

Technical Document

Niagara Hierarchies Guide



About this guide

This topic contains important information about the purpose, content, context, and intended audience for this document.

Product Documentation

This document is part of the Niagara technical documentation library. Released versions of Niagara software include a complete collection of technical information that is provided in both online help and PDF format. The information in this document is written primarily for Systems Integrators. To make the most of the information in this book, readers should have some training or previous experience with Niagara software, as well as experience working with JACE network controllers.

Document Content

This Niagara 4 Niagara Hierarchies Guide provides an introduction to the use of hierarchies to create any number of logical navigation trees which are not limited to representing the actual system structure. The use of hierarchies allows for easy access to system information and components.

This document provides conceptual information about the hierarchies feature, as well as procedural information describing how to set up and edit hierarchy definitions and assign hierarchies to roles.

- [Document change log](#)
Updates (changes and additions) to this document are listed below.
- [Related documentation](#)
Additional information is available in the following documents.

Document change log

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

June 6, 2022

Added Note for namespace and query search details in “NEQL query examples” topic.

Parent topic: [About this guide](#)

Related documentation

Additional information is available in the following documents.

- *Niagara Tagging Guide*
- *Niagara Relations Guide*
- *Niagara Station Security Guide*
- *Niagara Developer Guide*
- *Niagara System Database and System Indexing Guide*

Parent topic: [About this guide](#)

Common hierarchy tasks

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.

- **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.

- **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.

- **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.

- **Viewing implied tags using Spy view**

Implied tags and implied relations are automatically assigned to objects by rules in the installed **Smart Tag Dictionaries**. The implied tags and implied relations do not appear in the **Property Sheet** view or other more commonly used views. **Spy** view shows all of the direct and implied tags and relations on an object as well as other detailed data. Although intended to be used for diagnostic purposes, you can use **Spy** view to identify implied tags and/or relations already assigned to a component. This can be useful when developing hierarchies. Once identified, you can then create queries for those tags/relations in your hierarchy definition.

- **Accessing hierarchies scoped against the SystemDb**

You can access hierarchies scoped against the SystemDb, which allows the Supervisor station to navigate hierarchies with data based on entities from the remote JACE stations. The purpose of the System Database is to allow you to run queries against your entire Niagara system (the Supervisor and those Niagara stations connected to it), which allows your searches and hierarchies to span the entire system.

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) those changes are not automatically reflected in the individual hierarchy tree (in the Hierarchy space). To view changes in an individual hierarchy tree, right-click that hierarchy node in the Hierarchy space and select **Refresh Tree Node**. This updates the hierarchy tree 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.

Parent topic: [Common hierarchy tasks](#)

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.

- 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*.

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.
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.
3. In the right pane, click on AHU Points to open it's property sheet.
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 query to search only this specific location within the station. An alternative to limiting the scope is to use additional LevelDefs.

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**.
6. In the property sheet, click to expand the new Device level definition and perform the following action:

Property name	Action
Query	<p>Enter: n:device and nBld:ahu to return any objects tagged with n:device and nBld:ahu.</p> <hr/> <p>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 to determine how they are tagged or to apply tags.</p> <hr/>

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**.
8. In the property sheet, click to expand the new Points level definition and perform the following actions:

Property name	Action
Outbound Relation Ids	Enter: n:childPoint to return any child point components.
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

9. Click **Save**.
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.

There is added support for multiple relation IDs on a single relation level definition. You can enter more than one comma-separated tagID in the Inbound Relation IDs or Outbound Relation IDs properties. The results for any of these relation IDs display in the hierarchy.

Parent topic: [Common hierarchy tasks](#)

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.

- An existing Nav tree hierarchy in the **Hierarchy** space
 1. Expand the **HierarchyService** and double-click the hierarchy definition to edit.
 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
 3. Click **Save**.
 4. To view your changes in the hierarchy, right-click the Hierarchy space node in the Nav tree and click **Refresh Tree Node**.

Parent topic: [Common hierarchy tasks](#)

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.

- The hierarchy you wish to assign exists in the Hierarchy space.
 1. To open the **RoleManager** view, double-click the **RoleService**.
 2. Double-click to edit an existing role.
 3. In the **Edit** window, click the chevron icon to the right of Viewable Hierarchies.
 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

As an example, the Building Owner role is 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.

Parent topic: [Common hierarchy tasks](#)

Related reference

[Permissions](#)

Viewing implied tags using Spy view

Implied tags and implied relations are automatically assigned to objects by rules in the installed **Smart Tag Dictionaries**. The implied tags and implied relations do not appear in the **Property Sheet** view or other more commonly used views. **Spy** view shows all of the direct and implied tags and relations on an object as well as other detailed data. Although intended to be used for diagnostic purposes, you can use **Spy** view to identify implied tags and/or relations already assigned to a component. This can be useful when developing hierarchies. Once identified, you can then create queries for those tags/relations in your hierarchy definition.

- You are connected to your station.
- [One or more installed tag dictionaries. If necessary, add required tag dictionaries to the TagDictionaryService.](#)

Note: If tagging offline, it is possible that no dictionaries are available. In that situation, the system searches for tag dictionaries in alternate locations.

- [One or more installed tag dictionaries. If necessary, add required tag dictionaries to the TagDictionaryService](#)

Note: If tagging offline, it is possible that no dictionaries are available. In that situation, the system searches for tag dictionaries in alternate locations.

This procedure describes how to open the **Spy** view on a station component to see its implied tags:

Note: Invoking the **Edit Tags** window is another method for viewing the direct and implied tags assigned to a component.

1. In the Nav tree, right-click the component of interest and click **Views > Spy Remote** from the popup menu.
Spy information displays in the **Web Browser View**.
2. Scroll down to **Tags Implied**.
The implied tags assigned to the selected component are listed. Scroll up or down to view all of the tags and relations assigned to the component.

Parent topic: [Common hierarchy tasks](#)

Accessing hierarchies scoped against the SystemDb

You can access hierarchies scoped against the SystemDb, which allows the Supervisor station to navigate hierarchies with data based on entities from the remote JACE stations. The purpose of the System Database is to allow you to run queries against your entire Niagara system (the Supervisor and those Niagara stations connected to it), which allows your searches and hierarchies to span the entire system.

- You have an open connection to a Supervisor station (via Workbench or browser).
- The SystemDb is installed and configured on the Supervisor.
- In order for any tagged entities to resolve correctly in virtual hierarchies, you must have installed the same tag dictionaries on the Supervisor station as are installed in the subordinate stations.
- Entities in the subordinate stations are already tagged as needed.
- You have the hierarchy palette open.
- You have successfully indexed subordinate stations in your NiagaraNetwork, and Niagara virtuals must

be enabled and set up properly for user permissions.

Although not a prerequisite, you may want to copy existing hierarchy definitions configured on the subordinate stations and paste those into the HierarchyService on the Supervisor station. Also, you need to modify those definitions to point the hierarchy scope to the System Database. Otherwise, you need to create a properly scoped hierarchy definition as described here.

1. In the Nav Tree of the Supervisor station, expand the **Services** node.
2. In the hierarchy palette, drag the **Hierarchy** component to the station's **HierarchyService** and in the **Name** window, enter the desired name. Alternatively, there is a preconfigured SystemDbHierarchy component in the hierarchy palette that already has the SystemDb ("sys:") scope configured.
3. Open a **Property Sheet** view of the **HierarchyService**, expand **Hierarchy > Scope > Station** and in the **ScopeOrd** field replace the default value by entering: **SYS:**.
You have changed the Hierarchy scope to run against the SystemDb.
4. Continue to add level definitions as needed (based on objects and tags as used in the subordinate stations and how you wish to navigate those stations). For more details, see procedure "Setting up a Hierarchy Definition".
5. Right-click in the **Nav Tree** and select **Refresh Tree Node**.
6. Expand the station's **Hierarchy** space and navigate throughout the entities (including points and schedules) located on the subordinate stations as organized by the hierarchies.

For more details on SystemDb, see the *Niagara System Database and System Indexing Guide*.

Parent topic: [Common hierarchy tasks](#)

Related reference

[Hierarchy scopes](#)

Hierarchy reference

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*”.
- [Tags provide context in a hierarchy](#)
Hierarchies are based on the tags and relations 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 for individual hierarchies.
- [Hierarchy component](#)
Obtained from the **hierarchy** palette, the Hierarchy component creates the root level of a hierarchy Nav tree structure.
- [Caching hierarchies](#)
There is added support for caching hierarchies on the station (server-side) which improves hierarchy performance by greatly reducing the length of time it takes to render an expanded hierarchy on the client side (Workbench and web browsers). Using cached hierarchies makes the Workbench and web browser clients more responsive. However, the cache does reduce available heap memory on the station side.
- [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 Entity, are described here:
- [Context parameters](#)
Context parameters on a **LevelDef** can be used to pass context sensitive information to subsequent (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.
- [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.
- [Permissions](#)
Roles have a Viewable Hierarchies property that allows you to assign on a per role basis which hierarchies are visible to a user.

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*”.

Hierarchy Service properties

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, as shown.

Services : HierarchyService Property Sheet

HierarchyService Actions & Topics Slot Details

Display Name	Value	Comm.
Status	{ok}	
Fault Cause		
Enabled	<input checked="" type="checkbox"/> true	
Hierarchy Timeout	0h 0m 45s	

Property	Value	Description
Status	read-only	Reports the condition of the entity or process at last polling. {ok} indicates that the component is licensed and polling successfully. {down} indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection. {disabled} indicates that the Enable property is set to false. {fault} indicates another problem. Refer to Fault Cause for more information.
Fault Cause	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
Enabled	true or false	Activates (true) and deactivates (false) use of the object (network, device, point, component, table, schedule, descriptor, etc.).
Hierarchy Timeout	00000h 00m 00s (hours, minutes, seconds), 45 seconds (default)	This property allows you to configure the wait time for hierarchy query processing. When navigating a hierarchy, if the time it takes to process a hierarchy query exceeds the hierarchy timeout value, an error message displays in Niagara 4. You may increase the timeout value to allow a longer wait time for results, or decrease the timeout value to abandon hierarchy query processing earlier.

Hierarchy palette

The hierarchy palette 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 must add to a hierarchy definition to define the node levels within a hierarchy.

Parent topic: [Hierarchy reference](#)

Tags provide context in a hierarchy

Hierarchies are based on the tags and relations 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 for individual hierarchies.

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 when tagging components in the station. You do not have to create a custom tag dictionary to use Ad Hoc tags, they can be created as needed. Although, using tags in a tag dictionary ensures consistency which typically yields better results.

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.

Parent topic: [Hierarchy reference](#)

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.

Figure 1. Hierarchy component property sheet

Topology (Hierarchy)	
Query Context	>> ⌚
Status	{ok}
Fault Cause	
▼ Scope	Hierarchy Scope Container
▶ Station	Hierarchy Scope
▶ Tags	Hierarchy Tags
Cache Status	Cached
Cache Creation Time	05-Oct-2017 10:03 AM EDT
Cache On Station Started	<input checked="" type="radio"/> false
▶ GroupLevelDef	Group Level Def groupBy: n:geoCountry
▶ GroupLevelDef1	Group Level Def groupBy: n:geoState
▶ GroupLevelDef2	Group Level Def groupBy: n:geoCity
▶ GroupLevelDef3	Group Level Def groupBy: n:building
▶ GroupLevelDef4	Group Level Def groupBy: n:Floor
▶ QueryLevelDef	Query Level Def: c:PressureSP OR c:Temp...

Hierarchy properties

Property	Value	Description
Query Context	Config Facets window	<p>Sets up the current location's context as a facet. The current location is one item that could be placed in the query context. 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.</p> <p>The query context can hold anything to be used in LevelDef queries. It is comparable to a hierarchy scoped variable. You could create multiple copies of such a hierarchy definition and then customize each copy with the query context of the specific hierarchy.</p>
Status	read-only	<p>Reports the condition of the entity or process at last polling.</p> <p>{ok} indicates that the component is licensed and polling successfully.</p> <p>{down} indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection.</p> <p>{disabled} indicates that the Enable property is set to false.</p> <p>{fault} indicates another problem. Refer to Fault Cause for more information.</p>
Fault Cause (general)	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
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
Tags	GroupLevelDefn tags	<p>Applies additional tags. For example, you can set the display name of the hierarchy definition as you might for any other component. The display name will be used when the hierarchy is displayed under the hierarchy space and the <code>n:displayName</code> tag stores this display name. The name of the hierarchy definition and not the display name is used in hierarchy ords so hierarchy ords will continue to work even if the display name of a hierarchy is changed. The display name might be changed to make the hierarchy space more user friendly (display names can accept more than regular component names).</p>
Cache Status	read-only (defaults to Not Cached (default), Caching, Cached, Caching Failed, Not Cached On Started	Shows whether or not the hierarchy is cached or the status of a caching operation that is in progress. <code>Cached</code> indicates there is an existing cache of the hierarchy.

Property	Value	Description
		<p><code>Not Cached</code> indicates that either an existing cache was cleared, or the hierarchy has never been cached.</p> <p><code>Caching</code> indicates that there is a job running that is creating a cache of the hierarchy.</p> <p><code>Caching Failed</code> indicates that something went wrong during cache creation and a cache of the hierarchy does not exist. See the caching job log for details regarding the failure.</p> <p><code>Not Cached on Startup</code> indicates that the hierarchy property <code>Cache On Station Started</code> is true the station start did not start a caching job because either the <code>niagara.hierarchy.caching.disableOnStationStarted</code> or <code>niagara.hierarchy.caching.disabled</code> system property is set to true. A cache of the hierarchy does not exist.</p>
Cache Creation Time	read-only	Shows date/time that the current cache was created or null if a cache of the hierarchy does not exist.
Cache On Station Started	true, false (default)	<p>Configures if a cache job should run on station start.</p> <p>true starts a job once the station that builds a cache of this hierarchy starts.</p> <p>false starts nothing on station start.</p>
Property	Value	Description
Query Context	Config Facets window	<p>Sets up the current location's context as a facet. The current location is one item that could be placed in the query context. 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.</p> <p>The query context can hold anything to be used in <code>LevelDef</code> queries. It is comparable to a hierarchy scoped variable. You could create multiple copies of such a hierarchy definition and then customize each copy with the query context of the specific hierarchy.</p>
Status	read-only	<p>Reports the condition of the entity or process at last polling.</p> <p><code>{ok}</code> indicates that the component is licensed and polling successfully.</p> <p><code>{down}</code> indicates that the last check was unsuccessful, perhaps because of an incorrect property, or possibly loss of network connection.</p> <p><code>{disabled}</code> indicates that the <code>Enable</code> property is set to false.</p> <p><code>{fault}</code> indicates another problem. Refer to <code>Fault Cause</code> for more information.</p>

Property	Value	Description
Fault Cause (general)	read-only	Indicates the reason why a system object (network, device, component, extension, etc.) is not working (in fault). This property is empty unless a fault exists.
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
Tags	GroupLevelDefn tags	Applies additional tags. For example, you can set the display name of the hierarchy definition as you might for any other component. The display name will be used when the hierarchy is displayed under the hierarchy space and the <code>n:displayName</code> tag stores this display name. The name of the hierarchy definition and not the display name is used in hierarchy ords so hierarchy ords will continue to work even if the display name of a hierarchy is changed. The display name might be changed to make the hierarchy space more user friendly (display names can accept more than regular component names).
Cache Status	read-only (defaults to <code>Not_Cached</code> (default), <code>Caching</code> , <code>Cached</code> , <code>Caching Failed</code> , <code>Not_Cached On Started</code>)	Shows whether or not the hierarchy is cached or the status of a caching operation that is in progress. <code>Cached</code> indicates there is an existing cache of the hierarchy. <code>Not_Cached</code> indicates that either an existing cache was cleared, or the hierarchy has never been cached. <code>Caching</code> indicates that there is a job running that is creating a cache of the hierarchy. <code>Caching Failed</code> indicates that something went wrong during cache creation and a cache of the hierarchy does not exist. See the caching job log for details regarding the failure. <code>Not_Cached on Startup</code> indicates that the hierarchy property <code>Cache On Station Started</code> is true the station start did not start a caching job because either the <code>niagara.hierarchy.caching.disableOnStationStarted</code> or <code>niagara.hierarchy.caching.disabled</code> system property is set to true. A cache of the hierarchy does not exist.
Cache Creation Time	read-only	Shows date/time that the current cache was created or null if a cache of the hierarchy does not exist.
Cache On Station Started	true, false (default)	Configures if a cache job should run on station start. true starts a job once the station that builds a cache of this hierarchy starts. false starts nothing on station start.

Actions

These caching-related actions are applied manually per hierarchy definition via the right-click menu.

- **Create Cache** — builds the hierarchy cache; an existing cache is deleted before the new cache is built
- **Clear Cache** — deletes the existing hierarchy cache

Additionally, you could link the caching action to a trigger schedule so that the cache is recreated on a regular schedule, such as each night at 3:00 a.m.

Parent topic: [Hierarchy reference](#)

Related concepts

[Caching hierarchies](#)

Caching hierarchies

There is added support for caching hierarchies on the station (server-side) which improves hierarchy performance by greatly reducing the length of time it takes to render an expanded hierarchy on the client side (Workbench and web browsers). Using cached hierarchies makes the Workbench and web browser clients more responsive. However, the cache does reduce available heap memory on the station side.

Caching is applied manually per hierarchy definition. If you have more than one hierarchy definition, each one must be cached separately via an action on the right-click menu. And each hierarchy has a separate cache status visible in the property sheet view of the Hierarchy component.

As with the implied tag index, once the hierarchy cache is built, it does not change. Subsequent changes to the station, that is, editing a hierarchy definition or editing permissions, are not reflected in the cache until it is recreated. For example, if you take away permissions that let a particular user see some part of the hierarchy, it will continue to be visible to that user if the cache is not recreated. A best practice is that the user making such changes to the station also be responsible for clearing the existing cache and creating a new cache.

Unlike the implied tag index which it is built “on-the-fly” as tag queries are executed, the hierarchy cache is built at the time you initiate it. The job is launched and runs in a separate thread, and when the job completes a popup window displays a message confirming that hierarchy caching completed successfully. Note that you can expand a hierarchy while its caching job is running.

Figure 1. Cache information available in Remote Spy Page view

Cache Info	
cached element count	1772
estimated cache size [MB]	5.72796

The **Remote Spy Page** view for each hierarchy shows how many elements are cached and the size of the cached hierarchy. The cached hierarchy is stored in heap memory and is not persisted with the station. It must be recreated each time the station starts. The Cache On Station Started property can be set so recreating the cache when the station starts is automatic.

- [System properties for caching](#)
There are a couple of system properties that you can set which will inhibit the automatic caching functions. Useful for managing hierarchy caching on a platform with limited heap space.
- [Other caching mechanisms](#)
One thing to be aware of is that clients, such as Workbench or a web browser, have their own caching mechanism that is separate from hierarchy caching.

Parent topic: [Hierarchy reference](#)

Related reference

[Hierarchy component](#)

[System properties for caching](#)

System properties for caching

There are a couple of system properties that you can set which will inhibit the automatic caching functions. Useful for managing hierarchy caching on a platform with limited heap space.

- **`niagara.hierarchy.caching.disableOnStationStarted`**

In a situation where it appears that too much memory is being used (resulting in a station shutdown/crash), there is a system property that you can set (`niagara.hierarchy.caching.disableOnStationStarted`) for ignoring the Cache On Station Started property. The cache-related **Actions** will continue to work even if this system property is set. When this system property is set to “true” hierarchies will not be cached automatically when the station starts even if the Cache On Station Started property is “true”. The intention of this system property is to provide a method of preventing a reboot loop.

- **`niagara.hierarchy.caching.disabled`**

When set to “true”, this system property disables caching of any hierarchies. You can set this property. Additionally, it will be set automatically by the nre launcher for legacy platforms that do not have much heap space, such as the JACE-3 and JACE-6.

The **Create Cache** action will still be present on hierarchies, but, when you try to invoke the action, an error window appears with the following message:

“Hierarchy caching is disabled because the system property `niagara.hierarchy.caching.disabled` is set to true. This may be because the platform does not have sufficient station heap space.”

This new system property is in addition to `niagara.hierarchy.caching.disableOnStarted`, which only affects whether caches for hierarchies will actually be built on station startup when with the `CacheOnStationStarted` property set to true. If the `niagara.hierarchy.caching.disabled` property is set, then the `niagara.hierarchy.caching.disableOnStationStarted` property is disregarded and hierarchy caches are not built on station startup or when requested by the **Create Cache** action.

Parent topic: [Caching hierarchies](#)

Related concepts

[Caching hierarchies](#)

Other caching mechanisms

One thing to be aware of is that clients, such as Workbench or a web browser, have their own caching mechanism that is separate from hierarchy caching.

For example, if you were to expand a hierarchy in the web browser, the browser saves those objects in the web page. If you then change something that affects the contents of the hierarchy (hierarchy definition, station configuration, user permissions, etc.) you will not see this change in the browser but will continue to see what is saved to the web page. So you must refresh the web page in order to see those changes. The same is true in Workbench, you must refresh the tree node of the hierarchy space (individual hierarchies cannot be refreshed separately).

Parent topic: [Caching hierarchies](#)

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 Entity, 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. Marker tags should not be used in a GroupLevelDef.
 - A **ListLevelDef** defines a node based on one or more **NamedGroupDefs** (named group definitions). Each NamedGroupDef has a query in which both marker and value tags can be used. ListLevelDefs require one or more NamedGroupDefs within them.

In order to view the actual data, you must add devices and child elements underneath a group or list level definition. To do this, use the Entity level definitions (either QueryLevelDef or RelationLevelDef) to 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 tree 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 based on an Entity level definition typically is associated to a BComponent within the scope of the NEQL query, which is typically a station.

- **[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 or ListLevelDef components.
- **[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. Typically, the relationship is a child relationship (n:child).
- **[GroupLevelDef component](#)**
Obtained from the **hierarchy** palette, this component sets up a group to contain the results of one or more QueryLevelDefs that follow.
- **[ListLevelDef component](#)**
Obtained from the **hierarchy** palette, this component sets up groups based on NEQL queries contained in NamedLevelDefs. A ListLevelDef contain as child components one or more NamedGroupDefs and must be followed by at least one QueryLevelDef.
- **[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) within the ListLevelDef.

Parent topic: [Hierarchy reference](#)

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 or ListLevelDef components.

Property Sheet

Campus_QueryLevel (Query Level Def)

Query Context >> ⌚

Query

Include Grouping Queries ☒ true

Sort Ascending

Property	Value	Description
Query Context	Config Facets window	<p>Sets up the current location's context as a facet.</p> <p>The current location is one item that could be placed in the query context. 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.</p> <p>The query context can hold anything to be used in LevelDef queries. It is comparable to a hierarchy scoped variable. You could create multiple copies of such a hierarchy definition and then customize each copy with the query context of the specific hierarchy.</p>
Query	NEQL query	<p>Defines a NEQL query. You may use all the NEQL operators: and, or, etc.</p> <p>The Context expression feature of NEQL can also be included to access values in the query context. Facet keys (found in the query context) surrounded by braces will be replaced by the value of the facet. For example, in the query <code>hs:siteRef = {siteId}, "{siteId}"</code> will be replaced by the value of the siteId facet in the query context.</p>
Include Grouping Queries	true (default), false	<p>Configures the prepending of Group Level queries to the current NEQL query.</p> <p>true prepends preceding GroupLevelDef queries in the hierarchy to the current NEQL query.</p> <p>false prevents preceding GroupLevelDef queries from prepending to the current NEQL query. This results in LevelElems for all query results being appended to the grouping LevelElems instead of just those query results that also match preceding grouping values in the hierarchy.</p>
Sort	None, Ascending (default), Descending	Determines the order in which results display.

Parent topic: [About level definitions](#)



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. Typically, the relationship is a child relationship (n:child).

RelationLevelDef properties

Property Sheet


RelationLevelDef (Relation Level Def)


Query Context  

Inbound Relation Ids

Outbound Relation Ids

Filter Expression

Repeat Relation ☒ false 

Sort 

The Inbound Relation Id and Outbound Relation Id properties on a relation level definition replace the “RelationId” and “Inbound” properties present in prior releases. Note that during station start-up, if the existing bog file contains a RelationId value, this value is copied to either an Inbound Relation Id or an Outbound Relation Id based on the inbound flag value.

Additionally, there is added support for multiple relation IDs on a single relation level definition. You may add multiple Inbound or Outbound Relation Ids separated by a comma. The results for any of these relation IDs display in the hierarchy. Note that only a single LevelElem will be added for each distinct Entity related by the specified Ids, even if the Entity is related multiple ways (any duplicate relation levels will be removed).

Property	Value	Description
Query Context	Config Facets window	<p>Sets up the current location’s context as a facet.</p> <p>The current location is one item that could be placed in the query context. 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.</p> <p>The query context can hold anything to be used in LevelDef queries. It is comparable to a hierarchy scoped variable. You could create multiple copies of such a hierarchy definition and then customize each copy with the query context of the specific hierarchy.</p>
Inbound Relation Ids	Comma separated list of relationIds	Queries for inbound relations with the specified Ids. Any duplicate relation levels will be removed.
Outbound Relation Ids	Comma separated list of relationIds	Queries for outbound relations with the specified Ids. Any duplicate relation levels will be removed.
Repeat Relation	true, false (default)	<p>Configures how the system processes relations.</p> <p>true, causes the system to traverse the</p>

Property	Value	Description
		specified relations until there are no further results. For example, if the relation ID is n:child, the relation level def repeatedly evaluates until it reaches an Entity that has no children. false evaluates the relations level def only once.
Sort	None, Ascending (default), Descending	Determines the order in which results display.

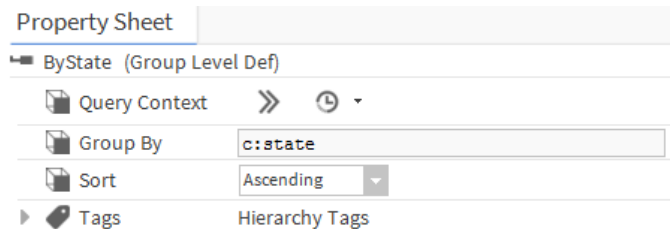
Parent topic: [About level definitions](#)

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.

Figure 1.



Property	Value	Description
Query Context	Config Facets window	Sets up the current location's context as a facet. The current location is one item that could be placed in the query context. 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. The query context can hold anything to be used in LevelDef queries. It is comparable to a hierarchy scoped variable. You could create multiple copies of such a hierarchy definition and then customize each copy with the query context of the specific hierarchy.
Group By	text string	Defines a single tag to use for grouping query results at the current level. For example, n:geoState groups all resulting data by the "n:geoState" tag on the Entity: AZ, CA, VA, etc.
Sort	None, Ascending (default), Descending	Determines the order in which results

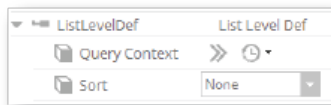
Property	Value	Description
		display.

Parent topic: [About level definitions](#)

ListLevelDef component

Obtained from the hierarchy palette, this component sets up groups based on NEQL queries contained in NamedLevelDefs. A ListLevelDef contains as child components one or more NamedGroupDefs and must be followed by at least one QueryLevelDef.

The NEQL queries in the child NamedGroupDefs can contain marker tags and value tags.



Type	Value	Description
Query	NEQL query	Defines a NEQL query. You may use all the NEQL operators: and, or, etc. The Context expression feature of NEQL can also be included to access values in the query context. Facet keys (found in the query context) surrounded by braces will be replaced by the value of the facet. For example, in the query <code>hs:siteRef = {siteId}, "{siteId}"</code> will be replaced by the value of the siteId facet in the query context.
Sort	None, Ascending (default), Descending	Determines the order in which results display.

Parent topic: [About level definitions](#)

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) within the ListLevelDef.



Property	Value	Description
Query	NEQL query	Defines a NEQL query. You may use all the NEQL operators: and, or, etc. The Context expression feature of NEQL can also be included to access values in the query context. Facet keys (found in the

Property	Value	Description
		query context) surrounded by braces will be replaced by the value of the facet. For example, in the query <code>hs:siteRef = {siteId}, "{siteId}"</code> will be replaced by the value of the <code>siteId</code> facet in the query context.

Parent topic: [About level definitions](#)

Context parameters

Context parameters on a **LevelDef** can be used to pass context sensitive information to subsequent (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."

Note: When adding facets to a Query Context, the facet values for tag names must be Strings. When comparing a tag value to values from the Query Context, make sure the resulting types are the same. In the "Query context example," 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).

Parent topic: [Hierarchy reference](#)

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.

The value in the Scope ORD subproperty limits the query to a specific location within the station, such as `station:|slot:/Model/Westerre/W1`. An alternative to limiting the scope is to use additional **LevelDefs** in a hierarchy definition.

Note: The "nspc" and "sys" ORD schemes support querying for indexed entities in the System Database. These ORD schemes allow you to query the entire SystemDb, as well as to execute scoped queries against the SystemDb. For details see the procedure, "Accessing hierarchies scoped against the SystemDb" in Chapter 1 of this guide. For complete details on the System Database and System Indexing, see the *Niagara System Database and System Indexing Guide*.

Parent topic: [Hierarchy reference](#)

Related tasks

[Accessing hierarchies scoped against the SystemDb](#)

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 *Niagara Station Security Guide*

Parent topic: [Hierarchy reference](#)

Related tasks

[Assigning a hierarchy to a role](#)

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.
- [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.
- [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.
- [NEQL query examples](#)
The Niagara Entity Query Language (NEQL) provides a mechanism for querying tagged entities in Niagara applications. This topic provides grammar and syntax examples to help you construct these queries.

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 1. All AHU Points hierarchy definition setup (left) and resulting hierarchy Nav tree (right)

The figure shows two side-by-side screenshots from a software interface. The left screenshot displays the 'Hierarchy' configuration window for 'AHU Points'. It includes sections for 'Query Context' (Status: {ok}, Fault Cause, Scope: Hierarchy Scope Container, Tags: Hierarchy Tags, Device: Query Level Def: n:device and hs:ahu), 'Query' (Query Context, Query: n:device and hs:ahu, Include Grouping Queries: true, Sort: Ascending), and 'Points' (Query Context, Inbound Relation Ids, Outbound Relation Ids: n:childPoint, Filter Expression, Repeat Relation: true, Sort: Ascending). The right screenshot shows the resulting 'Hierarchy' Nav tree. It has a root node 'All AHU Points' which branches into three AHU units: 'AHU-1', 'AHU-2', and 'AHU-3'. Each AHU unit has three child nodes: 'DamperPosition', 'ReturnTemp', and 'SupplyTemp'. Red icons (a red square with a white 'N') are visible next to the AHU units and their children, indicating they are newly created or modified.

Note: Red icons displaying in a hierarchy definition indicate the need to save due to the hierarchy being newly created or modified in some way.

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.
Tags	Hierarchy Tags	Tags added to LevelElems that are not BComponents	None in this example. When the navigation hierarchy is built, tags are 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)	Outbound Relation Ids: n:childPoint	Implied tag on each point.	Indicates the outbound relation direction setup on the object (target) component of a relation. Returns all entities with an outbound "n:childPoint" relation from each result returned from the previous QueryLevelDef (named Device). The n:childPoint relation is implied between devices and points underneath the device.
	Inbound Relation Ids:	Implied tag on each point.	Not used in this example.
	Filter Expression:	Set up here in the hierarchy definition.	Not used in this example.
	Repeat Relation: true	Set up here in the hierarchy definition.	When true, this cause entities traversed by the specified outbound "n:childPoint" relation to be added to subsequent levels of the hierarchy, if they existed. n:childPoint is a relation implied on devices so it is unlikely that there would be additional levels unless the relation was added manually.

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

There is added support for multiple relation IDs on a single relation level definition. You can enter more than one comma separated tagID in the Inbound Relation IDs or Outbound Relation IDs properties. The results for any of these relation IDs display in the hierarchy.

Parent topic: [Examples](#)

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.

Building setup

The Site component (from the Haystack palette) was used to represent each building in the Nav tree. The Site component is useful for this purpose since it is pre-tagged with a number of individual site-related tags which can be used to create alternate hierarchies. **Building1** and **Building2** are arbitrary names assigned to the components. Each building is assigned the following tags:

- The Haystack tag, **hs:site**, identifies the object as a physical site.
- The implied Haystack tag **hs:id=h:{identifier}** provides unique identification for each building via the alpha-numeric **{identifier}** notation. For **Building1** **hs:id** is **h:9f2** , and for **Building2** it is **h:9f3**.

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 assigned the following tags and relation:

- The Haystack tag, **hs:equip**, identifies the object as a physical device.
- The implied Haystack tag, **hs:id=h:{deviceID}**, provides unique identification for each device via the alpha-numeric **{deviceID}** identifier. In the example, **ModbusVAV1**'s **hs:id** is **h:83d**, and **ModbusVAV2**'s **hs:id** is **h:841**.
- The Haystack relation, **hs:siteRef** associates each device with the building in which it is located.

Points setup

VAV performance is monitored by two points (**ModbusClientPointDeviceExt**):

- **SupplyTemp**
- **ReturnTemp**

Each point is assigned the following Haystack tag and relation:

- The Haystack tag, **hs:point**, identifies the object as a point.
- The Haystack relation, **hs:equipRef**, associates each point with the equipment to which it belongs

Hierarchy definition

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

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 : hs:site	Name, value tag on each building component hs:site = h:9f2 or hs:site = h:9f3	Returns components tagged with hs:site.
equipment QueryLevelDef	Query Context: equipId=hs:id	hs:Id={identifier}, where {identifier} is a unique string that identifies each device. The equipment Id for ModbusVAV1 is h:83d, and that for ModbusVAV2 is h:841.	Establishes the context for the points level by storing the value of each device's hs:id in the config facet equipId.
	Query: hs:equip and hs:siteRef-> hs:id = {siteId}	Set up by the site query context in the previous level definition.	Query returns all devices assigned the hs:equip implied marker tag, and the NEQL traverse operator “->” instructs the query to “traverse” the siteRef relationship to determine if that Source Ord value matches the unique siteId context setup in the previous QueryLevelDef, the site level definition.
points QueryLevelDef	Query Context	none	
	Query: hs:point and hs:equipRef-> hs:id = {equipId}	Set up by the equipment query context in previous level definition.	The implied marker tag on each point, hs:point, returns all data for all points in the system, and the NEQL traverse operator “->” instructs the query to “traverse” the equipRef relationship to determine if that Source Ord value matches the unique equipId context setup in the previous QueryLevelDef, the equipment level definition.

Context processing

In this example:

- The siteRef=h:9f2 outbound relation on ModbusVAV1 matches the site query context value, siteId=h:9f2, for Building 1 causing ModbusVAV1 to appear under Building 1 in the resulting hierarchy.
- The siteRef=h:9f3 outbound relation on ModbusVAV2 matches the site query context value,

siteId=h:9f3, for Building 2 causing ModbusVAV2 to appear under Building 2 in the resulting hierarchy.

Continuing with the processing of points:

- The equipRef=h:83d relation on two points labeled SupplyTemp and ReturnTemp matches the equipment query context value, equipId=h:83d, for ModbusVAV1 causing these points to appear under ModbusVAV1.
- The equipRef=h:841 relation on two additional points SupplyTemp and ReturnTemp matches the equipment query context value, equipId=h:841, for ModbusVAV2 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 (BOords in this example).

Parent topic: [Examples](#)

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.

Figure 1. Multi-user hierarchies on a station

- **Facility Manager hierarchy** - To monitor usage for the entire building as well as by floor, the facility manager prefers that the data to be displayed by floor. Each floor expands in a similar fashion to Floor 1 in the screen capture. Westerre II follows the same structure.

Note: This hierarchy definition is the one that is fully explained in this example topic.

- **Systems Integrator hierarchy** - The systems integrator, whose main interest is monitoring device performance, prefers to view the same data by device.
- **Operations Center hierarchy** - Finally, the operations center monitors the data by tenant and individual office.

Definition for Facility Manager hierarchy

Now that you know the goal, the hierarchy structure for the Facility Manager is as follows.

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	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	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	Sets equipID to the value of

		{identifier} is a unique string that identifies the office) is an implied tag on each office.	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. 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).

Parent topic: [Examples](#)

NEQL query examples

The Niagara Entity Query Language (NEQL) provides a mechanism for querying tagged entities in Niagara applications. This topic provides grammar and syntax examples to help you construct these queries.

Note that NEQL only queries for tags. NEQL supports traversing defined entity relationships as well as parameterized queries and allows you to use the same syntax for queries in hierarchy level definitions as is used in **Search** queries.

In addition to using NEQL queries in a hierarchy definition or search, you might use these queries for testing purposes, say to find devices or points in a station. For example, to find all devices in a station enter the query: `neql:n:device`. Or using the absolute ORD form in Workbench, press **CTRL+L** to open the **ORD** dialog and enter the following:

```
ip:<host>|foxs:|station:|slot:/|neql:n:device
```

Note:

In Niagara, the “sys:” ORD scheme can be used to redirect NEQL queries to resolve against the System Database. For example, the following query searches for all devices known to the System Database:

```
ip:<host>|foxs:|sys:|neql:n:device
```

In Niagara, NEQL queries can be resolved over FOX. Additionally, NEQL results support tables so the results can be displayed in collection tables, reports, Px pages, etc. NEQL can be used in the following cases:

- used on a table on a Px page
- resolved from the ORD Field Editor
- resolved from the browser/path bar when typed in an Hx Profile
- used to generate reports

Finally, NEQL and BQL can be used together (although not supported in **Search**). You can append a BQL query to the end of a NEQL query for additional processing. Do this in any of the cases listed above. The following example shows a BQL `select` query appended to a NEQL query.

```
ip:<host>|foxs:|station:|slot:|neql:n:device|bql:select toDisplayPathString, status
```

The above query first finds all devices using the NEQL statement, and then the BQL statement processes against the devices returned from the NEQL query, displaying a two-column table with the display path of the devices shown in one column and the status of the devices shown in the other.

CAUTION: Appending a BQL query to the end of a NEQL query may take some processing time. You could experience processing delays when using such queries.

Query examples

The following examples are designed to help you construct queries.

To query for	Syntax example	Returns result
point tag	<code>n:point</code>	Any entity with the point tag (in the "n" namespace)
name tag = "foo"	<code>n:name = "foo"</code>	Any entity with the "foo" name tag (in the "n" namespace)
type tag = "baja:Folder"	<code>n:type = "baja:Folder"</code>	Any entity with the "baja:Folder" type tag (in the "n" namespace)
points that are NumericWritables with <code>hs:coolingCapacity > 4.03</code>	<code>type = "control:NumericWritable" and hs:coolingCapacity > 4.03</code>	Any entity with the "control:NumericWritable" type tag (in the "n" namespace) and the <code>coolingCapacity</code> tag (in the "hs" namespace) with a value greater than 4.03
everything	<code>true</code>	All entities
entities with names containing "Switch"	<code>n:name like ".*Switch.*"</code>	Any entity with a name that contains "Switch" case-sensitive
entities with <code>geoCity</code> tag with value not equal to Atlanta	<code>n:geoCity != "Atlanta"</code>	Any entity with the <code>geoCity</code> tag (in the n namespace) with a value that is not Atlanta
where <code>n:pxView</code> is a relation	<code>n:pxView->n:type</code>	Any entity with a <code>pxView</code> relation where the endpoint is a niagara type. This is all the entities that have px views.
entities with <code>t:foo</code> but not <code>t:herp</code>	<code>t:foo and not t:herp</code>	Any entity with the <code>foo</code> tag (in the t namespace) that does not also have the <code>herp</code> tag (in the t namespace).

To query for	Syntax example	Returns result
points that were built earlier than 2015 or whose primary function is backup	hs:yearBuilt < 2015 or hs:primaryFunction = "backup"	Any entity with either the yearBuilt tag (in the hs namespace) with value less than 2015 or the primaryFunction tag (in the hs namespace) with value "backup"
child entities of entities with the floor tag = 2	n:parent->hs:floor = 2	Any entity whose parent has the floor tag (in the hs namespace) with value 2

Note: Do not create a namespace for a tag dictionary if that namespace is already used for an existing ORD scheme (NEQL or BQL). Since the search service allows for different types of searches (for example, neql and bql), user must validate the search text and make sure they are not trying to enter an invalid ORD. Precede the query search with "neql:" to specify that it should run a neql query explicitly, for example: "neql:h:test".

NEQL grammar

The following list is the subset of the NEQL grammar that you can use to build NEQL queries for the “Query” portion of Query Level Defs and the “Filter Expression” portion of Relation Level Defs in Hierarchies. For a more comprehensive overview of NEQL and the complete grammar and examples, refer to NEQL documentation in the *Niagara Developer Guide*.

- <statement> := <full select> | <filter select> | <traverse>
- <full select> := select <tag list> where <predicate>
- <filter select> := <predicate>
- <traverse> := traverse <relation> (where <predicate>)
- <tag list> := <tag> (, <tag>)*
- <tag> := (<namespace>:)<key>
- <relation> := (<namespace>:)<key><direction>
- <namespace> := <word>
- <key> := <word>
- <direction> := -> | <-
- <predicate> := <condOr>
- <condOr> := <condAnd> (or <condAnd>)*
- <condAnd> := <term> (and <term>)*
- <term> := <cmp> | <tagPath> | <not>
- <cmp> := <comparable> <cmpOp> <comparable> | <like>
- <like> := <tagPath> like <regex>
- <cmpOp> := = | != | < | <= | > | >=
- <comparable> := <val> | <tagPath>
- <val> := <number> | <bool> | <str>
- <tagPath> := (<relation>)*<tag>
- <not> := not <negatable> | !<negatable>
- <negatable> := (<predicate>) | <tag> // note: parens around <predicate> signify actual paren characters, NOT optional syntax
- <number> := <int> | <double>
- <bool> := true | false
- <str> := "<chars>"
- <typeSpec> := <moduleName>:<typeName>
- <moduleName> := <word>
- <typeName> := <word>
- <word> := <chars>
- <regex> := "<chars>" // note: some regex edge cases not supported

Parent topic: [Examples](#)