

Introduction to Ansible for Network Engineers

Lab 1 - Ansible command line

The Ansible Command

From the command line of your Ansible server you can run the “ansible” command to perform simple tasks. For larger tasks, involving multiple changes or devices, an Ansible playbook is used. Playbooks are discussed in more detail in the Lab 3.

Ansible facts

Ansible uses a system of data gathering commands to automatically gather information about the device being managed. Examples of the data gathered:

- Interface IP addresses
- Hostname
- Version of software

Task 1 - Gathering facts for a Cisco device

Make sure you are in the lab1 directory, and your shell prompt starts with "(Lab)". Issue the "pwd" command to confirm your present working directory.

```
(Lab) → lab1 pwd
/home/student1/labs/lab1
```

The commands you will run have been recorded into text files so that you can copy and paste them.

```
(Lab) → lab1 ls
ansible.cfg  inventory  task1.txt  task2.txt  task3.txt
```

To view the command for the first task, use the "cat <filename>" command.

```
(Lab) → lab1 cat task1.txt
ansible cisco -m cisco.ios.ios_facts
(Lab) → lab1
```

Try the command from task1 yourself.

```
(Lab) → lab1 ansible cisco -m cisco.ios.ios_facts
r1 | SUCCESS => {
  "ansible_facts": {
    "ansible_net_api": "cliconf",
    "ansible_net_gather_network_resources": [],
    "ansible_net_gather_subset": [
      "default"
    ],
    "ansible_net_hostname": "csr1",
    "ansible_net_image": "bootflash:packages.conf",
    "ansible_net_iostype": "IOS-XE",
    "ansible_net_model": "CSR1000V",
    "ansible_net_operatingmode": "autonomous",
```

```
    "ansible_net_python_version": "3.11.6",
    "ansible_net_serialnum": "9BNAC028QW1",
    "ansible_net_system": "ios",
    "ansible_net_version": "17.03.01a",
    "ansible_network_resources": {}
  },
  "changed": false
}
(Lab) → lab1
```

Ansible has gathered basic facts about the configuration of r1, our Cisco router. The facts are displayed in a data format called Javascript Object Notation (JSON).

In JSON, variables are recorded as key, value pairs. For example, "variable_name": "variable_value". This creates a simple variable that holds a string of text.

Look at the output of the command you ran. On the 8th line of output, there is a variable named "ansible_net_hostname". Then looking to the right of the :, the value of this variable is "csr1".

Facts can be used to make decisions within Ansible, and for populating data in reports and templated files.

Question

What version of IOS is router r1 running?

Look for the ansible_net_version variable.

```
"ansible_net_version": "17.03.01a",
```

Task 2 - Gathering facts from a Juniper device

Cat the file "task2.txt" to see how to gather facts from a Juniper device. Run the command.

```
(Lab) → lab1 cat task2.txt
ansible juniper -m junipernetworks.junos.junos_facts
(Lab) → lab1 ansible juniper -m junipernetworks.junos.junos_facts
r2 | SUCCESS => {
  "ansible_facts": {
    "ansible_net_api": "netconf",
    "ansible_net_gather_network_resources": [],
    "ansible_net_gather_subset": [
      "default"
    ],
    "ansible_net_hostname": "vmx1",
    "ansible_net_model": "vmx",
    "ansible_net_python_version": "3.11.6",
    "ansible_net_serialnum": "VM65311E62C8",
    "ansible_net_system": "junos",
    "ansible_net_version": "18.2R1.9",
    "ansible_network_resources": {}
  },
  "changed": false
}
(Lab) → lab1
```

This output should look very similar to the output from the Cisco device. One of the strengths of Ansible is that differences between manufacturers can be abstracted away.

For example, whether it's a Juniper or Cisco device, the hostname is always captured in the fact "ansible_net_hostname".

The Ansible command line

So far, you have run two different commands for gathering facts from a Juniper or Cisco device. In later labs you will see how to write a single task that works with both manufacturers automatically.

Before looking at the final task for this lab, let's look at the structure of the ansible command itself.

```
ansible juniper -m junipernetworks.junos.junos_facts
```

How did ansible know what devices to run that command against? The first argument to the ansible command is the host or group name to apply the command to. In this case, the command is targeting the "juniper" group.

Looking at the second part of the command:

```
-m junipernetworks.junos.junos_facts
```

This tells Ansible which module you want to use. In Ansible, a module is a component that performs an action on the device. There are many modules provided with Ansible, or as part of the community contributed collections known as Ansible Galaxy. At this point, most common IT equipment has an Ansible module available for management.

Where do host or group names come from? Ansible uses an inventory system to configure the hosts and groups that Ansible will work with. Inventory can take the form of simple text files similar to a Windows INI file, YAML files, or be dynamically generated by running a program.

The inventory for each lab is stored in the inventory directory. Try using the cat command on "inventory/routers" to see the inventory for this lab.

```
(Lab) → lab1 cat inventory/routers
[cisco]
r1
[juniper]
r2
(Lab) → lab1
```

Lines that have a name surrounded with square brackets ('[' and ']') start a new group. The members of that group are listed afterwards, one member per line until a new group is declared. So the router, r1, is a member of group cisco, and the router r2 is a member of group juniper.

Nodes, or hosts, can be members of multiple groups.

Task 3 - Add a new inventory group

Before we add a new group to the inventory, it is helpful to learn a new command, ansible-inventory.

The ansible-inventory command outputs the inventory in JSON format. This is helpful for troubleshooting and could also be used to interface with other programs.

```
(Lab) → lab1 ansible-inventory --list
{
  "_meta": {
    "hostvars": {
      "r1": {
        "ansible_connection":
"ansible.netcommon.network_cli",
        "ansible_host": "172.20.20.10",
        "ansible_network_os": "cisco.ios.ios",
        "ansible_password": "admin",
        "ansible_user": "admin"
      },
```

```

        "r2": {
            "ansible_connection": "ansible.netcommon.netconf",
            "ansible_host": "172.20.20.11",
            "ansible_network_os": "junipernetworks.junos.junos",
            "ansible_password": "admin@123",
            "ansible_user": "admin"
        }
    },
    "all": {
        "children": [
            "ungrouped",
            "cisco",
            "juniper"
        ]
    },
    "cisco": {
        "hosts": [
            "r1"
        ]
    },
    "juniper": {
        "hosts": [
            "r2"
        ]
    }
}
(Lab) → lab1

```

Look at the “meta” section at the beginning of the ansible-inventory output. This includes all the inventory information for how ansible will contact each host. In a later lab, you will see where this configuration is stored. Ansible typically connects to network devices over ssh. For Junos it uses netconf over ssh, and for IOS it is a traditional ssh session.

Edit the file `inventory/routers` with nano. Add the 3 lines shown in the screenshot below and save the file.



```
GNU nano 7.2 routers Modified
[cisco]
r1
[juniper]
r2
[allrouters]
r1
r2
```

Add all-routers group to inventory

Now run the `ansible-inventory` command to verify your change.

```
(Lab) → lab1 ansible-inventory --list
{
  "_meta": {
    "hostvars": {
      "r1": {
        "ansible_connection":
"ansible.netcommon.network_cli",
        "ansible_host": "172.20.20.10",
        "ansible_network_os": "cisco.ios.ios",
        "ansible_password": "admin",
        "ansible_user": "admin"
      },
      "r2": {
        "ansible_connection": "ansible.netcommon.netconf",
        "ansible_host": "172.20.20.11",
        "ansible_network_os": "junipernetworks.junos.junos",
        "ansible_password": "admin@123",
        "ansible_user": "admin"
      }
    }
  }
}
```

```

    }
  },
  "all": {
    "children": [
      "ungrouped",
      "cisco",
      "juniper",
      "allrouters"
    ]
  },
  "allrouters": {
    "hosts": [
      "r1",
      "r2"
    ]
  },
  "cisco": {
    "hosts": [
      "r1"
    ]
  },
  "juniper": {
    "hosts": [
      "r2"
    ]
  }
}
(Lab) → lab1

```

There are a few interesting things to notice in this output. First, the allrouters group has been created and has members r1, and r2.

Looking above the allrouters group in the output we see the group "all". This group is predefined by ansible and will always include all hosts regardless of type. There is also a group called ungrouped, any nodes defined in the inventory before the first group are automatically added to this group.

Task 4 - gathering more facts.

The facts that are being gathered are useful, but there is a lot of detail we would like to see that is missing. By default, the facts gathered are limited to basic parameters. This task will show you all the facts that are available for Cisco IOS and Juniper Junos.

```
(Lab) → lab1 cat task4.txt
ansible juniper -m junipernetworks.junos.junos_facts -a
gather_subset=all,\!hardware
ansible cisco -m cisco.ios.ios_facts -a gather_subset=all
(Lab) → lab1
```

Before running these commands, look at what was added to the command line.

```
-a gather_subset=all,\!hardware
-a gather_subset=all
```

The “-a” argument tells Ansible that what follows is an argument to the module being invoked. In the case of the Juniper device, we want to gather all available facts but excluding facts under the hardware category. The VMX router used in this lab does not support the hardware category, only physical devices support that category. The “!” means negation for Ansible, so this says not hardware. The “\” is necessary to tell the Unix shell to treat the “!” as a normal character.

Try one or both of the one-liners in this task.

```
(Lab) → lab1 ansible juniper -m junipernetworks.junos.junos_facts
-a gather_subset=all,\!hardware
r2 | SUCCESS => {
  "ansible_facts": {
    "ansible_net_api": "netconf",
    "ansible_net_config": "## Last changed: 2023-10-19 12:17:32
UTC\nversion 18.2R1.9;\n\nsystem {\n  login {\n    user admin
{\n  uid 2000;\n  class super-user;\n
authentication {\n  encrypted-password
```

```

\"$6$kROaItFW$7DxteQ.7ySpjQBT1xm9EZg8j5avd/
qlaguFg0PQKkHbqtOodvNP95uGdMvMSCEd6SvPCy.RUqiBzalYI3aH3W1\";\n
}\n      }\n      }\n      root-authentication {\n          encrypted-
password
\"$6$qKj1CL8T$0gUJa6QxFloi76qPLpPaW0IfIg0e1JDoCYmifkfDsWo45MMU1z1esU
1z5mLpUNtNfBBdfp6owRO9v6e.XEOvo1\";\n      }\n      host-name vmx1;\n
management-instance;\n      services {\n          ssh;\n
extension-service {\n          request-response {\n
grpc {\n          clear-text {\n
port 57400;\n          }\n          max-
connections 4;\n          }\n          }\n          }\n
netconf {\n          ssh;\n          rfc-compliant;\n          }\n
}\n      syslog {\n          user * {\n          any emergency;\n
}\n          file messages {\n          any notice;\n
authorization info;\n          }\n          file interactive-commands
{\n          interactive-commands any;\n          }\n          }\n}
\nchassis {\n      fpc 0 {\n          pic 0 {\n          number-of-
ports 96;\n          }\n          }\n}\n\ninterfaces {\n      fxp0 {\n
unit 0 {\n          family inet {\n          address
10.0.0.15/24;\n          }\n          }\n          }\n}\n\nrouting-instances
{\n      mgmt_junos {\n          description management-instance;\n
routing-options {\n          static {\n          route
0.0.0.0/0 next-hop 10.0.0.2;\n          }\n          }\n          }\n}\",
    "ansible_net_gather_network_resources": [],
    "ansible_net_gather_subset": [
        "config",
        "interfaces",
        "default"
    ],
    "ansible_net_hostname": "vmx1",
    "ansible_net_interfaces": {
        ".local.": {
            "admin-status": "up",
            "macaddress": "Unspecified",
            "mtu": "Unlimited",
            "oper-status": "up",
            "speed": "Unlimited",
            "type": "Loopback"
        },

```

```

"fxp0": {
  "admin-status": "up",
  "macaddress": "0c:00:75:68:29:00",
  "mtu": "1514",
  "oper-status": "up",
  "speed": "1000mbps",
  "type": "Ethernet"
},
"ge-0/0/0": {
  "admin-status": "up",
  "macaddress": "0c:00:37:37:6a:01",
  "mtu": "1514",
  "oper-status": "up",
  "speed": "1000mbps",
  "type": null
},
"ge-0/0/1": {
  "admin-status": "up",
  "macaddress": "0c:00:a4:c3:86:02",
  "mtu": "1514",
  "oper-status": "up",
  "speed": "1000mbps",
  "type": null
},
"ge-0/0/2": {
  "admin-status": "up",
  "macaddress": "2c:6b:f5:ab:b6:02",
  "mtu": "1514",
  "oper-status": "down",
  "speed": "1000mbps",
  "type": null
},
"ge-0/0/3": {
  "admin-status": "up",
  "macaddress": "2c:6b:f5:ab:b6:03",
  "mtu": "1514",
  "oper-status": "down",
  "speed": "1000mbps",
  "type": null
}

```

```

    },
    "lo0": {
        "admin-status": "up",
        "macaddress": "Unspecified",
        "mtu": "Unlimited",
        "oper-status": "up",
        "speed": "Unspecified",
        "type": "Loopback"
    },
},
"ansible_net_model": "vmx",
"ansible_net_python_version": "3.11.6",
"ansible_net_serialnum": "VM65311E62C8",
"ansible_net_system": "junos",
"ansible_net_version": "18.2R1.9",
"ansible_network_resources": {}
},
"changed": false
}
(Lab) → lab1

```

The output for the Junos device facts has been truncated here. The output you'll receive in the lab is much longer because all possible interfaces are present. In this lab, only interfaces ge-0/0/0 through ge-0/0/4 are actually present on the virtual machine.

Below is the output for r1, the Cisco device.

```

(Lab) → lab1 ansible cisco -m cisco.ios.ios_facts -a
gather_subset=all
r1 | SUCCESS => {
    "ansible_facts": {
        "ansible_net_all_ipv4_addresses": [
            "10.0.0.15"
        ],
        "ansible_net_all_ipv6_addresses": [],
        "ansible_net_api": "cliconf",
        "ansible_net_config": "Building configuration...\n\nCurrent
configuration : 6066 bytes\n!\n! Last configuration change at

```

```
12:17:34 UTC Thu Oct 19 2023\n!\\version 17.3\\nservice timestamps  
debug datetime msec\\nservice timestamps log datetime msec\\n! Call-  
home is enabled by Smart-Licensing.\\nservice call-home\\nplatform qfp  
utilization monitor load 80\\nplatform punt-keepalive disable-kernel-  
core\\nplatform console serial\\n!\\nhostname csr1\\n!\\nboot-start-  
marker\\nboot-end-marker\\n!\\n!\\n!\\nno aaa new-model\\n!\\n!\\n!\\n!\\n!  
\\n!\\nip domain name example.com\\n!\\n!\\n!\\nlogin on-success log\\n!\\n!  
\\n!\\n!\\n!\\n!\\nsubscriber templating\\n! \\n! \\n! \\n!  
\\nmultilink bundle-name authenticated\\n!\\n!\\n!\\n!\\n!\\n!\\n!\\n!  
\\n!\\n!\\n!\\n!\\n!\\ncrypto pki trustpoint TP-self-signed-3177340814\\n  
enrollment selfsigned\\n subject-name cn=IOS-Self-Signed-  
Certificate-3177340814\\n revocation-check none\\n rsakeypair TP-self-  
signed-3177340814\\n!\\ncrypto pki trustpoint SLA-TrustPoint\\n  
enrollment pkcs12\\n revocation-check crl\\n!\\n!\\ncrypto pki  
certificate chain TP-self-signed-3177340814\\n certificate self-  
signed 01\\n 30820330 30820218 A0030201 02020101 300D0609 2A864886  
F70D0101 05050030 \\n 31312F30 2D060355 04031326 494F532D 53656C66  
2D536967 6E65642D 43657274 \\n 69666963 6174652D 33313737 33343038  
3134301E 170D3233 31303139 31323137 \\n 34335A17 0D333331 30313831  
23313734 335A3031 312F302D 06035504 03132649 \\n 4F532D53 656C662D  
5369676E 65642D43 65727469 66696361 74652D33 31373733 \\n 34303831  
34308201 22300D06 092A8648 86F70D01 01010500 0382010F 00308201 \\n  
0A028201 0100BD94 8ED680EF 711EE836 E0467793 0D81D0AB 4C8180CA  
8B6E3C8C \\n 39E9D984 444DF681 BACC1527 2E3E6EB0 A8CEDC3B 897C2DA1  
6F8692C1 701D4894 \\n 8C9932E2 AEF2329E 25D0669E 4CC7F126 A9146373  
E86018BF F7A38558 B3D219AE \\n 372BCCC6 D6D7E127 47C88613 3C4B7FCC  
90A13BD1 92623DA9 C1D77ED5 5FF29062 \\n 9A2A3961 DDF8865F A96FFC73  
23BA7E0F F5ECCD9B E24BE747 1D726F12 3B5C811B \\n 4B4F5746 3D1490A7  
6B9544A1 A9424255 AFC59704 ED41243E 313AF194 BC5FE502 \\n D7816561  
3C43E37A 91F88D95 6DE9C7B3 5ACFCB25 1B3D1ABE DEACEBBA E12B404B \\n  
D6FD2D96 B0A36F52 BA13E409 6D10BC8A 2D6C3DCF 61281A49 84D1576B  
D84B3CD1 \\n 93A6FF4E 96D50203 010001A3 53305130 0F060355 1D130101  
FF040530 030101FF \\n 301F0603 551D2304 18301680 145E9FDA 7180F443  
64444774 F4FBCC4F 8A065684 \\n 9F301D06 03551D0E 04160414 5E9FDA71  
80F44364 444774F4 FBCC4F8A 0656849F \\n 300D0609 2A864886 F70D0101  
05050003 82010100 2D5AEA9A 0EBD51D2 FB018177 \\n DF7E7D8B 68F18839  
63D91B85 EC697604 64402BB0 C55DC765 9182A6B2 EF47F727 \\n C38D3316  
F4B6A0BD 9EEED3A1 D8C5BCD1 4F35985F A2EE94FC 1A98924A 7495B599 \\n  
AEEAD0C2 675C263C 13011D01 E6FC0214 30CDA9BF 5D20F9DD CA36CCB0
```



```

    "description": null,
    "duplex": "Full",
    "ipv4": [
        {
            "address": "10.0.0.15",
            "subnet": "24"
        }
    ],
    "lineprotocol": "up",
    "macaddress": "0c00.3a37.9700",
    "mediatype": "Virtual",
    "mtu": 1500,
    "operstatus": "up",
    "type": "CSR vNIC"
},
"GigabitEthernet2": {
    "bandwidth": 1000000,
    "description": null,
    "duplex": "Full",
    "ipv4": [],
    "lineprotocol": "down",
    "macaddress": "0c00.dfb1.c701",
    "mediatype": "Virtual",
    "mtu": 1500,
    "operstatus": "administratively down",
    "type": "CSR vNIC"
},
"GigabitEthernet3": {
    "bandwidth": 1000000,
    "description": null,
    "duplex": "Full",
    "ipv4": [],
    "lineprotocol": "down",
    "macaddress": "0c00.806e.5702",
    "mediatype": "Virtual",
    "mtu": 1500,
    "operstatus": "administratively down",
    "type": "CSR vNIC"
}

```

```
    },
    "ansible_net_iostype": "IOS-XE",
    "ansible_net_memfree_mb": 458.6839714050293,
    "ansible_net_memtotal_mb": 698.7988090515137,
    "ansible_net_model": "CSR1000V",
    "ansible_net_neighbors": {},
    "ansible_net_operatingmode": "autonomous",
    "ansible_net_python_version": "3.11.6",
    "ansible_net_serialnum": "9BNAC028QW1",
    "ansible_net_system": "ios",
    "ansible_net_version": "17.03.01a",
    "ansible_network_resources": {}
  },
  "changed": false
}
(Lab) → lab1
```

Summary

In this lab, you looked at two commands:

- ansible
- ansible-inventory

You should be able to use these commands to verify changes to your inventory, and gather facts from IOS and Junos network devices. You also started learning about JSON.

Lab 2 – Introduction to JSON and YAML

In this lab, you will look at different types of variables, and learn how they are represented in JSON and YAML. Ansible uses JSON internally to represent the data passed to and from modules. YAML is used as the format for input to Ansible, i.e., Playbooks and Inventories.

Variable types

In Ansible there are three types of variables commonly used. Scalar variables (simple variables) hold a single value, usually a string. Lists and Dictionaries are collections that hold multiple values at the same time.

The difference between Lists and Dictionaries is in how you access the value you want. In a list, the values are accessed by a numerical index, for example first, second, third, etc. In a Dictionary, the values are accessed by name.

Lists

Examples of working with Lists in YAML and JSON.

YAML list

```
list_a: [ "hello", "world"]
```

The square brackets indicate a list, and then each value is provided in a comma separated list.

There is an alternative way to represent lists in YAML that is often easier to read, especially in longer lists.

```
list_a:
  - "hello"
  - "world"
```

You may add as many values as needed.

Note the indentation and the use of hyphens. The indentation is not just to make things easy to read, it is an important part of YAML. The indentation level tells Ansible how things are grouped together. Everything indented at the same level under "list_a" is grouped together as part of "list_a".

We can create more complex data structures, for example a list of lists.

```
list_of_lists:
  - [ "a-first_list_first_value", "b-first_list_second_value" ]
  - [ "c-second_list_first_value", "d-second_list_second_value" ]
```

A more compact version of the example above, the names of the values have been shorted to keep it easy to read.

```
list_of_lists: [ ["a","b"], ["c","d"] ]
```

Finally, the most verbose version of this example.

```
list_of_lists:
  - - "a"
    - "b"
  - - "c"
    - "d"
```

There are excellent free websites that offer YAML validation. You simply paste your YAML file into the website, and it will verify the file and help you to correct any errors in syntax. Of course, sensitive data should never be entered into these websites, it is best used as a learning tool.

[YAML Lint](#)

In JSON, lists are represented very similarly, however instead of using indentation to represent groupings curly braces '{' and '}' are used. JSON calls these arrays, but Ansible and YAML call them lists. To be consistent with Ansible this document will call them lists.

The example below recreates the list of lists example from the previous page.

```
list_of_lists: [ ["a","b"], ["c", "d"] ]
```

The syntax is identical to the first YAML example. This is not by accident, JSON files are also valid YAML files. YAML is a superset of JSON.

How do you access values from a list?

Look at the example YAML file below.

```
list_travel:
- "Denver"
- "New York"
- "London"
```

This list represents someone's travel plans for the summer. The first stop is Denver, then New York, and finally London. In Ansible, like almost all programming languages, we start counting from zero, not one.

```
list_travel[0] == "Denver"
list_travel[1] == "New York"
list_travel[2] == "London"
```

In later labs you will see many more examples of this.

Dictionaries

Dictionaries work almost exactly like Lists, the only difference is that instead of indexing the values by a number, we index by an arbitrary string.

Suppose we wanted to make a list of pets and what kind of animal they are. Given the name of a pet, we want to know if they are a dog, a cat, or a horse.

We can do that in YAML like this:

```
pets:  
  ranger: dog  
  aspen: dog  
  maple: horse  
  willow: cat
```

We call the index into the dictionary the key. The format is always “key: value” with the key on the left and the value on the right, with a colon for the separator.

How would we access the pets dictionary to find out what kind of animal aspen is?

```
pets["aspen"] == "dog"
```

Dictionaries can be nested just like lists.

```
pets:  
  aspen: { "gender": "female", "color": "black" }  
  ranger: { "gender": "male", "color": "brown" }
```

Another way to write this using indentation instead of curly braces is shown below.

```
pets:  
  aspen:  
    gender: female
```

```
    color: black
  ranger:
    gender: male
    color: brown
```

String quoting in YAML

The previous example didn't use quotes for the values in the dictionary. Most of the examples before then used quotes around strings. In YAML values are always strings, so if you don't quote your string it will usually work. There is a corner case where the YAML parser will fail to understand your string properly. For example:

```
pets:
  aspen: {{ gender['aspen'] }}
```

The curly braces in this example are from a templating language called Jinja that Ansible uses internally. Jinja is introduced in later labs. For now just note that the double curly braces are used to invoke Jinja.

Why does this example fail? We are really trying to create a string with Jinja, but Ansible's YAML parser can't tell if we are calling Jinja or creating a nested dictionary. When in doubt, add quotes!

The correct way to write this example would be:

```
pets:
  aspen: "{{{ gender['aspen'] }}}"
```

Finally, a quick example of dictionaries in JSON.

```
pets = { "aspen":"dog", "ranger":"dog", "willow":"cat",
  "maple":"horse" }
```


A dictionary is represented as a list of key value pairs surrounded by curly braces.

Ansible Inventory group and host variables

In Lab 1 you looked at the inventory file called routers. The ansible environment you are using has been configured to use the directory "inventory" for all inventory objects.

Within that directory are a few special sub-directories, named "host_vars" and "group_vars".

Login to the lab environment and cd into the lab2 directory.

Run the tree command, as shown in the output below.

```
(Lab) → lab2 tree
.
├── ansible.cfg
├── ansible.cfg.no_defaults
├── inventory
│   ├── group_vars
│   │   ├── cisco.yml
│   │   └── juniper.yml
│   ├── host_vars
│   │   ├── r1.yml
│   │   └── r2.yml
│   └── routers
```

```
4 directories, 7 files
```

```
(Lab) → lab2
```

The `tree` command nicely displays all of the files and subdirectories, by default from the current working directory. You can also do things like “`tree <some path>`” to see the tree starting from the specified path.

Under the inventory directory, you see two sub-directories `host_vars` and `group_vars`. These directories have special properties.

Host_vars

Inside the `host_vars` directory you can define additional variables for a given host. The format of the files are “`ansible_host_name.yml`”. So any variables defined in the file “`r1.yml`” are automatically associated with the host named `r1`.

It is critical to be careful with case on the file name. On some systems like Linux the file system is case sensitive. macOS is by default not case sensitive. If the case in the inventory file doesn’t match the case in the file name the variables will not be associated. Developing on a Mac and deploying on Linux is a great way to learn this lesson, the hard way.

Take a look at the `r1` and `r2` host variables.

```
(Lab) → lab2 cd inventory
(Lab) → inventory cat host_vars/r1.yml
---
ansible_host: 172.20.20.10
(Lab) → inventory cat host_vars/r2.yml
---
ansible_host: 172.20.20.11
(Lab) → inventory
```

The variable `ansible_host` defines the hostname or IP address used to contact that host. Your IP addresses will be different depending on which student account you are using.

Group_vars

The `group_vars` directory works the same way as the `host_vars` directory, but the variables are defined at the group level instead of the host level.

All hosts that belong to a group will inherit the variables defined for that group. This is a very convenient way to specify variables that are in common for many hosts. In this lab, the Juniper and Cisco devices each need different variables defined in order to make connectivity work.

Take a look at the cisco and juniper group variables.

```
(Lab) → inventory cat group_vars/cisco.yml
---
ansible_connection: ansible.netcommon.network_cli
ansible_network_os: cisco.ios.ios
ansible_user: admin
ansible_password: admin
(Lab) → inventory cat group_vars/juniper.yml
---
ansible_connection: ansible.netcommon.netconf
ansible_network_os: junipernetworks.junos.junos
ansible_user: admin
ansible_password: admin@123
(Lab) → inventory
```

All the variables here are specific to Ansible's connection to the Cisco and Juniper devices.

The `ansible_connection` variable defines the method used for communication with the device. Cisco IOS uses the standard Cisco CLI over SSH, whereas Juniper devices run Netconf over SSH.

The `ansible_network_os` variable defines the type of device to interact with.

Finally, the `ansible_user` and `ansible_password` define the SSH username and password used to access the device. You can also use SSH public keys and SSH agents to access the devices, but that is out of scope for this lab.

Summary

This lab looked at ways to present structured data in YAML and JSON. You also learned more about the Ansible inventory system.

Lab 3 – Ansible Playbooks

Ansible Playbooks

In Lab 1 you used one-liners to gather facts from the labs Cisco and Juniper devices. In Lab 2 you learned more about JSON and YAML.

In this lab you will create a playbook to gather facts from a device, and output those facts to the screen with the debug module. You will also learn about the “when” clause to execute tasks only when certain conditions are true.

Structure of a playbook

Playbooks are Ansible’s way for specifying the actions or state required for a given set of hosts. Playbooks are broken down into a set of plays, each play performs a set of tasks on the hosts specified for that play.

The format for a playbook is shown in the YAML document below.

```
---  
- name: Play 1  
  hosts: Hosts or groups for play 1  
  gather_facts: True  
  tasks:  
    - name: Task 1 for Play 1  
      module_name:
```

```

        module_option_1: value
        module_option_2: value
- name: Task 2 for Play 1
  module_name:
    module_option_1: value
- name: Play 2
  hosts: Hosts or groups for play 2
  gather_facts: False
  tasks:
- name: Task 1 for Play 2
  module_name:
    module_option_1: value
- name: Task 2 for Play 2
  module_name:
    module_option_1: value

```

Each play can have a different set of hosts that it works on. You can have lists of hosts or groups and logic to combine groups or hosts.

```

---
- name: Examples 1 all hosts group
  hosts: all
  ...
- name: Example 2 all hosts but exclude r2
  hosts: all,!r2
  ...
- name: Example 3 multiple hosts or groups
  hosts: cisco:juniper
  ...
- name: Example 4 hosts in two groups simultaneously
  hosts: cisco:&denver
- name: Example 5 all hosts whose name starts with switch
  hosts: switch*
  ...
- name: Example 6 all hosts named switch 1 through 7
  hosts: switch[1:7]

```

Look at the examples above. The plays are truncated to highlight the hosts portion of the definition. Multiple hosts or groups can be listed. They are separated by either a colon ':' or a comma ','.

In Example 2 the all hosts builtin group is selected. Then host r2 is excluded by saying "!r2". Remember that '!' means not in Ansible.

In Example 3 two groups are selected, cisco and juniper. Any host that belongs to the group cisco *OR* the group juniper will be included.

In Example 4 the opposite logic is employed. The ampersand '&' character means logical and. So any host that belongs to the group cisco *AND* the group Denver will be selected for this play.

Examples 5 and 6 demonstrate using simple regular expressions to pick multiple hosts or groups at once. The asterisks '*' character means match anything. So switch* will match any host or group whose name begins with switch. The [1:7] means match the range of values 1 through 7. Any host or group whose name starts with the word switch followed by a number 1 through 7 will match. It must be exactly in that format, "switch-7" will not match because the hyphen character was not included in the expression.

You can create arbitrary long selectors for the hosts. For example

```
---
- name: Long host selector example
  hosts: cisco:&denver:!r7
```

This example targets all hosts in group cisco that also belong to group Denver, except host r7.

For more details on host selection, see the Ansible documentation linked below.

[Ansible host selection documentation](#)

Disabling fact gathering

The process of gathering facts can be time consuming. On a typical router it might take Ansible 3-5 seconds to gather all of the facts for that device. On a play that targets many devices this can result in very slow playbook runs. You can disable fact gathering when you aren't using facts for a given play.

```
---
- name: A play without fact gathering
  hosts: all
  gather_facts: false
  tasks:
    - name: Task 1
      module_name:
        module_option: value
```

The default value is for `gather_facts` is `True`.

For each play you must include a list of tasks. There is no limit on the number of tasks you include, there must be at least one. Each task can use exactly one module.

Task 1 – Gathering Cisco facts in a playbook

For your first playbook, we will revisit the facts gathering from Lab 1.

In this playbook, you will use the `cisco.ios.ios_facts` module to gather facts from the `cisco` group of routers.

Make sure you are in the `lab3` directory.

```
(Lab) → lab3 pwd
```



```
/home/student1/labs/lab3  
(Lab) → lab3
```

Create the file `cisco_facts.yml` with your preferred editor in the `lab3` directory. The text you need to enter is available in the `task1.txt` file.

```
(Lab) → lab3 cat task1.txt  
Open cisco_facts.yml in your editor.
```

```
nano cisco_facts.yml
```

Add the basic playbook structure to the top of the file.

```
---  
- name: Gather cisco facts  
  hosts: cisco  
  gather_facts: false  
  tasks:  
    - name: Invoke the cisco.ios.ios_facts module to gather facts  
      cisco.ios.ios_facts:
```

Now that you've created the `cisco_facts.yml` playbook, you need to run it. Playbooks are run using the `ansible-playbook` command.

```
ansible-playbook <playbook-file-name>
```

Run your playbook now, your output should be similar to that shown below.

```
(Lab) → lab3 ansible-playbook cisco_facts.yml

PLAY [Gather cisco facts] *****

TASK [Invoke the cisco.ios.ios_facts module to gather facts] *****
ok: [r1]

PLAY RECAP *****
r1 : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0

(Lab) → lab3
```

Looking at the screenshot above you should recognize the text you entered for the name of the Play and the name of the Task.

Under each task, you will see the status for each host that task executed on.

```
TASK [Invoke the cisco.ios.ios_facts module to gather facts]
*****
ok: [r1]
```

In this case, only r1 is in the group cisco.

The play recap reports the overall status of the play. If anything was changed on a host, you will see changed=X where X is the number of hosts that changes were made to. Similarly, any hosts that had failures will be counted under failures=X.

```
PLAY RECAP
*****
r1 : ok=1 changed=0 unreachable=0
failed=0 skipped=0 rescued=0 ignored=0
```

Play results are color coded. Green means success, Red means failure, and Orange means something was changed.

Making playbooks into executable files

It is often convenient to make your playbooks into executable files that you can run directly instead of typing the `ansible-playbook` command. The procedure for this is two steps long.

- Add `#!/usr/bin/env ansible-playbook` as the first line of the file.
- Set the permissions of the file to be executable `chmod 755 file_name`
- Run the playbook with `./playbook-file-name`

```
(Lab) → lab3 chmod 755 cisco_facts.yml
(Lab) → lab3 ./cisco_facts.yml

PLAY [Gather cisco facts] *****
TASK [Invoke the cisco.ios.ios_facts module to gather facts] *****
ok: [r1]

PLAY RECAP *****
r1 : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0

(Lab) → lab3 █
```

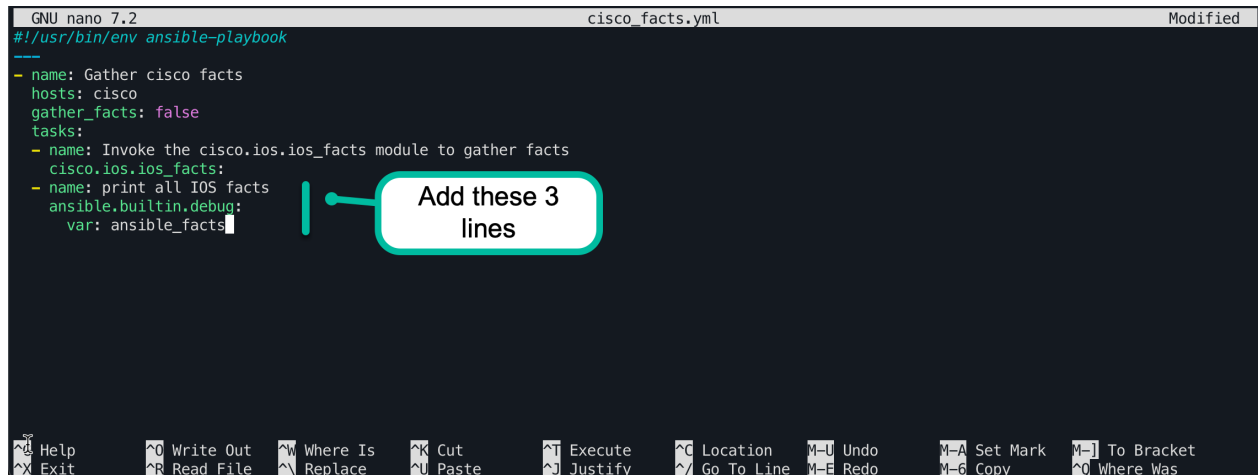
Set playbook permissions

Our playbook ran successfully, but what information did it actually gather?

Task 2 – Using the debug module to print out a variable at runtime

Ansible's debug module can be very useful for printing out values while a playbook is executing. The debug module will be invoked as a second task within our play.

You can cat task2.txt in the lab environment to see what lines of text to add to your playbook. After you've added the new task, run your playbook again.

A screenshot of a terminal window showing the GNU nano 7.2 editor editing a file named cisco_facts.yml. The file content is a YAML playbook with two tasks. The first task is 'Gather cisco facts' and the second is 'Invoke the cisco.ios.ios_facts module to gather facts'. A callout box with a green border and a pointer indicates that three lines should be added to the second task. The lines are: 'ansible.builtin.debug:', ' ansible_facts:', and ' var: ansible_facts'. The bottom of the screen shows the nano editor's command palette with various shortcuts like Help, Write Out, Where Is, Cut, Execute, Location, Undo, Set Mark, To Bracket, Exit, Read File, Replace, Paste, Justify, Go To Line, Redo, Copy, and Where Was.

```
GNU nano 7.2 cisco_facts.yml Modified
#!/usr/bin/env ansible-playbook

- name: Gather cisco facts
  hosts: cisco
  gather_facts: false
  tasks:
    - name: Invoke the cisco.ios.ios_facts module to gather facts
      cisco.ios.ios_facts:
    - name: print all IOS facts
      ansible.builtin.debug:
        var: ansible_facts
```

Add debug module to cisco_facts

How does this new task work?

The module being invoked is "ansible.builtin.debug". This module takes one argument, "var: name-of-variable-to-display", the key "var". The value for this key is the name of the variable to be printed. Here selecting "ansible_facts" will cause all of the gathered ansible facts to be printed.

The output from your playbook run should be similar to that shown below:

```
(Lab) → lab3 ./cisco_facts.yml
```

```
PLAY [Gather cisco facts]
*****
*****

TASK [Invoke the cisco.ios.ios_facts module to gather facts]
*****
*****
ok: [r1]
```

```

TASK [print all IOS facts]
*****
*****
ok: [r1] => {
  "ansible_facts": {
    "net_api": "cliconf",
    "net_gather_network_resources": [],
    "net_gather_subset": [
      "default"
    ],
    "net_hostname": "csr1",
    "net_image": "bootflash:packages.conf",
    "net_iostype": "IOS-XE",
    "net_model": "CSR1000V",
    "net_operatingmode": "autonomous",
    "net_python_version": "3.11.6",
    "net_serialnum": "9BNAC028QW1",
    "net_system": "ios",
    "net_version": "17.03.01a",
    "network_resources": {}
  }
}

PLAY RECAP
*****
*****
r1                                : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0

(Lab) → lab3

```

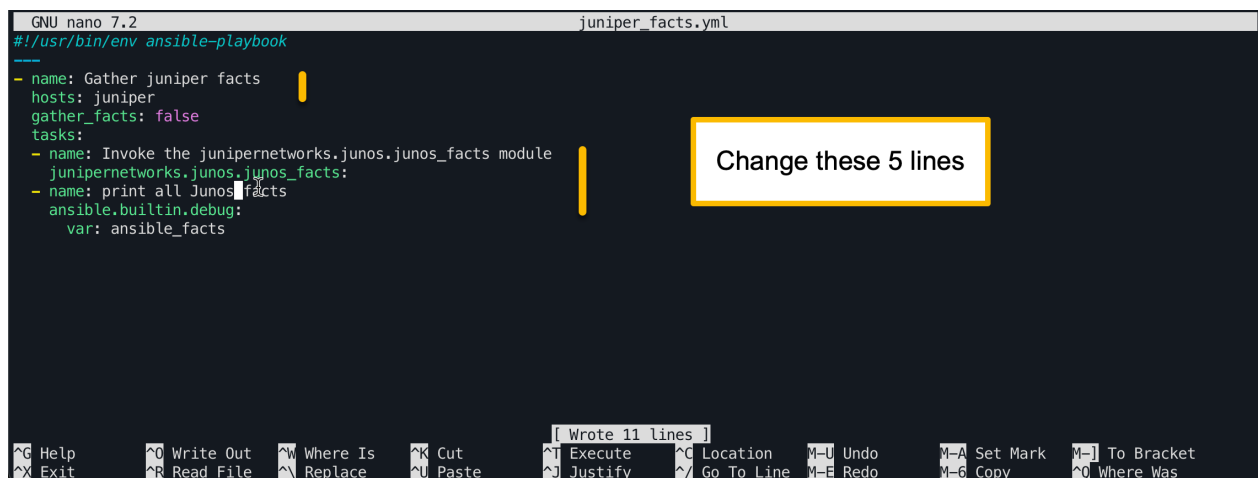
Task 3 – Juniper facts playbook

In this task you will create a new playbook with the same functionality as `cisco_facts.yml`, but aimed at Juniper devices.

First copy `cisco_facts.yml` to `juniper_facts.yml`

```
(Lab) → lab3 cp cisco_facts.yml juniper_facts.yml
(Lab) → lab3
```

You can find the changes you need to make by using the `cat` command with `task3.txt`



```
GNU nano 7.2 juniper_facts.yml
#!/usr/bin/env ansible-playbook
---
- name: Gather juniper facts
  hosts: juniper
  gather_facts: false
  tasks:
    - name: Invoke the junipernetworks.junos.junos_facts module
      junipernetworks.junos.junos_facts:
    - name: print all Junos facts
      ansible.builtin.debug:
        var: ansible_facts
```

Change these 5 lines

Junos facts playbook

When you run the new playbook you should see output similar to this:

```
(Lab) → lab3 chmod 755 juniper_facts.yml
(Lab) → lab3 ./juniper_facts.yml
```

```
PLAY [Gather juniper facts]
*****
*****

TASK [Invoke the junipernetworks.junos.junos_facts module]
*****
*****
```

```

ok: [r2]

TASK [print all Junos facts]
*****
*****

ok: [r2] => {
    "ansible_facts": {
        "net_api": "netconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [
            "default"
        ],
        "net_hostname": "vmx1",
        "net_model": "vmx",
        "net_python_version": "3.11.6",
        "net_serialnum": "VM65311E62C8",
        "net_system": "junos",
        "net_version": "18.2R1.9",
        "network_resources": {}
    }
}

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

r2                                : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0

(Lab) → lab3

```

Task 4 – Playbooks with multiple plays

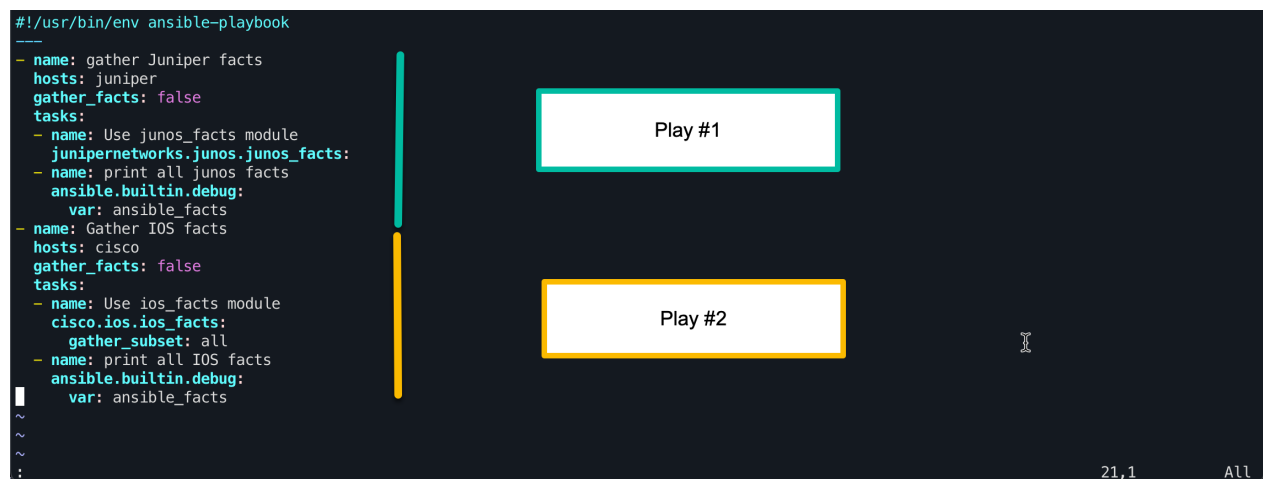
In this task you will combine the `cisco_facts.yml` and `juniper_facts.yml` playbooks into a single playbook with multiple plays.

Create a new playbook called facts.yml. Copy in the content of your cisco_facts.yml and juniper_facts.yml playbooks into this new playbook. Be sure to remove the extra header lines if you copy and paste.

```
#!/usr/bin/env ansible-playbook
---
```

The two lines above should only appear once in your playbook, at the very beginning.

You can cat task4.txt in your shell to see the outline of the task, and for cut and paste.

A screenshot of a terminal window showing an Ansible playbook. The first play is for 'gather Juniper facts' targeting 'juniper' hosts, using the 'junos_facts' module. The second play is for 'Gather IOS facts' targeting 'cisco' hosts, using the 'ios_facts' module. A vertical cyan bar highlights the first play, and a vertical yellow bar highlights the second play. To the right of each bar is a white box with a colored border: a cyan box labeled 'Play #1' and a yellow box labeled 'Play #2'. The terminal shows the playbook content with syntax highlighting. At the bottom right, it says '21,1 All'.

Two plays in a playbook

Run your playbook, you should have output similar to that below.

```
(Lab) → lab3 ./facts.yml
```

```
PLAY [gather Juniper facts]
*****
*****

TASK [Use junos_facts module]
*****
*****
```



```

ok: [r2]

TASK [print all junos facts]
*****
*****

ok: [r2] => {
    "ansible_facts": {
        "net_api": "netconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [
            "default"
        ],
        "net_hostname": "vmx1",
        "net_model": "vmx",
        "net_python_version": "3.11.6",
        "net_serialnum": "VM65311E62C8",
        "net_system": "junos",
        "net_version": "18.2R1.9",
        "network_resources": {}
    }
}

PLAY [Gather IOS facts]
*****
*****

TASK [Use ios_facts module]
*****
*****

ok: [r1]

TASK [print all IOS facts]
*****
*****

ok: [r1] => {
    "ansible_facts": {
        "net_api": "cliconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [

```

```

        "default"
    ],
    "net_hostname": "csr1",
    "net_image": "bootflash:packages.conf",
    "net_iostype": "IOS-XE",
    "net_model": "CSR1000V",
    "net_operatingmode": "autonomous",
    "net_python_version": "3.11.6",
    "net_serialnum": "9BNAC028QW1",
    "net_system": "ios",
    "net_version": "17.03.01a",
    "network_resources": {}
}

```

PLAY RECAP

```

*****
*****
r1                                     : ok=2    changed=0    unreachable=0
failed=0      skipped=0      rescued=0      ignored=0
r2                                     : ok=2    changed=0    unreachable=0
failed=0      skipped=0      rescued=0      ignored=0

```

(Lab) → lab3

Task 5 – introducing the when statement

Often in automation it is necessary to only perform a task under certain conditions that are determined at run time. Ansible has a conditional logic statement, “when”, that is used for this purpose.

Look at the example playbook below.

```

---
- name: Playbook with conditional logic
  hosts: all # We will use when to only target Juniper devices
  tasks:
    - name: Do something junos specific
      junipernetworks.junos.junos_facts:
        when: ansible_network_os == "junipernetwork.junos.junos"

```

This says to execute the `junos_facts` module when the variable `ansible_network_os` is equal to the string `"junipernetworks.junos.junos"`. When the comparison fails, `ansible_network_os` is not equal to `"junipernetworks.junos.junos"`, then this task is skipped. You will see that in the output of the `ansible-playbook` command.

The `ansible_network_os` variable is a predefined per host variable. In this lab, we configure this variable using group variables as explained at the end of Lab 2.

Another special variable that ansible predefines is `"inventory_hostname"`. This variable is always equal to the hostname the current task is executing against. Another way we could have written this check is shown below. This time the `inventory_hostname` variable is checked for presence in the group `juniper`.

```

---
- name: Playbook with conditional logic
  hosts: all # We will use when to only target Juniper devices
  tasks:
    - name: Do something junos specific
      junipernetworks.junos.junos_facts:
        when: inventory_hostname in group['juniper']

```

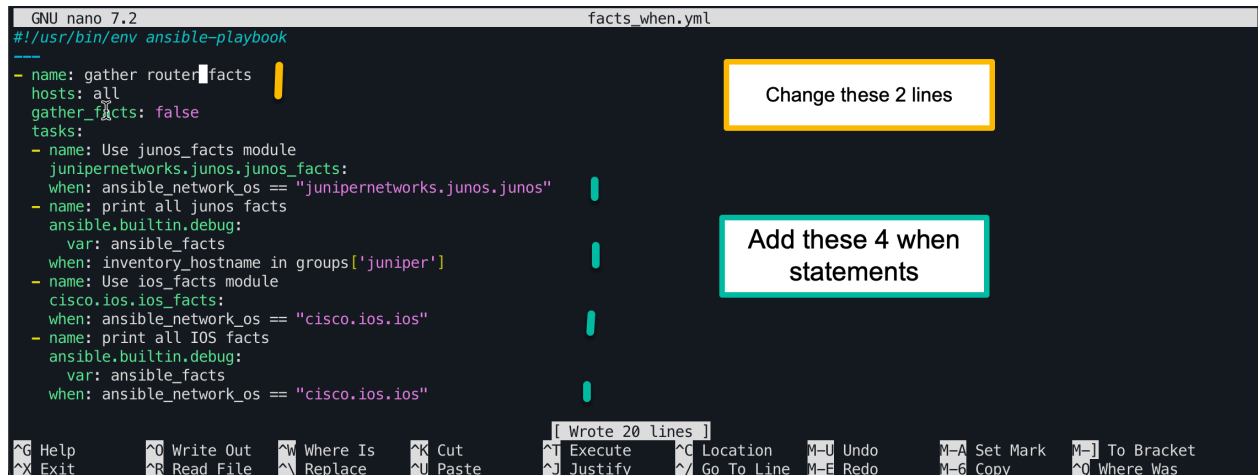
Ansible defines a large number of special variables. The complete list is found in the website below.

[Ansible special variables website](#)

As the final step in this task, rewrite the playbook from task 4 to operate across all hosts. Add conditional logic using when so junos_facts module is called only on Juniper devices. Similarly, only ios_facts is called only on IOS devices.

Name your new playbook facts_when.yml.

You can cat task5.txt to see the final playbook. The changes are shown below.



The screenshot shows a nano 7.2 editor window titled 'facts_when.yml'. The content is an Ansible playbook with the following structure:

```
#!/usr/bin/env ansible-playbook

- name: gather router facts
  hosts: all
  gather_facts: false
  tasks:
    - name: Use junos_facts module
      junipernetworks.junos.junos_facts:
        when: ansible_network_os == "junipernetworks.junos.junos"
    - name: print all junos facts
      ansible.builtin.debug:
        var: ansible_facts
        when: inventory_hostname in groups['juniper']
    - name: Use ios_facts module
      cisco.ios.ios_facts:
        when: ansible_network_os == "cisco.ios.ios"
    - name: print all IOS facts
      ansible.builtin.debug:
        var: ansible_facts
        when: ansible_network_os == "cisco.ios.ios"
```

Annotations on the image:

- A yellow box highlights the first two lines of the tasks section with the text "Change these 2 lines".
- A green box highlights the four 'when' statements with the text "Add these 4 when statements".
- A status bar at the bottom indicates "[Wrote 20 lines]".
- The bottom of the window shows standard nano editor shortcuts like ^C Help, ^O Write Out, etc.

When statements

You can use either style of when statement to choose the correct host type.

When you run your playbook you should see output similar to that below. Note that two tasks were skipped for each host because the when statement was false. I.e. the junos tasks were skipped when executing against the cisco devices.

(Lab) → lab3 ./facts_when.yml

```
PLAY [gather router facts]
*****
*****

TASK [Use junos_facts module]
*****
*****

skipping: [r1]
```

```

ok: [r2]

TASK [print all junos facts]
*****
*****
skipping: [r1]
ok: [r2] => {
    "ansible_facts": {
        "net_api": "netconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [
            "default"
        ],
        "net_hostname": "vmx1",
        "net_model": "vmx",
        "net_python_version": "3.11.6",
        "net_serialnum": "VM65311E62C8",
        "net_system": "junos",
        "net_version": "18.2R1.9",
        "network_resources": {}
    }
}

TASK [Use ios_facts module]
*****
*****
skipping: [r2]
ok: [r1]

TASK [print all IOS facts]
*****
*****
skipping: [r2]
ok: [r1] => {
    "ansible_facts": {
        "net_api": "cliconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [
            "default"
        ]
    }
}

```

```

    ],
    "net_hostname": "csr1",
    "net_image": "bootflash:packages.conf",
    "net_iostype": "IOS-XE",
    "net_model": "CSR1000V",
    "net_operatingmode": "autonomous",
    "net_python_version": "3.11.6",
    "net_serialnum": "9BNAC028QW1",
    "net_system": "ios",
    "net_version": "17.03.01a",
    "network_resources": {}
  }
}

PLAY RECAP
*****
*****
r1                                : ok=2    changed=0    unreachable=0
failed=0      skipped=2      rescued=0    ignored=0
r2                                : ok=2    changed=0    unreachable=0
failed=0      skipped=2      rescued=0    ignored=0

(Lab) → lab3

```

Summary

You learned the basic structure of Ansible playbooks. Remember it is a YAML document, and it is built as a list of plays. Each play can be executed against different hosts as needed. Each play consists of one or more tasks to be performed against the selected hosts.

You also saw how to execute tasks only when certain conditions are met using the `when` statement.

Lab 5 – Configuration backup and restore

In this Lab you will be introduced to the network configuration module, `junos_config` and `ios_config`. There is a configuration module like this for most network devices. This module allows you to backup and restore configurations, and apply native configuration file segments to a device.

In this lab you will look at two different approaches to dealing with variables while backing up a device. In both cases the goal is to include the current date in the name of the backup file. In one case you will use the `delegate_to` statement to force a task to be run on the Linux server instead of the network device. This will be the Cisco backup playbook.

The second approach is to access the facts for a different host, again the Linux server, while executing tasks against the network device. This is the approach taken in the Juniper backup playbook.

Finally, you will create a playbook to restore the configuration of a device from the backups taken. This task will introduce gathering input from the user at run time.

Task 1 and 2 – Backing up Cisco devices

In this task you will learn how to use the `delegate_to` statement. This statement can be applied to individual tasks, much like the `when` statement you looked at in Lab 3.

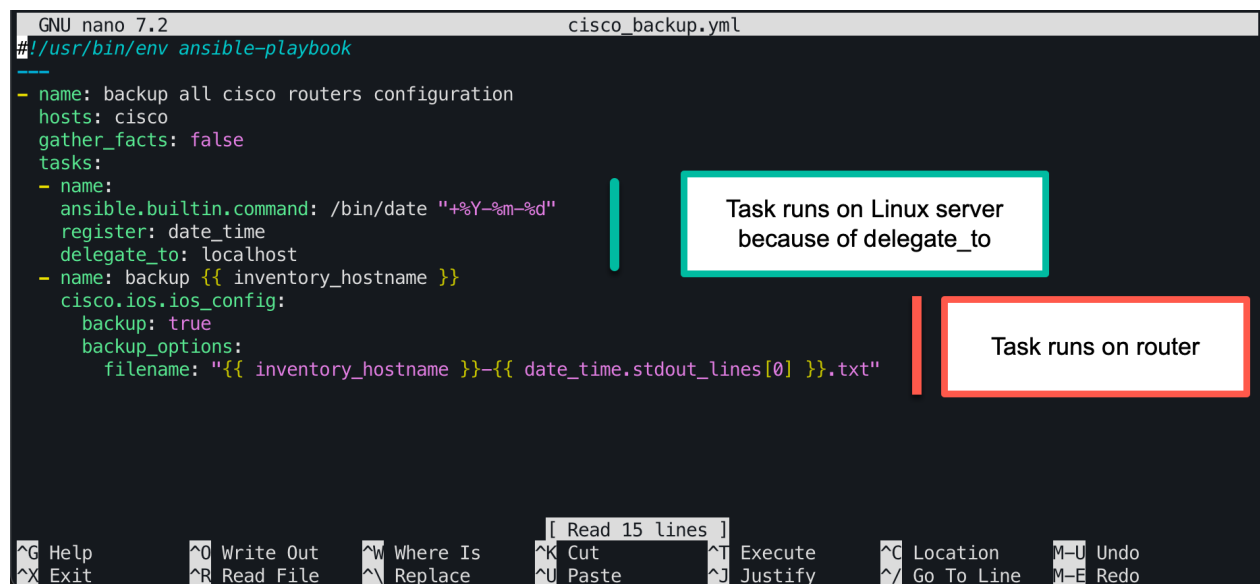
When you use `delegate_to` on a task, that task is performed on the host named with the statement, overriding the host selection for that play. This is commonly used to perform a task on the Ansible server within the same play that is executing on a network device.

Make sure you are in the `lab5` directory.

```
cd ~/labs/lab5
```

Use the `cat` command to view `task1.txt`.

Look at the screenshot of `cisco_backup.yml`.



The screenshot shows a terminal window with the GNU nano 7.2 editor open to the file `cisco_backup.yml`. The file contains an Ansible playbook with two tasks. The first task is highlighted in green, and the second task is highlighted in red. Two callout boxes provide additional context: a green box points to the `delegate_to: localhost` line in the first task, stating 'Task runs on Linux server because of delegate_to', and a red box points to the `cisco.ios.ios_config` block in the second task, stating 'Task runs on router'. The bottom of the screen shows the nano editor's command palette with various shortcuts like Help, Exit, Write Out, Read File, Where Is, Replace, Cut, Paste, Execute, Justify, Location, Go To Line, Undo, and Redo.

```
GNU nano 7.2 cisco_backup.yml
#!/usr/bin/env ansible-playbook
---
- name: backup all cisco routers configuration
  hosts: cisco
  gather_facts: false
  tasks:
  - name:
    ansible.builtin.command: /bin/date "+%Y-%m-%d"
    register: date_time
    delegate_to: localhost
  - name: backup {{ inventory_hostname }}
    cisco.ios.ios_config:
      backup: true
      backup_options:
        filename: "{{ inventory_hostname }}-{{ date_time.stdout_lines[0] }}.txt"
```

Cisco backup playbook

Start with the first task, highlighted in green above.

The `ansible.builtin.command` module is used on Linux/Unix devices to execute arbitrary commands in the shell. In this case the command is `/bin/date` with arguments of `" +%Y-%m-%d"`. What this command does is print the current date, in the format YEAR-MONTH-DATE.

Next we use the "register" statement. The register statement causes the output from the module to be captured into a variable. You may select whatever name you want for the variable, in this case `date_time` is used.

Finally, the `delegate_to` statement causes the `ansible.builtin.command` module to execute against localhost instead of the network device. Localhost is a predefined inventory host in Ansible, and is always equal to the host ansible is running on.

Now look at the second task, highlighted in red above.

This task invokes the `cisco.ios.ios_config` module. This module is used to access the running configuration, and alter the configuration.

The `backup` argument causes the module to write a backup of the existing configuration before making any changes to the device. Since no changes are specified in this playbook, only the backup is performed. By default the backup file is saved in a directory called `backup`, that is created in the same directory that the playbook is run from.

Finally, we override the name of the backup file using `inventory_hostname` and `date_time` variables.

It is worth taking time to carefully understand the structure of the `date_time` variable.

```
filename: "{{ inventory_hostname }}-  
{{ date_time.stdout_lines[0] }}.txt"
```

How do you know ahead of time what structure will be present in the registered variable. It depends on the return values of the module you invoked. The best way to see the structure is to use the `debug` module to print out the captured date.

Look at `task2.txt` using the `cat` command.

```
(Lab) → lab5 cat task2.yml
```

Create a playbook called `register.yml` with the following content.

```
#!/usr/bin/env ansible-playbook
---
- name: explore register variable structure.
  hosts: localhost
  gather_facts: false
  tasks:
    - name:
      ansible.builtin.command: /bin/date "+%Y-%m-%d"
      register: date_time
    - name: display date_time structure
      ansible.builtin.debug:
        var: date_time
(Lab) → lab5
```

Run the playbook and look at the JSON output.

```
(Lab) → lab5 ./register.yml
```

```
PLAY [explore register variable structure.]
*****
***

TASK [ansible.builtin.command]
*****
*****
changed: [localhost]

TASK [display date_time structure]
*****
*****
ok: [localhost] => {
  "date_time": {
    "changed": true,
    "cmd": [
      "/bin/date",
      "+%Y-%m-%d"
    ]
  }
}
```

```

    ],
    "delta": "0:00:00.002202",
    "end": "2023-10-19 16:19:43.513438",
    "failed": false,
    "msg": "",
    "rc": 0,
    "start": "2023-10-19 16:19:43.511236",
    "stderr": "",
    "stderr_lines": [],
    "stdout": "2023-10-19",
    "stdout_lines": [
        "2023-10-19"
    ]
}
}

PLAY RECAP
*****
*****
localhost           : ok=2    changed=1    unreachable=0
failed=0      skipped=0      rescued=0      ignored=0

(Lab) → lab5

```

Looking at the returned JSON structure, you'll see there are two keys that have the output we are looking for. The "stdout" key will certainly work. If the command had returned multiple lines of output, the "stdout_lines" key is more useful.

Look again at the last line of the `cisco_backup.yml` playbook.

```

    filename: "{{ inventory_hostname }}-
    {{ date_time.stdout_lines[0] }}.txt"

```

Now that you have seen the structure of the data captured in the registered variable it should make more sense. This Jinja template string will create a name of the form hostname-date.

Finally, run the `cisco_backup.yml` playbook.

```
(Lab) → lab5 ./cisco_backup.yml
PLAY [backup all cisco routers configuration] *****
TASK [ansible.builtin.command] *****
changed: [r1 -> localhost]
TASK [backup r1] *****
changed: [r1]
PLAY RECAP *****
r1 : ok=2 changed=2 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0

(Lab) → lab5
```

Notice the output says two devices were changed and two devices completed successfully. Why two devices? The first device is the localhost when the `date` command is run. The second device is the actual router, `r1`.

Why is `r1` changed? The router itself was not changed, but the backup file was created. The `ios_config` module reports that a change was made because it created the backup file.

Finally, use the `tree` command to see the backup file. Use the `more` command to view the contents of the file and verify it is a Cisco IOS configuration file.

```
(Lab) → lab5 tree backup
backup
├── r1-2023-10-19.txt
```

```
1 directory, 1 file
```

```
(Lab) → lab5
```

```
(Lab) → lab5 cat backup/r1-2023-10-19.txt | head -10
Building configuration...
```

Current configuration : 6066 bytes

!

! Last configuration change at 12:17:34 UTC Thu Oct 19 2023

!

version 17.3

service timestamps debug datetime msec

service timestamps log datetime msec

! Call-home is enabled by Smart-Licensing.

Task 3 – Backing up Juniper devices

You could certainly use the same playbook, simply adapting the tasks to use the `junos_config` module instead of `ios_config`.

The goal for this task is to look at a different way to access host variables. Instead of delegating tasks to `localhost`, we run a play against `localhost` to gather the date. Then in a second play the data gathered from the first play is used to populate the file-name for the backup task. This approach is far more efficient since the date command is only run once.

Look at task3 using the cat command.

```
(Lab) → lab5 cat task3.yml
Create a playbook named juniper_backup.yml.
```

The contents of the playbook should be:

```
#!/usr/bin/env ansible-playbook
---
- name: gather date and time
  hosts: localhost
  gather_facts: false
  tasks:
    - name:
      ansible.builtin.command: /bin/date "+%Y-%m-%d"
      register: date_time
- name: backup all juniper routers configuration
  hosts: juniper
  gather_facts: false
```

```

tasks:
- name: backup {{ inventory_hostname }}
  junipernetworks.junos.junos_config:
    backup: true
    backup_options:
      filename: "{{ inventory_hostname }}-{{ hostvars['localhost']
['date_time']['stdout_lines'][0] }}.txt"

```

The special variable `hostvars` is a dictionary that can be used to access any variable that has been defined during execution of the playbook. Plays are run in the order that are listed in the playbook, if you define the variable in the 3rd play of a playbook, it will exist for any plays after the 3rd play (ie play 4, play 5, etc.) Plays 1 and 2 in this case would not have access to the variable.

Run the `juniper_backup.yml` playbook, and verify the contents of the backup.

```

(Lab) → lab5 cat backup/r2-2023-10-19.txt | head -10
set version 18.2R1.9
set system login user admin uid 2000
set system login user admin class super-user
set system login user admin authentication encrypted-password
"$6$kROaItFW$7DxteQ.7ySpjQBT1xm9EZg8j5avd/
qlaguFg0PQKkHbqtOodvNP95uGdMvMSCEd6SvPCy.RUqiBzalyI3aH3W1"
set system root-authentication encrypted-password
"$6$qKj1CL8T$0gUJa6QxFloi76qPLpPaW0IfIg0e1JDoCYmifkFDsWo45MMU1z1esU1
z5mLpUNtNfBBdfp6owRO9v6e.XEOvol"
set system host-name vmx1
set system management-instance
set system services ssh
set system services extension-service request-response grpc clear-
text port 57400
set system services extension-service request-response grpc max-
connections 4

```

Task 4 – Combined Juniper and Cisco backup playbook

In this task you will combine the `cisco_facts` and `juniper_facts` playbooks into a single playbook called `backup.yml`. This playbook will use `when` statements to ensure tasks are run against the correct host.

Look at `task4.txt` using the `cat` command.

```
(Lab) → lab5 cat task4.txt
```

Create `backup.yml` combining the tasks from `cisco_backup.yml` and `juniper_backup.yml`.

You can use either style of capturing the date, that you prefer.

Your playbook should look similar to this:

```
---
- name: gather date and time
  hosts: localhost
  gather_facts: false
  tasks:
    - name:
      ansible.builtin.command: /bin/date "+%Y-%m-%d"
      register: date_time
- name: backup all routers configuration
  hosts: all
  gather_facts: false
  tasks:
    - name: backup {{ inventory_hostname }}
      junipernetworks.junos.junos_config:
        backup: true
        backup_options:
          filename: "{{ inventory_hostname }}-{{ hostvars['localhost']
['date_time']['stdout_lines'][0] }}.txt"
      when: inventory_hostname in groups['juniper']
    - name: backup {{ inventory_hostname }}
      cisco.ios.ios_config:
```



```
    backup: true
    backup_options:
        filename: "{{ inventory_hostname }}-{{ hostvars['localhost']
['date_time']['stdout_lines'][0] }}.txt"
    when: inventory_hostname in groups['cisco']
```

Run your new playbook to ensure it works. You should notice that the backup tasks did not create new files because the configuration of the devices hasn't changed and a backup with today's date already exists.

Task5 – Change the system hostname

In this task you will make a change to one of the routers hostname. In the next task, you will write a playbook to restore the configuration based on the backups taken in previous tasks.

Use the cat command to look at task5.txt.

```
(Lab) → lab5 cat task5.txt
Your hostname for r1 is clab-devpod-csr1.
Your hostname for r2 is clab-devpod-vmx1.
The username and password for r1 is admin/admin
The username and password for r2 is admin/admin@123
```

Login to one of the devices, your choice, and make a configuration change to the hostname.

```
For r1:
    config t
    hostname <new hostname>
For r2:
    config
    set system hostname <new hostname>
    commit
```

Your hostname for r1 and r2 will be different than what is shown here. Your task5.txt file will have the correct hostname for your session.

Task 6 – Restoring a configuration

In this task you will write a playbook to revert the configuration change made in Task 5. This task also introduces a new feature of Ansible, the ability to ask for user input before running a task.

Use the cat command to look at task6.txt.

```
(Lab) → lab5 cat task6.txt
Create a new playbook named restore.yml.
```

Your playbook should have content similar to this:

```
#!/usr/bin/env ansible-playbook
---
- name: restore configuration
  hosts: all
  gather_facts: false
  vars_prompt:
    - name: host
      prompt: "Enter the hostname as it appears in the inventory
(R1,R2):"
      private: false
    - name: date
      prompt: "Enter the date (YYYY-MM-DD):"
      private: false
  tasks:
    - name: restore {{ inventory_hostname }}
      junipernetworks.junos.junos_config:
        src: "backup/{{ inventory_hostname }}-{{ date }}.txt"
        update: replace
```

```

    when: inventory_hostname in groups['juniper'] and
inventory_hostname == host
- name: restore {{ inventory_hostname }}
  cisco.ios.ios_config:
    src: "backup/{{ inventory_hostname }}-{{ date }}.txt"
    save_when: changed
  when: inventory_hostname in groups['cisco'] and
inventory_hostname == host

```

This playbook uses the “src” argument to the config modules. This argument provides a text file containing the configuration to be loaded. You can load a full configuration or a partial configuration.

On Junos there is an additional argument, `src_format` where you can specify the type of configuration file, “set”, “xml”, “text”, or “json”. If you don’t specify the format, it will automatically attempt to determine it.

The `update` argument specifies how the update is handled on Junos. By default the configuration file provided is merged with the running or current configuration. For example, on Junos if you load a configuration file that configures an IP address on an interface logical unit, and the update method is set to merge, any IP address already configured on that logical unit will be left in place and the new IP address will be added.

If you set the update method to replace, then the provided configuration file will replace the existing running configuration.

By default Ansible modules on IOS affect only the running configuration of the device. To ensure the changes are saved to the startup configuration, the `ios_config` module has the `save_when` argument. The default value of `save_when` is `never`. You can set it to “never”, “changed”, “modified”, and “always”. Always will cause the module to save the running config to the startup config every time it runs, probably not what you want. Modified will save the running config anytime the running config is different

than the startup config. Finally, changed will save the running config to the startup config anytime Ansible has made a change to the configuration.

In the next lab you will see a different way to handle this, using Ansible handlers.

You will see more options for the config module in the next few labs.

This playbook introduces a new feature for a play, the “vars_prompt” option. This option will tell Ansible to prompt the user.

```
vars_prompt:
- name: host
  prompt: "Enter the hostname ..."
  private: false
```

The name argument provides the name of the variable this prompt will create. In this example, the variable is named host.

The prompt argument provides the text that Ansible will use to prompt the user.

The private argument is used to turn on or off the screen echo. If private is set to true, the whatever the user types will not be echoed back, in other words it will not be visible. This is used for gathering sensitive information like passwords.

After the prompts are run, you can use the new variables exactly like any other variable in Ansible.

In this playbook the use of the when statement has become more complicated.

```
when: inventory_hostname in groups['cisco'] and inventory_hostname
== host
```

This compound statement requires that the inventory_hostname be in the correct group for the task and that the inventory_hostname be equal to the hostname provided by the user. Remember, the conditions that you provide to the when statement are evaluated by Jinja. You can create powerful conditional logic in Jinja.

Run your restore playbook to revert the configuration change made in task 5.

```
(Lab) → lab5 ./restore.yml
Enter the hostname as it appears in the inventory (r1,r2):: r1
Enter the date (YYYY-MM-DD):: 2023-10-20

PLAY [restore configuration] *****

TASK [restore r1] *****
skipping: [r1]
skipping: [r2]

TASK [restore r1] *****
skipping: [r2]
[WARNING]: To ensure idempotency and correct diff the input configuration lines should be similar to how they
appear if present in the running configuration on device including the indentation
changed: [r1]

PLAY RECAP *****
r1                : ok=1    changed=1    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
r2                : ok=0    changed=0    unreachable=0    failed=0    skipped=2    rescued=0    ignored=0

(Lab) → lab5 ssh admin@clab-devpod-csr1
(admin@clab-devpod-csr1) Password:
```

Restore configuration - Idempotency

In the screenshot above, the first 2 lines of output are the vars_prompt gathering the necessary information.

If you chose to work with the Cisco device, you will see a warning in purple text, highlighted in green in the screenshot.

Idempotence

Idempotence is a mathematical word used to describe an operation where repeating that operation has no additional effect. Imagine walking into a dark room and flipping the light switch up to turn on the lights. If you use your finger to flip the switch up again, it has no effect. This is an idempotent operation. In contrast, adding one to number in a variable is not idempotent. Every time you do it the value gets bigger.

Ansible places great importance on making all of its operations idempotent. Running a playbook a second time when no changes have been made to the device should always result in the same state.

In IOS it is possible to have non idempotent operations. If you use abbreviated commands with the `ios_config` module, it will compare the abbreviated command with the running config that has the fully spelled out command. Because the two commands are not spelled character for character the same, Ansible will update the configuration. This will cause Ansible to make a configuration change every time the playbook is run.

A similar issue is possible in Junos if you supply abbreviated set commands in a configuration snippet.

Best practice is to always fully spell out the commands you are providing when using the config modules.

Summary

In this lab you learned how to:

- Prompt users for input
- Delegate tasks to different hosts
- Use the config module to backup and restore configurations
- The idea of idempotent operations

Lab 6 - Setting system configuration

In this lab you will look at two different approaches to setting system configuration. In the first approach you will use the config module that you used in Lab 5. In the second approach you will use task specific modules for IOS and Junos.

Both of these approaches to configuration management are valid. In combination with Jinja templates the configuration module is very powerful and flexible. The downside to the configuration module is that you are still forced to use the native configuration syntax of the device, i.e. set commands in Junos.

The task specific modules cover most of the configuration tasks you will need on a Junos or IOS device. The advantage of the task specific modules is that often they abstract away the configuration syntax from the user. You still need to understand the way the protocol works, but you don't always have to know the exact syntax.

Task 1 – Setting hostnames with junos_config.

In this task you will change the system hostname with the junos_config module. In the next task you will see an alternate way to perform this configuration.

Look at the task1.txt file with the cat command.

```
(Lab) → lab6 cat task1.txt
```

Create a new playbook `junos-hostname.yml`.
This playbook should change the hostname on `r2` using the `junos_config` module.

Your playbook should look like this.

```
#!/usr/bin/env ansible-playbook
---
- name: set hostname configuration
  hosts: r2
  gather_facts: false
  vars:
    hostname: vmx-a
  tasks:
    - name: Set system host-name on {{ inventory_hostname }}
      junipernetworks.junos.junos_config:
        lines:
          - set system host-name {{ hostname }}
```

This playbook introduces a few new things.

First, the `vars:` option for a play. This allows you to define play specific variables and is very useful when you will reuse the same value in multiple tasks. You see on the last line of the playbook that the variable is referenced with `{{ hostname }}`. Each line provided is processed by Jinja before being sent to the device.

In the `junos_config` module we have added the `lines` argument. This argument expects that each line provided will be a set or delete statement. Make sure that you fully spell out each command, otherwise your playbook will not be idempotent!

Run the playbook and verify that the hostname has changed successfully.

Task 2 – Setting hostnames with the junos_hostname module

In this task you will perform the same task as task 1, changing the system hostname on r2. However, this time you will use the junos_hostname module. This module has the advantage of being higher level and hiding all of the Junos specific commands.

Use the cat command to view task2.txt.

```
(Lab) → lab6 cat task2.txt
```

Create a new playbook junos_hostname2.yml.

Use the junos_hostname module to set the hostname back to "r2"

```
#!/usr/bin/env ansible-playbook
---
- name: set hostname configuration
  hosts: r2
  gather_facts: false
  vars:
    hostname: r2
  tasks:
    - name: Set system host-name on {{ inventory_hostname }}
      junipernetworks.junos.junos_hostname:
        config:
          hostname: "{{hostname}}"
```

It is important to make sure the argument to the hostname is quoted on the last line of the playbook. Remember, any time a string starts with a Jinja double curly bracket pair, it must be quoted to avoid a YAML parser error.

Run your playbook and verify that the hostname is back to "r2".

Task 3 – Setting hostname with ios_config module

The ios_config module behaves very similarly to the junos_config module. There are a few additional arguments in ios_config, for example when setting an IP address you specify the parent configuration mode i.e. the interface name. You will see more about this in a later lab.

Use the cat command to view task3.txt.

```
(Lab) → lab6 cat task3.yml
```

Create a new playbook cisco_hostname.yml.

Use the ios_config module to change the system hostname.

Your playbook should look like this:

```
#!/usr/bin/env ansible-playbook
---
- name: set hostname configuration
  hosts: r1
  gather_facts: false
  vars:
    hostname: csr-a
  tasks:
    - name: Set system host-name on {{ inventory_hostname }}
      cisco.ios.ios_config:
        lines:
          - hostname {{ hostname }}
```

Run your playbook and verify the hostname changed correctly.

Task4 – Setting hostname using the ios_hostname module

In this task you will rewrite the playbook from task 3 to use the ios_hostname module. The easiest way to do this is to copy junos_hostname2.yml and change the hostname and module lines.

Use the cat command to view task4.txt.

```
(Lab) → lab6 cat task4.txt
Create a new playbook cisco_hostname2.yml.
Use junos_hostname2.yml as your starting point.
Change the host, hostname, and module name.
```

```
#!/usr/bin/env ansible-playbook
---
- name: set hostname configuration
  hosts: r1
  gather_facts: false
  vars:
    hostname: r1
  tasks:
    - name: Set system host-name on {{ inventory_hostname }}
      cisco.ios.ios_hostname:
        config:
          hostname: "{{hostname}}"
```

Before moving on, take a moment to compare the playbook in task2 and the playbook in task4. With the exception of the module name and the hostname, these two playbooks are identical. Not all task specific modules are as similar as these, but when they are similar it provides an advantage for users.

Run your playbook and verify the hostname is changed back to r1.

Task 5 – Ansible Handlers

As mentioned in Lab 5, the IOS modules in Ansible do not automatically save the running configuration to the startup configuration when a change is made. This is an Ansible design decision.

When you looked at the `ios_config` module, there was an option “`save_when`” that allowed for specifying when the configuration should be saved. The task specific IOS modules do not offer this choice. Another method is needed to ensure the startup configuration matches the running configuration.

Ansible has a feature called Handlers. Handlers run at the end of the play and only when required. How does Ansible know if a handler is required to run? For any tasks that might make a change, the `notify` option is added. If the task makes a change it will notify the handler. Provided at least one task has notified that handler, it will be run at the end of the play.

There can be many handlers attached to a play, and tasks can notify one or more handler as needed.

Here is an example of a handler.

```
---
- name: Playbook with handler
  hosts: r1
  gather_facts: false
  tasks:
    - name: Task 1 - make a change
      cisco.ios.ios_hostname:
        config:
          hostname: crazy-hostname
      notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
```

```
commands:
  - command: copy running-config startup-config
    prompt: 'Destination filename.*'
    answer: startup-config
```

This handler introduces a new module, `ios_command`. There is an equivalent for Junos devices, `junos_command`.

The `ios_command` module allows you to send arbitrary commands to the CLI. If those commands require an interactive response, like the `copy run start` command does, then you supply the answer with the `prompt:` and `answer:` parameters as shown above.

The `prompt:` parameter accepts a regular expression in the prompt string, so you don't have to enter the exact text of the prompt. Just enough to be sure the answer is correct for that question.

If the commands being run do not require answering a prompt, then the `prompt` and `answer` parameters can be omitted. It is also possible to run multiple commands. The example below demonstrates this.

```
cisco.ios.ios_command:
  commands:
  - show inventory
  - show ip interface brief
```

For this task, create a new playbook `cisco_hostname3.yml` that includes the handler to save the running configuration when a change is made.

View the `task5.txt` file for the details, and suggested content of the playbook.

```
(Lab) → lab6 cat task5.txt
Copy the cisco_hostname2.yml playbook to cisco_hostname3.yml.
Add the notify parameter and handler to the new playbook.
Run the playbook with a different hostname and verify change is
saved
```

to the startup configuration.

```
#!/usr/bin/env ansible-playbook
---
- name: set hostname configuration
  hosts: r1
  gather_facts: false
  vars:
    hostname: crazyhostname
  tasks:
    - name: Set system host-name on {{ inventory_hostname }}
      cisco.ios.ios_hostname:
        config:
          hostname: "{{hostname}}"
      notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
        commands:
          - command: copy running-config startup-config
            prompt: 'Destination filename.*'
            answer: 'startup-config'
```

Run your new playbook and verify it worked correctly. Optionally, edit the playbook to restore the old hostname and run it again.

Summary

In this lab you learned how to apply configuration to a device using the vendor specific config module. You also learned how to use a task specific module. Also, you saw that the task specific modules provide identical syntax across vendors.

The ansible handler functionality was introduced along with the `ios_command` module.

Lab 7 – Configuring interface parameters

In this lab, you will look at how to configure an IP address on both a Cisco and Juniper device.

This lab will follow the format from Lab 6. First you will use the vendor-specific configuration module, then you will use the task-specific configuration module.

In this lab, you will configure a new loopback address, and an interface address on each router. Then verify connectivity between the two routers.

Task 1 – Configure IP address on IOS using the config module

In this task, you will configure a loopback address on R1, and an interface address on GigabitEthernet2.

When providing the correct line to the config module, you must also provide the parent line from the configuration. The parent line in this case is the interface <XYZ> line.

For example:

```
interface GigabitEthernet 2.          ! parent line
ip address 10.200.200.1 255.255.255.0
```

Use the cat command to view task1.txt.

```
#!/usr/bin/env ansible-playbook
---
- name: set IP address on R1
  hosts: r1
  gather_facts: false
  vars:
    ip_gig2: 10.200.200.1
    mask_gig2: 255.255.255.0
    ip_lo1: 172.31.1.1
    mask_lo1: 255.255.255.255
  tasks:
    - name: Configure lo1 on r1
      cisco.ios.ios_config:
        lines:
          - ip address {{ ip_lo1 }} {{ mask_lo1 }}
          - no shutdown
        parents:
          - interface Loopback1
      notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
        commands:
          - command: copy running-config startup-config
            prompt: 'Destination filename.*'
            answer: 'startup-config'
```

Run the playbook twice. You should see that both times it reported the configuration was changed. Why was that?

On IOS, Loopback interfaces default to not being shutdown. The “no shutdown” line is not displayed in the running configuration because it’s a default command. When Ansible doesn’t see the “no shutdown” line, it reapplies the configuration. Try commenting out the “no shutdown” line and see if your playbook becomes idempotent. You can comment out a line in YAML by starting it with the pound sign ‘#’.

After you comment out the “no shutdown” line, rerun your playbook. It should make no changes.

Now, add a second task to add the second interface configuration.

```
- name: Configure GigabitEthernet2 on r1
  cisco.ios.ios_config:
    lines:
      - ip address {{ ip_gig2 }} {{ mask_gig2 }}
      - no shutdown
    parents:
      - interface GigabitEthernet2
  notify: update-startup-config
```

Your final playbook should look like this:

```
#!/usr/bin/env ansible-playbook
---
- name: set IP address on R1
  hosts: r1
  gather_facts: false
  vars:
    ip_gig2: 10.200.200.1
    mask_gig2: 255.255.255.0
    ip_lo1: 172.31.1.1
    mask_lo1: 255.255.255.255
  tasks:
    - name: Configure lo1 on r1
      cisco.ios.ios_config:
        lines:
          - ip address {{ ip_lo1 }} {{ mask_lo1 }}
          - no shutdown
        parents:
          - interface Loopback1
      notify: update-startup-config
    - name: Configure GigabitEthernet2 on r1
      cisco.ios.ios_config:
        lines:
```

```

    - ip address {{ ip_gig2 }} {{ mask_gig2 }}
    - no shutdown
  parents:
    - interface GigabitEthernet2
  notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
        commands:
          - command: copy running-config startup-config
            prompt: 'Destination filename.*'
            answer: 'startup-config'

```

Task 2 – Setting IP address using the cisco_interface module

In this task, you will recreate the `cisco_ip.yml` playbook. Instead of using the `ios_config` module, you will use the `ios_l3_interfaces` module. This module is designed to allow all the common interface configuration tasks, while abstracting away the internal IOS syntax. There is also an `ios_l2_interfaces` module for configuring layer 2 features.

Use the `cat` command to view `task2.txt`.

(Lab) → `lab7 cat task2.txt`

Refactor your `cisco_ip.yml` playbook, and name it `cisco_ip2.yml`.

This module will use the `cisco.ios.ios_l3_interfaces` module.

```

#!/usr/bin/env ansible-playbook
---
- name: set IP address on R1
  hosts: r1
  gather_facts: false

```

```

vars:
  ip_gig2: 10.200.200.1
  mask_gig2: 24
  ip_lo1: 172.31.1.1
  mask_lo1: 32
tasks:
- name: Configure interfaces on r1
  cisco.ios.ios_l3_interfaces:
    config:
      - name: Loopback1
        ipv4:
          - address: "{{ ip_lo1 }}/{{ mask_lo1 }}"
            secondary: false
      - name: GigabitEthernet2
        ipv4:
          - address: "{{ ip_gig2 }}/{{ mask_gig2 }}"
            secondary: false
    notify: update-startup-config
handlers:
- name: update-startup-config
  cisco.ios.ios_command:
    commands:
      - command: copy running-config startup-config
        prompt: 'Destination filename.*'
        answer: 'startup-config'
(Lab) → lab7

```

A few important things to note on this module. Interface net masks are given in CIDR notation, not as subnet masks. Notice the mask_gig2 and mask_lo1 variables at the beginning of the play.

```

vars:
  ip_gig2: 10.200.200.1
  mask_gig2: 24
  ip_lo1: 172.31.1.1
  mask_lo1: 32

```

You do not need to add the `secondary: false` option to the IP addresses, that is the default value. It is present here just to demonstrate how secondary addresses could be configured if you set it to true.

Run the `cisco_ip2.yml` playbook and verify your configuration.

Task 3 – The loop statement

Look at the use of the `ios_l3_interfaces` module in Task2. Much of the configuration is repeated, and it would be nice if the code could be written more compactly.

Think about the variables and values for the interface configuration. If you had to write that information down on paper, you would probably make a table. Something like this:

Interface	IP Address	Mask	Secondary
Loopback1	172.31.1.1	32	False
GigabitEthernet2	10.200.200.1	24	False

Now look at the first row of that table. It is precisely what a dictionary records:

```
{ interface: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }
```

The same thing can be said for the second row. Putting this together, the table above can be represented as a list of dictionaries.

```
- { interface: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }  
- { interface: GigabitEthernet2, ip: 10.200.200.1, mask: 24, sec: False }
```

Ansible has a powerful tool for looping when you want to do the same task several times with different values in each time.

```
- name: Configure interfaces on r1
  cisco.ios.ios_l3_interfaces:
    config:
      - name: "{{ item.int }}"
        ipv4:
          - address: "{{ item.ip }}/{{ item.mask }}"
            secondary: "{{ item.sec }}"
    loop:
      - { int: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }
      - { int: GigabitEthernet2, ip: 10.200.200.1, mask: 24, sec:
False }
```

Look at the code above. The first five lines of code invoke the `cisco.ios.ios_l3_interfaces` module. That module is passed parameters `item.int`, `item.ip`, `item.mask`, and `item.sec`.

The variable `item` is a magic variable in Ansible. It represents the value(s) to be used during the loop. The first time through the loop `item` looks like this:

```
item = { int: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }
```

So `item` is a dictionary with four keys: `int`, `ip`, `mask`, and `sec`.

The second time through, the loop `item` looks like this:

```
item = { int: GigabitEthernet2, ip: 10.200.200.1, mask: 24, sec:
False }
```

The loop will run for however many items you provide. In this case, the loop had a list of two dictionaries, so the loop runs twice.

Take care with the indentation here. The loop keyword is indented at the same level as the module being declared, `cisco.ios.ios_l3_interfaces`. The indentation defines what the loop will cover.

Login to r1 and delete the Loopback1 interface.

Use the cat command to view task3.txt.

```
(Lab) → lab7 cat task3.txt
```

In this task you will rewrite the playbook from task2 to be more concise. Name the new playbook `cisco_ip3.yml`
Use the loop statement, and convert the variable declarations into a list of dictionaries as described in the lab guide.

Your playbook should look like this.

```
#!/usr/bin/env ansible-playbook
---
- name: set IP address on R1
  hosts: r1
  gather_facts: false
  tasks:
    - name: Configure interfaces on r1
      cisco.ios.ios_l3_interfaces:
        config:
          - name: "{{ item.int }}"
            ipv4:
              - address: "{{ item.ip }}/{{ item.mask }}"
                secondary: "{{ item.sec }}"
        loop:
          - { int: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }
          - { int: GigabitEthernet2, ip: 10.200.200.1, mask: 24, sec:
False }
      notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
```

```

commands:
  - command: copy running-config startup-config
    prompt: 'Destination filename.*'
    answer: 'startup-config'

```

Create the new playbook as described in the task3.txt file.

Now run your new playbook and verify the results. It should look something like this:

```

r1#conf t
Enter configuration commands, one per line. End with CNTL/Z.
r1(config)#no int lo1
r1(config)#^Z
r1#sh ip int brief
Interface                IP-Address      OK? Method Status          Protocol
GigabitEthernet1         10.0.0.15       YES manual up              up
GigabitEthernet2         10.200.200.1    YES manual up              up
GigabitEthernet3         unassigned      YES unset  administratively down down
r1#exit
Connection to clab-devpod-csr1 closed.
(Lab) → lab7 ./cisco_ip3.yml

PLAY [set IP address on R1] *****

TASK [Configure interfaces on r1] *****
changed: [r1] => (item={'int': 'Loopback1', 'ip': '172.31.1.1', 'mask': 32, 'sec': False})
ok: [r1] => (item={'int': 'GigabitEthernet2', 'ip': '10.200.200.1', 'mask': 24, 'sec': False})

PLAY RECAP *****
r1                        : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(Lab) → lab7 █

```

Configuring interfaces with loops

One of the really nice features of writing the playbook this way is how Ansible presents the loop information in the play results. You can see which interface was changed, and what values were assigned to it. It is very concise compared to other ways of writing this playbook.

Task 4 – Configuring interfaces on Junos

In this task, you will configure the loopback interface IP and ge-0/0/1.0 IP address for the vmx1 router. The goal is to configure enough functionality to allow deploying OSPF in the next lab.

The `junos_l3_interfaces` module does have some slight differences from the `ios_l3_interfaces` module. Junos does not use the secondary statement for additional IP addresses on an interface. Also, Junos has the notion of logical units that are separate from the interface name. Because of this, there are slight changes to config: statement versus the cisco version.

Use the `cat` command to view the contents of `task4.txt` in your shell.

```
(Lab) → lab7 cat task4.txt
Create a new playbook junos_ip.yml. This playbook should be modeled
on
cisco_ip3.yml, using a loop construct to define the parameters of
each
interface.
```

```
#!/usr/bin/env ansible-playbook
---
- name: set IP address on r2
  hosts: r2
  gather_facts: false
  tasks:
    - name: Configure interfaces on r2
      junipernetworks.junos.junos_l3_interfaces:
        config:
          - name: "{{ item.int }}"
            unit: "{{ item.unit }}"
            ipv4:
              - address: "{{ item.ip }}/{{ item.mask }}"
        loop:
```


- { int: lo0, unit: 0, ip: 172.31.1.2, mask: 32 }
- { int: ge-0/0/0, unit: 0, ip: 10.200.200.2, mask: 24 }

Run your new playbook and verify that r2 can ping r1's GigabitEthernet2 interface.

```
(Lab) → lab7 ssh admin@clab-devpod-vmx1
(admin@clab-devpod-vmx1) Password:
Last login: Fri Oct 20 21:29:19 2023 from 10.0.0.2
--- JUNOS 18.2R1.9 Kernel 64-bit  JNPR-11.0-20180614.6c3f819_buil
admin@r2> ping 10.200.200.1
PING 10.200.200.1 (10.200.200.1): 56 data bytes
64 bytes from 10.200.200.1: icmp_seq=0 ttl=255 time=342.332 ms
64 bytes from 10.200.200.1: icmp_seq=1 ttl=255 time=130.966 ms
^C
--- 10.200.200.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 130.966/236.649/342.332/105.683 ms

admin@r2> exit
```

Summary

You have learned how to create loops in tasks, allowing concise playbooks. You have seen how to use the cisco and juniper layer 3 interfaces modules. In the next two labs, you will configure static and dynamic routing between the two devices.

Lab 8 – Configuring static routes

This lab introduces the `static_routes` module. There is a version of this module for both Junos and IOS.

Task 1 – Configuring an IOS static route.

The static routes module provides a lot of options. In order to specify a route, most of these options are required.

The structure of a static route definition is shown below.

```
config:
  - address_families:
    - afi: ipv4
      routes:
        - dest: 172.31.1.2/32
          next_hops:
            - forward_router_address: 10.200.200.2
              distance_metric: 10
              tag: 50
```

The config statement contains a list of routes to be added.

Each route must specify the `address_families` it will represent. You can define multiple entries here, this is a list of `afi` entries.

Afi values can be "ipv4" or "ipv6". For each AFI value, you may define multiple routes.

The routes attribute is a list of destinations. Each destination can have multiple next_hops.

The `next_hop` is a list of dictionaries, each defining the parameters for one next hop. The `distance_metric` and `tag` are not required.

The indentation levels for the config parameter can be difficult to keep track of. The screenshot below has the different levels of indentation drawn in with vertical bars to help see them.

```
#!/usr/bin/env ansible-playbook
---
- name: Configure static route on IOS to r2
  hosts: r1
  gather_facts: false
  tasks:
    - name: Set static route on r1 to r2
      cisco.ios.ios_static_routes:
        config:
          - address_families:
              - afi: ipv4
                routes:
                  - dest: 172.31.1.2/32
                    next_hops:
                      - forward_router_address: 10.200.200.2
                        distance_metric: 10
```

Static route configuration

There are also many additional parameters available in this module. For example handling of VRFs, multicast, track objects, and multiple topologies.

ios static routes module documentation

Use the cat command to view the task1.txt file.

Create a new playbook names cisco_routes.yml that creates a new static route on r1 to r2's loopback address.

(Lab) → lab8 cat task1.txt

Create a playbook named cisco_route.yml to create the static route equivalent to:

```
ip route 172.31.1.2 255.255.255.255 10.200.200.2
```

Note: this module requires very careful indentation, copy and paste is recommended.

```
#!/usr/bin/env ansible-playbook
```

```
---
```

```
- name: Configure static route on IOS to r2
  hosts: r1
  gather_facts: false
  tasks:
    - name: Set static route on r1 to r2
      cisco.ios.ios_static_routes:
        config:
          - address_families:
              - afi: ipv4
                routes:
                  - dest: 172.31.1.2/32
                    next_hops:
                      - forward_router_address: 10.200.200.2
        notify: update-startup-config
  handlers:
    - name: update-startup-config
      cisco.ios.ios_command:
        commands:
          - command: copy running-config startup-config
            prompt: 'Destination filename.*'
            answer: 'startup-config'
```

Run your module and then verify that the route is installed on the router.

Task 2 – Configuring a Junos static route

The `junipernetworks.junos.junos_static_routes` module uses very similar syntax to the cisco equivalent. The junos version of this module does not support all of the optional parameters that the IOS version does. However, for the routes we are using in this lab, the configuration syntax is identical and you only need to change the ip addresses and module name.

It is important to note that in the IOS module, the parameter is “next_hops”. In the Junos module the parameter is “next_hop”, *NO S*.

At the time of writing this lab, there is a issue preventing the use of the `junos_static_routes` module. While this module is preferred, we will use the `junos_config` module to work around it.

The content of task2.txt is shown below.

```
(Lab) → lab8 cat task2.txt
```

```
Use the junos_config module to create a static route equivalent to:
set routing-options static route 172.31.1.1 next-hop 10.200.200.1
```

Edit the `juniper_route.yml` playbook to match the one shown below.

```
#!/usr/bin/env ansible-playbook
---
- name: Configure static route on Junos to r1
  hosts: r2
  gather_facts: false
  tasks:
    - name: set static route to r1 from r2
      junipernetworks.junos.junos_config:
        lines:
          - set routing-options static route 172.31.1.1 next-hop
10.200.200.1
```

Run the playbook and verify that you can ping the loopback address of r1 from r2, and the loopback address of r2 from r1.

Summary

You saw how to use the `ios_static_routes` module in this lab. You also saw how to configure static routes using the `junos_config` module. The `junos_config` module is more concise for this use case, but requires familiarity with the Junos CLI.

Lab 9 - OSPF configuration

This lab combines the work you did in previous labs to begin creating a playbook that captures the configuration of each router. Done properly, this ansible playbook can become a living document that captures all of the configuration on a device.

Task 1 – removing the static routes

Since you will be deploying OSPF, the static routes created in lab 8 are unnecessary. Ansible has a nice way of handling this. Each module supports a parameter called state. The state parameter defaults to “merged” which means the configuration provided will be merged with the existing configuration. When the existing configuration needs to be removed, changing the state parameter to “deleted” does exactly that. Most modules support the state parameter, but not all.

For this task, copy the playbooks from lab8 into the lab9 directory. Change the cisco playbook to include the state: deleted parameter at the end. The state: parameter must be aligned with the config: block.

The junos_config module doesn't support the state: deleted parameter. If the junos_static_routes module had worked, the same state deleted parameter would work. Instead, change the set command to a delete command in the lines parameter.

The file task1.txt is shown below.

```
(Lab) → lab9 cat task1.txt
```

```
This task is to delete the static routes from lab8.
```

Copy the playbooks from lab8 to lab9.

```
cp ../lab8/*.yaml .
```

Edit `cisco_route.yaml` and `juniper_route.yaml` to match the content shown below.

```
#!/usr/bin/env ansible-playbook
```

```
---
```

```
- name: Delete static route on IOS to r2
  hosts: r1
  gather_facts: false
  tasks:
    - name: Set static route on r1 to r2
      cisco.ios.ios_static_routes:
        config:
          - address_families:
              - afi: ipv4
                routes:
                  - dest: 172.31.1.2/32
                    next_hops:
                      - forward_router_address: 10.200.200.2
        state: deleted
```

```
#!/usr/bin/env ansible-playbook
```

```
---
```

```
- name: Delete static route on Junos to r1
  hosts: r2
  gather_facts: false
  tasks:
    - name: set static route to r1 from r2
      junipernetworks.junos.junos_config:
        lines:
          - delete routing-options static route 172.31.1.1 next-hop
10.200.200.1
(Lab) → lab9
```

Run the modified playbooks and verify the changes on each router.

Task 2 – Configuring OSPF with the ospfv2 modules

You will use the `cisco.ios.ios_ospfv2` and `junipernetworks.junos.junos_ospfv2` modules to configure OSPFv2 on R1 and R2. These modules arguments mirrors closely the normal for configuring OSPF on IOS and Junos.

The loopback addresses should be pingable after running your playbooks.

junos_ospfv2 module

A sample configuration for the `junos_ospfv2` module is shown below.

```
config:
  - areas:
    - area_id: 0.0.0.0
      interfaces:
        - name: ge-0/0/0.0
        - name: lo0.0
```

If you are familiar with Junos configuration, this config block mirrors the normal configuration very closely. Junos always associates the interfaces with the area in the configuration.

```
admin@r2# show protocols ospf
area 0.0.0.0 {
    interface ge-0/0/0.0;
    interface lo0.0;
}
```

There are many more possible options, for this lab the configuration presented is sufficient. The full module documentation is at the link:

ios_ospfv2 module

Like the junos module, the ios_ospfv2 module configuration closely mirrors traditional IOS configuration for OSPF. You will need to provide network statements and wildcard masks, process ids, etc.

Here is the configuration.

```
cisco.ios.ios_ospfv2:
  config:
    processes:
      - process_id: 1
        network:
          - address: 10.200.200.0
            wildcard_bits: 0.0.0.255
            area: 0
          - address: 172.31.1.1
            wildcard_bits: 0.0.0.0
            area: 0
```

This configuration is equivalent to:

```
router ospf 1
  network 10.200.200.0 0.0.0.255 area 0
  network 172.31.1.1 0.0.0.0 area 0
```

The task2.txt file is displayed below.

```
(Lab) → lab9 cat task2.txt
Configure OSPF on R2 and R1.
Create a playbook using the junos_ospfv2 module to configure ospfv2
on ge-0/0/0.0 and lo0.0
Your r2_ospf.yml playbook should be similar to:
```

```
#!/usr/bin/env ansible-playbook
```

```
---
```

```
- name: Configure OSPF on R2
  hosts: r2
  gather_facts: false
  tasks:
    - name: Configure OSPF on R2
      junipernetworks.junos.junos_ospfv2:
        config:
          - areas:
              - area_id: 0.0.0.0
                interfaces:
                  - name: ge-0/0/0.0
                  - name: lo0.0
```

Your r1_ospf.yml playbook should be similar to:

```
#!/usr/bin/env ansible-playbook
```

```
---
```

```
- name: Configure OSPF on r1
  hosts: r1
  gather_facts: false
  tasks:
    - name: Configure OSPF
      cisco.ios.ios_ospfv2:
        config:
          processes:
            - process_id: 1
              areas:
                - area_id: 0
              network:
                - address: 10.200.200.0
                  wildcard_bits: 0.0.0.255
                  area: 0
                - address: 172.31.1.1
                  wildcard_bits: 0.0.0.0
                  area: 0
          notify: update-startup-config
  handlers:
```

```
- name: update-startup-config
  cisco.ios.ios_command:
    commands:
      - command: copy running-config startup-config
        prompt: 'Destination filename.*'
        answer: 'startup-config'
```

Create and run r1_ospf.yml and r2_ospf.yml as described in the task file.

Run both playbooks and then login to each router and verify that an OSPF adjacency has formed.

Task 3 – Combining everything

Up to this point, the playbooks you have written perform a single task on the device. This is certainly a valid style of automation. Another approach commonly taken in the system administration world is to have a single playbook for each device that acts as the source of truth for that devices configuration.

In this task you will combine the work of Labs 6,7,9 into a single playbook for each router.

The task text file has been split into task3-cisco.txt and task3-juniper.txt because of the length of the files.

This playbook is a good example of how each task notifies the handler, but the handler is only run once at the end of the play.

Task3-cisco is shown below:

```
(Lab) → lab9 cat task3-cisco.txt
Create a playbook r1.yml that combines cisco_hostname3.yml from lab
6,
cisco_ip3.yml from lab 7, and r1_ospf.yml from this lab.
```

There should be a single play that uses the tasks from all 3 playbooks.

```
#!/usr/bin/env ansible-playbook
---
- name: R1 configuration
  hosts: r1
  gather_facts: false
  vars:
    hostname: csr1
  tasks:
    - name: Set system hostname on {{ inventory_hostname }}
      cisco.ios.ios_hostname:
        config:
          hostname: "{{hostname}}"
      notify: update-startup-config
    - name: Configure interfaces on r1
      cisco.ios.ios_l3_interfaces:
        config:
          - name: "{{ item.int }}"
            ipv4:
              - address: "{{ item.ip }}/{{ item.mask }}"
                secondary: "{{ item.sec }}"
        loop:
          - { int: Loopback1, ip: 172.31.1.1, mask: 32, sec: False }
          - { int: GigabitEthernet2, ip: 10.200.200.1, mask: 24, sec:
False }
      notify: update-startup-config
    - name: Configure OSPF
      cisco.ios.ios_ospfv2:
        config:
          processes:
            - process_id: 1
              areas:
                - area_id: 0
              network:
                - address: 10.200.200.0
                  wildcard_bits: 0.0.0.255
```

```

        area: 0
    - address: 172.31.1.1
      wildcard_bits: 0.0.0.0
      area: 0
  notify: update-startup-config
handlers:
- name: update-startup-config
  cisco.ios.ios_command:
    commands:
    - command: copy running-config startup-config
      prompt: 'Destination filename.*'
      answer: 'startup-config'

```

After completing r1.yml, run the playbook. If there are no changes, remove some of the previous configuration and run the playbook again to see how the handler behaves with multiple changes.

Task3-juniper.txt is shown below.

```

(Lab) → lab9 cat task3-juniper.txt
Create a playbook r2.yml that combines juniper_hostname2.yml from
lab 6,
junos_ip.yml from lab 7, and r2_ospf.yml from this lab.

```

There should be a single play that uses the tasks from all 3 playbooks.

```

#!/usr/bin/env ansible-playbook
---
- name: Configure R2
  hosts: r2
  gather_facts: false
  vars:
    hostname: r2
  tasks:
  - name: Set system host-name on {{ inventory_hostname }}
    junipernetworks.junos.junos_hostname:

```

```

    config:
      hostname: "{{hostname}}"
- name: Configure interfaces on r2
  junipernetworks.junos.junos_l3_interfaces:
    config:
      - name: "{{ item.int }}"
        unit: "{{ item.unit }}"
        ipv4:
          - address: "{{ item.ip }}/{{ item.mask }}"
  loop:
    - { int: lo0, unit: 0, ip: 172.31.1.2, mask: 32 }
    - { int: ge-0/0/0, unit: 0, ip: 10.200.200.2, mask: 24 }
- name: Configure OSPF on R2
  junipernetworks.junos.junos_ospfv2:
    config:
      - areas:
          - area_id: 0.0.0.0
            interfaces:
              - name: ge-0/0/0.0
              - name: lo0.0

```

(Lab) → lab9

Summary

You have configured OSPF using the task specific modules, and you have combined all of the work in this workshop into a single playbook that manages the state of each router.

Summary

In this workshop you have explored a lot of the features and capabilities of Ansible.

You learned about:

- Ansible facts and the inventory system
- Reading and write YAML documents
- Reading JSON
- Basic Jinja templating
- The structure of playbooks
- How to write a play with multiple tasks
- How to select the hosts that a play targets
- Using the `delegate_to` argument to run tasks on different hosts in the same play
- Using the `when` argument to run tasks conditionally
- Using the `debug` module to print the contents of variables at runtime
- Backup and restore the configuration of IOS and Junos devices using the `config` module.
- Use the `config` module to send configuration snippets to IOS and Junos devices
- Using task specific modules to configure hostname, interface IP addresses, static routes, and OSPF
- Using handlers to run tasks only when a change is made to a device
- Using the `loop` argument to repeat a task with different values each time

Thank you for participating in this workshop. Your feedback on the content of this workshop would be greatly appreciated.

Additional Resources

The ansible community has created a large number of modules, and high quality documentation.

[Ansible documentation home](#)

The Cisco IOS module documentation can be found here:

[] <https://docs.ansible.com/ansible/latest/collections/cisco/ios/index.html#plugins-in-cisco-ios>

The Juniper Junos module documentation can be found here:

[] <https://docs.ansible.com/ansible/latest/collections/junipernetworks/junos/index.html#plugins-in-junipernetworks-junos>

Jeff Geerling has created a number of useful YouTube videos around Ansible, and written a book on Ansible.

[Ansible 101 Youtube Playlist](#)

[Ansible for DevOps](#)

O'Reilly has published an excellent book on Ansible.

[Ansible Up and Running](#)