Lab 9 – Jinja2 Templating

Introduction

We can use Jinja2 templating with ansible to enable dynamic access to variables and to create special filters or used with loops and such. All the templating occurs on the Ansible controller/host before it is sent and executed on the target machine.

Whether you know or it or not we have already done some Jinja2 templating already. Anytime in the previous labs when we referenced “{{ variable }}” that was using Jinja2 templating. We will go over some more complex examples and ways to use it.

Please refer to the **Ansible-Pod-Info.docx** file for information on connecting to your Ansible host.

Don’t forget to change the XX in your inventory file based on your Pod information.

1. Network CLI Filters

1.1 One way we can use Jinja2 templating is by formatting data that we get back from output of commands.

Let’s say for example we wanted to parse the output of show interface into a better format that is easier to read and create vars out of them, well Ansible has a way to do this using **parse_cli** we will go over how to parse out the show interface output next.

First let’s copy our group_vars/nxos structure from the previous lab so we don’t have to re-make it and let us ensure we are in the proper folder as well.

We are also creating a parsers/nxos folder structure that we will put our spec file for the **parse_cli** command in.

```
cd ~/ansible_labs/lab9-jinja2

cp -R ../lab8-loops/group_vars .

cp ../lab8-loops/inventory .
```

```
cp ../lab8-loops/ansible.cfg .

mkdir parsers

mkdir parsers/nxos
```

1.2 Now we have a **group_vars** folder setup with our vault already configured. So now let us create a playbook that contains an example Jinja2 filter using **parse_cli**

```
cat > parse_cli.yaml <<EOF

- name: Parse CLI Example

  hosts: nxos

  tasks:

    - name: get show interface

      nxos_command:

        commands:

          - show interface

      register: iface_output

    - name: generate fact from interface output

      set_fact:

        nxos_ifaces: "{{ iface_output.stdout[0] |parse_cli('
./parsers/nxos/show_interface_parser.yaml')}}"

    - name: Debug output of nxos_ifaces

      debug: var=nxos_ifaces
```

```
EOF
```

1.3 Now let's create the **show_interface_parser.yaml** file to parse our output for us. We need to make sure this is created in the parsers/nxos directory since that is what our playbook is looking for it in.

```
cat > ./parsers/nxos/show_interface_parser.yaml <<EOF

vars:

  interface:

    name: "{{ item[0].name }}"

    link_protocol_state: "{{ item[0].oper }}"

    admin_state: "{{ item[5].admin }}"

    ipv4: "{{ item[1].match[0] }}"

    mtu: "{{ item[2].match[0] }}"

    port_mode: "{{ item[3].match[0] }}"

    duplex: "{{ item[4].match[0] }}"

    description: "{{ item[6].match[0] }}"


keys:

  interfaces:

    value: "{{ interface }}"

    start_block: "^Ethernet.+$"
```

```
    end_block: ".+interface reset"

    items:

      - "^(?P<name>Ethernet.+) is (?P<oper>\\\w+)"

      - "\\\s+Internet Address is (.+)"

      - "\\\s+ MTU (\\\d+)"

      - "\\\s+Port mode is (\\\w+)"

      - "\\\s+(\\\w+-duplex)"

      - "admin state is (?P<admin>\\\w+)"

      - "Description: (.+)"

EOF
```

1.4 Now let's run our playbook and see what kind of output we get here from this.

A few things that are going on above here in the playbook before we actually run this. The way the parse_cli option works is it allows us to give it output and have it formatted into JSON.

We use the spec file located in the parsers/nxos folder to tell the parse_cli command how to parse the output. By us setting the set_fact option we are able to create a new variable containing this new json formatted output that contains different items basically.

Let's run the playbook and see what kind of output we see from this.

```
ansible-playbook parse_cli.yaml --ask-vault
```

You should have some output similar to below

```
Vault password:


PLAY [Parse CLI Example] ********************************
**********************************************************
**



TASK [Gathering Facts] **********************************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [get show interface] *******************************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [generate fact from interface output] **************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [Debug output of nxos_ifaces] **********************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain] => {
```

```
    "nxos_ifaces": [
        {
            "admin_state": "up",
            "description": "Aggregate description",
            "duplex": "auto-duplex",
            "ipv4": null,
            "link_protocol_state": "down",
            "mtu": 9216,
            "name": "Ethernet1/1",
            "port_mode": null
        },
        {
            "admin_state": "up",
            "description": "Core uplink Jumbo MTU",
            "duplex": "auto-duplex",
            "ipv4": null,
            "link_protocol_state": "down",
            "mtu": 9216,
            "name": "Ethernet1/2",
            "port_mode": null
```

```
        },… More data follows this
```

Notice in the output how we get our items for each object here?

This gives us a variable that basically contains a list of dictionary objects here with each interface and their information. The [that you see at the beginning of the output and the] you see at the end designate this is a List and the { } are the dict/json objects.

How do you think we would change the debug output to only show us the first result, or say the first 10?

1.4 Now let's take this same playbook and expand on it. Let's say for example we want to create a Jinja2 template to loop through these interfaces and write to a file a list of interfaces where the MTU is 9216.

First let's create the playbook in the root of our directory.

```
cat > parse_cli_mtu.yaml <<EOF

- name: Parse CLI Example

  hosts: nxos

  tasks:

    - name: get show interface

      nxos_command:

        commands:

          - show interface

      register: iface_output

    - name: generate fact from interface output
```

```
      set_fact:

        nxos_ifaces: "{{ iface_output.stdout[0] |parse_cli('
./parsers/nxos/show_interface_parser.yaml')}}"

    - name: Write matching interfaces to file

      template:

       src: ./templates/interfaces.j2

       dest: ./jumbo_mtus.txt

EOF
```

1.5 Now we need to create our templates directory and interfaces.j2 Jinja2 template file. We are using a for loop and if condition to only write the objects to the file that match a MTU of 9216.

```
mkdir templates

cat > templates/interfaces.j2 <<EOF

Interfaces With Jumbo MTU:

{% for interface in nxos_ifaces %}

 {% if interface['mtu'] == 9216 %}

   {{ interface['name'] }}

 {% endif %}

{% endfor %}

EOF
```

Now we can run our playbook and see if it creates the file we specified jumbo_mtus.txt

```
ansible-playbook parse_cli_mtu.yaml --ask-vault
```

And we should get output similar to this:

```
Vault password:



PLAY [Parse CLI Example] ************************************
**********************************************************
**



TASK [Gathering Facts] **************************************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [get show interface] ***********************************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [generate fact from interface output] ******************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]
```

```
TASK [Write matching interfaces to file] ********************
*************************************************************
**

changed: [n9k-standalone-XX.localdomain]



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

n9k-standalone-XX.localdomain : ok=4    changed=1    unreach
able=0    failed=0
```

And now if we cat our jumbo_mtus.txt we should see something like below:

```
cat jumbo_mtus.txt

Interfaces With Jumbo MTU:

    Ethernet1/1

     Ethernet1/2
```

2. Creating Configuration With Jinja Templates

2.1 We can also use Jinja2 templating to create configurations for switches or other options that might not be directly supported in a certain module. In this next example we will use a jinja2 template to update some basic switch configuration.

There might be times where you cannot complete your entire needs with just modules, you might have to use a Jinja template and push actual configuration to a device.

Let's start by creating a new template file for our configuration template. In our example we will just showcase setting an interface configuration directly using a template and the **nxos_config** module.

In this template file we are just using actual input rather than using variables. These could also be variables using the {{ }} jinja constructor to specify variables.

In our jinja template we are just supplying what the configuration would normally see for an interface configuration and we are supplying it. This could be an entire switch configuration if we so chose.

```
cat > templates/int_config.j2 <<EOF

interface Ethernet1/1

  description This is a jinja desc

  mtu 1500

  no shutdown

EOF
```

2.2 Now let's create our playbook referencing this jinja template.

```
cat > int_config.yaml <<EOF

- name: Jinja Interface Example

  hosts: nxos

  tasks:

    - name: Write interface configuration to switch.

      nxos_config:

       src: ./templates/int_config.j2
```

```
       save_when: changed

EOF
```

2.3 Now we can run our playbook and see if it modifies our interface information

```
ansible-playbook int_config.yaml --ask-vault
```

We should see some output like below then if we login to our switch and run the show int command listed below we should see our configuration applied.

```
Vault password:



PLAY [Jinja Interface Example] ******************************
**********************************************************
**



TASK [Gathering Facts] **************************************
**********************************************************
**

ok: [n9k-standalone-XX.localdomain]



TASK [Write interface configuration to switch.] *************
**********************************************************
**

changed: [n9k-standalone-XX.localdomain]
```

```
PLAY RECAP *********************************************************
**

n9k-standalone-XX.localdomain : ok=2    changed=1    unreach
able=0    failed=0
```

2.4 Now let's ssh to our switch and run the below command to see if our configuration was applied.

```
show run interface Ethernet1/1
```

We should see something like below, notice how the MTU is gone now? It was at 9216 before but because it is at 1500 which is default it doesn't show it here.

```
interface Ethernet1/1

  description This is a jinja desc

  no switchport

  no shutdown
```

3. Knowledge Check - EOS Configuration Jinja

So in this knowledge check you will be using a Jinja template to apply some configuration to an Arista EOS device. We want you to accomplish the following.

Your working directory will be **~/ansible_labs/lab9-jinja2/knowledge_check3**

1. Using Jinja templating create the following the information on the switch:
    a. Vlan 2 with name production
    b. Vlan 3 with name dev

    c. Vlan 4 with name vpn
2. You will accomplish this using the variable option for your vlans and doing a for loop in the jinja template.
3. Remember you can use a group vars file to setup all your connection information and the vlan's that you will loop through.

Version Control Commit

Now add all your new files to your repository and push it up, reference the GitHub lab if you get stuck or ask for help.