Photo by Kirill Sh on Unsplash

Getting Started with Nornir for Python Network Automation

12 min read · May 26, 2024

Syed Asif Follow

Listen Share

Nornir is a Python library designed for network automation tasks. It allows Network Engineers to manage and automate their network devices using Python. Unlike tools like Ansible that use domain-specific languages, Nornir leverages the full power of Python, providing more flexibility and control over your automation scripts.

If you're familiar with Ansible, you know that you first set up your inventory, write tasks, and then execute them on all or selected devices concurrently. Nornir works similarly, but the key difference is that you use Python code instead of a domain-specific language.

Prerequisites and Key Points

Before diving into Nornir, you should have a good understanding of Python basics and a lab environment.

Setting Up a Network Lab with GNS3

Networking labs, providing a sandbox for testing, learning, and refining skills. In this blog, we'll delve into the…

medium.com

Remember, Nornir isn't meant to replace tools like Netmiko or Napalm; it's designed to work alongside them. Think of Nornir as a framework that organizes your automation tasks. For example, to SSH into network devices, you'll still use plugins like Netmiko. We'll cover how these tools integrate with Nornir in the upcoming sections.

Installing Nornir is easy. Just run the following `pip` install command:

```
pip install nornir
```

Overview of Nornir Components

Here's a quick overview of the main components of Nornir. Together, these elements create a powerful framework for network automation.

- **Inventory**: This is where you store information about your devices. Nornir's inventory system is flexible, allowing you to define devices, their credentials, and other details in a structured format.
- **Tasks**: These are the actions you want to perform on your devices, like sending commands or configurations. In Nornir, you write tasks using Python functions.

- **Plugins:** Nornir supports plugins to extend its functionality. Plugins can be used for tasks, inventory management, or adding new features.
- **Parallel Execution:** One of Nornir's strengths is its ability to run tasks in parallel across multiple devices. This feature speeds up network automation tasks significantly, especially for large networks.
- **Results:** Nornir has a powerful feature called Results. After executing tasks on your devices, Nornir collects and stores the outcomes in a Results object.

We will go through each of these components in detail with some examples.

Project Directory Structure

Here is my directory structure and the files:

```
(.venv) zolo@u22s:~/nornir-lab$ tree
.
├── config.yaml
├── defaults.yaml
├── groups.yaml
├── hosts.yaml
└── nornir_test.py

0 directories, 5 files
```

Configuring Nornir with YAML

Configuration File (`config.yaml`)

The `config.yaml` file is a configuration for Nornir that outlines how it should manage its inventory and execute tasks. It's written in YAML, a human-readable data format, making it easy to understand and modify.

```
# config.yaml
---
inventory:
  plugin: SimpleInventory
  options:
    host_file: 'hosts.yaml'
    group_file: 'groups.yaml'
    defaults_file: 'defaults.yaml'
```

```
runner:
  plugin: threaded
  options:
    num_workers: 5
```

- **Inventory**: Specifies how Nornir should load information about network devices. It uses the SimpleInventory plugin and points to three files (other inventory plugins can read from Ansible's inventory files or tools like NetBox):
- `hosts.yaml` for individual device details
- `groups.yaml` for settings common to groups of devices
- `defaults.yaml` for default settings applicable to all devices unless overridden in the other files.
- **Runner**: Controls how Nornir runs tasks across devices. Here, the threaded plugin is used with `num_workers` set to 5, meaning tasks will be executed in parallel on up to 5 devices at a time.

Host File (`hosts.yaml`)

This file contains details about each network device. For every device, you can specify parameters such as its hostname, IP address, platform type (e.g., Cisco, Arista), and credentials. Nornir uses this information to connect to and manage the devices individually.

```
# hosts.yaml
---
sw1:
  hostname: 172.16.10.11
  groups:
    - cisco_switch

R1:
  hostname: 172.16.10.12
  groups:
    - cisco_router
```

Groups File (`groups.yaml`)

The `groups.yaml` file is used to define common settings for groups of devices. For example, if you have several devices from the same vendor or within the same part of your network, you can group them and assign shared parameters like vendor or credentials. Devices `hosts.yaml` can be associated with one or more groups, inheriting the group's settings.

```
# groups.yaml
---
cisco_switch:
  platform: cisco_ios

cisco_router:
  platform: cisco_ios
```

Writing and Running Your First Nornir Script

Let's look at a simple example to understand how our first Nornir script works, using the inventory examples we discussed before (with Cisco devices).

```
# nornir hello script
from nornir import InitNornir

def say_hello(task):
    print("Hello, Nornir")

nr = InitNornir(config_file="config.yaml")
nr.run(task=say_hello)
```

```
# output
(.venv) zolo@u22s:~/nornir-lab$ python nornir_test.py
Hello, Nornir
Hello, Nornir
```

- **Importing Nornir**: The script starts by importing the `InitNornir` class from the Nornir library. This is essential for initializing our Nornir environment.
- **Defining a Task Function:** Next, we define a simple task function, `say_hello`, that takes `task` as an argument. This function simply prints a message, "Hello,

Nornir". In Nornir, tasks are functions that you want to execute on your network devices. The `task` argument represents the task being executed and carries information about the current device it's running on.

- **Initializing Nornir**: We then create an instance of Nornir using `InitNornir` , specifying `config.yaml` as the configuration file. This configuration includes our inventory setup with `hosts.yaml` , `groups.yaml` , and `defaults.yaml` , defining our network devices and their properties.
- **Running the Task**: Finally, we use the `.run()` method on our Nornir instance to execute the `say_hello` task across all devices specified in our inventory. Because our `config.yaml` specifies a runner with 5 workers, tasks can be executed in parallel on up to 5 devices at a time.
- **Output:** Given our inventory setup, the script prints “Hello, Nornir” once for each device in the inventory. Since we have two devices (sw1 and R1), we see the message printed twice, indicating the task executed successfully on each device.

Let’s look at our second example to see how to use the `print_result` plugin. If you have used Ansible before, you know it provides a nice output showing what's going on.

You can install the plugin using the `pip install` command:

```
pip install nornir_utils
```

```
# nornir print script
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result

def say_hello(task):
    return "Hello, Nornir"

nr = InitNornir(config_file="config.yaml")
result = nr.run(task=say_hello)
print_result(result)
```

In this updated example, the significant addition is the use of `print_result` from the `nornir_utils` plugin. This function is designed to neatly display the results of tasks executed by Nornir on your network devices.

- **Importing `print_result`** : We've added a new import statement to bring in the `print_result` function. This plugin is used to format and print the outcome of our tasks in a readable manner.
- **Storing and Printing Results**: Instead of directly printing a message within the `say_hello` task, we now return the message. The main script captures the output of the `nr.run` method in a variable named `result` . This variable holds detailed information about the task execution on each device. Finally, `print_result(result)` is called to display this information.

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_test2.py
say_hello**********************************************************************
* R1 ** changed : False *******************************************************
vvvv say_hello ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Hello, Nornir
^^^^ END say_hello ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : False ******************************************************
vvvv say_hello ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Hello, Nornir
^^^^ END say_hello ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- **Output**: The output from print_result shows a structured view of the task execution. For each device (R1 and sw1), it indicates the task name (say_hello), the status (changed: False), and the message returned by the task (Hello, Nornir). This format makes it easy to understand what happened during the script's execution and to troubleshoot if necessary.

Accessing the Host's Parameters

The `task.host` object allows us to access various parameters of the host on which the task is currently executing. You can retrieve specific details like:

- `task.host` : The name of the current device.
- `task.host.groups` : The group(s) the device belongs to.

- `task.host.hostname` : The hostname or IP address of the device.

By using `task.host` along with its attributes, we can dynamically insert each device's specific information into our task's return message.

```
# nornir access host script
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result

def say_hello(task):
    return f"Hello, {task.host} - {task.host.groups} - {task.host.hostname}"

nr = InitNornir(config_file="config.yaml")
result = nr.run(task=say_hello)
print_result(result)
```

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_test3.py
say_hello**********************************************************************
* R1 ** changed : False *******************************************************
vvvv say_hello ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Hello, R1 - [Group: cisco_device] - 172.16.10.12
^^^^ END say_hello ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : False ******************************************************
vvvv say_hello ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Hello, sw1 - [Group: cisco_device] - 172.16.10.11
^^^^ END say_hello ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This script shows how to use `task.host` to access and display details about each device, including its name, groups, and hostname.

Filtering Devices with Nornir

Here's another example of how you can run tasks on specific devices.

```
# nornir filter script
from nornir import InitNornir
from nornir_utils.plugins.functions import print_result


def say_hello(task):
    return f"Hello, {task.host} - {task.host.groups} - {task.host.hostname}"


nr = InitNornir(config_file="config.yaml")
nr = nr.filter(hostname="172.16.10.11")

result = nr.run(task=say_hello)
print_result(result)
```

In this script, we’re using the `filter` method to narrow down the devices based on their hostname. Specifically, we're filtering for devices with the hostname "172.16.10.11", which corresponds to switchs in our inventory. Then, we run the `say_hello` task only on these filtered devices. Finally, we print the results using the `print_result` function.

Introduction to the nornir_netmiko Plugin

Now, we’ve reached the really exciting part where we can actually execute commands on devices and see the output. You might think, like I did when I was just getting started, “Alright, I’ll just create a new function, import Netmiko’s ConnectHandler, and get on with it, right?”

But here’s a pleasant surprise: the awesome teams behind Nornir and Netmiko have already done a lot of the heavy lifting for us. They’ve created plug-ins that we can easily import. To get the netmiko plug-in, all you need to do is run `pip install nornir_netmiko`. This simple command fetches and installs everything you need to start sending commands to your network devices through your Nornir scripts.

Installing the nornir_netmiko Plugin:

```
pip install nornir_netmiko
```

Sending Commands with nornir_netmiko:

```
# nornir show cmd script
from nornir import InitNornir
from nornir_netmiko.tasks import netmiko_send_command
from nornir_utils.plugins.functions import print_result
```

```
nr = InitNornir(config_file="config.yaml")

results = nr.run(
    task=netmiko_send_command, command_string="show ip interface brief | excl c
)
print_result(results)
```

In this script, we're leveraging the nornir_netmiko plugin, particularly the `netmiko_send_command` function, to execute commands on network devices. After initializing Nornir, we call `nr.run`, passing in `netmiko_send_command` as the task. We specify the command we want to run on our devices with `command_string='show ip interface brief | excl down'`.

```
# Output of Sending Commands

(.venv) zolo@u22s:~/nornir-lab$ python nornir_netmiko_show.py
netmiko_send_command***********************************************************
* R1 ** changed : False *******************************************************
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Interface                  IP-Address      OK? Method Status                Pro
FastEthernet0/0            172.16.10.12    YES NVRAM  up                    up
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : False ******************************************************
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
Interface              IP-Address      OK? Method Status                Protoco
GigabitEthernet0/0     unassigned      YES unset  up                    up
GigabitEthernet3/3     unassigned      YES unset  up                    up
Vlan1                  172.16.10.11    YES NVRAM  up                    up
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Configuring Devices with Nornir and Netmiko

The `netmiko_send_config` function is utilized to push configuration commands to devices, specifically targeting router devices with `hostname="172.16.10.12"`.

After filtering for these devices, we execute `netmiko_send_config` to send configuration commands. The output mark `changed : True` indicates that the configuration was successfully applied, reflecting changes made on the devices.

```
# nornir config cmd script
from nornir import InitNornir
```

```
nr = nr.filter(hostname="172.16.10.12")
results = nr.run(task=netmiko_send_config, config_commands=["ntp server 1.1.1.1
print_result(results)
```

```
# Output
(.venv) zolo@u22s:~/nornir-lab$ python nornir_netmiko_conf.py
netmiko_send_config*************************************************************
* R1 ** changed : True *********************************************************
vvvv netmiko_send_config ** changed : True vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#ntp server 1.1.1.1
R1(config)#end
R1#
^^^^ END netmiko_send_config ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Dynamic Configuration with Host-Specific Data

I have made a slight change to the script to demonstrate a more dynamic feature of Nornir: accessing host-specific data within a task function for customized configurations across devices.

In the updated `hosts.yaml` file, you'll notice an additional data section under `R1`. This section allows us to define custom data applicable to devices. Here, we've specified an NTP server address (1.1.1.1) under data, making it accessible to devices associated with this router.

```
from nornir import InitNornir
from nornir_netmiko.tasks import netmiko_send_config
from nornir_utils.plugins.functions import print_result

def set_ntp(task):
    ntp_server = task.host["ntp"]
    task.run(task=netmiko_send_config, config_commands=[f"ntp server {ntp_serve
nr = InitNornir(config_file="config.yaml")
nr = nr.filter(hostname="172.16.10.12")

results = nr.run(task=set_ntp)
print_result(results)
```

The function `set_ntp` fetches the NTP server address using `task.host['ntp']`, dynamically inserting it into the configuration command. This method ensures that the NTP server setting applied to each device is retrieved from the inventory, allowing for centralized management of device configurations.

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_netmiko_data.py
set_ntp**********************************************************************
* R1 ** changed : True *******************************************************
vvvv set_ntp ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
---- netmiko_send_config ** changed : True -----------------------------------
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#ntp server 1.1.1.1
R1(config)#end
R1#
^^^^ END set_ntp ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In this example, you would have seen two different ways to run tasks: `task.run` and `nr.run`. Here's a brief explanation of the difference between the two:

- task.run: This is used within a task function to execute another task. Think of it as calling a sub-task within your main task. When you use it `task.run`, you're essentially saying, "While performing this task, go ahead and run these additional tasks as part of it."
- nr.run: On the other hand, `nr.run` is used to kick off tasks at the top level. This is the method you call when you want to start your automation process and execute tasks across your inventory of devices.

In summary, `nr.run` is used to initiate your automation tasks on your network devices, while `task.run` allowing you to organize and modularize your tasks by calling other tasks within a task.

Integrating Python NAPALM with Nornir

In this section, we will extend our network automation capabilities by integrating NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) with Nornir. NAPALM provides a common API to interact with different network devices, supporting several network operating systems like IOS, Junos, and EOS.

Installing NAPALM

First, we need to install NAPALM. You can install it using pip:

```
pip install napalm
```

Additionally, we need to install the Nornir NAPALM plugin:

```
pip install nornir_napalm
```

Let’s begin with a basic example of using NAPALM to retrieve data from our network devices. We’ll use NAPALM to get the interfaces’ IP addresses.

```
# nornir napalm get interfaces ip script
from nornir import InitNornir
from nornir_napalm.tasks import napalm_get
from nornir_utils.plugins.functions import print_result

nr = InitNornir(config_file="config.yaml")

results = nr.run(
    task=napalm_get, getters=["interfaces_ip"]
)
print_result(results)
```

In this script, we initialize Nornir and then run the `napalm_get` task with the getter `interfaces_ip` to retrieve the IP addresses of the interfaces.

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_napalm_int.py
napalm_get*****************************************************************
* R1 ** changed : False ***************************************************
vvvv napalm_get ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
{ 'interfaces_ip': { 'FastEthernet0/0': { 'ipv4': { '172.16.10.12': { 'prefix_l
^^^^ END napalm_get ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : False **************************************************
vvvv napalm_get ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
{'interfaces_ip': {'Vlan1': {'ipv4': {'172.16.10.11': {'prefix_length': 24}}}}}
^^^^ END napalm_get ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

We can also use NAPALM to configure devices. The following script demonstrates how to use the `napalm_configure` task to push configuration changes to devices.

```
# nornir napalm configure script
from nornir import InitNornir
from nornir_napalm.tasks import napalm_configure
from nornir_utils.plugins.functions import print_result

nr = InitNornir(config_file="config.yaml")

def configure_ntp(task):
    ntp_config = """
    ntp server 1.1.1.1
    """
    task.run(task=napalm_configure, configuration=ntp_config)

results = nr.run(task=configure_ntp)
print_result(results)
```

In this script, we define a `configure_ntp` function that uses `napalm_configure` to apply an NTP configuration to the devices. We then run this task across our inventory.

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_napalm_configure.py
configure_ntp**************************************************************
* R1 ** changed : True ****************************************************
vvvv configure_ntp ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
---- napalm_configure ** changed : True ----------------------------------------
+    ntp server 1.1.1.1
```

```
^^^^ END configure_ntp ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : True *******************************************************
vvvv configure_ntp ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
---- napalm_configure ** changed : True ---------------------------------------
+    ntp server 1.1.1.1
^^^^ END configure_ntp ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Let’s look at another example where we retrieve basic device facts using NAPALM.

```
# nornir napalm get facts script
from nornir import InitNornir
from nornir_napalm.plugins.tasks import napalm_get
from nornir_utils.plugins.functions import print_result


def get_facts(task):
    task.run(task=napalm_get, getters=["facts"])


nr = InitNornir(config_file="config.yaml")
result = nr.run(task=get_facts)
print_result(result)
```

In this script, the napalm_get task is used with the facts-getter to retrieve basic information about the devices, such as vendor, model, serial number, and uptime.

```
(.venv) zolo@u22s:~/nornir-lab$ python nornir_napalm_get.py
get_facts**********************************************************************
* R1 ** changed : False *******************************************************
vvvv get_facts ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
---- napalm_get ** changed : False --------------------------------------------
{ 'facts': { 'fqdn': 'R1.test.lab',
             'hostname': 'R1',
             'interface_list': ['FastEthernet0/0', 'FastEthernet0/1'],
             'model': '3725',
             'os_version': '3700 Software (C3725-ADVENTERPRISEK9-M), Version '
                           '12.4(15)T14, RELEASE SOFTWARE (fc2)',
             'serial_number': 'FTX0945W0MY',
             'uptime': 2040.0,
             'vendor': 'Cisco'}}
^^^^ END get_facts ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
* sw1 ** changed : False ******************************************************
vvvv get_facts ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
```

```
    ---- napalm_get ** changed : False --------------------------------------------
    { 'facts': { 'fqdn': 'SW1.test.lab',
                 'hostname': 'SW1',
                 'interface_list': [ 'GigabitEthernet0/0',
                                     'GigabitEthernet0/1',
                                     'GigabitEthernet0/2',
                                     'GigabitEthernet0/3',
                                     'GigabitEthernet1/0',
                                     'GigabitEthernet1/1',
                                     'GigabitEthernet1/2',
                                     'GigabitEthernet1/3',
                                     'GigabitEthernet2/0',
                                     'GigabitEthernet2/1',
                                     'GigabitEthernet2/2',
                                     'GigabitEthernet2/3',
                                     'GigabitEthernet3/0',
                                     'GigabitEthernet3/1',
                                     'GigabitEthernet3/2',
                                     'GigabitEthernet3/3',
                                     'Vlan1'],
                 'model': 'IOSv',
                 'os_version': 'vios_l2 Software (vios_l2-ADVENTERPRISEK9-M), '
                               'Experimental Version 15.2(20200924:215240) '
                               '[sweickge-sep24-2020-l2iol-release 135]',
                 'serial_number': '9BPOBFRTOEY',
                 'uptime': 3960.0,
                 'vendor': 'Cisco'}}
    ^^^^ END get_facts ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Nornir, integrated with NAPALM and supported by plugins like nornir_netmiko and nornir_utils, offers a robust solution for network automation tasks. This combination enhances automation capabilities, allowing for efficient retrieval and configuration of network device data. With NAPALM's multivendor support and common API, along with Nornir's powerful task management, network automation becomes more manageable, scalable, and tailored to your specific needs. This guide has demonstrated how to set up Nornir, create basic and advanced scripts, utilize host-specific data for dynamic configurations, and leverage plugins to efficiently manage and configure network devices.

Network-Automotion / nornir_lab · GitLab

Networking

Network Automation Tools

Python Programming

Nornir

Python3

Follow

Written by Syed Asif

147 followers · 51 following

Associate Electronics Engineer in Radio Technology

Responses (1)

Write a response

What are your thoughts?

P Pradyumna Kubear
Oct 6, 2024

well written and amazing

1 reply Reply