Department of Defense Joint Communications Unit (JCU)

This story was submitted by ***Derek Howard****, Senior Technical Advisor at the JCU.*

I was asked to share the story of the Joint Communications Unit's (JCU) automation journey, and I am happy to do that. I really enjoy telling the story and hope that we are able to help other elements in the DoD accomplish some of the things that we have within JCU.

First, I would like to reference the JCU mission statement to describe what JCU is and what they are responsible for.

What is JCU?

The Joint Communications Unit (JCU) is a technical unit of the United States Special Operations Command charged to standardize and ensure interoperability of communication procedures and equipment of the Joint Special Operations Command and its subordinate units.

The JCU was activated at Ft. Bragg, NC in 1980, after the failure of Operation Eagle Claw. The JCU has earned the reputation of "DoD's Finest Communicators".

Why Automation?

There were two primary reasons why we sought out automation for JCU. The team I was a part of was responsible for ensuring network configurations were compliant and standard across a number of communication packages. The issue with the compliance and standardization process was the lack of defined compliance and standards. We did not have a version-controlled network configuration, and we had massive configuration drift on the network devices. We thrived on just making it work, which led at times, to hours of troubleshooting equipment. This process of attempting to baseline these communication packages took up to a week at a time, and at the end they just worked, and there was no standardization across the systems.

The second reason we sought automation was the creation of the Modular Family of Systems (MFS). This new suite of expeditionary communication packages was designed to replace all of the existing equipment JCU used for tactical communications. The initial configuration plan used many Excel spreadsheets as an IPAM solution for multiple enclaves
(separate network classifications) and a Word document for each MFS Type (Lite, Medium, Heavy). Due to the modular nature and the requirement to support any possible scenario, the Word documents were approximately 27 pages long for some network enclaves and device types.

What is MFS?

MFS is the suite of expeditionary communications packages that JCU needs to utilize in order to meet their communications requirements.

The MFS suite consists of Lite, Medium, and Heavy variants that all provide the same capabilities from a configuration perspective, but support different numbers of users.
The components of the MFS suite include the Cisco ESR5915, ESR6300, CSR1000V, CSR8000V, and ESS3300.

The components were designed to be modular in nature and able to be quickly swapped in and out using embedded event manager scripts to create dynamic connections between routers.

My team had to configure and field a large number of these systems very quickly and guarantee that they would work. After building that first batch manually and going through the painful process of populating a Word document with the correct hostnames, subnets, IPs, etc. that make up the host variables of a specific device, and encountering many fat-finger mistakes, we quickly realized we needed to come up with a better process. It took us over three weeks to configure and test the first set of equipment.

Automation Tool

A combination of Ansible, GitLab and Jinja2 became our solution to provide a standardized, configured and working device. We took the Word documents of network configurations and separated the different sections into many Jinja2 templates and added a bunch of logic to meet the needs of each MFS type.

```
!
{% if device_type == 'mfs_router_lite' %}
!
access-list 1 remark CONTROL VTY
access-list 1 permit 192.157.0.0 0.0.255.255
access-list 1 deny any log
!
{% else %}
!
access-list 1 remark CONTROL VTY
access-list 1 permit 192.168.252.254
access-list 1 permit 192.157.0.0 0.0.255.255
access-list 1 permit 10.11.11.0 0.0.0.7
access-list 1 deny any log
!
{% endif %}
```

IP management and host variables were moved to YAML files for each kit, which contained the variables and IP assignments for each kit type.

```
---
Kit_config:
- KL082:
hostname: KL082-HV-BR
device_type: mfs_router
isp_network: 192.168.x.x #(ISP network address)
NIPR_data_network: 192.168.x.x #(NIPR Data network address)
NIPR_voice_network: 192.168.x.x #(NIPR Voice network address)
tun_source: Vlan21
NIPR_tun_source: Vlan23 #( Tunnel source for NIPR tunnels)
manet_ip: 192.168.x.x #{Manet IP address}
vlan10_network: 172.29.x.x #( Vlan10 network address)
dun_dmvpn_int: Vlan60 #( IP address for ISP tunnel)

- KL082-SW:
hostname: KL082-HV-SW
device_type: mfs_switch
mgmt_ip: 172.29.x.x #(management address, 1 less than vlan10 gateway)
default_gw: 172.29.x.x #(router vlan 10 IP)
```

A static host file using the ini format was created to manage the connection variables and define host type and primary IP of each device.

```
[KL082]
KL-082-BR  ansible_host=192.168.0.0   device_type=mfs_router
KL082-SW   ansible_host=192.168.0.1

[KL084]
KL-084-BR  ansible_host=192.168.0.2   device_type=mfs_router
KL084-SW   ansible_host=192.168.0.3


[all:vars]
ansible_network_os = ios
ansible_connection = network_cli
ansible_become = yes
ansible_become_method = enable
```

The automation provided the solution we were looking for and allowed us to begin configuring and fielding MFS packages. Once the initial concept had been fleshed out and tested, we moved the Ansible playbooks and GitLab project to AWX in order to provide a user interface to request network configurations for our users.

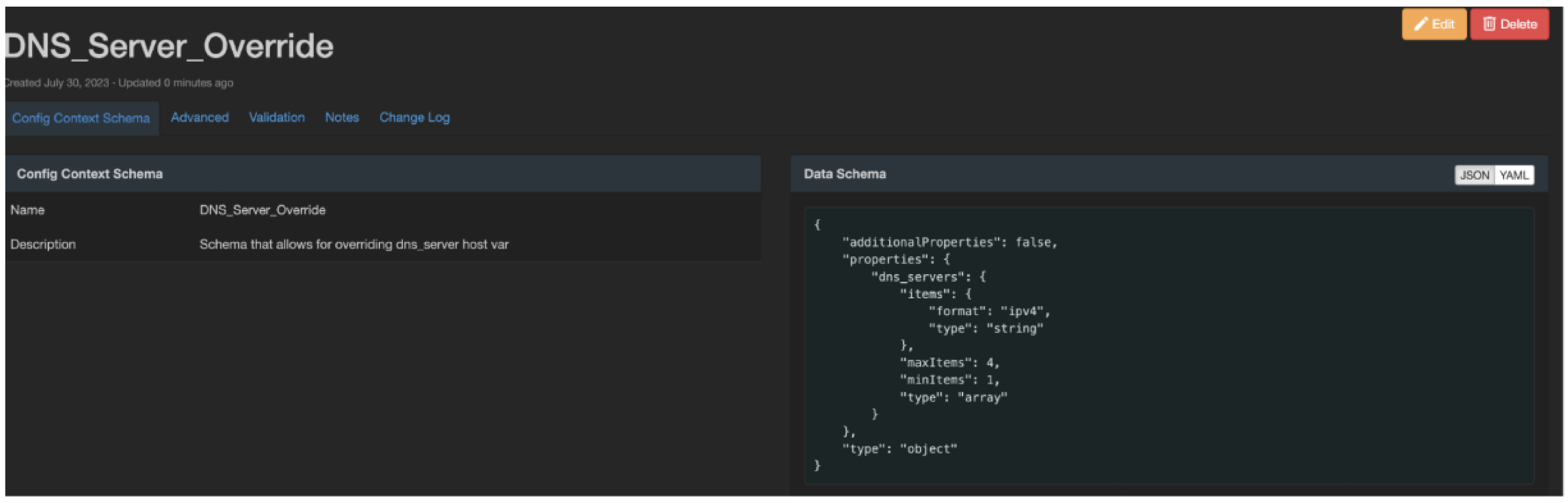
Why a Source of Truth?

We knew early on that using a static inventory and individual YAML files for host variables was never going to scale for us. It was very annoying to add or make a change to the inventory or YAML files. A branch needed to be created followed by a merge request before AWX could sync the project and finally create and deliver a config to a network device. If a mistake was made, this process would have to be repeated and take even longer. We wanted a smooth, quick, and streamlined process for adding and making changes to devices that would have an immediate effect when generating configurations.

After a lot of searching we found that a network source of truth (SOT) could solve both of the issues we were having with our inventory and our host variables. We currently use Nautobot as our network source of truth. Nautobot had a number of features that we thought could help us make fast and scalable changes to our inventories like config context and graphQL. Additionally the support from the Network to Code Slack channel and optional training available were very helpful for us.

We did some initial tests using Nautobot API calls for device objects and config context for group variables that did not scale for us; and as our device count grew, our inventory got increasingly slower. We found that we were pulling too much information from Nautobot.
We decided to make some changes to our group variable structure and moved to using graphQL. We found that using graphQL to query the exact information we needed from Nautobot to generate a config, reduced our inventory runtime from approximately 5 minutes to under 10 seconds.

Config Context gives us the ability to assign a variable to an object as a host variable in order to overwrite a group variable. This gave us some scalability in supporting more than just ourselves when it came to generating network configurations. The screenshot below shows the usage of config context schema and config context to allow users to overwrite a dns_server host variable. Config context schema ensures the user enters data in config context correctly so there will be no errors when generating device configurations.



DNS_Server_Override

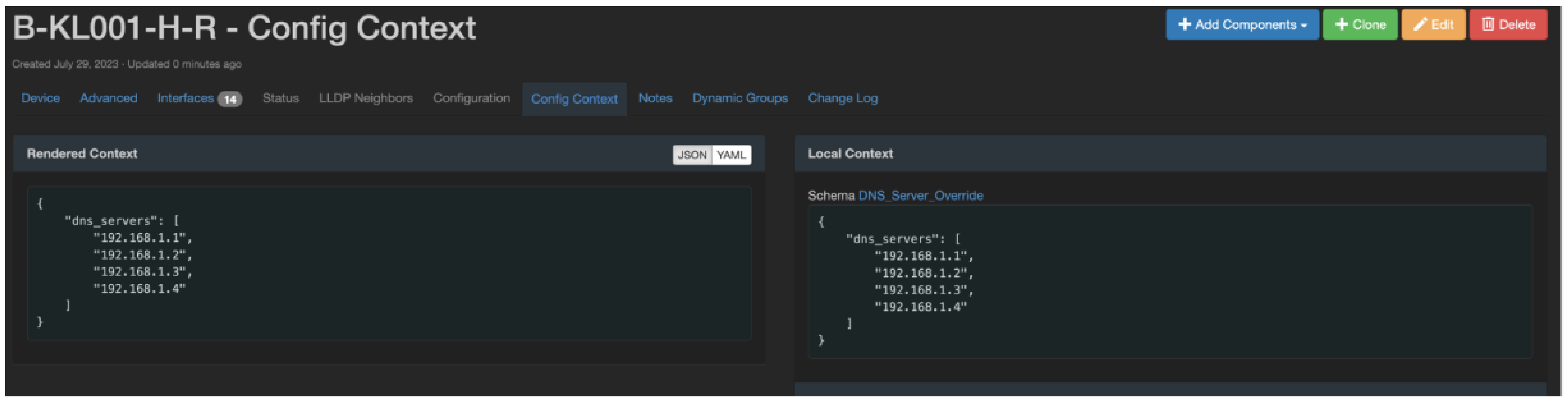
Created July 30, 2023 · Updated 0 minutes ago

Config Context Schema

Name	DNS_Server_Override
Description	Schema that allows for overriding dns_server host var

Data Schema

```
{
    "additionalProperties": false,
    "properties": {
        "dns_servers": {
            "items": {
                "format": "ipv4",
                "type": "string"
            },
            "maxItems": 4,
            "minItems": 1,
            "type": "array"
        }
    },
    "type": "object"
}
```



B-KL001-H-R - Config Context

Created July 29, 2023 · Updated 0 minutes ago

Rendered Context

```
{
    "dns_servers": [
        "192.168.1.1",
        "192.168.1.2",
        "192.168.1.3",
        "192.168.1.4"
    ]
}
```

Local Context

Schema DNS_Server_Override

```
{
    "dns_servers": [
        "192.168.1.1",
        "192.168.1.2",
        "192.168.1.3",
        "192.168.1.4"
    ]
}
```

Knowing that PKI authentication was going to be a requirement at some point, we wanted a solution that would integrate with Keycloak or other Identity Management solutions to give us two-factor authentication options.

We ended up creating a Python dynamic inventory to query Nautobot. This inventory would run every time a config generation request was made inside of AWX, ensuring that the config would contain the most recent changes from our preferred source of truth, Nautobot.query.

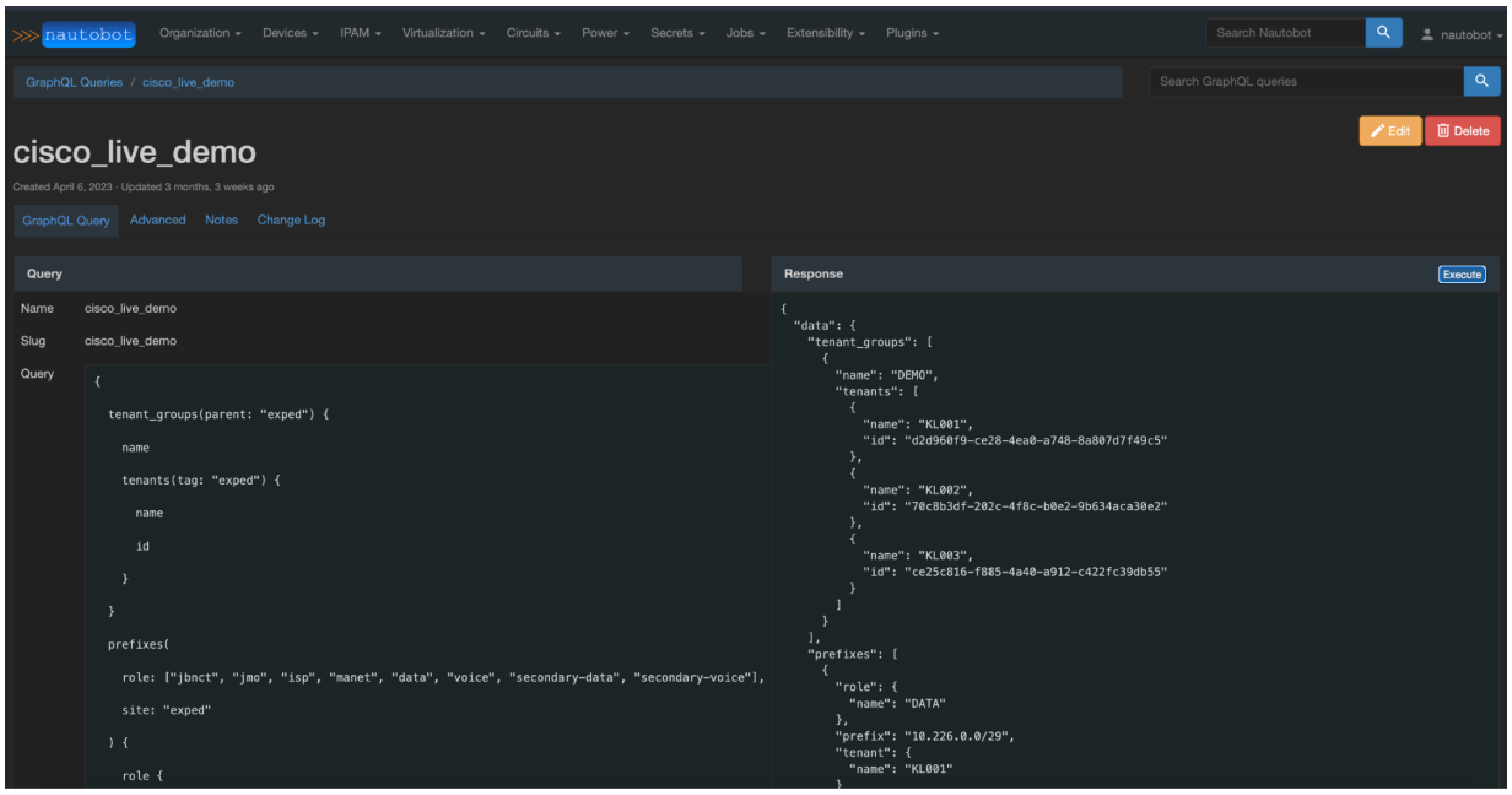
```python
def run_graphql_query(query_id: str) -> dict:
"""Runs an API call against Nautobot and runs a graphql query."""

graphql_url = f"{NAUTOBOT_BASE_URL}/extras/graphql-queries/{query_id}/run/"
data = requests.post(url=graphql_url, headers=HEADERS, verify=False)
return data.json()["data"]


if __name__ == "__main__":
# Pull config contexts, tenants, tenant groups, and hosts
exped_gql = run_graphql_query("a7c70e0a-18a0-4feb-934c-c753ea4b7def")

# Briefly format data received from above api calls
tenant_groups = exped_gql["tenant_groups"]
prefixes = exped_gql["prefixes"]
hosts = exped_gql["devices"]
```

The script would make an API call to Nautobot and return the results of a preconfigured graphQL query.



cisco_live_demo

Created April 6, 2023 · Updated 3 months, 3 weeks ago

Name cisco_live_demo

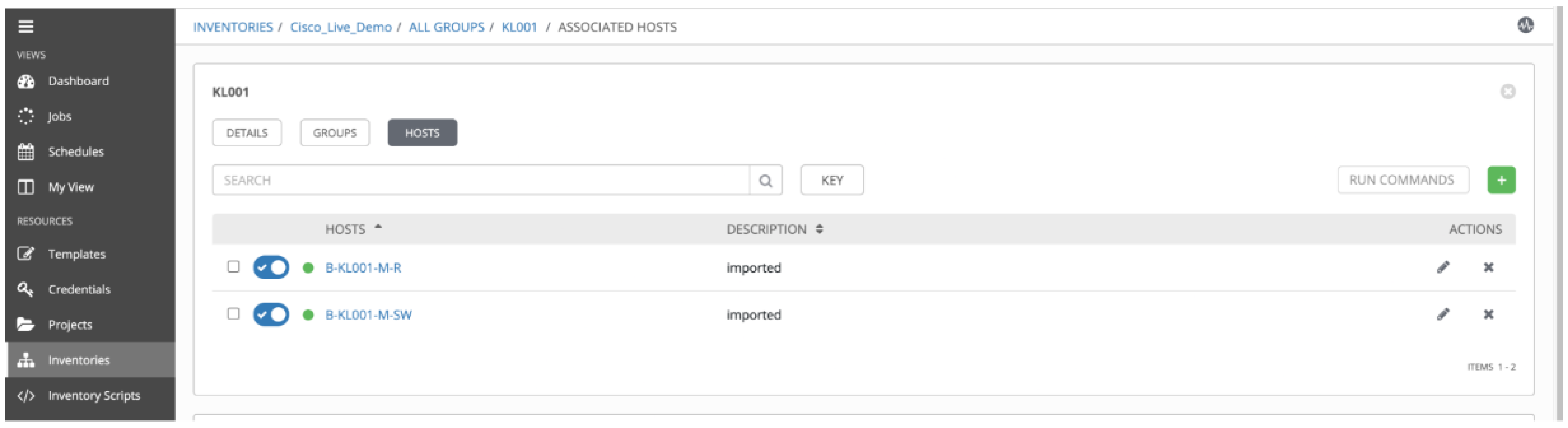
Slug cisco_live_demo

Query

```
{
  tenant_groups(parent: "exped") {
    name
    tenants(tag: "exped") {
      name
      id
    }
  }
  prefixes(
    role: ["jbnct", "jmo", "isp", "manet", "data", "voice", "secondary-data", "secondary-voice"],
    site: "exped"
  ) {
    role {
```

```
{
  "data": {
    "tenant_groups": [
      {
        "name": "DEMO",
        "tenants": [
          {
            "name": "KL001",
            "id": "d2d960f9-ce28-4ea0-a748-8a807d7f49c5"
          },
          {
            "name": "KL002",
            "id": "70c8b3df-202c-4f8c-b0e2-9b634aca30e2"
          },
          {
            "name": "KL003",
            "id": "ce25c816-f885-4a40-a912-c422fc39db55"
          }
        ]
      }
    ],
    "prefixes": [
      {
        "role": {
          "name": "DATA"
        },
        "prefix": "10.226.0.0/29",
        "tenant": {
          "name": "KL001"
        }
```

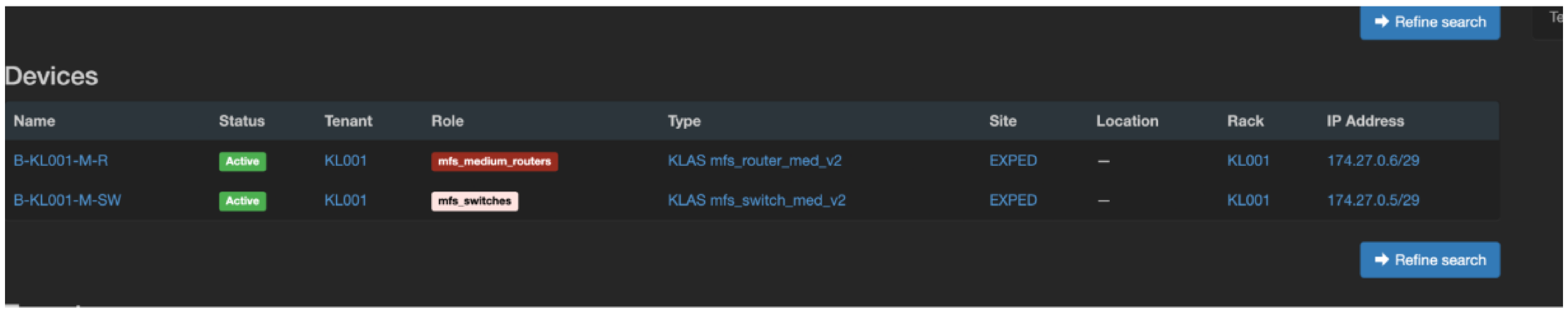
The results of the graphQL query are then parsed in the rest of the dynamic inventory and put into a format that AWX is able to consume.



HOSTS	DESCRIPTION	ACTIONS
B-KL001-M-R	imported	
B-KL001-M-SW	imported	

ITEMS 1 - 2

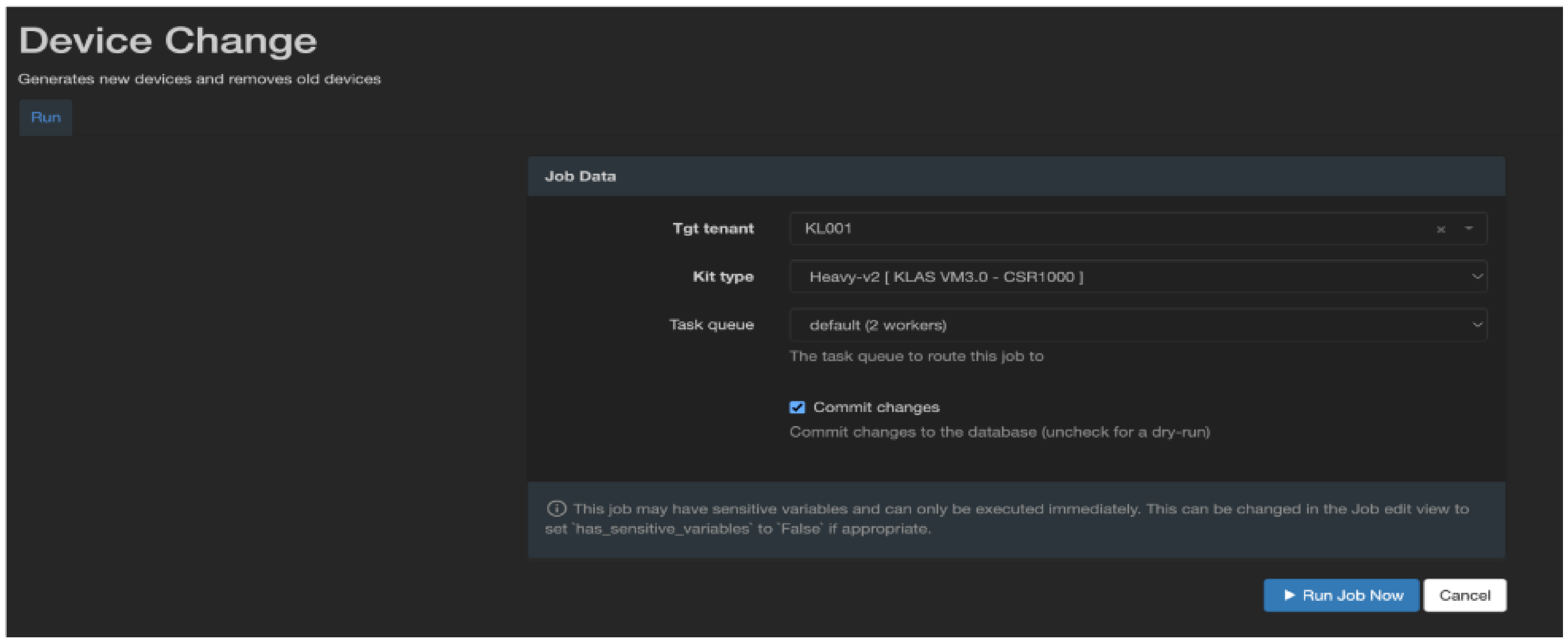
We also used the job feature to build a custom Python job that allowed us to add and/or change the device type of a tenant. This made it very easy for us to life cycle the Cisco ESR5915 with the ESR6300. Users were able to select the tenant object they wanted to change and then what they wanted to change it to. The next time they generated configurations for that tenant object, Ansible would have the latest information from Nautobot.
This screenshot shows that KL001 is currently a medium_v2, which is a Cisco 6300. If we wanted to change the device types of that tenant object inside of Nautobot manually we would need to delete all of the devices and re-create them from a new device type, while assigning IPs to interfaces and assigning primary IPs to the device and changing the hostname.



Devices

Name	Status	Tenant	Role	Type	Site	Location	Rack	IP Address
B-KL001-M-R	Active	KL001	mfs_medium_routers	KLAS mfs_router_med_v2	EXPED	—	KL001	174.27.0.6/29
B-KL001-M-SW	Active	KL001	mfs_switches	KLAS mfs_switch_med_v2	EXPED	—	KL001	174.27.0.5/29

Using the custom jobs option in Nautobot we are able to allow our users to run a single job that accomplishes all of the necessary changes in Nautobot to generate a new standardized config for that device type.



Device Change

Generates new devices and removes old devices

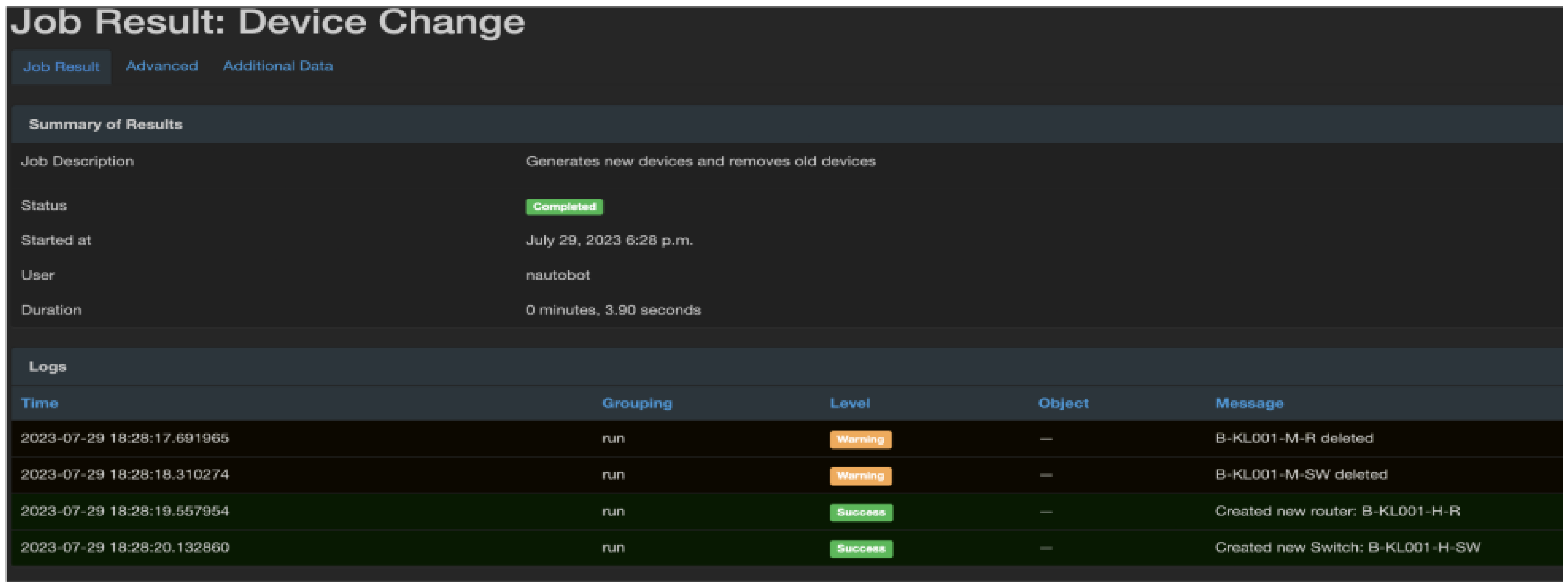
Job Data

Tgt tenant	KL001
Kit type	Heavy-v2 [KLAS VM3.0 - CSR1000]
Task queue	default (2 workers)

The task queue to route this job to

☑ Commit changes
Commit changes to the database (uncheck for a dry-run)

This job may have sensitive variables and can only be executed immediately. This can be changed in the Job edit view to set `has_sensitive_variables` to `False` if appropriate.



Job Result: Device Change

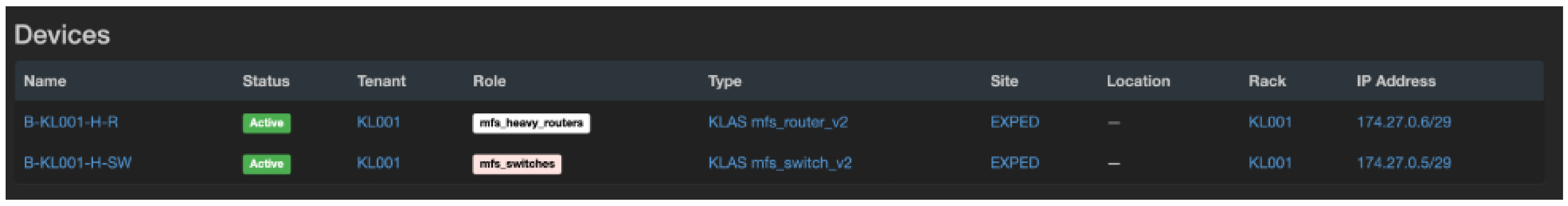
Summary of Results

Job Description	Generates new devices and removes old devices
Status	Completed
Started at	July 29, 2023 6:28 p.m.
User	nautobot
Duration	0 minutes, 3.90 seconds

Logs

Time	Grouping	Level	Object	Message
2023-07-29 18:28:17.691965	run	Warning	—	B-KL001-M-R deleted
2023-07-29 18:28:18.310274	run	Warning	—	B-KL001-M-SW deleted
2023-07-29 18:28:19.557954	run	Success	—	Created new router: B-KL001-H-R
2023-07-29 18:28:20.132860	run	Success	—	Created new Switch: B-KL001-H-SW

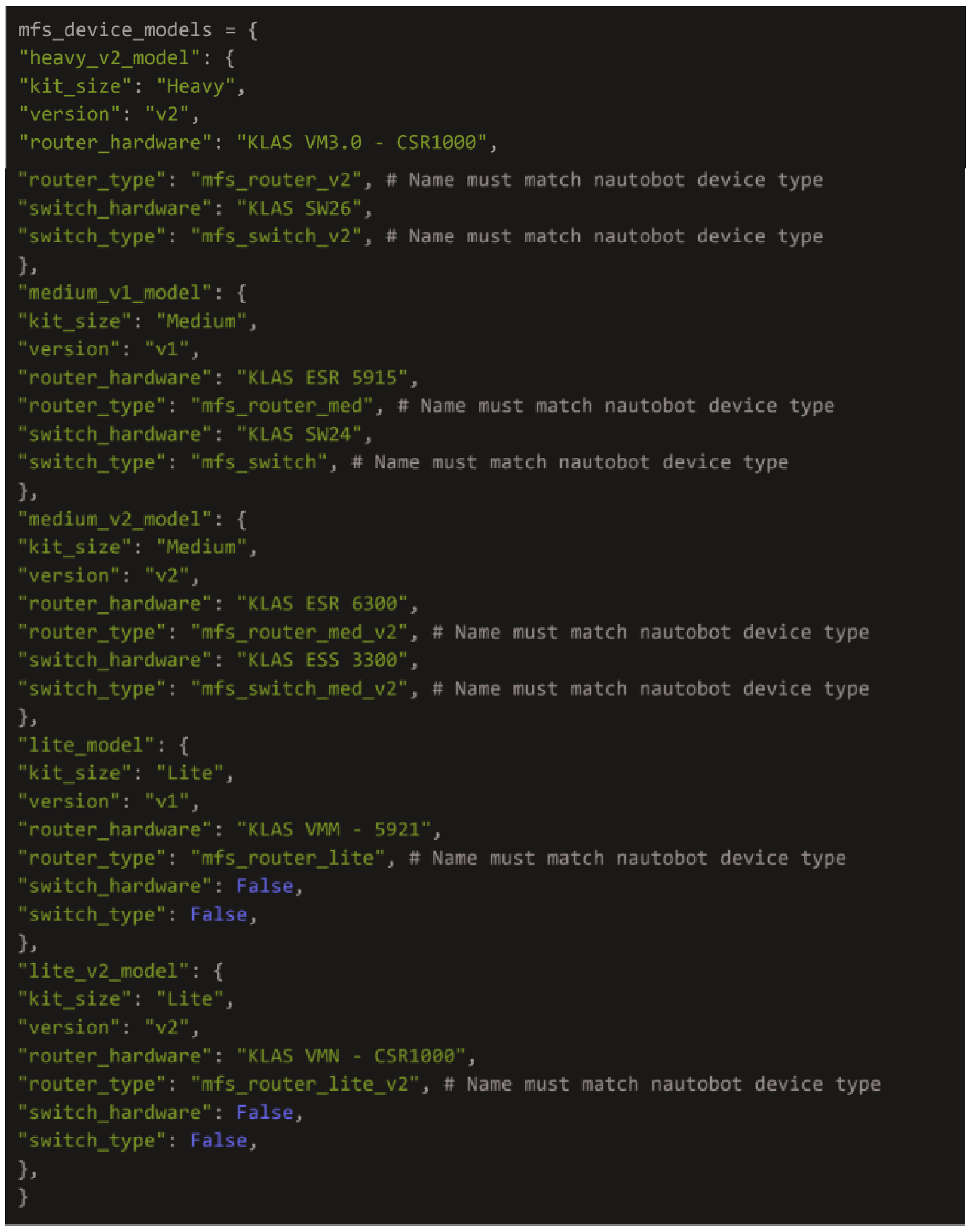
The final result of the job will be new devices within the existing tenant KL001 with updated information.



Devices

Name	Status	Tenant	Role	Type	Site	Location	Rack	IP Address
B-KL001-H-R	Active	KL001	mfs_heavy_routers	KLAS mfs_router_v2	EXPED	—	KL001	174.27.0.6/29
B-KL001-H-SW	Active	KL001	mfs_switches	KLAS mfs_switch_v2	EXPED	—	KL001	174.27.0.5/29

We provide a dictionary of options for the dropdown as a selection of choices, and then the rest of the jobs script deletes and creates the object based on what we deem the desired state of that device type.



```
mfs_device_models = {
"heavy_v2_model": {
"kit_size": "Heavy",
"version": "v2",
"router_hardware": "KLAS VM3.0 - CSR1000",
"router_type": "mfs_router_v2", # Name must match nautobot device type
"switch_hardware": "KLAS SW26",
"switch_type": "mfs_switch_v2", # Name must match nautobot device type
},
"medium_v1_model": {
"kit_size": "Medium",
"version": "v1",
"router_hardware": "KLAS ESR 5915",
"router_type": "mfs_router_med", # Name must match nautobot device type
"switch_hardware": "KLAS SW24",
"switch_type": "mfs_switch", # Name must match nautobot device type
},
"medium_v2_model": {
"kit_size": "Medium",
"version": "v2",
"router_hardware": "KLAS ESR 6300",
"router_type": "mfs_router_med_v2", # Name must match nautobot device type
"switch_hardware": "KLAS ESS 3300",
"switch_type": "mfs_switch_med_v2", # Name must match nautobot device type
},
"lite_model": {
"kit_size": "Lite",
"version": "v1",
"router_hardware": "KLAS VMM - 5921",
"router_type": "mfs_router_lite", # Name must match nautobot device type
"switch_hardware": False,
"switch_type": False,
},
"lite_v2_model": {
"kit_size": "Lite",
"version": "v2",
"router_hardware": "KLAS VMN - CSR1000",
"router_type": "mfs_router_lite_v2", # Name must match nautobot device type
"switch_hardware": False,
"switch_type": False,
},
}
```

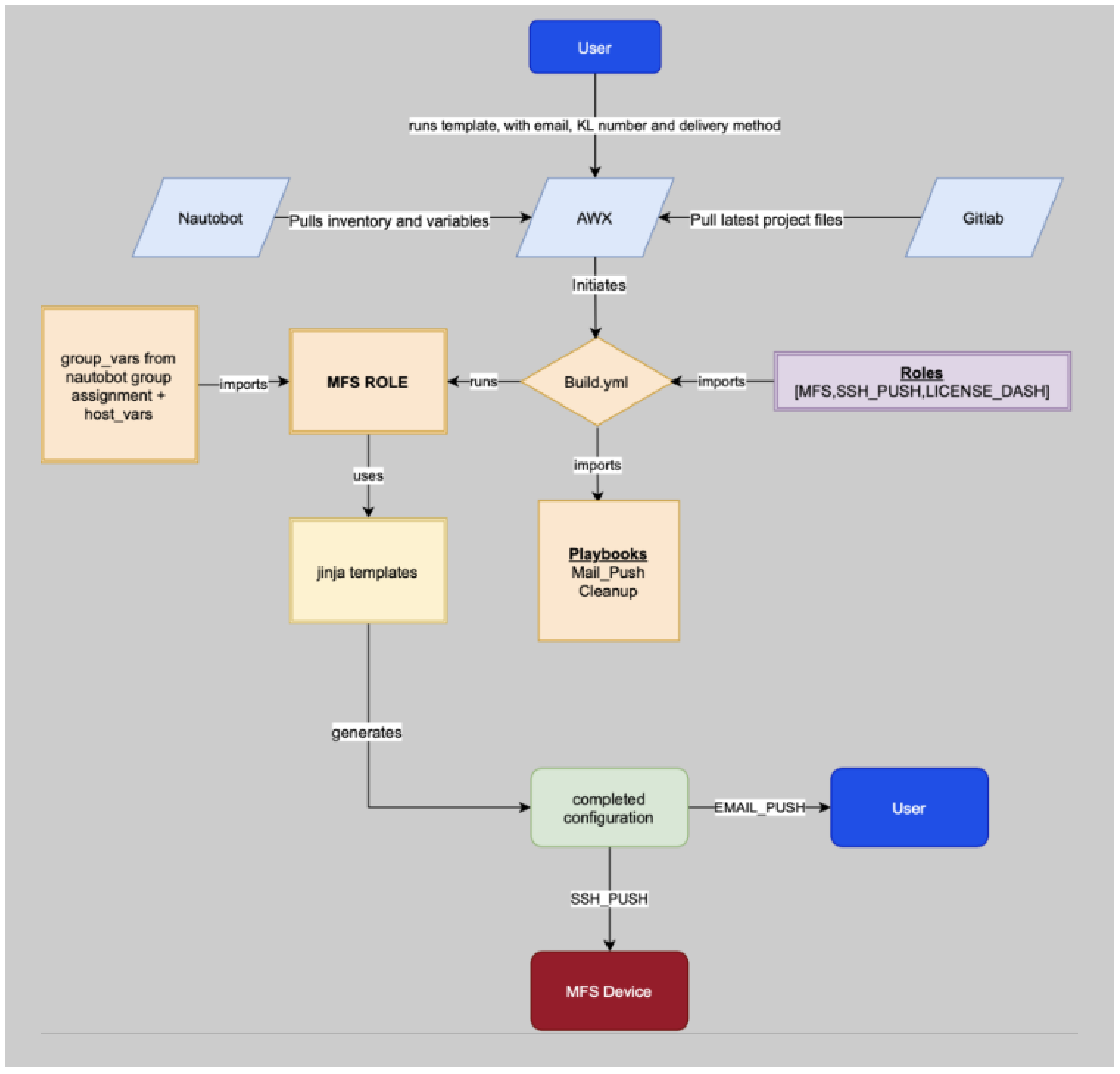
Using a network source of truth like Nautobot increases the speed and flexibility of your automation. It builds trust and allows for visibility into the desired state of network devices for all users. If used correctly it can correct the state of network devices and build a safe and secure environment for network operations.

Learning, Scaling, and Improving

Over time and with increased experience, we have made many changes to the automation process to include more complex logic in the Jinja2 templates, template inheritance with blocks, and CI/CD pipelines for testing network configurations to name a few. The process we use for automating network configurations continues to grow and improve as we learn more and gain experience.

Final Workflow

The final workflow of our automation process can be seen below, excluding the CI/CD pipelines used to validate Jinja syntax and network configurations, prior to allowing a merge request to succeed.


User
runs template, with email, KL number and delivery method
Nautobot
Pulls inventory and variables
AWX
Pull latest project files
Gitlab
Initiates
group_vars from nautobot group assignment + host_vars
imports
MFS ROLE
runs
Build.yml
imports
Roles [MFS,SSH_PUSH,LICENSE_DASH]
uses
imports
jinja templates
Playbooks Mail_Push Cleanup
generates
completed configuration
EMAIL_PUSH
User
SSH_PUSH
MFS Device

What else have we done and what else is on the roadmap?

- SSL Cert tracker (Nautobot Plugin)
- System baseline tracker (Nautobot Plugin)
- Domain Controller Replication (External powershell script)
- Sites and services standardization
- Automatic creation of A Records

I get asked a lot about how we were able to do this.

How did we get the time and skills?
How are we maintaining those skills or training new people that come into the organization?

Our leadership team is very supportive and encourages the teams within the organization to find new and more efficient ways to do business.
We are given time and space to work through difficult problem sets. We have dedicated team members who want to solve problems and enjoy learning new things.

We developed a four-month platform engineer onboarding course that runs twice a year. All qualified new hires that are coming to the unit go through this course and learn GitLab, Ansible, Jinja, network source of truth, Django, and Kubernetes.

If you are interested in coming to JCU to work on similar projects like Automation, Coding, Edge Computing, or Kubernetes, please go to https://www,jcu.mil and fill out an application.