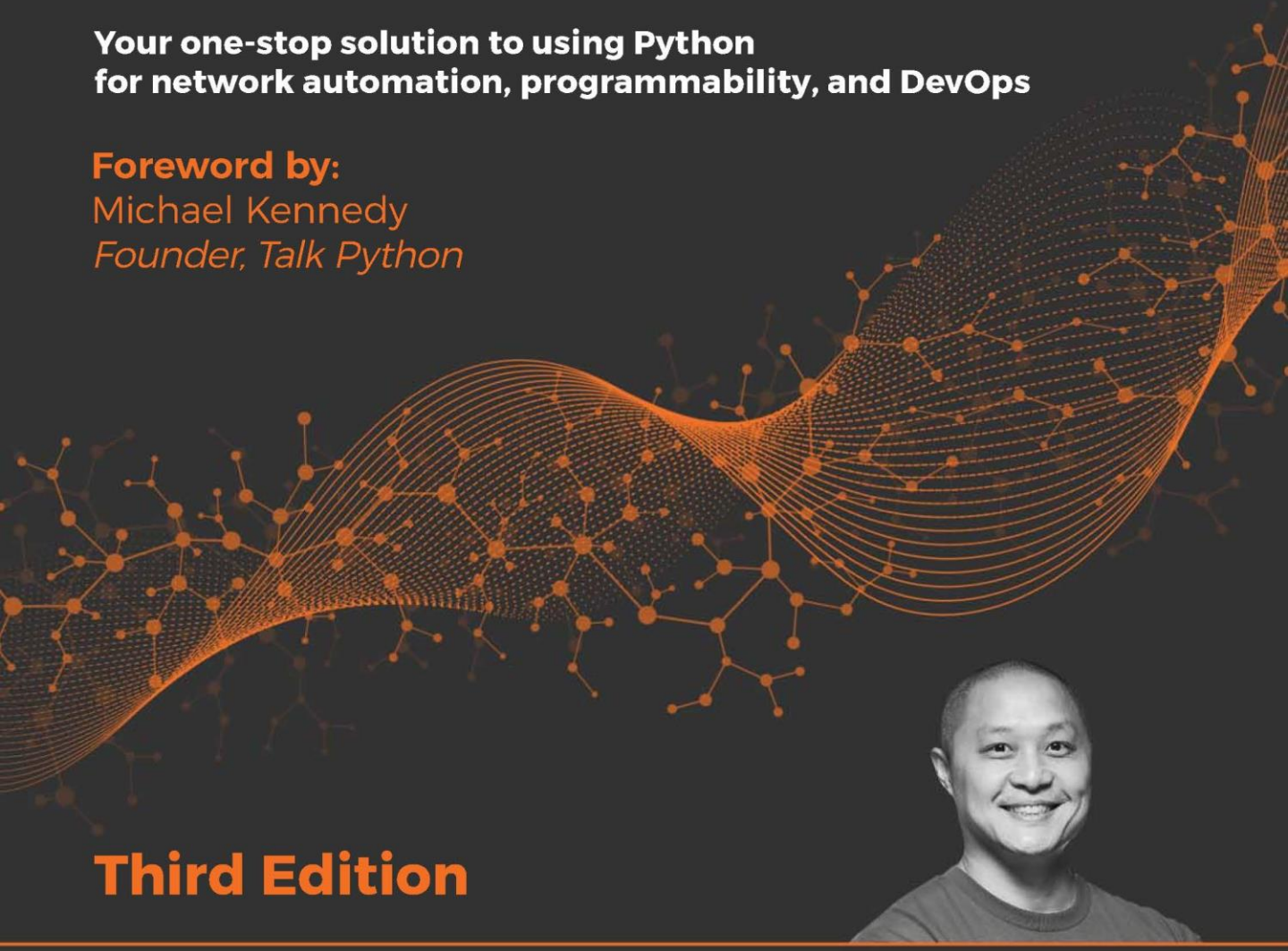
EXPERT INSIGHT

# Mastering Python Networking

**Your one-stop solution to using Python for network automation, programmability, and DevOps**

**Foreword by:**
Michael Kennedy
*Founder, Talk Python*





**Third Edition**

Eric Chou

Packt

# Mastering Python Networking

*Third Edition*



**Producer:** Tushar Gupta
**Acquisition Editor – Peer Reviews:** Suresh Jain
**Project Editor:** Tom Jacob
**Content Development Editor:** Ian Hough
**Technical Editor:** Karan Sonawane
**Copy Editor:** Safis Editing
**Proofreader:** Safis Editing
**Indexer:** Manju Arasan
**Presentation Designer:** Pranit Padwal

First published: June 2017
Second edition: August 2018
Third edition: January 2020

Production reference: 1280120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-83921-467-7

www.packt.com

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

# Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Contributors

## About the author

**Eric Chou** is a seasoned technologist with over 20 years of experience. He has worked on some of the largest networks in the industry while working at Amazon, Azure, and other Fortune 500 companies. Eric is passionate about network automation, Python, and helping companies build better security postures.

In addition to being the author of *Mastering Python Networking* (Packt), he is also the co-author of *Distributed Denial of Service (DDoS): Practical Detection and Defense* (O'Reilly Media).

Eric is also the primary inventor for two U.S. patents in IP telephony. He shares his deep interest in technology through his books, classes, and blog, and contributes to some of the popular Python open source projects.

---

I would like to thank the open source, network engineering, and Python community members and developers for generously sharing their compassion, knowledge, and code. Without them, many of the projects referenced in this book would not have been possible. I hope I had made small contributions to these wonderful communities in my own ways as well.

I would like to thank to the Packt team, Tushar, Tom, Ian, Alex, Jon, and many others, for the opportunity to collaborate on the third edition of the book. Special thanks to the technical reviewer, Rickard Körkkö, for generously agreeing to review the book.

Thank you, Mandy and Michael for writing the Forewords for this book. I can't express my appreciation enough. You guys rock!

To my parents and family, your constant support and encouragement made me who I am, I love you.

---

# 3
# APIs and Intent-Driven Networking

In *Chapter 2*, *Low-Level Network Device Interactions*, we looked at ways to interact with the network devices using Pexpect and Paramiko. Both of these tools use a persistent session that simulates a user typing in commands as if they are sitting in front of a Terminal. This works fine up to a point. It is easy enough to send commands over for execution on a device and capture the output. However, when the output becomes more than a few lines of characters, it becomes difficult for a computer program to interpret the output. The returned output from Pexpect and Paramiko is a series of characters meant to be read by a human being. The structure of the output consists of lines and spaces that are human-friendly but difficult to be understood by computer programs.

In order for our computer programs to automate many of the tasks we want to perform, we need to interpret the returned results and make follow-up actions based on the returned results. When we cannot accurately and predictably interpret the returned results, we cannot execute the next command with confidence.

Luckily, this problem was solved by the internet community. Imagine the difference between a computer and a human being when they are both reading a web page. The human sees words, pictures, and spaces interpreted by the browser; the computer sees raw HTML code, Unicode characters, and binary files. What happens when a website needs to become a web service for another computer? The same web resources need to accommodate both human clients and other computer programs. Doesn't this problem sound familiar to the one that we presented before?

The answer is the **application program interface** (**API**). It is important to note that an API is a concept and not a particular technology or framework, according to Wikipedia:

> *In computer programming, an application programming interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it's a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer.*

In our use case, the set of clearly defined methods of communication would be between our Python program and the destination device. The APIs from our network devices provide a separate interface for the computer programs. The exact API implementation is vendor-specific. One vendor will prefer XML while another might use JSON; some might provide HTTPS as the underlying transport protocol and others might provide Python libraries as wrappers. We will see examples of each in this chapter.

Despite the differences, the idea of an API remains the same: it is a communication method optimized for other computer programs.

In this chapter, we will look at the following topics:

- Treating **infrastructure as code** (**IaC**), intent-driven networking, and data modeling
- Cisco NX-API and the **Application Centric Infrastructure** (**ACI**)
- Juniper **Network Configuration Protocol** (**NETCONF**) and PyEZ
- Arista eAPI and pyeapi

We will start by examining why we want to treat infrastructure as code.

# Infrastructure-as-code

In a perfect world, network engineers and architects who design and manage networks should focus on what they want the network to achieve instead of the device-level interactions. But we all know the world is far from perfect. Many years ago when I worked as an intern for a second-tier ISP, wide-eyed and excited, one of my first assignments was to install a router on a customer's site to turn up their fractional frame relay link (remember those?). *How would I do that?* I asked. I was handed down a standard operating procedure for turning up frame relay links.

I went to the customer site, blindly typed in the commands, looked at the green lights flashing, then happily packed my bag and patted myself on the back for a job well done. As exciting as that assignment was, I did not fully understand what I was doing. I was simply following instructions without thinking about the implication of the commands I was typing in. How would I troubleshoot something if the light was red instead of green? I think I would have called the office and cried for help (tears optional).

Of course, network engineering is not about typing in commands into a device, but it is about building a way that allows services to be delivered from one point to another with as little friction as possible. The commands we have to use and the output that we have to interpret are merely means to an end. In other words, we should be focused on our intent for the network. **What we want our network to achieve is much more important than the command syntax we use to get the device to do what we want it to do**. If we further abstract that idea of describing our intent as lines of code, we can potentially describe our whole infrastructure as a particular state. The infrastructure will be described in lines of code with the necessary software or framework to enforce that state.

# Intent-driven networking

Since the publication of the first edition of this book, the terms **intent-based networking** (**IBN**) and **intent-driven networking** (**IDN**) have seen an uptick in use after major network vendors chose to use them to describe their next-generation devices. The two terms generally mean the same thing. *In my opinion, intent-driven networking is the idea of defining a state that the network should be in and having software code to enforce that state*. As an example, if my goal is to block port `80` from being externally accessible, that is how I should declare it as the intention of the network. The underlying software will be responsible for knowing the syntax of configuring and applying the necessary access-list on the border router to achieve that goal. Of course, IDN is an idea with no clear answer on the exact implementation. The software we use to enforce our declared intent can be a library, a framework, or a complete package that we purchase from a vendor.

In using an API, it is my opinion that it gets us closer to a state of intent-driven networking. In short, because we abstract the layer of a specific command executed on our destination device, we focus on our intent instead of the specific commands. For example, going back to our `block port 80` access-list example, we might use access-list and access-group on a Cisco and filter-list on a Juniper. However, in using an API, our program can start asking the executor for their intent while masking the kind of physical device the software is talking to. We can even use a higher-level declarative framework, such as Ansible, which we will cover in *Chapter 4*, *The Python Automation Framework – Ansible Basics*. But for now, let's focus on network APIs.

# Screen scraping versus API structured output

Imagine a common scenario where we need to log into the network device and make sure all the interfaces on the device are in an up/up state (both the status and the protocol are showing as `up`). For the human network engineers getting into a Cisco NX-OS device, it is simple enough to issue the `show ip interface brief` command in the Terminal to easily tell from the output which interface is up:

```
nx-osv-2# show ip int brief
IP Interface Status for VRF "default"(1) Interface IP Address Interface
Status
Lo0 192.168.0.2 protocol-up/link-up/admin-up
Eth2/1 10.0.0.6 protocol-up/link-up/admin-up
nx-osv-2#
```

The line break, white spaces, and the first line of the column title are easily distinguished by the human eye. In fact, they are there to help us line up, say, the IP addresses of each interface from line one to line two and three. If we were to put ourselves in the computer's position, all these spaces and line breaks only take us away from the really important output, which is: which interfaces are in the up/up state? To illustrate this point, we can look at the Paramiko output for the same operation:

```
>>> new_connection.send('show ip int brief/n')
16
>>> output = new_connection.recv(5000)
>>> print(output)
b'sh ip int briefrrnIP Interface Status for VRF "default"(1)r\nInterface
IP Address Interface StatusrnLo0 192.168.0.2 protocol-up/link-up/admin-up
rnEth2/1 10.0.0.6 protocol-up/link-up/admin-up r\nrnx- osv-2# '
>>>
```

If we were to parse out that data contained in the "output" variable, here is what I would do in a pseudo-code fashion (pseudo-code means a simplified representation of the actual code I would write) to subtract the text into the information I need:

1. Split each line via the line break.
2. I do not need the first line that contains the executed command of `show ip interface brief` and will disregard it.
3. Take out everything on the second line up until the VRF, and save it in a variable as we want to know which VRF the output is showing.

4. For the rest of the lines, because we do not know how many interfaces are there, we will use a regular expression statement to search if the line starts with interface names, such as `lo` for loopback and `Eth` for Ethernet interfaces.
5. We will need to split this line into three sections separated by a space, each consisting of the name of the interface, IP address, and then the interface status.
6. The interface status will then be split further using the forward slash (`/`) to give us the protocol, link, and the admin status.

Whew, that is a lot of work just for something that a human being can tell at a glance! You might be able to optimize the code and reduce the number of lines in the code; but in general, the steps are what we need to do when we need to screen scrap texts that are somewhat unstructured. There are many downsides to this method, but some of the bigger problems that I can see are listed as follows:

- **Scalability**: We spent so much time on painstaking details to parse out the outputs from each command. It is hard to imagine how we can do this for the hundreds of commands that we typically run.
- **Predictability**: There is really no guarantee that the output stays the same between different software versions. If the output is changed ever so slightly, it might just render our hard-fought battle of information gathering useless.
- **Vendor and software lock-in**: Perhaps the biggest problem is that once we spend all this time parsing the output for this particular vendor and software version, in this case, Cisco NX-OS, we need to repeat this process for the next vendor that we pick. I don't know about you, but if I were to evaluate a new vendor, the new vendor would be at a severe onboarding disadvantage if I have to rewrite all the screen scrap code again.

Let's compare that with an output from an NX-API call for the same `show ip interface brief` command. We will go over the specifics of getting this output from the device later in this chapter, but what is important here is to compare the following output to the previous screen scraping output:

```
{
"ins_api":{
"outputs":{
"output":{
"body":{ "TABLE_intf":[
{
"ROW_intf":{
"admin-state":"up",
```

```
"intf-name":"Lo0",
"iod":84,
"ip-disabled":"FALSE",
"link-state":"up",
"prefix":"192.168.0.2",
"proto-state":"up"
}
},
{
"ROW_intf":{
"admin-state":"up",
"intf-name":"Eth2/1",
"iod":36,
"ip-disabled":"FALSE",
"link-state":"up",
"prefix":"10.0.0.6",
"proto-state":"up"
}
}
],
"TABLE_vrf":[
{
"ROW_vrf":{
"vrf-name-out":"default"
}
},
{
"ROW_vrf":{
"vrf-name-out":"default"
}
}
]
},
"code":"200",
"input":"show ip int brief",
"msg":"Success"
}
```

```
},
"sid":"eoc",
"type":"cli_show",
"version":"1.2"
}
}
```

NX-API can return output in XML or JSON, and this is the JSON output that we are looking at. Right away, you can see the output is structured and can be mapped directly to the Python dictionary data structure. Once this is converted to a Python dictionary, there is no parsing required — you can simply pick the key and retrieve the value associated with the key. You can also see from the output that there are various metadata in the output, such as the success or failure of the command. If the command fails, there will be a message telling the sender the reason for the failure. You no longer need to keep track of the command the program issued, because it is already returned to you in the `input` field. There is also other useful metadata in the output, such as the NX-API version.

This type of exchange makes life easier for both vendors and operators. On the vendor side, they can easily transfer configuration and state information. They can add extra fields when the need to expose additional data arises using the same data structure. On the operator side, they can easily ingest the information and build their infrastructure around it. It is generally agreed by all that network automation and programmability is much needed by both network vendors and operators. The questions are usually centered on the format and structure of the automation. As you will see later in this chapter, there are many competing technologies under the umbrella of API. On the transport language alone, we have REST API, NETCONF, and RESTCONF, among others.

Ultimately, the overall market might decide about the final data format in the future. In the meantime, each of us can form our own opinions and help drive the industry forward.

# Data modeling for infrastructure-as-code

According to Wikipedia (`https://en.wikipedia.org/wiki/Data_model`), the definition for a data model is as follows:

> *A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to properties of real-world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner.*

The data modeling process is illustrated in the following diagram:

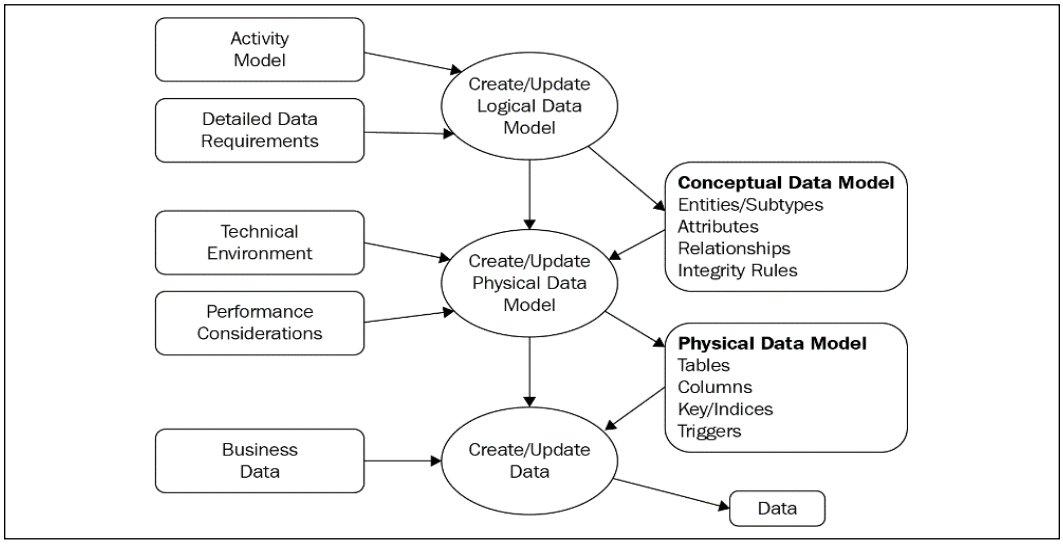



Figure 1: Data modeling process

When applying the data model concept to the network, we say the network data model is an abstract model that describes our network, be it a data center, campus, or global wide area network. If we take a closer look at a physical data center, a layer 2 Ethernet switch can be thought of as a device containing a table of MAC addresses mapped to each port. Our switch data model describes how the MAC address should be kept in a table, which includes the keys, additional characteristics (think of VLAN and private VLAN), and more. Similarly, we can move beyond devices and map the whole data center in a model. We can start with the number of devices in each of the access, distribution, and core layers, how they are connected, and how they should behave in a production environment. For example, if we have a fat-tree network, we can declare in the model how many links each of the spine routers have, the number of routes they should contain, and the number of next-hops each of the prefixes would have.

These characteristics can be mapped out in a format that can then be referenced against as the ideal state that we can check against using software programs.

# YANG and NETCONF

One of the relatively new network data modeling languages that is gaining traction is **Yet Another Next Generation** (**YANG**) (despite common belief, some of the IETF workgroups do have a sense of humor). It was first published in RFC 6020 in 2010, and has since gained traction among vendors and operators.

At the time of writing, the support for YANG has varied greatly from vendors. The adoption rate in production is relatively low. However, among the various data modeling formats, it is one that seemingly has the most momentum.

As a data modeling language, it is used to model the configuration of devices. It can also represent state data manipulated by the NETCONF protocol, NETCONF remote procedure calls, and NETCONF notifications. It aims to provide a common abstraction layer between the protocols used, such as NETCONF, and the underlying vendor-specific syntax for configuration and operations. We will take a look at some examples of its usage in this chapter.

Now that we have discussed the high-level concepts on API-based device management and data modeling, let us take a look at some of the examples from Cisco in their API structures and ACI platform.

# The Cisco API and ACI

Cisco Systems, the 800-pound gorilla in the networking space, have not missed out on the trend of network automation. In their push for network automation, they have made various in-house developments, product enhancements, partnerships, as well as many external acquisitions. However, with product lines spanning routers, switches, firewalls, servers (unified computing), wireless, the collaboration software and hardware, and analytic software, to name a few, it is hard to know where to start.

Since this book focuses on Python and networking, we will scope this section to the main networking products. In particular, we will cover the following:

- Nexus product automation with NX-API
- Cisco NETCONF and YANG examples
- The Cisco ACI for the data center
- The Cisco ACI for the enterprise

For the NX-API and NETCONF examples in this chapter, we can either use the Cisco DevNet always-on lab devices mentioned in *Chapter 2*, *Low-Level Network Device Interactions*, or a locally run Cisco VIRL virtual lab. Since Cisco ACI is a separate product at Cisco, they are licensed with the physical switches. For the following ACI examples, I would recommend using the DevNet or dCloud labs to get an understanding of the tools. If you are one of the lucky engineers who has a private ACI lab that you can use, please feel free to use it for the relevant examples.

We will use the same lab topology as we did in *Chapter 2*, *Low-Level Network Device Interactions*, with the exception of one of the devices running **NX-OSv**:

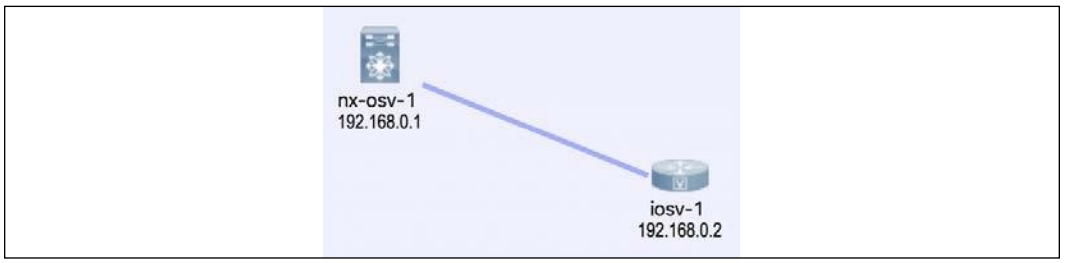



Figure 2: Lab topology

Let's take a look at NX-API.

# Cisco NX-API

Nexus is Cisco's primary product line of data center switches. The NX-API (`http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/6-x/programmability/guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_chapter_011.html`) allows the engineer to interact with the switch outside of the device via a variety of transports including SSH, HTTP, and HTTPS.

## Lab software installation and device preparation

Here are the Ubuntu packages that we will install. You may already have some of the packages, such as `pip` and `git`, from previous chapters:

```
(venv) $ sudo apt-get install -y python3-dev libxml2-dev libxslt1-dev
libffi-dev libssl-dev zlib1g-dev python3-pip git python3-requests
```

If you are using Python 2, use the following packages instead: `sudo apt- get install -y python-dev libxml2-dev libxslt1-dev libffi-dev libssl-dev zliblg-dev python-pip git python-requests`.

The `ncclient` (`https://github.com/ncclient/ncclient`) library is a Python library for NETCONF clients. We should also enable our virtual environment that we created in the last chapter, if not already. We will install `ncclient` from the GitHub repository so that we can install the latest version:

```
(venv) $ git clone https://github.com/ncclient/ncclient
(venv) $ cd ncclient/
(venv) $ python setup.py install
```

NX-API on Nexus devices is turned off by default, so we will need to turn it on. We can either use the user that is already created (if you are using VIRL auto-config), or create a new user for the NETCONF procedures:

```
feature nxapi
username cisco password 5 $1$Nk7ZkwH0$fyiRmMMfIheqE3BqvcL0C1 role
network- operator
username cisco role network-admin
username cisco passphrase lifetime 99999 warntime 14 gracetime 3
```

For our lab, we will turn on both HTTP and the sandbox configuration; keep in mind that they should be turned off in production:

```
nx-osv-2(config)# nxapi http port 80
nx-osv-2(config)# nxapi sandbox
```

We are now ready to look at our first NX-API example.

# NX-API examples

NX-API sandbox is a great way to play around with various commands, data formats, and even copy the Python script directly from the web page. In the last step, we turned it on for learning purposes. Again, the sandbox should be turned off in production.

Let's launch a web browser with the Nexus device's management IP and take a look at the various message formats, requests, and responses based on the CLI commands that we are already familiar with:

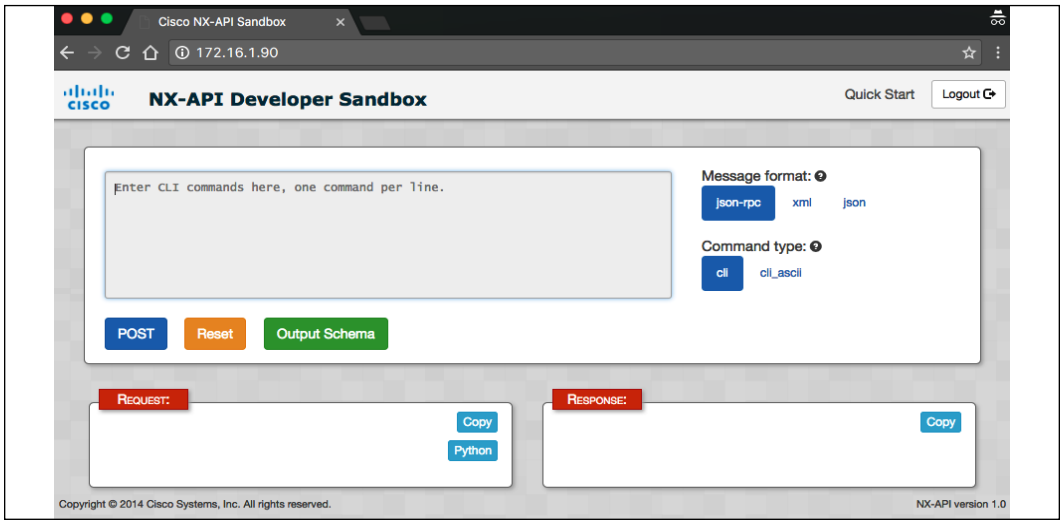



Figure 3: NX-API Developer Sandbox

In the following example, I have selected `JSON-RPC` and the `CLI` command type for the `show version` command. Click on **POST** and you will see both the **REQUEST** and **RESPONSE**:

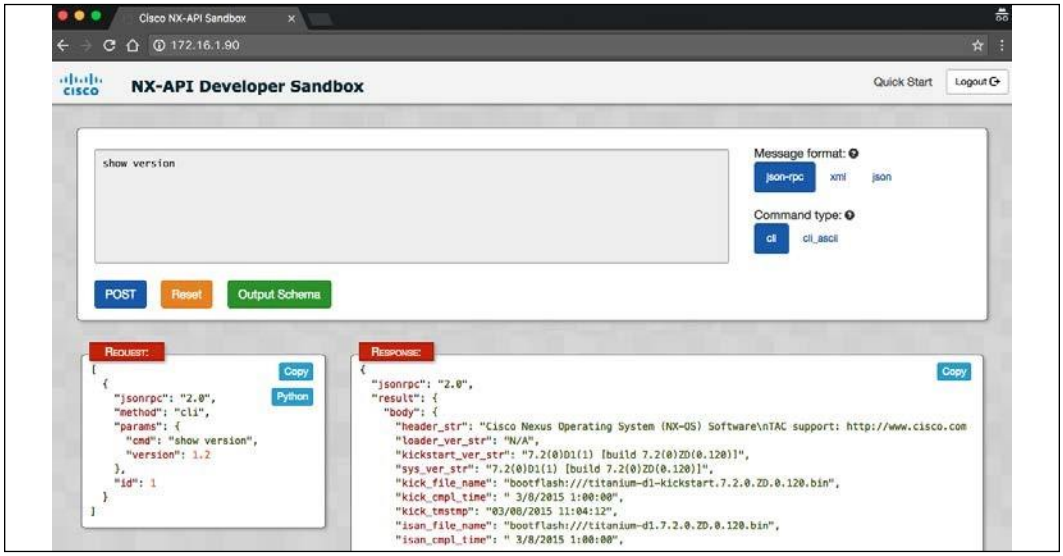



Figure 4: Cisco NX-API Developer Sandbox command output

The sandbox comes in handy if you are unsure about the supportability of the message format, or if you have questions about the response data field keys for the value you want to retrieve in your code.

In our first example, `cisco_nxapi_1.py`, we are just going to connect to the Nexus device and print out the capabilities exchanged when the connection was first made:

```
#!/usr/bin/env python3

from ncclient import manager

conn = manager.connect(
        host='172.16.1.90',
        port=22,
        username='cisco',
        password='cisco',
        hostkey_verify=False,
        device_params={'name': 'nexus'},
        look_for_keys=False
        )

for value in conn.server_capabilities:
    print(value)

conn.close_session()
```

The connection parameters of the host, port, username, and password are pretty self-explanatory. The device parameter specifies the kind of device the client is connecting to. We will see a different response in the Juniper NETCONF sections when using the `ncclient` library. The `hostkey_verify` bypasses the `known_host` requirement for SSH; if not, the host needs to be listed in the `~/.ssh/known_hosts` file. The `look_for_keys` option disables public-private key authentication, and uses the username and password combination for authentication.

Some people have reported problems with Python 3 and Paramiko to `https://github.com/paramiko/paramiko/issues/748`. In the second edition, the issue should have already been fixed by the underlying Paramiko behavior.

The output will show the XML and NETCONF supported features by this version of NX- OS:

```
(venv) $ python cisco_nxapi_1.py
urn:ietf:params:xml:ns:netconf:base:1.0
urn:ietf:params:netconf:base:1.0
urn:ietf:params:netconf:capability:validate:1.0
```

```
urn:ietf:params:netconf:capability:writable-running:1.0
urn:ietf:params:netconf:capability:url:1.0?scheme=file
urn:ietf:params:netconf:capability:rollback-on-error:1.0
urn:ietf:params:netconf:capability:candidate:1.0
urn:ietf:params:netconf:capability:confirmed-commit:1.0
```

Using ncclient and NETCONF over SSH is great because it gets us closer to the native implementation and syntax. We will use the same library later on in this book. For NX-API, we can also use HTTPS and JSON-RPC. In the earlier screenshot of **NX-API Developer Sandbox**, if you noticed, in the **REQUEST** box, there is a box labeled **Python**. If you click on it, you will be able to get an automatically converted Python script based on the request library.

The following script uses an external Python library named `requests`. `requests` is a very popular, self-proclaimed HTTP for the human library used by companies and agencies like Amazon, Google, the NSA, and others. You can find more information about it on the official site (`http://docs.python-requests.org/en/master/`).

For the `show version` example from the NX-API sandbox, the following Python script is automatically generated for us. I am pasting in the output without any modification:

```
"""
NX-API-BOT
"""
import requests
import json

"""
Modify these please
"""
url='http://YOURIP/ins'
switchuser='USERID'
switchpassword='PASSWORD'

myheaders={'content-type':'application/json-rpc'}
payload=[
  {
    "jsonrpc": "2.0",
    "method": "cli",
```

```
    "params": {
      "cmd": "show version",
      "version": 1.2
     },
     "id": 1
  }
]
response = requests.post(url,data=json.dumps(payload), headers=myheaders,
auth=(switchuser,switchpassword)).json()
```

In the `cisco_nxapi_2.py` script, you will see that I have only modified the URL, username, and password of the preceding file. The output was parsed to include only the software version. Here is the output:

```
(venv) $ python cisco_nxapi_2.py
7.3(0)D1(1)
```

The best part about using this method is that the same overall syntax structure works with both `configuration` commands as well as `show` commands. This is illustrated in the `cisco_nxapi_3.py` file, which configures the device with a new hostname with a command line. After command execution, you will see the device hostname being changed from `nx-osv-1` to `nx-osv`-new:

```
nx-osv-1-new# sh run | i hostname
hostname nx-osv-1-new
```

For multiline configuration, you can use the ID field to specify the order of operations. This is shown in `cisco_nxapi_4.py`. The following payload was listed for changing the description of the interface Ethernet 2/12 in the interface configuration mode:

```
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "interface ethernet 2/12",
      "version": 1.2
    },
    "id": 1
  },
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "description foo-bar",
```

```
      "version": 1.2
    },
    "id": 2
  },
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "end",
      "version": 1.2
    },
    "id": 3
  },
  {
    "jsonrpc": "2.0",
    "method": "cli",
    "params": {
      "cmd": "copy run start",
      "version": 1.2
    },

    "id": 4
    }
]
```

We can verify the result of the previous configuration script by looking at the running configuration of the Nexus device:

```
hostname nx-osv-1-new
...
interface Ethernet2/12
description foo-bar
shutdown
no switchport
mac-address 0000.0000.002f
```

In the next section, we will look at some examples for Cisco NETCONF and the YANG model.

# The Cisco YANG model

Earlier in this chapter, we looked at the possibility of expressing the network by using the data modeling language YANG. Let's look into it a little bit more with examples.

First off, we should know that the YANG model only defines the type of schema sent over the NETCONF protocol without dictating what the data should be. Secondly, it is worth pointing out that NETCONF exists as a standalone protocol, as we saw in the NX-API section. YANG, being relatively new, has a spotty supportability across vendors and product lines. For example, if we run a capability exchange script for a Cisco CSR 1000v running IOS-XE, we can see the different YANG model supported:

```
urn:cisco:params:xml:ns:yang:cisco-virtual-service?module=cisco- virtual-service&revision=2015-04-09

http://tail-f.com/ns/mibs/SNMP-NOTIFICATION-MIB/200210140000Z?module=SNMP-NOTIFICATION-MIB&revision=2002-10-14

urn:ietf:params:xml:ns:yang:iana-crypt-hash?module=iana-crypt-hash&revision=2014-04-04&features=crypt-hash-sha-512,crypt-hash-sha-256,crypt-hash-md5

urn:ietf:params:xml:ns:yang:smiv2:TUNNEL-MIB?module=TUNNEL-MIB&revision=2005-05-16

urn:ietf:params:xml:ns:yang:smiv2:CISCO-IP-URPF-MIB?module=CISCO-IP-URPF-MIB&revision=2011-12-29

urn:ietf:params:xml:ns:yang:smiv2:ENTITY-STATE-MIB?module=ENTITY-STATE-MIB&revision=2005-11-22

urn:ietf:params:xml:ns:yang:smiv2:IANAifType-MIB?module=IANAifType-MIB&revision=2006-03-31

<omitted>
```

Compare this to the output that we saw for NX-OS. Clearly, IOS-XE supports more YANG model features than NX-OS.

Industry-wide network data modeling, when supported, is clearly something that can be used across your devices, which is beneficial for network automation. However, given the uneven support of vendors and products, it is not yet mature enough to be used exclusively for the production network, in my opinion. Please take a look at the `cisco_yang_1.py` script for Cisco APIC-EM controller that shows how to parse out the NETCONF XML output with YANG filters called `urn:ietf:params:xml:ns:yang:ietf-interfaces` as a starting point to see the existing tag overlay.

You can check the latest vendor support on the YANG GitHub project page (`https://github.com/YangModels/yang/tree/master/vendor`).

# The Cisco ACI and APIC-EM

The Cisco ACI is meant to provide a centralized approach to all of the network components. In the data center context, it means that the centralized controller is aware of and manages the spine, leaf, and top of rack switches, as well as all the network service functions. This can be done through GUI, CLI, or API. Some might argue that the ACI is Cisco's answer to broader controller-based, software-defined networking.

One of the somewhat confusing points for ACI is the difference between ACI and APIC-EM. In short, ACI focuses on data center operations while APIC-EM focuses on enterprise modules. Both offer a centralized view and control of the network components, but each has its own focus and share of tool sets. For example, it is rare to see any major data center deploy a customer-facing wireless infrastructure, but a wireless network is a crucial part of enterprises today. Another example would be the different approaches to network security. While security is important in any network, in the data center environment, lots of security policies are pushed to the edge node on the server for scalability. In enterprise security, policies are somewhat shared between the network devices and servers.

Unlike NETCONF RPC, ACI API follows the REST model to use the HTTP verbs (`GET`, `POST`, and `DELETE`) to specify the operation that's intended.

We can look at the `cisco_apic_em_1.py` file, which is a modified version of the Cisco sample code on `lab2-1-get-network-device- list.py` (`https://github.com/CiscoDevNet/apicem-1.3-LL-sample-codes/blob/master/basic-labs/lab2-1-get-network-device-list.py`). The code illustrates the general workflow to interact with ACI and APIC-EM controllers.

The abbreviated version without comments and spaces are listed in the following section.

The first function named `getTicket()` uses HTTPS POST on the controller with the path of `/api/v1/ticket` with a username and password embedded in the header. This function will return the parsed response for a ticket that is only valid for a limited time:

```
def getTicket():
    url = "https://" + controller + "/api/v1/ticket"
    payload = {"username":"usernae","password":"password"}
    header = {"content-type": "application/json"}
    response= requests.post(url,data=json.dumps(payload),
headers=header, verify=False)
    r_json=response.json()
    ticket = r_json["response"]["serviceTicket"]
    return ticket
```

The second function then calls another path called `/api/v1/network-devices` with the newly acquired ticket embedded in the header, then parses the results:

```
    url = "https://" + controller + "/api/v1/network-device"
    header = {"content-type": "application/json", "X-Auth-Token":ticket}
```

This is a pretty common workflow for API interactions. The client will authenticate itself with the server in the first request and receive a time-based token. This token will be used in subsequent requests and will be served as a proof of authentication.

The output displays both the raw JSON response output as well as a parsed table. A partial output when executed against a DevNet lab controller is shown here:

```
Network Devices =
 {
  "version": "1.0",
  "response": [
  {
   "reachabilityStatus": "Unreachable",
   "id": "8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7",
   "platformId": "WS-C2960C-8PC-L",
 <omitted> "lineCardId": null,
  "family": "Wireless Controller",
  "interfaceCount": "12",
  "upTime": "497 days, 2:27:52.95"
  }
 ]
}
8dbd8068-1091-4cde-8cf5-d1b58dc5c9c7 Cisco Catalyst 2960-C Series
Switches
cd6d9b24-839b-4d58-adfe-3fdf781e1782 Cisco 3500I Series Unified Access
Points
<omitted>
55450140-de19-47b5-ae80-bfd741b23fd9 Cisco 4400 Series Integrated
Services Routers
ae19cd21-1b26-4f58-8ccd-d265deabb6c3 Cisco 5500 Series Wireless LAN
Controllers
```

As you can see, we only query a single controller device, but we are able to get a high-level view of all the network devices that the controller is aware of. In our output, the Catalyst 2960-C switch, 3500 Access Points, 4400 ISR router, and 5500 Wireless Controller can all be explored further. The downside is, of course, that the ACI controller only supports Cisco devices at this time.

**Cisco IOS-XE**

For the most part, Cisco IOS-XE scripts are functionally similar to scripts we have written for NX-OS. However, IOS-XE does have additional features that can benefit Python network programmability, such as on-box Python and a guest shell, `https://developer.cisco.com/docs/ios-xe/#!on-box-python-and-guestshell-quick-start-guide/onbox-python`.

Similar to the ACI controller, Cisco Meraki is a centrally-managed host that has visibility for multiple wired and wireless networks. Unlike the ACI controller, Meraki is cloud-based so is hosted outside of the on-premise location. Let us take a look at some of the Cisco Meraki features and examples in the next section.

# Cisco Meraki controller

Cisco Meraki is a cloud-based Wi-Fi centralized controller that simplifies IT management of devices. The approach is very similar to APIC with the exception that the controller is in a cloud-based public URL. The user typically receives the API key via the GUI, then it can be used in a Python script to retrieve the organization ID:

```
#!/usr/bin/env python3
import requests
import pprint

myheaders={'X-Cisco-Meraki-API-Key': <skip>}
url = 'https://dashboard.meraki.com/api/v0/organizations'
response = requests.get(url, headers=myheaders, verify=False)
pprint.pprint(response.json())
```

Let us execute the preceding script:

```
(venv) $ python cisco_meraki_1.py
 [{'id': '681155',
  'name': 'DeLab',
  'url': 'https://n6.meraki.com/o/49Gm_c/manage/organization/overview'},
```

```
{'id': '865776',
 'name': 'Cisco Live US 2019',
 'url': 'https://n22.meraki.com/o/CVQqTb/manage/organization/overview'},
{'id': '549236',
 'name': 'DevNet Sandbox',
 'url': 'https://n149.meraki.com/o/t35Mb/manage/organization/overview'},
{'id': '52636',
 'name': 'Forest City - Other',
 'url': 'https://n42.meraki.com/o/E_utnd/manage/organization/overview'}]
```

From there, the organization ID can be used to further retrieve information such as the inventory, network information, and so on:

```
#!/usr/bin/env python3
import requests
import pprint

myheaders={'X-Cisco-Meraki-API-Key': <skip>}
orgId = '549236'
url = 'https://dashboard.meraki.com/api/v0/organizations/' + orgId +
'/networks'
response = requests.get(url, headers=myheaders, verify=False)
pprint.pprint(response.json())
```

```
(venv) $ python cisco_meraki_2.py
<skip>
[{'disableMyMerakiCom': False,
  'disableRemoteStatusPage': True,
  'id': 'L_646829496481099586',
  'name': 'DevNet Always On Read Only',
  'organizationId': '549236',
  'productTypes': ['appliance', 'switch'],
  'tags': ' Sandbox ',
  'timeZone': 'America/Los_Angeles',
  'type': 'combined'},
 {'disableMyMerakiCom': False,
  'disableRemoteStatusPage': True,
  'id': 'N_646829496481152899',
  'name': 'test - mx65',
  'organizationId': '549236',
  'productTypes': ['appliance'],
  'tags': None,
  'timeZone': 'America/Los_Angeles',
```

```
  'type': 'appliance'},
<skip>
```

If you do not have a Meraki lab device, you can use the free DevNet lab located at, `https://developer.cisco.com/learning/tracks/meraki`, as I have done for this section.

We have seen examples of Cisco devices using NX-API, ACI, and the Meraki controller. In the next section, let us take a look at some of the Python examples working with Juniper Networks devices.

# The Python API for Juniper Networks

Juniper Networks have always been a favorite among the service provider crowd. If we take a step back and look at the service provider vertical, it would make sense that automating network equipment is on the top of their list of requirements. Before the dawn of cloud-scale data centers, service providers were the ones with the most network equipment. A typical enterprise network might only have a few redundant internet connections at the corporate headquarters with a few hub-and-spoke remote sites connected back to the HQ using the service provider's private **multiprotocol label switching** (**MPLS**) network. But to a service provider, they are the ones who need to build, provision, manage, and troubleshoot the connections and the underlying networks. They make their money by selling the bandwidth along with value-added managed services. It would make sense for the service providers to invest in automation to use the least amount of engineering hours to keep the network humming along. In their use case, network automation is the key to their competitive advantage.

In my opinion, the difference between a service provider's network needs compared to a cloud data center is that, traditionally, service providers aggregate more services into a single device. A good example would be MPLS, which almost all major service providers provide but rarely adapt in the enterprise or data center networks. Juniper, as they have been very successful, has identified this need of network programmability and excelled at fulfilling the service provider requirements of automating. Let's take a look at some of Juniper's automation APIs.

# Juniper and NETCONF

NETCONF is an IETF standard, which was first published in 2006 as `RFC 4741` and later revised in `RFC 6241`. Juniper Networks contributed heavily to both of the RFC standards. In fact, Juniper was the sole author of `RFC 4741`. It makes sense that Juniper devices fully support NETCONF, and it serves as the underlying layer for most of its automation tools and frameworks. Some of the main characteristics of NETCONF include the following:

1. It uses **extensible markup language** (**XML**) for data encoding.
2. It uses **remote procedure calls** (**RPC**), therefore in the case of HTTP(s) as the transport, the URL endpoint is identical while the operation intended is specified in the body of the request.
3. It is conceptually based on layers from top to bottom. The layers include the content, operations, messages, and transport:

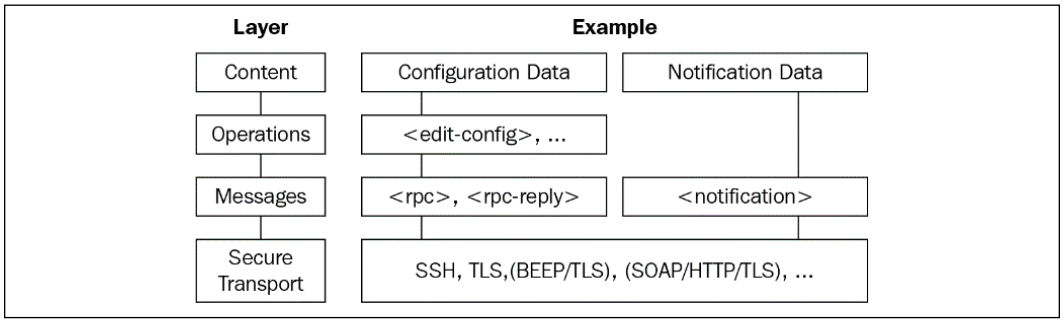



Figure 5: NETCONF model

Juniper Networks provide an extensive NETCONF XML management protocol developer guide (`https://www.juniper.net/techpubs/en_US/junos13.2/information-products/pathway-pages/netconf-guide/netconf.html#overview`) in their technical library. Let's take a look at its usage.

## Device preparation

In order to start using NETCONF, let's create a separate user as well as turning on the required services:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password "$1$0EkA.
XVf$cm80A0GC2dgSWJIYWv7Pt1"
set system services ssh
```

```
set system services telnet
set system services netconf ssh port 830
```

For the Juniper device lab, I am using an older, unsupported platform called **JunOS Olive**. It is solely used for lab purposes. You can use your favorite search engine to find out some interesting facts and history about Juniper Olive.

On the Juniper device, you can always take a look at the configuration either in a flat file or in XML format. The `flat` file comes in handy when you need to specify a one-liner command to make configuration changes:

```
netconf@foo> show configuration | display set
set version 12.1R1.9
set system host-name foo set system domain-name bar
<omitted>
```

The XML format comes in handy at times when you need to see the XML structure of the configuration:

```
netconf@foo> show configuration | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1R1/junos">
<configuration junos:commit-seconds="1485561328" junos:commit-
localtime="2017-01-27 23:55:28 UTC" junos:commit-user="netconf">
<version>12.1R1.9</version>
<system>
<host-name>foo</host-name>
<domain-name>bar</domain-name>
```

We installed the necessary Linux libraries and the `ncclient` Python library in *Lab software installation and device preparation* within *Cisco NX-API* section. If you have not done so, refer back to that section and install the necessary packages.

We are now ready to look at our first Juniper NETCONF example.

# Juniper NETCONF examples

We will use a pretty straightforward example to execute `show version`. We will name this file `junos_netconf_1.py`:

```
#!/usr/bin/env python3
```

```
from ncclient import manager

conn = manager.connect(
    host='192.168.24.252',
    port='830',
    username='netconf',
    password='juniper!',
    timeout=10,
    device_params={'name':'junos'},
    hostkey_verify=False)

result = conn.command('show version', format='text')
print(result.xpath('output')[0].text)
conn.close_session()
```

All the fields in the script should be pretty self-explanatory, with the exception of `device_params`. Starting with ncclient 0.4.1, the device handler was added to specify different vendors or platforms. For example, the name can be Juniper, CSR, Nexus, or Huawei. We also added `hostkey_verify=False` because we are using a self-signed certificate from the Juniper device.

The returned output is `rpc-reply` encoded in XML with an output element:

```
<rpc-reply message-id="urn:uuid:7d9280eb-1384-45fe-be48- b7cd14ccf2b7">
<output>
Hostname: foo
Model: olive
JUNOS Base OS boot [12.1R1.9]
JUNOS Base OS Software Suite [12.1R1.9]
<omitted>
JUNOS Runtime Software Suite [12.1R1.9] JUNOS Routing Software Suite
[12.1R1.9]
</output>
</rpc-reply>
```

We can parse the XML output to just include the output text:

```
print(result.xpath('output')[0].text)
```

In `junos_netconf_2.py`, we will make configuration changes to the device. We will start with some new imports for constructing new XML elements and the connection manager object:

```
#!/usr/bin/env python3
```

```
from ncclient import manager
from ncclient.xml_ import new_ele, sub_ele

conn = manager.connect(host='192.168.24.252', port='830',
username='netconf', password='juniper!', timeout=10, device_
params={'name':'junos'}, hostkey_verify=False)
```

We will lock the configuration and make configuration changes:

```
# lock configuration and make configuration changes conn.lock()

# build configuration
config = new_ele('system')
sub_ele(config, 'host-name').text = 'master'
sub_ele(config, 'domain-name').text = 'python'
```

Under the build configuration section, we create a new element of `system` with sub-elements of `host-name` and `domain-name`. If you were wondering about the hierarchy structure, you can see from the XML display that the node structure with `system` is the parent of `host-name` and `domain-name`:

```
<system>
    <host-name>foo</host-name>
    <domain-name>bar</domain-name>
...
</system>
```

After the configuration is built, the script will push the configuration and commit the configuration changes. These are the normal best practice steps (`lock`, `configure`, `unlock`, `commit`) for Juniper configuration changes:

```
# send, validate, and commit config conn.load_
configuration(config=config)
conn.validate()
commit_config = conn.commit()
print(commit_config.tostring)

# unlock config
conn.unlock()

# close session
conn.close_session()
```

Overall, the NETCONF steps map pretty well to what we would have done in the CLI steps. Please take a look at the `junos_netconf_3.py` script for a more reusable code. The following example combines the step-by-step example with a few Python functions:

```
# make a connection object
def connect(host, port, user, password):
    connection = manager.connect(host=host, port=port,
        username=user, password=password, timeout=10,
        device_params={'name':'junos'}, hostkey_verify=False)
    return connection

# execute show commands
def show_cmds(conn, cmd):
    result = conn.command(cmd, format='text')
    return result

# push out configuration
def config_cmds(conn, config):
    conn.lock()
    conn.load_configuration(config=config)
    commit_config = conn.commit()
    return commit_config.tostring
```

This file can be executed by itself, or it can be imported to be used by other Python scripts.

Juniper also provides a Python library to be used with their devices, called PyEZ. We will take a look at a few examples of using the library in the following section.

# Juniper PyEZ for developers

**PyEZ** is a high-level Python library implementation that integrates better with your existing Python code. By utilizing the Python API that wraps around the underlying configuration, you can perform common operations and configuration tasks without extensive knowledge of the Junos CLI.

Juniper maintains a comprehensive Junos PyEZ developer guide at `https://www.juniper.net/techpubs/en_US/junos-pyez1.0/information-products/pathway-pages/junos-pyez-developer-guide.html#configuration` on their technical library. If you are interested in using PyEZ, I would highly recommend at least a glance through the various topics in the guide.

# Installation and preparation

The installation instructions for each of the operating systems can be found on the *Installing Junos PyEZ* (`https://www.juniper.net/techpubs/en_US/junos-pyez1.0/topics/task/installation/junos-pyez-server-installing.html`) page. We will show the installation instructions for Ubuntu 18.04.

The following are some dependency packages, many of which should already be on the host from running previous examples:

```
(venv) $ sudo apt-get install -y python3-pip python3-dev libxml2-dev
libxslt1-dev libssl-dev libffi-dev
```

`PyEZ` packages can be installed via `pip`.

```
(venv) $ pip install junos-eznc
```

On the Juniper device, NETCONF needs to be configured as the underlying XML API for PyEZ:

```
set system services netconf ssh port 830
```

For user authentication, we can either use password authentication or an SSH key pair. Creating the local user is straightforward:

```
set system login user netconf uid 2001
set system login user netconf class super-user
set system login user netconf authentication encrypted-password "$1$0EkA.
XVf$cm80A0GC2dgSWJIYWv7Pt1"
```

For the `ssh` key authentication, first, generate the key pair on your management host if you have not done so for *Chapter 2*, *Low-Level Network Device Interactions*:

```
$ ssh-keygen -t rsa
```

By default, the public key will be called `id_rsa.pub` under `~/.ssh/`, while the private key will be named `id_rsa` under the same directory. Treat the private key like a password that you never share. The public key can be freely distributed. In our use case, we will copy the public key to the `/tmp` directory and enable the Python 3 HTTP server module to create a reachable URL:

```
(venv) $ cp ~/.ssh/id_rsa.pub /tmp
(venv) $ cd /tmp
(venv) $ python3 -m http.server
(venv) Serving HTTP on 0.0.0.0 port 8000 ...
```

For Python 2, use `python -m SimpleHTTPServer` instead.

From the Juniper device, we can create the user and associate the public key by downloading the public key from the Python 3 web server:

```
netconf@foo# set system login user echou class super-user authentication
load-key-file http://<management host ip>:8000/id_rsa.pub
/var/home/netconf/...transferring.file........100% of 394 B 2482 kBps
```

Now, if we try to `ssh` with the private key from the management station, the user will be automatically authenticated:

```
(venv) $ ssh -i ~/.ssh/id_rsa <Juniper device ip>
--- JUNOS 12.1R1.9 built 2012-03-24 12:52:33 UTC
echou@foo>
```

Let's make sure that both of the authentication methods work with PyEZ. Let's try the username and password combination:

```
>>> from jnpr.junos import Device
>>> dev = Device(host='<Juniper device ip, in our case 192.168.24.252>',
user='netconf', password='juniper!')
>>> dev.open()
Device(192.168.24.252)
>>> dev.facts
{'serialnumber': '', 'personality': 'UNKNOWN', 'model': 'olive', 'ifd_
style': 'CLASSIC', '2RE': False, 'HOME': '/var/home/juniper', 'version_
info': junos.version_info(major=(12, 1), type=R, minor=1, build=9),
'switch_style': 'NONE', 'fqdn': 'foo.bar', 'hostname': 'foo', 'version':
```

```
'12.1R1.9', 'domain': 'bar', 'vc_capable': False}
>>> dev.close()
```

We can also try to use the SSH key authentication:

```
>>> from jnpr.junos import Device
>>> dev1 = Device(host='192.168.24.252', user='echou', ssh_private_key_
file='/home/echou/.ssh/id_rsa')
>>> dev1.open()
Device(192.168.24.252)
>>> dev1.facts
{'HOME': '/var/home/echou', 'model': 'olive', 'hostname': 'foo', 'switch_
style': 'NONE', 'personality': 'UNKNOWN', '2RE': False, 'domain': 'bar',
'vc_capable': False, 'version': '12.1R1.9', 'serialnumber': '', 'fqdn':
'foo.bar', 'ifd_style': 'CLASSIC', 'version_info': junos.version_
info(major=(12, 1), type=R, minor=1, build=9)}
>>> dev1.close()
```

Great! We are now ready to look at some examples for PyEZ.

# PyEZ examples

In the previous interactive prompt, we already saw that when the device connects, the object automatically retrieves a few facts about the device. In our first example, `junos_pyez_1.py`, we were connecting to the device and executing an RPC call for `show interface em1`:

```
#!/usr/bin/env python3
from jnpr.junos import Device
import xml.etree.ElementTree as ET
import pprint

dev = Device(host='192.168.24.252', user='juniper', passwd='juniper!')

try:
    dev.open()
except Exception as err:
    print(err)
    sys.exit(1)

result = dev.rpc.get_interface_information(interface_name='em1',
terse=True)
pprint.pprint(ET.tostring(result))

dev.close()
```

The `Device` class has an `rpc` property that includes all operational commands. This is pretty awesome because there is no slippage between what we can do in CLI versus API. The catch is that we need to find out the corresponding `xml rpc` element tag for the CLI command. In our first example, how do we know `show interface em1` equates to `get_interface_information`? We have three ways of finding out this information:

1. We can reference the *Junos XML API Operational Developer Reference*
2. We can use the CLI and display the XML RPC equivalent and replace the dash (-) between the words with an underscore (_)
3. We can also do this programmatically by using the PyEZ library

I typically use the second option to get the output directly:

```
netconf@foo> show interfaces em1 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1R1/junos">
 <rpc>
 <get-interface-information>
 <interface-name>em1</interface-name>
 </get-interface-information>
 </rpc>
 <cli>
 <banner></banner>
 </cli>
</rpc-reply>
```

Here is an example of using PyEZ programmatically (the third option):

```
>>> dev1.display_xml_rpc('show interfaces em1', format='text')
'<get-interface-information>/n <interface-name>em1</interface- name>/n</
get-interface-information>/n'
```

Of course, we can make configuration changes as well. In the `junos_pyez_2.py` configuration example, we will import an additional `Config()` method from PyEZ:

```
#!/usr/bin/env python3
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
```

We will utilize the same block to connect to a device:

```
dev = Device(host='192.168.24.252', user='juniper',
    passwd='juniper!')
```

```
    try:
        dev.open()
    except Exception as err:
        print(err)
        sys.exit(1)
```

The `new Config()` method will load the XML data and make the configuration changes:

```
config_change = """
<system>
  <host-name>master</host-name>
  <domain-name>python</domain-name>
</system>
"""
cu = Config(dev)
cu.lock()
cu.load(config_change)
cu.commit()
cu.unlock()

dev.close()
```

The PyEZ examples are simple by design. Hopefully, they demonstrate the ways you can leverage PyEZ for your Junos automation needs. In the following example, let's take a look at how we can work with Arista network devices using Python libraries.

# The Arista Python API

**Arista Networks** have always been focused on large-scale data center networks. On its corporate profile page (`https://www.arista.com/en/company/company-overview`), it states the following:

> *"Arista Networks was founded to pioneer and deliver software-driven cloud networking solutions for large data center storage and computing environments."*

Notice that the statement specifically called out **large data centers**, which we know are exploding with servers, databases, and, yes, network equipment. It makes sense that automation has always been one of Arista's leading features. In fact, it has a Linux underpin behind their operating system, allowing many added benefits such as Linux commands and a built-in Python interpreter directly on the platform. From day one, Arista was open about exposing the Linux and Python features to the network operators.

Like other vendors, you can interact with Arista devices directly via eAPI, or you can choose to leverage their Python library. We will see examples of both. We will also look at Arista's integration with the Ansible framework in later chapters.

# Arista eAPI management

Arista's eAPI was first introduced in EOS 4.12 a few years ago. It transports a list of show or configuration commands over HTTP or HTTPS and responds back in JSON. An important distinction is that it is an RPC and **JSON-RPC**, instead of a pure RESTFul API that is served over HTTP or HTTPS. For all intents and purposes, the difference is that we make the request to the same URL endpoint using the same HTTP method (`POST`). But instead of using HTTP verbs (`GET`, `POST`, `PUT`, `DELETE`) to express our action, we simply state our intended action in the body of the request. In the case of eAPI, we will specify a `method` key with a `runCmds` value.

For the following examples, I am using a physical Arista switch running EOS 4.16.

## eAPI preparation

The eAPI agent on the Arista device is disabled by default, so we will need to enable it on the device before we can use it:

```
arista1(config)#management api http-commands
arista1(config-mgmt-api-http-cmds)#no shut
arista1(config-mgmt-api-http-cmds)#protocol https port 443
arista1(config-mgmt-api-http-cmds)#no protocol http
arista1(config-mgmt-api-http-cmds)#vrf management
```

As you can see, we have turned off the HTTP server and are using HTTPS as the sole transport instead. The management interfaces, by default, reside in a VRF called **management**. In my topology, I am accessing the device via the management interface; therefore, I have specified the VRF for eAPI management. You can check that API management state via the `show management api http-commands` command:

```
arista1#sh management
api http-commands Enabled: Yes
HTTPS server: running, set to use port 443 HTTP server: shutdown, set to use port 80
Local HTTP server: shutdown, no authentication, set to use port 8080
Unix Socket server: shutdown, no authentication
VRF: management
Hits: 64
```

```
Last hit: 33 seconds ago Bytes in: 8250
Bytes out: 29862
Requests: 23
Commands: 42
Duration: 7.086
seconds SSL Profile: none
QoS DSCP: 0
User Requests Bytes in Bytes out Last hit

---------- ------------ ------------ ------------ ----------
admin 23 8250 29862 33 seconds ago


URLs

----------------------------------------
Management1 : https://192.168.199.158:443


arista1#
```

After enabling the agent, you will be able to access the exploration page for eAPI by going to the device's IP address in a web browser. If you have changed the default port for access, just append it at the end. The authentication is tied into the method of authentication on the switch. We will use the username and password configured locally on the device. By default, a self-signed certificate will be used:

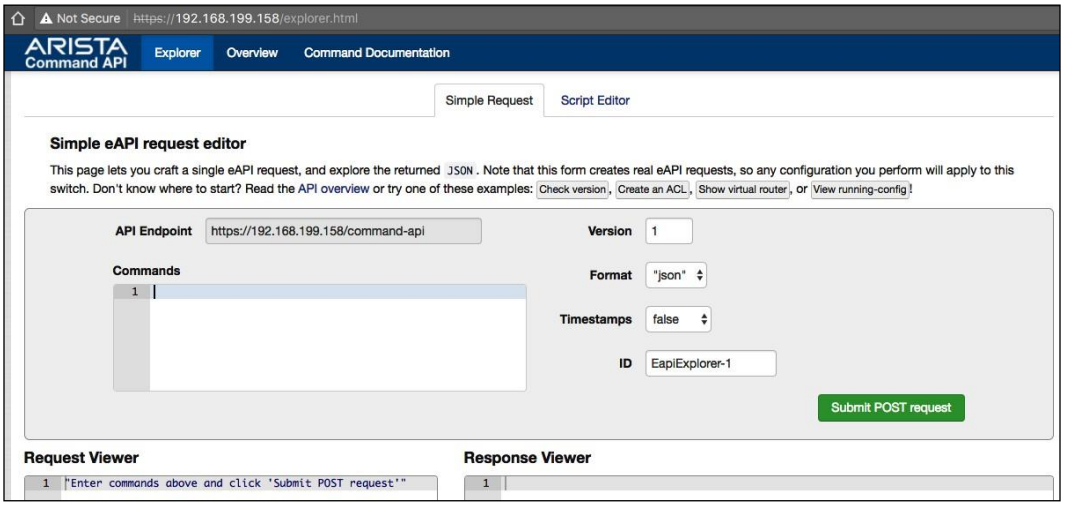



Figure 6: Arista EOS explorer

You will be taken to an explorer page where you can type in the CLI command and get a nice output for the body of your request. For example, if I want to see how to make a request body for `show version`, this is the output I will see from the explorer:

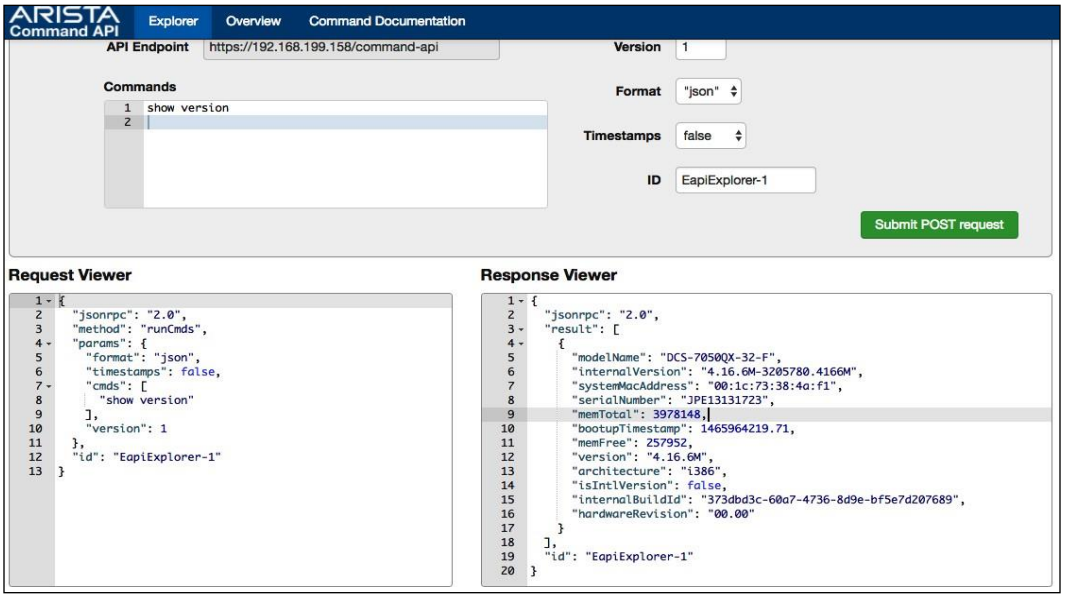



Figure 7: Arista EOS explorer viewer

The overview link will take you to the sample use and background information while the command documentation will serve as reference points for the show commands. Each of the command references will contain the returned value field name, type, and a brief description. The online reference scripts from Arista use `jsonrpclib` (`https://github.com/joshmarshall/jsonrpclib/`), which is what we will use. However, as of the time of writing this book, it has a dependency of Python 2.6+ and has not yet ported to Python 3; therefore, we will use Python 2.7 for these examples.

By the time you read this book, there might be an updated status. Please read the GitHub pull request (`https://github.com/joshmarshall/jsonrpclib/issues/38`) and the GitHub README (`https://github.com/joshmarshall/jsonrpclib/`) for the latest status.

Installation is straightforward using `easy_install` or `pip`:

```
(venv) $ pip install jsonrpclib
```

# eAPI examples

We can then write a simple program called `eapi_1.py` to look at the response text:

```
#!/usr/bin/python2

from __future__ import print_function
from jsonrpclib import Server
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
switch = Server("https://admin:arista@192.168.199.158/command-api")
response = switch.runCmds( 1, [ "show version" ] )
print('Serial Number: ' + response[0]['serialNumber'])
```

Note that, since this is Python 2, in the script, I used the `from future import print_function` to make future migration easier. The `ssl`-related lines are for Python version > 2.7.9. For more information, please see: `https://www.python.org/dev/peps/pep-0476/`.

This is the response I received from the previous `runCmds()` method:

```
[{u'memTotal': 3978148, u'internalVersion': u'4.16.6M- 3205780.4166M',
u'serialNumber': u'<omitted>', u'systemMacAddress': u'<omitted>',
u'bootupTimestamp': 1465964219.71, u'memFree': 277832, u'version':
u'4.16.6M', u'modelName': u'DCS-7050QX-32-F', u'isIntlVersion':
False, u'internalBuildId': u'373dbd3c-60a7-4736-8d9e-bf5e7d207689',
u'hardwareRevision': u'00.00', u'architecture': u'i386'}]
```

As you can see, the result is a list containing one dictionary item. If we need to grab the serial number, we can simply reference the item number and the key:

```
print('Serial Number: ' + response[0]['serialNumber'])
```

The output will contain only the serial number:

```
$ python eapi_1.py
Serial Number: <omitted>
```

To be more familiar with the command reference, I recommend that you click on the **Command Documentation** link on the eAPI page, and compare your output with the output of **show version** in the documentation.

As noted earlier, unlike REST, the JSON-RPC client uses the same URL endpoint for calling the server resources. You can see from the previous example that the `runCmds()` method contains a list of commands. For the execution of configuration commands, you can follow the same framework, and configure the device via a list of commands.

Here is an example of configuration commands named `eapi_2.py`. In our example, we wrote a function that takes the `switch` object and the list of commands as attributes:

```
#!/usr/bin/python2

from __future__ import print_function
from jsonrpclib import Server
import ssl, pprint

ssl._create_default_https_context = ssl._create_unverified_context

# Run Arista commands thru eAPI
def runAristaCommands(switch_object, list_of_commands):
    response = switch_object.runCmds(1, list_of_commands)
    return response


switch = Server("https://admin:arista@192.168.199.158/command-api")

commands = ["enable", "configure", "interface ethernet 1/3",
"switchport access vlan 100", "end", "write memory"]

response = runAristaCommands(switch, commands)
pprint.pprint(response)
```

Here is the output of the command's execution:

```
$ python2 eapi_2.py
[{}, {}, {}, {}, {}, {u'messages': [u'Copy completed successfully.']}]
```

Now, do a quick check on the `switch` to verify the command's execution:

```
arista1#sh run int eth 1/3
interface Ethernet1/3
    switchport access vlan 100
arista1#
```

Overall, eAPI is fairly straightforward and simple to use. Most programming languages have libraries similar to `jsonrpclib`, which abstracts away JSON-RPC internals. With a few commands, you can start integrating Arista EOS automation into your network.

# The Arista Pyeapi library

The Python client Pyeapi (`http://pyeapi.readthedocs.io/en/master/index.html`) library is a native Python library wrapper around eAPI. It provides a set of bindings to configure Arista EOS nodes. Why do we need Pyeapi when we already have eAPI? The answer is 'it depends.' Picking between Pyeapi versus eAPI is mostly a judgment call if you are already using Python for automation.

However, if you are in a non-Python environment, eAPI is probably the way to go. From our examples, you can see that the only requirement of eAPI is a JSON-RPC capable client. Thus, it is compatible with most programming languages. When I first started out in the field, Perl was the dominant language for scripting and network automation. There are still many enterprises that rely on Perl scripts as their primary automation tool. If you're in a situation where the company has already invested a ton of resources and the code base is in another language than Python, eAPI with JSON-RPC would be a good bet.

However, for those of us who prefer to code in Python, a native Python library means a more natural feeling in writing our code. It certainly makes extending a Python program to support the EOS node easier. It also makes keeping up with the latest changes in Python easier. For example, we can use Python 3 with Pyeapi!

At the time of writing this book, Python 3 (3.4+) support is officially a work in progress, as stated in the documentation (`http://pyeapi.readthedocs.io/en/master/requirements.html`). Please check the documentation for more details.

# Pyeapi installation

Installation is straightforward with `pip`:

```
(venv) $ pip install pyeapi
```

Note that pip will also install the `netaddr` library as it is part of the stated requirements (`http://pyeapi.readthedocs.io/en/master/requirements.html`) for Pyeapi.

By default, the Pyeapi client will look for an INI-style hidden (with a period in front) file called `eapi.conf` in your home directory. You can override this behavior by specifying the `eapi.conf` file path. It is generally a good idea to separate your connection credential and lock it down from the script itself. You can check out the Arista Pyeapi documentation (`http://pyeapi.readthedocs.io/en/master/configfile.html#configfile`) for the fields contained in the file.

Here is the file I am using in the lab:

```
cat ~/.eapi.conf
[connection:Arista1]
host: 192.168.199.158
username: admin
password: arista
transport: https
```

The first line, [`connection:Arista1`], contains the name that we will use in our Pyeapi connection; the rest of the fields should be pretty self-explanatory. You can lock down the file to be read-only for the user using this file:

```
$ chmod 400 ~/.eapi.conf
$ ls -l ~/.eapi.conf
-r-------- 1 echou echou 94 Jan 27 18:15 /home/echou/.eapi.conf
```

Now that Pyeapi is installed, let's get into some examples.

# Pyeapi examples

Now, we are ready to take a look around Pyeapi's usage. Let's start by connecting to the EOS node by creating an object in the interactive Python shell:

```
>>> import pyeapi
>>> arista1 = pyeapi.connect_to('Arista1')
```

We can execute `show` commands to the node and receive the output:

```
>>> import pprint
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
'encoding': 'json',
'result': {'fqdn': 'arista1', 'hostname': 'arista1'}}]
```

The configuration field can be either a single command or a list of commands using the `config()` method:

```
>>> arista1.config('hostname arista1-new')
[{}]
>>> pprint.pprint(arista1.enable('show hostname'))
[{'command': 'show hostname',
 'encoding': 'json',
```

```
 'result': {'fqdn': 'arista1-new', 'hostname': 'arista1-new'}}]
>>> arista1.config(['interface ethernet 1/3', 'description my_link'])
[{}, {}]
```

Note that command abbreviations (`show run` versus `show running-config`) and some extensions will not work:

```
>>> pprint.pprint(arista1.enable('show run'))
Traceback (most recent call last):
...
File "/usr/local/lib/python3.5/dist-packages/pyeapi/eapilib.py", line
396, in send
raise CommandError(code, msg, command_error=err, output=out) pyeapi.
eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show run' failed:
invalid command [incomplete token (at token 1: 'run')]
>>>
>>> pprint.pprint(arista1.enable('show running-config interface ethernet
1/3'))
Traceback (most recent call last):
...
pyeapi.eapilib.CommandError: Error [1002]: CLI command 2 of 2 'show
running-config interface ethernet 1/3' failed: invalid command
[incomplete token (at token 2: 'interface')]
```

However, you can always catch the results and get the desired value:

```
>>> result = arista1.enable('show running-config')
>>> pprint.pprint(result[0]['result']['cmds']['interface Ethernet1/3'])
{'cmds': {'description my_link': None, 'switchport access vlan 100':
None}, 'comments': []}
```

So far, we have been doing what we have been doing with eAPI for `show` and `configuration` commands. Pyeapi offers various APIs to make life easier. In the following example, we will connect to the node, call the VLAN API, and start to operate on the VLAN parameters of the device. Let's take a look:

```
>>> import pyeapi
>>> node = pyeapi.connect_to('Arista1')
>>> vlans = node.api('vlans')
>>> type(vlans)
<class 'pyeapi.api.vlans.Vlans'>
>>> dir(vlans)
[...'command_builder', 'config', 'configure', 'configure_interface',
```

```
'configure_vlan', 'create', 'default', 'delete', 'error', 'get', 'get_
block', 'getall', 'items', 'keys', 'node', 'remove_trunk_group', 'set_
name', 'set_state', 'set_trunk_groups', 'values']
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name':
'default'}}
>>> vlans.get(1)
{'vlan_id': 1, 'trunk_groups': [], 'state': 'active', 'name': 'default'}
>>> vlans.create(10) True
>>> vlans.getall()
{'1': {'vlan_id': '1', 'trunk_groups': [], 'state': 'active', 'name':
'default'}, '10': {'vlan_id': '10', 'trunk_groups': [], 'state':
'active', 'name': 'VLAN0010'}}
>>> vlans.set_name(10, 'my_vlan_10') True
```

Let's verify that VLAN 10 was created on the device:

```
arista1#sh vlan
VLAN Name Status Ports
----- ------------------------------ --------- -------------------
-----
1 default active
10 my_vlan_10 active
```

As you can see, the Python native API on the EOS object is really where Pyeapi excels beyond eAPI. It abstracts the lower-level attributes into the device object and makes the code cleaner and easier to read.

For a full list of ever-increasing Pyeapi APIs, check the official documentation (`http://pyeapi.readthedocs.io/en/master/api_modules/_list_of_modules.html`).

To round up this section, let's assume that we repeat the previous steps enough times that we would like to write another Python class to save us some work.

The `pyeapi_1.py` script is shown as follows:

```
#!/usr/bin/env python3

import pyeapi
```

```
class my_switch():

    def __init__(self, config_file_location, device):
        # loads the config file
        pyeapi.client.load_config(config_file_location)
        self.node = pyeapi.connect_to(device)
        self.hostname = self.node.enable('show hostname')[0]['result']
['hostname']
        self.running_config = self.node.enable('show running-config')

    def create_vlan(self, vlan_number, vlan_name):
        vlans = self.node.api('vlans')
        vlans.create(vlan_number)
        vlans.set_name(vlan_number, vlan_name)
```

As you can see from the script, we automatically connect to the node and set the hostname and load `running_config` upon connection. We also create a method to the class that creates VLAN by using the VLAN API. Let's try out the script in an interactive shell:

```
>>> import pyeapi_1
>>> s1 = pyeapi_1.my_switch('/tmp/.eapi.conf', 'Arista1')
>>> s1.hostname
'arista1'
>>> s1.running_config
[{'encoding': 'json', 'result': {'cmds': {'interface Ethernet27':
{'cmds':
{}, 'comments': []}, 'ip routing': None, 'interface face Ethernet29':
{'cmds': {}, 'comments': []}, 'interface Ethernet26': {'cmds': {},
'comments': []}, 'interface Ethernet24/4': h.':
<omitted>
'interface Ethernet3/1': {'cmds': {}, 'comments': []}}, 'comments': [],
'header': ['! device: arista1 (DCS-7050QX-32, EOS-4.16.6M)n!n']},
'command': 'show running-config'}]
>>> s1.create_vlan(11, 'my_vlan_11')
>>> s1.node.api('vlans').getall()
{'11': {'name': 'my_vlan_11', 'vlan_id': '11', 'trunk_groups': [],
'state':
'active'}, '10': {'name': 'my_vlan_10', 'vlan_id': '10', 'trunk_groups':
[], 'state': 'active'}, '1': {'name': 'default', 'vlan_id': '1', 'trunk_
groups': [], 'state': 'active'}}
>>>
```

We have now taken a look at Python scripts for three of the top vendors in

networking: Cisco Systems, Juniper Networks, and Arista Networks. In the next section, we will take a look at an open source network operating system that is gaining some momentum in the same space.

# VyOS example

VyOS is a fully open source network OS that runs on a wide range of hardware, virtual machines, and cloud providers (`https://vyos.io/`). Because of its open source nature, it is gaining wide support in the open source community. Many of the open source projects are using VyOS as the default platform for testing. In the last section of the chapter, we will look at a quick VyOS example.

The VyOS image can be downloaded in various formats: `https://wiki.vyos.net/wiki/Installation`. Once downloaded and initialized, we can install the Python library on our management host:

```
(venv) $ pip install vymgmt
```

The example script, `vyos_1.py`, is very simple:

```
#!/usr/bin/env python3

import vymgmt

vyos = vymgmt.Router('192.168.2.116', 'vyos', password='vyos')
vyos.login()
vyos.configure()
vyos.set("system domain-name networkautomationnerds.net")
vyos.commit()
vyos.save()
vyos.exit()
vyos.logout()
```

We can execute the script to change the system domain name:

```
(venv) $ python vyos_1.py
We can log in to the device to verify the change:
vyos@vyos:~$ show configuration | match domain
domain-name networkautomationnerds.net
```

As you can see from the example, the method we use for VyOS is pretty similar to the other examples we have seen before from proprietary vendors. This is mainly by design, as they provide an easy transition from using other vendor equipment to open source VyOS. We are getting close to the end of the chapter. There are some other libraries that are worth mentioning and should be kept an eye out for in development, which we will do in the next section.

# Other libraries

We'll finish this chapter by mentioning that there are several excellent efforts in terms of vendor-neutral libraries such as Nornir (`https://nornir.readthedocs.io/en/stable/index.html`), Netmiko (`https://github.com/ktbyers/netmiko`), and NAPALM (`https://github.com/napalm-automation/napalm`). We have seen some examples of them in the last chapter. For most of these vendor-neutral libraries, they are likely a step slower to support the latest platform or features. However, because the libraries are vendor-neutral, if you do not like vendor lock-in for your tools, then these libraries are good choices. Another benefit of using these vendor-neutral libraries is the fact that they are normally open source, so you can contribute back upstream for new features and bug fixes.

Cisco also recently released their pyATS framework (`https://developer.cisco.com/pyats/`) as well as the associated pyATS library (formerly Genie). We will take a look at pyATS and Genie in *Chapter 15*, *Test-Driven Development for Networks*.

# Summary

In this chapter, we looked at various ways to communicate and manage network devices from Cisco, Juniper, VyOS, and Arista. We looked at both direct communication with the likes of NETCONF and REST, as well as using vendor-provided libraries such as PyEZ and Pyeapi. These are different layers of abstractions, meant to provide a way to programmatically manage your network devices without human intervention.

In *Chapter 4*, *The Python Automation Framework – Ansible Basics*, we will take a look at a higher-level of vendor-neutral abstraction framework called Ansible. Ansible is an open source, general-purpose automation tool written in Python. It can be used to automate servers, network devices, load balancers, and much more. Of course, for our purpose, we will focus on using this automation framework for network devices.