Lab 9. Network Automation with Ansible

We will learn more about this, and what we can do to automate our network, in this lab by covering the following topics:

- Why automate network management?
- How Ansible manages networking devices
- How to enable network automation
- The available Ansible networking modules
- Connecting to network devices
- Environment variables for network devices
- Custom conditional statements for networking devices

Let's get started!

Enabling network automation

Before you can use Ansible for network automation, you need to make sure you have everything you need.

Based on the kind of connection method we are going to use, we need different dependencies. As an example, we are going to use a Cisco IOS device with [network_cli] connectivity.

The only requirements for Ansible network automation to work are as follows:

- Ansible 2.5+
- Proper connectivity with the network device

First, we need to check the Ansible version:

1. To ensure that you have a recent Ansible version, you can run the following command:

```
$ ansible --version
```

This will tell you the version of your Ansible installation.

2. If it's 2.5 or better, you can issue the following command (with the appropriate options) to check the connectivity of your network device:

```
$ ansible all -i n1.example.com, -c network_cli -u my_user -k -m ios_facts -e
ansible_network_os=ios all
```

This should return your device's facts, which proves that we are able to connect. As for any other target, Ansible is able to retrieve facts, and this is usually the first thing Ansible does when interacting with a target.

This is a key step since this allows Ansible to know the current state of the device and therefore act appropriately.

By running the [ios_facts] module on our target device, we are just executing this first standard step (so no changes will be performed on the device itself or its configurations), but this will confirm that Ansible is able to connect all the way to the device and perform commands on it.

Now, obviously, you could only actually run the preceding command and explore its behavior if you have access to a network device running Cisco IOS. We understand that not everyone will have the same networking equipment available to them for testing purposes (or indeed any!). Fortunately for us, a new breed of switches is becoming

available -- "white box" switches. These switches are made by a variety of manufacturers and are based on standardized hardware where you can install your own network operating system. One such operating system is Cumulus Linux, and a freely available test version of this, called Cumulus VX, is available for you to download.

At the time of writing, the download link for Cumulus VX is https://cumulusnetworks.com/products/cumulus-vx/. You will need to register to download it, but doing so gives you free access to the world of open networking.

Simply download the image appropriate to your hypervisor (for example, VirtualBox) and then run it just as you would run any other Linux virtual machine. Once you've done this, you can connect to the Cumulus VX switch, just like you would any other SSH device. For example, to run an ad hoc command to gather facts about all the switch port interfaces (which are enumerated as [swp1], [swp2], and [swpX] on Cumulus VX), you would run the following command:

```
$ ansible -i vx01.example.com, -u cumulus -m setup -a 'filter=ansible_swp*' all --ask-pass
```

If successful, this should result in pages of information about the switch port interface for your Cumulus VX-powered virtual switch. On my test system, the first part of this output looks like this:

```
vx01.example.com | SUCCESS => {
    "ansible_facts": {
        "ansible_swp1": {
            "active": false,
            "device": "swp1",
            "features": {
                "esp_hw_offload": "off [fixed]",
                "esp_tx_csum_hw_offload": "off [fixed]",
                "fcoe_mtu": "off [fixed]",
                "generic_receive_offload": "on",
                "generic_segmentation_offload": "on",
                "highdma": "off [fixed]",
...
```

As you can see, working with white box switches using an operating system such as Cumulus Linux has the advantage that you can connect using the standard SSH protocol, and you can even use the built-in [setup] module to gather facts about it. Working with other proprietary hardware is not much more difficult, but simply requires more parameters to be specified, as we showed earlier in this lab.

Now that you know the fundamentals of enabling network automation, let's learn how to discover the appropriate networking modules for our desired automation task in Ansible.

Reviewing the available Ansible networking modules

At the moment, there are thousands of modules on a total of more than 20 different networking platforms. Let's learn how to find the ones more relevant to you:

1. First of all, you need to know which device type you have and how Ansible calls it. On the https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html page, you can find the different device types that Ansible supports and how they are designated. In our example, we will use Cisco IOS as an example.
2. On the https://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html page, you can search for the category dedicated to the family of switches you need, and you'll be able to see all the modules you can use.

The list of modules is way too big and family-specific for us to talk about them in depth. This list is getting larger every release, with often hundreds of new additions in every release.

If you are familiar with how to configure the device in a manual fashion, you will quickly find the name of the modules fairly natural, so it will be easy for you to understand what they do. However, let's go through a handful of examples from the collection of Cisco IOS modules -- specifically, with reference to https://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html#ios:

- [ios_banner]: As the name suggests, this module will allow you to tweak and modify the login banner (what in many systems is called [motd]).
- [ios_bgp]: This module allows you to configure BGP routes.
- [ios_command]: This is the IOS equivalent of the Ansible [command] module, and it allows you to perform many different commands. As for the [command] module, this is a very powerful module, but it's better to use specific modules for the operation we are going to perform, if they are available.
- [ios_config]: This module allows us to make pretty much any changes to the configuration file of the device. As for the [ios_command] module, this is a very powerful module, but it's better to use specific modules for the operation we are going to perform, if they are available. The idempotency for this module is only guaranteed if no abbreviated commands are used.
- [ios_vlan]: This module allows the configuration of VLANs.

These are just a few examples, but there are many more modules for Cisco IOS (27, at the time of writing), and if you cannot find a specific module to perform the operation you want, you can always fall back to [ios_command] and [ios_config], which, thanks to their flexibility, will allow you to perform any operation you can think of.

In contrast, if you are working with a Cumulus Linux switch, you'll find there is just one module -- [nclu] (see https://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html#cumulus). This reflects the fact that all configuration work in Cumulus Linux is handled with this command. If you need to customize the message of the day or other aspects of the Linux operating system, you can do this in the normal manner (for example, using the [template] or [copy] modules, which we have demonstrated previously in this course).

Connecting to network devices

As we have seen, there are some peculiarities in Ansible networking, so specific configurations are required.

In order to manage network devices with Ansible, you need to have at least one to test on. Let's assume we have a Cisco IOS system available to us. It is accepted that not everyone will have such a device to test on, so the following is offered as a hypothetical example only.

Going by the https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html page, we can see that the correct [ansible_network_os] for this device is [ios] and that we can connect to it using both [network_cli] and [local]. Since [local] is deprecated, we are going to use [network_cli]. Follow these steps to configure Ansible so that you can manage IOS devices:

1. First, let's create the inventory file with our devices in the [routers] group:

```
[routers]
n1.example.com
n2.example.com

[cumulusvx]
vx01.example.com
```

2. To know which connection parameters to use, we will set Ansible's special connection variables so that they define the connection parameters. We'll do this in a group variables subdirectory of our playbook, so we will

need to create the [group_vars/routers.yml] file with the following content:

```
---
ansible_connection: network_cli
ansible_network_os: ios
ansible_become: True
ansible_become_method: enable
```

By virtue of these special Ansible variables, it will know how to connect to your devices. We covered some of these examples earlier in this course, but as a recap, Ansible uses the values of those variables to determine its behavior in the following ways:

- [ansible_connection]: This variable is used by Ansible to decide how to connect to the device. By choosing [network_cli], we are instructing Ansible to connect to the CLI over SSH mode, as we discussed in the previous paragraph.
- [ansible_network_os]: This variable is used by Ansible to understand the device family of the device we are going to use. By choosing [ios], we are instructing Ansible to expect a Cisco IOS device.
- [ansible_become]: This variable is used by Ansible so that we decide whether to perform privilege escalation on the device or not. By specifying [True], we are telling Ansible to perform privilege escalation.
- [ansible_become_method]: There are many different ways to perform privilege escalation on the various devices (normally [sudo] on a Linux server -- this is the default setting), and for Cisco IOS , we must set this to [enable].

With that, you have learned the necessary steps to connect to network devices.

To validate that the connection is working as expected (assuming you have access to a router running Cisco IOS), you can run this simple playbook, called [ios_facts.yaml]:

```
---
- name: Play to return facts from a Cisco IOS device
  hosts: routers
  gather_facts: False
  tasks:
    - name: Gather IOS facts
      ios_facts:
        gather_subset: all
```

You can run this using a command such as the following:

```
$ ansible-playbook -i hosts ios_facts.yml --ask-pass
```

If it returns successfully, this means that your configuration is correct and you've been able to give Ansible the necessary authorization to manage your IOS device.

Similarly, if you wanted to connect to a Cumulus VX device, you could add another group variables file called [group_vars/cumulusvx.yml] containing the following code:

```
---
ansible_user: cumulus
become: false
```

An analogous playbook that returns all the facts about our Cumulus VX switches could look like this:

```
---
- name: Simply play to gather Cumulus VX switch facts
  hosts: cumulusvx
  gather_facts: no

  tasks:
    - name: Gather facts
      setup:
        gather_subset: all
```

You can run this in a normal manner by using a command such as the following:

```
$ ansible-playbook -i hosts cumulusvx_facts.yml --ask-pass
```

If successful, you should see the following output from your playbook run:

```
SSH password:

PLAY [Simply play to gather Cumulus VX switch facts]
*********************************************************************************



TASK [Gather facts]
*********************************************************************************

ok: [vx01.example.com]

PLAY RECAP
*********************************************************************************

vx01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

This demonstrates the techniques for connecting to two different types of network devices in Ansible, including one you can test by yourself without access to any special hardware. Now, let's build on this by looking at how to set environment variables for network devices in Ansible.

Environment variables for network devices

Very often, the complexity of networks is high and the network systems are very varied. For those reasons, Ansible has a huge amount of variables that can help you tweak it so that you can make Ansible fit your environment.

Let's suppose you have two different networks (that is, one for computing and one for network devices) that can't communicate directly, but have to pass through a bastion host to reach one from the other. Since we have Ansible in the computing network, we will need to jump networks using the bastion host to configure an IOS router in the management network. Also, our target switch needs a proxy to reach the internet.

To connect to the IOS router in the database network, we will need to create a new group for our network devices, which are on a separate network. For this example, this might be specified as follows:

```
[bastion_routers]
n1.example.com
n2.example.com
```

```
[bastion_cumulusvx]
vx01.example.com
```

Following the creation of our updated inventory, we can create a new group variables file, such as [group_vars/bastion_routers.yaml], with the following content:

```
---
ansible_connection: network_cli
ansible_network_os: ios
ansible_become: True
ansible_become_method: enable
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q bastion.example.com"'
proxy_env:
    http_proxy: http://proxy.example.com:8080
```

We can also do the same for our Cumulus VX switches if they are behind a bastion server by creating a [group_vars/bastion_cumulusvx.yml] file:

```
---
ansible_user: cumulus
ansible_become: false
ansible_ssh_common_args: '-o ProxyCommand="ssh -W %h:%p -q bastion.example.com"'
proxy_env:
    http_proxy: http://proxy.example.com:8080
```

In addition to the options we discussed in the previous section, we now have two additional options:

- [ansible_ssh_common_args]: This is a very powerful option that allows us to add additional options to the SSH connections so that we can tweak their behavior. These options should be fairly straightforward to identify since you are already using them in your SSH configurations to simply SSH to the target machine. In this specific case, we are adding a [ProxyCommand], which is the SSH directive to perform a jump to a host (usually a bastion host) so that we can enter the target host securely.
- [http_proxy]: This option, which is below the [proxy_env] option, is key in environments where network isolation is strong, and therefore your machines can't interact with the internet unless they use a proxy.

Assuming you have set up passwordless (for example, SSH key-based) access to your bastion host, you should be able to run an ad hoc Ansible [ping] command against your Cumulus VX host, as follows:

```
$ ansible -i hosts -m ping -u cumulus --ask-pass bastion_cumulusvx
SSH password:

vx01.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

Note that the use of the bastion server becomes transparent -- you can carry on automating with Ansible as if you were on the same flat network. If you have access to a Cisco IOS-based device, you should be able to run a similar command against the [bastion_routers] group as well and achieve similarly positive results. Now that you have learned the necessary steps to set environment variables for network devices, and indeed access them with Ansible, even when they are on isolated networks, let's learn how to set conditional statements for networking devices.

Conditional statements for networking devices

Although there are no networking-specific Ansible conditionals, conditionals are fairly common in networking-related Ansible usage.

In networking, it's common to enable and disable ports. To have data pass through the cable, both ports at the ends of the cable should be enabled and result in a "connected" state (some vendors will use different names for this, but the idea is the same).

Let's suppose we have two Arista Networks EOS devices and we issued the ON status on the ports and need to wait for the connection to be up before proceeding.

To wait for the [Ethernet4] interface to be enabled, we will need to add the following task in our playbook:

```
- name: Wait for interface to be enabled
  eos_command:
      commands:
          - show interface Ethernet4 | json
      wait_for:
          - "result[0].interfaces.Ethernet4.interfaceStatus eq connected"
```

[eos_command] is the module that allows us to issue free-formed commands to an Arista Networks EOS device. The command itself needs to be specified in an array in the [commands] option. With the [wait_for] option, we can specify a condition, and Ansible will reiterate on the specified task until the condition is satisfied. Since the command's output is redirected to the [json] utility, the output will be a JSON, so we can traverse its structure using Ansible's ability to manipulate JSON data.

We can achieve similar results on Cumulus VX -- for example, we can query the facts gathered from the switch to see if port [swp2] is enabled. If it is not, then we will enable it; however, if it is enabled, we will skip the command. We can do this with a simple playbook, follows:

```
---
- name: Simple play to demonstrate conditional on Cumulus Linux
  hosts: cumulusvx

  tasks:
    - name: Enable swp2 if it is disabled
      nclu:
        commands:
          - add int swp2
        commit: yes
      when: ansible_swp2.active == false
```

Notice the use of the [when] clause in our task, meaning we should only issue the configuration directive if [swp2] is not active. If we were to run this playbook for the first time on an unconfigured Cumulus Linux switch, we should see an output similar to the following:

```
PLAY [Simple play to demonstrate conditional on Cumulus Linux]
************************************************************

TASK [Gathering Facts]
************************************************************
ok: [vx01.example.com]
```

```
TASK [Enable swp2 if it is disabled]
************************************************************
changed: [vx01.example.com]

PLAY RECAP
************************************************************
vx01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

As we can see, the [nclu] module committed our change to the switch configuration. However, if we were to run the playbook a second time, the output should be more like this:

```
PLAY [Simple play to demonstrate conditional on Cumulus Linux]
************************************************************

TASK [Gathering Facts]
************************************************************
ok: [vx01.example.com]

TASK [Enable swp2 if it is disabled]
************************************************************
skipping: [vx01.example.com]

PLAY RECAP
************************************************************
vx01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=1 rescued=0 ignored=0
```

This time, the task was skipped as the Ansible facts show that port [swp2] is already enabled. This is obviously an incredibly simple example, but it shows how you can work with conditionals on a network device very much in the same way that you have already seen conditionals being used on Linux servers, earlier in this course.

That concludes our brief look at network device automation with Ansible -- more in-depth work would require a look at network configurations and necessitate more hardware, so this is beyond the scope of this course. However, I hope that this information demonstrates to you that Ansible can be used effectively to automate and configure a wide array of network devices.

Summary

In this lab, you learned about the reasons for automating network management. You then looked at how Ansible manages network devices, how to enable network automation in Ansible, and how to locate the Ansible modules necessary to perform the automation tasks you wish to complete. Then, through practical examples, you learned how to connect to network devices, how to set environment variables (and connect to isolated networks via bastion hosts), and how to apply conditional statements to Ansible tasks for network device configuration.

In the next lab, we will learn how to manage Linux containers and cloud infrastructures using Ansible.

Questions

1. Which of these is NOT one of the four major connection types that Ansible uses for connecting to those network devices?

A) [netconf]

B) [network_cli]

C) [local]

D) [netstat]

E) [httpapi]

2. True or False: The [ansible_network_os] variable is used by Ansible to understand the device family of the device we are going to use.

A) True

B) False

3. True or False: In order to connect to an IOS router in a separate network, you need to specify the special connection variables for the host, possibly as inventory group variables.

A) True

B) False