Whitepaper
Public

Object Renaming in Cisco ACI

© 2023 Cisco and/or its affiliates. All rights reserved.

Contents

Introduction 3
Cisco ACI Concepts: A Refresher 3
Object Relations 4
The cost of hypothetical renaming 8
Why this structure? 8
Alternatives to renaming 9
Aliases **9**
Annotations **10**
The case of interface descriptions 10
Conclusion 12

© 2023 Cisco and/or its affiliates. All rights reserved.

Introduction

Cisco Application Centric Infrastructure (ACI) does not offer the option to rename objects after they have been created. Therefore, choosing a sensible naming convention for objects is of paramount importance to ensure smooth operation of ACI fabrics. Cisco provides guidelines to that effect[1] in the form of best practices. However, not everybody adheres to those best practices and as such, it is not uncommon for Cisco's technical staff to meet the dreadful " How can I rename objects in ACI?" question. This document explains why renaming objects is not possible and presents alternative options to alleviate an ill-fitted naming convention.

Cisco ACI Concepts: A Refresher

Under the covers, Cisco ACI relies on an object-oriented design based on an object model. Every aspect of ACI is stored as an object in a configuration database. Configuration items, faults, statistics, events, and logs are all objects defined in the object model. Each object belongs to a specific class, which broadly identifies the category (such as security and routing) to which it belongs. For example when you configure a tenant in ACI, you actually instruct the system to instantiate an object of class *fvTenant* (*fv* stands for fabric virtualization) and store that object somewhere in the configuration database. After an object is instantiated (a fancy word to indicate an object of a given type has been created), it receives a distinguished name (dn). For example when you create tenant *Foo*, its dn is *uni/tn-Foo*. That is valid for every ACI fabric in the world. The dn of tenant *Foo* is always *uni/tn-Foo*. You can determine the dn of most objects by right-clicking on an object in the Cisco Application Policy Infrastructure Controller (APIC) user interface and choosing **Share**.

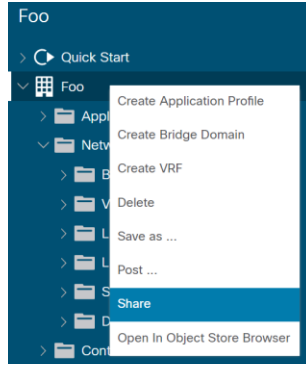


Figure 1 **The " Share" option in the user interface**

Share

https://10.48.168.3/#dn=uni/tn-Foo

Figure 2 **The dn of tenant Foo**

A key concept to understand is that each and every dn is unique in the entire configuration database of ACI. Duplicates never exist. Furthermore, looking at any dn reveals its place in that configuration database.

[1] https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/kb/b-Cisco-ACI-Naming-and-Numbering.html

© 2023 Cisco and/or its affiliates. All rights reserved.

Looking at tenant *Foo*, we observe its dn is *uni/tn-Foo*. The *uni* prefix indicates a higher-level object. That higher-level object is the parent of *tn-Foo*, and *tn-Foo* is therefore a child of *uni*. If you create VRF instance *main* under tenant *Foo*, the *dn* of VRF instance *main* is *uni/tn-Foo/ctx-main* (ctx stands for context). The VRF instance is a child of *tn-Foo* which is itself a child of *uni*. The concept of parent-child entities is very common in database systems design. Because each dn clearly indicates the position of a given object in the entire configuration database, readers familiar with databases have probably realized ACI stores objects using a tree, as shown in Figure 3.

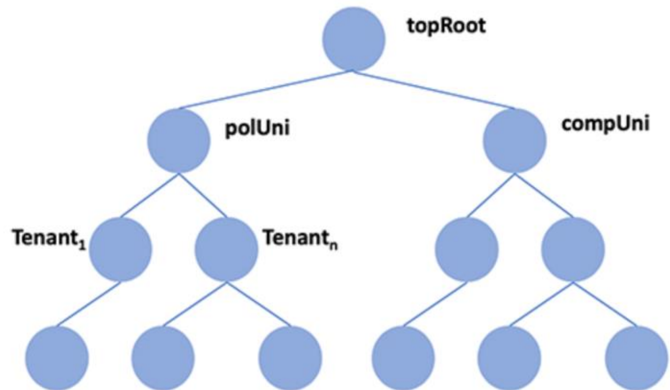


Figure 3 Cisco ACI uses a tree structure to store objects

Different techniques exist to store objects configured as a tree, each offering pros and cons. The next sections explore further relations between objects and discusses the implications of the ACI architectural model.

Object Relations

Continuing our journey into the ACI object model, let's now create a bridge domain called *primary* and attach it to VRF instance *main*. Intuitively, you might think the dn of that bridge domain is going to be *uni/tn-Foo/ctx-main/BD-primary*. Unfortunately, that is not the case. The dn of the bridge domain is actually *uni/tn-Foo/BD-primary*! However, it is linked to VRF instance *main*, as show in Figure 4.

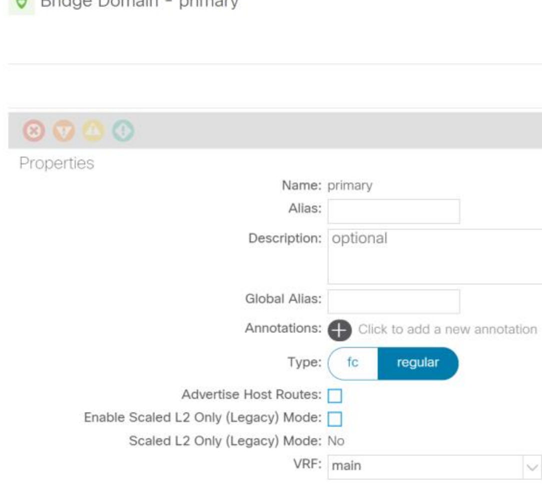


Figure 4 Bridge domain *primary* is related to VRF instance *main*

If the dn of the bridge domain does not indicate the relation to VRF instance *main*, then what does? As seen with tenant *Foo* and VRF instance *main*, children can themselves have children. But, they can also point to other objects to indicate a relation to that object! That is precisely how bridge domain *primary* expresses its relation to VRF instance *main*. If we look at the object model explorer (right-click on an object in the GUI and choose **Open in Object Store Browser**), we can see bridge domain *primary* has children of its own.

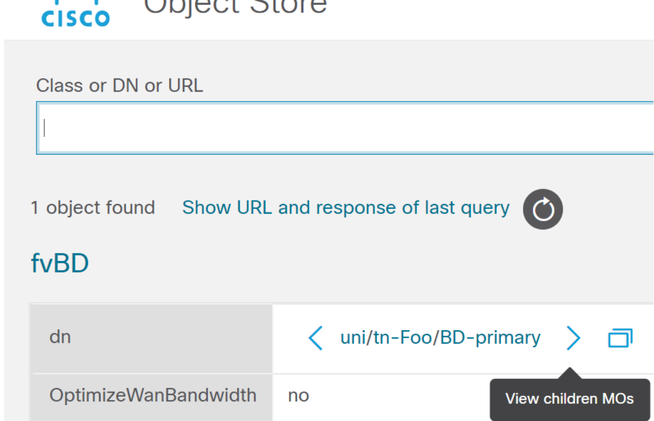
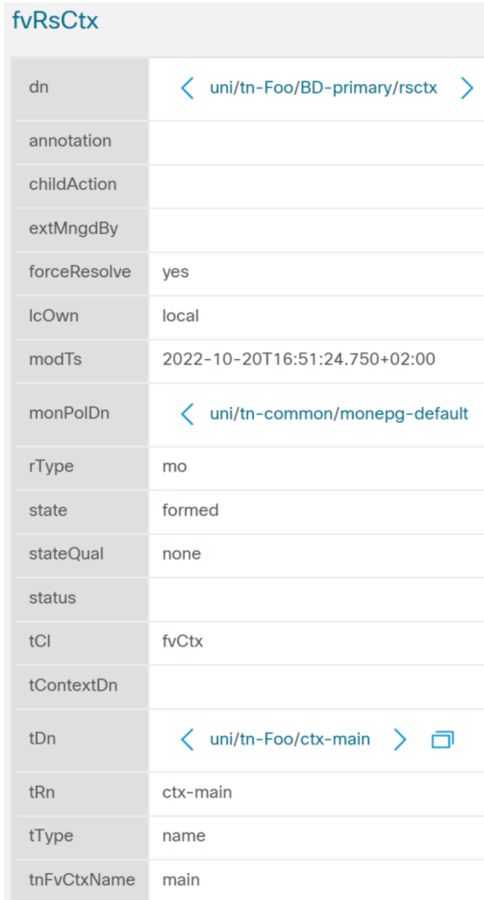


Figure 5 Viewing the bridge domain in the Object Store browser

Clicking **View children MOs** (MOs stands for managed objects, meaning any object stored in the configuration database) reveals one interesting child called *fvRsCtx*. *Rs* here indicates " relation source" and *Ctx* as seen previously refers to a VRF instance. Here is what the entire child looks like:

© 2023 Cisco and/or its affiliates. All rights reserved.



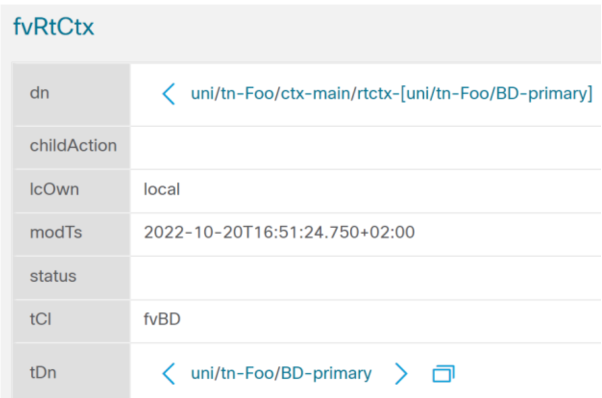
fvRsCtx

dn	uni/tn-Foo/BD-primary/rsctx
annotation	
childAction	
extMngdBy	
forceResolve	yes
lcOwn	local
modTs	2022-10-20T16:51:24.750+02:00
monPolDn	uni/tn-common/monepg-default
rType	mo
state	formed
stateQual	none
status	
tCl	fvCtx
tContextDn	
tDn	uni/tn-Foo/ctx-main
tRn	ctx-main
tType	name
tnFvCtxName	main

Figure 6 Child *fvRsCtx* details

Bridge domain *primary* points to VRF instance *main* through a child object! That child object has itself a class and a dn of course. Every managed object possesses a series of attributes as shown in Figure 6: *dn, annotation, childAction*, and so on are all attributes of a given managed object. In this particular *fvRsCtx* example, attribute *tDn* is of particular interest. It is the *target dn* attribute. It points to the dn of the object this bridge domain wants to establish a relation with. Attribute *tCl* represents the class of the target relation, a *fvCtx* object in this case, which is therefore a VRF instance.

It is interesting to know that a target object always knows which objects are pointing to it! If we explore VRF instance *main* in the Object Store Browser, we notice a child called *fvRtCtx* where *Rt* stands for relation target this time.



fvRtCtx

dn	uni/tn-Foo/ctx-main/rtctx-[uni/tn-Foo/BD-primary]
childAction	
lcOwn	local
modTs	2022-10-20T16:51:24.750+02:00
status	
tCl	fvBD
tDn	uni/tn-Foo/BD-primary

© 2023 Cisco and/or its affiliates. All rights reserved.

Figure 7 Child *fvRtCtx* details

Look at the *tDn* attribute: it points to bridge domain *primary*. We have essentially a linked list of objects, each aware of relation targets and relation sources. Relations are all established using distinguished names. Every single possible relation to and from a given object is documented in detail in the object model reference guide[2]. Some objects may contain dozens and dozens of relation targets and/or sources. Figure 8 represents a mere overview of the *fvTenant* object.

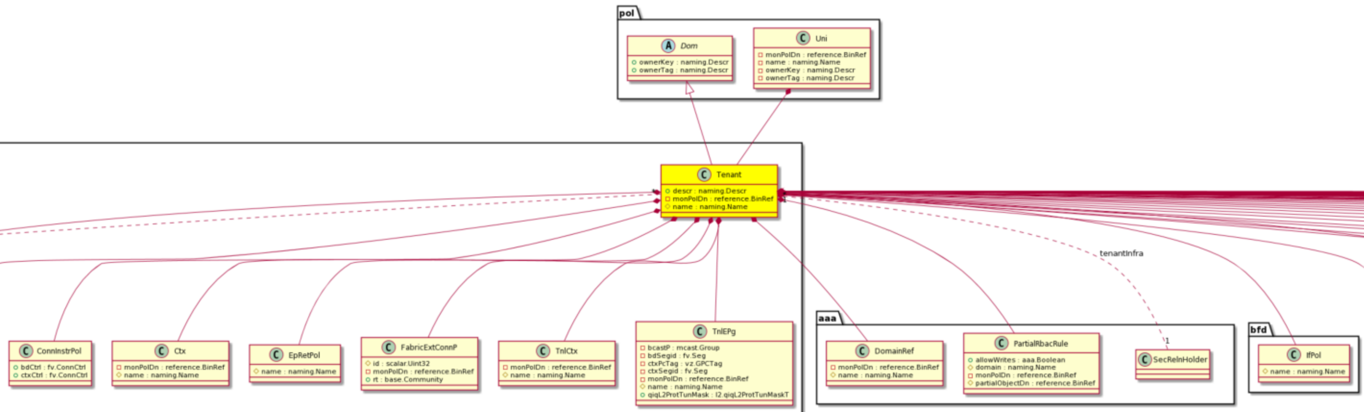


Figure 8 Partial representation of *fvTenant* relations

The entire ACI object model contains tens of thousands of objects. If you are curious and want to explore the object model of your ACI fabric, download the PyACI project[3] and run the *rmetagen.py* utility, pointing it to the fabric of your choice. After a few minutes, you will have an entire representation of the object model in the *~/.aci-meta* directory. A quick Python script reveals the number of objects in an ACI fabric running software release 6.0(1) in a lab environment:

```
 cisco@CSCO-W-PF2P9ZV8  ~/.aci-meta  python3
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import json
>>> f=open('./aci-meta.6.0(1g).json')
>>> j=json.loads(f.read())
>>> type(j)
<class 'dict'>
>>> len(j['classes'].keys())
17080
>>>
```

Figure 9 Exploring the object model with PyACI

This ACI fabric's configuration contains 17,080 distinct classes of objects! When you consider that figure, it is easy to imagine there are thousands and thousands of relation targets and sources formed between managed objects in this fabric. Picking just one simple object (the default LLDP interface policy – *uni/infra/lldpIfP-default*), we can see it contains 25 " Rt" children, indicating 25 managed objects point to that interface policy:

---

[2] https://developer.cisco.com/site/apic-mim-ref-api/

[3] https://github.com/datacenter/pyaci

© 2023 Cisco and/or its affiliates. All rights reserved.

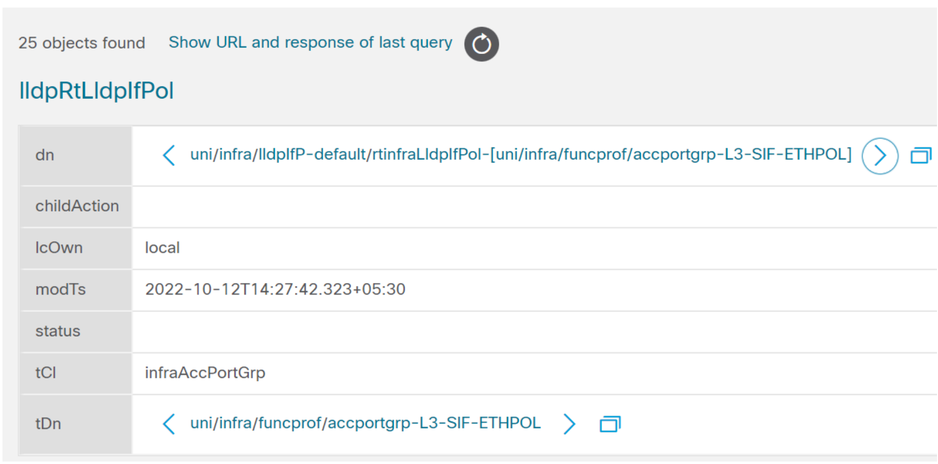


Figure 10 **Children of the default LLDP interface policy**

The cost of hypothetical renaming

Imagine that you could modify the name of that default LLDP interface policy. The immediate implication is that 25 objects must immediately be updated to point to the new *dn* resulting from this hypothetical renaming. Conversely, the LLDP interface policy itself now must update all its *Rt* children to reflect its new complete *dn*.

You probably recall that that the *dn* of every single managed object contains the name of the object itself. So, if you rename *lldpIfP-default* to *lldpIfP-somethingNew*, the *dn* of the policy reflects that change. Therefore, any managed object relying on the previous *dn* must update its corresponding attribute or attributes accordingly. Furthermore, while the renaming process walks through the configuration tree, no other changes to the LLDP interface policy can occur to ensure data consistency. This effectively means locking entire portions of the configuration tree for an unknown duration. Repeat this with the many potential simultaneous name changes and the result is an APIC busy primarily with patching relation sources, relation targets, and distinguished names. What happens in case of failure? Does APIC roll back all changes? How does it keep track of changes successfully executed versus those pending? Readers should also know that APIC is a distributed cluster replicating all information in three shards. This means renaming operations must also be replicated within the APIC cluster. This hypothetical renaming can turn out to be very computationally expensive!

Why this structure?

You might be wondering why Cisco ACI stores information this way. Why use relation targets and sources as children of managed objects? Why refer to distinguished names in object attributes? Why not generate a UUID per object and build a relational database using primary and foreign keys? One immediate benefit of this tree or graph structure is that looking at any given object directly reveals the other objects it either depends on, or the objects that depend on this object. Accessing a *managed object* using its *dn* is computationally inexpensive. It's similar to fetching an object using its primary key in a relational database. This is quite practical when rendering the configuration database in the User Interface for example. If you want to find out all bridge domains pointing to a VRF instance, there is no need to perform complex SQL joins with lookups in multiple tables. Likewise, determining how many bridge domains this VRF instance is the parent of is trivial: one simple lookup does the job. It also allows the APIC API to offer backend support for advanced queries such as " return all objects of class *fvTenant* and their children, only if the name of the bridge domain contains production" without requiring advanced SQL knowledge. Because ACI is built

© 2023 Cisco and/or its affiliates. All rights reserved.

around objects related to other objects, this tree or graph structure lends itself well to the job at hand. The model trades flexibility for performance. You cannot rename objects, but you get very rapid navigation through a potentially enormous configuration tree.

Alternatives to renaming

To err is human. Despite the best planning, everybody makes mistakes. Introducing a typo in the name of an object is just a keyboard stroke away. If the typo is not immediately caught and corrected, you risk being stuck with objects pointing to a name you did not mean to enter, thereby making operations of the fabric more difficult.

Cisco realizes these conditions and provides several ways to alleviate that burden:

1. Aliases
2. Annotations

Aliases

Aliases come in two forms:

1. Name Aliases
2. Global API Aliases

Name aliases, or simply aliases in the GUI (Graphical User Interface), are an attribute several managed objects possess. When configured, the GUI renders the values of the alias attribute in place of the object's real name. You can see several examples in Figure 11.

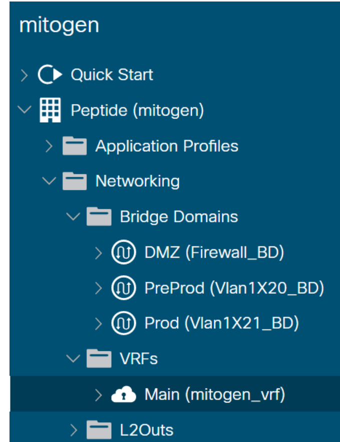


Figure 11 Using aliases in the GUI

The GUI places the original name of the managed object between parentheses and highlights the alias attribute. In Figure 11, the actual name of the tenant is *mitogen* while its alias is *Peptide*. Likewise for all bridge domains and the VRF instance. This is a very convenient way to correct a mis-named object. Now, this does not rename objects; it simply modifies an attribute of the object. The GUI is instructed to act upon seeing a non-empty alias attribute.

Even though most objects contain the alias attribute, not every managed object does. Also, the GUI does not render the alias attribute of every single managed object; it focuses primarily on tenant-related managed objects. Aliases need not be unique across the configuration though. They serve a purely cosmetic function.

© 2023 Cisco and/or its affiliates. All rights reserved.

Global aliases are only relevant in the context of API-driven operations. When configured, a global alias becomes a child (*tagAliasInst*) of the object it is attached to. The original managed object can then be accessed using its global alias. Suppose you create global alias *foo* for *uni/tn-Foo/ap-one/epg-web*. You can now perform API operations against EPG web using https://<apic>/api/alias/foo.json.

Annotations

Annotations (object class *tagAnnotation*) follow the industry-standard *key:value* pair metadata format. An arbitrary number of annotations can be attached to any managed objects in the configuration database. Annotations become children of the managed object they qualify. The GUI renders annotations for a certain number of managed objects as shown in Figure 12 with a tenant:

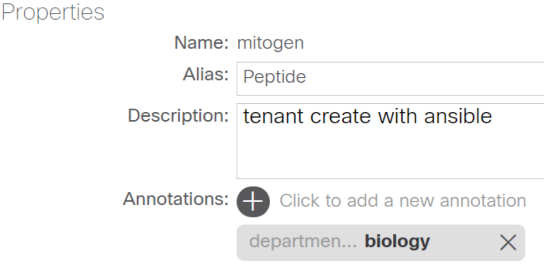


Figure 12 Annotating a tenant

The GUI lets you query annotations in a central location as shown in Figure 13.

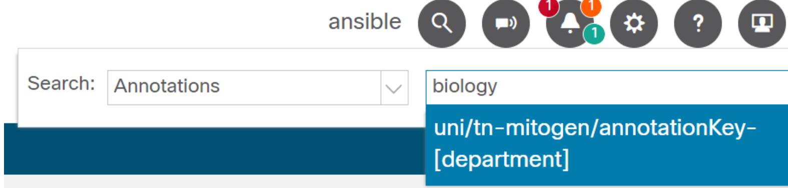


Figure 13 Searching for annotations

The case of interface descriptions

A common pain point brought forward by customers due to the inability to rename objects is how to pick the best interface naming convention. Customers are not sure which naming convention is going to work best for their environment and sometimes determine months into the deployment that another naming convention would have been more suitable. Most of the times, an ill-suited interface naming convention can be alleviated by using better interface descriptions. The easiest way to create a description for an interface in ACI is to go to the **Fabric > Inventory > Pod > Interfaces > Physical Interfaces** section of the GUI as shown in Figure 14.

© 2023 Cisco and/or its affiliates. All rights reserved.

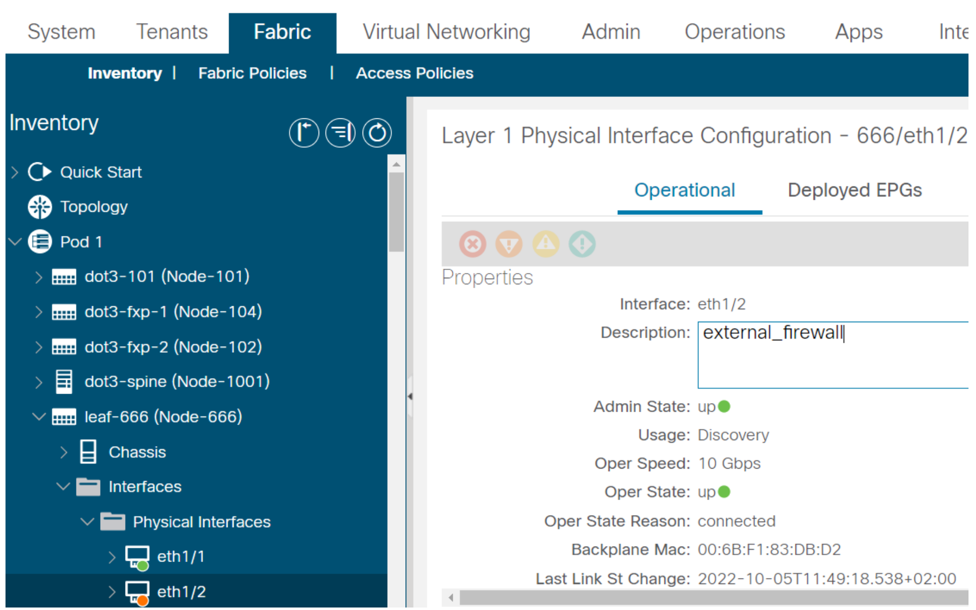


Figure 14 Adding an interface description in the GUI

Behind the scenes, the GUI configures an interface override policy, which you can find here:

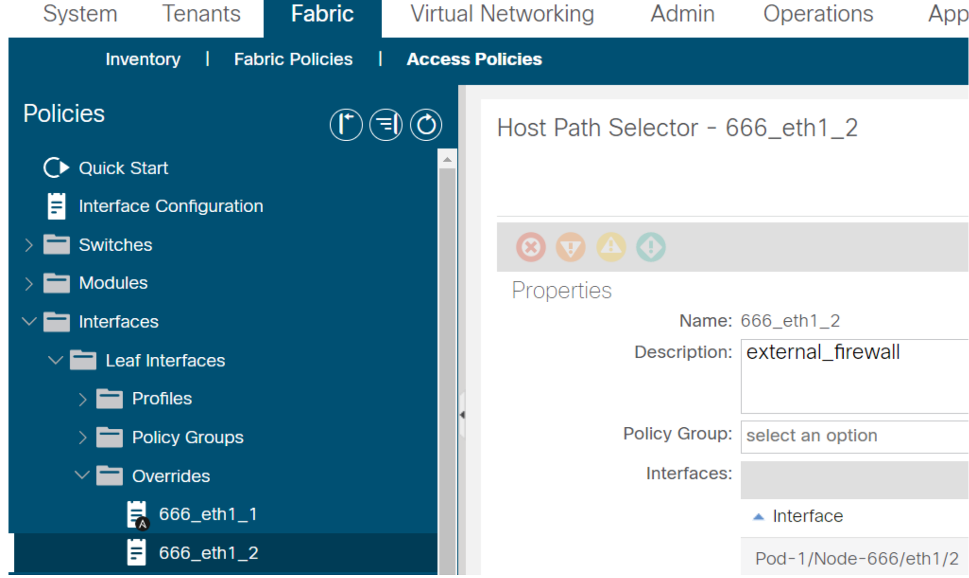


Figure 15 Interface override policy that is automatically created

The interface description is now reflected in the GUI and on the command line as shown in Figure 16.

```
apic1-dot3# fabric 666 show interface eth1/2 description
----------------------------------------------------------------
 Node 666 (leaf-666)
----------------------------------------------------------------
----------------------------------------------------------------
 Port          Type    Speed     Description
----------------------------------------------------------------
 Eth1/2        eth     inherit   external_firewall

apic1-dot3# _
```

Figure 16 Interface description on the command line

Programmatic access to interface descriptions is made simple with Ansible as shown in Figure 17.

```yaml
---
- hosts: 10.48.168.3
  connection: local
  gather_facts: no
  vars:
    aci_creds: &aci_login
      hostname: '{{ inventory_hostname }}'
      username: ansible
      validate_certs: no
      use_proxy: no
      use_ssl: yes
      private_key: ansible.key
  tasks:
  - name: Set Interface Description
    cisco.aci.aci_interface_description:
      <<: *aci_login
      pod_id: 1
      node_id: 666
      node_type: leaf
      interface: 1/1
      description: internet_router
      state: present
    delegate_to: localhost
```

Figure 17 Assigning interface descriptions with Ansible

Conclusion

Considering how and why Cisco ACI stores runtime and configuration data, renaming objects is not implemented due to the enormous computational cost and configuration lock nuisances that would ensue. Renaming an object is only possible if the original object is destroyed and replaced by a new one. Adopting a sensible naming convention is of critical importance. Should you make small mistakes despite solid planning efforts, aliases and annotation can provide relief. If your use case revolves around naming of interfaces, configuring suitable interface descriptions is just a simple task in the user interface or easily done through programmatic access.

Printed in USA

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam
The Netherlands

Cisco has more than 200 offices worldwide. Address, phone numbers, and fax numbers are listed on the Cisco Website at https://www.cisco.com/go/offices.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: https://www.cisco.com/go/trademarks. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

© 2023 Cisco and/or its affiliates. All rights reserved.