

Jenkins Pipeline: Initial Setup and Unit Tests

Overview

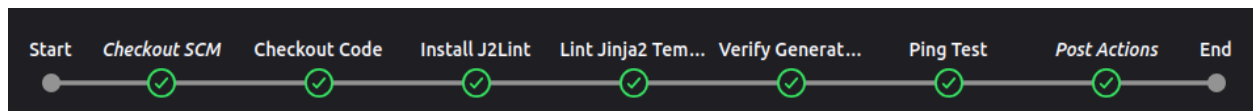
The Jenkins pipeline is configured to automate the configuration validation, file existence check, and connectivity check, enabling a reliable network automation CI/CD process. The pipeline stages are as follows:

```
pipeline {
    agent any

    stages {
        stage('Checkout Code') { /* Details */ }
        stage('Install J2Lint') { /* Details */ }
        stage('Lint Jinja2 Templates') { /* Details */ }
        stage('Verify Generated Config Exists') { /* Details */ }
        stage('Ping Test') { /* Details */ }
    }

    post {
        success { echo 'Jenkins Job successful. No errors found!' }
        failure { echo 'Jenkins Job failed. Please check the errors!' }
    }
}
```

Stages Breakdown



Stage 1: Checkout Code

- **Purpose:** Pulls the latest code, including templates, from the GitHub repository.
- **Implementation:** Uses checkout scm to sync the current repository state in Jenkins with the remote repository, ensuring up-to-date code for the CI/CD process.

- **Expected Output:** Repository code, including Jinja2 templates, is available for the next steps.

Stage 2: Install J2Lint

- **Purpose:** Ensures that j2lint is installed, which is used for linting Jinja2 templates.
- **Implementation:** Uses pip install --user j2lint to install j2lint in the user's environment if it isn't already installed.
- **Expected Output:** j2lint is available for the linting stage.

Stage 3: Lint Jinja2 Templates

- **Purpose:** Validates the syntax of Jinja2 templates to ensure there are no syntax errors.
- **Implementation:** Runs j2lint on all templates in the template-generator/templates/ directory. Errors are flagged if any syntax issues are found.
- **Expected Output:** Successful linting output or error messages indicating syntax issues.

```
[Pipeline] stage
[Pipeline] { (Lint Jinja2 Templates)
[Pipeline] sh
+ export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin:/home/student/.local/bin
+ j2lint template-generator/templates/access.j2 template-generator/templates/bgp.j2 template-generator/templates/
core.j2 template-generator/templates/dhcp.j2 template-generator/templates/edge.j2 template-generator/templates/
interfaces.j2 template-generator/templates/ospf.j2 template-generator/templates/ospfv3.j2 template-generator/
templates/rip.j2 template-generator/templates/snmp.j2 template-generator/templates/static_routes.j2 template-
generator/templates/virtual_router.j2 template-generator/templates/vlan.j2 template-generator/templates/
vlan_interfaces.j2
[Pipeline] }
[Pipeline] // stage
```

Screenshot 1: Linting successful

Stage 4: Verify Generated Config Exists

- **Purpose:** Ensures that at least one YAML configuration file is present in the generated-configs directory before proceeding.
- **Implementation:** Checks for .yaml files in generated-configs using a shell script. If no files are found, Jenkins raises an error, terminating the pipeline.
- **Expected Output:** Confirmation that a configuration file exists, or an error if none are found.

```
[Pipeline] stage
[Pipeline] { (Verify Generated Config Exists)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ls /home/student/git/csci5840/template-generator/generated-configs/r9_core.yaml
[Pipeline] echo
Configuration file exists: /home/student/git/csci5840/template-generator/generated-configs/r9_core.yaml
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
```

Screenshot 2: Configuration file exists

```
[Pipeline] stage
[Pipeline] { (Verify Generated Config Exists)
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ls /home/student/git/csci5840/template-generator/generated-configs/*.yaml
+ echo not found
[Pipeline] error
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Ping Test)
Stage "Ping Test" skipped due to earlier failure(s)
[Pipeline] getContext
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Jenkins Job failed. Please check the errors!
```

Screenshot 3: Configuration file does not exist

Stage 5: Ping Test

- **Purpose:** Confirms connectivity to the device before proceeding with configuration deployment.
- **Implementation:**
 - Identifies the latest YAML file in generated-configs and extracts the device name from the filename.
 - Executes a ping test to verify that the device is reachable. An error is raised if the device is unreachable.
- **Expected Output:** Successful ping response from the target device or an error message if the device is unreachable.

```
[Pipeline] sh
+ ls -t /home/student/git/csci5840/template-generator/generated-configs/r8_access.yaml
+ head -n 1
[Pipeline] sh
+ ping -c 4 r8
PING r8 (172.20.20.8) 56(84) bytes of data.
64 bytes from r8 (172.20.20.8): icmp_seq=1 ttl=64 time=0.078 ms
64 bytes from r8 (172.20.20.8): icmp_seq=2 ttl=64 time=0.098 ms
64 bytes from r8 (172.20.20.8): icmp_seq=3 ttl=64 time=0.082 ms
64 bytes from r8 (172.20.20.8): icmp_seq=4 ttl=64 time=0.087 ms

--- r8 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3056ms
rtt min/avg/max/mdev = 0.078/0.086/0.098/0.007 ms
[Pipeline] echo
Ping test successful for device: r8
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
```

Screenshot 4: Ping test

Post Conditions

- **Success:** Marks the Jenkins job as successful, enabling the “Push Config” button on the frontend.
- **Failure:** Provides error details, highlighting the stage that caused the failure, and displays these on the frontend.

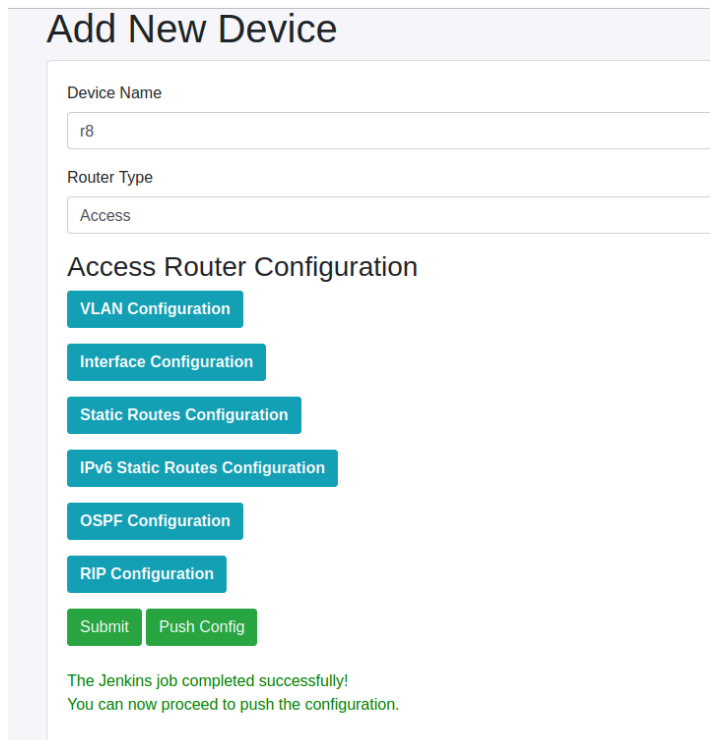
Frontend: Push Config Button

Once all Jenkins pipeline tests pass, the “Push Config” button on the frontend becomes active, allowing the user to proceed with pushing configurations to the device.



The screenshot shows a web interface with a navigation bar at the top containing 'Home', 'Grafana Dashboard', and 'Add New Device'. The main heading is 'Add New Device'. Below it is a form with two input fields: 'Device Name' with a placeholder 'Enter device name' and 'Router Type' with a placeholder 'Select Router Type'. At the bottom of the form are two green buttons: 'Submit' and 'Push Config'.

Screenshot 5: Before Jenkins job is run



The screenshot shows the same 'Add New Device' form, but now the 'Device Name' field contains 'r8' and the 'Router Type' field contains 'Access'. Below the form, there is a section titled 'Access Router Configuration' with a list of configuration options, each in a blue button: 'VLAN Configuration', 'Interface Configuration', 'Static Routes Configuration', 'IPv6 Static Routes Configuration', 'OSPF Configuration', and 'RIP Configuration'. At the bottom of this section are two green buttons: 'Submit' and 'Push Config'. Below the buttons, a green message states: 'The Jenkins job completed successfully! You can now proceed to push the configuration.'

Screenshot 6: After the Jenkins Job is successful

Add New Device

Device Name

r8

Router Type

Access

Access Router Configuration

VLAN Configuration

Interface Configuration

Static Routes Configuration

IPv6 Static Routes Configuration

OSPF Configuration

RIP Configuration

Submit Push Config

Jenkins job failed.
Please review the errors in the Jenkins console for more details.

Screenshot 7: After the Jenkins Job is failed

Push Config Workflow

Upon activation and clicking the “Push Config” button, the following steps are triggered:

1. Configuration Generation

- The generate_config.py script is executed, which:
 - Uses the YAML file in generated-configs to generate a device configuration.
 - Saves the generated configuration file as <device_name>.cfg in the generated-configs directory.
- **Expected Output:** Configuration file <device_name>.cfg is generated in the generated-configs directory.

```
student@csci5840-vm2-snir8112:~/git/csci5840/template-generator/generated-configs$ ls
r8_access.yaml  r8.cfg
```

Screenshot 8: .cfg generated

2. Push Config to Device

- Once the configuration file is generated, the script establishes an SSH connection to the device using Netmiko.
- The generated configuration (<device_name>.cfg) is pushed to the device.
- **Expected Output:** Configuration is successfully deployed on the device.

3. Cleanup and Confirmation

- Upon successful deployment, both the YAML file and the .cfg configuration file are deleted from the generated-configs directory.
- A success message is displayed on the frontend, confirming the deployment.
- **Expected Output:** The frontend confirms successful deployment, and the generated files are removed from the repository.

```
student@csci5840-vm2-snir8112:~/git/csci5840/template-generator/generated-configs$ ls
r8_access.yaml  r8.cfg
student@csci5840-vm2-snir8112:~/git/csci5840/template-generator/generated-configs$ ls
student@csci5840-vm2-snir8112:~/git/csci5840/template-generator/generated-configs$ ssh admin@
```

Screenshot 9: Automatic cleanup on successful push

Coverage

File 1: nethealth.py

This file contains the main functionality for your NetHealth script. It provides the core logic for connecting to network devices, extracting health metrics, and performing network health checks. Here's what each main component does:

1. sshInfo() Function:

- Reads the sshInfo.csv file and extracts device connection information (like IP, username, password, device type).
- Returns this data as a dictionary, which is used by other functions to connect to devices.

2. extract_cpu_usage() Function:

- Parses CPU usage information from command output.
- It looks specifically for a pattern like "5.6 sy" to extract the system usage percentage (the sy field), formatting it as 5.6%.

- If it doesn't find the expected pattern, it returns an error message.

3. extract_ospf_neighbors() Function:

- Parses OSPF neighbor information from command output, extracting details like Neighbor ID and State.
 - Helps verify the connectivity and status of OSPF neighbors by organizing the information in a human-readable format.
-

File 2: test_netman.py

This is the unit test file for nethealth.py. It uses Python's unittest library to perform automated tests on the NetHealth functions by connecting to devices listed in sshInfo.csv. Here's a breakdown of its main sections and what each test does:

1. Test Setup with setUp() Method:

- Runs before each test method.
- Calls sshInfo() to load the device information (from sshInfo.csv) into self.devices.
- Ensures that the device data loads successfully, which is critical for the rest of the tests.

2. test_cpu_usage():

- This test, when enabled, would connect to each device, run a CPU usage command, and check if extract_cpu_usage() can correctly parse the CPU usage.
- Verifies the format of the parsed CPU usage (should match a percentage format like 5.6%).
- Right now, this function is commented out, likely for debugging purposes due to prior parsing issues.

3. test_ospf_neighbors():

- Connects to each device and runs a command to retrieve OSPF neighbor information.
- Parses the output with extract_ospf_neighbors() to confirm if neighbors are detected.
- Checks that the parsed output contains "Neighbor:" to verify that neighbor data is present.

4. test_bgp_neighbors():

- Connects to each device, runs a BGP summary command, and verifies that BGP neighbor information is retrieved.
- Ensures the output is non-empty, which would indicate active BGP neighbors or connections.

5. Subtests with subTest:

- Each test uses subTest to handle multiple devices individually. If one device fails, it won't prevent the other devices from being tested.

How the Tests Run

When you run `coverage run -m unittest discover`, here's the sequence of events:

```
student@csci5840-vm2-snir8112:~/git/csci5840/scripts$ coverage run --source=test_netman -m unittest discover
.....
-----
Ran 9 tests in 8.825s
OK
```

Screenshot 10: Running the unit test file

1. Initialization:

- The setUp() method loads device information before each test, preparing self.devices for the following tests.

2. Testing Each Health Check:

- test_cpu_usage(), test_ospf_neighbors(), and test_bgp_neighbors() each connect to devices using the details in self.devices.
- Each test sends a command to check specific health data (CPU, OSPF, BGP) and verifies that the output meets expected formats or contains key data.

3. Output Verification:

- Each test uses assertions (assertRegex, assertIn, assertTrue) to confirm that the parsed data meets the expected criteria.
- If any test fails, it outputs a detailed error message for easier debugging.

```
student@csci5840-vm2-snir8112:~/git/csci5840/scripts$ coverage report -m
Name          Stmts  Miss  Cover   Missing
-----
test_netman.py    25     0   100%
TOTAL            25     0   100%
```

Screenshot 11: Coverage report

file:///home/student/git/csci5840/scripts/htmlcov/function_index.html					
Coverage report: 100%					
<div>FilesFunctionsClasses</div>					
coverage.py v7.6.4, created at 2024-10-28 21:03 -0600					
File ▲	function	statements	missing	excluded	coverage
test_netman.py	TestDeviceHealthChecks.setUp	2	0	0	100%
test_netman.py	TestDeviceHealthChecks.test_ospf_neighbors	9	0	0	100%
test_netman.py	TestDeviceHealthChecks.test_bgp_neighbors	7	0	0	100%
test_netman.py	(no function)	7	0	0	100%
Total		25	0	0	100%
coverage.py v7.6.4, created at 2024-10-28 21:03 -0600					

Screenshot 12: Coverage report in HTML