ACI Partner Rotation Program

Week 1 – N9K Standalone

Course Lab Guide

by Josh Beren (jberen) and Henry Luo (heluo),

Last Revision Date: 2/19/2016

Contributors: Norman Lau (nolau), Jeffrey Liu (jeffreli), and Kevin Echols II (kevechol)

Table of Contents

Lab 1.1.1 – Working with GitHub

Overview

In this lab, you will create a GitHub account, create a GitHub repository, and then make a commit to your new GitHub repository.

Procedures

1. Open your browser and go to http://www.github.com
2. Create an account by entering a username, email address, and password

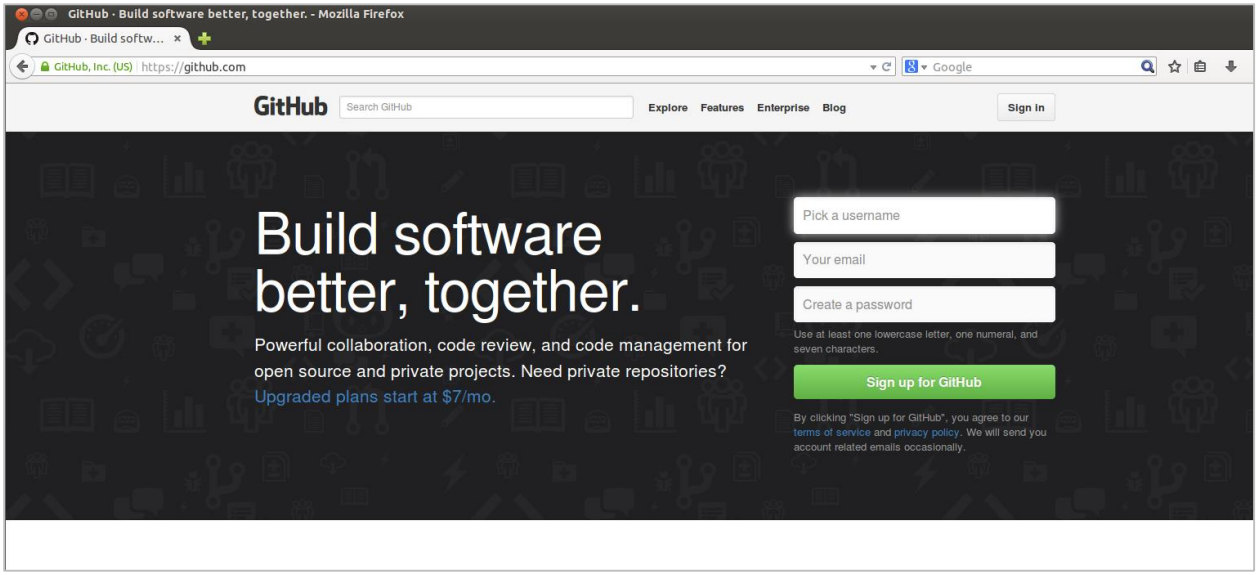

Figure 1 - Github.com

3. Select the FREE option when prompted with the next screen

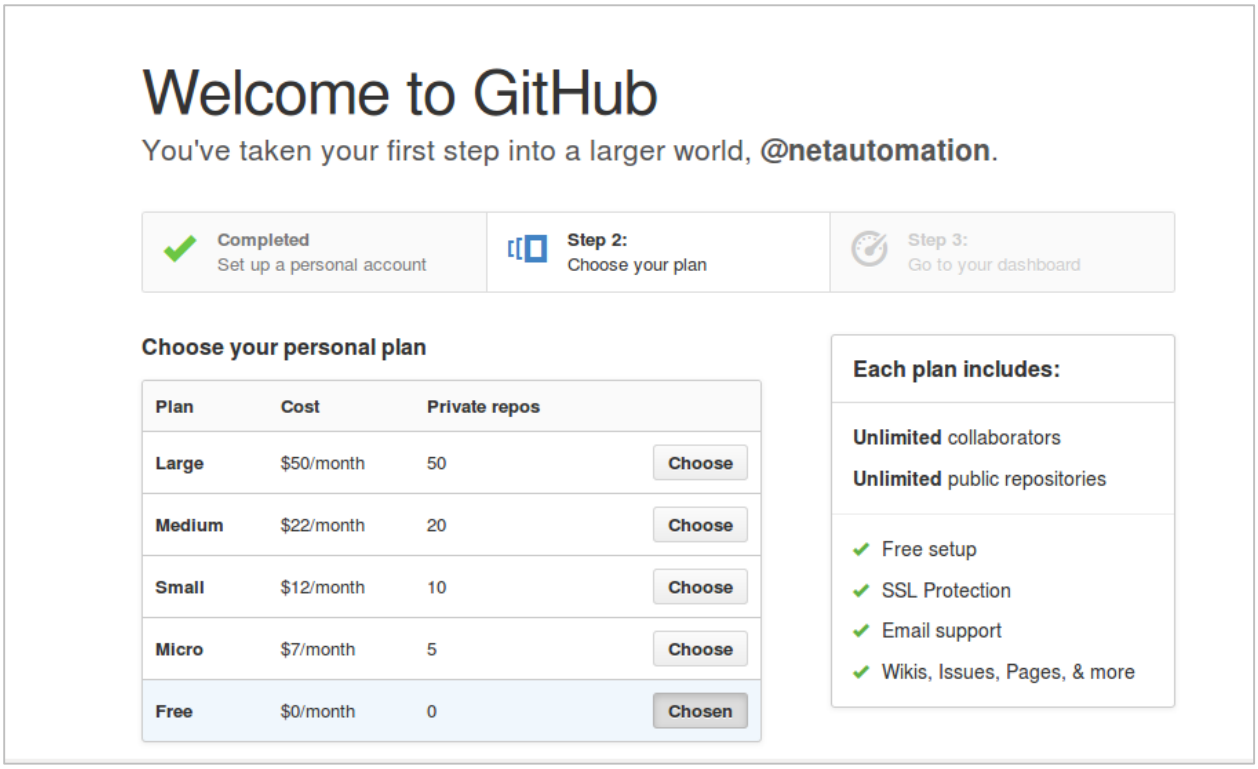

Figure 2 - Choose the Free Plan

4. On the next screen, click the **green button** in the bottom right hand portion of the screen and create a **New Repository**

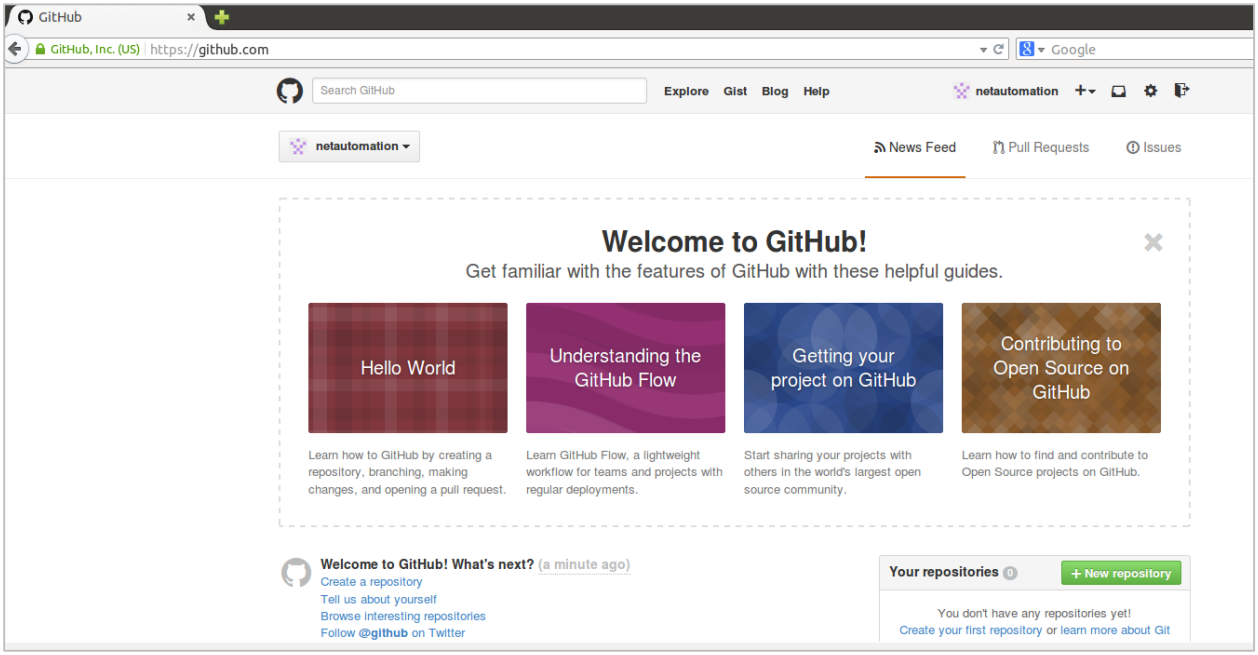

Figure 3 - Github Welcome Screen

5. Next, enter the repository name and description:
    - Name: `cisco_class`
    - Description: `First repo for programmability class`
    - Do **NOT** check the “Initialize” checkbox
    - Click the green button and “**Create Repository**”

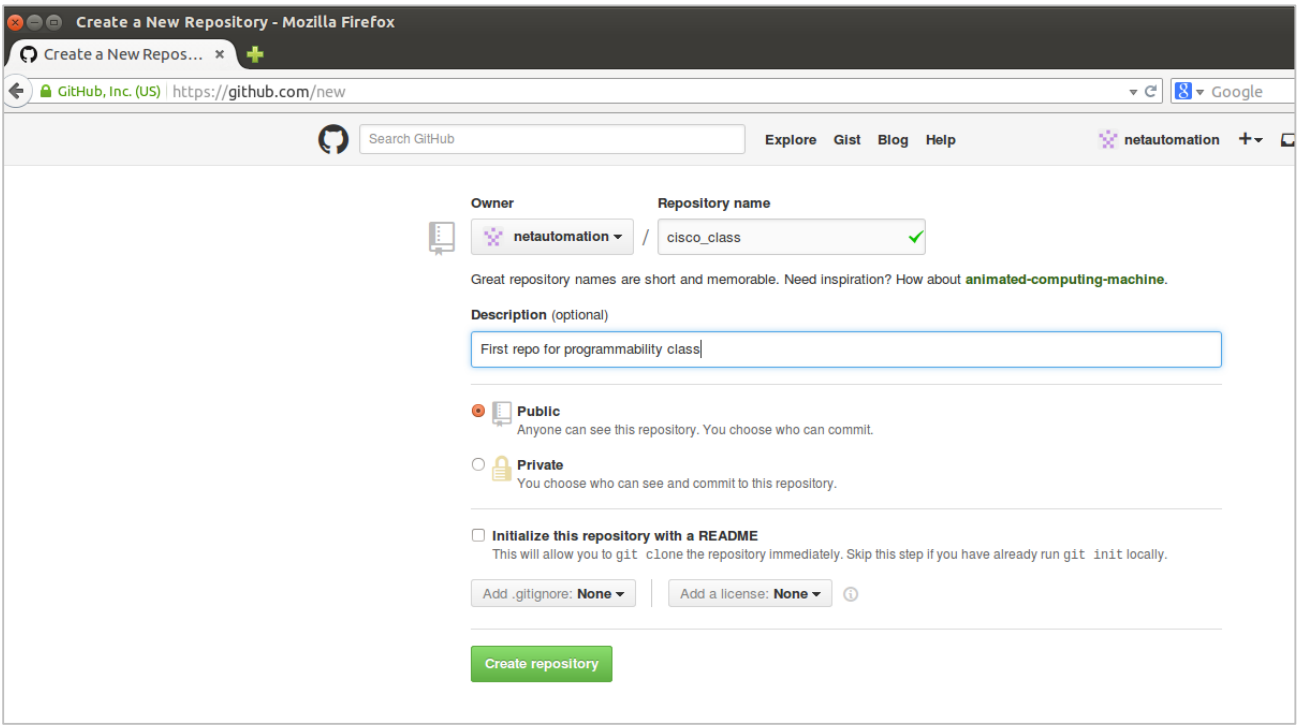

Figure 4 - Creating a Github Repository

6. You will then see the following screen

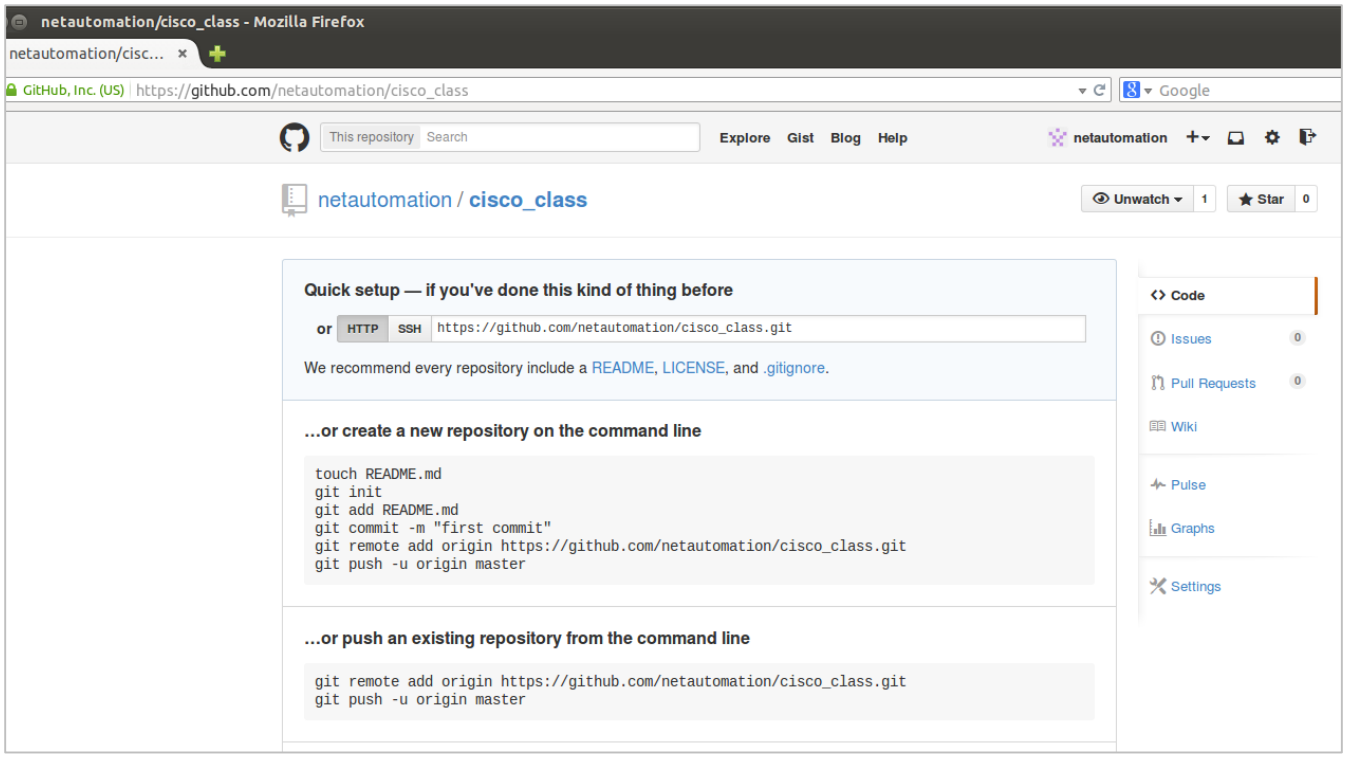

Figure 5 - Successful Repo Creation

At this point, you have officially created a GitHub repository.

7. Open a Linux Terminal window and navigate to the 'Training' folder (cd Training)
8. Create the `classfiles` directory –mkdir classfiles
9. Create a readme file in the `classfiles` directory

   Use the filename README.md (this is common practice for github repos)

   Issue the following commands:

- `touch README.md`

- `cat > README.md`

  This is the readme file!

- `CTRL+C` to back to the Linux prompt

- `cat README.md`

  This will ensure you have at least one line of text in the file.

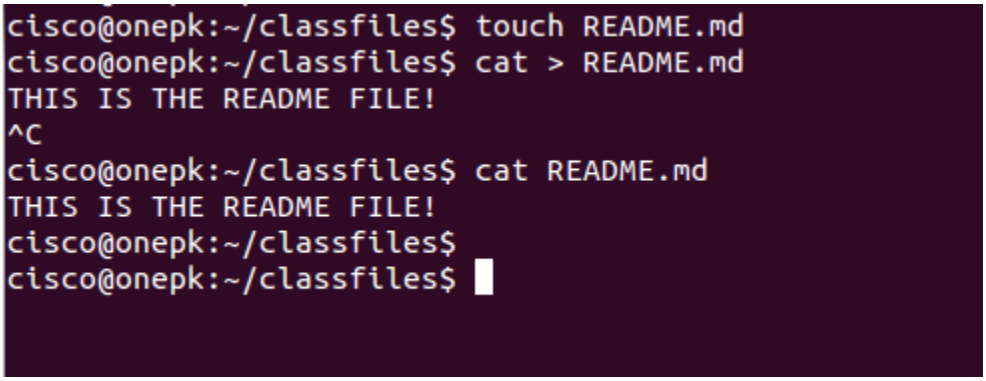

Figure 6 - Create the README.md file

10. Using `git push` to push the `readme` file to the GitHub you just created

    Make sure you are in the `classfiles` directory

    Issue the following commands:

- `git init`
- `git config --global user.name "YOUR GITHUB USERNAME"`
- `git config --global user.email YOUR_EMAIL_ADDRESS`
- `git add README.md`
- `git commit -m 'first and initial commit'`
- `git remote add origin https://<username>@github.com/<username>/cisco_class.git`
  - Note: at this point, the files are not yet pushed to github. They are being staged for commit. A message is required for the commit.
- `git push origin master`

```
cisco@onepk:~/classfiles$ git config --global user.name "netautomation"
cisco@onepk:~/classfiles$ git config --global user.email nycnetops@gmail.com
cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$ git init
Initialized empty Git repository in /home/cisco/classfiles/.git/
cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$ git add README.md
cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$ git commit -m 'first and initial commit'
[master (root-commit) 9b61690] first and initial commit
 1 files changed, 34 insertions(+)
 create mode 100644 README.md

cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$ git remote add origin https://netautomation@github.com/
netautomation/cisco_class.git
cisco@onepk:~/classfiles$
cisco@onepk:~/classfiles$ git push origin master

Password for 'https://netautomation@github.com':
To https://github.com/netautomation/cisco_class.git
 * [new branch]      master -> master
cisco@onepk:~/classfiles$
```

Figure 7 - From git init to git push

11. Validate the `git push`

- Go back to your repository on GitHub.com
- Click the README.md file so you can see the contents of the file.

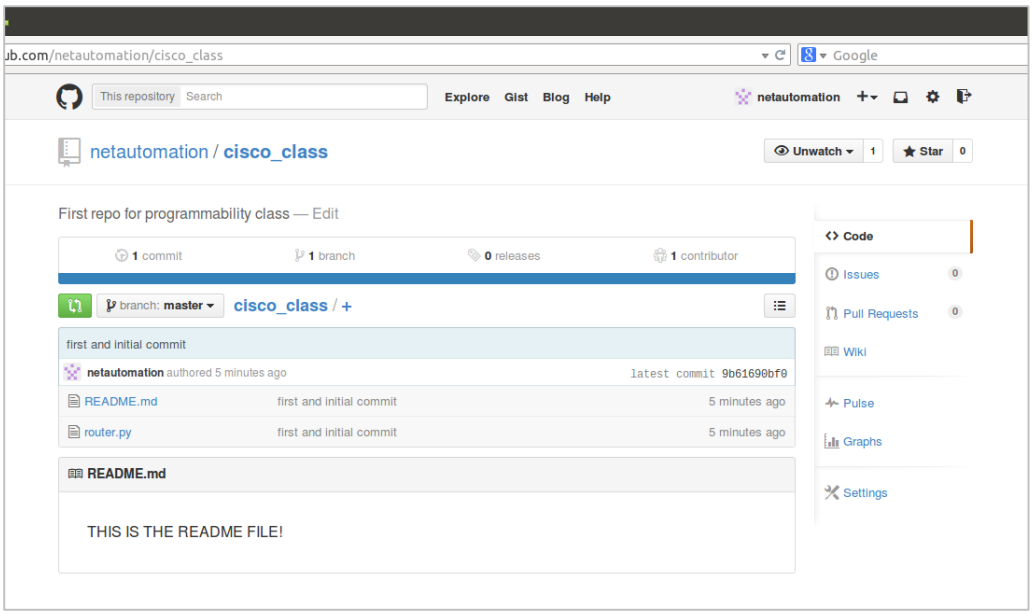

Figure 8 - Examine Github repository

The real power of GitHub is around collaboration. It is often collaboration around source code, but any type of file, including network configuration files can be stored in GitHub. Using additional `git` commands, you can do diffs on files, see what has changed, and “owners” of files can be notified if a “non-owner” tries to commit a change to their file. Then the “owner” can approve the change once it is validated. And every change has comments and an audit trail.

Lab 1.1.2 – Working with Python Dictionaries and Functions in the Python Shell

Overview

This lab will build on what you've learned in the previous lab while working in the Python Shell. This lab will continue to use the Python shell, but will focus on more advanced data types and programming concepts including Python dictionaries, functions, and for loops.

Procedures

1. Create three Python dictionaries. Type `python` to enter Python Shell

   Each dictionary will hold data that would normally be pulled from network devices, e.g. OS version, hostname, domain, management IP address, and model. This data will be referred to as "device facts."

   Each dictionary will be created in a different way to understand that there is more than one way to create a dictionary.

- `router1 = {'os_version':'3.1.1', 'hostname':'nyc_router1', 'model':'nexus 9396', 'domain':'cisco.com', 'mgmt_ip':'10.1.50.11'}`

- `router2 = dict( os_version='3.2.1', hostname='rtp_router2', model='nexus 9396', domain='cisco.com', mgmt_ip='10.1.50.12')`

- `os_version = '3.1.1'`
- `hostname = 'ROUTER3'`
- `model = 'nexus 9396'`
- `domain = 'lab.cisco.com'`
- `mgmt_ip = '10.1.50.13'`

- `router3 = {'os_version':os_version,'hostname': hostname,'model':model,'domain':domain,'mgmt_ip':mgmt_ip}`

```
>>> router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus 9396
','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
>>>
>>> router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9396
', domain='cisco.com', mgmt_ip='10.1.50.12')
>>>
>>> os_version = '3.1.1'
>>> hostname = 'ROUTER3'
>>> model = 'nexus 9396'
>>> domain = 'lab.cisco.com'
>>> mgmt_ip = '10.1.50.13'
>>>
>>> router3 = {'os_version':os_version,'hostname': hostname,'model':model,'domai
n':domain,'mgmt_ip':mgmt_ip}
>>>
```

Figure 9 - Creating Python dictionaries

2. Access & Print Key/Value Pairs. Remember a dictionary is made up of key/value pairs. For example, one key/value pair is `hostname` (key) and `nyc_router1` (value). Some other programming languages call these hash maps/tables.

   Perform the following tasks.

   - Print the router1 `hostname`

     ```
     print router1['hostname']
     ```

   - Print the `router1 os_version`

     ```
     print router1['os_version']
     ```

   - Print the `router3 hostname`

     ```
     print router3['hostname']
     ```

   - Update and then print the os_`version` for `router2` to be `'3.1.1'`

     ```
     router2['os_version'] = '3.1.1'

     print router2['os_version']
     ```

   - Update and then print the `model` for `router3` to be `'nexus 9504'`

     ```
     router3['model'] = 'nexus 9504'

     print router3['model']
     ```

Use the following figure for guidance.

```
>>>
>>> print router1['hostname']
nyc_router1
>>>
>>> print router1['os_version']
3.1.1
>>>
>>> print router3['hostname']
ROUTER3
>>>
>>> router2['os_version'] = '3.1.1'
>>> print router2['os_version']
3.1.1
>>>
>>> router3['model'] = 'nexus 9504'
>>> print router3['model']
nexus 9504
>>>
>>>
```

Figure 10 - Using Python Dictionaries - Key/Value Assignments

Print just the keys of `router3`. This will require using a built-in method call `keys()`. Feel free to use `dir(router3)` if you would like to see other built-in methods for dictionaries.

```
>>>
>>> print router3.keys()
['os_version', 'model', 'hostname', 'domain', 'mgmt_ip']
>>>
```

Figure 11 - Using keys()

Check to see if `router2` has `hostname` as one of keys in the dictionary

```
>>>
>>> router2.has_key('hostname')
True
>>>
```

Figure 12 – Using has_key

3. Create a function called `getRouter()`

   The function will accept one argument called `rtr`. All three dictionaries should be re-created in the new function. Use `def getRouter(rtr):` to create the function and then indent 4 spaces for each new line in the function.

   The function should contain all three dictionaries, but also a conditional block that will evaluate `rtr`. `rtr` is a variable that will be passed to the function. If `rtr` matches pre-defined strings as shown in the following figure, a particular router object (router1, router2, or router3) will be returned. Note: the strings being used to evaluate `rtr` do NOT need to match the `variable` name.

- After the last line, hit enter two times to be back at the `'>>>'` prompt.

- Following the commands outlined in the figure below, create a function at the Python shell.

```
def getRouter(rtr):
    router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus
9396','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
    router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9396',
domain='cisco.com', mgmt_ip='10.1.50.12')
    router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504',
domain='lab.cisco.com', mgmt_ip='10.1.50.13')

    if rtr == 'router1':
        return router1
    elif rtr == 'router2':
        return router2
    elif rtr == 'router3':
        return router3

    return 'No router found.'
```

```
>>>
>>> def getRouter(rtr):
...     router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus
9396','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
...     router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus
9396', domain='cisco.com', mgmt_ip='10.1.50.12')
...     router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504
', domain='lab.cisco.com', mgmt_ip='10.1.50.13')
...
...     if rtr == 'router1':
...         return router1
...     elif rtr == 'router2':
...         return router2
...     elif rtr == 'router3':
...         return router3
...
...     return 'No router found.'
...
>>>
```

Figure 13 - Creating a function

4. Make three function calls to `getRouter()`.

   Note: what is in the parentheses below is an argument (variable, object, etc.) and will be passed to `getRouter()` and will be accessed as `rtr` while in the function.

   Function call 1:

- `result1 = getRouter('router1')`
- `print result1`

   Function call 2:

- `getRouter('router3')`

   Function call 3:

- `result2 = getRouter('router4')`
- `result2`

   Because data is being returned from the function, this can be stored in a variable. If you are only printing data in the function, you don't necessarily need to return any data.

```
>>>
>>> result1 = getRouter('router1')
>>> print result1
{'os_version': '3.1.1', 'model': 'nexus 9396', 'hostname': 'nyc_router
1', 'domain': 'cisco.com', 'mgmt_ip': '10.1.50.11'}
>>>
>>> getRouter('router3')
{'os_version': '3.1.1', 'domain': 'lab.cisco.com', 'hostname': 'ROUTER
3', 'model': 'nexus 9504', 'mgmt_ip': '10.1.50.13'}
>>>
>>> result2 = getRouter('router4')
>>> result2
'No router found.'
>>>
```

Figure 14 - Calling functions

5. In the previous task, you evaluated `rtr` based on an insignificant random string in the function. Update the conditional to use a loop and check the actual `hostname` of the device. If `rtr` matches one of the hostnames, return that device. Don't forget to use the up arrow here to save time typing!

   Follow the commands in the figure below:

```
def getRouter(rtr):
    router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus
   9396','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
    router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9396',
   domain='cisco.com', mgmt_ip='10.1.50.12')
    router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504',
   domain='lab.cisco.com', mgmt_ip='10.1.50.13')

    router_list = [router1,router2,router3]
    for router in router_list:
        if rtr == router['hostname']:
            return router
    return 'No router found.'
```

```
>>>
>>> def getRouter(rtr):
...     router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus 9
396','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
...     router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9
396', domain='cisco.com', mgmt_ip='10.1.50.12')
...     router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504'
, domain='lab.cisco.com', mgmt_ip='10.1.50.13')
...
...     router_list = [router1,router2,router3]
...     for router in router_list:
...         if rtr == router['hostname']:
...             return router
...     return 'No router found.'
...
>>>
```

Figure 15 - Updating the function

If you can think of a different way to perform the conditional using containment (`in`), feel free to try that out too.

6. Make three function calls to `getRouter()`.

   Note: you are now passing the REAL hostname to `getRouter()`

   Function call 1:

- `result1 = getRouter('nyc_router1')`
- `print result1`

   Function call 2:

- `getRouter('router_blob')`

Function call 3:

- `result2 = getRouter('ROUTER3')`
- `result2`

```
>>>
>>> result1 = getRouter('nyc_router1')
>>> print result1
{'os_version': '3.1.1', 'model': 'nexus 9396', 'hostname': 'nyc_router1', 'domain
': 'cisco.com', 'mgmt_ip': '10.1.50.11'}
>>>
>>> getRouter('router_blob')
'No router found.'
>>>
>>> result2 = getRouter('ROUTER3')
>>> result2
{'os_version': '3.1.1', 'domain': 'lab.cisco.com', 'hostname': 'ROUTER3', 'model'
: 'nexus 9504', 'mgmt_ip': '10.1.50.13'}
>>>
```

Figure 16 - Calling functions with real hostnames

Lab 1.1.3 – Creating a Python Script

Overview

In this lab, you will build on the knowledge and experience gained from the previous labs and see how you can go from using the Python interactive interpreter to writing Python scripts. You will then learn how to create a Python script that accepts command line arguments from the Linux terminal.

Procedures

1. Open a **Terminal Window**
2. Create new directory called `classfiles`

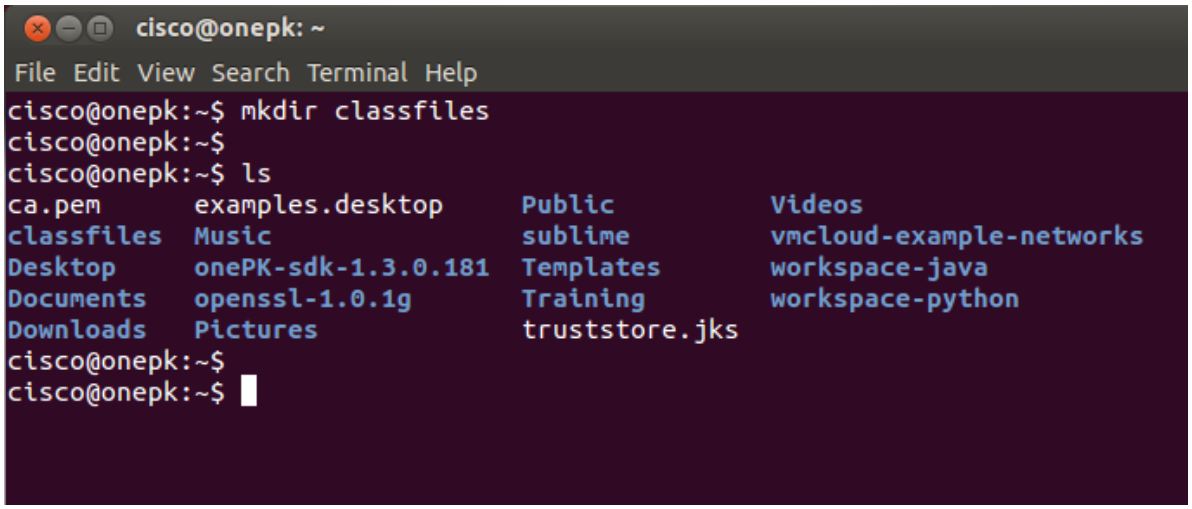

Figure 17 - mkdir classfiles

3. Open the **Sublime Text Editor**

- Find and double-click the **Sublime Text** Icon on the Ubuntu Desktop in the bottom left corner.

4. While in **Sublime Text**, do a **"Save As**" and save the empty file as `my_script1.py in the new classfiles` directory. Make sure you navigate to place it in the `classfiles` directory that is located in your (`cisco`) `home` directory.

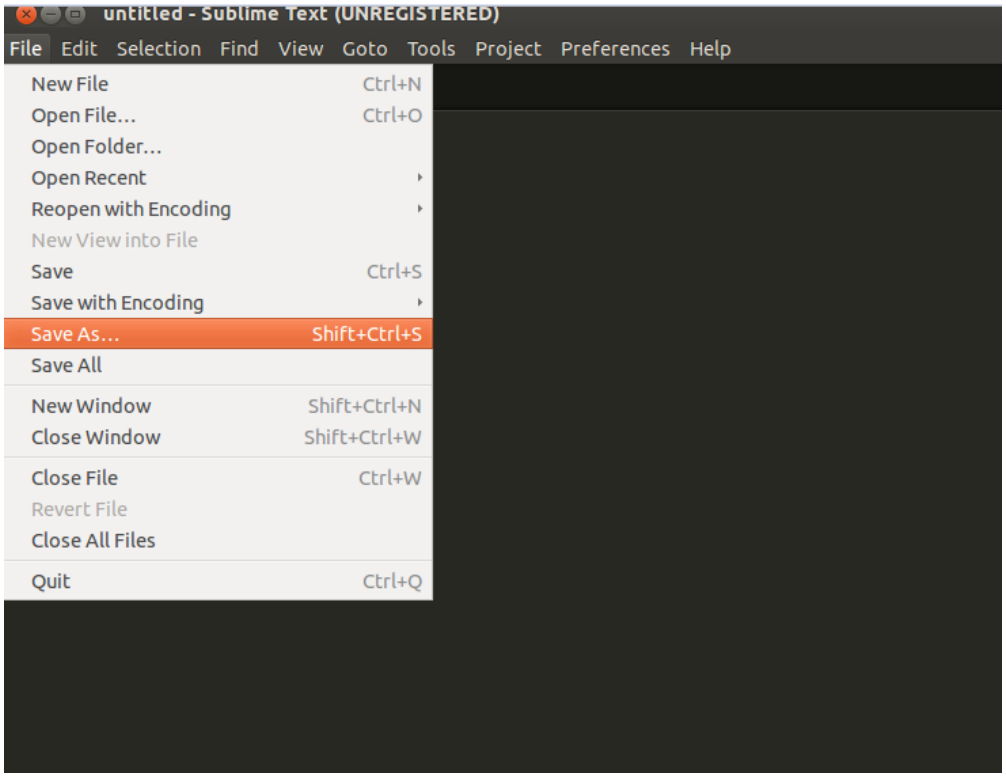

Figure 18 - Sublime Text (save as)

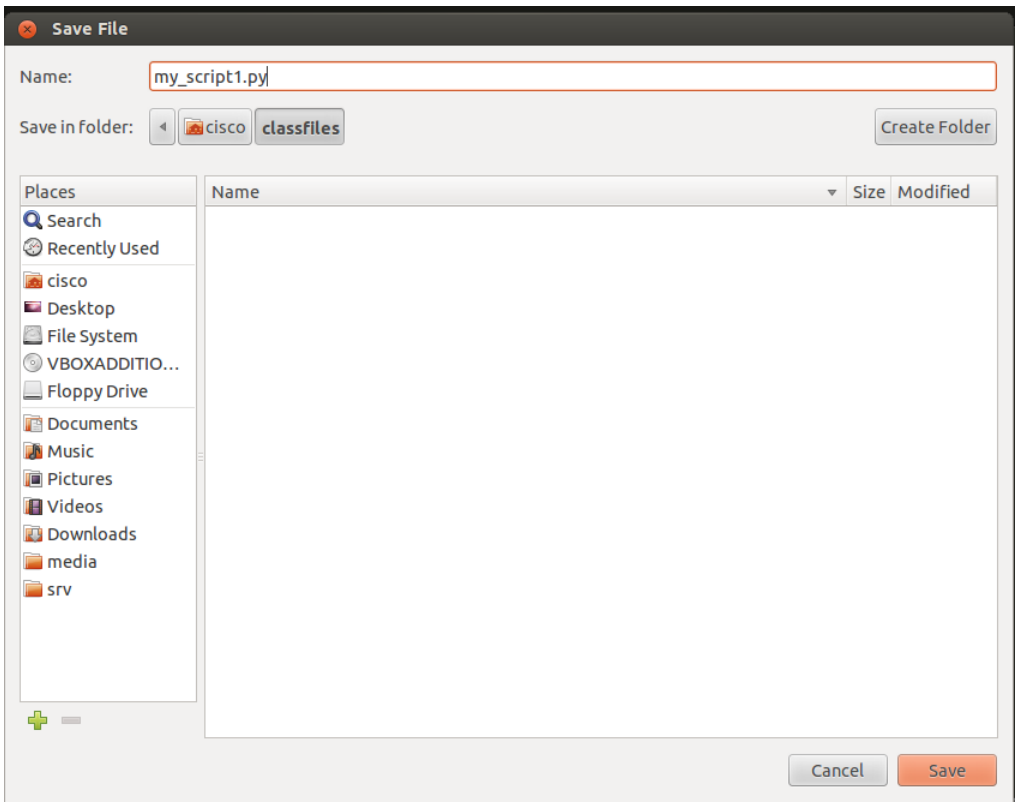

Figure 19 - my_script.py

5. Create a basic script that prints a single line of text. You will need to add three (3) things to the script.

- **Shebang** – the first line of a python always starts with a shebang or hashbang "#!". This tells the script where to find the version of Python being used. The following uses what is defined in the Linux Python environment variable. You can also specify a particular version here if you wish too. You won't need to worry about this at all for the course.

- **if __name__ == " __main__"** statement that is always required for code that will be executed as a Python script

- **What you want to print, i.e. the code**

```
#!/usr/bin/env python


if __name__ == "__main__":

    print 'It is about time I started programming.'
```

Figure 20 - The first script!

6. Go back to the terminal window and navigate to the `classfiles` directory

7. Execute the python script using the following command:

- `python my_script1.py`

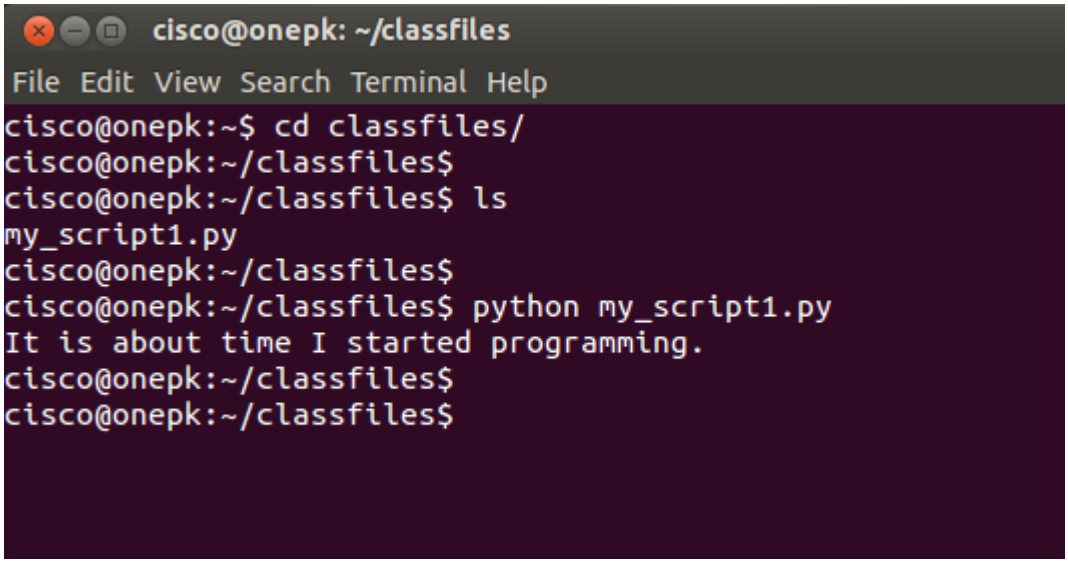

Figure 21 - Executing a Python script

- **Congratulations**. You just created and executed a Python script!

8. Create a new script called `router.py`

- This script should have the contents of what was used in the last task on the Python shell (remember those 3 dictionaries!). Reference the figure below for the function definition.

```
#!/usr/bin/env python

def getRouter(rtr):
    router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus
9396','domain':'cisco.com','mgmt_ip':'10.1.50.11'}
    router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9396',
domain='cisco.com', mgmt_ip='10.1.50.12')
    router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504',
domain='lab.cisco.com', mgmt_ip='10.1.50.13')

    router_list = [router1,router2,router3]
    for router in router_list:
        if rtr == router['hostname']:
            return router
    return 'No router found.'
```

The new part relative to the script should look like the following:

```
if __name__ == "__main__":
    result1 = getRouter('nyc_router1')
    print result1

    result2 = getRouter('ROUTER3')
    print result2
```

The function should be located above the “main” conditional block. Everything else is unchanged.

Note: it is often good practice to limit the amount of characters on a line to 80, but this is not done here to offer a simplistic view of the code.

At this point, you should recognize how the Python shell is different from writing Python scripts.

Ensure your script looks like the following figure.

Figure 22 - router.py

9. Execute the script from the Linux terminal (make sure you are in the `classfiles` directory)

   Use the following command:

- `python router.py`

```
cisco@onepk:~/classfiles$ python router.py
{'os_version': '3.1.1', 'model': 'nexus 9396', 'hostname': 'nyc_router1', 'domain': 'cisco.com',
'mgmt_ip': '10.1.50.11'}
{'os_version': '3.1.1', 'domain': 'lab.cisco.com', 'hostname': 'ROUTER3', 'model': 'nexus 9504',
'mgmt_ip': '10.1.50.13'}
cisco@onepk:~/classfiles$
```

10. Instead of hard-coding the hostname of the router you want to print, now enter the hostname through the Linux command line. Update the script to use command line arguments.

    You will use the built in module called `sys` that has a built-in variable called `argv`. Variables sent in from the command line get assigned to argv. argv is a Python list. As an example, if you execute a script like this from the Linux Terminal: `python script.py r1 r2 r3`, within the script (in sublime text) `argv` would be a list that resembles the following if you configured it manually:

- `argv = [script.py, 'r1', 'r2', 'r3']` – but remember, this happens automatically, you do not have to configure this list!

    You will notice that the first element in the `argv` list is always the script name. You'll see this when you `print args` below.

    Now instead of hard coding the router1 and router2 hostnames that get sent to `getRouter()` in the script, you will send the arguments from the command line. You will need to send the second and third arguments because, remember, the first is always the script name. After making the changes, your new code should look like this. Note the changes in RED/bold.

```
import sys (put this in line 2 of the script)
if __name__ == "__main__"

    args = sys.argv

    print args

    result1 = getRouter(args[1])
    print result1

    result2 = getRouter(args[2])
    print result2
```

    Use the following figure for guidance.

router.py

```
#!/usr/bin/env python

import sys

def getRouter(rtr):
    router1 = {'os_version':'3.1.1','hostname':'nyc_router1','model':'nexus 9396', \
    'domain':'cisco.com','mgmt_ip':'10.1.50.11'}
    router2 = dict( os_version='3.2.1', hostname='rtp_router2',model='nexus 9396', \
     domain='cisco.com', mgmt_ip='10.1.50.12')
    router3 = dict( os_version='3.1.1', hostname='ROUTER3',model='nexus 9504', \
     domain='lab.cisco.com', mgmt_ip='10.1.50.13')

    router_list = [router1,router2,router3]
    for router in router_list:
        if rtr == router['hostname']:
            return router
    return 'No router found.'

if __name__ == "__main__":

     args = sys.argv

     print args

     result1 = getRouter(args[1])
     print result1

     result2 = getRouter(args[2])
     print result2

```

Figure 23 - Using command line arguments

11. Execute the new `router.py` script using command line arguments as inputs to the script: `python router.py nyc_router1 rtp_router2`

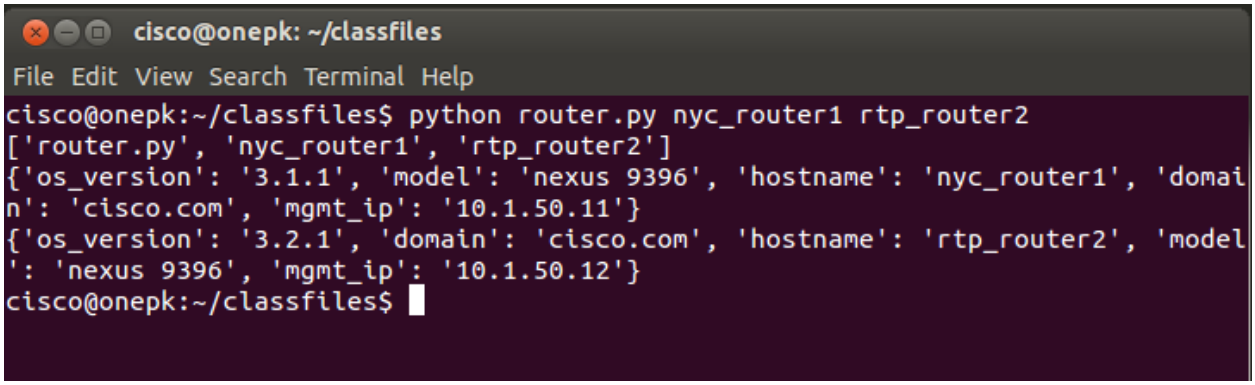

Figure 24 - Executing router.py

- Note: you have three `print` statements in the script and should notice three `print` statements in the output. The first `print` statement is `print args` – here you should notice that `args` is a `list` and the first element in the list is the script name. This is why we start with `args[1]` when using and passing `args` around within the script.

12. Update the script again (Fig 26) now to only print a single key value pair.

```
#print args

result1 = getRouter(args[1])

print 'HOSTNAME:', result1['hostname']

value = args[2]

print value.upper() + ': ' + result1[value]
```

The script should execute by running the following command (taking a look at the User Experience):

- `python router.py hostname_of_router dictionary_key`

- Instead of passing it two hostnames, you will pass it the hostname (arg1) of the router you want information about and the second argument will be, the exact information or dictionary key (arg2), you want out of the dictionary for that device.

- Use the following figure for guidance. Feel free to print args again too.

```
18
19  if __name__ == "__main__":
20
21      args = sys.argv
22
23      #print args
24
25      result1 = getRouter(args[1])
26
27      print 'HOSTNAME:', result1['hostname']
28
29      value = args[2]
30
31      print value.upper() + ': ' + result1[value]
32
```

Figure 25 - Updating the script

- Now execute the script by sending two command line arguments to the script. The first argument should be hostname of a device and the second should be the value you want to extract for that device.

  Issue the following command. It will print the model for router1.

  `python router.py nyc_router1 model`

- As always, make sure you pause and understand what is happening. Don't forget to add `print` statements where you need to throughout the script to ensure you understand what is happening at various points in the script.

```
cisco@onepk: ~/classfiles
File  Edit  View  Search  Terminal  Help
cisco@onepk:~/classfiles$ python router.py nyc_router1 model
HOSTNAME: nyc_router1
MODEL: nexus 9396
cisco@onepk:~/classfiles$
```

Figure 26 - Using multiple command line arguments

Lab 1.1.4 – Native Python on NX-OS

Overview

In this lab, you will build on the knowledge and experience gained from the previous labs and see how you can go from using the Python interactive interpreter on a Linux server to use the Python interactive interpreter on a Nexus switch.

Procedures

1. Open a Terminal Window and ssh into your Nexus 9000 switch:
    - Ask instructor for IP address, 172.31.217.x
    - username: admin
    - password: cisco123
    - example: `ssh admin@172.31.217.133`

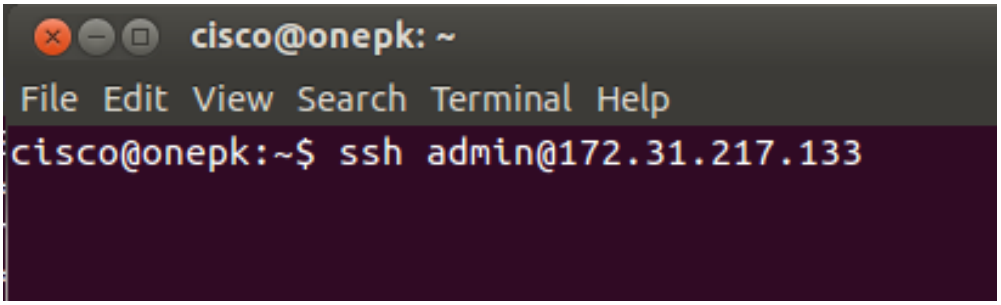

Figure 27 - ssh to nexus 9000

2. If it asks you, accept the connection and add the SSH key to the list of known hosts – type `yes` at the `(yes/no)` prompt. If you connected to the device already, you won't see this.

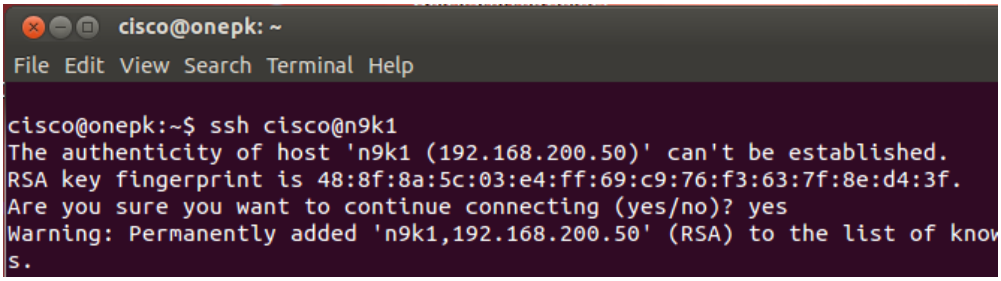

Figure 28 - Accept SSH key from 9K

3. Enter the Python interactive interpreter located on the switch by typing `python` and pressing `Enter`. The output should look very familiar. You should see it's the same Python shell found on servers.

```
N9K1#
N9K1# python
Python 2.7.5 (default, Oct  8 2013, 23:59:43)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> _
```

Figure 29 - Using the Python Interpreter on the Nexus 9000

4. Import cisco specific CLI modules (they are already loaded on the 9K)

   Issue the following command:

- `from cli import *`

   Examine the methods that were imported. Issue the following command:

- `dir()`

   You will notice a few different CLI methods including `cli`, `clid`, and `clip`.
   Each of them can be used to issue a CLI command, but each returns output differently.

```
>>> from cli import *
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'cli', 'cli_syntax_error'
, 'clid', 'clip', 'cmd_exec_error', 'data_type_error', 'json', 'shlex', 'structu
red_output_not_supported_error', 'subprocess', 'unexpected_error', 'xmltodict']
>>> _
```

Figure 30 - Importing cisco cli modules

5. Pass the Cisco CLI command `show ip int brief` to each of the CLI methods (`cli`, `clip`, and `clid`). Compare and contrast the outputs.

   Issue the following commands from the Python shell on the Nexus 9000 switch:

- `cli('show ip int brief').`

```
>>> cli('show ip int brief')
'IP Interface Status for VRF "default"(1)\nInterface            IP Address
Interface Status\nLo33                 192.168.33.3    protocol-up/link-up/admin
-up       \nLo88                 192.168.88.1    protocol-up/link-up/admin-up
    \nLo99                 192.168.99.1    protocol-up/link-up/admin-up       \n
Eth1/1               10.101.101.1    protocol-down/link-down/admin-up   \nEth1/2
              10.254.1.1      protocol-down/link-down/admin-up   \nEth1/3
         10.1.30.1       protocol-down/link-down/admin-up   \nEth2/1
   10.2.200.1      protocol-down/link-down/admin-up   \n'
>>>
```

Figure 31 - Using cli()

- `clid('show ip int brief')`.

  Notice the output is a Python dictionary, or nested dictionaries.

```
>>> clid('show ip int brief')
'{"TABLE_vrf": [{"ROW_vrf": {"vrf-name-out": "default"}}, {"ROW_vrf": {"vrf-name
-out": "default"}}, {"ROW_vrf": {"vrf-name-out": "default"}}, {"ROW_vrf": {"vrf-
name-out": "default"}}, {"ROW_vrf": {"vrf-name-out": "default"}}, {"ROW_vrf": {"
vrf-name-out": "default"}}, {"ROW_vrf": {"vrf-name-out": "default"}}], "TABLE_in
tf": [{"ROW_intf": {"intf-name": "Lo33", "prefix": "192.168.33.3", "ip-disabled"
: "FALSE", "iod": "6", "proto-state": "up", "link-state": "up", "admin-state": "
up"}}, {"ROW_intf": {"intf-name": "Lo88", "prefix": "192.168.88.1", "ip-disabled
": "FALSE", "iod": "7", "proto-state": "up", "link-state": "up", "admin-state":
"up"}}, {"ROW_intf": {"intf-name": "Lo99", "prefix": "192.168.99.1", "ip-disable
d": "FALSE", "iod": "8", "proto-state": "up", "link-state": "up", "admin-state":
 "up"}}, {"ROW_intf": {"intf-name": "Eth1/1", "prefix": "10.101.101.1", "ip-disa
bled": "FALSE", "iod": "9", "proto-state": "down", "link-state": "down", "admin-
state": "up"}}, {"ROW_intf": {"intf-name": "Eth1/2", "prefix": "10.254.1.1", "ip
-disabled": "FALSE", "iod": "10", "proto-state": "down", "link-state": "down", "
admin-state": "up"}}, {"ROW_intf": {"intf-name": "Eth1/3", "prefix": "10.1.30.1"
, "ip-disabled": "FALSE", "iod": "11", "proto-state": "down", "link-state": "dow
n", "admin-state": "up"}}, {"ROW_intf": {"intf-name": "Eth2/1", "prefix": "10.2.
200.1", "ip-disabled": "FALSE", "iod": "57", "proto-state": "down", "link-state"
: "down", "admin-state": "up"}}]}'
```

Figure 32 - Using clid()

- `clip('show ip int brief').`

  This output will mimic that of being on the CLI.

```
>>> clip('show ip int brief')
IP Interface Status for VRF "default"(1)
Interface            IP Address      Interface Status
Lo33                 192.168.33.3    protocol-up/link-up/admin-up
Lo88                 192.168.88.1    protocol-up/link-up/admin-up
Lo99                 192.168.99.1    protocol-up/link-up/admin-up
Eth1/1               10.101.101.1    protocol-down/link-down/admin-up
Eth1/2               10.254.1.1      protocol-down/link-down/admin-up
Eth1/3               10.1.30.1       protocol-down/link-down/admin-up
Eth2/1               10.2.200.1      protocol-down/link-down/admin-up
```

Figure 33 - Using clip()

6. Go back into the **Sublime Text Editor** and open the file called `interface_stats.py` located in the `~/Training` directory. Take a look at the source code.

```
7  if __name__ == "__main__":
8
9      length = len(sys.argv)
10
11     if  length == 1:
12         print '\n Input Requires an option. \n\n Available Options: '
13         print ' - crc\n - runts\n - coll\n'
14     else:
15         args = sys.argv
16         stats = json.loads(clid('show interface'))
17
18         if args[1] == 'help':
19             print '\n Available Options: ' + '\n - crc ' + '\n - runts\n - coll\n'
20         else:
21           if not (args[1] == 'crc' or args[1] == 'runts' or args[1] == 'coll'):
22                 print '\nInvalid Option'
23                 print 'Input Requres a Valid Option'
24                 print '\n Available Options: ' + '\n - crc ' + '\n - runts\n - coll\n'
25             else:
26                 for each in stats['TABLE_interface']['ROW_interface']:
27                     if each['interface'].startswith('Eth'):
28                         delta = 15 - len(each['interface'])
29                   spaces = delta * ' '
30                   if length == 2:
31                     occurences = 0
32                   elif length == 3:
33                       occurences = args[2]
34                   if args[1] == 'crc' and each['eth_crc'] >= occurences:
35                     print each['interface'] + ': ' + spaces + 'CRC errors: ' + each['eth_crc']
36                   elif args[1] == 'runts' and each['eth_runts'] >= occurences:
37                     print each['interface'] + ': ' + spaces + 'runts: ' + each['eth_runts']
38                   elif args[1] == 'coll' and each['eth_coll'] >= occurences:
39                     print each['interface'] + ': ' + spaces + 'collisions: ' + each['eth_coll']
40
```

Figure 34 - Examining interface_stats.py

- The following bullets outline the code in the figure above. Take a few moments to review the bullets while referencing the code:
  - The built-in function `len()` is being used to check to see how many arguments are being passed in from the command line
  - If there is only 1 argument passed, this means the user didn't input any arguments because remember that the script name is always the first argument. If this is true, the available options are output for the user: "crc," "runts," and "coll." This is a mini help menu with how to use the script.
  - Otherwise, the output of show interface is loaded into a new variable. You can disregard `json.loads` for now, but that command is helping store the output from the `clid` call as a dictionary. `json.loads` will be covered in more detail soon.
  - If help is the only (or first) argument a mini help menu will be displayed again, otherwise there are some other checks to ensure the right arguments are entered and it is not just based on quantity of arguments.

- If everything is input correctly, there is a for loop that loops through every Ethernet interface on the switch and prints either the runts, collisions, or crc errors for every interface. The second argument is an optional integer and if this is sent, only the interfaces with more than N [runts, collisions, or crc errors] will be displayed.

If you were on the Python shell, the user experience would like this:

```
python interface_stats.py runts 0
```

Figure 35 - Example: running interface_stats.py

- The above example would print all interfaces that have greater than 0 runts. To run this script on the Nexus, you will need to use the following command because the script is stored in the `scripts` directory in `bootflash`.

- **Note: if the script isn't already on the Nexus switch, you will need to transfer the file to the switch from your virtual machine. First, check to see if interface_stats.py is on the switch. It would be in the scripts directory in bootflash.**

```
N9K# dir bootflash:scripts
No such file or directory
N9K#
```

Figure 36 - dir bootflash

- This figure above shows that the file and directory do not exist on the Nexus 9000. If you see the `interface_stats.py` script, you can skip this step.

- You need to execute the following commands:
- `mkdir bootflash:scripts`
- `dir bootflash:` (to verify the proper `scripts` dir was created)

```
N9K# mkdir bootflash:scripts
N9K#
N9K# dir bootflash:
       3651    Oct 11 09:21:04 2014  20141011_162104_poap_6233_init.log
       3651    Oct 11 09:45:03 2014  20141011_164503_poap_5995_init.log
    2097212    Nov 07 13:58:14 2014  20141106_011849_poap_5995_1.log
     796689    Nov 07 14:10:59 2014  20141106_011849_poap_5995_2.log
    1048674    Nov 05 18:57:43 2014  20141106_011849_poap_5995_init.log
     118231    Nov 07 14:15:55 2014  20141107_221328_poap_5995_init.log
     606083    Nov 07 14:31:46 2014  20141107_222043_poap_5995_init.log
       4096    Nov 12 11:54:24 2014  home/
      16384    Oct 11 09:17:21 2014  lost+found/
  353457152    Oct 11 09:18:18 2014  n9000-dk9.6.1.2.I3.1.bin
       4096    Dec 11 14:41:20 2014  scripts/
       4096    Dec 11 13:50:23 2014  virt_strg_pool_bf_vdc_1/
       4096    Dec 11 13:50:11 2014  virtual-instance/
         55    Dec 11 13:50:05 2014  virtual-instance.conf

Usage for bootflash://sup-local
  762507264 bytes used
51142017024 bytes free
51904524288 bytes total
N9K#
```

- Now, transfer the script onto the switch using `scp`.
- In your virtual machine, navigate to the `Training` directory and issue an `ls` to ensure the `interface_stats.py` script is there.

```
cisco@onepk:~$ cd Training/
cisco@onepk:~/Training$
cisco@onepk:~/Training$ ls
addroute.py           device.py           N9K1.conf  onepk.py
Ansible               device.pyc          nxapi.py   Solutions
demo_config_build.py  interface_stats.py  nxapi.pyc
cisco@onepk:~/Training$
cisco@onepk:~/Training$
```

Within the `Training` directory, issue the following command:

- `scp interface_stats.py USERNAME@A.B.C.D:/scripts/interface_stats.py`

```
cisco@onepk:~/Training$ scp interface_stats.py admin@n9k1:/scripts/interface_stats.py
User Access Verification
Password:
interface_stats.py                                    100% 1582     1.5KB/s   00:00
cisco@onepk:~/Training$
```

- You can verify that the script is now in the proper directory by going back to the 9K and issuing the `dir bootflash:scripts` command again.

```
N9K# dir bootflash:scripts
       1582    Dec 11 14:52:06 2014  interface_stats.py

Usage for bootflash://sup-local
  762511360 bytes used
51142012928 bytes free
51904524288 bytes total
N9K#
```

7. Run the script

- `python bootflash:scripts/interface_stats.py runts 0`

```
N9K1# python bootflash:scripts/interface_stats.py runts 0
Ethernet1/1:     runts: 0
Ethernet1/2:     runts: 0
Ethernet1/3:     runts: 0
Ethernet1/4:     runts: 0
Ethernet1/5:     runts: 0
Ethernet1/6:     runts: 0
Ethernet1/7:     runts: 0
Ethernet1/8:     runts: 0
Ethernet1/9:     runts: 0
Ethernet1/10:    runts: 0
```

Figure 37 - Running interface_stats.py while on the Nexus 9K

- Run the script without any arguments. Go back to the code if you'd like to review what is happening.

```
N9K1# python bootflash:scripts/interface_stats.py

 Input Requires an option.

 Available Options:
 - crc
 - runts
 - coll

N9K1#
```

Figure 38 - Running interface_stats.py without a proper command line argument

8. Configure 10 new VLANs using the on box Python shell. You should use the built-in `cli` method

- Create a list that has 10 VLAN names:
  `'web','db','web2','db2','voice','video','srvs','test','prod','qa'`

- Using a for loop, configure the VLAN ID and VLAN name for each. This examples configures 10 contiguous VLANs: VLAN 10-19. Note: multiple commands can be executed within the same `cli` method call. Each command needs to be separated by a semi-colon.

- `from cli import *`
- `vlan_names = ['web','db','web2','db2','voice','video','srvs','test','prod','qa']`
- `for vlan in range(10,20):`
- `    cli('config t ; ' + 'vlan ' + str(vlan) + ' ; ' + 'name ' + vlan_names[vlan-10] + ' ; ')`

- Note: the `vlan_names` index is using vlan-10 to configure VLAN IDs 10-19. Example: the first iteration would be `vlan_names[0]` which is the first element in the `vlan_names` list. `vlan` is an integer and increments by 1 each iteration through the loop.

```
>>> from cli import *
>>>
>>> vlan_names = ['web','db','web2','db2','voice','video','srvs','test','prod','qa']
>>>
>>> for vlan in range(10,20):
...     cli('config t ; ' + 'vlan ' + str(vlan) + ' ; ' + 'name ' + vlan_names[vlan-10] + ' ; ')
...
```

Figure 39 - Creating 10 VLANs while on the Python Interpreter on the Nexus 9K

- Exit back to the Cisco CLI using `exit()` or `ctrl+D`

- Issue the `show vlan` command and verify all VLANs have been created properly

Figure 40 - Verifying VLANs were created

Lab 2.1.1 – Creating a Network Configuration Template

Overview

You've now learned the fundamentals of working in Python. We will now review how to create templates. Templates can be used for configuration templates, config files on Linux systems, or even reports. In this lab, you will use a templating language called Jinja2 that is built for native integration with Python as well as a structured data format called YAML to learn how to create configuration templates for network devices.

Procedures

1. In the **Sublime Text** text editor, open the file called `N9K1.conf` located in the cisco/`Training` directory. This file is a sample configuration of a Nexus switch.
   The top portion of the file will look like this:

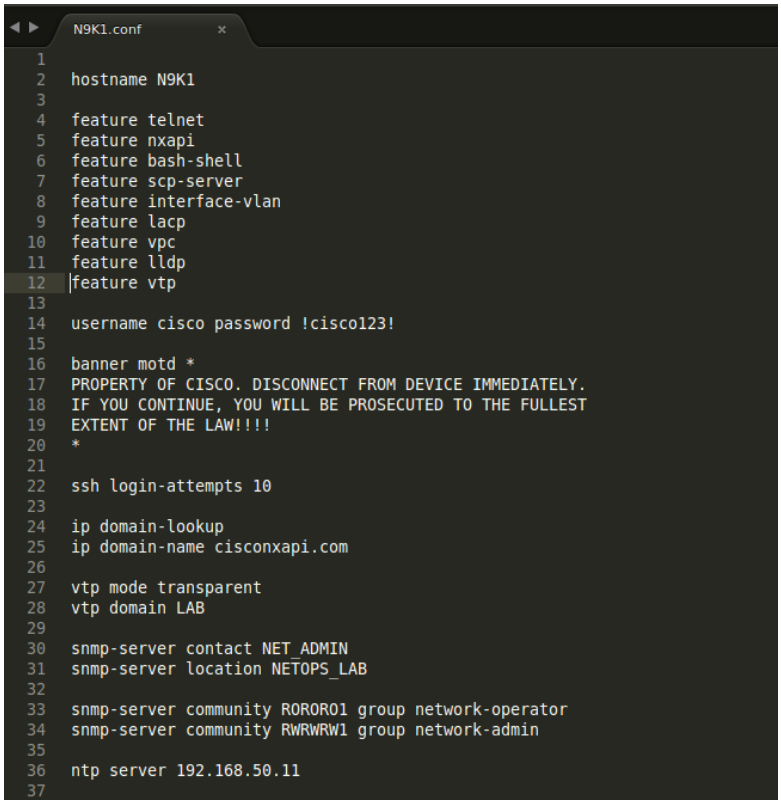

Figure 41 - NK91.conf (switch config file)

2. Create a jinja2 template file
   Copy the contents of the `N9K1.conf` file into a new file called `N9K1.j2` while in Sublime.

3. Prepare and populate the template file you just created (`N9K1.j2)`

   Replace parts of config with variable names as stated documented below. Remember, enclose each variable in double curly braces {{ example }}

- `hostname {{ hostname }}`
- ***< remove features >***
- `username {{ username }} password {{ password }}`
- `banner motd *`

  `{{ banner_motd }}`

  `*`
- ***<remove>*** `ssh login-attempts 10`
- ***<remove>*** `ip domain-lookup`
- `ip domain-name {{ domain }}`
- `vtp mode {{ vtp_mode }}`
- ***<remove>*** `vtp domain LAB`
- `snmp-server contact {{ snmp_contact }}`
- `snmp-server location {{ snmp_location }}`
- `snmp-server community {{ ro_string }} group network-operator`
- `snmp-server community {{ rw_string }} group network-admin`
- `ntp server {{ ntp_server }}`
- `{{ route_1 }}`
- `{{ route_2 }}`
- ***<remove the vlans >***
- `vrf context {{ vrf_name }}`
- `ip domain-name {{ domain }}`
- `{{ route }}`
- ***<remove>*** `interface Vlan1`
- `interface {{ intf_id_1 }}`
- `{{ switchport_1 }}`
- `ip address {{ ip_1 }}`
- `{{ state_1 }}`
- `interface {{ intf_id_2 }}`
- `{{ switchport_2 }}`
- `ip address {{ ip_2 }}`

```
      {{ state_2 }}
   interface {{ intf_id_3 }}
      {{ switchport_3 }}
      switch mode {{ mode_3 }}
      switchport trunk native vlan {{ native_vlan_3 }}
      switchport trunk allowed vlan {{ vlan_range_3 }}
      {{ state_3 }}
    interface {{ intf_id_4 }}
      {{ switchport_4 }}
      switch mode {{ mode_4 }}
      switchport trunk native vlan {{ native_vlan_4 }}
      switchport trunk allowed vlan {{ vlan_range_4 }}
      {{ state_4 }}
interface mgmt0
    vrf member management
    ip address {{ mgmt_ip }}
```

4. The resulting Jinja2 (j2) file should look like the following figures.

N9K1.j2 | N9K1.conf | N9K1.yml

```
1
2
3   hostname {{ hostname }}
4
5   username {{ username }} password {{ password }}
6
7   banner motd *
8   {{ banner_motd }}
9   *
10
11  ip domain-name {{ domain }}
12
13  vtp mode {{ vtp_mode }}
14
15  snmp-server contact {{ snmp_contact }}
16  snmp-server location {{ snmp_location }}
17  snmp-server community {{ ro_string }} group network-operator
18  snmp-server community {{ rw_string }} group network-admin
19
20  ntp server {{ ntp_server }}
21
22  {{ route_1 }}
23  {{ route_2 }}
24
25  vrf context {{ vrf_name }}
26    ip domain-name {{ domain }}
27    {{ route }}
28
29  interface {{ intf_id_1 }}
30    {{ switchport_1 }}
31    ip address {{ ip_1 }}
32    {{ state_1 }}
33
```

Figure 42 - Jinja2 template Part 1

```
interface {{ intf_id_2 }}
  {{ switchport_2 }}
  ip address {{ ip_2 }}
  {{ state_2 }}

interface {{ intf_id_3 }}
  {{ switchport_3 }}
  switch mode {{ mode_3 }}
  switchport trunk native vlan {{ native_vlan_3 }}
  switchport trunk allowed vlan {{ vlan_range_3 }}
  {{ state_3 }}

interface {{ intf_id_4 }}
  {{ switchport_4 }}
  switch mode {{ mode_4 }}
  switchport trunk native vlan {{ native_vlan_4 }}
  switchport trunk allowed vlan {{ vlan_range_4 }}
  {{ state_4 }}

interface mgmt0
  vrf member management
  ip address {{ mgmt_ip }}


```

Figure 43 - Jinja2 Template Part 2

5. Create a YAML variables file

   Create a new file while in **Sublime Text** and save the file as `N9K1.yml`. This will be the YAML file that holds the values that will be rendered with the Jinja2 template. Remember all YAML files start with three dashes/hyphens. Take the values that were replaced from the config file and store them in the YAML. The YAML file should look like the following figures.

```yaml
---

hostname: N9K1-HOSTNAME

username: admin
password: "cisco123"

banner_motd: "PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF THE LAW!!!!"

domain: cisconxapi.com

vtp_mode: transparent

snmp_contact: JOHN_CHAMBERS

snmp_location: CISCO_SJC

ro_string: RORORO
rw_string: RWRWRW

ntp_server: 192.168.50.11

route_1: "ip route 9.0.0.0/24 192.168.88.2"
route_2: "ip route 192.168.88.0/24 192.168.33.1"

vrf_name: management
route: "ip route 0.0.0.0/0 192.168.200.1"
```

```yaml
intf_id_1: Ethernet1/1
switchport_1: "no switchport"
ip_1: "10.101.101.1/30"
state_1: "no shutdown"

intf_id_2: Ethernet1/2
switchport_2: "no switchport"
ip_2: "10.254.1.1/30"
state_2: "no shutdown"

intf_id_3: Ethernet1/3
switchport_3: switchport
mode_3: trunk
native_vlan_3: 3000
vlan_range_3: "100-102,200-202"
state_3: "no shutdown"

intf_id4: Ethernet1/4
switchport_4: switchport
mode_4: trunk
native_vlan_4: 3000
vlan_range_4: "100-102,200-202"
state_4: "no shutdown"

mgmt_ip: 172.31.217.133/23
```

Figure 44 - Template Variables YAML

6. Render the template with variables.

- Go to a **Terminal Window** and navigate to the `Training` directory.

- At the prompt, render the template using the pre-created `demo_config_build.py` Python script. The script accepts two arguments: the jinja2 and yaml files. The order doesn't matter.

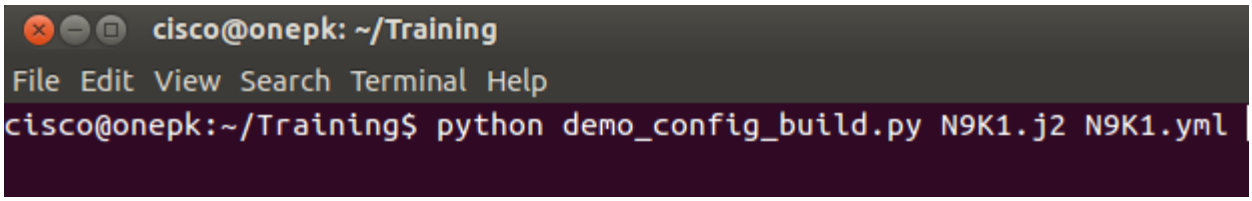

Figure 45 - Rendering the jinja2 template and YAML variables files together

The output will look like what is seen in the following figures.(next page)

```
cisco@onepk:~/Training$ python demo_config_build.py N9K1.j2 N9K1.yml

hostname N9K1-HOSTNAME

username admin password cisco123

banner motd *
PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT O
 THE LAW!!!!
*

ip domain-name cisconxapi.com

vtp mode transparent

snmp-server contact JOHN_CHAMBERS
snmp-server location CISCO_SJC

snmp-server community RORORO group network-operator
snmp-server community RWRWRW group network-admin

ntp server 192.168.50.11

ip route 9.0.0.0/24 192.168.88.2
ip route 192.168.88.0/24 192.168.33.1

vrf context management
  ip domain-name cisconxapi.com
  ip route 0.0.0.0/0 192.168.200.1
```

Figure 46 - Rendered Config Part 1

```
interface Ethernet1/1
  no switchport
  ip address 10.101.101.1/30
  no shutdown

interface Ethernet1/2
  no switchport
  ip address 10.254.1.1/30
  no shutdown

interface Ethernet1/3
  switchport
  switch mode trunk
  switchport trunk native vlan 3000
  switchport trunk allowed vlan 100-102,200-202
  no shutdown

interface Ethernet1/4
  switchport
  switchport mode trunk
  switchport trunk native vlan 3000
  switchport trunk allowed vlan 100-102,200-202
  no shutdown

interface mgmt0
  vrf member management
  ip address 172.31.217.133/23
cisco@onepk:~/Training$
```

Figure 47 - Rendered Config Part 2

Note: This example / demo script output the rendered configuration to the terminal, but with a small change to the script, it is possible to store in a text file.

Lab 2.1.2 – Working with Files: Importing YAML Variables into Python

Overview

In the previous lab, you created a basic template that was output to the terminal. In this lab, you will gain insight and learn how to import a YAML file into Python that will allow you to create and define variables in YAML, but then import them to Python and work with them just as if they were defined locally.

Procedures

1. Open a Linux Terminal and navigate to the Training directory, if you aren't already there.

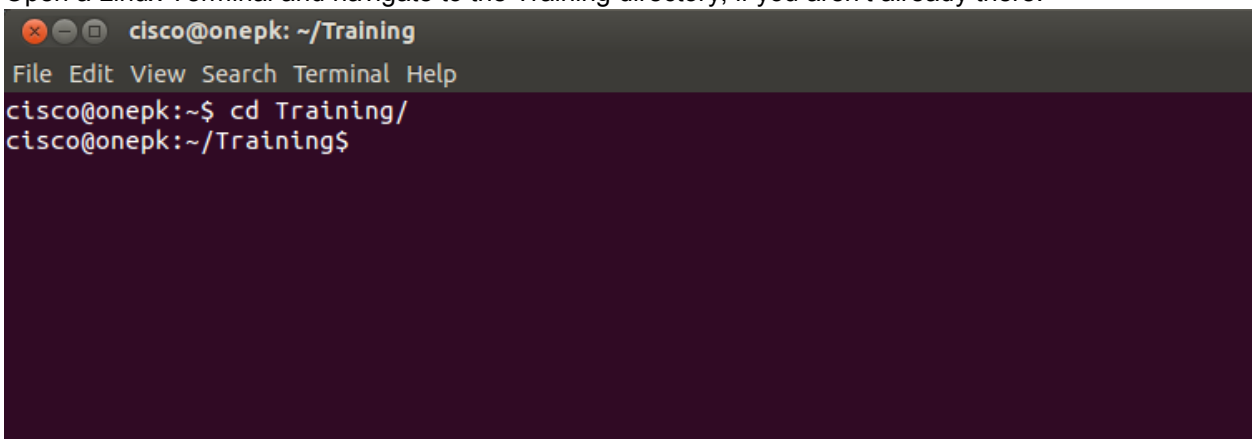

Figure 48 - Navigate to Training dir

2. Enter the Python Shell

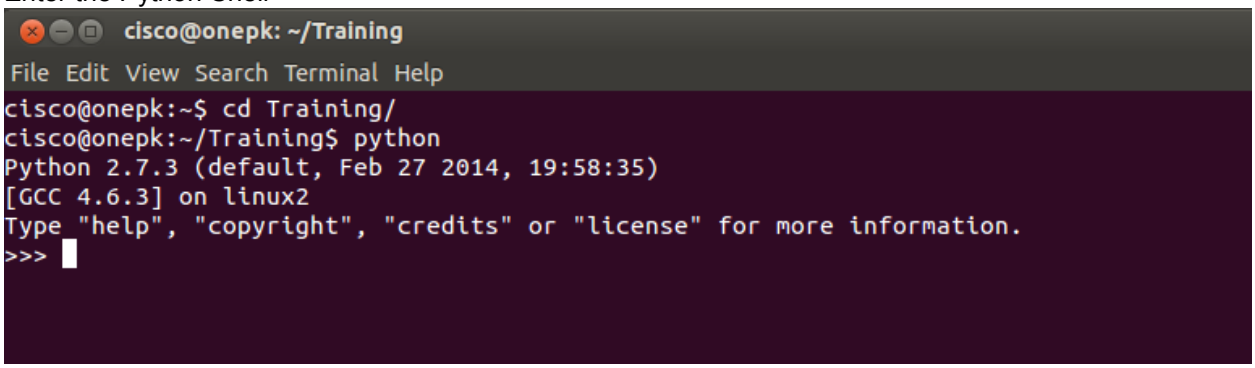

Figure 49 - Enter the Python shell

3. Import yaml
   Issue the following command: `import yaml`

```
cisco@onepk:~/Training$ python
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import yaml
>>>
>>>
```

Figure 50 - import yaml

4. Open and import the N9K1.yml file that was used from the previous lab

   Issue the following command:

- `my_vars = yaml.load(file('N9K1.yml','r'))`

   This will open the file in “read-only” mode and import the variables. The option “r” is used for “read-only”

```
>>> import yaml
>>>
>>> my_vars = yaml.load(file('N9K1.yml','r'))
>>>
>>>
```

Figure 51 - Load vars from file

5. Verify the data type of `my_vars`

   Issue the command:

- `dir(my_vars)`

   When you see the output, you should be able to tell what data type it is.

```
>>> dir(my_vars)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__
', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_
key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update'
, 'values', 'viewitems', 'viewkeys', 'viewvalues']
>>>
>>>
```

Figure 52 - Examine dir() of new dict var

   Were you able to tell that `my_vars` is a dictionary?

6. Print out the variables that were just imported

   Issue the following commands:

- `print my_vars['username']`
- `print my_vars['password']`
- `print my_vars['route']`

```
>>>
>>> print my_vars['username']
cisco
>>>
>>> print my_vars['password']
!cisco123!
>>>
>>> print my_vars['route']
ip route 0.0.0.0/0 192.168.200.1
>>>
>>>
```

Figure 53 - print vars from yaml file

You can also use the built-in methods for dictionaries.

Issue the following commands:

- `print my_vars.keys()`
- `print my_vars.values()`

```
>>> print my_vars.keys()
['native_vlan', 'username', 'domain', 'ip_2', 'ip_1', 'state_1', 'ntp_server', 'state_2', 'state_3', '
password', 'vtp_mode', 'mgmt_ip', 'intf_id_4', 'hostname', 'intf_id_1', 'intf_id_3', 'state_4', 'switc
hport_4', 'switchport_1', 'switchport_2', 'switchport_3', 'route_1', 'route_2', 'rw_string', 'snmp_loc
ation', 'mode_3', 'mode_4', 'intf_id_2', 'vlan_range', 'route', 'snmp_contact', 'banner_motd', 'vrf_na
me', 'ro_string']
>>>
>>> print my_vars.values()
[3000, 'cisco', 'cisconxapi.com', '10.254.1.1/30', '10.101.101.1/30', 'no shutdown', '192.168.50.11',
'no shutdown', 'no shutdown', '!cisco123!', 'transparent', '192.168.200.11/24', 'Ethernet1/4', 'N9K1-H
OSTNAME', 'Ethernet1/1', 'Ethernet1/3', 'no shutdown', 'switchport', 'no switchport', 'no switchport',
 'switchport', 'ip route 9.0.0.0/24 192.168.88.2', 'ip route 192.168.88.0/24 192.168.33.1', 'RWRWRW',
'CISCO_SJC', 'trunk', 'trunk', 'Ethernet1/2', '100-102,200-202', 'ip route 0.0.0.0/0 192.168.200.1', '
JOHN_CHAMBERS', 'PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF T
HE LAW!!!!', 'management', 'RORORO']
>>>
```

Figure 54 - print keys/values of my_vars

Note: You will work more with dictionaries in upcoming labs.

Lab 2.1.3 – Using Advanced Data Structures to Simplify Template Creation

Overview

This lab will build on what you accomplished in the previous configuration templating lab using a Jinja2 template and a YAML variables file. You will modify and update each of the files used to create the template by optimizing how variables are declared and used. You will use lists, dictionaries, and conditional statements in the Jinja2 template and then learn how to create these data structures in YAML.

Procedures

1. Open each of the files from the previous config template lab: `N9K1.j2` and `N9K1.yml`
2. Do a "**Save As**" on each and save the files with the new names `N9K1_v2.j2` and `N9K1_v2.yml`, respectively.
3. Go into the N9K1_v2.yml and make the following changes in the figures below. These changes will leverage data types other than integers and strings including lists and dictionaries.

   Note: the changes are for `snmp`, `routes`, `vlans`, and `interfaces`

```yaml
---

hostname: N9K1-HOSTNAME

username: admin
password: "cisco123"

banner_motd: "PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT

domain: cisconxapi.com

vtp_mode: transparent

snmp:
  { contact: JOHN_CHAMBERS, location: CISCO_SJC, ro_string: RORORO, rw_string: RWRWRW }

ntp_server: 192.168.50.11

routes:
  - "ip route 9.0.0.0/24 192.168.88.2"
  - "ip route 192.168.88.0/24 192.168.33.1"

vrf_name: management
route: "ip route 0.0.0.0/0 192.168.200.1"
```

```yaml
vlans:
  - { id: 10, name: web }
  - { id: 11, name: qa }
  - { id: 12, name: prod }
  - { id: 13, name: test }
  - { id: 14, name: srvs }
  - { id: 15, name: video }
  - { id: 16, name: voice }
  - { id: 17, name: db2 }
  - { id: 18, name: web2 }
  - { id: 19, name: db }
  - { id: 1000, name: vlan_1000 }
  - { id: 3000, name: dummy }

interfaces:
  - { intf: Ethernet1/1, switchport: "no switchport", ip: "10.101.101.1/30", state: "no shutdown"}
  - { intf: Ethernet1/2, switchport: "no switchport", ip: "10.254.1.1/30", state: "no shutdown"}
  - { intf: Ethernet1/3, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202", state: "no shutdown"}
  - { intf: Ethernet1/4, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202", state: "no shutdown"}

mgmt_ip: 172.31.217.133/23
```

Figure 55 - Updated YAML Part 2

4. Go into the N9K1_v2.j2 and make the following changes in the figures below.

- By using lists and dictionaries in the YAML, you will now use conditionals (if statements), for loops, and understand how to access dictionaries while using the Jinja2 templating language.

Note: Take notice of the "-" in the for loops and conditional blocks. This is removing whitespace before/after the blocks. Feel free to not use the "-" symbols to see how the formatting turns out when the template is rendered, but it may not be pretty!

```
hostname {{ hostname }}

username {{ username }} password {{ password }}

banner motd *
{{ banner_motd }}
*

ip domain-name {{ domain }}

vtp mode {{ vtp_mode }}

snmp-server contact {{ snmp.contact }}
snmp-server location {{ snmp.location }}
snmp-server community {{ snmp.ro_string }} group network-operator
snmp-server community {{ snmp.rw_string }} group network-admin

ntp server {{ ntp_server }}

{% for route in routes %}
{{ route }}
{%- endfor %}
```

Figure 56 - Updated j2 template Part 1

```
{% for vlan in vlans %}
vlan {{ vlan.id }}
{%- if vlan.name %}
  name {{ vlan.name }}
{%- endif %}
{%- endfor %}

vrf context {{ vrf_name }}
  ip domain-name {{ domain }}
  {{ route }}

{%- for interface in interfaces %}
interface {{ interface.intf }}
  {{ interface.switchport }}
{%- if interface.switchport == 'no switchport' %}
  ip address {{ interface.ip }}
{%- endif %}
{%- if interface.switchport == 'switchport' %}
  switchport mode {{ interface.mode }}
  switchport trunk native vlan {{ interface.native_vlan }}
  switchport trunk allowed vlan {{ interface.vlan_range }}
{%- endif %}
  {{ interface.state }}
{% endfor -%}

inteface mgmt0
  vrf member management
  ip address {{ mgmt_ip }}
```

Figure 57 - Updated j2 template Part 2

5. Render the new template with the new variables file.

Issue the following command:

- `python demo_config_build.py N9K1_v2.j2 N9K1_v2.yml`

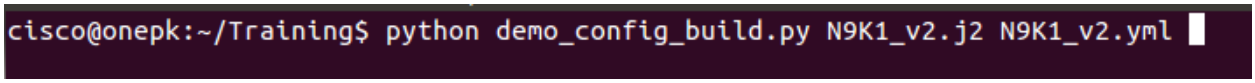

Figure 58 - Execute the render script using updates j2/YAML files

The output of the script should look like the following figures.

```
cisco@onepk:~/Training$ python demo_config_build.py N9K1_v2.j2 N9K1_v2.yml

hostname N9K1-HOSTNAME

username admin password cisco123

banner motd *
PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT
 OF THE LAW!!!!
*

ip domain-name cisconxapi.com

vtp mode transparent

snmp-server contact JOHN_CHAMBERS
snmp-server location CISCO_SJC
snmp-server community RORORO group network-operator
snmp-server community RWRWRW group network-admin

ntp server 192.168.50.11


ip route 9.0.0.0/24 192.168.88.2
ip route 192.168.88.0/24 192.168.33.1
```

Figure 59 - Updated Rendered Config Part 1

```
vlan 10
  name web
vlan 11
  name qa
vlan 12
  name prod
vlan 13
  name test
vlan 14
  name srvs
vlan 15
  name video
vlan 16
  name voice
vlan 17
  name db2
vlan 18
  name web2
vlan 19
  name db
vlan 1000
  name vlan_1000
vlan 3000
  name dummy

vrf context management
  ip domain-name cisconxapi.com
  ip route 0.0.0.0/0 192.168.200.1
```

Figure 60 - Updated Rendered Config Part 2

```
interface Ethernet1/1
  no switchport
  ip address 10.101.101.1/30
  no shutdown

interface Ethernet1/2
  no switchport
  ip address 10.254.1.1/30
  no shutdown

interface Ethernet1/3
  switchport
  switchport mode trunk
  switchport trunk native vlan 3000
  switchport trunk allowed vlan 100-102,200-202
  no shutdown

interface Ethernet1/4
  switchport
  switchport mode trunk
  switchport trunk native vlan 3000
  switchport trunk allowed vlan 100-102,200-202
  no shutdown
interface mgmt0
  vrf member management
  ip address 172.31.217.133/23
cisco@onepk:~/Training$ _
```

Figure 61 - Updated Rendered Config Part 3

Lab 2.2.1 – JSON in Python: Working with Dictionaries

Overview

In the previous Python labs, you learned how to create and use Python dictionaries. This lab will continue to explore Python dictionaries by using dictionary built-in methods, but also show dictionaries can be easily serialized and transposed as JSON objects.

Procedures

1. Open **a Linux Terminal** and go to the Python interactive shell
2. Create a Python dictionary

   Issue the following commands:

- `router = {}`
- `router['hostname'] = 'router1'`
- `router['location'] = 'nyc'`
- `router['vrf'] = 'production'`
- `router['domain'] = 'cisco.com'`
- `router['os_version'] = '3.1.2'`

```
>>> router = {}
>>> router['hostname'] = 'router1'
>>> router['location'] = 'nyc'
>>> router['vrf'] = 'production'
>>> router['domain'] = 'cisco.com'
>>> router['os_version'] = '3.1.2'
>>>
```

Figure 62 - Creating a dictionary

3. Print the dictionary

- `print router`

```
>>> print router
{'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'router1', 'location': 'nyc'
, 'vrf': 'production'}
>>>
```

Figure 63 - printing a dictionary

4. Use the `json` module to print the dictionary

   Dictionaries map very well into JSON because they are key/value pairs. Now `print` the dictionary using a JSON pretty print module called `dumps`. The output that you will see is now in the typical JSON format.

   Issue the following commands:

- `import json`
- `print json.dumps(router, indent=4)`

```
>>>
>>> import json
>>>
>>> print json.dumps(router, indent=4)
{
    "os_version": "3.1.2",
    "domain": "cisco.com",
    "hostname": "router1",
    "location": "nyc",
    "vrf": "production"
}
>>>
```

Figure 64 - Pretty Print Python Dictionary

5. Rename the variable from `router` to `router1`.
   Simply assign the contents of router to router1 using the following command:

- `router1 = router`

6. Create two new dictionaries.
   The only changes in these dictionaries from `router1` will be the hostname. The hostnames should be `router2` and `router3`. Reference the figure below for details.

```
>>>
>>> router1 = router
>>>
>>> router2 = {'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'router2', 'lo
cation': 'nyc', 'vrf': 'production'}
>>>
>>> router3 = {'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'router3', 'lo
cation': 'nyc', 'vrf': 'production'}
>>>
```

Figure 65 - Creating two more dictionaries

7. Create a `list` called `neighbors that has two elements: router2` and `router3`.

   Print `neighbors` to verify the contents.

```
>>>
>>> neighbors = [ router2, router3 ]
>>>
>>> print neighbors
[{'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'router2', 'location': 'nyc
', 'vrf': 'production'}, {'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'ro
uter3', 'location': 'nyc', 'vrf': 'production'}]
>>> _
```

Figure 66 - Creating a list of dictionaries

8. Create a new key/value pair in `router1`.

   We will be assigning the list called `neighbors` that was just created to the new key that is also being called `neighbors`. This will be a representation of router1's neighbors.

   Assign the list called `neighbors` to the new key. Pretty print `router1` to see the new output.

- `router1['neighbors'] = neighbors`
- `print json.dumps( router1, indent=4)`

```
>>>
>>> router1['neighbors'] = neighbors
>>>
>>> print json.dumps(router1, indent=4)
{
    "neighbors": [
        {
            "os_version": "3.1.2",
            "domain": "cisco.com",
            "hostname": "router2",
            "location": "nyc",
            "vrf": "production"
        },
        {
            "os_version": "3.1.2",
            "domain": "cisco.com",
            "hostname": "router3",
            "location": "nyc",
            "vrf": "production"
        }
    ],
    "domain": "cisco.com",
    "hostname": "router1",
    "os_version": "3.1.2",
    "location": "nyc",
    "vrf": "production"
}
>>>
```

Figure 67 - Creating and printing a nested dictionary

As you can see `router 1` is still a dictionary, but one of its values is a list that contains dictionaries. Thus, `router1` is a dictionary in which one of its keys (`neighbors`) contains a list that happens to be a list of dictionaries. Does this make sense?

How would you print the hostname of router2?

- `print router1['neighbors'][0]['hostname']`

9. Print all key/value pairs in `router1` using the built-in method called `iteritems`.

`iteritems` allows you to access keys and values in parallel so you don't have to loop through just using the `keys` built-in method.

```
for key, value in router1.iteritems():
    print 'KEY: ', key
    print 'VALUE: ', value
    print '=' * 20
```

```
>>> for key, value in router1.iteritems():
...     print 'KEY:', key
...     print 'VALUE', value
...     print '=' * 20
...
KEY: neighbors
VALUE [{'os_version': '3.1.2', 'domain': 'cisco.com', 'hostname': 'router2', 'lo
cation': 'nyc', 'vrf': 'production'}, {'os_version': '3.1.2', 'domain': 'cisco.c
om', 'hostname': 'router3', 'location': 'nyc', 'vrf': 'production'}]
====================
KEY: domain
VALUE cisco.com
====================
KEY: hostname
VALUE router1
====================
KEY: os_version
VALUE 3.1.2
====================
KEY: location
VALUE nyc
====================
KEY: vrf
VALUE production
====================
>>>
```

Figure 68 - Using dictionary built-in method iteritems

10. Using one or more loops, print the `hostname` of each of the `router1` `neighbors`.

    You will loop through the `router1` key/value pairs using `iteritems` looking for when `key` is equal to `neighbors`. You will then loop through `neighbors` and its elements to then `print` the `hostname` of each neighbor.

    Note: remember the two neighbors are stored in a `list`. This means `value` when `key` is equal to `neighbors` will be a `list` and each element in the `list` is a `dictionary`.

    Feel free to use additional print statements as you see fit to better understand this sequence.

```
for key,value in router1.iteritems():
    if key == 'neighbors':
        for each in value:
            for k,v in each.iteritems():
                if k == 'hostname':
                    print v
```

```
>>>
>>> for key,value in router1.iteritems():
...     if key == 'neighbors':
...         for each in value:
...             for k,v in each.iteritems():
...                 if k == 'hostname':
...                     print v
...
router2
router3
```

Figure 69 - Looping through a nested dictionary using conditionals and iteritems

Here is an alternative more optimal option:

```
for key,value in router1.iteritems():
    if key == 'neighbors':
        for each in value:
            print each['hostname']
```

```
>>>
>>> for key,value in router1.iteritems():
...     if key == 'neighbors':
...         for each in value:
...             print each['hostname']
...
router2
router3
```

Lab 3.1.1 – NX-API Sandbox

Overview

This lab covers an introduction to NX-API by using the NX-API sandbox that exists locally on each Nexus 9000. The sandbox environment is accessed through a web server on each switch and offers the user the ability to test out NX-API calls without a programming environment. It's great for learning, but also very helpful for testing and verification as you are building scripts and other applications.

Procedures

1. If it hasn't been done yet, you should enable NX-API
    - SSH to the Nexus 9000 assigned to you
    - Enter configuration mode and enable the `nxapi` feature
        - `config t`
        - `feature nxapi`

```
N9K1# config t
Enter configuration commands, one per line.  End with CNTL/Z.
N9K1(config)#
N9K1(config)# feature nxapi
N9K1(config)#
N9K1(config)# nxapi ?
  certificate  Https certificate configuration
  http         Http configuration
  https        Https configuration
```

Figure 70 - enable nxapi

- ***Do NOT change the default http configuration.***

2. Once NX-API is enabled, open the Firefox web browser and go to `http://` ***`(IP address of your switch)`***

- Login using the switch credentials
    - Username: `admin`
    - Password: `cisco123`

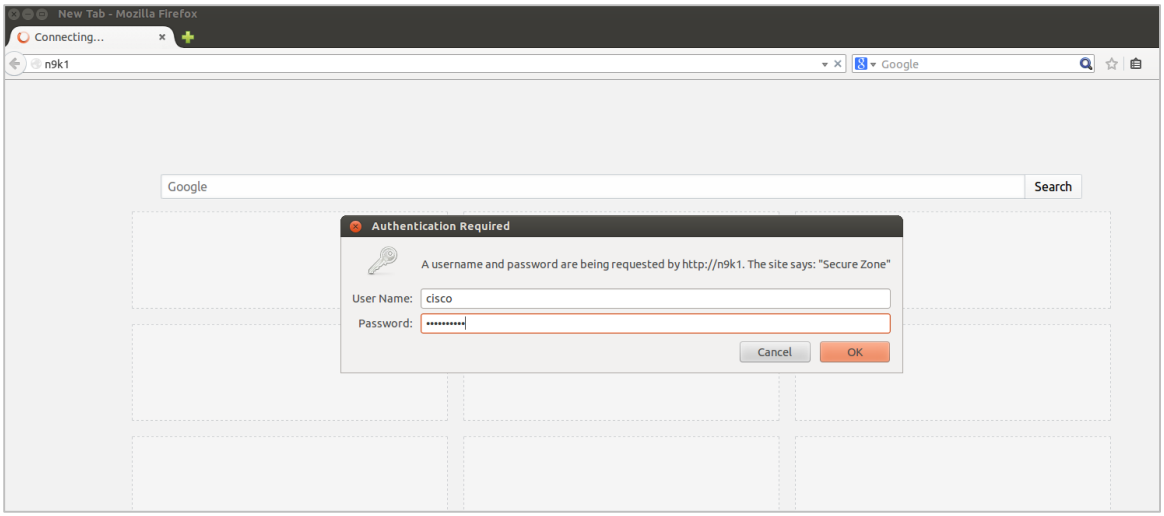

Figure 71 - NXAPI sandbox login

You will then see the following screen:

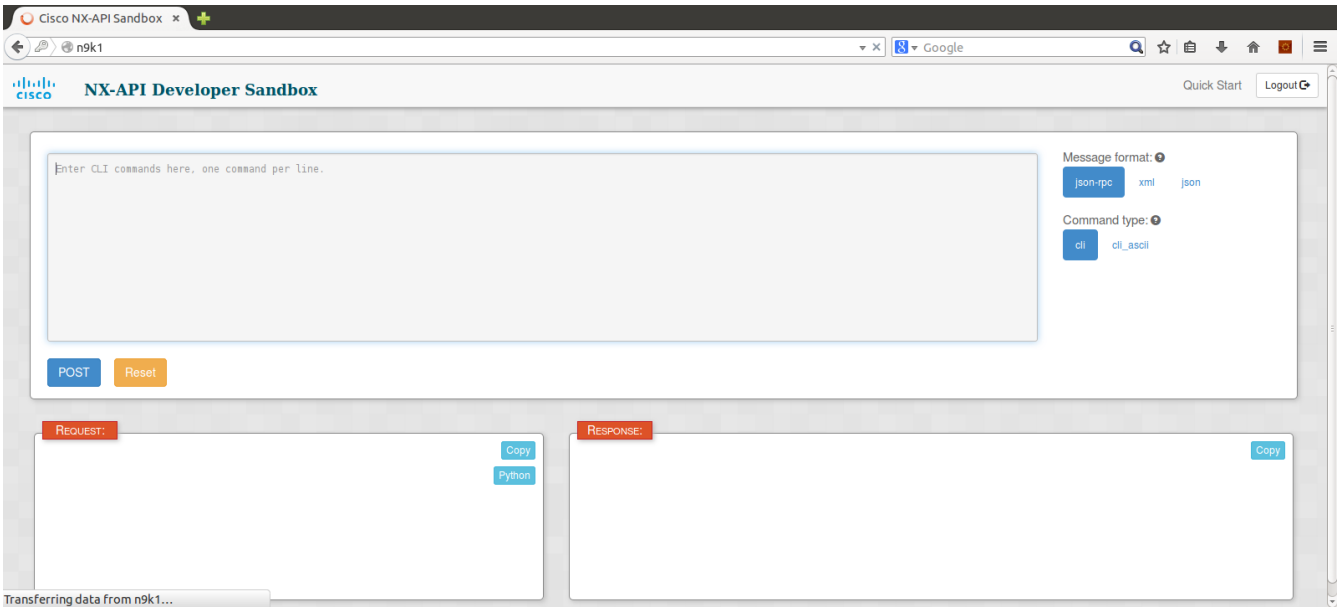

Figure 72 - NXAPI sandbox

3. Browse the NX-API sandbox and take a look at the documentation
4. Enter the command "`show version`"
5. Change the Message Format to `XML`
    - `type: cli_show`
    - `output_format: xml`
6. Click the **POST Request** button

- This will use the default values and send the ‘`show version`’ command to the switch. You will see the following output which shows the XML data that the switch returns from an API call that is wrapping the command ‘`show version`’

Format Message

```
<?xml version="1.0" encoding="UTF-8"?>
<ins_api>
  <type>cli_show</type>
  <version>0.1</version>
  <sid>eoc</sid>
  <outputs>
    <output>
      <body>
      <header_str>Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
Copyright (C) 2002-2014, Cisco and/or its affiliates.
All rights reserved.
The copyrights to certain works contained in this software are
owned by other third parties and used and distributed under their own
licenses, such as open source.  This software is provided "as is," and unless
otherwise stated, there is no warranty, express or implied, including but not
limited to warranties of merchantability and fitness for a particular purpose.
Certain components of this software are licensed under
the GNU General Public License (GPL) version 2.0 or
GNU General Public License (GPL) version 3.0  or the GNU
Lesser General Public License (LGPL) Version 2.1 or
Lesser General Public License (LGPL) Version 2.0.
A copy of each such license is available at
http://www.opensource.org/licenses/gpl-2.0.php and
http://opensource.org/licenses/gpl-3.0.html and
http://www.opensource.org/licenses/lgpl-2.1.php and
http://www.gnu.org/licenses/old-licenses/library.txt.
```

Figure 73 - 'show version' sandbox xml output part 1

```
</header_str>
      <bios_ver_str>07.06</bios_ver_str>
      <kickstart_ver_str>6.1(2)I2(1)</kickstart_ver_str>
      <bios_cmpl_time>03/02/2014</bios_cmpl_time>
      <kick_file_name>bootflash:///n9000-dk9.6.1.2.I2.1.bin</kick_file_name>
      <kick_cmpl_time> 3/15/2014 19:00:00</kick_cmpl_time>
      <kick_tmstmp>03/16/2014 04:26:07</kick_tmstmp>
      <chassis_id>Nexus9000 C9396PX Chassis</chassis_id>
      <cpu_name>Intel(R) Core(TM) i3-3227U C</cpu_name>
      <memory>16402720</memory>
      <mem_type>kB</mem_type>
      <proc_board_id>SAL1819S6BE</proc_board_id>
      <host_name>N9K1</host_name>
      <bootflash_size>21693714</bootflash_size>
      <kern_uptm_days>1</kern_uptm_days>
      <kern_uptm_hrs>23</kern_uptm_hrs>
      <kern_uptm_mins>15</kern_uptm_mins>
      <kern_uptm_secs>50</kern_uptm_secs>
      <rr_reason>Unknown</rr_reason>
      <rr_sys_ver>6.1(2)I2(1)</rr_sys_ver>
      <rr_service/>
      <manufacturer>Cisco Systems, Inc.</manufacturer>
     </body>
      <input>show version</input>
      <msg>Success</msg>
      <code>200</code>
    </output>
  </outputs>
</ins_api>
```

Figure 74 - 'show version' sandbox xml output part 2

7. Change the `output_format` to JSON and issue the same request.

This response should look familiar after working with Python dictionaries. This is the response from the switch when JSON is selected as the `output_format`.

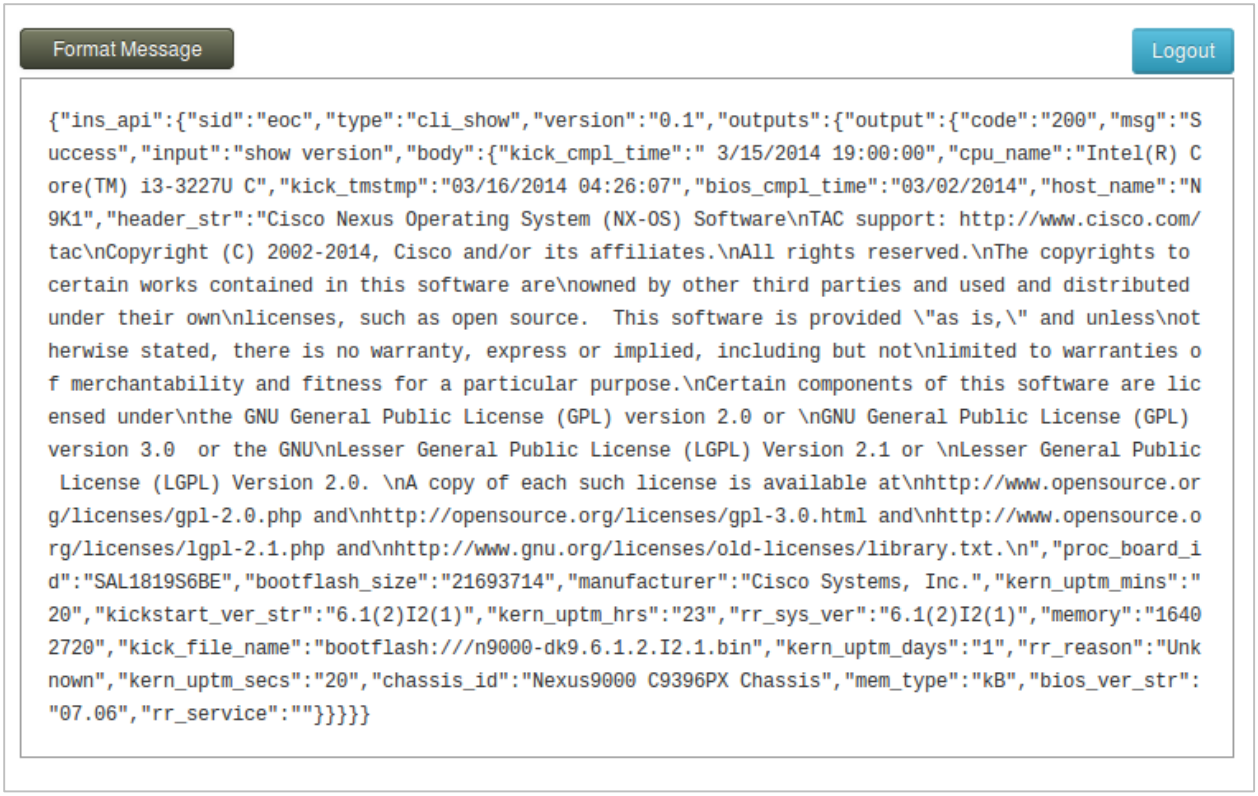

Figure 75 - 'show version' sandbox json output

By using the NX-API sandbox, you can get a feel for how the device is sending data back to the user who is using the API. We will use XML for our labs, but we will be converting the XML to dictionaries (JSON) due to the robustness of the XML interface. Not all commands are supported natively from the device in JSON.

8. Do the same thing for JSON-RPC

9. Configure a new VLAN with a VLAN ID and name using the NX-API sandbox

   Follow these steps:

   - Change the `format` to `xml`
   - Change the `type` to `cli_conf`
   - Change the input to be: `config t ; vlan 100 ; name my_first_vlan ;`
     - You can also do 1 command per line without semi colons
   - Note: keep the spaces in before/after the semi colons
   - Press the **POST Request** button

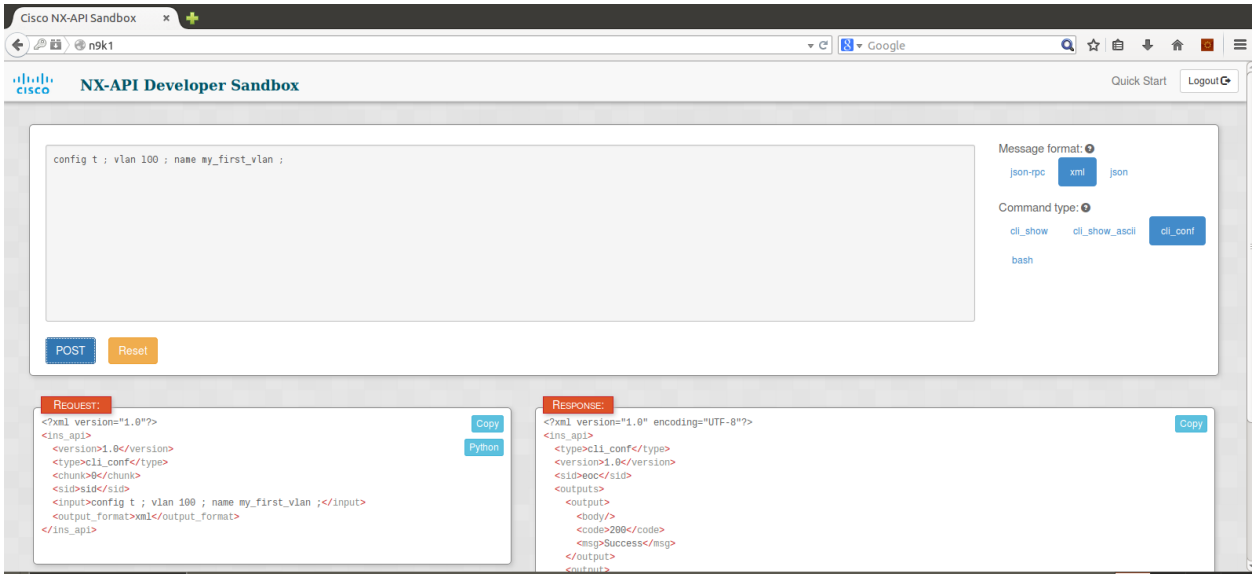

Figure 76 - Config a VLAN using NXAPI

Notice the code 200 and message of Success. The codes are standard HTTP response codes – you know if the command failed by the code. Try entering invalid commands and see what the body, code, and msg come back with. The body will have some context on what the error is.

10. Validate the creation of VLAN 100 using the NX-API sandbox

    Follow these steps:

    - Change the `type` to `cli_show`
    - Change the `input` to `show vlan id 100`
    - Press the **POST Request** button

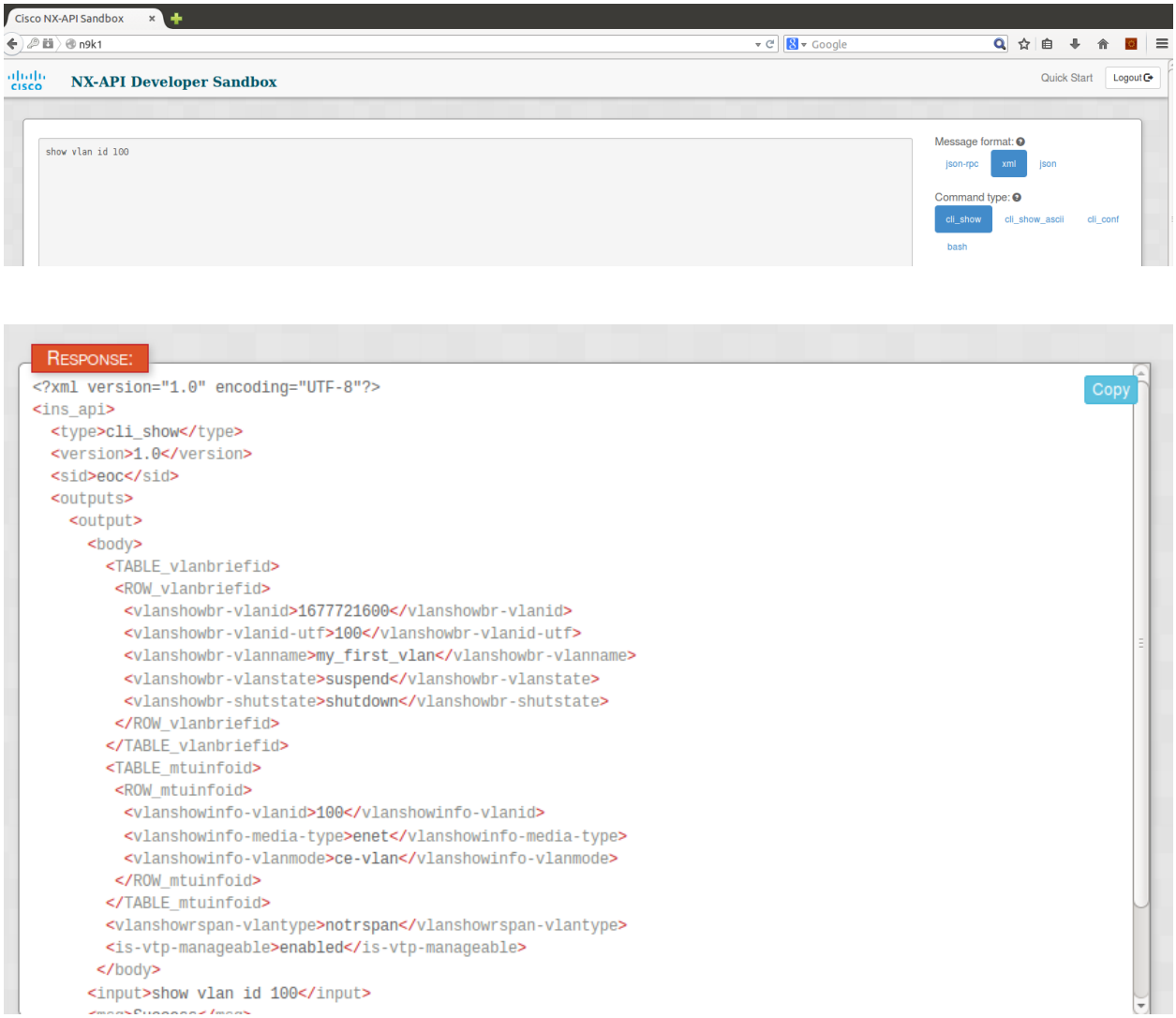

Figure 77 - Verifying VLAN change

You can also go back to the command line to validate if you wish.

11. Try the same command for a VLAN that does not exist using the NX-API:

- `show vlan id 123`

What happens? What is the msg? What is the code? How would you validate a VLAN is created?

12. Continue to explore the NX-API sandbox by issuing `show` and `config` commands

Lab 3.1.2 – NX-API on the Python Shell

Overview

After learning about the NX-API sandbox and how data is returned to a user from a Nexus 9000, the next step is working with NX-API in Python. In this lab, you will use the Python Interactive terminal to extract data from and make changes to a Nexus 9000 using NX-API.

Procedures

1. Make sure you have the NX-API sandbox environment still open in FireFox
2. Open a **Terminal Window** and enter the Python Interactive Shell
3. Import the device module to be used for NX-API

   Issue the following command:

- `from device import Device`

```
cisco@onepk:~$ python
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from device import Device
>>>
```

Figure 78 - import NXAPI Device module

The `Device` module is a class in another file that controls connectivity to/from devices running NX-API. It streamlines the process for getting started with NX-API. Do not worry about what is inside `device.py`, but if you are curious, go have a look. The file is in the `Training` directory.

4. Create a new `Device` object and assign it to a new variable called `sw1`

   This is the process for creating a new variable of type `Device` rather than of native types such as `string` or `integer`.

   When a variable of type `Device` is created, up to three variables: username (`username`), password (`password`), and IP address (`ip`), can be sent to initialize the object. The default username and password are already correct, so they

are not needed for this lab. The IP is also pre-configured with a default, but can be still be used for the lab to understand how it is used.

- `sw1 = Device(ip='DEVICE IP ADDRESS', username='admin', password='cisco123')`

```
>>> sw1 = Device(ip='172.31.217.133', username='admin', password='cisco123')
>>>
```

Figure 79 - Create a var object of type Device

5. Open a connection to the device. This one is deceiving because NX-API is stateless and it's not really opening a connection to the device. In reality, for this particular module, the username/password is being loaded when the `open` method is called. Whenever you use 3rd party modules, you'll need to understand the process of using them.

   Issue the following command:

- `sw1.open()`

   The full sequence will look like this:

```
>>> sw1 = Device(ip='172.31.217.133', username='admin', password='cisco123')
>>>
>>> sw1.open()
```

Figure 80 - Use required open method for this Device module

6. Make your first API call using NX-API

- The CLI command that we want to capture is: `show interface Ethernet1/1`

   Issue the following commands to make an NX-API call:

- `command = sw1.show('show interface Ethernet1/1')`
- `print command`

```
>>>
>>> command = sw1.show('show interface Ethernet1/1')
>>>
>>> print command
(<httplib.HTTPMessage instance at 0x2b821b8>, '<?xml version="1.0"?>\n<ins_api>\n  <typ
e>cli_show</type>\n  <version>0.1</version>\n  <sid>eoc</sid>\n  <outputs>\n    <output
>\n      <body>\n        <TABLE_interface>\n          <ROW_interface>\n          <interface
>Ethernet1/1</interface>\n          <state>down</state>\n          <state_rsn_desc>XCVR n
ot inserted</state_rsn_desc>\n          <admin_state>up</admin_state>\n          <share_s
tate>Dedicated</share_state>\n          <eth_hw_desc>1000/10000 Ethernet</eth_hw_desc>\n
          <eth_hw_addr>5087.89a1.d8d5</eth_hw_addr>\n          <eth_bia_addr>5087.89a1.d8
d6</eth_bia_addr>\n          <eth_ip_addr>10.101.101.1</eth_ip_addr>\n          <eth_ip_m
ask>30</eth_ip_mask>\n          <eth_ip_prefix>10.</eth_ip_prefix>\n          <eth_mtu>15
00</eth_mtu>\n          <eth_bw>10000000</eth_bw>\n          <eth_dly>10</eth_dly>\n
    <eth_reliability>255</eth_reliability>\n          <eth_txload>1</eth_txload>\n
   <eth_rxload>1</eth_rxload>\n          <medium>broadcast</medium>\n          <eth_duple
x>auto</eth_duplex>\n          <eth_speed>auto-speed</eth_speed>\n          <eth_beacon>o
ff</eth_beacon>\n          <eth_autoneg>on</eth_autoneg>\n          <eth_in_flowctrl>off<
/eth_in_flowctrl>\n          <eth_out_flowctrl>off</eth_out_flowctrl>\n          <eth_mdi
x>off</eth_mdix>\n          <eth_swt_monitor>off</eth_swt_monitor>\n          <eth_ethert
ype>0x8100</eth_ethertype>\n          <eth_eee_state>n/a</eth_eee_state>\n          <eth_
link_flapped>never</eth_link_flapped>\n          <eth_clear_counters>never</eth_clear_co
unters>\n          <eth_reset_cntr>0</eth_reset_cntr>\n          <eth_load_interval1_rx>3
0</eth_load_interval1_rx>\n          <eth_inrate1_bits>0</eth_inrate1_bits>\n          <e
th_inrate1_pkts>0</eth_inrate1_pkts>\n          <eth_load_interval1_tx>30</eth_load_inte
rval1_tx>\n          <eth_outrate1_bits>0</eth_outrate1_bits>\n          <eth_outrate1_pk
ts>0</eth_outrate1_pkts>\n          <eth_inucast>0</eth_inucast>\n          <eth_inmcast>
0</eth_inmcast>\n          <eth_inbcast>0</eth_inbcast>\n          <eth_inpkts>0</eth_inp
kts>\n          <eth_inbytes>0</eth_inbytes>\n          <eth_jumbo_inpkts>0</eth_jumbo_in
pkts>\n          <eth_runts>0</eth_runts>\n          <eth_giants>0</eth_giants>\n
 <eth_crc>0</eth_crc>\n          <eth_nobuf>0</eth_nobuf>\n          <eth_inerr>0</eth_in
err>\n          <eth_frame>0</eth_frame>\n          <eth_overrun>0</eth_overrun>\n
  <eth_underrun>0</eth_underrun>\n          <eth_ignored>0</eth_ignored>\n          <eth_
watchdog>0</eth_watchdog>\n          <eth_bad_eth>0</eth_bad_eth>\n          <eth_bad_pro
to>0</eth_bad_proto>\n          <eth_in_ifdown_drops>0</eth_in_ifdown_drops>\n          <
```

Figure 81 - XML return data from built-in method called show for 'show interface Ethernet1/1'

As you can see this output is in XML and it's not very human readable in this format. Let's fix that.

7. Convert the XML data to a Python Dictionary and then print it.

   Issue these commands:

- `import xmltodict`
  - This module will convert xml return data to a Python dictionary so it is easier to work with.
- `import json`
  - This way you can pretty print the Python dictionary. This isn't a requirement, but greatly simplifies readability.
  - Convert `command` to a dictionary

- `command = sw1.show('show interface Ethernet1/1')`
- `result = xmltodict.parse(command[1])`

    - `parse` is a method of the `xmltodict` module

- `print json.dumps(result, indent=4)`

  Note: the API call (`sw1.show()`)returns a `tuple` of two elements. A `tuple` is like a `list`, but always has the same number of elements. This particular tuple is always two elements. The first element (`command[0]`) is always the http headers and the second element (`command[1]`) is the important return data that needs to be analyzed. Feel free to print command[0] to see the `httplib` object. For this course, you can disregard command[0] – we'll just be working with command[1] all of the time.

```
>>> import xmltodict
>>>
>>> result = xmltodict.parse(command[1])
>>>
>>> import json
>>>
>>> print json.dumps(result, indent=4)
{
    "ins_api": {
        "type": "cli_show",
        "version": "0.1",
        "sid": "eoc",
        "outputs": {
            "output": {
                "body": {
                    "TABLE_interface": {
                        "ROW_interface": {
                            "interface": "Ethernet1/1",
                            "state": "down",
                            "state_rsn_desc": "XCVR not inserted",
                            "admin_state": "up",
                            "share_state": "Dedicated",
                            "eth_hw_desc": "1000/10000 Ethernet",
                            "eth_hw_addr": "5087.89a1.d8d5",
                            "eth_bia_addr": "5087.89a1.d8d6",
                            "eth_ip_addr": "10.101.101.1",
                            "eth_ip_mask": "30",
                            "eth_ip_prefix": "10.",
                            "eth_mtu": "1500",
                            "eth_bw": "10000000",
```

Figure 82 - import xmltodict and convert XML structure to Dictionary

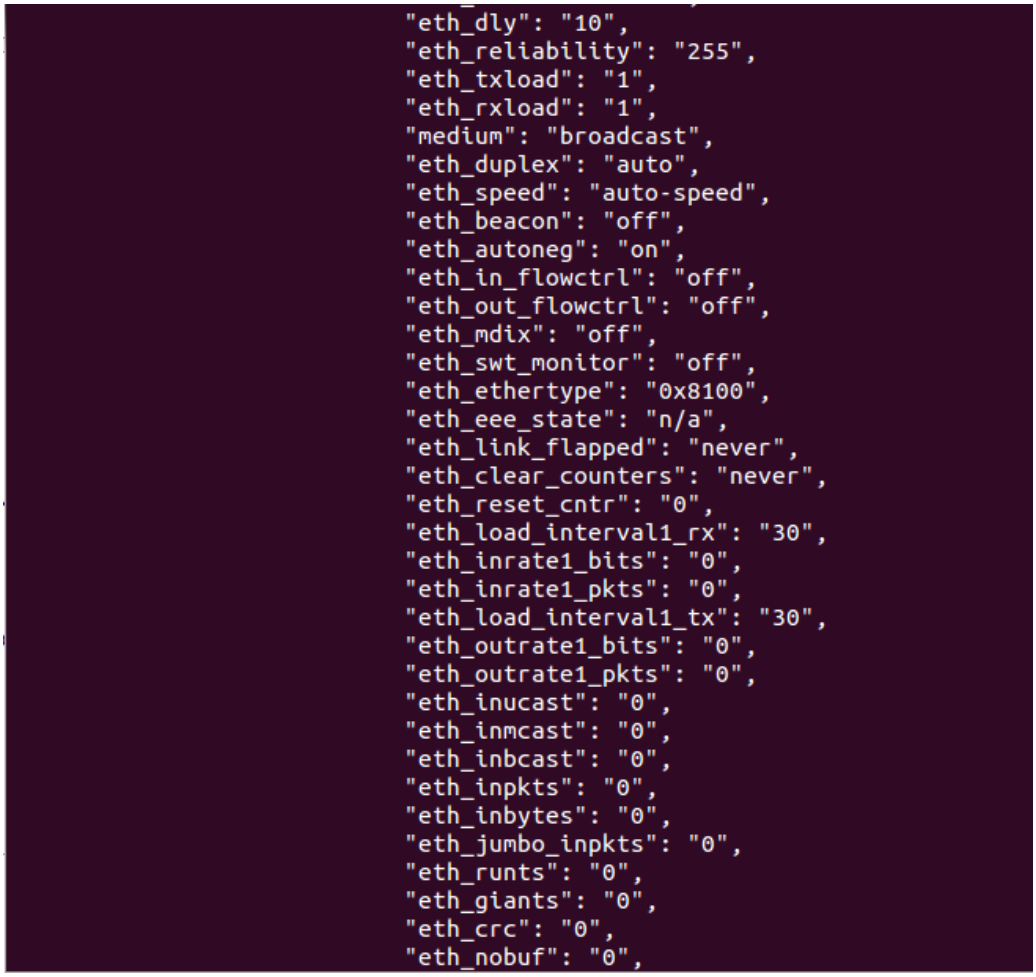

Figure 83 - Python Dictionary Part 2

Figure 84 - Python Dictionary Part 3

Note: the last part of dictionary has the original command, msg (status), and HTTP response code. These are frequently used for error checking.

8. Go the NX-API sandbox and compare this output to the sandbox for the same command: `show interface Ethernet1/1`. Do you now see how the sandbox can be used for testing and verification?
9. You should recall that the output from the previous figures is a nested Python dictionary.
10. Extract the `Ethernet1/1` **IP Address** and **subnet mask** from `result` into two separate values and print them both. Issue the following commands:

- `ip = result['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']['eth_ip_addr']`

- `print ip`

- `mask = result ['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']['eth_ip_mask']`

- `print mask`

- `print ip + '/' + mask`

```
>>>
>>> ip = result['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']['eth_ip_addr']
>>>
>>> print ip
10.101.101.1
>>>
>>> mask = result['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']['eth_ip_mask']
>>>
>>> print mask
30
>>>
>>> print ip + '/' + mask
10.101.101.1/30
>>>
```

Figure 85 - Working with the data returned from the device after XML to dictionary conversion

Please a take a minute to test this and really understand it. This part is absolutely critical to understand if you are going to work with NX-API.

11. Print the VLANs configured on the switch

    You should print them with the VLAN ID on one line and the VLAN name on a second line followed by a line of equal signs (“=”) to denote the next VLAN

    Take a look at the `show vlan` command in the NX-API sandbox. You should get in the habit of cross checking with the sandbox if possible. This helps you get a feel for what the nested dictionary will look like as you start coding. Find out what key/value pairs will be needed to print information about the VLANs using the CLI command: `show vlan`

- After reviewing the NX-API sandbox output, you should see that the keys you will need **per** VLAN are: `vlanshowbr-vlanid-utf` and `vlanshowbr-vlanname`

  Issue the following commands:

- `sh_vlan = sw1.show('show vlan')`
- `sh_vlan_dict = xmltodict.parse(sh_vlan[1])`

```
>>> sh_vlan = sw1.show('show vlan')
>>>
>>> sh_vlan_dict = xmltodict.parse(sh_vlan[1])
>>>
```

Figure 86 - 'show vlan' and NXAPI

- `print json.dumps( sh_vlan_dict, indent=4)`

```
>>> print json.dumps( sh_vlan_dict, indent=4)
{
    "ins_api": {
        "type": "cli_show",
        "version": "0.1",
        "sid": "eoc",
        "outputs": {
            "output": {
                "body": {
                    "TABLE_vlanbrief": {
                        "ROW_vlanbrief": [
                            {
                                "vlanshowbr-vlanid": "16777216",
                                "vlanshowbr-vlanid-utf": "1",
                                "vlanshowbr-vlanname": "default",
                                "vlanshowbr-vlanstate": "active",
                                "vlanshowbr-shutstate": "noshutdown",
                                "vlanshowplist-ifidx": "Ethernet2/3-10"
                            },
                            {
                                "vlanshowbr-vlanid": "167772160",
                                "vlanshowbr-vlanid-utf": "10",
                                "vlanshowbr-vlanname": "web",
                                "vlanshowbr-vlanstate": "active",
                                "vlanshowbr-shutstate": "noshutdown"
                            },
                            {
                                "vlanshowbr-vlanid": "184549376",
                                "vlanshowbr-vlanid-utf": "11",
                                "vlanshowbr-vlanname": "qa",
                                "vlanshowbr-vlanstate": "active",
                                "vlanshowbr-shutstate": "noshutdown"
                            },
```

Figure 87 - 'show vlan' python dictionary snippet

Note: this figure above is only showing 3 of the many VLANs configured. The rest of them follow the same format. The important takeaway from this output is the bracket **(" [")** next to the "`ROW_vlanbrief`" key. This means that `ROW_vlanbrief` is a **`list`** of dictionaries. Working with this dictionary and list combination will be very similar to how you worked with the nested `router1` dictionary that had a `list` of neighbors in the JSON lab.

To understand the list, print only the vlan name for VLAN 10.

- `vlan10_name = sh_vlan_dict['ins_api']['outputs']['output']['body']['TABLE_vlanbrief']['ROW_vlanbrief'][1]['vlanshowbr-vlanname']`

- `print vlan10_name`

```
>>> vlan10_name = sh_vlan_dict['ins_api']['outputs']['output']['body']['TABLE_vlanbrief']
['ROW_vlanbrief'][1]['vlanshowbr-vlanname']
>>>
>>> print vlan10_name
web
```

Figure 88 - printing a specific VLAN name from the nested dictionary

Now, finally print the VLAN ID and VLAN NAME for each VLAN.

- `vlans = sh_vlan_dict['ins_api']['outputs']['output']['body']['TABLE_vlanbrief']['ROW_vlanbrief']`
- `for each in vlans:`
- `    print 'VLAN ID: ', each['vlanshowbr-vlanid-utf']`
- `    print 'VLAN NAME: ', each['vlanshowbr-vlanname']`
- `    print '=' * 25`

Don't forget `vlans` is a `list`!

```
>>> vlans = sh_vlan_dict['ins_api']['outputs']['output']['body']['TABLE_vlanbrief']['ROW_
vlanbrief']
>>>
>>> for each in vlans:
...     print 'VLAN ID: ', each['vlanshowbr-vlanid-utf']
...     print 'VLAN NAME: ', each['vlanshowbr-vlanname']
...     print '=' * 25
...
```

Figure 89 - Print each VLAN ID & NAME

Before looking at the output, please understand the following. In this example, you assigned part of a nested dictionary structure to a new variable called `vlans` and the result is the `list` you saw in the previous step. Breaking things down like this make them easier to work with as you can see.

Final output:

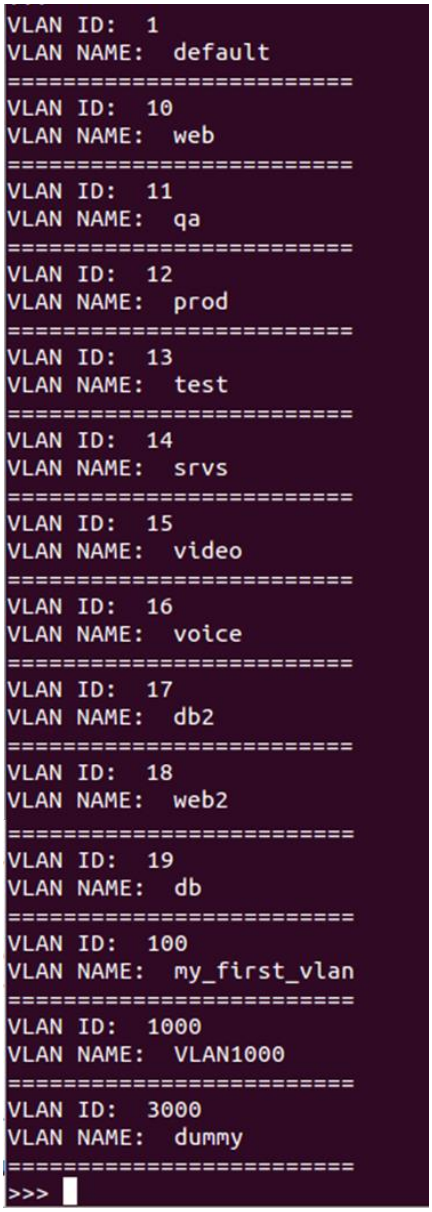

Figure 90 - VLANs output

12. Configure an IP address of `172.19.21.1/24` on `Ethernet1/24`

    Since you can combine multiple commands in an NX-API call, you can do this in 1 API or several API calls. For purposes of this task, you will use 3.

    - Convert the interface to a routed port (`no switchport`)
    - Configure the IP and subnet mask (`ip address 172.19.21.1/24`)
    - Enable the interface (`no shut`)

Issue these commands:

- `push_l3port = sw1.conf('config t ; interface Ethernet1/24 ; no switchport ')`
- `push_l3ip = sw1.conf('config t ; interface Ethernet1/24 ; ip address 172.19.21.1/24 ')`
- `cmd = 'config t ; interface Ethernet1/24 ;'`
- `push_state = sw1.conf(cmd + 'no shutdown')`

```
>>> push_l3port = sw1.conf('config t ; interface Ethernet1/24 ; no switchport ')
>>>
>>> push_l3ip = sw1.conf('config t ; interface Ethernet1/24 ; ip address 172.19.21.1/24 ')
>>>
>>> cmd = 'config t ; interface Ethernet1/24 ; '
>>>
>>> push_state = sw1.conf(cmd + 'no shutdown')
>>>
```

Figure 91 - Pushing a config using NXAPI

The figure above also shows that you can create a string like `cmd` to save from always typing in parameters like `config t` and the interface or particular config section you are working with.

13. Examine the data returned for configuration changes. This is important to verify the changes were successful. Examine the data that `push_state` has assigned to it after pushing the change.

    You will notice that the response code and msg are still returned even when pushing configuration changes using NX-API.

```
>>> push_state = sw1.conf(cmd + 'no shutdown')
>>>
>>> push_state_dict = xmltodict.parse(push_state[1])
>>>
>>> print json.dumps ( push_state_dict, indent=4)
{
    "ins_api": {
        "type": "cli_conf",
        "version": "0.1",
        "sid": "eoc",
        "outputs": {
            "output": {
                "body": null,
                "code": "200",
                "msg": "Success"
            }
        }
    }
}
>>>
```

Figure 92 - Return data even for config changes (push)

14. Log back into the CLI of the Nexus 9000 and verify the configuration was done properly.

```
N9K1# show run interface ethernet 1/24

!Command: show running-config interface Ethernet1/24
!Time: Mon Nov 17 14:34:22 2014

version 6.1(2)I3(1)

interface Ethernet1/24
  no switchport
  ip address 172.19.21.1/24
  no shutdown

N9K1#
```

Lab 3.1.3 – Writing a NX-API Script

Overview

In previous labs including the Intro to Python lab, you worked with “device facts” that were device attributes such as OS version, hostname, neighbors, etc. In this lab, you will create a script that gathers real device facts from the device itself. The device facts that should be gathered include hostname, OS version, management interface data (IP, interface name, speed, duplex), make/model of device, last reboot reason, uptime, bootflash, and serial numbers for all components within the device.

Procedures

1. SSH to the Nexus 9000
2. Review the output of the following commands from the CLI:
    - Show hardware
    - Show interface mgmt0
3. Review the output from each command in the NX-API sandbox
4. Test as needed from the Python Shell.
   You may want to convert the output from show hardware and show interface mgmt0 to Python dictionaries in the Python shell before writing the script to better understand the manipulation needed while writing the script.
5. Write a script to gather the facts stated above. In an effort to keep the script modular, use one function per command in addition to function `main()`. Each function should return a Python dictionary. The two Python dictionaries should be “joined” together in `main()` to create the final dictionary that should be called facts.

The output of the script should look like the following by the end of this lab:

```
cisco@onepk:~/Training$ python nxapi_facts.py
{
    "serial_numbers": {
        "DCB1809X07K": "N9K-PAC-650W",
        "DCB1809X07J": "N9K-PAC-650W",
        "SAL1819S6BE": "N9K-C9396PX",
        "SAL1807M5A1": "N9K-M12PQ"
    },
    "uptime": "2 day(s) 20 hour(s) 24 min(s) 36 sec(s)",
    "hostname": "N9K1",
    "bootflash": "21693714",
    "os_version": "6.1(2)I2(1)",
    "mgmt_intf": {
        "duplex": "half",
        "speed": "100 Mb/s",
        "ip_addr": "192.168.200.50/24",
        "name": "mgmt0"
    },
    "memory": "16402720kB",
    "last_reboot_reason": "Unknown",
    "type": "Nexus9000 C9396PX Chassis"
}
cisco@onepk:~/Training$
```

Figure 93 - Example script output

6. As you write the script, continue to build and test. Writing in Python and not needing to "compile" programs makes this very fast and enables you to iterate quickly.

The rest of the steps outline how to build and test the script.

7. Write the show interface function and test it (without the show hardware function).

   First, create a new empty file called `nxapi_facts.py` and store it in the `classfiles` directory.

   Use the following figure for guidance as needed.

nxapi_facts.py

```
#!/usr/bin/env python

import xmltodict
import json
from device import Device

def show_intf_mgmt(sw):

    getdata = sw.show('show interface mgmt0')

    show_intf_dict = xmltodict.parse(getdata[1])

    data = show_intf_dict['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']

    ip = data['eth_ip_addr']
    mask = data['eth_ip_mask']
    name = data['interface']
    speed = data['eth_speed']
    duplex = data['eth_duplex']

    mgmt_dict = { 'ip_addr': ip+'/'+mask, 'name':name, 'speed': speed, 'duplex': duplex }

    print json.dumps(mgmt_dict, indent=4)

    return mgmt_dict

def main():

    switch = Device(ip='172.31.217.138', username='admin', password='cisco123')
    switch.open()

    intf = show_intf_mgmt(switch)

if __name__ == "__main__":
    main()
```

Figure 94 - Writing a function in a script

From just this one part of the script, you can gain valuable output to ensure everything is working as expected:

Open a Linux Terminal and navigate to the `classfiles` directory. Run the script.

```
cisco@onepk:~/Training$ python nxapi_facts.py
{
    "duplex": "half",
    "speed": "100 Mb/s",
    "ip_addr": "192.168.200.50/24",
    "name": "mgmt0"
}
cisco@onepk:~/Training$
```

Figure 95 - Testing the function using the command line

8. Write the show hardware function and make the appropriate changes to `main()` to test it. The only change to `main()` is making sure you call `show_hardware()`

Use the following figure for guidance as needed.

```
63  def main():
64
65      switch = Device(ip='172.31.217.138', username='admin', password='cisco123')
66      switch.open()
67
68      intf = show_intf_mgmt(switch)
69
70      hw = show_hardware(switch)
71
72  if __name__ == "__main__":
73      main()
74
```

Figure 96 - make changes to main() and call the to be created show_hardware() function

Use the following figure for guidance as needed creating `show_hardware()`.

```
7  def show_hardware(sw):
8
9      getdata = sw.show('show hardware')
10
11     show_hw_dict = xmltodict.parse(getdata[1])
12     data = show_hw_dict['ins_api']['outputs']['output']['body']
13
14     hw_dict = {}
15     hw_dict['os_version'] = data['kickstart_ver_str']
16     hw_dict['type'] = data['chassis_id']
17     hw_dict['memory'] = data['memory'] + data['mem_type']
18     hw_dict['hostname'] = data['host_name']
19     hw_dict['bootflash'] = data['bootflash_size']
20     hw_dict['last_reboot_reason'] = data['rr_reason']
21     hw_dict['uptime'] = '{} day(s) {} hour(s) {} min(s) {} sec(s)'.format(data['kern_uptm_days'], \
22         data['kern_uptm_hrs'],data['kern_uptm_mins'],data['kern_uptm_secs'])
23
24     ser_nums = {}
25     ser_nums_data = show_hw_dict['ins_api']['outputs']['output']['body']['TABLE_slot'] \
26         ['ROW_slot']['TABLE_slot_info']['ROW_slot_info']
27
28     for each in ser_nums_data:
29
30         if 'serial_num' in each.keys():                #check sandbox output.  fans would cause error here
31             key = each['serial_num']
32             ser_nums[key] = each['model_num']
33
34     hw_dict['serial_numbers'] = ser_nums
35
36     print json.dumps(hw_dict, indent=4)
37
38     return hw_dict
39
```

Figure 97 - show_hardware() function

Notes:

- The example is using a nested a dictionary. There is the main dictionary being used and also separate dictionary being used for the serial numbers that is then assigned to the main dictionary.
- There is a new built-in method called `format` being used on strings in line 21. This streamlines the way to use variables in assignments or print statements.
- Line 30 could have used a "`try`" or `get()` statement instead, but those were not covered in class, so this is just making sure a serial number exists for the components. There is no serial numbers for the fans, so if you remove this and run the script, you will get a dictionary `key` error.

Checking the output:

```
cisco@onepk:~/Training$ python nxapi_facts.py
{
    "duplex": "half",
    "speed": "100 Mb/s",
    "ip_addr": "192.168.200.50/24",
    "name": "mgmt0"
}
{
    "serial_numbers": {
        "DCB1809X07K": "N9K-PAC-650W",
        "DCB1809X07J": "N9K-PAC-650W",
        "SAL1819S6BE": "N9K-C9396PX",
        "SAL1807M5A1": "N9K-M12PQ"
    },
    "uptime": "2 day(s) 19 hour(s) 35 min(s) 51 sec(s)",
    "hostname": "N9K1",
    "bootflash": "21693714",
    "os_version": "6.1(2)I2(1)",
    "memory": "16402720kB",
    "last_reboot_reason": "Unknown",
    "type": "Nexus9000 C9396PX Chassis"
}
cisco@onepk:~/Training$
```

Figure 98 - Running the script

9. Create the `facts` dictionary in `main()` and update it with the other two dictionaries because at this point they are still two independent dictionaries. We want ONE consolidated dictionary.

- Remove or comment out the existing `print` statements – one from each function.

- Rather than use the `update` method for both, create a new `key` for everything being returned from the `show_intf_mgmt()` function. This will simplify the readability of the final output as you will see soon.

Use the following figure for guidance as needed.

```
57  def main():
58
59      switch = Device(ip='172.31.217.138', username='admin', password='cisco123')
60      switch.open()
61
62      facts = {}
63
64      intf = show_intf_mgmt(switch)
65      facts['mgmt_intf'] = intf
66
67      hw = show_hardware(switch)
68      facts.update(hw)
69
70      print json.dumps( facts, indent=4)
71
72  if __name__ == "__main__":
73      main()
74
```

Figure 99 - Massasing the dictionaries and data to be output to terminal

New output:

```
cisco@onepk:~/Training$ python nxapi_facts.py
{
    "serial_numbers": {
        "DCB1809X07K": "N9K-PAC-650W",
        "DCB1809X07J": "N9K-PAC-650W",
        "SAL1819S6BE": "N9K-C9396PX",
        "SAL1807M5A1": "N9K-M12PQ"
    },
    "uptime": "2 day(s) 20 hour(s) 24 min(s) 36 sec(s)",
    "hostname": "N9K1",
    "bootflash": "21693714",
    "os_version": "6.1(2)I2(1)",
    "mgmt_intf": {
        "duplex": "half",
        "speed": "100 Mb/s",
        "ip_addr": "192.168.200.50/24",
        "name": "mgmt0"
    },
    "memory": "16402720kB",
    "last_reboot_reason": "Unknown",
    "type": "Nexus9000 C9396PX Chassis"
}
cisco@onepk:~/Training$
```

Figure 100 - Verify changes to script

10. Update the script so that it now accepts command line arguments when running the script from the Linux terminal.

- Remove or comment out the print statement in line 70.

- You should be able to send a valid dictionary `key` for `facts` and the appropriate `value` should be printed. If a `key` isn't given as an argument, continue to print the full `facts` dictionary.

  Note: don't forget to: `import sys`

  Remember, this is needed to use `sys.argv`

```
57  def main():
58
59      switch = Device(ip='172.31.217.138', username='admin', password='cisco123')
60      switch.open()
61
62      facts = {}
63
64      intf = show_intf_mgmt(switch)
65      facts['mgmt_intf'] = intf
66
67      hw = show_hardware(switch)
68      facts.update(hw)
69
70      args = sys.argv
71
72      if len(args) == 1:
73          print json.dumps(facts, indent=4)
74      else:
75          if args[1] in facts.keys():
76              print args[1].upper() + ': ' + json.dumps(facts[args[1]],indent=4)
77          else:
78              print 'Invalid Key.  Try again.'
79
80  if __name__ == "__main__":
81      main()
82
```

Figure 101 - Using command line arguments inside the script

Few sample outputs:

```
cisco@onepk:~/Training$ python nxapi_facts.py hostname
HOSTNAME: "N9K1"
cisco@onepk:~/Training$
```

Figure 102 – Example output key=hostname

```
cisco@onepk:~/Training$ python nxapi_facts.py serial_numbers
SERIAL_NUMBERS: {
    "DCB1809X07K": "N9K-PAC-650W",
    "DCB1809X07J": "N9K-PAC-650W",
    "SAL1819S6BE": "N9K-C9396PX",
    "SAL1807M5A1": "N9K-M12PQ"
}
cisco@onepk:~/Training$
```

Figure 103 - Example output key=serial_numbers

```
cisco@onepk:~/Training$ python nxapi_facts.py XXXX
Invalid Key.  Try again.
cisco@onepk:~/Training$
```

Figure 104 - Example output key=XXXX (invalid)

```
cisco@onepk:~/Training$ python nxapi_facts.py mgmt_intf
MGMT_INTF: {
    "duplex": "half",
    "speed": "100 Mb/s",
    "ip_addr": "192.168.200.50/24",
    "name": "mgmt0"
}
cisco@onepk:~/Training$
```

Figure 105 - Example output key=mgmt_intf

Lab 4.1.1 – Using Ansible as a Template Build Engine

Overview

You already learned how to use Jinja2 and YAML files and created a configuration file by rendering a template with a variables file using a demo python script. In this lab, you will see how Ansible can be used as a template build engine building on what you've done in previous labs. Using Ansible as a template build engine will further automate the process of creating configuration files off of base and host specific templates.

If for any reason you close the terminal window during this portion of the lab, you will have to rerun this command in the new terminal window you open

`source env-setup` in the `/home/cisco/Training/Ansible` directory

```
cisco@onepk:~/Training/Ansible$ source env-setup
cisco@onepk:~/Training/Ansible$
cisco@onepk:~/Training/Ansible$
```

Procedures

1. Open a Terminal Window
2. Navigate to the `/home/cisco/Training/Ansible` directory and issue an `ls` to see the contents of the directory

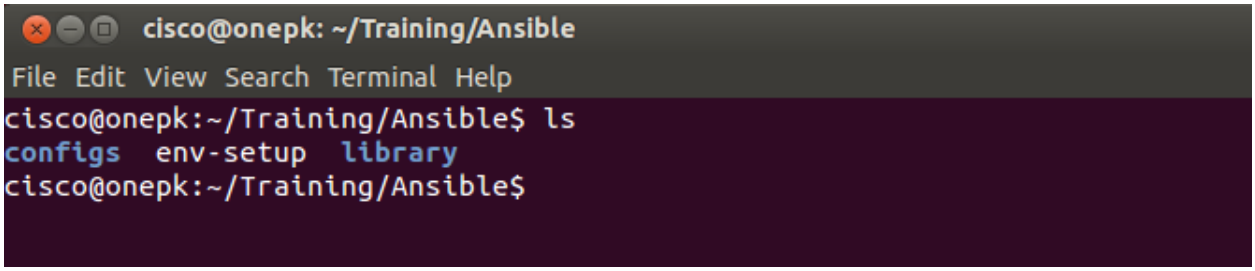

Figure 106 - Navigate to Ansible directory

You should see at least 2 directories and 1 file. The configs directory will be used in this lab. It should be empty. The library directory won't be used directly, but is where custom Ansible modules are stored. The environment setup contains a bash command to let Ansible know where to locate the hosts file that will be used.

3. Source the env-setup file

   `source` will execute the commands inside the file. This is just running a command to ensure the hosts file is set properly for your Ansible directory.

   Issue the command:

- `source env-setup`

```
cisco@onepk:~/Training/Ansible$ source env-setup
cisco@onepk:~/Training/Ansible$
cisco@onepk:~/Training/Ansible$
```

Figure 107 - Setup Ansible environment

4. Create 3 new directories in the Ansible directory and verify they are created

- `mkdir host_vars`
- `mkdir group_vars`
- `mkdir Templates`

```
cisco@onepk:~/Training/Ansible$ mkdir host_vars
cisco@onepk:~/Training/Ansible$ mkdir group_vars
cisco@onepk:~/Training/Ansible$ mkdir Templates
cisco@onepk:~/Training/Ansible$
cisco@onepk:~/Training/Ansible$ ls
configs  env-setup  group_vars  host_vars  library  Templates
cisco@onepk:~/Training/Ansible$
cisco@onepk:~/Training/Ansible$
```

Figure 108 - Create Ansible directories

5. Open **Sublime Text**
6. Open `Training/N9K1_v2.j2` and "**Save As**" `nexus9k.j2` in the **`Ansible/Templates`** directory

- **Remove** all hyphens **("-")** that are next to the "%" symbols in the for and if blocks.
- There are nine "%-" and one "-%". Click Find, Replace "%-" with "%" and "-%" with "%".

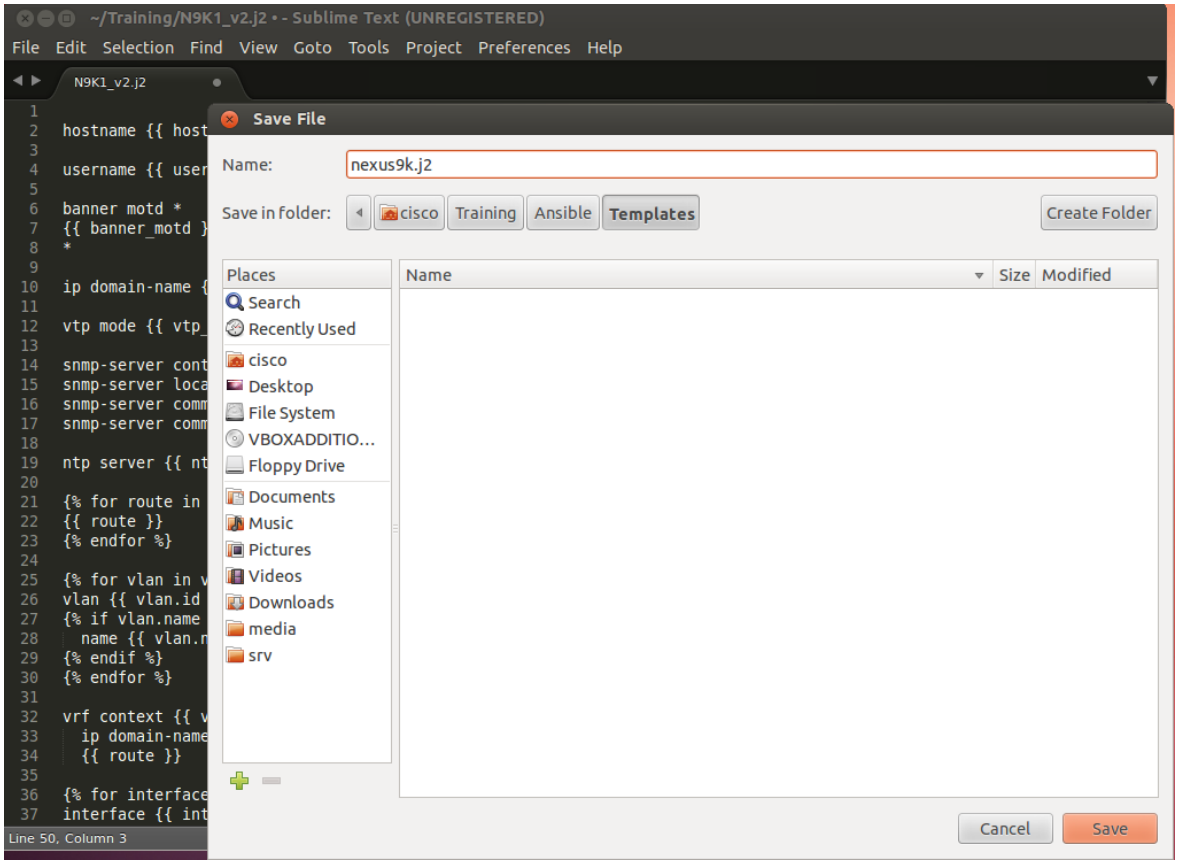

Figure 109 - Create nexus9k.j2

7. In the terminal window issue an `ls` and ensure you see the `nexus9k.j2` template there

- Did you remove all the hyphens in the previous step?

```
cisco@onepk:~/Training/Ansible$ ls Templates/
nexus9k.j2
cisco@onepk:~/Training/Ansible$
```

Figure 110 - Verify nexus9k.2 is in the proper directory

8. Open `Training/N9K1_v2.yml` and "**Save As**" `nexus.yml` in the **`Ansible/group_vars`** directory

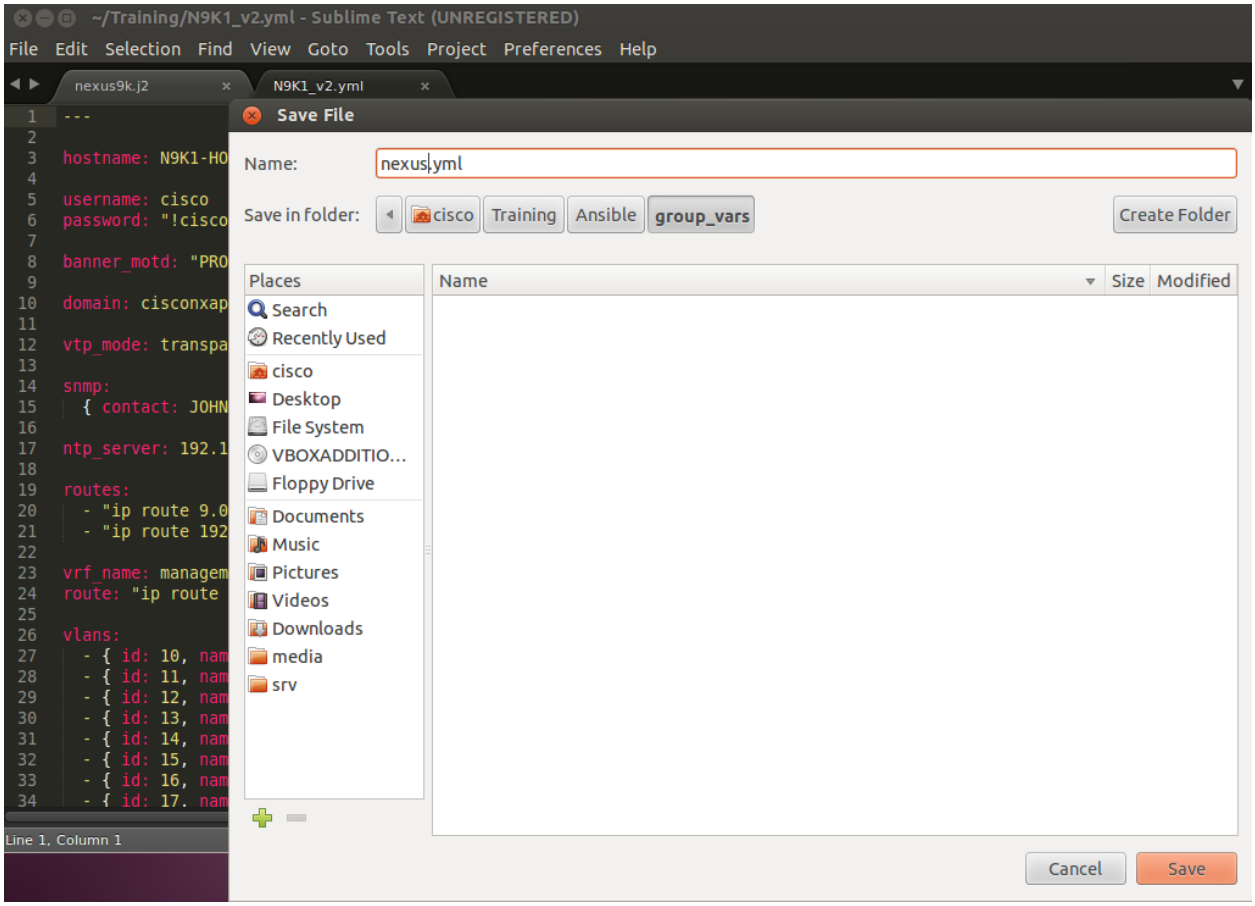

Figure 111 - Create nexus.yml vars file

9. In the terminal window issue an `ls` and ensure you see the `nexus.yml` variables file there.

```
cisco@onepk:~/Training/Ansible$ ls group_vars/
nexus.yml
cisco@onepk:~/Training/Ansible$
```

Figure 112 - Verify nexus.yml is in the proper location

10. Create an empty new `.yml` file called `n9k1.yml` and **Save** it in the **`host_vars`** directory. You can do this by calling to File-> New in Sublime.

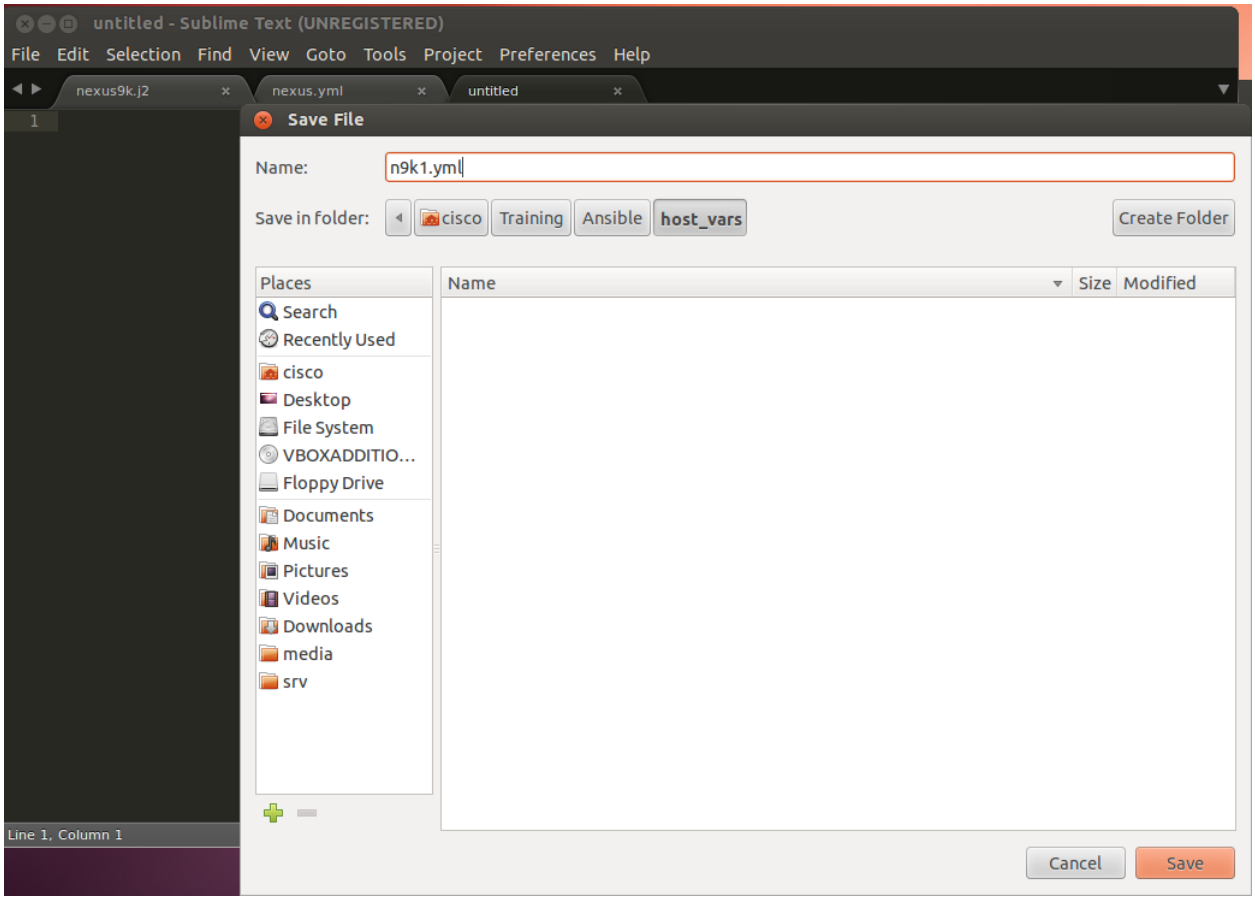

Figure 113 - Create empty vars file n9k1.yml

11. Add two variables to the `n9k1.yml` file and REMOVE them from `nexus.yml`

- Don't forget the three hyphens/dashes to start the yml file

- `<n9k1.yml>`
- `---`
- `hostname: N9K1-HOSTNAME`
- `mgmt_ip: 172.31.217.13x/23 *Use your assigned IP*`

- **Now remove these two variables from `nexus.yml`**

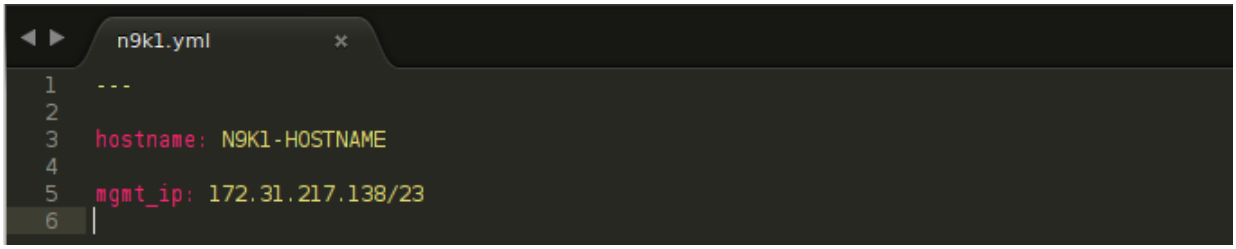

Figure 114 – host_vars/n9k1.yml

```
---

username: admin
password: "cisco123"
banner_motd: "PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF THE LAW!!!!"
domain: cisconxapi.com
vtp_mode: transparent
snmp:
  { contact: JOHN_CHAMBERS, location: CISCO_SJC, ro_string: RORORO, rw_string: RWRWRW }
ntp_server: 192.168.50.11
routes:
  - "ip route 9.0.0.0/24 192.168.88.2"
  - "ip route 192.168.88.0/24 192.168.33.1"
vrf_name: management
route: "ip route 0.0.0.0/0 192.168.200.1"
vlans:
  - { id: 10, name: web }
  - { id: 11, name: qa }
  - { id: 12, name: prod }
  - { id: 13, name: test }
  - { id: 14, name: srvs }
  - { id: 15, name: video }
  - { id: 16, name: voice }
  - { id: 17, name: db2 }
  - { id: 18, name: web2 }
  - { id: 19, name: db }
  - { id: 1000, name: vlan_1000 }
  - { id: 3000, name: dummy }

interfaces:
  - { intf: Ethernet1/1, switchport: "no switchport", ip: "10.101.101.1/30", state: "no shutdown"}
  - { intf: Ethernet1/2, switchport: "no switchport", ip: "10.254.1.1/30", state: "no shutdown"}
  - { intf: Ethernet1/3, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202",
  - { intf: Ethernet1/4, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202",
```

Figure 115 – group_vars/nexus.yml

- Note: whitespace and the rightmost side of the file was removed to simplify the viewing of the figure above.

12. Create the Ansible `hosts` file.

- Open a new empty file in **Sublime Text** and **"Save as"** `hosts` in the `Ansible` directory. No file extension is used on this file.

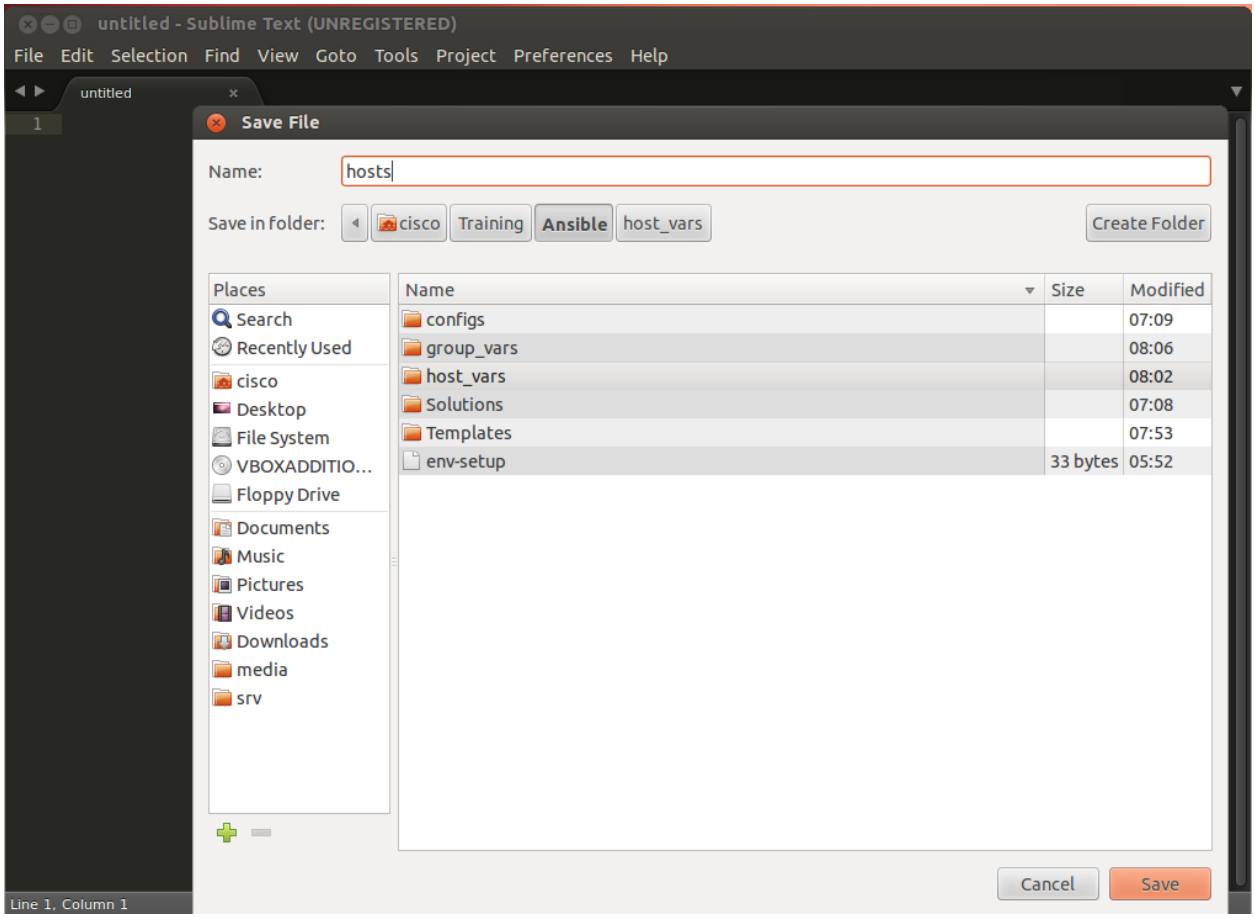

Figure 116 - create Ansible hosts file

13. Populate the `hosts` file with the required information

Note: variables can also be defined in the `hosts` file

Your hosts file should look like what is in the following figure:

Figure 117 - Ansible hosts file

`hosts` file breakdown:

- `[all:vars]`
    - This contains variables that can be used by `all` groups/hosts
- `configdir` is a variable and assigned the value of: "`/home/cisco/Training/Ansible/configs`"

- `[nexus]`
    - Denotes a device `group` called `nexus`
    - This group name is significant and maps to the `nexus.yml` file you created in the `group_vars` directory. The variables in the `nexus.yml` can now be used for devices in the **`nexus`** group.

- `n9k1`
- There is one device in the `nexus` group and it is called `n9k1`

14. The initial setup for Ansible is complete.

15. Create an Ansible playbook that will render the `nexus9k.j2` template with the `nexus.yml` variables.

- Create a new file called `config-builder.yml` and save it in the Ansible directory.

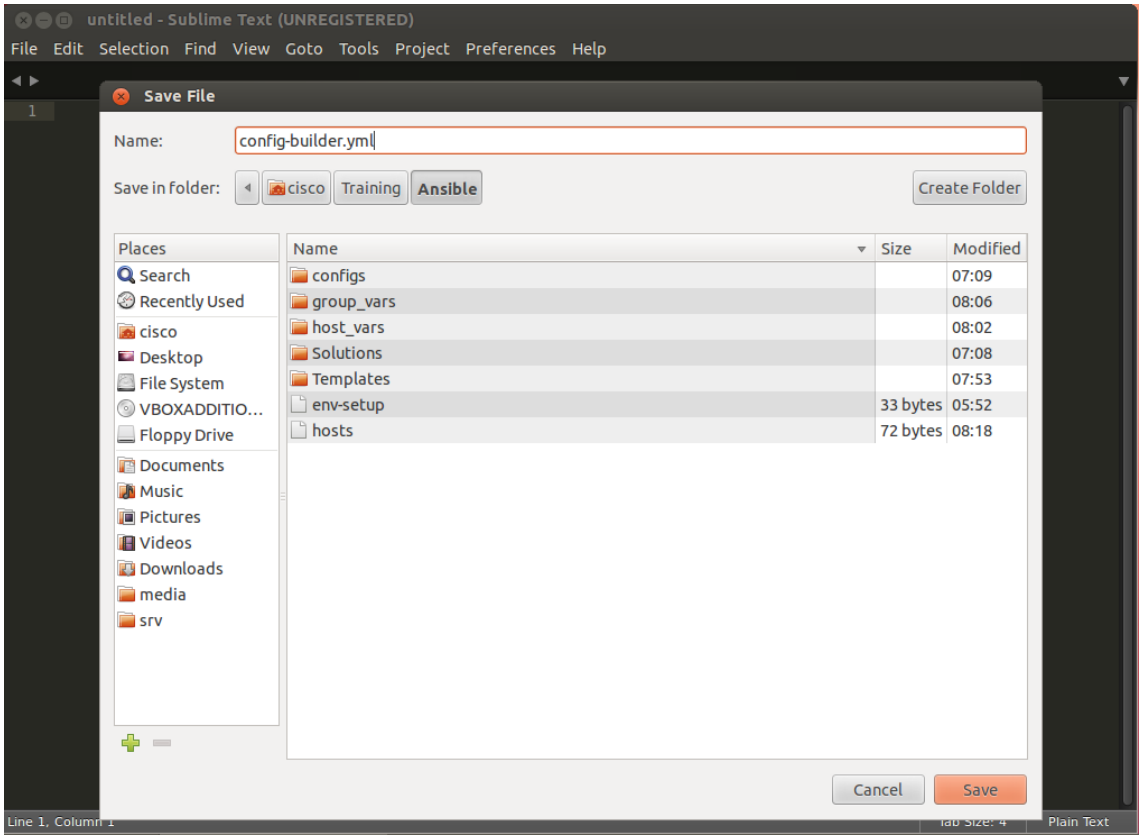

Figure 118 - Create an Ansible playbook

Populate the playbook to look like the following figure. As you can see below, Ansible uses YAML to define Ansible Playbooks.

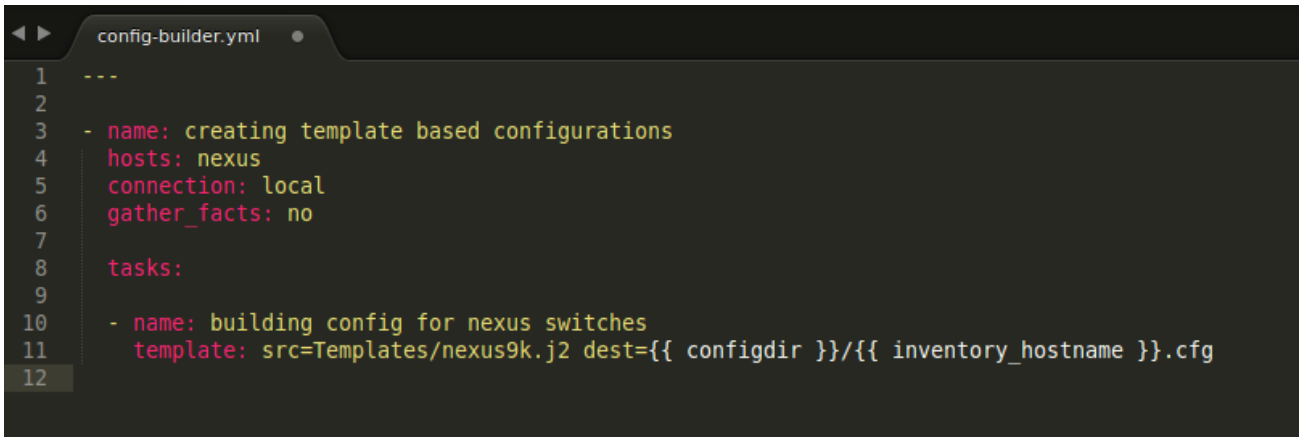

Figure 119 - Ansible Playbook - config-builder.yml

Playbook overview/breakdown

- Always starts with "---" because it's a YAML file
- `name:`

    - is an arbitrary value and this text is displayed when the playbook is executed
- `hosts:`
    - The group of hosts from `host` file that will have tasks executed against when the playbook is executed. This example is using the `nexus` group from the `hosts` file. `all` is also an option. `all` or `nexus` for this example would only `n9k1` because that's the only device in the `hosts` file
- `connection: local`
    - No remote connection will be made from the device. This is usually SSH because Ansible uses SSH, by default, to connect to target devices like servers. For most network devices, this will be `local` for offline configs or will use a native API local to the module
- `gather_facts: no`
    - When Ansible connects to servers, it has the option to gather "facts" about the devices. Attributes such as OS, IP info, etc. You are not using Ansible to connect to servers, so this will be no.

- `tasks:`
    - This is where the tasks you are automating are defined.
    - A task usually maps to one or more Ansible modules
- `template:`
    - This is calling the Ansible module called `template`
    - Reference http://docs.ansible.com/template_module.html for more detail on the template module
    - In this exercise, you are giving the `template` two key/value pairs
        - Location of the template (.j2)
        - Destination, i.e. where to place the final rendered file.
- `inventory_hostname`
    - This is an internal Ansible variable that is always the name of the host as defined in the `hosts` file. In this example, it will be `n9k1`.
- The tasks listed would be iterated for every host in the `hosts` file. This will run through once since there is only one host defined.

16. Execute the playbook

- Open a terminal window and navigate to the `Ansible` directory

  Issue the following command:

- `ansible-playbook config-builder.yml`

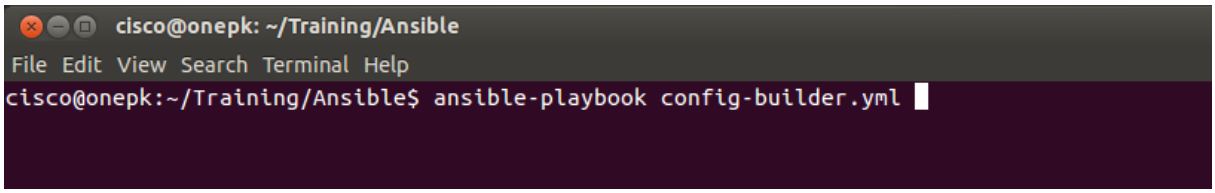

Figure 120 - Running the Playbook

Once you hit enter, you will see the following during execution.

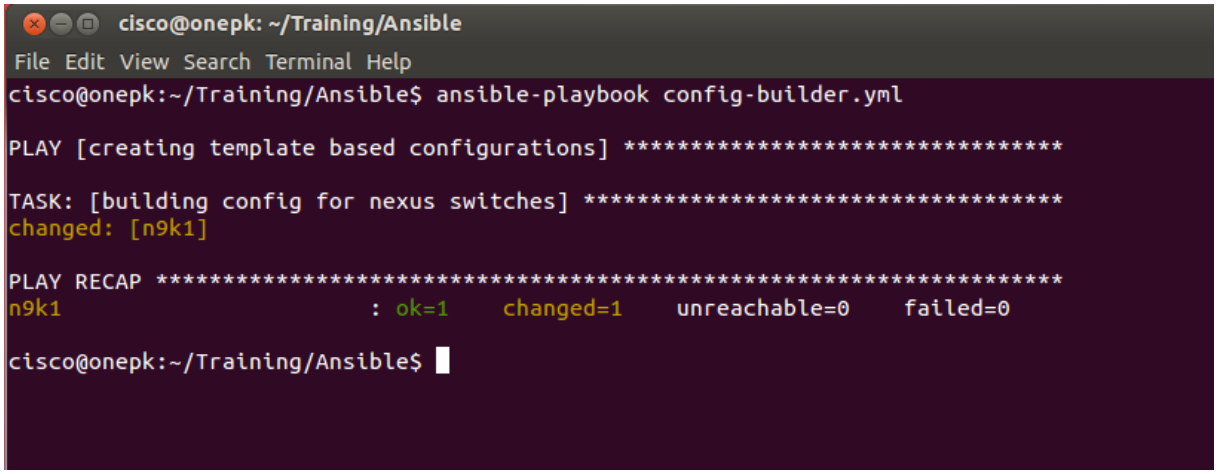

Figure 121 - Playbook output during/after execution

- Changed = 1 means that there was a change for that particular task. Here, you now know that the file was created based on the provided template.

17. Open the newly created config file. You will find it in `configs` directory

    Here is a snippet from the new file that was created called n9k1.cfg

n9k1.cfg

```
1
2  hostname N9K1-HOSTNAME
3
4  username cisco password !cisco123!
5
6  banner motd *
7  PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF THE LAW!!!!
8  *
9
10 ip domain-name cisconxapi.com
11
12 vtp mode transparent
13
14 snmp-server contact JOHN_CHAMBERS
15 snmp-server location CISCO_SJC
16 snmp-server community RORORO group network-operator
17 snmp-server community RWRWRW group network-admin
18
19 ntp server 192.168.50.11
20
21 ip route 9.0.0.0/24 192.168.88.2
22 ip route 192.168.88.0/24 192.168.33.1
23
24 vlan 10
25   name web
26
27 vlan 11
28   name qa
29
30 vlan 12
31   name prod
32
33 vlan 13
34   name test
35
36 vlan 14
37   name srvs
```

Figure 122 - New config file n9k1.cfg

18. Go back to the Linux terminal. Re-run the same playbook.

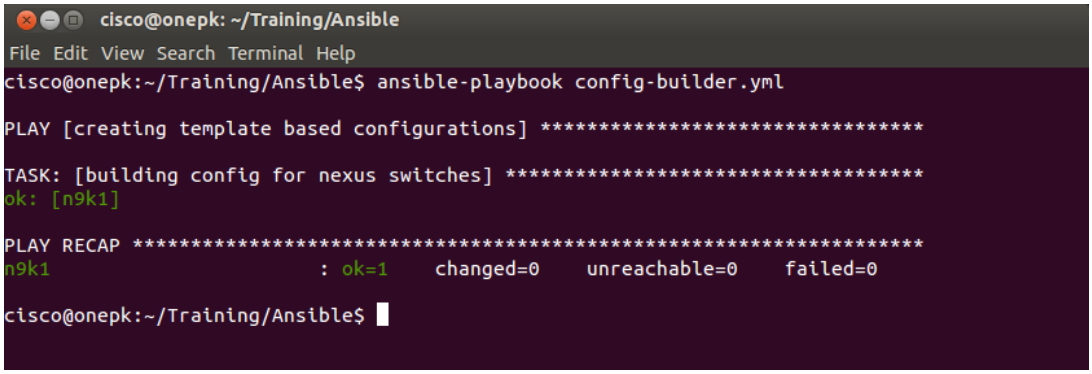

Figure 123 - Re-running the config-builder.yml playbook

Notice it is only green, and how there is no change and a new template is not created. It is smart enough not to generate the same file.

19. Open the hosts file and add 4 more hosts and then save the file.

The `[nexus]` group should now have the following hosts:

- `n9k1`
- `n9k2`
- `n9k3`
- `n9k4`
- `n9k5`

Figure 124 - Adding devices to the hosts file

20. Create 4 new files in the `host_vars` directory.

- The four new files will be `n9k2.yml`, `n9k3.yml`, `n9k4.yml`, and `n9k5.yml`.

The contents for each of the new files can be seen in the following Figures.

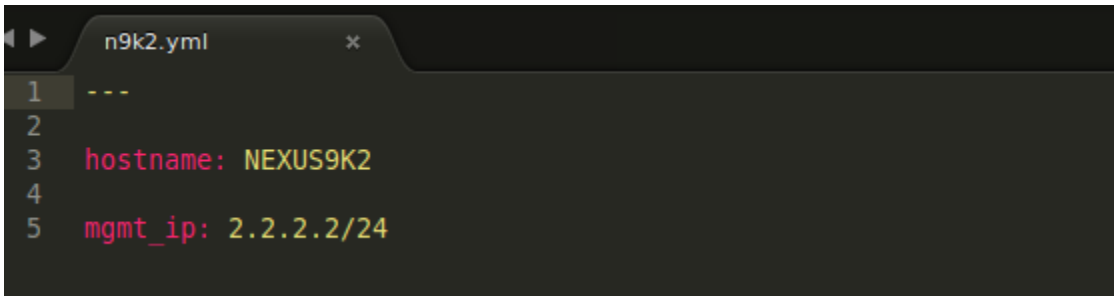

Figure 125 - n9k2.yml

Figure 126 - n9k3.yml

Figure 127 - n9k4.yml

Figure 128 - n9k5.yml

21. Go back to the Linux Terminal and re-run the `config-builder.yml` playbook

    Notice there are 4 new files created that have configs in them.

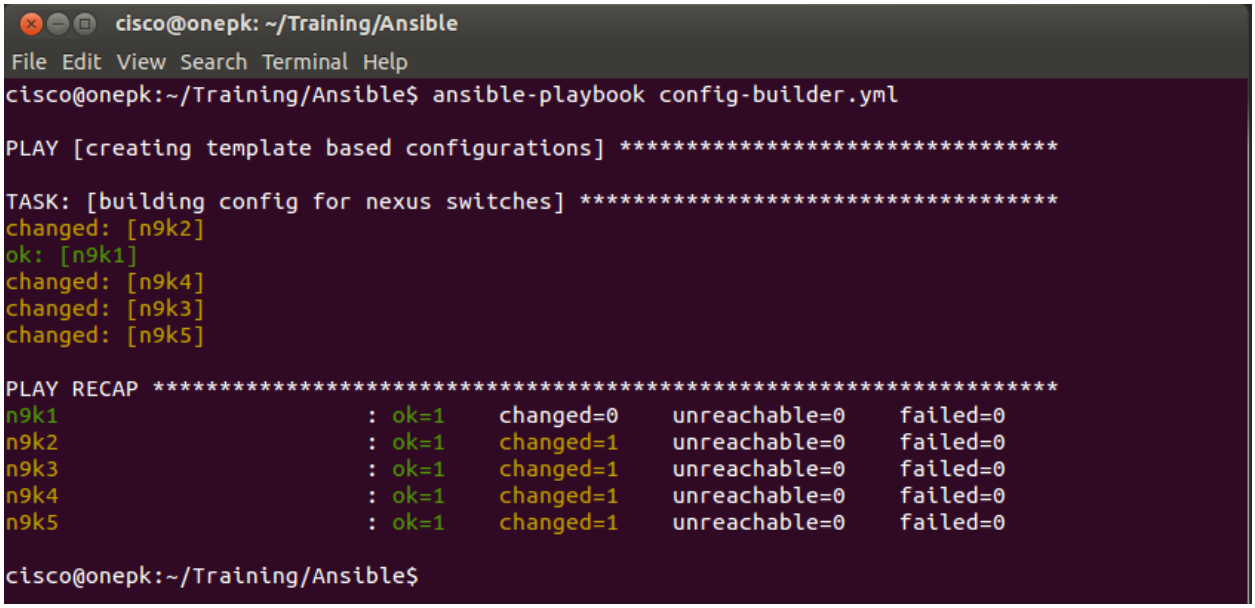

Figure 129 - config-builder.yml execution with 5 devices

22. Issue an `ls` on the `configs` directory and open them to ensure they are correct.

    You should realize the configuration file is being created from using 1 template and 2 variables files per host. Many more than 2 variable files can be used as well. It all depends on the groupings and roles of devices.

```
cisco@onepk:~/Training/Ansible$ ls configs/
n9k1.cfg  n9k2.cfg  n9k3.cfg  n9k4.cfg  n9k5.cfg
```

Figure 130 - Ensure new configs are created

23. Add 2 more hosts to the `hosts` file

    Instead of creating a new vars file for each, configure the vars in the `hosts` file.

    The updated `hosts` file should look like the following figure.

hosts

```
1  [all:vars]
2  configdir=/home/cisco/Training/Ansible/configs
3
4  [nexus]
5  n9k1
6  n9k2
7  n9k3
8  n9k4
9  n9k5
10 n9k6 hostname=NEXUS9K6 mgmt_ip=6.6.6.6/24
11 n9k7 hostname=NEXUS9K7 mgmt_ip=7.7.7.7/24
12
```

Figure 131 - Using host vars in the hosts and create two more devices

24. Go back to the Linux Terminal and re-run the config-builder.yml playbook

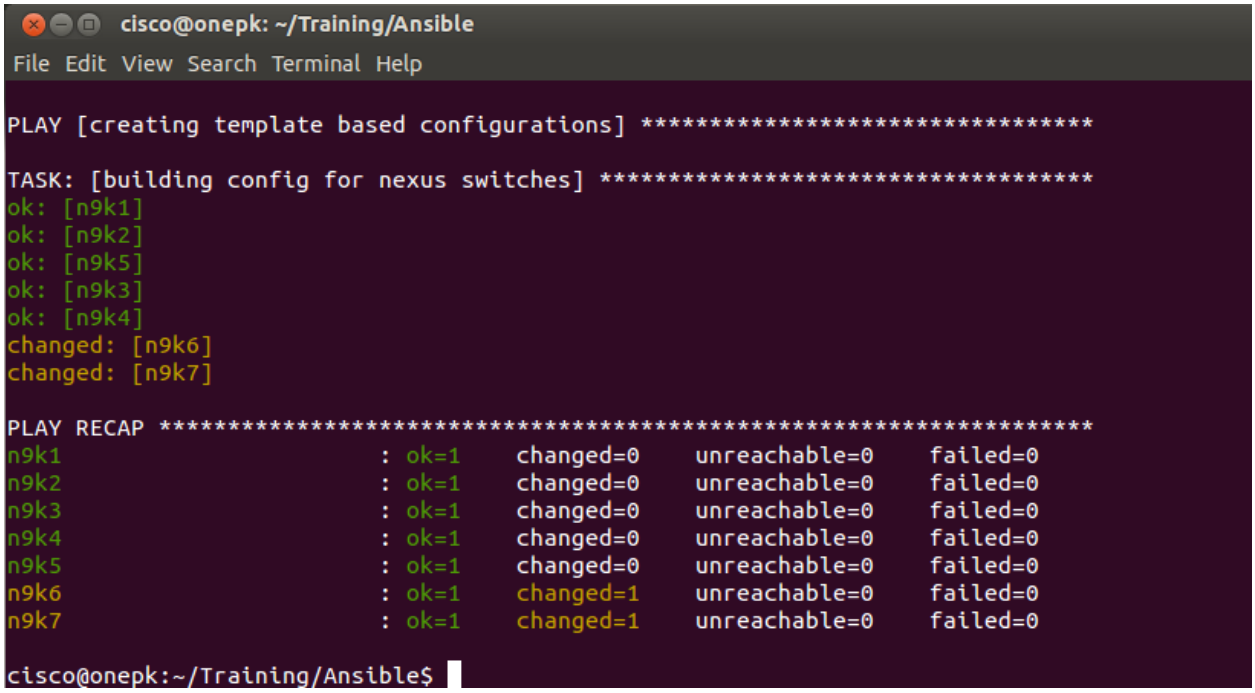

Figure 132 - Re-run the playbook with 7 devices

You know now that host vars can be configured in the hosts file or host based var/yml files.

25. Issue an `ls` on the `configs` directory and open them to ensure they are correct.

Figure 133 - Ensure all configs are created

Lab 4.1.2 – Configuration Automation with Ansible

Overview

In the previous lab, you learned the basics of working with Ansible. You used the hosts file and the Ansible core module called `template`. In this lab, you will use Ansible to configure a Cisco Nexus 9000 switch using custom Ansible modules that configure and manipulate VLANs and interfaces while also learning about loops within Ansible.

Procedures

1. Open the **Sublime Text** Editor

- Open the Ansible `hosts` file, create a new group called switches, and put the IP address of `n9k1` as the only entry in the group. This should be the IP address of YOUR switch.

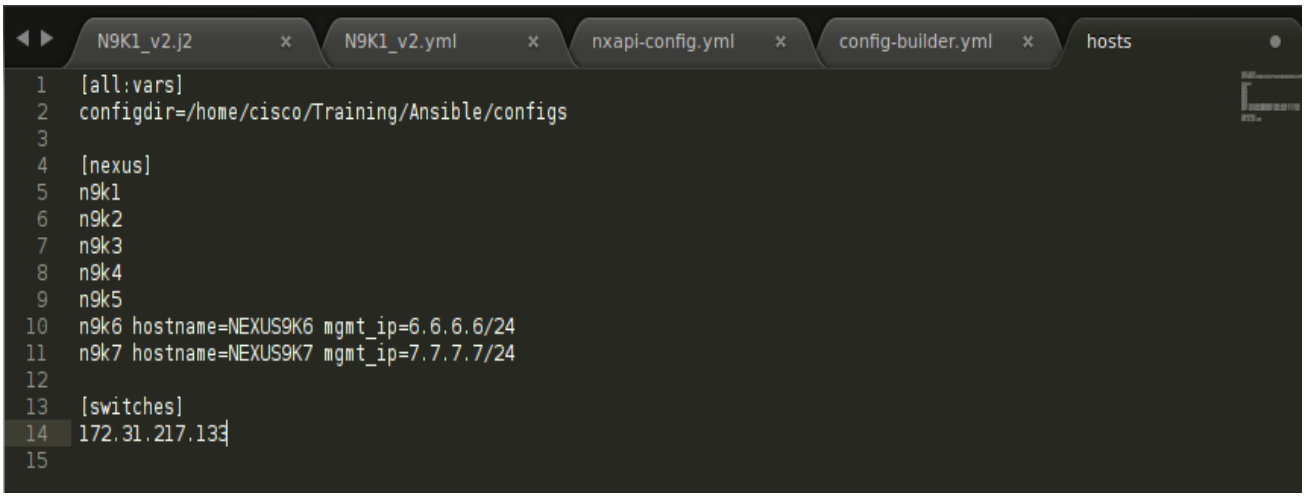

Figure 134 - Create new group in hosts file

2. Create a new playbook called `nxapi-config.yml`

   To do this, you will first open the `config-builder.yml` playbook that you used in the previous lab.

   After it is open, do a **Save As** and use the file name `nxapi-config.yml`. It should be saved in the `Ansible` directory.

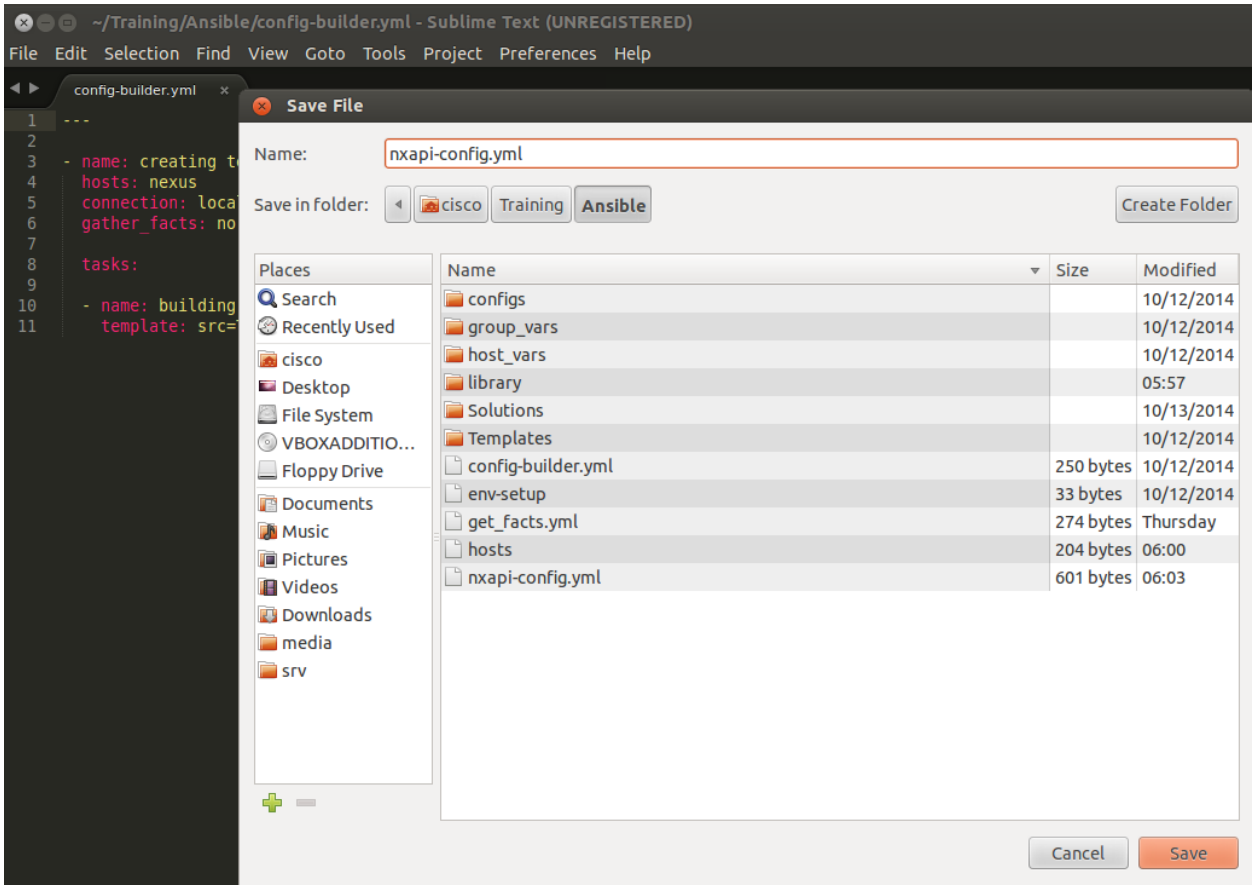

Figure 135 - Create nxapi_config.yml

3. Update the hosts group and name of the play that will be used for this playbook

- The previous lab used the group called `nexus` from the `hosts` file. This playbook and lab is going to use the newly created group called `switches`.

- Update the play name to be "`testing interface and vlan configs using nxapi modules`"

  Use the following figure for guidance.

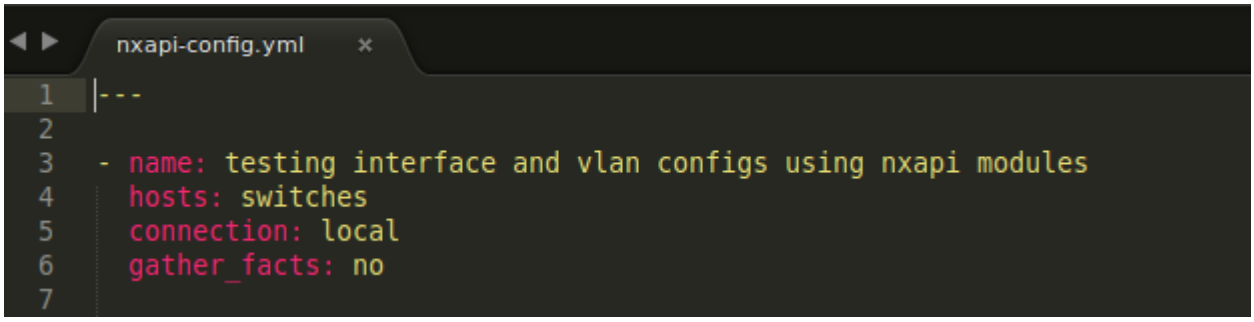

Figure 136 - Update name/hosts for nxapi-config.yml

4. Use the `nxapi_vlan` module to create a VLAN

   Remove the existing task from the previous lab.

   The new task will use the `nxapi_vlan` module. Reference the chart below for the acceptable parameters (key/value pairs) of this module.

- Configure VLAN 100 and give it a name of "`Cisco_100`" by using `nxapi_vlan`

- `tasks:`
-
- `- name: configure VLANs`
- `  nxapi_vlan:  device={{ inventory_hostname }} vlan_id=100 name=Cisco_100 state=present username=`*`USERNAME`* `password=`*`PASSWORD`*

  In this example, five key/value pairs are added to the `nxapi_vlan` module.
  - `device={{ inventory_hostname }}` is equivalent to the name of this device from the `hosts` file. This is sending the IP address of the switch to the module. This is required because the device connection is occurring in the module.
  - `vlan_id` and `name` are self-explanatory, but they are sending the VLAN attributes to the module on what should be configured and enabled
  - `state=present` is telling the module the VLAN should "exist" or be "`present`" on the device. You can see from the chart below that another option is "`absent`". If `absent` is used, the module will ensure the VLAN does not exist on the device.
  - `username=USERNAME` – USERNAME should be equal the username you've been using for the previous labs
  - `password=PASSWORD` – PASSWORD should be equal the password you've been using for the previous labs

You can use either format (no need to use both). Vertically stack the parameters **or** put them on one line.

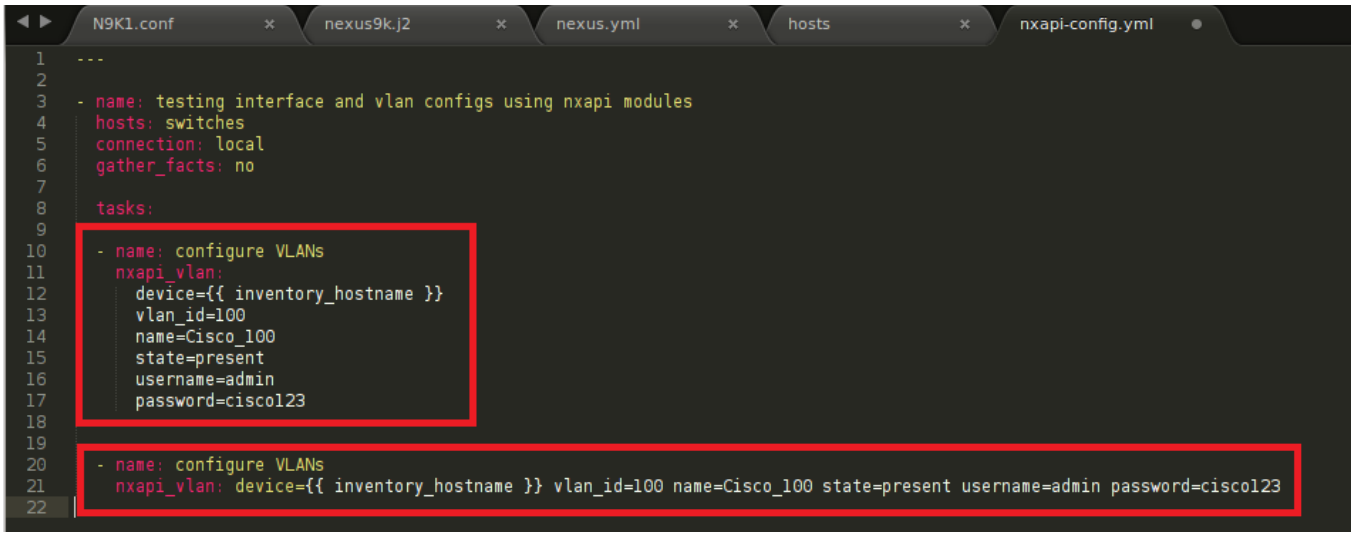

Figure 137 - nxapi_vlan task

Note: do not use spaces between key/value pairs!

- **Module**: nxapi_vlan

- **Description**: Manages VLAN resources in NX-OS

parameter	required	default	choices	comments
vlan_id	yes			
state	no	present	• present • absent	
name	no		•	string to be used as the vlan name
vlan_state	no	active	• active • suspend	equivalent to the 'state' command in vlan configuration mode
shut_state	no	noshutdown	• noshutdown • shutdown	equivalent to using shut/no shut in vlan configuration mode
username	no	admin	•	
password	no	cisco123	•	
device	yes		•	ip address of the device

5. Run the playbook from the Linux command line. First, save the playbook! Then go to a terminal window and navigate to the Ansible directory

   Issue the following command:

- `ansible-playbook nxapi-config.yml`

  You should see the output in the following figure.

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml

PLAY [testing interface and vlan configs using nxapi modules] *****************

TASK: [configure VLANs] *******************************************************
changed: [192.168.200.50]

PLAY RECAP ********************************************************************
192.168.200.50             : ok=1    changed=1    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 138 - nxapi-config.yml

Run the playbook again.

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml

PLAY [testing interface and vlan configs using nxapi modules] *****************

TASK: [configure VLANs] *******************************************************
ok: [192.168.200.50]

PLAY RECAP ********************************************************************
192.168.200.50             : ok=1    changed=0    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 139 - idempotent

Notice how there is no change in this execution because the module is idempotent. If the device is already in the desired state, it will not re-configure the same attribute.

- Now SSH to the device and issue a `show vlan id 100`. You should see that VLAN has been created.
- Remove the VLAN with a '`no vlan 100`' command on the switch.
- Run the playbook again, but use the `-v` flag. This will give more verbose output as the playbook runs.

- `ansible-playbook nxapi-config.yml –v`

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml -v

PLAY [testing interface and vlan configs using nxapi modules] ****************

TASK: [configure VLANs] ******************************************************
changed: [192.168.200.50] => {"changed": true, "commands": "vlan 100 ; state active ; noshutdown ; name
Cisco_100", "created": true, "current_resource": {}, "new_resource": {"name": "Cisco_100", "shut_state":
 "noshutdown", "vid": "100", "vlan_state": "active"}, "resource": {"name": "VLAN0100", "shut_state": "no
shutdown", "vid": "100", "vlan_state": "active"}, "state": "present"}

PLAY RECAP ********************************************************************
192.168.200.50             : ok=1    changed=1    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 140 –playbook verbose output

Note: everything returned that you can see with a –v can be stored as a variable during the rest of an Ansible play. These variables can be printed or used in conditionals within the Ansible language or even used in templates.

Next, go back to the playbook.

- Add the `vlan_state` and `shut_state` key/value pairs to be sent to the module. Configure them to have the values of `suspend` and `shutdown`, respectively. Remember to use the chart above for reference if you need assistance.

Add these new key/value pairs:

- `vlan_state=suspend` and `shut_state=shutdown`

Note: you can have these all on the same line or break them for easy readability when there are more 2-3.

nxapi-config.yml

```
1  ---
2
3  - name: testing interface and vlan configs using nxapi modules
4    hosts: switches
5    connection: local
6    gather_facts: no
7
8    tasks:
9
10   - name: configure VLANs
11     nxapi_vlan:
12       device={{ inventory_hostname }}
13       vlan_id=100
14       name=Cisco_100
15       state=present
16       vlan_state=suspend
17       shut_state=shutdown
18       username=admin
19       password=cisco123
20
```

Figure 141 - Update playbook with vlan_state and shut_state

- Run the playbook. If you use the –v flag, you can look at the commands getting sent to the device. You will notice that only the commands that are required are being sent to the device.

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml -v

PLAY [testing interface and vlan configs using nxapi modules] ****************

TASK: [configure VLANs] ******************************************************
changed: [192.168.200.50] => {"changed": true, "commands": "vlan 100 ; name Cisco_100 ; shutdown ; state
 suspend", "created": false, "current_resource": {"name": "VLAN0100", "shut_state": "noshutdown", "vid":
 "100", "vlan_state": "active"}, "new_resource": {"name": "Cisco_100", "shut_state": "shutdown", "vid":
"100", "vlan_state": "suspend"}, "resource": {"name": "Cisco_100", "shut_state": "shutdown", "vid": "100
", "vlan_state": "suspend"}, "state": "present"}

PLAY RECAP ********************************************************************
192.168.200.50             : ok=1    changed=1    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 142 - Adding key/value pairs for execution

Again, feel free to validate this on the Nexus 9000 switch.

6. Use the `nxapi_interface` module to configure interfaces on the Nexus 9000

   Reference the following chart to understand the available parameters to be used for the `nxapi_interface` module

   Module: nxapi_interface
   Description: Manages physical attributes of interfaces

parameter	required	default	choices	comments
interface	yes			full name of interface, i.e. ethernet1/1 (no short names, i.e. eth1/1)
config_state	no	configured	• configured • unconfigured • default •	• default will issue a 'default interface x/y' • unconfigure will issue a 'no interface x/y' command • unconfigure will default an ethernet interface
description	no		•	description of interface
admin_state	no	noshutdown	• noshutdown • shutdown	
username	no	cisco	•	
password	no	!cisco123!	•	
device	yes	•	•	IP Address of device

- Add the following tasks to the existing playbook. It will configure a description on `Ethernet1/40` and perform a '`shutdown`' on the interface. As you can see, the `{{ inventory_hostname }}` is also being used in the description.

- `- name: configure interface(s)`
- `  nxapi_interface:`
- `    interface=Ethernet1/40 device={{ inventory_hostname }} description="Configured by Ansible on Eth1/40 for {{ inventory_hostname }}" config_state=configured admin_state=shutdown username=USERNAME password=PASSWORD`

nxapi-config.yml

```
---

- name: testing interface and vlan configs using nxapi modules
  hosts: switches
  connection: local
  gather_facts: no

  tasks:

  - name: configure VLANs
    nxapi_vlan:
      device={{ inventory_hostname }}
      vlan_id=100
      name=Cisco_100
      state=present
      vlan_state=suspend
      shut_state=shutdown
      username=admin
      password=cisco123

  - name: configure interface(s)
    nxapi_interface:
      interface=Ethernet1/40 device={{ inventory_hostname }}
      description="Configured by Ansible on Eth1/40 for {{ inventory_hostname }}"
      config_state=configured admin_state=shutdown
      username=admin
      password=cisco123

```

Figure 143 - nxapi_interface in the playbook

Run the playbook.

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml

PLAY [testing interface and vlan configs using nxapi modules] *****************

TASK: [configure VLANs] *******************************************************
ok: [192.168.200.50]

TASK: [configure interface(s)] ************************************************
changed: [192.168.200.50]

PLAY RECAP ********************************************************************
192.168.200.50             : ok=2    changed=1    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 144 – nxapi-config Playbook execution

There was no change on the VLAN task, so that remained "green." But, this was the first time the interface module was being run, so there was a change.

Let's validate the change on the switch. SSH to the switch and do a `'show run interface ethernet1/40'`

```
N9K1# show run interface Eth1/40

!Command: show running-config interface Ethernet1/40
!Time: Mon Oct 20 14:35:08 2014

version 6.1(2)I2(1)

interface Ethernet1/40
  description Configured by Ansible on Eth1/40 for 192.168.200.50
  shutdown

N9K1#
```

Figure 145 - Validate interface config on switch

7. Using `with_items` in Ansible, automate the configuration of multiple interfaces

   This will push the same configuration to interfaces Ethernet1/40, Ethernet41, Ethernet42, and Ethernet 43 using only ONE task

   You will use a loop within Ansible by using a helper module called `with_items`.

   You will need to make three changes to existing playbook.

   - Add the `with_items` module right under the existing module being used and create a list for the interfaces you want to configure
   - Replace "`Ethernet1/40`" with `{{ item }}` for the `interface=Ethernet1/40` key/value pair
   - Replace "`Eth1/40`" with `{{ item }}` within the description

The parameters for the new task should be the following:

```
- name: configure interface(s)
  nxapi_interface:
    interface={{ item }} device={{ inventory_hostname }}
    description="Configured by Ansible on {{ item }} for {{ inventory_hostname }}"
    config_state=configured admin_state=shutdown
    with_items:
      - Ethernet1/40
      - Ethernet1/41
      - Ethernet1/42
      - Ethernet1/43
```

As you can see `{{ item }}` is a variable and its getting its value from each value in the list configured. For each item listed under `with_items` module, the task will be performed. So, the module `nxapi_interface` is basically being called four times.

At this point, the playbook should look like what is in the following figure:

nxapi-config.yml

```
---

- name: testing interface and vlan configs using nxapi modules
  hosts: switches
  connection: local
  gather_facts: no

  tasks:

  - name: configure VLANs
    nxapi_vlan:
      device={{ inventory_hostname }}
      vlan_id=100
      name=Cisco_100
      state=present
      vlan_state=suspend
      shut_state=shutdown
      username=admin
      password=cisco123

  - name: configure interface(s)
    nxapi_interface:
      interface={{ item }} device={{ inventory_hostname }}
      description="Configured by Ansible on {{ item }} for {{ inventory_hostname }}"
      config_state=configured admin_state=shutdown
      username=admin
      password=cisco123
    with_items:
      - Ethernet1/40
      - Ethernet1/41
      - Ethernet1/42
      - Ethernet1/43

```

Figure 146 - nxapi-config.yml FINAL

- Run the playbook using the –v flag.

- `ansible-playbook nxapi-config.yml –v`

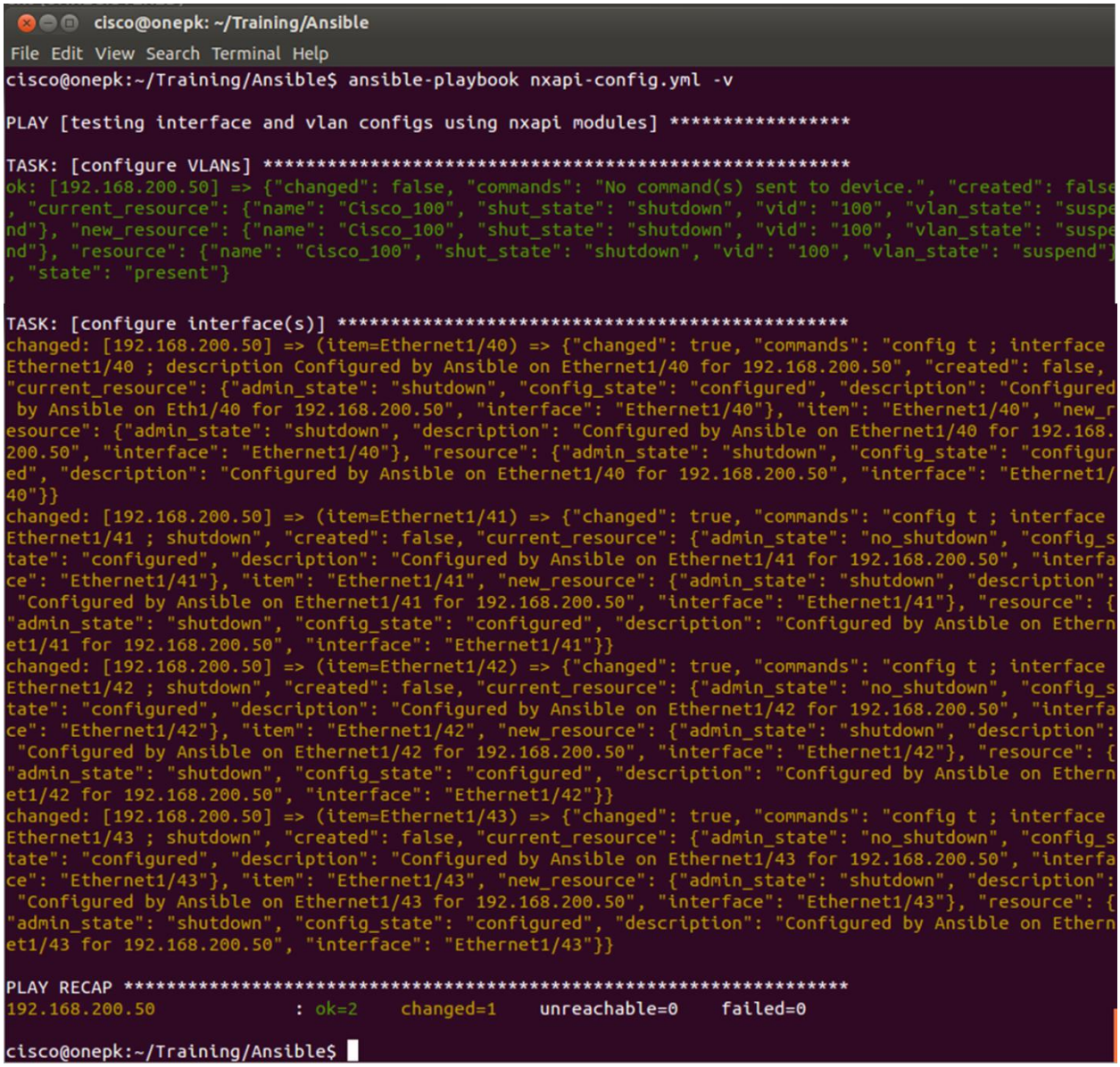

Figure 147 - Automate 4 interfaces

If you validate the configs, you should see that all 4 interfaces are configured properly.

```
interface Ethernet1/40
  description Configured by Ansible on Ethernet1/40 for 192.168.200.50
  shutdown

N9K1# show run interface Eth1/41

!Command: show running-config interface Ethernet1/41
!Time: Mon Oct 20 14:59:11 2014

version 6.1(2)I2(1)

interface Ethernet1/41
  description Configured by Ansible on Ethernet1/41 for 192.168.200.50
  shutdown

N9K1# show run interface Eth1/42

!Command: show running-config interface Ethernet1/42
!Time: Mon Oct 20 14:59:12 2014

version 6.1(2)I2(1)

interface Ethernet1/42
  description Configured by Ansible on Ethernet1/42 for 192.168.200.50
  shutdown

N9K1# show run interface Eth1/43

!Command: show running-config interface Ethernet1/43
!Time: Mon Oct 20 14:59:13 2014

version 6.1(2)I2(1)

interface Ethernet1/43
  description Configured by Ansible on Ethernet1/43 for 192.168.200.50
  shutdown
```

Figure 148 - Validate description and state for each interface

- Change the `config_state` in the playbook to `default`

  You can see the change in the following figure.

```
- name: configure interface(s)
  nxapi_interface:
    interface={{ item }} device={{ inventory_hostname }}
    description="Configured by Ansible on {{ item }} for {{ inventory_hostname }}"
    config_state=default admin_state=shutdown
    username=admin
    password=cisco123
  with_items:
    - Ethernet1/40
    - Ethernet1/41
    - Ethernet1/42
    - Ethernet1/43
```

Figure 149 - Changed config_state

Re-run the playbook.

```
cisco@onepk:~/Training/Ansible$ ansible-playbook nxapi-config.yml

PLAY [testing interface and vlan configs using nxapi modules] *****************

TASK: [configure VLANs] *******************************************************
ok: [192.168.200.50]

TASK: [configure interface(s)] ************************************************
changed: [192.168.200.50] => (item=Ethernet1/40)
changed: [192.168.200.50] => (item=Ethernet1/41)
changed: [192.168.200.50] => (item=Ethernet1/42)
changed: [192.168.200.50] => (item=Ethernet1/43)

PLAY RECAP ********************************************************************
192.168.200.50             : ok=2    changed=1    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 150 - Run the playbook to default intfs

Validate the changes.

```
N9K1# show run interface Eth1/40

!Command: show running-config interface Ethernet1/40
!Time: Mon Oct 20 15:02:55 2014

version 6.1(2)I2(1)

interface Ethernet1/40

N9K1# show run interface Eth1/41

!Command: show running-config interface Ethernet1/41
!Time: Mon Oct 20 15:02:59 2014

version 6.1(2)I2(1)

interface Ethernet1/41

N9K1# show run interface Eth1/42

!Command: show running-config interface Ethernet1/42
!Time: Mon Oct 20 15:03:01 2014

version 6.1(2)I2(1)

interface Ethernet1/42

N9K1# show run interface Eth1/43

!Command: show running-config interface Ethernet1/43
!Time: Mon Oct 20 15:03:03 2014

version 6.1(2)I2(1)

interface Ethernet1/43
```

Figure 151 - Validate configs are defaulted

Lab 4.1.3 – Extending Ansible – Creating a Custom Ansible Module

Overview

This is the final lab for working with Ansible and covers a fairly advanced topic. This lab covers how to create a custom Ansible module. In this lab, you will take the `nxapi_facts` Python script that you created in the NX-API script lab and convert that to be an Ansible module. If you recall, the script works by sending in a command line argument and then the script returns the value of the key being asked for as long it is in the "facts" dictionary. The module will work in the same fashion. The difference is that instead of using command line arguments, you will use key/value pairs within Ansible just as you have been using like with the `template`, `nxapi_interface`, and `nxapi_vlan` modules.

Procedures

1. Open the Python script, `classfiles/nxapi_facts.py`, that was created in the NX-API lab

nxapi_facts.py

```
#!/usr/bin/env python

import xmltodict
import json
import sys
from device import Device

def show_hardware(sw):

    getdata = sw.show('show hardware')

    show_hw_dict = xmltodict.parse(getdata[1])
    data = show_hw_dict['ins_api']['outputs']['output']['body']

    hw_dict = {}
    hw_dict['os_version'] = data['kickstart_ver_str']
    hw_dict['type'] = data['chassis_id']
    hw_dict['memory'] = data['memory'] + data['mem_type']
    hw_dict['hostname'] = data['host_name']
    hw_dict['bootflash'] = data['bootflash_size']
    hw_dict['last_reboot_reason'] = data['rr_reason']
    hw_dict['uptime'] = '{} day(s) {} hour(s) {} min(s) {} sec(s)'.format(data['kern_uptm_days'], \
        data['kern_uptm_hrs'],data['kern_uptm_mins'],data['kern_uptm_secs'])

    ser_nums = {}
    ser_nums_data = show_hw_dict['ins_api']['outputs']['output']['body']['TABLE_slot'] \
        ['ROW_slot']['TABLE_slot_info']['ROW_slot_info']

    for each in ser_nums_data:

        if 'serial_num' in each.keys():                #check sandbox output.  fans would cause error he
            key = each['serial_num']
            ser_nums[key] = each['model_num']

    hw_dict['serial_numbers'] = ser_nums

```

Figure 152 - Open nxapi_facts.py

2. **Save it As** `nxapi_facts` in the `Ansible/library` directory (**remove** the `.py` extension)

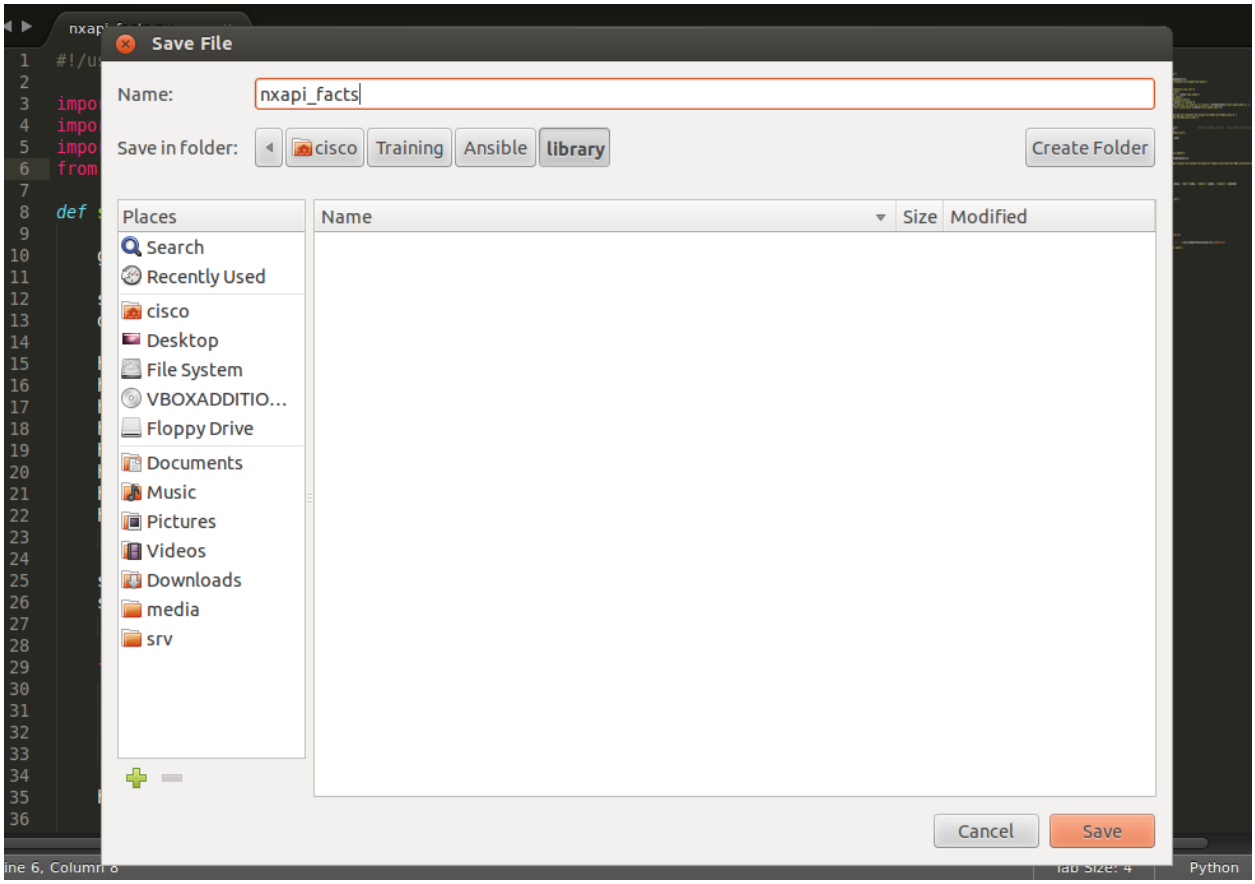

Figure 153 - Remove the .py extension

3. Replace the last two lines of the script

   The existing last two lines are:

- `if __name__ == "__main__":`
- `    main()`

   They should be replaced with:

- `from ansible.module_utils.basic import *`
- `main()`

```
80
81  from ansible.module_utils.basic import *
82  main()
83
```

Figure 154 - import required module for creating Ansible modules and call main()

4. Remove the print statements.

You cannot print while within Ansible modules. Instead, you will save the data that would have been printed into a variable called `msg`.

The existing print statements are:

- `print json.dumps(facts, indent=4)`
- `print args[1].upper() + ': ' + json.dumps(facts[args[1]],indent=4)`
- `print 'Invalid Key.  Try again.'`

They should be replaced with:

- `msg = facts`
- `msg = facts[args[1]]`
- `msg = 'Invalid Key.  Try again.'`

```
71
72      if len(args) == 1:
73          msg = facts
74      else:
75          if args[1] in facts.keys():
76              msg = facts[args[1]]
77          else:
78              msg = 'Invalid Key.  Try again.'
79
```

Figure 155 - Save status in var msg and do NOT print

5. Add an Ansible specific command to return the `msg` variable back to the Ansible playbook.

- This is added at the bottom of the module and represents the data that is printed when you run a playbook with the –v flag.

- `module.exit_json(status=msg)`
  This assigns the contents of msg to status and status is returned to the Playbook for analysis, debug, printing, etc.

  Following these changes, `main()` should look like the following:

```
57 def main():
58
59     switch = Device(ip='192.168.200.50')
60     switch.open()
61
62     facts = {}
63
64     intf = show_intf_mgmt(switch)
65     facts['mgmt_intf'] = intf
66
67     hw = show_hardware(switch)
68     facts.update(hw)
69
70     args = sys.argv
71
72     if len(args) == 1:
73         msg = facts
74     else:
75         if args[1] in facts.keys():
76             msg = facts[args[1]]
77         else:
78             msg = 'Invalid Key.  Try again.'
79
80     module.exit_json(status=msg)
81
82 if __name__ == "__main__":
83     main()
84
```

Figure 156 - Add module.exit_json

6. Ansible sends key/value pairs from the playbook into the module. These are sent as Python dictionaries. This module will accept two variables: the device IP address and the `fact` to be returned. Because command line arguments are no longer going to be used, the sys.argv and argv command line assignments need to be updated.

- Add the following to the top of `main()`

```
module = AnsibleModule(
       argument_spec = dict(
       device=dict(required=True),
       fact=dict(required=True)
     ),
     supports_check_mode = False
   )
```

The variable called `module` will be initialized at the top of function `main()`. This says two variables called `device` and `fact` will be sent from the playbook to the module.

The conditional also needs to be updated because we no longer need sys.argv and args.

The following is what the new variable assignments and conditional should look like:

```
57  def main():
58
59      module = AnsibleModule(
60      argument_spec = dict(
61        device=dict(required=True),
62        fact=dict(required=True)
63      ),
64      supports_check_mode = False
65    )
66      ip_addr = module.params['device']
67      fact = module.params['fact']
68
69      switch = Device(ip=ip_addr)
70      switch.open()
71
72      facts = {}
73      intf = show_intf_mgmt(switch)
74      facts['mgmt_intf'] = intf
75
76      hw = show_hardware(switch)
77      facts.update(hw)
78
79      if fact == 'all':
80          msg = facts
81      else:
82          if fact in facts.keys():
83              msg = facts[fact]
84          else:
85              msg = 'Invalid Key.  Try again.'
86
87      module.exit_json(status=msg)
88
89  from ansible.module_utils.basic import *
90  main()
```

Figure 157 - Add var module that is of type AnsibleModule and used to receive key/value pairs from playbook

Take your time to review the code. Does it make sense?

The conditional states that if `all` is specified as the fact, all attributes/values will be returned. Otherwise, if it is a valid fact, return that one fact (k/v pair). If the fact is not part of the `keys` in the `facts` dictionary, the "`Invalid key`" string will be returned.

7. If you didn't do this in the previous lab, create a new group in the `hosts` file

- Open the Ansible `hosts` file, create a new group called switches, and put the IP address of `n9k1` as the only entry in the group

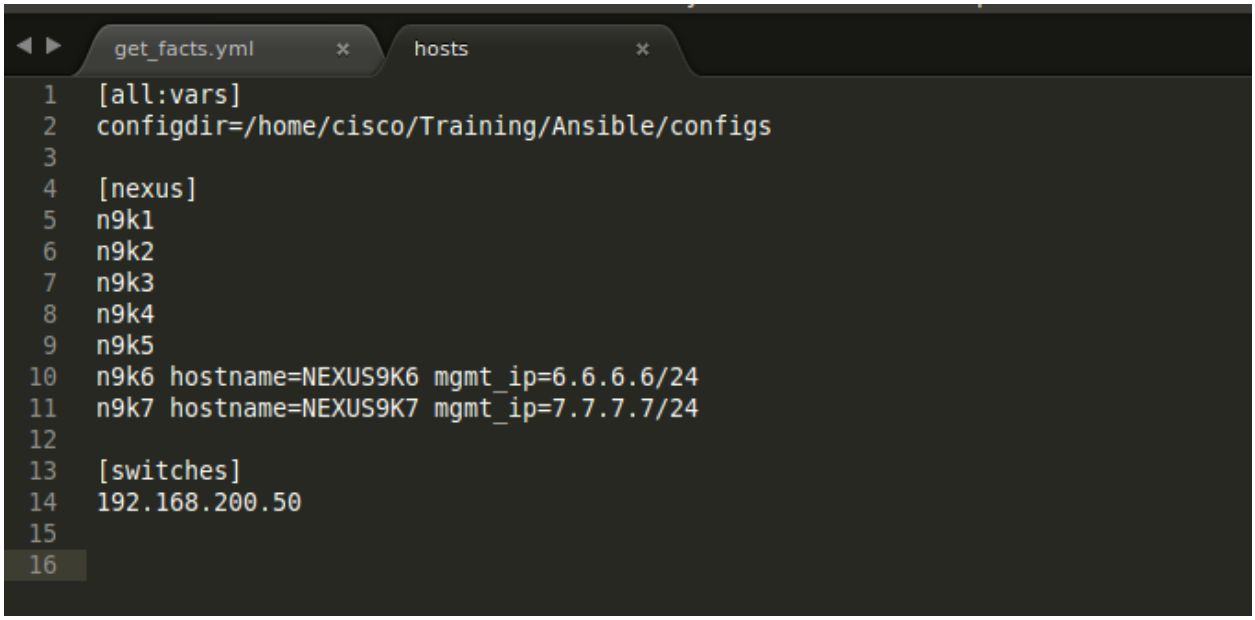

Figure 158 - Create new group in hosts file

8. Open the `config-builder.yml` playbook and do a **Save As**.

- Save the new playbook as `get_facts.yml` in the Ansible directory.

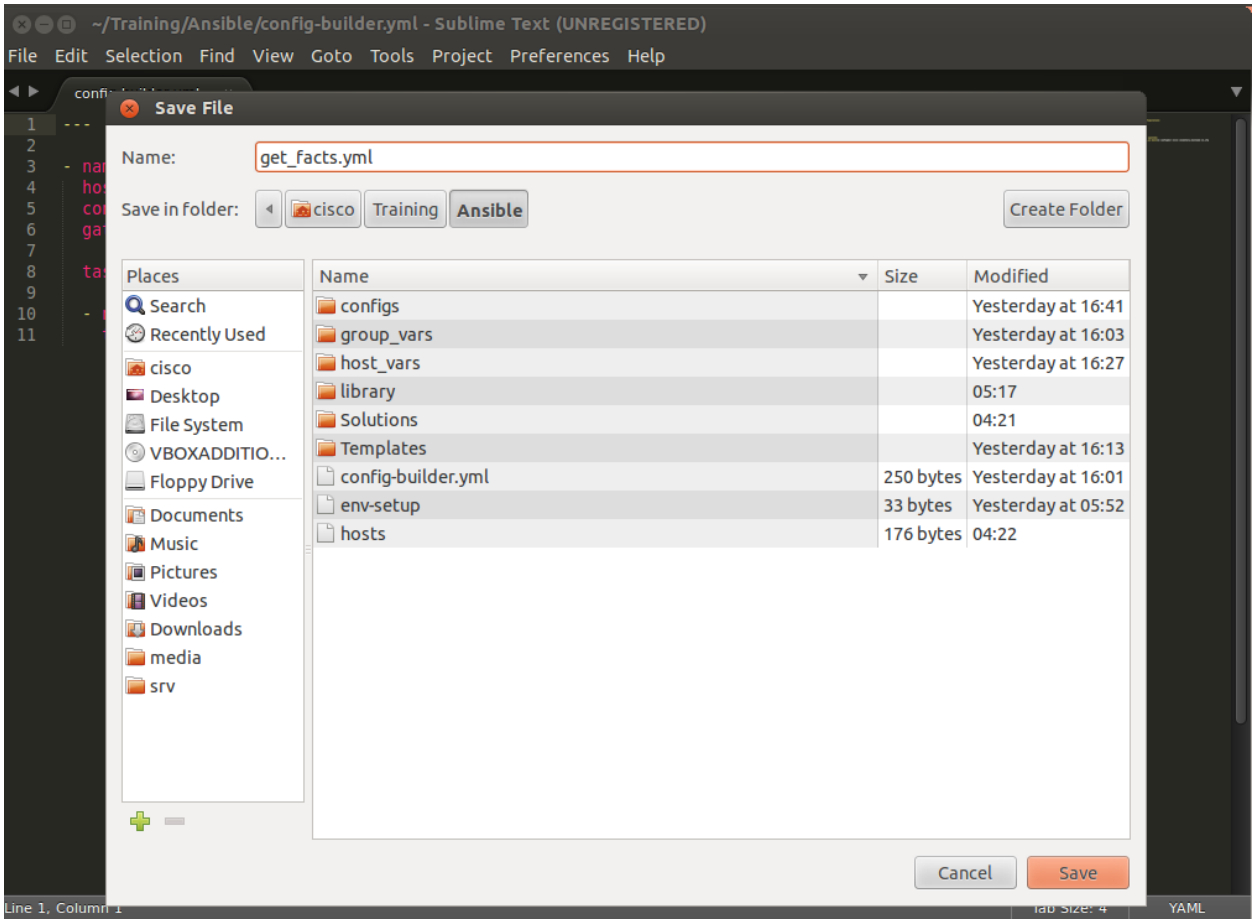

Figure 159 - Create new playbook - get_facts.yml

9. Update the Ansible Playbook `get_facts.yml`

   Make the following changes:

   - Update the hosts group from `nexus` to `switches`
   - Update the name of the play to `gather facts from nexus 9000 switches`
   - Remove the Template module from being used
   - Add the `get_facts` module and send two key/value pairs to the module
     - Key/value 1: device={{ inventory_hostname }}
     - Key/value 2: fact=all

   - Usage of the new module:

     ```
     - name: get facts from device
     ```

```
    nxapi_facts: device={{ inventory_hostname }} fact=all
```

10. In order to print the facts returned without using the –v flag, you need to store the return value in a new variable. This is done by using the Ansible `register` module

    Right beneath get_facts, the register module is used. It will look like this:

    - `- name: get facts from device`
    - `  nxapi_facts: device={{ inventory_hostname }} fact=all`
    - `  register: my_facts`

    In this example, the variable status that was sent back from the playbook will be accessed through `my_facts` like a Python dictionary using a "." Notation

11. Print the return data to the screen. Use the `debug` module to print raw data to the screen.

    A separate task will be used for this using the following syntax:

    - `debug: msg="{{ my_facts.status }}"`

    After these changes the playbook should look like the following figure:

get_facts.yml

```
1  ---
2
3  - name: gather facts from nexus 9000 switches
4    hosts: switches
5    connection: local
6    gather_facts: no
7
8    tasks:
9
10   - name: get facts from device
11     nxapi_facts: device={{ inventory_hostname }} fact=all
12     register: my_facts
13
14   - debug: msg="{{ my_facts.status }}"
15
16
```

Figure 160 - final get_facts.yml playbook

12. Run the playbook

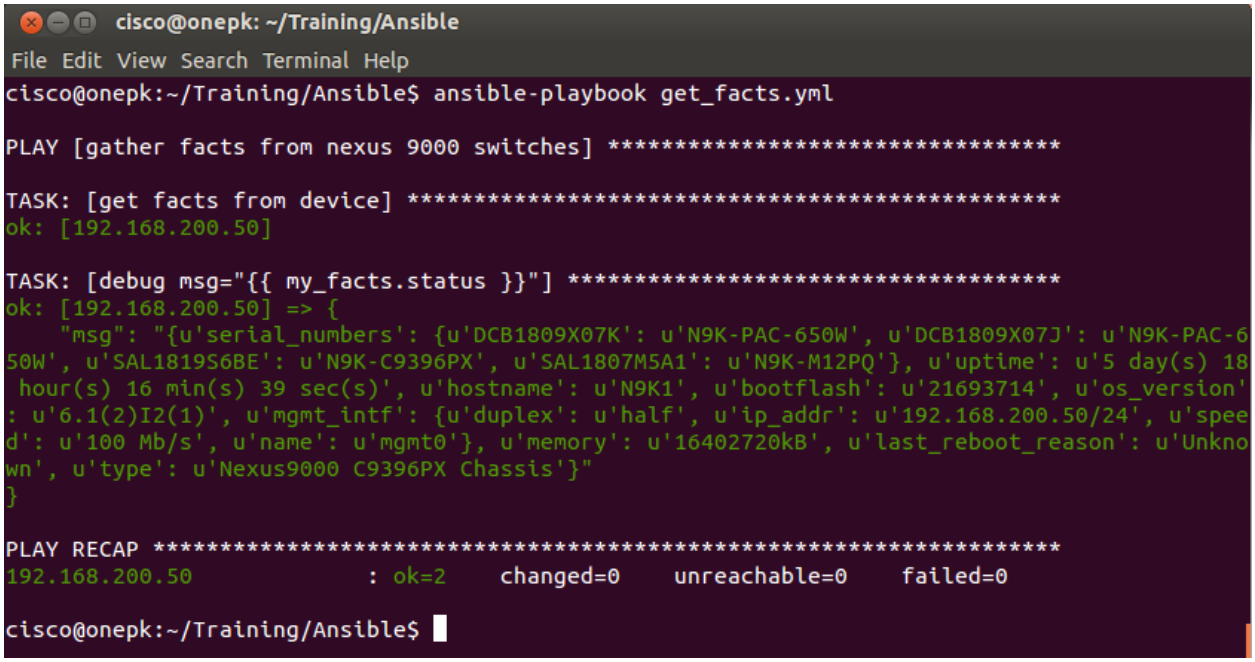

Figure 161 - Output from running get_facts.yml

13. In the playbook, change `fact=type`, and then re-run the playbook

```
get_facts.yml
1  ---
2
3  - name: gather facts from nexus 9000 switches
4    hosts: switches
5    connection: local
6    gather_facts: no
7
8    tasks:
9
10   - name: get facts from device
11     nxapi_facts: device={{ inventory_hostname }} fact=type
12     register: my_facts
13
14   - debug: msg="{{ my_facts.status }}"
15
16
17
18
```

Figure 162 - Modify key/value pairs sent to the module nxapi_facts

```
cisco@onepk: ~/Training/Ansible
File  Edit  View  Search  Terminal  Help
cisco@onepk:~/Training/Ansible$ ansible-playbook get_facts.yml

PLAY [gather facts from nexus 9000 switches] *********************************

TASK: [get facts from device] ************************************************
ok: [192.168.200.50]

TASK: [debug msg="{{ my_facts.status }}"] ************************************
ok: [192.168.200.50] => {
    "msg": "Nexus9000 C9396PX Chassis"
}

PLAY RECAP ********************************************************************
192.168.200.50             : ok=2    changed=0    unreachable=0    failed=0

cisco@onepk:~/Training/Ansible$
```

Figure 163 - Ansible Playbook for the final run

14. Because this data is now stored in `my_facts`, you can also create a report template using Jinja2. The variables wouldn't need to be in stored in YAML because they would be accessed from memory from `my_facts`. Feel free to give it a try!
15. For reference the following figures show the final playbook and module for this lab. The play

get_facts.yml

```
---

- name: gather facts from nexus 9000 switches
  hosts: switches
  connection: local
  gather_facts: no

  tasks:

  - name: get facts from device
    nxapi_facts: device={{ inventory_hostname }} fact=type
    register: my_facts

  - debug: msg="{{ my_facts.status }}"
```

Figure 164 - get_facts.yml

nxapi_facts

```
#!/usr/bin/env python

import xmltodict
import json
import sys
from device import Device

def show_hardware(sw):

    getdata = sw.show('show hardware')

    show_hw_dict = xmltodict.parse(getdata[1])
    data = show_hw_dict['ins_api']['outputs']['output']['body']

    hw_dict = {}
    hw_dict['os_version'] = data['kickstart_ver_str']
    hw_dict['type'] = data['chassis_id']
    hw_dict['memory'] = data['memory'] + data['mem_type']
    hw_dict['hostname'] = data['host_name']
    hw_dict['bootflash'] = data['bootflash_size']
    hw_dict['last_reboot_reason'] = data['rr_reason']
    hw_dict['uptime'] = '{} day(s) {} hour(s) {} min(s) {} sec(s)'.format(data['kern_uptm_days'], \
        data['kern_uptm_hrs'],data['kern_uptm_mins'],data['kern_uptm_secs'])

    ser_nums = {}
    ser_nums_data = show_hw_dict['ins_api']['outputs']['output']['body']['TABLE_slot'] \
        ['ROW_slot']['TABLE_slot_info']['ROW_slot_info']

    for each in ser_nums_data:

        if 'serial_num' in each.keys():                #check sandbox output.  fans would cause error here
            key = each['serial_num']
            ser_nums[key] = each['model_num']

    hw_dict['serial_numbers'] = ser_nums

    return hw_dict

def show_intf_mgmt(sw):

    getdata = sw.show('show interface mgmt0')

    show_intf_dict = xmltodict.parse(getdata[1])

    data = show_intf_dict['ins_api']['outputs']['output']['body']['TABLE_interface']['ROW_interface']

    ip = data['eth_ip_addr']
    mask = data['eth_ip_mask']
    name = data['interface']
    speed = data['eth_speed']
    duplex = data['eth_duplex']

    mgmt_dict = { 'ip_addr': ip+'/'+mask, 'name':name, 'speed': speed, 'duplex': duplex}

    return mgmt_dict

def main():

    module = AnsibleModule(
    argument_spec = dict(
      device=dict(required=True),
      fact=dict(required=True)
    ),
    supports_check_mode = False
  )
    ip_addr = module.params['device']
    fact = module.params['fact']

    switch = Device(ip=ip_addr)
    switch.open()

    facts = {}
    intf = show_intf_mgmt(switch)
    facts['mgmt_intf'] = intf

    hw = show_hardware(switch)
    facts.update(hw)

    if fact == 'all':
        msg = facts
    else:
        if fact in facts.keys():
            msg = facts[fact]
        else:
            msg = 'Invalid Key.  Try again.'

    module.exit_json(status=msg)

from ansible.module_utils.basic import *
main()

```

Figure 165 - get_facts

Appendix A

Lab 2.1.1

N9K1.j2 file

```
hostname {{ hostname }}

username {{ username }} password {{ password }}

banner motd *
{{ banner_motd}}
*

ip domain-name {{ domain }}

vtp mode {{ vtp_mode }}

snmp-server contact {{ snmp_contact }}
snmp-server location {{ snmp_location }}
snmp-server community {{ ro_string }} group network-operator
snmp-server community {{ rw_string }} group network-admin

ntp server {{ ntp_server }}

{{ route_1 }}
{{ route_2 }}

vrf context {{ vrf_name }}
  ip domain-name {{ domain }}
  {{ route }}

interface {{ intf_id_1 }}
  {{ switchport_1 }}
  ip address {{ ip_1 }}
  {{ state_1 }}

interface {{ intf_id_2 }}
  {{ switchport_2 }}
  ip address {{ ip_2 }}
  {{ state_2 }}

interface {{ intf_id_3 }}
  {{ switchport_3 }}
  switch mode {{ mode_3 }}
  switchport trunk native vlan {{ native_vlan_3 }}
  switchport trunk allowed vlan {{ vlan_range_3 }}
  {{ state_3 }}

interface {{ intf_id4 }}
  {{ switchport_4 }}
  switchport mode {{ mode_4 }}
  switchport trunk native vlan {{ native_vlan_4 }}
  switchport trunk allowed vlan {{ vlan_range_4 }}
  {{ state_4 }}
```

```
interface mgmt0
  vrf member management
  ip address {{ mgmt_ip }}
```

N9K1.yml file

```
---

hostname: N9K1-HOSTNAME

username: cisco
password: "!cisco123!"

banner_motd: "PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF THE LAW!!!!"

domain: cisconxapi.com

vtp_mode: transparent

snmp_contact: JOHN_CHAMBERS

snmp_location: CISCO_SJC

ro_string: RORORO
rw_string: RWRWRW

ntp_server: 192.168.50.11

route_1: "ip route 9.0.0.0/24 192.168.88.2"
route_2: "ip route 192.168.88.0/24 192.168.33.1"

vrf_name: management
route: "ip route 0.0.0.0/0 192.168.200.1"

intf_id_1: Ethernet1/1
switchport_1: "no switchport"
ip_1: "10.101.101.1/30"
state_1: "no shutdown"

intf_id_2: Ethernet1/2
switchport_2: "no switchport"
ip_2: "10.254.1.1/30"
state_2: "no shutdown"

intf_id_3: Ethernet1/3
switchport_3: switchport
mode_3: trunk
native_vlan_3: 3000
vlan_range_3: "100-102,200-202"
state_3: "no shutdown"

intf_id4: Ethernet1/4
switchport_4: switchport
mode_4: trunk
```

```
native_vlan_4: 3000
vlan_range_4: "100-102,200-202"
state_4: "no shutdown"

mgmt_ip: 192.168.200.11/24
```

Lab 2.1.3

N9K1_v2.yml file

```
---

hostname: N9K1-HOSTNAME

username: cisco
password: "!cisco123!"

banner_motd: "PROPERTY OF CISCO. IF YOU CONTINUE, YOU WILL BE PROSECUTED TO THE FULLEST EXTENT OF THE LAW!!!!"

domain: cisconxapi.com

vtp_mode: transparent

snmp:
  { contact: JOHN_CHAMBERS, location: CISCO_SJC, ro_string: RORORO, rw_string: RWRWRW }

ntp_server: 192.168.50.11

routes:
  - "ip route 9.0.0.0/24 192.168.88.2"
  - "ip route 192.168.88.0/24 192.168.33.1"

vrf_name: management
route: "ip route 0.0.0.0/0 192.168.200.1"

vlans:
  - { id: 10, name: web }
  - { id: 11, name: qa }
  - { id: 12, name: prod }
  - { id: 13, name: test }
  - { id: 14, name: srvs }
  - { id: 15, name: video }
  - { id: 16, name: voice }
  - { id: 17, name: db2 }
  - { id: 18, name: web2 }
  - { id: 19, name: db }
  - { id: 1000, name: vlan_1000 }
  - { id: 3000, name: dummy }

interfaces:
  - { intf: Ethernet1/1, switchport: "no switchport", ip: "10.101.101.1/30", state: "no shutdown"}
  - { intf: Ethernet1/2, switchport: "no switchport", ip: "10.254.1.1/30", state: "no shutdown"}
  - { intf: Ethernet1/3, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202", state: "no shutdown"}
  - { intf: Ethernet1/4, switchport: switchport, mode: trunk, native_vlan: 3000, vlan_range: "100-102,200-202", state: "no shutdown"}
```

```
mgmt_ip: 192.168.200.11/24
```

N9K1_v2.j2 file

```
hostname {{ hostname }}

username {{ username }} password {{ password }}

banner motd *
{{ banner_motd}}
*

ip domain-name {{ domain }}

vtp mode {{ vtp_mode }}

snmp-server contact {{ snmp.contact }}
snmp-server location {{ snmp.location }}
snmp-server community {{ snmp.ro_string }} group network-operator
snmp-server community {{ snmp.rw_string }} group network-admin

ntp server {{ ntp_server }}

{% for route in routes %}
{{ route }}
{%- endfor %}

{% for vlan in vlans %}
vlan {{ vlan.id }}
{%- if vlan.name %}
  name {{ vlan.name }}
{%- endif %}
{%- endfor %}

vrf context {{ vrf_name}}
  ip domain-name {{ domain }}
  {{ route }}

{%- for interface in interfaces %}
interface {{ interface.intf }}
  {{ interface.switchport }}
{%- if interface.switchport == 'no switchport' %}
  ip address {{ interface.ip }}
{%- endif %}
{%- if interface.switchport == 'switchport' %}
  switchport mode {{ interface.mode }}
  switchport trunk native vlan {{ interface.native_vlan }}
  switchport trunk allowed vlan {{ interface.vlan_range }}
{%- endif %}
  {{ interface.state }}
{% endfor -%}

interface mgmt0
  vrf member management
```

```
 ip address {{ mgmt_ip }}
```