Construct Infrastructure Automation Workflow

Note: Be sure to review the Objectives and Job Aids links above for required information. Password Information and Command Lists for Detailed Lab Steps are in the Job Aids link.

Task 1: Test Ansible Task

You will install Ansible inside a Python virtual environment, create a valid inventory file, and test your Ansible installation by executing a simple task on your server using the ping module. Your task will make sure that Ansible can connect to your server and execute commands.

Show Steps

- **Step 1:**

  Launch Visual Studio Code from your desktop.

  Tip:

- **Step 2:**

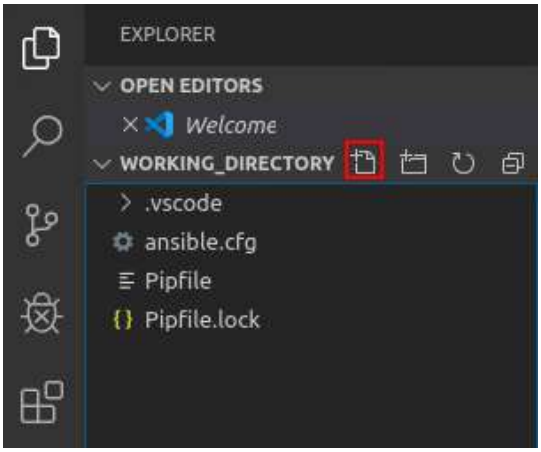
  Click **File > Open Folder**, then in the upper-left corner click **Home**, select **working_directory**, and click **OK**.

  Tip:

- **Step 3:**

  Now click the **New File** icon next to the WORKING_DIRECTORY folder name, and create a file named **inventory.cfg.**

  Tip:



- **Step 4:**

  In the created file, type in **server** on a single line. Save the file using **File > Save.**

- **Step 5:**

  Click **View > Terminal** to open a terminal in Visual Studio Code.

- **Step 6:**

  Type **pwd** to verify that you are in the /home/student/working_directory folder

  Tip:

```
student@student-vm:~/working_directory$ pwd
/home/student/working_directory
```

- **Step 7:**

  Install the Python virtual environment required Ansible packages by executing the **pipenv install** command.

  Tip:

```
student@student-workstation:~/working_directory$ pipenv install
Installing dependencies from Pipfile.lock (f1662d)…
       ████████████████████████████████████████ 28/28 — 00:00:04
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run
```

- **Step 8:**

  Now, execute the **pipenv shell** command to activate your new environment. Notice that your prompt now shows the name of your active virtual environment in parentheses (working_directory).

  Tip:

```
student@student-workstation:~/working_directory$ pipenv shell
Launching subshell in virtual environment…
student@student-workstation:~/working_directory$  . /home/student/.local/share/virtualenvs/working_directory-NLDM5Sxr/bin/activate
(working_directory) student@student-workstation:~/working_directory$
```

- **Step 9:**

  Use the command **ssh-keygen -t rsa** to create a new SSH key pair so that you will be able to connect to your server without a password. Press **enter** to use the default settings and overwrite the old key.

  Tip:

```
student@student-workstation:~/working-directory$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
/home/student/.ssh/id_rsa already exists.
Overwrite (y/n)? y
```

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:qYIDesta0q9gfllOrddwOumU1kiA1t9uNcG7F1R9p1E student@student-workstation
The key's randomart image is:
+---[RSA 2048]----+
|               oE|
|    o     .   o +|
|   o o     o . oo|
|  .   o ..  + .  |
|.     .oS. + .   |
|.o . o.+=.. o .  |
|+.* * o=*+ . .   |
|o* * +o=..  .    |
|.o=.. o..        |
+----[SHA256]-----+
```

- **Step 10:**

  Use the **ssh-copy-id root@server** command to copy the public key to the server for the root user. When asked, confirm the connection by entering **yes** and use the root's password **1234QWer**.

  Tip:

```
student@student-workstation:~/working_directory$ ssh-copy-id root@server
The authenticity of host 'server (192.168.0.20)' can't be established.
ECDSA key fingerprint is SHA256:XfYKvY//OzqVsnCMLHYexeN7V1pJtUoEQlUdn5ATWI0.
Are you sure you want to continue connecting (yes/no)? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
root@server's password: 1234QWer

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'root@server'"
and check to make sure that only the key(s) you wanted were added.
```

- **Step 11:**

  Use **ssh root@server** to test that you can connect without a password.

  Tip:

```
student@student-workstation:~/working_directory$ ssh root@server
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-62-generic x86_64)

Last login: Sun Dec  1 23:28:52 2019 from 192.168.0.10
root@server:~#
```

- **Step 12:**

  Enter **exit** to disconnect from the server.

  Tip:

```
root@server:~# exit
logout
Connection to server closed.
```

- **Step 13:**

  Use the command **ansible** to execute an ad-hoc command to confirm that you are able to execute commands. Use **server** as the pattern and the **ping** module.

  Tip:

  Previously you have inserted the hostname server into the inventory.cfg file, which Ansible uses this ad hoc command where executes a ping to the target server.

```
student@student-workstation:~/working_directory$ ansible server -m ping
server | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Task 2: Create Ansible Playbook

You will now construct a playbook and use it to provision your server. Your playbook will consist of one task, which will make sure that the SSH service is running and enabled on boot.

Show Steps

- **Step 1:**

  Create a new file inside your working_directory and name it **provision-server.yml**.

Note

Indentation is very important when writing playbooks. Always use spaces instead of tabs, and make sure there are spaces between every dash and after every colon character.

- **Step 2:**

  In the first line name the playbook **Initial system setup.**

  Tip:

```
- name: Initial system setup
```

- **Step 3:**

  In the second line with two spaces of indentation insert the server for hosts.

  Tip:

```
- name: Initial system setup
  hosts: server
```

- **Step 4:**

  Write a task with the name **Make sure ssh service is enabled upon system boot** and use the service module which enables the SSH service.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes
```

- **Step 5:**

  Save the provision-server.yml file and use the command **ansible-playbook --syntax-check provision-server.yml** to make sure that there are no syntax errors in your playbook.

  Tip:

```
student@student-workstation:~/working_directory$ ansible-playbook --syntax-check provision-server.yml
playbook: provision-server.yml
student@student-workstation:~/working_directory$
```

- **Step 6:**

  Execute the playbook using the command **ansible-playbook provision-server.yml** and check the returns of the tasks in the **PLAY RECAP -** the play report.

  Tip:

```
student@student-workstation:~/working_directory$ ansible-playbook provision-server.yml
PLAY [Initial system setup] ****************************************************************************************
TASK [Gathering Facts] *********************************************************************************************
ok: [server]
TASK [Make sure ssh service is enabled upon system boot] ***********************************************************
ok: [server]
PLAY RECAP *********************************************************************************************************
server                     : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
student@student-workstation:~/working_directory$
```

Note

Your play reports that two tasks returned "ok," because the Gathering Facts task does not make any changes, and the SSH service is already enabled upon system restart.

Task 3: Perform Initial System Setup

You will now write a Bash script for generating a Linux-compatible password hash, and then you will create a task that will create a new user admin that uses that password hash.

Show Steps

- **Step 1**:

  Create a new file named **make_password.sh** in which you will write a Bash script.

- **Step 2**:

  At the top of the file write the shebang and the /bin/bash as the interpreter.

  Tip:

  ```
  #!/bin/bash
  ```

- **Step 3**:

  Add a comment **force caller to enter password twice**.

  Tip:

  ```
  #!/bin/bash

  # force caller to enter password twice
  ```

- **Step 4**:

  Next use the **read** command with the switches **-s** and **-p** to read the command into the variable *pass1*. Set the output text to **"**Enter password: ".

  Tip:

  ```
  #!/bin/bash

  # force caller to enter password twice
  read -s -p 'Enter password: ' pass1
  ```

- **Step 5**:

  Again, write the same line, but change the variable to pass2 and the text to "Repeat password: ".

  Tip:

  ```
  #!/bin/bash

  # force caller to enter password twice
  read -s -p 'Enter password: ' pass1
  read -s -p 'Repeat password: ' pass2
  ```

- **Step 6**:

  Add a comment with the text "make sure passwords match".

  Tip:

  ```
  #!/bin/bash
  # force caller to enter password twice
  read -s -p 'Enter password: ' pass1
  read -s -p 'Repeat password: ' pass2

  # make sure passwords match
  ```

- **Step 7**:

  Write an if statement that checks whether the two passwords are not equal.

  Tip:

  ```
  #!/bin/bash
  # force caller to enter password twice
  read -s -p 'Enter password: ' pass1
  ```

```
read -s -p 'Repeat password: ' pass2

# make sure passwords match
if [ $pass1 != $pass2 ]; then
```

- **Step 8**:

  In case the two passwords do not match return the text "passwords do not match" and exit the script with an error 2. Close the if statement.

  Tip:

```
#!/bin/bash
# force caller to enter password twice
read -s -p 'Enter password: ' pass1
read -s -p 'Repeat password: ' pass2

# make sure passwords match
if [ $pass1 != $pass2 ]; then
        echo "passwords do not match"
        exit 2
fi
```

- **Step 9**:

  Add a comment "return sha512 hash"

  Tip:

```
#!/bin/bash
# force caller to enter password twice
read -s -p 'Enter password: ' pass1
read -s -p 'Repeat password: ' pass2

# make sure passwords match
if [ $pass1 != $pass2 ]; then
        echo "passwords do not match"
        exit 2
fi

# return sha512 hash
```

- **Step 10:**

  Pipe the output of the **echo $pass1** command into the command **mkpasswd --method=sha-512 --stdin** to output a password hash.

  Tip:

```
#!/bin/bash

# force caller to enter password twice
read -s -p 'Enter password: ' pass1
read -s -p 'Repeat password: ' pass2

# make sure passwords match
if [ $pass1 != $pass2 ]; then
        echo "passwords do not match"
        exit 2
fi

# return sha512 hash
echo $pass1 | mkpasswd --method=sha-512 --stdin
```

- **Step 11:**

  Save the file. In the terminal, use the **chmod** command with the numeric mode **700** to make the created script executable for the owner of the file.

  Tip:

```
student@student-workstation:~/working_directory$ chmod 700 make_password.sh
```

- **Step 12:**

  Run the script and provide **1234QWer** twice as a password. Copy the created hash.

  Tip:

```
student@student-workstation:~/working_directory$ ./make_password.sh
Enter password:
Repeat password:
$6$3m/sStC8g8qm/O$9q9fkou1VXVEr7TD/LWoAKAFXyQ7XXNHQYKTPcEJ/tSBlgjc8n1uYYDx58n.rOW2ZRcsoWr2AZfkrOsRL3FdC1
```

Note

Your password hash will not be the same as the one shown here.

- **Step 13:**

  Open the **provision-server.yml** file and create a task that will create an admin group for the admin user.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present
```

- **Step 14:**

  Create a task that will create an admin user, and use the password hash from your make_password.sh script for the password value.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present

    - name: Create admin user
      user:
        name: admin
        group: admin
        home: /home/admin
        password: $6$EsC9meJyM$eOumIORpN7GEzlt6A.Sev2OPv9wUjiTQniy2WZ2WYQogReCUtL6Zke8EpLsBE5JIqQkr5aKGSdUNS5z/bqDls/
        shell: /bin/bash
        state: present
```

Task 4: Configure Service

You will now install and configure a TFTP service (tftpd-hpa) that allows your routers to save their configuration every time a change occurs.

You need to make sure that routers can write their configuration inside the /var/lib/tftpboot directory by changing the configuration file with the lineinfile module.

Show Steps

- **Step 1:**

  Create a new task in your playbook. Name the task "Install TFTP server packages" and use the **package** module to install the **tftpd-hpa** package.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present

    - name: Create admin user
      user:
        name: admin
        group: admin
        home: /home/admin
        password: $6$EsC9meJyM$eOumIORpN7GEzlt6A.Sev2OPv9wUjiTQniy2WZ2WYQogReCUtL6Zke8EpLsBE5JIqQkr5aKGSdUNS5z/bqDls/
        shell: /bin/bash
```

```
      state: present

  - name: Install TFTP server packages
    package: name=tftpd-hpa state=present
```

- **Step 2:**

  Create a task named "Configure TFTP server for write access". Use the **lineinfile** module to add the line *TFTP_OPTIONS="--secure --verbose -4 -c"* to the file /etc/default/tftpd-hpa which will enable write operations for TFTP clients.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present

    - name: Create admin user
      user:
        name: admin
        group: admin
        home: /home/admin
        password: $6$EsC9meJyM$eOumIORpN7GEzlt6A.Sev2OPv9wUjiTQniy2WZ2WYQogReCUtL6Zke8EpLsBE5JIqQkr5aKGSdUNS5z/bqDls/
        shell: /bin/bash
        state: present

    - name: Install TFTP server packages
      package: name=tftpd-hpa state=present

    - name: Configure TFTP server for write access
      lineinfile:
        path: /etc/default/tftpd-hpa
        line: 'TFTP_OPTIONS="--secure --verbose -4 -c"'
        state: present
```

- **Step 3:**

  Create a task named "Configure /var/lib/tftpboot permissions" and the **file** module to set the permissions to the directory /var/lib/tftpboot. Set the owner to **root**, group to **tftp** and mode to **0775**.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present

    - name: Create admin user
      user:
        name: admin
        group: admin
        home: /home/admin
        password: $6$EsC9meJyM$eOumIORpN7GEzlt6A.Sev2OPv9wUjiTQniy2WZ2WYQogReCUtL6Zke8EpLsBE5JIqQkr5aKGSdUNS5z/bqDls/
        shell: /bin/bash
        state: present

    - name: Install TFTP server packages
      package: name=tftpd-hpa state=present

    - name: Configure TFTP server for write access
      lineinfile:
        path: /etc/default/tftpd-hpa
        line: 'TFTP_OPTIONS="--secure --verbose -4 -c"'
        state: present

    - name: Configure /var/lib/tftpboot permissions
      file: path=/var/lib/tftpboot owner=root group=tftp mode=0775
```

- **Step 4:**

  Create a task named "Start tftpd-hpa service and enable it upon system boot" and use the **service** module to enable the **tftpd-hpa** service. Set the state to **restarted**.

  Tip:

```
- name: Initial system setup
  hosts: server

  tasks:
    - name: Make sure ssh service is enabled upon system boot
      service: name=ssh enabled=yes

    - name: Create admin group
      group: name=admin state=present

    - name: Create admin user
      user:
        name: admin
        group: admin
        home: /home/admin
        password: $6$EsC9meJyM$eOumIORpN7GEzlt6A.Sev2OPv9wUjiTQniy2WZ2WYQogReCUtL6Zke8EpLsBE5JIqQkr5aKGSdUNS5z/bqDls/
        shell: /bin/bash
        state: present

    - name: Install TFTP server packages
      package: name=tftpd-hpa state=present

    - name: Configure TFTP server for write access
      lineinfile:
        path: /etc/default/tftpd-hpa
        line: 'TFTP_OPTIONS="--secure --verbose -4 -c"'
        state: present

    - name: Configure /var/lib/tftpboot permissions
      file: path=/var/lib/tftpboot owner=root group=tftp mode=0775

    - name: Start tftpd-hpa service and enable it upon system boot
      service: name=tftpd-hpa state=restarted enabled=yes
```

- **Step 5**:

  Save the file and check the syntax using the command **ansible-playbook --syntax-check provision-server.yml**.

  Tip:

```
student@student-workstation:~/working_directory$ ansible-playbook --syntax-check provision-server.yml

playbook: provision-server.yml
```

- **Step 6**:

  Execute the playbook using the **ansible-playbook provision-server.yml** command.

  Tip:

```
student@student-workstation:~/working_directory$ ansible-playbook provision-server.yml
<...ouput omitted...>
server                     : ok=8    changed=6    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
student@student-workstation:~/working_directory$
```

Task 5: Configure Device

Your Cisco NSO is not managing your routers yet. You will now write a Python script that interacts with Cisco NSO using the REST API, and create a new device named csr1 based on information in XML format.

Hide Steps

- **Step 1**:

  Create a new file in your working_directory and name it **csr1.xml.** This file will contain basic information that informs your Cisco NSO how to connect to your device.

- **Step 2**:

  Add a device tag with the attribute xmlns set to "http://tail-f.com/ns/ncs", add also the closing tag.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
</device>
```

- **Step 3**:

  Into the device element insert an element with the tag **name** and set its text to **csr1**.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
</device>
```

- **Step 4**:

  Into the device element add a tag **address** and set its text to **192.168.0.30**.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
</device>
```

- **Step 5**:

  Into the device element add a **port** tag and set its text to **22**.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
</device>
```

- **Step 6**:

  Into the device element add element **authgroup** and set its value to **default**.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
  <authgroup>default</authgroup>
</device>
```

- **Step 7**:

  Into the device element add tags **state** and **device-type** with their closing tags.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
  <authgroup>default</authgroup>
  <state>
  </state>
  <device-type>
  </device-type>
</device>
```

- **Step 8**:

  Add tag **admin-state** with the text set to **unlocked** into the state element.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
  <authgroup>default</authgroup>
  <state>
    <admin-state>unlocked</admin-state>
  </state>
  <device-type>
  </device-type>
</device>
```

- **Step 9**:

  Into the element device-type add a **cli** tag.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
  <authgroup>default</authgroup>
```

```
  <state>
    <admin-state>unlocked</admin-state>
  </state>
  <device-type>
    <cli>
    </cli>
  </device-type>
</device>
```

- **Step 10:**

  Finally, into the cli element add an **ned-id** tag with the attribute **xmlns:cisco-ios-cli-3.8="http://tail-f.com/ns/ned-id/cisco-ios-cli-3.8"** and text **cisco-ios-cli-3.8:cisco-ios-cli-3.8**.

  Tip:

```
<device xmlns="http://tail-f.com/ns/ncs">
  <name>csr1</name>
  <address>192.168.0.30</address>
  <port>22</port>
  <authgroup>default</authgroup>
  <state>
    <admin-state>unlocked</admin-state>
  </state>
  <device-type>
    <cli>
      <ned-id xmlns:cisco-ios-cli-3.8="http://tail-f.com/ns/ned-id/cisco-ios-cli-3.8">cisco-ios-cli-3.8:cisco-ios-cli-3.8</ned-id>
    </cli>
  </device-type>
</device>
```

- **Step 11:**

  Now, create a new file in your working_directory and name it **nso_device.py.** This script will make use of your XML file and send it to Cisco NSO.

- **Step 12:**

  At the top of the file import the requests module for sending HTTP requests to Cisco NSO.

  Tip:

```
import requests
```

- **Step 13:**

  Add REST API variables for credentials, URL, headers, and the source of XML data for the new device.

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
```

- **Step 14:**

  Using the requests module create a session object and store it in a variable named *api_session*.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
```

- **Step 15:**

  Set the auth attribute of the *api_session* object to *(API_USER, API_PASS)* tuple.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)
```

- **Step 16:**

  Add a comment for the next part of the code that will create an NSO device from the before created XML. Create a new variable *api_endpoint* and set its value to a concatenated string of API_BASE and "/api/running/devices/device/csr1". This URL points to the CSR1 device in Cisco NSO.

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
```

- **Step 17:**

  Use the **with open** to read the XML and send the *xml* using the **put** method of the *api_session* object to the URL stored in the variable *api_endpoint*. Set the headers to *API_HEAD* and save the response into the *api_response* variable.

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
```

- **Step 18:**

  Use the print method to print the *api_endpoint* variable and the returned status code stored in the *api_response*.

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
```

```
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')
```

- **Step 19:**

  Create a comment for a segment of the code in which you will instruct Cisco NSO to fetch SSH host keys from the CSR1 device.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
```

- **Step 20:**

  Set the variable *api_endpoint* to a concatenated string of *API_BASE* and "/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys". This URL points to the fetch-host-keys action in Cisco NSO.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
```

- **Step 21:**

  Send the request using the **post** method of the *api_session* object to the URL stored in the variable api_endpoint. Set the headers to *API_HEAD* and save the response into the api_response variable.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
```

```
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
```

- **Step 22:**

  Use the print method to print the *api_endpoint* variable and the returned status code stored in the *api_response*.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')
```

- **Step 23:**

  Write a comment for the last part of the code in which you will instruct Cisco NSO to get current configuration from the device and save it. This way Cisco NSO is aware of the device state.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to sync configuration from device
```

- **Step 24:**

  Set the variable *api_endpoint* to a concatenated string of *API_BASE* and "/api/running/devices/device/csr1/_operations/sync-from". This URL points to the sync-from action in NSO.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to sync configuration from device
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/_operations/sync-from'
```

- **Step 25:**

  Send the request using the **post** method of the *api_session* object to the URL stored in the variable *api_endpoint*. Set the headers to API_HEAD and save the response into the *api_response* variable.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to sync configuration from device
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/_operations/sync-from'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
```

- **Step 26:**

  Use the print method to print the *api_endpoint* variable and the returned status code stored in the *api_response*.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}

api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# create nso device from csr1.xml
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1'
with open('csr1.xml') as xml:
    api_response = api_session.put(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PUT: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to fetch ssh keys
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')

# tell nso to sync configuration from device
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/_operations/sync-from'
api_response = api_session.post(api_endpoint, headers=API_HEAD)
print(f'-> POST: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')
```

- **Step 27:**

  Save the file. Run your script using the command **python nso_device.py** and check the three responses. Your device can now be provisioned from Cisco NSO. When asked, install Linter Pylint by clicking the install button.

  Tip:

```
student@student-workstation:~/working_directory$ python nso_device.py
-> PUT: http://nso:8080/api/running/devices/device/csr1
  -> RESPONSE: 201
-> POST: http://nso:8080/api/running/devices/device/csr1/ssh/_operations/fetch-host-keys
  -> RESPONSE: 200
-> POST: http://nso:8080/api/running/devices/device/csr1/_operations/sync-from
  -> RESPONSE: 200
```

Task 6: Configure Archive

Your device is now ready to be configured by Cisco NSO. You will now write a Python script that enables archiving to the TFTP service running on your Student Workstation.

Show Steps

- **Step 1:**

  Create a new file in your working_directory, name it **archive.xml** and open it. This configuration for the CSR router will configure the archive path when the write-memory command is executed.

- **Step 2:**

  Add a **config** tag and its closing tag.

  Tip:

```
<config>
</config>
```

- **Step 3:**

  Into the config element add an **archive** tag with the **xmlns** attribute set to **urn:ios** and also add its closing tag.

  Tip:

```
<config>
  <archive xmlns="urn:ios">
  </archive>
</config>
```

- **Step 4:**

Into the archive element add a path tag with the text set to tftp://192.168.0.20/$h.

Tip:

```
<config>
  <archive xmlns="urn:ios">
    <path>tftp://192.168.0.20/$h</path>
  </archive>
</config>
```

- **Step 5:**

  Add a self-closing tag **write-memory** into the archive element.

  Tip:

```
<config>
  <archive xmlns="urn:ios">
    <path>tftp://192.168.0.20/$h</path>
    <write-memory/>
  </archive>
</config>
```

- **Step 6:**

  Create a new file in your working_directory and name it **nso_archive.py.** This script will make use of the created XML file and send it to Cisco NSO.

- **Step 7:**

  Edit the file. The first part of the script is the same as in **nso_device.py**.

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)
```

- **Step 8:**

  In the main part of the script you will use the archive.xml file to configure the CSR router using NSO. Write a short comment before the code.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# use archive.xml to configure csr1 using nso
```

- **Step 9:**

  Set the variable *api_endpoint* to a concatenated string of *API_BASE* and "/api/running/devices/device/csr1/config".

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
```

```
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# use archive.xml to configure csr1 using nso
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/config'
```

- **Step 10:**

  Use the with open to read the XML and send the xml using the **patch** method of the *api_session* object to the URL stored in the variable *api_endpoint*. Set the headers to *API_HEAD* and save the response into the *api_response* variable.

  Tip:

```
import requests

# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)

# use archive.xml to configure csr1 using nso
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/config'
with open('archive.xml') as xml:
    api_response = api_session.patch(api_endpoint, headers=API_HEAD, data=xml)
```

Note

The PATCH method is used on Cisco NSO objects that already exist; that way, you update a part of the device configuration.

- **Step 11:**

  Use the print method to print the *api_endpoint* variable and the returned status code stored in the *api_response*.

  Tip:

```
import requests
# credentials
API_USER = 'admin'
API_PASS = 'admin'
# nso server address
API_BASE = 'http://nso:8080'
# api headers
API_HEAD = {
    'Accept': 'application/vnd.yang.data+xml'
}
api_session = requests.Session()
api_session.auth = (API_USER, API_PASS)
# use archive.xml to configure csr1 using nso
api_endpoint = f'{API_BASE}/api/running/devices/device/csr1/config'
with open('archive.xml') as xml:
    api_response = api_session.patch(api_endpoint, headers=API_HEAD, data=xml)
print(f'-> PATCH: {api_endpoint}')
print(f'  -> RESPONSE: {api_response.status_code}')
```

- **Step 12:**

  Save the script and run it. Check the response.

  Tip:

```
student@student-workstation:~/working_directory$ python nso_archive.py
-> PATCH: http://nso:8080/api/running/devices/device/csr1/config
  -> RESPONSE: 204
```

Task 7: Verify Service Provisioning

You will now verify that your Student Workstation and TFTP service are working and that your router is using the service to archive the configuration every time that the configuration is saved.

Show Steps

- **Step 1:**

  Use SSH to connect to CSR1 router with username **cisco** and accept an RSA fingerprint. When prompted, use **cisco** for password.

  Tip:

```
student@student-workstation:~/working_directory$ ssh cisco@csr1
The authenticity of host 'csr1 (192.168.0.30)' can't be established.
RSA key fingerprint is SHA256:XQn/9LIaHRFpukxs49PbAa1wV7+A/Vtfr2ZI4Qnd2/Y.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'csr1' (RSA) to the list of known hosts.
Password:
csr1kv1#
```

- **Step 2:**

  Run the **show archive** command and verify that the router successfully wrote the configuration to your TFTP server.

  Tip:

```
csr1kv1# show archive
The maximum archive configurations allowed is 10.
The next archive file will be named tftp://192.168.0.20/csr1kv1-<timestamp>-1
 Archive #  Name
   1        tftp://192.168.0.20/csr1kv1-Nov--4-06-07-04-PST-0 <- Most Recent
   2
   3
   4
   5
   6
   7
   8
   9
   10
csr1kv1#
```

- **Step 3:**

  Run the **write** command to save the configuration and archive it to your server again.

  Tip:

```
csr1kv1# write
Building configuration...
[OK]!
csr1kv1#
```

- **Step 4:**

  Run the **show archive** command again and see that the most recent archive is now number 2.

  Tip:

```
csr1kv1# show archive
The maximum archive configurations allowed is 10.
The next archive file will be named tftp://192.168.0.20/csr1kv1-<timestamp>-2
 Archive #  Name
   1        tftp://192.168.0.20/csr1kv1-Nov--4-06-07-04-PST-0
   2        tftp://192.168.0.20/csr1kv1-Nov--4-06-22-29-PST-1 <- Most Recent
   3
   4
   5
   6
   7
   8
   9
   10
csr1kv1#
```

Lab Completion Instructions

You have now completed this lab exercise.
Please click '**End Session**'.

Choose '**Exit**'.