NAF Workshop WS:A2 - AutoCon3

Network Testing with NUTS

Urs Baumann, Steinn Örvar Bjarnarson

None

Table of contents

1. Workshop FAQ: Network Testing with NUTS

Proctor(s): Urs Baumann, Steinn Örvar Bjarnarson

Warning

We recommend using Python 3.12 for this lab

Use the specific version with `uv`

```
uv venv --python 3.12
```

1.1 General Information

1.1.1 1. What is this workshop about?

This hands-on workshop introduces NUTS (Network Unit Testing System) for network testing. Participants will learn the fundamentals of NUTS and pytest before applying them to test a simple network. The workshop also covers custom test case development and provides an introduction to the INPG Stack.

Topics:

- Introduction to NUTS and pytest
- Performing basic network testing with NUTS
- Customizing pytest test reports
- Developing custom test cases
- Overview of the INPG Stack (Infrahub, NUTS, Prometheus, Grafana)

This workshop is ideal for those looking to enhance their network automation and testing skills with Python.

1.1.2 2. What experience level is required?

This workshop is designed for participants with an intermediate level of experience in Python and network automation. You should be comfortable working with Python and have some familiarity with pytest, Napalm, and other network automation libraries. Prior experience with network testing or automation frameworks will be beneficial but is not strictly required.

1.2 Pre-Workshop Preparation

1.2.1 3. What do I need to install before the workshop?

- **Python (version 3.10 or above)**
- **Python venv**: To avoid messing up your local Python setup, you should be able to create a virtual Python environment or work in a dedicated container/VM.
- **Unix**: It is recommended that you work on a Unix-based operation system like Linux, MacOS or WSL.
- **Libraries**: NUTS, NAPALM, Nornir, Pytest
- **Software**: containerlab.dev, Arista cEOS 4.33.1F
- **IDE**: You can use the IDE you desire. The proctors are most comfortable with VSCode.

1.2.2 4. Are there any pre-reading materials?

There is no mandatory pre-reading, but it does not harm to familiarize yourself with the official documentation of the libraries we will cover:

- NUTS Documentation
- NUTS Git Repo
- containerlab
- uv
- Nornir
- Napalm Mock Driver
- Napalm Mock data example

1.2.3 5. Do I need to bring any equipment?

Yes, please bring a laptop with Python and the necessary libraries pre-installed. Ensure that your laptop is configured with the appropriate permissions to install and run software.

1.2.4 6. Can I use GitHub Codespaces?

Yes. All the labs work well with GitHub Codespaces. Make sure your free quota is not exceeded.

1.3 Own Ideas

1.3.1 7. Can I implement my own ideas?

Definitely, it is appriciated to consider how you can apply the learned material to your specific use case. However, time is limited, and the proctors cannot focus on each idea in detail.

1.3.2 8. Lab access?

You can set up and run your own lab on your computer or connect to an external lab. However, keep in mind that internet bandwidth is shared among all attendees. Graphical transmissions, such as remote desktop sessions or video streaming, are discouraged as they consume excessive bandwidth. For the best performance, use SSH-based access whenever possible.

1.4 Technical Questions

1.4.1 9. Will the workshop be hands-on?

Yes! This is a practical, hands-on workshop. You will be actively writing and running Python code to create CLI tools throughout the session.

1.5 Additional Information

1.5.1 10. Who do I contact if I have questions before the workshop?

For any questions or concerns prior to the workshop, feel free to contact Urs Baumann directly through the Network Automation Forum channels.

1.5.2 11. What if I cannot keep up with the pace of the workshop?

This workshop is designed for an intermediate skill level, but don’t worry if you fall behind. The proctors are happy to assist during the session, and you can also use breaks to catch up. Additionally, each section includes checkpoint solutions to help you stay on track. The goal is to ensure everyone can follow along and gain hands-on experience.

2. Agenda

2.1 Agenda

2.1.1 09:00-09:15

Theory - Introduction: Welcome & setup check, intro of proctors, workshop goals, Python environment verification, NUTS/NAPALM/containerlab prep.

2.1.2 09:15-09:30

Lab - Setup Environment and install NUTS

2.1.3 09:30-10:15

Theory - Introduction to NUTS and pytest: fundamentals, architecture, first test run and output walkthrough.

2.1.4 10:15-10:45

Lab - Basic Network Testing with NUTS: running standard tests, interpreting results, writing your first test.

2.1.5 10:45-11:15

Break - Coffee, a lot of coffee. ☕☕☕

2.1.6 11:15-11:30

Theory - Customising pytest reports for network testing scenarios.

2.1.7 11:30-12:30

Lab - Developing custom test cases for NUTS. Applying NUTS to real-world scenarios: config validation, operation state checks, and more.

2.1.8 12:30-12:45

Theory, Demo & Lab - Overview of the INPG Stack: Infrahub, NUTS, Prometheus, Grafana; walkthrough and their role in network testing.

Online lab: https://play.instruqt.com/opsmill/invite/xvvcexynkvxt

2.1.9 12:45-13:00

End - Questions, Demo, next steps, feedback!

3. Workshop Tools Overview

This page summarizes the essential tools and frameworks used in the workshop.

3.1 NUTS (Network Unit Testing System)

NUTS is a Python-based testing framework designed for network environments. It extends pytest to provide structured, reusable testing for network devices using real or simulated data.

- Documentation: https://nuts.readthedocs.io/en/latest/
- GitHub Repo: https://github.com/network-unit-testing-system/nuts

3.2 pytest

pytest is the de facto standard Python testing framework that is easy to extend and use for both unit and functional testing.

- Documentation: https://docs.pytest.org/

3.3 NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support)

A Python library that provides a unified API to configure and retrieve data from network devices.

- Documentation: https://napalm.readthedocs.io/
- Mock Driver Guide: https://napalm.readthedocs.io/en/latest/tutorials/mock_driver.html

3.4 Nornir

An automation framework that uses Python to manage inventory, run tasks and process network-related workflows.

- Documentation: https://nornir.readthedocs.io/

3.5 Containerlab

Containerlab enables container-based networking labs on a single machine or across a distributed environment. It simplifies lab topology setup for network testing.

- Documentation: https://containerlab.dev/

3.6 netlab

netlab is a tool to easily manage and run network labs. Starting with release 25.07, netlab includes an experimental integration with NUTS for network testing.

- Documentation: https://netlab.tools/
- NUTS integration: https://netlab.tools/extool/nuts/

3.7 uv

uv is a fast Python package installer and virtual environment manager.

- Documentation: https://docs.astral.sh/uv/

3.8 INPG Stack

A combined solution of: - **Infrahub**: Stores network inventory data. - **NUTS**: Network Unit Testing System - **Prometheus**: Collects and stores metrics. - **Grafana**: Visualizes and dashboards collected data.

This stack is designed to give context and observability to your network testing efforts.

- Prometheus: https://prometheus.io/
- Grafana: https://grafana.com/

4. Lab

For this lab, we use a prepared environment created with `netlab`. You can run the virtual lab with containerlab, but to conserve resources, everything is set up to run with the mock driver from NAPALM. Without starting the lab, only the test cases using NAPALM can be executed.

Use the following repository for your work. You can fork and clone it.

https://github.com/network-unit-testing-system/naf_workshop_nuts_lab

4.1 Setup

4.1.1 📝 Install

Follow the instructions in the README.md

Solution

```
$ uv venv --python 3.12
$ uv sync
Resolved 59 packages in 2ms
Audited 54 packages in 0.15ms
```

4.1.2 📝Initialize Project

Since version v3.5.0, NUTS includes a small helper script, `nuts-init`, to bootstrap the project structure. Set the test directory to `./tests`, the nornir config file to `./nr-config.yaml`, and add one Arista EOS host. The inventory is already provided in the `./mocked_inventory` directory. Specify the inventory as a path, but **do not let the script create the inventory**, as this would overwrite the existing files.

The helper script will create the `nr-config.yaml` file and a demo test case at `tests/test_lldp_neighbors_demo.yaml`.

Solution

```
$ uv run nuts-init
Test dir [./tests]:
Nornir config file [./nr_config.yaml]: ./nr-config.yaml
Inventory directory [./inventory]: ./mocked_inventory
Create simple inventory [y/N]: N
Add Cisco XE host [y/N]: N
Add Juniper Junos host [y/N]: N
Add Arista EOS host [y/N]: y
Add Cisco NX host [y/N]: N
Add Cisco XR host [y/N]: N
Use netmiko session logs [y/N]: N
```

nr-config.yaml

```yaml
inventory:
  plugin: SimpleInventory
  options:
    host_file: mocked_inventory/hosts.yaml
    group_file: mocked_inventory/groups.yaml
  transform_function:  # consider using "load_credentials" from nornir_utils
runner:
  plugin: threaded
  options:
    num_workers: 100
logging:
  enabled: false
```

Note

If you already have a nornir config file, you can use the pytest option `--nornir-config` (this option is automatically added when pytest discovers the nuts plugin)

```
pytest --nornir-config "config.yaml"
```

4.2 My first test

The bootstrap script has already created a demo test case. If you run pytest now, the test will fail with `No hosts found for filter ... in Nornir inventory.` This is because the bootstrap script does not know the naming of the hosts in your lab/inventory.

4.2.1 📝 No hosts found

Run the test using the pytest command. This test should fail because the hostname is not found in your inventory.

Solution

```
$ uv run pytest --disable-warnings -q

====================================== ERRORS ======================================
_______________ ERROR collecting tests/test_lldp_neighbors_demo.yaml ________________
.venv/lib/python3.11/site-packages/pluggy/_hooks.py:512: in __call__
    return self._hookexec(self.name, self._hookimpls.copy(), kwargs, firstresult)
.venv/lib/python3.11/site-packages/pluggy/_manager.py:120: in _hookexec
    return self._inner_hookexec(hook_name, methods, kwargs, firstresult)
.venv/lib/python3.11/site-packages/_pytest/python.py:271: in pytest_pycollect_makeitem
    return list(collector._genfunctions(name, obj))
.venv/lib/python3.11/site-packages/_pytest/python.py:498: in _genfunctions
    self.ihook.pytest_generate_tests.call_extra(methods, dict(metafunc=metafunc))
.venv/lib/python3.11/site-packages/pluggy/_hooks.py:573: in call_extra
    return self._hookexec(self.name, hookimpls, kwargs, firstresult)
.venv/lib/python3.11/site-packages/pluggy/_manager.py:120: in _hookexec
    return self._inner_hookexec(hook_name, methods, kwargs, firstresult)
.venv/lib/python3.11/site-packages/nuts/plugin.py:71: in pytest_generate_tests
    parametrize_args, parametrize_data = get_parametrize_data(
.venv/lib/python3.11/site-packages/nuts/yamlloader.py:202: in get_parametrize_data
    data["test_data"] = ctx.parametrize(data.get("test_data", []))
.venv/lib/python3.11/site-packages/nuts/context.py:191: in parametrize
    raise NutsSetupError(
E   nuts.helpers.errors.NutsSetupError: No hosts found for filter <Filter ({'name__any': ['arista-eos-demo-01']})> in Nornir inventory.
============================== short test summary info ==============================
ERROR tests/test_lldp_neighbors_demo.yaml::TestNapalmLldpNeighborsCount - nuts.helpers.errors.NutsSetupError: No hosts found for filter <Filter ({'name__a...
!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!!
3 warnings, 1 error in 0.65s
```

tests/test_lldp_neighbors_demo.yaml

```yaml
- test_class: TestNapalmLldpNeighborsCount
  test_data:
  - host: arista-eos-demo-01
    neighbor_count: 3  # Number of LLDP neighbors need to be updated
```

4.2.2 📝 Host r02

Change the hostname to `r02` in the test definition `tests/test_lldp_neighbors_demo.yaml` and run the test again. The test will still fail, but now the reason is "AssertionError: assert 3 == 5". The test checks if `r02` has 3 neighbors, but the device actually has 5 neighbors, so the test fails.

Solution

```
uv run pytest --disable-warnings -q
F                                                                                [100%]
===================================== FAILURES ======================================
______________ TestNapalmLldpNeighborsCount.test_neighbor_count[r02_] _______________

self = <nuts.base_tests.napalm_lldp_neighbors.TestNapalmLldpNeighborsCount object at 0x7f3dd2f5f910>
single_result = <nuts.helpers.result.NutsResult object at 0x7f3dd335f090>
neighbor_count = 3

    @pytest.mark.nuts("neighbor_count")
    def test_neighbor_count(self, single_result, neighbor_count):
>       assert neighbor_count == len(single_result.result)
E       AssertionError: assert 3 == 5
E        +  where 5 = len({'Ethernet1': {'parent_interface': 'Ethernet1', 'remote_chassis_id': '00:1C:73:F6:74:93', 'remote_host': 'r01', 'remot...e':
'Ethernet4', 'remote_chassis_id': '00:1C:73:B1:E5:CD', 'remote_host': 'r04', 'remote_port': 'Ethernet1', ...}, ...})
E        +    where {'Ethernet1': {'parent_interface': 'Ethernet1', 'remote_chassis_id': '00:1C:73:F6:74:93', 'remote_host': 'r01', 'remot...e': 'Ethernet4',
'remote_chassis_id': '00:1C:73:B1:E5:CD', 'remote_host': 'r04', 'remote_port': 'Ethernet1', ...}, ...} = <nuts.helpers.result.NutsResult object at
0x7f3dd335f090>.result

.venv/lib/python3.11/site-packages/nuts/base_tests/napalm_lldp_neighbors.py:74: AssertionError
============================== short test summary info ==============================
FAILED tests/test_lldp_neighbors_demo.yaml::TestNapalmLldpNeighborsCount::test_neighbor_count[r02_] - AssertionError: assert 3 == 5
1 failed, 3 warnings in 0.41s
```

tests/test_lldp_neighbors_demo.yaml

```yaml
- test_class: TestNapalmLldpNeighborsCount
  test_data:
  - host: r02
    neighbor_count: 3  # Number of LLDP neighbors need to be updated
```

4.2.3 📝 Host r02, 5 LLDP neighbors

Update the expected neighbor count to 5 and run the test again. This time, the test should pass.

Solution

```
$ uv run pytest --disable-warnings -v
================================ test session starts ================================
platform linux -- Python 3.11.12, pytest-7.4.4, pluggy-1.6.0 -- /workspaces/naf_workshop_nuts_lab/.venv/bin/python
cachedir: .pytest_cache
rootdir: /workspaces/naf_workshop_nuts_lab
plugins: nuts-3.5.0
collected 1 item

tests/test_lldp_neighbors_demo.yaml::TestNapalmLldpNeighborsCount::test_neighbor_count[r02_] PASSED [100%]

=========================== 1 passed, 3 warnings in 0.33s ===========================
```

tests/test_lldp_neighbors_demo.yaml

```yaml
- test_class: TestNapalmLldpNeighborsCount
  test_data:
    - host: r02
      neighbor_count: 5  # Number of LLDP neighbors need to be updated
```

4.3 Basic Network Tests

Once your first test is running successfully, it's time to add additional foundational network validation tests using the NUTS framework. These tests help ensure that devices are properly interconnected and that essential protocols are functioning as expected.

4.3.1 📝 LLDP Neighbor Verification

Verify that each router has the expected LLDP neighbors. This confirms that the physical topology matches the intended design.

Use the `TestNapalmLldpNeighbors` test bundle and create the file `tests/test_lldp_neighbors.yaml` with all the test definitions.

Solution

tests/test_lldp_neighbors.yaml

```yaml
- test_class: TestNapalmLldpNeighbors
  test_data:
    # r01
    - host: r01
      local_port: Ethernet1
      remote_host: r02
      remote_port: Ethernet1
    - host: r01
      local_port: Ethernet2
      remote_host: r03
      remote_port: Ethernet1

    # r02
    - host: r02
      local_port: Ethernet1
      remote_host: r01
      remote_port: Ethernet1
    - host: r02
      local_port: Ethernet2
      remote_host: r03
      remote_port: Ethernet2
    - host: r02
      local_port: Ethernet3
      remote_host: r03
      remote_port: Ethernet3
    - host: r02
      local_port: Ethernet4
      remote_host: r04
      remote_port: Ethernet1
    - host: r02
      local_port: Ethernet5
      remote_host: r04
      remote_port: Ethernet2

    # r03
    - host: r03
      local_port: Ethernet1
      remote_host: r01
      remote_port: Ethernet2
    - host: r03
      local_port: Ethernet2
      remote_host: r02
      remote_port: Ethernet2
    - host: r03
      local_port: Ethernet3
      remote_host: r02
      remote_port: Ethernet3
    - host: r03
      local_port: Ethernet4
      remote_host: r05
      remote_port: Ethernet1
    - host: r03
      local_port: Ethernet5
      remote_host: r05
      remote_port: Ethernet2

    # r04
    - host: r04
      local_port: Ethernet1
      remote_host: r02
      remote_port: Ethernet4
    - host: r04
      local_port: Ethernet2
      remote_host: r02
      remote_port: Ethernet5
    - host: r04
      local_port: Ethernet3
      remote_host: r05
      remote_port: Ethernet3
    - host: r04
      local_port: Ethernet4
      remote_host: r05
      remote_port: Ethernet4
    - host: r04
      local_port: Ethernet5
      remote_host: r06
      remote_port: Ethernet1

    # r05
    - host: r05
      local_port: Ethernet1
      remote_host: r03
      remote_port: Ethernet4
    - host: r05
      local_port: Ethernet2
      remote_host: r03
      remote_port: Ethernet5
    - host: r05
      local_port: Ethernet3
      remote_host: r04
      remote_port: Ethernet3
    - host: r05
      local_port: Ethernet4
    remote_host: r04
    remote_port: Ethernet4
  - host: r05
    local_port: Ethernet5
    remote_host: r07
    remote_port: Ethernet1

  # r06
  - host: r06
    local_port: Ethernet1
    remote_host: r04
    remote_port: Ethernet5
  - host: r06
    local_port: Ethernet2
    remote_host: r08
    remote_port: Ethernet1
  - host: r06
    local_port: Ethernet3
    remote_host: r09
    remote_port: Ethernet1

  # r07
  - host: r07
    local_port: Ethernet1
    remote_host: r05
    remote_port: Ethernet5
  - host: r07
    local_port: Ethernet2
    remote_host: r08
    remote_port: Ethernet2
  - host: r07
    local_port: Ethernet3
    remote_host: r09
    remote_port: Ethernet2

  # r08
  - host: r08
    local_port: Ethernet1
    remote_host: r06
    remote_port: Ethernet2
  - host: r08
    local_port: Ethernet2
    remote_host: r07
    remote_port: Ethernet2
  - host: r08
    local_port: Ethernet3
    remote_host: r09
    remote_port: Ethernet3
  - host: r08
    local_port: Ethernet4
    remote_host: r10
    remote_port: Ethernet1

  # r09
  - host: r09
    local_port: Ethernet1
    remote_host: r06
    remote_port: Ethernet3
  - host: r09
    local_port: Ethernet2
    remote_host: r07
    remote_port: Ethernet3
  - host: r09
    local_port: Ethernet3
    remote_host: r08
    remote_port: Ethernet3
  - host: r09
    local_port: Ethernet4
    remote_host: r10
    remote_port: Ethernet2

  # r10
  - host: r10
    local_port: Ethernet1
    remote_host: r08
    remote_port: Ethernet4
  - host: r10
    local_port: Ethernet2
    remote_host: r09
    remote_port: Ethernet4
```

4.3.2 📝 Uplink Interface Status

Ensure that the uplink interfaces are operational. This test checks that the interfaces are both administratively enabled and operationally up.

Use the `TestNapalmInterfaces` test bundle and create the file `tests/test_interfaces.yaml` with all the test definitions.

Solution

tests/test_interfaces.yaml

```yaml
- test_class: TestNapalmInterfaces
  test_data:
    # r01
    - &interface
      host: r01
      name: Ethernet1
      is_enabled: true
      is_up: true
      mtu: 1500
      speed: 1000
    - <<: *interface
      host: r01
      name: Ethernet2

    # r02
    - <<: *interface
      host: r02
      name: Ethernet1
    - <<: *interface
      host: r02
      name: Ethernet2
    - <<: *interface
      host: r02
      name: Ethernet3
    - <<: *interface
      host: r02
      name: Ethernet4
    - <<: *interface
      host: r02
      name: Ethernet5

    # r03
    - <<: *interface
      host: r03
      name: Ethernet1
    - <<: *interface
      host: r03
      name: Ethernet2
    - <<: *interface
      host: r03
      name: Ethernet3
    - <<: *interface
      host: r03
      name: Ethernet4
    - <<: *interface
      host: r03
      name: Ethernet5

    # r04
    - <<: *interface
      host: r04
      name: Ethernet1
    - <<: *interface
      host: r04
      name: Ethernet2
    - <<: *interface
      host: r04
      name: Ethernet3
    - <<: *interface
      host: r04
      name: Ethernet4
    - <<: *interface
      host: r04
      name: Ethernet5

    # r05
    - <<: *interface
      host: r05
      name: Ethernet1
    - <<: *interface
      host: r05
      name: Ethernet2
    - <<: *interface
      host: r05
      name: Ethernet3
    - <<: *interface
      host: r05
      name: Ethernet4
    - <<: *interface
      host: r05
      name: Ethernet5

    # r06
    - <<: *interface
      host: r06
      name: Ethernet1
    - <<: *interface
      host: r06
      name: Ethernet2
    - <<: *interface
      host: r06
      name: Ethernet3
    # r07
    - <<: *interface
      host: r07
      name: Ethernet1
    - <<: *interface
      host: r07
      name: Ethernet2
    - <<: *interface
      host: r07
      name: Ethernet3

    # r08
    - <<: *interface
      host: r08
      name: Ethernet1
    - <<: *interface
      host: r08
      name: Ethernet2
    - <<: *interface
      host: r08
      name: Ethernet3
    - <<: *interface
      host: r08
      name: Ethernet4

    # r09
    - <<: *interface
      host: r09
      name: Ethernet1
    - <<: *interface
      host: r09
      name: Ethernet2
    - <<: *interface
      host: r09
      name: Ethernet3
    - <<: *interface
      host: r09
      name: Ethernet4

    # r10
    - <<: *interface
      host: r10
      name: Ethernet1
    - <<: *interface
      host: r10
      name: Ethernet2
```

4.4 Pytest Reports

One of the advantages of building tests on top of Pytest is its powerful and extensible ecosystem. There are dozens of plugins that integrate with CI/CD pipelines, generate coverage data, or produce test reports in various formats—from terminal output to rich HTML dashboards.

When automating network tests, reporting is important. It's often the first thing your team, manager, or CI/CD pipeline will check to determine if everything is working correctly.

NUTS integrates seamlessly with Pytest's reporting features. For example:

- Use `pytest --junitxml=report.xml` to generate a JUnit-compatible XML report.
- Integrate with Allure, HTML, or even Slack notifications via plugins.

4.4.1 Customizing Reports

What if you want to include network-specific metadata in your reports? For example:

- Which host was tested,
- Which interface or protocol was involved,
- Which requirement number the test relates to.

In standard Pytest, you would typically use the record_property fixture inside your Python test functions to attach metadata to test results:

```python
def test_example(record_property):
    record_property("device", "router1")
    record_property("interface", "eth0")
```

However, NUTS test logic is driven by YAML, not Python functions. This means you don’t have access to `record_property` in the usual way. Instead, NUTS provides a dedicated Pytest hook:

```
def pytest_nuts_single_result(request: FixtureRequest, nuts_ctx: NutsContext, result: NutsResult) -> None:
    ...
```

You can define this hook in your `conftest.py`:

```
@pytest.hookimpl(tryfirst=True)
def pytest_nuts_single_result(request, nuts_ctx, result):
    test_extras = nuts_ctx.nuts_parameters.get("test_extras", {})
    if not isinstance(test_extras, dict):
        return
    for p_name, p_value in test_extras.get("properties", {}).items():
        request.node.user_properties.append((p_name, p_value))
```

If `test_extras` exists in the NUTS context, every key-value pair under the `properties` attribute will be appended to the Pytest node object and subsequently included in the JUnit report.

4.4.2 📝 Set the property "mocked"

Call the `pytest_nuts_single_result` hook to set, for all Nornir tests (when the context is of type `NornirContext`), the property "mocked" to the string "True" or "False" depending on the `eos` group platform.

Solution

tests/conftest.py

```
from pytest import  FixtureRequest
from nuts.context import NutsContext, NornirNutsContext
from nuts.helpers.result import NutsResult

def pytest_nuts_single_result(request: FixtureRequest, nuts_ctx: NutsContext, result: NutsResult) -> None:
    if isinstance(nuts_ctx, NornirNutsContext):
        # This is a Nornir context
        if eos := nuts_ctx.nornir.inventory.groups.get("eos"):
            if eos.platform == "mock":
                request.node.user_properties.append(("mocked", "True"))
                return
        request.node.user_properties.append(("mocked", "False"))
```

4.5 Custom Test Cases

In this section, we explore how to extend the NUTS framework by writing a custom test class.

In our lab topology, routers `r02`, `r03`, `r04`, and `r05` participate in an OSPF area, while routers `r06`, `r07`, `r08`, and `r09` are part of an IS-IS routing domain. Unfortunately, Napalm does not yet support OSPF or IS-IS neighbor information. Normally, in true open-source fashion, we would:

1. Implement IS-IS support in Napalm for at least three major platforms,
2. Write unit and integration tests,
3. Open a PR and participate in the review process,
4. Wait for a new Napalm release.

But... we have a workshop to run and a lunch break coming up. So instead, we'll write a custom NUTS test class that uses the NAPALM CLI function to retrieve the IS-IS neighbor output and asserts that the expected neighbors are present.

On GitHub, you can find an example repository, network-unit-testing-system/example-custom-netmiko-cli-test, demonstrating how to implement a custom test class. You can also refer to the documentation How To Write Your Own Test for further guidance.

We will take advantage of the fact that Arista EOS can provide the show command output in JSON format.

r06: `show isis neighbors | json`

```
r06#show isis neighbors | json
{
    "vrfs": {
        "default": {
            "isisInstances": {
                "Gandalf": {
                    "neighbors": {
                        "0000.0000.0008": {
                            "adjacencies": [
                                {
                                    "state": "up",
                                    "circuitId": "73",
                                    "routerIdV4": "0.0.0.0",
                                    "interfaceName": "Ethernet2",
                                    "lastHelloTime": 1748181597,
                                    "level": "level-2",
                                    "snpa": "P2P",
                                    "hostname": "r08",
                                    "details": {
                                        "stateChanged": 1748181281,
                                        "grSupported": "Supported",
                                        "interfaceAddressFamily": "ipv4",
                                        "srEnabled": false,
                                        "advertisedHoldTime": 30,
                                        "ip4Address": "10.1.0.50",
                                        "neighborAddressFamily": "ipv4",
                                        "areaIds": [
                                            "49.0001"
                                        ],
                                        "bfdIpv4State": "adminDown",
                                        "bfdIpv6State": "adminDown",
                                        "grState": ""
                                    }
                                }
                            ]
                        },
                        "0000.0000.0009": {
                            "adjacencies": [
                                {
                                    "state": "up",
                                    "circuitId": "6F",
                                    "routerIdV4": "0.0.0.0",
                                    "interfaceName": "Ethernet3",
                                    "lastHelloTime": 1748181597,
                                    "level": "level-2",
                                    "snpa": "P2P",
                                    "hostname": "r09",
                                    "details": {
                                        "stateChanged": 1748181281,
                                        "grSupported": "Supported",
                                        "interfaceAddressFamily": "ipv4",
                                        "srEnabled": false,
                                        "advertisedHoldTime": 30,
                                        "ip4Address": "10.1.0.54",
                                        "neighborAddressFamily": "ipv4",
                                        "areaIds": [
                                            "49.0001"
                                        ],
                                        "bfdIpv4State": "adminDown",
                                        "bfdIpv6State": "adminDown",
                                        "grState": ""
                                    }
                                }
                            ]
                        }
                    }
                }
            }
        }
    }
}
```

Note: If you do end up implementing this in Napalm after the workshop, the Internet will thank you—and so will we.

Warning

For Pytest to discover your custom module, it is important that the project's root folder is included in the Python path. You can ensure this by running Pytest as a Python module: `uv run python -m pytest`

4.5.1 📝 OSPF Neighbor Count

Implement a custom test case to support the following YAML test definition:

tests/test_ospf_neighbors.yaml

```yaml
- test_module: my_module.ospf
  test_class: TestOspfNeighborsCount
  test_data:
    - host: r02
      neighbor_count: 4
```

Solution

my_module/ospf.py

```python
from typing import Dict, Callable, List

import json
import pytest
from nornir.core.task import MultiResult, Result
from nornir_napalm.plugins.tasks import napalm_cli

from nuts.context import NornirNutsContext
from nuts.helpers.result import AbstractHostResultExtractor, NutsResult


class OspfExtractor(AbstractHostResultExtractor):
    def single_transform(self, single_result: MultiResult) -> int:
        count = 0

        result = self._simple_extract(single_result)[
            "show ip ospf neighbor summary | json"
        ]
        json_data = json.loads(result)
        ospf_inst = json_data.get("vrfs", {}).get("default", {}).get("instList", {})
        for ospf in ospf_inst.values():
            count += ospf.get("neighborSummary", {}).get("numFull", 0)
        return count


class OspfContext(NornirNutsContext):
    def nuts_task(self) -> Callable[..., Result]:
        return napalm_cli

    def nuts_arguments(self) -> Dict[str, List[str]]:
        return {"commands": ["show ip ospf neighbor summary | json"]}

    def nuts_extractor(self) -> OspfExtractor:
        return OspfExtractor(self)


CONTEXT = OspfContext


class TestOspfNeighborsCount:
    @pytest.mark.nuts("neighbor_count")
    def test_neighbor_count(self, single_result: NutsResult, neighbor_count: int) -> None:
        assert neighbor_count == single_result.result, (
            f"Expected neighbor count {neighbor_count} does not match the result {single_result.result}"
        )
```

4.5.2 📝 ISIS Neighbor Count

Implement a custom test case to support the following YAML test definition:

tests/test_isis_neighbors.yaml

```yaml
- test_module: my_module.isis
  test_class: TestIsisNeighborsCount
  test_data:
    - host: r06
      neighbor_count: 2
```

Solution

my_module/isis.py

```python
from typing import Dict, Callable, List

import json
import pytest
from nornir.core.task import MultiResult, Result
from nornir_napalm.plugins.tasks import napalm_cli

from nuts.context import NornirNutsContext
from nuts.helpers.result import AbstractHostResultExtractor, NutsResult


class IsisExtractor(AbstractHostResultExtractor):
    def single_transform(self, single_result: MultiResult) -> int:
        count = 0

        result = self._simple_extract(single_result)["show isis neighbors | json"]
        json_data = json.loads(result)

        instances = (
            json_data.get("vrfs", {}).get("default", {}).get("isisInstances", {})
        )
        for instance in instances.values():
            neighbors = instance.get("neighbors", {})
            for adjacencies in neighbors.values():
                for adjacency in adjacencies.get("adjacencies", []):
                    if adjacency.get("state") == "up":
                        count += 1
        return count


class IsisContext(NornirNutsContext):
    def nuts_task(self) -> Callable[..., Result]:
        return napalm_cli

    def nuts_arguments(self) -> Dict[str, List[str]]:
        return {"commands": ["show isis neighbors | json"]}

    def nuts_extractor(self) -> IsisExtractor:
        return IsisExtractor(self)


CONTEXT = IsisContext


class TestIsisNeighborsCount:
    @pytest.mark.nuts("neighbor_count")
    def test_neighbor_count(self, single_result: NutsResult, neighbor_count: int) -> None:
        assert neighbor_count == single_result.result, (
            f"Expected neighbor count {neighbor_count} does not match the result {single_result.result}"
        )
```