2 ]
  }
}
```

About the Json Path selector

The **`JsonPath`** component allows data to be interactively located and extracted from JSON structures using a special notation to represent the payload structure.

For the example below, the first item in the values array (1) can be selected using a **`JsonPath`** value of `$.data.values.[0]`:

```
{
  "type" : "line",
  "data" :
  {
    "labels" : ["Sunday", "Monday"],
    "values" : [ 1, 2 ]
  }
}
```


{ } JsonPathRouter (Json Path Router)

Enabled	● true
Last Result	Routed
Last Result Time	15-Nov-2019 08:55 AM EST
Last Input	{"type": "line", "data": { "labels": ["Su
Out	1
Path	$.data.values.[0]
Status	{ok}

In this example a single numeric value was selected. However it is possible to select a complete subset of the incoming JSON, for example: `$.data` would select the entire data object into the out slot, or `$.data.values` would select the entire JSON "values" array. Any expression containing a search with `$..labels`, for example, will return search results enclosed within an outer array.

Much more explanation of this powerful tool can be found at the following websites:

- `https://goessner.net/articles/JsonPath/`
- `http://jsonpath.com/`
- `https://www.baeldung.com/guide-to-jayway-jsonpath`

Applying a jsonPath selector

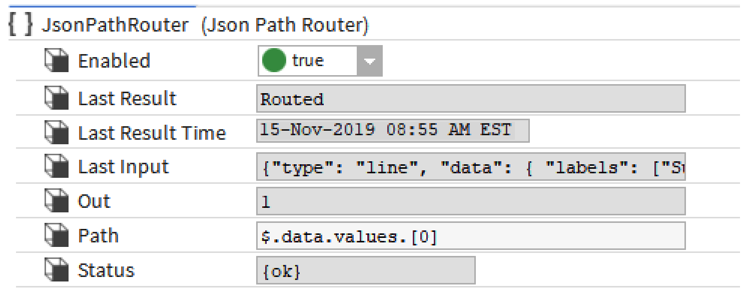
Selectors are components that apply selection criteria to an inbound message and display the result in an out slot. The **`JsonPath`** component allows data to be interactively located and extracted from JSON structures using a special notation to represent the payload structure.

Prerequisites: You have a schema generating an output that can be filtered.

The following task shows how to use a **JsonPath** component for data selection.

Step 1 Open the **jsonToolkit** palette, expand **Inbound→Selectors** and drag a **JsonPath** selector to a Wire Sheet and then open the selector's property sheet view.

Step 2 Configure the path property using the syntax `$.data.values.[0]`, as shown below, and save your changes.



{ } JsonPathRouter (Json Path Router)

Enabled	● true
Last Result	Routed
Last Result Time	15-Nov-2019 08:55 AM EST
Last Input	{"type": "line", "data": { "labels": ["S
Out	1
Path	$.data.values.[0]
Status	{ok}

The result of the configuration displays in the **Out** property.

For example, this path selects the first item in a values array (1): `$.data.values.[0]`. This is the schema payload:

```
{
  "messageType" : "line",
  "data" : [
  {
    "labels" : ["Sunday", "Monday"],
    "values" : [ 1, 2 ]
  }
}
```

This example selects a single numeric value, however, there are other possibilities for selecting a subset of the incoming JSON:

- `$.data` transfers the entire data object to the **Out** slot.
- `$.data.values` selects the entire JSON array.

Any expression containing a search with, for example, `$..labels` returns search results enclosed within an outer array.

These URLs to external web sites explain this powerful tool in detail.

- https://goessner.net/articles/JsonPath/
- http://jsonpath.com/
- https://www.baeldung.com/guide-to-jayway-jsonpath

Handlers and alarm acknowledgments

Message handlers are components designed to perform a specific task with the data routed and selected via the other inbound components. Handlers make acknowledging alarms possible.

If an alarm exported from a station includes the UUID, an Alarm Uuid Ack Handler can pass back that unique id. The expected format is shown below, where the array allows multiple alarms to be acknowledged at once.

```
{
  "user": "Maya",
  "alarms": [ "5cf9c8b2-1542-42ba-a1fd-5f753c777bc0" ]
}
```

The user value stored on the alarm record identifies which user acknowledged the alarm in the remote application. If the user key is omitted the component still tries to acknowledge the alarms using the fallback name "AlarmUuidAckUser".

NOTE: The Json Schema Service **`runAsUser`** is a prerequisite for this handler to work. The specified user must have admin write permissions for the alarm class of the records being acknowledged.

Two alarm handler properties configure this task:

- **`AckSource`** is a string appended to every AlarmRecord acknowledged. Its purpose is to allow auditing in future and is stored as **`AckSource`** in the alarm data.
- AckResult is a topic that reports the results of the alarm acknowledgment. Its purpose is to log or post process activity. Here is an example of the output it reports:

  `"Ack-ed alarm " + record`

  `"Already ack-ed in alarmDb " + record`

  `"Could not create BUuid from " + uuid`

Setpoint handler and writing to points

The SetPointHandler sets incoming setpoint values to control writable control points.

ID.

This is an example of setpoint handler JSON:

```
{
  "%idKey%" : "x",
  "%valueKey%" : y,
  ("%slotNameKey%" : "slotName")
}
```

The Control Points are located by handle ord in the form: `"%idKey%" : "323e"`.

These properties configure setpoint handlers:

- **`idKey`** is a top-level key in the JSON payload. It represents the point ID.
- **`valueKey`** is a top-level key in the JSON payload. It represents the value to set.
- **`slotNameKey`** is an optional top-level key in the JSON payload. It represents the slot name to write to.
- **`defaultWriteSlot`** defines which slot to write to by default if the payload does not specify a slot.
- **`runAsUser`** is a mandatory property for the setpoint handler to use.

The nested keys, **`override/duration`** and **`status`** are not currently supported.

Export setpoint handler and export registration

Like the **`SetpointHandler`**, the **`ExportSetpointHandler`** allows an external JSON message to change the value of a control point identified by the Id property of an export marker.

This handler locates target points in a station where a unique key from the cloud platform registered the points. Once the cloud platform returns a suitable identifier for a point with an export marker, this setpoint handler can apply write messages from the platform using the returned Id rather than the slot or handle ord (for example).

Export registration

The **`JsonExportRegistrationRouter`** and **`JsonExportDeregistrationRouter`** enable this behaviour of applying a unique identifier from an external system to an export marker.

This allows the cloud (or other external system) to assign it's own identifier or primary key to export-marked points in the Niagara station, which can be used to locate them in future or include them in exports to the cloud system.

The messages should be in this format:

```
{
  "messageType" : "registerId"
  "niagaraId" : "h:a032",
  "platformId" : "mooseForce123"
}
```

or

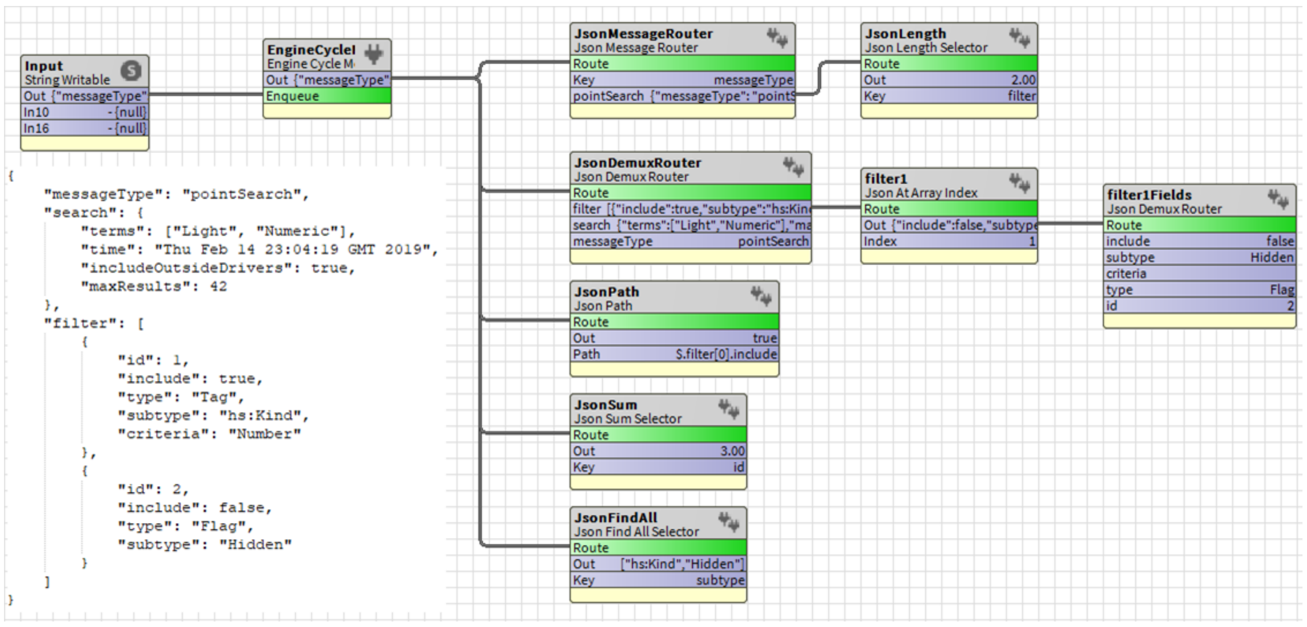
```
{
  "messageType" : "deregisterId"
  "platformId" : "mooseForce123",
}
```

NOTE: This class does not use the messageType, which would be used simply to route it to this handler and so can be changed as needed.

Example

This **Wire Sheet** and JSON loosely demonstrate some of the routers and selectors based upon a fictional point search JSON message.

Figure 21 Json Export Registration Handler example Wire Sheet and JSON



```
Input
String Writable
Out {"messageType"
In10 - {null}
In16 - {null}

EngineCycle1
Engine Cycle M
Out {"messageType"
Enqueue

JsonMessageRouter
Json Message Router
Route
Key messageType
pointSearch {"messageType": "pointS

JsonLength
Json Length Selector
Route
Out 2.00
Key filter

JsonDemuxRouter
Json Demux Router
Route
filter [{"include":true,"subtype":"hs:Kin
search {"terms":["Light","Numeric"],"ma
messageType pointSearch

filter1
Json At Array Index
Route
Out {"include":false,"subtype
Index 1

filter1Fields
Json Demux Router
Route
include false
subtype Hidden
criteria
type Flag
id 2

JsonPath
Json Path
Route
Out true
Path $.filter[0].include

JsonSum
Json Sum Selector
Route
Out 3.00
Key id

JsonFindAll
Json Find All Selector
Route
Out ["hs:Kind","Hidden"]
Key subtype
```

```
{
    "messageType": "pointSearch",
    "search": {
        "terms": ["Light", "Numeric"],
        "time": "Thu Feb 14 23:04:19 GMT 2019",
        "includeOutsideDrivers": true,
        "maxResults": 42
    },
    "filter": [
        {
            "id": 1,
            "include": true,
            "type": "Tag",
            "subtype": "hs:Kind",
            "criteria": "Number"
        },
        {
            "id": 2,
            "include": false,
            "type": "Flag",
            "subtype": "Hidden"
        }
    ]
}
```

Chapter 4 Developer guide

Topics covered in this chapter

- ♦ JSON schema types
- ♦ Relative topic builder
- ♦ Type Override example
- ♦ Inline JSON Writer
- ♦ Custom query style
- ♦ Builder class / API
- ♦ Useful methods
- ♦ How schema generation works
- ♦ Working with Apache Velocity
- ♦ Subscription examples with bajascript
- ♦ Inbound components

Developers can use JSON to create complex queries and apps. They can extend the Toolkit by creating their own query styles.

JSON schema types

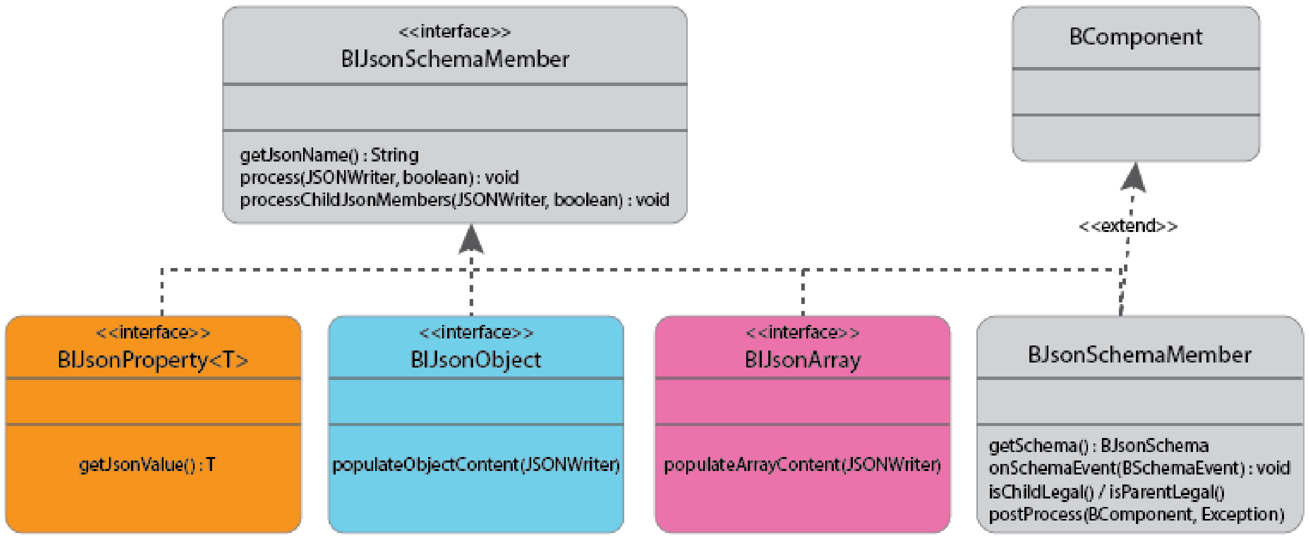
All components that contribute to the string output of the schema are called members and are nested under the schema. During generation, the system processes each member recursively (top down), appending each member's result to a JSON writer. This creates the final JSON output string.

Three interfaces represent three structural types of the JSON payload:

- Property (key/value pair)
- Object
- Array

A getJsonName() defines each schema member.

Figure 22 Schema types



<<interface>>
BIJsonSchemaMember
getJsonName() : String
process(JSONWriter, boolean) : void
processChildJsonMembers(JSONWriter, boolean) : void
BComponent
<<extend>>
<<interface>>
BIJsonProperty<T>
getJsonValue() : T
<<interface>>
BIJsonObject
populateObjectContent(JSONWriter)
<<interface>>
BIJsonArray
populateArrayContent(JSONWriter)
BJsonSchemaMember
getSchema() : BJsonSchema
onSchemaEvent(BSchemaEvent) : void
isChildLegal() / isParentLegal()
postProcess(BComponent, Exception)

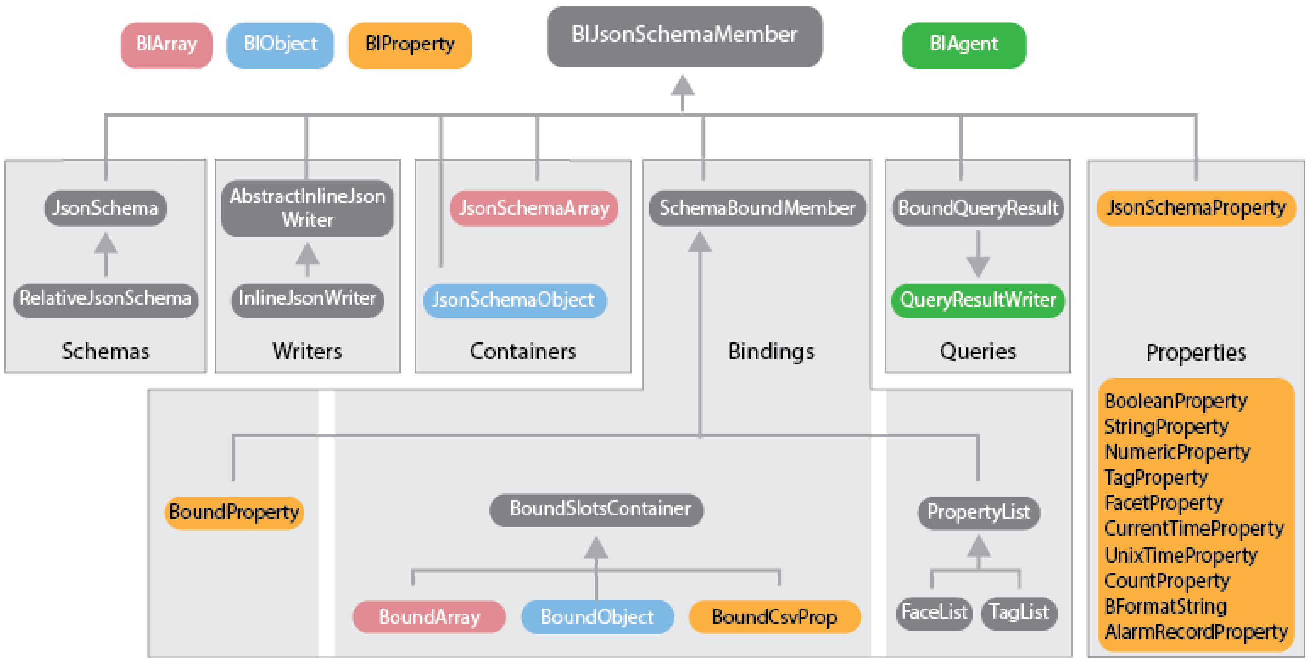
Three interfaces represent the three structural types of a JSON payload: property (key and value pair), object and array. All schema members have a name defined by getJsonName().

All schema members inherit the default processChildJsonMembers() behaviour, which allows us to recursively call process() on each member down through the nested schema structure.

All schema member types extend BJsonSchemaMember and most implement one of the three interface types. The base class lets us define the parent-child legal checks. This restricts nested types to just other BJsonSchemaMembers. This is where the JSON passes global schema events, for example, unsubscribe.

Different types of JSON schema members may be nested under a schema. These are logically grouped by common behaviour.

Figure 23 Json schema members



When developing against the toolkit, most of these classes are open to extension.

Example 1

Consider a requirement for a new key and value pair to represent a device’s startup time as a string value. You might simply extend the `BJsonSchemaProperty<T>` as type `<String>` using your own date format or type `<BAbsTime>` allowing the schema to render the date automatically using the schema date config. Now, you just need to implement `getJsonValue()` to return the appropriate value.

```
@NiagaraType
public class BDeviceTimeProperty extends BJsonSchemaProperty<BAbsTime>
{
/*+ ------------ BEGIN BAJA AUTO GENERATED CODE ------------ +*/
.....
/*+ ------------ END BAJA AUTO GENERATED CODE -------------- +*/

 @Override
 public BAbsTime getJsonValue()
 {
   return (BAbsTime) ..... // this will use the schemas date format config
 }
}
```

Example 2

This requirement is for an object that contains a key and value pair for each slot on the target component, but only those with a user defined 1 flag. You might extend BJsonSchemaBoundObject, hide the slotsToInclude slot, and override the method getPropertiesToIncludeInJson() to only return properties with the user defined flag.

```
@NiagaraType
@NiagaraProperty(name = "slotsToInclude", type = "jsonToolkit:SlotSelectionType",
defaultValue = "BSlotSelectionType.allVisibleSlots",flags = Flags.HIDDEN,
override = true) public class BUserDefinedFlags extends BJsonSchemaBoundObject
{
/*+ ------------ BEGIN BAJA AUTO GENERATED CODE ------------ +*/
.....
/*+ ------------ END BAJA AUTO GENERATED CODE -------------- +*/
 @Override
 public List <String>getPropertiesToIncludeInJson(BComplex resolvedTarget)
 {
  if (resolvedTarget == null)
  {
   return Collections.emptyList(); // or try to resolve it!
  }
   return Arrays.stream(resolvedTarget.getPropertiesArray())
    .filter(prop -> (resolvedTarget.getFlags(prop) & Flags.USER_DEFINED_1) != 0)
    .map(prop -> prop.getName())
    .collect(Collectors.toList());
 }
}
```

Relative topic builder

If the recipient requires a different topic or URL per point or device, the **`relativeTopicBuilder`** component is an example of building a topic (for MQTT) or path (for HTTP url) as the output from the current base item of a relative schema changes.

This program object is in the **Programs** folder of the **`jsonToolkit`** palette.

As an example, to update each item returned by the base query, you would link from the RelativeJsonSchema's **`Current Base Output`** topic to the **`Base Item Changed`** property, and then from the output slot to the publish points proxyExt.

Other properties of the base could be inserted to the topic as desired (not just the name).

The example that is included in the palette illustrates the `%s` variable substituted by this: `"/an/mqtt/example/%s"`.

Type Override example

At the core of the JSON Toolkit is a method that maps baja object types to JSON. This determines, for example, how any encountered BControlPoint, Facets, BAbsTime etc. should be encoded in the output.

The payload includes many variations for the supported Niagara types. Our approach to accommodating this is to allow a developer or power user the ability to override specific types as they are converted to JSON.

For a small JsonSchema, the example in the **`jsonToolkit`** palette demonstrates how to use a program `object[^1]` to replace units:

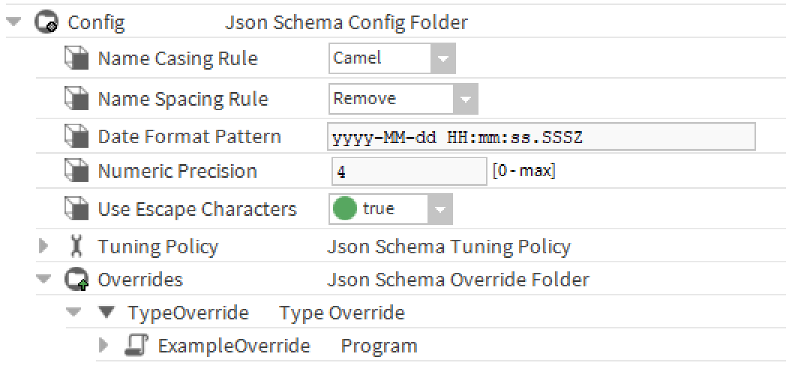
```
/**
   * Allows Json types to to be overridden when placed under JsonSchema/config/overrides/
   */
  public BValue onOverride(final BValue input)
  {
    if (input instanceof BUnit)
    {
      javax.baja.units.UnitDatabase unitDB = javax.baja.units.UnitDatabase.getDefault()
      javax.baja.units.UnitDatabase.Quantity quantity =
      unitDB.getQuantity(input.as(BUnit.class))
      if (quantity != null)
      {
        return BString.make(input.as(BUnit.class).getSymbol() + ":" + quantity.getName())
      }
    }

    // If we can't override the value then just return it as we found it
    return input
  }
```

[^1]: To improve maintainability and station loading time in the event that a program object is duplicated repeatedly, use the ProgramBuilder.

To use the program, drag this component into the **Config→Overrides** folder of the schema.

Figure 24 TypeOverride component in jsonSchema



Config	Json Schema Config Folder
Name Casing Rule	Camel
Name Spacing Rule	Remove
Date Format Pattern	yyyy-MM-dd HH:mm:ss.SSSZ
Numeric Precision	4 [0 - max]
Use Escape Characters	● true
Tuning Policy	Json Schema Tuning Policy
Overrides	Json Schema Override Folder
TypeOverride	Type Override
ExampleOverride	Program

Developers could also override the `doOverride(BValue value)` method in their own `BTypeOverride` variant.

Inline JSON Writer

This writer allows the schema to defer control to a developer's own code in the tree of schema members. This means that you can add any form of dynamic content into the schema output.

To add custom dynamic content, use a program object as per the example in the **Programs** folder of the **`jsonToolkit`** palette. Or you can extend `BAbstractInlineJsonWriter`. As code contained in a module is easier to maintain, extending the abstract class would be preferred where the program object may be widely distributed.

This palette example implements a method: `public BValue onOverride(final BInlineJsonWriter input)`, which you can customize to meet your project's needs. The `InlineJsonWriter` has two important methods:

- `JSONWriter jsonWriter = in.getJsonWriter();`
- `BComplex base = in.getCurrentBase();`

Demonstrated below:

```
/**
 * The override method allows control of the writer and current base to be passed
 * to the code below  * allowing JSON to be dynamically constructed within a schema.
 *
 * @param BInlineJsonWriter wraps two things:
 *          JSONWriter jsonWriter = in.getJsonWriter();
 *           BComplex base = in.getCurrentBase();
 *
 * @return BValue allows logging of the "result" when fine logging is enabled
 * (this does not need to match what happened to the JSON...)
 */
public BValue onOverride(final BInlineJsonWriter in)
{
    //current base is set by the parent schema as each point is submitted for publishing
    BComplex base = in.getCurrentBase()

    //if (base instanceof BComponent)

    JSONWriter jsonWriter = in.getJsonWriter()

    jsonWriter.key("highLimit")
    jsonWriter.value("1024")

    // do not close writer

    return null
}
```

Custom query style

Third–party systems may require query results to be formatted in a manner other than the options provided in the JSON Toolkit.

To render query data differently, extend `BQueryResultWriter` and register the class as an agent on `jsonToolkit:JsonSchemaBoundQueryResult`.

This example shows how to format the contents of the `QueryResultHolder` for an external system:

```
package com.tridiumx.jsonToolkit.outbound.schema.query

import static com.tridiumx.jsonToolkit.outbound.schema.support.JsonSchemaUtil.toJsonType

import java.util.concurrent.atomic.AtomicInteger
import javax.baja.nre.annotations.AgentOn
import javax.baja.nre.annotations.NiagaraType
import javax.baja.sys.BString
import javax.baja.sys.Sys
import javax.baja.sys.Type

import com.tridiumx.jsonToolkit.outbound.schema.query.style.BQueryResultWriter
import com.tridium.json.JSONWriter

/**
 * An example custom query result writer.
 *
 * @author Nick Dodd
 */
@NiagaraType(agent = @AgentOn(types = "jsonToolkit:JsonSchemaBoundQueryResult"))
public class BCowSayJson extends BQueryResultWriter
{
/*+ ------------ BEGIN BAJA AUTO GENERATED CODE ------------ +*/
/*@ $com.tridiumx.jsonToolkit.outbound.schema.query.style.BObjectsArray(4046064316)1.0$ @*
/* Generated Thu Dec 13 11:24:58 GMT 2018 by Slot-o-Matic (c) Tridium, Inc. 2012 */

////////////////////////////////////////////////////////////////
```

```
// Type
////////////////////////////////////////////////////////////////

  @Override
  public Type getType() { return TYPE }
  public static final Type TYPE = Sys.loadType(BCowSayJson.class)

/*+ ------------ END BAJA AUTO GENERATED CODE -------------- +*/

  @Override
  public BString previewText()
  {
    return BString.make("A demonstration result writer")
  }

  @Override
  public void appendJson(JSONWriter jsonWriter, QueryResultHolder result)
  {
    jsonWriter.object()

    try
    {
      jsonWriter.key("mooo01").value("____________________________")
      headerCsv(jsonWriter, result)
      dataCsv(jsonWriter, result)
      jsonWriter.key("mooo02").value("----------------------------")
      jsonWriter.key("mooo03").value("    \\   ^__^               ")
      jsonWriter.key("mooo04").value("     \\  (oo)\\_______      ")
      jsonWriter.key("mooo05").value("         (__)\\       )\\/\\")
      jsonWriter.key("mooo06").value("              ||----w |     ")
      jsonWriter.key("mooo07").value("              ||     ||     ")
    }
    finally
    {
      jsonWriter.endObject()
    }
  }

  private void headerCsv(JSONWriter jsonWriter, QueryResultHolder result)
  {
    jsonWriter.key("columns").value(String.join(",", result.getColumnNames()))
  }

  private void dataCsv(JSONWriter jsonWriter, QueryResultHolder result)
  {
    AtomicInteger rowCount = new AtomicInteger()

    result.getResultList().forEach( map - {

      jsonWriter.key("data" + rowCount.incrementAndGet())
      jsonWriter.array()
      try
      {
        map.values()
          .forEach(value - jsonWriter.value(toJsonType(value, getSchema().getConfig())))
      }
      finally
      {
        jsonWriter.endArray()
      }
    })

    processChildJsonMembers(jsonWriter, false)  // append any nested members content
    to the json
  }
}
```

Builder class / API

To support the programmatic creation of JSON schemata by developers, the `JsonSchemaBuilder` class provides suitable methods.

For example:

```
BJsonSchema schema =
        new JsonSchemaBuilder()
          .withUpdateStrategy(BJsonSchemaUpdateStrategy.onDemandOnly)
          .withQuery("Bacnet Query", "station:|slot:/Drivers/BacnetNetwork|bql:select
           out.value AS 'v', status from control:ControlPoint")
          .withRootObject()
          .withFixedNumericProperty("Version", BDouble.make(1.23))
          .withFixedObject("Config")
          .stepDown()
            .withBoundProperty("BacnetAddress", BOrd.make(String.format
             ("station:|slot:/Drivers/BacnetNetwork/%s/address", deviceName)))
            .withBoundObject("DeviceSettings", BOrd.make(String.format
             ("station:|slot:/Drivers/BacnetNetwork/%s/config/deviceObject", deviceName)))
          .stepUp()
          .withBoundQueryResult("Data", "Bacnet Query", BObjectsArray.TYPE.getTypeSpec())
      .build()
```

The above schema would result in this output:

```
{
      "Version":1.23,
      "Config":{
        "BacnetAddress":"192.168.1.24",
        "DeviceSettings":{
          "pollFrequency":"Normal",
          "status":"{ok}",
          "faultCause":"",
          "objectId":"device:100171",
          …….
    }
    },
    "Data":[
       {
         "v":0.45,
         "status":"{down,stale}"
       },
       ……*
       ]
    }
```

NOTE: This example has been trimmed for demonstration purposes.

Useful methods

These are some methods you might regularly use to create custom content.

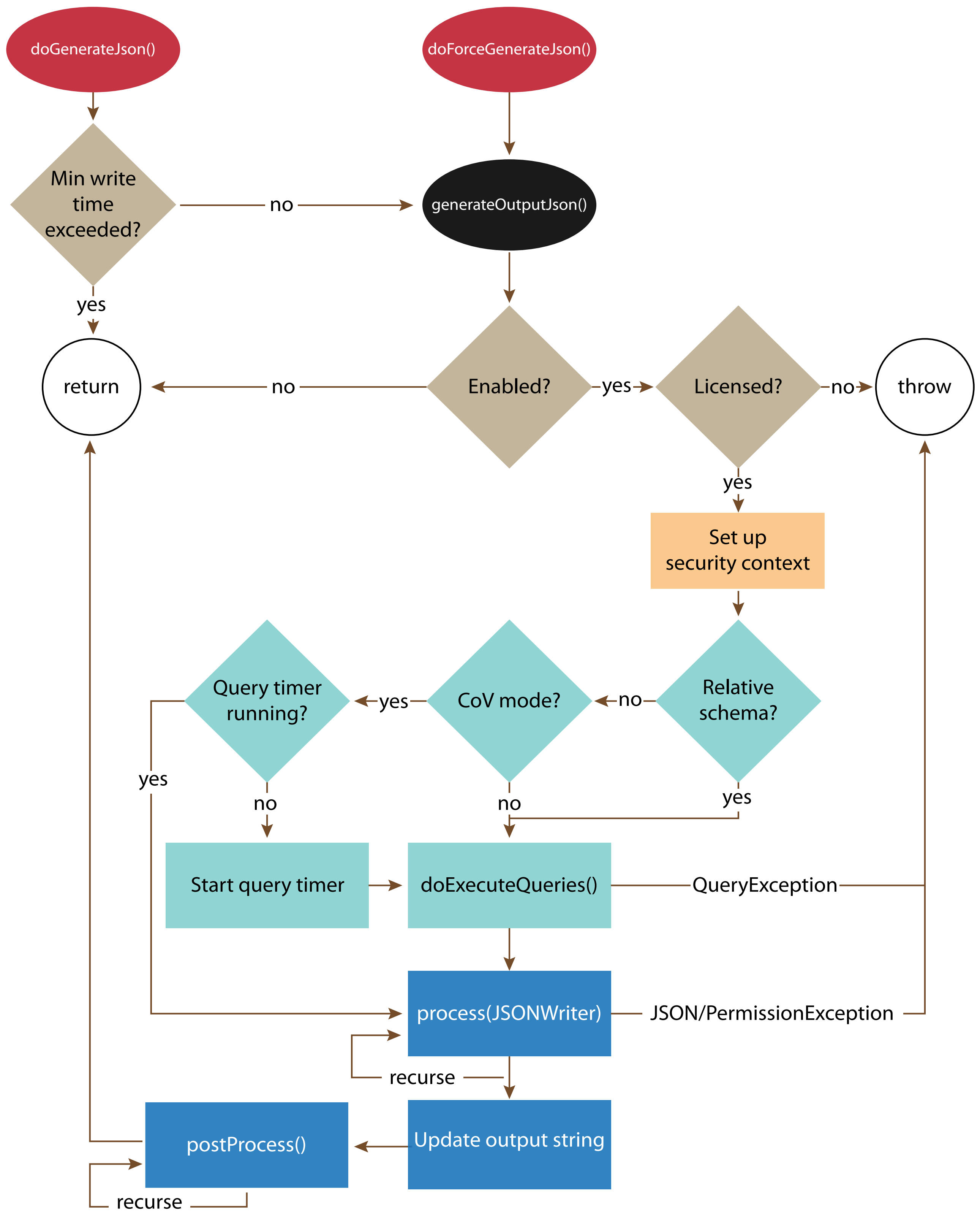
What to do (usage goal)	Class	Method
Override to return a different key. This skips the schema's config settings for name case/space handling.	BIJsonSchemaMember	getJsonName()
Override to append customized content to the current JSON stream via `json.key()` and `json.value()`, etc. The Boolean parameter indicates if the syntax of the keys are currently valid (for example, not inside an array)	BIJsonSchemaMember	process(JSONWriter json, boolean jsonKeysValid)
Override to react to events, such as the base item changing or subscription disabled.	BJsonSchemaMember	onSchemaEvent(BSchemaEvent event)
Quickly get a reference to the parent schema	BJsonSchemaMember	getSchema()
Write a JSON key with the schema's current case and space-handling rules applied.	JsonSchemaNameUtil	writeKey(BIJsonSchemaMember member, JSONWriter jsonWriter, String name)
Convert any Java value to a native JSON type (String or Number or Boolean) with some default handling of some baja types, and filter out sensitive types.	JsonSchemaUtil	toJsonType(Object value, BJsonSchemaConfigFolder config)
Convert core Java type values (Numerics or Strings or Booleans) to BValue equivalents. If the parameter is already a BValue, this method returns a copy.	JsonSchemaUtil	toBValue(Object value)
Get a live resolved reference to the ord bindings target.	BJsonSchemaBoundMember	getOrdTarget() / getTarget()
Override the schema's default behaviour for handling a subscription event from a binding target. Depending on the content, the schema's default behaviour is to unsubscribe, ignore or request schema generation.	BJsonSchemaBoundMember	handleSubscriptionEvent(Subscription subscription, BComponentEvent event)
Override to return a different set of slot values for the resolved target.	BJsonSchemaBoundSlotsContainer	getPropertiesToIncludeInJson (target)
Implement to perform any lifecycle, cleanup or reporting task after a schema has completed output generation (or failed, in which case, the exception is non-null).	BIPostProcessor	postProcess(BJsonSchema schema, Exception exception)
Extract values from incoming JSON payloads using various methods.	JsonKeyExtractUtil	lookup*()
Implement to handle an incoming JSON payload or throw a RoutingFailedException if unable to process the message.	BJsonInbound	routeValue(BString message, Context cx)
Override to locate a control point by another means than the handle ord, for example by slot path or name.	BJsonSetPointHandler	lookupTarget(BString msg, String id)

How schema generation works

Two actions cause the JSON schema to generate or regenerate it's output.

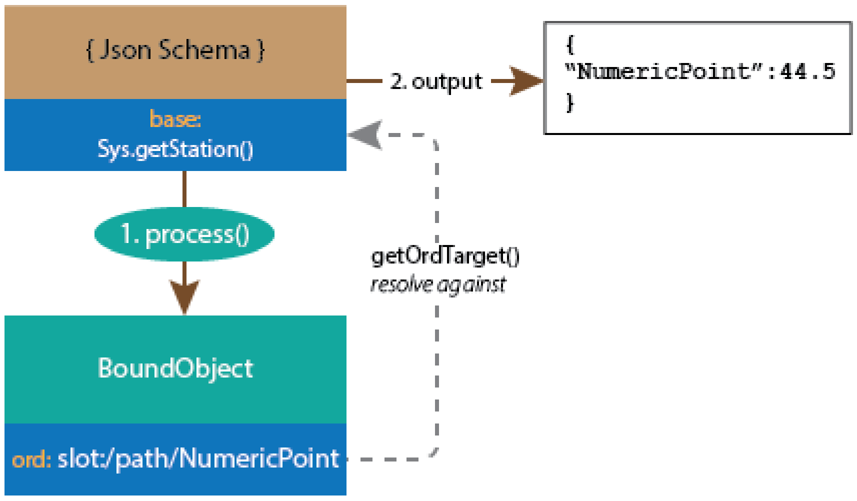
This charts the flow through the schema logic.

Figure 25 Generate actions



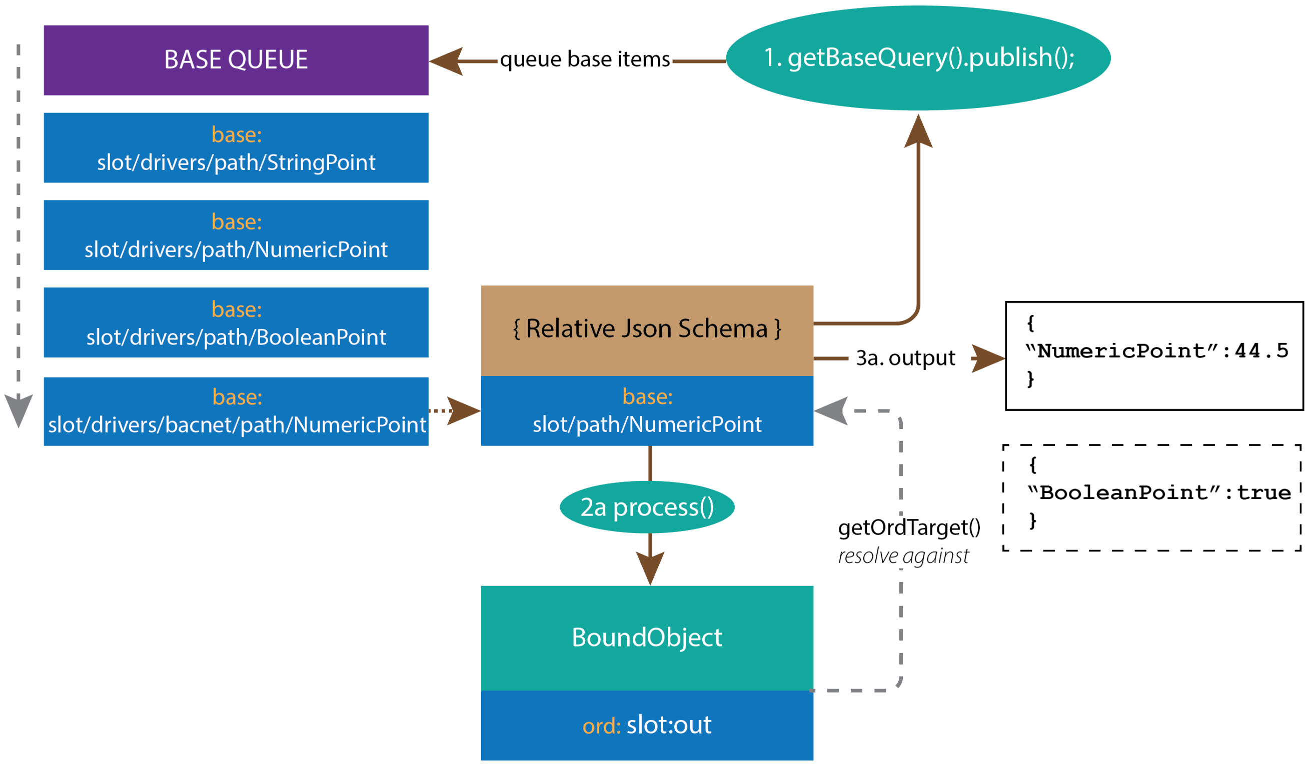
Binding ords resolve against the current base item of the schema. Unless you are using a relative JSON schema, this is the station that uses the current result of the base query. Currently, base queries resolve against the station.

Figure 26 Regular JSON schema with absolute ord bindings that resolve against the station



Relative JSON schema with relative ord bindings resolve against the current base item. This process repeats until there are no more base items, and results in several output strings.

Figure 27 Relative JSON schema with relative ord bindings that resolve against the current base item



External access to schema output

A URL like the following also allows access to the schema output via the JsonExporter:

```
http://127.0.0.1/ord/station:%7Cslot:/JsonSchema%7Cview:jsonToolkit:JsonExporter
```

This could allow access to an external application consuming data from Niagara.

Working with Apache Velocity

Apache Velocity is a Java-based template language anyone can use to reference objects defined in Java code. You can use it to expose the output of a JSON schema via the Jetty Web Server in Niagara 4. This tool may be beneficial for applications that expect to consume data provided by the Niagara station, for example, a visualization or machine-learning library.

Prerequisites:

Given JSON's origin as a data exchange format for the web, many libraries expect to receive input in this format. The Google Chart library is such an example. The following example is from the Google Chart project web site. Notice that the `var data` is populated with JSON data. Replacing hard-coded data with the output from a suitably-configured JSON schema in your station draws a chart from the Niagara station data.

```
<html>
 <head>
  <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>
  <script type="text/javascript">
     google.charts.load('current', {'packages':['corechart']});
     google.charts.setOnLoadCallback(drawChart);
     function drawChart() {
       var data = google.visualization.arrayToDataTable([
         ['Year', 'Sales', 'Expenses'],
         ['2004',  1000,      400],
         ['2005',  1170,      460],
         ['2006',  660,       1120],
         ['2007',  1030,      540]
       ]);
       var options = {
         title: 'Company Performance',
         curveType: 'function',
         legend: { position: 'bottom' }
       };

       var chart = new google.visualization.LineChart
       (document.getElementById('curve_chart'));

       chart.draw(data, options);
     }
   </script>
 </head>
 <body>
   <div id="curve_chart" style="width: 900px height: 500px"></div>
 </body>
</html>
```

Step 1 Create a new file `chart.vm` and paste into it the code example of a sample chart from the json-consuming-charting library of your choice.

Step 2 Replace the JSON data with a velocity variable, for example, `$schema.output`,

```
var data = google.visualization.arrayToDataTable([
        $schema.output
    ])
```

Step 3 After saving the file, open the **`axvelocity`** palette and add a **`VelocityServlet`** named “chart” to your station.

Step 4 Add a VelocityDocument below the servlet and change the **`Template File`** property to point to the `chart.vm` file you created earlier.

Step 5 Add a new `ContextOrdElement` named Schema to the `VelocityContext` of your VelocityDocument.

Step 6 Update the Schema Ord element to point to a suitable jsonSchema added to your station.

This schema could output live station data or the result of a query or transform. Both would be suitable for charting libraries, although it may be necessary to modify the time and date format form the schema default settings or to reduce the presented interval of data by using a `SeriesTransform Rollup` function.

So, what did we achieve? The template HTML file has a variable, which when accessed via the station's velocity servlet will be replaced with the output from our schema.

If you add a WebBrowser from the **`workbench`** palette to a Px Page and set the **`ord`** property to `http:\\127.0.0.1\velocity\chart`, you should see a chart when you view the page in a web browser. If not, use the developer tools to view the source code and ensure that the output of your schema is replacing the `$schema.output` variable.

Subscription examples with bajascript

Whilst Velocity is a very convenient means to inject data into an html document, one of many benefits of using bajascript in your application is support for subscriptions, which update the graphic as data change.

Of course, you could build this schema output in bajascript by executing queries or by directly subscribing to the components required, but a jsonSchema may reduce some of the work needed in JavaScript, allowing subscription only to the output slot, which can fetch the required data from the station.

Example html file for showing Chart.js

```
<!DOCTYPE html
<!-- @noSnoop --
<html
<head
  <titleHTML Page</title>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.7.3/Chart.min.js">
  </script>

  <script type='text/javascript' src='/requirejs/config.js'></script>
  <script type='text/javascript' src='/module/js/com/tridium/js/ext/require/require.min
  .js?'></script>

  <!-- note the special syntax for downloading JS file from the 'bajascript' folder
  you add in your station -->
  <script type='text/javascript' src='/ord/file:%5Ebajascript/basic.js%7Cview:web
  :FileDownloadView'></script>

</head>
<body>

  <canvas class="my-4 w-100" id="myChart" width="800" height="450"></canvas>

</body>
</html>
```

Example basic.js file to fetch chart data

The data array in the payload below uses bound properties. A single-column query would allow historical data to be used instead from a bql query on the history database.

```
// Subscribe to a Ramp. When it changes, print out the results.
require(['baja!'], function (baja) {
    "use strict"

    // A Subscriber is used to listen to Component events in Niagara.
    var sub = new baja.Subscriber()

    var update = function () {

      // Graphs
      var ctx = $('#myChart')

      var newJson = JSON.parse(this.getOutput())

      var myChart = new Chart(ctx, newJson)
    }

    // Attach this function to listen for changed events.
    sub.attach('changed', update)

    // Resolve the ORD to the Ramp and update the text.
    baja.Ord.make('station:|slot:/ChartsJS/JsonSchema').get({ok: update, subscriber: sub})
    })
```

Example schema output for chart

```
{
  "type": "line",
  "data": {
    "labels": [
      "Sunday",
      "Monday",
      "Tuesday",
      "Wednesday",
      "Thursday"
    ],
    "datasets": [
      {
        "data": [
          202,
          240,
          202,
          3,
          150
        ],
        "backgroundColor": "transparent",
        "borderColor": "#007bff",
        "borderWidth": 3,
        "lineTension": 0
      },
      {
        "data": [
          3,
          202,
          150,
          202,
          240
        ],
        "backgroundColor": "transparent",
        "borderColor": "#ff0033",
        "borderWidth": 3,
        "lineTension": 0
      }
    ]
  },
  "options": {
    "scales": {
      "yAxes": [
        {
          "ticks": {
            "beginAtZero": false
          }
        }
      ]
    },
    "legend": {
      "display": false
    },
    "title": {
      "display": true,
      "text": "Philips Hue Light Demo"
    },
    "tooltips": {
      "intersect": true,
      "mode": "index"
    },
    "hover": {
      "intersect": true,
      "mode": "nearest"
    }
  }
}
```

Inbound components

Inbound components route JSON messages to control points and devices.

To create a new inbound type, you extend one of the three main types: `BJsonRouter`, `BJsonSelector` or `BJsonHandler` and implement `routeValue(BString message, Context cx) throws RoutingFaile-dException`. You can create a new `RoutingFailedException` at any stage to report an error and update the lastResult slot.

When extending any of the `BJsonInbound` types, you may specify which property triggers an automatic re-routing of the last input with `Property[] getRerunTriggers()`. The helper interface `JsonKeyExtrac-tUtil` contains several methods for extracting values from a JSON payload.

Chapter 5 Components

Topics covered in this chapter

Components include services, folders and other model building blocks associated with a module. You may drag them to a **Property** or **Wire Sheet** from a palette.

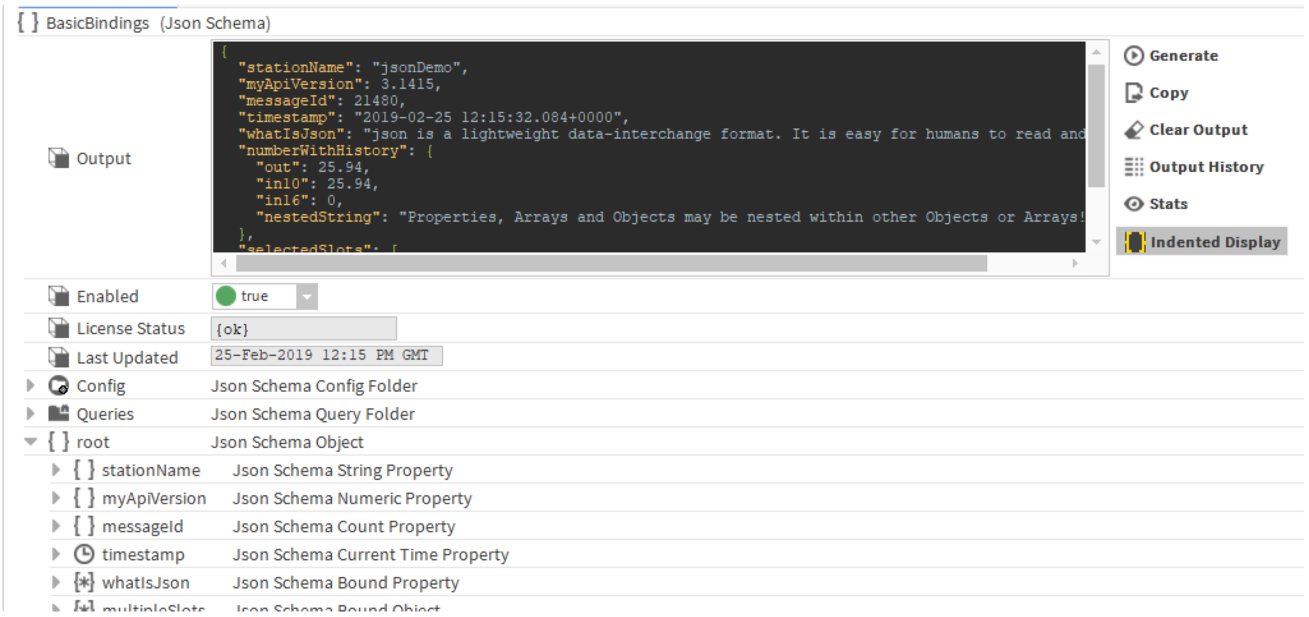
Descriptions included in the following topics appear as context-sensitive help topics when accessed by:

- Right-clicking on the object and selecting **Views→Guide Help**
- Clicking **Help→Guide On Target**

JsonSchema (Json Schema)

This component defines the schema, which includes the resulting output, configuration and queries properties, JSON entities, and actions.

Figure 28 JsonSchema properties



{ } BasicBindings (Json Schema)

Output

```
{
  "stationName": "jsonDemo",
  "myApiVersion": 3.1415,
  "messageId": 21480,
  "timestamp": "2019-02-25 12:15:32.084+0000",
  "whatIsJson": "json is a lightweight data-interchange format. It is easy for humans to read and
  "numberWithHistory": {
    "out": 25.94,
    "in10": 25.94,
    "in16": 0,
    "nestedString": "Properties, Arrays and Objects may be nested within other Objects or Arrays!
  },
  "selectedSlots": [
```

Enabled	true
License Status	{ok}
Last Updated	25-Feb-2019 12:15 PM GMT
Config	Json Schema Config Folder
Queries	Json Schema Query Folder
root	Json Schema Object
stationName	Json Schema String Property
myApiVersion	Json Schema Numeric Property
messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property
multipleSlots	Json Schema Bound Object

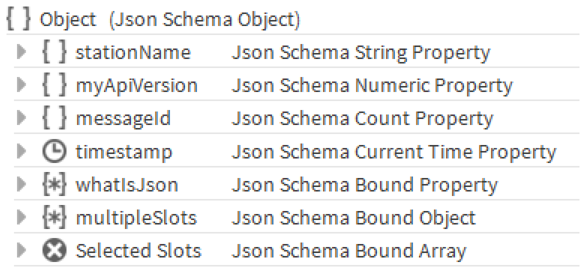
You add a schema to a station by dragging a JsonSchema from the palette to the **Config** folder in the Nav tree. From there, to access schema properties, expand the **Config** folder and double-click the schema.

Property	**Value**	**Description**
Output	container	Contains the generated JSON string.
License Status	read-only	Reports `{ok}` or `{fault}` based on the validity of the JSON license.
Last Updated	read-only	Reports when the schema was updated last.
Config	folder	Contains properties for customizing the schema.
Queries	folder	Contains the query ords.

root

This container holds JSON entities: objects, arrays, properties and bound properties.

Figure 29 root Json Schema Object



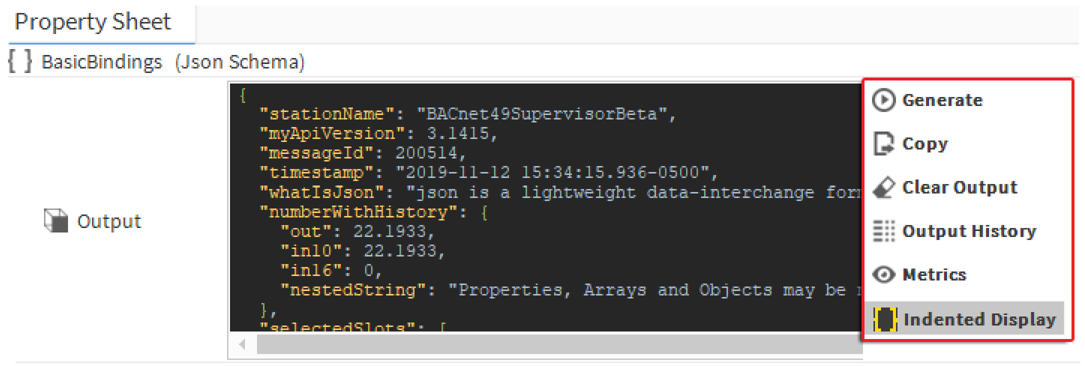
Object	(Json Schema Object)
stationName	Json Schema String Property
myApiVersion	Json Schema Numeric Property
messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property
multipleSlots	Json Schema Bound Object
Selected Slots	Json Schema Bound Array

A separate topic documents each of type of object, array and property.

Actions

These actions are available when you right-click on the **JsonSchema** or click the icons to the right of the `Output` box on the schema **Property Sheet**.

Figure 30 JSON Schema action buttons on the right side



```
{
  "stationName": "BACnet49SupervisorBeta",
  "myApiVersion": 3.1415,
  "messageId": 200514,
  "timestamp": "2019-11-12 15:34:15.936-0500",
  "whatIsJson": "json is a lightweight data-interchange for
  "numberWithHistory": {
    "out": 22.1933,
    "in10": 22.1933,
    "in16": 0,
    "nestedString": "Properties, Arrays and Objects may be 
  },
  "selectedSlots": [
```

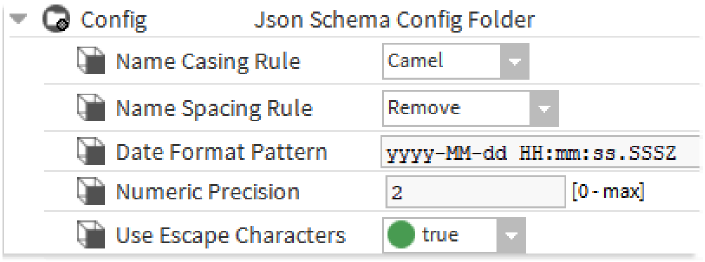
- **Generate Json** requests a rebuild and update of schema output. For relative schemas, this evaluates the `Base Query` and publishes results.
- **Force Generate Json** forces the generate action regardless of the current tuning settings.
- **Clear Cache** discards the last known values of bindings and cached query results.
- **Clear Output** sets the schema output to an empty string.
- **Execute Queries** forces an immediate execution of all the schemas queries. You can link this action to some appropriate logic to trigger execution when needed.
- **Unregister** (relative schema only) unsubscribes the registration from any base items that the relative schema monitors for updates.
- **Unsubscribe All** removes cloud registration from all export-marked entities in the station.

In addition to the standard properties (Enabled, Status, and Fault Cause), these properties are unique to the jsonSchema. The child containers are documented in following topics.

Config (Json Schema Config Folder)

This folder contains properties used to configure the entire schema.

Figure 31 Config folder properties



Config Json Schema Config Folder
Name Casing Rule: Camel
Name Spacing Rule: Remove
Date Format Pattern: yyyy-MM-dd HH:mm:ss.SSSZ
Numeric Precision: 2 [0 - max]
Use Escape Characters: true

To access these properties, expand **Config→JsonSchema**, right-click **Config** and click **Views→AX Property Sheet**.

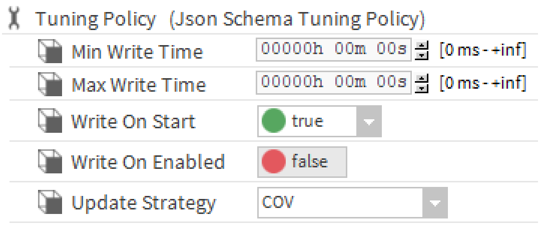
Property	**Value**	**Description**
Name Casing Rule	drop-down list (defaults to `Camel`)	Configures how the schema formats JSON keys. Establishing a standard provides naming convention uniformity. `Camel` begins key names with a lower-case letter and uses upper case to begin concatenated words (camelCaseKey). `Pascal` starts names with initial caps and concatenates all words (PascalCaseKey). `Upper` changes all letters to upper case (UPPERCASEKEY). `Lower` reduces all letters to lower case (lowercasekey). `Preserve` leaves the name unchanged as entered.
Name Spacing Rule	drop-down list (defaults to `Remove`)	Promotes uniformity by defining the use of spaces in JSON key names. `Remove` removes all spaces ("SpaceTemp" : ...). `Keep` leaves spaces unchanged ("Space Temp" : ...). `Add` injects a space between caseChanges `Hyphenate` replaces each space with a hyphen ("Space-Temp" : ...). `Underscore` replaces each space with an underscore ("Space_Temp" : ...). `URL Encode` adds a plus (+) between words ("Space+Temp" : ...).
Date Format Pattern	text	Defines a Java SimpleDateFormat pattern for the time used by the schema when it encounters AbsTime, for example, from a history query or the Current Time property. ISO 8601, for example, is `yyyy-MM-dd HH:mm:ss.SSSZ`.
Numeric Precision	number	Defines the number of decimal digits to show on exported floating point numbers. Values are rounded. Point facets are not used.
Use Escape Characters	`true` (default) or `false`	Turns on and off the use of escape characters around characters that otherwise would have special meaning. When `false`, the schema removes the escape characters it finds. For example, $20 becomes a “ ” or space character.
Tuning Policy	folder	Contains properties to configure performance.

Property	Value	Description
Overrides	folder	Contains override programs.
Debug	folder	Contains troubleshooting information.

Tuning Policy (Json Schema Tuning Policy)

These properties configure how a schema evaluates write requests and the acceptable freshness of read requests.

Figure 32 Tuning Policy properties


Tuning Policy (Json Schema Tuning Policy)
Min Write Time 00000h 00m 00s [0 ms - +inf]
Max Write Time 00000h 00m 00s [0 ms - +inf]
Write On Start true
Write On Enabled false
Update Strategy COV

To access these properties, expand **Config→JsonSchema→Config** and double–click **Tuning Policy**.

NOTE:

Clicking **Actions→Force Generate Json** overrides all tuning policy settings. Export markers applied to numeric points also have a **`CoV Tolerance`** property, which you can use to throttle output.

Property	Value	Description
Min Write Time	hours minutes seconds	Specifies the minimum amount of time allowed between schema generation, so that, for example, hundreds of concurrent CoV changes over a short time do not result in a deluge of JSON messages. The default value of zero (0) disables this rule causing all value changes to attempt to generate.
Max Write Time	hours minutes seconds	If nothing else triggers a generate, this property specifies the maximum amount of time to wait before regenerating. Any generation action resets this timer. The default value of zero (0) disables this rule resulting in no timed generation.
Write On Start	`true` (default) or `false`	Determines schema behaviour when a station starts. If `true`, a schema generation occurs when the station starts. If `false`, no generation occurs on station start.

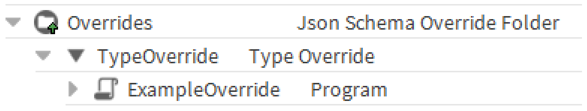
Property	Value	Description
Write On Enabled	`true` or `false` (default)	Determines schema behaviour when a status transitions from disabled to normal (enabled). If `true`, a generate occurs when the schema transitions from disabled to enabled. If `false`, no generation occurs.
Update Strategy	drop-down list	Manages the control strategy in the station. `COV` updates JSON at change of value. `On Demand Only` updates JSON only when you right-click on the schema component and click **Actions→Generate**.

Overrides (Json Schema Overrides Folder)

Configures how to convert specific data types to JSON. This definition overrides the default conversion behaviour and applies to anywhere the datatype is encountered in an entire schema.

Examples might be where facets should be replaced to a locally understood value, such as 'degC' to 'Celsius'; defining a different format for Simple types, such as Color and RelTime; or perhaps to manage expectations for +/- INF in a target platform.

Figure 33 An example of an Overrides folder



Overrides	Json Schema Override Folder
TypeOverride	Type Override
ExampleOverride	Program

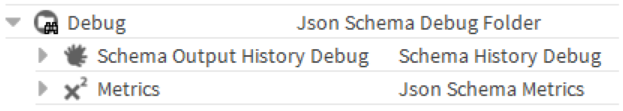
To access these slots, expand **Config→JsonSchema→Config**, right-click **Overrides** and click **Views→AX Property Sheet**.

This example contains a type override.

Debug (Json Schema Debug Folder)

This folder contains two slots. This information can help troubleshoot problems.

Figure 34 Debug containers



Debug	Json Schema Debug Folder
Schema Output History Debug	Schema History Debug
Metrics	Json Schema Metrics

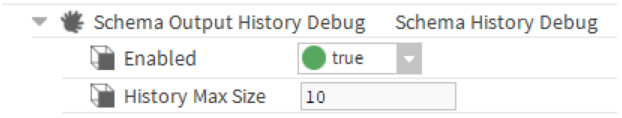
To access these containers, expand **Config→JsonSchema→Config**, right-click **Debug** and click **Views→AX Property Sheet**.

Container	Value	Description
Schema Output History Debug	Additional properties	Displays the recent history of output from a JSON schema.
Config, Debug, Metrics (JsonSchema)	read-only folder	Reports JSON statistics related to three aspects of activity: queries, data generation, and data subscription.

Schema Output History Debug (Schema History Debug)

The report this view provides lists the recent history of output from a JSON schema.

Figure 35 Schema Output History Debug properties



Schema Output History Debug	Schema History Debug
Enabled	● true
History Max Size	10

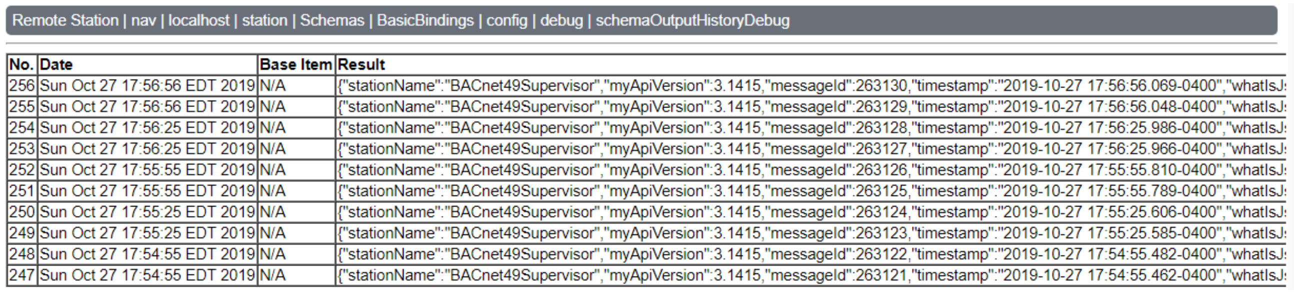
Right-clicking **Schema Output History Debug** followed by clicking **Views→Spy Local** or **Spy Remote** opens a **schemaOutputHistoryDebug** tab. This tab displays the recent history of output from the schema. This information is useful when the output updates rapidly, such as when a link calls a generate JSON in quick succession, or, in a relative schema, when output quickly changes once per base item.

In addition to the standard property (Enabled), this property supports the `Schema Output History Debug` component.

Property	Value	Description
History Max Size	number (defaults to 10 records)	Sets how many debug records to store in the station.

Debug report

Figure 36 Debug report



Remote Station | nav | localhost | station | Schemas | BasicBindings | config | debug | schemaOutputHistoryDebug

No.	Date	Base Item	Result
256	Sun Oct 27 17:56:56 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263130,"timestamp":"2019-10-27 17:56:56.069-0400","whatIsJ
255	Sun Oct 27 17:56:56 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263129,"timestamp":"2019-10-27 17:56:56.048-0400","whatIsJ
254	Sun Oct 27 17:56:25 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263128,"timestamp":"2019-10-27 17:56:25.986-0400","whatIsJ
253	Sun Oct 27 17:56:25 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263127,"timestamp":"2019-10-27 17:56:25.966-0400","whatIsJ
252	Sun Oct 27 17:55:55 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263126,"timestamp":"2019-10-27 17:55:55.810-0400","whatIsJ
251	Sun Oct 27 17:55:55 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263125,"timestamp":"2019-10-27 17:55:55.789-0400","whatIsJ
250	Sun Oct 27 17:55:25 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263124,"timestamp":"2019-10-27 17:55:25.606-0400","whatIsJ
249	Sun Oct 27 17:55:25 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263123,"timestamp":"2019-10-27 17:55:25.585-0400","whatIsJ
248	Sun Oct 27 17:54:55 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263122,"timestamp":"2019-10-27 17:54:55.482-0400","whatIsJ
247	Sun Oct 27 17:54:55 EDT 2019	N/A	{"stationName":"BACnet49Supervisor","myApiVersion":3.1415,"messageId":263121,"timestamp":"2019-10-27 17:54:55.462-0400","whatIsJ

To access this view, click the **Output History** button or right-click the `Schema Output History Debug` slot and click **Views→Spy Remote** or **Spy Local**.

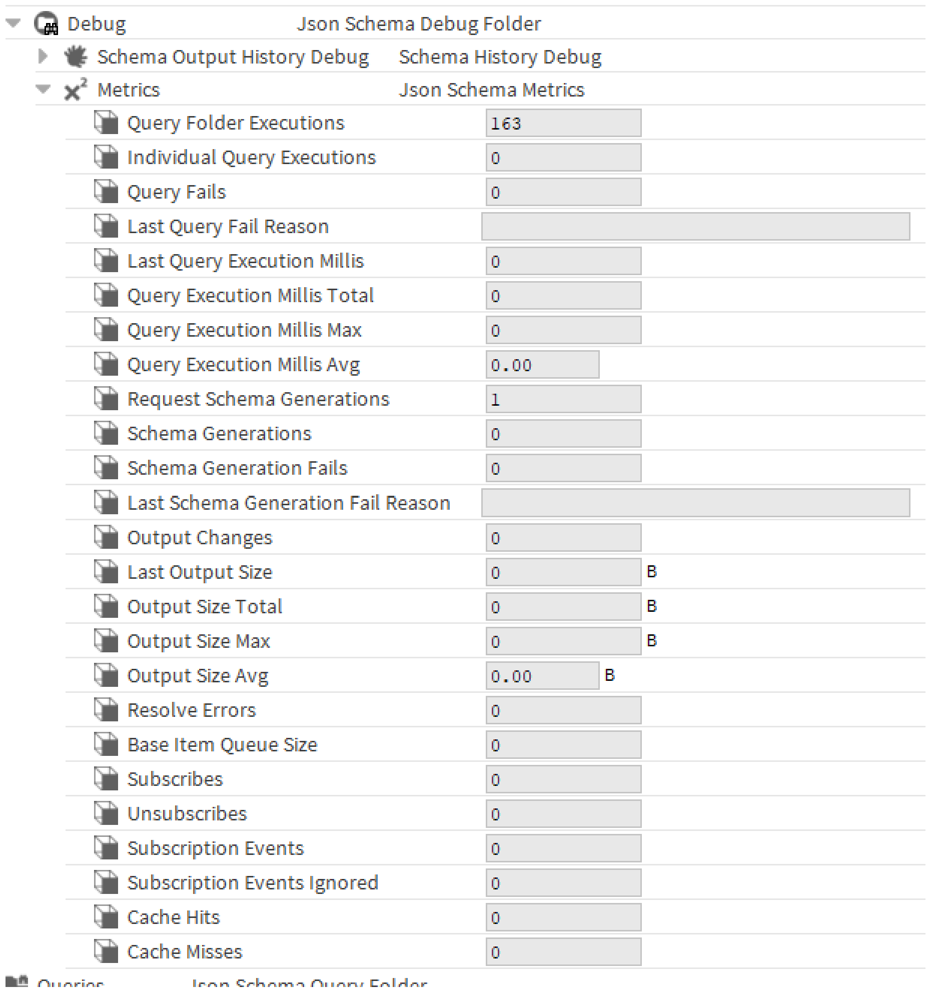
Column	Description
No.	Identifies the row. You configure the number of allowed rows by setting the `History Max Size` value on the Debug **Property Sheet**.
Date	Identifies when the history was written to the database.
Base Item	Identifies the slot from which the system generated the JSON.
Result	Shows the JSON payload.

Metrics (Json Schema Metrics)

This folder exposes schema generation, query execution and CoV subscription metrics. If needed, you can log or link individual metric values to generate alarms.

`Metrics` help with determining sizing and provisioning capacity on a cloud platform by estimating the traffic a station is likely to generate with a given schema. They may also assist in identifying performance problems. To assist debugging, use the reset action.

Figure 37 Metrics as reported from the schema



Debug	Json Schema Debug Folder
Schema Output History Debug	Schema History Debug
Metrics	Json Schema Metrics
Query Folder Executions	163
Individual Query Executions	0
Query Fails	0
Last Query Fail Reason	
Last Query Execution Millis	0
Query Execution Millis Total	0
Query Execution Millis Max	0
Query Execution Millis Avg	0.00
Request Schema Generations	1
Schema Generations	0
Schema Generation Fails	0
Last Schema Generation Fail Reason	
Output Changes	0
Last Output Size	0 B
Output Size Total	0 B
Output Size Max	0 B
Output Size Avg	0.00 B
Resolve Errors	0
Base Item Queue Size	0
Subscribes	0
Unsubscribes	0
Subscription Events	0
Subscription Events Ignored	0
Cache Hits	0
Cache Misses	0
Queries	Json Schema Query Folder

To access these metrics, expand **Config→JsonSchema→Config→Debug** and double–click **Metrics**.

The metrics provide three categories of performance information: query performance, generate performance, and subscription performance.

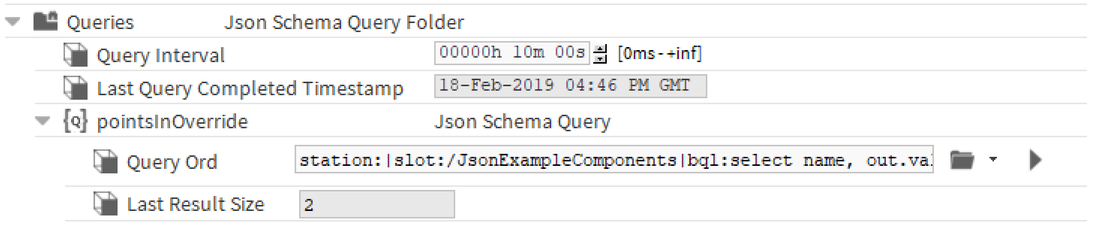
Queries	Generation	Subscription
Query Folder Executions	Request Schema Generations	Subscribes
Individual Query Executions	Schema Generations	Unsubscribes
Query Fails	Schema Generation Fails	Subscription Events
Last Query Fail Reason	Last Schema Generation Fail Reason	Subscription Events Ignored
Last Query Execution Millis	Output Changes	Cache Hits
Query Execution Millis Total	Last Output Size	Cache Misses
Query Execution Millis Max	Output Size Total	
Query Execution Millis Avg	Output Size Max	
	Output Size Avg	
	Resolve Errors	

Most metrics are self-explanatory. Execution millis report the number of milliseconds spent performing a query. Cache hits indicate the number of schema string generations that found a cached value for a binding. Cache misses indicate the number of schema string generations that found no cached value for a binding.

Queries (Json Schema Query Folder)

This folder under a JSON schema stores search queries whose results are then available to be used by the schema. Queries generate JSON payloads from the results of bql or neql searches. For example, a query may include a report of overridden points, active alarms, or history logs for a given point.

Figure 38 Queries folder properties



Queries Json Schema Query Folder
Query Interval 00000h 10m 00s [0ms-+inf]
Last Query Completed Timestamp 18-Feb-2019 04:46 PM GMT
pointsInOverride Json Schema Query
Query Ord station:|slot:/JsonExampleComponents|bql:select name, out.va
Last Result Size 2

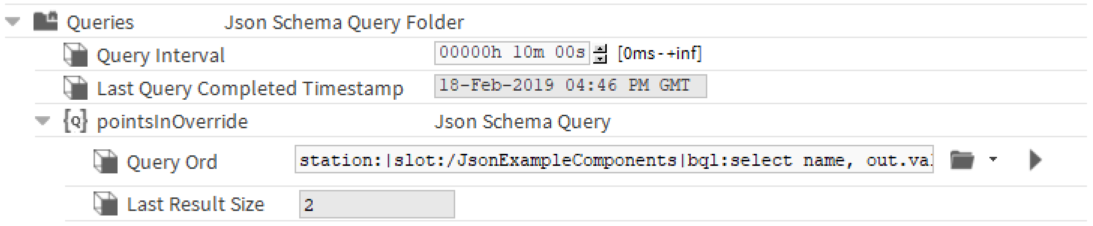
To access these properties, expand **Config→JsonSchema**, right-click **Queries** and click **Views→Ax Property Sheet**.

Property	Value	Description
Query Interval	time	Defines how often the schema executes its queries, which determines how up-to-date exported data are when the schema uses an Update Strategy of COV. If multiple queries exist, each time the schema executes it runs each query in parallel.
Last Query Completed Timestamp	read-only (defaults to null)	Reports the time the last query completed.
Queries, queries-MaxExecutionTime (hidden property on the Queries folder)	time	Increases the amount of time granted to complete all queries on each cycle. Failure to complete within this time causes the schema generation to fail.

Query (Json Schema Query)

This JSON entity sets up a database search. A query can be any valid transform, neql or bql statement, which returns a BITable.

Figure 39 Query properties



Queries Json Schema Query Folder
Query Interval 00000h 10m 00s [0ms-+inf]
Last Query Completed Timestamp 18-Feb-2019 04:46 PM GMT
pointsInOverride Json Schema Query
Query Ord station:|slot:/JsonExampleComponents|bql:select name, out.va
Last Result Size 2

To add a query to a schema (**JsonSchema** or **RelativeJsonSchema**), expand the **Query** folder in the palette and drag a query to the **Queries** folder in the schema.

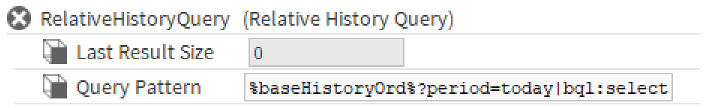
You access query properties by double-clicking the **JsonSchema** or **RelativeJsonSchema** node in the Nav tree, expanding the `Queries` folder followed by expanding the query itself.

Property	Value	Description
Query Ord	ord (defaults to null)	Identifies the target object of the query.
Last Result Size	read-only (defaults to 0)	Reports the size of the query result the last time the framework executed the search.

RelativeHistoryQuery (Relative History Query)

This query works in conjunction with a `RelativeJsonSchema`.

Figure 40 RelativeHistoryQuery properties



Last Result Size	0
Query Pattern	%baseHistoryOrd%?period=today|bql:select

You add a RelativeHistoryQuery under the `Queries` folder in the **RelativeJsonSchema**. You access these properties by double-clicking the **RelativeJsonSchema** node in the Nav tree and expanding the `Queries` folder.

Property	Value	Description
Last Result Size	read-only (defaults to 0)	Reports the size of the query result the last time the framework executed the search.
Query Pattern	bql	Prepends to a bql query so query data can be included in the payload for a given set of points or devices. For example: `%baseHistoryOrd%?period=today|bql: select timestamp, value`

Example

Here is an example of how to use the `Query Pattern` property to pre-pend the current base item to a bql query. This example includes query data in the payload for a given set of points or devices:

```
%baseHistoryOrd%?period=today|bql:select timestamp, value
```

You may use this with a base query to return a HistoryConfig or a HistoryExt (or the parent of these slots):

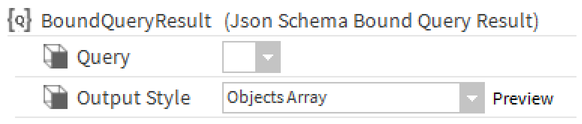
```
station:|slot:/JsonExampleComponents|bql:select * from history:HistoryConfig
```

CAUTION: When creating queries, bear in mind the potential performance implications of running queries frequently. To reduce the scope of the query, focus the first part of the ord on the location where the data are likely to be found, or use the stop keyword to prevent depth recursion.

BoundQueryResult (Json Schema Bound Query Result)

This entity determines where and how to insert the results of a query in the payload.

Figure 41 BoundQueryResult properties



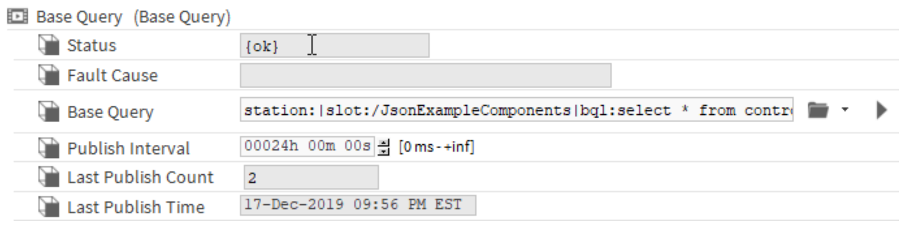
Query	
Output Style	Objects Array

To add this component, expand the **Query** folder in the palette and drag a **`BoundQueryResult`** to the root JSON schema **Object** of a relative JSON schema.

Property	**Value**	**Description**
Query	drop-down list	Associates this query result with a query ord as defined by a query under the **Queries** folder. This folder can contain multiple queries.
Output Style	drop-down list	Defines the output style to render the query in.

Base Query (Base Query)

A base query feeds base components to a schema, which the query resolves against the schema one at a time. When used with a relative schema, the base query allows you to limit the scope of your query and to scale within that as you add new points or components.



Base Query (Base Query)	
Status	{ok}
Fault Cause	
Base Query	station:|slot:/JsonExampleComponents|bql:select * from contr
Publish Interval	00024h 00m 00s [0 ms-+inf]
Last Publish Count	2
Last Publish Time	17-Dec-2019 09:56 PM EST

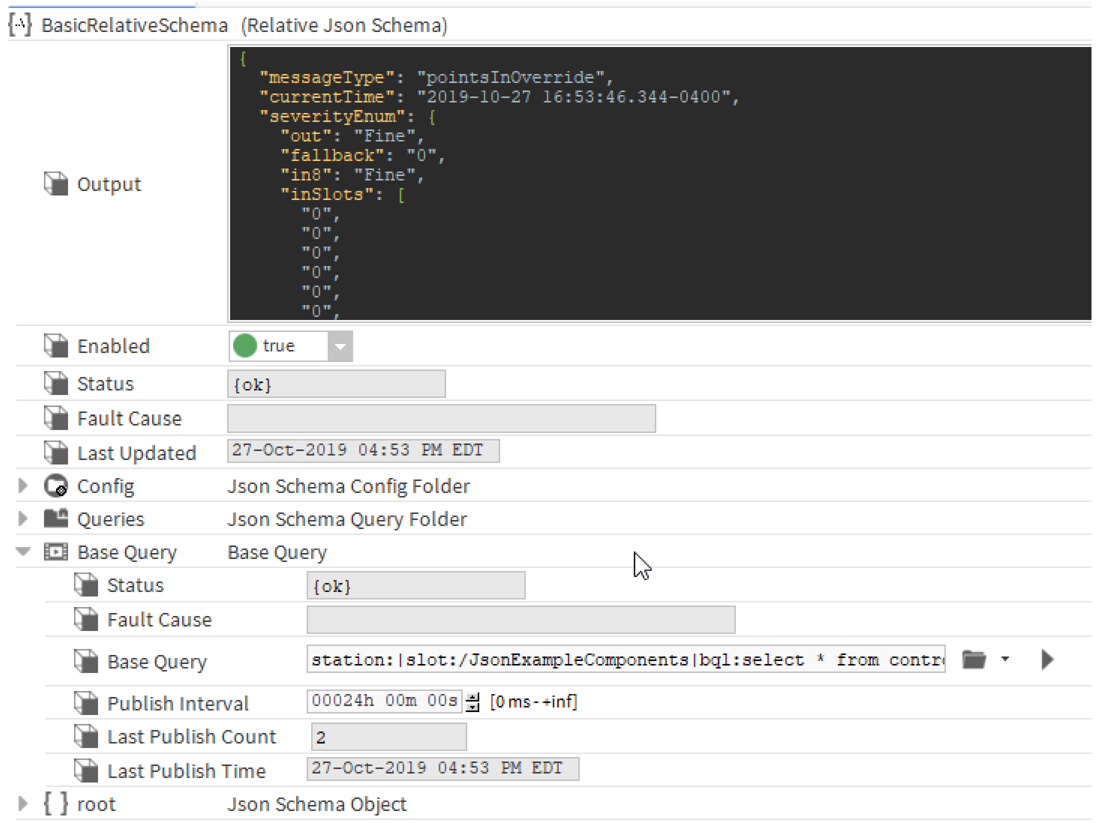
The **`Base Query`** component is located in the palette as part of any of the relative schema components (for example, **`BasicRelativeSchema`**, **`RelativeHistorySchema`**, and others).
In addition to the standard properties (Status and Fault Cause), these properties support the **`Base Query`**.

Property	**Value**	**Description**
Base Query	text	Defines the scope of the query.
Publish Interval	hours, minutes, seconds	Specifies the amount of time between query executions. It triggers a complete publish output (of every returned component) at the interval selected.
Last Publish Count	read-only	Indicates the number of times the query executed.
Last Publish Time	read-only	Indicates the last time the query executed.

RelativeJsonSchema (Relative Json Schema)

This schema enables the scaling of JSON payload generation, which provides faster processing than the speed available using multiple simple schemata.

Figure 42 RelativeJsonSchema properties


BasicRelativeSchema (Relative Json Schema)

Output

```
{
  "messageType": "pointsInOverride",
  "currentTime": "2019-10-27 16:53:46.344-0400",
  "severityEnum": {
    "out": "Fine",
    "fallback": "0",
    "in8": "Fine",
    "inSlots": [
      "0",
      "0",
      "0",
      "0",
      "0",
      "0",
```

Enabled	true
Status	{ok}
Fault Cause	
Last Updated	27-Oct-2019 04:53 PM EDT
Config	Json Schema Config Folder
Queries	Json Schema Query Folder
Base Query	Base Query
Status	{ok}
Fault Cause	
Base Query	station:|slot:/JsonExampleComponents|bql:select * from contr
Publish Interval	00024h 00m 00s [0 ms-+inf]
Last Publish Count	2
Last Publish Time	27-Oct-2019 04:53 PM EDT
root	Json Schema Object

You add a relative schema to a station by dragging a `RelativeJsonSchema` from the palette to the **Config** folder in the Nav tree. From there, to access schema properties, expand the **Config** folder and double-click the schema.

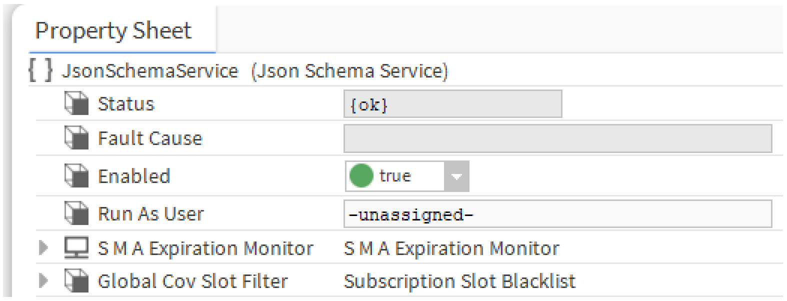
In addition to the standard properties (Enabled, Status, and Fault Cause), these properties are unique to JSON.

Property	Value	Description
Last Updated	read-only (defaults to null)	Reports when the relative schema was updated last.
Config	folder	Contains properties for configuring the relative schema.
Queries	folder	Contains the search arguments.
Base Query	additional properties	Defines a query that is intended to resolve targets in the station one at a time. An example might be all BACnet devices. The base query returns the objects that the schema resolves against. The schema objects (below the query) then pick out appropriate values.

JsonSchemaService (Json Schema Service)

This service supports JSON functionality and provides some station global filtering.

Figure 43 JsonSchemaService properties



Property Sheet	
{ } JsonSchemaService (Json Schema Service)	
Status	{ok}
Fault Cause	
Enabled	● true
Run As User	-unassigned-
S M A Expiration Monitor	S M A Expiration Monitor
Global Cov Slot Filter	Subscription Slot Blacklist

You access these properties by double-clicking the **JsonSchemaService** under the **Config→Services** folder in the Nav tree.

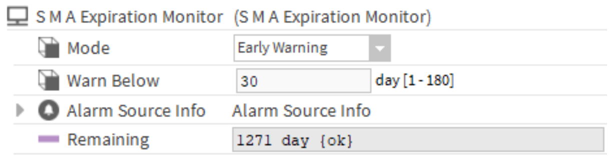
In addition to the standard properties (Status, Fault Cause, and Enabled), the following properties are unique to the JsonSchemaService:

Property	Value	Description
Run As User	text	Specifies the user account to assume in the event that a router processes an incoming change. This is mandatory when using the **`SetPointHandler`**, for example, so that the framework can limit any changes triggered by a cloud platform to areas where the platform should have write access within the station. This setting is also optionally used for JSON schema export data. This property is important for security. Only a super user can configure it. The framework requires it for incoming data used to update a **`SetPointHandler`**. The set operation succeeds only if a real user with operator-write permission on the slot issues the incoming JSON. This property is optional when exporting JSON with a schema. When set, the data value of the exported slot defaults to an empty string unless Run As User is a real user with operator-read permission on the slot.
S M A Expiration Monitor	additional properties	Configures a reminder of when the framework Software Maintenance Agreement is about to expire.
Global Cov Slot Filter	Additional properties	Provides some station global filtering by identifying which slots should be ignored when subscribed to bound values. The default list of slots includes a good example of why this function is necessary in that changes to a component's **`wsAnnotation`** property (which details the position and size of a component glyph on the **Wire Sheet**), should generally be excluded from the changes of value reported to any upstream consumer of data.

S M A Expiration Monitor (S M A Expiration Monitor)

Given the JSON Toolkit's requirement for active maintenance (SMA) on non-demo licenses, this monitor increasingly notifies you as the license expiration date approaches. It runs on startup, then every 24 hours since the last check to establish if the expiration date is within the warning period or expired, and generates an offNormal or Fault alarm accordingly.

Figure 44 S M A Expiration Monitor properties



S M A Expiration Monitor	(S M A Expiration Monitor)
Mode	Early Warning
Warn Below	30 day [1 - 180]
Alarm Source Info	Alarm Source Info
Remaining	1271 day {ok}

To configure these properties, expand **Config→Services**, double-click **JsonSchemaService** and expand **S M A Expiration Monitor**.

Although the alarms are likely the most accessibly notification method, the SMA monitor also logs messages to the station console and exposes the days remaining as a slot, which can be shown, for example, on a dashboard.

The station itself has an **`S M A Notification Setting`** under the **UserService**, which alerts you at the web login screen.

As an extension of S M A requires a reboot to install the new license, the monitor performs no further checks, once it detects an expired license, until the station starts again.

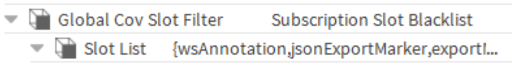
In addition to the standard **`Alarm Source Info`** properties, these properties are unique to the JSON Toolkit:

Property	Value	Description
Mode	drop-down list (defaults to `Early Warning`)	Configures when to activate an alarm regarding a pending license expiration. `Early Warning` generates an alarm before the license expires. `Once Expired` generates an alarm when the license expires and thereafter. `Disable Monitor` turns monitoring off.
Warn below	number of days from 1 to 180 (defaults to 30 days)	Configures when to start warning of the license expiration.
Remaining	read-only	Displays the number of days before the license expires.

Global Cov Slot Filter (Subscription Slot Blacklist)

This filter denotes which slots to ignore when subscribed to bound values.

Figure 45 Subscription Slot Blacklist



Global Cov Slot Filter	Subscription Slot Blacklist
Slot List	{wsAnnotation,jsonExportMarker,exportI...

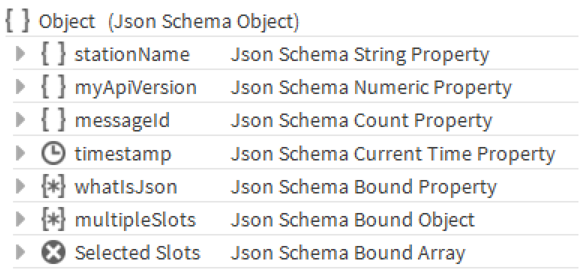
You access this list by expanding **Config→Services→JsonSchemaService** and double-clicking **Global Cov Slot Filter**.

The default list includes a good example of why this function is necessary, in that changes to a component's **`wsAnnotation`** property (which details the position and size of a component glyph on the Wire Sheet) should, generally, be excluded from the changes of value reported to any upstream consumer of data.

Object (Json Schema Object)

This is an empty, named container that holds the other schema entities, which set up the JSON payload.

Figure 46 Example of an object container with JSON entities



Object (Json Schema Object)

stationName	Json Schema String Property
myApiVersion	Json Schema Numeric Property
messageId	Json Schema Count Property
timestamp	Json Schema Current Time Property
whatIsJson	Json Schema Bound Property
multipleSlots	Json Schema Bound Object
Selected Slots	Json Schema Bound Array

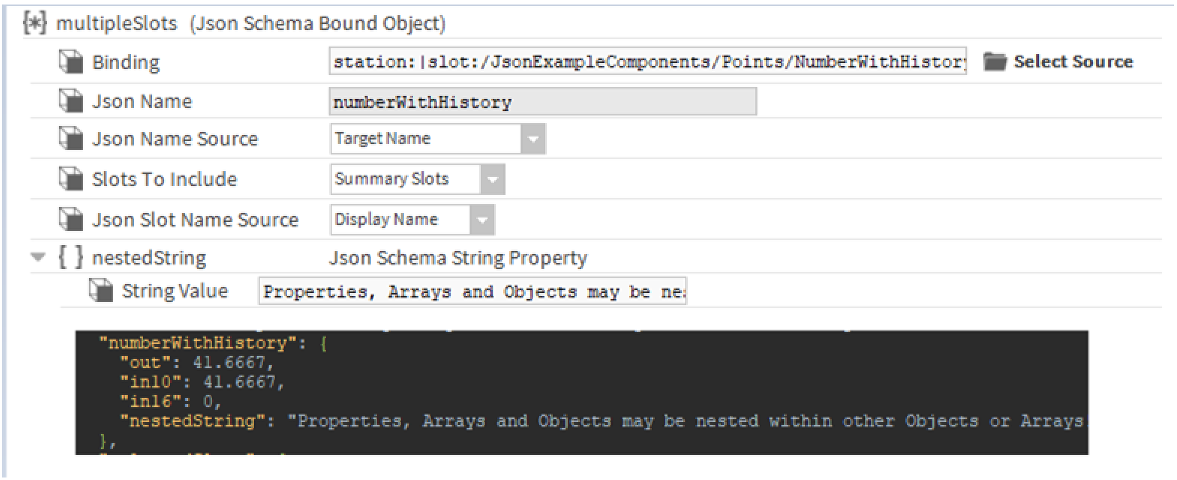
To add the root object to a schema, expand the **Objects** folder in the palette and drag an **Object** to the **JsonSchema** folder. To add another object to the schema, drag an **Object** from the palette to the root **Object** container under the schema.

An object is a container. It has no properties of its own or additional containers. Inside this container, the JSON objects and properties model the structure of the JSON message underneath the schema object. If you nest items in this container within each other in a tree structure, they will appear nested in the JSON string.

BoundObject (Json Schema Bound Object)

This entity is a named JSON object whose child name and value pairs are the slots within a target ord.

Figure 47 Example of a BoundObject



multipleSlots (Json Schema Bound Object)

Binding	station:|slot:/JsonExampleComponents/Points/NumberWithHistor
Json Name	numberWithHistory
Json Name Source	Target Name
Slots To Include	Summary Slots
Json Slot Name Source	Display Name
nestedString	Json Schema String Property
String Value	Properties, Arrays and Objects may be ne

```
"numberWithHistory": {
  "out": 41.6667,
  "in10": 41.6667,
  "in16": 0,
  "nestedString": "Properties, Arrays and Objects may be nested within other Objects or Arrays
},
```

To add a bound object to a schema, expand the **Objects** folder in the palette and drag a **`BoundObject`** to the schema folder, then double-click the bound object.

Property	Value	Description
Binding	ord, bql, neql, absolute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Json Name	read-only	Displays the name defined by the **`Json Name Source`**.
Json Name Source	drop-down list	Selects a name for the source object based on how it is defined elsewhere. Options are: `Display Name` is an explicitly-assigned name for the object. `Target Name`

Property	Value	Description

		`Target Display Name`
		`Target Parent Name`
		`Target Path` displays the ord for the object rather than a name.

Slots To Include	dop-down lists	Identifies which slots from the target to include in the resultant JSON. Options are: `All Slots` reports data from all slots in the target object. `All Visible Slots` excludes hidden slots. `All Summary Slots` includes only those with the summary flag set. `Selected Slots` manually selects slots from a list.
Json Slot Name Source	drop-down list	Selects the name for a specific source slot. Options are: `Display Name` is an explicitly-assigned name for the slot. `Name` selects an alternate name.

Array (Json Schema Array)

This is an empty, named container for other schema entities, which set up the JSON payload.

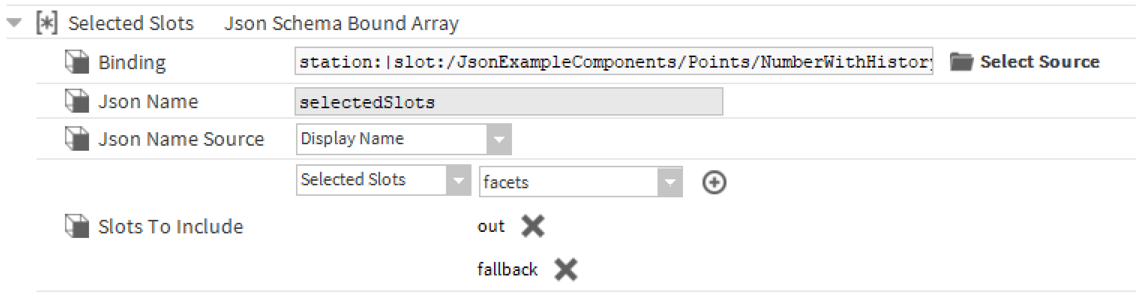
To add an array to a schema object, expand the **Arrays** folder in the palette and drag an **`Array`** to the root object folder in a schema.

An array has no properties of its own or additional containers.

BoundArray (Json Schema Bound Array

This is an empty named container for other schema entities.

Figure 48 BoundArray properties


Selected Slots Json Schema Bound Array
Binding station:|slot:/JsonExampleComponents/Points/NumberWithHistor
Json Name selectedSlots
Json Name Source Display Name
Slots To Include Selected Slots facets out fallback

To add a bound array to a schema object, expand the **Arrays** folder in the palette and drag a **`BoundArray`** to the root object folder in a schema, then double-click the **`BoundArray`** component.

Property	Value	Description
Binding	ord, bql, neql, ab-solute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Json Name	read-only	Displays the name defined by the **`Json Name Source`**.

Property	Value	Description

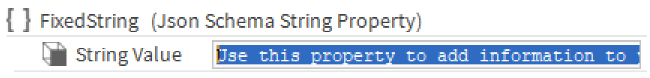
Json Name Source	drop-down list	Selects a name for the source object based on how it is defined elsewhere. Options are:
		`Display Name` is an explicitly-assigned name for the object.
		`Target Name`
		`Target Display Name`
		`Target Parent Name`
		`Target Path` displays the ord for the object rather than a name.

Slots To Include	drop-down lists	Identifies which slots from the target to include in the resultant JSON. Options are: `All Slots` reports data from all slots in the target object. `All Visible Slots` excludes hidden slots. `All Summary Slots` includes only those with the summary flag set. `Selected Slots` manually selects slots from a list.

FixedString (Json Schema String Property)

This property inserts a string value into the JSON payload.

Figure 49 FixedString property



String Value `Use this property to add information to`

To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`FixedString`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`Fixed-String`** component.

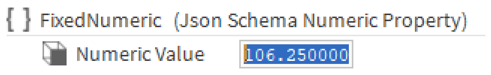
Fixed properties, such as names, appear as constants.

You can link in to these if you expect a name to vary. JSON includes the current value during the next generation event triggered by a CoV on a bound entity or by the invocation of the `Generate` action. Changing the value of a fixed property does not trigger a CoV generation event the same way that a bound equivalent does.

FixedNumeric (Json Schema Numeric Property)

This property inserts a fixed numeric value.

Figure 50 FixedNumeric property



Numeric Value `106.250000`

To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`FixedNumeric`** to the root object **`{ }`** or to an array **`[ ]`** under the root object. To configure its property, double-click it.

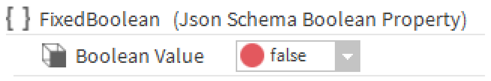
You can link in to this value if you expect it to vary. The next generation event includes the current value triggered by CoV on a bound entity or by the invocation of the `Generate` action. A change in the value of any fixed property does not trigger a CoV generation event in the way that a bound equivalent would.

Property	Value	Description
Numeric Value	single-digit number to six decimal places (defaults to 0.000000)	Sets up a numeric value.

FixedBoolean (Json Schema Boolean Property)

This property inserts a fixed Boolean value, which defaults to `false`.

Figure 51 Fixed Boolean property


{ } FixedBoolean (Json Schema Boolean Property)

Boolean Value: false

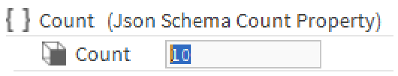
To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`FixedBoolean`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`Fixed-Boolean`** component.

You can link in to this value if you expect it to vary. The next generation event includes the current value triggered by CoV on a bound entity or by the invocation of the `Generate` action. A change in the value of any fixed property does not trigger a CoV generation event in the way that a bound equivalent would.

Count (Json Schema Count Property)

This fixed property defines a named value that increments by one each time the schema generates. You could use this property for message IDs.

Figure 52 Count property


{ } Count (Json Schema Count Property)

Count: 10

To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`Count`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`Count`** component.

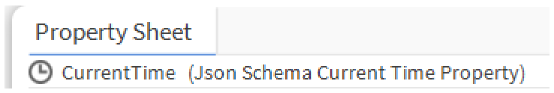
This property is a number that defaults to zero (0).

To return this value to zero, right-click the **`Count`** property and click **Actions→Reset**.

CurrentTime (Json Schema Current Time Property)

This fixed property inserts the current time as defined by the **`Date Format Pattern`** in the JSON schema object.

Figure 53 CurrentTime property


Property Sheet

CurrentTime (Json Schema Current Time Property)

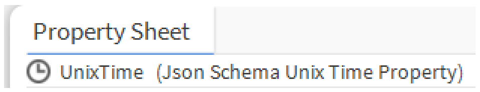
To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`CurrentTime`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`Cur-rentTime`** component.

The format for the current time is: `year-month-day hour:minute:second`

UnixTime (Json Schema Unix Time Property)

This fixed property inserts the current time as Unix time. This system for identifying a point in time is the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970. It is widely used in systems that run the Unix operating system.

Figure 54 UnixTime property

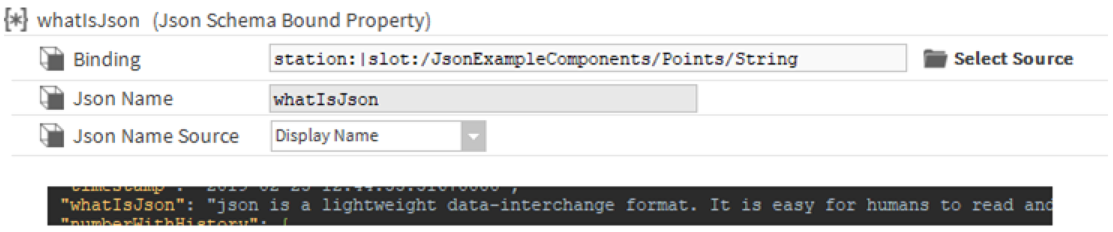


To add this property to a schema object or array, expand the **Properties** folder in the palette and drag a **`UnixTime`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`UnixTime`** component.

BoundProperty (Json Schema Bound Property)

This property inserts the current value of the object specified by the **`Binding`** property.

Figure 55 BoundProperty in a JsonSchema and RelativeJsonSchema

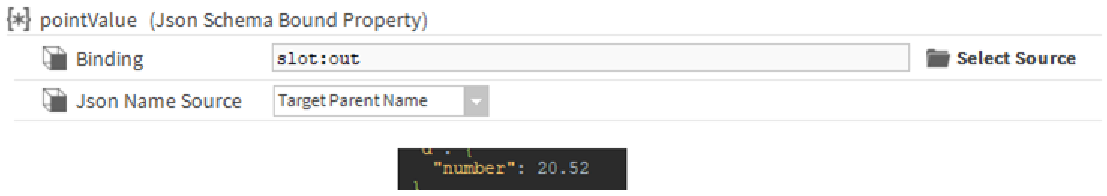


whatIsJson (Json Schema Bound Property)

Binding	station:|slot:/JsonExampleComponents/Points/String	Select Source
Json Name	whatIsJson	
Json Name Source	Display Name	

```
"whatIsJson": "json is a lightweight data-interchange format. It is easy for humans to read anc
```

Relative Schema



pointValue (Json Schema Bound Property)

Binding	slot:out	Select Source
Json Name Source	Target Parent Name	

```
"number": 20.52
```

To add a bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`BoundProperty`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`BoundProperty`** component.

Property	Value	Description
Binding	ord, bql, neql, absolute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Json Name	read-only	Displays the name defined by the **`Json Name Source`**.

Json Name Source	drop-down list	Selects a name for the source object based on how it is defined elsewhere. Options are:
		`Display Name` is an explicitly-assigned name for the object.
		`Target Name`
		`Target Display Name`
		`Target Parent Name`
		`Target Path` displays the ord for the object rather than a name.

BoundCSVProperty (Json Schema Bound Csv Property)

This bound property is a named JSON string, which renders child slots as a string, comma separated list (with no surrounding [] or {}).

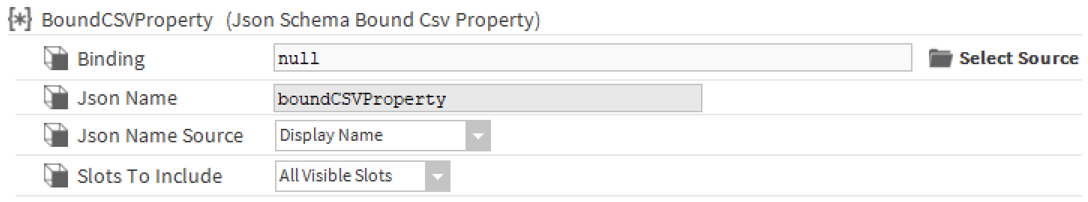


Figure 56 BoundCSVProperty properties

BoundCSVProperty (Json Schema Bound Csv Property)

Binding	null	Select Source
Json Name	boundCSVProperty	
Json Name Source	Display Name	
Slots To Include	All Visible Slots	

To add this bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`BoundCSVProperty`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the **`BoundCSVProperty`** component.

Property	Value	Description
Json Name	read-only	Displays the name defined by the **`Json Name Source`**.

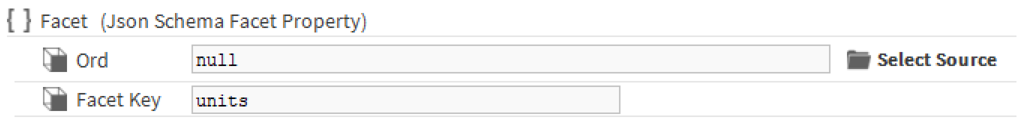
Json Name Source	drop-down list	Selects a name for the source object based on how it is defined elsewhere. Options are:
		`Display Name` is an explicitly-assigned name for the object.
		`Target Name`
		`Target Display Name`
		`Target Parent Name`
		`Target Path` displays the ord for the object rather than a name.

Slots To Include	dop-down lists	Identifies which slots from the target to include in the resultant JSON. Options are: `All Slots` reports data from all slots in the target object. `All Visible Slots` excludes hidden slots. `All Summary Slots` includes only those with the summary flag set. `Selected Slots` manually selects slots from a list.

Facet (Json Schema Facet Property)

This bound property defines a single facet value from a bound component to insert in the schema output, for example the units of the current point.

Figure 57 Facet property


{ } Facet (Json Schema Facet Property)

Ord: null

Facet Key: units

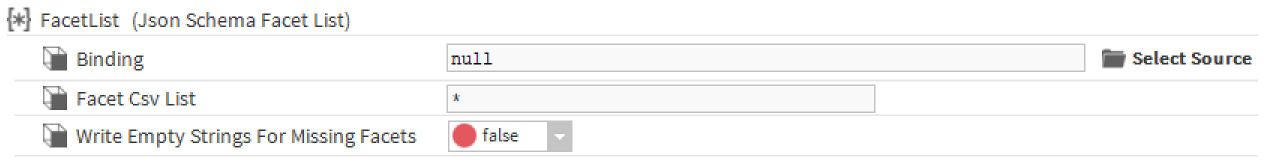
To add this bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`Facet`** to the root object **`{ }`** or to an array **`[ ]`** under the root object, then double-click the bound component.

Property	Value	Description
Ord	ord	Selects the ord to the component with the facet applied.
Facet Key	text	Defines the name of a facet. Facet keys should be added as follows: units, mix, max.

FacetList (Json Schema Facet List)

This bound property inserts a list of name/value facet properties based upon a comma separated list or * for all.

Figure 58 FacetList bound property


FacetList (Json Schema Facet List)

Binding	null	Select Source
Facet Csv List	*	
Write Empty Strings For Missing Facets	false	

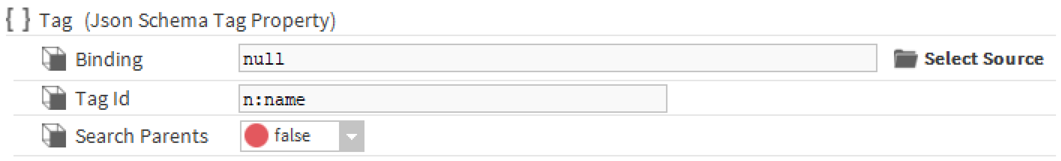
To add this bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`FacetList`** to an object `{ }` or to an array `[ ]` under the root object, then double-click the bound component.

Property	**Value**	**Description**
Binding	ord, bql, neql, absolute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Facet Csv List	text (defaults to * for all)	Inserts a list of name and value facet property pairs based upon a comma-separated list or asterisk (*) for all. Add facet keys as follows: units,mix,max.
Write Empty Strings For Missing Facets	`true` or `false` (default)	Determines if the JSON outputs an empty string when facets are missing.

Tag (Json Schema Tag Property)

This bound property inserts a single tag value from the bound component into the output.

Figure 59 Tag bound property


Tag (Json Schema Tag Property)

Binding	null	Select Source
Tag Id	n:name	
Search Parents	false	

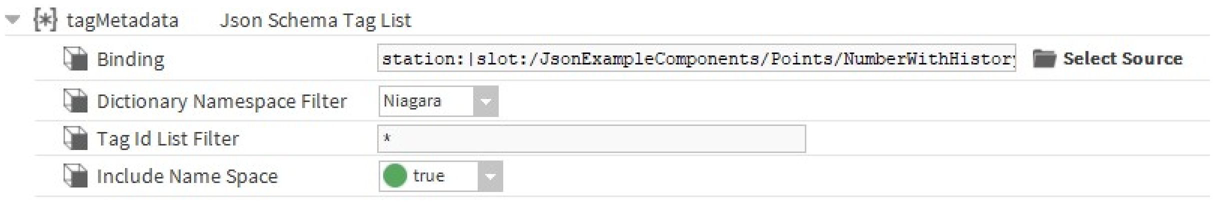
To add this bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`Tag`** to an object `{ }` or to an array `[ ]` under the root object, then double-click the bound component.

Property	**Value**	**Description**
Binding	ord, bql, neql, absolute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Tag Id	tag syntax (n:name)	Identifies a tag to use in the binding search.
Search Parents (Tag)	`true` or `false` (default)	Configures the search to include parent tags. If the search does not find a tag on the binding target, this property, when set to `true`, searches up the hierarchy for the closest component with a matching tag id.

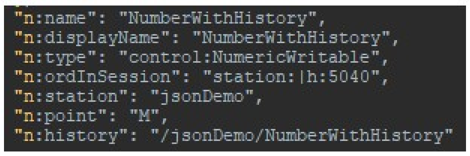
TagList (Json Schema Tag List)

This bound property defines a list of name/value properties based upon selected tags found upon a binding target.

Figure 60 TagList properties



tagMetadata	Json Schema Tag List
Binding	station:|slot:/JsonExampleComponents/Points/NumberWithHistory
Dictionary Namespace Filter	Niagara
Tag Id List Filter	*
Include Name Space	true



```
"n:name": "NumberWithHistory",
"n:displayName": "NumberWithHistory",
"n:type": "control:NumericWritable",
"n:ordInSession": "station:|h:5040",
"n:station": "jsonDemo",
"n:point": "M",
"n:history": "/jsonDemo/NumberWithHistory"
```

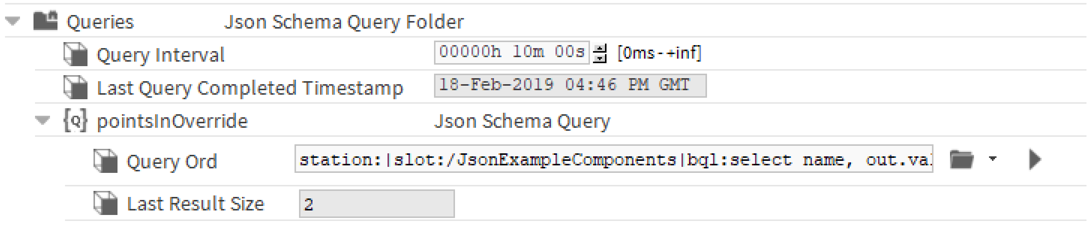
To add this bound property to a schema object or array, expand the **BoundProperties** folder in the palette and drag a **`TagList`** to the root object **{ }** or to an array **[ ]** under the root object, then double-click the bound component.

Property	Value	Description
Binding	ord, bql, neql, absolute path	Establishes a relationship between a target object, such as a point, slot, component, tag, etc. and its representation in the framework.
Dictionary Namespace Filter	drop-down list	Limits the search based on a tag dictionary name.
Tag Id List Filter	text	Identifies a comma-separated list to limit the tags to be included in the output. For example, n:name,n:type or * for all. If **`Include Name Space`** is set to `true`, the schema adds the tag dictionary prefix to the key (for example, hs:hvac).
Include Name Space	`true` (default) or `false`	Configures the search to include the tag dictionary prefix in the key.

Query (Json Schema Query)

This JSON entity sets up a database search. A query can be any valid transform, neql or bql statement, which returns a BITable.

Figure 61 Query properties



Queries	Json Schema Query Folder
Query Interval	00000h 10m 00s [0ms-+inf]
Last Query Completed Timestamp	18-Feb-2019 04:46 PM GMT
pointsInOverride	Json Schema Query
Query Ord	station:|slot:/JsonExampleComponents|bql:select name, out.va
Last Result Size	2

To add a query to a schema (**JsonSchema** or **RelativeJsonSchema**), expand the **Query** folder in the palette and drag a query to the **Queries** folder in the schema.

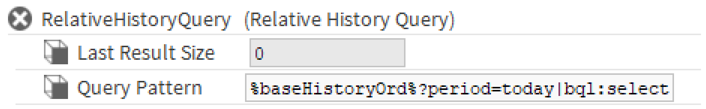
You access query properties by double-clicking the **JsonSchema** or **RelativeJsonSchema** node in the Nav tree, expanding the `Queries` folder followed by expanding the query itself.

Property	Value	Description
Query Ord	ord (defaults to null)	Identifies the target object of the query.
Last Result Size	read-only (defaults to 0)	Reports the size of the query result the last time the framework executed the search.

RelativeHistoryQuery (Relative History Query)

This query works in conjunction with a `RelativeJsonSchema`.

Figure 62 RelativeHistoryQuery properties


RelativeHistoryQuery (Relative History Query)
Last Result Size 0
Query Pattern %baseHistoryOrd%?period=today|bql:select

You add a RelativeHistoryQuery under the `Queries` folder in the **RelativeJsonSchema**. You access these properties by double-clicking the **RelativeJsonSchema** node in the Nav tree and expanding the `Queries` folder.

Property	Value	Description
Last Result Size	read-only (defaults to 0)	Reports the size of the query result the last time the framework executed the search.
Query Pattern	bql	Prepends to a bql query so query data can be included in the payload for a given set of points or devices. For example: `%baseHistoryOrd%?period=today|bql: select timestamp, value`

Example

Here is an example of how to use the `Query Pattern` property to pre-pend the current base item to a bql query. This example includes query data in the payload for a given set of points or devices:

```
%baseHistoryOrd%?period=today|bql:select timestamp, value
```

You may use this with a base query to return a HistoryConfig or a HistoryExt (or the parent of these slots):

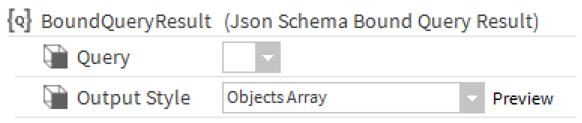
```
station:|slot:/JsonExampleComponents|bql:select * from history:HistoryConfig
```

CAUTION: When creating queries, bear in mind the potential performance implications of running queries frequently. To reduce the scope of the query, focus the first part of the ord on the location where the data are likely to be found, or use the stop keyword to prevent depth recursion.

BoundQueryResult (Json Schema Bound Query Result)

This entity determines where and how to insert the results of a query in the payload.

Figure 63 BoundQueryResult properties


BoundQueryResult (Json Schema Bound Query Result)
Query
Output Style Objects Array

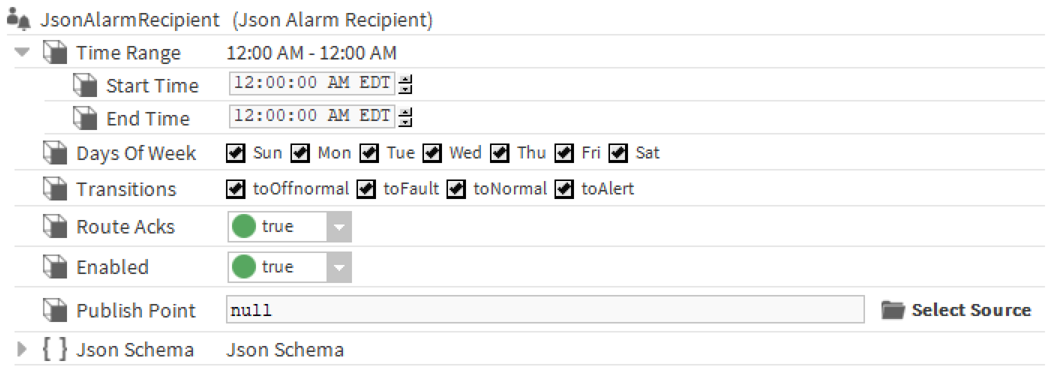
To add this component, expand the **Query** folder in the palette and drag a **`BoundQueryResult`** to the root JSON schema **Object** of a relative JSON schema.

Property	Value	Description
Query	drop-down list	Associates this query result with a query ord as defined by a query under the **Queries** folder. This folder can contain multiple queries.
Output Style	drop-down list	Defines the output style to render the query in.

JsonAlarmRecipient (Json Alarm Recipient)

This component configures the recipient of JSON alarm output.

Figure 64 JsonAlarmRecipient properties



JsonAlarmRecipient (Json Alarm Recipient)

Time Range	12:00 AM - 12:00 AM
Start Time	12:00:00 AM EDT
End Time	12:00:00 AM EDT
Days Of Week	✓ Sun ✓ Mon ✓ Tue ✓ Wed ✓ Thu ✓ Fri ✓ Sat
Transitions	✓ toOffnormal ✓ toFault ✓ toNormal ✓ toAlert
Route Acks	● true
Enabled	● true
Publish Point	null
Json Schema	Json Schema

To use this component, expand the **Alarm** node in the palette and drag a JsonAlarmRecipient to the **Config→Services→AlarmService** folder in the Nav tree.

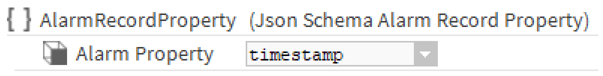
In addition to the standard properties (Days of the Week, Transitions, Publish Point and Enabled), these properties configure this alarm recipient.

Property	Value	Description
Time Range, Start Time (JsonAlarm-Recipient)	hour of the day	When used with **`End Time`**, sets when to begin reporting alarm records.
Time Range, End Time (JsonAlarm-Recipient)	hour of the day	When configured with **`Start Time`**, filters alarm records by defining when to stop returning records.
Publish Point	text (defaults to null)	Selects the source.

AlarmRecordProperty (Json Schema Alarm Record Property)

These properties are only supported on the JsonAlarmRecipients Schema.

Figure 65 Alarm record property



AlarmRecordProperty (Json Schema Alarm Record Property)

Alarm Property	timestamp

To use this property, expand the **Alarm** node in the palette and drag a **`AlarmRecordProperty`** to a schema's object folder.

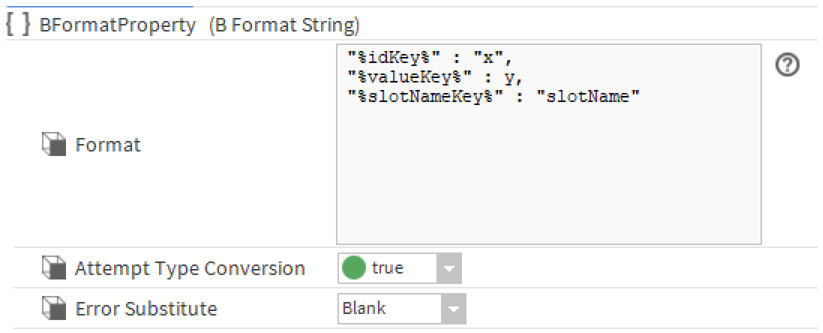
Each of these added to the schema includes the selected **`Alarm Property`** in the output. For example the sourceState, uuid, alarmClass etc. As with other schema Properties, the name is determined by renaming the property, for example AlarmRecordProperty -> timestamp.

Property	Value	Description
Alarm Property	drop-down list	Selects alarm properties to add to the JsonSchema.

BFormatProperty (B Format String)

This property defines alarm data to extract from the Niagara alarm database. For example, if an engineer uses the **`Metadata`** property of an AlarmExt to record the location of a point in a building, **`alarmData.location`** could fetch this information and include it in the payload.

Figure 66 BFormatProperty



Format
```
"%idKey%" : "x",
"%valueKey%" : y,
"%slotNameKey%" : "slotName"
```
Attempt Type Conversion: true
Error Substitute: Blank

To use this property, expand the **Alarm** node in the palette and drag a **`BFormatProperty`** to a schema's object folder.

Property	Value	Description

Format	B Format String	Defines the BFormat string. For example:
		`"%idKey%" : "x",`
		`"%valueKey%" : y,`
		`"%slotNameKey%" : "slotName"`

Attempt Type Conversion	`true` (default) or `false`	Converts Booleans and numbers in a formatted string to native JSON Booleans and numbers. `true` performs the conversion. `false` leaves Booleans and numbers as they are.
Error Substitute	drop-down list, defaults to `Blank`	Controls the role of an error substitute. `Ignore` does nothing. `Key Only` substitutes using the location ID. `Blank` substitutes nothing.

ExportMarker (Json Export Marker)

Provides a way to mark a component for data export to JSON. You use this method rather than binding to an ord, bql, neql, or an absolute path.

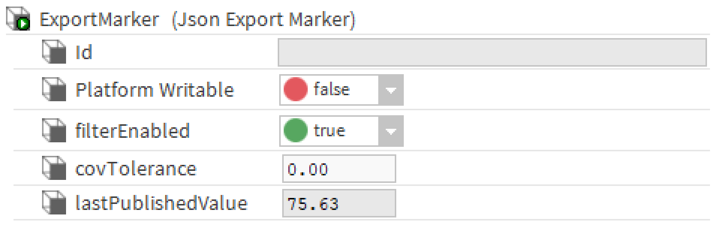
The toolkit provides three ways to select control point data for export:

- Add an absolute ord binding to a JSON schema.
- Use bql or neql to identify control points to a relative JSON schema.
- Add a **`JsonExportMarker`** to a component.

Marking a component offers several benefits beyond just marking points to include in a **`RelativeJson-Schema`**. For example, markers support the export of alarm and history data for specific points. Markers can store a unique identifier supplied by a third party platform. This can be used to differentiate between regis-teredpoints with an ID and unregistered points without an ID. For example, with markers JSON can send dif-ferent payloads prior to registration including more detailed information (units, min/max, descriptive tags) than should be sent upon every change of value.

When applied to a numeric point, a JSON export marker introduces a **`CovTolerance`** property to reduce unwanted updates from the station if a value changes only slightly. You may also use the export marker with incoming JSON payloads.

Figure 67 ExportMarker properties


ExportMarker (Json Export Marker)

Id	
Platform Writable	false
filterEnabled	true
covTolerance	0.00
lastPublishedValue	75.63

To use this marker, expand the **ExportMarker** node in the palette and drag an **ExportMarker** to a point in the station.

Property	Value	Description
Id (ExportMarker)		Provides an id from the cloud platform. The expectation is that this value will be unique, at least within each station as it may be used by the cloud platform as a primary key.
Platform Writable (ExportMarker)	`true` or `false` (default)	Used with the setpoint/override feature to prevent writes from the upstream platform.
filterEnabled	`true` (default) or `false`	Turns the filter on and off. When disabled, the schema ignores **`CovTolerance`**.
CovTolerance	number to two decimal places	Sets up an amount that defines a range of values within which a given value may vary without requiring the station to update the value. This eliminates the overhead required to update when a value changes only slightly.
lastPublishedValue	read-only	Reports the most recent value that was exported.

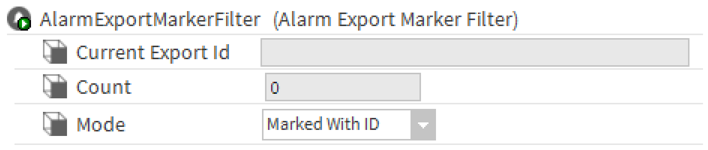
Examples

Example	JSON
Base query	`station:|slot:/|bql:select * from jsonToolkit:JsonExportMarker`
BoundProperty binding ord	`slot:..` (References the parent of the JsonMarker Base)

AlarmExportMarkerFilter (Alarm Export Marker Filter)

This filter selects specific alarms before the station passes the data to an alarm recipient. Typically, the recipient for the filtered alarms would be a **`JsonAlarmRecipient`**, but it could be an SNMP, BACnet, etc. recipient with the source alarm class linked to the In slot of the filter.

Figure 68 AlarmExportMarkerFilter properties


AlarmExportMarkerFilter (Alarm Export Marker Filter)

Current Export Id	
Count	0
Mode	Marked With ID

To use this filter, expand the **ExportMarker** node in the palette and drag an **`AlarmExportMarkerFilter`** to a point in the station.

Property	Value	Description
Current Export Id	read-only	Provides an ID for the export action. For HistoryExportMarkerFilters, this ID should be linked into the schema output to provide identifying information. Or you could even use a query to select data to include if the target system could infer useful data from it.
Count	read-only	Reports how many export marked alarms where processed in the last invocation. It resets when the station restarts.
Mode	drop-down list (defaults to `Marked With ID`)	Selects which alarm records to output to the alarm recipient. `Marked With Id` outputs records that have an ExportMarker on the source component with an Id set. `Marked` outputs records that have an ExportMarker on the source component. `Pass All` outputs all alarms. `Block All` outputs no alarms.

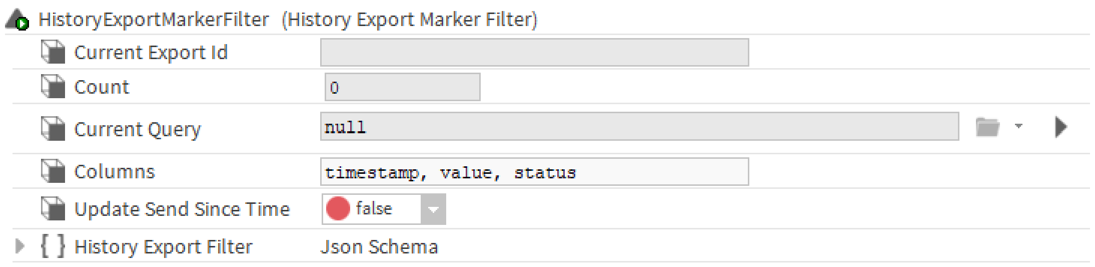
Action

Action	Description
Send Since	Queries the alarm database and passes existing records in to this filter (inclusive of the supplied timestamp) so that the framework can check them for a suitable export marker and then pass them on to the receiving JSON schema as required to create a new record for each alarm. The timestamp, being in the past, should help identify when this mode is active.

HistoryExportMarkerFilter (History Export Marker Filter)

This filter exports history data for points with an export marker. To do so, it adds a new query under the schema's **`Queries`** folder (if one does not already exist). A **`BoundQueryResult`** references this query.

Figure 69 HistoryExportMarkerFilter properties


HistoryExportMarkerFilter (History Export Marker Filter)

Current Export Id	
Count	0
Current Query	null
Columns	timestamp, value, status
Update Send Since Time	false
{ } History Export Filter	Json Schema

To use this filter, expand the **ExportMarker** node in the palette and drag an **HistoryExportMarkerFil-ter** to a location in the station.

You access these properties by double-clicking the **HistoryExportMarkerFilter** node in the Nav tree.

There is some overlap with the **RelativeHistoryComponent**, which can select point histories using many different criteria, and an appropriate BaseQuery may also be used to generate history for each export marked point. The **HistoryExportMarkerFilter** updates the timestamp stored on each ExportMarker, so that the schema sends only recent history records to the remote system (typically records added since the last export).

The History Export Filter container is a JsonSchema nested under the filter. It determines the payload for-mat, and the output from that schema to link to a target transport point to complete the export.

If a point with an **ExportMarker** has more than one history extension, the schema exports each in turn.

NOTE: Since the **ExportMarker** relies on being added to a local control point in the station, it is not possi-ble to match histories imported over BACnet or the **NiagaraNetwork** using this filter. Instead, use a **RelativeJsonSchema**.

In addition to the standard properties (Enabled, Status, and Fault Cause), the history export filter provides these properties.

Property	Value	Description
Current Export Id	read-only	Provides an ID for the export action. For HistoryExportMarkerFilters, this ID should be linked into the schema output to provide identifying information. Or you could even use a query to select data to include if the target system could infer useful data from it.
Count	read-only	Reports how many export marked histories where processed in the last invocation. It resets when the station restarts.
CurrentQuery		Identifies the query used by the **HistoryExportMarkerFil-ter**'s schema. The first query in the **Queries** folder is linked on start, but it does not have to be the only query, or output first data in the JSON schema.
Columns	CSV text	Defines the columns to appear in the filter. For example, `timestamp, value, status`.
Update Send Since Time	`true` or `false` (default)	Enables and disables the updating of the timestamp stored on the **ExportMarker** every time the schema exports history. If `true`, which means the most recent send time was updated, the schema sends only the changed records. If `false`, the schema sends all history records that meet the other criteria.

Actions

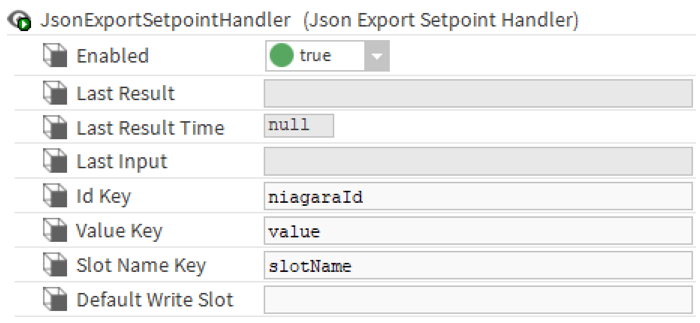
`Send Since Last Export` uses the timestamp stored in each ExportMarker to send only history records that have not yet been sent.

JsonExportSetpointHandler (Json Export Setpoint Handler)

This component allows an external JSON message to change the value of a control point identified by the ID property of an export marker.

Locating target points like this can support a station where a unique key registers the points from the cloud platform. Once the cloud platform returns a suitable identifier for an export-marked point, you can use this setpoint handler to apply write messages from the platform using the ID, rather than the Niagara slot or handle ord (for example).

Figure 70 JsonExportSetpointHandler properties



JsonExportSetpointHandler (Json Export Setpoint Handler)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Id Key	niagaraId
Value Key	value
Slot Name Key	slotName
Default Write Slot	

To add this handler to a station, expand the **ExportMarker** folder in the palette and drag this component to the router folder in the Nav tree.

In addition to the standard property (Enabled), these properties support the JsonExportSetpointHandler.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

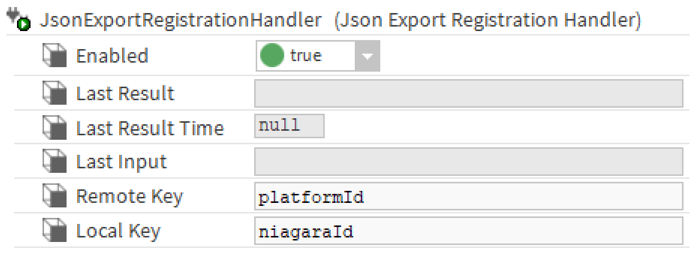
Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Id Key	text	Defines which top-level key in the JSON payload represents the point Id.
Value Key	text (defaults to value)	Defines which top-level key in the JSON payload represents the value to set.
Slot Name Key	text (defaults to slotName)	Defines the optional top-level key in the JSON payload that represents the slot name to write to.
Default Write Slot (JsonExportSet-pointHandler, SetPointHandler)	text	Defines the slot to write to by default if the payload does not specify the slot.

JsonExportRegistrationHandler (Json Export Registration Handler)

This component works with the **`JsonExportSetpointHandler`** to apply a unique identifier from an external system to an export marker.

This allows the cloud (or other external system) target to assign it's own identifier or primary key to export-marked points in the Niagara station, which can be used to locate them in future, or included in exports to that cloud system.

Figure 71 JsonExportRegistrationHandler properties



JsonExportRegistrationHandler (Json Export Registration Handler)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Remote Key	platformId
Local Key	niagaraId

To add this handler to a station, expand the **ExportMarker** folder in the palette and drag this component to the router folder in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonExportRegistrationHandler`**.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Remote Key	text (defaults to platformId)	Identifies the name of the JSON property that denotes the point's identifier in the remote system.
Local Key	text (defaults to niagaraId)	Identifies the name of the JSON property that denotes the point’s identifier in the Niagara station.

Syntax

The messages should be in this format:

```
{
  "messageType" : "registerId"
  "niagaraId" : "h:a032",
  "platformId" : "mooseForce123"
}
```

or

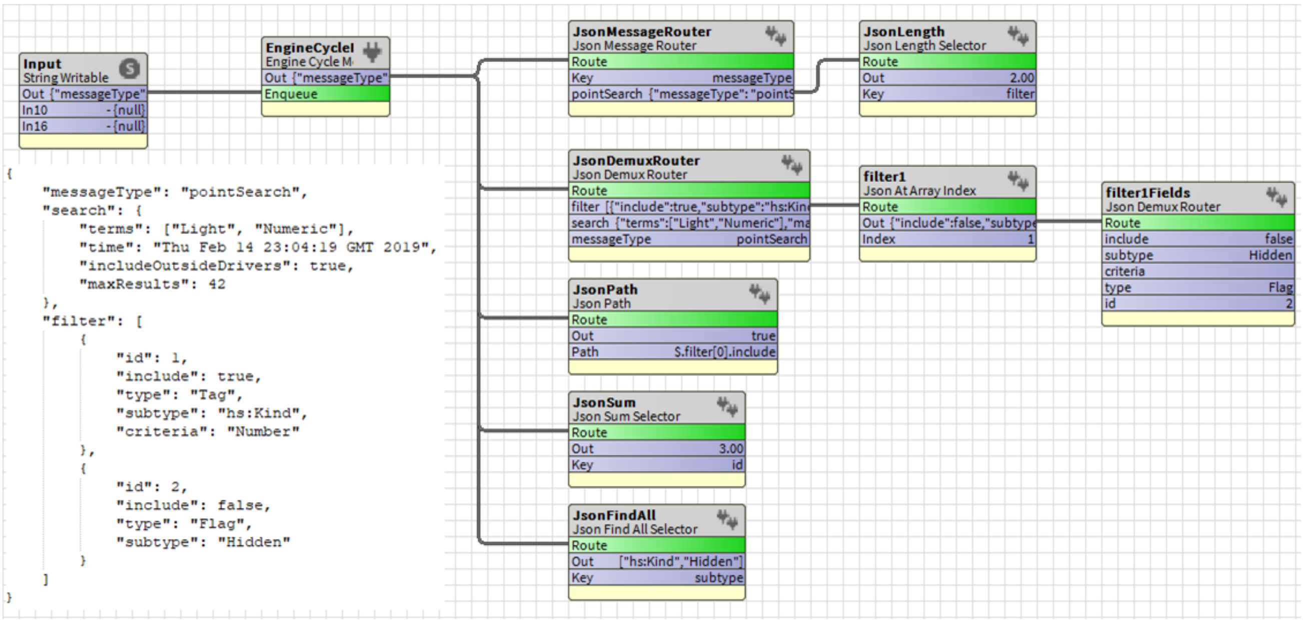
```
{
  "messageType" : "deregisterId"
  "platformId" : "mooseForce123",
}
```

NOTE: This class does not use the **`messageType`**, which would be used simply to route it to this handler and so can be changed as needed.

Example

This **Wire Sheet** and JSON loosely demonstrate some of the routers and selectors based upon a fictional point search JSON message.

Figure 72 Json Export Registration Handler example Wire Sheet and JSON



Input (String Writable)
Out {"messageType"
In10 - {null}
In16 - {null}

EngineCycle1 (Engine Cycle M)
Out {"messageType"
Enqueue

JsonMessageRouter (Json Message Router)
Route
Key messageType
pointSearch {"messageType":"pointS

JsonLength (Json Length Selector)
Route
Out 2.00
Key filter

JsonDemuxRouter (Json Demux Router)
Route
filter [{"include":true,"subtype":"hs:Kin
search {"terms":["Light","Numeric"],"ma
messageType pointSearch

filter1 (Json At Array Index)
Route
Out {"include":false,"subtype
Index 1

filter1Fields (Json Demux Router)
Route
include false
subtype Hidden
criteria
type Flag
id 2

JsonPath (Json Path)
Route
Out true
Path $.filter[0].include

JsonSum (Json Sum Selector)
Route
Out 3.00
Key id

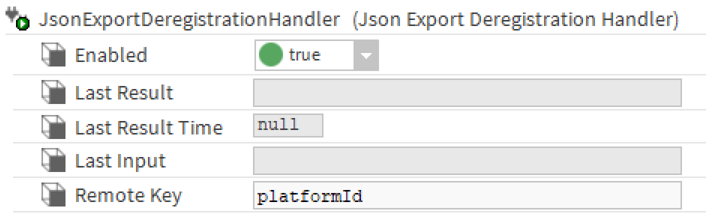
JsonFindAll (Json Find All Selector)
Route
Out ["hs:Kind","Hidden"]
Key subtype

```
{
    "messageType": "pointSearch",
    "search": {
        "terms": ["Light", "Numeric"],
        "time": "Thu Feb 14 23:04:19 GMT 2019",
        "includeOutsideDrivers": true,
        "maxResults": 42
    },
    "filter": [
        {
            "id": 1,
            "include": true,
            "type": "Tag",
            "subtype": "hs:Kind",
            "criteria": "Number"
        },
        {
            "id": 2,
            "include": false,
            "type": "Flag",
            "subtype": "Hidden"
        }
    ]
}
```

JsonExportDeregistrationHandler (Json Export Deregistration Handler)

This component works with the **`JsonExportSetpointHandler`** to remove a unique identifier from an external system to an export marker.

Figure 73 JsonExportDeregistrationHandler properties



JsonExportDeregistrationHandler (Json Export Deregistration Handler)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Remote Key	platformId

To add this handler to a station, expand the **ExportMarker** folder in the palette and drag this component to the router folder in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonExportDeregistrationHandler`**.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Remote Key	text (defaults to platformId)	Identifies the name of the JSON property that denotes the point's identifier in the remote system.

Syntax

The messages should be in this format:

```
{
  "messageType" : "registerId"
  "niagaraId" : "h:a032",
  "platformId" : "mooseForce123"
}
```

or

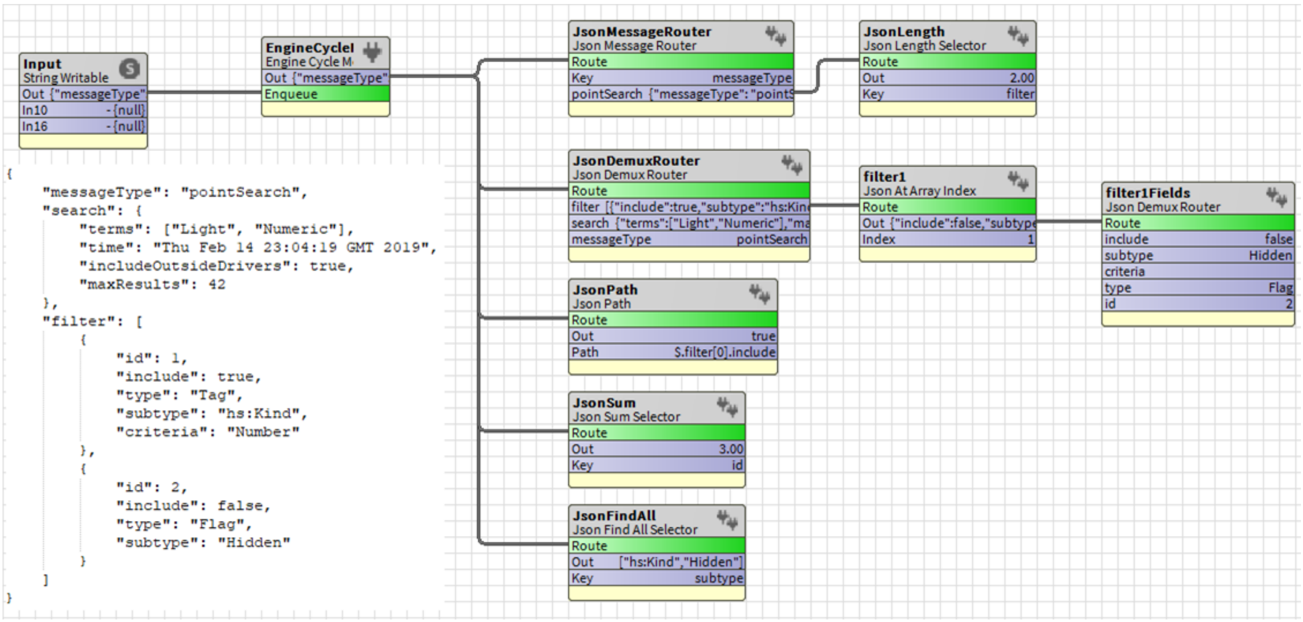
```
{
  "messageType" : "deregisterId"
  "platformId" : "mooseForce123",
}
```

NOTE: This class does not use the `messageType`, which would be used simply to route it to this handler and so can be changed as needed.

Example

This Wire Sheet and JSON loosely demonstrate some of the routers and selectors based upon a fictional point search JSON message.

Figure 74 Json Export Registration Handler example Wire Sheet and JSON



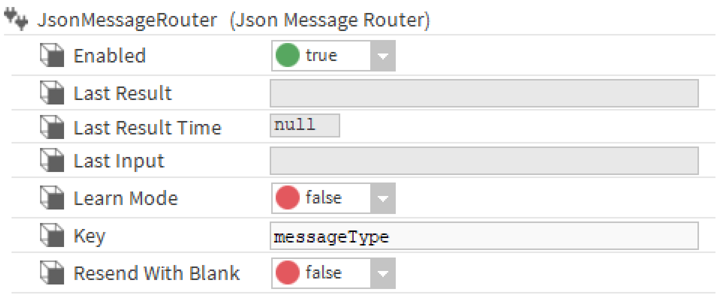
```
{
    "messageType": "pointSearch",
    "search": {
        "terms": ["Light", "Numeric"],
        "time": "Thu Feb 14 23:04:19 GMT 2019",
        "includeOutsideDrivers": true,
        "maxResults": 42
    },
    "filter": [
        {
            "id": 1,
            "include": true,
            "type": "Tag",
            "subtype": "hs:Kind",
            "criteria": "Number"
        },
        {
            "id": 2,
            "include": false,
            "type": "Flag",
            "subtype": "Hidden"
        }
    ]
}
```

JsonMessageRouter (Json Message Router)

This component transfers inbound messages to an onward component that is suitable for processing or handling the message.

This allows the cloud (or other external system) target to assign it's own identifier or primary key to export-marked points in the Niagara station, which can be used to locate them in future, or included in exports to that cloud system.

Figure 75 JsonMessageRouter properties



JsonMessageRouter (Json Message Router)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Learn Mode	● false
Key	messageType
Resend With Blank	● false

You add this router to a station by expanding the **Inbound→Routers** in the palette and dragging this component to the **Config** folder in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonMessageRouter`**.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.

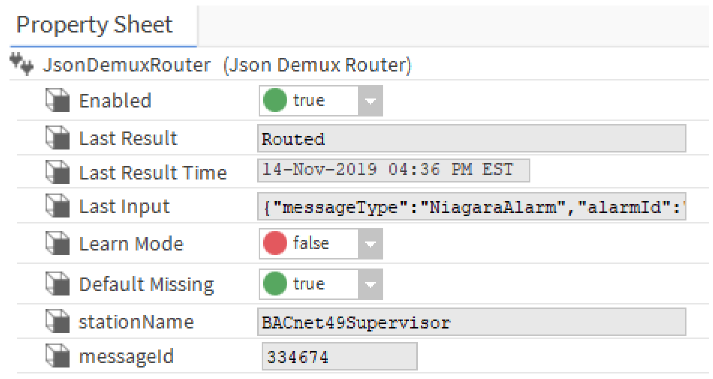
Property	Value	Description
Last Input	read-only	Reports the last message routed to a component.
Learn Mode	`true` or `false` (default)	`true` configures the JSON to add a dynamic slot on input for any newly-found message key.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)
Resend With Blank	`true` or `false` (default)	Turns on and off the resending of a message if a duplicate or matching message is received. `true` causes the router to send an empty string to the target slot, then resend the output. Without injecting an empty message, the link does not propagate the change, which could be an issue if the handler needed other values in place to respond to this message. `false` does not send the empty string to the target slot, which does not resend the output.

JsonDemuxRouter (Json Dmux Router)

Unlike the **`JsonMessageRouter`**, which forwards the whole JSON payload to the added slots intact, this component passes a selected part of the message to the added slots. It is a very basic method of selecting data of interest, and likely will become inefficient to use when faced with a large payload and chained routers. An approach with far more features is JSON Path.

The added slots must match the key name and should be either Boolean, numeric or string to match the JSON value.

Figure 76 JsonDemuxRouter properties


Property Sheet

JsonDemuxRouter (Json Demux Router)

Enabled	● true
Last Result	Routed
Last Result Time	14-Nov-2019 04:36 PM EST
Last Input	{"messageType":"NiagaraAlarm","alarmId":
Learn Mode	● false
Default Missing	● true
stationName	BACnet49Supervisor
messageId	334674

You add this router to a station by expanding the **Inbound→Routers** in the palette and dragging this component to the **Config** folder in the Nav tree.

In addition to the standard property (Enabled), these properties support the JsonDemuxRouter.

Property	Value	Description

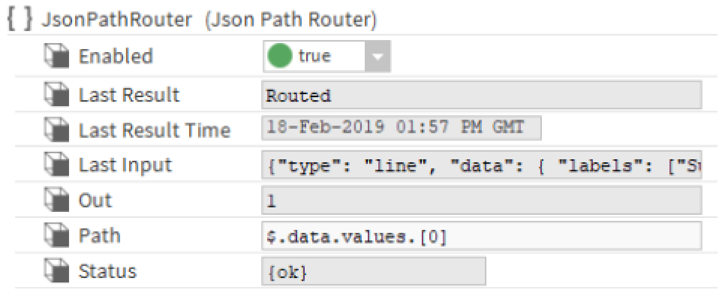
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Learn Mode	`true` or `false` (default)	`true` configures the JSON to add a dynamic slot on input for any newly-found message key.
Default Missing	`true` (default) or `false`	Can set a dynamic slot's value to its default if the value is missing. `true` sets the value to its default if the inbound JSON message did not include a value for the slot. `false` ignores setting the default value.

JsonPath (Json Path)

Selectors are components that take an inbound JSON message, apply some selection criteria to it, and set up the result an out slot. This might be a subset of the JSON. It could be, for example, the size of a message or the result of an aggregate function, such as the sum of a repeated value. This selector component allows data to be interactively located and extracted from JSON structures using a special notation to represent the payload structure.

Figure 77 JsonPath properties


{ } JsonPathRouter (Json Path Router)

Enabled	true
Last Result	Routed
Last Result Time	18-Feb-2019 01:57 PM GMT
Last Input	{"type": "line", "data": { "labels": ["S
Out	1
Path	$.data.values.[0]
Status	{ok}

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`JsonPath`** to a JSON message router node in the Nav tree.

In addition to the standard properties (Enabled and Status), these properties support the **`JsonPath`**.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

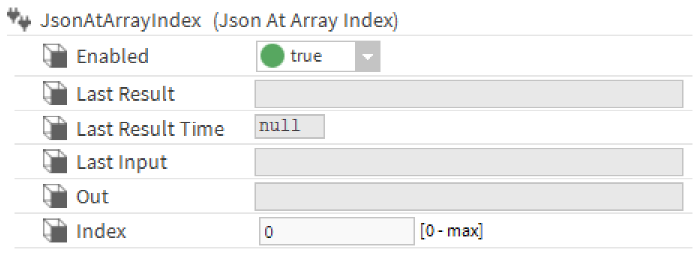
Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.

Property	Value	Description
Out	read-only	Displays the result.
Path (JsonPath)	text	Defines the path.

JsonAtArrayIndex (Json At Array Index)

This component selects a value in a JSON array by array index.

Figure 78 JsonAtArrayIndex properties



JsonAtArrayIndex (Json At Array Index)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Out	
Index	0 [0 - max]

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`JsonAtArrayIndex`** to a message router in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonAtArrayIndex`**.

Property	Value	Description

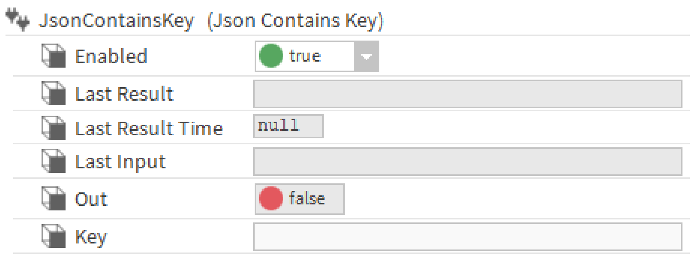
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find message key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Index	number	Defines the index in the JSON array.

JsonContainsKey (Json Contains Key)

This selector returns a Boolean value if the specified key is present in the payload.

Figure 79 JsonContainsKey properties



JsonContainsKey (Json Contains Key)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Out	false
Key	

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`JsonContainsKey`** to a message router in the Nav tree.

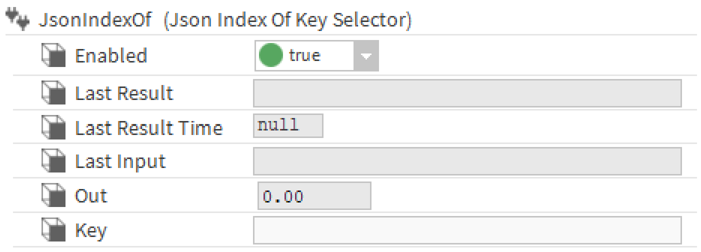
In addition to the standard property (Enabled), these properties support the JsonContainsKey.

Property	Value	Description
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output: `Unable to find messagge key:` `Problem parsing messageType`
Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)

JsonIndexOf (Json Index Of Key Selector)

This component returns the index of a given key within a JSON object.

Figure 80 JsonIdexOf properties



JsonIndexOf (Json Index Of Key Selector)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Out	0.00
Key	

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`Jso-nIndexOf`** to a message router in the Nav tree.

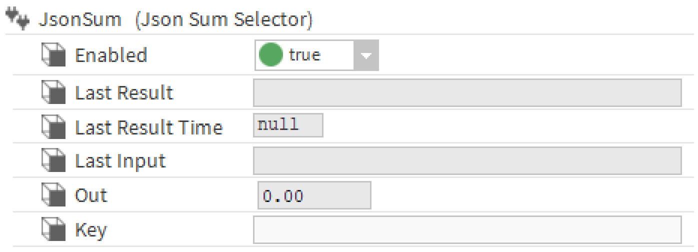
In addition to the standard property (Enabled), these properties support the **`JsonIndexOf`**.

Property	Value	Description
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output: `Unable to find messagge key:` `Problem parsing messageType`
Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)

JsonSum (Json Sum Selector)

This selector sums all values found in the payload that match the key (numeric values parsed only).

Figure 81 JsonSum properties


JsonSum (Json Sum Selector)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Out	0.00
Key	

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`Json-Sum`** to a message router in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonSum`**.

Property	**Value**	**Description**

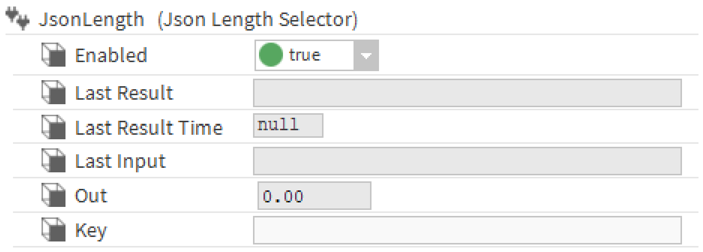
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)

JsonLength (Json Length Selector)

This selector returns the length of the first object or array that matches the key.

Figure 82 JsonLength properties


JsonLength (Json Length Selector)

Enabled	true
Last Result	
Last Result Time	null
Last Input	
Out	0.00
Key	

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`Json-Length`** to a message router in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonLength`**.

Property	Value	Description

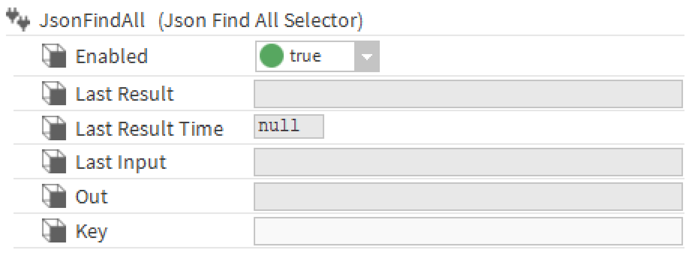
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)

JsonFindAll (Json Find All Selector)

This selector returns all values in an array that match the key.

Figure 83 JsonFindAll properties



JsonFindAll (Json Find All Selector)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Out	
Key	

You add this selector to a station by expanding **Inbound→Selectors** in the palette and dragging the **`Json-FindAll`** to a message router in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`JsonFindAll`**.

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

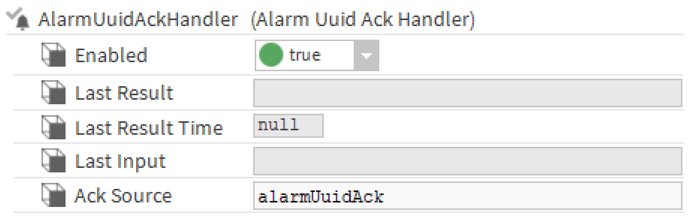
Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Out	read-only	Displays the result.
Key	text (defaults to messageType)	Defines which part of the incoming message to switch on)

AlarmUuidAckHandler (Alarm Uuid Ack Handler)

If the alarms exported from a station include a unique ID (UUID), this component passes back the UUID.

Message handlers are components designed to perform a specific task with the data routed and selected via the other inbound components.

Figure 84 AlarmUuidAckHandler properties



AlarmUuidAckHandler (Alarm Uuid Ack Handler)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Ack Source	alarmUuidAck

You add this handler to a station by expanding the **Inbound→Handlers** folder in the palette and dragging this component to a message router in the Nav tree.

In addition to the standard property (Enabled), these properties support the **`AlarmUuidAckHandler`**:

Property	Value	Description

Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Ack Source	text	Configures a string to append to every alarm record acknowledgement. This string can provide additional information for future auditing. The alarm data record stores it as AckSource.
Ack Result		Reports the results of the alarm acknowledgment for logging or post processing activity.

Example

The expected format for this component is:

```
{
  "user": "Maya",
  "alarms": [ "5cf9c8b2-1542-42ba-a1fd-5f753c777bc0" ]
}
```

This array allows the system to acknowledge multiple alarms at once.

The alarm record stores the user value, which identifies the user who acknowledged the alarm in the remote application. If the user key is omitted, the component still tries to acknowledge the alarms using the fallback name: **`AlarmUuidAckUser`**.

NOTE:

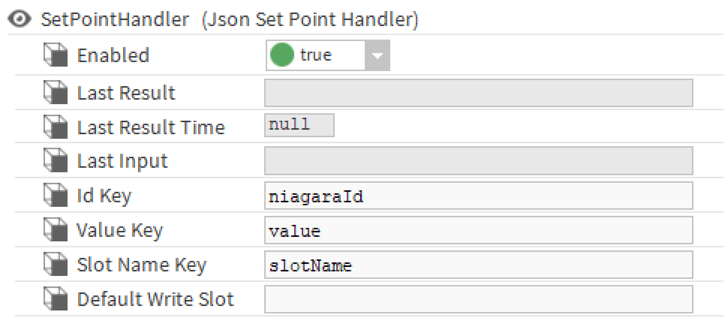
The **JsonSchemaService**’s **`Run As User`** property is a prerequisite for this handler to work. The specified user must have admin write permissions for the alarm class of the records being acknowledged.

SetPointHandler (Json Set Point Handler)

This handler sets incoming setpoint values to control writable control points.

Override, duration, the status parameter and nested keys are not supported.

Figure 85 SetpointHandler properties


SetPointHandler (Json Set Point Handler)

Enabled	● true
Last Result	
Last Result Time	null
Last Input	
Id Key	niagaraId
Value Key	value
Slot Name Key	slotName
Default Write Slot	

You add this handler to a station by expanding the **Inbound→Handlers** folder in the palette and dragging this component to a message router in the Nav tree.

NOTE: The **`Run As User`** property in the **JsonSchemaService** is required to use the **`SetPointHandler`**.

In addition to the standard property (Enabled), these properties support the **`SetPointHandler`**:

Property	Value	Description

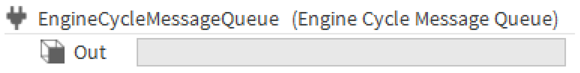
Last Result	read-only	Reports the results of the alarm acknowledgment to allow for logging or post-processing activity. Example output:
		`Unable to find messagge key:`
		`Problem parsing messageType`

Last Result Time	read-only	Reports when the handler ran last.
Last Input	read-only	Reports the last message routed to a component.
Id Key	text	Defines which top-level key in the JSON payload represents the point Id.
Value Key	text (defaults to value)	Defines which top-level key in the JSON payload represents the value to set.
Slot Name Key	text (defaults to slotName)	Defines the optional top-level key in the JSON payload that represents the slot name to write to.
Default Write Slot	text	Defines the slot to write to by default if the payload does not specify the slot.

EngineCycleMessageQueue (Engine Cycle Message Queue)

When the system generates JSON very quickly, this component can provide a buffer between the data source and destination control point to prevent potential discards within the same engine cycle. Using this component ensures that the JSON processes all messages.

Figure 86 EngineCycleMessageQueue property


EngineCycleMessageQueue (Engine Cycle Message Queue)

Out

You add this queue to a station by expanding the **Inbound→Handlers** folder in the palette and dragging this component to a message router in the Nav tree.

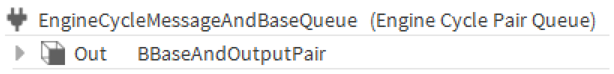
For example, you can link a string output slot to onward points or, where necessary, to an **`EngineCycleMessageQueue`**.

To buffer incoming messages when using this component, it is advisable to link from the readValue on a proxyExt rather than from the out slot of its parent point.

Property	Value	Description
Out	read-only	Displays the result.

EngineCycleMessageAndBaseQueue (Engine Cycle Pair Queue)

Figure 87 EngineCycleMessageAndBaseQueue property



EngineCycleMessageAndBaseQueue (Engine Cycle Pair Queue)
Out BBaseAndOutputPair

You add this queue to a station by expanding the **Inbound→Handlers** folder in the palette and dragging this component to a message router in the Nav tree.

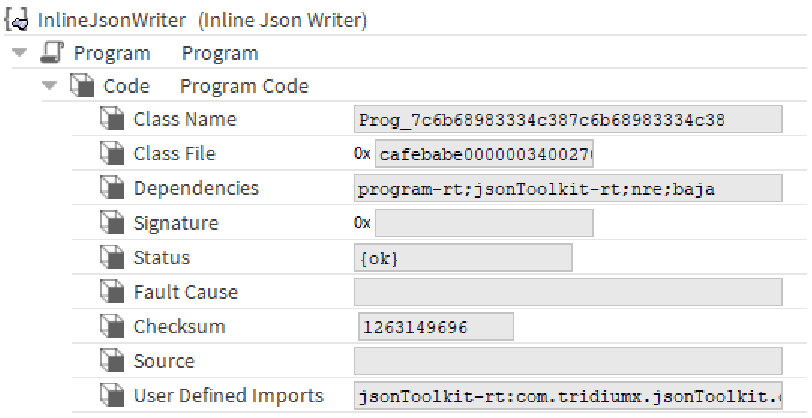
Property	Value	Description
Out		

InlineJsonWriter (Inline Json Writer)

This feature supports custom JSON code.

You achieve this using a program object as per the example in the **Programs** folder of the `jsonToolkit` palette. You can extend `BAbstractInlineJsonWriter`. Extending the abstract class would be preferred where the program object may be widely distributed, as code contained in a module is easier to maintain.

Figure 88 InlineJsonWriter code properties



InlineJsonWriter (Inline Json Writer)
Program Program
Code Program Code

Class Name	Prog_7c6b68983334c387c6b68983334c38
Class File	0x cafebabe000000340027
Dependencies	program-rt;jsonToolkit-rt;nre;baja
Signature	0x
Status	{ok}
Fault Cause	
Checksum	1263149696
Source	
User Defined Imports	jsonToolkit-rt:com.tridiumx.jsonToolkit.

To use this program object, drag it from the **Programs** folder in the `jsonToolkit` palette to the **Config** folder in the station. To open this **AX Property Sheet**, double-click the `InlineJsonWriter` component in the station.

To view the example code, right–click the **Program** node, click**Views→Program Editor** and click the **Edit** tab.

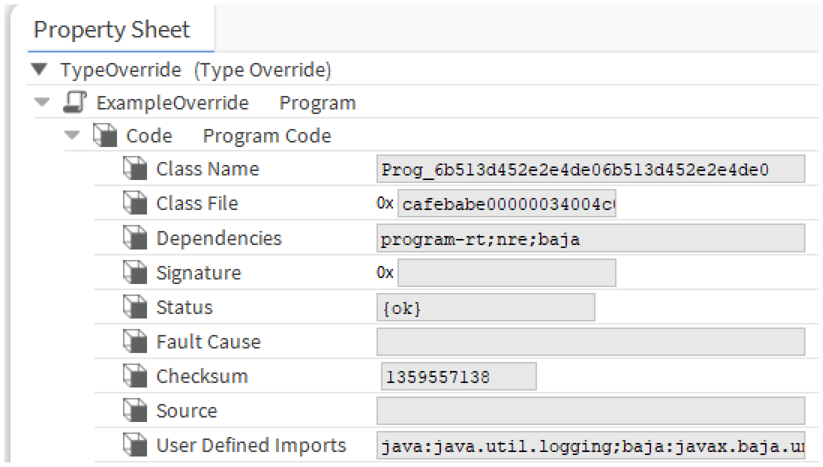
In addition to the standard properties (Status and Fault Cause), these properties support the `InlineJsonWriter`.

Property	Value	Description
Class Name	read-only	Reports the name that describes this object. Each object is created from a single class. One class can instantiate multiple objects.
Class File	read-only	Reports the name of the file that contains the custom program.
Dependencies	read-only	Identifies the dependent modules.
Signature	read-only	Identifies the mathematical scheme used to verify the authenticity of the program.

TypeOverride (Type Override)

This component is an example of a program to override a data type.

Figure 89 TypeOverride properties



Property Sheet

TypeOverride (Type Override)

ExampleOverride Program

Code Program Code

Class Name	Prog_6b513d452e2e4de06b513d452e2e4de0
Class File	0x cafebabe00000034004c
Dependencies	program-rt;nre;baja
Signature	0x
Status	{ok}
Fault Cause	
Checksum	1359557138
Source	
User Defined Imports	java:java.util.logging;baja:javax.baja.ui

To use this program object, drag it from the **Programs** folder in the **`jsonToolkit`** palette to the **Config** folder in the station. To open its **AX Property Sheet**, double-click the **`TypeOverride`** object in the station.

To view the example code, right–click the **ExampleOverride** node, click**Views→Program Editor** and, if needed, click the **Edit** tab.

In addition to the standard properties (Status and Fault Cause), these properties support the **`TypeOverr-ide`** example. These properties are part of the **`Program`** component from the Program module and are not specific to the JSON Toolkit.

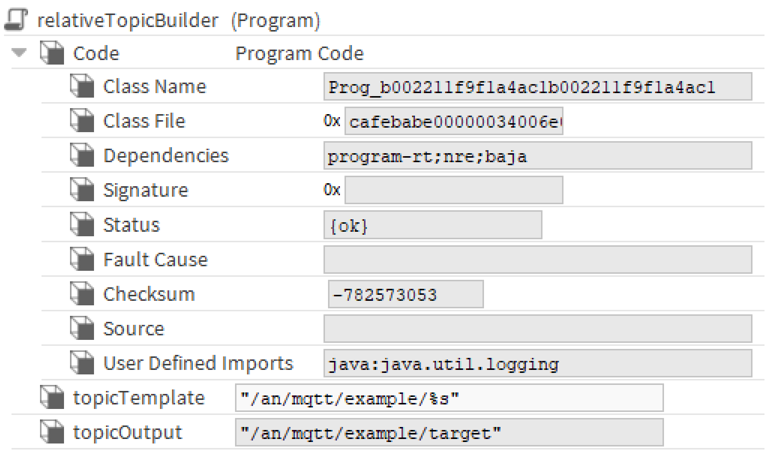
Property	Value	Description
Class Name	read-only	Reports the name that describes this object. Each object is created from a single class. One class can instantiate multiple objects.
Class File	read-only	Reports the name of the file that contains the custom program.
Dependencies	read-only	Identifies the dependent modules.
Signature	read-only	Identifies the mathematical scheme used to verify the authenticity of the program.

Property	Value	Description
Source	read-only	Displays the program's source code.
User Defined Imports	read-only	Displays user-defined custom imports of the types used in the source code.

relativeTopicBuilder (Program)

This program object uses an instance-based class file to implement your component logic. You view and edit the program using the **`ProgramEditor`**.

Figure 90 relativeTopicBuilder properties



relativeTopicBuilder (Program)

Code	Program Code
Class Name	Prog_b002211f9f1a4ac1b002211f9f1a4ac1
Class File	0x cafebabe00000034006e
Dependencies	program-rt;nre;baja
Signature	0x
Status	{ok}
Fault Cause	
Checksum	-782573053
Source	
User Defined Imports	java:java.util.logging
topicTemplate	"/an/mqtt/example/%s"
topicOutput	"/an/mqtt/example/target"

To use this object, drag it from the **Programs** folder in the **`jsonToolkit`** palette to the **Config** folder in the station. To open this **AX Property Sheet**, double-click the **`relativeTopicBuilder`** component in the station.

In addition to the standard properties (Status and Fault Cause), these properties support the **`relativeTopicBuilder`**.

Property	Value	Description
Class Name	read-only	Reports the name that describes this object. Each object is created from a single class. One class can instantiate multiple objects.
Class File	read-only	Reports the name of the file that contains the custom program.
Dependencies	read-only	Identifies the dependent modules.
Signature	read-only	Identifies the mathematical scheme used to verify the authenticity of the program.
Source	read-only	Displays the program's source code.
User Defined Imports	read-only	Displays user-defined custom imports of the types used in the source code.

Property	Value	Description
topicTemplate	text	Defines a template string for the output topic in a Java-format style. For example, `%s` represents a replaceable substring. The schema resolves these against the object it passes to the input slot and writes the result to the **`topicOutput`**.
topicOutput	read-only	Identifies the destination for the output JSON.

Actions

`Execute` takes some input, does whatever the action should (including

Index

Glossary

binding	A relationship between a widget in a station and a data source, such as a point, slot, component, tag, etc. The most common binding, a value binding, provides information for presentation as text or a graphic. Bindings include mouse-over and right-click actions, and a way to animate any property of its parent widget using converters that convert the target object into a property value.
payload	The objects, arrays and key/value pairs contained between open and close curly brackets that conform to JSON syntax.
subscription	A method for updating a station with the current value of a remote component. When a remote component's value changes, subscription synchronizes the related proxy point's value in the station with the current value of the remote component. Subscription occurs in real time.