NAF Workshop WS:C4 - AutoCon2

Leveraging Python Ecosystem for Quick CLI Tool Creation

Urs Baumann

None

Table of contents

1. Workshop FAQ: Leveraging Python Ecosystem for Quick CLI Tool Creation

Proctor: Urs Baumann

Warning

At the moment NAPALM does not support Python 3.13 (PR)

Getting the path to a older Python version

```
whereis python3.12
```

Use the older version with Poetry

```
poetry env use /usr/local/python/3.12.1/bin/python3
```

1.1 General Information

1.1.1 1. What is this workshop about?

This workshop focuses on leveraging Python libraries like Nornir, Pydantic, Rich, and Typer to create powerful command-line interface (CLI) tools. Attendees will walk through hands-on exercises and real-world examples to build practical and user-friendly tools.

1.1.2 2. What experience level is required?

This workshop is suitable for all levels, from beginners to experts. Familiarity with Python basics is advised. The material will provide extra tasks for faster attendees.

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 or MacOS.
- **Libraries**: Poetry, Nornir, Pydantic, Rich, Typer, and others will be installed as part of the setup.
- **IDE**: You can use the IDE you desire. The proctor is 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:

- Poetry
- Nornir
- Pydantic
- Rich
- Typer

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 proctor cannot focus on each idea in detail.

1.3.2 8. Lab access?

You can run your own lab on your computer or connect to your lab. The internet is shared to all attendees and therefore graphical transmissions are not recommended.

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?

The workshop is structured to accommodate various skill levels. If you fall behind, the proctor will be available to help, and you can use the break to catch up. The lab containes checkpoints with provided solutions.

2. Agenda

2.1 Agenda

2.1.1 09:00-09:15:

Theory - Introduction: Why do we use Poetry and not pip? Other alternatives?

2.1.2 09:15-09:30:

Lab - Setup Environment and install poetry.

Optional: What is pipx and why is it handy?

2.1.3 09:30-10:00:

Theory - Introduction to Nornir, Inventory, Tasks

Demo - Live Demo of some code on how to deploy L3VPN Services in a network.

2.1.4 10:00-10:30:

Lab - Connect to Network Device to get some info and look up more info on an API.

Optional: How does the NAPALM mock device driver work and can you use it in pytest?

2.1.5 10:30-10:40:

Demo - Live Demo of how rich can provide nice-looking output on the shell.

2.1.6 10:40-11:00

Break - Coffee, a lot of coffee.

2.1.7 11:00-11:10:

Lab – Using Rich

Optional: Can you save the rich output as SVG?

2.1.8 11:10-11:30:

Theory - Introduction to Pydantic, benefits of Models

Demo - Demo Pydentify, JSONSchema, Models

2.1.9 11:30-12:00:

Lab - Build Pydantic model with multiple layers, load, and dump data. Validate

Optional: Transform data, custom validation, JSONSchema, load object from YAML

2.1.10 12:00-12:15:

Theory - Typer, (and FastAPI as a contrast), Demo how Nettowel works.

2.1.11 12:15-12:30:

Lab - Combine Nornir, Rich, Pydantic and Typer to a nice CLI tool.

Optional: Use Pytest to test the CLI tool, build package with poetry, publish on pypi registry

2.1.12 12:45-13:00:

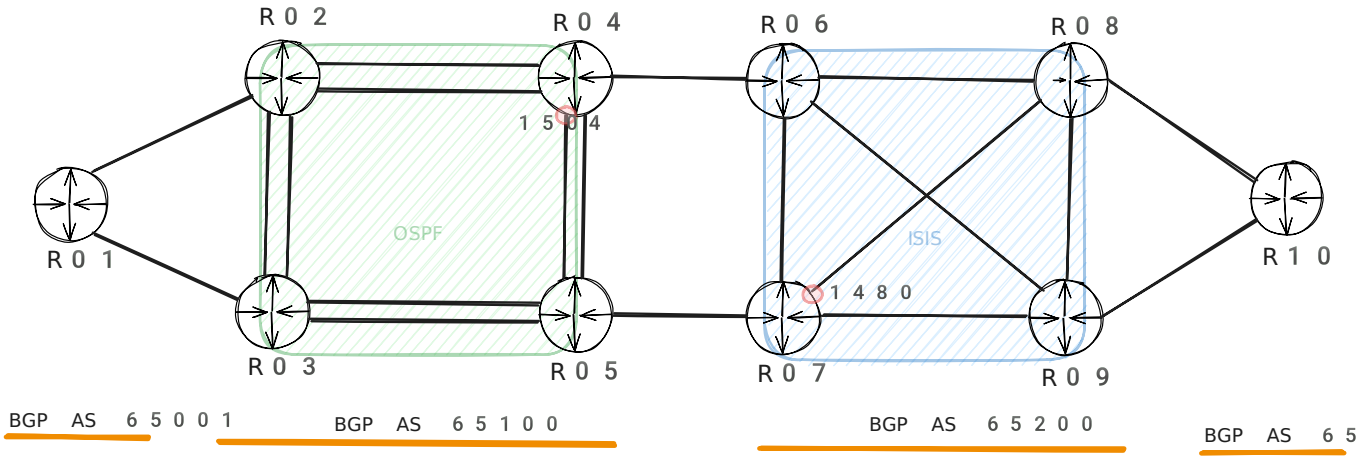
End - Questions, Demo, next steps, feedback!

3. Use Case

The workshop is designed around a use case where you are tasked with building a small CLI tool to interact with the network and extract the MTU (Maximum Transmission Unit) of various interfaces. The goal is to create a user-friendly tool that your coworkers can easily install and use.

Please note that this use case is intended for educational purposes only. There are other tools and solutions available that can accomplish the same task.

3.1 Topology



3.2 Commands

3.2.1 Interfaces

Display the names of all network interfaces along with their MTU sizes. If no hostname is specified, the output will include all network devices in the inventory. If a minimum MTU value is provided, any interfaces with an MTU size below that threshold will be highlighted in red in the output.

3.2.2 Neighbors

For a specific network device, the LLDP neighbors are gathered, and for each connection, the interface names and their MTU sizes are displayed. If the MTU size does not match, it will be highlighted in red. Additionally, if a minimum MTU value is specified, any interfaces with an MTU size below that threshold will also be highlighted in red in the output.

3.2.3 Path

For a given destination (specified as an IPv4 address in CIDR notation) and a starting network device, the path is discovered recursively. For each hop, the MTU is compared, and any mismatch is highlighted in red. If a minimum MTU value is specified, any MTU lower than this threshold will also be marked red in the output.

Although for a small number of devices it would be easier to collect the data in parallel from all hosts, the goal of this exercise is to demonstrate how complex workflows can also be implemented.

Keep in mind that there is a high chance you may not complete this command within the workshop time.

3.3 Mocked Data

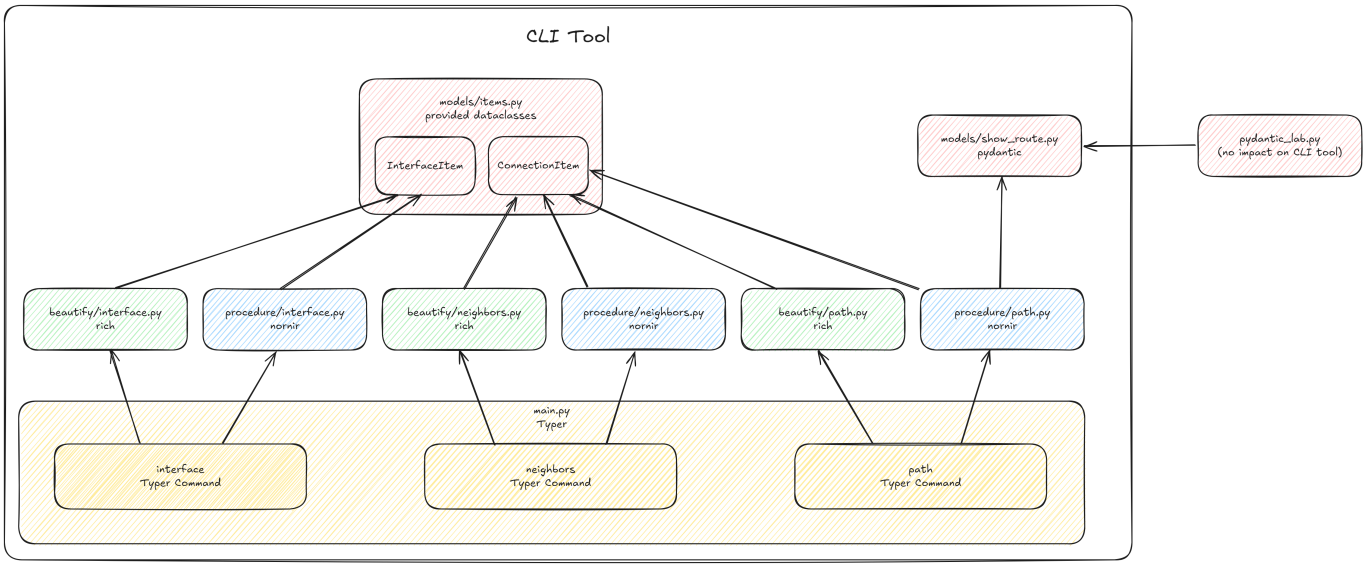
o avoid the need for setting up a lab with 10 routers, mocked data for the NAPALM Mock Driver is provided. The lab itself was created using Netlab and is also available if you'd like to work with a real lab setup.

The mocked data was collected with a simple CLI tool built using Nornir, NAPALM, and Typer. 😉

3.4 Code

You can find the code here: https://github.com/ubaumann/mtu_tool

The following graphic illustrates how the various exercises will ultimately come together to form a CLI tool.



4. Poetry

Poetry is a popular Python package manager widely used in the Python community. However, in recent times, `uv` is gaining increasing attention. Poetry offers several advantages over traditional methods like `pip`, `requirements.txt`, and `setup.py`. With Poetry, project dependencies are clearly defined in the `pyproject.toml` file, and the exact versions of installed packages are locked in the `poetry.lock` file, ensuring consistent environments across different setups.

4.1 Links:

- Poetry
- uv (alternative)

5. LAB Poetry

To get hands-on experience, we'll create a new Python project. However, for the remainder of the lab, we'll be using an existing Poetry setup.

If you haven't installed Poetry yet, please follow the installation instructions.

5.1 Setup a new project and add dependencies and development dependencies.

- Create a new project with Poetry. Our example use case is a small MTU tool.
- Add dependencies to the project
- rich
- nornir
- nornir-napalm
- typer
- Add development dependencies
- pytest
- black
- Install the created project
- Open a shell with loaded virtual environment
- Display outdated packages

Solution

```
poetry new mtu-tool
poetry add rich nornir nornir-napalm typer
poetry add --group=dev pytest black
poetry install
poetry shell
poetry show --outdated
```

5.2 Links:

- Poetry basic usage

5.3 Additional Task

Use the test Python repository on PyPI to publish your project using Poetry.

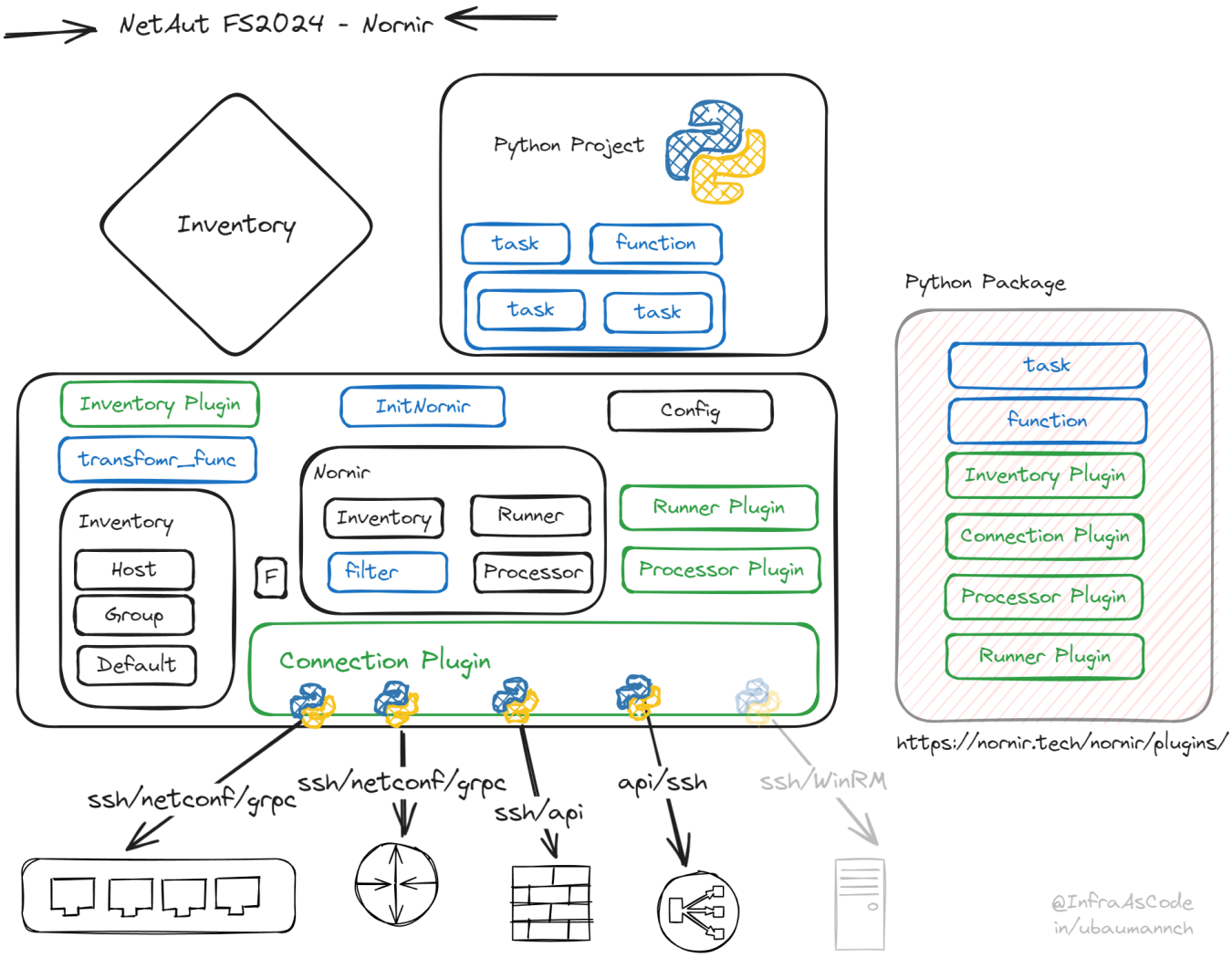
5.4 Next Steps

Explore the pipx chapter or review the differences between `uv` and Poetry.

6. Nornir

Pluggable multi-threaded framework with inventory management to help operate collections of devices

Nornir is a powerful framework, and since version 3, it provides only core functionality. All additional features are now available through separate plugins.



6.1 Links:

- Nornir Docs
- Nornir Plugis

7. LAB Network API

Using Nornir and the plugins to interact with the device APIs. If you'd like to work with your own lab setup, feel free to do so. This lab uses a mocked NAPALM driver to minimize resource usage and ensure it runs on any notebook.

7.1 Prerequisites

The prepared use case utilizes the NAPALM Mock driver to simulate interactions with network devices. For more details and the full use case, refer to the Use Case chapter.

To get started, clone the repository (https://github.com/ubaumann/mtu_tool).

The following elements are already set up:

- Mocked data for the NAPALM driver
- A simple inventory
- Project setup with Poetry

7.2 Procedures

All three tasks require a Nornir object. To facilitate this, the `helper.py` file includes a helper function called `init_nornir`.

```python
from typing import Optional

from nornir import InitNornir
from nornir.core import Nornir


def init_nornir(configuration_file: str = "config.yaml", password: Optional[str] = None) -> Nornir:
    """
    Helper function to init the nornir object.
    We could do stuff like setting the default password here
    """
    nr = InitNornir(config_file=configuration_file)
    if password:
        nr.inventory.defaults.password = password
    return nr
```

7.2.1 Interfaces `mtu_tool/procedure/interfaces.py`

Implement the `interfaces` function, which takes a Nornir object and an optional `hostname`. If a `hostname` is provided, the Nornir inventory should be filtered to include only the specified host.

The function must use the `napalm_get` task from the nornir-napalm plugin to retrieve interface information using the `get_interfaces` getter. The `InterfaceItem` dataclass from `mtu_tool.models.itms` defines the expected structure for the interface data. The function should return a dictionary where the keys are hostnames and the values are lists of `InterfaceItem` objects. Additionally, the function should return the `AggregatedResult` from the Nornir task run.

```python
from typing import Dict, List, Optional, Tuple

from nornir.core import Nornir
from nornir.core.task import AggregatedResult
from nornir_napalm.plugins.tasks import napalm_get

from mtu_tool.models.itms import InterfaceItem


def interfaces(
    nr: Nornir,
    hostname: Optional[str] = None,
) -> Tuple[Dict["str", List[InterfaceItem]], AggregatedResult]:
    """Collect the mtu for all interfaces"""

    # TODO

    return data, result
```

```python
if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    data, result = interfaces(nr)
    print_result(result)
    for host, int_data in data.items():
        for d in int_data:
            print(f"{host}: {d.name:<15} {d.mtu}")

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/interfaces.py
    r01: Ethernet2       1500
    r01: Ethernet1       1500
    r01: Management0     1500
    r01: Loopback0       65535
    r02: Ethernet2       1500
    r02: Ethernet4       1500
    r02: Ethernet1       1500
    r02: Ethernet3       1500
    r02: Ethernet5       1500
    r02: Management0     1500
    r02: Loopback0       65535
    ...
    """
```

One possible solution

```python
from typing import Dict, List, Optional, Tuple

from nornir.core import Nornir
from nornir.core.task import AggregatedResult
from nornir_napalm.plugins.tasks import napalm_get

from mtu_tool.exceptions import NoHostFoundException
from mtu_tool.models.itms import InterfaceItem


def interfaces(
    nr: Nornir,
    hostname: Optional[str] = None,
) -> Tuple[Dict["str", List[InterfaceItem]], AggregatedResult]:
    """Collect the mtu for all interfaces"""

    if hostname:
        nr = nr.filter(name=hostname)
        if len(nr.inventory) == 0:
            raise NoHostFoundException(f"Host {hostname} not found in inventory")

    result = nr.run(
        task=napalm_get,
        getters=[
            "get_interfaces",
        ],
    )

    data = {}
    for host, mulit_result in result.items():
        data_interface = []
        interfaces = mulit_result[0].result.get("get_interfaces", {})
        for int_name, int_data in interfaces.items():
            data_interface.append(InterfaceItem(name=int_name, mtu=int_data.get("mtu")))
        data[host] = data_interface

    return data, result


if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    data, result = interfaces(nr)
    print_result(result)
    for host, int_data in data.items():
        for d in int_data:
            print(f"{host}: {d.name:<15} {d.mtu}")

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/interfaces.py
    r01: Ethernet2       1500
    r01: Ethernet1       1500
    r01: Management0     1500
    r01: Loopback0       65535
    r02: Ethernet2       1500
    r02: Ethernet4       1500
    r02: Ethernet1       1500
    r02: Ethernet3       1500
    r02: Ethernet5       1500
    r02: Management0     1500
    r02: Loopback0       65535
    ...
    """
```

7.2.2 Neighbors `mtu_tool/procedure/neighbors.py`

Implement the `neighbors` function. The function takes similar arguments as before, but now the `hostname` is required. The objective is to retrieve all LLDP neighbors from the specified device and return connection details for each link between the device and its neighbors. To achieve this, the function will need to execute `nornir.run` twice.

```python
from typing import List, Tuple

from nornir.core import Nornir
from nornir.core.task import AggregatedResult
from nornir.core.filter import F
from nornir_napalm.plugins.tasks import napalm_get

from mtu_tool.models.itms import ConnectionItem


def neighbors(
    nr: Nornir,
    hostname: str,
) -> Tuple[List[ConnectionItem], AggregatedResult]:
```

```
    """Collect the mtu for all local and neighbor interfaces"""

    # TODO

    return connections, result_interfaces


if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    connections, result = neighbors(nr, hostname="r06")
    print_result(result)
    for c in connections:
        print(
            f"{c.local_interface} {c.local_mtu} - {c.neighbor_name} {c.neighbor_interface} {c.neighbor_mtu}"
        )

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/neighbors.py
    Ethernet1 1500 - r04 Ethernet5 1500
    Ethernet2 1500 - r08 Ethernet1 1500
    Ethernet3 1500 - r09 Ethernet1 1500
    """
```

- Use the `napalm_get` task with the `get_lldp_neighbors` getter to retrieve all LLDP neighbors.
- Compile a list of all neighbor names, including the provided `hostname`.
- Use Nornir's `F` function to filter the inventory, selecting only the hosts whose names match the neighbors or the `hostname` (`F(name__any=...)`).
- Collect `get_interfaces` data for all the filtered hosts.
- For each LLDP neighbor, create a ConnectionItem using the relevant data from the `get_interfaces` results, and add it to a list.

One possible solution

```python
from typing import List, Tuple

from nornir.core import Nornir
from nornir.core.task import AggregatedResult
from nornir.core.filter import F
from nornir_napalm.plugins.tasks import napalm_get

from mtu_tool.exceptions import NoHostFoundException
from mtu_tool.models.itms import ConnectionItem


def neighbors(
    nr: Nornir,
    hostname: str,
) -> Tuple[List[ConnectionItem], AggregatedResult]:
    """Collect the mtu for all local and neighbor interfaces"""

    nr_host = nr.filter(name=hostname)
    if len(nr_host.inventory) == 0:
        raise NoHostFoundException(f"Host {hostname} not found in inventory")

    result_lldp = nr_host.run(
        task=napalm_get,
        getters=[
            "get_lldp_neighbors",
        ],
    )
    hosts = [hostname]
    get_lldp_neighbors = result_lldp[hostname][0].result.get("get_lldp_neighbors", {})
    for lldp_data in get_lldp_neighbors.values():
        for lldp_neighbor in lldp_data:
            n = lldp_neighbor["hostname"]
            if n not in hosts:
                hosts.append(n)

    result_interfaces = nr.filter(F(name__any=hosts)).run(
        task=napalm_get,
        getters=[
            "get_interfaces",
        ],
    )

    get_interfaces_local = result_interfaces[hostname][0].result.get(
        "get_interfaces", {}
    )

    connections = []
    for local_interface, lldp_data in get_lldp_neighbors.items():
        local_mtu = get_interfaces_local[local_interface]["mtu"]
        for lldp_neighbor in lldp_data:
            neighbor_name = lldp_neighbor["hostname"]
            neighbor_interface = lldp_neighbor["port"]
            neighbor_mtu = result_interfaces[neighbor_name][0].result.get(
                "get_interfaces", {}
            )[neighbor_interface]["mtu"]

            connections.append(
                ConnectionItem(
                    local_name=hostname,
                    local_interface=local_interface,
                    local_mtu=local_mtu,
                    neighbor_name=neighbor_name,
                    neighbor_interface=neighbor_interface,
                    neighbor_mtu=neighbor_mtu,
                )
            )

    return connections, result_interfaces


if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    connections, result = neighbors(nr, hostname="r06")
    print_result(result)
    for c in connections:
        print(
            f"{c.local_interface} {c.local_mtu} - {c.neighbor_name} {c.neighbor_interface} {c.neighbor_mtu}"
        )

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/neighbors.py
    Ethernet1 1500 - r04 Ethernet5 1500
    Ethernet2 1500 - r08 Ethernet1 1500
    Ethernet3 1500 - r09 Ethernet1 1500
    """
```

7.2.3 Path `mtu_tool/procedure/path.py`

Congratulations on reaching this point! The following procedure is definitely challenging and can be time-consuming. For educational purposes, the goal is to recursively find all possible paths from a given network device to a defined destination.

The mocked data contains information to help trace the path to all loopback IP addresses, where the loopback addresses correspond to the router's name number. For example, device `r01` has the IP address `10.0.0.1`, and `r10` has `10.0.0.10`. To determine the next hop for `10.0.0.10`, use the command `show ip route 10.0.0.10/32 | json`.

Note that multiple paths may exist, so the returned paths should be a list of lists, each containing `ConnectionItem` objects, along with the corresponding `AggregatedResult`. For better clarity, the `AggregatedResult` should not only include the results of the final Nornir run, but also capture the results of previous runs in the process.

```
import logging
from typing import Dict, Tuple, List
from ipaddress import IPv4Interface

from nornir.core import Nornir
from nornir.core.task import Task, Result, AggregatedResult
from nornir_napalm.plugins.tasks import napalm_get, napalm_cli

from mtu_tool.models.itms import ConnectionItem



def path(
    nr: Nornir,
    hostname: str,
    destination: IPv4Interface,
) -> Tuple[List[List[ConnectionItem]], AggregatedResult]:

    # TODO

    return paths, path_result


if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    paths, result = path(nr, "r01", "10.0.0.10/32")
    print_result(
        result,
        # severity_level=logging.DEBUG,
    )

    for i, p in enumerate(paths):
        print(f"{'=' * 16} Path {i} {'=' * 16}")
        for step in p:
            print(
                f"{step.local_name} {step.local_interface} {step.local_mtu} -> {step.neighbor_mtu} {step.neighbor_interface} {step.neighbor_name}"
            )

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/path.py
    ================ Path 0 ================
    r01 Ethernet1 1500 -> 1500 Ethernet1 r02
    r02 Ethernet4 1500 -> 1500 Ethernet1 r04
    r04 Ethernet5 1500 -> 1500 Ethernet1 r06
    r06 Ethernet2 1500 -> 1500 Ethernet1 r08
    r08 Ethernet4 1500 -> 1500 Ethernet1 r10
    ================ Path 1 ================
    r01 Ethernet1 1500 -> 1500 Ethernet1 r02
    r02 Ethernet4 1500 -> 1500 Ethernet1 r04
    ...
    """
```

One possible solution

```python
import logging
from typing import Dict, Tuple, List
from ipaddress import IPv4Interface
from copy import copy

from nornir.core import Nornir
from nornir.core.task import Task, Result, AggregatedResult
from nornir_napalm.plugins.tasks import napalm_get, napalm_cli

from mtu_tool.models.itms import ConnectionItem
from mtu_tool.models.show_route import Model as RouteModel

from mtu_tool.exceptions import PathProcedureError, NoHostFoundException


# Use a cash to interact only once with a device/mock
CACHE: Dict[str, Result] = {}


def get_next_hop(task: Task, destination: IPv4Interface) -> Result:
    if cached := CACHE.get(task.host.name):
        return cached

    show_cmd = f"show ip route {destination} | json"
    result_cli = task.run(
        task=napalm_cli,
        commands=[show_cmd],
        severity_level=logging.DEBUG,
    )
    result_getters = task.run(
        task=napalm_get,
        getters=[
            "get_lldp_neighbors",
            "get_interfaces",
        ],
        severity_level=logging.DEBUG,
    )

    # The CLI command returns a JSON string, use Pydantic Model
    show_routes = RouteModel.model_validate_json(result_cli[0].result[show_cmd])

    # default VRF hardcoded for now
    routes = show_routes.vrfs["default"].routes
    if len(routes) == 0:
        raise Exception("No route found")

    # Collect "interface: mtu" mapping
    interface_mtus = {
        name: data["mtu"]
        for name, data in result_getters[0].result["get_interfaces"].items()
    }

    next_hopes = []
    if not routes[str(destination)].directlyConnected:
        for via in routes[str(destination)].vias:
            interface_data = result_getters[0].result["get_interfaces"][via.interface]
            lldp_data = result_getters[0].result["get_lldp_neighbors"][via.interface]
            # For now we just take the first lldp neighbor. Feel free to improve
            # If DNS PTR works for all p2p links, we could use it to get the next hop host
            next_hopes.append(
                ConnectionItem(
                    local_name=task.host.name,
                    local_interface=via.interface,
                    local_mtu=interface_data["mtu"],
                    neighbor_name=lldp_data[0]["hostname"],
                    neighbor_interface=lldp_data[0]["port"],
                    # The remote MTU needs to be updated in the recursion
                )
            )
    result_data = {
        "next_hops": next_hopes,
        "interface_mtus": interface_mtus,
    }
    result = Result(host=task.host, result=result_data)
    CACHE[task.host.name] = result
    return result


def path(
    nr: Nornir,
    hostname: str,
    destination: IPv4Interface,
) -> Tuple[List[List[ConnectionItem]], AggregatedResult]:

    nr_host = nr.filter(name=hostname)
    if len(nr_host.inventory) == 0:
        raise NoHostFoundException(f"Host {hostname} not found in inventory")

    path_result = AggregatedResult("path")

    path: List[ConnectionItem] = []  # first path list
    paths = [path]  # list of all existing paths

    def split_path(path):
        """
```

```
        Generator that first yields the original list and then yields copies of the list passed as an argument. Each copy is added to the 'paths' collection.

        Parameters:
        path (list): The original list to be yielded and copied.

        Yields:
        list: The original list.

        Yields:
        list: Copies of the original list.
        """
        path_copy = copy(path)
        yield path
        while True:
            new_path = copy(path_copy)
            paths.append(new_path)
            yield new_path

    def walk_path(
        hostname: str, interface: str, path, aggregated_result: AggregatedResult
    ) -> int:

        nr_host = nr.filter(name=hostname)

        result = nr_host.run(
            task=get_next_hop,
            destination=destination,
        )
        if hostname not in aggregated_result:
            # If hostname is already in aggregated_result we are working with a cashed result
            aggregated_result.update(result)

        next_hops: List[ConnectionItem] = result[hostname][0].result["next_hops"]
        interface_mtus = result[hostname][0].result["interface_mtus"]

        # generator copies the path and adds it to the paths list if multiple paths exist
        split_path_generator = split_path(path)
        for nh in next_hops:
            passed_path = next(split_path_generator)
            # Add next hop information to the list to have the right order and beeing able to copy the path
            passed_path.append(nh)
            remote_mtu = walk_path(
                nh.neighbor_name,
                nh.neighbor_interface,
                passed_path,
                aggregated_result,
            )
            # walk_path returns the MTU uf the remote and the next hop information added to the list can be updated now
            nh.neighbor_mtu = remote_mtu

        return int(interface_mtus[interface])

    # Start recursion
    try:
        walk_path(
            hostname=hostname,
            interface="Loopback0",
            path=path,
            aggregated_result=path_result,
        )
    except TypeError as exc:
        raise PathProcedureError(
            "Most likely a job failed and did return an exception and not a dict",
            path_result,
        ) from exc
    return paths, path_result


if __name__ == "__main__":
    from nornir_rich.functions import print_result
    from mtu_tool.helpers import init_nornir

    nr = init_nornir()

    paths, result = path(nr, "r01", "10.0.0.10/32")
    print_result(
        result,
        # severity_level=logging.DEBUG,
    )

    for i, p in enumerate(paths):
        print(f"{'=' * 16} Path {i} {'=' * 16}")
        for step in p:
            print(
                f"{step.local_name} {step.local_interface} {step.local_mtu} -> {step.neighbor_mtu} {step.neighbor_interface} {step.neighbor_name}"
            )

    """
    mtu-tool-py3.10ins@ubuntu-L:~/mtu_tool$ python mtu_tool/procedure/path.py
    ================ Path 0 ================
    r01 Ethernet1 1500 -> 1500 Ethernet1 r02
    r02 Ethernet4 1500 -> 1500 Ethernet1 r04
    r04 Ethernet5 1500 -> 1500 Ethernet1 r06
    r06 Ethernet2 1500 -> 1500 Ethernet1 r08
    r08 Ethernet4 1500 -> 1500 Ethernet1 r10
    ================ Path 1 ================
    r01 Ethernet1 1500 -> 1500 Ethernet1 r02
    r02 Ethernet4 1500 -> 1500 Ethernet1 r04
    ...
    """
```