Technical Document

Hierarchies Guide

August 18, 2015

niagara4

Hierarchies Guide

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

Confidentiality

Trademark notice

Copyright and patent notice

Contents

About this guide

This document provides tasks and conceptual information about the hierarchies features in Niagara 4.

Document change log

Updates (changes and additions) to this document are listed below.

- Initial release publication: August 18, 2015

Related documentation

Additional information is available in the following documents.

- *Tagging Guide*
- *Relations Guide*
- *Station Security Guide*

Chapter 1 Common hierarchy tasks

Topics covered in this chapter

- ♦ Preliminary preparations
- ♦ Setting up a hierarchy definition
- ♦ Editing a hierarchy definition
- ♦ Assigning a hierarchy to a role

The Hierarchy Service provides an efficient method of creating one or more logical navigation trees for the Niagara system. You manage (set up and edit) hierarchy definitions on a station under the **HierarchyService**. Hierarchy definitions are not legal anywhere else in the station. To set up new hierarchy definition, drag and drop the **Hierarchy** component onto the **HierarchyService** node or property sheet. Currently, the **Hierarchy Service** can contain an unlimited number of hierarchy definitions.

When you save a hierarchy definition, it executes against the system and the resulting Nav tree hierarchy is saved in the station's **Hierarchy** space. The navigation hierarchy name matches that of the hierarchy definition. In order to modify a navigation hierarchy, you must make necessary changes in the hierarchy definition and save. Then simply right-click the **Hierarchy** node and refresh the nav tree to update the navigation hierarchy.

Preliminary preparations

The following preparations should be done prior to setting up a hierarchy.

1. Lay out how you want your hierarchy to look on a piece of paper before you begin assigning tags and creating components for buildings, offices, etc.

   NOTE: It may be helpful to model the desired navigation nodes in the station using folders to represent objects such as offices or floors.

2. Make a list of tags you plan to use and to which components you will assign the tags. You are not limited to using tags from one tag dictionary. You can use any tags that are available, as well as individual Ad Hoc tags which you create as needed (they do not need to be in a tag dictionary). For any tags that you create, use a consistent tag naming convention, such as acme:AHU, acme:building, etc.

   NOTE: Only the tags that you add to an object appear on its property sheet. To view both implied tags and directly added tags for a component, right-click the component in the Nav tree and select **Edit Tags**, then click the tabs to view either Direct or Implied tags.

3. Confirm that the necessary tag dictionaries are installed. If desired, create a custom tag dictionary containing a collection of tags that you create in order to simplify the tagging process.

   NOTE: The Niagara tag dictionary is installed by default. Also, the Haystack open source tag dictionary, included in the installation, can be added from the **haystack** palette.

4. Confirm devices and points are already discovered and tagged. If not, add the necessary tags. You can assign multiple tags to any component. For example, one piece of equipment might have the following tags applied: n:device, acme:AHU2, acme:equipRef, etc. For details on adding tags, see the *Tagging Guide*.

   NOTE: Tags are case sensitive. Make sure you use the correct case when entering tags in your hierarchy definition queries otherwise the queries will return nothing.

5. Confirm any additional components (any model components representing buildings, offices, etc.) are already created and tagged. If not, add necessary tags.

6. Confirm any necessary relationships between components are already added. If not, add any necessary relations. For example, you may need to add a relation between a folder in your model representing a particular floor and several air handler units. For details on adding relations, see the *Relations Guide*.

NOTE: When editing a hierarchy definition (in the **HierarchyService**) the changes are not automatically reflected in the resulting Nav tree hierarchy (in the **Hierarchy** space). To view changes in the Nav tree hierarchy, right-click the **Hierarchy** node and select **Refresh Tree Node**. This updates the hierarchy according to the current definition.

Once you have tagged all components as needed and added any desired relations between components you are ready to define your hierarchy.

Additional Tips

- You can speed up hierarchy creation by copying and pasting level definitions within the current hierarchy, or from one hierarchy to another. Then make any necessary edits to the pasted level definitions.
- You can create multiple hierarchies for the same station in order to have different users navigate the station differently. For example a Facilities Manager might navigate the system differently than an Operator.
- When defining a hierarchy, a common practice is to frequently save and evaluate the resulting hierarchy. In this way, you are able to identify where an additional level definition is required. Tweak the resulting hierarchy by making iterative passes in this manner.

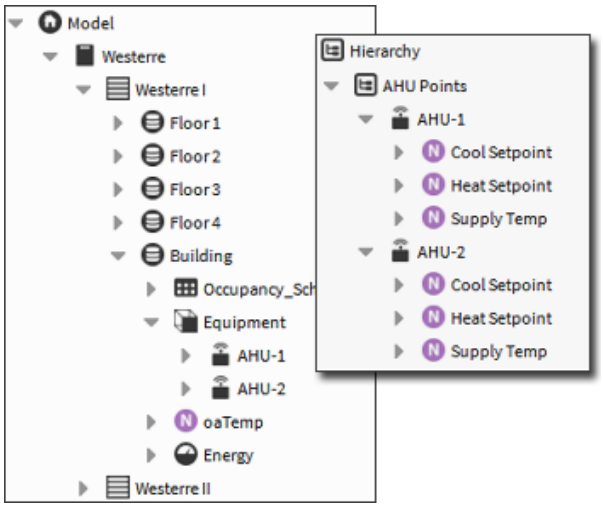
Setting up a hierarchy definition

This procedure demonstrates the steps to define a hierarchy to easily navigate to Air Handler Units in a specific building in order to monitor child points and performance.

Prerequisites:

- The **hierarchy** and **tagdictionary** modules must be installed on your system.
- The **hierarchy** palette is open in the side bar.
- All devices, points and other components are already tagged and any necessary relations already added.
- A plan for the desired navigation hierarchy already determined during preliminary preparations. For more details see the section, *Preliminary preparations*.

Shown here is the modelled site structure (left) and the resulting navigation hierarchy (right).



Step 1 Navigate to the station's **Config→Services** node in the Nav tree and double-click the **HierarchyService**.

The **Hierarchy Service** Property Sheet displays in the right pane.

Step 2 Drag and drop the **Hierarchy** component from the palette side bar to the **Hierarchy Service** Property Sheet (or to **HierarchyService** node in the Nav tree), and in the **Name** dialog enter the hierarchy name: "`AHU Points`" and click **OK**.

The new hierarchy definition appears in the **Hierarchy Service** Property Sheet and under the **HierarchyService** in the Nav tree.

Step 3 In the right pane, click on **AHU Points** to open it's property sheet.

Step 4 Click to expand the **Scope** and **Station** properties and enter the following **Scope ORD**: `station:|slot:/Model/Westerre/W1`

This `Scope ORD` causes the hierarchy query to search only this specific location within the station. An alternative to limiting the scope is to use additional LevelDefs.

NOTE: Red icons appear in a hierarchy definition as you make changes. This alerts you that there are changes that have not yet been saved.

Step 5 Click and drag a **QueryLevelDef** component from the **hierarchy** palette to the **AHU Points** Property Sheet and in the **Name** dialog enter "`Device`" and click **OK**.

Step 6 In the property sheet, click to expand the new **Device** component and perform the following action:

Property	Action
Query	Enter: `n:device and nBld:ahu` to return any objects tagged with `n:device` and `nBld:ahu`.

NOTE: Typically, you identify the tags applied to AHUs in the station and make note of them, by examining tags on one or more of these devices during preliminary preparations. Without prior knowledge, you need to examine the devices at this point to determine how they are tagged or to apply tags.

Step 7 Click and drag a **RelationLevelDef** component from the **hierarchy** palette to the **AHU Points** Property Sheet and in the **Name** dialog enter "`Points`" and click **OK**.

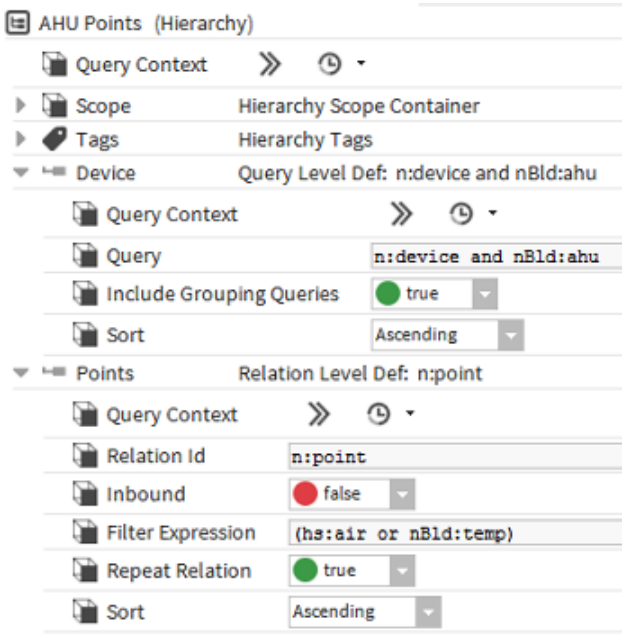
Step 8 In the property sheet, click to expand the new **Points** component and perform the following action:

Property	Action
Relation Id	Enter: `n:point` to return any point components.
Inbound	Select: **false** **NOTE:** Examining the **Relation Sheet** view of one or more points under one of the devices reveals that the relation direction is outbound so this property must be set to **false**.
Filter Expression	Enter additional tags to filter results: `hs:air or nBld:temp` to return any objects tagged with either `hs:air` or `nBld:temp`.
Repeat Relation	Select: **true**

Step 9 Click **Save**.

Step 10 In the Nav tree, right-click the **Hierarchy** space, select `Refresh Tree Node`, and expand the nodes to view the resulting **AHU Points** hierarchy.

Example: AHU Points hierarchy definition



Editing a hierarchy definition

Editing a Nav tree hierarchy is done by making modification in the corresponding hierarchy definition located under the **HierarchyService** node. You can add, remove, or reorder the LevelDefs in an existing hierarchy definition, as well as edit the configured properties (NEQL text, **GroupBy** tags and facets) for any LevelDef.

Prerequisites:

- An existing Nav tree hierarchy in the **Hierarchy** space

Step 1 Expand the **HierarchyService** and double-click the hierarchy definition to edit.

Step 2 Make any of the following changes as needed:

- Click existing LevelDefs to edit configured properties.
- Add additional LevelDefs.
- Delete existing LevelDefs.
- Reorder existing LevelDefs

Step 3 Click **Save**.

Step 4 To view your changes in the hierarchy, right-click the **Hierarchy** node in the Nav tree and click **Refresh Tree Node**.

Assigning a hierarchy to a role

The visibility of any particular hierarchy is given on a role by role basis. More than one hierarchy can be assigned to a role. The role(s) assigned to a user determines which hierarchies are visible to that user.

Prerequisites:

- The hierarchy you wish to assign exists in the Hierarchy space.

Step 1 To open the **RoleManager** view, double-click the **RoleService**.

Step 2 Double-click to edit an existing role.

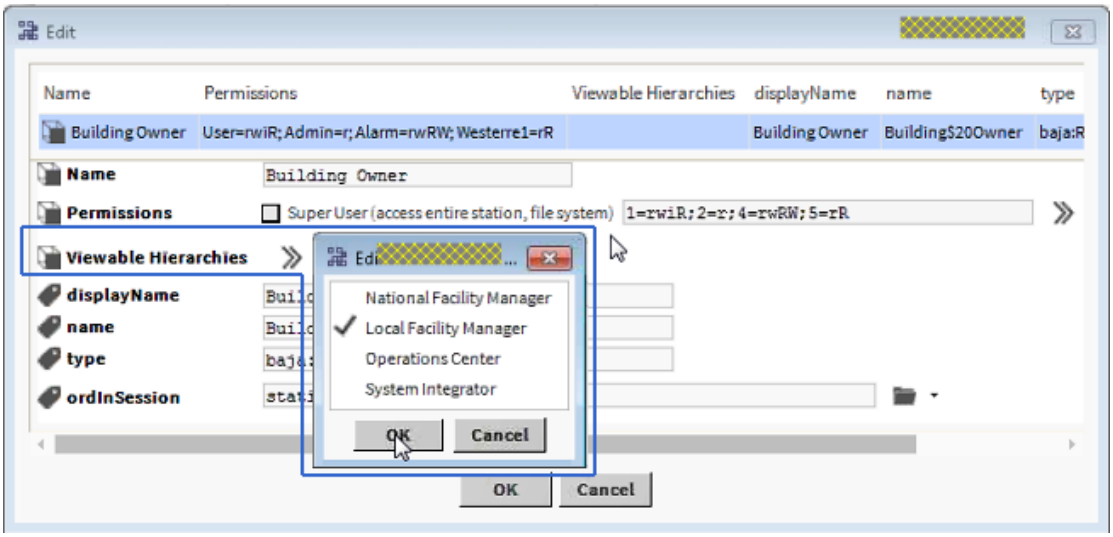
Step 3 In the **Edit** window, click the chevron icon to the right of **`Viewable Hierarchies`**.

Step 4 In the **Edit Viewable Hierarchies** window, click to the left of the desired hierarchy to select it, and click **OK**.

The hierarchy will be visible under the Hierarchy space to any user assigned that role who logs in to the station.

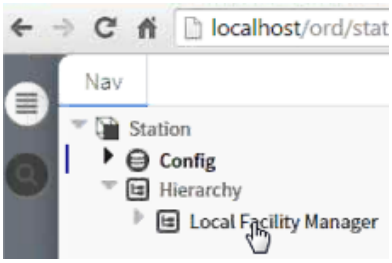
NOTE: Assigning a hierarchy to a role only controls visibility of the top level of that hierarchy. The visibility of elements under any assigned hierarchy are still restricted by the assigned role and its category permissions.

Example



This example shows the `Building Owner` role being edited to assign the `Local Facility Manager` hierarchy.

After a user assigned the `Building Owner` role logs in to the station, the **Local Facility Manager** hierarchy is visible under the Hierarchy space.



Next, the particular **`Role`** must be assigned to a user for the hierarchy to be visible to that user.

Chapter 2 Hierarchy concepts

Topics covered in this chapter

- About the Hierarchy Service
- Tags provide context in a hierarchy
- Hierarchy component
- About level definitions
- Context parameters
- Hierarchy scopes
- Permissions

The following topics describe the basic hierarchy concepts and hierarchy components.

About the Hierarchy Service

The Hierarchy Service provides an efficient method of creating a logical navigation tree for the Niagara system. Rather than defining each element of the tree in a Nav file, the Hierarchy Service allows you to define the navigation tree based on a set of level definition rules referred to as "LevelDefs".

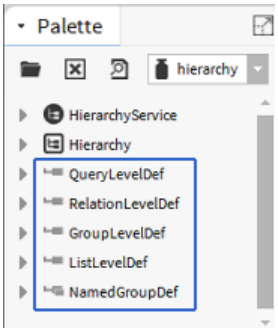
The **`hierarchy`** module is required in order to use hierarchies.

The **HierarchyService**, installed by default in the station's **Services** directory, is the parent container for all hierarchy definitions.

The default view for the **Hierarchy Service** is the Property Sheet view.

The **hierarchy** palette (shown here) provides the **HierarchyService** component, the **Hierarchy** component which you use to create a new hierarchy definition, as well as default level definition components (LevelDefs) which you add to a hierarchy definition to define the node levels within a hierarchy.

Figure 1 The hierarchy palette showing default LevelDef components



Tags provide context in a hierarchy

Hierarchies are based on the tags associated with each object (device, point and component). Tags tell the system, for example, that a specific device is located in a specific building and that a specific point belongs to a specific piece of equipment. The **HierarchyService** uses this contextual information to set up the structure.

All types of tags may be used to structure hierarchies, including implied (default) tags, such as `n:device` and `n:point` as well as Haystack dictionary tags (tags that begin with `hs:`), custom-built dictionary tags, and Ad Hoc tags that you might create while defining the hierarchy. You do not have to create a custom tag dictionary to use Ad Hoc tags in a hierarchy.

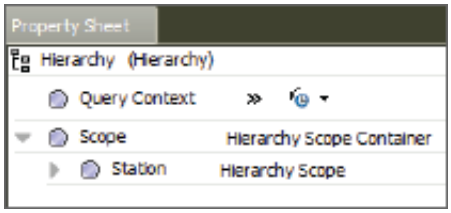
Before creating one or more hierarchies, configure your devices with the tags that will yield the hierarchies that you need.

NOTE: As a convenience, you can fine tune a hierarchy by editing the tags on a component where it appears in the Hierarchy space, rather than navigating to the component in the station logic. Whether your changes are made in the station logic or in the Hierarchy space, they are applied to the same component.

Hierarchy component

Obtained from the **hierarchy** palette, the **`Hierarchy`** component creates the root level of a hierarchy Nav tree structure.

The name of the **`Hierarchy`** component becomes the collective name for the root node in the tree. Examples might be a company or department name, a geographic region or the name of a group of devices that are being monitored together.



Property	Value	Description
`Query Context`	Config Facets dialog box	Sets up the current location's context as a facet. The facet value is compared with the value of the context tag on a device or point at a lower level in the navigation tree.
`Scope`	`station:`	Causes the hierarchy query to search the local station database.
`Scope Ord`	`station:| slot:/...`	Causes the hierarchy query to search within a specific location in the station database

About level definitions

Level definitions (LevelDefs) are used under the **HierarchyService** to define hierarchies. Each hierarchy is defined as a tree of LevelDefs where there is a unique LevelDef for each node of the tree. The two basic types of LevelDefs, Group and Element, are described here:

- Group and list level definitions, basically placeholder folders, set up the structure.
  - A **GroupLevelDef** defines a node based on distinct tag values assigned to devices, points or other components, and provides simple grouping.
  - A **ListLevelDef** works with marker tags and defines a node based on one or more **NamedGroupDef**s (named group definitions). ListLevelDefs require one or more NamedGroupDefs below them.

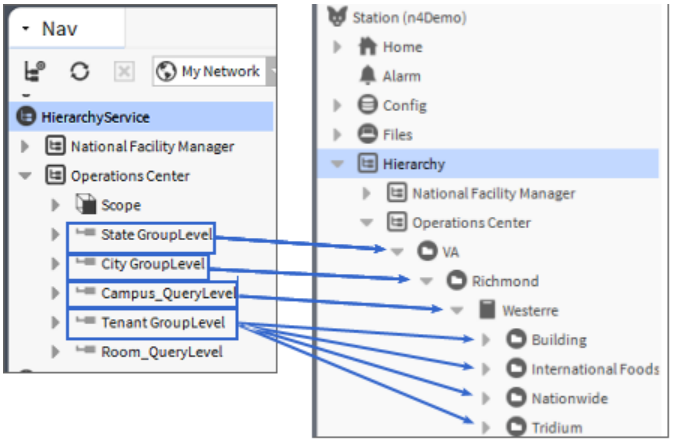
In order to view the actual data, you must add devices and child elements underneath a group or list level definition. You do this using the Element level definitions (either QueryLevelDef or RelationLevelDef).

- Element level definitions set up each NEQL query.
  - A **QueryLevelDef** defines the tags to search on.
  - A **RelationLevelDef** defines a relationship with a parent element to search on.

Level elements (LevelElems) are the nodes presented in an expanded hierarchy in the station Hierarchy space as shown in the following image, where each node in the hierarchy is represented with a LevelElem.

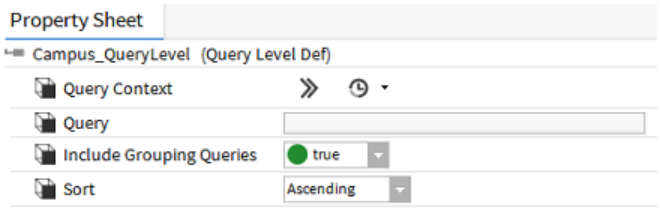
LevelElems result from running the NEQL query at each level of the defined hierarchy. An individual LevelElem typically is associated to a BComponent within the scope of the NEQL query, which is typically a station.

Figure 2 Group and Element LevelDefs in hierarchy definition (left) determine the individual LevelElem nodes in the resulting hierarchy (right)



QueryLevelDef component

Obtained from the **hierarchy** palette, this component sets up a NEQL query that returns the data displayed in a hierarchy. QueryLevelDef is added to a hierarchy definition usually following one or more **`GroupLevelDef`** component or a **`ListLevelDef`**.

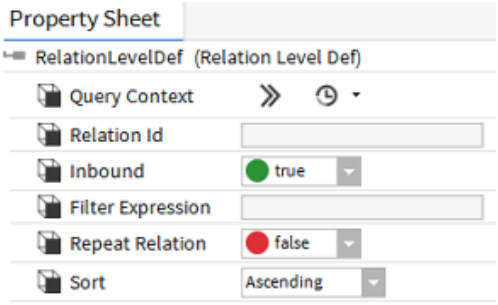


Property	Value	Description
`Query Context`	Config Facets dialog box	Sets up the current location's context as a facet. The facet value is compared with the value of the context tag on a device or point at a lower level in the navigation tree.
`Query`	NEQL query	This property is a NEQL query. You may use all the NEQL operators: and, or, etc. One query has special syntax: `facetname={identifier}`, where `facetname` is the key name you set up for **Query Context** on the definition of a previous node; and `{identifier}` is the tag that identifies the results to display at the current node.

Property	Value	Description
Include Grouping Queries	`true` (default), `false`	**True**: Preceding GroupLevelDef queries in the hierarchy are appended to the current NEQL query. **False**: Prevents preceding GroupLevelDef queries from appending to the current NEQL query, which allows for greater flexibility in structuring a query.
Sort	`None`, `Ascending` (default), `Descending`	Determines the order in which results display.

RelationLevelDef component

Obtained from the **hierarchy** palette, this component defines a query that returns data for all objects that are related to the level immediately above it. The relationship is usually a child relationship (n:child).

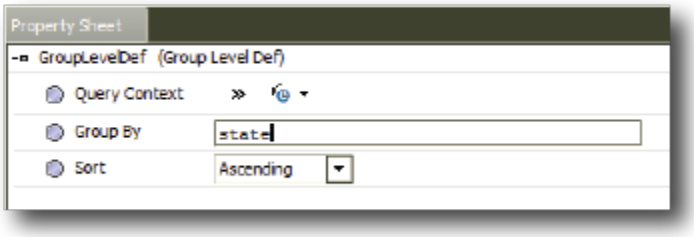


Property	Value	Description
Query Context	Config Facets dialog box	Sets up the current location's context as a facet. The facet value is compared with the value of the context tag on a device or point at a lower level in the navigation tree.
Relation Id	NEQL query	You may use all the NEQL operators: and, or, etc.
Inbound	`true` (default), `false`	Indicates the relation direction.
Filter Expression		Further limits results returned by RelationId by the tags applied to the object at the other end of the relation
Repeat Relation	`true`, `false`	When true, displays query results for devices and points that are related to the **Query Tags** property defined in the **QueryLevelDef** above. When false, ignores results for related devices.
Sort	`None`, `Ascending` (default), `Descending`	Determines the order in which results display.

GroupLevelDef component

Obtained from the **hierarchy** palette, this component sets up a group to contain the results of one or more **`QueryLevelDefs`** that follow.

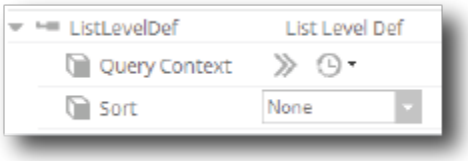
NOTE: At least one **`QueryLevelDef`** is required after a **`GroupLevelDef`**, at any subsequent position.



Property	Value	Description
Query Context	Config Facets dialog box	Sets up the current location's context as a facet. The facet value is compared with the value of the context tag on a device or point at a lower level in the navigation tree.
Group By	Text	Defines one or more tags to use for grouping query results at the current level. For example, `state` groups all resulting data by the state tag on the device: AZ, CA, VA, etc.
Sort	`None`, `Ascending` (default), `Descending`	Determines the order in which results display.

ListLevelDef component

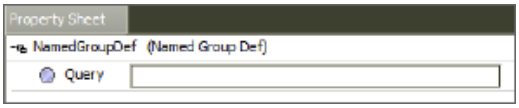
Obtained from the **hierarchy** palette, this component works with marker tags. **ListLevelDef** must be followed by one or more **NamedLevelDefs** and at least one **QueryLevelDef**.



Type	Value	Description
Query Context	Config Facets dialog box	Sets up the current location's context as a facet. The facet value is compared with the value of the context tag on a device or point at a lower level in the navigation tree.
Sort	`None`, `Ascending` (default), `Descending`	Determines the order in which results display.

NamedGroupDef component

Obtained from the **hierarchy** palette, this component works in conjunction with **ListLevelDef** and **RelationLevelDef**. It allows you to add one or more placeholder folders (nodes) under the **ListLevelDef**.



Property	Value	Description
Query	NEQL query	This property is a NEQL query. You may use all the NEQL operators: and, or, etc.

Context parameters

Context parameters on a **LevelDef** can be used to pass context sensitive information to lower level definitions. You can use a **Query Context** to store any name value pair, but a more powerful use is to store context sensitive data via facets.

If we add a String facet to a **Query Context** where the value of that facet is a tag name, the tag name is evaluated against the results returned by the **LevelDef** *and* the value stored in the **Query Context**. See also, Query context example, page 21

NOTE: When adding facets to a **Query Context**, the facet values for tag names must be Strings.

When comparing tag value to values from the **Query Context**, make sure the resulting types are the same. From the Query context example, page 21, with `equipId=hs:id` and `hs:equipRef={equipId}`, the "type" for the values: `hs:id` and `hs:equipRef` must be the same (BOrds in this example).

Hierarchy scopes

The **Scope** container under each hierarchy can contain one or more **HierarchyScopes** over which the hierarchy can be generated. The default is the station (or Component Space) scope. In later releases, more scopes will be available over which to build hierarchies.

Permissions

Roles have a **Viewable Hierarchies** property that allows you to assign on a per role basis which hierarchies are visible to a user.

Users with the **Admin** role can always view all hierarchies and due to their super user permissions can view everything under those hierarchies. All other users are assigned permissions to view a hierarchy via their role(s).

In the **Role Service**, when editing a role in the **Role Manager**, you can select which hierarchies are visible to that role. A user will be able to see all hierarchies that are assigned to their role.

NOTE: Assigning a hierarchy to a role only controls visibility of the top level of that hierarchy. The visibility of elements under any assigned hierarchy are still restricted via the **Role** and its **Category** permissions.

For more details on categories, roles, and permissions, refer to the "Authorization Management" section of the *Station Security Guide*

Chapter 3 Examples

Topics covered in this chapter

- Display all points example
- Query context example
- Multi-user example
- NEQL query examples

The following examples are designed to help you plan your tag requirements in advance.

Display all points example

This is a straight-forward example of how to create a hierarchy that displays real-time results from all points configured for three AHU units owned by the same company.

Figure 3 The resulting Nav tree

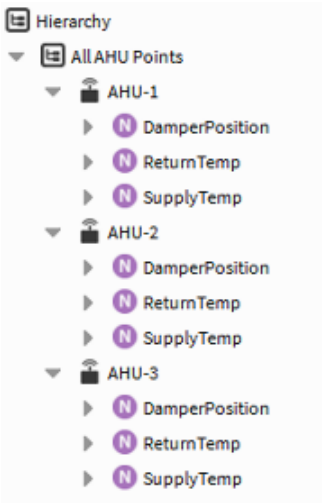
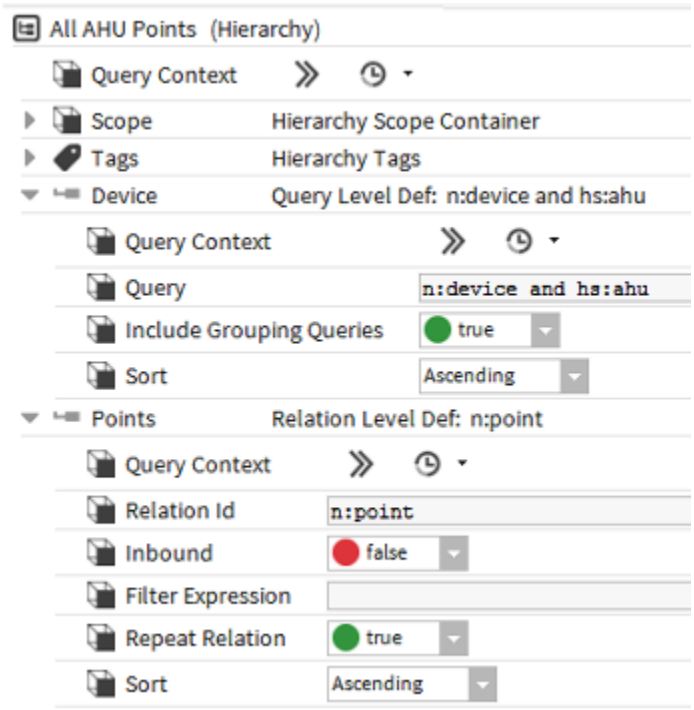


Figure 4 All AHU Points hierarchy definition setup



NOTE: Red icons on a hierarchy definition indicate that a new hierarchy has been created or changes have been made the hierarchy needs to be saved.

This table explains each property in the hierarchy definition.

Level name	Property	Where set up	Comments
Scope	**Scope**: `Station`	Station selected by default in the hierarchy definition.	This property limits the range of the hierarchy to the station database. In later Niagara 4 releases, **Scope** will allow queries and hierarchy generation based on other specified spaces, such as **System Db**.

Tags	**Hierarchy Tags**	Tags added to components that are not BComponents	None in this example. When the navigation hierarchy is built, picks up tags applied to LevelElems (nodes) that are not BComponents
Device (QueryLevelDef)	**Query Context**	Selected here in the hierarchy definition.	Not used in this example.
	Query: `n:device`	Direct tag added to each AHU.	Returns all components tagged as "devices".
	`and hs:ahu`	Direct tag added to each AHU.	Further narrows the search results to only devices tagged with "hs:ahu".
	Sort: `Ascending`	Selected here in the hierarchy definition.	Defines the display sequence.
Points (RelationLevelDef)	**Relation Id**: `n:point`	Implied tag on each point.	Returns all points under the station beginning at the station root. Results are limited to AHU child relations tagged with n:point.
	Inbound: `false`	Selected here in the hierarchy definition.	Indicates the relation direction setup on the object (target) component of a relation.
	Filter Expression:	Set up here in the hierarchy definition.	Not used in this example.
	Repeat Relation: `true`	Set up here in the hierarchy definition.	Causes the hierarchy to include any children of the points. Device children are any components under the device, such as alarms, histories, schedules, etc. Children of each point (grandchildren of the device) include point extensions.

NOTE:

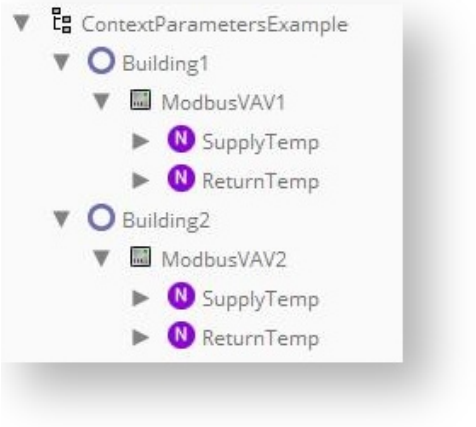
Referencing the implied `n:point` relation between devices and their points, you can easily create hierarchy displays that omit of the Points folder.

Query context example

This example has two buildings, each with a Modbus network and Modbus Variable Air Volume (VAV) controller. The goal is to monitor supply and return temperatures for each controller. Specifically, this example demonstrates how the query context works.

Hierarchy result

The resulting Nav tree hierarchy under the Hierarchy space looks like this:



The software monitors the same two points for each device. Without the query context all four points would appear below each VAV device.

Building setup

Each building is a system component set up under the **Drivers** folder in the Nav tree. `Building1` and `Building2` are arbitrary names assigned to the buildings when they are set up as components. Each building is tagged with two Haystack tags:

- The Haystack tag `hs:site` identifies the object as a physical site.
- Like a serial number, the implied Haystack tag `hs:id=h:[identifier]`, where `[identifier]` is a four-digit number that provides unique identification for each building. For Building1 `hs:id` is `h:6496`, and for Building2 it is `h:6497`.

Device setup

The network and VAV controllers are set up under the **Drivers** folder. `ModbusVAV1` and `ModbusVAV2` are the arbitrary equipment names. Each device is tagged with these tags:

- The Haystack tag `hs:equip` identifies the object as a physical device.
- Like a serial number, the implied Haystack tag `hs:id=h:[deviceID]`, (where `[deviceID]` is a four-digit identifier) provides unique identification for each device. In the example, ModbusVAV1's `hs:id` is `h:3efc`, and ModbusVAV2's `hs:id` is `h:3f00`.
- The Haystack tag `hs:siteRef=h:[buildingId]` (where `[buildingId]` is the building number) associates each device with the building in which it is located: for ModbusVAV1, `hs:siteRef=6496` and for ModbusVAV2, `hs:siteRef=6497`.

Points setup

VAV performance is monitored by two points (ModbusClientPointDeviceExt):

- `SupplyTemp`
- `ReturnTemp`

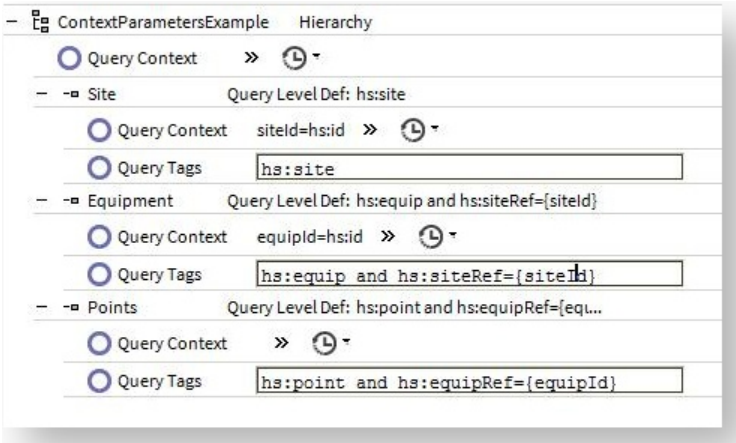
Each point has these Haystack tags associated with it:

- `hs:point` identifies this object as a point.
- `hs:equipRef=h:[deviceID]` (where `[deviceID]` is a four-digit identifier) associates each point with the equipment to which it belongs. For the points associated with ModbusVAV1, `hs:equipRef=h:3efc`, and for the points associated with ModbusVAV2, `hs:equipRef=h:3f00`

Hierarchy definition

The hierarchy definition consists of three **QueryLevelDefs**.

Figure 5 Context Parameter Example hierarchy definition



This table explains each property in the hierarchy definition.

Level Name	Property	Where set up	Processing
site (QueryLevelDef)	**Query Context**: `siteId=hs:id`	Implied name, value tag on each building component: `hs:Id=[identifier]`, where `[identifier]` is a unique string that identifies the site (Building 1 and Building 2)	Establishes the context for the equipment level (next level in the tree) by storing the value of each building's `hs:id` (Building 1 or Building 2) in the config facet `siteId`.
	Query Tag: `hs:site`	Name, value tag on each building component `hs:site = 6496` or`hs:site = 6497`	Returns components tagged with `hs:site`.
equipment (QueryLevelDef)	**Query Context**: `equipId=hs:id`	`hs:Id=[identifier]`, (where `[identifier]` is a unique string that identifies the equipment). The equipment Id for ModbusVAV1 is h:3efc, and that for ModbusVAV2 is 3f00.	Establishes the context for the points level by storing the value of each device's `hs:id` in the config facet `equipId`.
	Query Tag: `hs:equip`	Implied marker tag on each equipment component.	Returns all devices tagged with the implied tab: `hs:equip`.
	Query Tag: `hs:siteRef={siteId}`	Set up by the **site** query context in this hierarchy definition.	Compares the value of the device's `hs:siteRef` tag with the `siteId` context passed down from the **Site** level in the definition.
points (QueryLevelDef)	**Query Tag**: `hs:point`	Implied marker tag on each point.	Returns all data for all points in the system
	Query Tag: `hs:equipRef={equipId}`	Set up by the **equipment** query context in this hierarchy definition.	Compares the value of the device's `equipRef` with the `equipId` passed down from the **equipment** level in the definition.

Context processing

In this example:

- The `siteId=h:6496` for Building 1 matches the `siteRef=h:6496` tag on ModbusVAV1 causing ModbusVAV1 to appear under Building 1 in the hierarchy.
- The `siteId=h:6497` for Building 2 matches the `siteRef=h:6497` tag on ModbusVAV2, causing ModbusVAV2 to appear under Building 2 in the hierarchy.

Continuing with the processing of points:

- The `equipId=h:3efc` matches the `equipRef=h:3efc` tag on two points labeled `SupplyTemp` and `ReturnTemp`, causing these points to appear under ModbusVAV1.
- The `equipId=h:3f00` matches the `equipRef=h:3f00` tag on two additional points `SupplyTemp` and `ReturnTemp`, causing them to appear under ModbusVAV2.

Without using config facets to store the site and equipment contexts, a query would place all equipment below each building and all points below each piece of equipment regardless of where the equipment and points actually belong.

NOTE: When adding facets to the **`Query Context`**, the facet values for tag names should be Strings.

When comparing tag value to values from the **`Query Context`**, make sure the resulting types are the same. from the example above with `equipId=hs:id` and `hs:equipRef={equipId}`, the types in the values of `hs:id` and `hs:equipRef` must be the same (BOrds in this example).

Multi-user example

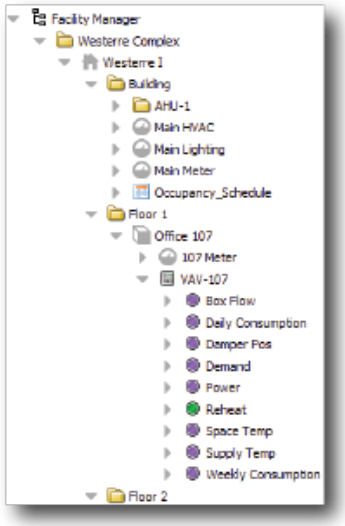
This example features a campus of two multi-tenant buildings: Westerre I and Westerre II. The Niagara system monitors each building's lighting systems and HVAC equipment. The example illustrates how to structure hierarchies in a Supervisor station for three users: facilities manager, system integrator, and operations center.

Three hierarchies

All three users require a hierarchy that displays data by building, but under the node for each building, their tree structures differ.

- To monitor usage for the entire building as well as by floor, the facility manager prefers that the data to be displayed by floor:

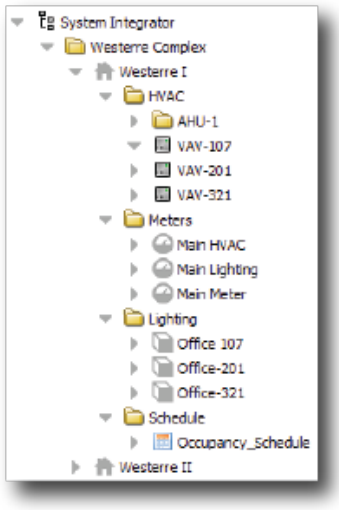
Figure 6 Facility Manager hierarchy expanded to show VAV points



Each floor expands in a similar fashion to Floor 1 in the screen capture. Westerre II follows the same structure. This is the hierarchy definition fully explained in this example topic.

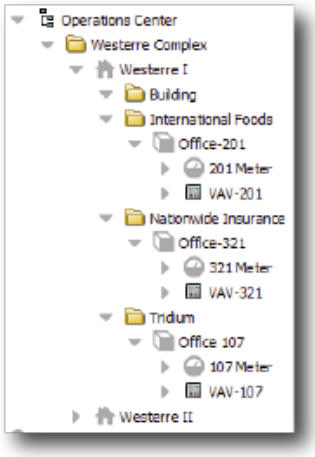
- The systems integrator, whose main interest is monitoring device performance, prefers to view the same data by device.

Figure 7 Systems Integrator hierarchy fully expanded



- Finally, the operations center monitors the data by tenant and individual office.

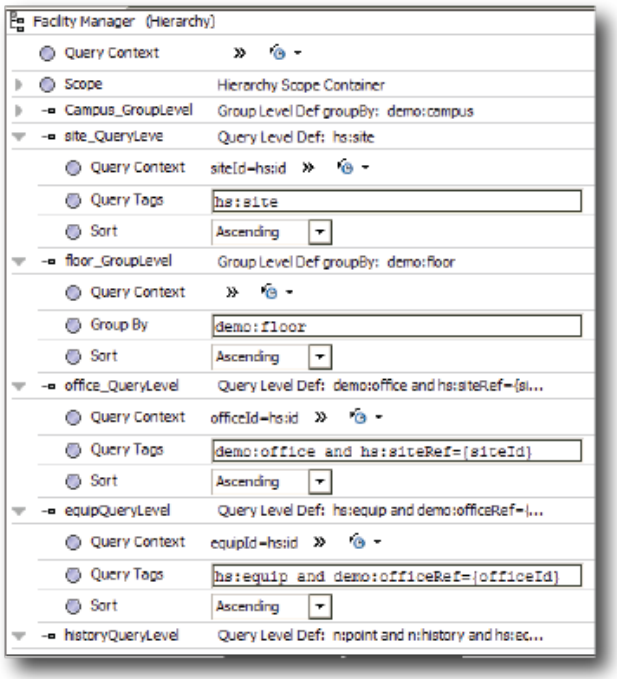
Figure 8 Operations Center hierarchy fully expanded



Definition for Facility Manager hierarchy

Now that you know the goal, here is what the hierarchy structure looks like for the FacilityManager:

Figure 9 Facility Manager hierarchy definition



All tags, including implied (default) and dictionary tags are available for constructing hierarchies. The table that follows explains each of six levels for the Facility Manager's hierarchy definition. After studying this example you should be able to understand the hierarchies for the Systems Integrator and Operations Center. The level names (left column) were set up by the hierarchy designer.

Level name	Property	Where set up	Result
Campus_GroupLevel	**Group By:** `demo:campus`	Name, value tag on each building component; the value of `campus` is "Westerre Complex"	"Westerre Complex" appears as the first node in the hierarchy under "Facility Manager"

site_QueryLevel	**Query Context**: `siteId=hs:id`	Implied tag on each building component: `hs:Id=[identifier]`, (where `[identifier]` is a unique string that identifies the site)	Sets `siteId` equal to the value of the site's `hs:id` and passes `siteId` down to the next level in the hierarchy definition
	Query: `hs:site`	Marker tag on each building component	Identifies the component as a location, causing the site names to appear as nodes under "Westerre Complex"
floor_GroupLevel	**Group By**: `demo:floor`	Name, value tag on each office: `demo:=Floor [n]`, (where `[n]` is 1, 2, 3 or 4)	Identifies the floor on which each office is located, causing the floor to appear in the hierarchy
office_QueryLevel	**Query Context**: `officeId=hs:id`	`hs:Id=[identifier]`, (where `[identifier]` is a unique string that identifies the office) is an implied tag on each office.	Sets `officeId` equal to the value of the office's `hs:id` and passes `officeId` down to the next level in the hierarchy definition.
	Query: `demo:office`	Marker tag on each office	Identifies the component as an office, causing the office names to appear as nodes under each floor in the hierarchy
	Query: `hs:siteRef={siteId}`	`siteRef` is a name, value tag on each office. The value of this tag is the hs:id of the site (building) in which the office is located.	Compares the `hs:siteRef` tag on the office) with the `siteId` passed down from the **site_QueryLevel**. A match ensures that the office appears under the correct building in the hierarchy.
equip_QueryContext	**Query**: `hs:equip`	On each device	Identifies the device as a piece of equipment. This tag causes the equipment names to appear as nodes under each office in the hierarchy.
	Query: `demo:officeRef={officeId}`	`officeRef` is a name/value tag on each device. The value of this tag is the `hs:id` of the office in which the device is located.	Compares the `demo:officeRef` tag on the equipment with the `officeId` passed down from the **office_QueryLevel**. A match ensures that the equipment appears under the correct office in the hierarchy.
	Query Context: `equipId=hs:id`	`hs:Id=[identifier]`, (where `[identifier]` is a unique string that identifies the office) is an implied tag on each office.	Sets `equipID` to the value of the device's `hs:id` and passes `equipId` down to the next level in the hierarchy definition.
history_QueryLevel	**Query**: `n:point`	Implied tag on each point.	Returns data from all device points.
	Query: `n:history`	Implied tag on each point.	Returns all histories for the device.
	Query: `hs:equipRef={equipId}`	Name, value tag on each point. The value of this tag is the `hs:id` of the device in which the point is located.	Compares the `hs:equipRef` tag on each point with the `equipId` passed down from the **equip_QueryLevel**. A match ensures that the point appears under the correct device in the hierarchy.

About assigning hierarchies to Roles

Roles have a **`Viewable Hierarchies`** property that allows you to assign on a per role basis which hierarchies are visible to a user. By assigning a hierarchy to a specific role you are able to control the visibility of the entire hierarchy (and grouping elements within the hierarchy). Only the users assigned to that role are able to view the hierarchy.

NOTE: Users with the **Admin** role can always view all hierarchies and due to their super user permissions can view everything under those hierarchies. All other users are assigned permissions to view one or more hierarchies via their role(s).

NEQL query examples

The Niagara Entity Query Language (NEQL) provides a mechanism for querying tagged entities in Niagara applications. NEQL only queries for tags. NEQL supports parameterized queries and allows you to use the same syntax as is used in Search. The following examples are designed to help you construct queries.

Examples

To query for	Syntax example	Returns result
point tag	`n:point`	Any entity with the point tag (in the "n" namespace)
name tag = "foo"	`n:name = "foo"`	Any entity with the "foo" name tag (in the "n" namespace)
type tag = "baja:Folder"	`n:type = "baja:Folder"`	Any entity with the "baja:Folder" type tag (in the "n" namespace)
point tag and name tag = "foo" and type tag = "baja:Folder" and hs:coolingCapacity > 4.03	`point and name = "foo" and type = "baja:Folder" and hs:coolingCapacity > 4.03`	Any entity with the point tag and any entity with the foo tag (in the "n" namespace) and any entity with the "baja:Folder" type tag (in the "n" namespace) and any entity with the coolingCapacity tag (in the "hs" namespace) with a value greater than 4.03
foo tag	`t:foo`	Any entity with the foo tag (in the t namespace)
entities containing child entities tagged with "foo"	`n:child->t:foo`	Any parent (`n:child->t:foo`) of any entity with the foo tag
entities containing other entities with child entities tagged with "foo"	`n:child->n:child->t:foo`	Any grandparent (`n:child->n:child->t:foo`) of any entity with the foo tag
entities with t:foo tag or entities containing child entities with t:foo tag or entities containing other entities with child entities tagged with t:foo tag	`t:foo` or `n:child->t:foo` or `n:child->n:child->t:foo`	Any entity with the foo tag (in the t namespace) or any parent of any entity with the foo tag or any grandparent of any entity with the foo tag
where `n:pxView` is a relation	`n:pxView->n:type`	Any entity with a pxView relation where the endpoint is a niagara type. This is all the entities that have px views.

NEQL Grammar

<statement> := <full select> | <filter select>

<full select> := select (<tag list>) where <predicate>

<filter select> := <predicate>

<tag list> := <tag> (, <tag>)*

<tag> := <word>(: <word>) // either namespace:key or key

<predicate> := <condOr>
<condOr> := <condAnd> ("or" <condAnd>)*
<condAnd> := <term> ("and" <term>)*
<term> := <val> | <cmp>
<cmp> := <val> <cmpOp> <val>
<cmpOp> := "=" | "!=" | "<" | "<=" | ">" | ">="
<val> := <number> | <bool> | <str> | <tag>
<bool> := "true" or "false"
<number> := <int> | <double>
<bool> := true | false
<str> := " <chars> "

Index