EXPERT INSIGHT

python™

Mastering Python Networking

Utilize Python packages and frameworks for network automation, monitoring, cloud, and management

Fourth Edition



Eric Chou

packt

Mastering Python Networking

Fourth Edition

Utilize Python packages and frameworks for network automation, monitoring, cloud, and management

Eric Chou

BIRMINGHAM—MUMBAI

"Python" and the Python Logo are trademarks of the Python Software Foundation.

Mastering Python Networking

Fourth Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavoured to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Aaron Tanna
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Namrata Katare
Content Development Editor: Liam Thomas Draper
Copy Editor: Safis Editing
Technical Editor: Srishty Bhardwaj
Proofreader: Safis Editing
Indexer: Rekha Nair
Presentation Designer: Ganesh Bhadwalkar
Developer Relations Marketing Executive: Meghal Patel

First published: April 2016
Second edition: August 2018
Third edition: January 2020
Fourth edition: January 2023

Production reference: 1130123

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

ISBN 978-1-80323-461-8

www.packt.com

To my wife, Joanna, who inspired me to explore.

To my children, Mikaelyn and Esmie, may you find the fire to explore the world as well.

Foreword

The book you are holding in your hands or are reading on your screen has a power that can be yours if you take time to study it. Programming can amplify your existing knowledge and skillset.

Many people are told they should learn programming and Python for their own sake. Programming skills are in demand, so you should be a programmer. That is probably good advice. But better advice would be to answer the question, "How can you take your existing expertise and leap ahead of peers by automating and extending that experience with software skills?" This book aims to do just that for network professionals. You'll learn about Python in the context of network configuration, administration, monitoring and more.

If you are tired of logging in and typing in a bunch of commands to configure your network, Python is for you. If you need to be certain the network configuration is solid and repeatable, Python is for you. If you need to monitor, in real-time, what is happening on the network, well, you guessed it, Python is for you.

You are probably in agreement about learning software skills that can be applied to network engineering. After all, terms like **Software-Defined Networking** (**SDN**) have been all the buzz in the last few years. But why Python? Maybe you should learn JavaScript or Go or some other language. Maybe you should just double down on Bash and shell scripting.

Python is well suited for network engineering for two reasons.

Firstly, as Eric will demonstrate throughout this book, there are many Python libraries (sometimes called packages) designed specifically on network engineering. With libraries such as Ansible, you can create complex network and server configurations using simple configuration files. Using Pexpect or Paramiko, you'll be able to program against remote legacy systems as if they had their own scripting API. If the gear you're configuring has an API, chances are you can use a purpose-built Python library to work with it. So clearly, Python is well-suited for the job.

Secondly, Python is special amongst programming languages. Python is what I call a *full spectrum language*. My definition of this term is that it is both a language that is incredibly easy to get started (`print ("hello world")` anyone?) and also very powerful, being the technology behind incredible software such as YouTube.

This is not normal. We have solid beginner languages for quickly building software. Visual Basic comes to mind here. So does Matlab and other commercial languages. Yet, when these are pushed too far, they fall down badly. Can you imagine Linux, Firefox, or an intensive video game created with any of these? No way.

At the other end of the spectrum, we also have very powerful languages such as C++, .NET, Java and many others. C++ is, in fact, the language used to build some Linux kernel modules and large open-source software such as Firefox to some degree. Yet, they are not beginner-friendly. You have to learn about pointers, compilers, linkers, headers, classes, accessibility (public/private), and on and on just to get started.

Python lives in both realms making it incredibly easy to be productive with just a few lines of code and simple programming concepts. Yet it grows to be the language of choice for some of the world's most significant software behind YouTube, Instagram, Reddit, and others. Microsoft chose Python as their language to implement the CLI for Azure (although you don't have to know or use Python to use their CLI of course).

So, here's the deal. Programming is a superpower. It can take your network engineering expertise and launch it into the stratosphere. Python is one of the world's fastest growing and most popular programming languages. And Python has many highly polished libraries for working with networks in many facets. Therefore, this book, *Mastering Python Networking*, combines all of these and will change the way you think about networking. Enjoy the journey.

Michael Kennedy
Portland, OR
Founder of Talk Python

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, Microsoft, and other Fortune 500 companies. Eric is passionate about network engineering, Python, and helping companies build better automation culture.

Eric is the author of other top-selling books on security, data processing, and programming. Eric holds multiple U.S. patents in IP telephony and networking. He shares his deep interest in technology through his books, online classes, speaking engagements, and contributing to open source projects.

I would like to thank the open source community members 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 made small contributions to the wonderful communities in my own way as well.

I would like to thank the Packt team for the opportunity to collaborate on the fourth edition of this book. Special thanks to the technical reviewer, Josh VanDeraa, for generously agreeing to review this book.

To my family, your constant support and encouragement make me want to be the best version of myself. I love you.

About the reviewer

Josh VanDeraa is a 20-year networking veteran who has been doing network automation for the past 7 years. He has worked in large enterprise retail, travel, managed services, and most recently, professional services industries. He has worked on networks of all sizes, bringing multiple different network automation solutions to the table to drive real value with Python, Ansible, and Python web framework solutions.

Josh is the author of *Open Source Network Management* and maintains a blog site (`https://josh-v.com`) to provide additional content to those on the web.

I'd like to thank my family first and foremost, who have allowed me the time to dive in and review the topic.

I would also like to thank Eric for taking the time to share his experiences. This is a tough thing to do and Eric has done a masterful job at putting time and real-life experiences into the book to be consumed. I love how the book has evolved over the revisions, with my first reading of the book coming in the second edition.

And lastly, I would like to thank the greater network automation community. This is a journey for all of the individuals, no matter the organization. Please join, teach, and share your experiences with the community so network automation can continue to evolve!

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



Table of Contents

Preface xxi

Chapter 1: Review of TCP/IP Protocol Suite and Python 1

An overview of the Internet 3

Servers, hosts, and network components • 4

The rise of data centers • 5

Enterprise data centers • 5

Cloud data centers • 6

Edge data centers • 7

The OSI model 8

The client-server model 10

Network protocol suites 11

The transmission control protocol • 12

Functions and characteristics of TCP • 12

TCP messages and data transfer • 12

The user datagram protocol • 13

The Internet protocol • 14

IP network address translation (NAT) and network security • 15

IP routing concepts • 15

Python language overview 16

Python versions • 17

Operating system • 17

Running a Python program • 18
Python built-in types • 19
The None type • 20
Numerics • 20
Sequences • 20
Mapping • 23
Sets • 24
Python operators • 25
Python control flow tools • 26
Python functions • 28
Python classes • 29
Python modules and packages • 30
Summary 31

Chapter 2: Low-Level Network Device Interactions 33

The challenges of the CLI 34
Constructing a Virtual lab 35
Physical devices • 36
Virtual devices • 36
Cisco modeling labs • 37
CML tips • 39
Cisco DevNet • 40
GNS3 and others • 42
Python virtual environment 43
Python pexpect library 44
Pexpect installation • 44
Pexpect overview • 45
Our first Pexpect program • 50
More Pexpect Features • 51
Pexpect and SSH • 53
Pexpect complete example • 53

The Python Paramiko library 54

Installation of Paramiko • 55

Paramiko overview • 55

First Paramiko program • 58

More Paramiko features • 59

Paramiko for servers • 59

More Paramiko examples • 61

The Netmiko library 62

The Nornir framework 64

Downsides of Pexpect and Paramiko compared to other tools • 66

Idempotent network device interaction • 67

Bad automation speeds up bad things • 67

Summary 68

Chapter 3: APIs and Intent-Driven Networking 69

Infrastructure-as-Code (IaC) 70

Intent-driven networking • 71

Screen scraping versus API structured output • 72

Data modeling for IaC • 76

YANG and NETCONF • 77

Cisco API examples 77

Cisco NX-API • 78

Lab preparation • 79

NX-API examples • 79

The Cisco YANG Model • 84

The Cisco ACI examples • 85

Cisco Meraki controller 89

The Python API for Juniper Networks 91

Juniper and NETCONF • 91

Device preparation • 92

Juniper NETCONF examples • 93

Juniper PyEZ for developers • 96

Installation and preparation • 96

PyEZ examples • 98

The Arista Python API 100

Arista eAPI management • 101

eAPI preparation • 101

eAPI examples • 104

The Arista Pyeapi library • 106

Pyeapi installation • 106

Pyeapi examples • 107

VyOS example 111

Other libraries 112

Summary 112

Chapter 4: The Python Automation Framework – Ansible 115

Ansible – A More Declarative Framework 117

Ansible Versions • 118

Our First Ansible Network Example 119

The Control Node Installation • 120

Lab Topology • 121

The Variable Files • 122

Our First Playbook • 123

The Advantages of Ansible 125

Agentless • 125

Idempotence • 126

Simple and Extensible • 127

Ansible Content Collections 127

More Ansible Network Examples 128

Inventory Nesting • 129

Ansible Conditionals • 131

Configuration Change • 133

Ansible Network Facts • 135
Ansible Loops • 137
Standard Loops • 137
Looping over Dictionaries • 139
Templates • 142
The Jinja Template Variables • 145
Jinja Template Loops • 146
Jinja Template Conditional • 147
Summary 149

Chapter 5: Docker Containers for Network Engineers 151

Docker Overview 153
Advantages of Docker • 153
Building Python applications in Docker 154
Installing Docker • 154
Useful Docker commands • 155
Building hello world • 156
Building our application • 157
Sharing Docker images • 161
Container orchestration with Docker-compose • 163
Container networking 166
Container host network • 168
Custom bridge network • 169
Other container network options • 170
Containers in the network engineering field 170
Containerlab • 171
Docker and Kubernetes 175
Summary 176

Chapter 6: Network Security with Python 177

The Lab Setup **178**

Python Scapy **185**

Installing Scapy • 186

Interactive examples • 188

Packet captures with Scapy • 190

The TCP port scan • 192

The ping collection • 196

Common attacks • 197

Scapy resources • 198

Access lists **198**

Implementing access lists with Ansible • 199

MAC access lists • 203

The Syslog search **205**

Searching with the regular expression module • 206

Other tools **208**

Private VLANs • 208

UFW with Python • 209

Further reading **210**

Summary **210**

Chapter 7: Network Monitoring with Python – Part 1 213

Lab Setup **214**

SNMP **215**

Setup • 217

PySNMP • 219

Python for Data Visualization **225**

Matplotlib • 225

Installation • 225

Matplotlib – the first example • 226

Matplotlib for SNMP results • 228

Additional Matplotlib resources • 232

Pygal • 233

Installation • 233

Pygal – the first example • 233

Pygal for SNMP results • 235

Additional Pygal resources • 237

Python for Cacti 238

Installation • 238

Python script as an input source • 240

Summary 243

Chapter 8: Network Monitoring with Python – Part 2 245

Graphviz 246

Lab setup • 247

Installation • 248

Graphviz examples • 249

Python with Graphviz examples • 251

LLDP neighbor graphing • 252

Information retrieval • 255

Python parser script • 256

Testing the playbook • 261

Flow-based monitoring 263

NetFlow parsing with Python • 264

Python Socket and Struct • 265

ntop traffic monitoring 269

Python extension for ntop • 272

sFlow • 276

SFlowtool and sFlow-RT with Python • 277

Summary 281

Chapter 9: Building Network Web Services with Python 283

Comparing Python web frameworks **285**
Flask and lab setup **288**
Introduction to Flask **289**
Flask versions • 289
Flask examples • 289
The HTTPie client • 290
URL routing • 293
URL variables • 294
URL generation • 295
The jsonify return • 296
Network resource API **297**
Flask-SQLAlchemy • 297
The network content API • 299
The devices API • 303
The device ID API • 305
Network dynamic operations **306**
Asynchronous operations • 309
Authentication and Authorization **312**
Running Flask in containers **315**
Summary **319**

Chapter 10: Introduction to Async IO 321

Asynchronous operations overview **322**
Python multiprocessing • 323
Python multithreading • 324
Python asyncio module **326**
The Scrapli project **331**
Scrapli example • 331
Scrapli async example • 334
Summary **338**

Chapter 11: AWS Cloud Networking 339

AWS setup **340**
The AWS CLI and Python SDK • 342
AWS network overview **345**
Virtual Private Cloud **353**
Route tables and route targets • 358
Automation with CloudFormation • 360
Security Groups and Network ACLs • 364
Elastic IP • 367
NAT gateways • 368
Direct Connect and VPN **369**
VPN gateways • 369
Direct Connect • 370
Network scaling services **371**
Elastic Load Balancing • 372
Route 53 DNS service • 372
CloudFront CDN services • 373
Other AWS network services **373**
Summary **374**

Chapter 12: Azure Cloud Networking 375

Azure and AWS network service comparison **376**
Azure setup **378**
Azure administration and APIs **381**
Azure service principals • 384
Python versus PowerShell • 387
Azure global infrastructure **388**
Azure virtual networks **389**
Internet access • 392
Network resource creation • 396

VNet service endpoints • 398

VNet peering • 399

VNet routing 402

Network security groups • 407

Azure VPNs 409

Azure ExpressRoute 412

Azure network load balancers 414

Other Azure network services 416

Summary 416

Chapter 13: Network Data Analysis with Elastic Stack 419

What is the Elastic Stack? 420

Lab topology 421

Elastic Stack as a service 428

First End-to-End example 430

Elasticsearch with a Python client 435

Data ingestion with Logstash 437

Data ingestion with Beats 440

Search with Elasticsearch 447

Data visualization with Kibana 452

Summary 458

Chapter 14: Working with Git 459

Content management considerations and Git 460

Introduction to Git 461

Benefits of Git • 461

Git Terminology • 462

Git and GitHub • 463

Setting up Git 464

Gitignore • 465

Git Usage Examples **466**
Git Branch **471**
GitHub Example • 473
Collaborating with Pull Requests • 479
Git with Python **482**
GitPython • 482
PyGitHub • 483
Automating Configuration Backup **485**
Collaborating with Git **487**
Summary **488**

Chapter 15: Continuous Integration with GitLab **491**

The traditional change management process **492**
Introduction to continuous integration **494**
Installing GitLab **495**
GitLab runners **499**
First GitLab example **501**
GitLab network example **509**
Summary **513**

Chapter 16: Test-Driven Development for Networks **515**

Test-driven development overview **516**
Test definitions • 517
Topology as code **518**
XML parsing example • 521
Python's unittest module • 524
More on Python testing • 528
pytest examples • 528
Writing tests for networking **531**
Testing for reachability • 532
Testing for network latency • 533

Testing for security • 534

Testing for transactions • 534

Testing for network configuration • 535

Testing for Ansible • 535

pyATS and Genie 536

Summary 544

Other Books You May Enjoy 549

Index 553

Preface

As Charles Dickens wrote in *A Tale of Two Cities, "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness."* These seemingly contradictory words perfectly describe the chaos and mood felt during a time of change and transition. We are undoubtedly experiencing a similar time with the rapid changes in the network engineering field. As software development becomes more integrated into all engineering stacks, the traditional command-line interface and vertically integrated network control methods are no longer the best ways to manage today's networks.

For network engineers, the changes we are seeing are full of excitement and opportunities and yet are challenging, particularly for those who need to quickly adapt and keep up. This book has been written to help ease the transition for networking professionals by providing a practical guide that addresses how to evolve from a traditional platform to one built on software-driven and development practices.

In this book, we use Python as the programming language of choice to master network engineering tasks. Python is an easy-to-learn, high-level programming language that can effectively complement network engineers' creativity and problem-solving skills to streamline daily operations. Python is becoming an integral part of many large-scale networks, and through this book, I hope to share with you the lessons I've learned.

Since the publication of the first three editions of this book, I have been fortunate to have had interesting and meaningful conversations with many of the book's readers. I am humbled by the success of the first three editions and took to heart the feedback I was given. In this fourth edition, I tried to incorporate many of the newer libraries, updated existing examples with the latest software and newer hardware platforms, added two new chapters, and significantly modified several chapters. I believe the changes are more reflective of today's network engineering environment.

A time of change presents great opportunities for technological advancement. The concepts and tools in this book have helped me tremendously in my career, and I hope they can do the same for you.

Who this book is for

This book is ideal for IT professionals and operations engineers who already manage groups of network devices and would like to expand their knowledge of using Python and other tools to overcome network challenges. Basic knowledge of networking and Python is recommended.

What this book covers

Chapter 1, *Review of TCP/IP Protocol Suite and Python*, reviews the fundamental technologies for internet communication today, from the OSI and client-server model to the TCP, UDP, and IP suites. The chapter will review the basics of the Python language, such as types, operators, loops, functions, and packages.

Chapter 2, *Low-Level Network Device Interactions*, uses practical examples to illustrate how to use Python to execute commands on a network device. It will also discuss the challenges of having a CLI-only interface in automation. The chapter will use the Pexpect, Paramiko, Netmiko, and Nornir libraries in the examples.

Chapter 3, *APIs and Intent-Driven Networking*, discusses the network devices that support **Application Programming Interfaces** (**APIs**) and other high-level interaction methods. It also illustrates tools that allow the abstraction of low-level tasks while focusing on the intent of the network engineers. A discussion about and examples of Cisco NX-API, Meraki, Juniper PyEZ, Arista Pyeapi, and Vyatta VyOS will appear in the chapter.

Chapter 4, *The Python Automation Framework – Ansible*, discusses the basics of Ansible. Ansible is an open source, Python-based automation framework. The framework moves one step further from APIs and focuses on declarative task intent. In this chapter, we will cover the advantages of using Ansible and its high-level architecture and see some practical examples of Ansible with network devices.

Chapter 5, *Docker Containers for Network Engineers,* explores containers and explains how Docker is the new standard in application development. In this chapter, we will introduce Docker as a tool by introducing the overall concepts and building example applications with it.

Chapter 6, *Network Security with Python*, introduces several Python tools to help you secure your network. It will discuss using Scapy for security testing, using Ansible to quickly implement access lists, and using Python for network forensic analysis.

Chapter 7, *Network Monitoring with Python – Part 1*, covers monitoring the network using various tools. The chapter contains some examples using SNMP and PySNMP for queries to obtain device information. Matplotlib and Pygal examples will be shown for graphing the results. The chapter will end with a Cacti example using a Python script as an input source.

Chapter 8, *Network Monitoring with Python – Part 2*, covers more network monitoring tools. The chapter will start with using Graphviz to graph the network from LLDP information. We will move to use examples with push-based network monitoring using NetFlow and other technologies. We will use Python to decode flow packets and ntop to visualize the results.

Chapter 9, *Building Network Web Services with Python*, shows you how to use the Python Flask web framework to create API endpoints for network automation. The network API offers benefits such as abstracting the requester from network details, consolidating and customizing operations, and providing better security by limiting the exposure of available operations.

Chapter 10, *Introduction to Async IO*, covers Async IO, the new Python 3 package that allows us to perform tasks at the same time. We will cover topics such as multiprocessing, parallelism, threading, and other concepts. We will also cover examples from the Scrapli project.

Chapter 11, *AWS Cloud Networking*, shows how we can use AWS to build a virtual network that is functional and resilient. We will cover virtual private cloud technologies such as CloudFormation, VPC routing tables, access lists, Elastic IP, NAT gateways, Direct Connect, and other related topics.

Chapter 12, *Azure Cloud Networking*, covers the network services by Azure and how to build network services with the service. We will discuss Azure VNet, Express Route and VPN, Azure network load balancers, and other related network services.

Chapter 13, *Network Data Analysis with Elastic Stack*, shows how we can use Elastic Stack as a set of tightly integrated tools to help us analyze and monitor our network. We will cover areas from installation, configuration, data import with Logstash and Beats, and searching data using Elasticsearch, to visualization with Kibana.

Chapter 14, *Working with Git*, illustrates how we can leverage Git for collaboration and code version control. Practical examples of using Git for network operations will be used in this chapter.

Chapter 15, *Continuous Integration with GitLab*, uses GitLab to automatically create operations pipelines that can save us time and increase reliability.

Chapter 16, *Test-Driven Development for Networks*, explains how to use Python's `unittest` and `pytest` to create simple tests to verify our code. We will also see examples of writing tests for our network to verify reachability, network latency, security, and network transactions.

To get the most out of this book

To get the most out of this book, some basic hands-on network operation knowledge and Python knowledge is recommended. Most of the chapters can be read in any order, except for *Chapter 4* and *Chapter 5*, which introduce base technologies that will be used later in the book. Besides the basic software and hardware tools introduced at the beginning of the book, new tools relevant to each of the chapters will be introduced in the respective chapters.

It is highly recommended to follow and practice the examples shown in your network lab.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Python-Networking-Fourth-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/D2Ttl`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The auto-config also generated `vty` access for both Telnet and SSH."

A block of code is set as follows:

```
# This is a comment
print("hello world")
```

Any command-line input or output is written as follows:

```
$ python3
Python 3.10.6 (main, Nov  2 2022, 18:53:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: " In the coming section, we will continue with the SNMP theme of network monitoring but with a fully featured network monitoring system called **Cacti**."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

If you are interested in getting in touch with the Author, please visit `https://members.networkautomation.community/`

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Mastering Python Networking, Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781803234618`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Review of TCP/IP Protocol Suite and Python

Welcome to the new and exciting age of network engineering! When I started working as a network engineer at the turn of the millennium some 20 years ago, the role was distinctly different than the network engineering role of today. At the time, network engineers mainly possessed domain-specific knowledge to manage and operate local and wide area networks using the command-line interface. While they might occasionally cross over the discipline wall to handle tasks normally associated with systems administration and developers, there was no explicit expectation for a network engineer to write code or understand programming concepts. This is no longer the case today.

Over the years, the DevOps and **Software–Defined Networking** (**SDN**) movement, among other factors, have significantly blurred the lines between network engineers, systems engineers, and developers.

The fact that you have picked up this book suggests that you might already be an adopter of network DevOps, or maybe you are considering going down that path of checking out network programmability. Maybe you have been working as a network engineer for many years, just as I had, and wanted to know what the buzz around the Python programming language is all about. You might even already have been fluent in the Python programming language but wonder what its applications are in the network engineering field.

If you fall into any of these camps or are simply just curious about Python in the network engineering field, I believe this book is for you:

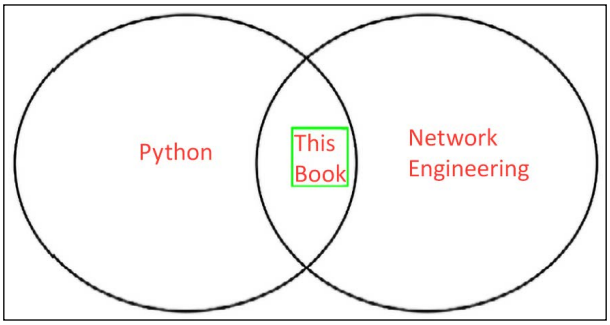


Figure 1.1: The intersection between Python and network engineering

There are already many great books written that dive into the topics of network engineering and Python separately. I do not intend to repeat their efforts with this book. Instead, this book assumes you have some hands-on experience in managing networks, as well as a basic understanding of network protocols. It's helpful if you're already familiar with Python as a programming language, but you do not need to be an expert. We will cover some Python basics later in the chapter as a baseline of Python knowledge. Again, you do not need to be an expert in Python or network engineering to read this book. This book intends to build on the basic foundations of network engineering and Python to help readers learn and practice various applications that can make their lives easier.

In this chapter, we will do a general review of some networking and Python concepts. The rest of the chapter should set the level of expectation of the prior knowledge required to get the most out of this book. If you want to brush up on the contents of this chapter, there are lots of free or low-cost resources to bring you up to speed. I would recommend the free Khan Academy (`https://www.khanacademy.org/`) and the official Python tutorials from the Python Software Foundation at `https://www.python.org/`.

This chapter will pay a very quick visit to the relevant networking topics at a high level without going too much into the details. There is simply not enough space in one chapter to do a deep dive on any of the topics. To be honest, most of us do not get to a deep level when doing day-to-day jobs. What is the level of required networking knowledge? Judging from my experience of working in the field, a typical network engineer or developer might not remember the exact **Transmission Control Protocol** (**TCP**) state machine to accomplish their daily tasks (I know I don't), but they would be familiar with the basics of the **Open Systems Interconnection** (**OSI**) model, the TCP and **User Datagram Protocol** (**UDP**) operations, different IP header fields, and other fundamental concepts. That is what we will cover in this chapter.

We will also look at a high-level review of the Python language first; just enough for those readers who do not code in Python daily to have ground to walk on for the rest of the book.

Specifically, we will cover the following topics:

- An overview of the Internet
- The OSI and client-server model
- TCP, UDP, and IP protocol suites
- Python syntax, types, operators, and loops
- Extending Python with functions, classes, and packages

Of course, the information presented in this chapter is not exhaustive; please check out the references for further information if required.

As network engineers, often we are challenged by the scale and complexity of the networks we need to manage. They range from small home-based networks, medium-size networks that make a small business go, to large multi-national enterprise networks spanning across the globe. The most extensive network of them all is, of course, the Internet. Without the Internet, there would be no email, websites, API, streaming media, or cloud computing as we know it. Therefore, before we dive deeper into the specifics of protocols and Python, let us begin with an overview of the Internet.

An overview of the Internet

What is the Internet? This seemingly easy question might receive different answers depending on each of our backgrounds. The Internet means different things to different people; the young, the old, students, teachers, business people, and poets could all give different answers to the same question.

To a network engineer, the Internet is a global computer network consisting of a web of inter-networks connecting large and small networks together. In other words, it is *a network of networks without a centralized owner*. Take your home network as an example. It might consist of a device that integrates the functions of routing, Ethernet switching, and wireless access points connecting your smartphones, tablets, computers, and internet-enabled TVs together for the devices to communicate with each other. This is your **Local Area Network (LAN)**.

When your home network needs to communicate with the outside world, it passes information from your LAN to a larger network, often appropriately named the **Internet Service Provider (ISP)**. The ISP is typically thought of as a business that you pay to get online. They can do this by aggregating small networks into bigger networks that they maintain.

Your ISP network often consists of many edge nodes that aggregate the traffic to their core network. The core network's function is to interconnect these edge networks via a high-speed network.

At some of the more specialized edge nodes, called Internet exchange points, your ISP is connected to other ISPs to pass your traffic appropriately to your destination. The return path from your destination to your home computer, tablet, or smartphone may or may not follow the same path through all these in-between networks back to your original device, while the source and destination remain the same. This asymmetrical behavior is designed to be fault-tolerant so that no one node can take down the whole connection.

Let's look at the components making up this web of networks.

Servers, hosts, and network components

Hosts are end nodes on the network that communicate with other nodes. In today's world, a host can be a traditional computer, or it can be your smartphone, tablet, or TV. With the rise of the **Internet of Things** (**IoT**), the broad definition of a host can be expanded to include an **Internet Protocol** (**IP**) camera, TV set-top boxes, and the ever-increasing types of sensors we use in agriculture, farming, automobiles, and more. With the explosion of the number of hosts connected to the Internet, they all need to be addressed, routed, and managed. The demand for proper networking has never been greater.

Most of the time when we are on the Internet, we make requests for services. This could be viewing a web page, sending or receiving emails, transferring files, and other online activities. These services are provided by **servers**. As the name implies, servers provide services to multiple nodes and generally have higher levels of hardware specification. In a way, servers are particular "super nodes" on the network that offer additional capabilities to their peers. We will look at servers again later, in the *The client-server model* section.

If you think of servers and hosts as cities and towns, the *network components* are the roads and highways that connect them together. In fact, the term information superhighway comes to mind when describing the network components that transmit the ever-increasing bits and bytes across the globe. In the Seven-Layer OSI model that we will look at in *The OSI model*, these network components are layer-one to layer-three devices that sometimes venture into layer four as well. They are layer-two and layer-three routers and switches that direct traffic, as well as layer-one transports such as fiber optic cables, coaxial cables, twisted copper pairs, and some **Dense Wavelength Division Multiplexing** (**DWDM**) equipment, to name a few.

Collectively, hosts, servers, storage, and network components make up the internet as we know it today.

The rise of data centers

In the last section, we looked at the different roles that servers, hosts, and network components play in the inter-network. Because of the higher hardware capacity that servers demand, they are often put together in a central location to be managed more efficiently. We often refer to these locations as data centers. They can generally be classified into three broad categories:

- Enterprise data centers
- Cloud data centers
- Edge data centers

Let us take a look at enterprise data centers first.

Enterprise data centers

In a typical enterprise, the company generally has business needs for internal tools such as emailing, document storage, sales tracking, ordering, HR tools, and a knowledge-sharing intranet. These services become file and mail servers, database servers, and web servers. Unlike user computers, these are generally high-end computers that require higher power, cooling, and high bandwidth network connections. A byproduct of the hardware is also the amount of noise it makes, which is not suitable for a normal workspace. The servers are generally placed in a central location, called the **Main Distribution Frame** (**MDF**), in the enterprise building to provide the necessary power feed, power redundancy, cooling, and network connectivity.

To connect to the MDF, the user's traffic is generally aggregated at locations closer to the user, which are sometimes called the **Intermediate Distribution Frame** (**IDF**), before they are bundled up and connected to the MDF. It is not unusual for the IDF-MDF spread to follow the physical layout of the enterprise building or campus. For example, each building floor can consist of an IDF that aggregates to the centralized MDF on another floor in the same building. If the enterprise consists of several buildings, further aggregation can be done by combining the buildings' traffic before connecting them to the enterprise data center.

Many enterprise data centers (sometimes referred to as campus networks) follow the three-layer network design. The layers are the access layer, the distribution layer, and the core layer. Of course, as with any design, there are no hard rules or one-size-fits-all model; the three-layer designs are just a general guide. As an example, to overlay the three-layer design to our User-IDF-MDF example earlier, the access layer is analogous to the ports each user connects to, the IDF can be thought of as the distribution layer, while the core layer consists of the connection to the MDF and the enterprise data centers. This is, of course, a generalization of enterprise networks, as some of them will not follow the same model.

Cloud data centers

With the rise of cloud computing and software, or **Infrastructure as a Service** (**IaaS**), the data centers the cloud providers have built are big in scale, sometimes referred to as hyper-scale data centers. What we referred to as cloud computing is the on-demand availability of computing resources offered by the likes of Amazon AWS, Microsoft Azure, and Google Cloud without the user having to manage the resources directly. Many web-scale service providers, such as Facebook, can also be grouped into this category.

Because of the number of servers they need to house, cloud data centers generally demand a much, much higher capacity of power, cooling, and network capacity than any enterprise data center. Even after working on cloud providers' data centers for many years, every time I visit a cloud provider's data center, I am still amazed at their scale. Just to give examples of their sheer scale, cloud data centers are so big and power-hungry that they are typically built close to power plants where they can get the cheapest power rate, without losing too much efficiency during the transportation of the power. Their cooling needs are so high that some are forced to be creative about where the data center is built. Facebook, for example, has built its Lulea data center in northern Sweden (just 70 miles south of the Arctic Circle) in part to leverage the cold temperature for cooling. Any search engine can give you some astounding numbers when it comes to the science of building and managing cloud data centers for the likes of Amazon, Microsoft, Google, and Facebook. The Microsoft data center in West Des Moines, Iowa, for example, consisted of 1.2 million square feet of the facility on 200 acres of land and required the city to spend an estimated $65 million in public infrastructure upgrades.

At the cloud provider scale, the services they need to provide are generally not cost-effective or feasible to be housed in a single server. The services are spread between a fleet of servers, sometimes across many racks, to provide redundancy and flexibility for service owners.

The latency and redundancy requirements, as well as the physical spread of servers, put a tremendous amount of pressure on the network. The interconnections required to connect the server fleets equate to an explosive growth of network equipment such as cables, switches, and routers. These requirements translate into the amount of equipment that needs to be racked, provisioned, and managed. A typical network design would be a multi-staged Clos network:

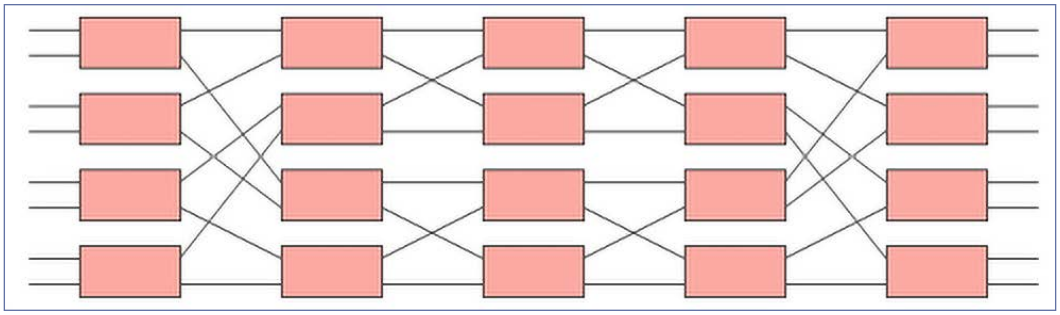


Figure 1.2: Clos network

In a way, cloud data centers and the increasing adaptions in other networks are where network automation becomes a necessity for the speed, flexibility, and reliability automation can provide. If we follow the traditional way of managing network devices via a Terminal and command-line interface, the number of engineering hours required will not allow the service to be available in a reasonable amount of time. Not to mention that human repetition is error-prone, inefficient, and a terrible waste of engineering talent. To add further complexity, there is often the need to quickly change some of the network configurations to accommodate rapidly changing business needs, such as redesigning a three-tier campus network into a CLOS-based topology.

Personally, cloud data center networking is where I started my path of network automation with Python several years ago, and I've never looked back since.

Edge data centers

If we have sufficient computing power at the data center level, why keep anything else but at these data centers? All the connections from clients worldwide can be routed back to the data center servers, and we can call it a day, right? The answer, of course, depends on the use case. The most significant limitation in routing the request and session all the way back from the client to a large data center is the latency introduced in the transport. In other words, significant latency is where the network becomes a bottleneck.

Of course, any elementary physics textbook can tell you that the network latency number would never be zero: even as fast as light can travel in a vacuum, it still takes time for physical transportation. In the real world, latency would be much higher than light in a vacuum. Why? Because the network packet must traverse through multiple networks, sometimes through an undersea cable, slow satellite links, 4G or 5G cellular links, or Wi-Fi connections.

How can we reduce network latency? One solution would be to reduce the number of networks the end-user requests traverse through. We can try to be as closely connected to the end-user as possible, perhaps meeting the user at the edge where the requests enter our network. We can place enough resources at these edge locations to serve the request. This is especially common for servicing media content such as music and videos.

Let's take a minute and imagine that you are building the next generation of video streaming services. To increase customer satisfaction with smooth streaming, you would want to place the video server as close to the customer as possible, either inside or very near to the customer's ISP. Also, for redundancy and connection speed, the upstreaming of the video server farm would not just be connected to one or two ISPs, but all the ISPs that we can connect to reduce the hop count, thus reducing the number of devices we need to pass thru. All the connections would have as much bandwidth as needed to decrease peak-hour latency. This need gave rise to the peering exchange's edge data centers of big ISP and content providers. Even when the number of network devices is not as high as cloud data centers, they too can benefit from network automation in terms of the increased reliability, flexibility, security, and visibility network automation brings.

If we extend the concept of edge nodes and get creative, we can see that some of the latest technologies, such as self-driving cars and **Software-Defined Wide-Area-Networks (SD-WANs)** are also applications of edge nodes. The self-driving car needs to make split-second decisions based on its sensors. SD-WAN routers need to route packets locally, without the need to consult a central "brain." These are all concepts of intelligent edge nodes.

As with many complex subjects, we can tackle the complexity by dividing the subject into smaller digestible pieces. Networking breaks the complexity by using layers to model the functions of its elements. Over the years, there have been different networking models. We will look at two of the most important models in this book, starting with the OSI model.

The OSI model

No network book is complete without first going over the OSI model. The model is a conceptual model that componentizes the telecommunication functions into different layers. The model defines seven layers, and each layer sits independently on top of another one with defined structures and characteristics.

For example, in the network layer, IP is located on top of the different data link layers, such as Ethernet or frame relay. The OSI reference model is a good way to normalize different and diverse technologies into a set of common languages that people can agree on. This greatly reduces the scope for parties working on individual layers and allows them to look at specific tasks in depth without worrying too much about compatibility:

OSI Model			
Layer		Protocol data unit (PDU)	Function
Host Layers	7. Application	Data	High-level APIs, including resource sharing, remote file access
	6. Presentation		Translation of data between a networking service and an application; including character encoding, data compression, and encryption / decryption
	5. Session		Managing communication sessions, i.e. continuous exchange of Information in the form of multiple back-end-forth transmissions between two nodes
	4. Transport	Segment (TCP) / Datagram (UDP)	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
Media Layers	3. Network	Packet	Structuring and managing a multi-node network, including addressing routing and traffic control
	2. Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
	1. Physical	Bit	Transmission and reception of raw bit streams over a physical medium

Figure 1.3: OSI model

The OSI model was initially worked on in the late 1970s and was later published jointly by the **International Organization for Standardization (ISO)**, what is now known as the **Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T)**. It is widely accepted and commonly referred to when introducing a new topic in telecommunication.

Around the same time as the OSI model development, the Internet was taking shape. The reference model the original designer used for the Internet is often referred to as the TCP/IP model. The TCP and the IP were the original protocol suites contained in the design. This is somewhat similar to the OSI model in the sense that they divide end-to-end data communication into abstraction layers.

What is different in the TCP/IP model is it combines layers 5 to 7 in the OSI model in the **Application** layer, while the **Physical** and **Data link** layers are combined in the **Link** layer:

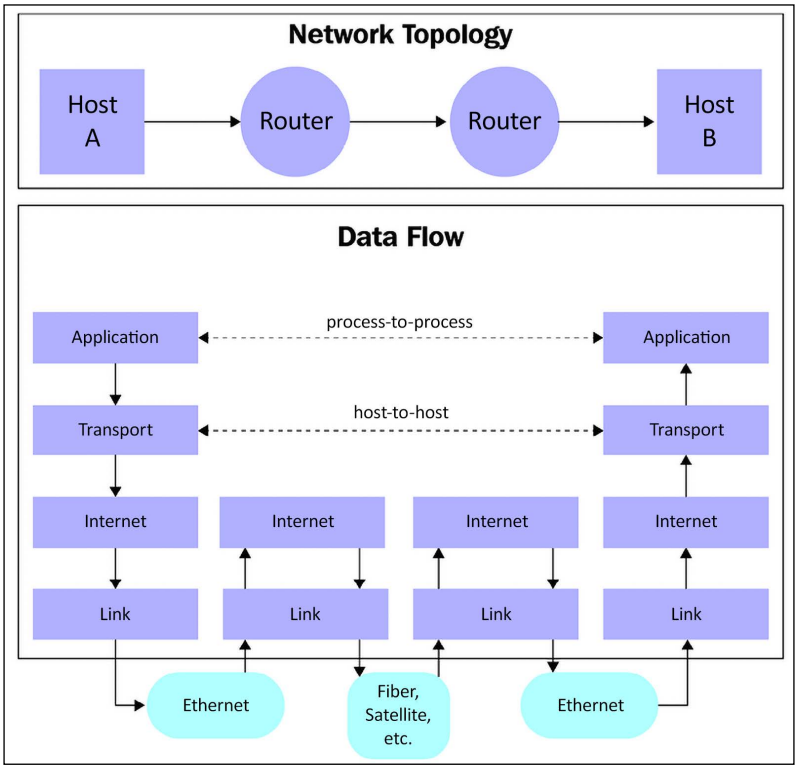


Figure 1.4: Internet protocol suite

Both the OSI and TCP/IP models are useful for providing standards for end-to-end data communication. We will refer to either the OSI or the TCP/IP model when needed, such as when we are discussing the web framework in the upcoming chapters. Just like models at the transport layer, there are also reference models that govern communication at the application level. In the modern network, the client-server model is what most applications are based on. We will look at the client-server model in the next section.

The client-server model

The client-server reference models demonstrated a standard way for data to communicate between two nodes. Of course, by now, we all know that not all nodes are created equal. Even in the earliest **Advanced Research Projects Agency Network (ARPANET)** days, there were workstation nodes, and there were server nodes with the purpose of providing content to other workstation nodes.

These server nodes typically have higher hardware specifications and are managed more closely by engineers. Since these nodes provide resources and services to others, they are appropriately referred to as servers. Servers typically sit idle, waiting for clients to initiate requests for their resources. This model of distributed resources that are requested by the client request is referred to as the client-server model.

Why is this important? If you think about it for a minute, the importance of networking is greatly highlighted by this client-server model. Without the need to transfer services between clients and servers, there is not a lot of need for network interconnections. It is the need to transfer bits and bytes from the client to the server that shines a light on the importance of network engineering. Of course, we are all aware of how the biggest network of them all, the Internet, has been transforming the lives of all of us and is continuing to do so.

You might be asking, how can each node determine the time, speed, source, and destination every time they need to talk to each other? This brings us to network protocols.

Network protocol suites

In the early days of computer networking, protocols were proprietary and closely controlled by the company that designed the connection method. If you were using **Novell's IPX/SPX** protocol in your hosts, the same hosts would not be able to communicate with Apple's **AppleTalk** hosts, and vice versa. These proprietary protocol suites generally have analogous layers to the OSI reference model and follow the client-server communication method but are not compatible with each other. The proprietary protocols generally only work in LANs that are closed, without the need to communicate with the outside world. When traffic does need to move beyond the local LAN, typically a protocol translation device, such as a router, is used to translate from one protocol to another. For example, to connect an AppleTalk-based network to the Internet, a router would be used to connect and translate the AppleTalk protocol to an IP-based network. The additional translation is usually not perfect, but since most of the communication happened within the LAN in the early days, it was accepted by the network administrators.

However, as the need for inter-network communication rises beyond the LAN, the need for standardizing the network protocol suites becomes greater. The proprietary protocols eventually gave way to the standardized protocol suites of TCP, UDP, and IP, which greatly enhanced the ability of one network to talk to another. The Internet, the greatest network of them all, relies on these protocols to function properly. In the next few sections, we will look at each of the protocol suites.

The transmission control protocol

TCP is one of the main protocols used for the Internet today. If you have opened a web page or have sent an email, you have come across the TCP protocol. The protocol sits at layer 4 of the OSI model, and it is responsible for delivering the data segment between two nodes in a reliable and error-checked manner. The TCP consists of a 160-bit header that contains, among other fields, source and destination ports, a sequence number, an acknowledgment number, control flags, and a checksum:

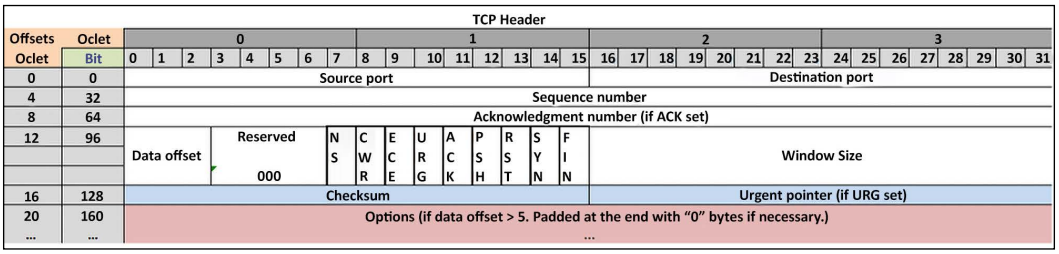


Figure 1.5: TCP header

Functions and characteristics of TCP

TCP uses datagram sockets or ports to establish host-to-host communication. The standard body, called the **Internet Assigned Numbers Authority** (**IANA**), designates well-known ports to indicate certain services, such as `port 80` for HTTP (web) and `port 25` for SMTP (mail). The server in the client-server model typically listens on one of these well-known ports in order to receive communication requests from the client. The TCP connection is managed by the operating system with the socket representing the local endpoint for the connection.

The protocol operation consists of a state machine, which the machine needs to keep track of when it is listening for an incoming connection during the communication session, as well as releasing resources once the connection is closed. Each TCP connection goes through a series of states such as `Listen`, `SYN-SENT`, `SYN-RECEIVED`, `ESTABLISHED`, `FIN-WAIT`, `CLOSE-WAIT`, `CLOSING`, `LAST-ACK`, `TIME-WAIT`, and `CLOSED`. The different states help in managing the TCP messages.

TCP messages and data transfer

The biggest difference between TCP and UDP, which is its close cousin on the same layer, is that it transmits data in an ordered and reliable fashion. The fact that the TCP operation guarantees delivery is often referred to TCP as a connection-oriented protocol. It does this by first establishing a three-way handshake to synchronize the sequence number between the transmitter and the receiver, `SYN`, `SYN-ACK`, and `ACK`.

The acknowledgment is used to keep track of subsequent segments in the conversation. Finally, at the end of the conversation, one side will send a `FIN` message, and the other side will `ACK` the `FIN` message as well as sending a `FIN` message of its own. The `FIN` initiator will then `ACK` the `FIN` message that it received.

As many of us who have troubleshot a TCP connection can tell you, the operation can get quite complex. One can certainly appreciate that, most of the time, the operation just happens silently in the background.

A whole book could be written about the TCP protocol; in fact, many excellent books have been written on the protocol.

As this section is a quick overview, if interested, The *TCP/IP Guide* (`http://www.tcpipguide.com`) is an excellent free resource that you can use to dig deeper into the subject.

The user datagram protocol

The UDP is also a core member of the protocol suites used. Like TCP, it operates on layer 4 of the OSI model that is responsible for delivering data segments between the application and the IP layer. Unlike TCP, the header is only 64 bits, which only consists of a source and destination port, length, and checksum. The lightweight header makes it ideal for applications that prefer faster data delivery without setting up the session between two hosts or needing reliable data delivery. Perhaps it's hard to imagine with today's fast internet connections, but the lightweight header made a big difference to the speed of transmission in the early days of `X.21` and frame relay links.

Besides the speed difference, not having to maintain various states, such as TCP, also saves computer resources on the two endpoints:

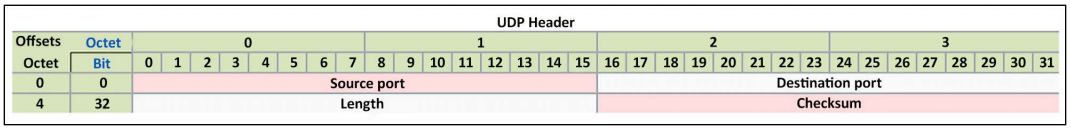


Figure 1.6: UDP header

You might now wonder why UDP was ever used at all in the modern age; given the lack of reliable transmissions, wouldn't we want all the connections to be reliable and error-free? If you think about multimedia video streaming or Skype calls, those applications benefit from a lighter header when the application just wants to deliver the datagram as quickly as possible. You can also consider the fast **Domain Name System (DNS)** lookup process based on the UDP protocol. The tradeoff between accuracy and latency usually tips to the side of low latency.

When the address you type in on the browser is translated into a computer-understandable address, the user will benefit from a lightweight process, since this has to happen before even the first bit of information is delivered to you from your favorite website.

Again, this section does not do justice to the topic of UDP, and you are encouraged to explore the topic through various resources if you are interested in learning more about UDP.

The Wikipedia article on UDP, `https://en.wikipedia.org/wiki/User_Datagram_Protocol`, is a good starting point to learn more about UDP.

The Internet protocol

As network engineers will tell you, we live at the IP layer, which is layer 3 on the OSI model. IP has the job of addressing and routing between end nodes, among others. The addressing of an IP is probably its most important job. The address space is divided into two parts: the network and the host portion. The subnet mask is used to indicate which portion in the network address consists of the network and which portion is the host by matching the network portion with a 1 and the host portion with a 0. IPv4 expresses the address in the dotted notation, for example, `192.168.0.1`.

The subnet mask can either be in a dotted notation (`255.255.255.0`) or use a forward slash to express the number of bits that should be considered in the network bit (`255.255.255.0` or `/24`):

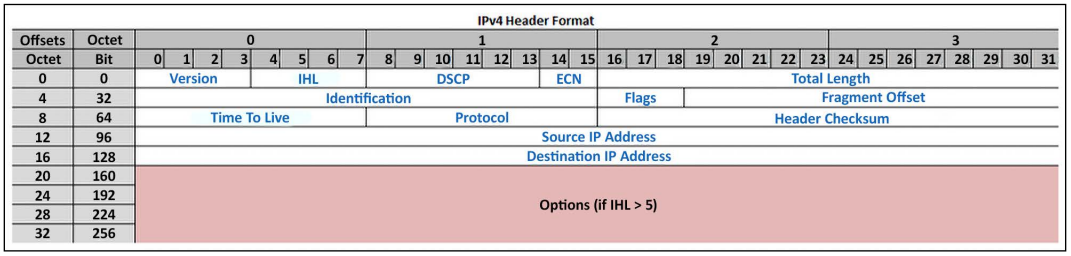


Figure 1.7: IPv4 header

The IPv6 header, the next generation of the IP header of IPv4, has a fixed portion and various extension headers:

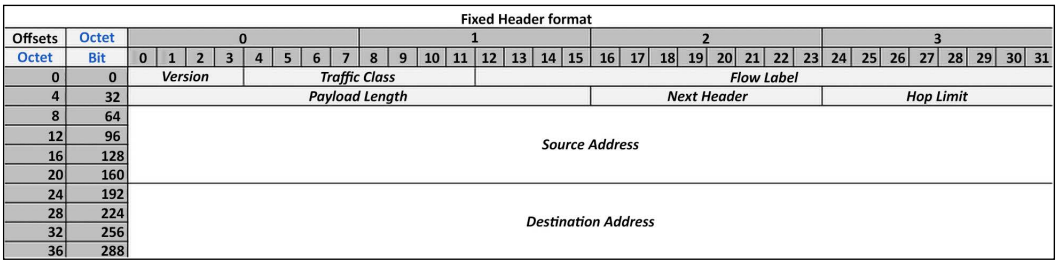


Figure 1.8: IPv6 header

The IPv6 **Next Header** field in the fixed header section can indicate an extension header to be followed that carries additional information. It can also identify the upper-layer protocol such as TCP and UDP. The extension headers can include routing and fragment information. For example, the extension header can include how the original packet is fragmented so the destination node can reassemble the packet accordingly. As much as the protocol designer would like to move from IPv4 to IPv6, the Internet today is still mainly addressed with IPv4, with some of the service provider networks addressed with IPv6 natively.

IP network address translation (NAT) and network security

NAT is typically used for translating a range of private IPv4 addresses into publicly routable IPv4 addresses. But it can also mean a translation between IPv4 and IPv6, such as at a carrier edge when they use IPv6 inside of the network that needs to be translated to IPv4 when the packet leaves the network. Sometimes, NAT6 to IPv6 is used as well for security reasons.

Security is a continuous process that integrates all the aspects of networking, including automation and Python. This book aims to use Python to help you manage the network; security will be addressed as part of the following chapters in the book, such as using Python to implement access lists, search for breaches in the log, and so on. We will also look at how we can use Python and other tools to gain visibility in the network, such as a graphic network topology dynamically based on the network device information.

IP routing concepts

IP routing is about having the intermediate devices between the two endpoints transmit the packets between them based on the IP header. For all communication that happens on the Internet, the packet will traverse through various intermediate devices. As mentioned, the intermediate devices consist of routers, switches, optical gears, and various other gears that might not examine beyond the network and transport layer. In a road trip analogy, you might travel in the United States from the city of San Diego in California to the city of Seattle in Washington. The IP source address is analogous to San Diego and the destination IP address can be thought of as Seattle. On your road trip, you will stop by many different intermediate spots, such as Los Angeles, San Francisco, and Portland; these can be thought of as the intermediary routers and switches between the source and destination.

Why was this important? In a way, this book is about managing and optimizing these intermediate devices. In the age of mega data centers that span the size of multiple American football fields, the need for efficient, agile, reliable, and cost-effective ways to manage the network becomes a major point of competitive advantage for companies. In future chapters, we will dive into how we can use Python programming to effectively manage a network.

Now that we've looked at network reference models and protocol suites, we're ready to dive into the Python language itself. In this chapter, we'll begin with a broad overview of Python.

Python language overview

In a nutshell, this book is about making our network engineering lives easier with Python. But what is Python and why is it the language of choice of many DevOps engineers? In the words of the Python Foundation Executive Summary (`https://www.python.org/doc/essays/blurb/`):

> *"Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level, built-in data structure, combined with dynamic typing and dynamic binding, makes it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy-to-learn syntax emphasizes readability and therefore reduces the cost of program maintenance."*

If you are somewhat new to programming, the "object-oriented," and "dynamic semantics" mentioned in the summary probably do not mean much to you. But I think we can all agree that "rapid application development" and "simple, easy-to-learn syntax" sound good. Python, as an interpreted language, means there is little to no compilation process required before execution, so the time needed to write, test, and edit Python programs is significantly reduced. For simple scripts, if your script fails, a print statement could be all you need to troubleshoot the issue.

Using the interpreter also means that Python is easily ported to different operating systems, such as Windows and Linux. A Python program written on one operating system can be used on another with little to no change.

The functions, modules, and packages encourage code reuse by breaking a large program into simple reusable pieces. The object-oriented nature of Python takes it one step further for grouping the components into objects. In fact, all Python files are modules that can be reused or imported into another Python program. This makes it easy to share programs between engineers and encourages code reuse. Python also has a *batteries-included* mantra, which means for everyday tasks, you need not download any additional packages outside of the Python language itself. To achieve this goal without the code being too bloated, a set of Python modules, a.k.a. standard libraries, are installed when you install the Python interpreter. For common tasks such as regular expressions, mathematical functions, and JSON decoding, all you need is the *import* statement, and the interpreter will move those functions into your program. This *batteries-included* mantra is what I would consider one of the killer features of the Python language.

Lastly, the fact that Python code can start in a relatively small-sized script with a few lines of code and grow into a complete production system is convenient for network engineers. As many of us know, the network typically grows organically without a master plan. A language that can grow with your network is invaluable. You might be surprised to see that a language deemed a scripting language by many is being used for whole production systems of many cutting-edge companies (organizations using Python: `https://wiki.python.org/moin/OrganizationsUsingPython`).

If you have ever worked in an environment where you must switch between working on different vendor platforms, such as Cisco IOS and Juniper Junos, you know how painful it is to switch between syntaxes and usage when trying to achieve the same task. Since Python is flexible enough for both small and large programs, there is no such dramatic context switching. It is just the same Python code from small to large!

For the rest of the chapter, we will take a high-level tour of the Python language. If you are already familiar with the basics, feel free to quickly scan through it or skip ahead to *Chapter 2*.

Python versions

As many readers will already be aware, Python has been going through a transition from Python 2 to Python 3 for the last few years. Python 3 was released in 2008, over 10 years ago, **with active development**, with the most recent release of 3.10. Unfortunately, Python 3 is not backward compatible with Python 2.

At the time of writing the fourth edition of this book, in mid-2022, the Python community has all but moved over to Python 3. In fact, Python 2 officially entered end-of-life as of January 1, 2020. The latest Python 2.x release, 2.7, was released over six years ago in mid-2010. Since Python 2 is end-of-life and is no longer maintained by the Python Software Foundation, we should all be using Python 3. In this book, we will be using the latest stable Python 3 release, Python 3.10. Python 3.10 has many exciting features, such as a stable asynchronous I/O that is very helpful in network automation. This book will use Python 3 for the code examples unless otherwise stated. When applicable, we will point out the Python 2 and Python 3 differences.

Operating system

As mentioned, Python is cross-platform. Python programs can be run on Windows, Mac, and Linux. In reality, certain care needs to be taken when you need to ensure cross-platform compatibilities, such as taking care of the subtle differences between backslashes in Windows filenames and the steps to activate Python virtual environments on different operating systems. Since this book is written for DevOps, systems, and network engineers, Linux is the preferred platform for the intended audience, especially in production.

The code in this book will be tested on Linux Ubuntu 22.04 LTS machines. As of this writing, Python 3.10.4 is the default version that ships with 22.04, so we do not need to install Python 3 separately. I will also try my best to make sure the code runs the same on the Windows and the macOS platforms.

If you are interested in the OS details, they are as follows:

```
$ uname -a
Linux network-dev-4 5.15.0-39-generic #42-Ubuntu SMP Thu Jun 9 23:42:32
UTC 2022 x86_64 x86_64 x86_64 GNU/Linux

$ lsb_release -a
No LSB modules are available.
Distributor ID:    Ubuntu
Description:       Ubuntu 22.04 LTS
Release:    22.04
Codename:   jammy
```

Running a Python program

Python programs are executed by an interpreter, which means the code is fed through this interpreter to be executed by the underlying operating system. There are several different implementations of the interpreter by the Python development community, such as IronPython and Jython. In this book, we will use the most common Python interpreter today, CPython. Whenever we mention Python in this book, we refer to CPython unless otherwise indicated.

One way you can use Python is by taking advantage of the interactive prompt. This is useful when you want to quickly test a piece of Python code or concept without writing a whole program.

This is typically done by simply typing in the `Python3` keyword:

```
$ python3
Python 3.10.4 (main, Apr  2 2022, 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
```

The interactive mode is one of Python's most useful features. In the interactive shell, you can type any valid statement or sequence of statements and immediately get a result back. I typically use the interactive shell to explore a feature or library that I am not familiar with.

The interactive mode can also be used for more complex tasks such as experimenting with data structure behaviors, for example, mutable versus immutable data types. Talk about instant gratification!

On Windows, if you do not get a Python shell prompt back, you might not have the program in your system search path. The latest Windows Python installation program provides a checkbox for adding Python to your system path; make sure that is checked during installation. Or, you can add the program to the path manually by going to **Environment settings**.

A more common way to run the Python program, however, is to save your Python file and run it via the interpreter after. This will save you from typing in the same statements repeatedly. Python files are just regular text files typically saved with the `.py` extension. In the *Nix world, you can also add the **shebang** (#!) line on top to specify the interpreter that will be used to run the file. The # character can be used to specify comments that will not be executed by the interpreter. The file `helloworld.py` has the following statements:

```
# This is a comment
print("hello world")
```

This can be executed as follows:

```
$ python helloworld.py
hello world
```

Let us take a look at the basic Python building structures, built-in data types.

Python built-in types

In computer programming, data types typically refer to a way for the computer program to know the kind of value a variable can have, such as a word or a number. Python implements dynamic typing, or duck typing, and tries to determine the object's type as you declare it automatically. Python has several standard types built into the interpreter:

- **Numerics**: `int`, `float`, complex, and `bool` (the subclass of `int` with a `True` or `False` value)
- **Sequences**: `str`, `list`, `tuple`, and `range`
- **Mappings**: `dict`
- **Sets**: `set` and `frozenset`
- **None**: The `null` object

We will briefly examine the different types in Python. If they do not make sense to you at this moment, they might make more sense when we apply them in examples in the section that follows.

The None type

The `None` type denotes an object with no value. The `None` type is returned in functions that do not explicitly return anything such as a function but just do some math calculation and exit. The `None` type is also used in function arguments to error out if the caller does not pass in an actual value. For example, we can specify in a function 'if a==None, raise an error'.

Numerics

Python numeric objects are basically numbers. With the exception of Booleans, the numeric types of `int`, `long`, `float`, and `complex` are all signed, meaning they can be positive or negative. A Boolean is a subclass of the integer, which can be one of two values: 1 for True, and 0 for False. In practice, we are almost always testing Booleans with True or False instead of the numerics 1 and 0. The rest of the numeric types are differentiated by how precisely they can represent the number; in Python 3, `int` does not have a maximum size while in Python 2 `int` indicates whole numbers with a limited range. Floats are numbers using the double-precision representation (64-bit) on the machine.

Sequences

Sequences are ordered sets of objects with an index of non-negative integers. Since there are many different sequence types in Python (str, lists, tuple, etc), let us use the interactive interpreter to illustrate the different sequence types.

Please feel free to type along on your own computer.

Sometimes, it surprises people that `string` (think of words) is a sequence type. But if you look closely, strings are a series of characters put together. Strings are enclosed by either single, double, or triple quotes.

Note in the following examples, the quotes must match. A beginning double quote needs to be matched with an ending double quote. Triple quotes allow the string to span different lines:

```
>>> a = "networking is fun"
>>> b = 'DevOps is fun too'
>>> c = """what about coding?
... super fun!"""
>>>
```

The other two commonly used sequence types are lists and tuples. Lists are sequences of arbitrary objects. Lists can be created by enclosing objects in square brackets. Just like strings, lists are indexed by non-zero integers that start at zero.

The values of a list are retrieved by referencing the index number:

```
>>> vendors = ["Cisco", "Arista", "Juniper"]
>>> vendors[0]
'Cisco'
>>> vendors[1]
'Arista'
>>> vendors[2]
'Juniper'
```

Tuples are like lists, created by enclosing values in parentheses. Like lists, the values in the tuple are retrieved by referencing its index number. Unlike lists, the values cannot be modified after creation:

```
>>> datacenters = ("SJC1", "LAX1", "SFO1")
>>> datacenters[0]
'SJC1'
>>> datacenters[1]
'LAX1'
>>> datacenters[2]
'SFO1'
```

Some operations are common to all sequence types, such as returning an element by index. A sequence can also be sliced for a portion of its elements:

```
>>> a
'networking is fun'
>>> a[1]
'e'
>>> vendors
['Cisco', 'Arista', 'Juniper']
>>> vendors[1]
'Arista'
>>> datacenters
('SJC1', 'LAX1', 'SFO1')
>>> datacenters[1]
'LAX1'
>>>
>>> a[0:2]
'ne'
```

```
>>> vendors[0:2]
['Cisco', 'Arista']
>>> datacenters[0:2]
('SJC1', 'LAX1')
>>>
```

Remember that the index starts at 0. Therefore, the index of 1 is the second element in the sequence.

There are also common functions that can be applied to sequence types, such as checking the number of elements and finding the minimum and maximum values amongst all the elements:

```
>>> len(a)
17
>>> len(vendors)
3
>>> len(datacenters)
3
>>>
>>> b = [1, 2, 3, 4, 5]
>>> min(b)
1
>>> max(b)
5
```

There are various methods that are applicable only to strings. It is worth noting that these methods do not modify the underlying string data itself and always return a new string. In short, mutable objects such as lists and dictionaries can be changed after creation, and immutable objects, for example, strings, cannot. If you want to use the new returned value for other operations, you will need to catch the return value and assign it to a different variable:

```
>>> a
'networking is fun'
>>> a.capitalize()
'Networking is fun'
>>> a.upper()
'NETWORKING IS FUN'
>>> a
'networking is fun'
>>> b = a.upper()
>>> b
```

```
'NETWORKING IS FUN'
>>> a.split()
['networking', 'is', 'fun']
>>> a
'networking is fun'
>>> b = a.split()
>>> b
['networking', 'is', 'fun']
>>>
```

Here are some of the common methods for a list. The Python list data type is a very useful structure in terms of putting multiple items together and iterating through them one at a time. For example, we can make a list of data center spine switches and apply the same access list to all of them by iterating through them one by one. Since a list's value can be modified after creation (unlike tuples), we can also expand and contract the existing list as we move along the program:

```
>>> routers = ['r1', 'r2', 'r3', 'r4', 'r5']
>>> routers.append('r6')
>>> routers
['r1', 'r2', 'r3', 'r4', 'r5', 'r6']
>>> routers.insert(2, 'r100')
>>> routers
['r1', 'r2', 'r100', 'r3', 'r4', 'r5', 'r6']
>>> routers.pop(1)
'r2'
>>> routers
['r1', 'r100', 'r3', 'r4', 'r5', 'r6']
```

Python lists are great for storing data, but it is a bit tricky at times to keep track of data if we need to reference it by location. If that is an issue, we can use a different Python data type. Let's look at Python mapping type next.

Mapping

Python provides one mapping type, called the **dictionary**. The dictionary data type is what I think of as a poor man's database because it contains objects that can be indexed by keys. This is often referred to as an *associated array or hashing table* in other programming languages. If you have used any of the dictionary-like objects in other languages, you will know how powerful this type can be, because you can refer to the object with a human-readable key.

This key, instead of just a number, will make much more sense to the poor guy who is trying to maintain and troubleshoot the code. That guy could be you only a few months after you wrote the code, trying to troubleshoot the code at 2 AM.

The object in the dictionary value can also be another data type, such as a list. As we have used square brackets for lists and round braces for tuples, we use curly braces to create a dictionary. Here is an example of how we can use a dictionary to represent our data center devices:

```
>>> datacenter1 = {'spines': ['r1', 'r2', 'r3', 'r4']}
>>> datacenter1['leafs'] = ['l1', 'l2', 'l3', 'l4']
>>> datacenter1
{'leafs': ['l1', 'l2', 'l3', 'l4'], 'spines': ['r1',
'r2', 'r3', 'r4']}
>>> datacenter1['spines']
['r1', 'r2', 'r3', 'r4']
>>> datacenter1['leafs']
['l1', 'l2', 'l3', 'l4']
```

The Python dictionary is one of my favorite data containers to use in network scripts and I use it all the time. However, there are other data containers that can come in handy in different use cases – a set is one of them.

Sets

A **set** is used to contain an unordered collection of objects. Unlike lists and tuples, sets are unordered and cannot be indexed by numbers. But there is one character that makes sets stand out as useful: the elements of a set are never duplicated. Imagine you have a list of IPs that you need to put in an access list. The only problem with this list of IPs is that they are full of duplicates.

Now, think about how many lines of code you would use to loop through the list of IPs to sort out unique items, one at a time. Now consider this: the built-in set type would allow you to eliminate duplicate entries with just one line of code. Personally, the Python set data type is not used that much in my code, but when I need it, I am always very thankful it exists. Once the set or sets are created, they can be compared with each other using the `union`, `intersection`, and `differences`:

```
>>> a = "hello"
# Use the built-in function set() to convert the string to a set
>>> set(a)
{'h', 'l', 'o', 'e'}
>>> b = set([1, 1, 2, 2, 3, 3, 4, 4])
```

```
>>> b
{1, 2, 3, 4}
>>> b.add(5)
>>> b
{1, 2, 3, 4, 5}
>>> b.update(['a', 'a', 'b', 'b'])
>>> b
{1, 2, 3, 4, 5, 'b', 'a'}
>>> a = set([1, 2, 3, 4, 5])
>>> b = set([4, 5, 6, 7, 8])
>>> a.intersection(b)
{4, 5}
>>> a.union(b)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> 1 *
{1, 2, 3}
```

Now that we have looked at different data types, we will take a tour of Python operators next.

Python operators

Python has some *numeric operators* that you would expect from any programming language, such as +, -, and so on; note that the truncating division (//, also known as **floor division**) truncates the result to an integer and a floating-point but only the integer value is returned. The modulo (%) operator returns the remainder value in the division:

```
>>> 1 + 2
3
>>> 2 - 1
1
>>> 1 * 5
5
>>> 5 / 1 #returns float
5.0
>>> 5 // 2 # // floor division
2
>>> 5 % 2 # modulo operator
1
```

There are also *comparison operators*. Note the double equals sign is used for comparison and a single equals sign is used for variable assignment:

```
>>> a = 1
>>> b = 2
>>> a == b
False
>>> a > b
False
>>> a < b
True
>>> a <= b
True
```

We can also use two of the common membership operators to test whether an object is in a sequence type:

```
>>> a = 'hello world'
>>> 'h' in a
True
>>> 'z' in a
False
>>> 'h' not in a
False
>>> 'z' not in a
True
```

The Python operators allow us to perform simple operations efficiently. In the next section, we will look at how we can use control flows to repeat these operations.

Python control flow tools

The `if`, `else`, and `elif` statements control conditional code execution. Unlike some other programming languages, Python uses indentation to structure blocks. There can be any number of indentation spaces as long as they align. Common practice is typically to use 2 or 4 spaces. The format of the conditional statement is as follows:

```
if expression:
  do something
elif expression:
  do something if the expression meets
```

```
elif expression:
  do something if the expression meets
...
else:
  statement
```

Here is a simple example:

```
>>> a = 10
>>> if a > 1:
...   print("a is larger than 1")
... elif a < 1:
...   print("a is smaller than 1")
... else:
...   print("a is equal to 1")
...
a is larger than 1
>>>
```

The `while` loop will continue to execute until the condition is `False`, so be careful with this one if you don't want to continue to execute (and crash your process):

```
while expression:
  do something

>>> a = 10
>>> b = 1
>>> while b < a:
...   print(b)
...   b += 1
...
1
2
3
4
5
6
7
8
9
```

The `for` loop works with any object that supports iteration; this means all the built-in sequence types, such as `lists`, `tuples`, and `strings`, can be used in a `for` loop. The letter `i` in the following for loop is an iterating variable, so you can typically pick something that makes sense within the context of your code:

```
for i in sequence:
  do something
>>> a = [100, 200, 300, 400]
>>> for number in a:
...   print(number)
...
100
200
300
400
```

Now that we have looked at Python data types, operators, and control flows, we are ready to group them together into reusable code pieces called functions.

Python functions

Often times when you find yourself copying and pasting some pieces of code, it is a good sign you should break it up into self-contained chunks of functions. This practice allows for better modularity, is easier to maintain, and allows for code reuse. Python functions are defined by using the `def` keyword with the function name, followed by the function parameters. The body of the function consists of the Python statements that are to be executed. At the end of the function, you can choose to return a value to the function caller. By default, it will return the `None` object if you do not specify a return value:

```
def name(parameter1, parameter2):
  statements
  return value
```

We will see a lot more examples of functions in the following chapters, but here is a quick example. In the following example, we use positional parameters, so the first element is referred to by the first variable in the function. Another way of referring to parameters is keywords with default values, such as `def subtract(a=10, b=5)`:

```
>>> def subtract(a, b):
...   c = a - b
```

```
...   return c
...
>>> result = subtract(10, 5)
>>> result
5
```

Python functions are great for grouping tasks together. Can we group different functions into a bigger piece of reusable code? Yes, we can do that via Python classes.

Python classes

Python is an **object-oriented programming** (**OOP**) language. The way Python creates objects is with the `class` keyword. A Python object is most commonly a collection of functions (methods), variables, and attributes (properties). Once a class is defined, you can create instances of the class. The class serves as a blueprint for subsequent instances.

The topic of OOP is outside the scope of this chapter, but here is a simple example of a `router` object definition to illustrate the point:

```
>>> class router(object):
...   def __init__(self, name, interface_number, vendor):
...     self.name = name
...     self.interface_number = interface_number
...     self.vendor = vendor
...
>>>
```

Once defined, we can create as many instances of that class as we'd like:

```
>>> r1 = router("SFO1-R1", 64, "Cisco")
>>> r1.name
'SFO1-R1'
>>> r1.interface_number
64
>>> r1.vendor
'Cisco'
>>>
>>> r2 = router("LAX-R2", 32, "Juniper")
>>> r2.name
'LAX-R2'
```

```
>>> r2.interface_number
32
>>> r2.vendor
'Juniper'
>>>
```

Of course, there is a lot more to Python objects and OOP. We will look at more examples in future chapters.

Python modules and packages

Any Python source file can be used as a module and any functions and classes you define in that source file can be reused by other Python scripts. To load the code, the file referencing the module needs to use the `import` keyword. Three things happen when the file is imported:

1. The file creates a new namespace for the objects defined in the source file.
2. The caller executes all the code contained in the module.
3. The file creates a name within the caller that refers to the module being imported. The name matches the name of the module.

Remember the `subtract()` function that we defined using the interactive shell? To reuse the function, we can put it into a file named `subtract.py`:

```
def subtract(a, b):
  c = a - b
  return c
```

In a file within the same directory of `subtract.py`, you can start the Python interpreter and import this function:

```
>>> import subtract
>>> result = subtract.subtract(10, 5)
>>> result
5
```

This works because, by default, Python will first search for the current directory for the available modules. Remember the standard library that we mentioned a while back? You guessed it, those are just Python files being used as modules.

If you are in a different directory, you can manually add a search path location using the `sys` module with `sys.path`.

Can we group multiple modules together in Python? Yes, Python packages allow a collection of modules to be grouped together. This further organizes Python modules for more namespace protection and better reusability. A package is defined by creating a directory with a name you want to use as the namespace, then placing the module source file under that directory.

In order for Python to recognize the directory as a Python package, just create a `__init__.py` file in this directory. The `__init__.py` file can be an empty file. In the same example as the `subtract.py` file, let's say if you were to create a directory called `math_stuff`, we can create a `__init__.py` file in the directory:

```
$ mkdir math_stuff
$ touch math_stuff/__init__.py
$ tree
.
├── helloworld.py
└── math_stuff
    ├── __init__.py
    └── subtract.py
1 directory, 3 files
$
```

The way to refer to the module is to include the package name using dot notation, for example, `math_stuff.subtract`:

```
>>> from math_stuff.subtract import subtract
>>> result = subtract(10, 5)
>>> result
5
>>>
```

As you can see, modules and packages are great ways to organize large code files and make sharing Python code a lot easier.

Summary

In this chapter, we covered the OSI model and reviewed network protocol suites, such as TCP, UDP, and IP. They work as the layers that handle the addressing and communication negotiation between any two hosts. The protocols were designed with extensibility in mind and have largely been unchanged from their original design. Considering the explosive growth of the internet, that is quite an accomplishment.

We also quickly reviewed the Python language, including built-in types, operators, control flows, functions, classes, modules, and packages. Python is a powerful, production-ready language that is also easy to read. This makes the language an ideal choice when it comes to network automation. Network engineers can leverage Python to start with simple scripts and gradually move on to other advanced features.

In *Chapter 2*, *Low-Level Network Device Interactions*, we will start to look at using Python to programmatically interact with network equipment.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



2

Low-Level Network Device Interactions

In *Chapter 1*, *Review of TCP/IP Protocol Suite and Python*, we looked at the theories and specifications behind network communication protocols. We also took a quick tour of the Python language. In this chapter, we will start to dive deeper into the management of network devices using Python. In particular, we will examine the different ways in which we can use Python to programmatically communicate with legacy network routers and switches.

What do I mean by legacy network routers and switches? While it's hard to imagine any networking device coming out today without an **application programming interface** (**API**) for programmatic communication, it is a known fact that many of the network devices deployed in previous years did not contain API interfaces. The intended method of management for those devices was through **command-line interfaces** (**CLIs**) using terminal programs, which were originally developed with a human engineer in mind. The management relied on the engineer's interpretation of the data returned from the device for appropriate action. As one can imagine, as the number of network devices and the complexity of the network grew, it became increasingly difficult to manually manage them one by one.

Python has several great libraries and frameworks that can help with these tasks, such as Pexpect, Paramiko, Netmiko, NAPALM, and Nornir, amongst others. It is worth noting that there are several overlaps between these libraries in terms of code, dependencies, and the maintainers of the projects. For example, the Netmiko library was created by Kirk Byers in 2014 based on the Paramiko SSH library. Carl Montanari created the Scrapli library to take advantage of the latest Python 3 asyncio concurrency features. In recent years, Kirk, Carl, David Barroso from the NAPALM project, and others teamed up to create the awesome Nornir framework to provide a pure Python network automation framework.

For the most part, the libraries are flexible enough to be used together or separately. For example, Ansible (covered in *Chapter 4*, *The Python Automation Framework – Ansible*) uses both Paramiko and Ansible-NAPALM as the underlying libraries for its network modules.

With so many libraries in existence today, it's not possible to cover all of them in a reasonable number of pages. In this chapter, we will cover Pexpect first, then move on with examples from Paramiko. Once we understand the basics and operations of Paramiko, it is easy to branch out to other libraries, such as Netmiko and NAPALM. In this chapter, we will take a look at the following topics:

- The challenges of the CLI
- Constructing a virtual lab
- The Python Pexpect library
- The Python Paramiko library
- Examples from other libraries
- The downsides of Pexpect and Paramiko

We have briefly discussed the shortfalls of managing network devices via the command-line interface. It has proven to be ineffective in network management with even moderate-sized networks. This chapter will introduce Python libraries that can work with that limitation. First, let us discuss some of the challenges with the CLI in more detail.

The challenges of the CLI

I started my IT career at an ISP help desk back in the early 2000s. I remember watching the network engineers typing in what seemed like cryptic commands into a text terminal. Like magic, the network devices would then bend to their will and behave in the way they intended. In time, I got to learn and embrace these magic commands that I could type into the terminal. As network engineers, these CLI-based commands are like secret codes we share with each other in this world we call network engineering. Manually typing in the command was just something we all had to do to get the job done, no harm, no foul.

However, it was right around the year 2014 when we started to see the industry coming to a consensus about the clear need to move away from manual, human-driven CLIs toward an automatic, computer-centric automation API. Make no mistake, we still need to directly communicate with the device when making network designs, bringing up an initial proof of concept, and deploying the topology for the first time. However, once the network is deployed, the network management requirement is now to consistently make the same changes reliably across all network devices.

These changes need to be error-free, and the engineers need to repeat the steps without being distracted or feeling tired. This requirement sounds like an ideal job for computers and our favorite programming language, Python.

Of course, if the network devices can only be managed with the command line, the main challenge becomes how we can replicate the previous manual interactions between the router and the administrator automatically with a computer program. In the command line, the router will output a series of information and will expect the administrator to enter a series of manual commands based on the engineer's interpretation of the output. For example, in a Cisco **Internetwork Operating System (IOS)** device, you have to type in *enable* to get into a privileged mode, and upon receiving the returned prompt with the # sign, you then type in `configure terminal` in order to go into the configuration mode. The same process can further be expanded into the interface configuration mode and routing protocol configuration mode. This is in sharp contrast to a computer-driven, programmatic mindset. When the computer wants to accomplish a single task, say, put an IP address on an interface, it wants to structurally give all the information to the router at once, and it would expect a single *yes* or *no* answer from the router to indicate the success or failure of the task.

The solution, as implemented by both Pexpect and Paramiko, is to treat the interactive process as a child process and watch over the interaction between the child process and the destination device. Based on the returned value, the parent process will decide the subsequent action, if any.

I am sure we are all anxious to get started on using the Python libraries, but first, we will need to construct our network lab in order to have a network to test our code against. We will begin by looking at different ways we can build our network labs.

Constructing a Virtual lab

Before we dive into the Python libraries and frameworks, let's examine the options of putting together a lab for the benefit of learning. As the old saying goes, "practice makes perfect" – we need an isolated sandbox to safely make mistakes, try out new ways of doing things, and repeat some of the steps to reinforce concepts that were not clear on the first try.

To put together a network lab, we basically have two options: physical devices or virtual devices. Let's look at the advantages and disadvantages of the respective options.

Physical devices

This option consists of putting together a lab with physical network devices that you can see and touch. If you are lucky enough, you might even be able to construct a lab that is an exact replication of your production environment. The advantages and disadvantages of a physical lab are as follows:

- **Advantages**: It is an easy transition from lab to production. The topology is easier to understand for managers and fellow engineers who can look at and work on the devices if need be. The comfort level with physical devices is extremely high because of familiarity.
- **Disadvantages**: It is relatively expensive to pay for devices that will only be used in a lab. Also, physical devices require engineering hours to rack and stack and are not very flexible once constructed.

Virtual devices

Virtual devices are emulations or simulations of actual network devices. They are either provided by the vendors or by the open source community. The advantages and disadvantages of virtual devices are as follows:

- **Advantages**: Virtual devices are easier to set up, relatively cheap, and can make changes to the topology quickly.
- **Disadvantages**: They are usually scaled-down versions of their physical counterparts. Sometimes there are feature gaps between the virtual and the physical device.

Of course, deciding on a virtual or physical lab is a personal decision derived from a trade-off between the cost, ease of implementation, and the risk of having a gap between lab and production environments. In some of the places I have worked, the virtual lab was used when doing an initial proof-of-concept, while the physical lab was used when we moved closer to the final design.

In my opinion, as more and more vendors decide to produce virtual appliances, the virtual lab is the way to proceed in a learning environment. The feature gap of the virtual appliance is relatively small and specifically documented, especially when the virtual instance is provided by the vendor. The cost of the virtual appliance is relatively small compared to buying physical devices. The time to build using virtual devices is much shorter because they are just software programs.

For this book, I will use a combination of physical and virtual devices for concept demonstration, with a preference for virtual devices. For the examples we will see, the differences should be transparent. If there are any known differences between the virtual and physical devices pertaining to our objectives, I will make sure to list them.

For the code examples in the book, I will try to make the network topology as simple as possible while still being able to demonstrate the concept at hand. Each virtual network usually consists of not more than a few nodes, and we will reuse the same virtual network for multiple labs if possible.

For the examples in this book, I will utilize Cisco Modeling Labs, `https://www.cisco.com/c/en/us/products/cloud-systems-management/modeling-labs/index.html`, as well as other virtual platforms, such as Arista vEOS. As we will see in the next section, Cisco provides CML in both a paid version and a free, hosted version on Cisco DevNet (`https://developer.cisco.com/site/devnet/`) based on availability. The use of CML is optional. You can use any lab devices you have, but it might make it easier to follow along with the book examples. Also worth noting is that Cisco has strict software license requirements for device images, so by purchasing or using the free hosted CML, you will be less likely to violate their software license requirements.

Cisco modeling labs

I remember when I first started to study for my **Cisco Certified Internetwork Expert** (**CCIE**) lab exam, I purchased some used Cisco equipment from eBay to study with. Even with a used equipment discount, each router and switch still cost hundreds of US dollars. To save money, I purchased some really outdated Cisco routers from the 1980s (search for Cisco AGS routers in your favorite search engine for a good chuckle), which significantly lacked features and horsepower, even for lab standards. As much as it made for an interesting conversation with family members when I turned them on (they were really loud), putting the physical devices together was not fun. They were heavy and clunky, and it was a pain to connect all the cables, and to introduce link failure, I would literally have to unplug a cable.

Fast-forward a few years. Dynamips was created, and I fell in love with how easy it was to create different network scenarios. This was especially important when trying to learn a new concept. All I needed was the IOS images from Cisco and a few carefully constructed topology files, and I could easily build a virtual network to test my knowledge on. I had a whole folder of network topologies, pre-saved configurations, and different versions of images, as called for by different scenarios. The addition of a GNS3 frontend gave the whole setup a beautiful GUI facelift. With GNS3, you can just click and drop your links and devices; you can even print out the network topology for your manager or client right out of the GNS3 design panel. The only disadvantage of GNS3 was the tool not being officially blessed by Cisco, and the perceived lack of credibility because of it.

In 2015, the Cisco community decided to fulfill this need by releasing the Cisco **Virtual Internet Routing Lab** (**VIRL**), `https://learningnetwork.cisco.com/s/virl`. This quickly became my go-to tool as the network lab when developing, learning, and practicing network automation code.

A few years after the introduction of VIRL, Cisco released **Cisco Modeling Labs (CML)**, `https://developer.cisco.com/modeling-labs/`. It is a great network simulation platform with an easy-to-use HTML UI and a comprehensive API.

At the time of writing, the single-user license for CML is 199 USD (keep in mind that there is a free, hosted version on Cisco DevNet). In my opinion, the CML platform offers a few advantages over other alternatives and the cost is a bargain:

- **Ease of use**: As mentioned, all the images for IOSv, IOS-XRv, NX-OSv, ASAv, and other images are included in a single download.
- **Official**: CML is a widely used tool internally at Cisco and within the network engineering community. In fact, CML is used extensively for the new Cisco DevNet Expert Lab exam. Because of its popularity, bugs get fixed quickly, new features are carefully documented, and useful knowledge is widely shared among its users.
- **Third-party KVM images integration**: CML permits users to upload third-party VM images, such as Windows VM, that are not bundled by default.
- **Others**: The CML tool offers many other features, such as dashboard list view, multiuser grouping, Ansible integration, and pyATS integration.

We will not use all of the CML features in this book, but it is nice to know the tool is so feature-rich and is constantly being updated. Again, I want to stress the importance of having a lab to follow along for the book examples but it does not need to be Cisco CML. The code examples provided in this book should work across any lab device, as long as it runs the same software type and version.

CML tips

The CML website (`https://developer.cisco.com/modeling-labs/`) and documentation (`https://developer.cisco.com/docs/modeling-labs/`) offer lots of guidance and information, from installation to usage. The lab topology will be included in the respective chapters in the book's GitHub repository (`https://github.com/PacktPublishing/Mastering-Python-Networking-Fourth-Edition`). The lab images can be directly imported to the lab via the **Import** button:



Figure 2.1: CML Console Image Lab Image

For the labs, each of the devices will have its management interface connected to an unmanaged switch, which in turn connects to an external connection for access:

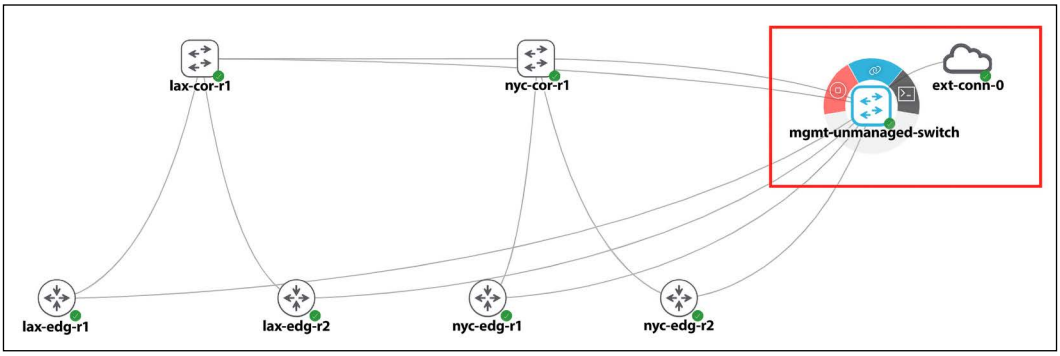


Figure 2.2: Unmanaged Switch for Management Interface Access

You will need to change the IP address of the management interface to fit your own lab's schema. For example, in the `2_DC_Topology.yaml` file in *Chapter 2*, the IP address of `lax-edg-r1 GigabitEthernet0/0 0` is `192.168.2.51`. You will need to change this IP address according to your own lab.

If you are using virtual lab software other than CML, you can open the topology file with any text editor (such as Sublime Text, shown below) and see each of the devices' configurations. You can then copy and paste the configuration into your own lab devices:

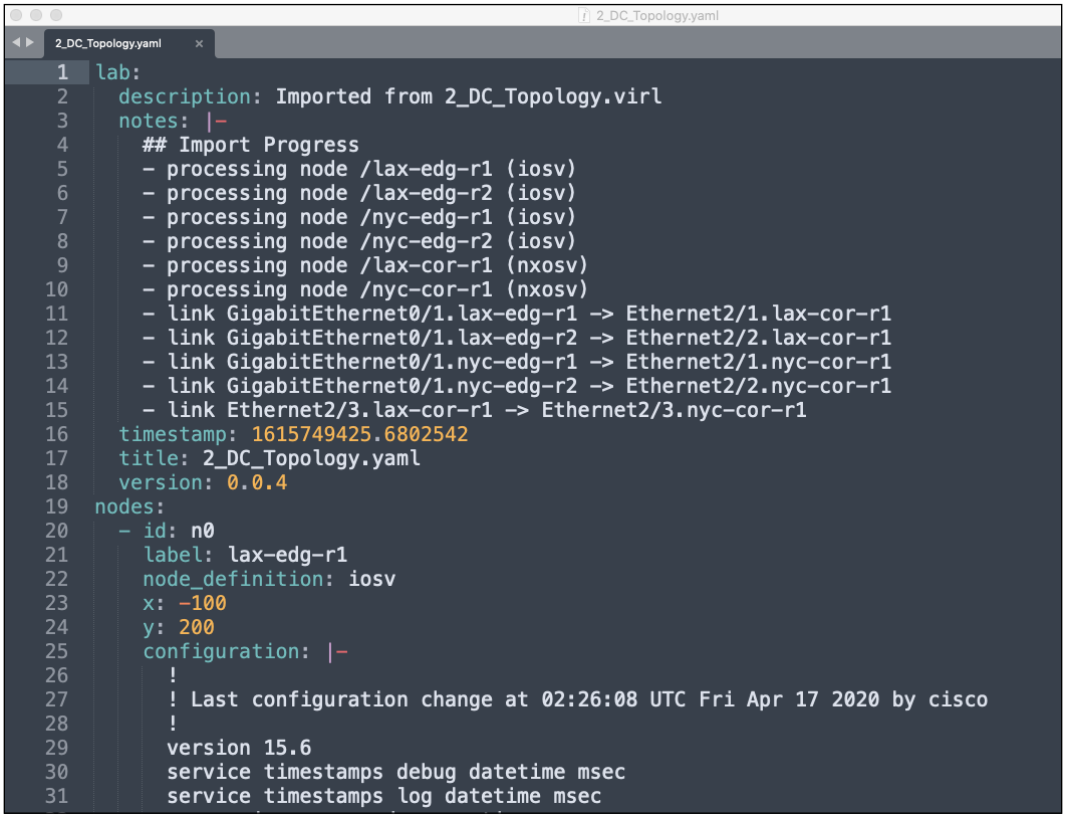


Figure 2.3: Topology File Viewed with Text Editor

We talked about Cisco DevNet briefly earlier in this section. Let us explore more about DevNet in the next section.

Cisco DevNet

Cisco DevNet (`https://developer.cisco.com/site/devnet/`) is the premier, all-in-one website when it comes to network automation resources at Cisco. It is free to sign up and provides free remote labs, free video courses, guided learning tracks, documentation, and much more.

The Cisco DevNet Sandbox (`https://developer.cisco.com/site/sandbox/`) is a great alternative if you do not already have a lab at your own disposal or want to try out new technologies. Some of the labs are always on, while others you need to reserve. The lab availability will depend on usage.

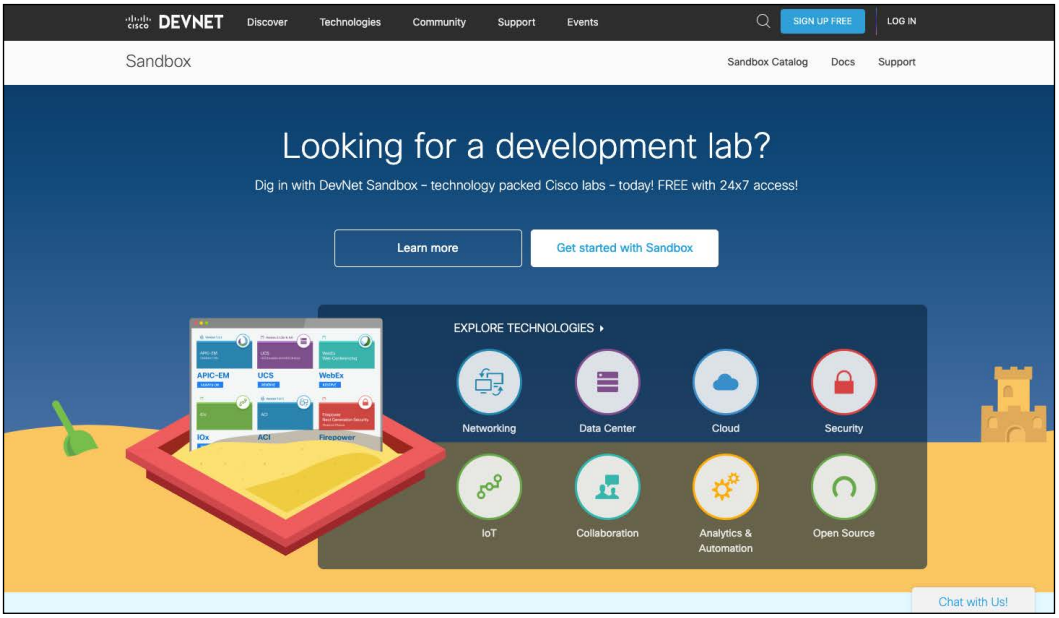


Figure 2.4: Cisco DevNet Sandbox



Since its inception, Cisco DevNet has become the de facto destination for all things related to network programmability and automation at Cisco. If you are interested in pursuing Cisco certifications in automation, DevNet offers different tracks from associate to expert level of validation; more information can be found at `https://developer.cisco.com/certification/`.

GNS3 and others

There are a few other virtual labs that I have used and would recommend. GNS3 is one of them:

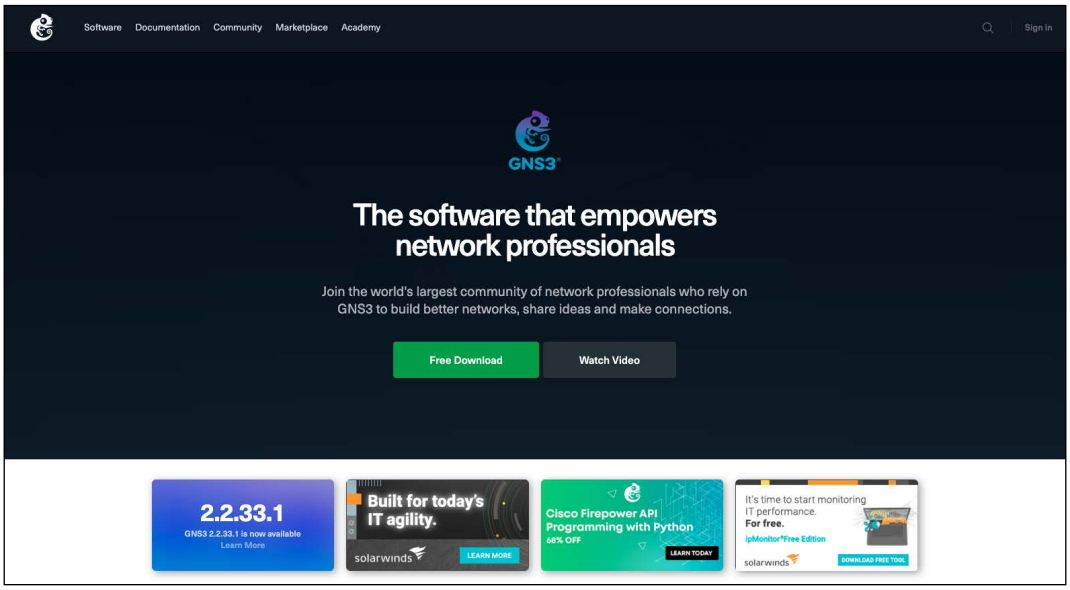


Figure 2.5: GNS3 Website

As mentioned previously, GNS3 is what a lot of us use to study for certification tests and to practice for labs. The tool has really grown up from the early days of being the simple frontend for Dynamips into a viable commercial product. GNS3 is vendor-neutral, which can be helpful if we want to build a multi-vendor lab. This is typically done either by making a clone of the image (such as Arista vEOS) or by directly launching the network device image via other hypervisors (such as KVM).

Another multi-vendor network emulation environment that has gotten a lot of great reviews is the **Emulated Virtual Environment Next Generation** (**Eve-NG**): `http://www.eve-ng.net/`. I personally do not have enough experience with the tool, but many of my colleagues and friends in the industry use it for their network labs. If you are familiar with containers, containerlab (`https://containerlab.dev/`) can also be an alternative for you.

There are also other standalone virtualized platforms, such as Arista vEOS (`https://www.arista.com/en/cg-veos-router/veos-router-overview`), Juniper vMX (`https://www.juniper.net/us/en/products/routers/mx-series/vmx-virtual-router-software.html`), and Nokia SR-Linux (`https://www.nokia.com/networks/data-center/service-router-linux-NOS/`), which you can use as standalone virtual appliances during testing.

They are great complementary tools for testing platform-specific features. Many of them are offered as paid products on public cloud provider marketplaces for easier access.

Now that we have built our network lab, we can start to experiment with Python libraries that can help with management and automation. We will begin with enabling the Python virtual environment. Then we will install and use the Pexpect library for some examples.

Python virtual environment

Let us start by using the Python virtual environment. The Python virtual environment allows us to manage separate package installations for different projects by creating a "virtual" isolated Python installation and installing packages into that virtual installation. By using a virtual environment, we do not need to worry about breaking the packages installed globally or from other virtual environments. We will start by installing the `python3.10-venv` package, then create the virtual environment itself:

```
$ sudo apt update
$ sudo apt install python3.10-venv
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
(venv) $ deactivate
```

From the output, we saw that we use the `venv` module from our installation, create a virtual environment called "venv" and then activate it. While the virtual environment is activated, you will see the (`venv`) label in front of your hostname, indicating that you are in that virtual environment. When finished, you can use the `deactivate` command to exit the virtual environment. If interested, you can learn more about Python virtual environments here: `https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/#installing-virtualenv`.



Always activate the virtual environment before you work on the code to isolate environments.

Once we have activated the virtual environment, we can move on to install the Pexpect library.

Python pexpect library

Pexpect is a pure Python module for spawning child applications, controlling them, and responding to expected patterns in their output. Pexpect works like Don Libes' `Expect`. Pexpect allows our script to spawn a child application and control it as if a human were typing commands; more information can be found on Pexpect's documentation page: `https://pexpect.readthedocs.io/en/stable/index.html`.



Nowadays, we typically use libraries, such as Nornir, that abstract this line-by-line, low-level interaction. However, it is still useful to understand the interaction at least at a high level. If you are the impatient kind, just skim through the following Pexpect and Paramiko sections.

Similar to the original **Tool Command Language** (**TCL**) `Expect` module by Don Libes, Pexpect launches, or spawns, another process and watches over it in order to control the interaction. The Expect tool was originally developed to automate interactive processes such as FTP, Telnet, and rlogin, and was later expanded to include network automation. Unlike the original Expect, Pexpect is entirely written in Python, which does not require TCL or C extensions to be compiled. This allows us to use the familiar Python syntax and its rich standard library in our code.

Pexpect installation

The Pexpect installation process is straightforward:

```
(venv) $ pip install pexpect
```

Let's do a quick test to make sure the package is usable; make sure we start the Python interactive shell from the virtual environment:

```
(venv) $ python
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pexpect
>>> dir(pexpect)
['EOF', 'ExceptionPexpect', 'Expecter', 'PY3', 'TIMEOUT', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__path__', '__revision__', '__spec__', '__
version__', 'exceptions', 'expect', 'is_executable_file', 'pty_spawn',
'run', 'runu', 'searcher_re', 'searcher_string', 'spawn', 'spawnbase',
```

```
 'spawnu', 'split_command_line', 'sys', 'utils', 'which']
    >>> exit()
```

Pexpect overview

For this chapter, we will use the `2_DC_Topology` and work on the two IOSv devices, **lax-edg-r1** and **lax-edg-r2**:

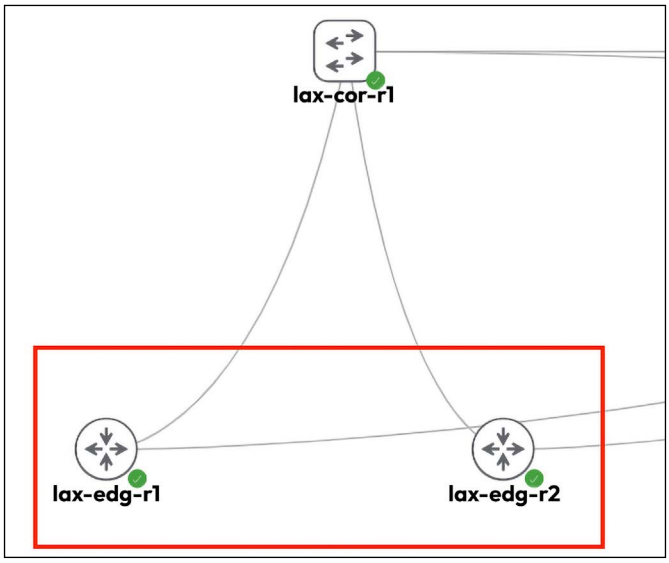


Figure 2.6: lax-edg-r1 and lax-edg-r2

The devices will each have a management address in the `192.16.2.x/24` range. In the example, `lax-edg-r1` will have `192.168.2.51` and `lax-edg-r2` will have `192.168.2.52` as the management IP. If this is the first time the device is powered up, it will need to generate an RSA key for SSH:

```
lax-edg-r2(config)#crypto key generate rsa
```

For older IOSv software images, we might also need to add the following lines to the `ssh` configuration (`~/.ssh/config`) depending on your platform:

```
Host 192.168.2.51
  HostKeyAlgorithms +ssh-rsa
  KexAlgorithms +diffie-hellman-group-exchange-sha1

Host 192.168.2.52
  HostKeyAlgorithms +ssh-rsa
  KexAlgorithms +diffie-hellman-group-exchange-sha1
```

With the devices ready, let's take a look at how you would interact with the router if you were to telnet into the device:

```
(venv) $ $ telnet 192.168.2.51
Trying 192.168.2.51...
Connected to 192.168.2.51.
Escape character is '^]'.
<skip>
User Access Verification
Username: cisco
Password:
```

The device configuration uses the username of `cisco`, and the password is also `cisco`. Notice that the user is already in the privileged mode because of the privilege assigned in the configuration:

```
lax-edg-r1#sh run | i cisco
enable password cisco
username cisco privilege 15 secret 5 $1$SXY7$Hk6z8OmtloIzFpyw6as2G.
 password cisco
 password cisco
```

The auto-config also generated `vty` access for both telnet and SSH:

```
line con 0
 password cisco
line aux 0
line vty 0 4
 exec-timeout 720 0
 password cisco
 login local
 transport input telnet ssh
```

Let's see a Pexpect example using the Python interactive shell:

```
>>> import pexpect
>>> child = pexpect.spawn('telnet 192.168.2.51')
>>> child.expect('Username')
0
>>> child.sendline('cisco')
6
```

```
>>> child.expect('Password')
0
>>> child.sendline('cisco')
6
>>> child.expect('lax-edg-r1#')
0
>>> child.sendline('show version | i V')
19
>>> child.before
b":
\r\n*************************************************************************\
r\n* IOSv is strictly limited to use for evaluation, demonstration and
IOS  *\r\n* education. IOSv is provided as-is and is not supported by
Cisco's      *\r\n* Technical Advisory Center. Any use or disclosure,
in whole or in part, *\r\n* of the IOSv Software or Documentation to
any third party for any       *\r\n* purposes is expressly prohibited
except as otherwise authorized by     *\r\n* Cisco in writing.
*\r\n*************************************************************************\
r\n"
>>> child.sendline('exit')
5
>>> exit()
```



Starting from Pexpect version 4.0, you can run Pexpect on the Windows platform. But, as noted in the Pexpect documentation, running Pexpect on Windows should be considered experimental for now.

In the previous interactive example, Pexpect spawns off a child process and watches over it in an interactive fashion. There are two important methods shown in the example, `expect()` and `sendline()`. The `expect()` line indicates the string in the Pexpect process looks for when the returned string is considered done. This is the expected pattern. In our example, we knew the router had sent us all the information when the hostname prompt (`lax-edg-r1#`) was returned. The `sendline()` method indicates which words should be sent to the remote device as the command. There is also a method called `send()`, but `sendline()` includes a linefeed, which is similar to pressing the *Enter* key at the end of the words you sent in the previous telnet session. From the router's perspective, it is just as if someone typed in the text from a Terminal. In other words, we are tricking the routers into thinking they are interfacing with a human being when they are actually communicating with a computer.

The `before` and `after` properties will be set to the text printed by the child application. The `before` properties will be set to the text printed by the child application up to the expected pattern. The `after` string will contain the text that was matched by the expected pattern. In our case, the `before` text will be set to the output between the two expected matches (`lax-edg-r1#`), including the `show version` command. The `after` text is the router hostname prompt:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('lax-edg-r1#')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor
board ID 9Y0KJ2ZL98EQQVUED5T2Q\r\n'
>>> child.after
b'iosv-1#'
```

If you are wondering about the `b'` in front of the return, it is a Python byte string (`https://docs.python.org/3.10/library/stdtypes.html`).

What would happen if you expected the wrong term? For example, if we typed in `username` with the lowercase "u" instead of `Username` after spawning the child application, the Pexpect process would look for a string of `username` from the child process. In that case, the Pexpect process would just hang because the word `username` would never be returned by the router. The session would eventually time out, or we could manually exit out via *Ctrl + C*.

The `expect()` method waits for the child application to return a given string, so in the previous example, if you wanted to accommodate both lowercase and uppercase `u`, you could use the following term:

```
>>> child.expect('[Uu]sername')
```

The square bracket serves as an `or` operation that tells the child application to expect a lowercase or uppercase "u" followed by `sername` as the string. What we are telling the process is that we will accept either `Username` or `username` as the expected string. For more information on these different types of matching using a regular expression, go to: `https://docs.python.org/3.10/library/re.html`.

The `expect()` method can also contain a list of options instead of just a single string; these options can also be regular expressions themselves. Going back to the previous example, we can use the following list of options to accommodate the two different possible strings:

```
>>> child.expect(['Username', 'username'])
```

Generally speaking, use the regular expression for a single `expect` string when we can fit the different letters in a regular expression, whereas use the possible options if we need to catch completely different responses from the device, such as a password rejection. For example, if we use several different passwords for our login, we want to catch `% Login invalid` as well as the device prompt.

One important difference between Pexpect regular expressions and Python regular expressions is that Pexpect matching is non-greedy, which means they will match as little as possible when using special characters. Because Pexpect performs regular expressions on a stream, it cannot look ahead, as the child process generating the stream may not be finished. This means the special dollar sign character `$` typically matching the end of the line is useless because `.+` will always return no characters, and the `.*` pattern will match as little as possible. In general, just keep this in mind and be as specific as you can be on the `expect` match strings.

Let's consider the following scenario:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('lax-edg-r1')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor
board ID 9Y0KJ2ZL98EQQVUED5T2Q\r\n'
>>>
```

Hmm... Something is not quite right here. Compare it to the Terminal output before; the output you expect would be `hostname lax-edg-r1`:

```
iosv-1#sh run | i hostname
hostname lax-edg-r1
```

Taking a closer look at the expected string will reveal the mistake. In this case, we were missing the hash (#) sign behind the `lax-edg-r1` hostname. Therefore, the child application treated the second part of the return string as the expected string:

```
>>> child.sendline('show run | i hostname')
22
>>> child.expect('lax-edg-r1#')
0
>>> child.before
b'#show run | i hostname\r\nhostname lax-edg-r1\r\n'
```

You can see a pattern emerging from the usage of Pexpect after a few examples. The user maps out the sequence of interactions between the Pexpect process and the child application. With some Python variables and loops, we can start to construct a useful program that will help us gather information and make changes to network devices.

Our first Pexpect program

Our first program, `chapter2_1.py`, extends what we did in the last section with some additional code:

```
#!/usr/bin/env python

import pexpect

devices = {'iosv-1': {'prompt': 'lax-edg-r1#', 'ip': '192.168.2.51'},
           'iosv-2': {'prompt': 'lax-edg-r2#', 'ip': '192.168.2.52'}}
username = 'cisco'
password = 'cisco'

for device in devices.keys():
    device_prompt = devices[device]['prompt']
    child = pexpect.spawn('telnet ' + devices[device]['ip'])
    child.expect('Username:')
    child.sendline(username)
    child.expect('Password:')
    child.sendline(password)
    child.expect(device_prompt)
    child.sendline('show version | i V')
```

```
    child.expect(device_prompt)
    print(child.before)
    child.sendline('exit')
```

We used a nested dictionary in line 5:

```
devices = {'iosv-1': {'prompt': 'lax-edg-r1#', 'ip': '192.168.2.51'},
           'iosv-2': {'prompt': 'lax-edg-r2#', 'ip': '192.168.2.52'}}
```

The nested dictionary allows us to refer to the same device (such as `lax-edg-r1`) with the appropriate IP address and prompt symbol. We can then use those values for the `expect()` method later on in the loop.

The output prints out the `show version | i V` output on the screen for each of the devices:

```
$ python chapter2_1.py
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor
board ID 98U40DKV403INHIULHYHB\r\n'
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\n'
```

Now that we have seen a basic example of Pexpect, let us go deeper into more features of the library.

More Pexpect Features

In this section, we will look at more Pexpect features that might come in handy when certain situations arise.

If you have a slow or fast link to your remote device, the default `expect()` method timeout is 30 seconds, which can be increased or decreased via the `timeout` argument:

```
>>> child.expect('Username', timeout=5)
```

You can choose to pass the command back to the user using the `interact()` method. This is useful when you just want to automate certain parts of the initial task:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('lax-edg-r1#')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
```

```
-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor board ID
98U40DKV403INHIULHYHB\r\n'
>>> child.interact()
show version | i V
Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version
15.8(3)M2, RELEASE SOFTWARE (fc2)
Processor board ID 98U40DKV403INHIULHYHB
lax-edg-r1#exit
Connection closed by foreign host.
>>>
```

You can get a lot of information about the `child.spawn` object by printing it out in string format:

```
>>> str(child)
"<pexpect.pty_spawn.spawn object at 0x7f068a9bf370>\ncommand: /usr/bin/
telnet\nargs: ['/usr/bin/telnet', '192.168.2.51']\nbuffer (last 100
chars): b''\nbefore (last 100 chars): b'TERPRISEK9-M), Version 15.8(3)
M2, RELEASE SOFTWARE (fc2)\\r\\nProcessor board ID 98U40DKV403INHIULHYHB\\
r\\n'\nafter: b'lax-edg-r1#'\nmatch: <re.Match object; span=(165, 176),
match=b'lax-edg-r1#'>\nmatch_index: 0\nexitstatus: 1\nflag_eof: False\
npid: 25510\nchild_fd: 5\nclosed: False\ntimeout: 30\ndelimiter: <class
'pexpect.exceptions.EOF'>\nlogfile: None\nlogfile_read: None\nlogfile_
send: None\nmaxread: 2000\nignorecase: False\nsearchwindowsize: None\
ndelaybeforesend: 0.05\ndelayafterclose: 0.1\ndelayafterterminate: 0.1"
>>>
```

The most useful debug tool for Pexpect is to log the output in a file:

```
>>> child = pexpect.spawn('telnet 192.168.2.51')
>>> child.logfile = open('debug', 'wb')
```

For more information on Pexpect features, check out: `https://pexpect.readthedocs.io/en/stable/api/index.html`

We have been working with Telnet so far in our examples, which leaves our communication in clear text during the session. In modern networks, we typically use **secure shell** (**SSH**) for management. In the next section, we will take a look at Pexpect with SSH.

Pexpect and SSH

Pexpect has a subclass called `pxssh`, which specializes in setting up SSH connections. The class adds methods for login, logout, and various tricky things to handle the different situations in the `ssh` login process. The procedures are mostly the same, with the exception of `login()` and `logout()`:

```
>>> from pexpect import pxssh
>>> child = pxssh.pxssh()
>>> child.login('192.168.2.51', 'cisco', 'cisco', auto_prompt_reset=False)
True
>>> child.sendline('show version | i V')
19
>>> child.expect('lax-edg-r1#')
0
>>> child.before
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor
board ID 98U40DKV403INHIULHYHB\r\n'
>>> child.logout()
>>>
```

Notice the `auto_prompt_reset=False` argument in the `login()` method. By default, `pxssh` uses the shell prompt to synchronize the output. But since it uses the PS1 option for most of bash-shell or c-shell, they will error out on Cisco or other network devices.

Pexpect complete example

As the final step, let's put everything you have learned so far about Pexpect into a script. Putting code into a script makes it easier to use in a production environment, as well as easier to share with your colleagues. We will write our second script, `chapter2_2.py`:

```
#!/usr/bin/env python

import getpass
from pexpect import pxssh

devices = {'lax-edg-r1': {'prompt': 'lax-edg-r1#', 'ip': '192.168.2.51'},
           'lax-edg-r2': {'prompt': 'lax-edg-r2#', 'ip': '192.168.2.52'}}
```

```
commands = ['term length 0', 'show version', 'show run']

username = input('Username: ')
password = getpass.getpass('Password: ')

# Starts the loop for devices
for device in devices.keys():
    outputFileName = device + '_output.txt'
    device_prompt = devices[device]['prompt']
    child = pxssh.pxssh()
    child.login(devices[device]['ip'], username.strip(), password.strip(),
auto_prompt_reset=False)
    # Starts the loop for commands and write to output
    with open(outputFileName, 'wb') as f:
        for command in commands:
            child.sendline(command)
            child.expect(device_prompt)
            f.write(child.before)

    child.logout()
```

The script further expands from our first Pexpect program with the following additional features:

- It uses SSH instead of Telnet.
- It supports multiple commands instead of just one by making the commands into a list (line 8) and loops through the commands (starting at line 20).
- It prompts the user for their username and password instead of hardcoding them in the script for better security posture.
- It writes the output in two files, `lax-edg-r1_output.txt` and `lax-edg-r2_output.txt`.

After the code is executed, we should see the two output files in the same directory. Besides Pexpect, Paramiko is another popular Python library used to handle interactive sessions.

The Python Paramiko library

Paramiko is a Python implementation of the SSHv2 protocol. Just like the `pxssh` subclass of Pexpect, Paramiko simplifies the SSHv2 interaction between the host and the remote device. Unlike `pxssh`, Paramiko focuses only on SSHv2 with no Telnet support. It also provides both client and server operations.

Paramiko is the low-level SSH client behind the high-level automation framework Ansible for its network modules. We will cover Ansible in *Chapter 4*, *The Python Automation Framework – Ansible*. Let's take a look at the Paramiko library.

Installation of Paramiko

Installing Paramiko is pretty straightforward with Python `pip`. However, there is a hard dependency on the cryptography library. The library provides low-level, C-based encryption algorithms for the SSH protocol.

The installation instruction for Windows, macOS, and other flavors of Linux can be found at: `https://cryptography.io/en/latest/installation/`.

We will show the Paramiko installation steps for our Ubuntu 22.04 virtual machine:

```
sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
pip install cryptography
pip install paramiko
```

Let us test the library's usage by importing it with the Python interpreter:

```
$ python
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>> exit()
```

Now we are ready to take a look at Paramiko in the next section.

Paramiko overview

Let's look at a quick Paramiko example using the Python 3 interactive shell:

```
>>> import paramiko, time
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('192.168.2.51', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> new_connection = connection.invoke_shell()
>>> output = new_connection.recv(5000)
>>> print(output) b"\
r\n**********************************************************************\
```

```
r\n* IOSv is strictly limited to use for evaluation, demonstration and
IOS  *\r\n* education. IOSv is provided as-is and is not supported by
Cisco's      *\r\n* Technical Advisory Center. Any use or disclosure,
in whole or in part, *\r\n* of the IOSv Software or Documentation to
any third party for any       *\r\n* purposes is expressly prohibited
except as otherwise authorized by     *\r\n* Cisco in writing.
*\r\n*************************************************************
**\r\nlax-edg-r1#"
>>> new_connection.send("show version | i V\n")
19
>>> time.sleep(3)
>>> output = new_connection.recv(5000)
>>> print(output)
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\nProcessor
board ID 98U40DKV403INHIULHYHB\r\nlax-edg-r1#'
>>> new_connection.close()
>>>
```

The `time.sleep()` function inserts a time delay to ensure all the outputs were captured. This is particularly useful on a slower network connection or a busy device. This command is not required but is recommended depending on your situation.

Even if we are seeing the Paramiko operation for the first time, the beauty of Python and its clear syntax means that we can make a pretty good educated guess at what the program is trying to do:

```
>>> import paramiko
>>> connection = paramiko.SSHClient()
>>> connection.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> connection.connect('192.168.2.51', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
```

The first four lines create an instance of the `SSHClient` class from Paramiko. The next line sets the policy that the client should use regarding keys; in this case, `lax-edg-r1` might not be in either the system host keys or the application's keys. In our scenario, we will automatically add the key to the application's `HostKeys` object. At this point, if you log on to the router, you will see all the login sessions from Paramiko.

The next few lines invoke a new interactive shell from the connection and a repeatable pattern of sending a command and retrieving the output. Finally, we close the connection.

Some readers who have used Paramiko before might be familiar with the `exec_command()` method instead of invoking a shell. Why do we need to invoke an interactive shell instead of using `exec_command()` directly? Unfortunately, `exec_command()` on Cisco IOS only allows a single command. Consider the following example with `exec_command()` for the connection:

```
>>> connection.connect('192.168.2.51', username='cisco', password='cisco',
look_for_keys=False, allow_agent=False)
>>> stdin, stdout, stderr = connection.exec_command('show version | i
V\n')
>>> stdout.read()
b'Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.8(3)M2, RELEASE SOFTWARE (fc2)rnProcessor board ID
98U40DKV403INHIULHYHBrn'
>>>
```

Everything works great; however, if you look at the number of sessions on the Cisco device, you will notice that the connection is dropped by the Cisco device without you closing the connection. Because the SSH session is no longer active, `exec_command()` will return an error if you want to send more commands to the remote device:

```
>>> stdin, stdout, stderr = connection.exec_command('show version | i
V\n')
Traceback (most recent call last):
<skip>
raise SSHException('SSH session not active') paramiko.ssh_exception.
SSHException: SSH session not active
>>>
```

In the previous example, the `new_connection.recv()` command displayed what was in the buffer and implicitly cleared it out for us. What would happen if you did not clear out the received buffer? The output would just keep on filling up the buffer and would overwrite it:

```
>>> new_connection.send("show version | i V\n")
 19
>>> new_connection.send("show version | i V\n")
19
>>> new_connection.send("show version | i V\n")
19
>>> new_connection.recv(5000)
```

```
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)rnProcessor
board ID 98U40DKV403INHIULHYHBrnlax-edg-r1#show version | i VrnCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)rnProcessor board ID 98U40DKV403INHIULHYHBrnlax-
edg-r1#show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)rnProcessor
board ID 98U40DKV403INHIULHYHBrnlax-edg-r1#'
>>>
```

For consistency of the deterministic output, we will retrieve the output from the buffer each time we execute a command.

First Paramiko program

Our first program will use the same general structure as the Pexpect program we have put together. We will loop over a list of devices and commands while using Paramiko instead of Pexpect. This will give us a good compare and contrast of the differences between Paramiko and Pexpect.

If you have not done so already, you can download the code, `chapter2_3.py`, from the book's GitHub repository at `https://github.com/PacktPublishing/Mastering-Python-Networking-Fourth-Edition`. I will list the notable differences here:

```
devices = {'lax-edg-r1': {'ip': '192.168.2.51'},
           'lax-edg-r2': {'ip': '192.168.2.52'}}
```

We no longer need to match the device prompt using Paramiko; therefore, the device dictionary can be simplified:

```
commands = ['show version', 'show run']
```

There is no sendline equivalent in Paramiko; instead, we manually include the newline break in each of the commands:

```
def clear_buffer(connection):
    if connection.recv_ready():
        return connection.recv(max_buffer)
```

We include a new method to clear the buffer for sending commands, such as `terminal length 0` or `enable`, because we do not need the output for those commands. We simply want to clear the buffer and get to the execution prompt. This function will later be used in the loop, such as in line 25 of the script:

```
output = clear_buffer(new_connection)
```

The rest of the program should be pretty self-explanatory, similar to what we have seen in this chapter. The last thing I would like to point out is that since this is an interactive program, we place a buffer and wait for the command to be finished on the remote device before retrieving the output:

```
time.sleep(5)
```

After we clear the buffer, we will wait five seconds between the execution of commands. This will give the device adequate time to respond if it is busy.

More Paramiko features

We will look at Paramiko a bit later in *Chapter 4*, *The Python Automation Framework – Ansible,* when we discuss Ansible, as Paramiko is the underlying transport for many of the network modules. In this section, we will take a look at some of the other features of Paramiko.

Paramiko for servers

Paramiko can be used to manage servers through SSHv2 as well. Let's look at an example of how we can use Paramiko to manage servers. We will use key-based authentication for the SSHv2 session.

In this example, I used another Ubuntu virtual machine on the same hypervisor as the destination server. You can also use a server on the CML simulator or an instance in one of the public cloud providers, such as Amazon AWS EC2.

We will generate a public-private key pair for our Paramiko host:

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

This command, by default, will generate a public key named `id_rsa.pub`, as the public key under the user home directory `~/.ssh` along with a private key named `id_rsa`. Treat the private key with the same attention as you would for private passwords that you do not want to share with anybody else.

You can think of the public key as a business card that identifies who you are. Using the private and public keys, the message will be encrypted by your private key locally and decrypted by the remote host using the public key. We should copy the public key to the remote host. In production, we can do this via out-of-band using a USB drive; in our lab, we can simply copy the public key to the remote host's `~/.ssh/authorized_keys` file. Open up a Terminal window for the remote server so you can paste in the public key.

Copy the content of `~/.ssh/id_rsa.pub` on your management host with Paramiko:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa <your public key>
```

Then, paste it to the remote host under the `user` directory; in this case, I am using `echou` for both sides:

```
<Remote Host>$ vim ~/.ssh/authorized_keys
ssh-rsa <your public key>
```

You are now ready to use Paramiko to manage the remote host. Notice in this example that we will use the private key for authentication as well as the `exec_command()` method for sending commands:

```
>>> import paramiko
>>> key = paramiko.RSAKey.from_private_key_file('/home/echou/.ssh/id_rsa')
>>> client = paramiko.SSHClient()
>>> client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> client.connect('192.168.199.182', username='echou', pkey=key)
>>> stdin, stdout, stderr = client.exec_command('ls -l')
>>> stdout.read()
b'total 44ndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Desktopndrwxr-xr-x 2
echou echou 4096 Jan 7 10:14 Documentsndrwxr-xr-x 2 echou echou 4096 Jan 7
10:14 Downloadsn-rw-r--r-- 1 echou echou 8980 Jan 7 10:03
examples.desktopndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Musicndrwxr-
xr-x
echou echou 4096 Jan 7 10:14 Picturesndrwxr-xr-x 2 echou echou 4096 Jan 7
10:14 Publicndrwxr-xr-x 2 echou echou 4096 Jan 7 10:14 Templatesndrwxr-
xr-x
2 echou echou 4096 Jan 7 10:14 Videosn'
>>> stdin, stdout, stderr = client.exec_command('pwd')
>>> stdout.read()
b'/home/echou'
>>> client.close()
>>>
```

Notice that in the server example, we do not need to create an interactive session to execute multiple commands. You can now turn off password-based authentication in your remote host's SSHv2 configuration for more secure key-based authentication with automation enabled.

Why do we want to know about using private keys as authentication methods? More and more network devices, such as Cumulus and Vyatta switches, are moving toward using Linux shell and public-private key authentication as a security mechanism. For some operations, we will use a combination of SSH session and key-based authentication for authentication.

More Paramiko examples

In this section, let's make the Paramiko program more reusable. There is one downside of our existing script: we need to open up the script every time we want to add or delete a host, or whenever we need to change the commands we want to execute on the remote host.

This is due to the fact that both the host and command information are statically entered inside the script. Hardcoding the host and command has a higher chance of making mistakes when making a change. By making both the host and command files read in as parameters for the script, we can make the script more flexible. Users (and future us) can simply modify these text files when you need to make host or command changes.

We have incorporated the change in the script named `chapter2_4.py`.

Instead of hardcoding the commands, we broke the commands into a separate `commands.txt` file. Up to this point, we have been using `show` commands; in this example, we will make configuration changes. In particular, we will change the logging buffer size to `30000` bytes:

```
$ cat commands.txt
config t
logging buffered 30000
end
copy run start
```

The device's information is written into a `devices.json` file. We chose JSON format for the device's information because JSON data types can be easily translated into Python dictionary data types:

```
$ cat devices.json
{
    "lax-edg-r1": {
        "ip": "192.168.2.51"
    },
    "lax-edg-r2": {
        "ip": "192.168.2.52"
    }
}
```

In the script, we made the following changes:

```
with open('devices.json', 'r') as f:
    devices = json.load(f)
with open('commands.txt', 'r') as f:
    commands = f.readlines()
```

Here is an abbreviated output from the script execution:

```
$ python chapter2_4.py
Username: cisco
Password:
b'terminal length 0\r\nlax-edg-r1#config t\r\nEnter configuration
commands, one per line.  End with CNTL/Z.\r\nlax-edg-r1(config)#'
b'logging buffered 30000\r\nlax-edg-r1(config)#'
b'end\r\nlax-edg-r1#'
b'copy run start'
<skip>
```

Do a quick check to make sure the change has taken place in both `running-config` and `startup-config`:

```
lax-edg-r1#sh run | i logging
logging buffered 30000
```

The Paramiko library is a general-purpose library intended for working with interactive command-line programs. For network management, there is another library, Netmiko, a fork from Paramiko, that is purpose-built for network device management. We will take a look at it in the upcoming section.

The Netmiko library

Paramiko is a great library to do low-level interactions with Cisco IOS and other vendor devices. But as you have noticed from previous examples, we are repeating many of the same steps between `lax-edg-r1` and `lax-edg-r2` for device login and execution. Once we start to develop more automation commands, we also start to repeat ourselves in capturing terminal outputs and formatting them into a usable format. Wouldn't it be great if somebody could write a Python library that simplifies these low-level steps and share it with other network engineers?

Ever since 2014, Kirk Byers (`https://github.com/ktbyers`) has been working on open-source initiatives to simplify the management of network devices. In this section, we will take a look at an example of the Netmiko (`https://github.com/ktbyers/netmiko`) library that he created.

First, we will install the `netmiko` library using `pip`:

```
(venv) $ pip install netmiko
```

We can use the example published on Kirk's website, `https://pynet.twb-tech.com/blog/automation/netmiko.html`, and apply it to our labs. We will start by importing the library and its `ConnectHandler` class. Then we will define our `device` parameter as a Python dictionary and pass it to the `ConnectHandler`. Notice that we are defining a `device_type` of `cisco_ios` in the `device` parameter:

```
>>> from netmiko import ConnectHandler
>>> net_connect = ConnectHandler(
...     device_type="cisco_ios",
...     host="192.168.2.51",
...     username="cisco",
...     password="cisco",
... )
```

This is where the simplification begins. Notice that the library automatically determines the device prompt as well as formatting the returned output from the `show` command:

```
>>> net_connect.find_prompt()
'lax-edg-r1#'
>>> output = net_connect.send_command('show ip int brief')
>>> print(output)
Interface                  IP-Address      OK? Method Status
Protocol
GigabitEthernet0/0         192.168.2.51    YES NVRAM  up
up
GigabitEthernet0/1         10.0.0.1        YES NVRAM  up
up
Loopback0                  192.168.0.10    YES NVRAM  up
up
```

Let's see another example for the second Cisco IOS device in our lab and send a `configuration` command instead of a `show` command. Note that the `command` attribute is a list that can contain multiple commands:

```
>>> net_connect_2 = ConnectHandler(
...     device_type="cisco_ios",
...     host="192.168.2.52",
...     username="cisco",
...     password="cisco",
... )
>>> output = net_connect_2.send_config_set(['logging buffered 19999'])
>>> print(output)
configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
lax-edg-r2(config)#logging buffered 19999
lax-edg-r2(config)#end
lax-edg-r2#
>>> exit()
```

How cool is that? Netmiko automatically took care of the nitty-gritty stuff for us, allowing us to focus on the commands themselves. The `netmiko` library is a great time saver and is used by many network engineers. In the next section, we will take a look at the Nornir (`https://github.com/nornir-automation/nornir`) framework, which also aims to simplify low-level interactions.

The Nornir framework

Nornir (`https://nornir.readthedocs.io/en/latest/`) is a pure Python automation framework intended to be used directly from Python. We will start with installing `nornir` in our environment:

```
(venv)$ pip install nornir nornir_utils nornir_netmiko
```

Nornir expects us to define an inventory file, `hosts.yaml`, consisting of the device information in a YAML format. The information specified in this file is no different than what we have previously defined using the Python dictionary in the Netmiko example:

```
---
lax-edg-r1:
    hostname: '192.168.2.51'
    port: 22
    username: 'cisco'
```

```
    password: 'cisco'
    platform: 'cisco_ios'

lax-edg-r2:
    hostname: '192.168.2.52'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'
```

We can use the `netmiko` plugin from the `nornir` library to interact with our device, as illustrated in the `chapter2_5.py` file:

```
#!/usr/bin/env python

from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_command


nr = InitNornir()

result = nr.run(
    task=netmiko_send_command,
    command_string="show arp"
)

print_result(result)
```

The execution output is shown as follows:

```
(venv) $ python chapter2_5.py

netmiko_send_
command************************************************************
* lax-edg-r1 ** changed : False ****************************************
******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
```

```
Internet  10.0.0.1               -   5254.001e.e911  ARPA
GigabitEthernet0/1
Internet  10.0.0.2              17   fa16.3e00.0001  ARPA
GigabitEthernet0/1
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^
* lax-edg-r2 ** changed : False ***************************************
******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.128.1             17   fa16.3e00.0002  ARPA
GigabitEthernet0/1
Internet  10.0.128.2              -   5254.0014.e052  ARPA
GigabitEthernet0/1
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^
```



There are other plugins in Nornir besides Netmiko, such as the popular NAPALM library (`https://github.com/napalm-automation/napalm`). Please feel free to check out Nornir's project page for the latest plugins: `https://nornir.readthedocs.io/en/latest/plugins/index.html`.

We have taken a pretty huge leap forward in this chapter in automating our network using Python. However, some of the methods we have used feel like workarounds for automation. We attempted to trick the remote devices into thinking they were interacting with a human on the other end. Even if we use libraries such as Netmiko or the Nornir framework, the underlying approach remains the same. Just because somebody else has done the work to help abstract the grunt work of the low-level interaction, we are still susceptible to the downsides of dealing with CLI-only devices.

Looking ahead, let us discuss some of the downsides of Pexpect and Paramiko compared to other tools in preparation for our discussion on API-driven methods in the next chapters.

Downsides of Pexpect and Paramiko compared to other tools

The biggest downside of our current method for automating CLI-only devices is that the remote devices do not return structured data. They return data that is ideal for fitting on a terminal to be interpreted by a human, not by a computer program. The human eye can easily interpret a space, while a computer only sees a return character.

We will take a look at a better way in the upcoming chapter. As a prelude to *Chapter 3*, *APIs and Intent-Driven Networking,* let's discuss the idea of idempotency.

Idempotent network device interaction

The term *idempotency* has different meanings, depending on its context. But in this book's context, the term means that when a client makes the same call to a remote device, the result should always be the same. I believe we can all agree that this is necessary. Imagine a scenario where each time you execute the same script, you get a different result back. I find that scenario very scary. How can you trust your script if that is the case? It would render our automation effort useless because we need to be prepared to handle different returns.

Since Pexpect and Paramiko are blasting out a series of commands interactively, the chance of having a non-idempotent interaction is higher. Going back to the fact that the return results needed to be screen scraped for useful elements, the risk of difference is much higher. Something on the remote end might have changed between the time we wrote the script and the time when the script is executed for the 100th time. For example, if the vendor makes a screen output change between releases without us updating the script, the script might break our network.

If we need to rely on the script for production, we need the script to be idempotent as much as possible.

Bad automation speeds up bad things

Bad automation allows you to poke yourself in the eye a lot faster, it is as simple as that. Computers are much faster at executing tasks than human engineers. If we had the same set of operating procedures executed by a human versus a script, the script would finish faster than humans, sometimes without the benefit of having a solid feedback loop between steps. The internet is full of horror stories of when someone pressed the *Enter* key and immediately regretted it.

We need to minimize the chances of bad automation scripts screwing things up. We all make mistakes; carefully testing your script before any production work and having a small blast radius are two keys to making sure you can catch your mistake before it comes back and bites you. No tool or human can eliminate mistakes completely, but we can strive to minimize the mistakes. As we have seen, as great as some of the libraries we have used in this chapter are, the underlying CLI-based method is inherently faulty and error-prone. We will introduce the API-driven method in the next chapter, which addresses some of the CLI-driven management deficiencies.

Summary

In this chapter, we covered low-level ways to communicate directly with network devices. Without a way to programmatically communicate and make changes to network devices, there is no automation. We looked at several libraries in Python that allow us to manage devices that were meant to be managed by the CLI. Although useful, it is easy to see how the process can be somewhat fragile. This is mostly due to the fact that the network gear in question was meant to be managed by human beings and not computers.

In *Chapter 3*, *APIs and Intent-Driven Networking*, we will look at network devices supporting API and intent-driven networking.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



3

APIs and Intent-Driven Networking

In *Chapter 2*, *Low-Level Network Device Interactions*, we looked at ways to interact with network devices using Python libraries such as Pexpect, Paramiko, Netmiko, and Nornir. Paramiko and similar libraries use persistent sessions that simulate a user typing in commands as if sitting in front of a Terminal. This approach works fine up to a point. It is easy to send commands 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 Paramiko, for example, 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.

The key point is: 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.

This is a similar issue faced by the internet community in general, not just with network automation, whenever computers need to communicate with each other. Imagine the difference between a computer and a human being when they both read 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. Essentially, the web server needs to transfer information to another computer in the most optimized way. How do we do that?

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. As defined by 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, such as our Python scripts. The exact API implementation is vendor and sometimes product specific. One vendor will prefer XML, while another might use JSON; one product might use HTTPS as the underlying transport protocol, and others might provide Python libraries called SDKs to be used with the device. We will see many different vendors and product examples in this chapter.

Despite the differences, the concept 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, **Application Centric Infrastructure** (**ACI**), and Meraki examples
- 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 (IaC)

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, 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, 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? No doubt I would have had to call the office and ask a more senior engineer for help.

Network engineering is not about typing commands into a device; 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 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 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, IDN 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 we purchase from a vendor.

When using an API, it is my opinion that it gets us closer to a state of IDN. 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 router and `filter-list` on a Juniper router. However, by 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*. 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:

```
lax-edg-r1#sh ip int brief
Interface                  IP-Address      OK? Method Status
Protocol
GigabitEthernet0/0         192.168.2.51    YES NVRAM  up
up
GigabitEthernet0/1         10.0.0.1        YES NVRAM  up
up
Loopback0                  192.168.0.10    YES NVRAM  up
```

The line break, white spaces, and the first line of the column title are easily distinguished by the human eye. They are there to help us line up, say, the IP addresses of each interface from line one to lines two and three. If we were to put ourselves in the computer's position for capturing the information, all these spaces and line breaks only take us away from the 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 `show interface brief` command:

```
>>> new_connection.send('show ip int brief/n')
16
>>> output = new_connection.recv(5000)
>>> print(output)
b'show ip interface brief\r\nInterface                  IP-
Address      OK? Method Status                Protocol\r\
nGigabitEthernet0/0         192.168.2.51    YES NVRAM  up
up      \r\nGigabitEthernet0/1         10.0.0.1        YES NVRAM  up
up      \r\nLoopback0                  192.168.0.10    YES NVRAM  up
up      \r\nlax-edg-r1#'
>>>
```

If we were to parse out that data contained in the output variable, this 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 discard it.
3. Take out everything on the second line up until the hostname prompt, and save it in a variable.
4. For the rest of the lines, because we do not know how many interfaces there are, we will use a regular expression statement to search if the line starts with interface names, such as `lo` for loopback and `GigabitEthernet` 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 while space to give us the protocol, the link, and the admin status.

Whew, that is a lot of work just for something that a human being can tell at a glance! The steps are what we need to do when we need to screen scrap unstructured texts. 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 we typically run.
- **Predictability**: There is 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**: Once we spend all this effort parsing the output for this particular vendor and software version, in this case, Cisco IOS, we need to repeat this process for the next vendor 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 (the full output is in the course code repository):

```
{
"ins_api":{
"outputs":{
```

```
"output":{
"body":{ "TABLE_intf":[
{
"ROW_intf":{
"admin-state":"up",
"intf-name":"Lo0",
"iod":84,
"ip-disabled":"FALSE",
"link-state":"up",
"prefix":"192.168.2.50",
"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. Right away, we 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, no extensive parsing is required — we can simply pick the key and retrieve the value associated with the key. We can also see from the output that there is 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. We no longer need to keep track of the command 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, we can easily ingest the information and build our infrastructure automation around it. It is agreed by all that network automation and programmability are beneficial to both network vendors and operators. The questions are usually about the transport, format, and structure of the automation messages. As we 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.

Data modeling for IaC

According to Wikipedia (`https://en.wikipedia.org/wiki/Data_model`), the definition of 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:

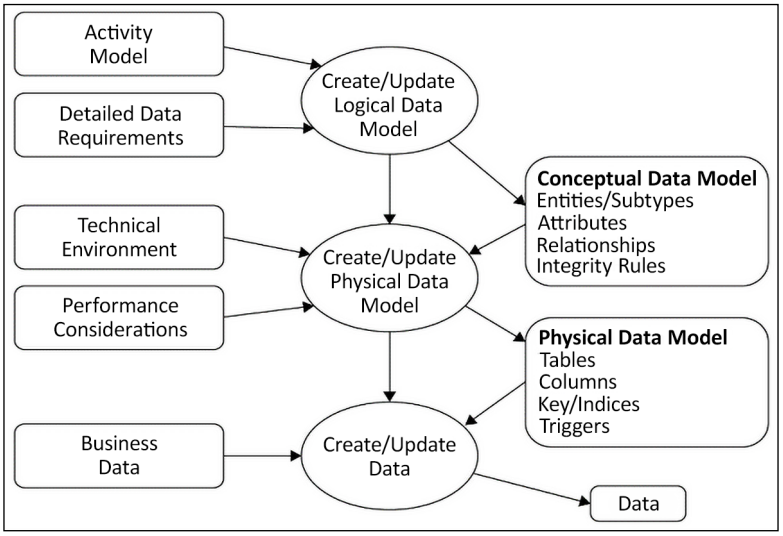


Figure 3.1: Data example process

When applying the data model concept to the network, we can say the network data model is an abstract model that describes our network. If we take a closer look at a physical data center, a layer 2 Ethernet switch can be considered 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 data 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.

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

YANG and NETCONF

One of the network data modeling languages is YANG, an interesting acronym for **Yet Another Next Generation** (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.

As a data modeling language, YANG 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, such as NETCONF, and the underlying vendor-specific syntax for configuration and operations. We will look at some examples of YANG later in this chapter.

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

Cisco API examples

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, and many external acquisitions. However, with product lines spanning routers, switches, firewalls, servers (unified computing), wireless, collaboration software and hardware, and analytic software, it is hard to know where to start.

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

- Nexus with NX-API
- Cisco NETCONF and YANG examples
- Cisco **Application Centric Infrastructure** (**ACI**)
- Cisco Meraki examples

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 CML virtual lab.

We will use the same lab topology as we did in *Chapter 2*, *Low-Level Network Device Interactions*, and focus on the devices running **NX-OSv**, **lax-cor-r1**, and **nyc-cor-r1**:

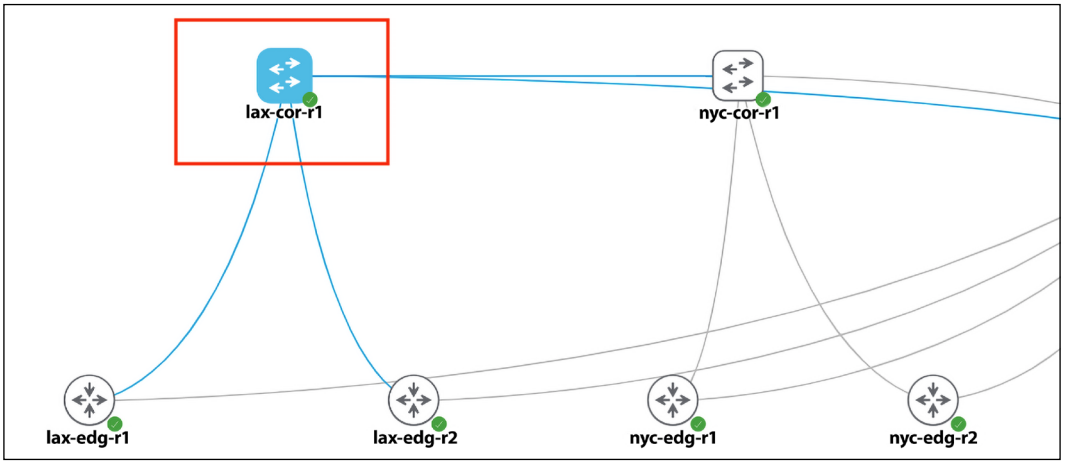


Figure 3.2: Lab NX-OSv Device

Let's take a look at the Cisco NX-API example first.

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 preparation

Remember to activate our Python virtual environment. This should be a given requirement whenever we perform labs from now on:

```
$ source venv/bin/activate
```

The `ncclient` (`https://github.com/ncclient/ncclient`) library is a Python library for NETCONF clients. We will also install a popular Python HTTP client library called Requests (`https://pypi.org/project/requests/`). We can install both via pip:

```
$ pip install ncclient==0.6.13
$ pip install requests==2.28.1
```

NX-API on Nexus devices is turned off by default, so we will need to turn it on. We will also need a user. In this case, we will just use the existing user of `cisco`:

```
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 the `nxapi http` and the `nxapi sandbox` configuration; keep in mind that both should be turned off in production:

```
lax-cor-r1(config)# nxapi http port 80
lax-cor-r1(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 look at the various message formats, requests, and responses based on the CLI commands that we are already familiar with:

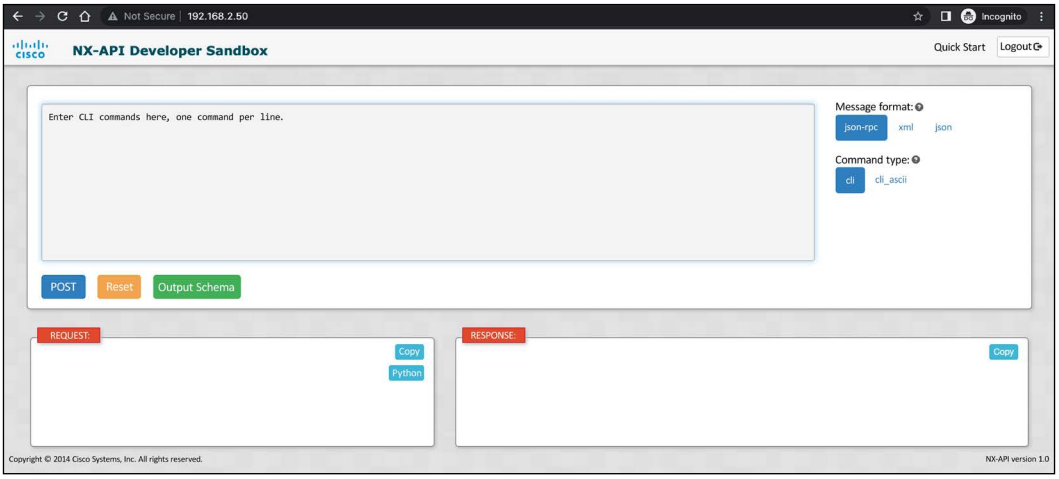


Figure 3.3: NX-API Developer Sandbox

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

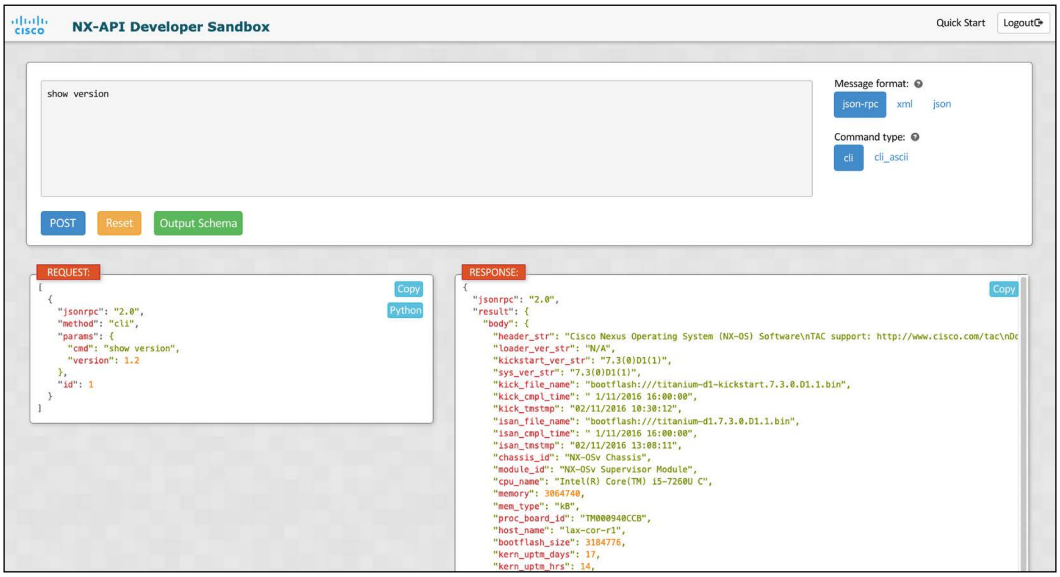


Figure 3.4: NX-API Sandbox example output

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='192.168.2.50',
        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()
```

In our example, we are using the `ncclient` library to connect to the device. The connection parameters of the host, port, username, and password are self-explanatory. The device parameter specifies the kind of device the client is connecting to. The `hostkey_verify` bypasses the `known_host` requirement for SSH; if it's not set to false, the host fingerprint will need 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.

The output will show the XML- and NETCONF-supported features of 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 other vendors as a comparison. 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 Requests library.

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 modifications:

```
"""
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 taken the script the NX-API sandbox generated and only modified the URL, username, and password of the script. 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 configuration and `show` commands. This is illustrated in the `cisco_nxapi_3.py` file, configuring the device with a new hostname. After command execution, you will see the device hostname being changed from `lax-cor-r1` to `lax-cor-r1-new`:

```
lax-cor-r1-new# sh run | i hostname
hostname lax-cor-r1-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:

```
interface Ethernet2/12
  description foo-bar
  shutdown
  no switchport
  mac-address 0000.0000.002f
```

In the next example, we will see how we can use YANG with NETCONF.

The Cisco YANG Model

Let us take a look at Cisco's YANG model support with an example. First, 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. Thirdly, YANG has different 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 YANG models supported on the platform:

```
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>
```

YANG has somewhat uneven support among different vendors and products. I have included a `cisco_yang_1.py` script with the book code repository for a Cisco IOS-XE always-on sandbox provided by Cisco Devnet that shows how to parse out the NETCONF XML output with YANG filters called `urn:ietf:params:xml:ns:yang:ietf-interfaces`.

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

The Cisco ACI examples

The Cisco Application Centric Infrastructure, or ACI, is meant to provide a centralized controller approach to all network components we defined within its management scope. In the data center context, 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 a GUI, CLI, or API. Some might argue that the ACI is Cisco's answer to broader controller-based, software-defined networking.

The ACI API follows the REST model to use the HTTP verbs (`GET`, `POST`, and `DELETE`) to specify the intended operation. In our example, we can use the Cisco DevNet always-on lab ACI device (`https://devnetsandbox.cisco.com/RM/Topology`):

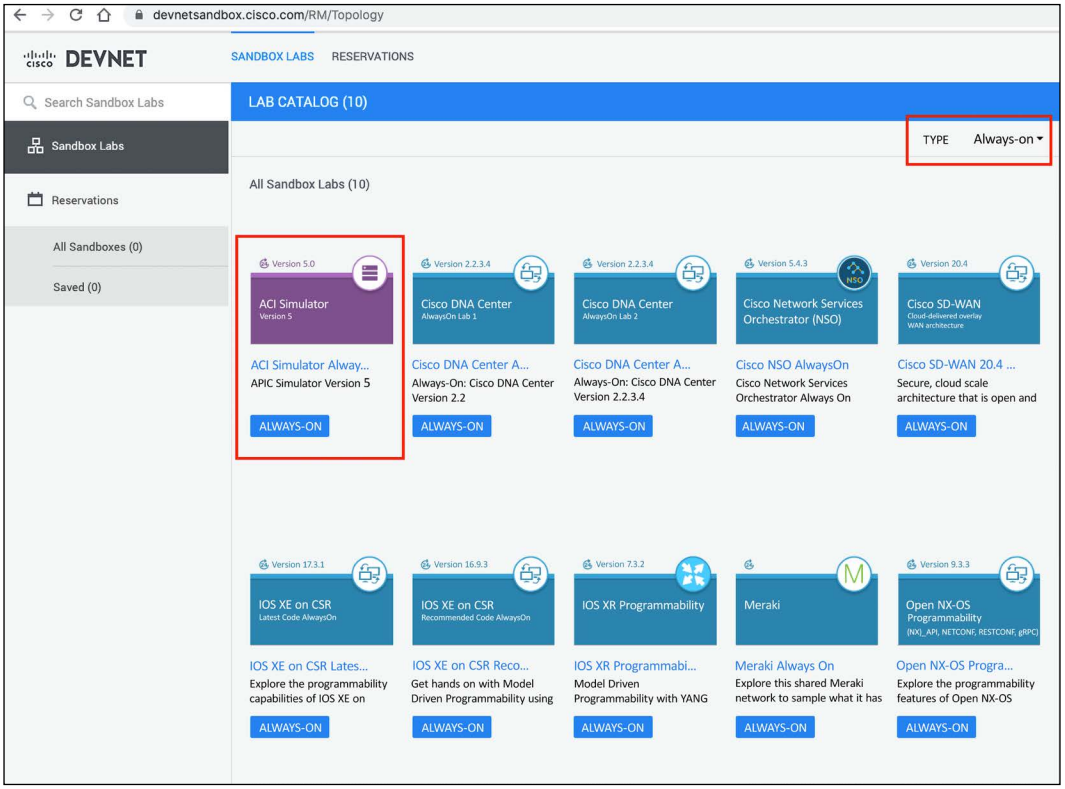


Figure 3.5: Cisco DevNet Sandboxes



Always check the latest Cisco DevNet page for the latest device information, username, and password as they may have changed since the book was written.

The controller is the brain of the network that maintains visibility to all network devices:

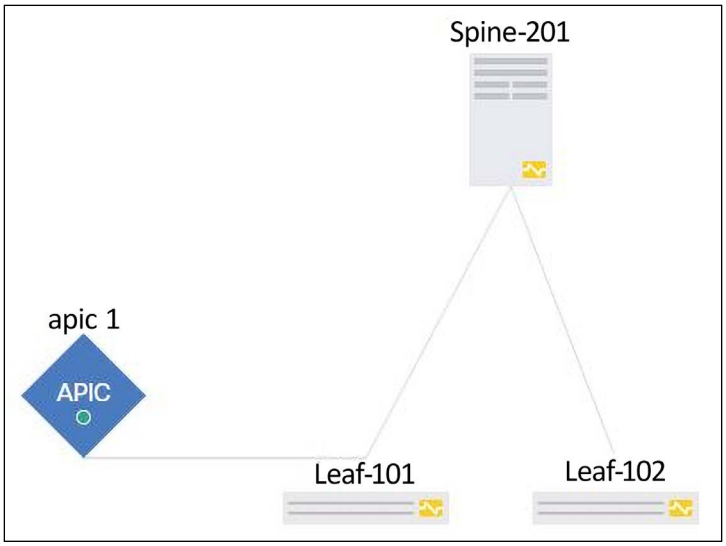


Figure 3.6: Cisco ACI Controller

We can use a web browser to log on to the controller and take a look at the different tenants:

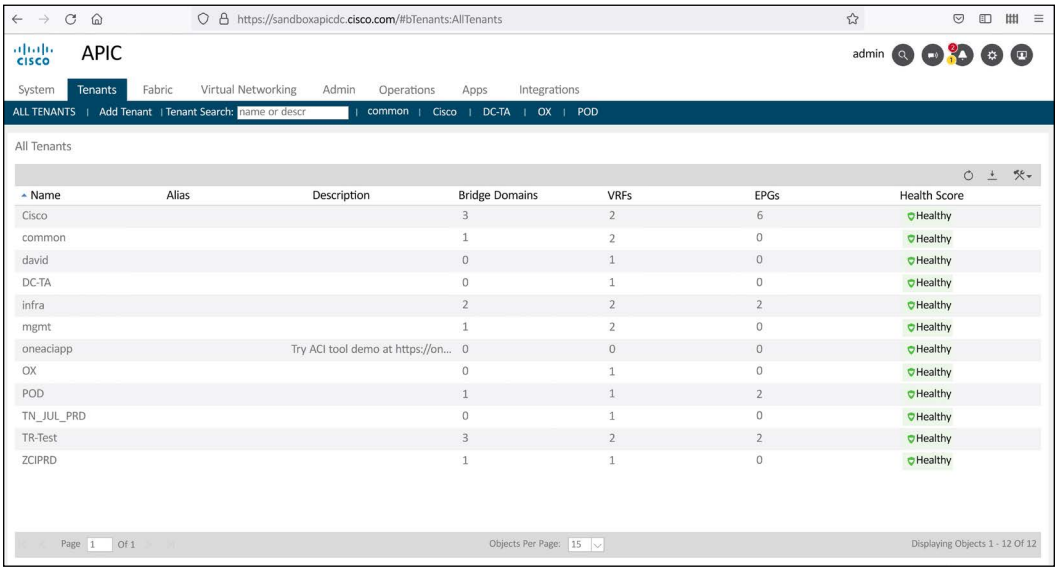


Figure 3.7: Cisco ACI Tenants

Let's use a Python interactive prompt to see how we can interact with the ACI controller. We will start by importing the correct libraries and defining the destination URL as well as the login credentials:

```
>>> import requests, json
>>> URL = 'https://sandboxapicdc.cisco.com'
>>> PASSWORD = "<password>"
>>> LOGIN = "admin"
>>> AUTH_URL = URL + '/api/aaaLogin.json'
```

We can then make a request and convert the response into JSON format:

```
>>> r = requests.post(AUTH_URL,
json={"aaaUser":{"attributes":{"name":LOGIN,"pwd":PASSWORD}}},
verify=False)
>>> r_json = r.json()
>>> r_json
{'totalCount': '1', 'imdata': [{'aaaLogin': {'attributes': {'token':
_<skip>}
```

We can grab the token from the response and use it as an authentication cookie for future requests to the controller. In the example below, we query for the `cisco` tenant that we saw in the controller tenant section:

```
>>> token = r_json["imdata"][0]["aaaLogin"]["attributes"]["token"]
>>> cookie = {'APIC-cookie':token}
>>> QUERY_URL = URL + '/api/node/class/fvTenant.json?query-target-
filter=eq(fvTenant.name,"Cisco")'
>>> r_cisco = requests.get(QUERY_URL, cookies=cookie, verify=False)
>>> r_cisco.json()
{'totalCount': '1', 'imdata': [{'fvTenant': {'attributes': {'annotation':
'', 'childAction': '', 'descr': '', 'dn': 'uni/tn-Cisco', 'extMngdBy': '',
'lcOwn': 'local', 'modTs': '2022-08-06T14:05:15.893+00:00', 'monPolDn':
'uni/tn-common/monepg-default', 'name': 'Cisco', 'nameAlias': '',
'ownerKey': '', 'ownerTag': '', 'status': '', 'uid': '15374', 'userdom':
':all:'}}}]}
>>> print(r_cisco.json()['imdata'][0]['fvTenant']['attributes']['dn'])
uni/tn-Cisco
```

As you can see, we only query a single controller device, but we can get a high-level view of all the network devices that the controller is aware of. This is pretty neat! 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. IOS-XE has 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 ACI, Cisco Meraki is a centrally managed controller that has visibility for multiple wired and wireless networks. Unlike the ACI controller, Meraki is cloud based, so it is hosted outside of the on-premises location. Let us look at some of the Cisco Meraki features and examples in the next section.

Cisco Meraki controller

Cisco Meraki is a cloud-based centralized controller that simplifies IT management of devices. The approach is very similar to ACI with the exception that the controller has 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 script, `cisco_meraki_1.py`, which is a simple request toward an always-on Meraki controller provided by Cisco DevNet:

```
(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())
```

Let's take a look at the output from the `cisco_meraki_2.py` script:

```
(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>
```

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 has 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 at the top of their list of requirements. Before the dawn of cloud-scale data centers, service providers had the most network equipment to manage. For example, a typical enterprise network might have a few redundant internet connections at the corporate headquarters with a few hub-and-spoke remote sites connected back to the HQ using a private **multiprotocol label switching (MPLS)** network. But to a service provider, they are the ones needing to build, provision, manage, and troubleshoot the MPLS 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 fewest engineering hours to keep the network humming along. In their use case, network automation is a 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 adopt in enterprise or data center networks. Juniper Networks has identified this need for 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 that 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 (RPCs)**. Therefore, if HTTP(s) is the transport protocol, the URL endpoint is identical while the operation intended is specified in the request's body.

3. It is conceptually based on layers from top to bottom. The layers include the content, operations, messages, and transport:

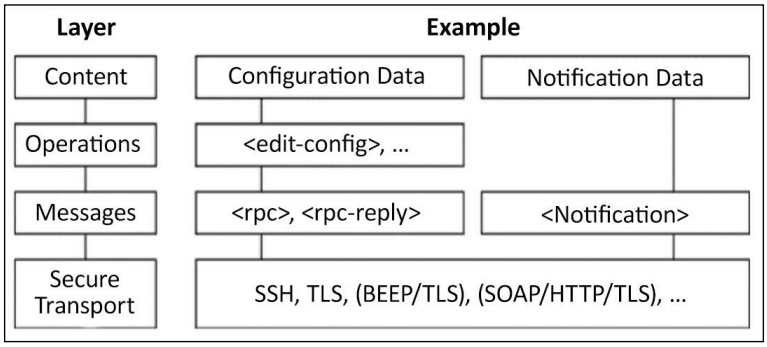


Figure 3.8: NETCONF model

Juniper Networks provides 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 its technical library. Let's take a look at its usage.

Device preparation

To start using NETCONF, let's create a separate user as well as turn on the required services:

```
set system login user juniper uid 2001
set system login user juniper class super-user
set system login user juniper 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 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 the *Lab software installation and device preparation within Cisco NX-API* section. If you have not done so, go 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='192.168.2.70',
    port='830',
    username='juniper',
    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='192.168.2.70', port='830',
username='juniper', 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'
```

In 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 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.

`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. We can create a new user or use the existing user. For `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` in 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 ...
```

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 192.168.2.70>',
user='juniper', password='juniper!')
>>> dev.open()
Device(192.168.2.70)
>>> 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='192.168.2.70', user='echou', ssh_private_key_
file='/home/echou/.ssh/id_rsa')
>>> dev1.open()
Device(192.168.2.70)
>>> 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 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='192.168.2.70', 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 awesome because there is no slippage between what we can do in CLI versus API. The catch is that we need to find 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='192.168.2.70', 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 look at how we can work with Arista network devices using Python libraries.

The Arista Python API

Arista Networks has 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 is an industry leader in data-driven, client to cloud networking for large data center, campus, and routing 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 underpinning behind its 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 in this chapter.

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 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. 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 request's body. 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, we can 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. The authentication is tied to 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:

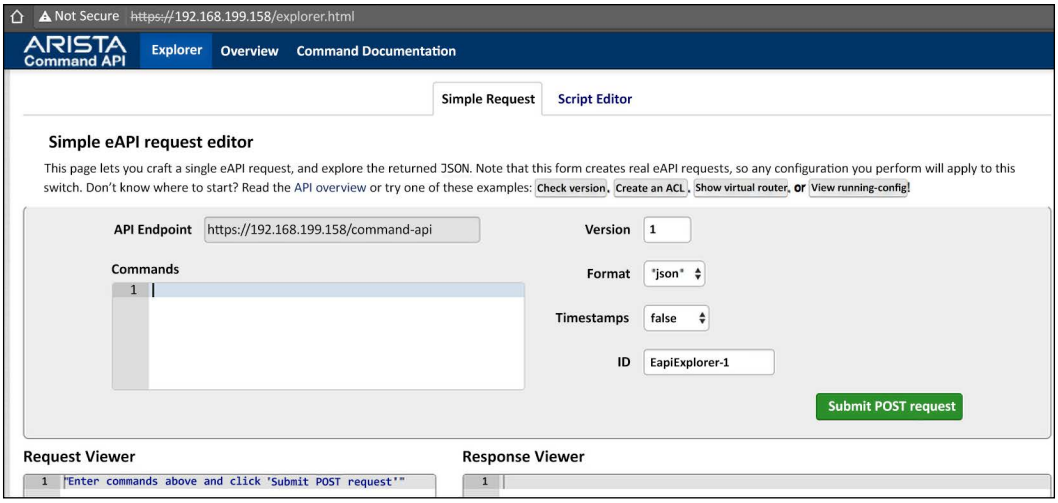


Figure 3.9: Arista EOS explorer

We will be taken to an explorer page, where we can type in the CLI command and get a nice output for the body of our 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:

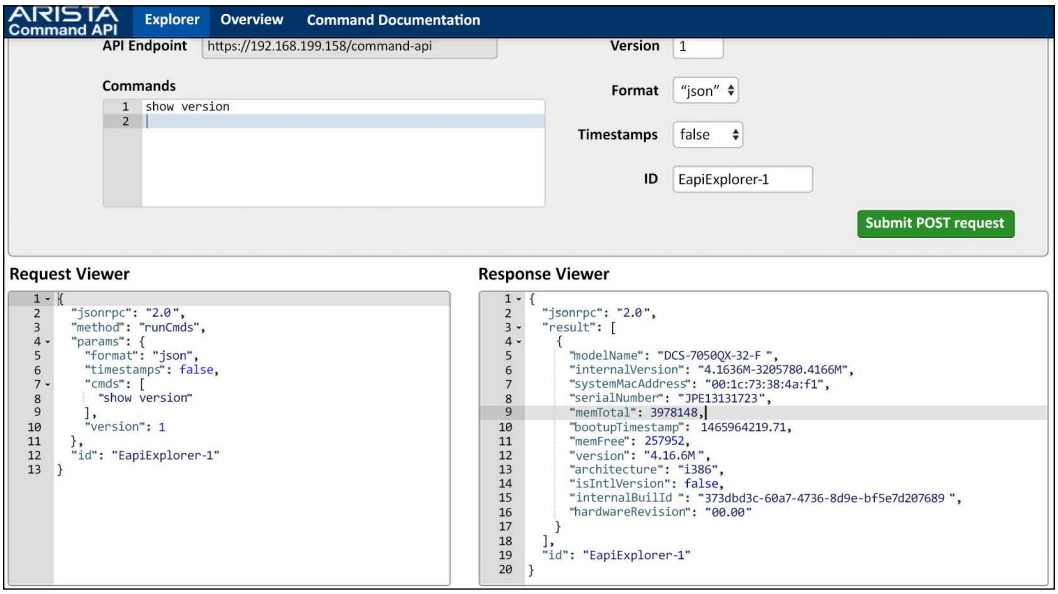


Figure 3.10: Arista EOS explorer viewer

The overview link will take us to the sample use and background information, while the command documentation will serve as reference points for the show commands. Each command reference 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 we will use.

The examples in this section mainly use Python 2.7 as jsonrpclib was not ported to Python 3 for a long time. However, per GitHub pull request `https://github.com/joshmarshall/jsonrpclib/issues/38`, Python 3 support should be included.

Installation is straightforward using `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'])
```

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 clicking on the **Command Documentation** link on the eAPI page and comparing our 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. We 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 steps and configure the device via a list of commands.

Here is an example of configuration commands in a file 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 and eAPI is mostly a judgment call.

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 a language other 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 such as Pyeapi 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 (`https://pyeapi.readthedocs.io/en/master/requirements.html`)!

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

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')]
```

We 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 we can see, the Python native API on the EOS object is 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 we can see from the script, we automatically connect to the node, set the hostname, and load `running_config` upon connection. We also create a method of 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 looked 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 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 it's 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 domain name 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`), NAPALM (`https://github.com/napalm-automation/napalm`), and Scrapli (`https://carlmontanari.github.io/scrapli/`). We have seen some of their examples in the last chapter. Most of these vendor-neutral libraries 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, these libraries are good choices. Another benefit of using vendor-neutral libraries is that they are normally open source, so you can contribute upstream for new features and bug fixes.

Summary

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

In *Chapter 4*, *The Python Automation Framework*, 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.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



4

The Python Automation Framework – Ansible

The previous two chapters incrementally introduced different ways to interact with network devices. In *Chapter 2*, *Low-Level Network Device Interactions*, we discussed the Pexpect and Paramiko libraries, which manage an interactive session to control interactions. In *Chapter 3*, *APIs and Intent-Driven Networking*, we started to think of our network in terms of API and intent. We looked at various APIs that contain a well-defined command structure and provide a structured way of getting feedback from the device. As we moved from *Chapter 2*, *Low-Level Network Device Interactions*, to *Chapter 3*, *APIs and Intent-Driven Networking*, we began to think about our intent for the network. We gradually began to express our network as code.

In this chapter, let's expand upon the idea of translating our intention into network requirements. If you have worked on network designs, chances are the most challenging part of the process is not the different pieces of network equipment but rather the qualifying and translating of business requirements into the actual network design. Your network design needs to solve business problems. For example, you might be working within a larger infrastructure team that needs to accommodate a thriving online e-commerce site that experiences slow site response times during peak hours. How do you determine whether the network is the problem? If the slow response on the website was indeed due to network congestion, which part of the network should you upgrade? Can the rest of the system take advantage of the greater speed and feed?

The following diagram is an illustration of a simple process of the steps that we might go through when trying to translate our business requirements into a network design:

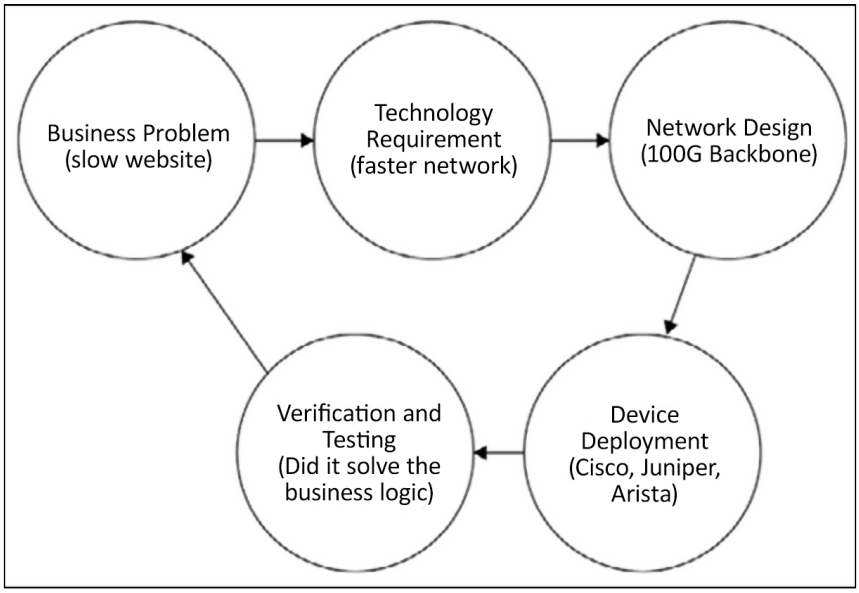


Figure 4.1: Business logic to network deployment

In my opinion, network automation is not just about faster configuration change. It should also solve business problems while accurately and reliably translating our intention into device behavior. These are the goals that we should keep in mind as we march on our network automation journey. In this chapter, we will look at a Python-based framework called **Ansible**, which allows us to declare our intention for the network and abstract even more from the API and CLI.

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

- An introduction to Ansible
- The advantages of Ansible
- The Ansible architecture
- Ansible advanced topics

Let's begin by looking at an overview of the Ansible framework.

Ansible – A More Declarative Framework

Imagine yourselves in a hypothetical situation: you woke up one morning in a cold sweat from a nightmare you had about a potential network security breach. You realize your network contains valuable digital assets that should be protected. You have been doing your job as a network administrator, so it is pretty secure, but you want to put more security measures around your network devices just to be sure.

To start with, you break the objective down into two actionable items:

- Upgrading the devices to the latest version of the software. The steps include the following:
    1. Uploading the image to the device
    2. Instructing the device to boot from the new image
    3. Proceeding to reboot the device
    4. Verifying that the device is running with the new software image
- Configuring the appropriate access control list on the networking devices, which includes the following:
    1. Constructing the access list on the device
    2. Configuring the access list on the interface under the interface configuration section

Being an automation-focused network engineer, you want to write scripts to reliably configure the devices and receive feedback from the operations. You begin to research the necessary commands and APIs for each step, validate them in the lab, and finally, deploy them in production. Having done a fair amount of work for OS upgrade and ACL deployment, you hope the scripts are transferable to the next generation of devices.

Wouldn't it be nice if there was a tool that could shorten this design-develop-deployment cycle? In this chapter, we will work with an open-source automation framework called Ansible. It is a framework that can simplify the process of going from business logic to completing the job without getting bogged down with specific network commands. It can configure systems, deploy software, and orchestrate a combination of tasks.

Ansible is written in Python and has emerged as one of the leading automation tools for Python developers. It is also one of the most supported automation frameworks by network vendors. In the *'Python Developers Survey 2020'* by JetBrains, Ansible is ranked #1 for configuration management tools:

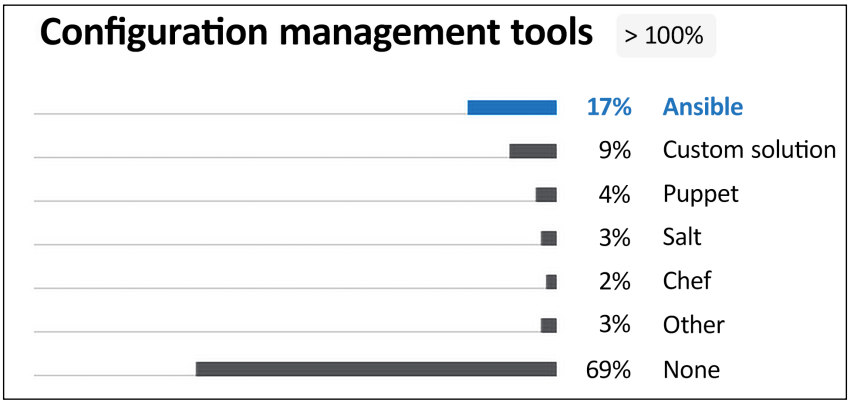


Figure 4.2: Python Developers Survey 2020 Results (source: https://www.jetbrains.com/lp/python-developers-survey-2020/)

Ever since version 2.10, Ansible has separated the release schedule for **ansible-core** and community packages. It is a bit confusing, so let's take a look at the differences.

Ansible Versions

Prior to version 2.9, Ansible has a fairly straightforward versioning system, going from 2.5, 2.6, 2.7, and so on (`https://docs.ansible.com/ansible/latest/roadmap/old_roadmap_index.html`). Starting from version 2.10, we see the jump from Ansible project 2.10, 3.0, 4.0, and so on (`https://docs.ansible.com/ansible/latest/roadmap/ansible_roadmap_index.html#ansible-roadmap`). What gives? The Ansible team wants to separate the core engine, modules, and plugins from the wider community-curated modules and plugins. This allows the core team to move faster with the core features while allowing time for the community to catch up with the maintenance of their code.

When we talk about "Ansible," we are referring to a collection of community packages at that level, say, version 3.0. In the version, it will specify a version of `ansible-core` (initially called `ansible-base`) it requires. For example, Ansible 3.0 requires ansible-core 2.10 and above, while Ansible 4.0 requires ansible-core 2.11+. In this structure, we can upgrade ansible-core to the latest version while keeping the community packages in an older release if needed.



If want to learn more about the versioning split, Ansible provides a useful Q&A page when they first adopted Ansible 3.0, `https://www.ansible.com/blog/ansible-3.0.0-qa`.

Let's move ahead and look at an Ansible example.

Our First Ansible Network Example

Ansible is an IT automation tool. Its main attributes are simplicity and ease of use with minimum moving parts. It manages machines in an agent-less manner (more on this later) and relies on the existing operating system credentials and remote Python software to run its code. Ansible is installed on a centralized machine called the controlled node and executes on the machine it wishes to control, called the managed node.

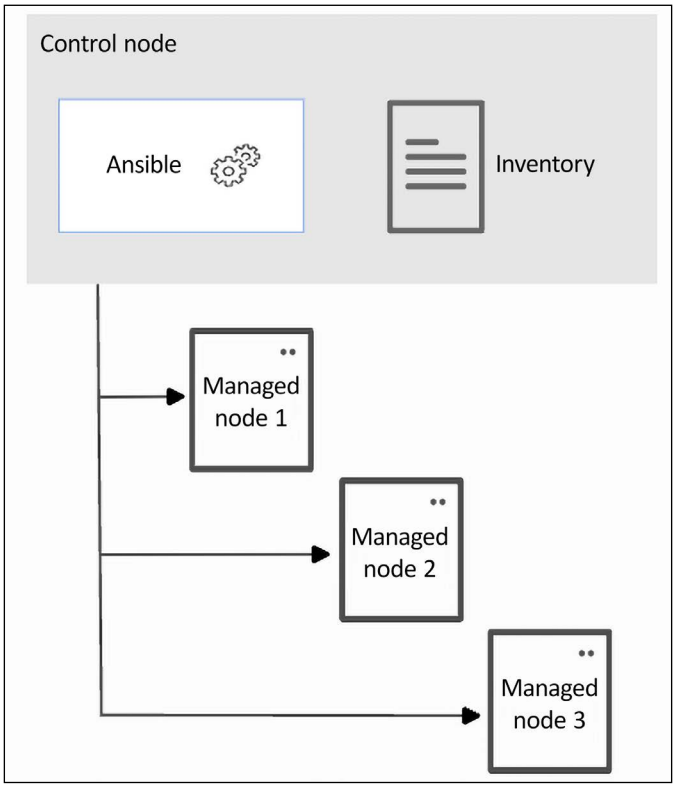


Figure 4.3: Ansible Architecture (Source: https://docs.ansible.com/ansible/latest/getting_started/index.html)

As with most IT infrastructure automation, Ansible started by managing servers. Most servers have Python installed or are capable of running Python code; Ansible would leverage this feature by pushing the code to the managed node and having it run locally on the managed node. However, as we know, most network devices are not capable of running native Python code; therefore, when it comes to network automation, the Ansible configuration is run locally first before making the changes to the remote devices.



For more information on how network automation is different, check out this document from Ansible, `https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html`.

Let us install Ansible on the control node.

The Control Node Installation

We will install Ansible on the Ubuntu host in our lab. The only requirements for the control node are Python 3.8 or newer as well as Python's `pip` package management system

```
(venv) $ pip install ansible
```

We can check the installed Ansible version as well as other package-level information via the '—version' switch:

```
(venv) $ ansible --version
ansible [core 2.13.3]
  config file = None
  configured module search path = ['/home/echou/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location = /home/echou/Mastering_Python_
Networking_Fourth_Edition/venv/lib/python3.10/site-packages/ansible
  ansible collection location = /home/echou/.ansible/collections:/usr/
share/ansible/collections
  executable location = /home/echou/Mastering_Python_Networking_Fourth_
Edition/venv/bin/ansible
  python version = 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0]
  jinja version = 3.1.2
  libyaml = True
```



If you are interested in installing Ansible on specific operation systems using their respective package management systems, please consult the Ansible documentation, `https://docs.ansible.com/ansible/latest/installation_guide/installation_distros.html`.

There is some important information displayed in the output. The most important is the Ansible core version (2.13.3) and configuration file (none for now). This is all we need to get started with Ansible, we can start to construct our first automation task.

Lab Topology

Ansible is known to have many different ways to accomplish the same task. For example, we can define Ansible configuration files in different locations. We can also specify host-specific variables in a variety of places such as an inventory, in playbooks, in roles, and the command line. This is too confusing for people who are just getting started with Ansible. In this chapter, I will use just one way of doing things that makes the most sense to me. Once we learn the basics, we can always consult the documentation to find out other ways to accomplish a task.

For the first example, we will use the same lab topology that we have been using and run the task against the two IOSv devices, `lax-edg-r1` and `lax-edg-r2`.

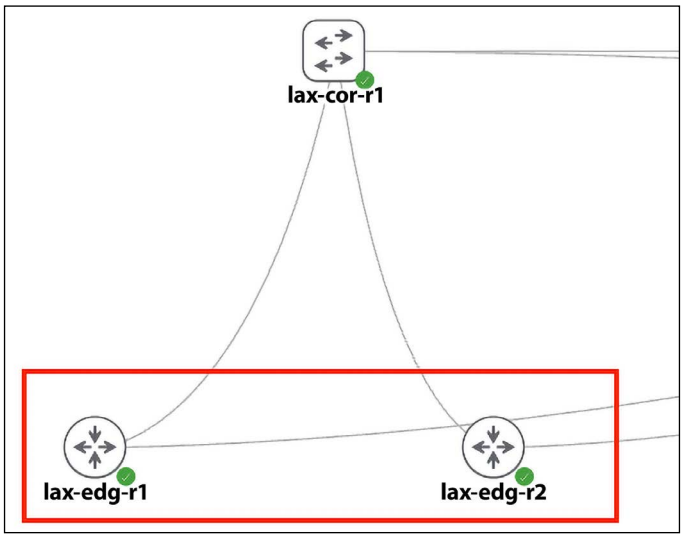


Figure 4.4: Lab Topology

The first thing we will need to think about is how to define the hosts we want to manage. In Ansible, we use an inventory file to define the host we intend to manage. Let us create a file called `hosts` and put the following text in the file:

```
[ios_devices]
iosv-1
iosv-2
```

This type of file is in an INI format (`https://en.wikipedia.org/wiki/INI_file`), which states I have a group of devices called the `ios_devices` with members of `iosv-1` and `iosv-2`.

We should now specify the particular variables associated with each host.

The Variable Files

There are many places we can put the variables associated with a host. Let us create a folder called `host_vars` and two files with the names of the files identical to the hosts we specified in the inventory file. The directory and filenames are important because that is how Ansible matches up the variables with the host. Below is an output to show the directory and files within that directory:

```
$ tree host_vars/
host_vars/
├── iosv-1
└── iosv-2
```

The file is where we will put the necessary information belonging to the host. For example, we can specify the IP address, username, password, and other information. Here is the output of the `iosv-1` file for our lab:

```
$ cat host_vars/iosv-1
---
ansible_host: 192.168.2.51
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

This file is in YAML format (`https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html`). The '`---`' symbol indicates the start of the document. Below the start symbol, we have many key-value pairs. The keys all start with `ansible` and the value is separated from the key with a colon. The `ansible_host`, `ansible_user`, and `ansible_ssh_pass` should be changed to values matching your own lab. How do I know these names? Ansible documentation is our best friend here. Ansible has a standard way of naming these parameters listed in its documentation, `https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html`.



Prior to Ansible 2.8, network modules did not have a standard way of naming the parameters, which is very confusing. Since version 2.8, the network modules have become much better at standardizing the parameters with the rest of the Ansible modules.

Once we have defined the respective files for the host variables, we are ready to construct an Ansible playbook.

Our First Playbook

Playbooks are Ansible's blueprint to describe what you would like to do to the managed nodes, using modules. This is where we will be spending the majority of our time as operators when working with Ansible. What are modules? In a simplified view, modules are pre-built codes that we can use to accomplish a certain task. Similar to Python modules, the code can come with the default Ansible installation or it can be installed separately.

If we use an analogy of building a tree house with Ansible, the playbook will be the instruction manual, the modules will be the tools we use, and the inventory will be the components we work on.

The playbook is designed to be human-readable in YAML format (`https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html`). We will write our first playbook, named `ios_config_backup.yml`, as follows:

```
- name: Back Up IOS Device Configurations
  hosts: all
  gather_facts: false
  tasks:
    - name: backup
      ios_config:
        backup: yes
```

Note the - in front of name, it specifies a list item in YAML. Everything in the same list item should have the same indentation. We set gather_facts to false because most network tasks are executed locally before making the changes to the devices. The gather_facts were mainly used when the managed nodes were servers to gather server information before any of the tasks were executed.

There are two key-value pairs in the list item, hosts and tasks. The hosts variable with the value of all specifies that we will work on all the hosts in the inventory file. The tasks key has another list item in the value, which uses the ios_config module (https://docs.ansible.com/ansible/latest/collections/cisco/ios/ios_config_module.html#ansible-collections-cisco-ios-ios-config-module). The ios_config module is one of the collections of modules installed along with Ansible. It also has a variety of arguments. We use the backup argument and set it to yes to indicate we will back up the devices' running-config.

The next task we will do is to use the new LibSSH connection plugin for Ansible. By default, Ansible network SSH connections use the Paramiko library. However, the Paramiko library does not guarantee FIPS readiness and is a bit slow when we need to connect to multiple devices. We will install LibSSH as follows:

```
(venv) $ pip install ansible-pylibssh
```

We will specify the usage in a new ansible.cfg file. We will create the file in the same directory as our playbook with the following content. In the same configuration file, we will also set host_key_checking to be false; this is to prevent an error if the host is not initially in the known_hosts list in the ssh setup:

```
[defaults]
host_key_checking = False

[persistent_connection]
ssh_type = libssh
```

Finally, we can execute the playbook via the ansible-playbook command with the -i switch to indicate the inventory file:

```
$ ansible-playbook -i hosts ios_config_backup.yml

PLAY [Back Up IOS Device Configurations] ********************************
**********************************************

TASK [backup] ***********************************************************
```

```
**************************************************
changed: [iosv-2]
changed: [iosv-1]

PLAY RECAP *********************************************************
**************************************************
iosv-1                     : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
iosv-2                     : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Just like magic, if we take a look at our working directory where the playbook is executed, we will see a folder named `backup` with the two devices' running configurations with timestamps! This command can now be scheduled to run via `cron` to run nightly to back up all of our devices' configurations.

Congratulations on executing your first Ansible playbook! Even with a playbook as simple as ours, this is a very useful automation task that we were able to accomplish in a short amount of time. We will expand on this playbook in just a bit, but first, let's take a look at why Ansible is a good fit for network management. Remember that Ansible modules are written in Python; that is one advantage for a Pythonic network engineer, right?

The Advantages of Ansible

There are many infrastructure automation frameworks besides Ansible—namely Chef, Puppet, and SaltStack. Each framework offers its unique features; no one right framework fits all organizations. In this section, let's take a look at some of the advantages of Ansible and why I believe it is a good tool for network automation.

The advantages will be listed with limited comparison to other frameworks to not start a flame war. Other frameworks might adopt some of the same philosophies or certain aspects of Ansible, but rarely do they contain all of the features that I will be mentioning. It is the combination of all the following features and philosophies that makes Ansible ideal for network automation.

Agentless

Unlike some of its peers, Ansible does not require a strict master-client model. No software or agent needs to be installed on the client that communicates back to the server. Outside of the Python interpreter, which many platforms have by default, there is no additional software needed.

For network automation modules, instead of relying on remote host agents, Ansible uses SSH or API calls to push the required changes to the remote host. This further reduces the need for a Python interpreter. This is huge for network device management, as network vendors are typically reluctant to put third-party software on their platforms. SSH, on the other hand, already exists on the network equipment. As we saw from *Chapter 3*, *APIs and Intent-Driven Networking*, newer network devices also provide an API layer, which can also be leveraged by Ansible.

Because there is no agent on the remote host, Ansible uses a `push` model to push the changes to the device, as opposed to the `pull` model, where the agent pulls the information from the master server. The `push` model is more deterministic as everything originates from the control machine. In a `pull` model, the timing of the `pull` might vary from client to client and therefore results in timing variance.

Again, the importance of being agentless cannot be stressed enough when working with the existing network equipment. This is usually one of the major reasons network operators and vendors embrace Ansible.

Idempotence

According to Wikipedia, idempotence is the property of certain operations in mathematics and computer science that can be applied multiple times without changing the result beyond the initial application (`https://en.wikipedia.org/wiki/Idempotence`). In more common terms, it means that running the same procedure repeatedly does not change a system after the first time. Ansible aims to be idempotent, which is good for network operations that require a certain order of operations. In our first playbook example, there is a 'changed' value when the playbook is run; this value will be 'false' if there was no change made on the remote device.

The advantage of idempotence is best compared to the Pexpect and Paramiko scripts we have written. Remember that these scripts were written to push out commands like an engineer was sitting at the terminal. If you were to execute the script 10 times, the script would make the same changes 10 times. If we write the same task via the Ansible playbook, the existing device configuration will be checked first, and the playbook will only execute if the changes do not exist. If we execute the playbook 10 times, the change will only be applied during the first run, with the next 9 runs suppressing the configuration change.

Being idempotent, we can repeatedly execute the playbook without worrying that there will be unnecessary changes made. This is important as we need to automatically check for state consistency without extra overhead.

Simple and Extensible

Ansible is written in Python and uses YAML for the playbook language, both of which are relatively easy to learn. Remember the Cisco IOS syntax? This is a domain-specific language that is only applicable when you are managing Cisco IOS devices or other similarly structured equipment; it is not a general-purpose language beyond its limited scope. Luckily, unlike some other automation tools, there is no extra **domain-specific language** (**DSL**) to learn for Ansible because YAML and Python are both widely used as general-purpose languages.

Ansible is extensible. As illustrated by the preceding example, Ansible starts with automating server (primarily Linux) workloads in mind. It then branches out to manage Windows machines with PowerShell. As more and more people in the network industry have started to adopt Ansible, network automation is now the main staple in Ansible workgroups.

The simplicity and extensibility speak well for future-proofing. The technology world is evolving fast, and we are constantly trying to adapt. Wouldn't it be great to learn a technology once and continue to use it, regardless of the latest trend? Ansible's track record speaks well for future technology adaptation.

Now that we've covered some of the advantages of Ansible, let's build on what we have learned so far with more features.

Ansible Content Collections

Let's start by listing out all the modules we have at hand with the default Ansible installation. They are organized into Content Collections (`https://www.ansible.com/products/content-collections`), sometimes called collections for abbreviation. We can list out the collections via the `ansible-galaxy collection list` command. Some of the notable network collections are listed out below:

```
(venv) $ ansible-galaxy collection list

# /home/echou/Mastering_Python_Networking_Fourth_Edition/venv/lib/
python3.10/site-packages/ansible_collections
Collection                    Version
----------------------------- -------
…
ansible.netcommon             3.1.0
arista.eos                    5.0.1
cisco.aci                     2.2.0
```

```
cisco.asa                       3.1.0
cisco.dnac                      6.5.3
cisco.intersight                1.0.19
cisco.ios                       3.3.0
cisco.iosxr                     3.3.0
cisco.ise                       2.5.0
cisco.meraki                    2.10.1
cisco.mso                       2.0.0
cisco.nso                       1.0.3
cisco.nxos                      3.1.0
cisco.ucs                       1.8.0
community.ciscosmb              1.0.5
community.fortios               1.0.0
community.network               4.0.1
dellemc.enterprise_sonic        1.1.1
f5networks.f5_modules           1.19.0
fortinet.fortimanager           2.1.5
fortinet.fortios                2.1.7
mellanox.onyx                   1.0.0
openstack.cloud                 1.8.0
openvswitch.openvswitch         2.1.0
vyos.vyos                       3.0.1
```

As shown from the list, even with the default installation, there is a large collection of network-related modules we can use. They range from enterprise software to open-source projects. Taking a look at the list and reading up on the ones that are of interest in your production environment would be a good start. Ansible documentation also provides a full list of all the available collections, `https://docs.ansible.com/ansible/latest/collections/index.html`. The collections can also be expanded via the `agalaxy install` command, `https://docs.ansible.com/ansible/latest/user_guide/collections_using.html`.

More Ansible Network Examples

Our first Ansible network example took us from being a noob to running our first useful network automation task. Let's try to build from the foundation and learn more features.

We will begin by seeing how we can build an inventory file that includes all of our network devices. If you recall, we have two data centers, each with core and edge devices:

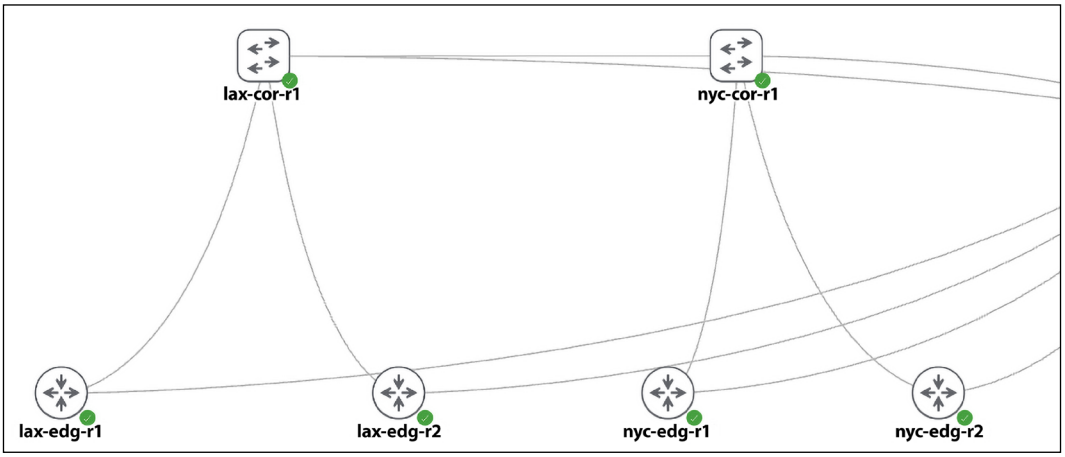


Figure 4.5: Full Lab Topology

In this example, we will include all of the devices in our inventory file.

Inventory Nesting

We can build an inventory file that includes nesting. For example, we can put together a host file named `hosts_full` that includes children from one group to another:

```
[lax_cor_devices]
lax-cor-r1

[lax_edg_devices]
lax-edg-r1
lax-edg-r2

[nyc_cor_devices]
nyc-cor-r1

[nyc_edg_devices]
nyc-edg-r1
nyc-edg-r2
```

```
[lax_dc:children]
lax_cor_devices
lax_edg_devices

[nyc_dc:children]
nyc_cor_devices
nyc_edg_devices

[ios_devices:children]
lax_edg_devices
nyc_edg_devices

[nxos_devices:children]
nyc_cor_devices
lax_cor_devices
```

In the file, we group the devices via both roles and features using the `[<name>:children]` format. To work with this new inventory file, we will need to update the `host_vars` directory to include the respective names of the devices:

```
(venv) $ tree host_vars/
host_vars/
…
├── lax-cor-r1
├── lax-edg-r1
├── lax-edg-r2
├── nyc-cor-r1
├── nyc-edg-r1
└── nyc-edg-r2
```

We will also need to change the `ansible_host` and `ansible_network_os` accordingly, using `lax-cor-r1` as an example:

```
(venv) $ cat host_vars/lax-cor-r1
---
ansible_host: 192.168.2.50
…
ansible_network_os: nxos
…
```

Now we can use the parent's group name to include its children. For example, in the `nxos_config_backup.yml` playbook, we only specified the parent group of `nxos_devices` instead of `all`:

```
- name: Back Up NX-OS Device Configurations
  hosts: nxos_devices
  gather_facts: false
  tasks:
    - name: backup
      nxos_config:
        backup: yes
```

When we execute this playbook, it will automatically include its children, `lax_cor_devices` and `nyc_cor_devices`. Also note that we use a separate `nxos_config` module (`https://docs.ansible.com/ansible/latest/collections/cisco/nxos/nxos_config_module.html#ansible-collections-cisco-nxos-nxos-config-module`) to accommodate the new device type.

Ansible Conditionals

Ansible conditionals are similar to conditional statements in programming languages. Ansible uses conditional keywords to only run a task when a given condition is met. In many cases, the execution of a play or task may depend on the value of a fact, variable, or the previous task result. For example, if you have a play to upgrade router images, you want to include a step to make sure the new router image is on the device before you move on to the next play of rebooting the router.

In this example, we will look at the `when` clause, which is supported for all modules. The `when` clause is useful when you need to check the output of a variable or a play execution result and act accordingly. Some of the conditions are as follows:

- Equal to (`eq`)
- Not equal to (`neq`)
- Greater than (`gt`)
- Greater than or equal to (`ge`)
- Less than (`lt`)
- Less than or equal to (`le`)
- Contains

Let's take a look at the following playbook named `ios_conditional.yml`:

```
---
- name: IOS command output for when clause
  hosts: ios_devices
  gather_facts: false
  tasks:
    - name: show hostname
      ios_command:
        commands:
            - show run | i hostname
      register: output

    - name: show output with when conditions
      when: output.stdout == ["hostname nyc-edg-r2"]
      debug:
        msg: '{{ output }}'
```

In the playbook, there are two tasks. In the first task, we use the `register` module to save the output of the command `show run | i hostname` in a variable called `output`. The `output` variable contains a `stdout` list with the output. We use the `when` clause to only show the output when the hostname is `nyc-edg-r2`. Let's execute the playbook:

```
(venv) $ ansible-playbook -i hosts_full ios_conditional.yml

PLAY [IOS command output for when clause] ********************************
**************************************************

TASK [show hostname] *****************************************************
**************************************************
ok: [lax-edg-r1]
ok: [nyc-edg-r2]
ok: [lax-edg-r2]
ok: [nyc-edg-r1]

TASK [show output with when conditions] **********************************
**************************************************
skipping: [lax-edg-r1]
skipping: [lax-edg-r2]
```

```
skipping: [nyc-edg-r1]
ok: [nyc-edg-r2] => {
    "msg": {
        "changed": false,
        "failed": false,
        "stdout": [
            "hostname nyc-edg-r2"
        ],
        "stdout_lines": [
            [
                "hostname nyc-edg-r2"
            ]
        ]
    }
}

PLAY RECAP *********************************************************************
****************************************************
lax-edg-r1                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
lax-edg-r2                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r1                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r2                 : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

We can see the output of `lax-edg-r1`, `lax-edg-r2`, and `nyc-edg-r1` were skipped because they did not meet the condition. Furthermore, we can see the `changed=0` output for all the devices. This is in accordance with the idempotency feature of Ansible.

Configuration Change

We can combine the conditional with configuration changes—for example, in the following playbook, `ios_conditional_config.yml`:

```
---
- name: IOS command output for when clause
  hosts: ios_devices
  gather_facts: false
```

```
  tasks:
    - name: show hostname
      ios_command:
        commands:
            - show run | i hostname
      register: output

    - name: show output with when conditions
      when: output.stdout == ["hostname nyc-edg-r2"]
      ios_config:
        lines:
            - logging buffered 30000
```

We would only change the logging buffer when the condition is met. Here is the output when we execute the playbook for the first time:

```
(venv) $ ansible-playbook -i hosts_full ios_conditional_config.yml
<skip>
TASK [show output with when conditions] ********************************
**************************************************
skipping: [lax-edg-r1]
skipping: [lax-edg-r2]
skipping: [nyc-edg-r1]
[WARNING]: To ensure idempotency and correct diff the input configuration
lines should be similar to how they appear if
present in the running configuration on device
changed: [nyc-edg-r2]

PLAY RECAP *************************************************************
**************************************************
lax-edg-r1                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
lax-edg-r2                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r1                 : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r2                 : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

The `nyc-edg-r2` device console will show the configuration is changed:

```
*Sep 10 01:53:43.132: %SYS-5-LOG_CONFIG_CHANGE: Buffer logging: level
debugging, xml disabled, filtering disabled, size (30000)
```

However, when we run the playbook for the second time, the same change is NOT applied again because it is already changed:

```
<skip>
TASK [show output with when conditions] ********************************
**************************************************
skipping: [lax-edg-r1]
skipping: [lax-edg-r2]
skipping: [nyc-edg-r1]
ok: [nyc-edg-r2]
```

How cool is that? With a simple playbook, we can safely apply a configuration change to only the devices we want to apply the change to with idempotency in check.

Ansible Network Facts

Prior to 2.5, Ansible networking shipped with a number of vendor-specific fact modules. As a result, the naming and usage of the facts were different between vendors. Starting with version 2.5, Ansible started to standardize its network fact modules. The Ansible network fact modules gather information from a system and store the results in facts prefixed with `ansible_net_`. The data collected by these modules is documented in the *return values* in the module documentation. This is useful as we can gather network facts and only perform tasks based on them.

As an example of the `ios_facts` module, below is the content of the `ios_facts_playbook`:

```
---
- name: IOS network facts
  connection: network_cli
  gather_facts: false
  hosts: ios_devices
  tasks:
    - name: Gathering facts via ios_facts module
      ios_facts:
      when: ansible_network_os == 'ios'

    - name: Display certain facts
```

```
      debug:
        msg: "The hostname is {{ ansible_net_hostname }} running {{
ansible_net_version }}"

    - name: Display all facts for hosts
      debug:
        var: hostvars
```

We are introducing a concept of variables in this playbook. The double curly brackets of `{{ }}` indicates it is a variable and the value of the variable should be presented for the output.

Upon execution of the playbook, here is a partial output:

```
(venv) $ ansible-playbook -i hosts_full ios_facts_playbook.yml
…
TASK [Display certain facts] ***************************************
******
ok: [lax-edg-r1] => {
    "msg": "The hostname is lax-edg-r1 running 15.8(3)M2"
}
ok: [lax-edg-r2] => {
    "msg": "The hostname is lax-edg-r2 running 15.8(3)M2"
}
ok: [nyc-edg-r1] => {
    "msg": "The hostname is nyc-edg-r1 running 15.8(3)M2"
}
ok: [nyc-edg-r2] => {
    "msg": "The hostname is nyc-edg-r2 running 15.8(3)M2"
}
…
TASK [Display all facts for hosts] *********************************
******
ok: [lax-edg-r1] => {
    "hostvars": {
        "lax-cor-r1": {
…
            "ansible_facts": {
                "net_api": "cliconf",
                "net_gather_network_resources": [],
```

```
                "net_gather_subset": [
                    "default"
                ],
                "net_hostname": "lax-edg-r1",
                "net_image": "flash0:/vios-adventerprisek9-m",
                "net_iostype": "IOS",
                "net_model": "IOSv",
                "net_python_version": "3.10.4",
                "net_serialnum": "98U40DKV403INHIULHYHB",
                "net_system": "ios",
                "net_version": "15.8(3)M2",
                "network_resources": {}
            },
…
```

We can now leverage the facts to combine with our conditional clause to customize our operations.

Ansible Loops

Ansible provides a number of looping functions in the playbook: standard loops, looping over files, sub-elements, `do-until`, and many more. In this section, we will look at two of the most commonly used loop forms: standard loops and looping over hash values.

Standard Loops

Standard loops in playbooks are often used to easily perform similar tasks multiple times. The syntax for standard loops is very easy: the `{{ item }}` variable is the placeholder looping over the `loop` list. In our next example, `standard_loop.yml`, we will loop over the items in the `loop` list with the `echo` command showing the output from our `localhost`.

```
- name: Echo Loop Items
  hosts: "localhost"
  gather_facts: false
  tasks:
    - name: echo loop items
      command: echo "{{ item }}"
      loop:
        - 'r1'
        - 'r2'
        - 'r3'
```

```
        - 'r4'
        - 'r5'
```

Let's go ahead and execute the playbook:

```
(venv) $ ansible-playbook -i hosts_full standard_loop.yml

PLAY [Echo Loop Items] *********************************************************
****************************************************

TASK [echo loop items] *********************************************************
****************************************************
changed: [localhost] => (item=r1)
changed: [localhost] => (item=r2)
changed: [localhost] => (item=r3)
changed: [localhost] => (item=r4)
changed: [localhost] => (item=r5)

PLAY RECAP *********************************************************************
****************************************************
localhost                  : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Using the same concept, we can systematically add VLANs to our devices. Here is an example of adding three VLANs to a host with a playbook titled `standard_loop_vlan_example.yml`:

```
- name: Add Multiple Vlans
  hosts: "nyc-cor-r1"
  gather_facts: false
  connection: network_cli
  vars:
    vlan_numbers: [100, 200, 300]
  tasks:
    - name: add vlans
      nxos_config:
        lines:
            - vlan {{ item }}
      loop: "{{ vlan_numbers }}"
      register: output
```

The playbook output is as follows:

```
(venv) $ ansible-playbook -i hosts_full standard_loop
_vlan_example.yml

PLAY [Add Multiple Vlans] ******************************************
**************************************************

TASK [add vlans] ***************************************************
**************************************************
changed: [nyc-cor-r1] => (item=100)
changed: [nyc-cor-r1] => (item=200)
changed: [nyc-cor-r1] => (item=300)
[WARNING]: To ensure idempotency and correct diff the input configuration
lines should be similar to how they appear if
present in the running configuration on device

PLAY RECAP *********************************************************
**************************************************
nyc-cor-r1                 : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

As we can see from the playbook, the loop list can be read from a variable, which gives greater flexibility to the structure of your playbook:

```
…
 vars:
    vlan_numbers: [100, 200, 300]
  tasks:
    …
     loop: "{{ vlan_numbers }}"
```

The standard loop is a great time saver when it comes to performing redundant tasks in a playbook. Let us see how we can loop over a dictionary in the next section.

Looping over Dictionaries

When we need to generate a configuration, we often have an entity with more than one attribute associated with it. If you think about the VLAN example in the last section, each VLAN would have several unique attributes, such as a description, a gateway IP address, and possibly others. Often, we can use a dictionary to represent the entity to incorporate multiple attributes into it.

Let's expand on the previous example to include a dictionary variable in `standard_loop_vlan_example_2.yml`. We defined the dictionary values for three `vlan` each with a nested dictionary for the description and the IP address:

```
---
- name: Add Multiple Vlans
  hosts: "nyc-cor-r1"
  gather_facts: false
  connection: network_cli
  vars:
    vlans: {
        "100": {"description": "floor_1", "ip": "192.168.10.1"},
        "200": {"description": "floor_2", "ip": "192.168.20.1"},
        "300": {"description": "floor_3", "ip": "192.168.30.1"}
      }
  tasks:
    - name: add vlans
      nxos_config:
        lines:
            - vlan {{ item.key }}
      with_dict: "{{ vlans }}"
    - name: configure vlans
      nxos_config:
        lines:
          - description {{ item.value.description }}
          - ip address {{ item.value.ip }}/24
        parents: interface vlan {{ item.key }}
      with_dict: "{{ vlans }}"
```

In the playbook, we configure the first task to add the VLANs by using the key of the items. In the second task, we proceed with configuring the VLAN interfaces using the values within each of the items. Note that we use the `parents` parameter to uniquely identify the section the commands should be checked against. This is due to the fact that the description and the IP address are both configured under the `interface vlan <number>` subsection in the configuration.

Before we execute the command, we need to make sure the layer 3 interface feature is enabled on the `nyc-cor-r1` device:

```
nyc-cor-r1(config)# feature interface-vlan
```

We can run the playbook as we have done previously. We can see the dictionary being looped through:

```
(venv) $ ansible-playbook -i hosts_full standard_loop_vlan_example_2.yml

PLAY [Add Multiple Vlans] ******************************************************
*********************************************

TASK [add vlans] ***************************************************************
*********************************************
changed: [nyc-cor-r1] => (item={'key': '100', 'value': {'description':
'floor_1', 'ip': '192.168.10.1'}})
changed: [nyc-cor-r1] => (item={'key': '200', 'value': {'description':
'floor_2', 'ip': '192.168.20.1'}})
changed: [nyc-cor-r1] => (item={'key': '300', 'value': {'description':
'floor_3', 'ip': '192.168.30.1'}})
[WARNING]: To ensure idempotency and correct diff the input configuration
lines should be similar to how they appear if
present in the running configuration on device

TASK [configure vlans] *********************************************************
*********************************************
changed: [nyc-cor-r1] => (item={'key': '100', 'value': {'description':
'floor_1', 'ip': '192.168.10.1'}})
changed: [nyc-cor-r1] => (item={'key': '200', 'value': {'description':
'floor_2', 'ip': '192.168.20.1'}})
changed: [nyc-cor-r1] => (item={'key': '300', 'value': {'description':
'floor_3', 'ip': '192.168.30.1'}})

PLAY RECAP *********************************************************************
*********************************************
nyc-cor-r1                 : ok=2    changed=2    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

We can verify the end result on the device:

```
nyc-cor-r1# sh run
interface Vlan100
  description floor_1
  ip address 192.168.10.1/24
```

```
interface Vlan200
  description floor_2
  ip address 192.168.20.1/24

interface Vlan300
  description floor_3
  ip address 192.168.30.1/24
```

For more loop types of Ansible, feel free to check out the corresponding documentation (`https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html`).

Looping over dictionaries takes some practice the first few times you use them. But just like standard loops, looping over dictionaries will be an invaluable tool in our tool belt. An Ansible loop is a tool that can save us time and make the playbook more readable. In the next section, we will look at Ansible templates that allow us to make systematic changes to text files commonly used for network device configuration.

Templates

Ever since I started working as a network engineer, I have always used some kind of network templating system. In my experience, many of the network devices have sections of the network configuration that are identical, especially if these devices serve the same role in the network.

Most of the time, when we need to provision a new device, we use the same configuration in the form of a template, replace the necessary fields, and copy the file over to the new device. With Ansible, you can automate all of the work by using the templating feature (`https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html`).

Ansible uses Jinja (`https://jinja.palletsprojects.com/en/3.1.x/`) templating to enable dynamic expressions and access to variables and facts. Jinja has its own syntax and method of doing loops and conditionals; fortunately, we just need to know the very basics of it for our purpose. The Ansible template module is an important tool that we will be using in our daily tasks, and we will spend more of this section exploring it. We will learn the syntax by gradually building up our playbook from some simple tasks to more complex ones.

The basic syntax for template usage is very simple; we just need to specify the source file and the destination location that we want to copy it to.

Let us create a new directory called `Templates` and start to create our playbooks. We will create an empty file for now:

```
(venv) $ mkdir Templates
(venv) $ cd Templates/
(venv) $ touch file1
```

Then we will use the following playbook, `template_1.yml`, to copy `file1` to `file2`. Note that the playbook is executed on the control machine only:

```
---
- name: Template Basic
  hosts: localhost
  tasks:
    - name: copy one file to another
      template:
        src=/home/echou/Mastering_Python_Networking_Fourth_Edition/
Chapter04/Templates/file1
        dest=/home/echou/Mastering_Python_Networking_Fourth_Edition/
Chapter04/Templates/file2
```

Executing the playbook will create a new file:

```
(venv) $ ansible-playbook -i hosts template_1.yml

PLAY [Template Basic] **********************************************
**********************************************

TASK [Gathering Facts] *********************************************
**********************************************
ok: [localhost]

TASK [copy one file to another] ************************************
**********************************************
changed: [localhost]

PLAY RECAP *********************************************************
**********************************************
localhost                  : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

```
(venv) $ ls file*
file1  file2
```

In our templates, the source files can have any extension, but since they are processed through the Jinja2 template engine, let's create a text file called `nxos.j2` as the template source. The template will follow the Jinja convention of using double curly braces to specify the variables, as well as using the curly brace plus the percentage sign to specify commands:

```
hostname {{ item.value.hostname }}

feature telnet
feature ospf
feature bgp
feature interface-vlan

{% if item.value.netflow_enable %}
feature netflow
{% endif %}

username {{ item.value.username }} password {{ item.value.password }}
role network-operator

{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}

{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
  ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}
```

We can now put together a playbook to create network configuration templates based on the `nxos.j2` file.

The Jinja Template Variables

The `template_2.yml` playbook expands on the previous template example with the following additions:

- The source file is `nxos.j2`.
- The destination filename is now a variable taken from the `nexus_devices` variable defined in the playbook.
- Each of the devices within `nexus_devices` contains the variables that would be substituted or looped over within the template.

The playbook might look more complex than the last one, but if you take out the variable definition portion, it is very similar to our simple template playbook from earlier:

```
---
- name: Template Looping
  hosts: localhost

  vars:
    nexus_devices: {
        "nx-osv-1": {
            "hostname": "nx-osv-1",
            "username": "cisco",
            "password": "cisco",
            "vlans": [100, 200, 300],
            "l3_vlan_interfaces": True,
            "vlan_interfaces": [
                {"int_num": "100", "ip": "192.168.10.1"},
                {"int_num": "200", "ip": "192.168.20.1"},
                {"int_num": "300", "ip": "192.168.30.1"}
            ],
            "netflow_enable": True
        },
        "nx-osv-2": {
            "hostname": "nx-osv-2",
            "username": "cisco",
            "password": "cisco",
            "vlans": [100, 200, 300],
            "l3_vlan_interfaces": False,
```

```
            "netflow_enable": False
        }
    }
  tasks:
    - name: create router configuration files
      template:
        src=/home/echou/Mastering_Python_Networking_Fourth_Edition/
Chapter04/Templates/nxos.j2
        dest=/home/echou/Mastering_Python_Networking_Fourth_Edition/
Chapter04/Templates/{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
```

Let us not execute the playbook just yet; we still need to take a look at the `if` conditional statements and `for` loops enclosed within the `{% %}` symbols from the `Jinja2` template.

Jinja Template Loops

There are two `for` loops in our `nxos.j2` template; one loops over the VLANs and the other loops over the VLAN interfaces:

```
{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}
{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}
  ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}
```

If you recall, we can also loop through a list as well as a dictionary in Jinja. In our example, the `vlans` variable is a list, while the `vlan_interfaces` variable is a list of dictionaries.

The `vlan_interfaces` loop is nested inside a conditional. This is the last thing that we will incorporate into our playbook before we execute the playbook.

Jinja Template Conditional

Jinja supports an `if` conditional check. We have added this conditional statement in two locations within the `nxos.j2` template; one is with the `netflow` variable and the other is the `l3_vlan_interfaces` variable. Only when the condition is `True` will we execute the statements within the block:

```
<skip>
{% if item.value.netflow_enable %}
feature netflow
{% endif %}
<skip>
{% if item.value.l3_vlan_interfaces %}
<skip>
{% endif %}
```

In the playbook, we have declared `netflow_enable` to be `True` for `nx-os-v1` and `False` for `nx-osv-2`:

```
  vars:
    nexus_devices: {
        "nx-osv-1": {
            <skip>
            "netflow_enable": True
        },
        "nx-osv-2": {
            <skip>
            "netflow_enable": False
        }
    }
```

Finally, we are ready to run our playbook:

```
(venv) $ ansible-playbook -i hosts template_2.yml

PLAY [Template Looping] ********************************************************
**********************************************

TASK [Gathering Facts] *********************************************************
**********************************************
```

```
ok: [localhost]

TASK [create router configuration files] ********************************
*****************************************
changed: [localhost] => (item={'key': 'nx-osv-1', 'value': {'hostname':
'nx-osv-1', 'username': 'cisco', 'password': 'cisco', 'vlans': [100, 200,
300], 'l3_vlan_interfaces': True, 'vlan_interfaces': [{'int_num': '100',
'ip': '192.168.10.1'}, {'int_num': '200', 'ip': '192.168.20.1'}, {'int_
num': '300', 'ip': '192.168.30.1'}], 'netflow_enable': True}})
changed: [localhost] => (item={'key': 'nx-osv-2', 'value': {'hostname':
'nx-osv-2', 'username': 'cisco', 'password': 'cisco', 'vlans': [100, 200,
300], 'l3_vlan_interfaces': False, 'netflow_enable': False}})

PLAY RECAP **************************************************************
*****************************************
localhost                  : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Do you remember that the destination files are named after the `{{ item.key }}.conf`? Two files have been created with the device names:

```
$ ls nx-os*
nx-osv-1.conf
nx-osv-2.conf
```

Let's check the similarities and differences of the two configuration files to make sure all of our intended changes are in place. Both files should contain the static items, such as `feature ospf`, the hostnames and other variables should be substituted accordingly, and only `nx-osv-1.conf` should have `netflow` enabled as well as the layer 3 `vlan` interface configuration:

```
$ cat nx-osv-1.conf
hostname nx-osv-1
feature telnet
feature ospf
feature bgp
feature interface-vlan
feature netflow
username cisco password cisco  role network-operator
vlan 100
vlan 200
```

```
vlan 300
interface 100
  ip address 192.168.10.1/24
interface 200
  ip address 192.168.20.1/24
interface 300
  ip address 192.168.30.1/24
```

Let's take a look at the `nx-osv-2.conf` file:

```
$ cat nx-osv-2.conf
hostname nx-osv-2
feature telnet
feature ospf
feature bgp
feature interface-vlan
username cisco password cisco  role network-operator
vlan 100
vlan 200
vlan 300
```

Neat, huh? This can certainly save us a ton of time for something that previously required repeated copying and pasting. Personally, the template module was a big game-changer for me. This module alone was enough to motivate me to learn and use Ansible a few years ago.

Summary

In this chapter, we took a grand tour of the open-source automation framework Ansible. Unlike Pexpect-based and API-driven network automation scripts, Ansible provides a higher layer of abstraction called a playbook to automate our network devices.

Ansible is a full-featured automation framework capable of managing large infrastructures. Our focus is on managing network devices, but Ansible is capable of managing servers, databases, cloud infrastructures, and more. We have only touched the surface of its capabilities. If you feel Ansible is a tool you would like to learn more about, the Ansible documentation is an excellent source of reference. The Ansible community is friendly and welcoming if you would like to get involved.

In *Chapter 5*, *Docker Containers for Network Engineers*, we will start to learn about Docker and the world of containers.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



5

Docker Containers for Network Engineers

Computer hardware virtualization has revolutionized and changed the way we approach infrastructure. Gone are the days when we must dedicate hardware to a single host and operating system. We now have the option to share precious hardware such as CPU, memory, and disk space with multiple virtual machines, each with its own operating system and applications. Because software executed on these virtual machines is separated from the underlying hardware resources, we are free to allocate a different combination of hardware resources to virtual machines based on their specific needs. Nowadays, it is hard to imagine a world without virtual machines.

As much as virtual machines are great for application building, they do take a while to build, spin up, and, ultimately, tear down. The reason is that the virtualization technology associated with virtual machines completely simulates the actual hardware for which the hardware is indistinguishable from the guest virtual machines.

The question might now be: is there a way to speed up the life cycle of applications with even more virtualization? The answer is yes, with the help of containers.

Containers and virtual machines are similar in that they both allow sharing of computing resources amongst different isolated applications. The difference is that virtual machines are abstracted at the Hypervisor level, whereas containers are abstracted within the operating system by a container engine. Containers are often referred to as OS-level virtualization.

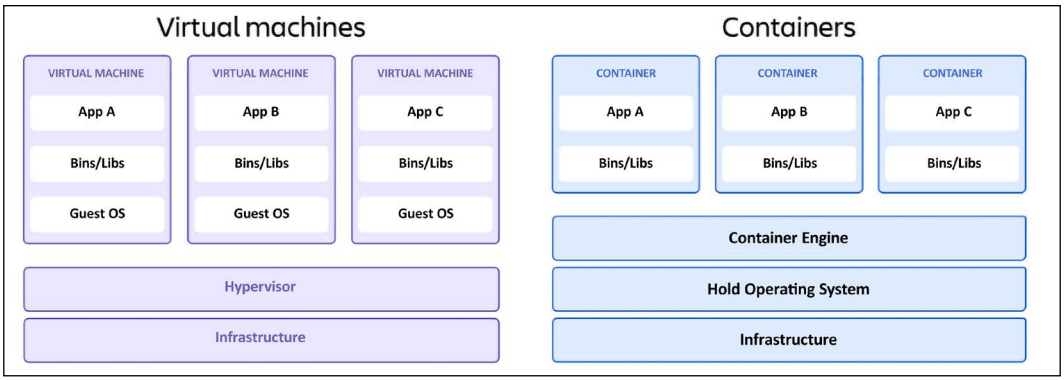


Figure 5.1: Virtual Machine and Container Comparison (source: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms)

In a full virtual machine, we can install different operating systems, such as Windows and Linux. Because container virtualization is being handled by the operating system, each container will have the same operating system. However, the application and its associated resources will be isolated and run independently of each other. The container engine will separate the configuration, software bundle, and libraries from each container.

Container virtualization is not new; **Linux containers** (**LXC**), Solaris containers, Docker, and Podman are examples of such implementation. In this chapter, we will look at the most popular container technology today, Docker. We will discuss the following topics related to Docker containers:

- Docker overview
- Building Python applications with Docker
- Container networking
- Containers in the network engineering field
- Docker and Kubernetes

We will be using containers for some of the technologies we will learn in this book; this is a good place to start getting familiar with containers.

Let's start by looking at a high-level overview of Docker.

Docker Overview

Docker is a set of products and tools that supports the delivery of containers. It was started by the company dotCloud in 2008 (renamed to Docker, Inc. in 2013). The set of tools includes the container technology of Docker, the container engine called Docker Engine, the cloud-based repository of containers called Docker Hub, and the desktop graphical user interface software called Docker Desktop.

Docker has two versions, **Docker Community Edition (Docker-CE)** and **Docker Enterprise Edition (Docker-EE)**. Docker-CE is a free and open-source platform based on the Apache 2.0 license, while Docker-EE is a premium version geared toward enterprises. When the term "Docker" is mentioned in this book, we are referring to the Community Edition.

There are three main components in a Docker container environment:

1. Building and Development: These include the tools used to build a container, including the CLI commands, the images, and the repositories where we get the various base images. In Docker, we use a Dockerfile to specify most of the building steps for a container.
2. Docker Engine: This is the daemon running in the background. We can use the Docker command to manage the daemon.
3. Container Orchestration: During development, we will typically use `Docker-compose` from Docker to manage a multi-container environment. In production, a common tool is a Google-originated tool called Kubernetes (`https://kubernetes.io/`).

In the next section, we will discuss the advantages of Docker.

Advantages of Docker

There are many advantages of Docker. We will summarize some of them here:

1. Docker containers are fast to deploy and destroy.
2. Containers reset gracefully. The containers are transient and ephemeral, leaving no residual artifacts when restarted. This leaves a clean state whenever a new container is spawned.
3. It is self-contained and deterministic. Containers are often delivered with configuration files with instructions on how the container can be rebuilt. We can be sure each container image is built in the same way.

4. It allows seamless integration between application development and DevOps. Because of the advantages stated above, many companies have deployed. Docker images directly in the production environment. The container can be reproduced exactly as the developer intended and tested into production.

Now that we have a general understanding of Docker, it is time to build our first Python applications in a Docker container.

Building Python applications in Docker

A Docker container is a very popular way to build Python applications.

Installing Docker

Of course, we will need to install Docker to start using it. We will follow DigitalOcean's excellent installation guide for Ubuntu 22.04 (`https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-22-04`). If you are using other versions of the Linux distribution, you can simply use the drop-down menu from the documentation to pick a different version. For installation on Mac or Windows, my recommendation would be to install Docker Desktop (`https://docs.docker.com/desktop/`). It will include the Docker Engine, CLI client, and GUI application.

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
$ sudo apt install apt-transport-https ca-certificates curl software-
properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/
keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/
ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/
docker.list > /dev/null
$ sudo apt update
$ apt-cache policy docker-ce
$ sudo apt install docker-ce
```



There are some optional but useful post-installation steps for Linux at `https://docs.docker.com/engine/install/linux-postinstall/`.

We can check the status of our Docker installation:

```
$ sudo systemctl status docker

● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor
preset: enabled)
     Active: active (running) since Sun 2022-09-11 15:02:27 PDT; 5s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
```

In the next section, we will see how we can build a Python application in Docker containers.

Useful Docker commands

We will need to use some commands to build, run, and test our containers.



For more Docker CLI references, check out the documentation: `https://docs.docker.com/engine/reference/run/`.

Here are some of the commands we will be using in this chapter:

- `docker run`: `docker run` is used to specify the image to derive the container from (by default, it is Docker Hub), network settings, name, and other settings.
- `docker container ls`: lists the containers; by default, it only lists currently running containers.
- `docker exec`: runs a command on a running container.
- `docker network`: used when we need to manage Docker networks, such as to create, list, and remove Docker networks.
- `docker image`: manages Docker images.

There are many more CLI commands, but these are enough to get us started. For a complete reference, check out the link provided in the information box.

Building hello world

The first step is to make sure we have reachability to Docker Hub to retrieve an image. To do so, Docker provides a very simple `hello-world` app:

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which
sent it
    to your terminal.
<skip>
```

We can see the various steps the Docker client needed to do to display the message. We can display the Docker processes that ran:

```
$ docker ps -a
CONTAINER ID   IMAGE                  COMMAND           CREATED
STATUS                          PORTS     NAMES
3cb4f91b6388   hello-world            "/hello"          About a minute
ago   Exited (0) About a minute ago             fervent_torvalds
```

We can see the `hello-world` image information:

```
$ docker images hello-world
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    feb5d9fea6a5   11 months ago   13.3kB
```

Now we can build our first Python application.

Building our application

Let's start by thinking about what we will build. Since we built a few Ansible playbooks in the last chapter, how about we containerize the `ios_config_backup.yml` playbook so we can share this with other team members?

We will create a new folder to keep all the files together. If you recall, for us to build a Docker image, there is a special file called a Dockerfile. We will also create such a file in the directory:

```
$ mkdir ansible_container && cd ansible_container
$ touch Dockerfile
```

We will also copy the *host_vars* folder, *ansible.cfg*, *hosts*, and *ios_config_backup.yml* files into this folder. We should also make sure the playbook runs as expected before we build the Docker container from it.

Docker builds itself in a layered fashion, starting with a base image. In the Dockerfile, we will specify the following lines:

```
# Getting base image
FROM ubuntu:22.04

# No need for interactive prompt
ENV DEBIAN_FRONTEND=noninteractive
```

The lines starting with a "#" mark are comments, just like in Python. The `FROM` keyword specifies the base image we will retrieve from the default Docker Hub. All the official Ubuntu images can be found on the site, `https://hub.docker.com/_/ubuntu`. In the `ENV` statement, we specified no need for interactive prompts.



The Dockerfile reference can be viewed at `https://docs.docker.com/engine/reference/builder/`.

Let us build this image:

```
$ docker build --tag ansible-docker:v0.1 .
```

The `build` command builds from the Dockerfile in the local directory while tagging the final image to be `ansible-docker` with version 0.1. Once completed, we can view the image:

```
$ docker images
REPOSITORY              TAG             IMAGE ID       CREATED
SIZE
ansible-docker          v0.1            e99f103e2d36   3 seconds ago
864MB
```



If we need to remove an image before the rebuild, we can delete the image with "`docker rmi <image id>`."

We can start the container based on the image:

```
$ docker run -it --name ansible-host1 ansible-docker:v0.1
root@96108c94e1d2:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:        22.04
Codename:       jammy
root@96108c94e1d2:/#
```

It will drop us into the bash shell prompt, and the container will stop itself once we exit. In order for it to run in a detached mode, we will need to start it with a "`-d`" flag. Let's go ahead and delete the container and recreate it with the flag:

```
$ docker ps -a
CONTAINER ID   IMAGE                 COMMAND   CREATED         STATUS
PORTS     NAMES
<container id>   ansible-docker:v0.1   "bash"    2 minutes ago   Exited
(0) 52 seconds ago             ansible-host1
$ docker rm <container id>
$ docker run -it -d --name ansible-host1 ansible-docker:v0.1
```



Remember to substitute your container ID. A nice shortcut to delete all containers in one setting is `docker rm -f $(docker ps -a -q)`.

The container now runs in detached mode, and we can execute an interactive prompt on the container:

```
$ docker ps
CONTAINER ID   IMAGE                 COMMAND   CREATED              STATUS
PORTS     NAMES
d3b6a6ec90e5   ansible-docker:v0.1   "bash"    About a minute ago   Up 58
seconds             ansible-host1
$ docker exec -it ansible-host1 bash
root@d3b6a6ec90e5:/# ls
```

We can go ahead and stop the container, then delete it:

```
$ docker stop ansible-host1
$ docker rm ansible-host1
```

We will introduce a few more Dockerfile commands:

```
# Getting base image
FROM ubuntu:22.04

# No need for interactive prompt
ENV DEBIAN_FRONTEND=noninteractive

# Run any command, i.e. install packages
RUN apt update && apt install -y python3.10 python3-pip ansible vim
RUN pip install ansible-pylibssh

# specify a working directory
WORKDIR /app
COPY . /app
```

The `RUN` command executes the shell commands as if we were typing them in the shell. We can specify the working directory as `/app` on the container, then copy everything in the current working directory (`host_vars`, hosts, playbook, etc.) to the `/app` directory on the remote container.

```
$ docker images
<find the image id>
$ docker rmi <image id>
$ docker build --tag ansible-docker:v0.1 .
```



We will keep the same tag, but if we would like to make it a new release, we can always tag it as `v0.2`.

We will launch the container again and execute the `ansible-playbook`:

```
$ docker run -it -d --name ansible-host1 ansible-docker:v0.1
docker exec -it ansible-host1 bash
root@5ef5e9c85065:/app# pwd
/app
root@5ef5e9c85065:/app# ls
ansible.cfg  dockerfile  host_vars  hosts  ios_config_backup.yml
root@5ef5e9c85065:/app# ansible-playbook -i hosts ios_config_backup.yml

PLAY [Back Up IOS Device Configurations] ******************************
**********************************

TASK [backup] *********************************************************
**********************************
changed: [iosv-2]
changed: [iosv-1]

PLAY RECAP ************************************************************
**********************************
iosv-1                     : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
iosv-2                     : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0

root@5ef5e9c85065:/app# ls backup/
iosv-1_config.2022-09-12@23:01:07  iosv-2_config.2022-09-12@23:01:07
```

Once the container is launched, we can start and stop via the hostname:

```
$ docker stop ansible-host1
$ docker start ansible-host1
```

Congratulations on working through the complete container workflow! This might not seem much now, but it is a big step. The steps might seem a bit foreign now, but don't worry, they will become more familiar as we get more practice under our belt.

Sharing Docker images

The last step will be to share the container images. One way to do it would be to tar zip the directory and share the file. Another way is to push the image to a repository accessible to whoever needs access. Docker Hub is one of the most popular repositories, but many others exist. They generally offer several different subscription price tiers.

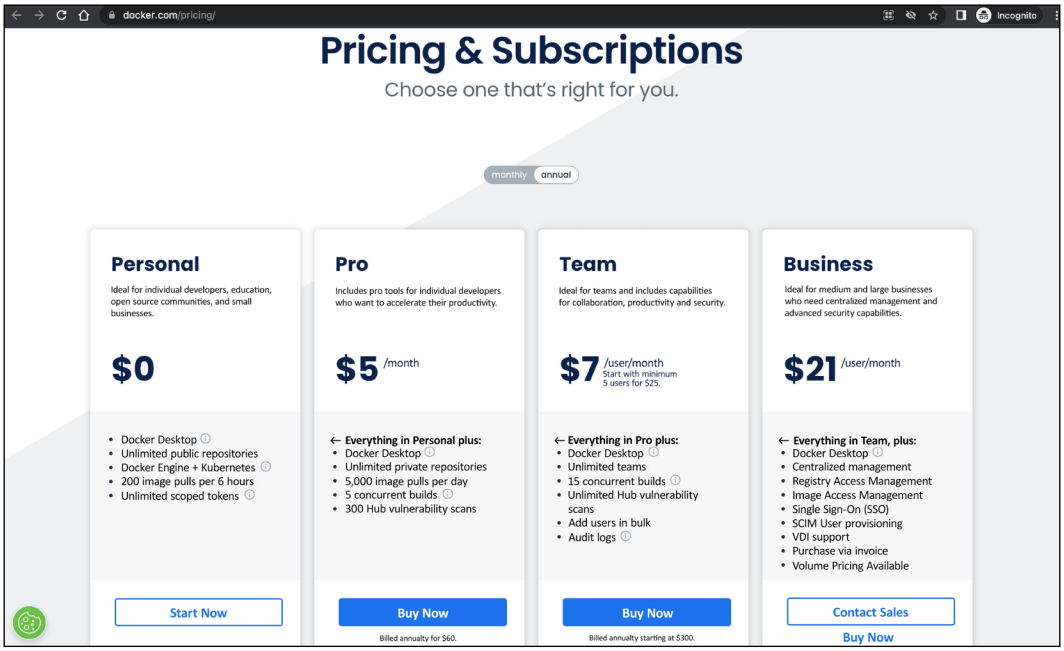


Figure 5.2: Docker Hub Pricing (source: `https://www.docker.com/pricing/`*)*

Besides sharing the container image, having an accessible repository is crucial in a DevOps **CI/CD (Continuous Integration/Continuous Delivery)** process. For example, we might be checking in the code with an automated build and test process. Once all the validation test passes, we can automatically push the image to the repository and deploy it to production. We will create a private repository on Docker Hub:

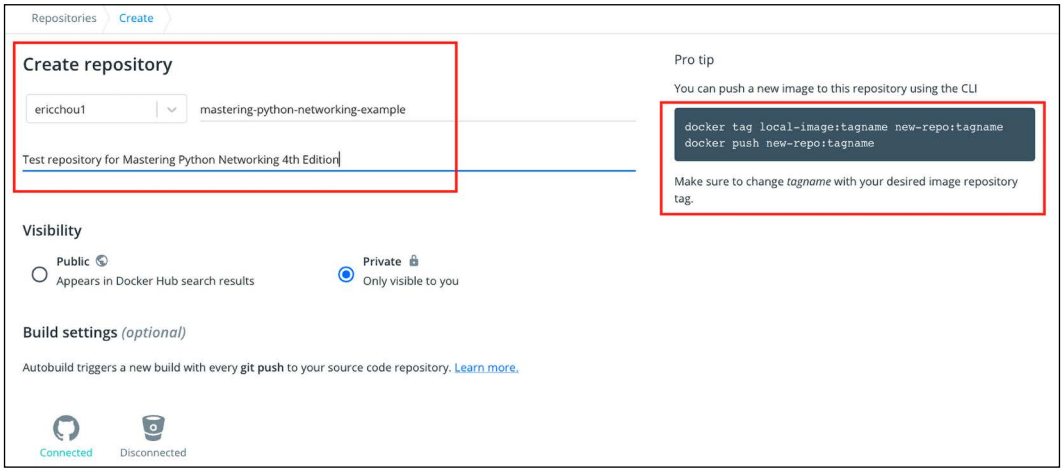


Figure 5.3: Docker Hub Repository

Then we will log in via the Docker CLI:

```
$ docker login
```

Then we can tag the existing image following the remote repository, then push toward it. Notice in the output below that the destination tag name matches the repository name on Docker Hub. This allows flexibility in local naming while conforming to the remote team naming conventions.

```
$ docker tag ansible-docker:v0.1 ericchou1/mastering-python-networking-
example:ch05-ansible-dockerv-v0.1
$ docker push ericchou1/mastering-python-networking-example:ch05-ansible-
dockerv-v0.1
```

Once the image finishes uploading, we can access the image and we can use it directly or use it as a base image in another Dockerfile.

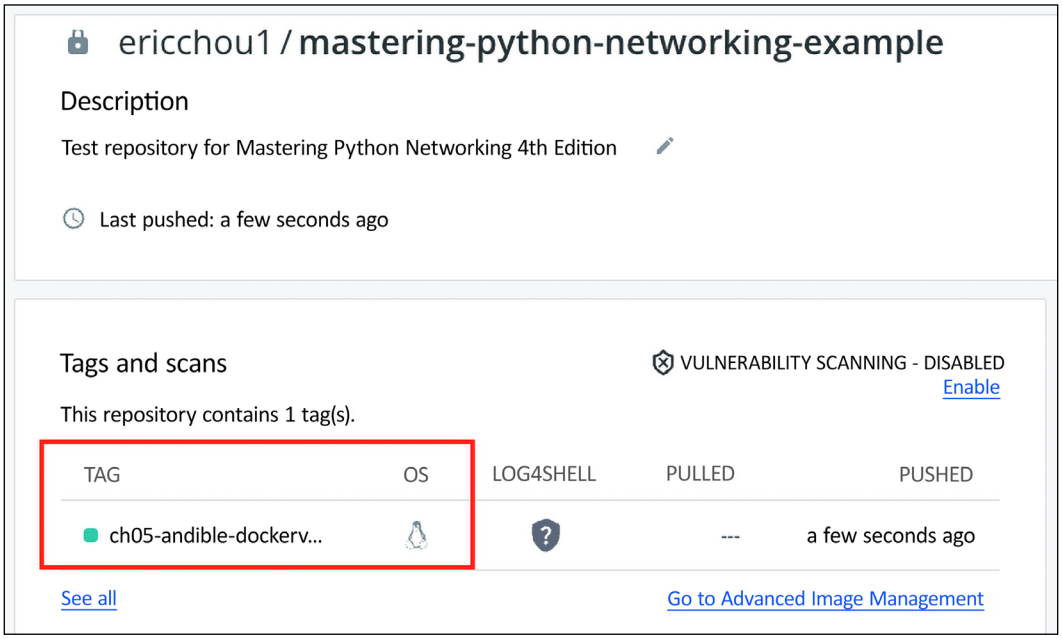


Figure 5.4: New Uploaded Image

In the next section, we will see how to coordinate multi-container setup locally during development.

Container orchestration with Docker-compose

Modern applications often have interdependencies with each other. For example, for a web application, we usually have a "stack" of applications. The popular LAMP stack is an acronym denoting Linux, Apache, MySQL, and PHP/Python to specify the components required to deliver a web application. In the world of Docker, we can use docker-compose (`https://docs.docker.com/compose/`) to specify how multiple containers should be built and run simultaneously.

If you have installed Docker Desktop for Mac or Windows, docker-compose is already included. In the Linux environment, docker-compose needs to be installed separately. We will follow DigitalOcean's guide for docker-compose (`https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-22-04`):

```
$ mkdir -p ~/.docker/cli-plugins/
$ curl -SL https://github.com/docker/compose/releases/download/v2.3.3/
docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-compose
$ chmod +x ~/.docker/cli-plugins/docker-compose
$ docker compose version
Docker Compose version v2.3.3
```

Docker-compose uses a YAML file named `docker-compose.yml` to construct the environment. There are lots of knobs to specify different service dependencies, persistent volumes, and opening public ports. Let's put together a simple example:

```
version: '3.9'
services:
  ansible:
    build:
      dockerfile: dockerfile
  db:
    image: postgres:14.1-alpine
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    ports:
      - '5432:5432'
    volumes:
      - db:/var/lib/postgresql/data
volumes:
  db:
    driver: local
```

Here is what the file specifies:

1. The file specifies two services, `ansible` and `db`. Each of the services is similar to the `docker run` commands.

2. The `ansible` service builds with the current Dockerfile in the current working directory named `dockerfile`.
3. We map the host port 5434 to the container port 5434.
4. We specify two environmental variables for the Postgres database.
5. We use a volume named `db` so that the database information written is persisted in the volume.



For more information on Docker-compose, please visit `https://docs.docker.com/compose/`.

We can run the combined service with the *docker-compose* command:

```
$ docker compose up
…
 Container ansible_container-db-1       Created
0.0s
  Container ansible_container-ansible-1  Created
0.0s
ansible_container-db-1       |
ansible_container-db-1       | PostgreSQL Database directory appears to
contain a database; Skipping initialization
ansible_container-db-1       |
ansible_container-db-1       | 2022-09-13 00:18:45.195 UTC [1] LOG:
starting PostgreSQL 14.1 on x86_64-pc-linux-musl, compiled by gcc (Alpine
10.3.1_git20211027) 10.3.1 20211027, 64-bit
ansible_container-db-1       | 2022-09-13 00:18:45.196 UTC [1] LOG:
listening on IPv4 address "0.0.0.0", port 5432
ansible_container-db-1       | 2022-09-13 00:18:45.196 UTC [1] LOG:
listening on IPv6 address "::", port 5432
ansible_container-db-1       | 2022-09-13 00:18:45.198 UTC [1] LOG:
listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
ansible_container-db-1       | 2022-09-13 00:18:45.201 UTC [21] LOG:
database system was shut down at 2022-09-13 00:18:36 UTC
ansible_container-db-1       | 2022-09-13 00:18:45.204 UTC [1] LOG:
database system is ready to accept connections
…
```

The services are launched concurrently. We can then tear down both services:

```
$ docker compose down
[+] Running 3/3
  Container ansible_container-db-1       Removed
0.2s
  Container ansible_container-ansible-1  Removed
0.0s
  Network ansible_container_default      Removed
0.1s
```

We have only built simple applications thus far in the book. This might make more sense when we learn about building a Web API later in the book. For now, it is good to consider how we can launch multiple containers via docker-compose.

As network engineers, it would be interesting to know how networking is done in a Docker environment. That is the subject of the next section.

Container networking

Container networking is not an easy topic to cover because of its scope and the number of technologies it touches. The space spans from Linux networking, how the particular type of Linux (Ubuntu, Red Hat, etc.) implements networking, to Docker's implementation of networking. Adding to the complexity is the fact that Docker is a fast-moving project, and many third-party plugins are available.

In this section, we will stick to the basics of the networking options offered by Docker by default. We will then briefly explain the options of overlay, Macvlan, and network plugins.

When we launch a container, it can reach the internet by default. Let's do a quick test by launching an Ubuntu container and attaching to it:

```
$ docker run -it ubuntu:22.04
<container launches and attached>
root@dcaa61a548be:/# apt update && apt install -y net-tools iputils-ping
root@dcaa61a548be:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
<skip>
```

```
root@dcaa61a548be:/# ping -c 1 www.cisco.com
PING e2867.dsca.akamaiedge.net (104.71.231.76) 56(84) bytes of data.
64 bytes from a104-71-231-76.deploy.static.akamaitechnologies.com
(104.71.231.76): icmp_seq=1 ttl=53 time=11.1 ms

--- e2867.dsca.akamaiedge.net ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 11.147/11.147/11.147/0.000 ms
```

We can see the host has a private IP different from our host's IP. It can also reach the Ubuntu repository to install software as well as ping the outside network. How does it do that? By default, Docker created three types of networks: `bridge`, `host`, and `none`. Let's launch a second Terminal window while keeping the host running in the first Terminal window:

```
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
78e7ab7ea276   bridge    bridge    local
93c142329fc9   host      host      local
da9fe0ed2308   none      null      local
```

The *none* network option is straightforward. It disables all networking and makes the container sit on a network island by itself. This leaves us with the `bridge` and `host` options. By default, Docker puts the host in the bridge network, `docker0`, with a **virtual Ethernet (veth)** interface (`https://man7.org/linux/man-pages/man4/veth.4.html`) to allow it to communicate to the internet:

```
$ ip link show
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 02:42:86:7f:f2:40 brd ff:ff:ff:ff:ff:ff
21: veth3fda84e@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master docker0 state UP mode DEFAULT group default
    link/ether 9a:f8:83:ae:cb:ea brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

If we launch another container, we will see an additional veth interface created and put into the same bridge group. By default, they can communicate with each other.

Container host network

We can also share the host network with the container. Let's start an Ubuntu container in the host network. We will also install Python 3.10 and other software packages:

```
$ docker run -it --network host ubuntu:22.04
root@network-dev-4:/# apt update && apt install -y net-tools iputils-ping
python3.10 vim
root@network-dev-4:/# ifconfig ens160
ens160: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.126  netmask 255.255.255.0  broadcast 192.168.2.255
```

If we check now, we can see the container now shares the same IP as the host network. We can create a simple HTML page and start the Python3 built-in web server on the container:

```
root@network-dev-4:/# cat index.html
<html>
<head></head>
<body><h1>Hello Networkers!</h1></body>
</html>
root@network-dev-4:/# python3.10 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

If we open up the IP address with port 8000 in a browser, we can see the page we created!



Figure 5.5: Index Page of Container Host



If you have a firewall on your host (such as iptables or ufw) turned on, make sure to open up port 8000 so you can see the page.

The host network option is useful when we need to expose containers for public service.

Custom bridge network

We can also create custom bridge networks and group containers together. We will create the network first:

```
$ docker network create network1
```

We can now assign the containers to the custom bridge network:

```
$ docker run -it --network network1 ubuntu:22.04
root@41a977cd9c5b:/# apt update && apt install -y net-tools iputils-ping
root@41a977cd9c5b:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.2  netmask 255.255.0.0  broadcast 172.18.255.255
<skip>
root@41a977cd9c5b:/# ping -c 1 www.cisco.com
PING e2867.dsca.akamaiedge.net (23.206.3.148) 56(84) bytes of data.
64 bytes from a23-206-3-148.deploy.static.akamaitechnologies.com
(23.206.3.148): icmp_seq=1 ttl=53 time=13.2 ms
```

The host is now in its custom bridge network. It has network access to the public internet and other containers in the same bridge network. If we want to expose a particular port to a container in the custom bridge network, we can use the `–publish` option to map the port to the `local host`:

```
$ docker run -it --network network1 --publish 8000:8000 ubuntu:22.04
```

We can remove the network via the `docker network rm`:

```
$ docker network ls
NETWORK ID     NAME       DRIVER    SCOPE
30aa5d7887bc   network1   bridge    local

$ docker network rm network1
```

The custom network option is great for developing multi-container projects that need isolation from each other. Up to this point, we have been looking at networking options in a single host. In the next section, we will see the options for inter-host communication between containers.

Other container network options

If we look closely at the `docker network ls` output, we can see the columns of `driver` and `scope`. Docker's network subsystem is pluggable, using drivers. The core networking functions were provided by the default drivers of `bridge`, `host`, and `none`.

Other notable drivers are listed below:

- Overlay: The overlay network creates a distributed network among multiple Docker daemon hosts.
- Macvlan: The macvlan network option is meant for applications needing to be directly connected to the physical network.
- Third-party network plugins: We can install third-party network plugins (`https://hub.docker.com/search?q=&type=plugin`) for additional features. For example, the vSphere-storage plugin (`https://hub.docker.com/r/vmware/vsphere-storage-for-docker`) enables customers to address persistent storage requirements for containers in a vSphere environment.

An overlay network driver is probably the option we will need to use beyond the development stage. It is meant to handle the routing of the packets to and from the Docker daemon host and the correct destination container. For example, an overlay ingress network would handle the incoming traffic and load balance to the correct container. Due to its complexity, this is typically handled by the orchestration tool of choice, such as Swarm or Kubernetes. If we use a public cloud provider, such as Google Kubernetes Engine, they might even handle this overlay network for us.

Containers in the network engineering field

Container technologies are transforming how infrastructure is built in modern days. We now have an additional layer of abstraction we can use to overcome limitations on physical space, power, cooling, and other factors. This is especially true of the need to move toward more environmentally-friendly data centers.

There are many new challenges and opportunities associated with the new container-based world:

- Networking in the container world. As we saw in the last section, there are lots of options that exist when it comes to networking in containers.
- DevOps. One of the challenges when trying to implement DevOps practices in network engineering is the lack of options for flexible, virtualized network devices. Containers can potentially solve that problem if we can virtualize our network along with the hosts.

- Lab and Testing. If we can virtualize our network via container images, this makes lab and testing much easier.

We will discuss DevOps in *Chapter 12*, *Continuous Integration with GitLab*; in the next section, we will look at a new way to test and run containerized network operating systems.

Containerlab

Containerlab (`https://containerlab.dev/`) is a way to run containerized network operating systems. It is a project started by the team at Nokia led by Roman Dodin (`https://twitter.com/ntdvps`). The team is also responsible for developing **SR Linux** (**Service Router Linux**), an open **network operating system** (**NOS**). Although born out of Nokia, Containerlab has multi-vendor support with Arista cEOS, Azure SONiC, Juniper cRPD, and many others. Let's do a quick example to illustrate the workflow of Containerlab. To install, we can follow the installation steps (`https://containerlab.dev/install/`) for Debian-based systems. To isolate the installation, we can create a new directory:

```
$ mkdir container_lab && cd container_lab
$ echo "deb [trusted=yes] https://apt.fury.io/netdevops/ /" | sudo tee -a
/etc/apt/sources.list.d/netdevops.list
$ sudo apt update && sudo apt install containerlab
```

We will define a `clab` file to define the topology, image, and starting configurations. There are several example labs under `/etc/containerlab/lab-examples/`. We will use the two-node lab example (`https://github.com/srl-labs/containerlab/blob/main/lab-examples/srl02/srl2.cfg`) with two SR Linux devices connected over an Ethernet interface. Since SR Linux container images can be downloaded over a public repository, this will save us the step of needing to download the container image separately. We will call this lab topology `srl02.clab.yml`:

```
# topology documentation: http://containerlab.dev/lab-examples/two-srls/
# https://github.com/srl-labs/containerlab/blob/main/lab-examples/srl02/
srl02.clab.yml
name: srl02

topology:
  nodes:
    srl1:
      kind: srl
      image: ghcr.io/nokia/srlinux
      startup-config: srl1.cfg
```

```
    srl2:
      kind: srl
      image: ghcr.io/nokia/srlinux
      startup-config: srl2.cfg

  links:
    - endpoints: ["srl1:e1-1", "srl2:e1-1"]
```

As indicated in the file, the topology consists of nodes and links. The nodes are the NOS systems, while the links define how they are connected. The two device configuration files are vendor-specific, in this case, SR Linux configurations:

```
$ cat srl1.cfg
set / interface ethernet-1/1
set / interface ethernet-1/1 subinterface 0
set / interface ethernet-1/1 subinterface 0 ipv4
set / interface ethernet-1/1 subinterface 0 ipv4 address 192.168.0.0/31
set / interface ethernet-1/1 subinterface 0 ipv6
set / interface ethernet-1/1 subinterface 0 ipv6 address
2002::192.168.0.0/127

set / network-instance default
set / network-instance default interface ethernet-1/1.0

$ cat srl2.cfg
set / interface ethernet-1/1
set / interface ethernet-1/1 subinterface 0
set / interface ethernet-1/1 subinterface 0 ipv4
set / interface ethernet-1/1 subinterface 0 ipv4 address 192.168.0.1/31
set / interface ethernet-1/1 subinterface 0 ipv6
set / interface ethernet-1/1 subinterface 0 ipv6 address
2002::192.168.0.1/127
```

We can now launch the lab with `containerlab deploy`:

```
$ sudo containerlab deploy --topo srl02.clab.yml
[sudo] password for echou:
INFO[0000] Containerlab v0.31.1 started
INFO[0000] Parsing & checking topology file: srl02.clab.yml
…
```



Technically, we do not need the `–topo` option to specify a topology. Containerlab will look for an `*.clab.y*ml` topology file by default. However, I find it a good practice to specify a topology file in case we have several topology files in the same directory.

If successful, we will see the device information. The device names are in the format of `clab-{ lab name }-{ device name }`:

```
+---+-----------------+--------------+----------------------+------+-----
----+----------------+----------------------+
| # |      Name       | Container ID |         Image        | Kind |
State  |  IPv4 Address  |     IPv6 Address     |
+---+-----------------+--------------+----------------------+------+-----
----+----------------+----------------------+
| 1 | clab-srl02-srl1 | 7cae81c710d8 | ghcr.io/nokia/srlinux | srl  |
running | 172.20.20.2/24 | 2001:172:20:20::2/64 |
| 2 | clab-srl02-srl2 | c75f274284ef | ghcr.io/nokia/srlinux | srl  |
running | 172.20.20.3/24 | 2001:172:20:20::3/64 |
+---+-----------------+--------------+----------------------+------+-----
----+----------------+----------------------+
```

We can access the device via `ssh` to the device; the default username and passwords are both `admin`:

```
$ ssh admin@172.20.20.3
admin@172.20.20.3's password:
Using configuration file(s): []
Welcome to the srlinux CLI.
Type 'help' (and press <ENTER>) if you need any help using this.
--{ running }--[  ]--
A:srl1# show version
------------------------------------------------------------------------------
--------------------------------------------------
Hostname             : srl1
Chassis Type         : 7220 IXR-D2
Part Number          : Sim Part No.
Serial Number        : Sim Serial No.
System HW MAC Address: 1A:85:00:FF:00:00
Software Version     : v22.6.3
```

```
Build Number         : 302-g51cb1254dd
Architecture         : x86_64
Last Booted          : 2022-09-12T03:12:15.195Z
Total Memory         : 1975738 kB
Free Memory          : 219406 kB
------------------------------------------------------------------------------
------------------------------------------------------
--{ running }--[  ]--
A:srl1#
A:srl1# quit
```

A directory is created with the associated files for the lab:

```
$ ls clab-srl02/*
clab-srl02/ansible-inventory.yml  clab-srl02/topology-data.json

clab-srl02/ca:
root  srl1  srl2

clab-srl02/srl1:
config  topology.yml

clab-srl02/srl2:
config  topology.yml
```

We can also see there is an additional bridge network created with the two veth interfaces connected to the bridge network:

```
(venv) $ ip link show
11: br-4807fa9091c5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP mode DEFAULT group default
    link/ether 02:42:72:7a:9d:af brd ff:ff:ff:ff:ff:ff
13: veth3392afa@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master br-4807fa9091c5 state UP mode DEFAULT group default
    link/ether be:f0:1a:f2:12:23 brd ff:ff:ff:ff:ff:ff link-netnsid 1
15: veth7417e97@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master br-4807fa9091c5 state UP mode DEFAULT group default
    link/ether 92:53:d3:ac:20:93 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

We can tear down the lab with the `containerlab destroy` command:

```
$ sudo containerlab destroy --topo srl02.clab.yml
[sudo] password for echou:
INFO[0000] Parsing & checking topology file: srl02.clab.yml
INFO[0000] Destroying lab: srl02
INFO[0001] Removed container: clab-srl02-srl2
INFO[0001] Removed container: clab-srl02-srl1
INFO[0001] Removing containerlab host entries from /etc/hosts file
```

I don't know about you, but Containerlab is the easiest way to launch a networking lab that I have seen. With more vendor support, it might one day become the only lab and testing software we need for network testing.

In the next section, we will briefly discuss the relationship between Docker and Kubernetes with a very brief overview of Kubernetes.

Docker and Kubernetes

As we have seen, Docker images and orchestration can be done with the tools provided by the Docker community. However, it is almost impossible to think about Docker containers without Kubernetes. This is because when it comes to container orchestration, Kubernetes is becoming the de facto standard in doing so. There is not enough space to cover Kubernetes in this chapter, but because of its strong ties to container orchestration, we should at least know the basics about Kubernetes.

Kubernetes (`https://kubernetes.io/`) was originally developed by Google, but the project is now managed by the Cloud Native Computing Foundation. It is an open-source container orchestration system that automatically deploys, scales, and manages containers. The project was well-received by the community right from the beginning since it had a proven track record of scale with Google's internal usage.

Kubernetes uses a master as the controlling unit that manages worker nodes to deploy containers. Each worker node can have one or more pods, which are the smallest units of units in Kubernetes. The pods are where the containers will be deployed. When the containers are deployed, they are generally grouped into different types of sets spread across the pods.

Most public cloud providers (AWS, Azure, Google, and DigitalOcean) offer managed Kubernetes clusters that users can try. The Kubernetes documentation (`https://kubernetes.io/docs/home/`) also offers many tutorials for step-by-step guides to learn more about the technology.

Summary

In this chapter, we learned about container virtualization. Containers are similar to virtual machines in their ability to isolate computing resources but different in the sense that they are lightweight and fast to deploy.

We saw how to use Docker containers to build Python applications and docker-compose to build multi-container applications on a single host.

Later in the chapter, we learned how networks are constructed with Docker containers by using the default bridge, custom bridges, and host options. Containers can also help with network operating system testing using the Containerlab project.

In the next chapter, we will look at how we can use Python in network security.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



6

Network Security with Python

In my opinion, network security is a tricky topic to write about. The reason is not a technical one but rather has to do with setting up the correct scope. The boundaries of network security are so wide that they touch all seven layers of the OSI model. From layer 1 of wiretapping to layer 4 of the transport protocol vulnerability, to layer 7 of man-in-the-middle spoofing, network security is everywhere. The issue is exacerbated by all the newly discovered vulnerabilities, which sometimes seem to be a daily occurrence. This does not even include the human social engineering aspect of network security.

As such, in this chapter, I would like to set the scope for what we will discuss. As we have been doing up to this point, we will primarily focus on using Python for network device security at OSI layers 3 and 4. We will look at Python tools that we can use to manage individual network devices for security purposes, as well as using Python as a glue to connect different components. Hopefully, we can treat network security holistically using Python in different OSI layers.

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

- The lab setup
- Python Scapy for security testing
- Access lists
- Forensic analysis with Syslog and **Uncomplicated Firewall** (**UFW**) using Python
- Other tools, such as a MAC address filter list, private VLAN, and Python IP table binding

Let's begin by looking at our lab setup for this chapter.

The Lab Setup

The devices being used in this chapter are a bit different from the previous chapters. In the previous chapters, we isolated a particular set of devices. For this chapter, we will use a few more Linux hosts in our lab to illustrate the function of the tools we will use. The connectivity and operating system information are important as they have ramifications regarding the security tools we will show later in this chapter. For example, if we want to apply an access list to protect the server, we need to know what the topology looks like and in which direction the client is making their connections. The Ubuntu host connections are a bit different than what we have seen so far, so please refer to this lab section when you see the example later if needed.

We will use the same Cisco CML tool with the NYC nodes with two additional Ubuntu hosts. The lab topology is provided with the course files.

The way to add a Linux host in CML is the same as adding network nodes, simply click on **add nodes** and pick Ubuntu as the selection. We will name the outside host connecting to `nyc-cor-r1` as the client, and the host behind `nyc-cor-edg-r1` as the server:

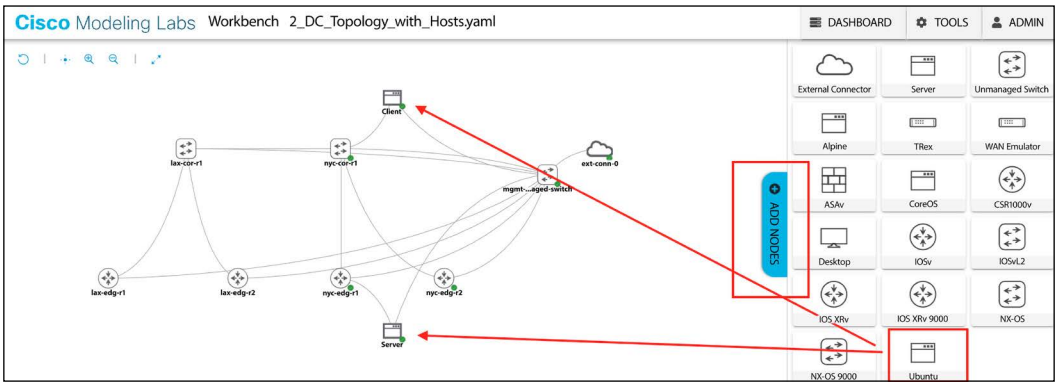


Figure 6.1: Adding Ubuntu Hosts

This is a good point to review and learn about Ubuntu Linux networking. We will spend some time listing out the Ubuntu Linux networking options of the setup. Here is an overview of the lab topology:

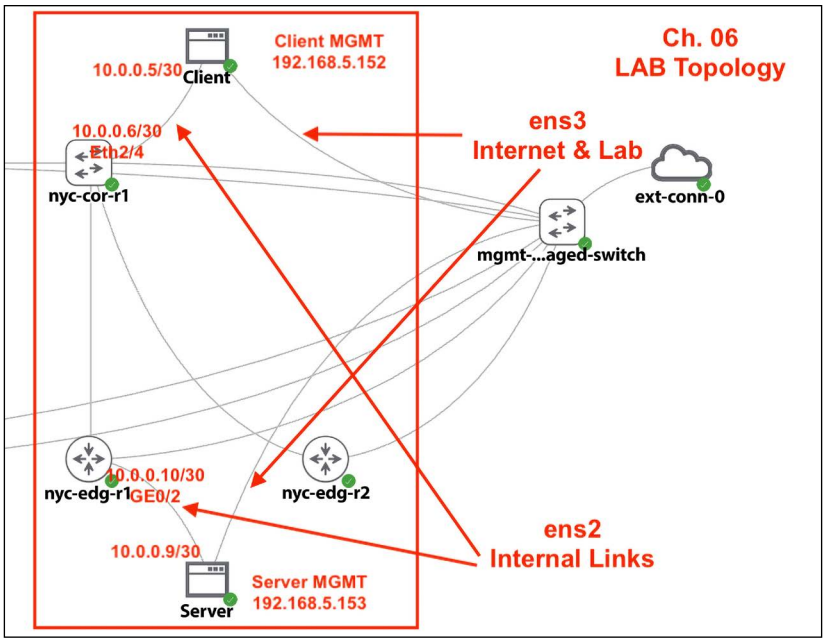


Figure 6.2: Lab Topology



The IP addresses listed will probably be different in your lab. They are listed here to easily reference in the remainder of the chapter code examples.

We will add two dual-home links to the hosts, one for the default gateway going to the unmanaged switch for management and internet. The other link is used to route the internet traffic. As illustrated, we will rename the host on the top as the client, and the bottom host as the server using the `hostname <name>` command. This is analogous to an internet client trying to access a corporate server within our network. The version of Ubuntu Linux in the CML software is 18.04 LTS:

```
ubuntu@client:~$ lsb_release -a
No LSB modules are available.
Distributor ID:    Ubuntu
Description:       Ubuntu 18.04.3 LTS
Release:    18.04
Codename:   bionic
```

To list and turn up the links, we can use the `ip link` and `ifconfig` commands:

```
ubuntu@client:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:1e:bc:51 brd ff:ff:ff:ff:ff:ff
3: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:19:54:b5 brd ff:ff:ff:ff:ff:ff
ubuntu@ubuntu:~$ sudo ifconfig ens3 up
```

For the hosts, when we first turn it up, it will have an initial network configuration under `/etc/netplan/50-cloud-init.yaml`. We will back it up and create our own:

```
ubuntu@ubuntu:/etc/netplan$ cd /etc/netplan/
ubuntu@ubuntu:/etc/netplan$ cp 50-cloud-init.yaml 50-cloud-init.yaml.bak
ubuntu@ubuntu:/etc/netplan$ sudo rm 50-cloud-init.yaml
ubuntu@ubuntu:/etc/netplan$ sudo touch 50-cloud-init.yaml
```

For the two network links, we will use the following configuration to configure the default gateway for `ens3` (management and internet) as well as the internal link:

```
ubuntu@client:~$ cat /etc/netplan/50-cloud-init.yaml
```

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      dhcp4: no
      dhcp6: no
      addresses: [192.168.2.152/24]
      gateway4: 192.168.2.1
      nameservers:
        addresses: [192.168.2.1,8.8.8.8]
```

```
    ens2:
      dhcp4: no
      dhcp6: no
      addresses: [10.0.0.5/30]
```

To allow the network change to take effect, we can use the `netplan apply` command:

```
ubuntu@ubuntu:/etc/netplan$ sudo netplan apply
```

Here is a quick output for the server side:

```
ubuntu@server:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:12:9c:5f brd ff:ff:ff:ff:ff:ff
3: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:0e:f7:ab brd ff:ff:ff:ff:ff:ff
ubuntu@server:~$ cat /etc/netplan/50-cloud-init.yaml
```

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      dhcp4: no
      dhcp6: no
      addresses: [192.168.2.153/24]
      gateway4: 192.168.2.1
      nameservers:
        addresses: [192.168.2.1,8.8.8.8]
    ens2:
      dhcp4: no
      dhcp6: no
      addresses: [10.0.0.9/30]
```

We will put the connected network into the existing OSPF network. Here is the configuration of `nyc-cor-r1`:

```
nyc-cor-r1# config t
Enter configuration commands, one per line.  End with CNTL/Z.
nyc-cor-r1(config)# int ethernet 2/4
nyc-cor-r1(config-if)# ip add 10.0.0.6/24
nyc-cor-r1(config-if)# ip router ospf 200 area 0.0.0.200
nyc-cor-r1(config-if)# no shut
nyc-cor-r1(config-if)# end
nyc-cor-r1# ping 10.0.0.5
PING 10.0.0.5 (10.0.0.5): 56 data bytes
36 bytes from 10.0.0.6: Destination Host Unreachable
Request 0 timed out
64 bytes from 10.0.0.5: icmp_seq=1 ttl=63 time=4.888 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=63 time=2.11 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=63 time=2.078 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=63 time=0.965 ms
^C
--- 10.0.0.5 ping statistics ---
5 packets transmitted, 4 packets received, 20.00% packet loss
round-trip min/avg/max = 0.965/2.51/4.888 ms
nyc-cor-r1#
```

The configuration of `nyc-cor-edg-r1` is as follows:

```
nyc-edg-r1#confi t
Enter configuration commands, one per line.  End with CNTL/Z.
nyc-edg-r1(config)#int gig 0/2
nyc-edg-r1(config-if)#ip add 10.0.0.10 255.255.255.252
nyc-edg-r1(config-if)#no shut
nyc-edg-r1(config-if)#end
nyc-edg-r1#ping 10.0.0.9
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 10.0.0.9, timeout is 2 seconds:
.!!!!
Success rate is 80 percent (4/5), round-trip min/avg/max = 1/3/7 ms
nyc-edg-r1#
```

```
nyc-edg-r1#confi t
Enter configuration commands, one per line.  End with CNTL/Z.
nyc-edg-r1(config)#router ospf 200
nyc-edg-r1(config-router)#net
nyc-edg-r1(config-router)#network 10.0.0.8 0.0.0.3 area 200
nyc-edg-r1(config-router)#end
nyc-edg-r1#
```

Here comes the part that might be a bit tricky for engineers who might be new to host-based networking. By default, the host has a routing preference as well. The default gateway we added for `ens3` will allow us to use the lab gateway for the "destination of last resort." We can see the routing table on the host via the `route` command:

```
ubuntu@client:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0
ens3
10.0.0.4        0.0.0.0         255.255.255.252 U     0      0        0
ens2
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0
ens3
```

We will use the following command to route traffic from the client to the server via the `route` command:

```
ubuntu@client:~$ sudo route add -net 10.0.0.8/30 gw 10.0.0.6
ubuntu@client:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0
ens3
10.0.0.4        0.0.0.0         255.255.255.252 U     0      0        0
ens2
10.0.0.8        10.0.0.6        255.255.255.252 UG    0      0        0
ens2
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0
ens3
```

We will do the same on the server side:

```
ubuntu@server:~$ sudo route add -net 10.0.0.4/30 gw 10.0.0.10
ubuntu@server:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0        0
ens3
10.0.0.4        10.0.0.10       255.255.255.252 UG    0      0        0
ens2
10.0.0.8        0.0.0.0         255.255.255.252 U     0      0        0
ens2
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0
ens3
```

To verify the client-to-server path, let's ping and trace the route to make sure that traffic between our hosts is going through the network devices instead of the default route:

```
# Install on both client and server
ubuntu@ubuntu:~$ sudo apt install traceroute

# From Server to Client
ubuntu@server:~$ ping -c 1 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=62 time=3.38 ms

--- 10.0.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.388/3.388/3.388/0.000 ms
ubuntu@server:~$ traceroute 10.0.0.5
traceroute to 10.0.0.5 (10.0.0.5), 30 hops max, 60 byte packets
 1  10.0.0.10 (10.0.0.10)  2.829 ms  5.832 ms  7.396 ms
 2  * * *
 3  10.0.0.5 (10.0.0.5)  11.458 ms  11.459 ms  11.744 ms

# From Client to Server
ubuntu@client:~$ ping -c 1 10.0.0.9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
```

```
64 bytes from 10.0.0.9: icmp_seq=1 ttl=62 time=3.32 ms

--- 10.0.0.9 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.329/3.329/3.329/0.000 ms
ubuntu@client:~$ traceroute 10.0.0.9
traceroute to 10.0.0.9 (10.0.0.9), 30 hops max, 60 byte packets
 1  10.0.0.6 (10.0.0.6)  3.187 ms  3.318 ms  3.804 ms
 2  * * *
 3  10.0.0.9 (10.0.0.9)  11.845 ms  12.030 ms  12.035 ms
```

The final task is to prepare the host for the remainder of the chapter with updated repositories:

```
$ sudo apt update && sudo apt upgrade -y
$ sudo apt install software-properties-common -y
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt install -y python3.10 python3.10-venv
$ python3.10 -m venv venv
$ source venv/bin/activate
```

Great! We have our lab; we are now ready to look at some security tools and measures using Python.

Python Scapy

Scapy (`https://scapy.net`) is a powerful Python-based interactive packet crafting program. Outside of some expensive commercial programs, very few tools can do what Scapy can do, to my knowledge. It is one of my favorite tools in Python.

The main advantage of Scapy is that it allows us to craft our packet from a very basic level. In the words of Scapy's creator:

> *"Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more..... with most other tools, you won't build something the author did not imagine. These tools have been built for a specific goal and can't deviate much from it."*

Let's now take a look at the tool.

Installing Scapy

Scapy has had an interesting path when it comes to Python 3 support. Back in 2015, there was an independent fork of Scapy from version 2.2.0 that aimed to support Python 3, named `Scapy3k`. In this book, we are using the main code base from the original Scapy project. If you have read the previous edition of the book and used a Scapy version that was only compatible with Python 2, please take a look at the Python 3 support per Scapy release:

Python versions support

Scapy version	Python 2.2-2.6	Python 2.7	Python 3.4-3.6	Python 3.7	Python 3.8	Python 3.9	Python 3.10-3.11
2.3.3	✅	✅	❌	❌	❌	❌	❌
2.4.0	❌	✅	✅	❌	❌	❌	❌
2.4.2	❌	✅	✅	✅	❌	❌	❌
2.4.3-2.4.4	❌	✅	✅	✅	✅	❌	❌
2.4.5	❌	✅	✅	✅	✅	✅	❌
2.5.0	❌	✅	✅	✅	✅	✅	✅

Python versions support

Scapy version	Python 2.2-2.6	Python 2.7	Python 3.4-3.6	Python 3.7	Python 3.8
2.2.X					
2.3.3					
2.4.0					
2.4.2					
2.4.3-2.4.4					

Figure 6.3: Python version support (source: https://scapy.net/download/)

We will install the official release from the source:

```
(venv) ubuntu@[server|client]:~$ git clone https://github.com/secdev/
scapy.git
(venv) ubuntu@[server|client]:~$ cd scapy/
(venv) ubuntu@[server|client]:~/scapy$ sudo python3 setup.py install
(venv) ubuntu@[server|client]:~/scapy$ pip install scapy
```

Following installation, we can launch the Scapy interactive shell by typing in **scapy** in the Command Prompt:

```
(venv) ubuntu@client:~$ sudo scapy
…

                     aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.5.0rc1.dev16
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | What is dead may never die!
       scccccp///pSP///p          p//Y   |                      -- Python 2
      sY/////////y  caa           S//P   |
       cayCyayP//Ya              pY/Ya
        sY/PsY////YCc           aC//Yp
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs
```

Here is a quick test to make sure we can access the `Scapy` library from Python 3:

```
(venv) ubuntu@client:~$ python3.10
Python 3.10.7 (main, Sep  7 2022, 15:23:21) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> exit()
```

Awesome! Scapy is now installed and can be executed from our Python interpreter. Let's take a look at its usage via the interactive shell in the next section.

Interactive examples

In our first example, we will craft an **Internet Control Message Protocol** (**ICMP**) packet on the client and send it to the server. On the server side, we will use `tcpdump` with a host filter to see the packet coming in:

```
## Client Side
ubuntu@client:~/scapy$ sudo scapy
>>> send(IP(dst="10.0.0.9")/ICMP())
.
Sent 1 packets.

# Server side
ubuntu@server:~/scapy$ sudo tcpdump -i ens2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens2, link-type EN10MB (Ethernet), capture size 262144 bytes
02:02:24.402707 Loopback, skipCount 0, Reply, receipt number 0, data (40
octets)
02:02:24.658511 IP 10.0.0.5 > server: ICMP echo request, id 0, seq 0,
length 8
02:02:24.658532 IP server > 10.0.0.5: ICMP echo reply, id 0, seq 0, length
8
```

As you can see, it is very simple to craft a packet from Scapy. Scapy allows you to build the packet layer by layer using the slash (`/`) as the separator. The `send` function operates at the layer 3 level, which takes care of routing and layer 2 for you. There is also a `sendp()` alternative that operates at layer 2, which means you will need to specify the interface and link layer protocol.

Let's look at capturing the returned packet by using the `send-request` (`sr`) function. We are using a special variation of `sr`, called `sr1`, which only returns one packet that answers from the packet sent:

```
>>> p = sr1(IP(dst="10.0.0.9")/ICMP())
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x0 len=28 id=5717 flags= frag=0 ttl=62
proto=icmp chksum=0x527f src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echo-reply
```

```
code=0 chksum=0xffff id=0x0 seq=0x0 |<Padding  load='\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

One thing to note is that the `sr()` function returns a tuple containing answered and unanswered lists:

```
>>> p = sr(IP(dst="10.0.0.9")/ICMP())
.Begin emission:
.....Finished sending 1 packets.
*
Received 7 packets, got 1 answers, remaining 0 packets
>>> type(p)
<class 'tuple'>
```

Now, let's take a look at what is contained inside the tuple:

```
>>> ans, unans = sr(IP(dst="10.0.0.9")/ICMP())
.Begin emission:
...Finished sending 1 packets.
..*
Received 7 packets, got 1 answers, remaining 0 packets
>>> type(ans)
<class 'scapy.plist.SndRcvList'>
>>> type(unans)
<class 'scapy.plist.PacketList'>
```

If we were to only take a look at the answered packet list, we could see that it is a `NamedTuple` containing the packet that we have sent as well as the returned packet:

```
>>> for i in ans:
...     print(type(i))
...
<class 'scapy.compat.NamedTuple.<locals>._NT'>
>>>
>>>
>>> for i in ans:
...     print(i)
...
QueryAnswer(query=<IP  frag=0 proto=icmp dst=10.0.0.9 |<ICMP  |>>,
```

```
answer=<IP  version=4 ihl=5 tos=0x0 len=28 id=10871 flags= frag=0 ttl=62
proto=icmp chksum=0x3e5d src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echo-reply
code=0 chksum=0xffff id=0x0 seq=0x0 |<Padding  load='\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>)
```

Scapy also provides a layer 7 construct, such as a DNS query. In the following example, we are querying an open DNS server for the resolution of `www.google.com`:

```
>>> p = sr1(IP(dst="8.8.8.8")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.google.
com")))
Begin emission:
Finished sending 1 packets.
......*
Received 7 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x20 len=76 id=20467 flags= frag=0 ttl=58
proto=udp chksum=0x5d3e src=8.8.8.8 dst=192.168.2.152 |<UDP  sport=domain
dport=domain len=56 chksum=0xf934 |<DNS  id=0 qr=1 opcode=QUERY aa=0
tc=0 rd=1 ra=1 z=0 ad=0 cd=0 rcode=ok qdcount=1 ancount=1 nscount=0
arcount=0 qd=<DNSQR  qname='www.google.com.' qtype=A qclass=IN |>
an=<DNSRR  rrname='www.google.com.' type=A rclass=IN ttl=115 rdlen=4
rdata=142.251.211.228 |> ns=None ar=None |>>>
>>>
```

Let's take a look at some other Scapy features. We'll begin by using Scapy for packet captures.

Packet captures with Scapy

As network engineers, we constantly have to capture packets on the wire during troubleshooting. We typically use Wireshark or similar tools, but Scapy can also be used to easily capture packets on the wire:

```
>>> a = sniff(filter="icmp", count=5)
>>> a.show()
0000 Ether / IP / ICMP 192.168.2.152 > 8.8.8.8 echo-request 0 / Raw
0001 Ether / IP / ICMP 8.8.8.8 > 192.168.2.152 echo-reply 0 / Raw
0002 Ether / IP / ICMP 192.168.2.152 > 8.8.8.8 echo-request 0 / Raw
0003 Ether / IP / ICMP 8.8.8.8 > 192.168.2.152 echo-reply 0 / Raw
0004 Ether / IP / ICMP 192.168.2.152 > 8.8.8.8 echo-request 0 / Raw
```

We can look at the packets in some more detail, including the raw format:

```
>>> for packet in a:
...     print(packet.show())
...
###[ Ethernet ]###
  dst       = 08:b4:b1:18:01:39
  src       = 52:54:00:19:54:b5
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 38166
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xd242
     src       = 192.168.2.152
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x6596
        id        = 0x502f
        seq       = 0x1
        unused    = ''
###[ Raw ]###
           load      = '\\xaa7%c\x00\x00\x00\x00\\xb2\\xcb\x01\x00\x00\
x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\
x1f !"#$%&\'()*+,-./01234567'
<skip>
```

We have seen the basic workings of Scapy. Let's now move on and see how we can use Scapy for certain aspects of common security testing.

The TCP port scan

The first step for any potential hackers is almost always trying to learn which service is open on the network to focus their efforts on the attack. Of course, we need to open certain ports to service our customers; that is part of the risk we must accept. However, we should close any other open port that needlessly exposes a larger attack surface. We can use Scapy to do a simple TCP open port scan to scan our own host.

We can send a `SYN` packet and see whether the server will return with `SYN-ACK` for various ports. Let's start with Telnet, TCP port 23:

```
>>> p = sr1(IP(dst="10.0.0.9")/TCP(sport=666,dport=23,flags="S"))
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>> p.show()
###[ IP ]###
  version= 4
  ihl= 5
  tos= 0x0
  len= 40
  id= 14089
  flags= DF
  frag= 0
  ttl= 62
  proto= tcp
  chksum= 0xf1b9
  src= 10.0.0.9
  dst= 10.0.0.5
  \options\
###[ TCP ]###
     sport= telnet
     dport= 666
     seq= 0
     ack= 1
     dataofs= 5
     reserved= 0
```

```
      flags= RA
      window= 0
      chksum= 0x9911
      urgptr= 0
      options= []
```

Note that, in the output here, the server is responding with a `RESET+ACK` for TCP port `23`. However, TCP port `22` (SSH) is open; therefore, a `SYN-ACK` is returned:

```
>>> p = sr1(IP(dst="10.0.0.9")/TCP(sport=666,dport=22,flags="S")).show()
###[ IP ]###
  version= 4
<skip>
  proto= tcp
  chksum= 0x28bf
  src= 10.0.0.9
  dst= 10.0.0.5
  \options\
###[ TCP ]###
     sport= ssh
     dport= 666
     seq= 1671401418
     ack= 1
     dataofs= 6
     reserved= 0
     flags= SA
<skip>
```

We can also scan a range of destination ports from `20` to `22`; note that we are using `sr()` for send-receive instead of the `sr1()` send-receive-one-packet variant:

```
>>> ans,unans = sr(IP(dst="10.0.0.9")/
TCP(sport=666,dport=(20,22),flags="S"))
>>> for i in ans:
...     print(i)
...
QueryAnswer(query=<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666
dport=ftp_data flags=S |>>, answer=<IP  version=4 ihl=5 tos=0x0 len=40
id=0 flags=DF frag=0 ttl=62 proto=tcp chksum=0x28c3 src=10.0.0.9
dst=10.0.0.5 |<TCP  sport=ftp_data dport=666 seq=0 ack=1 dataofs=5
```

```
reserved=0 flags=RA window=0 chksum=0x9914 urgptr=0 |<Padding  load='\x00\
x00\x00\x00\x00\x00' |>>>)
QueryAnswer(query=<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666
dport=ftp flags=S |>>, answer=<IP  version=4 ihl=5 tos=0x0 len=40 id=0
flags=DF frag=0 ttl=62 proto=tcp chksum=0x28c3 src=10.0.0.9 dst=10.0.0.5
|<TCP  sport=ftp dport=666 seq=0 ack=1 dataofs=5 reserved=0 flags=RA
window=0 chksum=0x9913 urgptr=0 |<Padding  load='\x00\x00\x00\x00\x00\x00'
|>>>)
QueryAnswer(query=<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666
dport=ssh flags=S |>>, answer=<IP  version=4 ihl=5 tos=0x0 len=44 id=0
flags=DF frag=0 ttl=62 proto=tcp chksum=0x28bf src=10.0.0.9 dst=10.0.0.5
|<TCP  sport=ssh dport=666 seq=4214084277 ack=1 dataofs=6 reserved=0
flags=SA window=29200 chksum=0x4164 urgptr=0 options=[('MSS', 1460)]
|<Padding  load='\x00\x00' |>>>)
```

We can also specify a destination network instead of a single host. As you can see from the `10.0.0.8/29` block, hosts `10.0.0.9`, `10.0.0.10`, and `10.0.0.14` returned with `SA`, which corresponds to the two network devices and the host:

```
>>> ans,unans = sr(IP(dst="10.0.0.8/29")/
TCP(sport=666,dport=(22),flags="S"))
>>> for i in ans:
...     print(i)
...
(<IP  frag=0 proto=tcp dst=10.0.0.14 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=7289 flags= frag=0 ttl=64
proto=tcp chksum=0x4a41 src=10.0.0.14 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=1652640556 ack=1 dataofs=6 reserved=0 flags=SA window=17292
chksum=0x9029 urgptr=0 options=[('MSS', 1444)] |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=62
proto=tcp chksum=0x28bf src=10.0.0.9 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=898054835 ack=1 dataofs=6 reserved=0 flags=SA window=29200
chksum=0x9f0d urgptr=0 options=[('MSS', 1460)] |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.10 |<TCP  sport=666 dport=ssh flags=S
|>>, <IP  version=4 ihl=5 tos=0x0 len=44 id=38021 flags= frag=0 ttl=254
proto=tcp chksum=0x1438 src=10.0.0.10 dst=10.0.0.5 |<TCP  sport=ssh
dport=666 seq=371720489 ack=1 dataofs=6 reserved=0 flags=SA window=4128
chksum=0x5d82 urgptr=0 options=[('MSS', 536)] |>>)
>>>
```

Based on what we have learned so far, we can make a simple script for reusability, `scapy_tcp_scan_1.py`:

```
#!/usr/bin/env python3
from scapy.all import *
import sys
def tcp_scan(destination, dport):
    ans, unans = sr(IP(dst=destination)/
TCP(sport=666,dport=dport,flags="S"))
    for sending, returned in ans:
        if 'SA' in str(returned[TCP].flags):
            return destination + " port " + str(sending[TCP].dport) + " is
open."
        else:
            return destination + " port " + str(sending[TCP].dport) + " is
not open."
def main():
    destination = sys.argv[1]
    port = int(sys.argv[2])
    scan_result = tcp_scan(destination, port)
    print(scan_result)
if __name__ == "__main__":
    main()
```

In the script, we start with the suggested importing of `scapy` and the `sys` module for taking in arguments. The `tcp_scan()` function is similar to what we have seen up to this point, the only difference being that we functionalized it so that we can acquire inputs from arguments and then call the `tcp_scan()` function in the `main()` function.

Remember that access to the low-level network requires root access; therefore, our script needs to be executed as `sudo`. Let's try the script on port `22` (SSH) and port `80` (HTTP):

```
ubunbu@client:~$ sudo python3 scapy_tcp_scan_1.py "10.0.0.14" 22
Begin emission:
......Finished sending 1 packets.
*
Received 7 packets, got 1 answers, remaining 0 packets
10.0.0.14 port 22 is open.
ubuntu@client:~$ sudo python3 scapy_tcp_scan_1.py "10.0.0.14" 80
```

```
Begin emission:
...Finished sending 1 packets.
*
Received 4 packets, got 1 answers, remaining 0 packets
10.0.0.14 port 80 is not open.
```

This was a relatively lengthy example of the TCP scan script, which demonstrated the power of crafting your packet with Scapy. We tested the steps in the interactive shell and finalized the usage with a simple script. Now, let's look at some more examples of Scapy's usage for security testing.

The ping collection

Let's say our network contains a mix of Windows, Unix, and Linux machines, with network users adding their machines from the **Bring Your Own Device** (**BYOD**) policy; they may or may not support an ICMP ping. We can now construct a file with three types of common pings for our network – the ICMP, TCP, and UDP pings – in `scapy_ping_collection.py`:

```
#!/usr/bin/env python3
from scapy.all import *
def icmp_ping(destination):
    # regular ICMP ping
    ans, unans = sr(IP(dst=destination)/ICMP())
    return ans
def tcp_ping(destination, dport):
    ans, unans = sr(IP(dst=destination)/TCP(dport=dport,flags="S"))
    return ans
def udp_ping(destination):
    ans, unans = sr(IP(dst=destination)/UDP(dport=0))
    return ans
def answer_summary(ans):
    for send, recv in ans:
        print(recv.sprintf("%IP.src% is alive"))
```

We can then execute all three types of pings on the network in one script:

```
def main():
    print("** ICMP Ping **")
    ans = icmp_ping("10.0.0.13-14")
    answer_summary(ans)
    print("** TCP Ping ***")
```

```
    ans = tcp_ping("10.0.0.13", 22)
    answer_summary(ans)
    print("** UDP Ping ***")
    ans = udp_ping("10.0.0.13-14")
    answer_summary(ans)
if __name__ == "__main__":
    main()
```

At this point, hopefully, you will agree with me that by having the ability to construct your custom packet, you can be in charge of the type of operations and tests that you would like to run. Along the same thought of constructing our custom packets using Scapy, we can construct our packets to perform security tests on our network.

Common attacks

In this example, let's look at how we can construct our packet to conduct some classic attacks, such as *Ping of Death* (`https://en.wikipedia.org/wiki/Ping_of_death`) and *Land Attack* (`https://en.wikipedia.org/wiki/Denial-of-service_attack`). These are network penetration tests that you previously had to pay for with similar commercial software. With Scapy, you can conduct the test while maintaining full control and adding more tests in the future.

The first attack sends the destination host with a bogus IP header, such as an IP header length of two and an IP version of three:

```
def malformed_packet_attack(host):
       send(IP(dst=host, ihl=2, version=3)/ICMP())
```

The `ping_of_death_attack` consists of the regular ICMP packet with a payload bigger than 65,535 bytes:

```
def ping_of_death_attack(host):
    # https://en.wikipedia.org/wiki/Ping_of_death
    send(fragment(IP(dst=host)/ICMP()/("X"*60000)))
```

The `land_attack` wants to redirect the client response back to the client and exhausts the host's resources:

```
def land_attack(host):
    # https://en.wikipedia.org/wiki/Denial-of-service_attack
    send(IP(src=host, dst=host)/TCP(sport=135,dport=135))
```

These are pretty old vulnerabilities or classic attacks that a modern operating system is no longer susceptible to. For our Ubuntu 20.04 host, none of the preceding attacks will bring it down. However, as more security issues are being discovered, Scapy is a great tool for initiating tests against our network and host without waiting for the impacted vendor to give you a validation tool. This is especially true for the zero-day (published without prior notification) attacks that seem more and more common on the internet. Scapy is a tool that can do a lot more than what we can cover in this chapter, but luckily, there are lots of open source resources on Scapy that we can reference.

Scapy resources

We have spent much effort working with Scapy in this chapter. This is partially due to the high regard in which I hold the tool. I hope you agree that Scapy is a great tool to keep in our toolset as network engineers. The best part about Scapy is that it is constantly being developed with an engaged community of users.

I would highly recommend at least going through the Scapy tutorial at `http://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial`, as well as any of the documentation that is of interest to you.

Of course, network security is more than just crafting packets and testing vulnerabilities. In the next section, we'll take a look at automating the access list that is commonly used to protect sensitive internal resources.

Access lists

The network access lists are usually the first line of defense against outside intrusions and attacks. Generally speaking, routers and switches process packets at a much faster rate than servers by utilizing high-speed memory hardware such as **ternary content-addressable memory** (**TCAM**). They do not need to see the application layer information. Instead, they just examine the layer 3 and layer 4 headers and decide whether the packets can be forwarded. Therefore, we generally utilize network device access lists as a first step in safeguarding our network resources.

As a rule of thumb, we want to place access lists as close to the source (client) as possible. Inherently, we also trust the inside host and distrust clients beyond our network boundary. The access list is, therefore, usually placed in the inbound direction on the external-facing network interface(s). In our lab scenario, this means we will place an inbound access list at Ethernet2/2 on `nyc-cor-r1`, which is directly connected to the client host.

If you are unsure of the direction and placement of the access list, a few points might help here:

- Think of the access list from the perspective of the network device.
- Simplify the packets in terms of just source and destination IPs and use one host as an example.
- In our lab, traffic from our server to the client will have a source IP of `10.0.0.9`, with a destination IP of `10.0.0.5`.
- The traffic from the client to the server will have a source IP of `10.0.0.5`, and a destination IP of `10.0.0.9`.

Obviously, every network is different, and how the access list should be constructed depends on the services provided by your server. But, as an inbound border access list, you should do the following:

- Deny RFC 3030 special-use address sources, such as `127.0.0.0/8`.
- Deny RFC 1918 space, such as `10.0.0.0/8`.
- Deny our own space as the source IP; in this case, `10.0.0.4/30`.
- Permit inbound TCP ports `22` (SSH) and `80` (HTTP) to host `10.0.0.9`.
- Deny everything else.

Here is a good list of bogon networks to block: `https://ipinfo.io/bogon`.

Knowing what to add is only half of the step. In the next section, let's take a look at how to implement the intended access list with Ansible.

Implementing access lists with Ansible

The easiest way to implement this access list would be to use Ansible. We have already looked at Ansible in previous chapters, but it is worth repeating the advantages of using Ansible in this scenario:

- **Easier management**: For a long access list, we can utilize the `include` statement to break the access list into more manageable pieces. The smaller pieces can then be managed by other teams or service owners.
- **Idempotency**: We can schedule the playbook at regular intervals and only the necessary changes will be made.
- **Each task is explicit**: We can separate the construct of the entries as well as apply the access list to the proper interface.

- **Reusability**: In the future, if we add additional external-facing interfaces, we just need to add the device to the list of devices for the access list.
- **Extensible**: You will notice that we can use the same playbook for constructing the access list and apply it to the right interface. We can start small and expand to separate playbooks in the future as needed.

The `host` file is pretty standard. We will also follow our standards of putting the variables in the `host_vars` folder:

```
[nxosv-devices]
nyc-cor-r1

[iosv-devices]
nyc-edg-r1
```

```
$ cat host_vars/nyc-cor-r1
---
ansible_host: 192.168.2.60
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: nxos
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

We will declare the variables in the playbook:

```
---
- name: Configure Access List
  hosts: "nxosv-devices"
  gather_facts: false
  connection: local
  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ ansible_username }}"
      password: "{{ ansible_password }}"
```

To save space, we will only illustrate denying the RFC 1918 space. Implementing the denial of RFC 3030 and our own space will be identical to the steps used for the RFC 1918 space. Note that we did not deny `10.0.0.0/8` in our playbook because our configuration currently uses the `10.0.0.0` network for addressing. Of course, we could perform the single host permit first and deny `10.0.0.0/8` in a later entry, but in this example, we just chose to omit it:

```
  tasks:
    - nxos_acl:
        name: border_inbound
        seq: 20
        action: deny
        proto: tcp
        src: 172.16.0.0/12
        dest: any
        log: enable
        state: present
    - nxos_acl:
        name: border_inbound
        seq: 30
        action: deny
        proto: tcp
        src: 192.168.0.0/16
        dest: any
        state: present
        log: enable
<skip>
```

Note that we are allowing the established connection sourcing from the server inside to be allowed back in. We use the final explicit `deny ip any` statement as a high-sequence number (`1000`) so that we can insert any new entries later on.

We can then apply the access list to the correct interface:

```
    - name: apply ingress acl to Ethernet 2/4
      nxos_acl_interface:
        name: border_inbound
        interface: Ethernet2/4
        direction: ingress
        state: present
```

This may seem like a lot of work for a single access list. For an experienced engineer, using Ansible to do this task will take longer than just logging in to the device and configuring the access list. However, remember that this playbook can be reused many times in the future, so it will save you time in the long run.

It is my experience that often, for a long access list, a few entries will be for one service, a few entries will be for another, and so on. The access lists tend to grow organically over time, and it becomes very hard to keep track of the origin and purpose of each entry. The fact that we can break them apart makes the management of a long access list much simpler.

Now, let's execute the playbook and `verify` on `nx-osv-1`:

```
$ ansible-playbook -i hosts access_list_nxosv.yml

PLAY [Configure Access List] ***************************************
*******************************

TASK [nxos_acl] ****************************************************
*******************************
ok: [nyc-cor-r1]

<skip>

TASK [nxos_acl] ****************************************************
*******************************
ok: [nyc-cor-r1]

TASK [apply ingress acl to Ethernet 2/4] ***************************
*******************************
changed: [nyc-cor-r1]

PLAY RECAP *********************************************************
*******************************
nyc-cor-r1                 : ok=7    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
<skip>
```

We should log in to `nyc-cor-r1` to verify the changes:

```
nyc-cor-r1# sh ip access-lists border_inbound

IP access list border_inbound
        20 deny tcp 172.16.0.0/12 any log
        30 deny tcp 192.168.0.0/16 any log
        40 permit tcp any 10.0.0.9/32 eq 22 log
        50 permit tcp any 10.0.0.9/32 eq www log
        60 permit tcp any any established log
        1000 deny ip any any log

nx-osv-1# sh run int eth 2/4
!
interface Ethernet2/1
  description to Client
  no switchport
  mac-address fa16.3e00.0001
  ip access-group border_inbound in
  ip address 10.0.0.6/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
```

We have seen the implementation of IP access lists that check layer 3 information on the network. In the next section, let's look at how to restrict device access in a layer 2 environment.

MAC access lists

In the case where you have a layer 2 environment, or where you are using non-IP protocols on Ethernet interfaces, you can still use a MAC address access list to allow or deny hosts based on MAC addresses. The steps are similar to the IP access list, but the match will be based on MAC addresses. Recall that for MAC addresses, or physical addresses, the first six hexadecimal symbols belong to an **organizationally unique identifier** (**OUI**). So, we can use the same access list matching pattern to deny a certain group of hosts.

We are testing this on IOSv with the `ios_config` module. For older Ansible versions, the change will be pushed out every single time the playbook is executed. For newer Ansible versions, the control node will check for changes first and only make changes when needed.

The host file and the top portion of the playbook are similar to the IP access list; the `tasks` portion is where the different modules and arguments are used:

```
<skip>
 tasks:
    - name: Deny Hosts with vendor id fa16.3e00.0000
      ios_config:
        lines:
          - access-list 700 deny fa16.3e00.0000 0000.00FF.FFFF
          - access-list 700 permit 0000.0000.0000 FFFF.FFFF.FFFF
    - name: Apply filter on bridge group 1
      ios_config:
        lines:
          - bridge-group 1
          - bridge-group 1 input-address-list 700
        parents
          - interface GigabitEthernet0/1
```

We can execute the playbook and verify its application on `iosv-1`:

```
$ ansible-playbook -i hosts access_list_mac_iosv.yml
TASK [Deny Hosts with vendor id fa16.3e00.0000] *************************
************************************************
changed: [nyc-edg-r1]
TASK [Apply filter on bridge group 1] ***********************************
************************************************
changed: [nyc-edg-r1]
```

As we have done before, let's log in to the device to verify our change:

```
nyc-edg-r1#sh run int gig 0/1
!
interface GigabitEthernet0/1
 description to nyc-cor-r1
 <skip>
 bridge-group 1
 bridge-group 1 input-address-list 700
end
```

As more virtual networks become popular, layer 3 information sometimes becomes transparent to the underlying virtual links. In these scenarios, the MAC access list becomes a good option if you need to restrict access to those links. In this section, we have used Ansible to automate the implementation of both layer 2 and layer 3 access lists. Now, let's change gears a bit but stay within the security context and look at how to pick up necessary security information from `syslogs` using Python.

The Syslog search

There are plenty of documented network security breaches that took place over an extended period. In these slow breaches, we quite often saw signs and traces in logs indicating that there were suspicious activities. These can be found in both server and network device logs. The activities were not detected, not because there was a lack of information, but rather because there was **too much** information. The critical information we were looking for is usually buried deep in a mountain of information that is hard to sort out.

Besides Syslog, UFW is another great source of log information for servers. It is a frontend to IP tables, which is a server firewall. UFW makes managing firewall rules very simple and logs a good amount of information. Refer to the *Other Tools* section for more information on UFW.

In this section, we will try to use Python to search through the Syslog text in order to detect the activities that we were looking for. Of course, the exact terms that we will search for depend on the device we are using. For example, Cisco provides a list of messages to look for in Syslog for any access-list violation logging. It is available at `http://www.cisco.com/c/en/us/about/security-center/identify-incidents-via-syslog.html`.

For more understanding of access control list logging, go to `http://www.cisco.com/c/en/us/about/security-center/access-control-list-logging.html`.

For our exercise, we will use a Nexus switch anonymized Syslog file containing about 65,000 lines of log messages. This file is included for you in the book's GitHub repository:

```
$ wc -l sample_log_anonymized.log
65102 sample_log_anonymized.log
```

We have inserted some Syslog messages from the Cisco documentation (`http://www.cisco.com/c/en/us/support/docs/switches/nexus-7000-series-switches/118907-configure-nx7k-00.html`) as the log message that we should be looking for:

```
2014 Jun 29 19:20:57 Nexus-7000 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured
from vty by admin on console0
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf: Ethernet4/1, Pro
tocol: "ICMP"(1), Hit-count = 2589
2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1, Dst IP: 172.16.10.10, Src Port: 0, Dst Port: 0, Src Intf:
Ethernet4/1, Pro tocol: "ICMP"(1), Hit-count = 4561
```

We will be using simple examples with regular expressions. If you are already familiar with the regular expression module in Python, feel free to skip the rest of the section.

Searching with the regular expression module

For our first search, we will simply use the regular expression module to search for the terms we are looking for. We will use a simple loop to do the following:

```
#!/usr/bin/env python3
import re, datetime
startTime = datetime.datetime.now()
with open('sample_log_anonymized.log', 'r') as f:
    for line in f.readlines():
        if re.search('ACLLOG-5-ACLLOG_FLOW_INTERVAL', line):
            print(line)
endTime = datetime.datetime.now()
elapsedTime = endTime - startTime
print("Time Elapsed: " + str(elapsedTime))
```

It took about four-hundredths of a second to search through the log file:

```
$ python3 python_re_search_1.py
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
Time Elapsed: 0:00:00.047249
```

It is recommended to compile the search term for a more efficient search. It will not impact us much since the script is already pretty fast. Python's interpretative nature might make it slower. However, it will make a difference when we search through a larger text body, so let's make the change:

```
searchTerm = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')
with open('sample_log_anonymized.log', 'r') as f:
for line in f.readlines():
if re.search(searchTerm, line):
    print(line)
```

The timing result is actually slower:

```
Time Elapsed: 0:00:00.081541
```

Let's expand the example a bit. Assuming we have several files and multiple terms to search through, we will copy the original file to a new file:

```
$ cp sample_log_anonymized.log sample_log_anonymized_1.log
```

We will also include searching for the `PAM: Authentication failure` term. We will add another loop to search both files:

```
term1 = re.compile('ACLLOG-5-ACLLOG_FLOW_INTERVAL')
term2 = re.compile('PAM: Authentication failure')
fileList = ['sample_log_anonymized.log', 'sample_log_anonymized_1.log']
for log in fileList:
    with open(log, 'r') as f:
        for line in f.readlines():
            if re.search(term1, line) or re.search(term2, line):
                print(line)
```

We can now see the difference in performance by expanding our search terms and the number of messages:

```
$ python3 python_re_search_2.py
2016 Jun 5 16:49:33 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM:
Authentication failure for illegal user AAA from 172.16.20.170 -
sshd[4425]
2016 Sep 14 22:52:26.210 NEXUS-A %DAEMON-3-SYSTEM_MSG: error: PAM:
Authentication failure for illegal user AAA from 172.16.20.170 -
sshd[2811]
```

```
<skip>
2014 Jun 29 19:21:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
2014 Jun 29 19:26:18 Nexus-7000 %ACLLOG-5-ACLLOG_FLOW_INTERVAL: Src IP:
10.1 0.10.1,
<skip>
Time Elapsed: 0:00:00.330697
```

Of course, when it comes to performance tuning, it is a never-ending, impossible race to zero, and the performance sometimes depends on the hardware you are using. But the important point is to regularly perform audits of your log files using Python so that you can catch the early signals of any potential breach.

We have looked at some of the key ways in which we can enhance our network security in Python, but there are several other powerful tools that can make this process easier and more effective. In the final section of this chapter, we will explore some of these tools.

Other tools

There are other network security tools that we can use and automate with Python. Let's take a look at two of the most commonly used ones.

Private VLANs

Virtual local area networks (VLANs) have been around for a long time. They are essentially broadcast domains where all hosts can be connected to a single switch but are partitioned out to different domains, so we can separate the hosts according to which hosts can see others via broadcasts. Let's consider a map based on IP subnets. For example, in an enterprise building, I would likely see one IP subnet per physical floor: `192.168.1.0/24` for the first floor, `192.168.2.0/24` for the second floor, and so on. In this pattern, we use a `/24` block for each floor. This gives a clear delineation of my physical network as well as my logical network. A host wanting to communicate beyond its subnet will need to traverse through its layer 3 gateway, where I can use an access list to enforce security.

What happens when different departments reside on the same floor? Perhaps the finance and sales teams are on the second floor, and I would not want the sales team's hosts in the same broadcast domain as the finance team. I can break the subnet down further, but that might become tedious and break the standard subnet scheme previously set up. This is where a private VLAN can help.

The private VLAN essentially breaks up the existing VLAN into sub-VLANs. There are three categories within a private VLAN:

- **The Promiscuous (P) port**: This port is allowed to send and receive layer 2 frames from any other port on the VLAN; this usually belongs to the port connecting to the layer 3 routers.
- **The Isolated (I) port**: This port is only allowed to communicate with P ports, and it is typically connected to hosts when you do not want it to communicate with other hosts in the same VLAN.
- **The Community (C) port**: This port is allowed to communicate with other C ports in the same community and with P ports.

We can again use Ansible or any other Python scripts introduced so far to accomplish this task. By now, we should have enough practice and confidence to implement this feature via automation, so I will not repeat the steps here. Being aware of the private VLAN feature will come in handy when you need to isolate ports even further in a layer 2 VLAN.

UFW with Python

We briefly mentioned UFW as the front end for IP tables on Ubuntu hosts. Here is a quick overview:

```
$ sudo apt-get install ufw
$ sudo ufw status
$ sudo ufw default outgoing
$ sudo ufw allow 22/tcp
$ sudo ufw allow www
$ sudo ufw default deny incoming
We can see the status of UFW:
$ sudo ufw status verbose Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed) New
profiles: skip
To Action From
-- ------ ----
22/tcp ALLOW IN Anywhere
80/tcp ALLOW IN Anywhere
22/tcp (v6) ALLOW IN Anywhere (v6)
80/tcp (v6) ALLOW IN Anywhere (v6)
```

As you can see, the advantage of UFW is that it provides a simple interface to construct otherwise complicated IP table rules. There are several Python-related tools we can use with UFW to make things even simpler:

- We can use the Ansible UFW module to streamline our operations. More information is available at `https://docs.ansible.com/ansible/latest/collections/community/general/ufw_module.html`.
- There are Python wrapper modules around UFW as an API (visit `https://gitlab.com/dhj/easyufw`). This can make integration easier if you need to dynamically modify UFW rules based on certain events.
- UFW itself is written in Python. Therefore, you can use existing Python knowledge if we need to extend the current command sets. More information is available at `https://launchpad.net/ufw`.

UFW proves to be a good tool to safeguard your network server.

Further reading

Python is a very common language used in many security-related fields. A few of the books I would recommend are listed as follows:

- **Violent Python**: A cookbook for hackers, forensic analysts, penetration testers, and security engineers, by T.J. O'Connor (ISBN-10: 1597499579)
- **Black Hat Python**: Python programming for hackers and pen-testers, by Justin Seitz (ISBN-10: 1593275900)

I have personally used Python extensively in my research work on **Distributed Denial of Service (DDoS)** at A10 Networks. If you are interested in learning more, the guide can be downloaded for free at `https://www.a10networks.com/resources/ebooks/distributed-denial-service-ddos/`.

Summary

In this chapter, we looked at network security with Python. We used the Cisco CML tool to set up our lab with hosts and network devices, consisting of NX-OSv and IOSv types. We took a tour around Scapy, which allows us to construct packets from the ground up.

Scapy can be used in interactive mode for quick testing. Once testing is completed in interactive mode, we can put the steps into a file for more scalable testing. It can be used to perform various network penetration testing for known vulnerabilities.

We also looked at how we can use both an IP access list and a MAC access list to protect our network. They are usually the first line of defense in our network protection. Using Ansible, we can deploy access lists consistently and quickly to multiple devices.

Syslog and other log files contain useful information that we should regularly comb through to detect any early signs of a breach. Using Python regular expressions, we can systematically search for known log entries that can point us to security events that require our attention. Besides the tools we have discussed, private VLAN and UFW are among some other useful tools that we can use for more security protection.

In *Chapter 7*, *Network Monitoring with Python – Part 1*, we will look at how to use Python for network monitoring. Monitoring allows us to know what is happening in our network, as well as the state of the network.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



7

Network Monitoring with Python – Part 1

Imagine you get a call from your company's network operations center at 2:00 a.m. The person on the other end says: "Hi, we are facing a difficult issue that is impacting production services. We suspect it might be network related. Can you check for us?" For this type of urgent, open-ended question, what would be the first thing you do? Most of the time, the thing that comes to mind would be: What changed in the time between when the network was working and when something went wrong? We would check our monitoring tool and see if any key metrics have changed in the last few hours. Better yet, we may have received monitoring alerts for any metrics that deviated from the normal baseline numbers.

Throughout this book, we have been discussing various ways to systematically make predictable changes to our network, intending to keep the network running as smoothly as possible. However, networks are not static – far from it. They are probably one of the most fluid parts of the entire infrastructure. By definition, a network connects different parts of the infrastructure, constantly passing traffic back and forth.

There are lots of moving parts that can cause our network to stop working as expected: hardware failures, software with bugs, human mistakes despite their best intentions, and many more. It is not a question of whether things will go wrong but of when and what will go wrong when it happens. We need ways to monitor our network to make sure it works as expected and hopefully be notified when it does not.

In the upcoming two chapters, we will look at various ways to perform network monitoring tasks. Many tools we have looked at thus far can be tied together or directly managed by Python. Like many tools we have looked at, network monitoring has two parts.

First, we need to know what monitoring-related information the equipment can transmit. Second, we need to identify what useful, actionable information we can interpret from the data.

In this chapter, we will begin by looking at a few tools that allow us to monitor the network effectively:

- The lab setup
- The **Simple Network Management Protocol (SNMP)** and related Python libraries to work with SNMP
- Python visualization libraries:
    - Matplotlib and examples
    - Pygal and examples
- Python integration with MRTG and Cacti for network visualization

This list is not exhaustive, and there is certainly no lack of commercial vendors in the network monitoring space. The basics of network monitoring that we will look at, however, carry well for both open source and commercial tools.

Lab Setup

The lab for this chapter consists of IOSv devices to simplify device configuration. We will use the same lab for this and the next chapters. The topology is as follows:

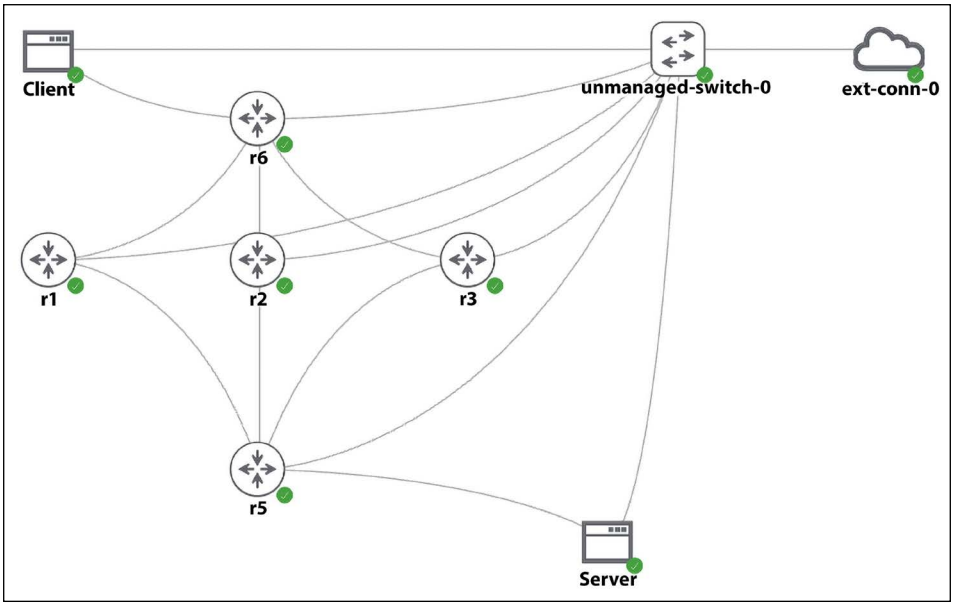


Figure 7.1: Lab topology

The device configuration is as follows:

Device	Management IP	Loopback IP
r1	192.168.2.218	192.168.0.1
r2	192.168.2.219	192.168.0.2
r3	192.168.2.220	192.168.0.3
r5	192.168.2.221	192.168.0.4
r6	192.168.2.222	192.168.0.5

The Ubuntu hosts information is as follows:

Device Name	External Link Eth0	Internal IP Eth1
Client	192.168.2.211	10.0.0.9
Server	192.168.2.212	10.0.0.5

The Linux hosts are `tinycore-linux` (`http://tinycorelinux.net/`) migrated from previous versions of VIRL. The default username and password are both `cisco`. If we need to change the interface IP and default gateways, they can be done via the following commands:

```
cisco@Client:~$ sudo ifconfig eth0 192.168.2.211 netmask 255.255.255.0
cisco@Client:~$ sudo route add default gw 192.168.2.1
cisco@Server:~$ sudo ifconfig eth0 192.168.2.212 netmask 255.255.255.0
cisco@Server:~$ sudo route add default gw 192.168.2.1
```

The two Ubuntu hosts will be used to generate traffic across the network so that we can look at some non-zero counters. The lab file is included in the book's GitHub repository.

SNMP

SNMP is a standardized protocol used to collect and manage devices. Although the standard allows you to use SNMP for device management, in my experience, most network administrators prefer to keep SNMP as an information collection mechanism only. Since SNMP operates on UDP, which is connectionless, and considering the relatively weak security mechanism in versions 1 and 2, making device changes via SNMP tends to make network operators uneasy. SNMP version 3 has added cryptographic security and new concepts and terminology to the protocol, but how SNMP version 3 is adapted varies among network device vendors.

SNMP is widely used in network monitoring and has been around since 1988 as part of RFC 1065. The operations are straightforward, with the network manager sending `GET` and `SET` requests toward the device and the device with the SNMP agent responding with the information per request. The most widely adopted standard is SNMPv2c, defined in RFC 1901 – RFC 1908. It uses a simple community-based security scheme for security. It has also introduced new features, such as the ability to get bulk information. The following diagram displays the high-level operation for SNMP:

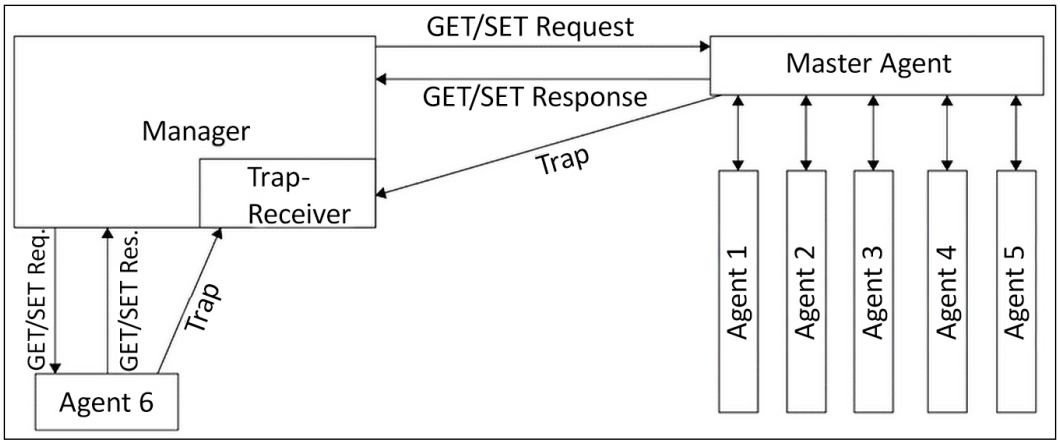


Figure 7.2: SNMP operations

The information residing in the device is structured in the **management information base (MIB)**. The MIB uses a hierarchical namespace containing an **object identifier (OID)**, which represents the information that can be read and fed back to the requester. When we talk about using SNMP to query device information, we are talking about using the management station to query the specific OID that represents the information we are after. A common OID structure, such as a systems and interfaces OID, is shared among vendors. Besides common OID, each vendor can also supply an enterprise-level OID specific to them.

As an operator, we must put some effort into consolidating information into an OID structure in our environment to retrieve useful information. This can sometimes be a tedious process of finding one OID at a time. For example, you might be requesting a device OID and receive a value of 10,000. What is that value? Is that interface traffic? Is it in bytes or bits? Or maybe it represents the number of packets? How do we know? We will need to consult either the standard or the vendor documentation to find out. There are tools that help with this process, such as a MIB browser that can provide more metadata to the value. But, at least in my experience, constructing an SNMP-based monitoring tool for your network can sometimes feel like a cat-and-mouse game of trying to find that one missing value.

Some of the main points to take away from the operation are as follows:

- The implementation relies heavily on the amount of information the device agent can provide. This, in turn, relies on how the vendor treats SNMP: as a core feature or an added feature.
- SNMP agents generally require CPU cycles from the control plane to return a value. Not only is this inefficient for devices with, say, large BGP tables, it is also not feasible to use SNMP to query the data at small intervals.
- The user needs to know the OID to query the data.

Since SNMP has been around for a while, I assume you have some experience with it already. Let's jump directly into package installation and our first SNMP example.

Setup

First, let's ensure we have the SNMP managing device and agent working in our setup. The SNMP bundle can be installed on either the hosts (client or server) in our lab or the managing device on the management network. As long as the SNMP manager has IP reachability to the device and the managed device allows the inbound connection, SNMP should work. In production, you should only install the software on the management host and only allow SNMP traffic in the control plane.

In this lab, we have installed SNMP on both the Ubuntu host on the management network and the client host in the lab:

```
$ sudo apt update && sudo apt upgrade
$ sudo apt-get install snmp
```

The next step is to turn on and configure the SNMP options on the network devices. There are many optional parameters you can configure on the network device, such as contact, location, chassis ID, and SNMP packet size. The SNMP configuration options are device-specific, and you should check the documentation for the particular device. For IOSv devices, we will configure an access list to limit only the desired host for querying the device and tying the access list with the SNMP community string. In our example, we will use the word `secret` as the read-only community string and `permit_snmp` as the access list name:

```
!
ip access-list standard permit_snmp
 permit <management station> log
 deny   any log
!
snmp-server community secret RO permit_snmp
!
```

The SNMP community string is acting as a shared password between the manager and the agent; therefore, it needs to be included any time you want to query the device.

As mentioned earlier in this chapter, finding the right OID is oftentimes half of the battle when working with SNMP. We can use tools such as Cisco SNMP Object Navigator (`https://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseOID.do?local=en`) to find specific OIDs to query.

Alternatively, we can just start walking through the SNMP tree, starting from the top of Cisco's enterprise tree at `.1.3.6.1.4.1.9`. We will perform the walk to make sure that the SNMP agent and the access list are working:

```
$ snmpwalk -v2c -c secret 192.168.2.218 .1.3.6.1.4.1.9
iso.3.6.1.4.1.9.2.1.1.0 = STRING: "
Bootstrap program is IOSv
"
iso.3.6.1.4.1.9.2.1.2.0 = STRING: "reload"
iso.3.6.1.4.1.9.2.1.3.0 = STRING: "iosv-1"
iso.3.6.1.4.1.9.2.1.4.0 = STRING: "virl.info"
<skip>
```

We can be more specific about the OID we need to query as well:

```
$ snmpwalk -v2c -c secret 192.168.2.218 .1.3.6.1.4.1.9.2.1.61.0
iso.3.6.1.4.1.9.2.1.61.0 = STRING: "cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web http://www.cisco.com"
```

As a matter of demonstration, what if we type in the wrong value by `1` digit, from `0` to `1` at the end of the last OID? This is what we would see:

```
$ snmpwalk -v2c -c secret 192.168.2.218 .1.3.6.1.4.1.9.2.1.61.1
iso.3.6.1.4.1.9.2.1.61.1 = No Such Instance currently exists at this OID
```

Unlike API calls, there are no useful error codes or messages; it simply states that the OID does not exist. This can be pretty frustrating at times.

The last thing to check would be that the access list we configured will deny unwanted SNMP queries. Because we had the log keyword for both the `permit` and `deny` entries in the access list, only `172.16.1.123` is permitted to query the devices:

```
*Sep 17 23:32:10.155: %SEC-6-IPACCESSLOGNP: list permit_snmp permitted 0
192.168.2.126 -> 0.0.0.0, 1 packet
```

As you can see, the biggest challenge in setting up SNMP is finding the right OID. Some OIDs are defined in standardized MIB-2; others are under the enterprise portion of the tree. Vendor documentation is the best bet, though. Some tools can help, such as a MIB browser; you can add MIBs (again, provided by the vendors) to the browser and see the description of the enterprise-based OIDs. A tool such as Cisco's SNMP Object Navigator (`http://snmp.cloudapps.cisco.com/Support/SNMP/do/BrowseOID.do?local=en`) proves to be very valuable when you need to find the correct OID of the object you are looking for.

PySNMP

PySNMP is a cross-platform, pure Python SNMP engine implementation developed by Ilya Etingof (`https://github.com/etingof`). It abstracts a lot of SNMP details for you, as great libraries do, and supports both Python 2 and Python 3.

PySNMP requires the PyASN1 package. The following is taken from Wikipedia:

> *"ASN.1 is a standard and notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunication and computer networking."*

PyASN1 conveniently provides a Python wrapper around ASN.1. Let's install the package first. Note that since we are using a virtual environment, we will use the virtual environment's Python interpreter:

```
(venv) $ cd /tmp
(venv) $ git clone https ://github.com/etingof/pyasn1.git
(venv) $ cd pyasn1
(venv) $ git checkout 0.2.3
(venv) $ python3 setup.py install # notice the venv path
```

Next, install the PySNMP package:

```
(venv) $ cd /tmp
(venv) $ git clone https://github.com/etingof/pysnmp
(venv) $ cd pysnmp/
(venv) $ git checkout v4.3.10
(venv) $ python3 setup.py install # notice the venv path
```



We are using an older version of PySNMP due to the fact that `pysnmp.entity.rfc3413.oneliner` was removed starting with version 5.0.0 (`https://github.com/etingof/pysnmp/blob/a93241007b970c458a0233c16ae2ef82dc107290/CHANGES.txt`). If you use `pip` to install the packages, the examples will probably break.

Let's look at how to use PySNMP to query the same Cisco contact information we used in the previous example. We will import the necessary module and create a `CommandGenerator` object first:

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> cmdGen = cmdgen.CommandGenerator()
>>> cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"
```

We can perform SNMP using the `getCmd` method. The result is unpacked into various variables; of these, we care most about `varBinds`, which contains the query result:

```
>>> errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
        cmdgen.CommunityData('secret'),
        cmdgen.UdpTransportTarget(('192.168.2.218', 161)),
        cisco_contact_info_oid)
>>> for name, val in varBinds:
        print('%s=%s' % (name.prettyPrint(), str(val)))

SNMPv2-SMI::enterprises.9.2.1.61.0=cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web http://www.cisco.com
>>>
```

Note that the response values are PyASN1 objects. The `prettyPrint()` method will convert some of these values into a human-readable format, but the result in our return variable was not converted. We converted it into a string manually.

We can write a script based on the preceding interactive example. We will name it `pysnmp_1.py` with error checking. We can also include multiple OIDs in the `getCmd()` method:

```
#!/usr/bin/env/python3
from pysnmp.entity.rfc3413.oneliner import cmdgen
cmdGen = cmdgen.CommandGenerator()
system_up_time_oid = "1.3.6.1.2.1.1.3.0"
cisco_contact_info_oid = "1.3.6.1.4.1.9.2.1.61.0"
errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
    cmdgen.CommunityData('secret'),
    cmdgen.UdpTransportTarget(('192.168.2.218', 161)),
    system_up_time_oid,
    cisco_contact_info_oid
)
# Check for errors and print out results
if errorIndication:
    print(errorIndication)
else:
    if errorStatus:
        print('%s at %s' % (
            errorStatus.prettyPrint(),
            errorIndex and varBinds[int(errorIndex)-1] or '?'
            )
        )
    else:
        for name, val in varBinds:
            print('%s = %s' % (name.prettyPrint(), str(val)))
```

The result will be unpacked and list the values of the two OIDs:

```
$ python pysnmp_1.py
SNMPv2-MIB::sysUpTime.0 = 599083
SNMPv2-SMI::enterprises.9.2.1.61.0 = cisco Systems, Inc.
170 West Tasman Dr.
San Jose, CA  95134-1706
U.S.A.
Ph +1-408-526-4000
```

```
Customer service 1-800-553-6387 or +1-408-526-7208
24HR Emergency 1-800-553-2447 or +1-408-526-7209
Email Address tac@cisco.com
World Wide Web http://www.cisco.com
```

In the following example, we will persist the values we received from the queries to perform other functions, such as visualization, with the data. For our example, we will use `ifEntry` within the MIB-2 tree for interface-related values to be graphed.

You can find a number of resources that map out the `ifEntry` tree; here is a screenshot of the Cisco SNMP Object Navigator site that we accessed previously for `ifEntry`:



Figure 7.3: SNMP ifEntry OID tree

A quick test will illustrate the OID mapping of the interfaces on the device:

```
$ snmpwalk -v2c -c secret 172.16.1.189 .1.3.6.1.2.1.2.2.1.2
iso.3.6.1.2.1.2.2.1.2.1 = STRING: "GigabitEthernet0/0"
iso.3.6.1.2.1.2.2.1.2.2 = STRING: "GigabitEthernet0/1"
iso.3.6.1.2.1.2.2.1.2.3 = STRING: "GigabitEthernet0/2"
iso.3.6.1.2.1.2.2.1.2.4 = STRING: "Null0"
iso.3.6.1.2.1.2.2.1.2.5 = STRING: "Loopback0"
```

From the documentation, we can map the values of `ifInOctets(10)`, `ifInUcastPkts(11)`, `ifOutOctets(16)`, and `ifOutUcastPkts(17)` into their respective OID values. From a quick check of the CLI and MIB documentation, we can see that the value of the `GigabitEthernet0/0` packets output maps to OID `1.3.6.1.2.1.2.2.1.17.1`. We will follow the rest of the same process to map out the rest of the OIDs for the interface statistics. When checking between the CLI and SNMP, keep in mind that the values should be close but not the same since there might be some traffic on the wire between the time of the CLI output and the SNMP query time:

```
r1#sh int gig 0/0 | i packets
  5 minute input rate 0 bits/sec, 0 packets/sec
  5 minute output rate 0 bits/sec, 0 packets/sec
     6872 packets input, 638813 bytes, 0 no buffer
     4279 packets output, 393631 bytes, 0 underruns
$ snmpwalk -v2c -c secret 192.168.2.218 .1.3.6.1.2.1.2.2.1.17.1
iso.3.6.1.2.1.2.2.1.17.1 = Counter32: 4292
```

If we are in a production environment, we will probably write the results into a database. But since this is just an example, we will write the query values to a flat file. We will write the `pysnmp_3.py` script for information queries and write the results to the file. In the script, we have defined various OIDs that we need to query:

```
# Hostname OID
system_name = '1.3.6.1.2.1.1.5.0'
# Interface OID
gig0_0_in_oct = '1.3.6.1.2.1.2.2.1.10.1'
gig0_0_in_uPackets = '1.3.6.1.2.1.2.2.1.11.1'
gig0_0_out_oct = '1.3.6.1.2.1.2.2.1.16.1'
gig0_0_out_uPackets = '1.3.6.1.2.1.2.2.1.17.1'
```

The values were consumed in the `snmp_query()` function, with `host`, `community`, and `oid` as input:

```
def snmp_query(host, community, oid):
    errorIndication, errorStatus, errorIndex, varBinds = cmdGen.getCmd(
        cmdgen.CommunityData(community),
        cmdgen.UdpTransportTarget((host, 161)),
        oid
    )
```

All of the values are put in a dictionary with various keys and written to a file called `results.txt`:

```
result = {}
result['Time'] = datetime.datetime.utcnow().isoformat()
result['hostname'] = snmp_query(host, community, system_name)
result['Gig0-0_In_Octet'] = snmp_query(host, community, gig0_0_in_oct)
result['Gig0-0_In_uPackets'] = snmp_query(host, community, gig0_0_in_
uPackets)
result['Gig0-0_Out_Octet'] = snmp_query(host, community, gig0_0_out_oct)
result['Gig0-0_Out_uPackets'] = snmp_query(host, community, gig0_0_out_
uPackets)
with open('/home/echou/Master_Python_Networking/Chapter7/results.txt',
'a') as f:
    f.write(str(result))
    f.write('\n')
```

The outcome will be a file with results showing the interface packets represented at the time of the query:

```
$ cat results.txt
{'Gig0-0_In_Octet': '3990616', 'Gig0-0_Out_uPackets': '60077', 'Gig0-
0_In_uPackets': '42229', 'Gig0-0_Out_Octet': '5228254', 'Time': '2017-03-
06T02:34:02.146245', 'hostname': 'iosv-1.virl.info'}
{'Gig0-0_Out_uPackets': '60095', 'hostname': 'iosv-1.virl.info', 'Gig0-
0_Out_Octet': '5229721', 'Time': '2017-03-06T02:35:02.072340', 'Gig0-0_In_
Octet': '3991754', 'Gig0-0_In_uPackets': '42242'}
<skip>
```

We can make this script executable and schedule a `cron` job to be executed every 5 minutes:

```
$ chmod +x pysnmp_3.py
# crontab configuration
*/5 * * * * /home/echou/Mastering_Python_Networking_Fourth_Edition/
Chapter07/pysnmp_3.py
```

As mentioned previously, in a production environment, we would put the information in a database. For a SQL database, you can use a unique ID as the primary key. In a NoSQL database, we might use time as the primary index (or key) because it is always unique, followed by various key-value pairs.

We will wait for the script to be executed a few times for the values to be populated. If you are the impatient type, you can shorten the `cron` job interval to 1 minute. After you see enough values in the `results.txt` file to make an interesting graph, we can move on to the next section to see how we can use Python to visualize the data.

Python for Data Visualization

We gather network data to gain insight into our network. One of the best ways to know what the data means is to visualize it with graphs. This is true for almost all data, but especially true for time series data in the context of network monitoring. How much data was transmitted over the network in the last week? What is the percentage of the TCP protocol among all of the traffic? These are values we can glean from using data-gathering mechanisms such as SNMP, and we can produce visualization graphs with some of the popular Python libraries.

In this section, we will use the data we collected from the last section using SNMP and use two popular Python libraries, Matplotlib and Pygal, to graph them.

Matplotlib

Matplotlib (`http://matplotlib.org/`) is a Python 2D plotting library for the Python language and its NumPy mathematical extension. It can produce publication-quality figures, such as plots, histograms, and bar graphs, with a few lines of code.

NumPy is an extension of the Python programming language. It is open source and widely used in various data science projects. You can learn more about it at `https://en.wikipedia.org/wiki/NumPy`.

Let's begin with the installation.

Installation

The installation can be done using the Linux package management system for the distribution or Python `pip`. In the latest version of Matplotlib, we will also install `python3-tk` for display:

```
(venv) $ pip install matplotlib
(venv) $ sudo apt install python3-tk
```

Now, let's get into our first example.

Matplotlib – the first example

For the following examples, the output figures are displayed as the standard output by default. Typically, the standard output is your monitor screen. During development, it is often easier to try out the code initially and produce the graph on the standard output first before finalizing the code with a script. If you have been following along with this book via a virtual machine, it is recommended that you use the VM window instead of SSH so that you can see the graphs. If you do not have access to the standard output, you can save the figure and view it after downloading it (as you will see soon). Note that you will need to set the `$DISPLAY` variable in some of the graphs we produce in this section.

The following screenshot of the Ubuntu desktop is used in this chapter's visualization example. As soon as the `plt.show()` command is issued in the Terminal window, **Figure 1** will appear on the screen. When you close the figure, you will return to the Python shell:

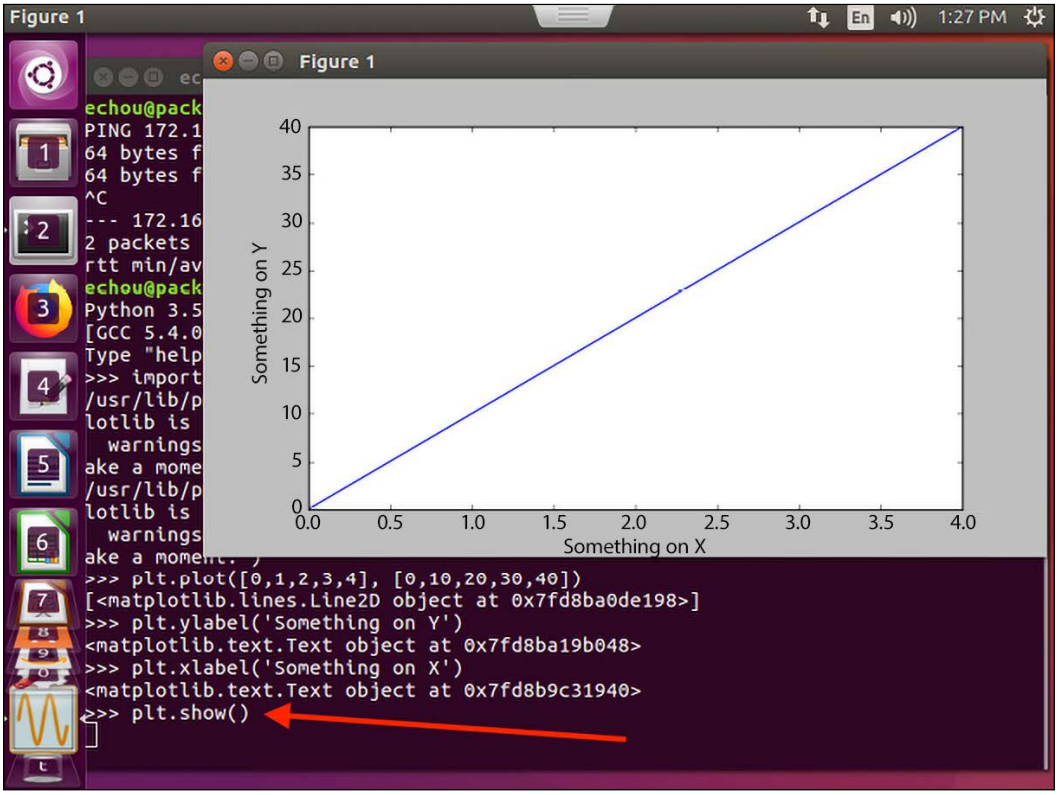


Figure 7.4: Matplotlib visualization with the Ubuntu desktop

Let's look at the line graph first. A line graph simply gives two lists of numbers that correspond to the x-axis and y-axis values:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([0,1,2,3,4], [0,10,20,30,40])
[<matplotlib.lines.Line2D object at 0x7f932510df98>]
>>> plt.ylabel('Something on Y')
<matplotlib.text.Text object at 0x7f93251546a0>
>>> plt.xlabel('Something on X')
<matplotlib.text.Text object at 0x7f9325fdb9e8>
>>> plt.show()
```

The graph will be a line graph:

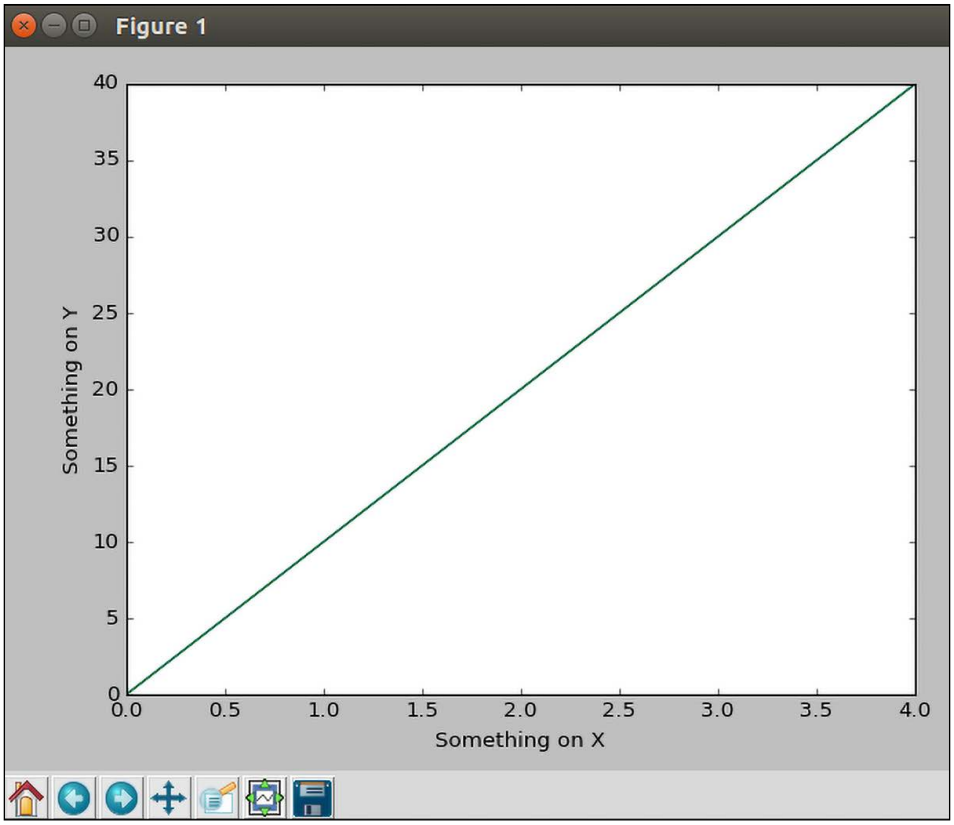


Figure 7.5: Matplotlib line graph

Alternatively, if you do not have access to standard output or have saved the figure first, you can use the `savefig()` method:

```
>>> plt.savefig('figure1.png') or
>>> plt.savefig('figure1.pdf')
```

With this basic knowledge of graphing plots, we can now graph the results we receive from SNMP queries.

Matplotlib for SNMP results

In our first Matplotlib example, `matplotlib_1.py`, we will import the `dates` module besides `pyplot`. We will use the `matplotlib.dates` module instead of the Python standard library `dates` module.

Unlike the Python `dates` module, the `matplotlib.dates` library will convert the date value internally into a float type, which is required by Matplotlib:

```
import matplotlib.pyplot as plt
import matplotlib.dates as dates
```

Matplotlib provides sophisticated date plotting capabilities; you can find more information on this at `https://matplotlib.org/stable/api/dates_api.html`.

In the script, we will create two empty lists, each representing the *x*-axis and *y*-axis values. Note that on line 12, we used the built-in `eval()` Python function to read the input as a dictionary instead of a default string:

```
x_time = []
y_value = []
with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) reads in each line as dictionary instead of string
        line = eval(line)
        # convert to internal float
        x_time.append(dates.datestr2num(line['Time']))
        y_value.append(line['Gig0-0_Out_uPackets'])
```

In order to read the *x*-axis value back in a human-readable date format, we will need to use the `plot_date()` function instead of `plot()`. We will also tweak the size of the figure a bit, as well as rotate the value on the *x*-axis so that we can read the value in full:

```
plt.subplots_adjust(bottom=0.3)
plt.xticks(rotation=80)
plt.plot_date(x_time, y_value)
plt.title('Router1 G0/0')
plt.xlabel('Time in UTC')
plt.ylabel('Output Unicast Packets')
plt.savefig('matplotlib_1_result.png')
plt.show()
```

The final result will display the **Router1 G0/0** and **Output Unicast Packets**, as follows:

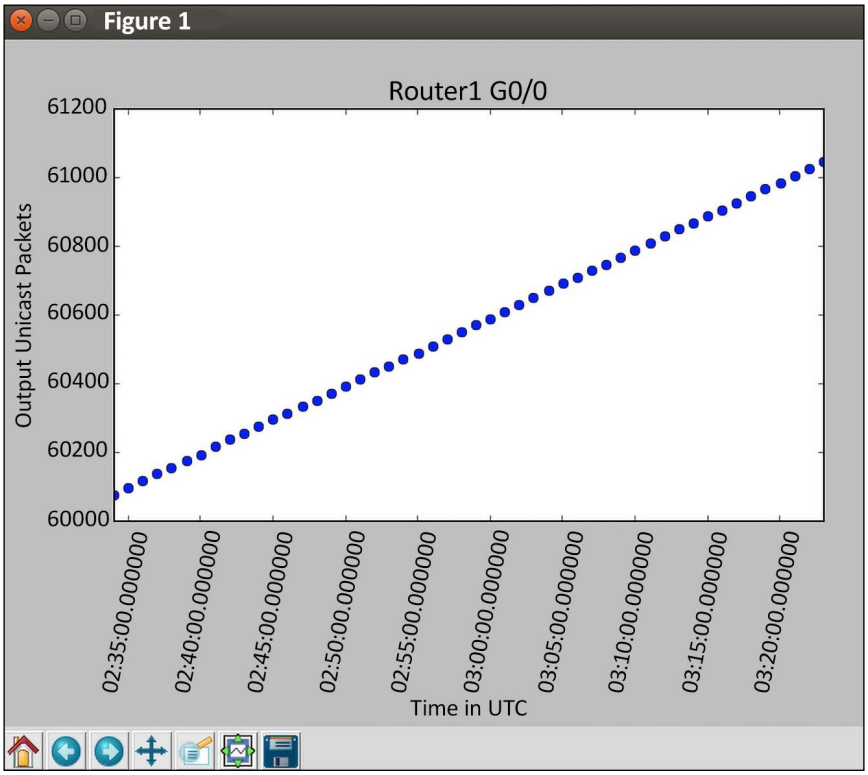


Figure 7.6: Router1 Matplotlib graph

Note that if you prefer a straight line instead of dots, you can use the third optional parameter in the `plot_date()` function:

```
plt.plot_date(x_time, y_value, "-")
```

We can repeat the steps for the rest of the values for output octets, input unicast packets, and input as individual graphs. However, in our next example, that is, `matplotlib_2.py`, we will show you how to graph multiple values against the same time range, as well as additional Matplotlib options.

In this case, we will create additional lists and populate the values accordingly:

```
x_time = []
out_octets = []
out_packets = []
in_octets = []
in_packets = []
with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) reads in each line as dictionary instead of string
        line = eval(line)
        # convert to internal float
        x_time.append(dates.datestr2num(line['Time']))
        out_packets.append(line['Gig0-0_Out_uPackets'])
        out_octets.append(line['Gig0-0_Out_Octet'])
        in_packets.append(line['Gig0-0_In_uPackets'])
        in_octets.append(line['Gig0-0_In_Octet'])
```

Since we have identical *x*-axis values, we can just add the different *y*-axis values to the same graph:

```
# Use plot_date to display x-axis back in date format
plt.plot_date(x_time, out_packets, '-', label='Out Packets')
plt.plot_date(x_time, out_octets, '-', label='Out Octets')
plt.plot_date(x_time, in_packets, '-', label='In Packets')
plt.plot_date(x_time, in_octets, '-', label='In Octets')
```

Also, add `grid` and `legend` to the graph:

```
plt.title('Router1 G0/0')
plt.legend(loc='upper left')
plt.grid(True)
plt.xlabel('Time in UTC')
plt.ylabel('Values')
plt.savefig('matplotlib_2_result.png')
plt.show()
```

The final result will combine all of the values in a single graph. Note that some of the values in the upper-left corner are blocked by the legend. You can resize the figure and/or use the pan/zoom option to move around the graph to see the values:

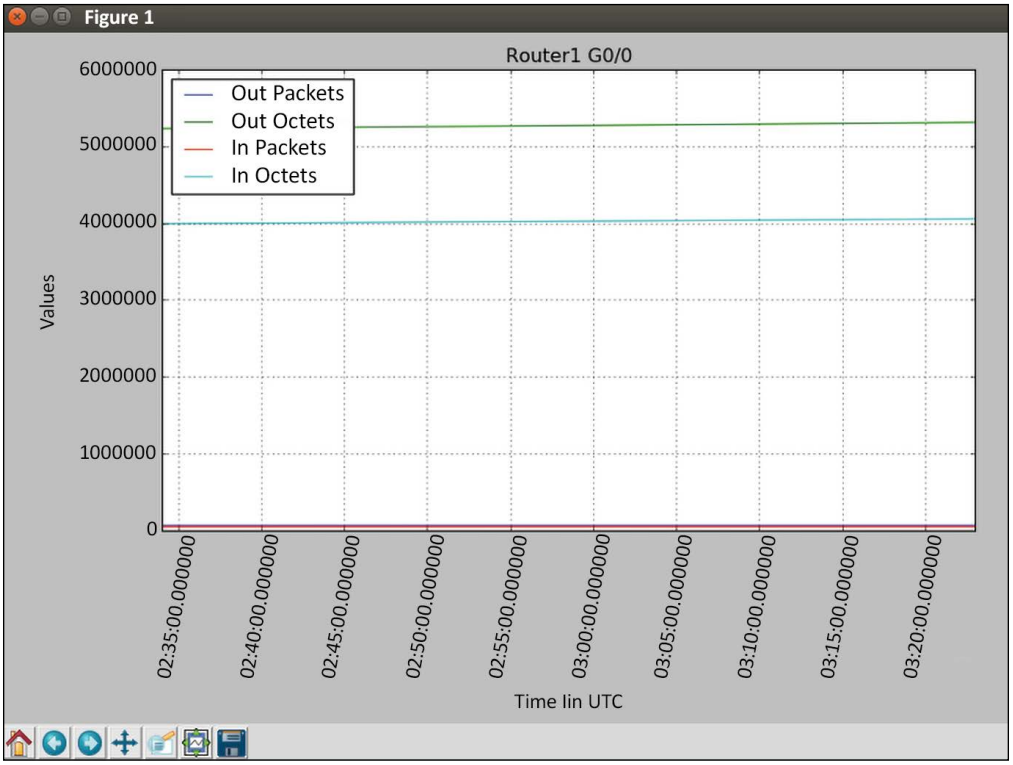


Figure 7.7: Router1 – Matplotlib multiline graph

There are many more graphing options available in Matplotlib; we are certainly not limited to plot graphs. For example, in `matplotlib_3.py`, we can use the following mock data to graph the percentage of different traffic types that we can see on the wire:

```
#!/usr/bin/env python3
# Example from http://matplotlib.org/2.0.0/examples/pie_and_polar_charts/
pie_demo_features.html
import matplotlib.pyplot as plt
# Pie chart, where the slices will be ordered and plotted counter-
clockwise:
labels = 'TCP', 'UDP', 'ICMP', 'Others'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # Make UDP stand out
```

```
fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
circle.
plt.savefig('matplotlib_3_result.png')
plt.show()
```

The preceding code leads to this pie chart from `plt.show()`:

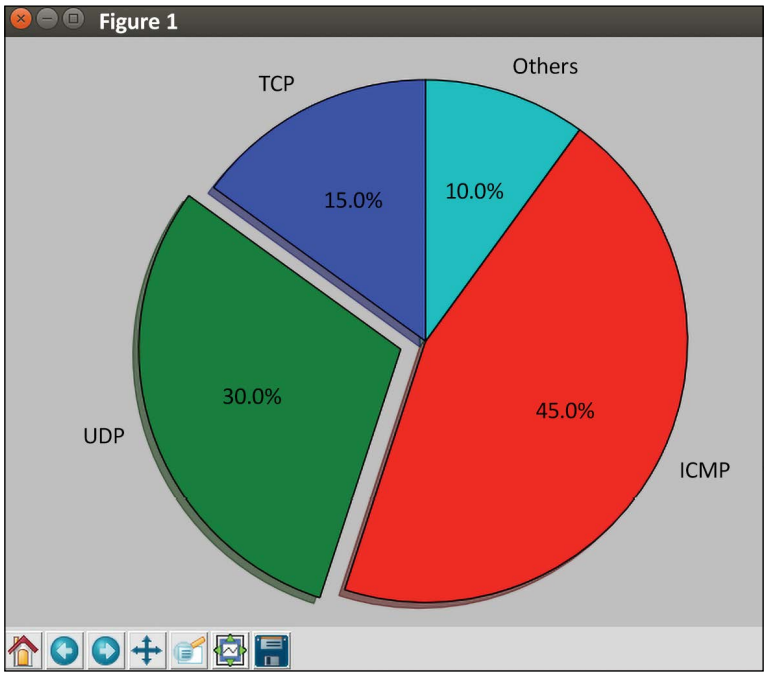


Figure 7.8: Matplotlib pie chart

In this section, we have used Matplotlib to graph our network data into more visually appealing graphs to help us understand the state of our network. This was done with bar graphs, line charts, and pie charts, which are appropriate for the data at hand. Matplotlib is a powerful tool that is not limited to Python. As an open-source tool, many additional Matplotlib resources can be leveraged to learn about the tool.

Additional Matplotlib resources

Matplotlib is one of the best Python plotting libraries, able to produce publication-quality figures. Like Python, it aims to make complex tasks simple. With over 10,000 stars (and counting) on GitHub, it is also one of the most popular open-source projects.

Its popularity directly translates into faster bug fixes, a friendly user community, extensive documentation, and general usability. Using the package has a bit of a learning curve, but it is well worth the effort.

In this section, we barely scratched the surface of Matplotlib. You'll find additional resources at `https://matplotlib.org/stable/index.html` (the Matplotlib project page) and `https://github.com/matplotlib/matplotlib` (the Matplotlib GitHub repository).

In the coming section, we will take a look at another popular Python graph library: **Pygal**.

Pygal

Pygal (`https://www.pygal.org/en/stable/`) is a dynamic **Scalable Vector Graphics** (**SVG**) charting library written in Python. The biggest advantage of Pygal, in my opinion, is that it produces write SVG graphs easily and natively. There are many advantages of SVG over other graph formats. Two of the main advantages are that it is web browser-friendly and it provides scalability without sacrificing image quality. In other words, you can display the resulting image in any modern web browser and zoom in and out of the image without losing the details of the graph. Did I mention that we can do this in a few lines of Python code? How cool is that?

Let's get Pygal installed, then move on to the first example.

Installation

The installation is done via `pip`:

```
(venv)$ pip install pygal
```

Pygal – the first example

Let's look at the line chart example demonstrated on Pygal's documentation, available at `http://pygal.org/en/stable/documentation/types/line.html`:

```
>>> import pygal
>>> line_chart = pygal.Line()
>>> line_chart.title = 'Browser usage evolution (in %)'
>>> line_chart.x_labels = map(str, range(2002, 2013))
>>> line_chart.add('Firefox', [None, None,    0, 16.6,   25,   31, 36.4,
45.5, 46.3, 42.8, 37.1])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.add('Chrome',  [None, None, None, None, None, None,    0,
3.9, 10.8, 23.8, 35.3])
<pygal.graph.line.Line object at 0x7f4883c52b38>
```

```
>>> line_chart.add('IE',      [85.8, 84.6, 84.7, 74.5,   66, 58.6, 54.7,
44.8, 36.2, 26.6, 20.1])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.add('Others',  [14.2, 15.4, 15.3,  8.9,    9, 10.4,  8.9,
5.8,  6.7,  6.8,  7.5])
<pygal.graph.line.Line object at 0x7f4883c52b38>
>>> line_chart.render_to_file('pygal_example_1.svg')
```

In this example, we created a line object with the `x_labels` automatically rendered as strings for 11 units. Each of the objects can be added with the label and the value in a list format, such as Firefox, Chrome, and IE.

The interesting bit to focus on is the fact that each of the line chart items has the exact number of matching numbers to the number of x units. When there is no value, for example, the years 2002 – 2007 for Chrome, the value **None** is entered.

Here's the resulting graph, as viewed in the Firefox browser:

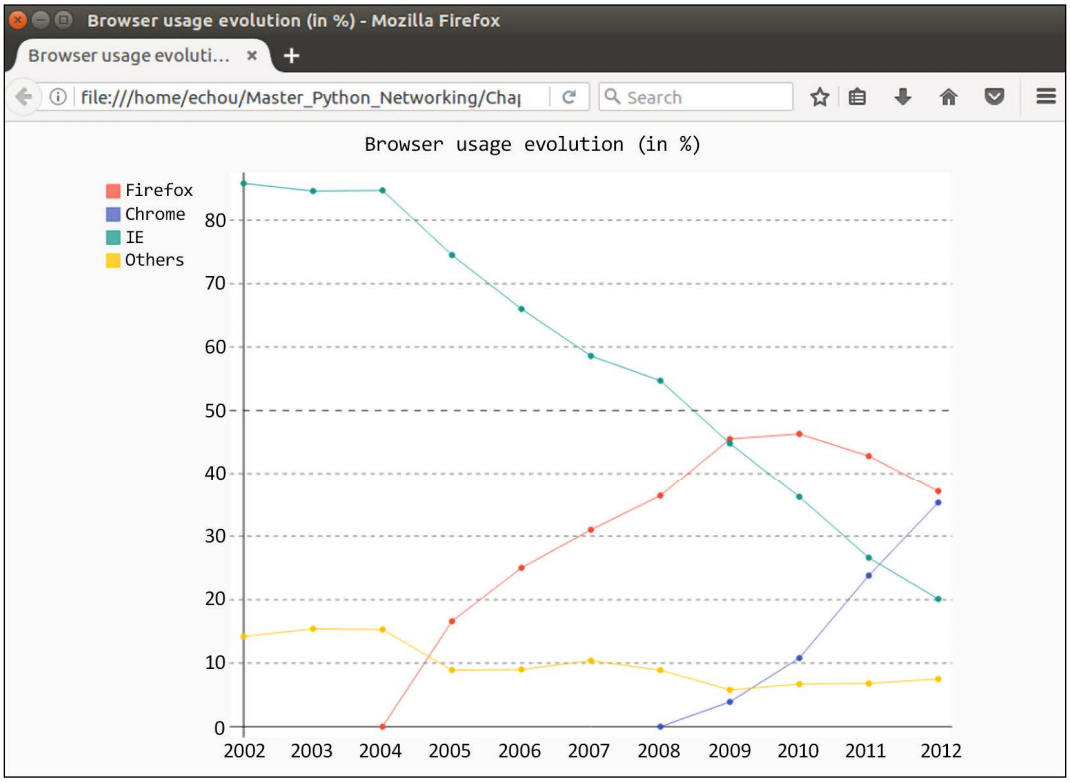


Figure 7.9: Pygal sample graph

Now that we can see the general usage of Pygal, we can use the same method to graph the SNMP results we have in hand. We will do this in the coming section.

Pygal for SNMP results

For the Pygal line graph, we can largely follow the same pattern as our Matplotlib example, where we create lists of values by reading the file. We no longer need to convert the *x*-axis value into an internal float, as we did for Matplotlib; however, we do need to convert the numbers in each of the values we would have received into float:

```
#!/usr/bin/env python3
import pygal
x_time = []
out_octets = []
out_packets = []
in_octets = []
in_packets = []
with open('results.txt', 'r') as f:
    for line in f.readlines():
        # eval(line) reads in each line as dictionary instead of string
        line = eval(line)
        x_time.append(line['Time'])
        out_packets.append(float(line['Gig0-0_Out_uPackets']))
        out_octets.append(float(line['Gig0-0_Out_Octet']))
        in_packets.append(float(line['Gig0-0_In_uPackets']))
        in_octets.append(float(line['Gig0-0_In_Octet']))
```

We can use the same mechanism that we saw to construct the line graph:

```
line_chart = pygal.Line()
line_chart.title = "Router 1 Gig0/0"
line_chart.x_labels = x_time
line_chart.add('out_octets', out_octets)
line_chart.add('out_packets', out_packets)
line_chart.add('in_octets', in_octets)
line_chart.add('in_packets', in_packets)
line_chart.render_to_file('pygal_example_2.svg')
```

The outcome is similar to what we have already seen, but the graph is now in an SVG format that can be easily displayed on a web page. It can be viewed in a modern web browser:

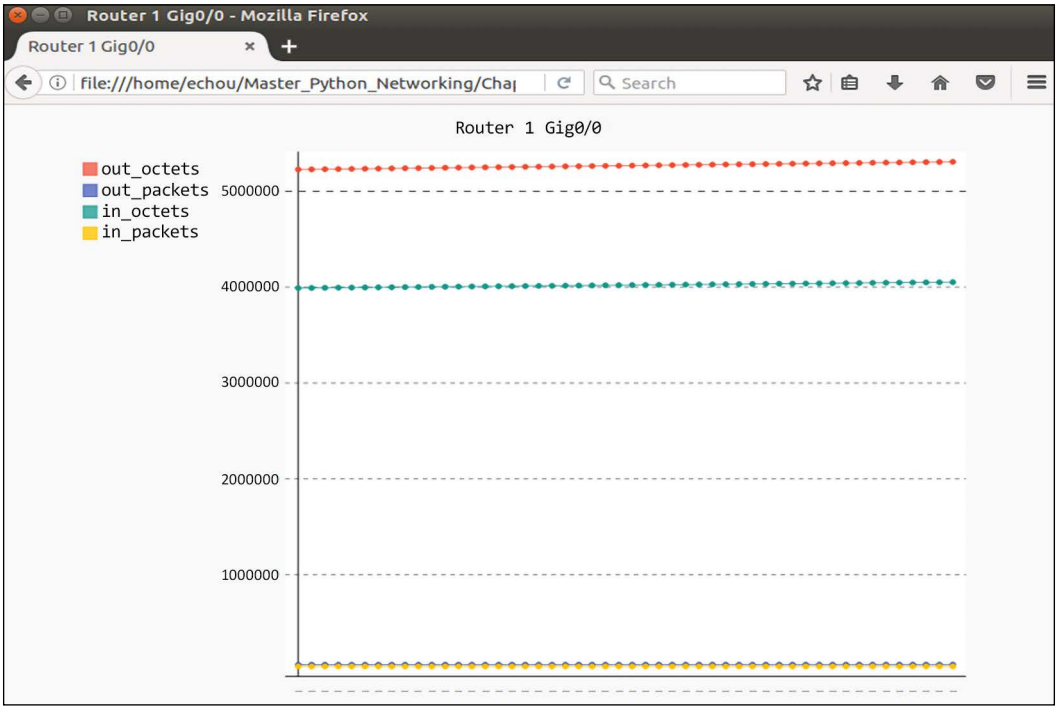


Figure 7.10: Router 1— Pygal multiline graph

Just like Matplotlib, Pygal provides many more options for graphs. For example, to graph the pie chart we saw previously in Matplotlib, we can use the `pygal.Pie()` object. This is shown in `pygal_2.py`:

```
#!/usr/bin/env python3
import pygal
line_chart = pygal.Pie()
line_chart.title = "Protocol Breakdown"
line_chart.add('TCP', 15)
line_chart.add('UDP', 30)
line_chart.add('ICMP', 45)
line_chart.add('Others', 10)
line_chart.render_to_file('pygal_example_3.svg')
```

The resulting SVG file is shown here:

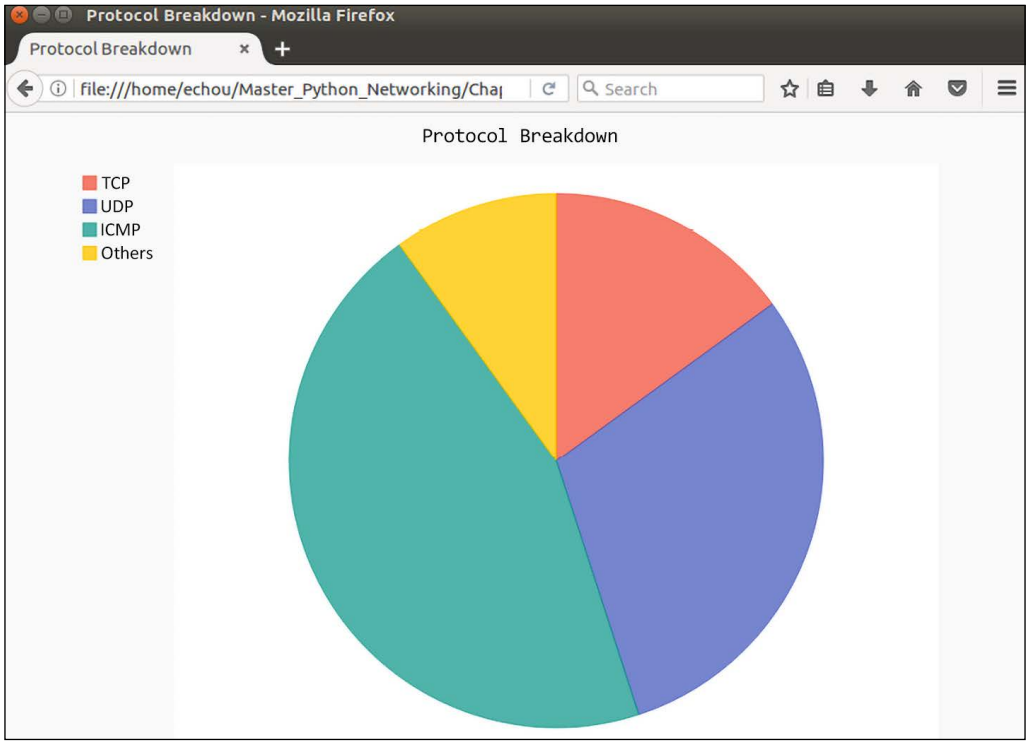


Figure 7.11: Pygal pie chart

Pygal is a great tool when it comes to generating production-ready SVG graphs. If this is the type of graph required, look no further than the Pygal library. In this section, we have examined examples of using Pygal to generate graphs for our network data. Similar to Matplotlib, there are many additional resources to help us to learn about Pygal if you're interested.

Additional Pygal resources

Pygal provides many more customizable features and graphing capabilities for the data you collect from basic network monitoring tools such as SNMP. We demonstrated a simple line graph and pie graph in this section. You can find more information about the project here:

- **Pygal documentation**: `http://www.pygal.org/en/stable/index.html`
- **Pygal GitHub project page**: `https://github.com/Kozea/pygal`

In the coming section, we will continue with the SNMP theme of network monitoring but with a fully featured network monitoring system called **Cacti**.

Python for Cacti

In my early days as a junior network engineer at a regional ISP, we used the open-source cross-platform **Multi Router Traffic Grapher (MRTG)** (`https://en.wikipedia.org/wiki/Multi_Router_Traffic_Grapher`) tool to check the traffic load on network links. We relied on the tool almost exclusively for traffic monitoring. I was amazed at how good and useful an open-source project could be. It was one of the first open-source high-level network monitoring systems that abstracted the details of SNMP, the database, and HTML for network engineers. Then came the **round-robin database tool (RRDtool)** (`https://en.wikipedia.org/wiki/RRDtool`). In its first release in 1999, it was referred to as "MRTG Done Right." It greatly improved the database and poller performance in the backend.

Released in 2001, Cacti (`https://en.wikipedia.org/wiki/Cacti_(software)`) is an open-source web-based network monitoring and graphing tool designed as an improved frontend for RRDtool. Because of the heritage of MRTG and RRDtool, you will notice a familiar graph layout, templates, and SNMP poller. As a packaged tool, the installation and usage will need to stay within the boundary of the tool. However, Cacti offers a custom data query feature that we can use Python to handle. In this section, we will see how we can use Python as an input method for Cacti.

First, we'll go through the installation process.

Installation

Because Cacti is an all-in-one tool, including web frontend, collection scripts, and database backend, unless you already have experience with Cacti, I would recommend installing the tool on a standalone VM or a container in our lab. The following instructions will be shown for a VM, but a container Dockerfile would be similar.

Installation on Ubuntu is straightforward when using APT on the Ubuntu management VM:

```
$ sudo apt-get install cacti
```

It will trigger a series of installation and setup steps, including the MySQL database, web server (Apache or `lighttpd`), and various configuration tasks. Once it's installed, navigate to `http://<ip>/cacti` to get started. The last step is to log in with the default username and password (`admin/admin`); you will be prompted to change the password.

During installation, when in doubt, go with the default option and keep it simple.

Once logged in, we can follow the documentation to add a device and associate it with a template. There is a Cisco router premade template that you can go with. Cacti has good documentation at `http://docs.cacti.net/` for adding a device and creating your first graph, so we will quickly look at some screenshots that you can expect to see:

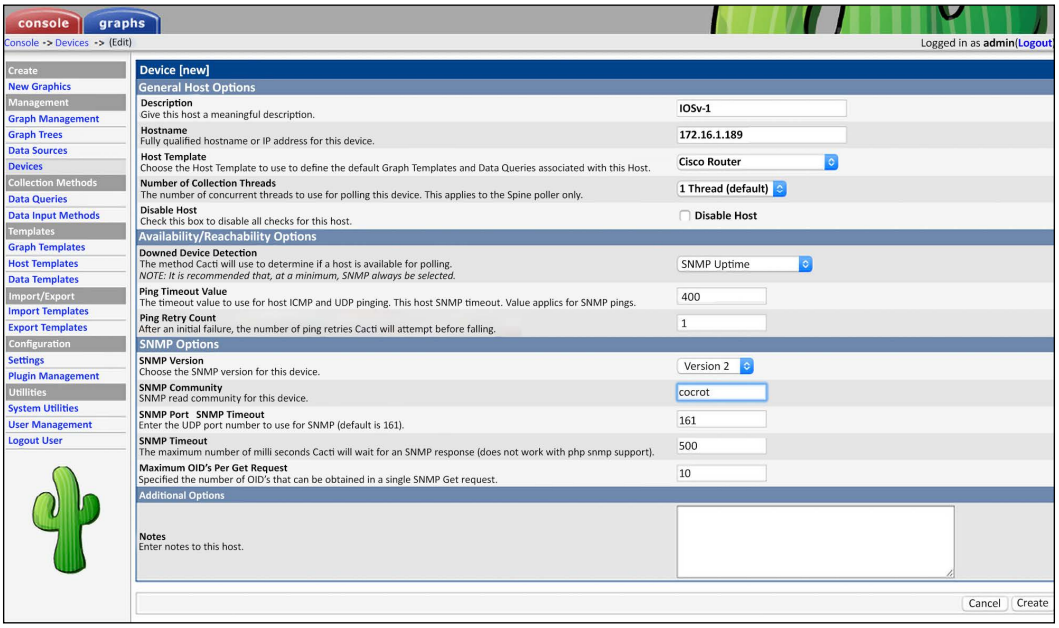


Figure 7.12: Cacti device edit page

A sign indicating the SNMP communication is working is when you can see the device uptime:

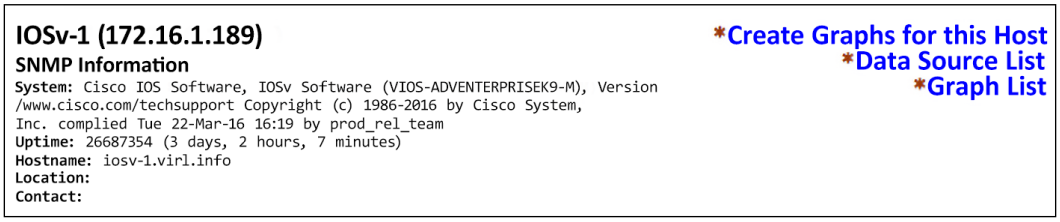


Figure 7.13: Device edit result page

You can add graphs to the device for interface traffic and other statistics:

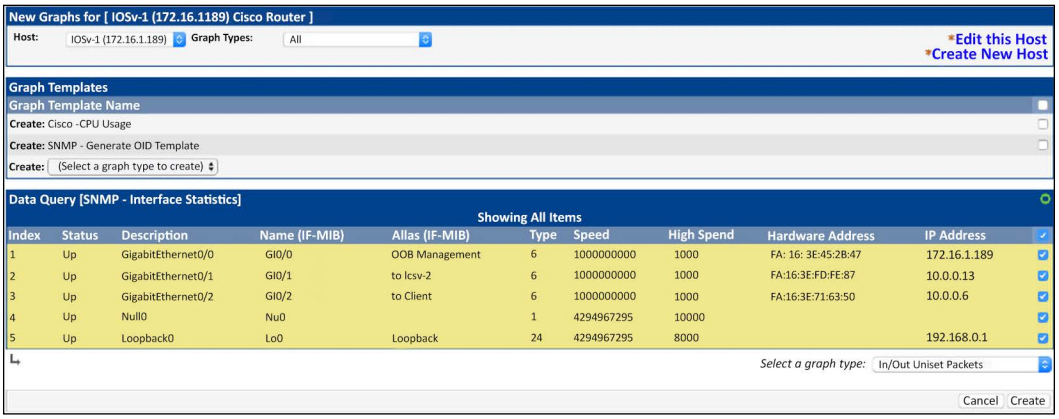


Figure 7.14: New graphs for the device

After some time, you will start seeing traffic, as shown here:

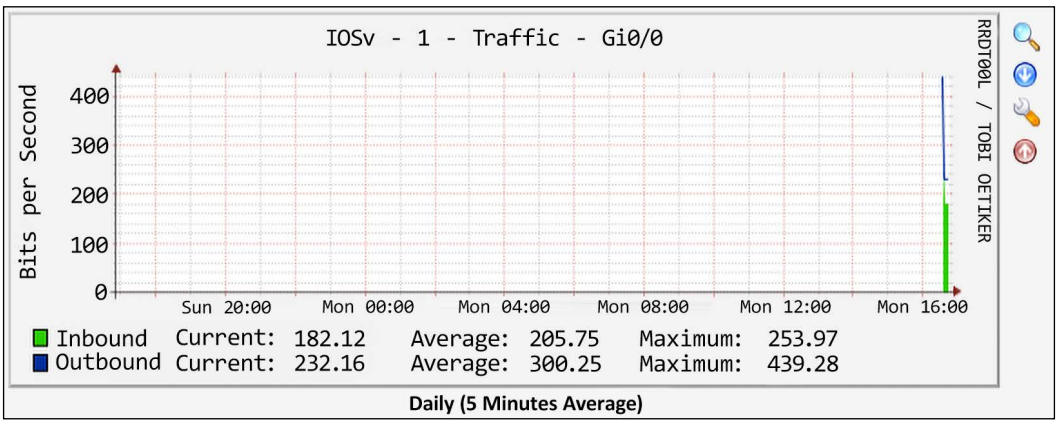


Figure 7.15: 5-minute average graph

We are now ready to look at how to use Python scripts to extend Cacti's data-gathering functionality.

Python script as an input source

There are two documents that we should read before we try to use our Python script as an input source:

- **Data input methods**: `http://www.cacti.net/downloads/docs/html/data_input_methods.html`

- **Making your scripts work with Cacti**: `http://www.cacti.net/downloads/docs/html/making_scripts_work_with_cacti.html`

You might wonder what the use cases are for using a Python script as an extension for data inputs. One of the use cases would be to provide monitoring to resources that do not have a corresponding OID, for example, if we would like to know how to graph how many times the access list `permit_snmp` has allowed the host `172.16.1.173` to conduct an SNMP query.



The example assumes the SNMP station is at IP `172.16.1.173`; please substitute the IP for the current lab management station IP.

We know we can see the number of matches via the CLI:

```
iosv-1#sh ip access-lists permit_snmp | I 172.16.1.173 10 permit
172.16.1.173 log (6362 matches)
```

However, the chances are there are no OIDs associated with this value (or we can just pretend that there are none). This is where we can use an external script to produce an output that can be consumed by the Cacti host.

We can reuse the Pexpect script we discussed in *Chapter 2*, *Low-Level Network Device Interactions*, `chapter1_1.py`. We will rename it `cacti_1.py`. Everything should be the same as the original script, except that we will execute the CLI command and save the output:

```
<skip>
for device in devices.keys():
…
    child.sendline''sh ip access-lists permit_snmp | i 172.16.1.17'')
    child.expect(device_prompt)
    output = child.before
```

The output in its raw form will appear as follows:

```
''sh ip access-lists permit_snmp | i 172.16.1.173rn 10 permit 172.16.1.173
log (6428 matches)r''
```

We will use the `split()` function for the string to only leave the number of matches and print them out on standard output in the script:

```
print(str(output).split'''')[1].split()[0])
```

To test this, we can see the number of increments by executing the script a number of times:

```
$ ./cacti_1.py
6428
$ ./cacti_1.py
6560
$ ./cacti_1.py
6758
```

We can make the script executable and put it into the default Cacti script location:

```
$ chmod a+x cacti_1.py
$ sudo cp cacti_1.py /usr/share/cacti/site/scripts/
```

The Cacti documentation, available at `http://www.cacti.net/downloads/docs/html/how_to.html`, provides detailed steps on how to add the script result to the output graph.

These steps include adding the script as a data input method, adding the input method to a data source, and then creating a graph to be viewed:

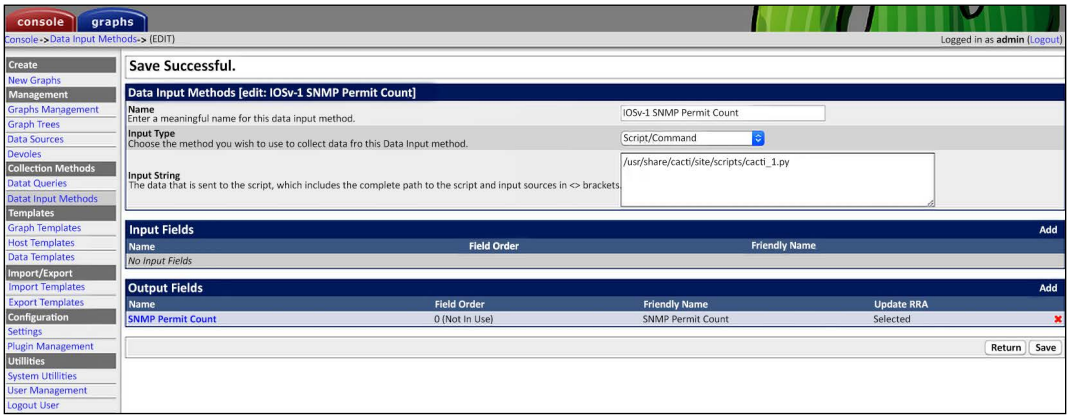


Figure 7.16: Data input method results page

SNMP is a common way to provide network monitoring services to devices. RRDtool with Cacti as the frontend provides a good platform for all network devices via SNMP. We can also use Python scripts to extend information gathering beyond SNMP.

Summary

In this chapter, we explored ways to perform network monitoring via SNMP. We configured SNMP-related commands on network devices and used our network management VM with an SNMP poller to query the devices. We used the PySNMP module to simplify and automate our SNMP queries. We also learned how to save the query results in a flat file or database to be used for future examples.

Later in this chapter, we used two different Python visualization packages, Matplotlib and Pygal, to graph SNMP results. Each package has its distinct advantages. Matplotlib is a mature, feature-rich library widely used in data science projects. Pygal can natively generate SVG-format graphs that are flexible and web-friendly. We saw how to generate line and pie graphs relevant to network monitoring.

Toward the end of this chapter, we looked at an all-inclusive network monitoring tool named Cacti. It primarily uses SNMP for network monitoring, but we saw how we could use Python scripts as an input source to extend the platform's monitoring capabilities when SNMP OID is not available on the remote host.

In *Chapter 8*, *Network Monitoring with Python – Part 2*, we will continue to discuss the tools we can use to monitor our networks and gain insight into whether the network is behaving as expected. We will look at flow-based monitoring using NetFlow, sFlow, and IPFIX. We will also use tools such as Graphviz to visualize our network topology and detect any topological changes.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



8

Network Monitoring with Python – Part 2

In *Chapter 7, Network Monitoring with Python – Part 1*, we used SNMP to query information from network devices. We did this using an SNMP manager to query the SNMP agent residing on the network device. The SNMP information is structured in a hierarchy format with a specific object ID as a way to represent the value of the object. Most of the time, the value we care about is a number, such as CPU load, memory usage, or interface traffic. We can graph this data against time to give us a sense of how the value has changed over time.

We can typically classify the SNMP approach as a `pull` method as we constantly ask the device for a particular answer. This method adds a burden to the device because it needs to spend a CPU cycle on the control plane to find answers from the subsystem, package the answer in an SNMP packet, and transport the answer back to the poller. If you have ever been to a family reunion where you have that one family member who keeps asking you the same questions repeatedly, that would be analogous to the SNMP manager polling the managed node.

Over time, if we have multiple SNMP pollers querying the same device every 30 seconds (you would be surprised how often this happens), the management overhead will become substantial. In the same family reunion example we have given, instead of one family member, imagine many people interrupting you every 30 seconds to ask you a question. I don't know about you, but I know I would be very annoyed even if it was a simple question (or worse, if all of them were asking the same question).

Another way we can provide more efficient network monitoring is to reverse the relationship between the management station from a *pull* to a *push* model. In other words, the information can be pushed from the device toward the management station in an agreed-upon format. This concept is what flow-based monitoring is based on. In a flow-based model, the network device streams the traffic information, called flow, to the management station. The format can be the Cisco proprietary NetFlow (version 5 or 9), the industry-standard IPFIX, or the open source sFlow format. In this chapter, we will spend some time looking into NetFlow, IPFIX, and sFlow with Python.

Not all monitoring comes in the form of time series data. You can represent information such as network topology and Syslog in a time series format if you want to, but this is not ideal. We can use Python to check network topology information and see whether the topology has changed over time. We can use tools, such as Graphviz, with a Python wrapper to illustrate the topology. As already seen in *Chapter 6*, *Network Security with Python*, Syslog contains security information. Later in this book, we will look at using the Elastic Stack (Elasticsearch, Logstash, Kibana, and Beat) as an efficient way to collect and index network security and log information.

Specifically, in this chapter, we will cover the following topics:

- Graphviz, which is an open source graph visualization software that can help us quickly and efficiently graph our network
- Flow-based monitoring, such as NetFlow, IPFIX, and sFlow
- Using ntop to visualize the flow of information

Let's start by looking at how to use Graphviz to monitor network topology changes.

Graphviz

Graphviz is an open source graph visualization software. Imagine we have to describe our network topology to a colleague without the benefit of a picture. We might say our network consists of three layers: core, distribution, and access.

The core layer comprises two routers for redundancy, and both of the routers are full-meshed toward the four distribution routers; the distribution routers are also full-meshed toward the access routers. The internal routing protocol is OSPF, and externally, we use BGP for peering with our service provider. While this description lacks some details, it is probably enough for your colleague to paint a pretty good high-level picture of your network.

Graphviz works similarly to the process by describing the graph in a text format that Graphviz can understand in a text file. We can then feed the file to the Graphviz program to construct the graph. Here, the graph is described in a text format called DOT (`https://en.wikipedia.org/wiki/DOT_(graph_description_language)`) and Graphviz renders the graph based on the description. Of course, because the computer lacks human imagination, the language has to be very precise and detailed.

For Graphviz-specific DOT grammar definitions, take a look at `http://www.graphviz.org/doc/info/lang.html`.

In this section, we will use the **Link Layer Discovery Protocol (LLDP)** to query the device neighbors and create a network topology graph via Graphviz. Upon completing this extensive example, we will see how we can take something new, such as Graphviz, and combine it with things we have already learned (network LLDP) to solve interesting problems (automatically graph the current network topology).

Let's start by looking over the lab we will be using.

Lab setup

We will use the same lab topology as the last chapter. To recap, we have a three-tier topology, with r6 being the external facing edge device and r5 the top-of-rack router connecting to the server.

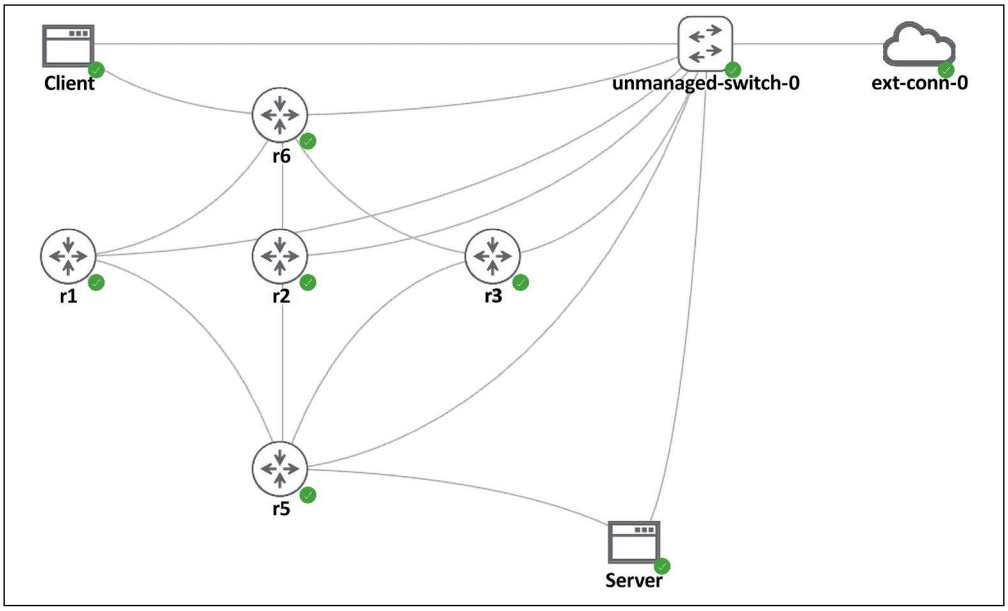


Figure 8.1: Lab topology

The devices are vIOS devices to save on lab resources and simplify configuration:

- Nodes virtualized by NX-OS and IOS-XR are much more memory-intensive than IOS.
- If you wish to use NX-OS, consider using NX-API or other API calls that will return structured data.

The devices have the following information:

Device	Management IP	Loopback IP
r1	192.168.2.218	192.168.0.1
r2	192.168.2.219	192.168.0.2
r3	192.168.2.220	192.168.0.3
r5	192.168.2.221	192.168.0.4
r6	192.168.2.222	192.168.0.5

The Ubuntu host's information is as follows:

Device Name	External Link Eth0	Internal IP Eth1
Client	192.168.2.211	10.0.0.9
Server	192.168.2.212	10.0.0.5

For our example, we will use LLDP (`https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol`). It is a vendor-neutral link-layer neighbor discovery protocol. Let us proceed to install the necessary software packages.

Installation

Graphviz can be obtained via `apt`:

```
$ sudo apt-get install graphviz
```

After the installation is complete, note that verification is performed by using the `dot` command:

```
$ dot -V
dot - graphviz version 2.43.0 (0)$ dot -V
```

We will use the Python wrapper for Graphviz, so let's install it now while we are at it:

```
(venv)$ pip install graphviz
>>> import graphviz
>>> graphviz.__version__
'0.20.1'
>>> exit()
```

Let's take a look at how we can use the software.

Graphviz examples

Like most popular open source projects, the documentation of Graphviz (`https://www.graphviz.org/documentation/`) is extensive. The challenge for someone new to the software is often the starting point, going from zero to one. For our purpose, we will focus on the dot graph, which draws directed graphs as hierarchies (not to be confused with the DOT language, which is a graph description language).

Let's start with some of the basic concepts:

- Nodes represent our network entities, such as routers, switches, and servers
- The edges represent the links between the network entities
- The graph, nodes, and edges each have attributes (`https://www.graphviz.org/doc/info/attrs.html`) that can be tweaked
- After describing the network, we can output the network graph (`https://www.graphviz.org/doc/info/output.html`) in either PNG, JPEG, or PDF format

Our first example, `chapter8_gv_1.gv`, is an undirected dot graph consisting of four nodes (`core`, `distribution`, `access1`, and `access2`). The edges, represented by the dash (`-`) sign, join the core node to the distribution node, as well as the distribution node to both of the access nodes:

```
graph my_network {
    core -- distribution;
    distribution -- access1;
    distribution -- access2;
}
```

The graph can be output in the `dot -T<format> source -o <output file>` command line:

```
$ mkdir output
$ dot -Tpng chapter8_gv_1.gv -o output/chapter8_gv_1.png
```

The resultant graph can be viewed from the following output folder:

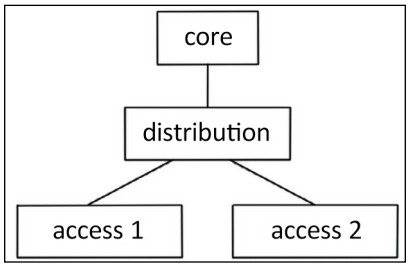


Figure 8.2: Graphviz undirected dot graph example

Just like *Chapter 7*, *Network Monitoring with Python – Part 1*, it might be easier to work in the Linux desktop window while working with these graphs so you can see the graphs right away.

Note that we can use a directional graph by specifying the graph as a digraph and using the arrow (->) sign to represent the edges. There are several attributes we can modify in the case of nodes and edges, such as the node shape, edge labels, etc. The same graph can be modified as follows in `chapter8_gv_2.gv`:

```
digraph my_network {
    node [shape=box];
    size = "50 30";
    core -> distribution [label="2x10G"];
    distribution -> access1 [label="1G"];
    distribution -> access2 [label="1G"];
}
```

We will output the file in PDF this time:

```
$ dot -Tpdf chapter8_gv_2.gv -o output/chapter8_gv_2.pdf
```

Take a look at the directional arrows in the new graph:

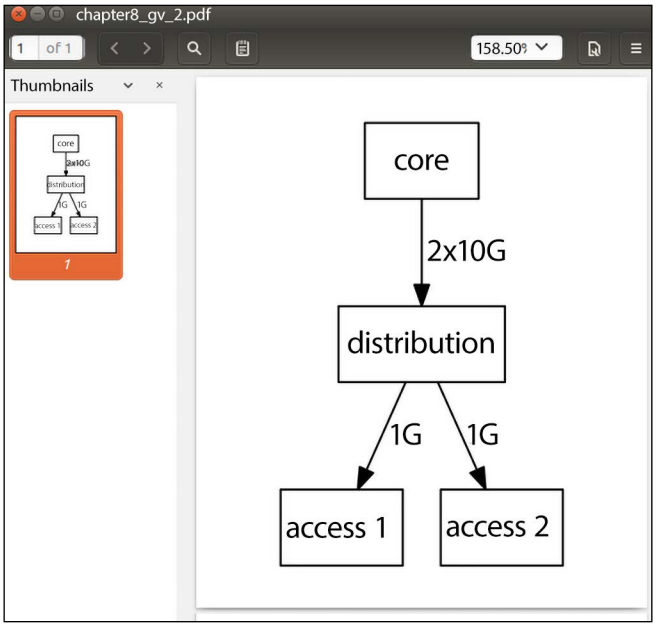


Figure 8.3: Network graph with directional arrows and line descriptions

Now let's take a look at the Python wrapper around Graphviz.

Python with Graphviz examples

We can reproduce the same topology graph as before using the Python Graphviz package and construct the same three-layer network topology:

```
>>> from graphviz import Digraph
>>> my_graph = Digraph(comment="My Network")
>>> my_graph.node("core")
>>> my_graph.node("distribution")
>>> my_graph.node("access1")
>>> my_graph.node("access2")
>>> my_graph.edge("core", "distribution")
>>> my_graph.edge("distribution", "access1")
>>> my_graph.edge("distribution", "access2")
```

The code produces what you would normally write in the DOT language but in a more Pythonic way. You can view the source of the graph before the graph generation:

```
>>> print(my_graph.source)
// My Network
digraph {
    core
    distribution
    access1
    access2
    core -> distribution
    distribution -> access1
    distribution -> access2
}
```

The graph can be rendered by the `render()` method. By default, the output format is PDF:

```
>>> my_graph.render("output/chapter8_gv_3.gv")
'output/chapter8_gv_3.gv.pdf'
```

The Python package wrapper closely mimics all the API options of Graphviz. You can find documentation about the options on the Graphviz Read the Docs website (`http://graphviz.readthedocs.io/en/latest/index.html`). You can also refer to the source code on GitHub for more information (`https://github.com/xflr6/graphviz`). We are now ready to use the tool to map out our network.

LLDP neighbor graphing

In this section, we will use the example of mapping out LLDP neighbors to illustrate a problem-solving pattern that has helped me over the years:

1. Modularize each task into smaller pieces, if possible. In our example, we can combine a few steps, but if we break them into smaller pieces, we will be able to reuse and improve them more easily.
2. Use an automation tool to interact with the network devices, but keep the more complex logic aside at the management station. For example, the router has provided an LLDP neighbor output that is a bit messy. In this case, we will stick with the working command and the output and use a Python script at the management station to parse out the output we need.
3. When given choices for the same task, pick the one that can be reused. In our example, we can use low-level Pexpect, Paramiko, or Ansible playbooks to query the routers. In my opinion, Ansible is a more reusable option, so that is what I have picked.

To get started, since LLDP is not enabled on the routers by default, we will need to configure them on the devices first. By now, we know we have a number of options to choose from; in this case, I chose the Ansible playbook with the `ios_config` module for the task. The `hosts` file consists of five routers:

```
$ cat hosts
[devices]
r1
r2
r3
r5-tor
r6-edge
[edge-devices]
r5-tor
r6-edge
```

Each host contains the corresponding names in the `host_vars` folder. We are showing `r1` as an example:

```
---
ansible_host: 192.168.2.218
ansible_user: cisco
ansible_ssh_pass: cisco
ansible_connection: network_cli
ansible_network_os: ios
ansbile_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

The `cisco_config_lldp.yml` playbook consists of one play with the `ios_lldp` module:

```
---
- name: Enable LLDP
  hosts: "devices"
  gather_facts: false
  connection: network_cli
  tasks:
    - name: enable LLDP service
      ios_lldp:
        state: present
```

```
    register: output
  - name: show output
    debug:
      var: output
```

The `ios_lldp` Ansible module is new in version 2.5 and later. Use the `ios_config` module if you are using an older version of Ansible.

Run the playbook to turn on `lldp`:

```
$ ansible-playbook -i hosts cisco_config_lldp.yml
<skip>
PLAY RECAP *********************************************************************
*************
r1                         : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r2                         : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r3                         : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r5-tor                     : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
r6-edge                    : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Since the default `lldp` advertise timer is 30 seconds, we should wait a bit for `lldp` advertisements to be exchanged between the devices. We can verify that LLDP is indeed active on the routers and the neighbors it has discovered:

```
r1#sh lldp
Global LLDP Information:
    Status: ACTIVE
    LLDP advertisements are sent every 30 seconds
    LLDP hold time advertised is 120 seconds
    LLDP interface reinitialisation delay is 2 seconds

r1#sh lldp neighbors
Capability codes:
```

```
    (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
    (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID           Local Intf     Hold-time  Capability      Port ID
r6.virl.info        Gi0/1          120        R               Gi0/1
r5.virl.info        Gi0/2          120        R               Gi0/1

Total entries displayed: 2
```

In older versions of CML, e.g. VIRL, or other lab software, you might see LLDP neighbors on the `G0/0` MGMT interfaces. What we really care about are the `G0/1` and `G0/2` interfaces that are directly connected to other peers. This information will come in handy as we prepare to parse the output and construct our topology graph.

Information retrieval

We can now use another Ansible playbook, namely `cisco_discover_lldp.yml`, to execute the LLDP command on the device and copy the output of each device to a `tmp` directory.

Let's create the `tmp` directory:

```
$ mkdir tmp
```

The playbook will have three tasks. The first task will execute the `show lldp` neighbor command on each of the devices, the second task will display the output, and the third task will copy the output to a text file in the output directory:

```
  tasks:
    - name: Query for LLDP Neighbors
      ios_command:
        commands: show lldp neighbors
      register: output
    - name: show output
      debug:
        var: output
    - name: copy output to file
      copy: content="{{ output.stdout_lines }}" dest="./tmp/{{ inventory_
hostname }}_lldp_output.txt"
```

After execution, the `./tmp` directory now contains all the routers' output (showing LLDP neighbors) in its own file:

```
$ ls -l tmp
total 20
-rw-rw-r-- 1 echou echou 413 Sep 18 10:44 r1_lldp_output.txt
-rw-rw-r-- 1 echou echou 413 Sep 18 10:44 r2_lldp_output.txt
-rw-rw-r-- 1 echou echou 413 Sep 18 10:44 r3_lldp_output.txt
-rw-rw-r-- 1 echou echou 484 Sep 18 10:44 r5-tor_lldp_output.txt
-rw-rw-r-- 1 echou echou 484 Sep 18 10:44 r6-edge_lldp_output.txt
```

`r1_lldp_output.txt`, as with the rest of the output files, contains the `output.stdout_lines` variable from the Ansible playbook for each device:

```
$ cat tmp/r1_lldp_output.txt
[["Capability codes:", "    (R) Router, (B) Bridge, (T) Telephone, (C)
DOCSIS Cable Device", "    (W) WLAN Access Point, (P) Repeater, (S)
Station, (O) Other", "", "Device ID           Local Intf     Hold-
time  Capability      Port ID", "r6.virl.info        Gi0/1          120
R              Gi0/1", "r5.virl.info        Gi0/2          120        R
Gi0/1", "", "Total entries displayed: 2"]]
```

So far, we have worked on retrieving information from network devices. Now we are ready to tie everything together with a Python script.

Python parser script

We can now use a Python script to parse the LLDP neighbor output from each device and construct a network topology graph from the results. The purpose is to automatically check the device to see whether any LLDP neighbors have disappeared due to link failure or other issues. Let's look at the `cisco_graph_lldp.py` file and see how that is done.

We start with the necessary imports of the packages: an empty list that we will populate with tuples of node relationships. We also know that `Gi0/0` on the devices is connected to the management network; therefore, we are only searching for `Gi0/[1234]` as our regular expression pattern in the `show LLDP neighbors` output:

```
import glob, re
from graphviz import Digraph, Source
pattern = re.compile('Gi0/[1234]')
device_lldp_neighbors = []
```

We will use the `glob.glob()` method to traverse the `./tmp` directory of all the files, parse out the device name, and find the neighbors that the device is connected to. There are some embedded print statements in the script that we can comment out for the final version; if the statements are uncommented, we can see the parsed result:

```
$ python cisco_graph_lldp.py
device: r6-edge
  neighbors: r2
  neighbors: r1
  neighbors: r3
device: r2
  neighbors: r5
  neighbors: r6
device: r3
  neighbors: r5
  neighbors: r6
device: r5-tor
  neighbors: r3
  neighbors: r1
  neighbors: r2
device: r1
  neighbors: r5
  neighbors: r6
```

The fully populated edge list contains tuples that consist of the device and its neighbors:

```
Edges: [('r6-edge', 'r2'), ('r6-edge', 'r1'), ('r6-edge', 'r3'), ('r2',
'r5'), ('r2', 'r6'), ('r3', 'r5'), ('r3', 'r6'), ('r5-tor', 'r3'), ('r5-
tor', 'r1'), ('r5-tor', 'r2'), ('r1', 'r5'), ('r1', 'r6')]
```

We can now construct the network topology graph using the Graphviz package. The most important part is the unpacking of the tuples that represent the edge relationship:

```
my_graph = Digraph("My_Network")
my_graph.edge("Client", "r6-edge")
my_graph.edge("r5-tor", "Server")
# construct the edge relationships
for neighbors in device_lldp_neighbors:
    node1, node2 = neighbors
    my_graph.edge(node1, node2)
```

If we were to print out the resulting source dot file, it would be an accurate representation of our network:

```
digraph My_Network {
        Client -> "r6-edge"
        "r5-tor" -> Server
        "r6-edge" -> r2
        "r6-edge" -> r1
        "r6-edge" -> r3
        r2 -> r5
        r2 -> r6
        r3 -> r5
        r3 -> r6
        "r5-tor" -> r3
        "r5-tor" -> r1
        "r5-tor" -> r2
        r1 -> r5
        r1 -> r6
}
```

Sometimes, it is confusing to see the same link twice; for example, the `r2` to `r5-tor` link appeared twice in the previous diagram for each of the directions of the link. As network engineers, we understand that sometimes a fault in the physical link will result in a unidirectional link, which we don't want to see.

If we were to graph the diagram as is, the placement of the nodes would be a bit funky. The placement of the nodes is auto-rendered. The following diagram illustrates the rendering in a default layout as well as the `neato` layout, namely, a digraph (`My_Network`, `engine='neato'`):

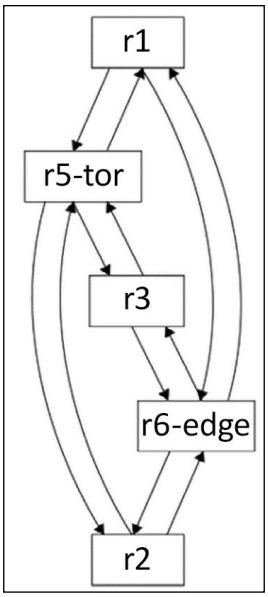


Figure 8.4: Topology graph 1

The `neato` layout represents an attempt to draw undirected graphs with even less hierarchy:

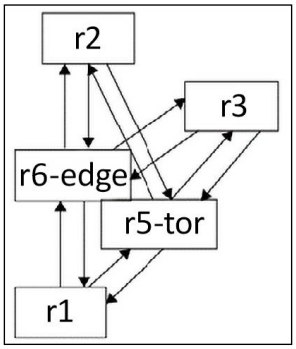


Figure 8.5: Topology graph 2

Sometimes, the default layout presented by the tool is just fine, especially if your goal is to detect faults instead of making them visually appealing. However, in this case, let's see how we can insert raw DOT language knobs into the source file. From research, we know that we can use the `rank` command to specify the level where some nodes can stay on the same level. However, there is no option presented in the Graphviz Python API. Luckily, the dot source file is just a string, which we can insert as raw dot comments using the `replace()` method with the following:

```
source = my_graph.source
original_text = "digraph My_Network {"
new_text = 'digraph My_Network {\n{rank=same Client "r6-edge"}\n{rank=same
r1 r2 r3}\n'
new_source = source.replace(original_text, new_text)
print(new_source)
new_graph = Source(new_source)
new_graph.render("output/chapter8_lldp_graph.gv")
```

The end result is a new source that we can render the final topology graph from:

```
digraph My_Network {
{rank=same Client "r6-edge"}
{rank=same r1 r2 r3}
        Client -> "r6-edge"
        "r5-tor" -> Server
        "r6-edge" -> r2
        "r6-edge" -> r1
        "r6-edge" -> r3
        r2 -> r5
        r2 -> r6
        r3 -> r5
        r3 -> r6
        "r5-tor" -> r3
        "r5-tor" -> r1
        "r5-tor" -> r2
        r1 -> r5
        r1 -> r6
}
```

The graph is now good to go with the correct hierarchy:

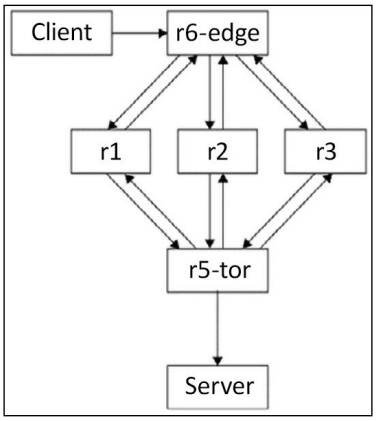


Figure 8.6: Topology graph 3

We have used the Python script to automatically retrieve network information from the devices and automatically graph the topology. It is quite a bit of work, but the reward is the consistency and the assurance that the graph always represents the latest state of the actual network. Let's follow up with some verification that our script can detect the latest state change of the network with the necessary graph.

Testing the playbook

We are now ready to incorporate a test to check whether the playbook can accurately depict the topology change when a link change happens.

We can test this by shutting down the `Gi0/1` and `Go0/2` interfaces on `r6-edge`:

```
r6#confi t
Enter configuration commands, one per line.  End with CNTL/Z.
r6(config)#int gig 0/1
r6(config-if)#shut
r6(config-if)#int gig 0/2
r6(config-if)#shut
r6(config-if)#end
r6#
```

When the LLDP neighbor passes the hold timer, they will disappear from the LLDP table on `r6-edge`:

```
r6#sh lldp neighbors
Capability codes:
    (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
    (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other
Device ID           Local Intf     Hold-time  Capability      Port ID
r1.virl.info        Gi0/0          120        R               Gi0/0
r2.virl.info        Gi0/0          120        R               Gi0/0
r3.virl.info        Gi0/0          120        R               Gi0/0
r5.virl.info        Gi0/0          120        R               Gi0/0
r3.virl.info        Gi0/3          120        R               Gi0/1

Device ID           Local Intf     Hold-time  Capability      Port ID
Total entries displayed: 5
```

If we execute the playbook and the Python script, the graph will automatically show `r6-edge` only connects to `r3` and we can start to troubleshoot why that is the case:

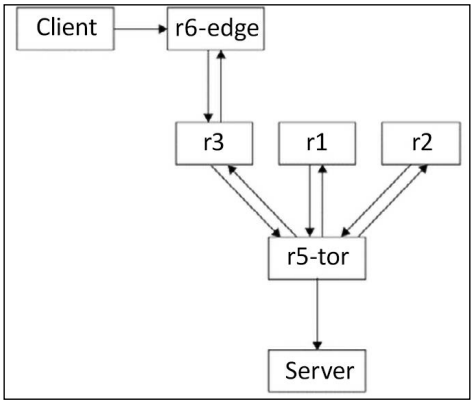


Figure 8.7: Topology graph 4

This is a relatively long example demonstrating multiple tools working together to solve a problem. We used the tools we have learned – Ansible and Python – to modularize and break tasks into reusable pieces.

We then used a new tool, namely, Graphviz, to help monitor the network for non-time series data, such as network topology relationships.

In the next section, we will change direction a bit and look into monitoring our network with network flows collected by our network equipment.

Flow-based monitoring

As mentioned in the chapter introduction, besides polling technology, such as SNMP, we can also use a push strategy, which allows the device to push network information toward the management station. NetFlow and its closely associated cousins, IPFIX and sFlow, are examples of such information pushed from the direction of the network device toward the management station. We can argue that the `push` method is more sustainable since the network device is inherently in charge of allocating the necessary resources to push the information. If the device CPU is busy, for example, it can skip the flow export process in favor of a more critical task such as routing packets.

A flow, as defined by IETF (`https://www.ietf.org/proceedings/39/slides/int/ip1394-background/tsld004.htm`), is a sequence of packets moving from an application sending something to the application receiving it. If we refer back to the OSI model, a flow is what constitutes a single unit of communication between two applications. Each flow comprises some packets; some flows have more packets (such as a video stream), while some have just a few (such as an HTTP request). If you think about flows for a minute, you'll notice that routers and switches might care about packets and frames, but the application and user usually care more about the network flows.

Flow-based monitoring usually refers to NetFlow, IPFIX, and sFlow:

- **NetFlow**: NetFlow v5 is a technology where the network device caches flow entries and aggregates packets by matching the set of tuples (source interface, source IP/port, destination IP/port, and so on). Once a flow is completed, the network device exports the flow characteristics, including total bytes and packet counts, to the management station.
- **IPFIX**: IPFIX is the proposed standard for structured streaming and is similar to NetFlow v9, also known as Flexible NetFlow. Essentially, it is a definable flow export, which allows the user to export nearly anything that the network device knows about. The flexibility often comes at the expense of simplicity compared to NetFlow v5. The configuration of IPFIX is more complex than the traditional NetFlow v5. Additional complexity makes it less ideal for introductory learning. However, once you are familiar with NetFlow v5, you can parse IPFIX as long as you match the template definition.
- **sFlow**: sFlow has no notion of a flow or packet aggregation by itself. It performs two types of sampling of packets. It randomly samples one out of "n" packets/applications and has a time-based sampling counter. It sends the information to the management station, and the station derives the network flow information by referring to the type of packet sample received along with the counters. As it doesn't perform any aggregation on the network device, you can argue that sFlow is more scalable than NetFlow and IPFIX.

The best way to learn about each one of these is probably to dive right into examples. Let's get into some of the flow-based examples in the following section.

NetFlow parsing with Python

We can use Python to parse the NetFlow datagram transported on the wire. This allows us to look at the NetFlow packet in detail and troubleshoot any NetFlow issues that are not working as expected.

First, let's generate traffic between the client and server across the lab network. We can use the built-in HTTP server module from Python to quickly launch a simple HTTP server on the VIRL host acting as the server. Open a new Terminal window to the server host and start the HTTP server; let's keep the window open:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

For Python 2, the module is named `SimpleHTTPServer`, for example, `python2 -m SimpleHTTPServer`.

In a separate Terminal window, `ssh` to the client. We can create a short `while` loop in a Python script to continuously send `HTTP GET` to the web server:

```
cisco@Client:~$ cat http_get.py
import requests
import time
while True:
    r = requests.get("http://10.0.0.5:8000")
    print(r.text)
    time.sleep(5)
```

The client should get a very plain HTML page every 5 seconds:

```
cisco@Client:~$ python3 http_get.py
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<skip>
</body>
</html>
```

If we look back to the server Terminal window, we should also see the requests continuously coming in from the client every 5 seconds:

```
cisco@Server:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.0.9 - - [02/Oct/2019 00:55:57] "GET / HTTP/1.1" 200 -
10.0.0.9 - - [02/Oct/2019 00:56:02] "GET / HTTP/1.1" 200 -
10.0.0.9 - - [02/Oct/2019 00:56:07] "GET / HTTP/1.1" 200 –
```

The traffic from the client to the server traverses through the network devices, and we can export NetFlow from any of the devices in between. Since `r6-edge` is the first hop for the client host, we will have this router export NetFlow to the management host at port `9995`.

In this example, we use only one device for demonstration; therefore, we manually configure it with the necessary commands. In the next section, when we enable NetFlow on all the devices, we will use an Ansible playbook to configure all the routers at once.

The following configurations are necessary for exporting NetFlow on Cisco IOS devices:

```
!
ip flow-export version 5
ip flow-export destination 192.168.2.126 9995 vrf Mgmt-intf
!
interface GigabitEthernet0/4
 description to Client
 ip address 10.0.0.10 255.255.255.252
 ip flow ingress
 ip flow egress
<skip>
```

Next, let's look at the Python parser script that helps us separate the different network flow fields we received from network devices.

Python Socket and Struct

The script, `netFlow_v5_parser.py`, was modified from Brian Rak's blog post at `http://blog.devicenull.org/2013/09/04/python-netflow-v5-parser.html`. The modification was mainly for Python 3 compatibility and parsing additional NetFlow version 5 fields. The reason we chose NetFlow v5 instead of NetFlow v9 is that v9 is more complex and uses templates to map out the fields, making it more difficult to learn in an introductory session. However, since NetFlow version 9 is an extended format of the original NetFlow version 5, all the concepts we introduced in this section apply to it.

Because NetFlow packets are represented in bytes over the wire, we will use the Python struct module included in the standard library to convert bytes into native Python data types.

You can find more information about the two modules at `https://docs.python.org/3.10/library/socket.html` and `https://docs.python.org/3.10/library/struct.html`.

In the script, we will start by using the `socket` module to bind and listen for the UDP datagrams. With `socket.AF_INET`, we intend on listening to the IPv4 address sockets; with `socket.SOCK_DGRAM`, we specify that we'll see the UDP datagram:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('0.0.0.0', 9995))
```

We will start a loop and retrieve information off the wire 1,500 bytes at a time:

```
while True:
    buf, addr = sock.recvfrom(1500)
```

The following line is where we begin to deconstruct or unpack the packet. The first argument of `!HH` specifies the network's big-endian byte order with the exclamation point (big-endian) as well as the format of the C type (`H = 2` byte unsigned short integer):

```
(version, count) = struct.unpack('!HH',buf[0:4])
```

The first 4 bytes include the version and the number of flows exported in this packet. If you do not remember the NetFlow version 5 header off the top of your head (that was a joke, by the way; I only read the header when I want to fall asleep quickly), here is a glance:

Table B-3 Version 5 Header Format

Bytes	Contents	Description
0-1	version	NetFlow export format version number
2-3	count	Number of flows exported in this packet (1-30)
4-7	SysUptime	Current time in milliseconds since the export device booted
8-11	unix_secs	Current count of seconds since 0000 UTC 1970
12-15	unix_nsecs	Residual nanoseconds since 0000 UTC 1970
16-19	flow-sequence	Sequence counter of total flows seen
20	engine_type	Type of flow-switching engine
21	engine_id	Slot number of the flow-switching engine
22-23	sampling_interval	First two bits hold the sampling mode; remaining 14 bits hold value of sampling interval

Figure 8.8: NetFlow v5 header (source: http://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html#wp1006108)

The rest of the header can be parsed accordingly, depending on the byte location and data type. Python allows us to unpack several header items in a single line:

```
    (sys_uptime, unix_secs, unix_nsecs, flow_sequence) = struct.
unpack('!IIII', buf[4:20])
    (engine_type, engine_id, sampling_interval) = struct.unpack('!BBH',
buf[20:24])
```

The `while` loop that follows will fill the `nfdata` dictionary with the flow record that unpacks the source address and port, destination address and port, packet count, and byte count and print the information out on the screen:

```
    nfdata = {}
    for i in range(0, count):
        try:
            base = SIZE_OF_HEADER+(i*SIZE_OF_RECORD)
            data = struct.unpack('!IIIIHH',buf[base+16:base+36])
            input_int, output_int = struct.unpack('!HH',
buf[base+12:base+16])
            nfdata[i] = {}
            nfdata[i]['saddr'] = inet_ntoa(buf[base+0:base+4])
            nfdata[i]['daddr'] = inet_ntoa(buf[base+4:base+8])
            nfdata[i]['pcount'] = data[0]
            nfdata[i]['bcount'] = data[1]
            nfdata[i]['stime'] = data[2]
            nfdata[i]['etime'] = data[3]
            nfdata[i]['sport'] = data[4]
            nfdata[i]['dport'] = data[5]
            print(i, " {0}:{1} -> {2}:{3} {4} packts {5} bytes".format(
                nfdata[i]['saddr'],
                nfdata[i]['sport'],
                nfdata[i]['daddr'],
                nfdata[i]['dport'],
                nfdata[i]['pcount'],
                nfdata[i]['bcount']),
                )
```

The output of the script allows you to visualize the header as well as the flow content at a glance. In the following output, we can see both BGP control packets (TCP port 179) as well as HTTP traffic (TCP port 8000) on r6-edge:

```
$ python3 netFlow_v5_parser.py
Headers:
NetFlow Version: 5
Flow Count: 6
System Uptime: 116262790
Epoch Time in seconds: 1569974960
Epoch Time in nanoseconds: 306899412
Sequence counter of total flow: 24930
0  192.168.0.3:44779 -> 192.168.0.2:179 1 packts 59 bytes
1  192.168.0.3:44779 -> 192.168.0.2:179 1 packts 59 bytes
2  192.168.0.4:179 -> 192.168.0.5:30624 2 packts 99 bytes
3  172.16.1.123:0 -> 172.16.1.222:771 1 packts 176 bytes
4  192.168.0.2:179 -> 192.168.0.5:59660 2 packts 99 bytes
5  192.168.0.1:179 -> 192.168.0.5:29975 2 packts 99 bytes
**********
Headers:
NetFlow Version: 5
Flow Count: 15
System Uptime: 116284791
Epoch Time in seconds: 1569974982
Epoch Time in nanoseconds: 307891182
Sequence counter of total flow: 24936
0  10.0.0.9:35676 -> 10.0.0.5:8000 6 packts 463 bytes
1  10.0.0.9:35676 -> 10.0.0.5:8000 6 packts 463 bytes
<skip>
11  10.0.0.9:35680 -> 10.0.0.5:8000 6 packts 463 bytes
12  10.0.0.9:35680 -> 10.0.0.5:8000 6 packts 463 bytes
13  10.0.0.5:8000 -> 10.0.0.9:35680 5 packts 973 bytes
14  10.0.0.5:8000 -> 10.0.0.9:35680 5 packts 973 bytes
```

Note that, in NetFlow version 5, the record size is fixed at 48 bytes; therefore, the loop and script are relatively straightforward.

However, in the case of NetFlow version 9 or IPFIX, after the header, there is a template FlowSet (`http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html`) that specifies the field count, field type, and field length. This allows the collector to parse the data without knowing the data format in advance. We will need to build additional logic in the Python script for NetFlow version 9.

By parsing the NetFlow data in a script, we gained a solid understanding of the fields, but this is very tedious and hard to scale. As you may have guessed, other tools save us the problem of parsing NetFlow records one by one. Let's look at one such tool, called **ntop**, in the coming section.

ntop traffic monitoring

Like the PySNMP script in *Chapter 7, Network Monitoring with Python – Part 1,* and the NetFlow parser script in this chapter, we can use Python scripts to handle low-level tasks on the wire. However, there are tools such as Cacti, which is an all-in-one open source package that includes data collection (pollers), data storage (RRDs), and a web frontend for visualization. These tools can save you a lot of work by packing the frequently used features and software in one package.

In the case of NetFlow, there are several open source and commercial NetFlow collectors we can choose from. If we do a quick search for the top N open source NetFlow analyzers, we will see several comparison studies for different tools.

Each one has its strengths and weaknesses; which one to use is a matter of preference, platform, and appetite for customization. I would recommend choosing a tool that would support both v5 and v9, and potentially sFlow. A secondary consideration would be whether the tool is written in a language that we can understand; I would imagine having Python extensibility would be a nice thing.

Two of the open source NetFlow tools I like and have used before are NfSen (with NFDUMP as the backend collector) and `ntop` (or `ntopng`). Between the two of them, `ntop` is the better-known traffic analyzer; it runs on both Windows and Linux platforms and integrates well with Python. Therefore, let's use `ntop` as an example in this section.

Similar to Cacti, ntop is an all-in-one tool. I recommend installing ntop on a separate host than the management station in production or a container on the management station.

The installation of our Ubuntu host is straightforward:

```
$ sudo apt-get install ntop
```

The installation process will prompt for the necessary interface for listening and setting the administrator password. By default, the `ntop` web interface listens on port `3000`, while the probe listens on UDP port `5556`. On the network device, we need to specify the location of the NetFlow exporter:

```
!
ip flow-export version 5
ip flow-export destination 192.168.2.126 5556 vrf Mgmt-intf
!
```

By default, IOSv creates a VRF called `Mgmt-intf` and places `Gi0/0` under VRF.

We will also need to specify the direction of traffic exports, such as `ingress` or `egress`, under the interface configuration:

```
!
interface GigabitEthernet0/0
...
ip flow ingress
ip flow egress
...
```

For your reference, I have included the Ansible playbook, `cisco_config_netflow.yml`, to configure the lab device for the NetFlow export.

`r5-tor` and `r6-edge` have two more interfaces than `r1`, `r2`, and `r3`; therefore, there is an additional playbook to enable the additional interfaces for them.

Execute the playbook and make sure the changes were applied properly on the devices:

```
$ ansible-playbook -i hosts cisco_config_netflow.yml
TASK [configure netflow export station] ********************************
*************************************
changed: [r2]
changed: [r1]
changed: [r3]
changed: [r5-tor]
changed: [r6-edge]
TASK [configure flow export on Gi0/0] **********************************
*************************************
```

```
ok: [r1]
ok: [r3]
ok: [r2]
ok: [r5-tor]
ok: [r6-edge]
<skip>
```

It is always a good idea to verify the device configuration after the playbook is run, so let's spot-check on r2:

```
r2#sh run
!
interface GigabitEthernet0/0
 description OOB Management
 vrf forwarding Mgmt-intf
 ip address 192.168.2.126 255.255.255.0
 ip flow ingress
 ip flow egress
<skip>
!
ip flow-export version 5
ip flow-export destination 192.168.2.126 5556 vrf Mgmt-intf
!
```

Once everything is set up, you can check the **ntop** web interface for local IP traffic:

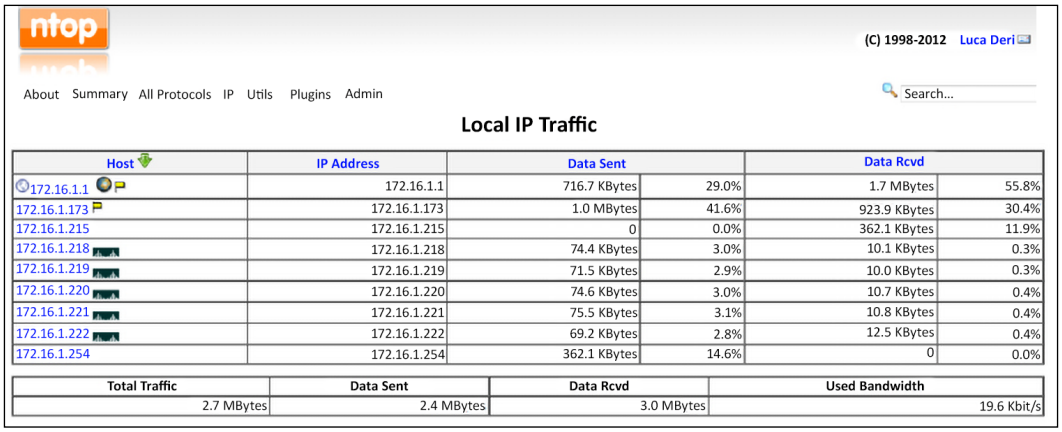


Figure 8.9: ntop local IP traffic

One of the most often used features of ntop is using it to look at the Top Talkers graph:

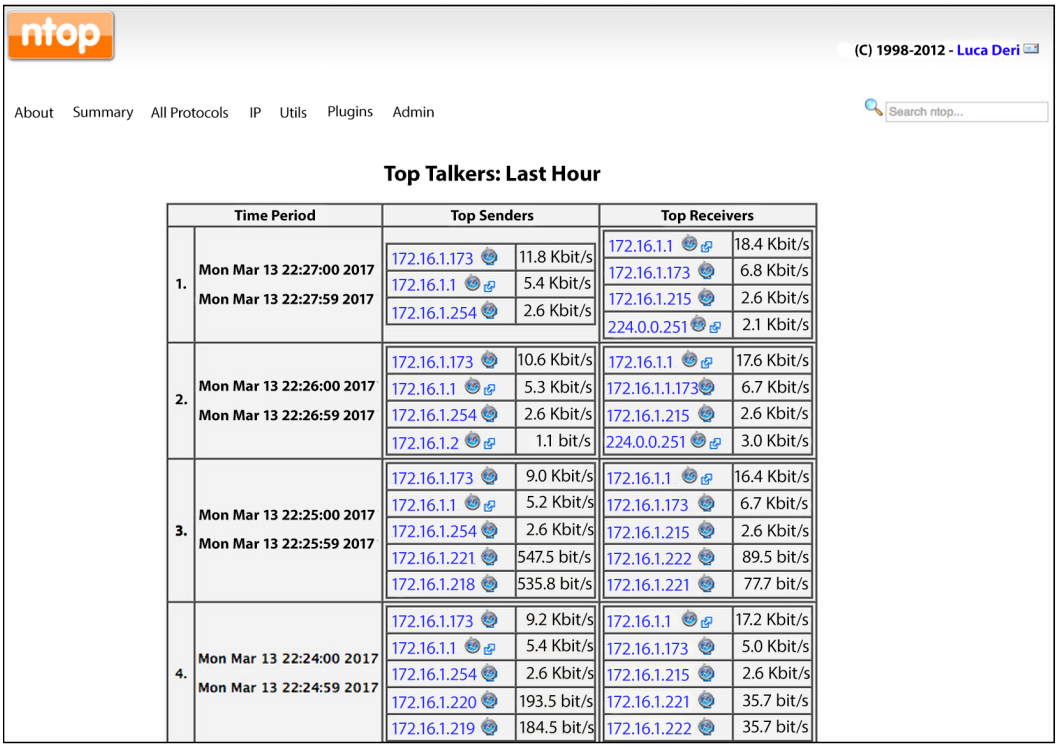


Figure 8.10: ntop top talkers

The ntop reporting engine is written in C; it is fast and efficient, but the need to have adequate knowledge of C to do something as simple as changing the web frontend does not fit the modern agile development mindset.

After a few false starts with Perl in the mid-2000s, the good folks at ntop finally settled on embedding Python as an extensible scripting engine. Let's take a look.

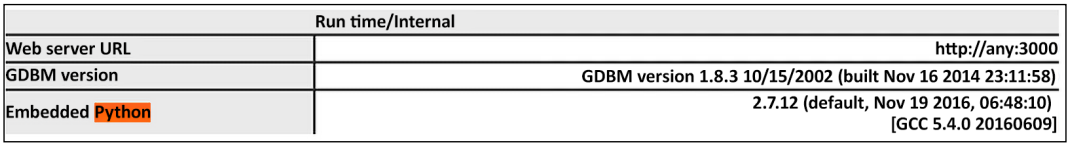
Python extension for ntop

We can use Python to extend ntop through the ntop web server. The ntop web server can execute Python scripts. At a high level, the scripts will involve the following:

- Methods to access the state of ntop
- The Python CGI module to process forms and URL parameters
- Making templates that generate dynamic HTML pages

- Each Python script can read from `stdin` and print out `stdout/stderr`. The `stdout` script is the returned HTTP page.

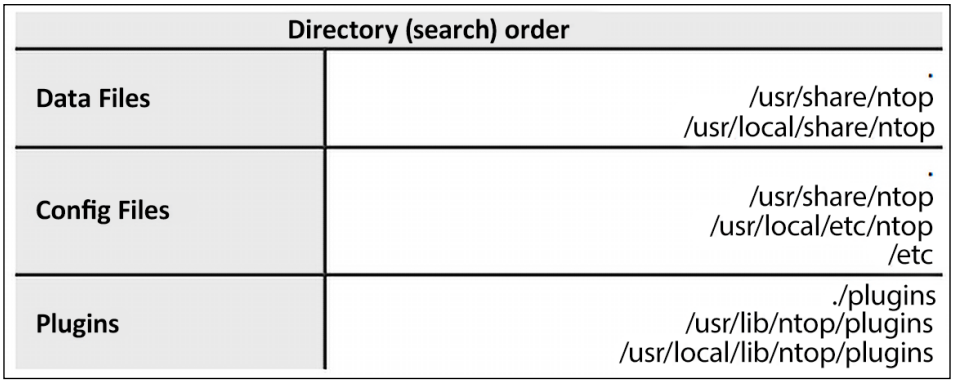
Several resources come in handy with Python integration. Under the web interface, you can click on **About | Show Configuration** to see the Python interpreter version as well as the directory for your Python script:



Run time/Internal	
Web server URL	http://any:3000
GDBM version	GDBM version 1.8.3 10/15/2002 (built Nov 16 2014 23:11:58)
Embedded Python	2.7.12 (default, Nov 19 2016, 06:48:10) [GCC 5.4.0 20160609]

Figure 8.11: Python version

You can also check the various directories where the Python script should reside:



Directory (search) order	
Data Files	. /usr/share/ntop /usr/local/share/ntop
Config Files	. /usr/share/ntop /usr/local/etc/ntop /etc
Plugins	./plugins /usr/lib/ntop/plugins /usr/local/lib/ntop/plugins

Figure 8.12: Plugin directories

Under **About | Online Documentation | Python ntop Engine**, there are links for the Python API as well as the tutorial:

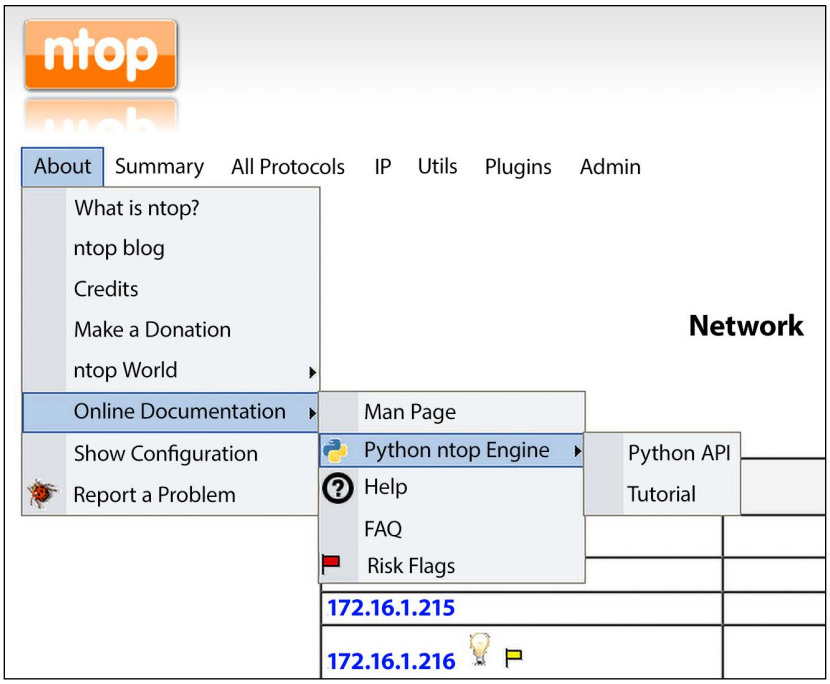


Figure 8.13: Python ntop documentation

As mentioned, the `ntop` web server directly executes the Python script placed under the designated directory:

```
$ pwd
/usr/share/ntop/python
```

We will place our first script, namely, `chapter8_ntop_1.py`, in the directory. The Python `CGI` module processes forms and parses URL parameters:

```
# Import modules for CGI handling
import cgi, cgitb
import ntop
# Parse URL cgitb.enable();
```

`ntop` implements three Python modules; each one of them has a specific purpose:

- **ntop**: This module interacts with the `ntop` engine.
- **Host**: This module drills down into a specific host's information.
- **Interfaces**: This module represents the information about the localhost interfaces.

In our script, we will use the ntop module to retrieve the ntop engine information as well as using the `sendString()` method to send the HTML body text:

```
form = cgi.FieldStorage();
name = form.getvalue('Name', default="Eric")
version = ntop.version()
os = ntop.os()
uptime = ntop.uptime()
ntop.printHTMLHeader('Mastering Python Networking', 1, 0) ntop.
sendString("Hello, "+ name +"<br>")
ntop.sendString("Ntop Information: %s %s %s" % (version, os, uptime))
ntop.printHTMLFooter()
```

We will execute the Python script using `http://<ip>:3000/python/<script name>`. Here is the result of our `chapter8_ntop_1.py` script:

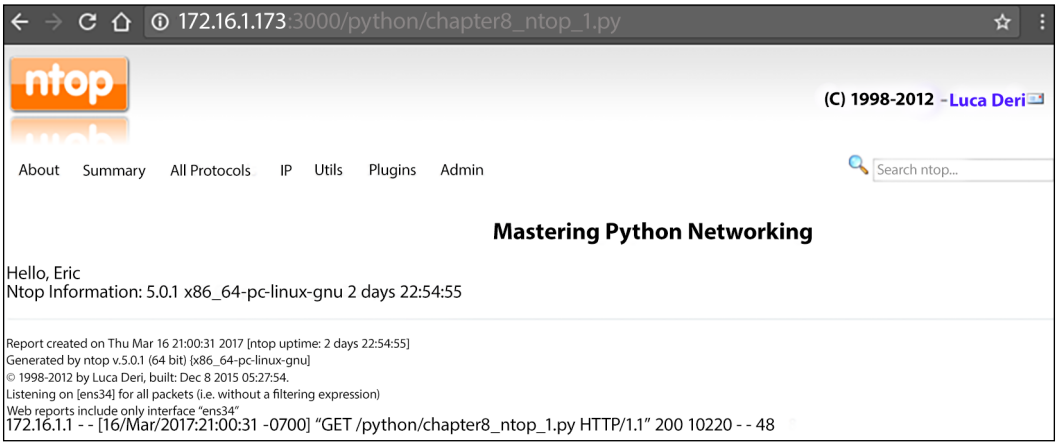


Figure 8.14: ntop script result

We can look at another example that interacts with the interface module, `chapter8_ntop_2.py`. We will use the API to iterate through the interfaces:

```
import ntop, interface, json
ifnames = []
try:
for i in range(interface.numInterfaces()):
    ifnames.append(interface.name(i))
except Exception as inst:
    print(type(inst)) # the exception instance
```

```
    print(inst.args) # arguments stored in .args
    print(inst) #   str _ allows args to printed directly
<skip>
```

The resulting page will display the ntop interfaces:

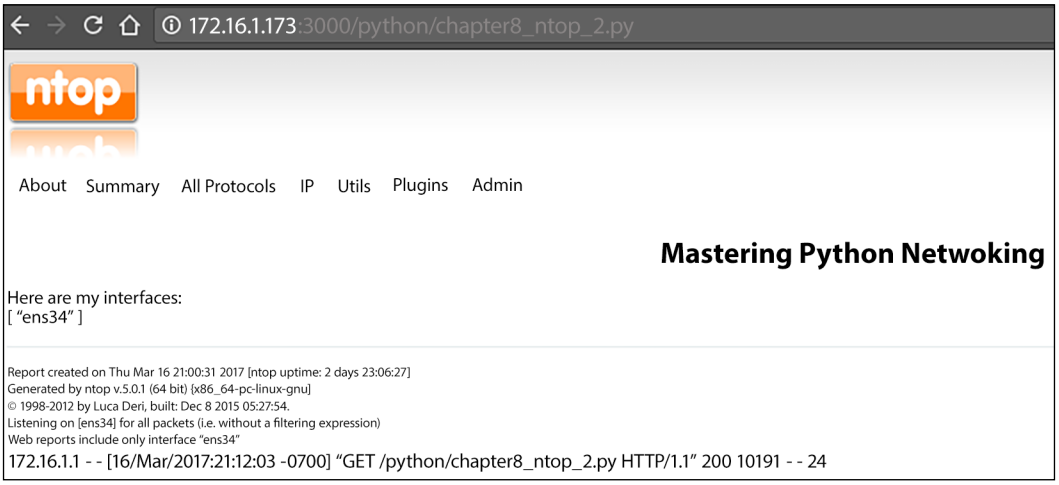


Figure 8.15: ntop interface information

Besides the community version, ntop also offers a few commercial products that you can choose from. With the active open source community, commercial backing, and Python extensibility, ntop is a good choice for your NetFlow monitoring needs.

Next, let's take a look at NetFlow's cousin: sFlow.

sFlow

sFlow, which stands for sampled flow, was originally developed by InMon (`http://www.inmon.com`) and later standardized by way of RFC. The current version is v5. Many in the industry believe the primary advantage of sFlow is its scalability.

sFlow uses random [one in `n`] packet flow samples along with the polling interval of counter samples to estimate the traffic; this is less CPU-intensive than NetFlow for network devices. sFlow's statistical sampling is integrated with the hardware and provides real-time, raw exports.

For scalability and competitive reasons, sFlow is generally preferred over NetFlow for newer vendors, such as Arista Networks, Vyatta, and A10 Networks. While Cisco supports sFlow on its Nexus line of products, sFlow is generally "not" supported on Cisco platforms.

SFlowtool and sFlow-RT with Python

Unfortunately, at this point, sFlow is something our CML lab devices do not support (not even with the NX-OSv virtual switches). You can use a Cisco Nexus 3000 switch or other vendor switches, such as Arista, that support sFlow. Another good option for the lab is to use an Arista vEOS virtual instance. I have access to a Cisco Nexus 3048 switch running 7.0 (3), which I will use for this section as the sFlow exporter.

The configuration of Cisco Nexus 3000 for sFlow is straightforward:

```
Nexus-2# sh run | i sflow feature sflow
sflow max-sampled-size 256
sflow counter-poll-interval 10
sflow collector-ip 192.168.199.185 vrf management sflow agent-ip
192.168.199.148
sflow data-source interface Ethernet1/48
```

The easiest way to ingest sFlow is to use `sflowtool`. For installation instructions, refer to the documentation at `http://blog.sflow.com/2011/12/sflowtool.html`:

```
$ wget http://www.inmon.com/bin/sflowtool-3.22.tar.gz
$ tar -xvzf sflowtool-3.22.tar.gz
$ cd sflowtool-3.22/
$ ./configure
$ make
$ sudo make install
```

I am using an older version of `sFlowtool` in the lab. The newer versions work the same.

After the installation, you can launch `sflowtool` and look at the datagram Nexus 3048 is sending on the standard output:

```
$ sflowtool
startDatagram =================================
datagramSourceIP 192.168.199.148
datagramSize 88
unixSecondsUTC 1489727283
datagramVersion 5
agentSubId 100
agent 192.168.199.148
packetSequenceNo 5250248
```

```
sysUpTime 4017060520
samplesInPacket 1
startSample ----------------------
sampleType_tag 0:4 sampleType COUNTERSSAMPLE sampleSequenceNo 2503508
sourceId 2:1
counterBlock_tag 0:1001
5s_cpu 0.00
1m_cpu 21.00
5m_cpu 20.80
total_memory_bytes 3997478912
free_memory_bytes 1083838464 endSample ----------------------
endDatagram =================================
```

There are a number of good usage examples on the sflowtool GitHub repository (`https://github.com/sflow/sflowtool`); one of them is to use a script to receive the `sflowtool` input and parse the output. We can use a Python script for this purpose. In the `chapter8_sflowtool_1.py` example, we will use `sys.stdin.readline` to receive the input and use a regular expression search to print out only the lines containing the word agent when we see the sFlow packets:

```
#!/usr/bin/env python3
import sys, re
for line in iter(sys.stdin.readline, ''):
    if re.search('agent ', line):
         print(line.strip())
```

The script can be piped to `sflowtool`:

```
$ sflowtool | python3 chapter8_sflowtool_1.py
agent 192.168.199.148
agent 192.168.199.148
```

There are a number of other useful output examples, such as `tcpdump`, output as NetFlow version 5 records, and a compact line-by-line output. This makes `sflowtool` flexible for different monitoring environments.

ntop supports sFlow, which means you can directly export your sFlow to the ntop collector. If your collector is only NetFlow-aware, you can use the `-c` option for the `sflowtool` output in the NetFlow version 5 format:

```
$ sflowtool --help
```

```
...
tcpdump output:
-t - (output in binary tcpdump(1) format)
-r file - (read binary tcpdump(1) format)
-x - (remove all IPV4 content)
-z pad - (extend tcpdump pkthdr with this many zeros
e.g. try -z 8 for tcpdump on Red Hat Linux 6.2)
NetFlow output:
-c hostname_or_IP - (netflow collector host)
-d port - (netflow collector UDP port)
-e - (netflow collector peer_as (default = origin_as))
-s - (disable scaling of netflow output by sampling rate)
-S - spoof source of netflow packets to input agent IP
```

Alternatively, you can also use InMon's sFlow-RT (`http://www.sflow-rt.com/index.php`) as your sFlow analytics engine. What sets sFlow-RT apart from an operator perspective is its vast RESTful API, which can be customized to support your use cases. You can also easily retrieve the metrics from the API. You can take a look at its extensive API reference at: `http://www.sflow-rt.com/reference.php`.

Note that sFlow-RT requires Java to run the following:

```
$ sudo apt-get install default-jre
$ java -version
openjdk version "1.8.0_121"
OpenJDK Runtime Environment (build 1.8.0_121-8u121-b13-0ubuntu1.16.04.2-
b13)
OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
```

Once installed, downloading and running sFlow-RT is straightforward (`https://sflow-rt.com/download.php`):

```
$ wget http://www.inmon.com/products/sFlow-RT/sflow-rt.tar.gz
$ tar -xvzf sflow-rt.tar.gz
$ cd sflow-rt/
$ ./start.sh
2017-03-17T09:35:01-0700 INFO: Listening, sFlow port 6343
2017-03-17T09:35:02-0700 INFO: Listening, HTTP port 8008
```

We can point the web browser to HTTP port `8008` and verify the installation:

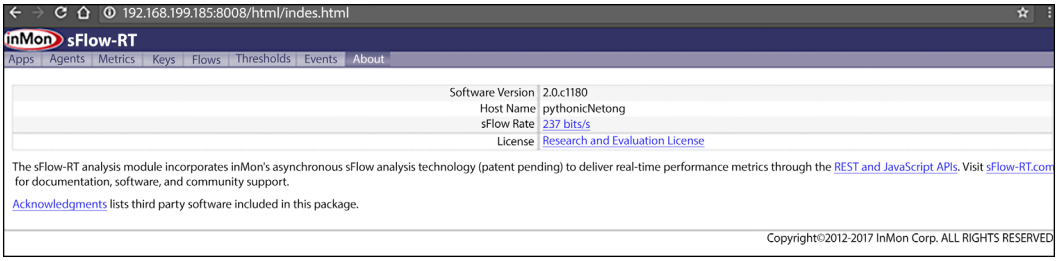


Figure 8.16: sFlow-RT version

As soon as sFlow-RT receives any sFlow packets, the agents and other metrics will appear:

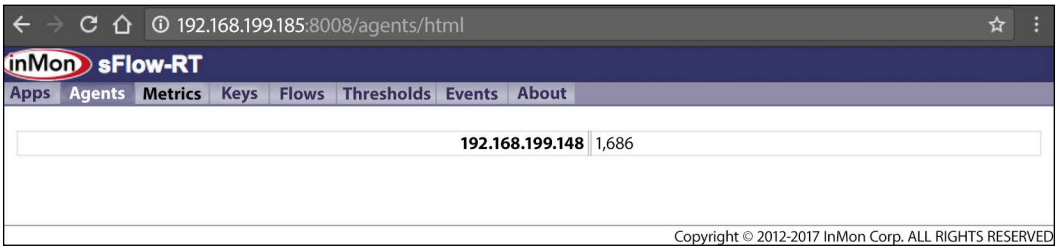


Figure 8.17: sFlow-RT agent IP

Here are two examples of using Python requests to retrieve information from sFlow-RT's REST API:

```
>>> import requests
>>> r = requests.get("http://192.168.199.185:8008/version")
>>> r.text '2.0-r1180'
>>> r = requests.get("http://192.168.199.185:8008/agents/json")
>>> r.text
'{"192.168.199.148": {n "sFlowDatagramsLost": 0,n
"sFlowDatagramSource": ["192.168.199.148"],n "firstSeen": 2195541,n
"sFlowFlowDuplicateSamples": 0,n "sFlowDatagramsReceived": 441,n
"sFlowCounterDatasources": 2,n "sFlowFlowOutOfOrderSamples": 0,n
"sFlowFlowSamples": 0,n "sFlowDatagramsOutOfOrder": 0,n "uptime":
4060470520,n "sFlowCounterDuplicateSamples": 0,n "lastSeen":
3631,n "sFlowDatagramsDuplicates": 0,n "sFlowFlowDrops": 0,n
"sFlowFlowLostSamples": 0,n "sFlowCounterSamples": 438,n
"sFlowCounterLostSamples": 0,n "sFlowFlowDatasources": 0,n
"sFlowCounterOutOfOrderSamples": 0n}}'
```

Consult the reference documentation for additional REST endpoints available for your needs.

In this section, we looked at sFlow-based monitoring examples both as a standalone tool as well as part of the integration with `ntop`. sFlow is one of the newer flow formats that intends to address scalability issues faced with traditional `netflow` formats, and it's worth us spending some time to see whether it is the right tool for the network monitoring tasks at hand. We are close to the end of this chapter, so let's look at what we have covered.

Summary

In this chapter, we looked at additional ways in which we can utilize Python to enhance our network monitoring efforts. We began using Python's Graphviz package to create network topology graphs with real-time LLDP information reported by the network devices. This allows us to effortlessly show the current network topology, as well as to easily notice any link failures.

Next, we used Python to parse NetFlow version 5 packets to enhance our understanding and troubleshooting of NetFlow. We also looked at how to use ntop and Python to extend ntop for NetFlow monitoring. sFlow is an alternative packet sampling technology. We used `sflowtool` and sFlow-RT to interpret sFlow results.

In *Chapter 9*, *Building Network Web Services with Python*, we will explore how to use the Python web framework Flask to build network web services.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



9

Building Network Web Services with Python

In the previous chapters, we were a consumer of the APIs provided by others. In *Chapter 3*, *APIs and Intent-Driven Networking*, we saw that we can use an `HTTP POST` request to NX-API at the `http://<your device ip>/ins` URL with the `CLI` command embedded in the `HTTP POST` body to execute commands remotely on the Cisco Nexus device; the device then returns the command execution output in its HTTP response return. In *Chapter 8*, *Network Monitoring with Python – Part 2*, we used the `HTTP GET` method for our sFlow-RT at `http://<your host ip>:8008/version` with an empty body to retrieve the version of the sFlow-RT software. These request-response exchanges are examples of RESTful web services.

According to Wikipedia (`https://en.wikipedia.org/wiki/Representational_state_transfer`):

> *"Representational state transfer (REST) or RESTful web services is one way of providing interoperability between computer systems on the internet. REST-compliant web services allow requesting systems to access and manipulate the textual representation of web resources using a uniform and predefined set of stateless operations."*

As noted, the use of RESTful web services using the HTTP protocol is only one of many methods of information exchange on the web; other forms of web services also exist. However, it is the most commonly used web service today, with the associated `GET`, `POST`, `PUT`, and `DELETE` verbs as a predefined way of exchanging information.



If you are wondering about HTTPS versus HTTP, for our discussion, we are treating HTTPS as a secure extension of HTTP (`https://en.wikipedia.org/wiki/HTTPS`) and the same underlying protocol as a RESTful API.

On the provider side, one of the advantages of providing RESTful services to users is the ability to hide internal operations from the user. For example, in the case of sFlow-RT, if we were to log in to the device to see the version of the software installed instead of using its RESTful API, we would need more in-depth knowledge of the tool to know where to check. However, by providing the resources as a URL, the API provider abstracts the version-checking operations from the requester, making the operation much simpler. The abstraction also provides a layer of security as it can open up the endpoints only as needed.

As the master of our network universe, RESTful web services provide many notable benefits that we can enjoy, such as the following:

- You can abstract the requester from learning about the internals of the network operations. For example, we can provide a web service to query the switch version without the requester knowing the exact CLI command or the switch API.
- We can consolidate and customize operations that uniquely fit our network needs, such as a resource to upgrade all our top-of-rack switches.
- We can provide better security by only exposing operations as needed. For example, we can provide read-only URLs (`GET`) to core network devices and read-write URLs (`GET`/`POST`/`PUT`/`DELETE`) to access-level switches.

In this chapter, we will use one of the most popular Python web frameworks, **Flask**, to create our RESTful web service for our network. In this chapter, we will learn about the following:

- Comparing Python web frameworks
- Introduction to Flask
- Operations involving static network content
- Operations involving dynamic network operations
- Authentication and authorization
- Running our web app in containers

Let's start by looking at the available Python web frameworks and why we chose Flask.

Comparing Python web frameworks

Python is known for its great many web frameworks. There is a running joke in the Python community about whether you can ever work as a full-time Python developer without working with any Python web frameworks. There are several Python web developer conferences, including DjangoCon US (`https://djangocon.us/`), DjangoCon EU (`https://djangocon.eu/`), FlaskCon (`https://flaskcon.com/`), Python Web Conference (`https://pythonwebconf.com/`), and many local meetups. Each of the conferences attracts hundreds of attendees every year. Did I mention Python has a thriving web development community?

If you sort the Python web frameworks at `https://hotframeworks.com/languages/python`, you can see that there is no shortage of choices when it comes to Python and web frameworks:

Framework	Score
Django	93
Flask	86
Tornado	71
FastAPI	66
AIOHTTP	64
Bottle	63
Pyramid	62
web.py	60
Sanic	58
web2py	58
CherryPy	55
Falcon	55
Grok	46
Zope	45
TurboGears	43
Quart	42
Masonite	39
Tipfy	32

Figure 9.1: Python Web Framework Rankings
(source: https://hotframeworks.com/languages/python)

In the most recent 2021 Python Developer Survey, Flask slightly edged Django as the most popular web framework:

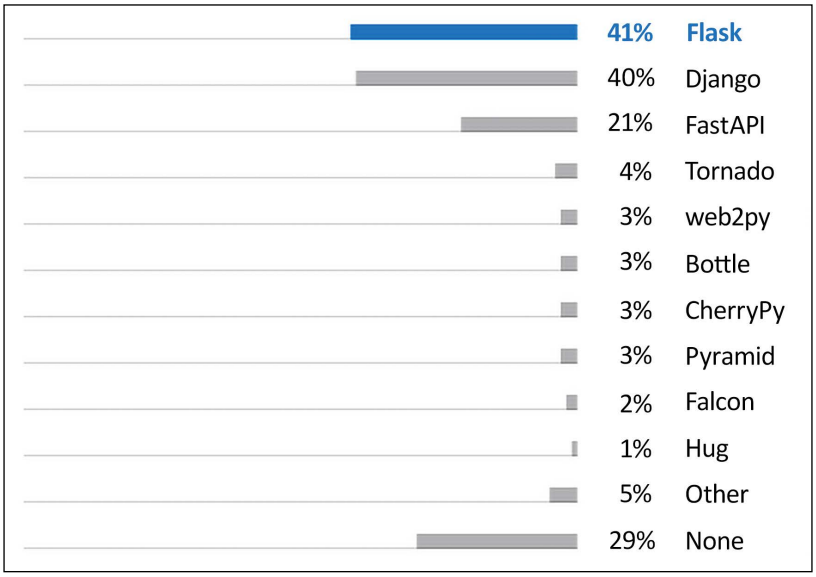


Figure 9.2: Python Developer Survey 2021 (source: https://lp.jetbrains.com/python-developers-survey-2021/)

With so many options to choose from, which framework should we pick? Trying all the frameworks one by one would be time-consuming. The question of which web framework is better is also a passionate topic among web developers. If you ask this question on any of the forums, such as Quora, or search on Reddit, get ready for some highly opinionated answers and heated debates.



Speaking of Quora and Reddit, here's an interesting fact: both Quora and Reddit were written in Python. Reddit uses Pylons (`https://www.reddit.com/wiki/faq#wiki_so_what_python_framework_do_you_use.3F`), while Quora started with Pylons but replaced a portion of the framework with its in-house code (`https://www.quora.com/What-languages-and-frameworks-are-used-to-code-Quora`).

Of course, I am biased toward programming languages (Python!) and web frameworks (Flask and Django!). In this section, I hope to convey my reasoning behind choosing one over the other for any particular project. Let's pick the top two frameworks from the preceding HotFrameworks list and compare them:

- **Django**: The self-proclaimed "web framework for perfectionists with deadlines" is a high-level Python web framework that encourages rapid development and clean, pragmatic design (`https://www.djangoproject.com/`). It is a large framework with pre-built code that provides an administrative panel and built-in content management.
- **Flask**: This is a microframework for Python and is based on Werkzeug, Jinja2, and other applications (`https://palletsprojects.com/p/flask/`). By being a microframework, Flask intends on keeping the core small and easy to extend when needed. The "micro" in microframework does not mean that Flask lacks functionality, nor does it mean it cannot work in a production environment.

I use Django for some of the larger projects while using Flask for quick prototypes. The Django framework has a strong opinion on how things should be done; any deviation from it would sometimes leave the user feeling that they are "fighting with the framework." For example, if you look at the Django database documentation (`https://docs.djangoproject.com/en/4.0/ref/databases/`) you will notice that the framework supports several different SQL databases. However, they are all variants of a SQL database such as MySQL, PostgreSQL, SQLite, and others.

What if we want to use a NoSQL database such as MongoDB or CouchDB? It might be possible, but it could leave us in our own hands because there is no official support from Django. Being an opinionated framework is certainly not a bad thing. It is just a matter of opinion (no pun intended).

Keeping the core code small and extending it when needed is very appealing when we need something simple and fast. The initial example in the documentation to get Flask up and running consists of only six lines of code and is easy to understand, even if you don't have any prior experience. Since Flask is built with extensions in mind, writing our extensions, such as a decorator, is relatively easy. Even though it is a microframework, the Flask core still includes the necessary components, such as a development server, debugger, integration with unit tests, RESTful request dispatching, and more, to get you started quickly.

As you can see, Django and Flask are the two most popular Python web frameworks by almost all measures. We can't really go wrong picking either as our starting point. The popularity that comes with both frameworks means both have extensive community contribution and support, and can quickly develop modern features.

For the ease of deployment, I feel Flask is an ideal choice for us when it comes to building network web services to start with.

Flask and lab setup

In this chapter, we will continue to use a virtual environment to isolate the Python environment and dependencies. We can start a new virtual environment, or we can continue to use the existing virtual environment that we have been using. My preference is to start a new virtual environment. I will call mine `ch09-venv`:

```
$ python3 -m venv ch09-venv
$ source ch09-venv/bin/activate
```

In this chapter, we will install quite a few Python packages. To make life easier, I have included a `requirements.txt` file on this book's GitHub repository; we can use it to install all the necessary packages (remember to activate your virtual environment). You should see packages being downloaded and successfully installed during the installation process:

```
(ch09-venv) $ cat requirements.txt
click==8.1.3
Flask==2.2.2
itsdangerous==2.1.2
Jinja2==3.1.2
MarkupSafe==2.1.1
Werkzeug==2.2.2
…

(ch09-venv) $ pip install -r requirements.txt
```

For our network topology, we will use the `2_DC_Topology` that we have been using in the first few chapters, as shown here:

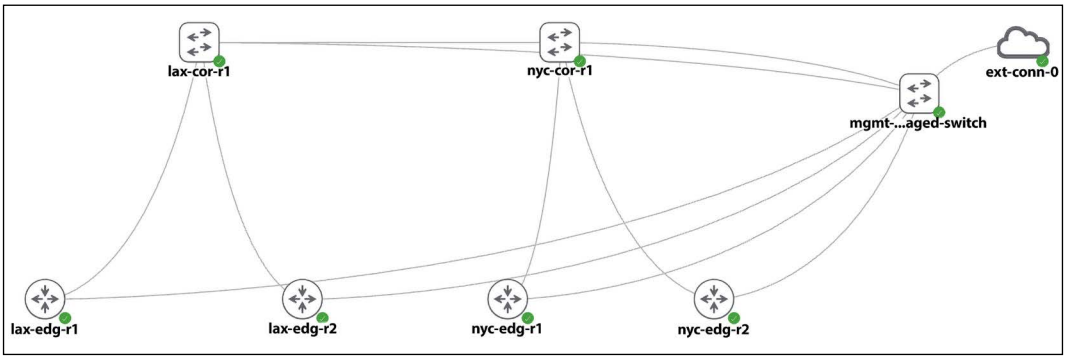


Figure 9.3: Lab Topology

Let's take a look at Flask in the next section.

Please note that, from here on out, I will assume that you will always execute from the virtual environment and that you have installed the necessary packages in the `requirements.txt` file.

Introduction to Flask

Like most popular open source projects, Flask has very good documentation, which is available at `https://flask.palletsprojects.com/en/2.0.x/`. If you'd like to dig deeper into Flask, the project documentation would be a great place to start.

I highly recommend Miguel Grinberg's work (`https://blog.miguelgrinberg.com/`) related to Flask. His blog, book, and video training have taught me a lot about Flask. Miguel's class, *Building Web APIs with Flask*, inspired me to build my first Flask-based API, and inspired the writing of this chapter. You can take a look at his published code on GitHub: `https://github.com/miguelgrinberg/`.

Flask versions

As of the time of writing, the latest version of Flask is version 2.2.2. Flask version 2.0.0 was released in May 2021 from version 1.1.4. There were several big changes introduced in the release, thus the big number jump in release numbers. Some of the big changes are listed below:

- Flask 2.0 officially dropped the support for Python 2 and Python 3.5.
- Python 3 type hinting is supported.
- The HTTP method decorator was introduced.

These changes probably mean very little at this point because we are just getting started with Flask. For now, just keep in mind the big version change if we were searching for answers and examples. If possible, look for examples based on version 2 and beyond.

Flask examples

Our first Flask application is contained in one single file, `chapter9_1.py`:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_networkers():
    return 'Hello Networkers!'
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

This is a simple design pattern for Flask apps. We create an instance of the `Flask` class with the first argument as the name of the application's module package. In this case, we used a single module that can be started as an application; later on, we will see how we can import it as a package. We then use the route decorator to tell Flask which URL should be handled by the `hello_networkers()` function; in this case, we indicated the root path. We end the file with the usual name scope, checking when the script is run by itself (`https://docs.python.org/3.10/library/__main__.html`).

We also add the `host` and `debug` options, which allow more verbose output and allow us to listen on all the host interfaces. We can run this application using the development server:

```
(ch09-venv) $ python chapter9_1.py
 * Serving Flask app 'chapter9_1'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.2.126:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 218-520-639
```



If you receive an error of **Address already in use** from the development server, change the port Flask Development Server runs on via the `port=xxxx` option, `https://flask.palletsprojects.com/en/2.2.x/server/`.

Now that we have a server running, let's test the server response with an HTTP client.

The HTTPie client

We have already installed `HTTPie` (`https://httpie.org/`) as part of the installation from the `requirements.txt` file. This book is printed in black and white, so the example does not show color highlighting, but in your installation, you can see `HTTPie` has better syntax highlighting for HTTP transactions. It also has a more intuitive command-line interaction with the RESTful HTTP server.

We can use it to test our first Flask application (more examples on `HTTPie` to follow). We will start a second terminal window on the management host, activate the virtual environment, and type the following in:

```
(ch09-venv) $ http http://192.168.2.126:5000
HTTP/1.1 200 OK
Connection: close
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Wed, 21 Sep 2022 02:54:54 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

Hello Networkers!
```

As a comparison, if we are using `curl`, we will need to use the `-i` switch to achieve the same output: `curl -i http://192.168.2.126:5000`.

We will use `HTTPie` as our client for this chapter; it is worth taking a minute or two to take a look at its usage. We will use the free website HTTPBin (`https://httpbin.org/`) to demonstrate the use of `HTTPie`. The usage of `HTTPie` follows this simple pattern:

```
$ http [flags] [METHOD] URL [ITEM]
```

Following the preceding pattern, a `GET` request is very straightforward, as we have seen with our Flask development server:

```
(ch09-venv) $ http GET https://httpbin.org/user-agent
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 35
Content-Type: application/json
Date: Wed, 21 Sep 2022 02:56:07 GMT
Server: gunicorn/19.9.0

{
    "user-agent": "HTTPie/3.2.1"
}
```

JSON is the default implicit content type for `HTTPie`. If your HTTP body contains just strings, no other operation is needed. If you need to apply non-string JSON fields, use `:=` or other documented special characters. In the following example, we want the `"married"` variable to be a `Boolean` instead of a `string`:

```
(ch09-venv) $ http POST https://httpbin.org/post name=eric twitter=at_
ericchou married:=true
…
Content-Type: application/json
…

{…
    "headers": {
        "Accept": "application/json, */*;q=0.5",
        …
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/3.2.1",
      …
    },
    "json": {
        "married": true,
        "name": "eric",
        "twitter": "at_ericchou"
    },
    "url": "https://httpbin.org/post"
}
```

As you can see, `HTTPie` is a big improvement from the traditional curl syntax and makes testing the REST API a breeze.



More usage examples are available at `https://httpie.io/docs/cli/usage`.

Getting back to our Flask program, a large part of API building is based on the flow of URL routing. Let's look deeper at the `app.route()` decorator.

URL routing

We added two additional functions and paired them up with the appropriate `app.route()` route in `chapter9_2.py`:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
    return 'You are at index()'
@app.route('/routers/')
def routers():
    return 'You are at routers()'
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

The result is that different endpoints are passed to different functions. We can verify this with two `http` requests:

```
# Server side
$ python chapter9_2.py
<skip>
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
# client side
$ http http://192.168.2.126:5000
<skip>
You are at index()
$ http http://192.168.2.126:5000/routers/
<skip>
You are at routers()
```

As the requests are made from the client side, the server screen will see the requests coming in:

```
(ch09-venv) $ python chapter9_2.py
<skip>
192.168.2.126 - - [20/Sep/2022 20:00:27] "GET / HTTP/1.1" 200 -
192.168.2.126 - - [20/Sep/2022 20:01:05] "GET /routers/ HTTP/1.1" 200 -
```

As we can see, the different endpoints correspond to different functions; whatever was returned from the function is what the server returns to the requester. Of course, the routing will be pretty limited if we have to keep it static all the time. There are ways to pass dynamic variables from the URL to Flask; we will look at an example of this in the next section.

URL variables

We can pass dynamic variables to the URL, as seen in the `chapter9_3.py` examples:

```
<skip>
@app.route('/routers/<hostname>')
def router(hostname):
    return 'You are at %s' % hostname

@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return 'You are at %s interface %d' % (hostname, interface_number)
<skip>
```

In the two functions, we pass in dynamic information such as the hostname and interface number at the time when the client is making the request. Note that, in the `/routers/<hostname>` URL, we pass the `<hostname>` variable as a string; in `/routers/<hostname>/interface/<int:interface_number>` we specify the `int` variable should only be an integer. Let's run the example and make some requests:

```
# Server Side
(ch09-venv) $ python chapter9_3.py
(ch09-venv) # Client Side
$ http http://192.168.2.126:5000/routers/host1
HTTP/1.0 200 OK
<skip>
You are at host1
(venv) $ http http://192.168.2.126:5000/routers/host1/interface/1
HTTP/1.0 200 OK
<skip>
You are at host1 interface 1
```

If the `int` variable is NOT an integer, an error will be thrown:

```
(venv) $ http http://192.168.2.126:5000/routers/host1/interface/one
HTTP/1.0 404 NOT FOUND
<skip>
<!doctype html>
<html lang=en>
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL
manually please check your spelling and try again.</p>
```

The converter includes integers, float, and path (it accepts slashes).

Besides matching static routes with dynamic variables, we can also generate URLs upon application launch. This is very useful when we do not know the endpoint variable in advance or if the endpoint is based on other conditions, such as the values queried from a database. Let's take a look at an example of this.

URL generation

In `chapter9_4.py`, we wanted to dynamically create a URL during application launch in the form of `/<hostname>/list_interfaces`, where the hostname could be `r1`, `r2`, or `r3`. We already know we can statically configure three routes and three corresponding functions, but let's see how we can do that upon application launch:

```
from flask import Flask, url_for
app = Flask(__name__)
@app.route('/<hostname>/list_interfaces')
def device(hostname):
    if hostname in routers:
        return 'Listing interfaces for %s' % hostname
    else:
        return 'Invalid hostname'
routers = ['r1', 'r2', 'r3']
for router in routers:
    with app.test_request_context():
        print(url_for('device', hostname=router))
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

Upon its execution, we will have a few nice, logical URLs that loop around the routers list without statically defining each:

```
# server side
$ python chapter9_4.py
<skip>
/r1/list_interfaces
/r2/list_interfaces
/r3/list_interfaces
# client side
(venv) $ http http://192.168.2.126:5000/r1/list_interfaces
<skip>
Listing interfaces for r1
(venv) $ http http://192.168.2.126:5000/r2/list_interfaces
<skip>
Listing interfaces for r2
# bad request
(venv) $ http http://192.168.2.126:5000/r1000/list_interfaces
<skip>
Invalid hostname
```

For now, you can think of `app.text_request_context()` as a dummy request object necessary for demonstration purposes. If you are interested in the local context, feel free to look at `https://werkzeug.palletsprojects.com/en/2.2.x/local/`. The dynamic generation of URL endpoints greatly simplifies our code, saves time, and makes the code easier to read.

The jsonify return

Another time-saver in Flask is the `jsonify()` return, which wraps `json.dumps()` and turns the JSON output into a response object with `application/json` as the content type in the HTTP header. We can tweak the `chapter9_3.py` script a bit, as illustrated in `chapter9_5.py`:

```
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/routers/<hostname>/interface/<int:interface_number>')
def interface(hostname, interface_number):
    return jsonify(name=hostname, interface=interface_number)
if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=True)
```

With a few lines, the return result is now a `JSON` object with the appropriate header:

```
$ http http://192.168.2.126:5000/routers/r1/interface/1
HTTP/1.0 200 OK
Content-Length: 38
Content-Type: application/json
Date: Tue, 08 Oct 2019 21:48:51 GMT
Server: Werkzeug/0.16.0 Python/3.6.8
{
    "interface": 1,
    "name": "r1"
}
```

Combine all the Flask features we have learned so far, and we are now ready to build an API for our network.

Network resource API

When we have network devices in production, each of the devices will have a certain state and information that you would like to keep in a persistent location so that you can easily retrieve them later on. This is often done in terms of storing data in a database. We saw many examples of such information storage in the monitoring chapters.

However, we would not normally give other non-network administrative users who might want this information direct access to the database; nor would they want to learn all the complex SQL query language. For these cases, we can leverage Flask and the **Flask-SQLAlchemy** extension of Flask to give them the necessary information via a network API.

You can learn more about Flask-SQLAlchemy at `https://flask-sqlalchemy.palletsprojects.com/en/2.x/`.

Flask-SQLAlchemy

SQLAlchemy and the Flask-SQLAlchemy extension are database abstraction and object-relational mappers, respectively. It's a fancy way to use the `Python` object for a database. To make things simple, we will use SQLite as the database, which is a flat file that acts as a self-contained SQL database. We will look at the content of `chapter9_db_1.py` as an example of using Flask-SQLAlchemy to create a network database and insert a few table entries into the database. This is a multiple-step process, and we will look at the steps in this section.

To begin, we will create a Flask application and load the configuration for SQLAlchemy, such as the database path and name, then create the `SQLAlchemy` object by passing the application to it:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
# Create Flask application, load configuration, and create
# the SQLAlchemy object
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
```

We can then create a device `database` object and its associated primary key and various columns:

```
# This is the database model object
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(120), index=True)
    vendor = db.Column(db.String(40))
    def __init__(self, hostname, vendor):
        self.hostname = hostname
        self.vendor = vendor
    def __repr__(self):
        return '<Device %r>' % self.hostname
```

We can invoke the `database` object, create entries, and insert them into the database table. Keep in mind that anything we add to the session needs to be committed to the database in order to be permanent:

```
if __name__ == '__main__':
    db.create_all()
    r1 = Device('lax-dc1-core1', 'Juniper')
    r2 = Device('sfo-dc1-core1', 'Cisco')
    db.session.add(r1)
    db.session.add(r2)
    db.session.commit()
```

We will run the Python script and check for the existence of the database file:

```
$ python chapter9_db_1.py
$ ls -l network.db
-rw-r--r-- 1 echou echou 28672 Sep 21 10:43 network.db
```

We can use the interactive prompt to check the database table entries:

```
>>> from flask import Flask
>>> from flask_sqlalchemy import SQLAlchemy
>>> app = Flask(__name__)
>>> app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
>>> db = SQLAlchemy(app)
>>> from chapter9_db_1 import Device
>>> Device.query.all()
[<Device 'lax-dc1-core1'>, <Device 'sfo-dc1-core1'>]
>>> Device.query.filter_by(hostname='sfo-dc1-core1')
<flask_sqlalchemy.BaseQuery object at 0x7f09544a0e80>
>>> Device.query.filter_by(hostname='sfo-dc1-core1').first()
<Device 'sfo-dc1-core1'>
```

We can also create new entries in the same manner:

```
>>> r3 = Device('lax-dc1-core2', 'Juniper')
>>> db.session.add(r3)
>>> db.session.commit()
>>> Device.query.filter_by(hostname='lax-dc1-core2').first()
<Device 'lax-dc1-core2'>
```

Let's go ahead and delete the `network.db` file so it does not conflict with our other examples using the same `db` name:

```
$ rm network.db
```

Now we are ready to move on to build our network content API.

The network content API

Before we dive into the code of building our API, let's take a moment to think about the API structure we will create. Planning for an API is usually more an art than a science; it really depends on your situation and preference. What I suggest in this section is, by no means, the only way, but for now, stay with me for the purposes of getting started.

Recall that, in our diagram, we have four Cisco IOSv devices. Let's pretend that two of them, `lax-edg-r1` and `lax-edg-r2`, are in the network role of the spine. The other two devices, `nyc-edg-r1` and `nyc-edg-r2`, are in our network service as leaves. These are arbitrary choices and can be modified later on, but the point is that we want to serve data about our network devices and expose them via an API.

To make things simple, we will create two APIs, a devices group API and a single-device API:

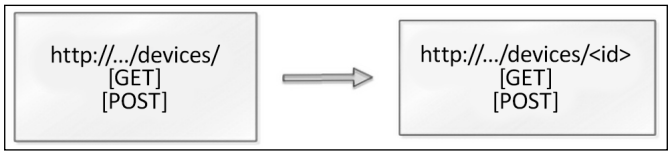


Figure 9.4: Network Content API

The first API will be our `http://192.168.2.126/devices/` endpoint, which supports two methods: `GET` and `POST`. The `GET` request will return the current list of devices, while the `POST` request with the proper JSON body will create the device. Of course, you can choose different endpoints for creation and querying, but in this design, we choose to differentiate the two by the HTTP methods.

The second API will be specific to our device in the form of `http://192.168.2.126/devices/<device id>`. The API with the `GET` request will show the details of the device that we have entered into the database.

The `PUT` request will modify the entry with the update. Note that we use `PUT` instead of `POST`. This is typical of HTTP API usage; when we need to modify an existing entry, we will use `PUT` instead of `POST`.

At this point, you should have a good idea about what your API will look like. To better visualize the final result, I will jump ahead and show the result quickly before we take a look at the code. If you want to follow the example, feel free to launch `chapter9_6.py` as the Flask server.

A `POST` request to the `/devices/` API will allow you to create an entry. In this case, I would like to create our network device with attributes such as hostname, loopback IP, management IP, role, vendor, and the operating system it runs on:

```
$ http POST http://192.168.2.126:5000/devices/ 'hostname'='lax-edg-r1'
'loopback'='192.168.0.10' 'mgmt_ip'='192.168.2.51' 'role'='spine'
'vendor'='Cisco' 'os'='15.8'
HTTP/1.1 201 CREATED
Connection: close
Content-Length: 3
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:01:33 GMT
Location: http://192.168.2.126:5000/devices/1
Server: Werkzeug/2.2.2 Python/3.10.4

{}
```

I can repeat the preceding step for the three additional devices:

```
$ http POST http://192.168.2.126:5000/devices/ 'hostname'='lax-edg-r2'
'loopback'='192.168.0.11' 'mgmt_ip'='192.168.2.52' 'role'='spine'
'vendor'='Cisco' 'os'='15.8'
$ http POST http://192.168.2.126:5000/devices/ 'hostname'='nyc-edg-r1'
'loopback'='192.168.0.12' 'mgmt_ip'='192.168.2.61' 'role'='leaf'
'vendor'='Cisco' 'os'='15.8'
$ http POST http://192.168.2.126:5000/devices/ 'hostname'='nyc-edg-r2'
'loopback'='192.168.0.13' 'mgmt_ip'='192.168.2.62' 'role'='leaf'
'vendor'='Cisco' 'os'='15.8'
```

If we use the same API endpoint with the `GET` request, we will be able to see the list of network devices that we created:

```
$ http GET http://192.168.2.126:5000/devices/
HTTP/1.1 200 OK
Connection: close
Content-Length: 193
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:07:16 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "device": [
        "http://192.168.2.126:5000/devices/1",
        "http://192.168.2.126:5000/devices/2",
        "http://192.168.2.126:5000/devices/3",
        "http://192.168.2.126:5000/devices/4"
    ]
}
```

Similarly, using the `GET` request for `/devices/<id>` will return specific information related to the device:

```
$ http GET http://192.168.2.126:5000/devices/1
HTTP/1.1 200 OK
Connection: close
Content-Length: 199
Content-Type: application/json
```

```
Date: Wed, 21 Sep 2022 18:07:50 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "hostname": "lax-edg-r1",
    "loopback": "192.168.0.10",
    "mgmt_ip": "192.168.2.51",
    "os": "15.8",
    "role": "spine",
    "self_url": "http://192.168.2.126:5000/devices/1",
    "vendor": "Cisco"
}
```

Let's pretend we have downgraded the `lax-edg-r1` operating system from `15.6` to `14.6`. We can use the `PUT` request to update the device record:

```
$ http PUT http://192.168.2.126:5000/devices/1 'hostname'='lax-edg-r1'
'loopback'='192.168.0.10' 'mgmt_ip'='192.168.2.51' 'role'='spine'
'vendor'='Cisco' 'os'='14.6'
HTTP/1.1 200 OK
# Verification
$ http GET http://192.168.2.126:5000/devices/1
HTTP/1.1 200 OK
Connection: close
Content-Length: 199
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:10:37 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "hostname": "lax-edg-r1",
    "loopback": "192.168.0.10",
    "mgmt_ip": "192.168.2.51",
    "os": "14.6",
    "role": "spine",
    "self_url": "http://192.168.2.126:5000/devices/1",
    "vendor": "Cisco"
}
```

Now, let's look at the code in `chapter9_6.py` that created the preceding APIs. What's cool, in my opinion, is that all of these APIs were done in a single file, including the database interaction. Later on, when we outgrow the APIs at hand, we can always separate the components, such as having a separate file for the database class.

The devices API

The `chapter9_6.py` file starts with the necessary imports. Note that the following request import is the `request` object from the client and not the `requests` package that we were using in the previous chapters:

```
from flask import Flask, url_for, jsonify, request
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
```

We declared a `database` object with `id` as the primary key and string fields for `hostname`, `loopback`, `mgmt_ip`, `role`, `vendor`, and `os`:

```
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(64), unique=True)
    loopback = db.Column(db.String(120), unique=True)
    mgmt_ip = db.Column(db.String(120), unique=True)
    role = db.Column(db.String(64))
    vendor = db.Column(db.String(64))
    os = db.Column(db.String(64))
```

The `get_url()` function under the `Device` class returns a URL from the `url_for()` function. Note that the `get_device()` function that's called is not defined just yet under the `/devices/<int:id>` route:

```
    def get_url(self):
        return url_for('get_device', id=self.id, _external=True)
```

The `export_data()` and `import_data()` functions are mirror images of each other. One is used to get the information from the database to the user (`export_data()`) when we use the `GET` method. The other is to get information from the user to the database (`import_data()`) when we use the `POST` or `PUT` method:

```
    def export_data(self):
        return {
            'self_url': self.get_url(),
            'hostname': self.hostname,
            'loopback': self.loopback,
            'mgmt_ip': self.mgmt_ip,
            'role': self.role,
            'vendor': self.vendor,
            'os': self.os
        }
    def import_data(self, data):
        try:
            self.hostname = data['hostname']
            self.loopback = data['loopback']
            self.mgmt_ip = data['mgmt_ip']
            self.role = data['role']
            self.vendor = data['vendor']
            self.os = data['os']
        except KeyError as e:
            raise ValidationError('Invalid device: missing ' + e.args[0])
        return self
```

With the `database` object in place and the import and export functions created, the URL dispatch is straightforward for device operations. The `GET` request will return a list of devices by querying all the entries in the `devices` table and returning the URL of each entry. The `POST` method will use the `import_data()` function with the global `request` object as the input. It will then add the device and commit the information to the database:

```
@app.route('/devices/', methods=['GET'])
def get_devices():
    return jsonify({'device': [device.get_url()
                               for device in Device.query.all()]})
@app.route('/devices/', methods=['POST'])
```

```
def new_device():
    device = Device()
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({}), 201, {'Location': device.get_url()}
```

If you look at the `POST` method, the returned body is an empty JSON body, with the status code `201` (created), as well as extra headers:

```
HTTP/1.0 201 CREATED
Content-Length: 2
Content-Type: application/json Date: ...
Location: http://192.168.2.126:5000/devices/4
Server: Werkzeug/2.2.2 Python/3.10.4
```

Let's look at the API that queries and returns information for individual devices.

The device ID API

The route for individual devices specifies that the ID should be an integer, which can act as our first line of defense against a bad request. The two endpoints follow the same design pattern as our `/devices/` endpoint, where we use the same `import` and `export` functions:

```
@app.route('/devices/<int:id>', methods=['GET'])
def get_device(id):
    return jsonify(Device.query.get_or_404(id).export_data())
@app.route('/devices/<int:id>', methods=['PUT'])
def edit_device(id):
    device = Device.query.get_or_404(id)
    device.import_data(request.json)
    db.session.add(device)
    db.session.commit()
    return jsonify({})
```

Note that the `query_or_404()` method provides a convenient way of returning `404 (not found)` if the database query returns negative for the ID passed in. This is a pretty elegant way of providing a quick check on the database query.

Finally, the last part of the code creates the database table and starts the Flask development server:

```
if __name__ == '__main__':
    db.create_all()
    app.run(host='0.0.0.0', debug=True)
```

This is one of the longer Python scripts in this book, so we took more time to explain it in detail. The script provides a way to illustrate how we can utilize the database in the backend to keep track of the network devices and only expose them to the external world as APIs using Flask.

In the next section, we will take a look at how to use the API to perform asynchronous tasks on either individual devices or a group of devices.

Network dynamic operations

Our API can now provide static information about the network; anything we can store in the database can be returned to the requester. It would be great if we could interact with our network directly, such as a query for device information or to push configuration changes to the device.

We will start this process by leveraging a script we have already seen in *Chapter 2*, *Low-Level Network Device Interactions*, for interacting with a device via Pexpect. We will modify the script slightly into a function we can repeatedly use in `chapter9_pexpect_1.py`:

```
import pexpect
def show_version(device, prompt, ip, username, password):
    device_prompt = prompt
    child = pexpect.spawn('telnet ' + ip)
    child.expect('Username:')
    child.sendline(username)
    child.expect('Password:')
    child.sendline(password)
    child.expect(device_prompt)
    child.sendline('show version | i V')
    child.expect(device_prompt)
    result = child.before
    child.sendline('exit')
    return device, result
```

We can test the new function via the interactive prompt:

```
>>> from chapter9_pexpect_1 import show_version
>>> print(show_version('lax-edg-r1', 'lax-edg-r1#', '192.168.2.51',
'cisco', 'cisco'))
('lax-edg-r1', b'show version | i V\r\nCisco IOS Software, IOSv Software
(VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)\r\
nProcessor board ID 98U40DKV403INHIULHYHB\r\n')
```

Make sure our Pexpect script works before proceeding. The following code assumes that we have entered the necessary database information from the previous section.

We can add a new API for querying the device version in `chapter9_7.py`:

```
from chapter9_pexpect_1 import show_version
<skip>
@app.route('/devices/<int:id>/version', methods=['GET'])
def get_device_version(id):
    device = Device.query.get_or_404(id)
    hostname = device.hostname
    ip = device.mgmt_ip
    prompt = hostname+"#"
    result = show_version(hostname, prompt, ip, 'cisco', 'cisco')
    return jsonify({"version": str(result)})
```

The result will be returned to the requester:

```
$ http GET http://192.168.2.126:5000/devices/1/version
HTTP/1.1 200 OK
Connection: close
Content-Length: 216
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:19:52 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "version": "('lax-edg-r1', b'show version | i V\\r\\nCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)\\r\\nProcessor board ID 98U40DKV403INHIULHYHB\\
r\\n')"
}
```

We can also add another endpoint that will allow us to perform a bulk action on multiple devices based on their common fields. In the following example, the endpoint will take the `device_role` attribute in the URL and match it up with the appropriate device(s):

```
@app.route('/devices/<device_role>/version', methods=['GET'])
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all() if
device.role == device_role]
    result = {}
    for id in device_id_list:
        device = Device.query.get_or_404(id)
        hostname = device.hostname
        ip = device.mgmt_ip
        prompt = hostname + "#"
        device_result = show_version(hostname, prompt, ip, 'cisco',
'cisco')
        result[hostname] = str(device_result)
    return jsonify(result)
```

Of course, looping through all the devices in `Device.query.all()` is not efficient, as in the preceding code. In production, we will use a SQL query that specifically targets the role of the device.

When we use the RESTful API, we can see that all the spine, as well as leaf, devices can be queried at the same time:

```
$ http GET http://192.168.2.126:5000/devices/spine/version
HTTP/1.1 200 OK
Connection: close
Content-Length: 389
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:20:57 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "lax-edg-r1": "('lax-edg-r1', b'show version | i V\\r\\nCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)\\r\\nProcessor board ID 98U40DKV403INHIULHYHB\\
r\\n')",
```

```
    "lax-edg-r2": "('lax-edg-r2', b'show version | i V\\r\\nCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)\\r\\n')"
}
```

As illustrated, the new API endpoints query the device(s) in real time and return the result to the requester. This works relatively well when you can guarantee a response from the operation within the timeout value of the transaction (30 seconds, by default) or if you are OK with the HTTP session timing out before the operation is completed. One way to deal with the timeout issue is to perform the tasks asynchronously. We will look at how to do so in the next section.

Asynchronous operations

Asynchronous operations, when executing tasks out of the normal time sequence, are, in my opinion, an advanced topic of Flask.

Luckily, Miguel Grinberg (`https://blog.miguelgrinberg.com/`), whose Flask work I am a big fan of, provides many posts and examples on his blog and his GitHub repository. For asynchronous operations, the example code in `chapter9_8.py` referenced Miguel's GitHub code on the `Raspberry Pi` file (`https://github.com/miguelgrinberg/oreilly-flask-apis-video/blob/master/camera/camera.py`) for the background decorator. We will start by importing a few more modules:

```
from flask import Flask, url_for, jsonify, request,\
    make_response, copy_current_request_context
from flask_sqlalchemy import SQLAlchemy
from chapter9_pexpect_1 import show_version
import uuid
import functools
from threading import Thread
```

The background decorator takes in a function and runs it as a background task using the thread and UUID for the task ID. It returns the status code `202` (accepted) and the location of the new resources for the requester to check. We will make a new URL for status checking:

```
@app.route('/status/<id>', methods=['GET'])
def get_task_status(id):
    global background_tasks
    rv = background_tasks.get(id)
```

```
    if rv is None:
        return not_found(None)
    if isinstance(rv, Thread):
        return jsonify({}), 202, {'Location': url_for('get_task_status',
id=id)}
    if app.config['AUTO_DELETE_BG_TASKS']:
        del background_tasks[id]
    return rv
```

Once we retrieve the resource, it is deleted. This is done by setting `app.config['AUTO_DELETE_BG_TASKS']` to `true` at the top of the app. We will add this decorator to our version endpoints without changing the other part of the code because all of the complexity is hidden in the decorator (how cool is that?):

```
@app.route('/devices/<int:id>/version', methods=['GET'])
@background
def get_device_version(id):
    device = Device.query.get_or_404(id)
<skip>
@app.route('/devices/<device_role>/version', methods=['GET'])
@background
def get_role_version(device_role):
    device_id_list = [device.id for device in Device.query.all() if
device.role == device_role]
<skip>
```

The end result is a two-part process. We will perform the `GET` request for the endpoint and receive the location header:

```
$ http GET http://192.168.2.126:5000/devices/spine/version
HTTP/1.1 202 ACCEPTED
Connection: close
Content-Length: 3
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:25:25 GMT
Location: /status/bb57f6cac4c64e0aa2e67415eb7cabd0
Server: Werkzeug/2.2.2 Python/3.10.4

{}
```

We can then make a second request to the location to retrieve the result:

```
$ http GET http://192.168.2.126:5000/status/
bb57f6cac4c64e0aa2e67415eb7cabd0
HTTP/1.1 200 OK
Connection: close
Content-Length: 389
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:28:30 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "lax-edg-r1": "('lax-edg-r1', b'show version | i V\\r\\nCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)\\r\\nProcessor board ID 98U40DKV403INHIULHYHB\\
r\\n')",
    "lax-edg-r2": "('lax-edg-r2', b'show version | i V\\r\\nCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2,
RELEASE SOFTWARE (fc2)\\r\\n')"
}
```

To verify that the status code `202` is returned when the resource is not ready, we will use the following script, `chapter9_request_1.py`, to immediately make a request to the new resource:

```
import requests, time

server = 'http://192.168.2.126:5000'
endpoint = '/devices/1/version'

# First request to get the new resource
r = requests.get(server+endpoint)
resource = r.headers['location']
print("Status: {} Resource: {}".format(r.status_code, resource))

# Second request to get the resource status
r = requests.get(server+"/"+resource)
print("Immediate Status Query to Resource: " + str(r.status_code))
```

```
print("Sleep for 2 seconds")
time.sleep(2)
# Third request to get the resource status
r = requests.get(server+"/"+resource)
print("Status after 2 seconds: " + str(r.status_code))
```

As you can see in the result, the status code is returned while the resource is still being run in the background as `202`:

```
$ python chapter9_request_1.py
Status: 202 Resource: /status/960b3a4a81d04b2cb7206d725464ef71
Immediate Status Query to Resource: 202
Sleep for 2 seconds
Status after 2 seconds: 200
```

Our APIs are coming along nicely! Because our network resource is valuable, we should secure API access to only authorized personnel. We will add basic security measures to our API in the next section.

Authentication and Authorization

For basic user authentication, we will use Flask's `httpauth` (`https://flask-httpauth.readthedocs.io/en/latest/`) extension, written by Miguel Grinberg, as well as the password functions in Werkzeug. The `httpauth` extension should have been installed as part of the `requirements.txt` installation at the beginning of this chapter. The new file illustrating the security feature is named `chapter9_9.py`. In the script, we will start with a few more module imports:

```
from werkzeug.security import generate_password_hash, check_password_hash
from flask_httpauth import HTTPBasicAuth
```

We will create an `HTTPBasicAuth` object as well as the `user database` object. Note that, during the user creation process, we will pass the password value; however, we are only storing `password_hash` instead of the cleartext `password` itself:

```
auth = HTTPBasicAuth()
<skip>
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True)
    password_hash = db.Column(db.String(128))
```

```
    def set_password(self, password):
        self.password_hash = generate_password_hash(password)
    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

The `auth` object has a `verify_password` decorator that we can use, along with Flask's `g` global context object that was created when the user request started. Because `g` is global, if we save the user to the `g` variable, it will live through the entire transaction:

```
@auth.verify_password
def verify_password(username, password):
    g.user = User.query.filter_by(username=username).first()
    if g.user is None:
        return False
    return g.user.verify_password(password)
```

There is a handy `before_request` handler that can be used before any API endpoint is called. We will combine the `auth.login_required` decorator with the `before_request` handler that will be applied to all the API routes:

```
@app.before_request
@auth.login_required
def before_request():
    pass
```

Lastly, we will use the `unauthorized` error handler to return a `response` object for the `401` unauthorized error:

```
@auth.error_handler
def unathorized():
    response = jsonify({'status': 401, 'error': 'unauthorized',
                        'message': 'please authenticate'})
    response.status_code = 401
    return response
```

Before we can test user authentication, we will need to create users in our database:

```
>>> from chapter9_9 import db, User
>>> db.create_all()
>>> u = User(username='eric')
>>> u.set_password('secret')
```

```
>>> db.session.add(u)
>>> db.session.commit()
>>> exit()
```

Once you start your Flask development server, try to make a request, like we did previously. You should see that, this time, the server will reject the request with a `401` unauthorized error:

```
$ http GET http://192.168.2.126:5000/devices/
HTTP/1.1 401 UNAUTHORIZED
Connection: close
Content-Length: 82
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:39:06 GMT
Server: Werkzeug/2.2.2 Python/3.10.4
WWW-Authenticate: Basic realm="Authentication Required"

{
    "error": "unahtorized",
    "message": "please authenticate",
    "status": 401
}
```

We will now need to provide the authentication header for our requests:

```
$ http --auth eric:secret GET http://192.168.2.126:5000/devices/
HTTP/1.1 200 OK
Connection: close
Content-Length: 193
Content-Type: application/json
Date: Wed, 21 Sep 2022 18:39:42 GMT
Server: Werkzeug/2.2.2 Python/3.10.4

{
    "device": [
        "http://192.168.2.126:5000/devices/1",
        "http://192.168.2.126:5000/devices/2",
        "http://192.168.2.126:5000/devices/3",
        "http://192.168.2.126:5000/devices/4"
    ]
}
```

We now have a decent RESTful API set up for our network. When our user wants to retrieve network device information, they can query for the static content of the network. They can also perform network operations for a single device or a group of devices. We also added basic security measures to ensure that only the users we created can retrieve the information from our API. The cool part is that this is all done within a single file in less than 250 lines of code (less than 200 if you subtract the comments)!



For more information on user session management, logging in, logging out, and remembering user sessions, I highly recommend using the Flask-Login (`https://flask-login.readthedocs.io/en/latest/`) extension.

We have now abstracted the underlying vendor API away from our network and replaced them with our RESTful API. By providing the abstraction, we are free to use what is required in the backend, such as Pexpect, while providing a uniform frontend to our requester. We can even take a step forward and replace the underlying network device without impacting the users making API calls to us. Flask provides this abstraction in a compact and easy-to-use way for us. We can also run Flask with a smaller footprint, such as by using containers.

Running Flask in containers

Containers have become very popular in the last few years. They offer more abstractions and virtualization beyond hypervisor-based virtual machines. For interested readers, we will offer a simple example of how we can run our Flask app in a Docker container.

We will build our example based on the free DigitalOcean Docker tutorial on building containers on Ubuntu 20.04 machines (`https://www.digitalocean.com/community/tutorials/how-to-build-and-deploy-a-flask-application-using-docker-on-ubuntu-20-04`). If you are new to containers, I would highly recommend that you go through that tutorial and return to this section after.

Let's make sure Docker is installed:

```
$ sudo docker –version
Docker version 20.10.18, build b40c2f6
```

We will make a directory named `TestApp` to house our code:

```
$ mkdir TestApp
$ cd TestApp/
```

In the directory, we will make another directory called app and create the `__init__.py` file:

```
$ mkdir app
$ touch app/__init__.py
```

Under the app directory is where we will contain the logic of our application. Since we have been using a single-file app up to this point, we can simply copy over the contents of our `chapter9_6.py` file to the `app/__init__.py` file:

```
$ cat app/__init__.py
from flask import Flask, url_for, jsonify, request
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///network.db'
db = SQLAlchemy(app)
@app.route('/')
def home():
    return "Hello Python Networking!"
<skip>
class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(64), unique=True)
    loopback = db.Column(db.String(120), unique=True)
    mgmt_ip = db.Column(db.String(120), unique=True)
    role = db.Column(db.String(64))
    vendor = db.Column(db.String(64))
    os = db.Column(db.String(64))
<skip>
```

We can also copy the SQLite database file we created to this directory:

```
$ tree app/
app/
├── __init__.py
├── network.db
```

We will place the `requirements.txt` file in the `TestApp` directory:

```
$ cat requirements.txt
Flask==1.1.1
```

```
Flask-HTTPAuth==3.3.0
Flask-SQLAlchemy==2.4.1
Jinja2==2.10.1
MarkupSafe==1.1.1
Pygments==2.4.2
SQLAlchemy==1.3.9
Werkzeug==0.16.0
httpie==1.0.3
itsdangerous==1.1.0
python-dateutil==2.8.0
requests==2.20.1
```



Due to a conflict with the `tiangolo/uwsgi-nginx-flask` image and some later versions of Flask packages, this requirements file is reverting to Flask 1.1.1. The portion of code we worked on works in both version 1.1.1 and the latest Flask version.

We will create the `main.py` file as our entry point and an `ini` file for `uwsgi`:

```
$ cat main.py
from app import app
$ cat uwsgi.ini
[uwsgi]
module = main
callable = app
master = true
```

We will use a pre-made Docker image and create a `Dockerfile` that builds the Docker image:

```
$ cat Dockerfile
FROM tiangolo/uwsgi-nginx-flask:python3.7-alpine3.7
RUN apk --update add bash vim
RUN mkdir /TestApp
ENV STATIC_URL /static
ENV STATIC_PATH /TestApp/static
COPY ./requirements.txt /TestApp/requirements.txt
RUN pip install -r /TestApp/requirements.txt
```

Our `start.sh` shell script will build the image, run it as a daemon in the background, then forward port `8000` to the Docker container:

```
$ cat start.sh
#!/bin/bash
app="docker.test"
docker build -t ${app} .
docker run -d -p 8000:80 \
  --name=${app} \
  -v $PWD:/app ${app}
```

We can now use the `start.sh` script to build the image and launch our container:

```
$ sudo bash start.sh
Sending build context to Docker daemon  48.13kB
Step 1/7 : FROM tiangolo/uwsgi-nginx-flask:python3.8
python3.8: Pulling from tiangolo/uwsgi-nginx-flask
85bed84afb9a: Pulling fs layer
5fdd409f4b2b: Pulling fs layer
<skip>
```

Our Flask now runs in the container that can be viewed from our host machine port `8000`:

```
$ sudo docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED
STATUS         PORTS                                          NAMES
25c83da6082c   docker.test   "/entrypoint.sh /sta…"   2 minutes ago   Up 2
minutes   443/tcp, 0.0.0.0:8000->80/tcp, :::8000->80/tcp   docker.test
```

We can see the **management host IP** displayed in the address bar as follows:

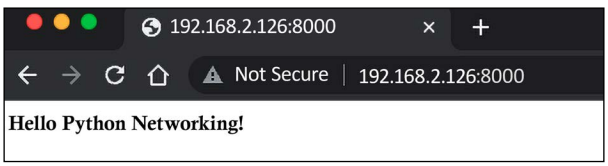


Figure 9.5: Management Host IP Forwarding

We can see the **Flask API endpoint** as follows:



Figure 9.6: API Endpoint

Once we are done, we can use the following commands to stop and delete the container:

```
$ sudo docker stop <container id>
$ sudo docker rm <containter id>
```

We can also delete the Docker image:

```
$ sudo docker images -a -q #find the image id
$ sudo docker rmi <image id>
```

As we can see, running Flask in a container gives us even more flexibility and the option to deploy our API abstraction in production. Containers, of course, offer their complexity and add more management tasks, so we need to weigh up the benefits and overhead when it comes to our deployment methods. We are close to the end of this chapter, so let's look at what we have done so far before looking forward to the next chapters.

Summary

In this chapter, we started to move on to the path of building RESTful APIs for our network. We looked at popular Python web frameworks, namely Django and Flask, and compared and contrasted the two. By choosing Flask, we can start small and expand on features by using Flask extensions.

In our lab, we used the virtual environment to separate the Flask installation base from our global site packages. The lab network consists of several IOSv nodes, two of which we have designated as spine routers while the other two as leaf routers. We took a tour of the basics of Flask and used the simple `HTTPie` client to test our API setup.

Among the different setups of Flask, we placed special emphasis on URL dispatch as well as URL variables because they are the initial logic between the requesters and our API system. We took a look at using Flask-SQLAlchemy and SQLite to store and return network elements that are static in nature. For operation tasks, we also created API endpoints while calling other programs, such as Pexpect, to accomplish configuration tasks. We improved the setup by adding asynchronous handling and user authentication to our API. We also looked at how to run our Flask API application in a Docker container.

In Chapter 10, Introduction to Async IO, we will shift gears to look at one of the newer features in Python 3, Async IO, and how it can be applied to network engineering.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



10

Introduction to Async IO

In the previous chapters, we have been interacting with the network devices directly via API or other Python libraries that abstracted us from low-level interactions with a remote device. When we need to interact with multiple devices, we use loops to allow us to pragmatically execute commands. One issue that we might start to see is that the end-to-end process begins to slow down when we need to interact with many devices. The bottleneck is usually the time spent waiting between the time we send the command until we receive the proper response from the remote device. If we need to spend 5 seconds of wait time per operation, we could wait for a few minutes when we need to operate on 30 devices.

This is partially true because our operations are sequential. We are only operating on one device at a time, in sequence. What if we can process multiple devices at the same time? That would speed things up, right? Yes, you are correct. But it is not as simple as "telling" our Python script to "go for" many devices simultaneously. We must consider the way computers schedule tasks, the language limitation, and the available tools at hand.

In this chapter, we will discuss Async IO, the Python package that allows us to perform multiple tasks at the same time. We will also discuss related topics such as multiprocessing, parallelism, threading, and others. Asynchronous operations in Python is a topic that I would consider medium to advanced level. The async IO module itself was only introduced in Python 3.4. It also went through rapid changes between Python 3.4 to Python 3.7. Regardless, it is a very relatable topic for network automation. I believe it is worth a study for any network engineer looking to be familiar with network automation.

In this chapter, we will discuss the following topics related to Async IO:

- Asynchronous operations overview
- Multiprocessing and threading
- Python asyncio module
- The `Scrapli` project



For more information on Python-related asynchronous operations, Real Python (`https://realpython.com/search?q=asyncio`) and Python documentation (`https://docs.python.org/3/library/asyncio.html`) both offer good resources for learning.

Let's start by looking at an overview of asynchronous operations.

Asynchronous operations overview

In the *Zen of Python;* we know one of the guiding principles in Python is to preferably have "one best way to do something." When it comes to asynchronous operations, it is a bit complicated. We know it would help if we could do multiple tasks simultaneously but determining the correct solution might not be straightforward.

First, we will need to determine what is slowing down our program. Typically, the bottleneck can be either CPU-bound or I/O-bound. In a CPU-bound situation, the program pushes the CPU to its limit. Operations such as solving mathematical questions or image processing are examples of CPU-bound programs. For example, when we pick an encryption algorithm for VPN, we know the more complex the algorithm, the more CPU it will consume. For CPU-bound tasks, the way to mitigate the bottleneck is to increase the CPU power or allow the task to use multiple CPUs simultaneously.

In an IO-bound operation, the program spends most of its time waiting for some output from an input it has completed. When we make an API call to a device, we cannot move on to the next step until we receive what we need as the answer. If time is significant, this is time that otherwise could have been useful in doing something else. The way to mitigate IO-bound tasks is to work on multiple tasks simultaneously.

If the work at hand is limited by CPU power or Input-Output latency, we can try to run multiple operations at once. This is called parallelism. Of course, not all tasks can be parallelized. As the great Warren Buffet saying goes, "You can't produce a baby in one month by getting nine women pregnant." However, if your task can be parallelized, we have a few parallel processing options: multiprocessing, threading, or the new asyncio module.

Python multiprocessing

Python's multiprocessing allows CPU-bound tasks to be broken up into sub-tasks and spawning subprocesses to handle them. This is well-suited for CPU-bound tasks because it allows multiple CPUs to work simultaneously. If we look back at the history of computing, we notice that around 2005, a single CPU can no longer get any faster. We simply cannot fit more transistors onto a single CPU due to interference and heat issues. The way we have gotten more computing power is by having multi-core CPUs. This is beneficial in allowing us to spread our tasks among the multi-core CPUs.

In Python's multiprocessing module, processes are spawned by creating a `Process` object and then calling its `start()` method. Let's see a simple example, `multiprocess_1.py`:

```
#!/usr/bin/env python3
# Modified from
# https://docs.python.org/3/library/multiprocessing.html
from multiprocessing import Process
import os

def process_info():
    print('process id:', os.getpid())

def worker(number):
    print(f'Worker number {number}')
    process_info()


if __name__ == '__main__':
    for i in range(5):
        p = Process(target=worker, args=(i,))
        p.start()
```

In the example, we have a worker function that calls another `process_info()` function to get the process ID. Then we start the Process object five times, each one targeting the worker function. The output for the execution is below:

```
(venv) $ python multiprocess_1.py
Worker number 0
process id: 109737
Worker number 2
process id: 109739
Worker number 3
process id: 109740
Worker number 1
process id: 109738
Worker number 4
process id: 109741
```

As we can see, each process has its process and process ID. Multiprocessing is great for CPU-bound tasks. If the work is IO-bound, before the asyncio module, our best bet is to use the threading module.

Python multithreading

As many of us know, Python has a **Global Interpreter Lock**, or **GIL**. It is used by the Python interpreter (CPython, to be exact) to assure that only one thread executes Python byte code at a time. This is mainly a safety measure to protect against race conditions in memory leaks. But it can become a performance bottleneck for IO-bound tasks.



For more information, check out the article at `https://realpython.com/python-gil/`.

One way to allow multiple threads to run is by using the `threading` module. It allows a program to run multiple operations concurrently. We can see a simple example in `threading_1.py`:

```
#!/usr/bin/env python3
# Modified from https://pymotw.com/3/threading/index.html
import threading

# Get thread ID
def thread_id():
    print('thread id:', threading.get_ident())

# Worker function
def worker(number):
    print(f'Worker number {number}')
    thread_id()


threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

The script is similar to our multiprocess example, with the exception of displaying the thread ID instead of the process ID. The output for the script execution is below:

```
(venv) $ python threading_1.py
Worker number 0
thread id: 140170712495680
Worker number 1
thread id: 140170704102976
Worker number 2
thread id: 140170695710272
Worker number 3
thread id: 140170704102976
Worker number 4
thread id: 140170695710272
```

The threading module is a good option to mitigate the Python GIL with multiple threads. However, when Python passes the task to a thread, the main process has limited visibility in the threading process. The threads are harder to deal with, especially when coordinating between different threads and handling errors if they arise. For IO-bound tasks, instead of threads, asyncio in Python 3 is another great option.

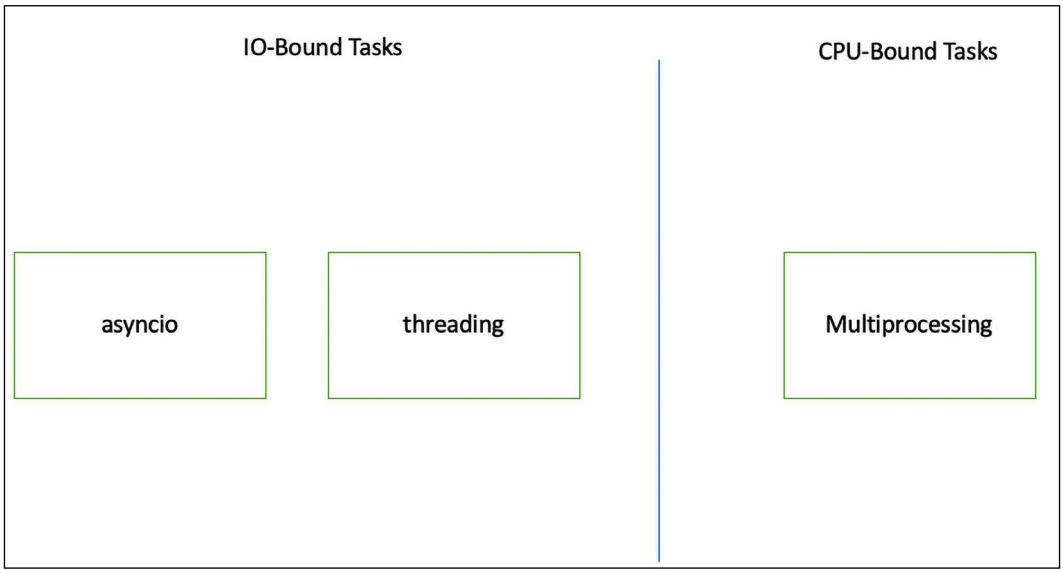


Figure 10.1: CPU-bound vs. IO-bound Python modules

Let's dig deeper into the asyncio module.

Python asyncio module

We can think of the asyncio module as Python's way of allowing us to write code to run tasks concurrently. It uses the newly introduced `async` and `await` keywords. It can help us improve performance for many operations that might be IO-bound, such as web servers, databases, and, of course, communication toward devices over a network. The asyncio module is the foundation of popular new frameworks, such as FastAPI (`https://fastapi.tiangolo.com/`).

However, it is important to point out that asyncio is neither multiprocessing nor multithreaded. It is designed to be single-threaded with a single process. Python asyncio uses *cooperative multiprocessing* to give the feeling of concurrency.

Unlike threading, Python controls the process from end to end instead of passing the threading process to the operating system. This lets Python know when the task is started and completed, thus coordinating between the processes. When we "pause" part of the code while waiting for results, Python will move on to other parts of the code before coming back to the "paused" code.

This is an important concept to grasp before writing our asyncio code. We need to decide which part of the code can be paused to allow Python to temporarily move on from it. We have to tell Python, "Hey, I am just waiting for something. Go do something else and come back to check on me."

Let us start with a simple example of the asyncio module syntax in `asyncio_1.py`:

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

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')
    await asyncio.sleep(2)
    print('... and again.')

asyncio.run(main())
```

When we execute it, here is the output:

```
$ python asyncio_1.py
Hello ...
... World!
... and again.
```

There are several things we can take note of in this example:

1. The asyncio module is in the standard library for Python 3.10.
2. The `async` keyword is used in front of the function. In asyncio, this is called a coroutine.
3. The `await` keyword is waiting for the return of some operations.
4. Instead of simply calling the function/coroutine, we use `asyncio.run()` to do so.

At the heart of the asyncio module are coroutines, defined with the `async` keyword. A coroutine is a specialized version of a Python generator function that can temporarily give back control to the Python interpreter while waiting.



Generator functions are a type of function that can be iterated over like a list but do so without loading the content into memories first. This is useful when, for example, the dataset is so large that it might overwhelm a computer's memory. For more information, check out this documentation: `https://wiki.python.org/moin/Generators`.

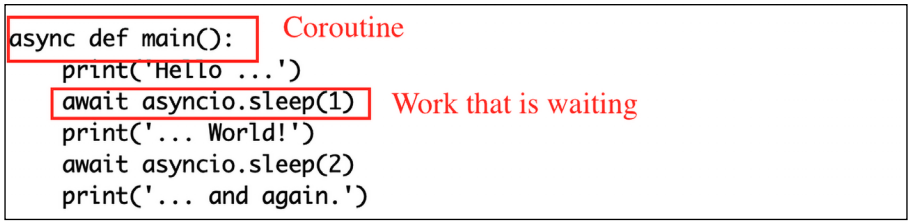


Figure 10.2: Coroutine with async and await

Let's take this example further and see how we can build on it. The following examples were taken from the excellent tutorial from *RealPython.com* (`https://realpython.com/async-io-python/#the-asyncio-package-and-asyncawait`). We will start with a synchronous count function with `sync_count.py`:

```
#!/usr/bin/env python3
# Modified from https://realpython.com/async-io-python/#the-asyncio-
package-and-asyncawait countsync.py example
import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    count()
    count()
    count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
```

```
    elapsed = time.perf_counter() - s
    print(f"Completed in {elapsed:0.2f} seconds.")
```

Upon execution, we can see the script executes in three seconds by faithfully executing the function three times sequentially:

```
(venv) $ python sync_count.py
One
Two
One
Two
One
Two
Completed in 3.00 seconds.
```

Now, let's see if we can build an asynchronous version of it, `async_count.py`:

```
#!/usr/bin/env python3
# example from https://realpython.com/async-io-python/#the-asyncio-
package-and-asyncawait countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"Completed in {elapsed:0.2f} seconds.")
```

When we execute this file, we see that similar tasks were completed in 1/3 of the time:

```
(venv) $ python async_count.py
One
One
One
Two
Two
Two
Completed in 1.00 seconds.
```

Why is that? It is because now when we are counting and hit the sleep pause, we give the control back to the interpreter to allow it to process other tasks.

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"Completed in {elapsed:0.2f} seconds.")
```


Figure 10.3: Event Loop

There are a few important points to note in this example:

1. The `sleep()` function is changed to an `asyncio.sleep()` function. It is an `awaitable` function.
2. Both the `count()` and `main()` functions are now coroutines.
3. We used `ansyncio.gather()` to collect all the coroutines.
4. The `asyncio.run()` is a loop that runs until everything is completed.

From the example, we can see there are several changes we need to make to regular functions to allow the performance gain offered by asyncio. Remember we talked about cooperative multiprocessing? Asyncio requires all components within the Python programs to work together to achieve this goal.

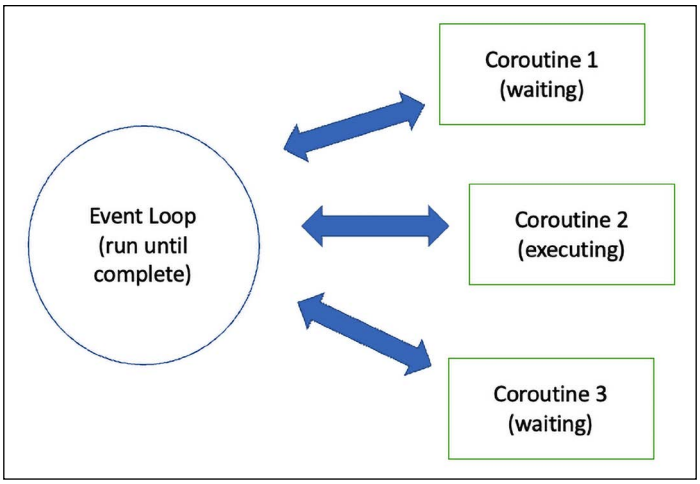


Figure 10.4: Event loop

In the next section, we will look at the Scrapli project that helps us speed up the network device interaction process by taking advantage of the Python 3 asyncio feature.

The Scrapli project

Scrapli is an open-source network library (`https://github.com/carlmontanari/scrapli`) that uses Python 3's asyncio capabilities to help connect to network devices faster. It was created by Carl Montanari (`https://github.com/carlmontanari`) while working on his network automation projects. The installation is straightforward:

```
(venv) $ pip install scrapli
(venv) $ mkdir scrapli && cd scrapli
```

Let's go ahead and start using Scrapli for our network device communication.

Scrapli example

We can use the following example, `scrapli_example_1.py`, to perform a `show` command on our lab NX-OS device, `lax-cor-r1`:

```
# Modified from https://github.com/carlmontanari/scrapli
from scrapli import Scrapli

device = {
   "host": "192.168.2.50",
   "auth_username": "cisco",
```

```
    "auth_password": "cisco",
    "auth_strict_key": False,
    "ssh_config_file": True,
    "platform": "cisco_nxos",
}

conn = Scrapli(**device)
conn.open()
response = conn.send_command("show version")
print(response.result)
```

Executing the script will give us the `show version` output. Notice this is in a string format:

```
(venv) $ python scrapli_example_1.py
Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
…
Software
  loader:    version N/A
  kickstart: version 7.3(0)D1(1)
  system:    version 7.3(0)D1(1)

Hardware
  cisco NX-Osv Chassis ("NX-Osv Supervisor Module")
  IntelI CITM) i5-7260U C with 3064740 kB of memory.
  Processor Board ID TM000940CCB

  Device name: lax-cor-r1
  bootflash:    3184776 kB
…
```

On the surface, it might not look any different than some of the other libraries we have seen. But underneath the hood, the core drivers and associated platforms are using the asyncio module that can be turned into an `awaitable` coroutine:

Platform/OS	Scrapli Driver	Scrapli Async Driver	Platform Name
Cisco IOS-XE	IOSXEDriver	AsyncIOSXEDriver	cisco_iosxe
Cisco NX-OS	NXOSDriver	AsyncNXOSDriver	cisco_nxos
Cisco IOS-XR	IOSXRDriver	AsyncIOSXRDriver	cisco_iosxr
Arista EOS	EOSDriver	AsyncEOSDriver	arista_eos
Juniper JunOS	JunosDriver	AsyncJunosDriver	juniper_junos

Figure 10.5: Scrapli Core Drivers (source: https://carlmontanari.github.io/scrapli/user_guide/basic_usage/)



We can verify the code by going to the project's GitHub page, `https://github.com/carlmontanari/scrapli`. The NXOS Async driver, `https://github.com/carlmontanari/scrapli/blob/main/scrapli/driver/core/cisco_nxos/async_driver.py`, can be traced back to the base async driver, `https://github.com/carlmontanari/scrapli/blob/main/scrapli/driver/base/async_driver.py`, as well as the base driver, `https://github.com/carlmontanari/scrapli/blob/main/scrapli/driver/base/base_driver.py`. This is part of the beauty of open-source projects, and we have the freedom to explore and build on each other's knowledge. Thank you, Carl!

The core drivers include Cisco IOS-XE, Cisco NX-OS, Cisco IOS-XR, Arista EOS, and Juniper JunOS. By simply specifying the platform, Scrapli is able to correlate it with the particular driver. There is also a `scrapli_community` project (`https://github.com/scrapli/scrapli_community`) that extends beyond the core drivers.

In our lab, we specify additional `ssh` configurations. Therefore, we need to set `ssh_config_file` to be `true`:

```
$ cat ~/.ssh/config
…
Host 192.168.2.50
  HostKeyAlgorithms +ssh-rsa
  KexAlgorithms +diffie-hellman-group-exchange-sha1
```



Scrapli's documentation, `https://carlmontanari.github.io/scrapli/`, is a good place to start. Packet Coders, `https://www.packetcoders.io/`, also offers good network automation classes, including Scrapli.

We can now put this awaitable task into an asyncio run loop.

Scrapli async example

In this example, we will be more precise about the driver and transport. We will install the `asyncssh` plugin (`https://carlmontanari.github.io/scrapli/api_docs/transport/plugins/asyncssh/`) from Scrapli to be used:

```
(venv) $ pip install scrapli[asyncssh]
```

The script, `scraplie_example_2.py`, is listed below:

```
#!/usr/bin/env python3
# Modified from
# https://github.com/carlmontanari/scrapli/blob/main/examples/async_usage/
async_multiple_connections.py
import asyncio
from scrapli.driver.core import AsyncNXOSDriver

async def gather_cor_device_version(ip, username, password):
    device = {
        "host": ip,
        "auth_username": username,
        "auth_password": password,
        "auth_strict_key": False,
        "ssh_config_file": True,
        "transport": "asyncssh",
```

```
        "driver": AsyncNXOSDriver
    }

    driver = device.pop("driver")
    conn = driver(**device)
    await conn.open()
    response = await conn.send_command("show version")
    await conn.close()
    return response

async def main():
    results = await asyncio.gather(
                        gather_cor_device_version('192.168.2.50', 'cisco',
'cisco'),
                        gather_cor_device_version('192.168.2.60', 'cisco',
'cisco')
                    )
    for result in results:
        print(result.result)


if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"Completed in {elapsed:0.2f} seconds.")
```

The script creates two new coroutines, one for gathering device information and the other for collecting the coroutine tasks within the `main()` function. We also created an `asyncio.run()` loop to run the `main()` function when the script is executed by itself. Let's execute the script:

```
(venv) $ python scrapli_example_2_async.py
Cisco Nexus Operating System (NX-OS) Software
…
  loader:    version N/A
  kickstart: version 7.3(0)D1(1)
  system:    version 7.3(0)D1(1)
…
```

```
  Device name: lax-cor-r1
  bootflash:    3184776 kB
…
  Device name: nyc-cor-r1
  bootflash:    3184776 kB
…
Completed in 1.37 seconds.
```

Besides the `show version` output from the two devices, we also saw that the execution was completed in little over 1 second.

Let's compare the performance difference between synchronous and asynchronous operations. Scrapli provides a `GenericDriver` for synchronous operations. In the example script `scrapli_example_3_sync.py`, we will use the `GenericDriver` to gather the information repeatedly. Just for illustration purposes, the script connects to each of the devices three times:

```
#!/usr/bin/env python3
# Modified from
# https://github.com/carlmontanari/scrapli/blob/main/examples/async_usage/
async_multiple_connections.py
import asyncio
# from scrapli.driver.core import Paramiko
from scrapli.driver import GenericDriver

def gather_cor_device_version(ip, username, password):
    device = {
        "host": ip,
        "auth_username": username,
        "auth_password": password,
        "auth_strict_key": False,
        "ssh_config_file": True,
        "driver": GenericDriver
    }

    driver = device.pop("driver")
    conn = driver(**device)
    conn.open()
```

```
    response = conn.send_command("show version")
    conn.close()
    return response

def main():
    results = []
    for device in [
                    '192.168.2.50',
                    '192.168.2.60',
                    '192.168.2.50',
                    '192.168.2.60',
                    '192.168.2.50',
                    '192.168.2.60',
                    '192.168.2.50',
                    '192.168.2.60',
                  ]:
        results.append(gather_cor_device_version(device, 'cisco',
'cisco'))
    return results


if __name__ == "__main__":
    import time
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"Completed in {elapsed:0.2f} seconds.")
```

There is also a comparable async version, `scrapli_example_3_async.py`. When we run the two scripts, here is the performance difference:

```
(venv) $ python scrapli_example_3_sync.py
Completed in 5.97 seconds.
(venv) $ python scrapli_example_3_async.py
Completed in 4.67 seconds.
```

This might not seem much of an improvement, but as we scale up our operations, the performance gain will become more significant.

Summary

In this chapter, we learned about the concepts of asynchronous processing. We touched on the concepts behind CPU-bound and IO-bound tasks. We previously addressed the bottlenecks caused by them with multiprocessing and multithreading.

Starting with Python 3.4, the new asyncio module was introduced to address IO-bound tasks. It is similar to multithreading but uses a special cooperative multitasking design. They use special keywords – the `async` keyword to create functions that are special types of Python generators and the `await` keyword to specify tasks that can be temporarily "paused." The asyncio module can then collect these tasks and run them in a loop until completed.

In the latter part of the chapter, we learned about using Scrapli, a project created by Carl Montanari for the network engineering community. It is designed to utilize the asyncio feature in Python 3 for network device management.

Asyncio is not easy. The new terminology of async, await, loop, and generators can feel overwhelming. The asyncio module has also been under rapid development from Python version 3.4 to 3.7, making some online documents outdated. Hopefully, the information presented in this chapter can help us understand this useful feature.

In the next chapter, we will switch gears toward cloud computing and the network features surrounding cloud computing.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



11

AWS Cloud Networking

Cloud computing is one of the major trends in computing today and has been for many years. Public cloud providers have transformed the start-up industry and what it means to launch a service from scratch. We no longer need to build our own infrastructure; we can pay public cloud providers to rent a portion of their resources for our infrastructure needs. Nowadays, walking around any technology conferences or meetups, we will be hard-pressed to find someone who has not learned about, used, or built services based in the cloud. Cloud computing is here, and we had better get used to working with it.

There are several cloud computing service models, roughly divided into **Software-as-a-Service** (**SaaS** — `https://en.wikipedia.org/wiki/Software_as_a_service`), **Platform-as-a-Service** (**PaaS** — `https://en.wikipedia.org/wiki/Cloud_computing#Platform_as_a_service_(PaaS)`), and **Infrastructure-as-a-Service** (**IaaS** — `https://en.wikipedia.org/wiki/Infrastructure_as_a_service`). Each service model offers a different level of abstraction from the user's perspective. For us, networking is part of the IaaS offering and the focus of this chapter.

Amazon Web Services (**AWS** — `https://aws.amazon.com/`) was the first company to offer IaaS public cloud services and was the clear leader in the space by market share in 2022 (`https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/`). If we define the term **Software-Defined Networking** (**SDN**) as a group of software services working together to create network constructs – IP addresses, access lists, load balancers, and **Network Address Translation** (**NAT**) – we can make the argument that AWS is the world's largest implementer of SDN. They utilize the massive scale of their global network, data centers, and servers to offer an amazing array of networking services.

If you are interested in learning about Amazon's scale and networking, I would highly recommend taking a look at James Hamilton's AWS re:Invent 2014 talk: `https://www.youtube.com/watch?v=JIQETrFC_SQ`. It is a rare insider's view of the scale and innovation at AWS.

In this chapter, we will discuss the networking services offered by the AWS cloud services and how we can use Python to work with them:

- AWS setup and networking overview
- Virtual private cloud
- Direct Connect and VPN
- Networking scaling services
- Other AWS network services

Let's begin by looking at how to set up AWS.

AWS setup

If you do not already have an AWS account and wish to follow along with these examples, please log on to `https://aws.amazon.com/` and sign up. The process is pretty straightforward; you will need a credit card and some way to verify your identity, such as a mobile phone that can accept text messages.

A good thing about AWS when you are just getting started is that they offer many services in a free tier (`https://aws.amazon.com/free/`), where you can use the services for free up to a certain level. For example, we will use the **Elastic Compute Cloud (EC2)** service in this chapter; the free tier for EC2 is the first 750 hours per month for its t2.micro or t3.micro instances for the first 12 months.

I recommend always starting with the free tier and gradually increasing your tier when the need arises. Please check the AWS site for the latest offerings:

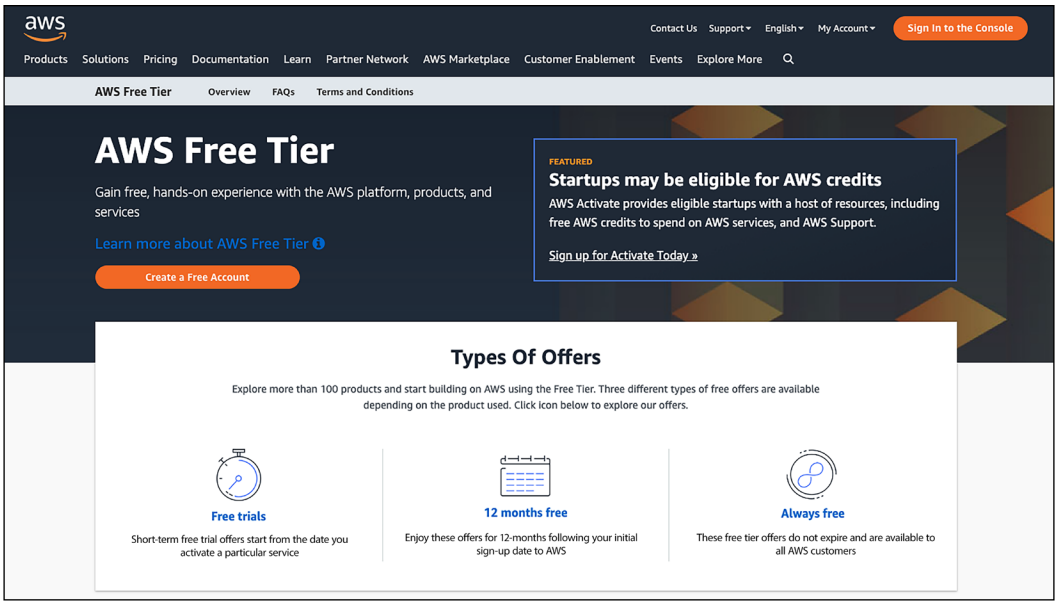


Figure 11.1: AWS free tier

Once you have an account, you can sign in via the AWS console (`https://console.aws.amazon.com/`) and take a look at the different services offered by AWS.



The AWS console layout is constantly changing. By the time you read this chapter, your screen might look different than what is shown. However, the AWS networking concepts will not change. We should always pay attention to the concept, and we should be ok despite any layout changes.

The console is where we can configure all the services and look at our monthly bills:

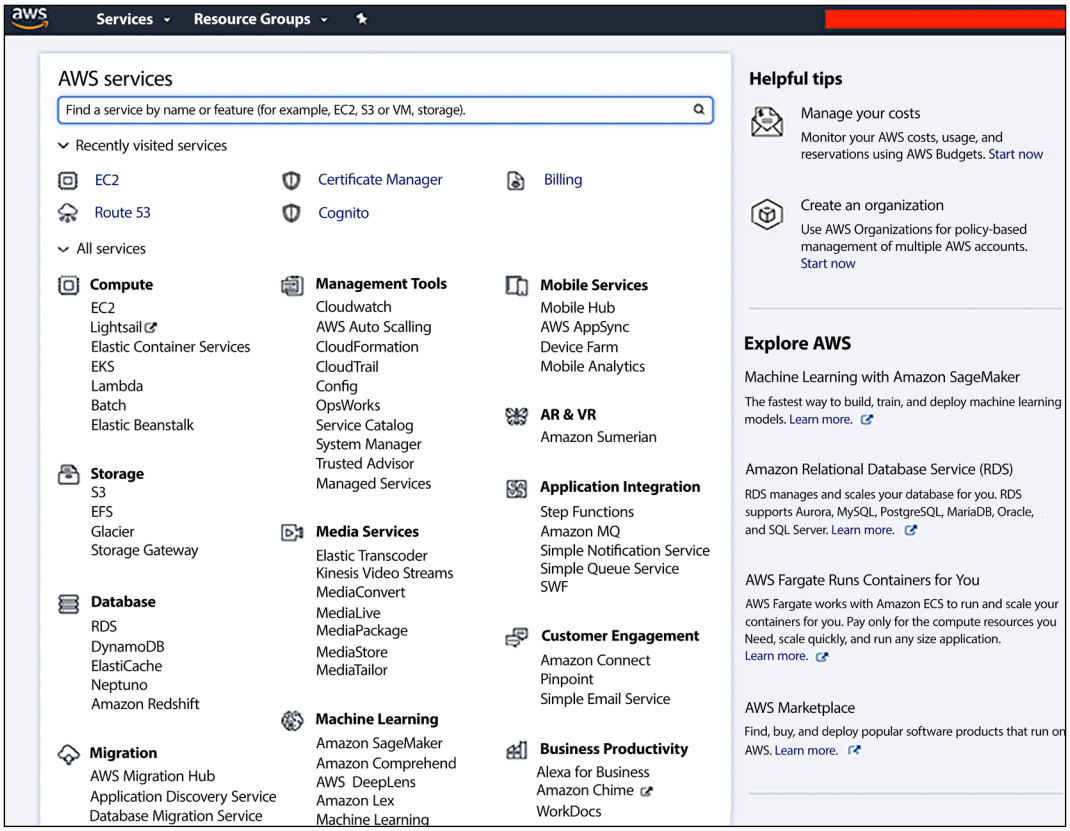


Figure 11.2: The AWS console

Now that we have set up our account, let's take a look at using the AWS CLI tool as well as the Python SDK to manage our AWS resources.

The AWS CLI and Python SDK

Besides the console, we can also manage AWS services via the **command line interface (CLI)** and various SDKs. **The AWS CLI is a Python package that can be installed via PIP** (`https://docs.aws.amazon.com/cli/latest/userguide/installing.html`). Let's install it on our Ubuntu host:

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
$ which aws
```

```
/usr/local/bin/aws
$ aws --version
aws-cli/2.7.34 Python/3.9.11 Linux/5.15.0-47-generic exe/x86_64.ubuntu.22
prompt/off
```

Once the AWS CLI is installed, for easier and more secure access, we will create a user and configure the AWS CLI with the user credentials. Let's go back to the AWS console and select **Identity and Access Management (IAM)** for user and access management:

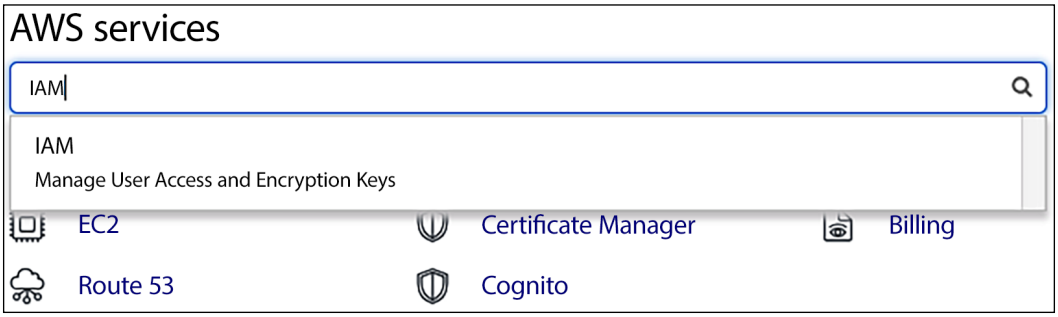


Figure 11.3: AWS IAM

We can choose **Users** on the left panel to create a user:



Figure 11.4: AWS IAM users

Select **Programmatic access** and assign the user to the default administrator group:

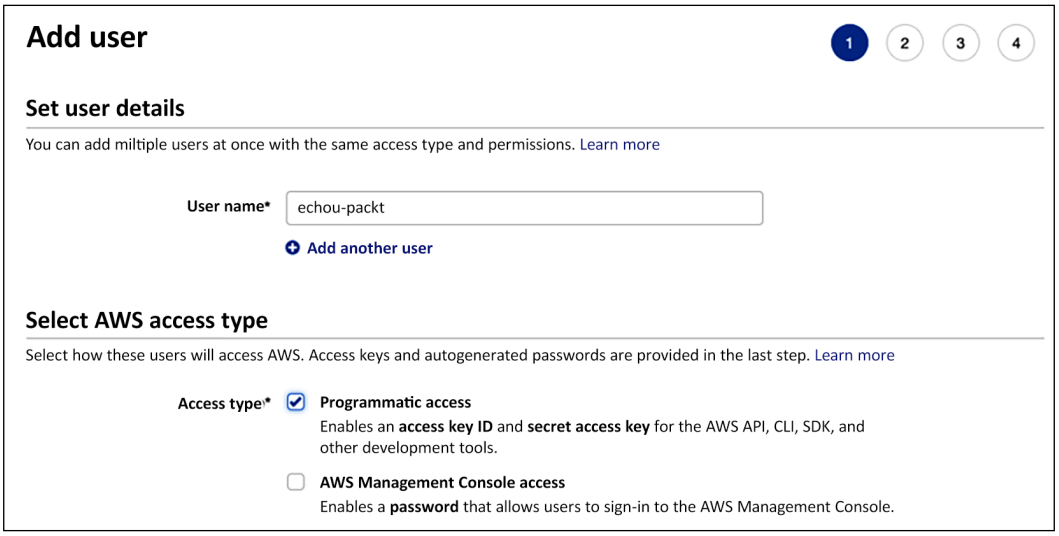


Figure 11.5: AWS IAM add user

The next step will add the user to the group; we can add the user to the administrator group for now. We do not need to add any tag for this user. The last step will show an **Access key ID** and a **Secret access key**. Copy them into a text file and keep it in a safe place:

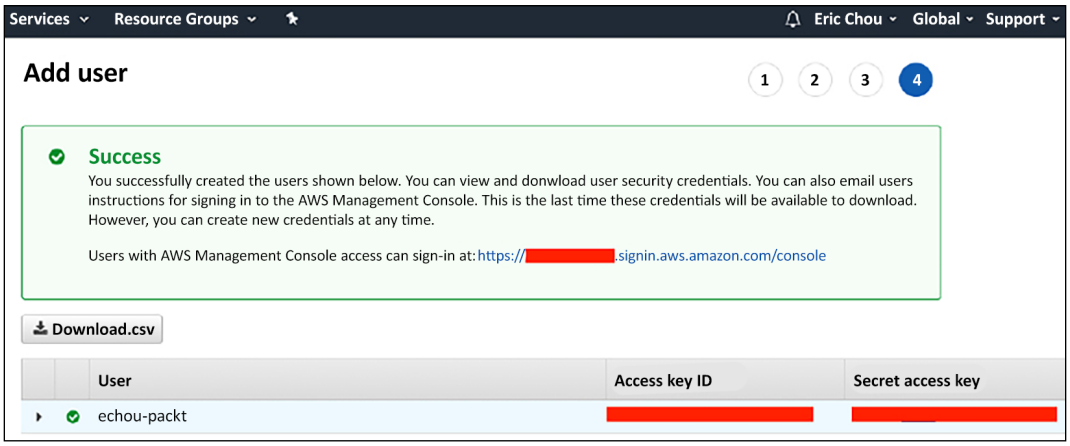


Figure 11.6: AWS IAM user security credentials

We will complete the AWS CLI authentication credential setup via `aws configure` in the terminal. We will go over AWS Regions in the upcoming section. We will use `us-east-1` for now since that is the Region with the most services. We can always come back to the settings later to change the Region:

```
$ aws configure
AWS Access Key ID [None]: <key>
AWS Secret Access Key [None]: <secret>
Default region name [None]: us-east-1
Default output format [None]: json
```

We will also install the AWS Python SDK, Boto3 (`https://boto3.readthedocs.io/en/latest/`):

```
(venv) $ pip install boto3
(venv) $ python
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>> boto3.__version__
'1.24.78'
>>> exit()
```

We are now ready to move on to the subsequent sections, starting with an introduction to AWS cloud networking services.

AWS network overview

When we discuss AWS services, we need to start at the top, with Regions and **Availability Zones (AZs)**. They have big implications for all of our services. At the time of writing this book, AWS has listed 27 geographic Regions and 87 **AZs** worldwide. In the words of AWS Global Cloud Infrastructure (`https://aws.amazon.com/about-aws/global-infrastructure/`):

> *"The AWS Cloud infrastructure is built around Regions and Availability Zones (AZs). AWS Regions provide multiple, physically separated and isolated Availability Zones which are connected with low latency, high throughput, and highly redundant networking."*

For a nice visualization of AWS Regions that can be filtered by AZ, Region, and so on, please check

out `https://aws.amazon.com/about-aws/global-infrastructure/regions_az/`.

Some of the services AWS offers are global (such as the IAM user we created), but most of the services are Region-based. The Regions are geographic footprints, such as US-East, US-West, EU-London, Asia-Pacific-Tokyo, etc. What this means for us is that we should build our infrastructure in a region that is closest to our intended users. This will reduce the latency of the service for our customers. If our users are on the **East Coast** of the United States, we should pick **US East (N. Virginia)** or **US East (Ohio)** as our Region if the service is Regional-based:

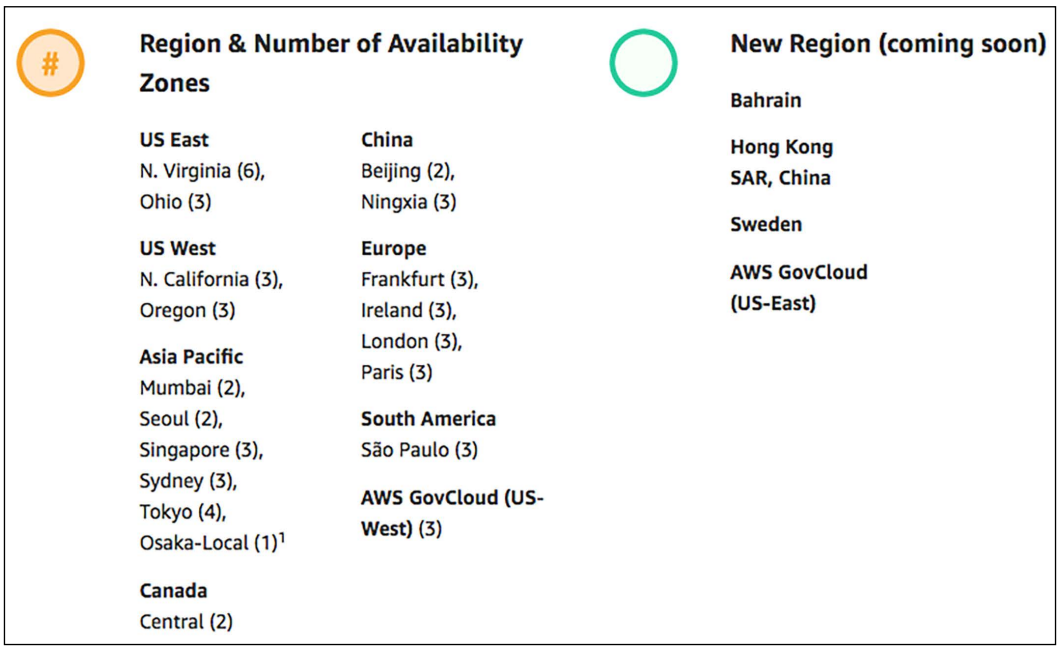


Figure 11.7: AWS Regions

Besides user latency, AWS Regions also have both service and cost implications. Users who are new to AWS might find it surprising that not all services are offered in all Regions. The services we will look at in this chapter are offered in most Regions, but some newer services might only be offered in selected Regions.

In the example that follows, we can see that **Alexa for Business** and **Amazon Chime** are only offered in the Northern Virginia Region in the United States:

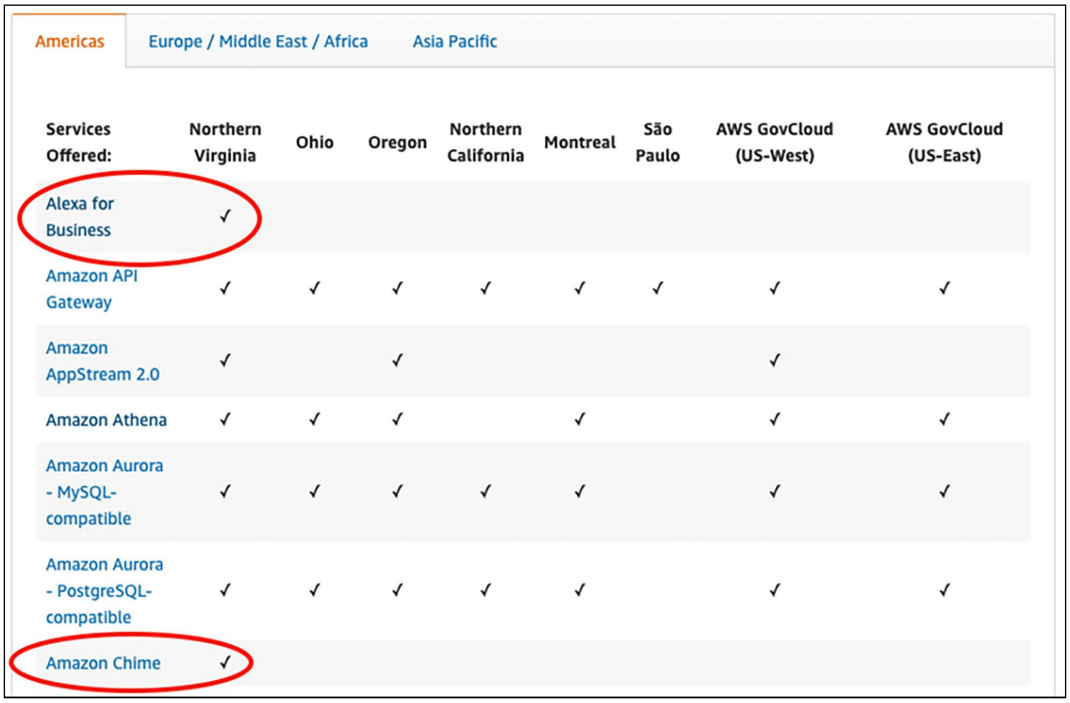
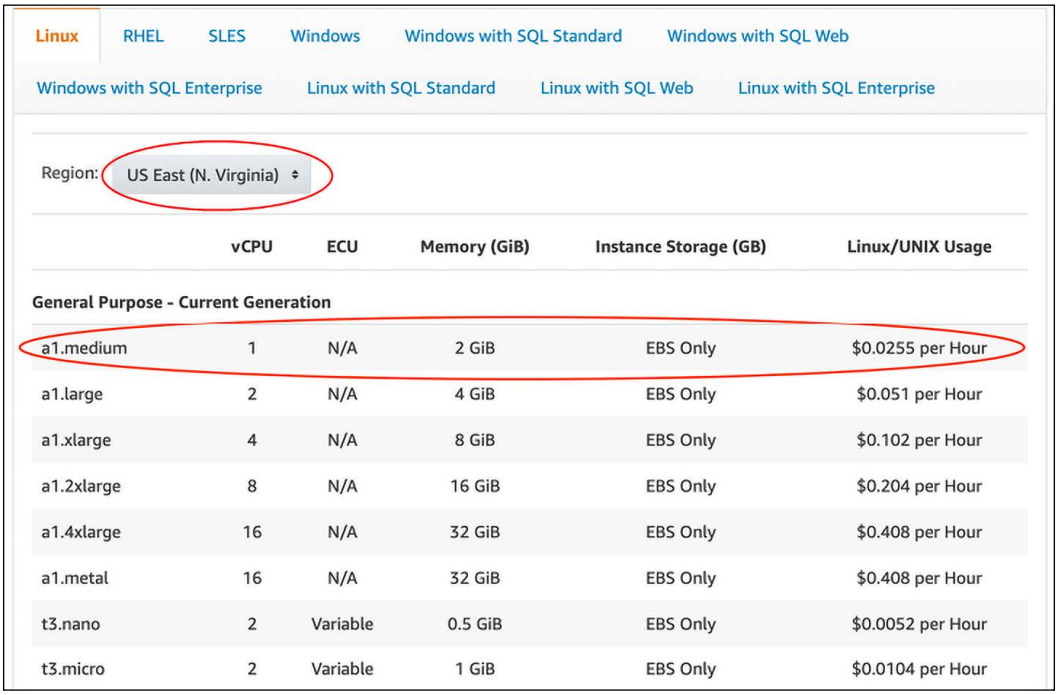


Figure 11.8: AWS services per Region

Besides service availability, the cost of an offering might be slightly different between Regions. For example, for the EC2 service we will look at in this chapter, the cost for an **a1.medium** instance is **USD 0.0255 per hour** in **US East (N. Virginia)**; the same instance costs 14% more, at **USD 0.0291 per hour**, in **EU (Frankfurt)**:



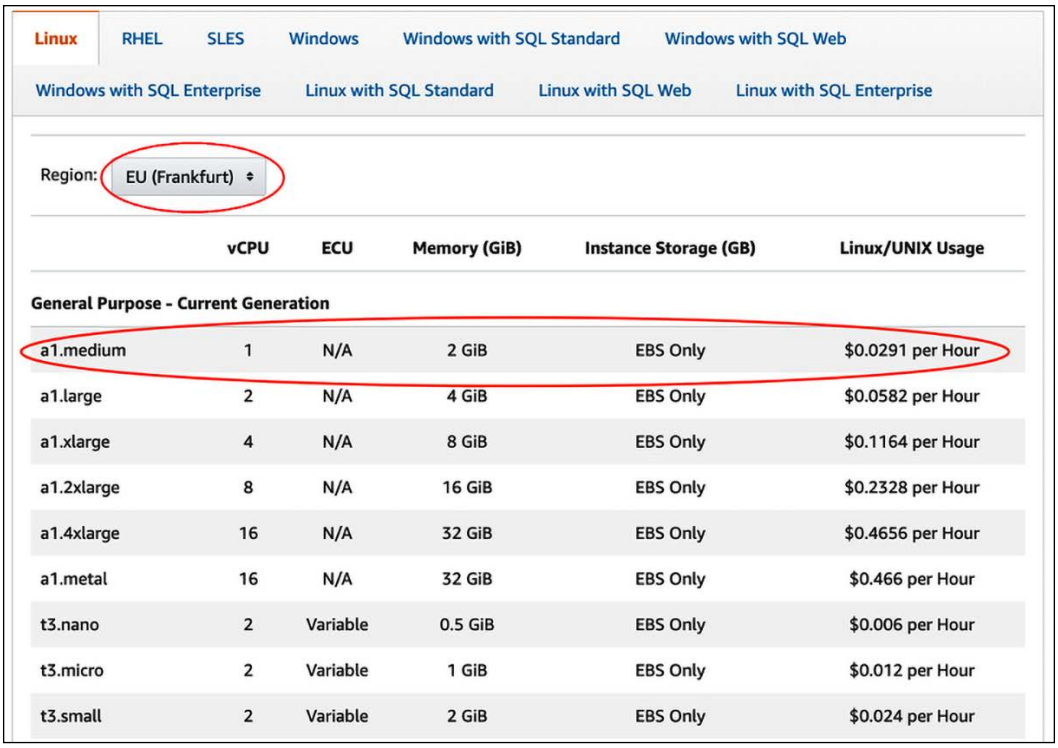
Linux RHEL SLES Windows Windows with SQL Standard Windows with SQL Web

Windows with SQL Enterprise Linux with SQL Standard Linux with SQL Web Linux with SQL Enterprise

Region: US East (N. Virginia)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
a1.medium	1	N/A	2 GiB	EBS Only	$0.0255 per Hour
a1.large	2	N/A	4 GiB	EBS Only	$0.051 per Hour
a1.xlarge	4	N/A	8 GiB	EBS Only	$0.102 per Hour
a1.2xlarge	8	N/A	16 GiB	EBS Only	$0.204 per Hour
a1.4xlarge	16	N/A	32 GiB	EBS Only	$0.408 per Hour
a1.metal	16	N/A	32 GiB	EBS Only	$0.408 per Hour
t3.nano	2	Variable	0.5 GiB	EBS Only	$0.0052 per Hour
t3.micro	2	Variable	1 GiB	EBS Only	$0.0104 per Hour

Figure 11.9: AWS EC2 US East price



Linux | RHEL | SLES | Windows | Windows with SQL Standard | Windows with SQL Web | Windows with SQL Enterprise | Linux with SQL Standard | Linux with SQL Web | Linux with SQL Enterprise

Region: EU (Frankfurt)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
a1.medium	1	N/A	2 GiB	EBS Only	$0.0291 per Hour
a1.large	2	N/A	4 GiB	EBS Only	$0.0582 per Hour
a1.xlarge	4	N/A	8 GiB	EBS Only	$0.1164 per Hour
a1.2xlarge	8	N/A	16 GiB	EBS Only	$0.2328 per Hour
a1.4xlarge	16	N/A	32 GiB	EBS Only	$0.4656 per Hour
a1.metal	16	N/A	32 GiB	EBS Only	$0.466 per Hour
t3.nano	2	Variable	0.5 GiB	EBS Only	$0.006 per Hour
t3.micro	2	Variable	1 GiB	EBS Only	$0.012 per Hour
t3.small	2	Variable	2 GiB	EBS Only	$0.024 per Hour

Figure 11.10: AWS EC2 EU price

When in doubt, choose US East (N. Virginia); it is the oldest Region and most likely the cheapest, with the most service offerings.

Not all Regions are available to all users. For example, **GovCloud** and the **China** Region are not available to users in the United States by default. You can list the Regions available to you via `aws ec2 describe-regions`:

```
$ aws ec2 describe-regions
{
    "Regions": [
        {
            "Endpoint": "ec2.eu-north-1.amazonaws.com",
            "RegionName": "eu-north-1",
            "OptInStatus": "opt-in-not-required"
        },
        {
            "Endpoint": "ec2.ap-south-1.amazonaws.com",
            "RegionName": "ap-south-1",
            "OptInStatus": "opt-in-not-required"
        },
<skip>
```

As stated by Amazon, all Regions are completely independent of one another. Therefore, most resources are not replicated across Regions. This means that if we have multiple Regions offering the same service, say **US-East** and **US-West**, and need the services to back each other up, we will need to replicate the necessary resources ourselves.

We can choose our desired Region in the AWS console, in the top-right corner, with the drop-down menu:

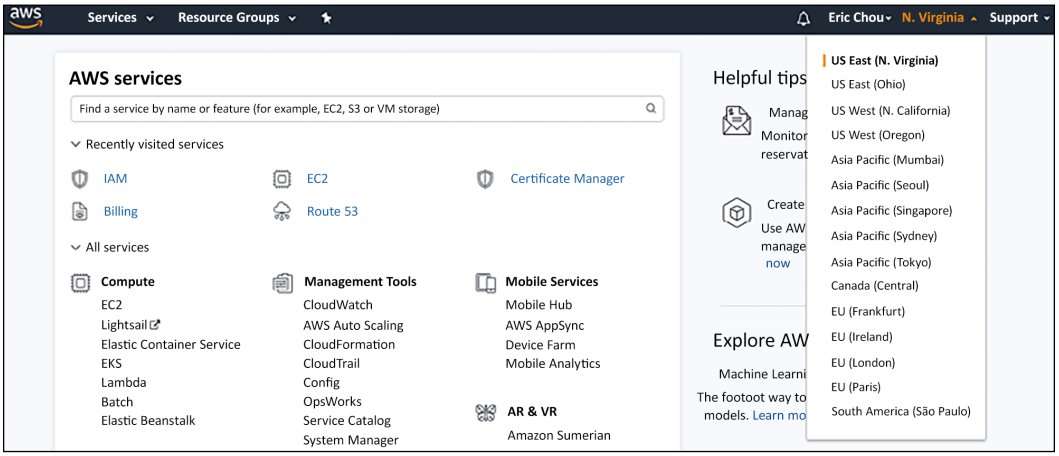


Figure 11.11: AWS Regions

We can only view the services available within the Region on the portal. For example, if we have EC2 instances in the US East Region and select the US West Region, none of our EC2 instances will show up. I have made this mistake several times and wondered where all of my instances went!

There are many AZs within each Region. AZs are labeled using a combination of the Region and an alphabetical letter, such as `us-east-1a`, `us-east-1b`, and so on. Each Region has multiple AZs – typically three or more. Each AZ has its isolated infrastructure with a redundant power supply, intra-data center networking, and facilities. All AZs in a Region are connected through low-latency fiber routes that are typically within 100 km of each other within the same Region:

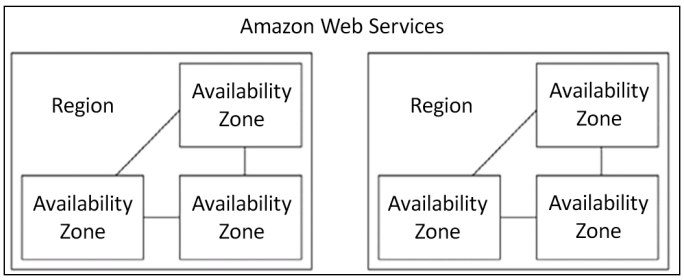


Figure 11.12: AWS Regions and AZs

Unlike Regions, many of the resources we build in AWS can be copied across AZs automatically. For example, we can configure our managed relational database (Amazon RDS) to be replicated across AZs. The concept of AZs is very important when it comes to service redundancy, and its constraints are important to us for the network services we will build.

AWS independently maps AZs to identifiers for each account. For example, my AZ, `us-east-1a`, might not be the same as `us-east-1a` for another account, even though they are both labeled as `us-east-1a`.

We can check the AZs in a Region in the AWS CLI:

```
$ aws ec2 describe-availability-zones --region us-east-1
{
    "AvailabilityZones": [
        {
            "State": "available",
            "Messages": [],
            "RegionName": "us-east-1",
            "ZoneName": "us-east-1a",
            "ZoneId": "use1-az2"
        },
```

```
        {
            "State": "available",
            "Messages": [],
            "RegionName": "us-east-1",
            "ZoneName": "us-east-1b",
            "ZoneId": "use1-az4"
        },
<skip>
```

Why do we care about Regions and AZs so much? As we will see in the coming few sections, AWS networking services are usually bound by the Region and AZ. A **virtual private cloud** (**VPC**), for example, must reside entirely in one Region, and each subnet needs to reside entirely in one AZ. On the other hand, NAT gateways are AZ-bound, so we will need to create one per AZ if we need redundancy.

We will go over both services in more detail, but their use cases are offered here as examples of how Regions and AZs are the basis of the AWS network services offering:

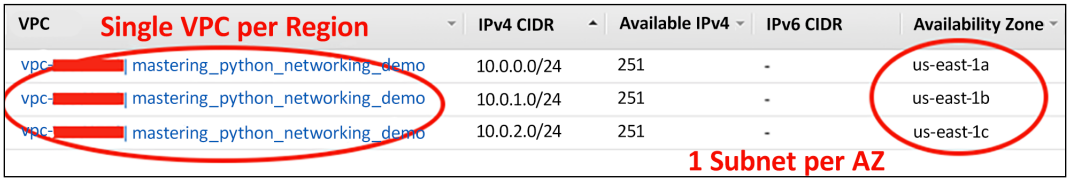


Figure 11.13: VPCs and AZs per Region

AWS edge locations are part of the **AWS CloudFront** content delivery network in 90+ cities across 48 countries as of May 2022 (`https://aws.amazon.com/cloudfront/features/`). These edge locations are used to distribute content with low latency to customers. The edge nodes have a smaller footprint than the full data center Amazon builds for the Region and AZs. Sometimes, people mistake the edge locations' point-of-presence for full AWS Regions. If the footprint is listed as an edge location, AWS services such as EC2 or S3 will not be offered. We will revisit edge locations in the **AWS CloudFront CDN services** section.

AWS transit centers are one of the least documented aspects of AWS networks. They were mentioned in James Hamilton's 2014 AWS re:Invent keynote (`www.youtube.com/watch?v=JIQETrFC_SQ`) as the aggregation points for different AZs in the Region. To be fair, we do not know if the transit center still exists and functions the same way after all these years. However, it is fair to make an educated guess about the placement of the transit center and its correlation with the AWS Direct Connect service, which we will look at later in this chapter.



James Hamilton, a VP and distinguished engineer from AWS, is one of the most influential technologists at AWS. If there is anybody whom I would consider authoritative when it comes to AWS networking, it would be him. You can read more about his ideas on his blog, Perspectives, at `https://perspectives.mvdirona.com/`.

It is impossible to cover all of the services related to AWS in one chapter. There are some relevant services not directly related to networking that we do not have the space to cover, but we should be familiar with:

- The IAM service, `https://aws.amazon.com/iam/`, is the service that enables us to manage access to AWS services and resources securely.
- **Amazon Resource Names (ARNs)**, `https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html`, uniquely identify AWS resources across all of AWS. These resource names are important when we need to identify a service, such as DynamoDB and API Gateway, that needs access to our VPC resources.
- Amazon **Elastic Compute Cloud (EC2)**, `https://aws.amazon.com/ec2/`, is the service that enables us to obtain and provision compute capacities, such as Linux and Windows instances, via AWS interfaces. We will use EC2 instances throughout this chapter in our examples.

For the sake of learning, we will exclude the AWS GovCloud (US) and China Regions, neither of which uses the AWS global infrastructure, and each has its own unique features and limitations.

This was a relatively long introduction to AWS network services, but an important one. These concepts and terms will be referred to in the rest of the chapters. In the upcoming section, we will look at the most important concept (in my opinion) in AWS networking: VPC.

Virtual Private Cloud

Amazon VPC (`https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html`) enables customers to launch AWS resources in a virtual network dedicated to the customer's account. It is truly a customizable network that allows you to define your IP address range, add and delete subnets, create routes, add VPN gateways, associate security policies, connect EC2 instances to your own data center, and much more.

In the early days, when VPC was unavailable, all EC2 instances in an AZ were on a single, flat network that was shared among all customers. How comfortable would the customer be with putting their information in the cloud? Not very, I'd imagine. Between the launch of EC2 in 2007 and the launch of VPC in 2009, VPC functions were some of the most requested features of AWS.

The packets leaving your EC2 host in a VPC are intercepted by the Hypervisor. The Hypervisor will check the packets against a mapping service that understands your VPC construct. Then, the packets are encapsulated with the real AWS servers' source and destination addresses. The encapsulation and mapping service enables the flexibility of VPC but also some of the limitations (multicast, sniffing) of VPC. This is, after all, a virtual network.

Since December 2013, all EC2 instances are VPC-only; you can no longer create an EC2 instance that is non-VPC (EC2-Classic), nor would you want to. If we use a launch wizard to create our EC2 instance, it will automatically be put into a default VPC with a virtual internet gateway for public access. In my opinion, only the most basic use cases should use the default VPC. In most cases, we should define our own non-default, customized VPC.

Let's create the following VPC using the AWS console in **us-east-1**:

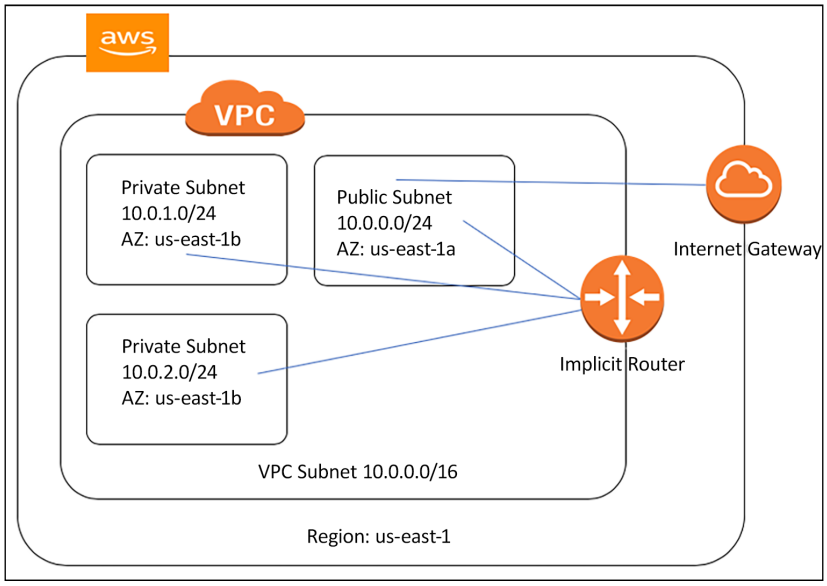


Figure 11.14: Our first VPC in US-East-1

If you recall, VPC is AWS Region-bound, and the subnets are AZ-based. Our first VPC will be based in `us-east-1`; the three subnets will be allocated to two different AZs in `us-east-1a` and `us-east-1b`.

Using the AWS console to create the VPC and subnets is pretty straightforward, and AWS provides several good tutorials online. I have listed the steps with the associated locations of each on the VPC dashboard:

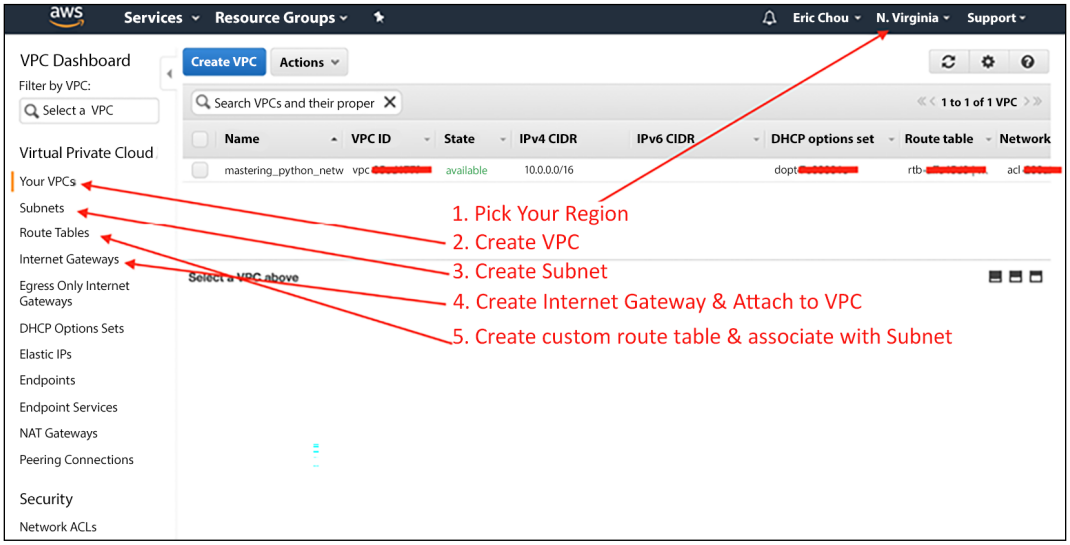


Figure 11.15: Steps for creating the VPC, subnet, and other features

The first two steps are point-and-click processes that most network engineers can work through, even without prior experience. By default, the VPC only contains the local route, `10.0.0.0/16`. Now, we will create an internet gateway and associate it with the VPC:

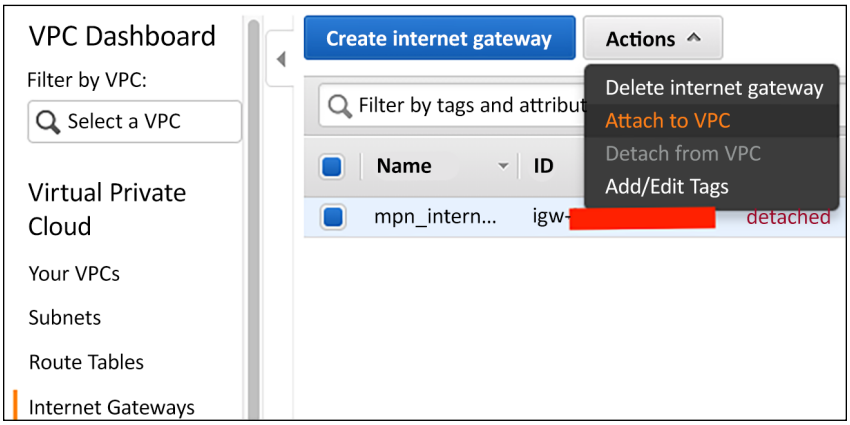


Figure 11.16: AWS internet gateway-to-VPC assignment

We can then create a custom route table with a default route pointing to the internet gateway, allowing internet access. We will associate this route table with our subnet in `us-east-1a`, `10.0.0.0/24`, thus allowing the VPC to have internet access:

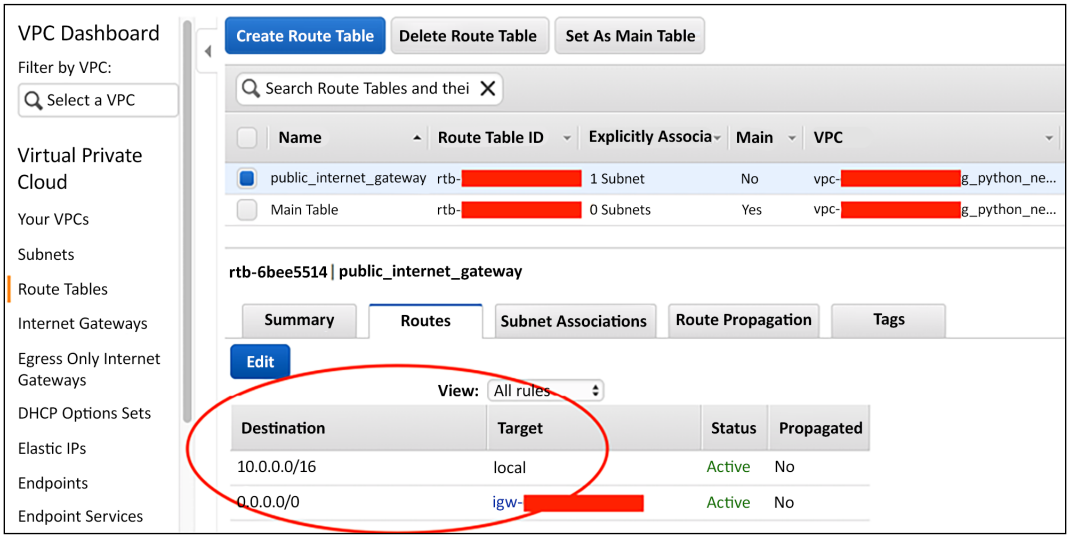


Figure 11.17: Route table

Let's use the Boto3 Python SDK to see what we have created; I used `mastering_python_networking_demo` as the tag for the VPC, which we can use as the filter:

```
#!/usr/bin/env python3
import json, boto3
region = 'us-east-1'
vpc_name = 'mastering_python_networking_demo'
ec2 = boto3.resource('ec2', region_name=region)
client = boto3.client('ec2')
filters = [{'Name':'tag:Name', 'Values':[vpc_name]}]
vpcs = list(ec2.vpcs.filter(Filters=filters))
for vpc in vpcs:
    response = client.describe_vpcs(
                VpcIds=[vpc.id,]
               )
    print(json.dumps(response, sort_keys=True, indent=4))
```

This script will allow us to query the Region for the VPC we created programmatically:

```
(venv) $ python Chapter11_1_query_vpc.py
{
    " ResponseMetadata " : {
        <skip>
        " HTTPStatusCode " : 200,
        " RequestId " : " 9416b03f-<skip> " ,
        " RetryAttempts " : 0
    },
    " Vpcs " : [
        {
            " CidrBlock " : " 10.0.0.0/16 ",
            " CidrBlockAssociationSet " : [
                {
                    " AssociationId " : " vpc-cidr-assoc-<skip> ",
                    "CidrBlock": "10.0.0.0/16",
                    "CidrBlockState": {
                        "State": "associated"
                    }
                }
            ],
            "DhcpOptionsId": "dopt-<skip>",
            "InstanceTenancy": "default",
            "IsDefault": false,
            "OwnerId": "<skip>",
            "State": "available",
            "Tags": [
                {
                    "Key": "Name",
                    "Value": "mastering_python_networking_demo"
                }
            ],
            "VpcId": "vpc-<skip>"
        }
    ]
}
```

The Boto3 VPC API documentation can be found at `https://boto3.readthedocs.io/en/latest/reference/services/ec2.html#vpc`.

If we created EC2 instances and put them in different subnets as is, the hosts would be able to reach each other across subnets. You may be wondering how the subnets can reach one another within the VPC since we only created an internet gateway in subnet 1a. In a physical network, the network needs to connect to a router to reach beyond its own local network.

It is not so different in VPC, except it is an **implicit router** with a default routing table of the local network, which in our example is `10.0.0.0/16`. This implicit router was created when we created our VPC. Any subnet that is not associated with a custom routing table is associated with the main table.

Route tables and route targets

Routing is one of the most important topics in network engineering. It is worth looking at how it is done in AWS VPC more closely. We've already seen that we had an implicit router and the main routing table when we created the VPC. In the last example, we created an internet gateway, a custom routing table with a default route pointing to the internet gateway using the route target, and we associated the custom routing table with a subnet.

So far, only the concept of the route target is where VPC is a bit different than traditional networking. We can roughly equate the route target with the next hop in traditional routing.

In summary:

- Each VPC has an implicit router
- Each VPC has the main routing table with the local route populated
- You can create custom-routing tables
- Each subnet can follow a custom-routing table or the default main routing table
- The route table route target can be an internet gateway, NAT gateway, VPC peers, and so on

We can use Boto3 to look at the custom route tables and associations with the subnets in `Chapter11_2_query_route_tables.py`:

```
#!/usr/bin/env python3
import json, boto3
region = 'us-east-1'
vpc_name = 'mastering_python_networking_demo'
ec2 = boto3.resource('ec2', region_name=region)
```

```
client = boto3.client('ec2')
response = client.describe_route_tables()
print(json.dumps(response['RouteTables'][0], sort_keys=True, indent=4))
```

The main routing table is implicit and not returned by the API. Since we only have one custom route table, this is what we will see:

```
(venv) $ python Chapter11_2_query_route_tables.py
{
    " Associations " : [
        <skip>
    ],
    " OwnerId " : " <skip> ",
    " PropagatingVgws " : [],
    " RouteTableId " : " rtb-<skip> ",
    " Routes " : [
        {
            "DestinationCidrBlock": "10.0.0.0/16",
            "GatewayId": "local",
            "Origin": "CreateRouteTable",
            "State": "active"
        },
        {
            "DestinationCidrBlock": "0.0.0.0/0",
            "GatewayId": "igw-041f287c",
            "Origin": "CreateRoute",
            "State": "active"
        }
    ],
    "Tags": [
        {
            "Key": "Name",
            "Value": "public_internet_gateway"
        }
    ],
    "VpcId": "vpc-<skip>"
}
```

We already created the first public subnet. We will create two more private subnets, `us-east-1b` and `us-east-1c`, following the same steps. The result will be three subnets: a `10.0.0.0/24` public subnet in `us-east-1a`, and `10.0.1.0/24` and `10.0.2.0/24` private subnets in `us-east-1b` and `us-east-1c`, respectively.

We now have a working VPC with three subnets: one public and two private. So far, we have used the AWS CLI and the Boto3 library to interact with AWS VPC. Let's take a look at another automation tool from AWS, **CloudFormation**.

Automation with CloudFormation

AWS CloudFormation (`https://aws.amazon.com/cloudformation/`) is one way in which we can use a text file to describe and launch the resource that we need. We can use CloudFormation to provision another VPC in the **us-west-1** Region:

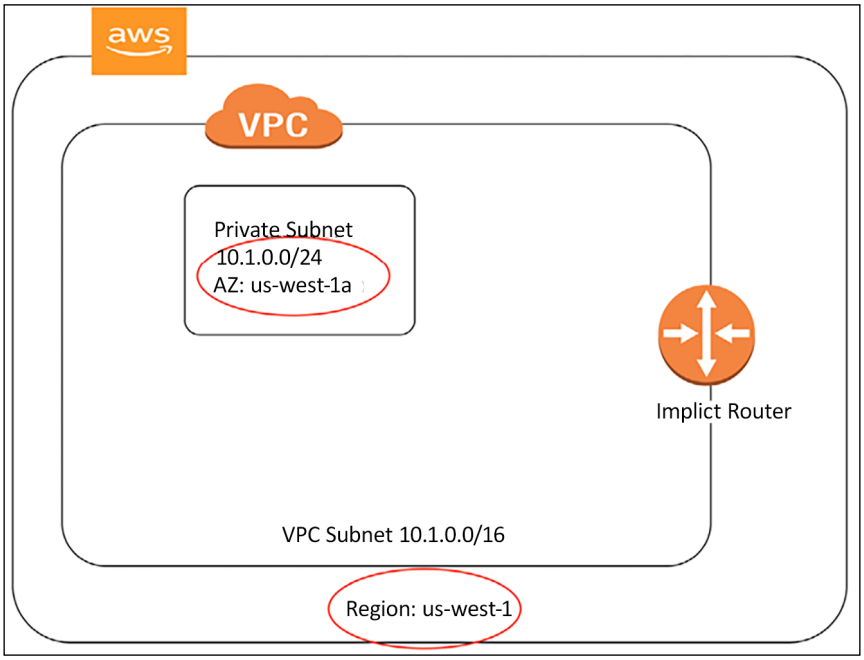


Figure 11.18: VPC for us-west-1

The CloudFormation template can be in YAML or JSON; we will use YAML for our first template for provisioning, `Chapter10_3_cloud_formation.yml`:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Create VPC in us-west-1
Resources:
```

```
  myVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: '10.1.0.0/16'
      EnableDnsSupport: 'false'
      EnableDnsHostnames: 'false'
      Tags:
        - Key: Name
        - Value: 'mastering_python_networking_demo_2'
```

We can execute the template via the AWS CLI. Notice that we specify the `us-west-1` region in our execution:

```
(venv) $ aws --region us-west-1 cloudformation create-stack --stack-name
'mpn-ch10-demo' --template-body file://Chapter11_3_cloud_formation.yml
{
"StackId": "arn:aws:cloudformation:us-west-1:<skip>:stack/mpn-ch10-
demo/<skip>"
}
```

We can verify the status via the AWS CLI:

```
(venv) $ aws --region us-west-1 cloudformation describe-stacks --stack-
name mpn-ch10-demo
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-west-1:<skip>:stack/mpn-
ch10-demo/bbf5abf0-8aba-11e8-911f-500cadc9fefe",
            "StackName": "mpn-ch10-demo",
            "Description": "Create VPC in us-west-1",
            "CreationTime": "2018-07-18T18:45:25.690Z",
            "LastUpdatedTime": "2018-07-18T19:09:59.779Z",
            "RollbackConfiguration": {},
            "StackStatus": "UPDATE_ROLLBACK_COMPLETE",
            "DisableRollback": false,
            "NotificationARNs": [],
            "Tags": [],
            "EnableTerminationProtection": false,
            "DriftInformation": {
```

```
                "StackDriftStatus": "NOT_CHECKED"
            }
        }
    ]
}
```

The last CloudFormation template created a VPC without any subnet. Let's delete that VPC and use the following template, `Chapter11_4_cloud_formation_full.yml`, to create both the VPC and the subnet. Notice that we will not have the VPC-ID before VPC creation, so we will use a special variable to reference the VPC-ID in the subnet creation. This same technique can be used for other resources, such as the routing table and internet gateway:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Create subnet in us-west-1
Resources:
  myVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: '10.1.0.0/16'
      EnableDnsSupport: 'false'
      EnableDnsHostnames: 'false'
      Tags:
        - Key: Name
          Value: 'mastering_python_networking_demo_2'
  mySubnet:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref myVPC
      CidrBlock: '10.1.0.0/24'
      AvailabilityZone: 'us-west-1a'
      Tags:
        - Key: Name
          Value: 'mpn_demo_subnet_1'
```

We can execute and verify the creation of the resources as follows:

```
(venv) $ aws --region us-west-1 cloudformation create-stack --stack-name
mpn-ch10-demo-2 --template-body file://Chapter11_4_cloud_formation_full.
yml
{
"StackId": "arn:aws:cloudformation:us-west-1:<skip>:stack/mpn-ch10- demo-
2/<skip>"
}
$ aws --region us-west-1 cloudformation describe-stacks --stack-name mpn-
ch10-demo-2
{
"Stacks": [
{
"StackStatus": "CREATE_COMPLETE",
...
"StackName": "mpn-ch10-demo-2", "DisableRollback": false
}
]
}
```

We can verify the VPC and subnet information from the AWS console. Remember to pick the right Region from the drop-down menu in the top right-hand corner:

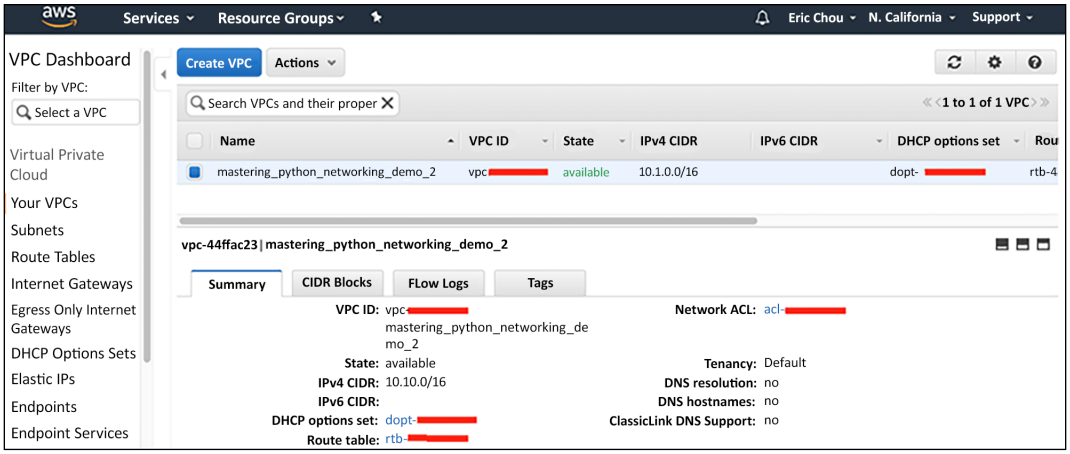


Figure 11.19: VPC in us-west-1

We can also take a look at the subnet:

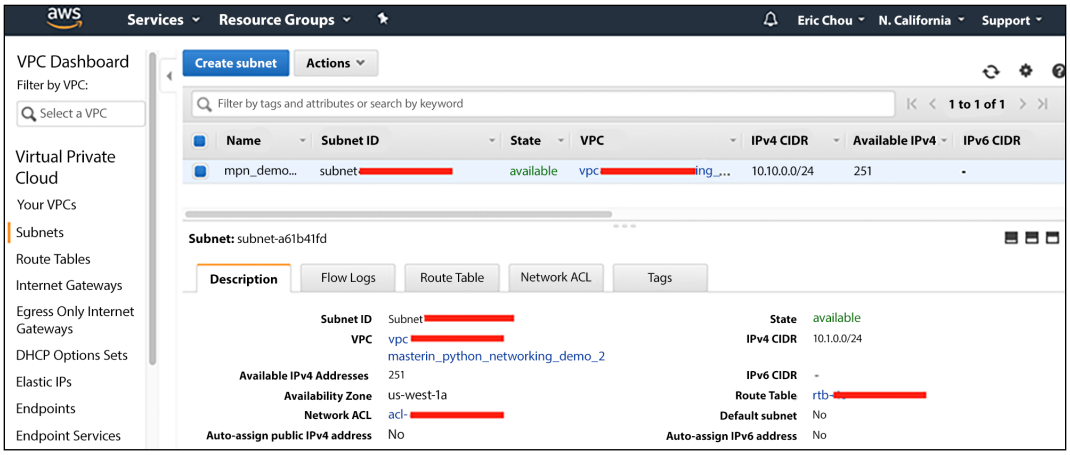


Figure 11.20: Subnet in us-west-1

We now have two VPCs on the two coasts of the United States. They are currently behaving like two islands, each by themselves. This may or may not be your desired state of operation. If we want the two VPCs to be connected, we can use VPC peering (`https://docs.aws.amazon.com/AmazonVPC/latest/PeeringGuide/vpc-peering-basics.html`) to allow direct communication.

There are a few VPC peering limitations, such as no overlapping IPv4 or IPv6 CIDR blocks being allowed. There are also additional limitations for inter-region VPC peering. Make sure you look over the documentation.

VPC peering is not limited to the same account. You can connect VPCs across different accounts, as long as the request was accepted and the other aspects (security, routing, and DNS name) are taken care of.

In the upcoming section, we will take a look at VPC security groups and network **access control lists** (**ACLs**).

Security Groups and Network ACLs

AWS **Security Groups** and **Network ACLs** can be found under the **Security** section of your VPC:

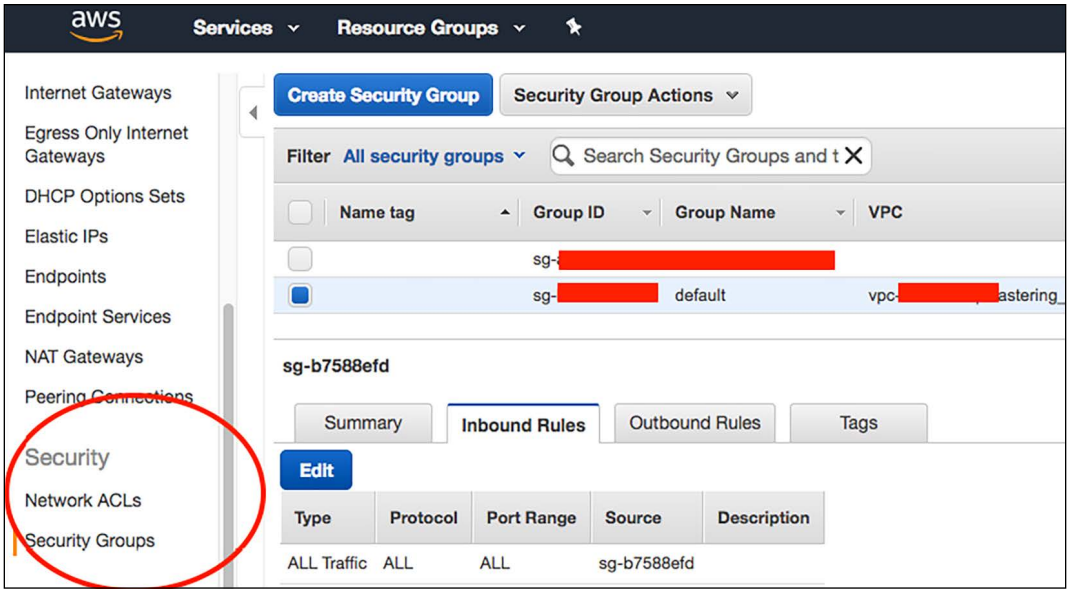


Figure 11.21: VPC security

A security group is a stateful virtual firewall that controls inbound and outbound access to resources. Most of the time, we use a security group to limit public access to our EC2 instance. The current limitation is 500 security groups in each VPC. Each security group can contain up to 50 inbound and 50 outbound rules.

You can use the following sample script, `Chapter11_5_security_group.py`, to create a security group and two simple ingress rules:

```
#!/usr/bin/env python3
import boto3
ec2 = boto3.client('ec2')
response = ec2.describe_vpcs()
vpc_id = response.get('Vpcs', [{}])[0].get('VpcId', '')
# Query for security group id
response = ec2.create_security_group(GroupName='mpn_security_group',
                                     Description='mpn_demo_sg',
                                     VpcId=vpc_id)
security_group_id = response['GroupId']
data = ec2.authorize_security_group_ingress(
    GroupId=security_group_id,
    IpPermissions=[
```

```
        {'IpProtocol': 'tcp',
         'FromPort': 80,
         'ToPort': 80,
         'IpRanges': [{'CidrIp': '0.0.0.0/0'}]},
        {'IpProtocol': 'tcp',
         'FromPort': 22,
         'ToPort': 22,
         'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}
    ])
print('Ingress Successfully Set %s' % data)
# Describe security group
#response = ec2.describe_security_groups(GroupIds=[security_group_id])
print(security_group_id)
```

We can execute the script and receive confirmation of the creation of the security group, which can be associated with other AWS resources:

```
(venv) $ python Chapter11_5_security_group.py
Ingress Successfully Set {'ResponseMetadata': {'RequestId': '<skip>',
'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'AmazonEC2', 'content-
type': 'text/xml;charset=UTF-8', 'date': 'Wed, 18 Jul 2018 20:51:55 GMT',
'content-length': '259'}, 'RetryAttempts': 0}} sg-<skip>
```

Network **ACLs** are an additional layer of security that is stateless. Each subnet in the VPC is associated with a network ACL. Since an ACL is stateless, you will need to specify both inbound and outbound rules.

The important differences between security groups and ACLs are as follows:

- Security groups operate at the network interface level, whereas ACLs operate at the subnet level.
- For a security group, we can only specify `allow` rules and not `deny` rules, whereas ACLs support both `allow` and `deny` rules.
- A security group is stateful, so return traffic is automatically allowed; return traffic in ACLs must be specifically allowed.

Let's look at one of the coolest features of AWS networking: Elastic IP. When I initially learned about Elastic IPs, I was blown away by their ability to assign and reassign IP addresses dynamically.

Elastic IP

An **Elastic IP** (**EIP**) is a way to use a public IPv4 address that's reachable from the internet.



IPv6 is not currently supported in EIP as of late 2022.

An EIP can be dynamically assigned to an EC2 instance, network interface, or other resources. A few characteristics of an EIP are as follows:

- An EIP is associated with the account and is region-specific. For example, an EIP in `us-east-1` can only be associated with resources in `us-east-1`.
- You can disassociate an EIP from a resource and re-associate it with a different resource. This flexibility can sometimes be used to ensure high availability. For example, you can migrate from a smaller EC2 instance to a larger EC2 instance by reassigning the same IP address from the small EC2 instance to the larger one.
- There is a small hourly charge associated with EIPs.

You can request an EIP from the portal. After the assignment, you can associate it with the desired resources:

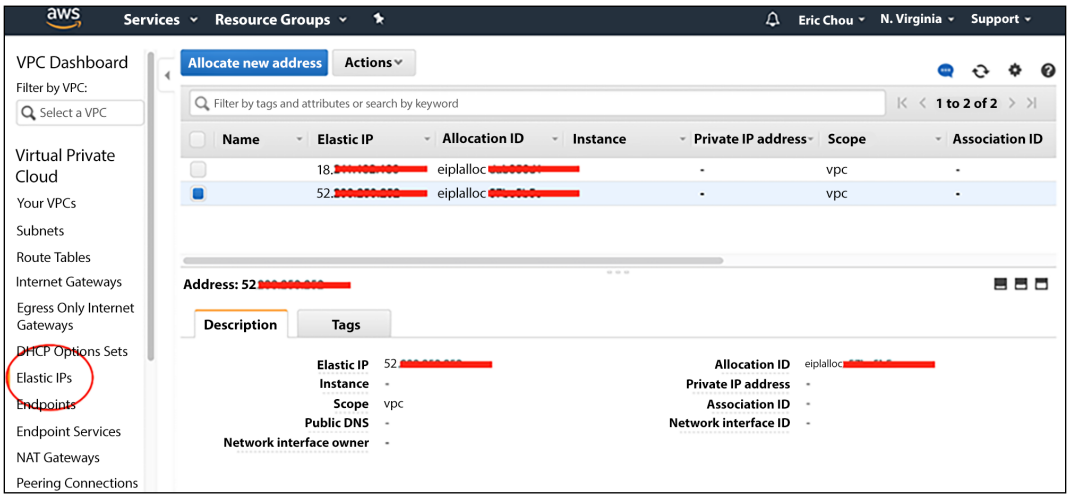


Figure 11.22: Elastic IPs

Unfortunately, EIPs are limited to five per Region to discourage waste (`https://docs.aws.amazon.com/vpc/latest/userguide/amazon-vpc-limits.html`). However, this number can be increased via a ticket to AWS Support if needed.

In the upcoming section, we will look at how we can use NAT gateways to allow communication for private subnets with the internet.

NAT gateways

To allow the hosts in our EC2 public subnet to be accessed from the internet, we can allocate an EIP and associate it with the network interface of the EC2 host. However, at the time of writing, there is a limit of five Elastic IPs per EC2-VPC (`https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Appendix_Limits.html#vpc-limits-eips`). Sometimes, it would be nice to allow the host in a private subnet outbound access when needed, without creating a permanent one-to-one mapping between the EIP and the EC2 host.

A **NAT gateway** can help by allowing the hosts in the private subnet temporary outbound access by performing NAT. This operation is similar to **port address translation** (**PAT**), which we normally perform on the corporate firewall. To use a NAT gateway, we can perform the following steps:

1. Create a NAT gateway in a subnet with access to the internet gateway via the AWS CLI, Boto3 library, or AWS console. The NAT gateway will need to be assigned an EIP.
2. Point the default route in the private subnet to the NAT gateway.
3. The NAT gateway will follow the default route to the internet gateway for external access.

This operation can be illustrated in the following diagram:

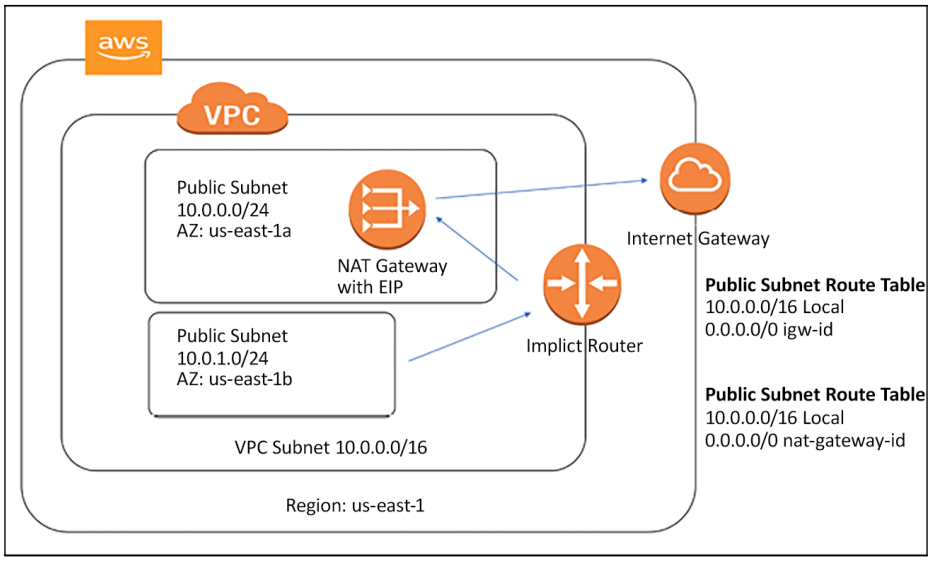


Figure 11.23: NAT gateway operations

One of the most common questions about NAT gateways typically involves which subnet the NAT gateway should reside in. The rule of thumb is to remember that the NAT gateway needs public access. Therefore, it should be created in the subnet with public internet access with an available EIP assigned to it:

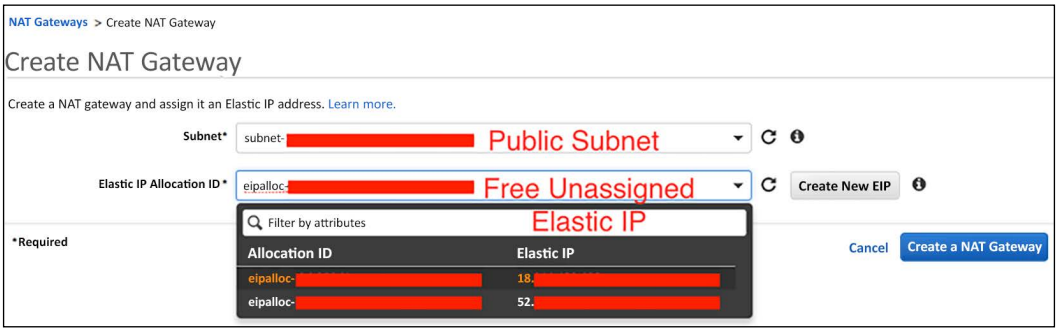


Figure 11.24: NAT gateway creation



Please remember to remove any of the AWS services you are not using to avoid charges.

In the upcoming section, we will look at how to connect our shiny virtual network in AWS to our physical network.

Direct Connect and VPN

Up to this point, our VPC has been a self-contained network that resides in the AWS network. It is flexible and functional, but to access the resources inside the VPC, we will need to access them with their internet-facing services, such as SSH and HTTPS.

In this section, we will look at the ways AWS allows us to connect to the VPC from our private network: an IPSec VPN gateway and Direct Connect.

VPN gateways

The first way to connect our on-premises network to VPC is with traditional IPSec VPN connections. We will need a publicly accessible device to establish VPN connections to AWS's VPN devices.

The customer gateway needs to support route-based IPSec VPNs, where the VPN connection is treated as a connection that a routing protocol and normal user traffic can traverse. Currently, AWS recommends using **Border Gateway Protocol** (**BGP**) to exchange routes.

On the VPC side, we can follow a similar routing table where we can route a particular subnet toward the **virtual private gateway (VPG)** target:

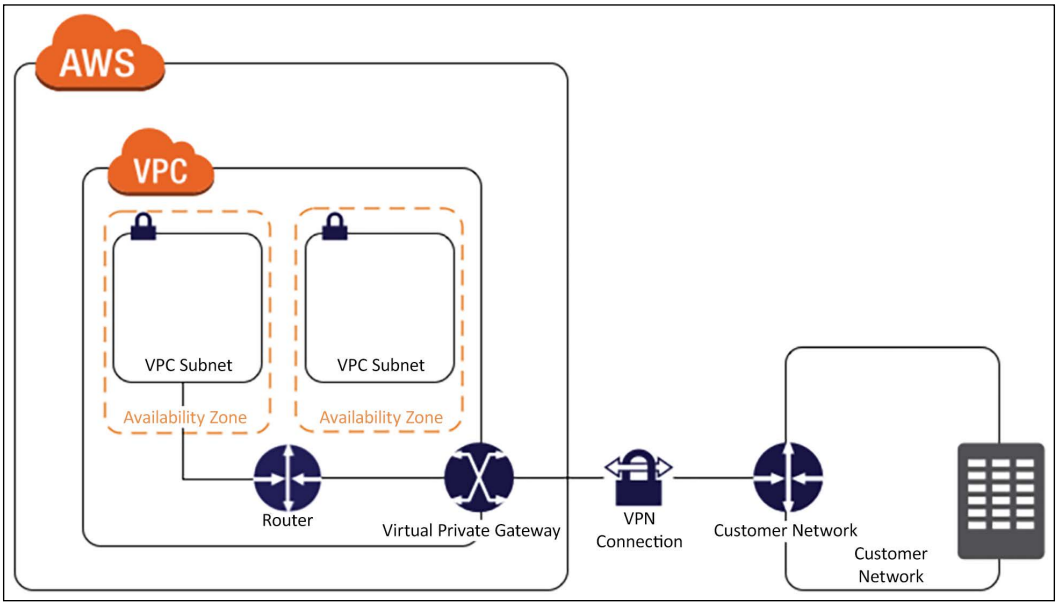


Figure 11.25: VPC VPN connection

Besides an IPSec VPN, we can also use a dedicated circuit to connect, which is termed **Direct Connect**.

Direct Connect

The IPSec VPN connection we looked at is an easy way to provide connectivity for on-premises equipment to AWS cloud resources. However, it suffers the same faults that IPSec over the internet always does: it is unreliable, and we have very little control over its reliability. There is very little performance monitoring and no **service-level agreement (SLA)** until the connection reaches a part of the internet that we can control.

For all of these reasons, any production-level, mission-critical traffic is more likely to traverse through the second option Amazon provides, that is, AWS Direct Connect. AWS Direct Connect lets customers connect their data center and colocation to their AWS VPC with a dedicated virtual circuit.

The somewhat difficult part of this operation is usually bringing our network to where we can connect with AWS physically, typically in a carrier hotel.

You can find a list of the AWS Direct Connect locations here: `https://aws.amazon.com/directconnect/details/`. The Direct Connect link is just a fiber patch connection that you can order from the particular carrier hotel to patch the network to a network port and configure the dot1q trunk's connectivity.

There are also increasingly more connectivity options for Direct Connect via a third-party carrier with **Multi-Protocol Label Switching** (**MPLS**) circuits and aggregated links. One of the most affordable options that I found and use is Equinix Cloud Exchange Fabric (`https://www.equinix.com/services/interconnection-connectivity/cloud-exchange/`). By using Equinix Cloud Exchange Fabric, we can leverage the same circuit and connect to different cloud providers at a fraction of the cost of dedicated circuits:

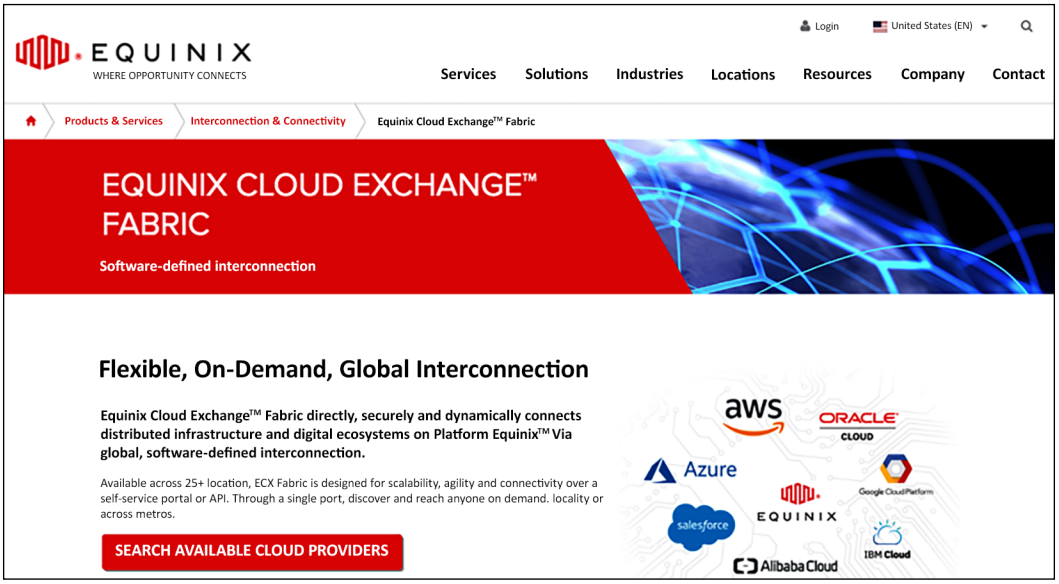


Figure 11.26: Equinix Cloud Exchange Fabric

In the upcoming section, we will look at some of the network scaling services AWS offers.

Network scaling services

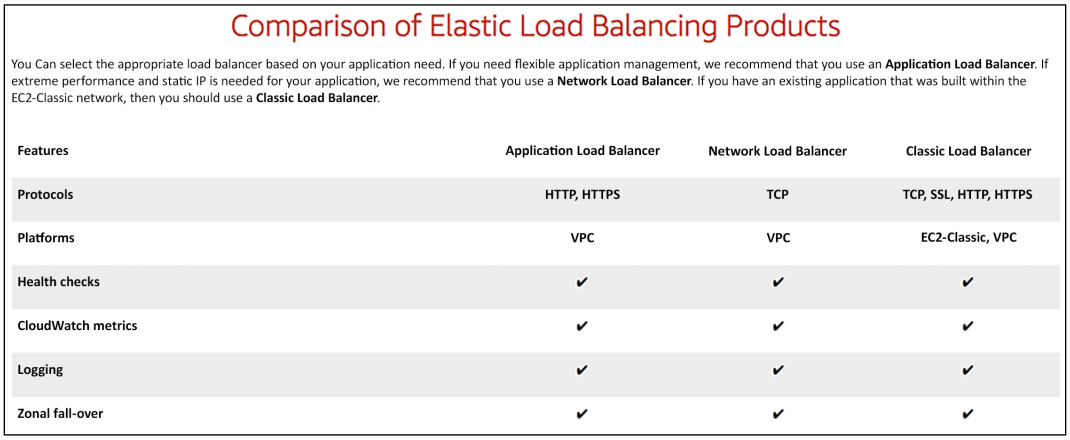
Many of the network services AWS offers do not have direct network implications, such as DNS and content distribution networks. They are relevant in our discussion due to their close relationship with the network and the application's performance.

Elastic Load Balancing

Elastic Load Balancing (**ELB**) allows incoming traffic from the internet to be automatically distributed across multiple EC2 instances. Like load balancers in the physical world, this allows us to have better redundancy and fault tolerance while reducing the per-server load. ELB comes in two flavors: application and network load balancing.

The network load balancer handles web traffic via **HTTP** and **HTTPS**; the application load balancer operates on a TCP level. If your application runs on **HTTP** or **HTTPS**, it is generally a good idea to go with the Application Load Balancer. Otherwise, using the Network Load Balancer is a good bet.

A detailed comparison of the application and Network Load Balances can be found at `https://aws.amazon.com/elasticloadbalancing/details/`:



Comparison of Elastic Load Balancing Products

You Can select the appropriate load balancer based on your application need. If you need flexible application management, we recommend that you use an **Application Load Balancer**. If extreme performance and static IP is needed for your application, we recommend that you use a **Network Load Balancer**. If you have an existing application that was built within the EC2-Classic network, then you should use a **Classic Load Balancer**.

Features	Application Load Balancer	Network Load Balancer	Classic Load Balancer
Protocols	HTTP, HTTPS	TCP	TCP, SSL, HTTP, HTTPS
Platforms	VPC	VPC	EC2-Classic, VPC
Health checks	✔	✔	✔
CloudWatch metrics	✔	✔	✔
Logging	✔	✔	✔
Zonal fall-over	✔	✔	✔

Figure 11.27: ELB comparison

ELB offers a way to load balance traffic once it enters the resource in our Region. The AWS Route 53 DNS service allows geographic load balancing between Regions, sometimes called Global Server Load Balancing.

Route 53 DNS service

We all know what domain name services are – Route 53 is AWS's DNS service. Route 53 is a full-service domain registrar where you can purchase and manage domains directly from AWS. Regarding network services, DNS allows a way to load balance between geographic regions using service domain names in a round-robin fashion between Regions.

We need the following items before we can use DNS for load balancing:

- A load balancer in each of the intended load balance Regions

- A registered domain name. We do not need Route 53 to be the domain registrar
- Route 53 is the DNS service for the domain

We can then use the Route 53 latency-based routing policy with a health check in an active-active environment between the two elastic load balancers. In the next section, we will focus on the content delivery network built by AWS, called CloudFront.

CloudFront CDN services

CloudFront is Amazon's **content delivery network (CDN)**, which reduces the latency of content delivery by physically serving the content closer to the customer. The content can be static web page content, videos, applications, APIs, or, most recently, Lambda functions. CloudFront edge locations include the existing AWS Regions and many other locations around the globe. The high-level operation of CloudFront is as follows:

1. Users access your website for one or more objects.
2. DNS routes the request to the Amazon CloudFront edge location closest to the user's request.
3. The CloudFront edge location will either service the content via the cache or request the object from the origin.

AWS CloudFront and CDN services, in general, are typically handled by application developers or DevOps engineers. However, it is always good to be aware of their operations.

Other AWS network services

There are lots of other AWS network services that we do not have the space to cover here. Some of the more popular services are listed in this section:

- **AWS Transit VPC** (`https://aws.amazon.com/blogs/aws/aws-solution-transit-vpc/`): This is a way to connect multiple VPCs to a common VPC that serves as a transit center. This is a relatively new service, but it can minimize the number of connections that you need to set up and manage. This can also serve as a tool when you need to share resources between separate AWS accounts.
- **Amazon GuardDuty** (`https://aws.amazon.com/guardduty/`): This is a managed threat detection service that continuously monitors for malicious or unauthorized behavior to help protect our AWS workloads. It monitors API calls or potentially unauthorized deployments.
- **AWS WAF** (`https://aws.amazon.com/waf/`): This is a web application firewall that helps protect web applications from common exploits. We can define customized web security rules to allow or block web traffic.

- **AWS Shield** (`https://aws.amazon.com/shield/`): This is a managed **Distributed Denial of Service** (**DDoS**) protection service that safeguards applications running on AWS. The protection service is free for all customers at the basic level; the advanced version of AWS Shield is a fee-based service.

There are lots of new and exciting AWS networking services constantly being announced, such as the ones we have looked at in this section. Not all of them are foundational services such as VPC or NAT gateways; however, they all serve useful purposes in their respective fields.

Summary

In this chapter, we looked at AWS cloud networking services. We reviewed the AWS network definitions of Region, Availability Zone, edge locations, and transit center. Understanding the overall AWS network gives us a good idea of some of the limitations and constraints of the other AWS network services. Throughout this chapter, we used the AWS CLI, the Python Boto3 library, and CloudFormation to automate some tasks.

We covered AWS VPC in depth, with the configuration of the route table and route targets. The example on security groups and network ACLs took care of the security for our VPC. We also looked at EIPs and NAT gateways for allowing external access.

There are two ways to connect AWS VPC to on-premise networks: Direct Connect and IPSec VPN. We briefly looked at each and the advantages of using them. Toward the end of this chapter, we looked at network scaling services offered by AWS, including ELB, Route 53 DNS, and CloudFront.

In the next chapter, we will look at the networking services offered by another public cloud provider, Microsoft Azure.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



12

Azure Cloud Networking

As we saw in *Chapter 11*, *AWS Cloud Networking*, cloud-based networking helps us connect our organization's cloud-based resources. A **virtual network** (**VNet**) can be used to segment and secure our virtual machines. It can also connect our on-premise resources to the cloud. As the first pioneer in this space, AWS is often regarded as the market leader, with the biggest market share. In this chapter, we will look at another important public cloud provider, Microsoft Azure, focusing on their cloud-based network products.

Microsoft Azure originally started as a project codenamed "Project Red Dog" in 2008 and was publicly released on February 1, 2010. At the time, it was named "Windows Azure" before being renamed "Microsoft Azure" in 2014. Since AWS released its first product, S3, in 2006, it essentially had a 6-year lead over Microsoft Azure. Attempting to catch up with AWS was no small task, even for a company with Microsoft's vast amount of resources. At the same time, Microsoft has its unique competitive advantages from years of successful products and relationships with its enterprise customer base.

As Azure focuses on leveraging the existing Microsoft product offerings and customer relationships, there are some important implications regarding Azure cloud networking. For example, one of the main drivers for a customer to establish an ExpressRoute connection with Azure, their AWS Direct Connect equivalent, might be a better experience with Office 365. Another example might be that the customer already has a service-level agreement with Microsoft that can be extended to Azure.

In this chapter, we will discuss the networking services offered by Azure and how we can use Python to work with them. Since we already introduced some of the cloud networking concepts in the last chapter, we will draw on those lessons, comparing AWS and Azure networking when applicable.

In particular, we will discuss:

- The Azure setup and a networking overview.
- Azure **virtual networks** (in the form of VNets). An Azure VNet is similar to an AWS VPC. It provides customers with a private network in the Azure cloud.
- ExpressRoute and VPNs.
- Azure Network Load Balancers.
- Other Azure network services.

We already learned many of the important cloud networking concepts in the last chapter. Let's leverage that knowledge and start by comparing the services offered by Azure and AWS.

Azure and AWS network service comparison

When Azure launched, they were more focused on **Software-as-a-Service** (**SaaS**) and **Platform-as-a-Service** (**PaaS**), with less of a focus on **Infrastructure-as-a-Service** (**IaaS**). For SaaS and PaaS, the networking services at the lower layers are often abstracted away from the user. For example, the SaaS offering of Office 365 is often offered as a remotely hosted endpoint that can be reached over the public internet. The PaaS offering of building web applications using Azure App Service is often done via a fully managed process, via popular frameworks such as .NET or Node.js.

The IaaS offering, on the other hand, requires us to build our infrastructure in the Azure cloud. As the undisputed leader in the space, much of the target audience already has experience with AWS. To help with the transition, Azure provides an "AWS to Azure Service Comparison" (`https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services`) on their website. This is a handy page that I often visit when I am confused about the equivalent Azure offering in comparison to AWS, especially when the service name is not directly illustrative of the service it provides. (I mean, can you tell what SageMaker is from looking at the name? I rest my case.)

I often use this page for competitive analysis as well. For example, when I need to compare the cost of a dedicated connection with AWS and Azure, I start with this page to verify that the equivalent service of AWS Direct Connect is Azure ExpressRoute, then use the link to get more details about the service.

If we scroll down on the page to the **Networking** section, we can see that Azure offers many similar products to AWS, such as VNet, VPN Gateway, and Load Balancer. Some of the services may have different names, such as Route 53 and Azure DNS, but the underlying services are the same.

Networking

Area	AWS service	Azure service	Description
Cloud virtual networking	Virtual Private Cloud (VPC)	Virtual Network	Provides an isolated, private environment in the cloud. Users have control over their virtual networking environment, including selection of their own IP address range, creation of subnets, and configuration of route tables and network gateways.
Cross-premises connectivity	AWS VPN Gateway	Azure VPN Gateway	Connects Azure virtual networks to other Azure virtual networks, or customer on-premises networks (Site To Site). Allows end users to connect to Azure services through VPN tunneling (Point To Site).
DNS management	Route 53	Azure DNS	Manage your DNS records using the same credentials and billing and support contract as your other Azure services
	Route 53	Traffic Manager	A service that hosts domain names, plus routes users to Internet applications, connects user requests to datacenters, manages traffic to apps, and improves app availability with automatic failover.
Dedicated network	Direct Connect	ExpressRoute	Establishes a dedicated, private network connection from a location to the cloud provider (not over the Internet).
Load balancing	Network Load Balancer	Load Balancer	Azure Load Balancer load-balances traffic at layer 4 (TCP or UDP).
	Application Load Balancer	Application Gateway	Application Gateway is a layer 7 load balancer. It supports SSL termination, cookie-based session affinity, and round robin for load-balancing traffic.

Figure 12.1: Azure networking services (source: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services)

There are some feature differences between Azure and AWS networking products. For example, for global traffic load balancing using DNS, AWS uses the same Route 53 product, while Azure breaks it into a separate product called Traffic Manager. When we dig deeper into the products, some differences might make a difference depending on usage. For example, Azure Load Balancer, by default, allows session affinity, a.k.a. a sticky session, whereas the AWS load balancer needs to be configured explicitly.

But for the most part, the high-level network products and services from Azure are similar to what we learned from AWS. This is the good news. The bad news is that just because the features are the same, it does not mean we can have a 1:1 overlay between the two.

The building tools are different, and the implementation details can sometimes throw off someone new to the Azure platform. We will point out some of the differences when we discuss the products in the following sections. Let's begin by talking about the setup process for Azure.

Azure setup

Setting up an Azure account is straightforward. Just like AWS, there are many services and incentives that Azure offers to attract users in the highly competitive public cloud market. Please check out the `https://azure.microsoft.com/en-us/free/` page for the latest offerings. At the time of writing, Azure is offering many popular services free for 12 months and 40+ other services as always free:

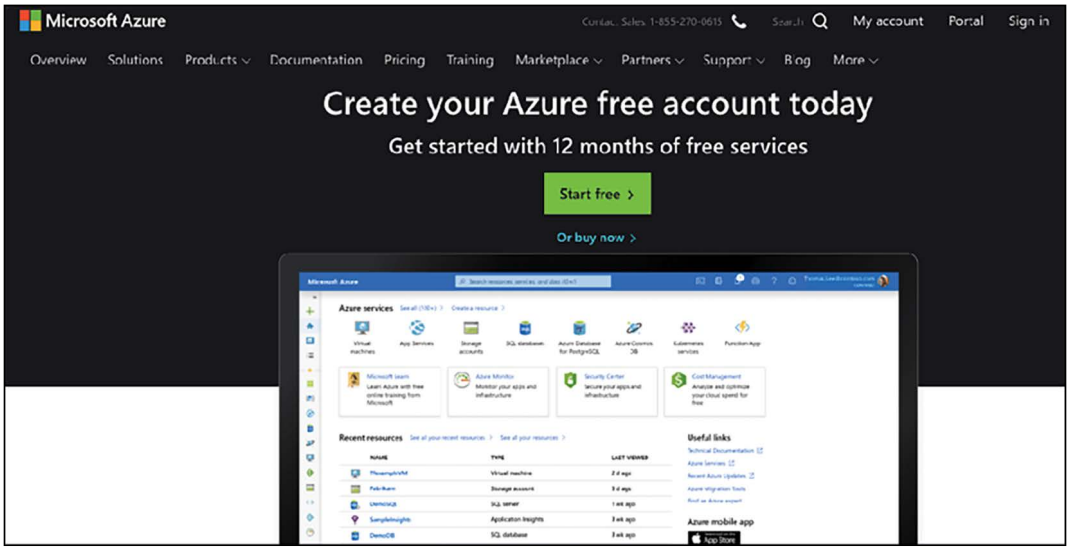


Figure 12.2: Azure portal (source: https://azure.microsoft.com/en-us/free/)

After the account is created, we can see the services available on the Azure portal at `https://portal.azure.com`:

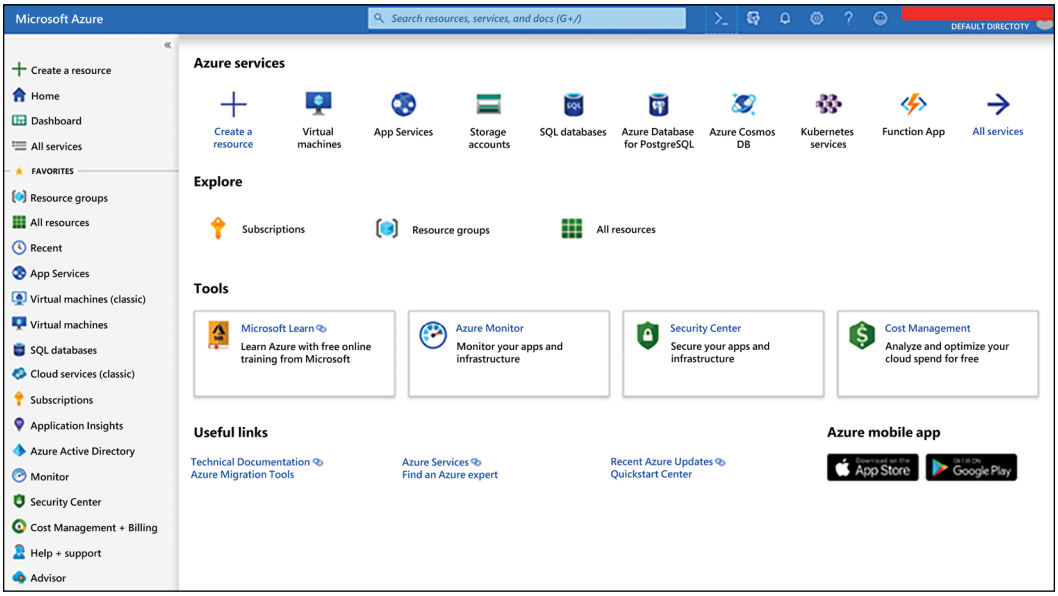


Figure 12.3: Azure services



The web pages might change by the time you read this chapter. They are generally intuitive navigation changes that are easy to maneuver, even if they look a little different.

Before any service can be launched, however, we will need to provide a payment method. This is done by adding a subscription service:

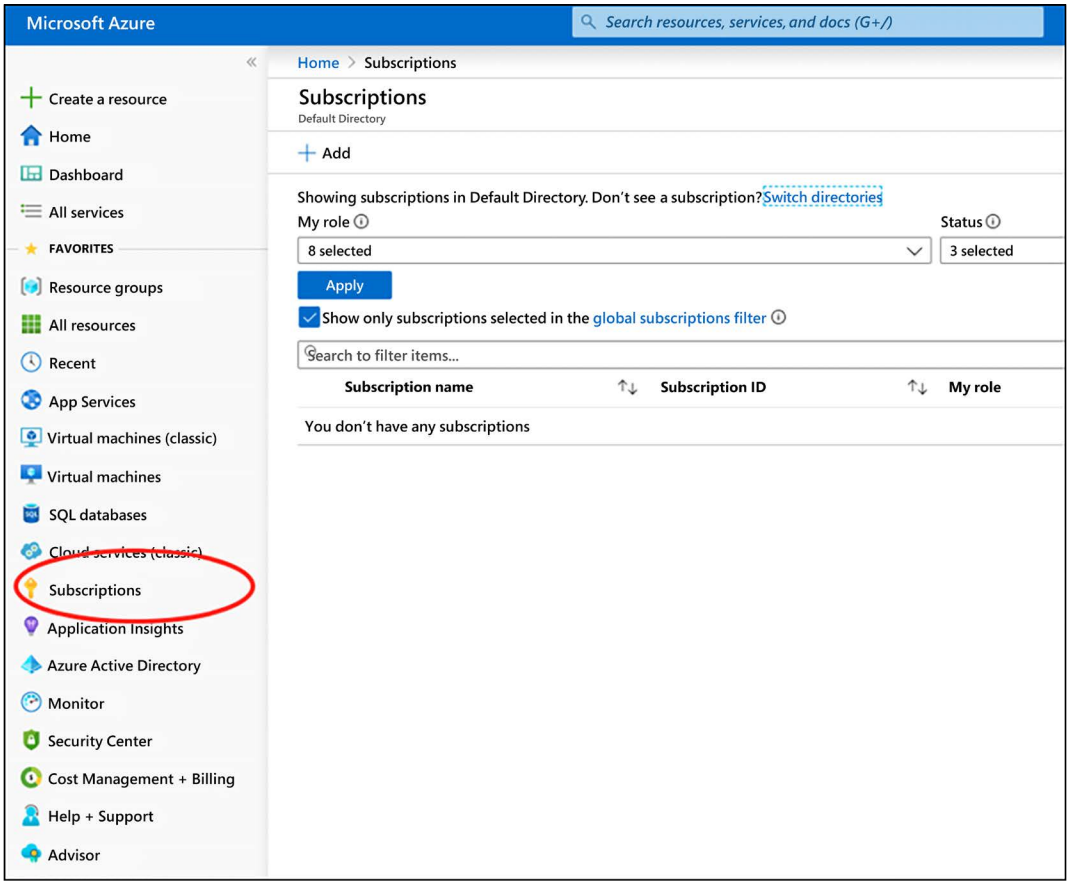


Figure 12.4: Azure subscriptions

I would recommend adding a pay-as-you-go plan, which has no upfront costs and no long-term commitment, but we also have the option to purchase various levels of support with the subscription plan.

Once the subscription is added, we can start looking at the various ways to administer and build in the Azure cloud, as detailed in the following section.

Azure administration and APIs

The Azure portal is the sleekest and most modern portal of the top public cloud providers, including AWS and Google Cloud. We can change the settings of the portal from the settings icon on the top management bar, including the language and region:



Figure 12.5: Azure portal in different languages

There are many ways to manage Azure services: the portal, the Azure CLI, RESTful APIs, and the various client libraries. Besides the point-and-click management interface, the Azure portal also provides a handy shell called Azure Cloud Shell.

It can be launched from the top right-hand corner of the portal:

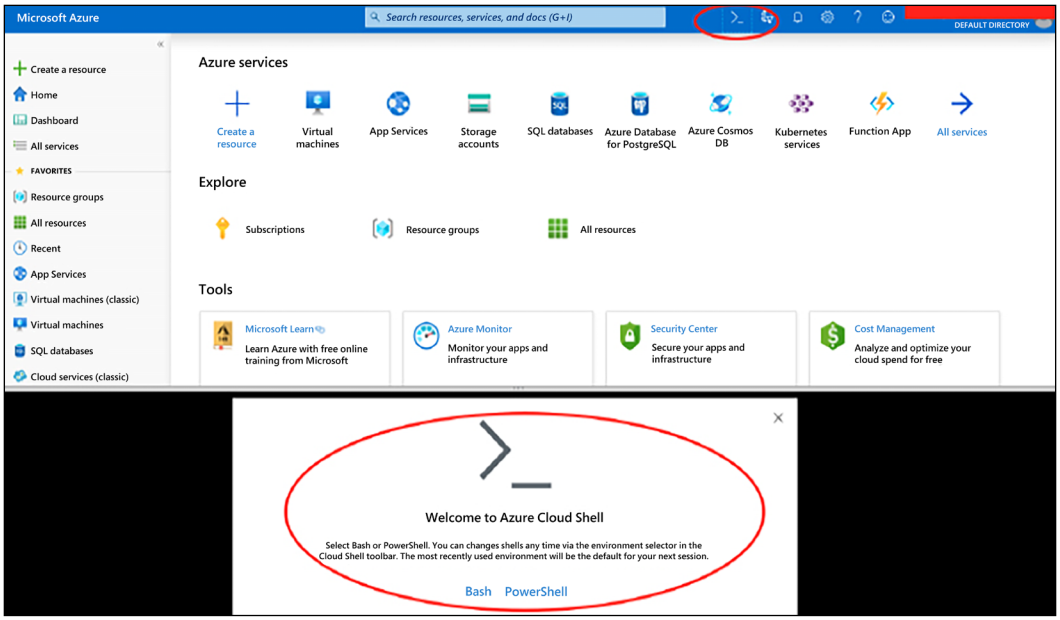


Figure 12.6: Azure Cloud Shell

When it is launched for the first time, you will be asked to pick between **Bash** and **PowerShell**. The shell interface can be switched later, but they cannot run simultaneously:

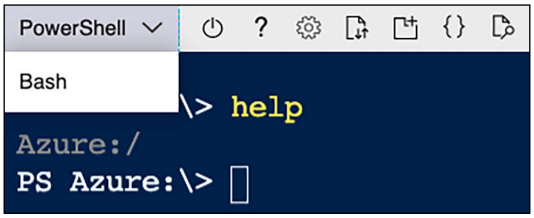


Figure 12.7: Azure Cloud Shell with PowerShell

My personal preference is the **Bash** shell, which allows me to use the pre-installed Azure CLI and Python SDK:

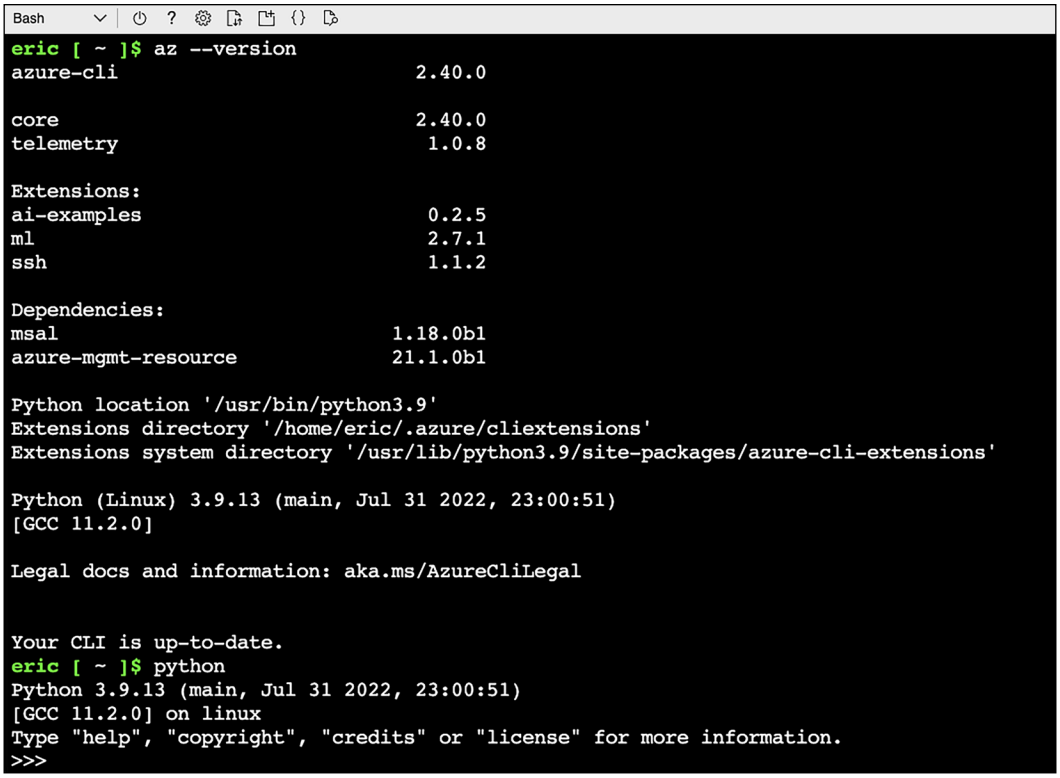


Figure 12.8: Azure AZ tool and Python in Cloud Shell

Cloud Shell is very handy because it is browser-based and thus accessible from virtually anywhere. It is assigned per unique user account and automatically authenticated with each session, so we do not need to worry about generating a separate key for it. But since we will be using the Azure CLI quite often, let's install a local copy on the management host:

```
(venv) $ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
(venv) $ az --version
azure-cli                         2.40.0

core                              2.40.0
telemetry                          1.0.8

Dependencies:
msal                            1.18.0b1
azure-mgmt-resource             21.1.0b1
```

Let's also install the Azure Python SDK on our management host. Starting with version 5.0.0, the Azure Python SDK requires us to install service-specific packages listed at `https://aka.ms/azsdk/python/all`:

```
(venv) $ pip install azure-identity
(venv) $ pip install azure-mgmt-compute
(venv) $ pip install azure-mgmt-storage
(venv) $ pip install azure-mgmt-resource
(venv) $ pip install azure-mgmt-network
```

The Azure for Python Developers page, `https://docs.microsoft.com/en-us/azure/python/`, is an all-inclusive resource for getting started with Azure using Python. The Azure SDK for Python page, `https://learn.microsoft.com/en-us/azure/developer/python/sdk/azure-sdk-overview`, provides detailed documentation on using the Python libraries for Azure resource management.

We are now ready to take a look at some of the service principles of Azure and launch our Azure services.

Azure service principals

Azure uses the concept of service principal objects for automated tools. The network security best practice of least privilege grants any person or tool just enough access to perform their job and no more. An Azure service principal restricts resources and the level of access based on roles. To get started, we will use the role automatically created for us by the Azure CLI and use the Python SDK to test the authentication. Use the `az login` command to receive a token:

```
(venv) $ az login --use-device-code
To sign in, use a web browser to open the page https://microsoft.com/
devicelogin and enter the code <your code> to authenticate.
```

Follow the URL and paste in the code you see on the command line and authenticate with the Azure account we created earlier:

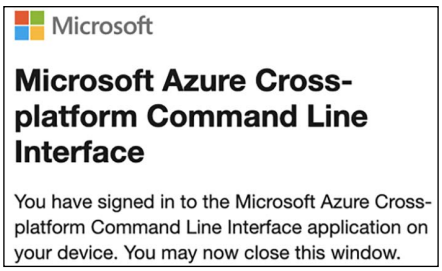


Figure 12.9: Azure Cross-platform Command Line Interface

We can create the credential file in `json` format and move that to the Azure directory. The Azure directory was created when we installed the Azure CLI tool:

```
(venv) $ az ad sp create-for-rbac --sdk-auth > credentials.json
(venv) $ cat credentials.json
{
  "clientId": "<skip>",
  "clientSecret": "<skip>",
  "subscriptionId": "<skip>",
  "tenantId": "<skip>",
  "<skip>"
}
(venv) echou@network-dev-2:~$ mv credentials.json ~/.azure/
```

Let's secure the credential file and export it as an environment variable:

```
(venv) $ chmod 0600 ~/.azure/credentials.json
(venv) $ export AZURE_AUTH_LOCATION=~/.azure/credentials.json
```

We will also export the various credentials into our environment:

```
$ cat ~/.azure/credentials.json
$ export AZURE_TENANT_ID="xxx"
$ export AZURE_CLIENT_ID="xxx"
$ export AZURE_CLIENT_SECRET="xxx"
$ export SUBSCRIPTION_ID="xxx"
```

We will grant role access to the subscription:

```
(venv) $ az ad sp create-for-rbac --role 'Owner' --scopes '/
subscriptions/<subscription id>'
{
  "appId": "<appId>",
  "displayName": "azure-cli-2022-09-22-17-24-24",
  "password": "<password>",
  "tenant": "<tenant>"
}
(venv) $ az login --service-principal --username "<appId>" --password
"<password>" --tenant "<tenant>"
```



For more information on Azure RBAC, visit `https://learn.microsoft.com/en-us/cli/azure/create-an-azure-service-principal-azure-cli`.

If we browse to the **Access control** section in the portal (**Home -> Subscriptions -> Pay-As-You-Go -> Access control**), we will be able to see the newly created role:

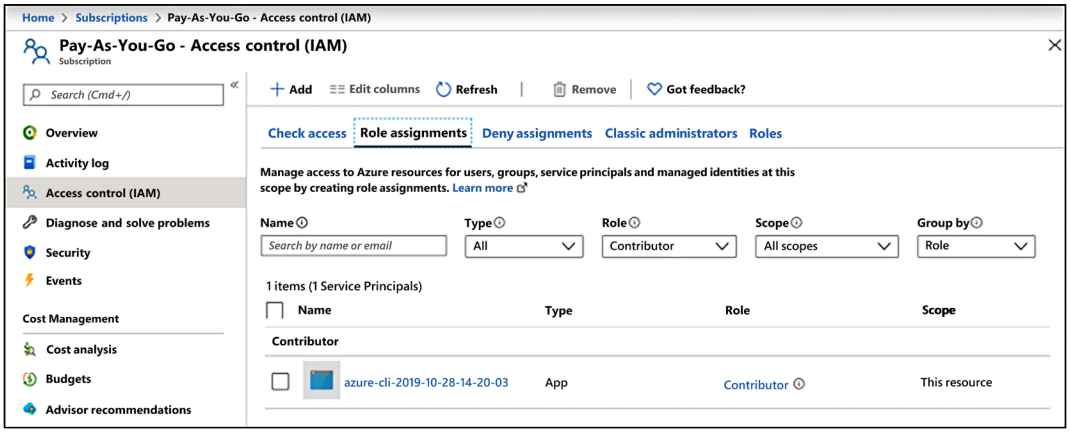


Figure 12.10: Azure pay-as-you-go IAM



There are many example codes for using Python SDK to manage a network using the Azure Python SDK on the GitHub page, `https://github.com/Azure-Samples/azure-samples-python-management/tree/main/samples/network`. The *Getting Started Guide*, `https://learn.microsoft.com/en-us/samples/azure-samples/azure-samples-python-management/network/`, can also be useful.

We will use a simple Python script, `Chapter12_1_auth.py`, to import the library for client authentication and network management:

```
#!/usr/bin/env python3
import os
import azure.mgmt.network
from azure.identity import ClientSecretCredential

credential = ClientSecretCredential(
    tenant_id=os.environ.get("AZURE_TENANT_ID"),
    client_id=os.environ.get("AZURE_CLIENT_ID"),
    client_secret=os.environ.get("AZURE_CLIENT_SECRET")
)
```

```
subscription_id = os.environ.get("SUBSCRIPTION_ID")
network_client = azure.mgmt.network.
NetworkManagementClient(credential=credential, subscription_
id=subscription_id)
print("Network Management Client API Version: " + network_client.DEFAULT_
API_VERSION)
```

If the file executes without an error, we have successfully authenticated with the Python SDK client:

```
(venv) $ python Chapter12_1_auth.py
Network Management Client API Version: 2022-01-01
```

While reading the Azure documentation, you may have noticed a combination of PowerShell and Python. In the next section, let's briefly consider the relationship between Python and PowerShell.

Python versus PowerShell

There are many programming languages and frameworks that Microsoft has either developed from the ground up or has implemented major dialects for, including C#, .NET, and PowerShell. It is no surprise that .NET (with C#) and PowerShell are somewhat first-class citizens in Azure. In much of the Azure documentation, you will find direct references to PowerShell examples. There are often opinionated discussions on the web forums on which tool, Python or PowerShell, is better suited to managing Azure resources.



As of July 2019, we can also run PowerShell Core on the Linux and macOS operating systems in the preview release, `https://docs.microsoft.com/en-us/powershell/scripting/install/installing-powershell-core-on-linux?view=powershell-6`.

We will not get into a debate on language superiority. I do not mind using PowerShell when required – I find it easy and intuitive – and I agree that sometimes the Python SDK lags behind PowerShell in implementing the latest Azure features. But since Python is at least part of the reason you picked up this book, we will stick to the Python SDK and the Azure CLI for our examples.

Initially, the Azure CLI was offered as PowerShell modules for Windows and the Node.js-based CLI for other platforms. But as the tool has grown in popularity, it is now a wrapper around the Azure Python SDK, as explained in this article on *Python.org*: `https://www.python.org/success-stories/building-an-open-source-and-cross-platform-azure-cli-with-python/`.

In the remaining sections of this chapter, when we are introducing a feature or concept, we will oftentimes turn to the Azure CLI for demonstration purposes. Rest assured that if something is available as an Azure CLI command, it is available in the Python SDK if we need to directly code it in Python.

Having covered Azure administration and the associated APIs, let's move on to discussing Azure global infrastructure.

Azure global infrastructure

Similar to AWS, an Azure global infrastructure consists of regions, **Availability Zones (AZs)**, and edge locations. At the time of writing, Azure has 60+ regions and more than 200+ physical data centers, as illustrated on the product page (`https://azure.microsoft.com/en-us/global-infrastructure/`):

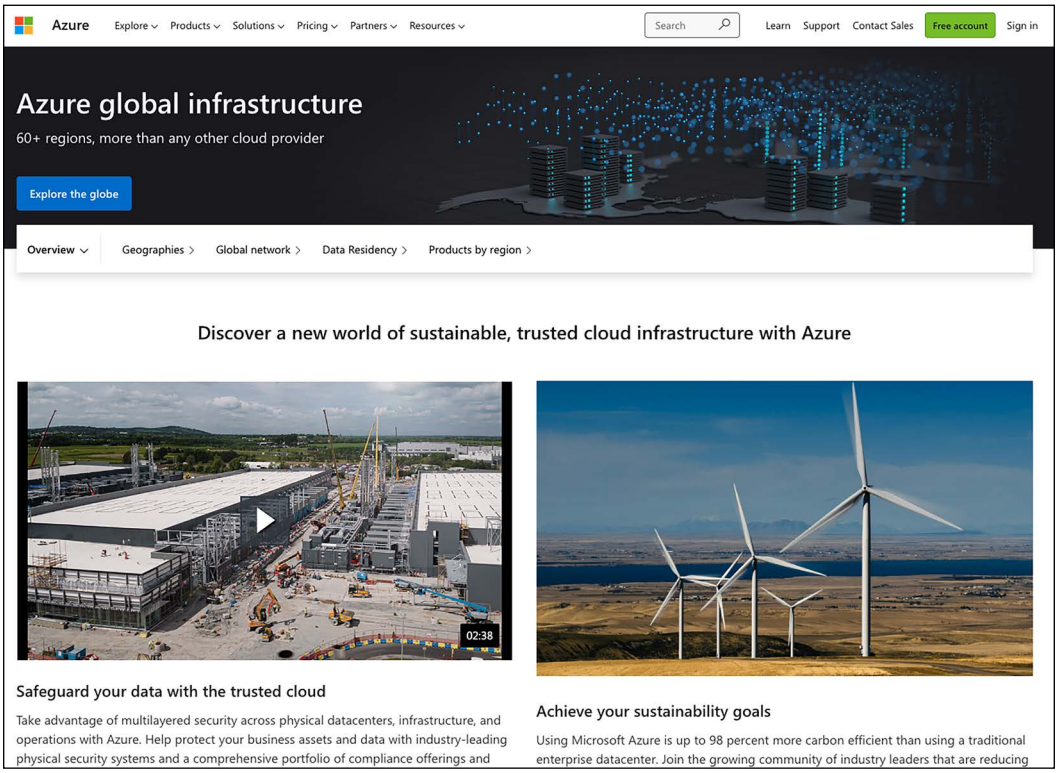


Figure 12.11: Azure global infrastructure (source: https://azure.microsoft.com/en-us/global-infrastructure/)

Like AWS, Azure products are offered via regions, so we need to check service availability and pricing based on regions. We can also build redundancy into the service by building the service in multiple AZs. However, unlike AWS, not all Azure regions have AZs, and not all Azure products support them. In fact, Azure did not announce the general availability of AZs until 2018, and they are only offered in select regions.

This is something to be aware of when picking our region. I recommend picking regions with AZs such as West US 2, Central US, and East US 1.

If we build in a region without AZs, we will need to replicate the service across different regions, typically in the same geography. We will discuss Azure geography next.



On the Azure global infrastructure page, the regions with Availability Zones are marked with a star in the middle.

Unlike AWS, Azure regions are also organized into a higher-order category of geographies. A geography is a discrete market, typically containing one or more regions. Besides lower latency and better network connectivity, replicating the service and data across regions in the same geography is necessary for government compliance. An example of replication across regions would be the regions of Germany. If we needed to launch services for the German market, the government mandates strict data sovereignty within the border, but none of the German regions have Availability Zones. We would need to replicate the data between different regions in the same geography, that is, Germany North, Germany Northeast, Germany West Central, and so on.

As a rule of thumb, I typically prefer regions that have Availability Zones to keep things similar across different cloud providers. Once we have determined the region that best fits our use case, we are ready to build our VNet in Azure.

Azure virtual networks

When we wear the network engineer hat in the Azure cloud, **Azure virtual networks (VNets)** are where we spend most of our time. Similar to a traditional network that we would build in our data center, they are the fundamental building blocks for our private networks in Azure. We will use a VNet to allow our VMs to communicate with each other, with the internet, and with our on-premise network through a VPN or ExpressRoute.

Let's begin by building our first VNet using the portal. We will start by browsing the **virtual network page** via **Create a Resource -> Networking -> Virtual network**:

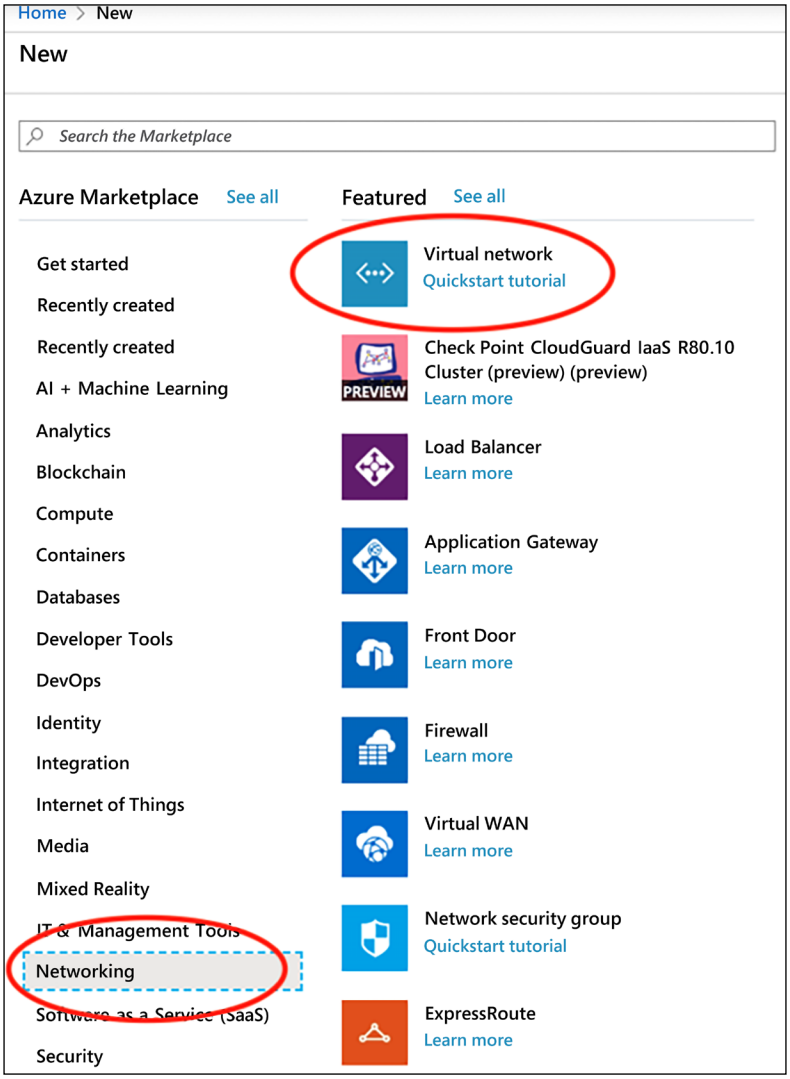


Figure 12.12: Azure VNet

Each VNet is scoped to a single region and we can create multiple subnets per VNet. As we will see later, multiple VNets in different regions can connect to each other via VNet peering.

From the VNet creation page, we will create our first network with the following credentials:

```
Name: WEST-US-2_VNet_1
Address space: 192.168.0.0/23
Subscription: <pick your subscription>
Resource group: <click on new> -> 'Mastering-Python-Networking'
Location: West US 2
Subnet name: WEST-US-2_VNet_1_Subnet_1
Address range: 192.168.1.0/24
DDoS protection: Basic
Service endpoints: Disabled
Firewall: Disabled
```

Here is a screenshot of the necessary fields. If there are any missing fields that are required, they will be highlighted in red. Click on **Create** when finished:

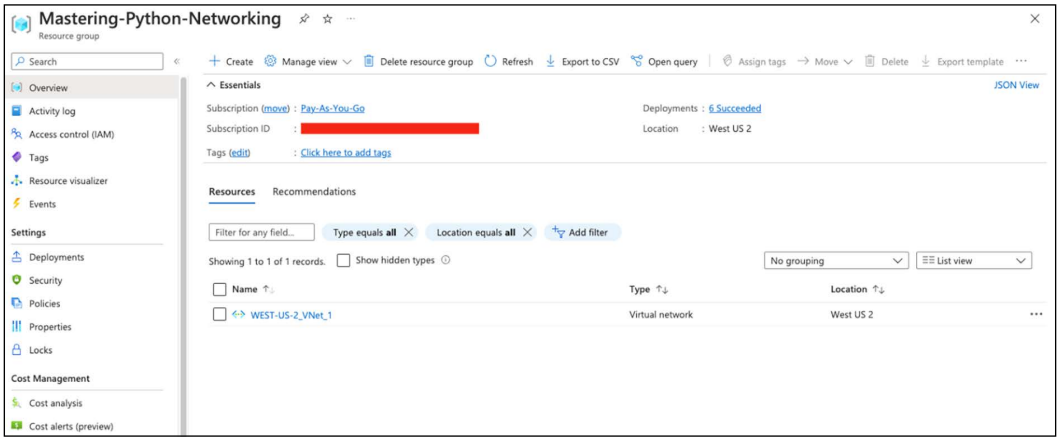


Figure 12.13: Azure VNet creation

Once the resource is created, we can navigate to it via **Home -> Resource groups -> Mastering-Python-Networking**:

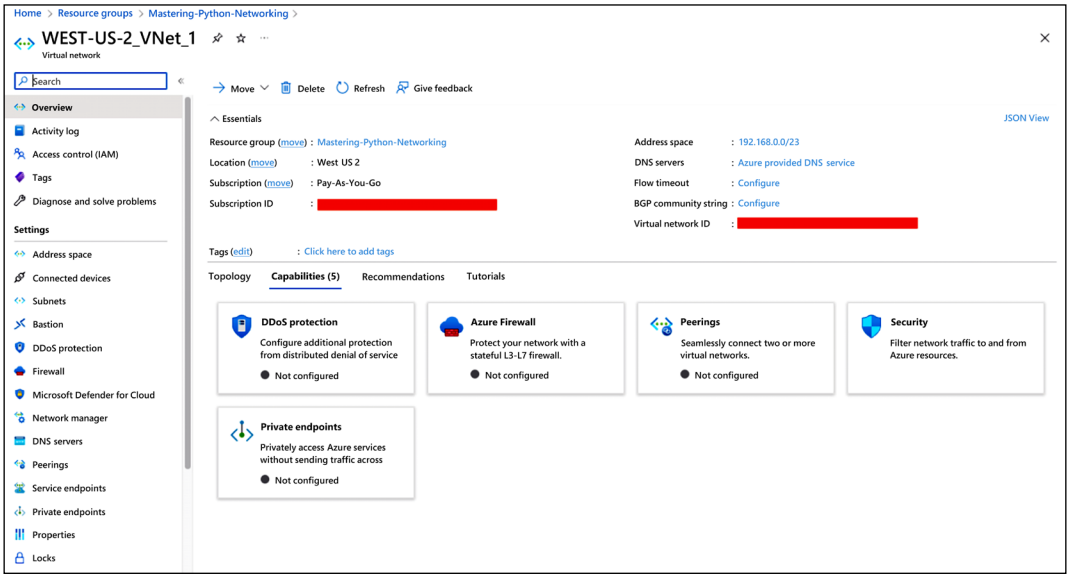


Figure 12.14: Azure VNet overview

Congratulations, we just created our first VNet in the Azure cloud! Our network needs to communicate with the outside world to be useful. We will look at how we can do that in the next section.

Internet access

By default, all resources within a VNet can carry out outbound communication with the internet; we do not need to add a NAT gateway as we do in AWS. For inbound communication, we will need to assign a public IP directly to the VM or use a load balancer with a public IP. To see this working, we will create VMs within our network.

We can create our first VM from **Home -> Resource groups -> Mastering-Python-Networking -> New -> Create a virtual machine:**

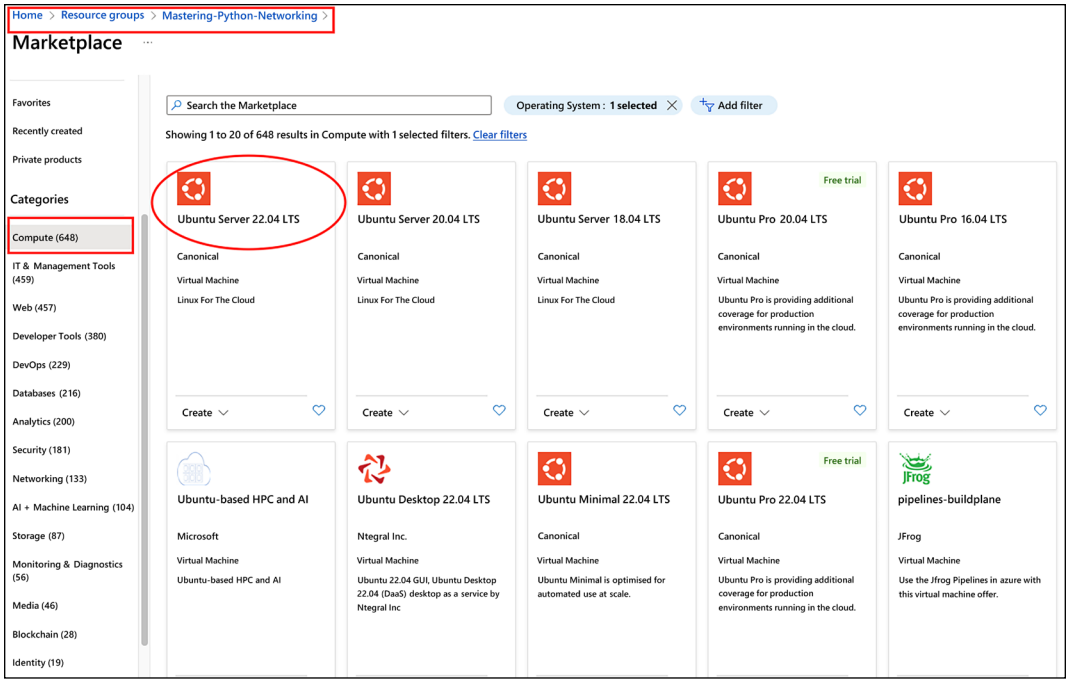


Figure 12.15: Azure creating a VM

I will pick **Ubuntu Server 22.04 LTS** as the VM and use the name `myMPN-VM1` when prompted. I will pick the region `West US 2`. We can choose password authentication or an SSH public key as the authentication method and allow an SSH inbound connection. Since we are using it for testing, we can pick the smallest instance in the B-Series to minimize our cost:

B-Series	Ideal for workloads that do not need continuous full CPU performance							
B2s	General purpose	2	4	4	1280	8	Supported	$30.37
B1s	General purpose	1	1	2	320	4	Supported	$7.59
B2ms	General purpose	2	8	4	1920	16	Supported	$60.74
B1ls	General purpose	1	0.5	2	320	4	Supported	$3.80
B4ms	General purpose	4	16	8	2880	32	Supported	$121.18
B1ms	General purpose	1	2	2	640	4	Supported	$15.11
B8ms	General purpose	8	32	16	4320	64	Supported	$243.09

Figure 12.16: Azure compute B-Series

We can leave the other options as their default settings, pick a small disk size, and check **delete with VM**. We will put the VM into the subnet that we created, as well as assigning a new public IP:

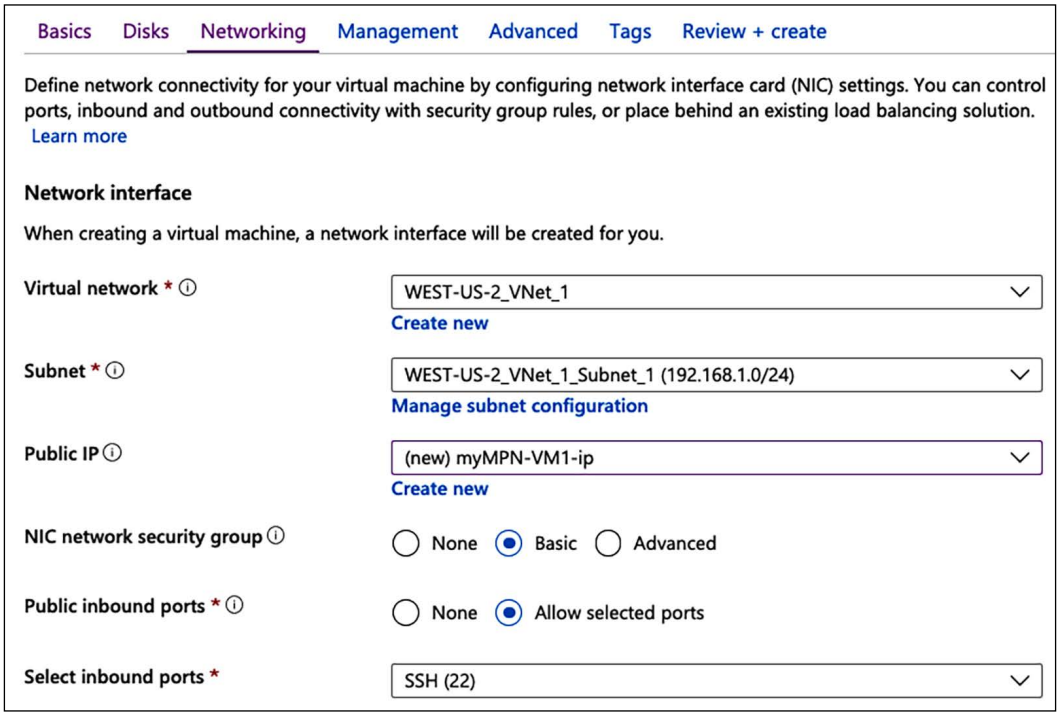


Figure 12.17: Azure network interface

After the VM is provisioned, we can `ssh` to the machine with the public IP and the user we created. The VM has only one interface that is within our private subnet; it is also mapped to the public IP that Azure automatically assigned. This public-to-private IP translation is done automatically by Azure.

```
echou@myMPN-VM1:~$ sudo apt install net-tools
echou@myMPN-VM1:~$ ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.4  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::20d:3aff:fe06:68a0  prefixlen 64  scopeid 0x20<link>
        ether 00:0d:3a:06:68:a0  txqueuelen 1000  (Ethernet)
        RX packets 2344  bytes 2201526 (2.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
```

```
        TX packets 1290  bytes 304355 (304.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
echou@myMPN-VM1:~$ ping -c 1 www.google.com
PING www.google.com (142.251.211.228) 56(84) bytes of data.
64 bytes from sea30s13-in-f4.1e100.net (142.251.211.228): icmp_seq=1
ttl=115 time=47.7 ms

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 47.668/47.668/47.668/0.000 ms
```

We can repeat the same process to create a second VM named `myMPN-VM2`. The VM can be configured with `SSH` inbound access but no public IP:

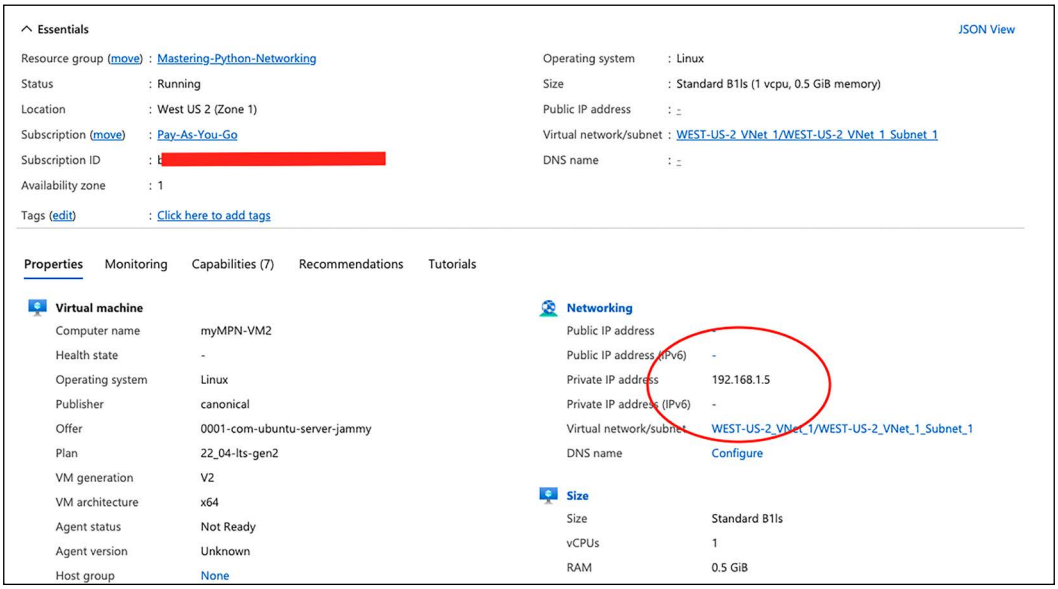


Figure 12.18: Azure VM IP addresses

After the VM creation, we can `ssh` to `myMPN-VM2` from `myMPN-VM1` with the private IP:

```
echou@myMPN-VM1:~$ ssh echou@192.168.1.5
echou@myMPN-VM2:~$ who
echou    pts/0        2022-09-22 16:43 (192.168.1.4)
```

We can test the internet connection by trying to access the `apt` package update repositories:

```
echou@myMPN-VM2:~$ sudo apt update
Hit:1 http://azure.archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://azure.archive.ubuntu.com/ubuntu jammy-updates InRelease [114
kB]
Get:3 http://azure.archive.ubuntu.com/ubuntu jammy-backports InRelease
[99.8 kB]
Get:4 http://azure.archive.ubuntu.com/ubuntu jammy-security InRelease [110
kB]
Get:5 http://azure.archive.ubuntu.com/ubuntu jammy/universe amd64 Packages
[14.1 MB]
Fetched 23.5 MB in 6s (4159 kB/s)
```

With our VM inside of VNet able to access the internet, we can create additional network resources for our network.

Network resource creation

Let's look at an example of using the Python SDK to create network resources. In the following example, `Chapter12_2_network_resources.py`, we will use the `subnet.create_or_update` API to create a new `192.168.0.128/25` subnet in the VNet:

```
#!/usr/bin/env python3
# Reference example: https://github.com/Azure-Samples/azure-samples-
python-management/blob/main/samples/network/virtual_network/manage_subnet.
py
#
import os
from azure.identity import ClientSecretCredential
import azure.mgmt.network
from azure.identity import DefaultAzureCredential
from azure.mgmt.network import NetworkManagementClient
from azure.mgmt.resource import ResourceManagementClient

credential = ClientSecretCredential(
    tenant_id=os.environ.get("AZURE_TENANT_ID"),
    client_id=os.environ.get("AZURE_CLIENT_ID"),
    client_secret=os.environ.get("AZURE_CLIENT_SECRET")
)
```

```
subscription_id = os.environ.get("SUBSCRIPTION_ID")
GROUP_NAME = "Mastering-Python-Networking"
VIRTUAL_NETWORK_NAME = "WEST-US-2_VNet_1"
SUBNET = "WEST-US-2_VNet_1_Subnet_2"
network_client = azure.mgmt.network.NetworkManagementClient(
    credential=credential, subscription_id=subscription_id)

# Get subnet
subnet = network_client.subnets.get(
    GROUP_NAME,
    VIRTUAL_NETWORK_NAME,
    SUBNET
)
print("Get subnet:\n{}".format(subnet))

subnet = network_client.subnets.begin_create_or_update(
    GROUP_NAME,
    VIRTUAL_NETWORK_NAME,
    SUBNET,
    {
        "address_prefix": "192.168.0.128/25"
    }
).result()
print("Create subnet:\n{}".format(subnet))
```

We will receive the following creation result message when we execute the script:

```
(venv) $ python3 Chapter12_2_subnet.py
{'additional_properties': {'type': 'Microsoft.Network/virtualNetworks/
subnets'}, 'id': '/subscriptions/<skip>/resourceGroups/Mastering-Python-
Networking/providers/Microsoft.Network/virtualNetworks/WEST-US-2_VNet_1/
subnets/WEST-US-2_VNet_1_Subnet_2', 'address_prefix': '192.168.0.128/25',
'address_prefixes': None, 'network_security_group': None, 'route_table':
None, 'service_endpoints': None, 'service_endpoint_policies': None,
'interface_endpoints': None, 'ip_configurations': None, 'ip_configuration_
profiles': None, 'resource_navigation_links': None, 'service_association_
links': None, 'delegations': [], 'purpose': None, 'provisioning_state':
'Succeeded', 'name': 'WEST-US-2_VNet_1_Subnet_2', 'etag': 'W/"<skip>"'}
```

The new subnet can also be seen on the portal:

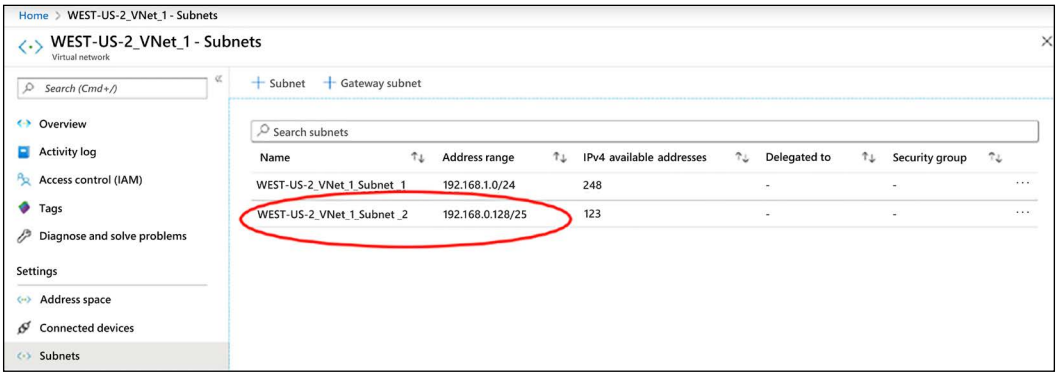


Figure 12.19: Azure VNet subnets

For more examples of using the Python SDK, check out `https://github.com/Azure-Samples/azure-samples-python-management`.

If we create a VM within the new subnet, even across subnet boundaries, the hosts in the same VNet can reach each other with the same implicit router we saw with AWS.

There are additional VNet services available to us when we need to interact with other Azure services. Let's take a look.

VNet service endpoints

VNet service endpoints can extend the VNet to other Azure services over a direct connection. This allows traffic from the VNet to the Azure service to remain on the Azure network. Service endpoints need to be configured with an identified service within the region of the VNet.

They can be configured via the portal with restrictions to the service and subnet:

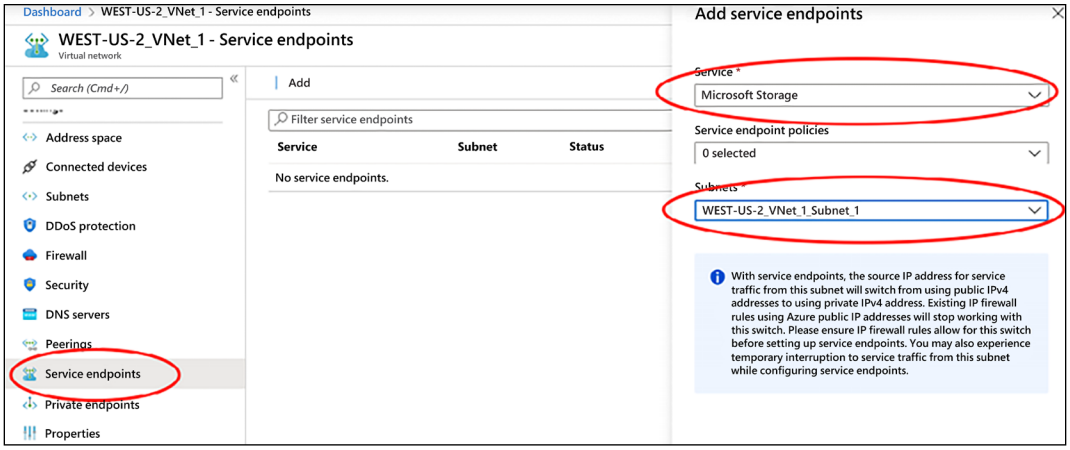


Figure 12.20: Azure service endpoints

Strictly speaking, we do not need to create VNet service endpoints when we need to have the VMs in the VNet communicate with the service. Each VM can access the service through the public IP mapped, and we can use network rules to permit only the necessary IPs. However, using the VNet service endpoints allows us to access the resources using the private IP within Azure without the traffic traversing the public internet.

VNet peering

As mentioned at the beginning of the section, each VNet is limited to a region. For region-to-region VNet connectivity, we can leverage VNet peering. Let's use the following two functions in `Chapter11_3_vnet.py` to create a VNet in the `US-East` region:

```
<skip>
def create_vnet(network_client):
    vnet_params = {
        'location': LOCATION,
        'address_space': {
            'address_prefixes': ['10.0.0.0/16']
        }
    }
```

```
    creation_result = network_client.virtual_networks.create_or_update(
        GROUP_NAME,
        'EAST-US_VNet_1',
        vnet_params
    )
    return creation_result.result()
<skip>
def create_subnet(network_client):
    subnet_params = {
        'address_prefix': '10.0.1.0/24'
    }
    creation_result = network_client.subnets.create_or_update(
        GROUP_NAME,
        'EAST-US_VNet_1',
        'EAST-US_VNet_1_Subnet_1',
        subnet_params
    )
    return creation_result.result()
```

To allow VNet peering, we need to peer bi-directionally from both VNets. Since we have been using the Python SDK up to this point, for learning purposes, let's look at an example with the Azure CLI.

We will grab the VNet name and ID from the `az network vnet list` command:

```
(venv) $ az network vnet list
<skip>
"id": "/subscriptions/<skip>/resourceGroups/Mastering-Python-Networking/
providers/Microsoft.Network/virtualNetworks/EAST-US_VNet_1",
    "location": "eastus",
    "name": "EAST-US_VNet_1"
<skip>
"id": "/subscriptions/<skip>/resourceGroups/Mastering-Python-Networking/
providers/Microsoft.Network/virtualNetworks/WEST-US-2_VNet_1",
    "location": "westus2",
    "name": "WEST-US-2_VNet_1"
<skip>
```

Let's check the existing VNet peering for our `West US 2` VNet:

```
(venv) $ az network vnet peering list -g "Mastering-Python-Networking"
--vnet-name WEST-US-2_VNet_1
[]
```

We will execute the peering from the `West US` to `East US` VNet, then repeat in the reverse direction:

```
(venv) $ az network vnet peering create -g "Mastering-Python-Networking"
-n WestUSToEastUS --vnet-name WEST-US-2_VNet_1 --remote-vnet "/
subscriptions/<skip>/resourceGroups/Mastering-Python-Networking/providers/
Microsoft.Network/virtualNetworks/EAST-US_VNet_1"
(venv) $ az network vnet peering create -g "Mastering-Python-
Networking" -n EastUSToWestUS --vnet-name EAST-US_VNet_1 --remote-vnet
"/subscriptions/b7257c5b-97c1-45ea-86a7-872ce8495a2a/resourceGroups/
Mastering-Python-Networking/providers/Microsoft.Network/virtualNetworks/
WEST-US-2_VNet_1"
```

Now if we run the check again, we will be able to see the VNet successfully peered:

```
(venv) $ az network vnet peering list -g "Mastering-Python-Networking"
--vnet-name "WEST-US-2_VNet_1"
[
  {
    "allowForwardedTraffic": false,
    "allowGatewayTransit": false,
    "allowVirtualNetworkAccess": false,
    "etag": "W/\"<skip>\"",
    "id": "/subscriptions/<skip>/resourceGroups/Mastering-Python-
Networking/providers/Microsoft.Network/virtualNetworks/WEST-US-2_VNet_1/
virtualNetworkPeerings/WestUSToEastUS",
    "name": "WestUSToEastUS",
    "peeringState": "Connected",
    "provisioningState": "Succeeded",
    "remoteAddressSpace": {
      "addressPrefixes": [
        "10.0.0.0/16"
      ]
    },
<skip>
```

We can also verify the peering on the Azure portal:

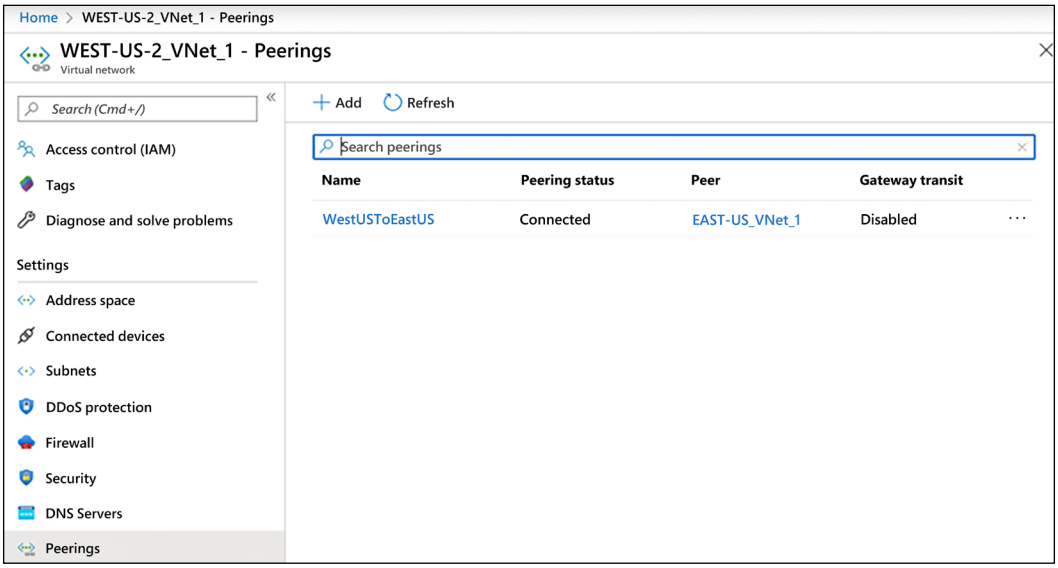


Figure 12.21: Azure VNet peering

Now that we have several hosts, subnets, VNets, and VNet peering in our setup, we should look at how routing is done in Azure. That is what we will do in the next section.

VNet routing

As a network engineer, implicit routes added by the cloud provider have always been a bit uncomfortable for me. In traditional networking, we need to cable up the network, assign IP addresses, configure routing, implement security, and make sure everything works. It can sometimes be complex, but every packet and route is accounted for. For virtual networks in the cloud, the underlay network is already completed by Azure and some network configuration on the overlay network needs to happen automatically for the host to work at launch time, as we saw earlier.

Azure VNet routing is a bit different from AWS. In the AWS chapter, we saw the routing table implemented at the VPC network layer. But if we browse to the Azure VNet setting on the portal, we will not find a routing table assigned to the VNet.

If we drill deeper into the **subnet setting**, we will see a routing table drop-down menu, but the value it is displaying is **None**:

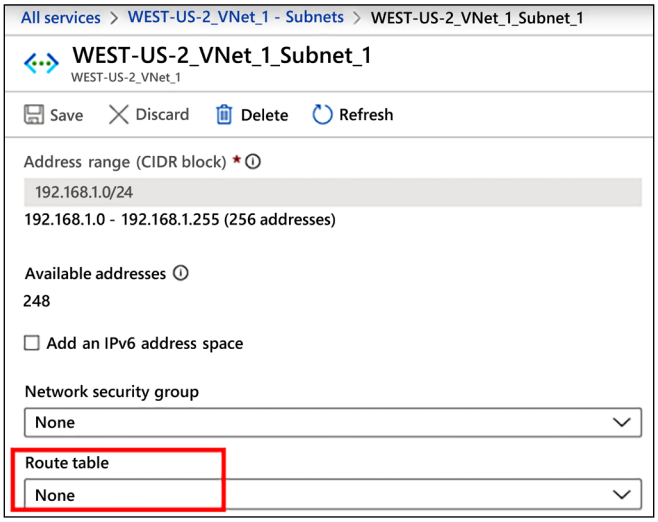


Figure 12.22: Azure subnet routing table

How can we have an empty routing table with the hosts in that subnet able to reach the internet? Where can we see the routes configured by Azure VNet? The routing has been implemented at the host and NIC levels. We can see it via **All services -> Virtual Machines -> myNPM-VM1 -> Networking (left panel) -> Topology (top panel)**:

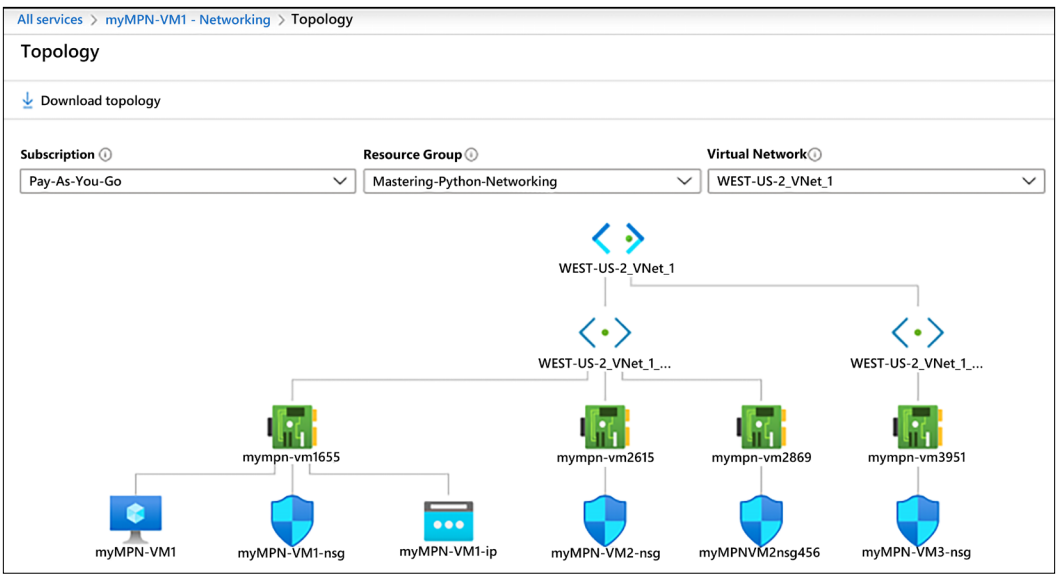


Figure 12.23: Azure network topology

The network is being shown on the NIC level with each NIC attached to a VNet subnet on the north side and other resources such as VM, **Network Security Group** (**NSG**), and IP on the south side. The resources are dynamic; at the time of the screen capture, I only had `myMPN-VM1` running, therefore it is the only one with a VM and IP address attached, while the other VMs only have NSGs attached.

We will cover NSG in the next section.

If we click on the NIC, **mympn-vm1655** in our topology, we can see the settings associated with the NIC. Under the **Support + troubleshooting** section, we will find the **Effective routes** link, where we can see the current routing associated with the NIC:

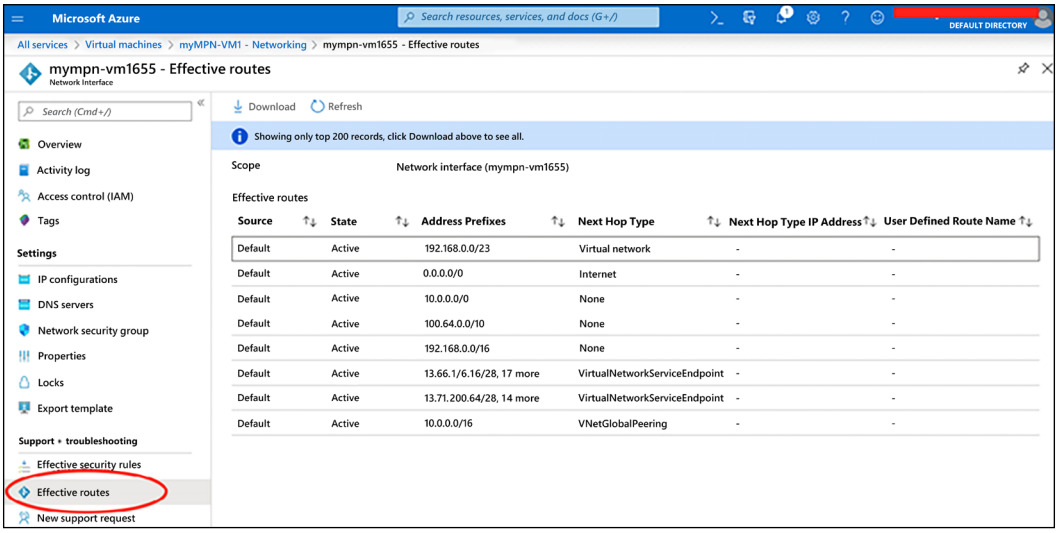


Figure 12.24: Azure VNet effective routes

If we want to automate the process, we can use the Azure CLI to find the NIC name and then show the routing table:

```
(venv) $ az vm show --name myMPN-VM1 --resource-group 'Mastering-Python-
Networking'
<skip>
"networkProfile": {
    "networkInterfaces": [
      {
        "id": "/subscriptions/<skip>/resourceGroups/Mastering-Python-
Networking/providers/Microsoft.Network/networkInterfaces/mympn-vm1655",
```

```
        "primary": null,
        "resourceGroup": "Mastering-Python-Networking"
      }
    ]
  }
<skip>
(venv) $ az network nic show-effective-route-table --name mympn-vm1655
--resource-group "Mastering-Python-Networking"
{
  "nextLink": null,
  "value": [
    {
      "addressPrefix": [
        "192.168.0.0/23"
      ],
<skip>
```

Great! That was one mystery solved, but what are those next hops in the routing table? We can reference the VNet traffic routing document: `https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-udr-overview`. A few important notes:

- If the source indicates that the route is **Default**, these are system routes that cannot be removed but can be overwritten with custom routes.
- VNet next hops are the routes within the custom VNet. In our case, this is the `192.168.0.0/23` network, not just the subnet.
- Traffic routed to the **None** next hop type is dropped, similar to the **Null** interface routes.
- The **VNetGlobalPeering** next hop type is what was created when we established VNet peering with other VNets.
- The **VirtualNetworkServiceEndpoint** next hop type was created when we enabled service endpoints in our VNet. The public IP is managed by Azure and changes from time to time.

How do we override the default routes? We can create a route table and associate it with subnets. Azure selects the routes with the following priority:

- User-defined route
- BGP route (from a Site-to-Site VPN or ExpressRoute)
- System route

We can create a route table in the **Networking** section:

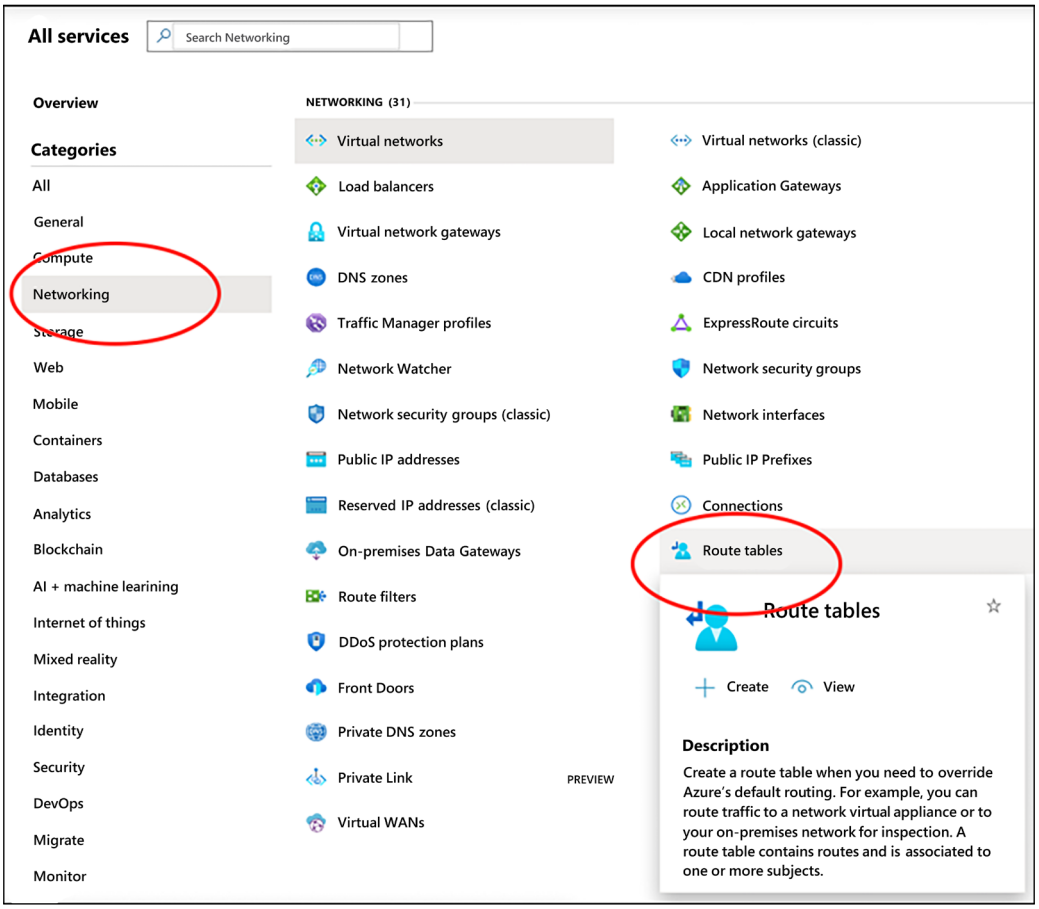


Figure 12.25: Azure VNet route tables

We can also create a route table, create a route within the table, and associate the route table with a subnet via the Azure CLI:

```
(venv) $ az network route-table create --name TempRouteTable --resource
"Mastering-Python-Networking"
(venv) $ az network route-table route create -g "Mastering-Python-
Networking" --route-table-name TempRouteTable -n TempRoute  --next-hop-
type VirtualAppliance --address-prefix 172.31.0.0/16 --next-hop-ip-address
10.0.100.4
(venv) $ az network vnet subnet update -g "Mastering-Python-Networking"
-n WEST-US-2_Vnet_1_Subnet_1 --vnet-name WEST-US-2_VNet_1 --route-table
TempRouteTable
```

Let's take a look at the primary security measure in VNet: NSGs.

Network security groups

VNet security is primarily implemented by NSGs. Just like traditional access lists or firewall rules, we need to think of network security rules in a single direction at a time. For example, if we want to have host `A`, in `subnet 1` communicate freely with host `B` in `subnet 2` over port `80`, we need to implement the necessary rules for both inbound and outbound directions for both hosts.

As we saw from previous examples, an NSG can be associated with the NIC or the subnet, so we also need to think in terms of security layers. Generally speaking, we should implement the more restrictive rules at the host level while the more relaxed rules are applied at the subnet level. This is similar to traditional networking.

When we created our VMs, we set a permit rule for SSH TCP port 22 inbound. Let's take a look at the security group that was created for our first VM, **myMPN-VM1-nsg**:

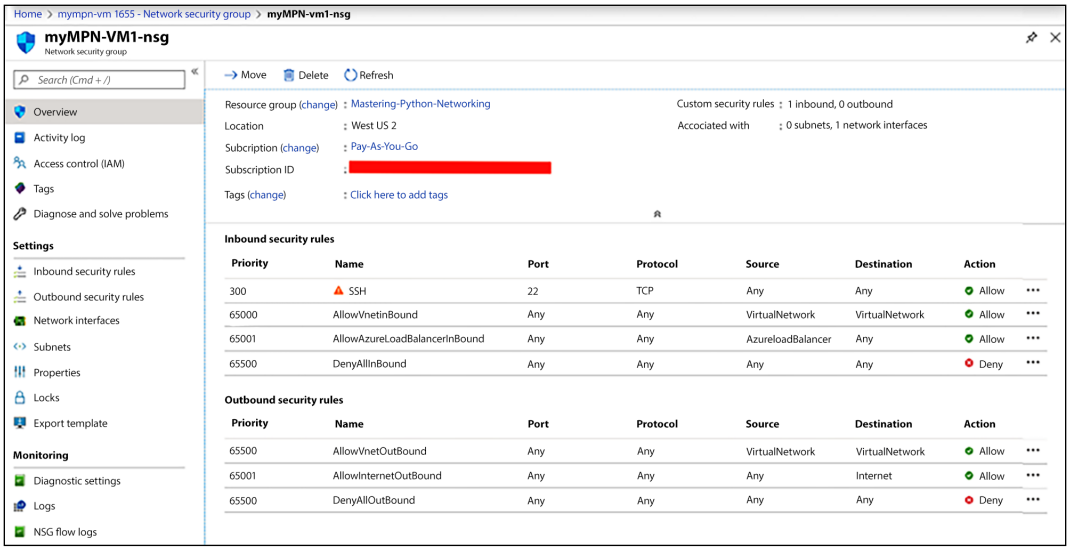


Figure 12.26: Azure VNet NSG

There are a few things worth pointing out:

- The priority level of system-implemented rules is high, at 65,000 and above.
- By default, virtual networks can freely communicate with each other in both directions.
- By default, internal hosts are allowed internet access.

Let's implement an inbound rule on the existing NSG group from the portal:

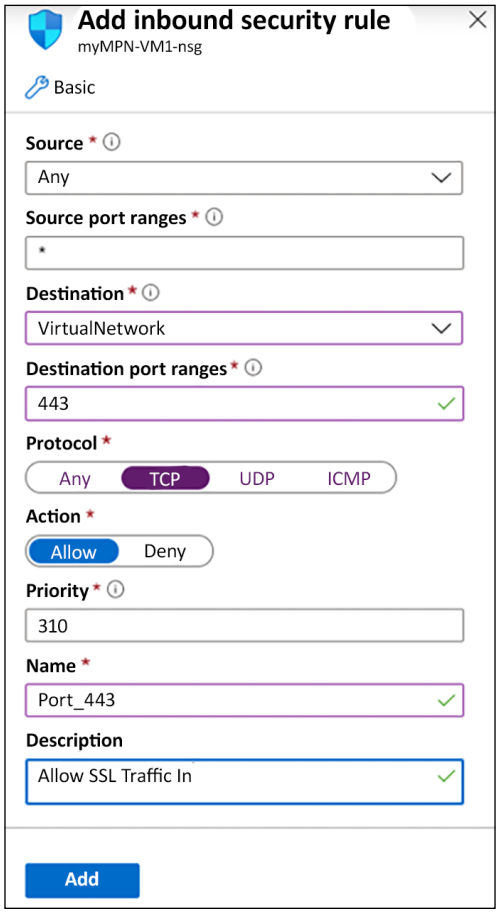


Figure 12.27: Azure security rule

We can also create a new security group and rules via the Azure CLI:

```
(venv) $ az network nsg create -g "Mastering-Python-Networking" -n TestNSG
(venv) $ az network nsg rule create -g "Mastering-Python-Networking"
--nsg-name TestNSG -n Allow_SSH --priority 150 --direction Inbound
--source-address-prefixes Internet --destination-port-ranges 22 --access
Allow --protocol Tcp --description "Permit SSH Inbound"
(venv) $ az network nsg rule create -g "Mastering-Python-Networking"
--nsg-name TestNSG -n Allow_SSL --priority 160 --direction Inbound
--source-address-prefixes Internet --destination-port-ranges 443 --access
Allow --protocol Tcp --description "Permit SSL Inbound"
```

We can see the new rules that were created as well as the default rules:

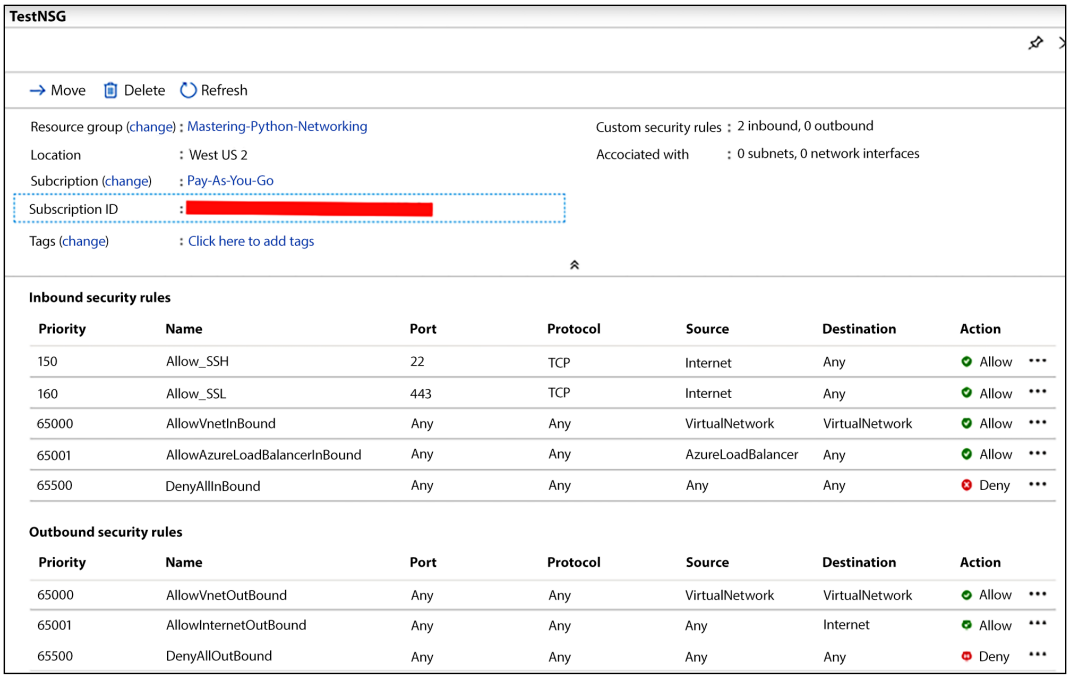


Figure 12.28: Azure security rules

The last step would be to bind this NSG to a subnet:

```
(venv) $ az network vnet subnet update -g "Mastering-Python-Networking" -n
WEST-US-2_VNet_1_Subnet_1 --vnet-name WEST-US-2_VNet_1 --network-security-
group TestNSG
```

In the next two sections, we will look at the two primary ways to extend Azure virtual networks to our on-premises data center: Azure VPN and Azure ExpressRoute.

Azure VPNs

As the network continues to grow, there might come a time when we need to connect the Azure VNet to our on-premise location. A VPN gateway is a type of VNet gateway that can encrypt the traffic between a VNet and our on-premise network and remote clients. Each VNet can only have one VPN gateway, but multiple connections can be built on the same VPN gateway.

More information about Azure VPN gateways can be found at this link: `https://docs.microsoft.com/en-us/azure/vpn-gateway/`.

VPN gateways are actually VMs themselves, configured with encryption and routing services, but cannot be directly configured by the user. Azure provides a list of SKUs based on the type of tunnel, number of concurrent connections, and total throughput (`https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpn-gateway-settings#gwsku`):

Gateway SKUs by tunnel, connection, and throughput

SKU	S2S/VNet-to-VNet Tunnels	P2S SSTP Connections	P2S IKEv2/OpenVPN Connections	Aggregate Throughput Benchmark	BGP	Zone-redundant
Basic	Max. 10	Max. 128	Not Supported	100 Mbps	Not Supported	No
VpnGw1	Max. 30*	Max. 128	Max. 250	650 Mbps	Supported	No
VpnGw2	Max. 30*	Max. 128	Max. 500	1 Gbps	Supported	No
VpnGw3	Max. 30*	Max. 128	Max. 1000	1.25 Gbps	Supported	No
VpnGw1AZ	Max. 30*	Max. 128	Max. 250	650 Mbps	Supported	Yes
VpnGw2AZ	Max. 30*	Max. 128	Max. 500	1 Gbps	Supported	Yes
VpnGw3AZ	Max. 30*	Max. 128	Max. 1000	1.25 Gbps	Supported	Yes

Figure 12.29: Azure VPN gateway SKUs (source: https://docs.microsoft.com/en-us/azure/vpn-gateway/point-to-site-about)

As we can see from the preceding table, the Azure VPN is divided into two different categories: **Point-to-Site (P2S)** VPN and **Site-to-Site (S2S)** VPN. The P2S VPN allows secure connections from an individual client computer, mainly used by telecommuters. The encryption method can be SSTP, IKEv2, or OpenVPN connection. When picking the type of VPN Gateway SKU for P2S, we will want to focus on the second and third columns on the SKU chart for the number of connections.

For a client-based VPN, we can use either SSTP or IKEv2 as the tunneling protocol:

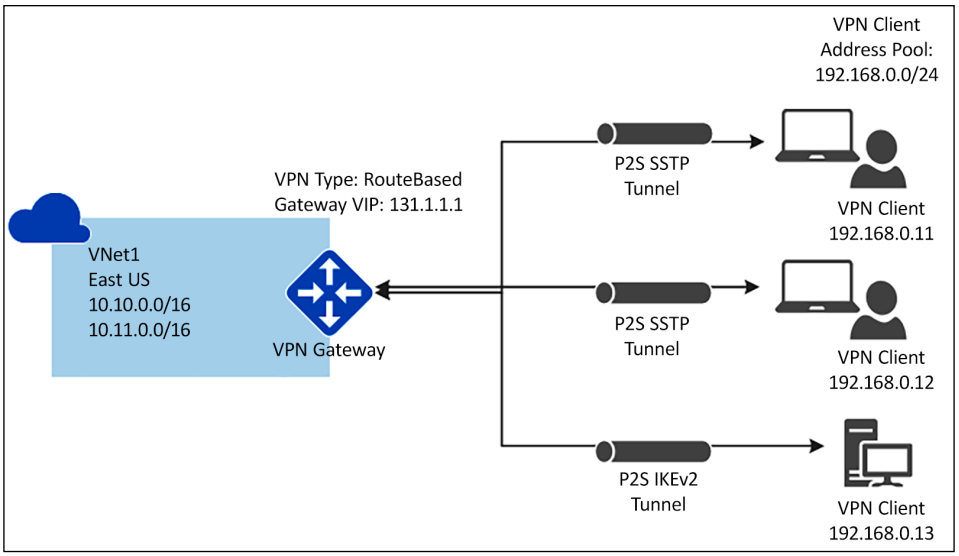


Figure 12.30: Azure Site-to-Site VPN gateway (source: https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways)

Besides client-based VPNs, another type of VPN connection is a Site-to-Site or multi-site VPN connection. The encryption method will be IPSec over IKE and a public IP will be required for both Azure and the on-premise network, as illustrated by the following diagram:

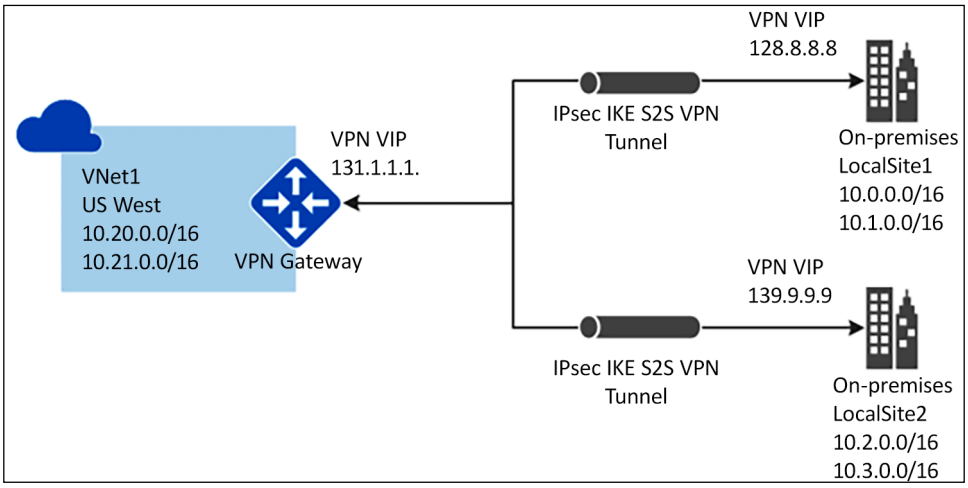


Figure 12.31: Azure client VPN gateway (source: https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways)

A full example of creating an S2S or P2S VPN is more than what we can cover in this section. Azure provides tutorials for S2S (`https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-howto-site-to-site-resource-manager-portal`), as well as P2S VPN (`https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-howto-site-to-site-resource-manager-portal`).

The steps are pretty straightforward for engineers who have configured VPN services before. The only point that may be a bit confusing and is not called out in the document is the fact that the VPN gateway device should live in a dedicated gateway subnet within the VNet with a `/27` IP block assigned:

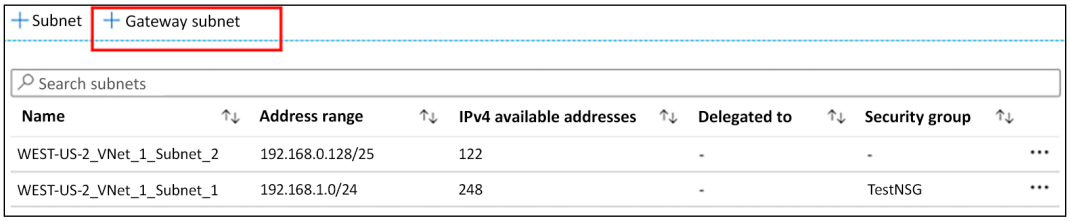


Figure 12.32: Azure VPN gateway subnet

A growing list of validated Azure VPN devices can be found at `https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpn-devices`, with links to their respective configuration guides.

Azure ExpressRoute

When organizations need to extend an Azure VNet to on-premises sites, it makes sense to start with a VPN connection. However, as the connection takes on more mission-critical traffic, the organization might want a more stable and reliable connection. Similar to AWS Direct Connect, Azure offers ExpressRoute as a private connection facilitated by a connectivity provider. As we can see from the diagram, our network is connected to Azure's partner edge network before it is transitioned to Azure's edge network:

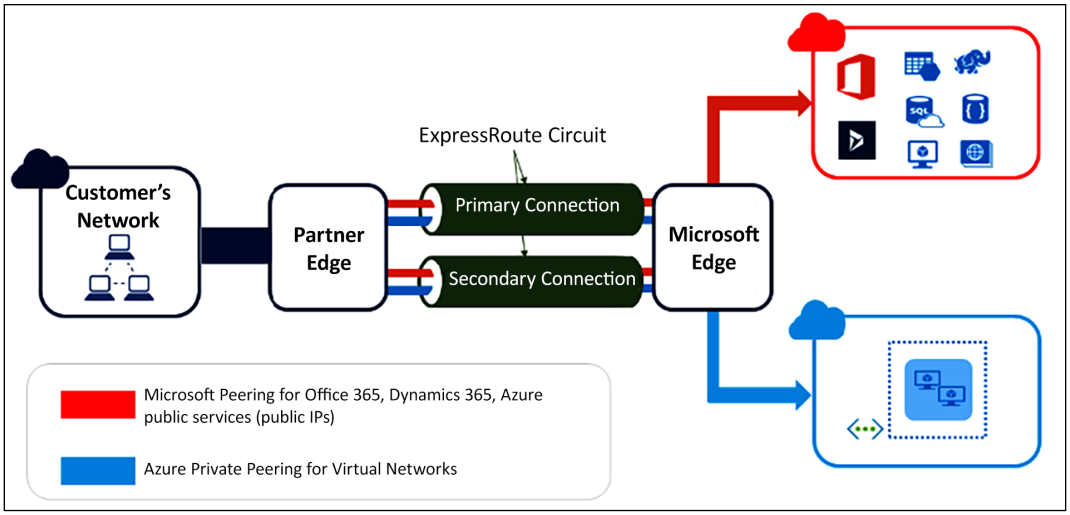


Figure 12.33: Azure ExpressRoute circuits (source: https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction)

The advantages of ExpressRoute include:

- More reliable since it does not traverse through the public internet.
- A faster connection with lower latency since a private connection is likely to have fewer hops between on-premise equipment to Azure.
- Better security measures since it is a private connection, especially if a company relies on Microsoft services such as Office 365.

The disadvantages of ExpressRoute can be:

- More difficulty setting up, both in terms of business and technical requirements.
- Higher cost commitment upfront, since the port charge and connection charges are often fixed. Some of the costs can be offset by a reduction in internet costs if it replaces a VPN connection. However, the total cost of ownership is typically higher with ExpressRoute.

A more detailed overview of ExpressRoute can be found at `https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction`. One of the biggest differences from AWS Direct Connect is the fact that ExpressRoute can offer connections across regions in geography. There is also a premium add-on that allows global connectivity to Microsoft services as well as QoS support for Skype for Business.

Similar to Direct Connect, ExpressRoute requires the user to connect to Azure with a partner or meet Azure at a certain designated location with ExpressRoute Direct (yes, the term is confusing). This is typically the biggest hurdle for enterprises to get over since they will need to either build their data center at one of the Azure locations, connect with a carrier (MPLS VPN), or work with a broker as a go-between for connection. These options typically require business contracts, longer-term commitments, and committed monthly costs.

To start, my recommendation would be similar to in *Chapter 11*, *AWS Cloud Networking*, which is to use an existing carrier broker for connection to a carrier hotel. From the carrier hotel, either directly connect to Azure or use an intermediary such as Equinix FABRIC (`https://www.equinix.com/interconnection-services/equinix-fabric`).

In the next section, we will look at how we can distribute incoming traffic efficiently when our service grows beyond just a single server.

Azure network load balancers

Azure offers load balancers in both the basic and standard SKU. When we discuss the load balancer in this section, we are referring to the Layer 4 TCP and UDP load distribution service instead of the Application Gateway Load Balancer (`https://azure.microsoft.com/en-us/services/application-gateway/`), which is a layer-7 load-balancing solution.

The typical deployment model is usually a one- or two-layer load distribution for an inbound connection from the internet:

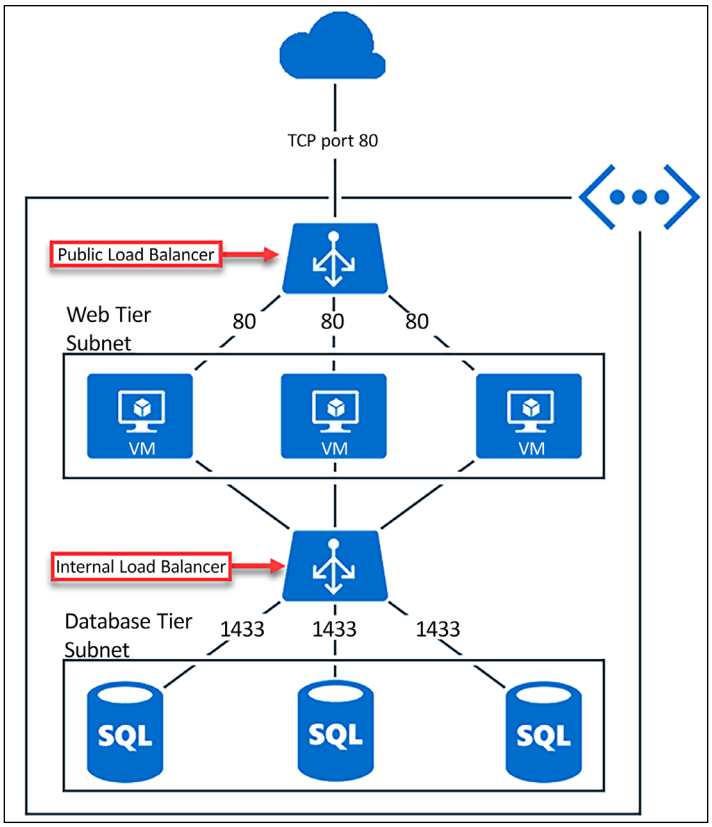


Figure 12.34: Azure Load Balancer (source: https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview)

The load balancer hashes the incoming connection on a 5-tuple hash (source and destination IP, source and destination port, and protocol) and distributes the flow to one or more destinations. The Standard Load Balancer SKU is a superset of the basic SKU, therefore new designs should adopt the Standard Load Balancer.

As with AWS, Azure is constantly innovating with new network services. We have covered the foundational services in this chapter; let's take a look at some of the other notable services.

Other Azure network services

Some of the other Azure network services that we should be aware of are:

- **DNS services**: Azure has a suite of DNS services (`https://docs.microsoft.com/en-us/azure/dns/dns-overview`), both public and private. It can be used for geographical load balancing for network services.
- **Container networking**: Azure has been making a push toward containers in recent years. More information about Azure network capabilities for containers can be found at `https://docs.microsoft.com/en-us/azure/virtual-network/container-networking-overview`.
- **VNet TAP**: Azure VNet TAP allows you to continuously stream your VM network traffic to a network packet collector or analytical tool (`https://docs.microsoft.com/en-us/azure/virtual-network/virtual-network-tap-overview`).
- **Distributed Denial of Service Protection**: Azure DDoS protection provides defense against DDoS attacks (`https://docs.microsoft.com/en-us/azure/virtual-network/ddos-protection-overview`).

Azure network services are a big part of the Azure cloud family and continue to grow at a fast rate. We have only covered a portion of the services in this chapter, but hopefully, it has given you a good foundation from which to begin to explore other services.

Summary

In this chapter, we took a look at the various Azure cloud network services. We discussed the Azure global network and various aspects of virtual networks. We used both the Azure CLI and the Python SDK to create, update, and manage those network services. When we need to extend Azure services to an on-premise data center, we can use either VPN or ExpressRoute for connectivity. We also briefly looked at various Azure network products and services.

In the next chapter, we will revisit the data analysis pipeline with an all-in-one stack: the Elastic Stack.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



13

Network Data Analysis with Elastic Stack

In *Chapter 7*, *Network Monitoring with Python – Part 1*, and *Chapter 8*, *Network Monitoring with Python – Part 2*, we discussed the various ways to monitor a network. In the two chapters, we looked at two different approaches for network data collection: we can either retrieve data from network devices such as SNMP, or we can listen for the data sent by network devices using flow-based exports. After the data is collected, we will need to store the data in a database, then analyze the data to gain insights to decide what the data means. Most of the time, the analyzed results are displayed in a graph, whether a line graph, bar graph, or pie chart. We can use individual tools such as PySNMP, Matplotlib, and Pygal for each step, or we can leverage all-in-one tools such as Cacti or ntop for monitoring. The tools introduced in those two chapters gave us basic monitoring and understanding of the network.

We then moved on to *Chapter 9*, *Building Network Web Services with Python*, to build API services to abstract our network from higher-level tools. In *Chapter 11*, *AWS Cloud Networking*, and *Chapter 12*, *Azure Cloud Networking*, we extended our on-premises network to the cloud using AWS and Azure. We have covered much ground in these chapters and have a solid set of tools to help us make our network programmable.

Starting with this chapter, we will build on our toolsets from previous chapters and look at other tools and projects that I have found useful in my journey once I was comfortable with the tools covered in previous chapters. In this chapter, we will take a look at an open source project, Elastic Stack (`https://www.elastic.co`), that can help us with analyzing and monitoring our network beyond what we have seen before.

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

- What is the Elastic (or ELK) Stack?
- Elastic Stack installation
- Data ingestion with Logstash
- Data ingestion with Beats
- Search with Elasticsearch
- Data visualization with Kibana

Let's begin by answering the question: what exactly is the Elastic Stack?

What is the Elastic Stack?

The Elastic Stack is also known as the "ELK" Stack. So, what is it? Let's see what the developers have to say in their own words (`https://www.elastic.co/what-is/elk-stack`):

> *"ELK" is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a serverside data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch. The Elastic Stack is the next evolution of the ELK Stack.*

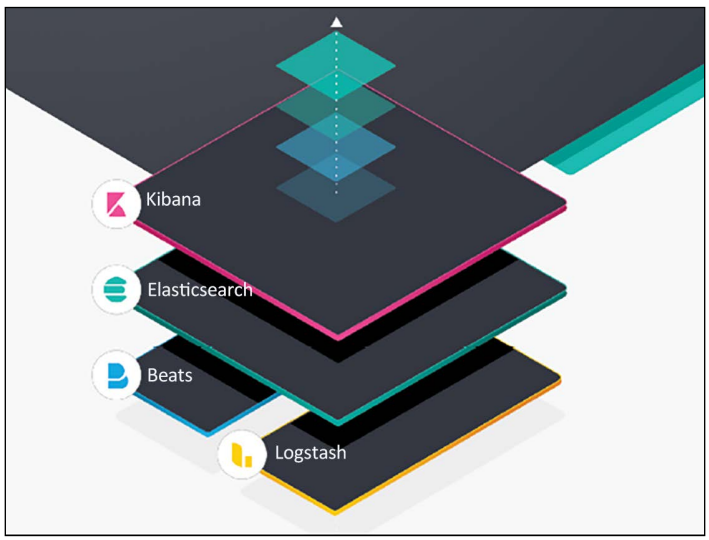


Figure 13.1: Elastic Stack (source: https://www.elastic.co/what-is/elk-stack)

As we can see from the statement, the Elastic Stack is a collection of different projects working together to cover the whole spectrum of data collection, storage, retrieval, analytics, and visualization. What is nice about the stack is that it is tightly integrated, but each component can also be used separately. If we dislike Kibana for visualization, we can easily plug in Grafana for the graphs. What if we have other data ingestion tools that we want to use? No problem, we can use the RESTful API to post our data to Elasticsearch. At the center of the stack is Elasticsearch, an open source, distributed search engine. The other projects were created to enhance and support the search function. This might sound a bit confusing at first, but as we look deeper at the components of the project, it will become clearer.

Why did they change the name of ELK Stack to Elastic Stack? In 2015, Elastic introduced a family of lightweight, single-purpose data shippers called Beats. They were an instant hit and continue to be very popular, but the creators could not come up with a good acronym for the "B" and decided to just rename the whole stack to Elastic Stack.

We will focus on the network monitoring and data analysis aspects of the Elastic Stack. Still, the stack has many use cases, including risk management, e-commerce personalization, security analysis, fraud detection, and more. It is being used by various organizations, from web companies such as Cisco, Box, and Adobe, to government agencies such as NASA JPL, the United States Census Bureau, and others (`https://www.elastic.co/customers/`).

When we talk about Elastic, we are referring to the company behind the Elastic Stack. The tools are open source and the company makes money by selling support, hosted solutions, and consulting around open source projects. The company stock is publicly traded on the New York Stock Exchange with the ESTC symbol.

Now that we have a better idea of what the ELK Stack is, let's take a look at the lab topology for this chapter.

Lab topology

For the network lab, we will reuse the network topology we used in *Chapter 8*, *Network Monitoring with Python – Part 2*. The network gear will have the management interfaces in the `192.168.2.0/24` management network with the interconnections in the `10.0.0.0/8` network and the subnets in `/30s`.

Where can we install the ELK Stack in the lab? In production, we should run the ELK Stack in a dedicated cluster. In our lab, however, we can quickly spin up a testing instance via Docker containers. If a refresher of Docker is needed, please refer to *Chapter 5*, *Docker Containers for Network Engineers*.

Following is a graphical representation of our network lab topology:

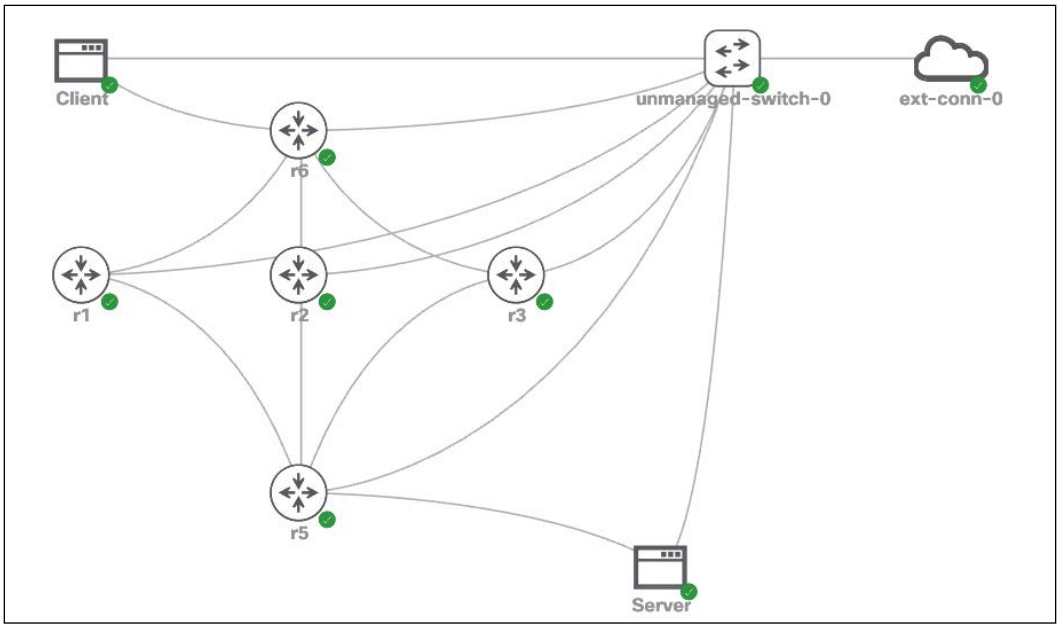


Figure 13.2: Lab Topology

Device	Management IP	Loopback IP
r1	192.168.2.218	192.168.0.1
r2	192.168.2.219	192.168.0.2
r3	192.168.2.220	192.168.0.3
r5	192.168.2.221	192.168.0.4
r6	192.168.2.222	192.168.0.5

The Ubuntu hosts information is as follows:

Device Name	External Link Eth0	Internal IP Eth1
Client	192.168.2.211	10.0.0.9
Server	192.168.2.212	10.0.0.5

To run multiple containers, we should allocate at least 4 GB RAM or more to the host. Let's start Docker Engine, if not done already, then pull the image from Docker Hub:

```
$ sudo service docker start
$ docker network create elastic
$ docker pull docker.elastic.co/elasticsearch/elasticsearch:8.4.2
$ docker run --name elasticsearch --rm -it --network elastic -p 9200:9200
-p 9300:9300 -e "discovery.type=single-node" -t docker.elastic.co/
elasticsearch/elasticsearch:8.4.2
```

When the Docker container is run, the generated default Elastic user password and Kibana enrollment token are output to the terminal; please take a note of them as we will need them later. You might need to scroll up the screen a bit to find them:

```
->  Password for the elastic user (reset with 'bin/elasticsearch-reset-
password -u elastic'):
  <password>
->  Configure Kibana to use this cluster:
* Run Kibana and click the configuration link in the terminal when Kibana
starts.
* Copy the following enrollment token and paste it into Kibana in your
browser (valid for the next 30 minutes):
  <token>
```

Once the Elasticsearch container runs, we can test out the instance by browsing to `https://<your ip>:9200`:

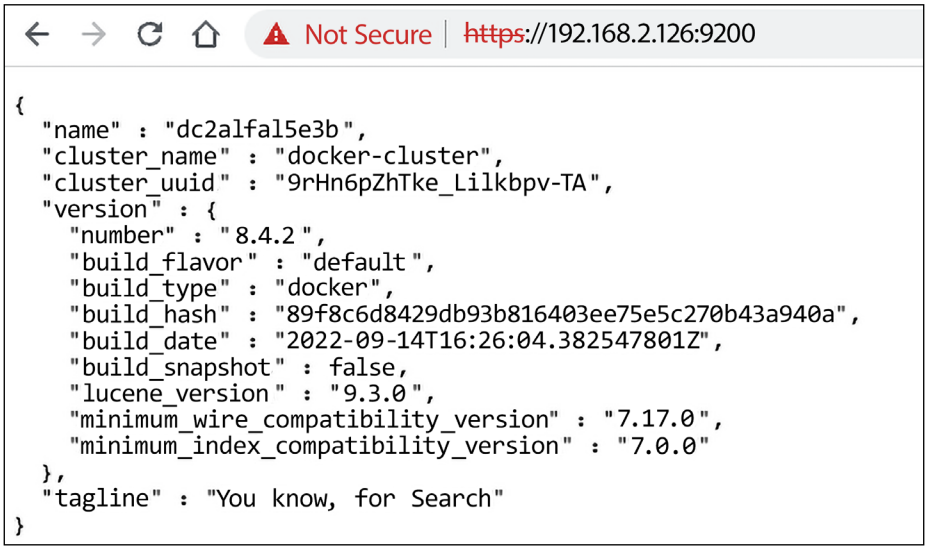


Figure 13.3: Elasticsearch Initial Result

We can then pull and run the Kibana container image from a separate terminal:

```
$ docker pull docker.elastic.co/kibana/kibana:8.4.2
$ docker run --name kibana --rm -it --network elastic -p 5601:5601 docker.
elastic.co/kibana/kibana:8.4.2
```

Once Kibana boots up, we can access it via port 5601:

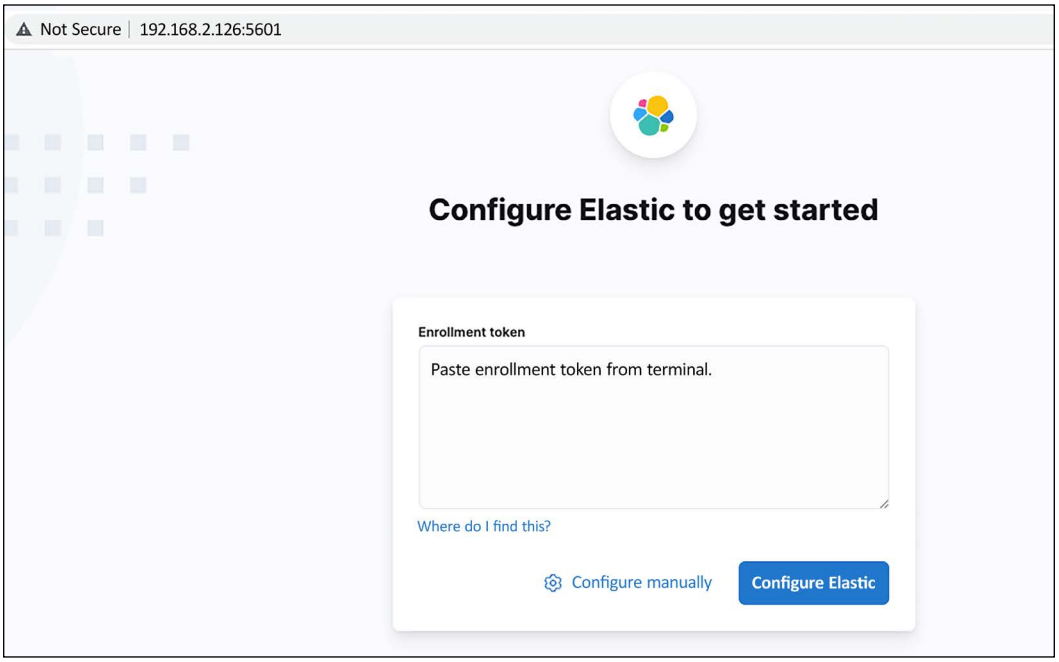


Figure 13.4: Kibana Start Page

Notice it is asking for the enrolment token we jotted down before. We can paste that in and click on **Configure Elastic**. It will prompt us for a token, which is now displayed on the Kibana terminal. Once that is authenticated, Kibana will start to configure Elastic:

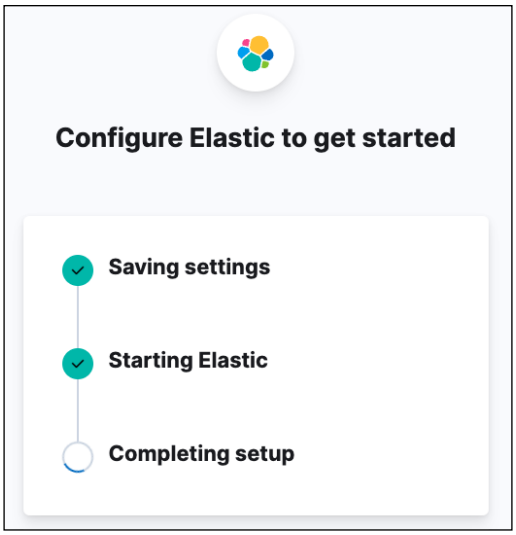


Figure 13.5: Configurating Elastic

Finally, we should be able to access the Kibana interface at `http://<ip>:5601`. We do not need any integration at this point; we will pick **Explore on my own**:

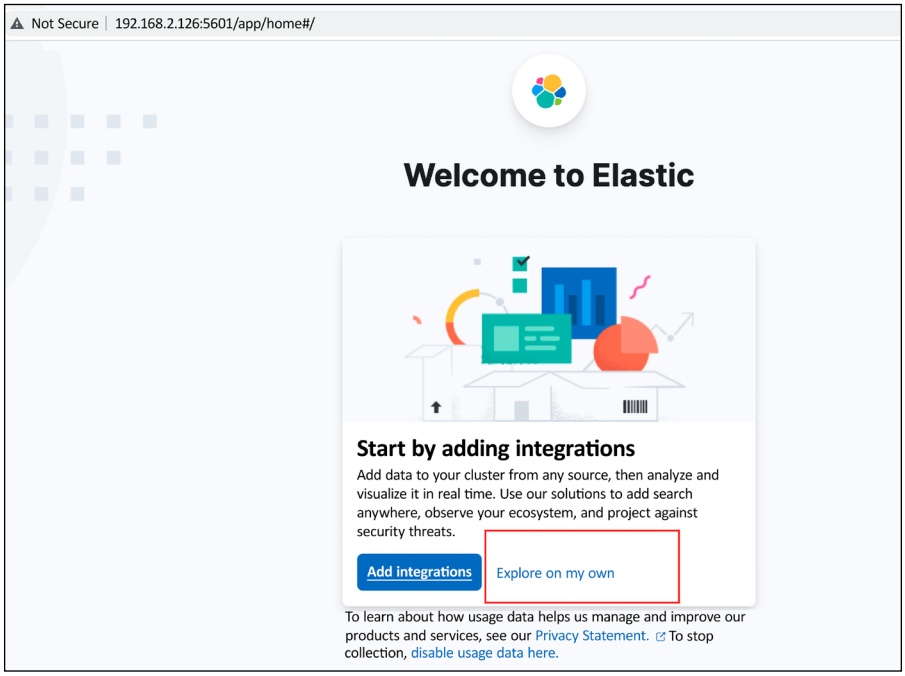


Figure 13.6:

We will be presented with an option to load some sample data. This is a great way to get our feet wet with the tool, so let's import this data:

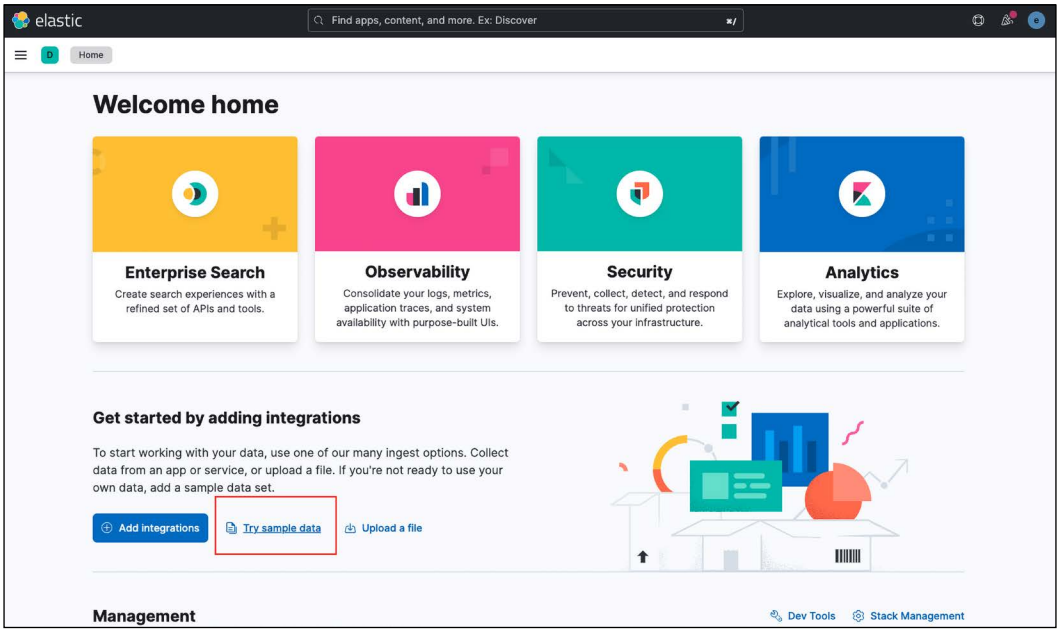


Figure 13.7: Kibana Home Page

We will choose **Try sample data** and add the sample eCommerce orders, sample flight data, and sample web logs:

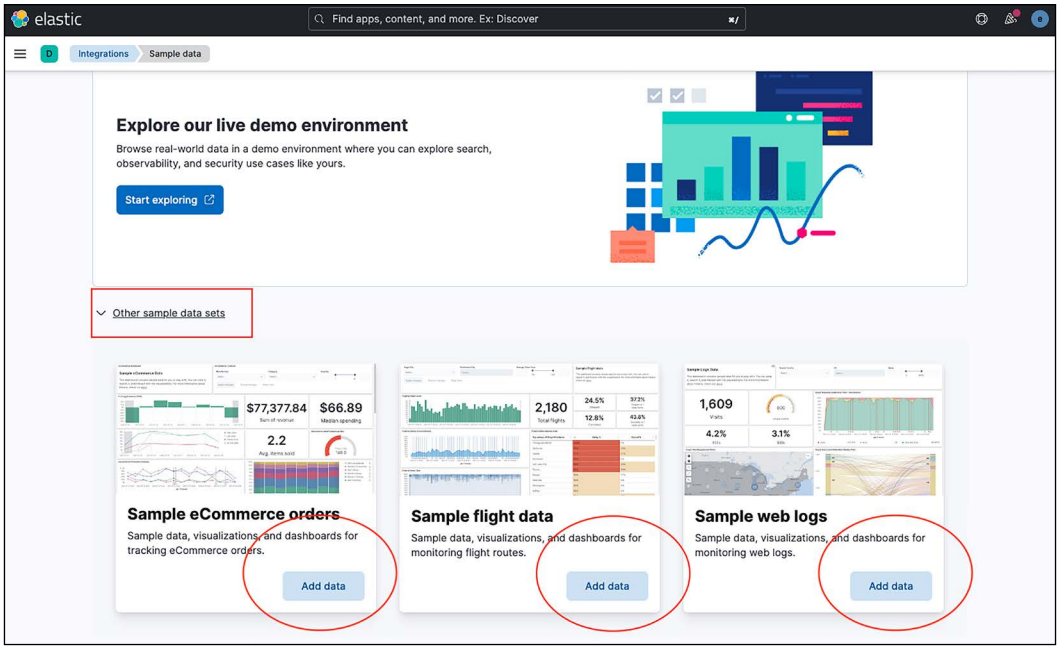


Figure 13.8: Adding Sample Data

To summarize, we now have Elasticsearch and Kibana running as containers with forwarded ports on the management host:

```
$ docker ps
CONTAINER ID   IMAGE
COMMAND                  CREATED              STATUS              PORTS
NAMES
f7d6d8842060   docker.elastic.co/kibana/
kibana:8.4.2                "/bin/tini -- /usr/l…"   42 minutes
ago       Up 42 minutes       0.0.0.0:5601->5601/tcp, :::5601->5601/tc
p                                             kibana
dc2a1fa15e3b   docker.elastic.co/elasticsearch/elasticsearch:8.4.2
"/bin/tini -- /usr/l…"   46 minutes ago       Up 46 minutes
0.0.0.0:9200->9200/tcp, :::9200->9200/tcp, 0.0.0.0:9300->9300/tcp,
:::9300->9300/tcp   elasticsearch
```

Great! We are almost done. The last piece of the puzzle is Logstash. Since we will be working with different Logstash configuration files, modules, and plugins, we will install it on the management host with a package instead of a Docker container. Logstash requires Java to run:

```
$ sudo apt install openjdk-11-jre-headless
$ java --version
openjdk 11.0.16 2022-07-19
OpenJDK Runtime Environment (build 11.0.16+8-post-Ubuntu-0ubuntu122.04)
OpenJDK 64-Bit Server VM (build 11.0.16+8-post-Ubuntu-0ubuntu122.04, mixed
mode, sharing)
```

We can download the Logstash bundled package:

```
$ wget https://artifacts.elastic.co/downloads/logstash/logstash-8.4.2-
linux-x86_64.tar.gz
$ tar -xvzf logstash-8.4.2-linux-x86_64.tar.gz
$ cd logstash-8.4.2/
```

We will modify a few fields in the Logstash configuration file:

```
$ vim config/logstash.yml
# change the following fields
node.name: mastering-python-networking
api.http.host: <your host ip>
api.http.port: 9600-9700
```

We will not start Logstash just yet. We will wait until we have installed the network-related plugins and created the necessary configuration file later in the chapter to start the Logstash process.

Let's take a moment to look at deploying the ELK Stack as a hosted service in the next section.

Elastic Stack as a service

Elasticsearch is a popular service available as a hosted option by both Elastic.co and other cloud providers. Elastic Cloud (`https://www.elastic.co/cloud/`) does not have an infrastructure of its own, but it offers the option to spin up deployments on AWS, Google Cloud Platform, or Azure. Because Elastic Cloud is built on other public cloud VM offerings, the cost will be a bit more than getting it directly from a cloud provider, such as AWS:

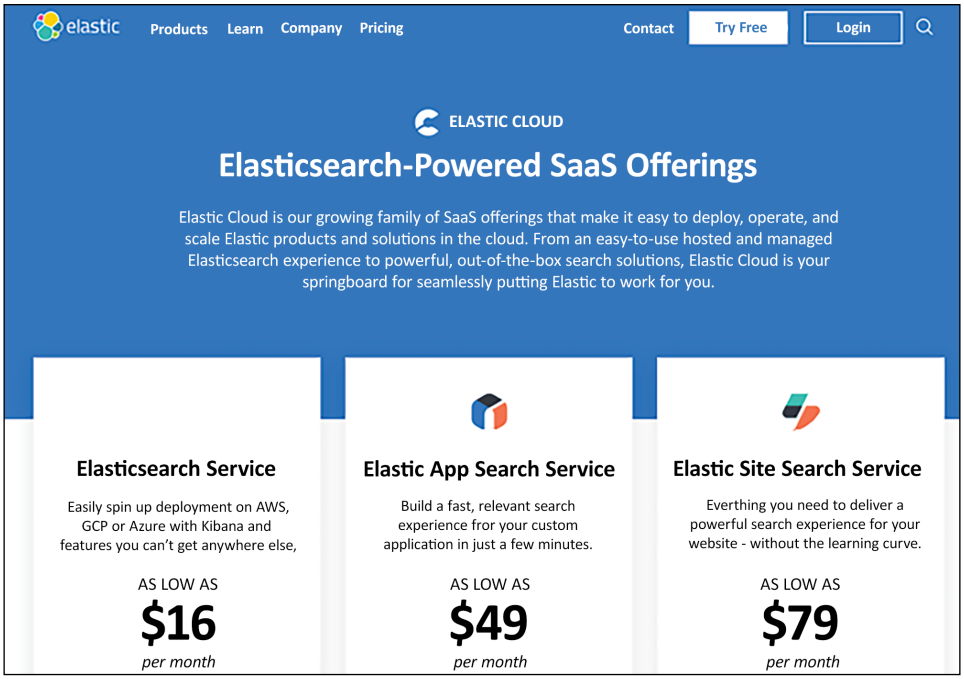


Figure 13.9: Elastic Cloud Offerings

AWS offers the hosted OpenSearch product (`https://aws.amazon.com/opensearch-service/`) tightly integrated with the existing AWS offerings. For example, AWS CloudWatch Logs can be streamed directly to the AWS OpenSearch instance (`https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_OpenSearch_Stream.html`).

From my own experience, as attractive as the Elastic Stack is for its advantages, it is a project that I feel is easy to get started but hard to scale without a steep learning curve. The learning curve is even steeper when we do not deal with Elasticsearch on a daily basis. If you, like me, want to take advantage of the features Elastic Stack offers but do not want to become a full-time Elastic engineer, I would highly recommend using one of the hosted options for production.

Which hosted provider to choose depends on your preference of cloud provider lockdown and if you want to use the latest features. Since Elastic Cloud is built by the folks behind the Elastic Stack project, they tend to offer the latest features faster than AWS. On the other hand, if your infrastructure is fully built in the AWS cloud, having a tightly integrated OpenSearch instance saves you the time and effort required to maintain a separate cluster.

Let's look at an end-to-end example from data ingestion to visualization in the next section.

First End-to-End example

One of the most common pieces of feedback from people new to Elastic Stack is the amount of detail you need to understand to get started. To get the first usable record in the Elastic Stack, the user needs to build a cluster, allocate master and data nodes, ingest the data, create the index, and manage it via the web or command-line interface. Over the years, Elastic Stack has simplified the installation process, improved its documentation, and created sample datasets for new users to get familiar with the tools before using the stack in production.



Running the components in Docker containers helps with some of the pain of installation but increases the complexity of maintenance. It is a balancing act to choose between running them in a virtual machine vs. containers.

Before we dig deeper into the different components of the Elastic Stack, it is helpful to look at an example that spans Logstash, Elasticsearch, and Kibana. By going over this end-to-end example, we will become familiar with the function that each component provides. When we look at each component in more detail later in the chapter, we can compartmentalize where the particular component fits into the overall picture.

Let's start by putting our log data into Logstash. We will configure each of the routers to export the log data to the Logstash server:

```
r[1-6]#sh run | i logging
logging host <logstash ip> vrf Mgmt-intf transport udp port 5144
```

On our Elastic Stack host, with all of the components installed, we will create a simple Logstash configuration that listens on UDP port 5144 and outputs the data to the console in JSON format as well as the Elasticsearch host:

```
echou@elk-stack-mpn:~$ cd logstash-8.4.2/
echou@elk-stack-mpn:~/logstash-8.4.2$ mkdir network_configs
echou@elk-stack-mpn:~/logstash-8.4.2$ touch network_configs/simple_config.
cfg
echou@elk-stack-mpn:~/logstash-8.4.2$ cat network_configs/simple_config.
conf
input {
  udp {
    port => 5144
    type => "syslog-ios"
  }
```

```
}
output {
  stdout { codec => json }
  elasticsearch {
    hosts => ["https://<elasticsearch ip>:9200"]
    ssl => true
    ssl_certificate_verification => false
    user => "elastic"
    password => "<password>"
    index => "cisco-syslog-%{+YYYY.MM.dd}"
  }
}
```

The configuration file consists of only an input section and an output section without modifying the data. The type, `syslog-ios`, is a name we picked to identify this index. In the `output` section, we configure the index name with variables representing today's date. We can run the Logstash process directly from the binary directory in the foreground:

```
$ ./bin/logstash -f network_configs/simple_config.conf
Using bundled JDK: /home/echou/Mastering_Python_Networking_Fourth_Edition/
logstash-8.4.2/jdk
[2022-09-23T13:46:25,876][INFO ][logstash.inputs.udp      ][main]
[516c12046954cb8353b87ba93e5238d7964349b0fa7fa80339b72c6baca637bb]
UDP listener started {:address=>"0.0.0.0:5144", :receive_buffer_
bytes=>"106496", :queue_size=>"2000"}
<skip>
```

By default, Elasticsearch allows automatic index generation when data is sent to it. We can generate some log data on the router by resetting the interface, reloading BGP, or simply going into the configuration mode and exiting out. Once there are some new logs generated, we will see the `cisco-syslog-<date>` index being created:

```
{"@timestamp":"2022-09-23T20:48:31.354Z", "log.level": "INFO",
"message":"[cisco-syslog-2022.09.23/B7PH3hxNSHqAegikXyp9kg]
create_mapping", "ecs.version": "1.2.0","service.name":"ES_
ECS","event.dataset":"elasticsearch.server","process.thread.
name":"elasticsearch[24808013b64b][masterService#updateTask]
[T#1]","log.logger":"org.elasticsearch.cluster.metadata.
MetadataMappingService","elasticsearch.cluster.uuid":"c-j9Dg8YTh2PstO3JFP9
AA","elasticsearch.node.id":"Pa4x3YJ-TrmFn5Pb2tObVw","elasticsearch.node.
name":"24808013b64b","elasticsearch.cluster.name":"docker-cluster"}
```

At this point, we can do a quick `curl` to see the index created on Elasticsearch. The `curl` command use the `insecure` flag to accommodate the self-signed certificate. The URL is in the "`https://<username>:<password>@<ip><port>/<path>`" format. "`_cat/indices/cisco*`" shows the category of indices, then match the indices name:

```
$ curl -X GET --insecure "https://elastic:-Rel0twWMUk8L-
ZtZr=I@192.168.2.126:9200/_cat/indices/cisco*"
yellow open cisco-syslog-2022.09.23 B7PH3hxNSHqAegikXyp9kg 1 1 9 0 21kb
21kb
```

We can now use Kibana to create the index by going to **Menu -> Management -> Stack Management**:

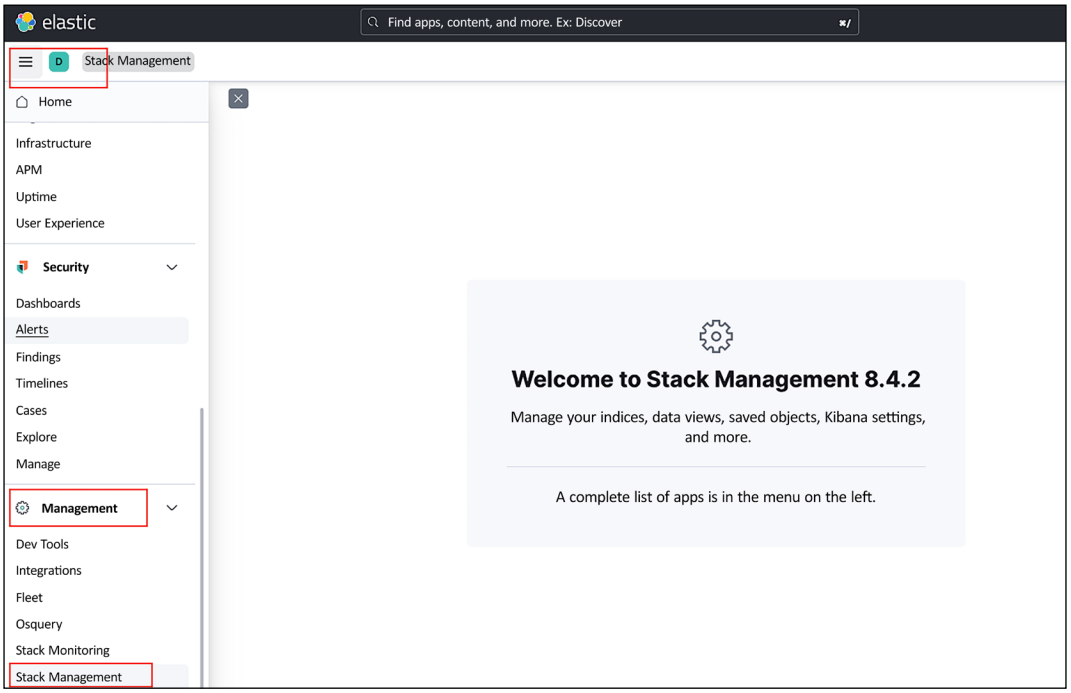


Figure 13.10: Stack Management

Under **Data -> Index Management**, we can see the newly created **cisco-syslog** index:

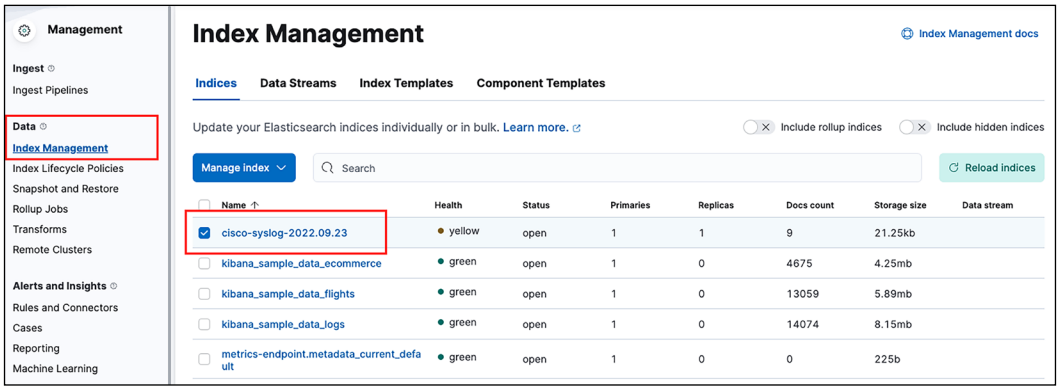


Figure 13.11: Index Management

We can now move to **Stack Management -> Kibana -> Data Views** to create a data view.

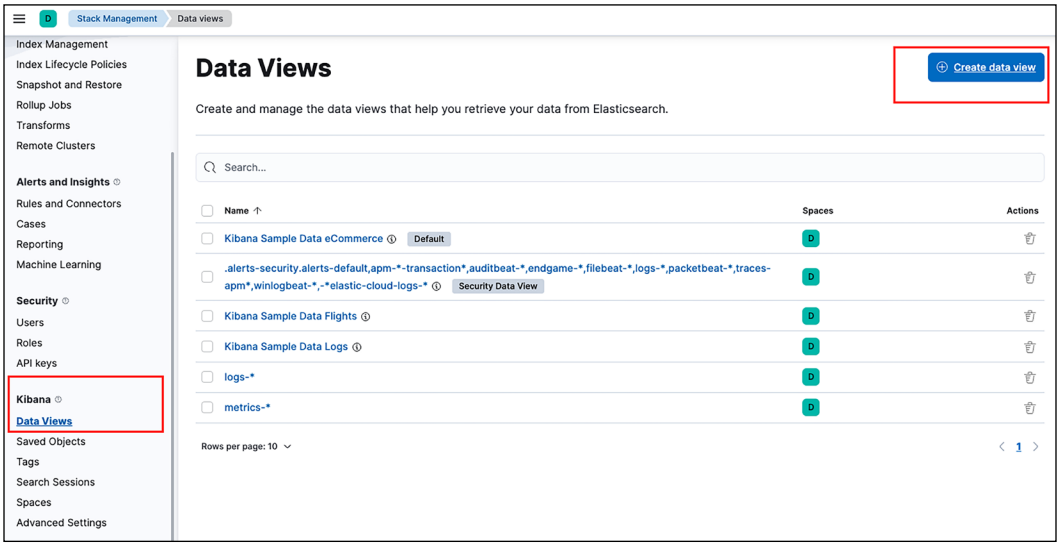


Figure 13.12: Create New Data Views Step 1

Since the index is already in Elasticsearch, we will only need to match the index name. Remember that our index name is a variable based on time; we can use a star wildcard (*) to match all the current and future indices starting with **cisco-syslog**:

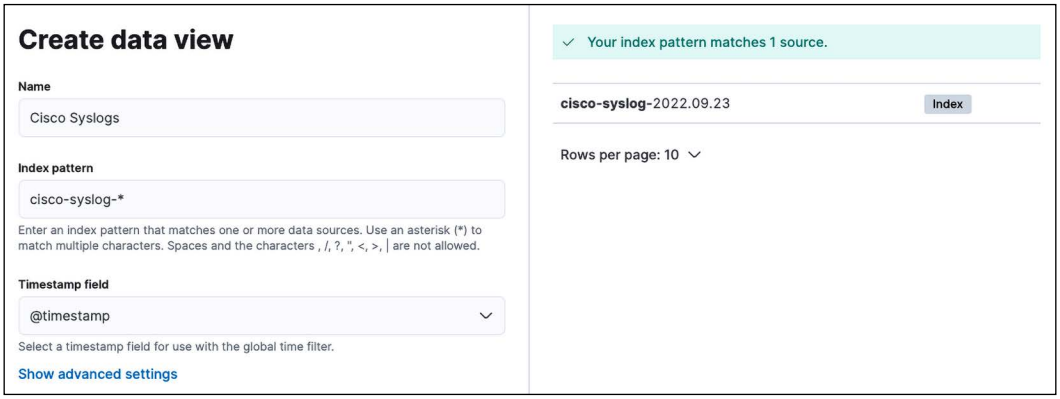


Figure 13.13: Create New Data Views Step 2

Our index is time-based, that is, we have a field that can be used as a timestamp, and we can search based on time. We should specify the field that we designated as the timestamp. In our case, Elasticsearch was already smart enough to pick a field from our syslog for the timestamp; we just need to choose it in the second step from the drop-down menu.

After the index pattern is created, we can use the **Menu -> Discover** (under **Analytics**) tab to look at the entries. Make sure you pick the right indices and the time range:

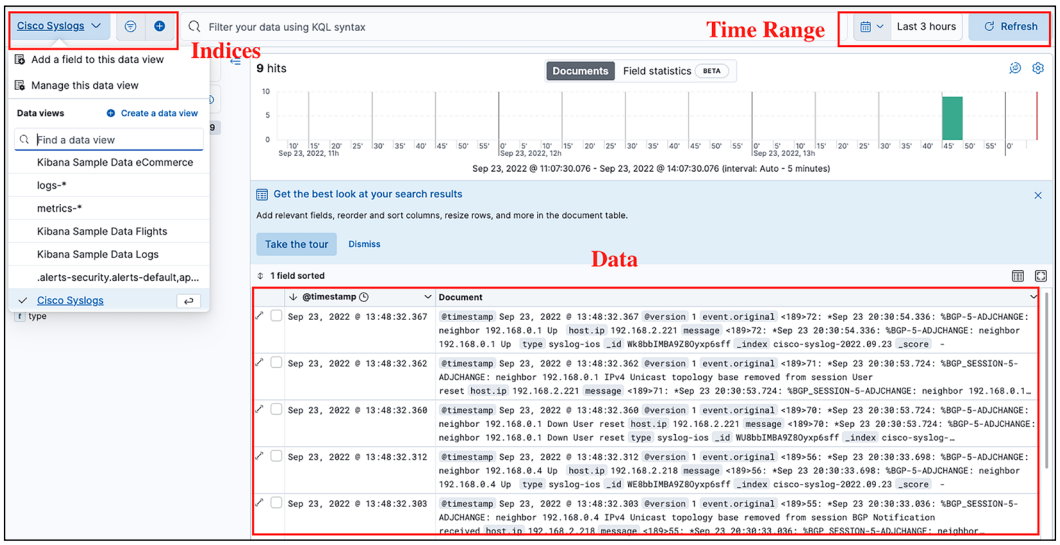


Figure 13.14: Elasticsearch Index Document Discovery

After we have collected some more log information, we can stop the Logstash process by using *Ctrl + C* on the Logstash process. This first example shows how we can leverage the Elastic Stack pipeline from data ingestion, storage, and visualization. The data ingestion used in Logstash (or Beats) is a continuous data stream that automatically flows into Elasticsearch. The Kibana visualization tool provides a way for us to analyze the data in Elasticsearch in a more intuitive way, then create a permanent visualization once we are happy with the result. There are more visualization graphs we can create with Kibana, which we will see more examples of later in the chapter.

Even with just one example, we can see that the most important part of the workflow is Elasticsearch. It is the simple RESTful interface, storage scalability, automatic indexing, and quick search result that gives the stack the power to adapt to our network analysis needs.

In the next section, we will look at how we can use Python to interact with Elasticsearch.

Elasticsearch with a Python client

We can interact with Elasticsearch via its HTTP RESTful API using a Python library. For instance, in the following example, we will use the `requests` library to perform a `GET` operation to retrieve information from the Elasticsearch host. For example, we know that `HTTP GET` for the following URL endpoint can retrieve the current indices starting with `kibana`:

```
$ curl -X GET --insecure "https://elastic:-Rel0twWMUk8L-
ZtZr=I@192.168.2.126:9200/_cat/indices/kibana*"
green open kibana_sample_data_ecommerce QcLgMu7CTEKNjeJeBxaD3w 1 0  4675 0
4.2mb 4.2mb
green open kibana_sample_data_logs      KPcJfMoSSaSs-kyqkuspKg 1 0 14074 0
8.1mb 8.1mb
green open kibana_sample_data_flights   q8MkYKooT8C5CQzbMMNTpg 1 0 13059 0
5.8mb 5.8mb
```

We can use the `requests` library to make a similar function in a Python script, `Chapter13_1.py`:

```
#!/usr/bin/env python3
import requests
from requests.packages.urllib3.exceptions import InsecureRequestWarning

# disable https verification check warning
requests.packages.urllib3.disable_warnings(InsecureRequestWarning)

def current_indices_list(es_host, index_prefix):
```

```
    current_indices = []
    http_header = {'content-type': 'application/json'}
    response = requests.get(es_host + "/_cat/indices/" + index_prefix +
"*", headers=http_header, verify=False)
    for line in response.text.split('\n'):
        if line:
            current_indices.append(line.split()[2])
    return current_indices

if __name__ == "__main__":
    username = 'elastic'
    password = '-Rel0twWMUk8L-ZtZr=I'
    es_host = 'https://'+username+':'+password+'@192.168.2.126:9200'
    indices_list = current_indices_list(es_host, 'kibana')
    print(indices_list)
```

Executing the script will give us a list of indices starting with `kibana`:

```
$ python Chapter13_1.py
['kibana_sample_data_ecommerce', 'kibana_sample_data_logs', 'kibana_
sample_data_flights']
```

We can also use the Python Elasticsearch client, `https://elasticsearch-py.readthedocs.io/en/master/`. It is designed as a thin wrapper around Elasticsearch's RESTful API to allow for maximum flexibility. Let's install it and run a simple example:

```
(venv) $ pip install elasticsearch
```

The example, `Chapter13_2`, simply connects to the Elasticsearch cluster and does a search for anything that matches the indices that start with `kibana`:

```
#!/usr/bin/env python3
from elasticsearch import Elasticsearch

es_host = Elasticsearch(["https://elastic:-Rel0twWMUk8L-
ZtZr=I@192.168.2.126:9200/"],
                        ca_certs=False, verify_certs=False)

res = es_host.search(index="kibana*", body={"query": {"match_all": {}}})
print("Hits Total: " + str(res['hits']['total']['value']))
```

By default, the result will return the first 10,000 entries:

```
$ python Chapter13_2.py
Hits Total: 10000
```

Using the simple script, the advantage of the client library is not obvious. However, the client library is very helpful when we create a more complex search operation, such as a scroll where we need to use the returned token per query to continue executing the subsequent queries until all the results are returned. The client can also help with more complicated administrative tasks, such as when we need to re-index an existing index. We will see more examples using the client library in the remainder of the chapter.

In the next section, we will look at more data ingestion examples from our Cisco device syslogs.

Data ingestion with Logstash

In the last example, we used Logstash to ingest log data from network devices. Let's build on that example and add a few more configuration changes in `network_config/config_2.cfg`:

```
input {
  udp {
    port => 5144
    type => "syslog-core"
  }
  udp {
    port => 5145
    type => "syslog-edge"
  }
}
filter {
 if [type] == "syslog-edge" {
    grok {
      match => { "message" => ".*" }
      add_field => [ "received_at", "%{@timestamp}" ]
    }
  }
}
output {
  stdout { codec => json }
  elasticsearch {
```

```
    hosts => ["https://192.168.2.126:9200"]
    <skip>
  }
}
```

In the input section, we will listen on two UDP ports, `5144` and `5145`. When the logs are received, we will tag the log entries with either `syslog-core` or `syslog-edge`. We will also add a filter section to the configuration to specifically match the `syslog-edge` type and apply a regular expression section, `Grok`, for the message section. In this case, we will match everything and add an extra field, `received_at`, with the value of the timestamp.

For more information on Grok, take a look at the following documentation: `https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html`.

We will change `r5` and `r6` to send syslog information to UDP port `5145`:

```
r[5-6]#sh run | i logging
logging host 192.168.2.126 vrf Mgmt-intf transport udp port 5145
```

When we start the Logstash server, we will see that both ports are now listening:

```
$ ./bin/logstash -f network_configs/config_2.conf
<skip>
[2022-09-23T14:50:42,097][INFO ][logstash.inputs.udp      ][main]
[212f078853a453d3d8a5d8c1df268fd628577245cd1b66acb06b9e1cb1ff8a10]
UDP listener started {:address=>"0.0.0.0:5144", :receive_buffer_
bytes=>"106496", :queue_size=>"2000"}
[2022-09-23T14:50:42,106][INFO ][logstash.inputs.udp      ][main]
[6c3825527b168b167846f4ca7dea5ef55e1437753219866bdcc2eb51aee53c84]
UDP listener started {:address=>"0.0.0.0:5145", :receive_buffer_
bytes=>"106496", :queue_size=>"2000"}
```

By separating out the entries using different types, we can specifically search for the types in the Kibana **Discover** dashboard:

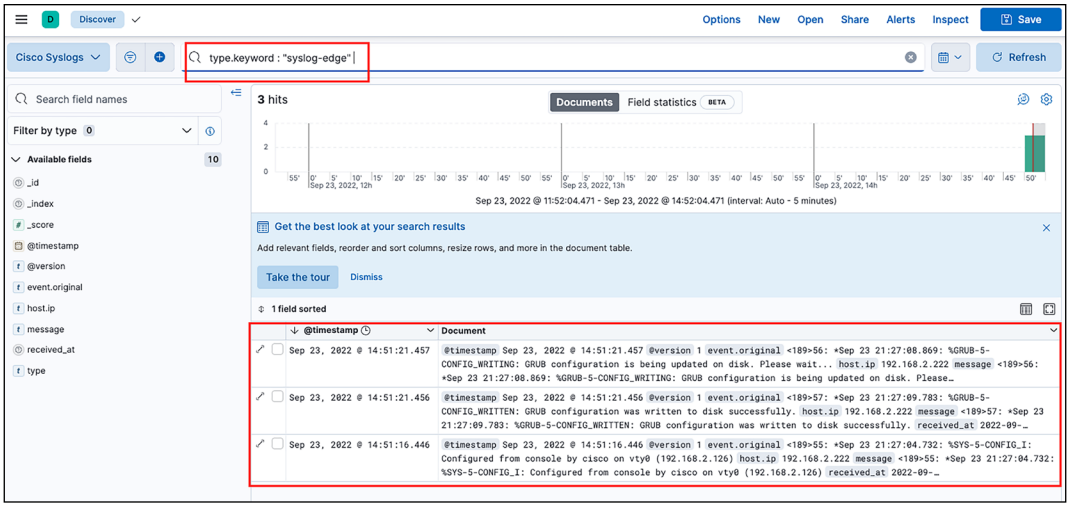


Figure 13.15: Syslog Index

If we expand on the entry with the `syslog-edge` type, we can see the new field that we added:



Figure 13.16: Syslog Timestamp

The Logstash configuration file provides many options in the input, filter, and output. In particular, the **Filter** section provides ways for us to enhance the data by selectively matching the data and further processing it before outputting it to Elasticsearch. Logstash can be extended with modules; each module provides a quick end-to-end solution for ingesting data and visualizations with purpose-built dashboards.

For more information on the Logstash modules, take a look at the following document: `https://www.elastic.co/guide/en/logstash/8.4/logstash-modules.html`.

Elastic Beats are similar to Logstash modules. They are single-purpose data shippers, usually installed as an agent, that collect data on the host and send the output data either directly to Elasticsearch or Logstash for further processing.

There are hundreds of different downloadable Beats, such as Filebeat, Metricbeat, Packetbeat, Heartbeat, and so on. In the next section, we will see how we can use Filebeat to ingest Syslog data into Elasticsearch.

Data ingestion with Beats

As good as Logstash is, the data ingestion process can get complicated and hard to scale. If we expand on our network log example, we can see that even with just network logs, it can get complicated trying to parse different log formats from IOS routers, NXOS routers, ASA firewalls, Meraki wireless controllers, and more. What if we need to ingest log data from Apache web logs, server host health, and security information? What about data formats such as NetFlow, SNMP, and counters? The more data we need to aggregate, the more complicated it can get.

While we cannot completely get away from aggregation and the complexity of data ingestion, the current trend is to move toward a more lightweight, single-purpose agent that sits as close to the data source as possible. For example, we can have a data collection agent installed directly on our Apache server specialized in collecting web log data; or we can have a host that only collects, aggregates, and organizes Cisco IOS logs. Elastic Stack collectively calls these lightweight data shippers Beats: `https://www.elastic.co/products/beats`.

Filebeat is a version of Elastic Beats software intended for forwarding and centralizing log data. It looks for the log file we specified in the configuration to be harvested; once it has finished processing, it will send the new log data to an underlying process that aggregates the events and outputs to Elasticsearch. In this section, we will look at using Filebeat with the Cisco modules to collect network log data.

Let's install Filebeat and set up the Elasticsearch host with the bundled visualization template and index:

```
$ $ curl -L -O https://artifacts.elastic.co/downloads/beats/filebeat/
filebeat-8.4.2-amd64.deb
$ sudo dpkg -i filebeat-8.4.2-amd64.deb
```

The directory layout can be confusing because they are installed in various `/usr`, `/etc/`, and `/var` locations:

Type	Description	Location
home	Home of the Filebeat installation.	`/user/share/filebeat`
bin	The location for the binary files	`/usr/share/filebeat/bin`
config	The location for configuration files.	`/etc/filebeat`
data	The location for persistent data files.	`/var/lib/filebeat`
logs	The location for the logs created by Filebeat.	`/var/log/filebeat`

Figure 13.17: Elastic Filebeat File Locations (source: https://www.elastic.co/guide/en/beats/filebeat/8.4/directory-layout.html)

We will make a few changes to the configuration file, `/etc/filebeat/filebeat.yml`, for the location of Elasticsearch and Kibana:

```
output.elasticsearch:
  # Array of hosts to connect to.
  hosts: ["192.168.2.126:9200"]

  # Protocol - either 'http' (default) or 'https'.
  protocol: "https"

  # Authentication credentials - either API key or username/password.
  username: "elastic"
  password: "changeme"
  ssl.verification_mode: none
setup.kibana:
  host: "192.168.2.126:5601"
```

Filebeat can be used to set up the index templates and example Kibana dashboards:

```
$ sudo filebeat setup --index-management -E output.logstash.enabled=false
-E 'output.elasticsearch.hots=["https://elastic:-Rel0twWMUk8L-
ZtZr=I@192.168.2.126:9200/"]'
$ sudo filebeat setup –dashboards
```

Let's enable the `cisco` module for Filebeat:

```
$ sudo filebeat modules enable cisco
Enabled cisco
```

Let's configure the `cisco` module for `syslog` first. The file is located under `/etc/filebeat/modules.d/cisco.yml`. In our case, I am also specifying a custom log file location:

```
- module: cisco
  ios:
    enabled: true
    var.input: syslog
    var.syslog_host: 0.0.0.0
    var.syslog_port: 514
    var.paths: ['/home/echou/syslog/my_log.log']
```

We can start, stop, and check the status of the Filebeat service using the common Ubuntu Linux command `service Filebeat [start|stop|status]`:

```
$ sudo service filebeat start
$ sudo service filebeat status
● filebeat.service - Filebeat sends log files to Logstash or directly to
Elasticsearch.
     Loaded: loaded (/lib/systemd/system/filebeat.service; disabled;
vendor preset: enabled)
     Active: active (running) since Fri 2022-09-23 16:06:09 PDT; 3s ago
<skip>
```

Modify or add UDP port 514 for syslog on our devices. We should be able to see the syslog information under the **filebeat-*** index search:

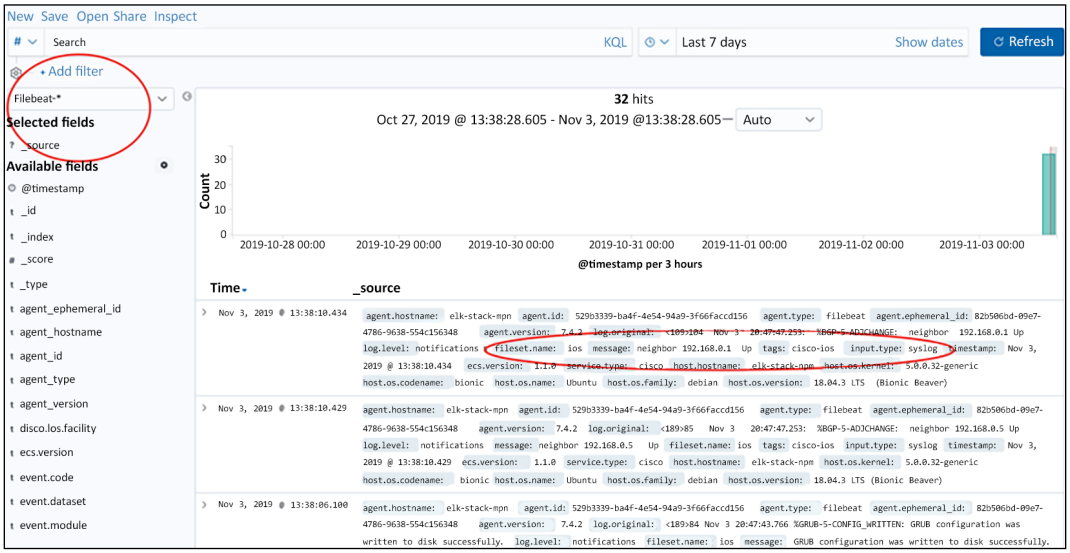


Figure 13.18: Elastic Filebeat Index

If we compare that to the previous syslog example, we can see that there are a lot more fields and meta information associated with each record, such as `agent.version`, `event.code`, and `event.severity`:

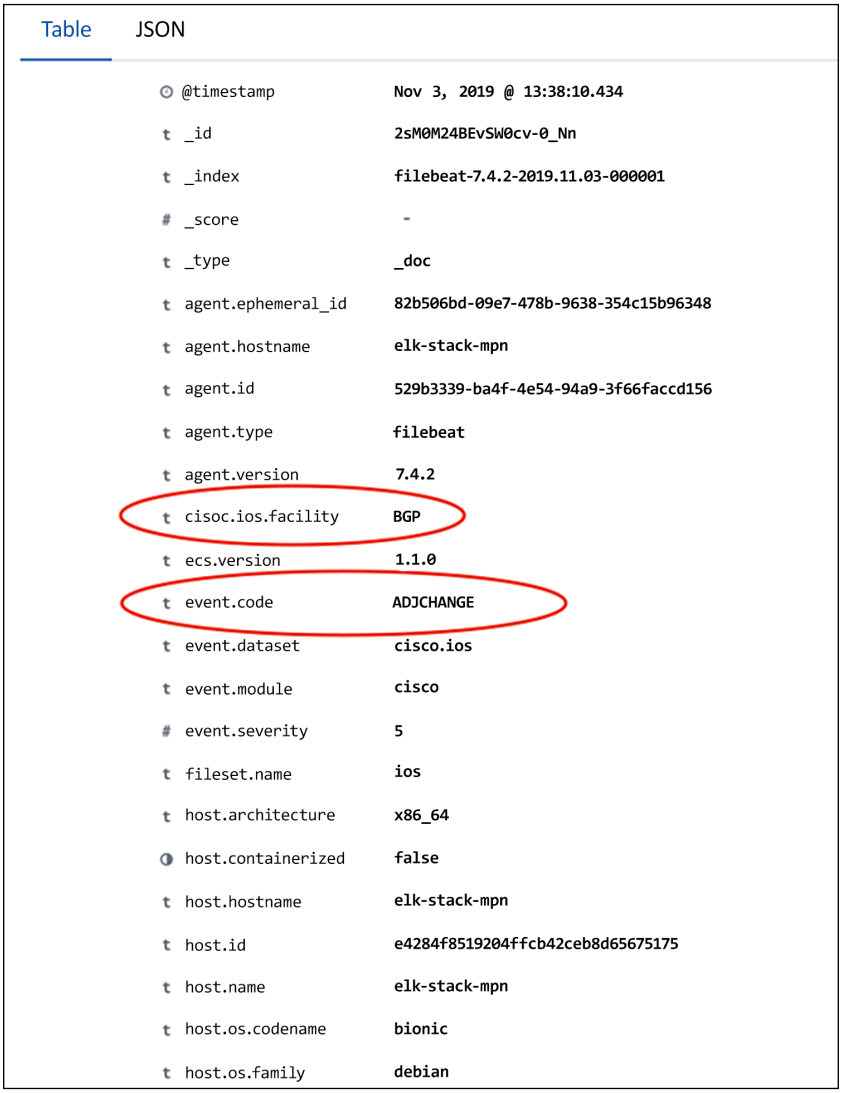


Figure 13.19: Elastic Filebeat Cisco Log

Why do the extra fields matter? Among other advantages, the fields make search aggregation easier, and this, in turn, allows us to graph the results better. We will see graphing examples in the upcoming section where we discuss Kibana.

Besides the `cisco` module, there are modules for Palo Alto Networks, AWS, Google Cloud, MongoDB, and many more. The most up-to-date module list can be viewed at `https://www.elastic.co/guide/en/beats/filebeat/8.4/filebeat-modules.html`.

What if we want to monitor NetFlow data? No problem, there is a module for that! We will run through the same process with the Cisco module by enabling the module and setting up the dashboard:

```
$ sudo filebeat modules enable netflow
$ sudo filebeat setup -e
```

Then, configure the module configuration file, `/etc/filebeat/modules.d/netflow.yml`:

```
- module: netflow
  log:
    enabled: true
    var:
      netflow_host: 0.0.0.0
      netflow_port: 2055
```

We will configure the devices to send the NetFlow data to port `2055`. If you need a refresher, please read the relevant configuration in *Chapter 8*, *Network Monitoring with Python – Part 2*. We should be able to see the new `netflow` data input type:

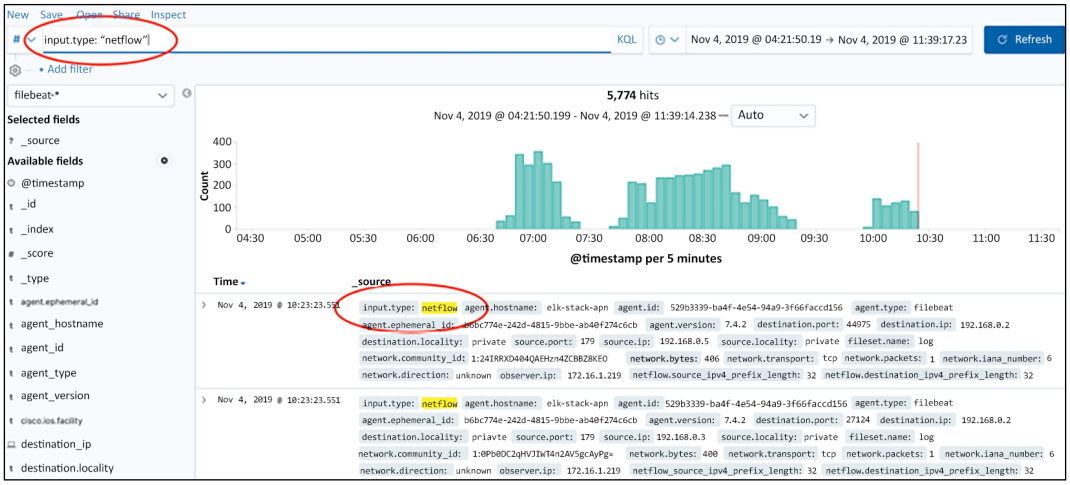


Figure 13.20: Elastic NetFlow Input

Remember that each module came pre-bundled with visualization templates? Not to jump ahead too much into visualization, but if we click on the **visualization** tab on the left panel, then search for **netflow**, we can see a few visualizations that were created for us:

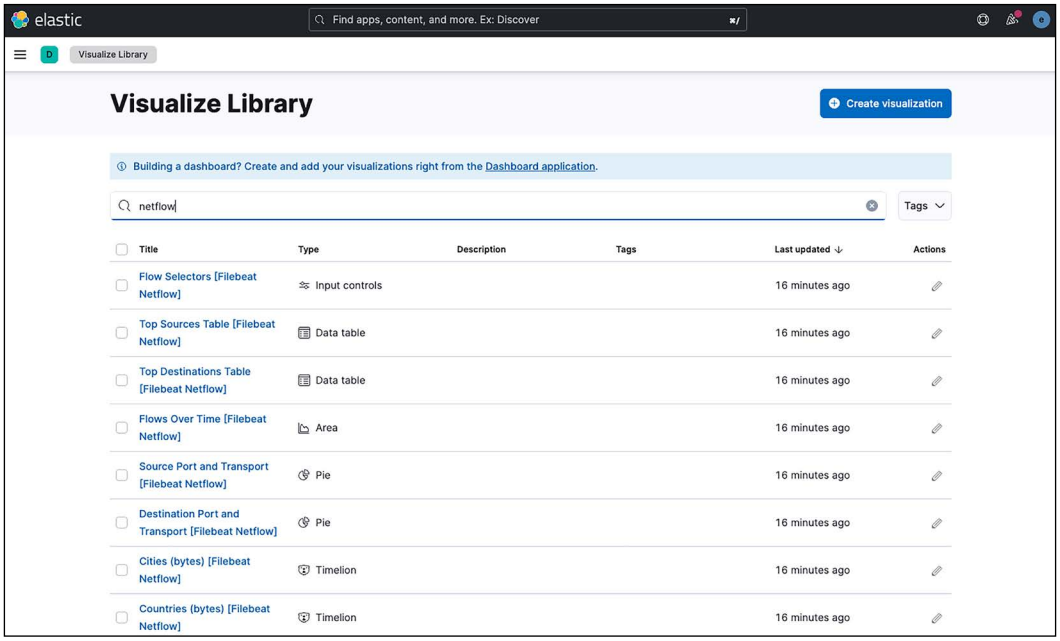


Figure 13.21: Kibana Visualization

Click on the **Conversation Partners [Filebeat Netflow]** option, which will give us a nice table of the top talkers that we can reorder by each of the fields:

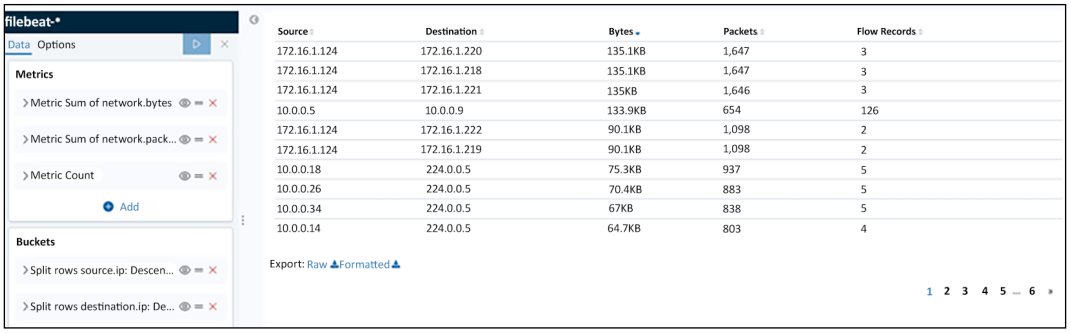


Figure 13.22: Kibana Table

In the next section, we will focus on the Elasticsearch part of the ELK Stack.

Search with Elasticsearch

We need more data in Elasticsearch to make the search and graph more interesting. I would recommend reloading a few of the lab devices to have the log entries for interface resets, BGP and OSPF establishments, as well as device boot-up messages. Otherwise, feel free to use the sample data we imported at the beginning of this chapter for this section.

If we look back at the `Chapter13_2.py` script example, when we searched, there were two pieces of information that could potentially change from each query: the index and query body. What I typically like to do is to break that information into input variables that I can dynamically change at runtime to separate the logic of the search and the script itself. Let's make a file called `query_body_1.json`:

```
{
  "query": {
    "match_all": {}
  }
}
```

We will create a script, `Chapter13_3.py`, that uses `argparse` to take the user input at the command line:

```
import argparse
parser = argparse.ArgumentParser(description='Elasticsearch Query
Options')
parser.add_argument("-i", "--index", help="index to query")
parser.add_argument("-q", "--query", help="query file")
args = parser.parse_args()
```

We can then use the two input values to construct the search the same way we have done before:

```
# load elastic index and query body information
query_file = args.query
with open(query_file) as f:
    query_body = json.loads(f.read())
# Elasticsearch instance
es_host = Elasticsearch(["https://elastic:<pass> @192.168.2.126:9200/"],
              ca_certs=False, verify_certs=False)
# Query both index and put into dictionary
index = args.index
res = es.search(index=index, body=query_body)
print(res['hits']['total']['value'])
```

We can use the `help` option to see what arguments should be supplied with the script. Here are the results when we use the same query against the two different indices we created:

```
$ python Chapter13_3.py --help
usage: Chapter12_3.py [-h] [-i INDEX] [-q QUERY]
Elasticsearch Query Options
optional arguments:
  -h, --help            show this help message and exit
  -i INDEX, --index INDEX
                        index to query
  -q QUERY, --query QUERY
                        query file
$ python3 Chapter13_3.py -q query_body_1.json -i "cisco*"
50
$ python3 Chapter13_3.py -q query_body_1.json -i "filebeat*"
10000
```

When developing our search, it usually takes a few tries before we get the result we are looking for. One of the tools Kibana provides is a developer console that allows us to play around with the search criteria and view the search results on the same page. The tool is under the menu section *Management for Dev Tools*.

For example, in the following figure, we execute the same search we have done now and we're able to see the returned JSON result. This is one of my favorite tools on the Kibana interface:

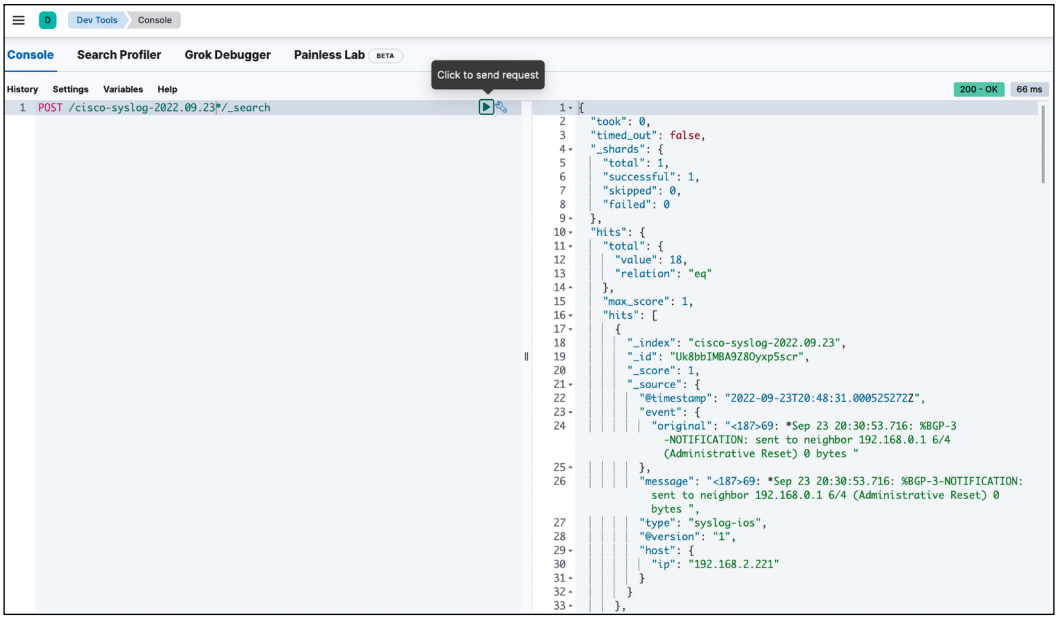


Figure 13.23: Kibana Dev Tools

Much of the network data is based on time, such as the log and NetFlow data we have collected. The values are taken at a snapshot in time, and we will likely group the value in a time scope. For example, we might want to know, "who are the NetFlow top talkers in the last 7 days?" or "which device has the most BGP reset messages in the last hour?" Most of these questions have to do with aggregation and time scope. Let's look at a query that limits the time range, `query_body_2.json`:

```
{
  "query": {
    "bool": {
    "filter": [
      {
        "range": {
          "@timestamp": {
            "gte": "now-10m"
          }
        }
      }
    ]
    }
  }
}
```

This is a Boolean query, `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html`, which means it can take a combination of other queries. In our query, we use the filter to limit the time range to be the last 10 minutes. We copy the `Chapter13_3.py` script to `Chapter13_4.py` and modify the output to grab the number of hits as well as a loop over the actual returned results list:

```
<skip>
res = es.search(index=index, body=query_body)
print("Total hits: " + str(res['hits']['total']['value']))
for hit in res['hits']['hits']:
    pprint(hit)
```

Executing the script will show that we only have 23 hits in the last 10 minutes:

```
$ python Chapter13_4.py -i "filebeat*" -q query_body_2.json
Total hits: 23
```

We can add another filter option in the query to limit the source IP via `query_body_3.json`:

```
{
  "query": {
    "bool": {
      "must": {
        "term": {
          "source.ip": "192.168.0.1"
        }
      },
<skip>
```

The result will be limited by both the source IP of r1's loopback IP in the last 10 minutes:

```
$ python Chapter12_4.py -i "filebeat*" -q query_body_3.json
Total hits: 18
```

Let's modify the search body one more time to add an aggregation, `https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket.html`, that takes a sum of all the network bytes from our previous search:

```
{
  "aggs": {
        "network_bytes_sum": {
          "sum": {
```

```
            "field": "network.bytes"
        }
    }
  },
  <skip>
}
```

The result will be different every time we run the script `Chapter13_5.py`. The current result is about 1 MB for me when I run the script consecutively:

```
$ python Chapter13_5.py -i "filebeat*" -q query_body_4.json
1089.0
$ python Chapter13_5.py -i "filebeat*" -q query_body_4.json
990.0
```

As you can see, building a search query is an iterative process; you typically start with a wide net and gradually narrow the criteria to fine-tune the results. In the beginning, you will probably spend a lot of time reading the documentation and searching for the exact syntax and filters. As you gain more experience under your belt, the search syntax will become easier. Going back to the previous visualization we saw from the `netflow` module setup for the NetFlow top talker, we can use the inspection tool to see the **Request** body:

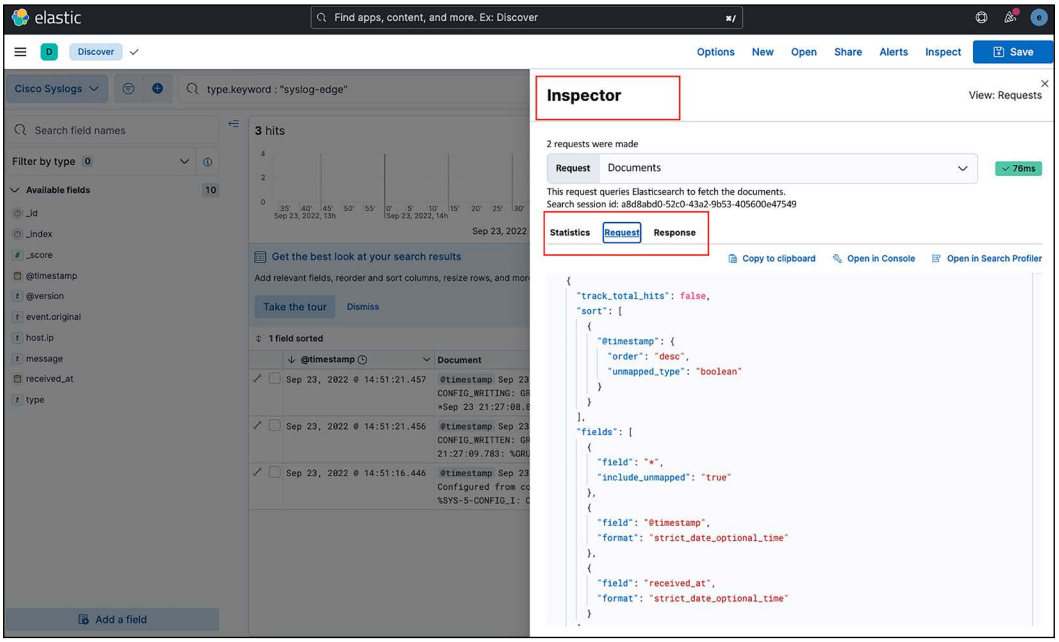


Figure 13.24: Kibana Request

We can put that into a query JSON file, `query_body_5.json`, and execute it with the `Chapter13_6.py` file. We will receive the raw data that the graph was based on:

```
$ python Chapter13_6.py -i "filebeat*" -q query_body_5.json
{'1': {'value': 8156040.0}, 'doc_count': 8256, 'key': '10.0.0.5'}
{'1': {'value': 4747596.0}, 'doc_count': 103, 'key': '172.16.1.124'}
{'1': {'value': 3290688.0}, 'doc_count': 8256, 'key': '10.0.0.9'}
{'1': {'value': 576446.0}, 'doc_count': 8302, 'key': '192.168.0.2'}
{'1': {'value': 576213.0}, 'doc_count': 8197, 'key': '192.168.0.1'}
{'1': {'value': 575332.0}, 'doc_count': 8216, 'key': '192.168.0.3'}
{'1': {'value': 433260.0}, 'doc_count': 6547, 'key': '192.168.0.5'}
{'1': {'value': 431820.0}, 'doc_count': 6436, 'key': '192.168.0.4'}
```

In the next section, let's take a deeper look at the visualization part of the Elastic Stack: Kibana.

Data visualization with Kibana

So far, we have used Kibana to discover data, manage indices in Elasticsearch, use developer tools to develop queries, and use a few other features. We also saw the pre-populated visualization charts from NetFlow, which gave us the top talker pair from our data. In this section, we will walk through the steps of creating our own graphs. We will start by creating a pie chart.

A pie chart is great at visualizing a portion of a component in relation to the whole. Let's create a pie chart based on the Filebeat index that graphs the top 10 source IP addresses based on the number of record counts. We will select **Dashboard -> Create dashboard -> Create visualization -> Pie**:

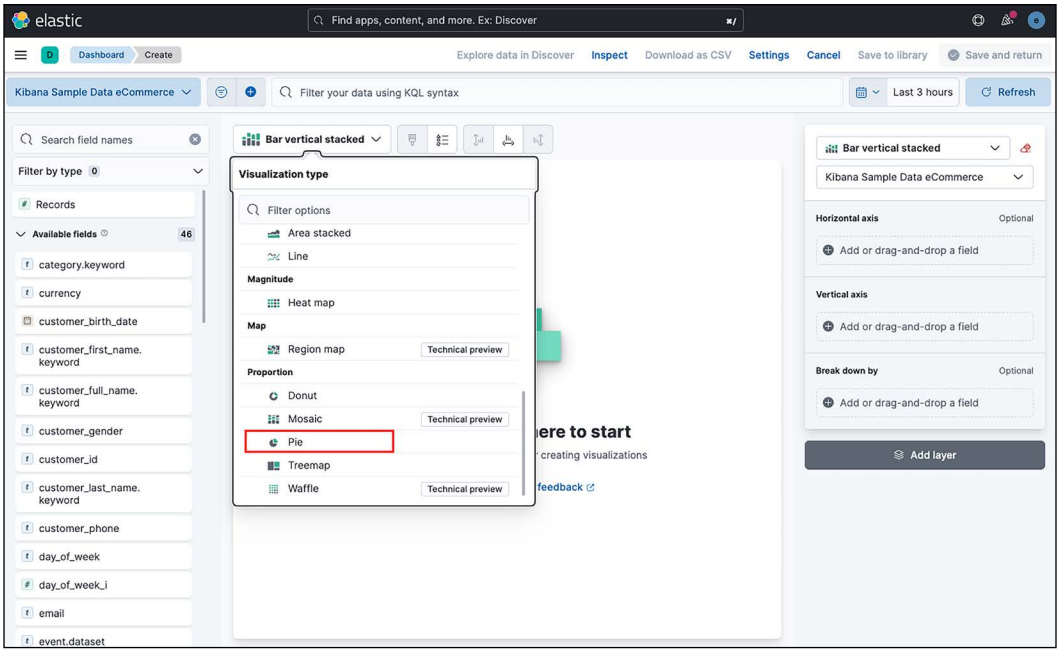


Figure 13.25: Kibana Pie Chart

Then we will type **netflow** in the search bar to pick our **[Filebeat NetFlow]** indices:

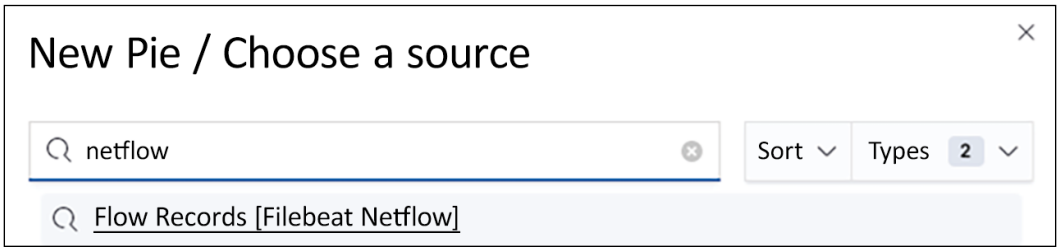


Figure 13.26: Kibana Pie Chart Source

By default, we are given the total count of all the records in the default time range. The time range can be dynamically changed:



Figure 13.27: Kibana Time Range

We can assign a custom label for the graph:

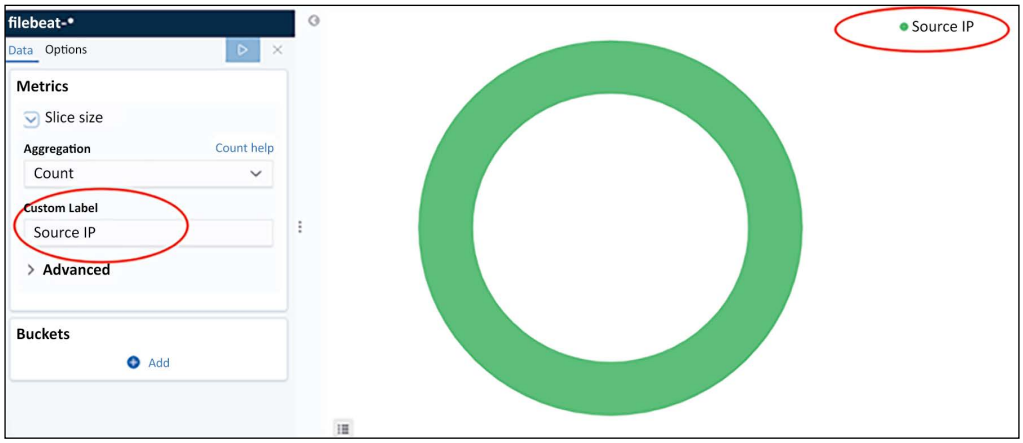


Figure 13.28: Kibana Chart Label

Let's click on the **Add** option to add more buckets. We will choose to split the slices, pick the terms for aggregation, and select the **source.ip** field from the drop-down menu. We will leave the **order Descending** but increase **Size** to **10**.

The change will only be applied when you click the **Apply** button at the top. It is a common mistake to expect the change to happen in real time when using a modern website and not by clicking on the **Apply** button:

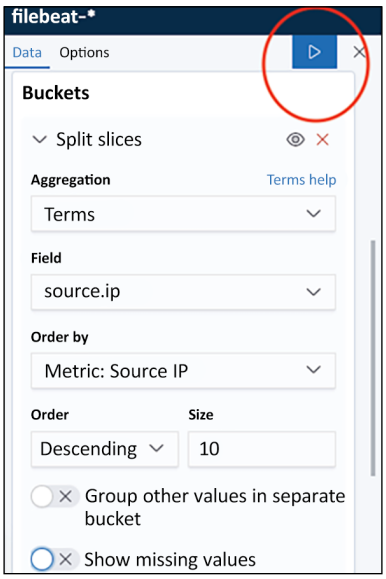


Figure 13.29: Kibana Play Button

We can click on **Options** at the top to turn off **Donut** and turn on **Show labels**:

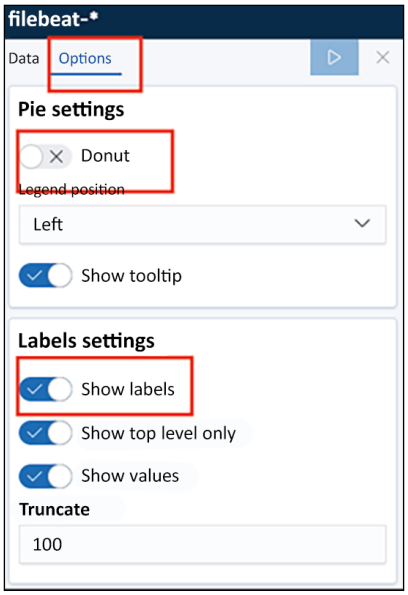


Figure 13.30: Kibana Chart Options

The final graph is a nice pie chart showing the top IP sources based on the number of document counts:

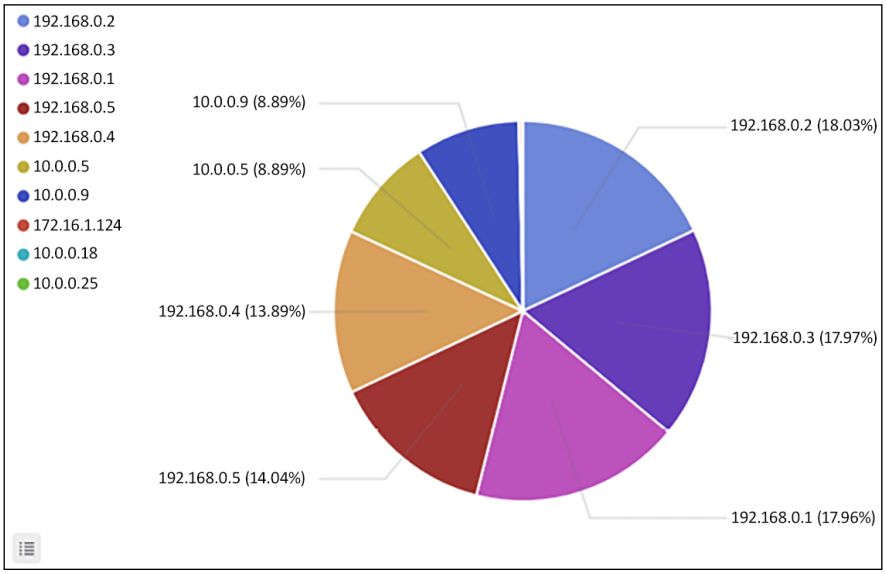


Figure 13.31: Kibana Pie Chart

As with Elasticsearch, the Kibana graph is also an iterative process that typically takes a few tries to get right. What if we split the result into different charts instead of slices on the same chart? Yeah, that is not very visually appealing:

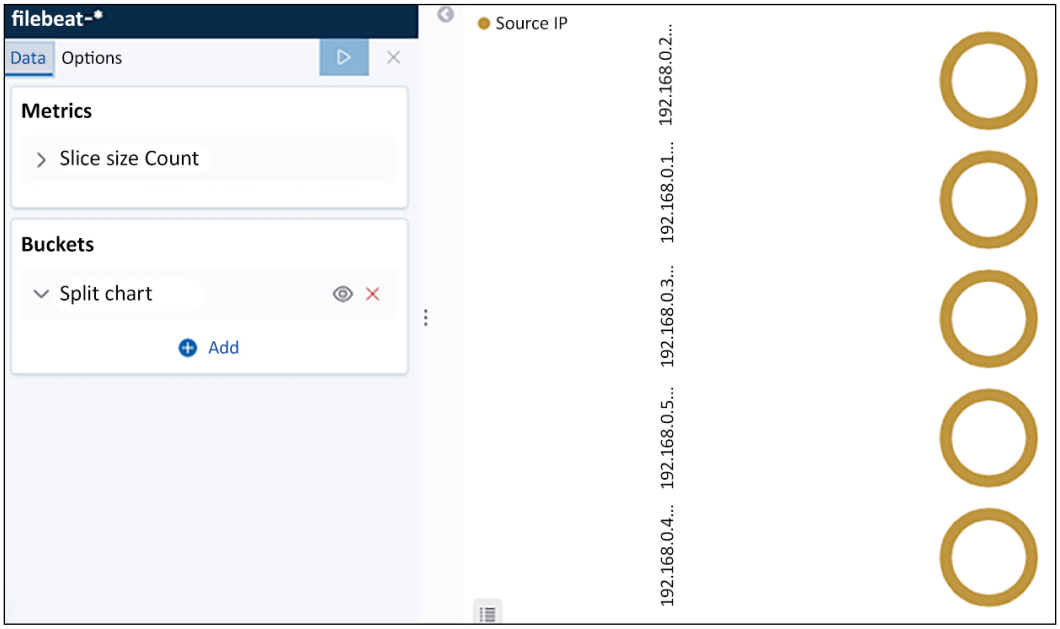


Figure 13.32: Kibana Split Chart

Let's stick to splitting things into slices on the same pie chart and change the time range to **Last 1 hour**, then save the chart so that we can come back to it later.

Note that we can also share the graph either in an embedded URL (if Kibana is accessible from a shared location) or a snapshot:

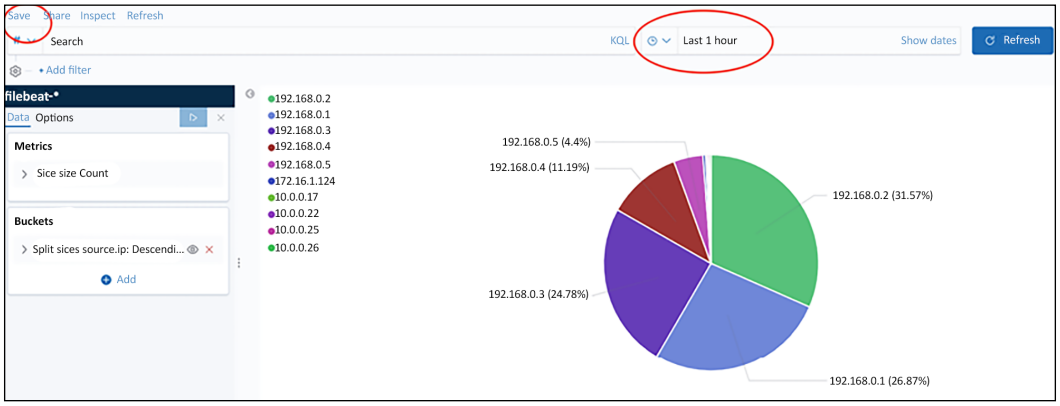


Figure 13.33: Kibana Save Chart

We can also do more with the metrics operations. For example, we can pick the data table chart type and repeat our previous bucket breakdown with the source IP. But we can also add a second metric by adding up the total number of network bytes per bucket:

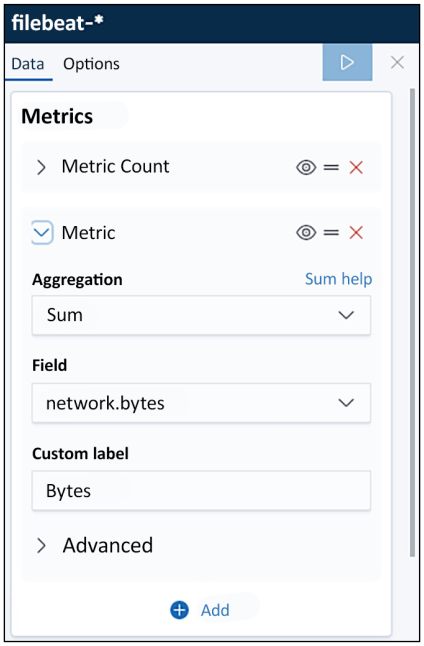


Figure 13.34: Kibana Metrics

The result is a table showing both the number of document counts as well as the sum of the network bytes. This can be downloaded in CSV format for local storage:

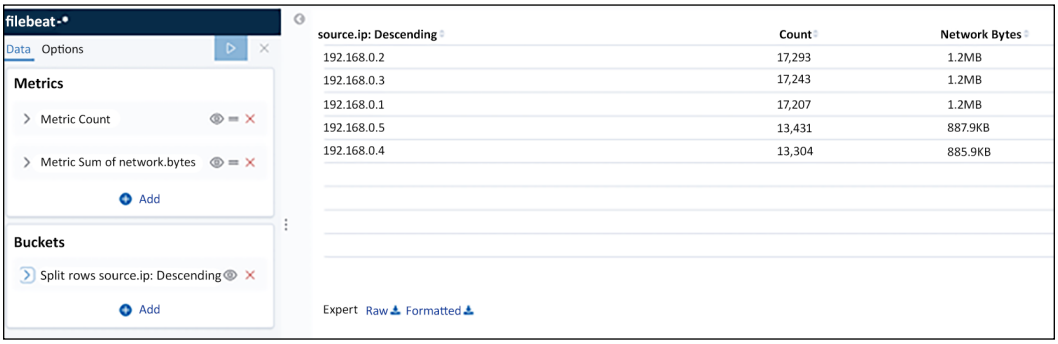


Figure 13.35: Kibana Tables

Kibana is a very powerful visualization tool in the Elastic Stack. We are just scratching the surface of its visualization capabilities. Besides many other graph options to better tell the story of your data, we can also group multiple visualizations onto a dashboard to be displayed. We can also use Timelion (`https://www.elastic.co/guide/en/kibana/8.4/timelion.html`) to group independent data sources for a single visualization or use Canvas (`https://www.elastic.co/guide/en/kibana/current/canvas.html`) as a presentation tool based on data in Elasticsearch.

Kibana is typically used at the end of the workflow to present our data meaningfully. We have covered the basic workflow from data ingestion to storage, retrieval, and visualization in the span of a chapter. It still amazes me that we can accomplish so much in a short period with the aid of an integrated, open source stack such as Elastic Stack.

Summary

In this chapter, we used the Elastic Stack to ingest, analyze, and visualize network data. We used Logstash and Beats to ingest the network syslog and NetFlow data. Then we used Elasticsearch to index and categorize the data for easier retrieval. Finally, we used Kibana to visualize the data. We used Python to interact with the stack and help us gain more insights into our data. Together, Logstash, Beats, Elasticsearch, and Kibana present a powerful all-in-one project that can help us understand our data better.

In the next chapter, we will look at using Git for network development with Python.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



14

Working with Git

We have worked on various aspects of network automation with Python, Ansible, and many other tools. In the examples you have been following along with in the first 13 chapters of this book, we have used over 150 files containing over 5,300 lines of code. That's pretty good for network engineers who may have been working primarily with the command-line interface before reading this book! With our new scripts and tools, we are ready to go out and conquer our network tasks, right? Well, not so fast, my fellow network ninjas.

There are several things we need to consider before we get into the meat of the tasks. We'll run through these considerations and talk about how the version-control (or source-control) system Git can help us out.

We'll cover the following topics:

- Content management considerations and Git
- An introduction to Git
- Setting up Git
- Git usage examples
- Git with Python
- Automating configuration backup
- Collaborating with Git

First, let's talk about what exactly are these considerations and the role Git can play in helping us to manage them.

Content management considerations and Git

The first thing that we must consider when creating code files is how to keep them in a location where they can be retrieved and used by us and others. Ideally, this location would be the only central place where the file is kept but also have backup copies available if needed. After the initial release of the code, we might add features and fix bugs in the future, so we would like a way to track these changes and keep the latest ones available for download. If the new changes do not work, we would like ways to roll back the changes and reflect the differences in the history of the file. This would give us a good idea of the evolution of the code files.

The second question is about the collaborative process between our team members. If we work with other network engineers, we will most likely need to work collectively on the files. These can be Python scripts, Ansible Playbooks, Jinja2 templates, INI-style configuration files, and many others. The point is that any kind of text-based file should be tracked with multiple inputs that everybody on the team should be able to see.

The third question is accountability. Once we have a system that allows for multiple inputs and changes, we need to mark these changes with an appropriate track record to reflect the owner of the change. The track record should also include a brief reason for the change so that the person reviewing the history can get an understanding of why the change was made.

These are some of the main challenges a version-control (or source-control) system, such as Git, tries to solve. To be fair, the process of version control can exist in forms other than a dedicated software system. For example, if I open my Microsoft Word program, the file constantly saves itself, and I can go back in time to revisit the changes or roll back to a previous version. That is one form of version control; however, the Word doc is hard to scale beyond my laptop. The version-control system we are focused on in this chapter is a standalone software tool with the primary purpose of tracking software changes.

There is no shortage of different source-control tools in software engineering, both proprietary and open-source. Some popular open-source version-control systems are CVS, SVN, Mercurial, and Git. In this chapter, we will focus on the source-control system Git. Many of the software we have used in this book use the same version control system to track changes, collaborate on features, and communicate with its users. We will be taking a more in-depth look at the tool. Git is the de facto version-control system for many large, open-source projects, including Python and the Linux kernel.



As of February 2017, the CPython development process has moved to GitHub. It has been a work in progress since January 2015. For more information, check out PEP 512 at: `https://www.python.org/dev/peps/pep-0512`.

Before we dive into the working examples of Git, let's look at the history and advantages of the Git system.

Introduction to Git

Git was created by Linus Torvalds, the creator of the Linux kernel, in April 2005. With his dry wit, he has affectionately called the tool "the information manager from hell." In an interview with the Linux Foundation, Linus mentioned that he felt source-control management was just about the least interesting thing in the computing world (`https://www.linuxfoundation.org/blog/2015/04/10-years-of-git-an-interview-with-git-creator-linus-torvalds/`). Nevertheless, he created the tool after a disagreement between the Linux kernel developer community and BitKeeper, the proprietary system they were using at the time.

What does the name Git stand for? In British English slang, a git is an insult denoting an unpleasant, annoying, childish person. With his dry humor, Linus said he is an egotistical bastard and that he named all of his projects after himself. First Linux, now Git. However, some suggested that the name is short for **Global Information Tracker** (**GIT**). You can be the judge of which explanation you like better.

The project came together quickly. About 10 days after its creation (yeah, you read that right), Linus felt the basic ideas for Git were right and started to commit the first Linux kernel code with Git. The rest, as they say, is history. More than ten years after its creation, it is still meeting all the expectations of the Linux kernel project. It took over as the version-control system for many other open-source projects despite many developers' inherent inertia in switching source-control systems. For the Python code base, after many years of hosting the code at Mercurial (`https://hg.python.org/`), the project was switched to Git on GitHub in February 2017.

Now that we've been through the history of Git let's take a look at some of its benefits.

Benefits of Git

The success of hosting large and distributed open-source projects, such as the Linux kernel and Python, speaks to the advantages of Git. I mean, if this tool is good enough for the software development for the most popular operating system (in my opinion) and the most popular programming language (again, my opinion only) in the world, it is probably good enough for my hobby project.

The popularity of Git is especially significant given that it is a relatively new source-control tool, and people do not tend to switch to a new tool unless it offers significant advantages over the old tool. Let's look at some of the benefits of Git:

- **Distributed development**: Git supports parallel, independent, and simultaneous development in private repositories offline. Many other version control systems require constant synchronization with a central repository. The distributed and offline nature of Git allows significantly greater flexibility for the developers.
- **Scale to handle thousands of developers**: The number of developers working on different parts of some open-source projects is in the thousands. Git supports the integration of their work reliably.
- **Performance**: Linus was determined to make sure Git was fast and efficient. To save space and transfer time for the sheer volume of updates for the Linux kernel code alone, compression and a delta check were used to make Git fast and efficient.
- **Accountability and immutability**: Git enforces a change log on every commit that changes a file, so there is a trail for all the changes and the reason behind them. The data objects in Git cannot be modified after they are created and placed in the database, making them immutable. This further enforces accountability.
- **Atomic transactions**: The integrity of the repository is ensured as the different but related change is performed either altogether or not at all. This will ensure the repository is not left in a partially changed or corrupted state.
- **Complete repositories**: Each repository has a complete copy of all historical revisions of every file.
- **Free, as in freedom**: The origin of the Git tool was born out of the disagreement between Linux and BitKeeper VCS as to whether software should be free and whether one should reject commercial software on principle, so it makes sense that the tool has a very liberal usage license.

Let's look at some of the terms used in Git before we go deeper into it.

Git Terminology

Here are some Git terms we should be familiar with:

- **Ref**: The name that begins with `refs` and points to an object.
- **Repository**: This is a database that contains all of a project's information, files, metadata, and history. It contains a collection of refs for all the collections of objects.

- **Branch**: This is an active line of development. The most recent commit is the `tip` or the `HEAD` of that branch. A repository can have multiple branches, but your `working tree` or `working directory` can only be associated with one branch. This is sometimes referred to as the current or `checked out` branch.
- **Checkout**: This is the action of updating all or part of the working tree to a particular point.
- **Commit**: This is a point in time in Git history, or it can mean storing a new snapshot in the repository.
- **Merge**: This is the action to bring the content of another branch into the current branch. For example, I am merging the `development` branch with the `master` branch.
- **Fetch**: This is the action of getting the content from a remote repository.
- **Pull**: Fetching and merging a repository.
- **Tag**: This is a mark at a point in time in a repository that is significant.

This is not a complete list; please refer to the Git glossary, `https://git-scm.com/docs/gitglossary`, for more terms and their definitions.

Finally, before getting into the actual setup and uses of Git, let's talk about the important distinction between Git and GitHub; one that is easily overlooked by engineers unfamiliar with the two.

Git and GitHub

Git and GitHub are not the same things. Sometimes, for engineers who are new to version-control systems, this is confusing. Git is a revision-control system, while GitHub, `https://github.com/`, is a centralized hosting service for Git repositories. The company, GitHub, was launched in 2008 and was acquired by Microsoft in 2018 but continued to operate independently.

Because Git is a decentralized system, GitHub stores a copy of our project's repository, just like any other distributed offline copies. We often designate the GitHub repository as the project's central repository, and all other developers push and pull their changes to and from that repository.

After GitHub was acquired by Microsoft in 2018, `https://blogs.microsoft.com/blog/2018/10/26/microsoft-completes-github-acquisition/`, many in the developer community worried about the independence of GitHub. As described in the press release, "GitHub will retain its developer-first ethos, operate independently, and remain an open-source platform." GitHub takes this idea of being the centralized repository in a distributed system further by using the `fork` and `pull requests` mechanisms. For projects hosted on GitHub, the project maintainers typically encourage other developers to `fork` the repository, or make a copy of the repository, and work on it as their copied repository.

After making changes, they can send a `pull request` to the main project, and the project maintainers can review the changes and `commit` the changes if they see fit. GitHub also adds the web interface to the repositories beside the command line; this makes Git more user-friendly.

Now that we've differentiated Git and GitHub, we can start properly! First, let's talk about setting up Git.

Setting up Git

So far, we have been using Git just to download files from GitHub. In this section, we will go a bit further by setting up Git locally so we can start committing our files. I will use the same Ubuntu 22.04 LTS management host in the example. If you are using a different version of Linux or other operating systems, a quick search of the installation process should land you in the right set of instructions.

If you have not done so already, install Git via the `apt` package-management tool:

```
$ sudo apt update
$ sudo apt install -y git
$ git --version
git version 2.34.1
```

Once `git` is installed, we need to configure a few things so that our commit messages can contain the correct information:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@domain.com"
$ git config --list
user.name=Your Name
user.email=email@domain.com
```

Alternatively, you can modify the information in the `~/.gitconfig` file:

```
$ cat ~/.gitconfig
[user]
name = Your Name
email = email@domain.com
```

There are many options in Git that we can change, but the name and email are the ones that allow us to commit the change without getting a warning. Personally, I like to use the Vim text editor, instead of the default Emac, for typing commit messages:

```
(optional)
$ git config --global core.editor "vim"
$ git config --list
user.name=Your Name
user.email=email@domain.com
core.editor=vim
```

Before we move on to using Git, let's go over the idea of a `gitignore` file.

Gitignore

There are files you do not want Git to check into GitHub or other repositories, such as files with passwords, API keys, or other sensitive information. The easiest way to prevent files from being accidentally checked into a repository is to create a `.gitignore` file in the repository's top-level folder. Git will use the `gitignore` file to determine which files and directories should be ignored before committing. The `gitignore` file should be committed to the repository as early as possible and be shared with other users.

Imagine the panic you would feel if you accidentally checked your group API key into a public Git repository. It is usually helpful to create the `gitignore` file when creating a new repository. In fact, GitHub provides an option to do just that when you create a repository on its platform. This file can include language-specific files. For example, let's exclude the Python `Byte-compiled` files:

```
# Byte-compiled / optimized / DLL files
  pycache /
*.py[cod]
*$py.class
```

We can also include files that are specific to your operating system:

```
# OSX
# =========================
.DS_Store
.AppleDouble
.LSOverride
```

You can learn more about `.gitignore` on GitHub's help page: `https://help.github.com/articles/ignoring-files/`. Here are some other references:

- Gitignore manual: `https://git-scm.com/docs/gitignore`
- GitHub's collection of `.gitignore` templates: `https://github.com/github/gitignore`

- Python language `.gitignore` example: `https://github.com/github/gitignore/blob/master/Python.gitignore`
- The `.gitignore` file for this book's repository: `https://github.com/PacktPublishing/Mastering-Python-Networking-Fourth-Edition/blob/main/.gitignore`.

I see the `.gitignore` file as a file that should be created simultaneously with any new repository. That is why this concept is introduced as early as possible. We will look at some of the Git usage examples in the next section.

Git Usage Examples

In my experience, when we work with Git, we will likely use the command line and the various options. The graphical tools are useful when we need to trace back changes, look at logs, and compare commit differences, but we rarely use them for normal branching and commits. We can look at Git's command-line option by using the `help` option:

```
$ git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

We will create a `repository` and create a file inside the repository:

```
$ mkdir TestRepo-1
$ cd TestRepo-1/
$ git init
Initialized empty Git repository in /home/echou/Mastering_Python_
Networking_third_edition/Chapter13/TestRepo-1/.git/
$ echo "this is my test file" > myFile.txt
```

When the repository was initialized with Git, a new hidden folder of `.git` was added to the directory. It contains all the Git-related files:

```
$ ls -a
.  ..  .git  myFile.txt
$ ls .git/
branches  config  description  HEAD  hooks  info  objects  refs
```

There are several locations where Git receives its configurations in a hierarchical format. The files are read from the `system`, `global`, and `repository` by default. The more specific the repository's location, the higher the override preference. For example, the repository configuration will override the global configuration. You can use the `git config -l` command to see the aggregated configuration:

```
$ ls .git/config
.git/config
$ ls ~/.gitconfig
/home/echou/.gitconfig
$ git config -l
user.name=Eric Chou
user.email=<email>
core.editor=vim
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

When we create a file in the repository, it is not tracked. For `git` to be aware of the file, we need to add the file:

```
$ git status
On branch master
Initial commit
Untracked files:
     (use "git add <file>..." to include in what will be committed)
myFile.txt
nothing added to commit but untracked files present (use "git add" to
track)
$ git add myFile.txt
$ git status
On branch master
Initial commit
Changes to be committed:
   (use "git rm --cached <file>..." to unstage)
new file: myFile.txt
```

When you add the file, it is in a staged status. To make the changes official, we will need to commit the change:

```
$ git commit -m "adding myFile.txt"
[master (root-commit) 5f579ab] adding myFile.txt
 1 file changed, 1 insertion(+)
 create mode 100644 myFile.txt
$ git status
On branch master
nothing to commit, working directory clean
```

In the last example, we provided the `commit` message with the `-m` option when we issued the `commit` statement. If we did not use the option, we would have been taken to a page to provide the commit message. In our scenario, we configured the text editor to be Vim, so we can use it to edit the message.

Let's make some changes to the file and `commit` it again. Notice that after the file has been changed, Git knows the file has been modified:

```
$ vim myFile.txt
$ cat myFile.txt
this is the second iteration of my test file
$ git status
On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: myFile.txt
$ git add myFile.txt
$ git commit -m "made modifications to myFile.txt"
[master a3dd3ea] made modifications to myFile.txt
1 file changed, 1 insertion(+), 1 deletion(-)
```

The `git commit` number is an `SHA-1 hash`, which is an important feature. If we had followed the same step on another computer, our `SHA-1 hash` value would be the same. This is how Git knows the two repositories are identical even when they are worked on in parallel.



If you have ever wondered about the `SHA-1 hash` value being accidentally or purposely modified to overlap, there is an interesting article on the GitHub blog about detecting this `SHA-1 hash` collision: `https://github.blog/2017-03-20-sha-1-collision-detection-on-github-com/`.

We can show the history of the commits with `git log`. The entries are shown in reverse chronological order; each commit shows the author's name and email address, the date, the log message, as well as the internal identification number of the commit:

```
$ git log
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:49:02 2019 -0800
    made modifications to myFile.txt
commit 5d7c1c8543c8342b689c66f1ac1fa888090ffa34
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:46:32 2019 -0800
    adding myFile.txt
```

We can also show more details about the change using the commit ID:

```
(venv) $ git show ff7dc1a40e5603fed552a3403be97addefddc4e9
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:49:02 2019 -0800
    made modifications to myFile.txt
diff --git a/myFile.txt b/myFile.txt
index 6ccb42e..69e7d47 100644
--- a/myFile.txt
+++ b/myFile.txt
@@ -1 +1 @@
-this is my test file
+this is the second iteration of my test file
```

If you need to revert the changes you have made, you can choose between `revert` and `reset`. The former changes all the files for a specific commit back to their state before the commit:

```
$ git revert ff7dc1a40e5603fed552a3403be97addefddc4e9
[master 75921be] Revert "made modifications to myFile.txt"
 1 file changed, 1 insertion(+), 1 deletion(-)
$ cat myFile.txt
this is my test file
```

The `revert` command will keep the `commit` you reverted and make a new `commit`. You will be able to see all the changes up to that point, including the revert:

```
$ git log
commit 75921bedc83039ebaf70c90a3e8d97d65a2ee21d (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 09:00:23 2019 -0800
    Revert "made modifications to myFile.txt"
    This reverts commit ff7dc1a40e5603fed552a3403be97addefddc4e9.
     On branch master
     Changes to be committed:
            modified:   myFile.txt
```

The `reset` option will reset the status of your repository to an older version and discard all the changes in between:

```
$ git reset --hard ff7dc1a40e5603fed552a3403be97addefddc4e9
HEAD is now at ff7dc1a made modifications to myFile.txt
$ git log
commit ff7dc1a40e5603fed552a3403be97addefddc4e9 (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:49:02 2019 -0800
    made modifications to myFile.txt
commit 5d7c1c8543c8342b689c66f1ac1fa888090ffa34
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 08:46:32 2019 -0800
    adding myFile.txt
```

I like to keep all the history, including any rollbacks I have done. Therefore, when I need to roll back a change, I usually pick `revert` instead of `reset`. In this section, we have seen how we can work with individual files. In the next section, let's look at how we can work with a collection of files that is grouped into a particular `bundle`, called `branch`.

Git Branch

A `branch` in `git` is a line of development within a repository. Git allows many branches and thus different lines of development within a repository. By default, we have the master branch.



A few years back, the default branch for GitHub was renamed "main": `https://github.com/github/renaming`. We will see both in the field.

There are many reasons for branching; there are no hard-set rules about when to branch or work on the master/main branch directly. Most of the time, we create a branch when there is a bug fix, a customer software release, or a development phase. In our example, let us create a branch that represents development, appropriately named the `dev` branch:

```
$ git branch dev
$ git branch
  dev
* master
```

Notice we need to specifically move into the `dev branch` after creation. We do that with `checkout`:

```
$ git checkout dev
Switched to branch 'dev'
$ git branch
* dev
  master
```

Let's add a second file to the `dev` branch:

```
$ echo "my second file" > mySecondFile.txt
$ git add mySecondFile.txt
$ git commit -m "added mySecondFile.txt to dev branch"
[dev a537bdc] added mySecondFile.txt to dev branch
 1 file changed, 1 insertion(+)
 create mode 100644 mySecondFile.txt
```

We can go back to the `master` branch and verify that the two lines of development are separate. Note that when we switch to the master branch, there is only one file in the directory:

```
$ git branch
* dev
  master
$ git checkout master
Switched to branch 'master'
$ ls
myFile.txt
$ git checkout dev
Switched to branch 'dev'
$ ls
myFile.txt  mySecondFile.txt
```

To have the contents in the `dev` branch be written into the `master` branch, we will need to `merge` them:

```
$ git branch
* dev
  master
$ git checkout master
Switched to branch 'master'
$ git merge dev master
Updating ff7dc1a..a537bdc
Fast-forward
 mySecondFile.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 mySecondFile.txt
$ git branch
  dev
* master
$ ls
myFile.txt  mySecondFile.txt
```

We can use `git rm` to remove a file. To see how it works, let's create a third file and remove it:

```
$ touch myThirdFile.txt
$ git add myThirdFile.txt
$ git commit -m "adding myThirdFile.txt"
```

```
[master 169a203] adding myThirdFile.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 myThirdFile.txt
$ ls
myFile.txt  mySecondFile.txt  myThirdFile.txt
$ git rm myThirdFile.txt
rm 'myThirdFile.txt'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    myThirdFile.txt
$ git commit -m "deleted myThirdFile.txt"
[master 1b24b4e] deleted myThirdFile.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 myThirdFile.txt
```

We will be able to see the last two changes in the log:

```
$ git log
commit 1b24b4e95eb0c01cc9a7124dc6ac1ea37d44d51a (HEAD -> master)
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 10:02:45 2019 -0800
    deleted myThirdFile.txt
commit 169a2034fb9844889f5130f0e42bf9c9b7c08b05
Author: Eric Chou <echou@yahoo.com>
Date:   Fri Nov 8 10:00:56 2019 -0800
    adding myThirdFile.txt
```

We have gone through most of the basic operations we would use for Git. Let's look at how to use GitHub to share our repository.

GitHub Example

In this example, we will use GitHub as the centralized location to synchronize our local repository and share it with other users.

We will create a repository on GitHub. GitHub has always been free for creating public open-source repositories. Starting in January 2019, it also offers unlimited free private repositories. In this case, we will create a private repository and add the license and `.gitignore` file:

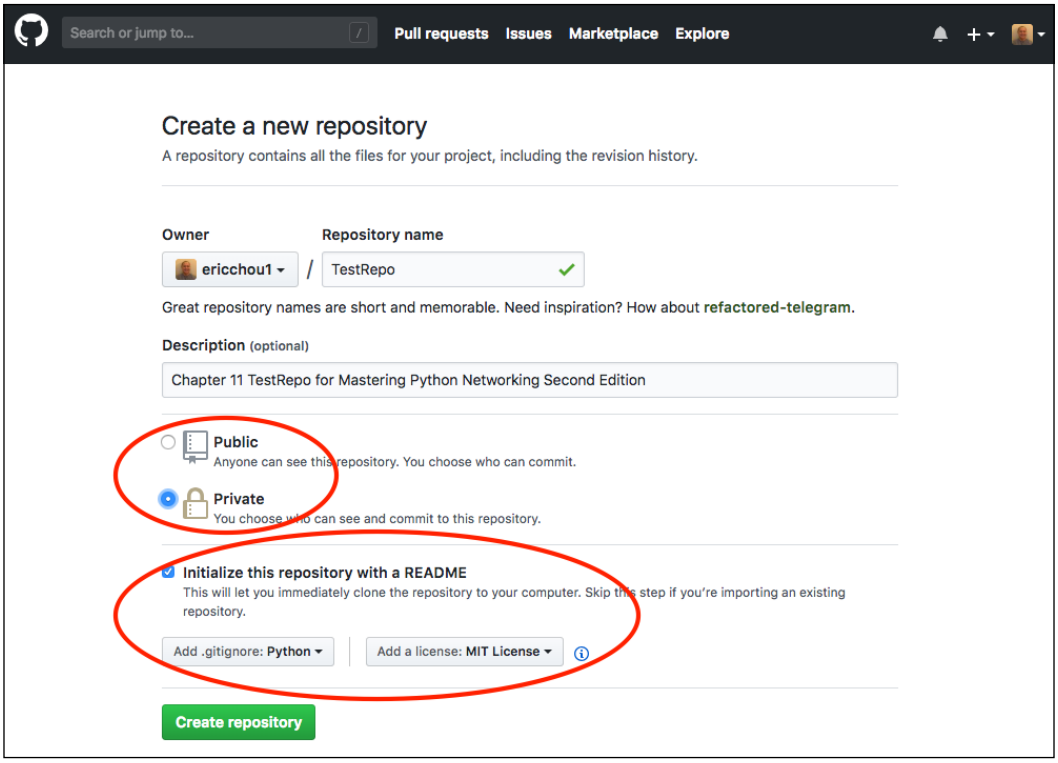


Figure 14.1: Creating a private repository in GitHub

Once the repository is created, we can find the URL for it:

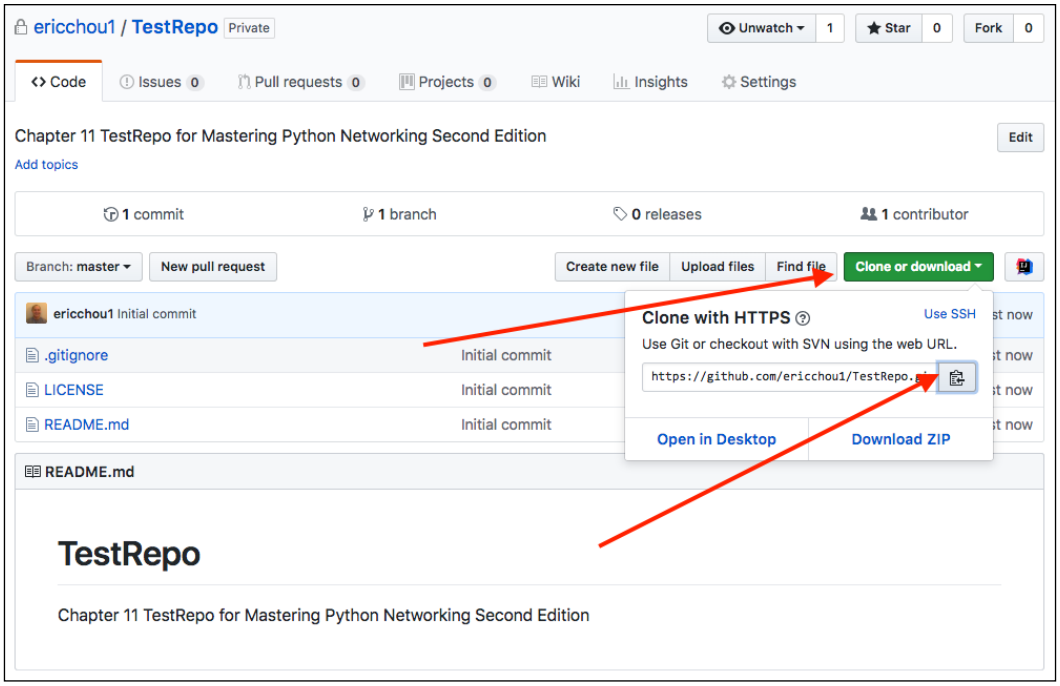


Figure 14.2: GitHub repository URL

We will use this URL to create a remote target, which we will use as a "source of truth" for our project. We will name the remote target `gitHubRepo`:

```
$ git remote add gitHubRepo https://github.com/ericchou1/TestRepo.git
$ git remote -v
gitHubRepo  https://github.com/ericchou1/TestRepo.git (fetch)
gitHubRepo  https://github.com/ericchou1/TestRepo.git (push)
```

Since we chose to create `README.md` and `LICENSE` files during creation, the remote repository and local repository are not the same.

GitHub switched to a **Personal Access Token** (**PAT**) as the term to be entered as a password a few years ago: `https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token`. To generate a token, click on the profile logo **-> Settings -> Developer settings -> Personal Access Tokens**. We need to use this token as our password when prompted in the command line.

If we were to push local changes to the new GitHub repository, we would receive the following error (remember to change the branch name to main if that is your default branch):

```
$ git push gitHubRepo master
Username for 'https://github.com': <skip>
Password for 'https://echou@yahoo.com@github.com': <remember to use your
personal access token>
To https://github.com/ericchou1/TestRepo.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ericchou1/TestRepo.
git'
```

We will go ahead and use `git pull` to get the new files from GitHub:

```
$ git pull gitHubRepo master
Username for 'https://github.com': <skip>
Password for 'https://<username>@github.com': <personal access token>
From https://github.com/ericchou1/TestRepo
* branch master -> FETCH_HEAD
Merge made by the 'recursive' strategy.
.gitignore | 104
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ LICENSE |
21 ++++++++++++
README.md | 2 ++
3 files changed, 127 insertions(+)
create mode 100644 .gitignore
create mode 100644 LICENSE
create mode 100644 README.md
```

Now we will be able to push the contents over to GitHub:

```
$ git push gitHubRepo master
Username for 'https://github.com': <username>
Password for 'https://<username>@github.com': <personal access token>
```

```
Counting objects: 15, done.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (15/15), 1.51 KiB | 0 bytes/s, done. Total 15 (delta
1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/ericchou1/TestRepo.git a001b81..0aa362a master ->
master
```

We can verify the content of the GitHub repository on the web page:

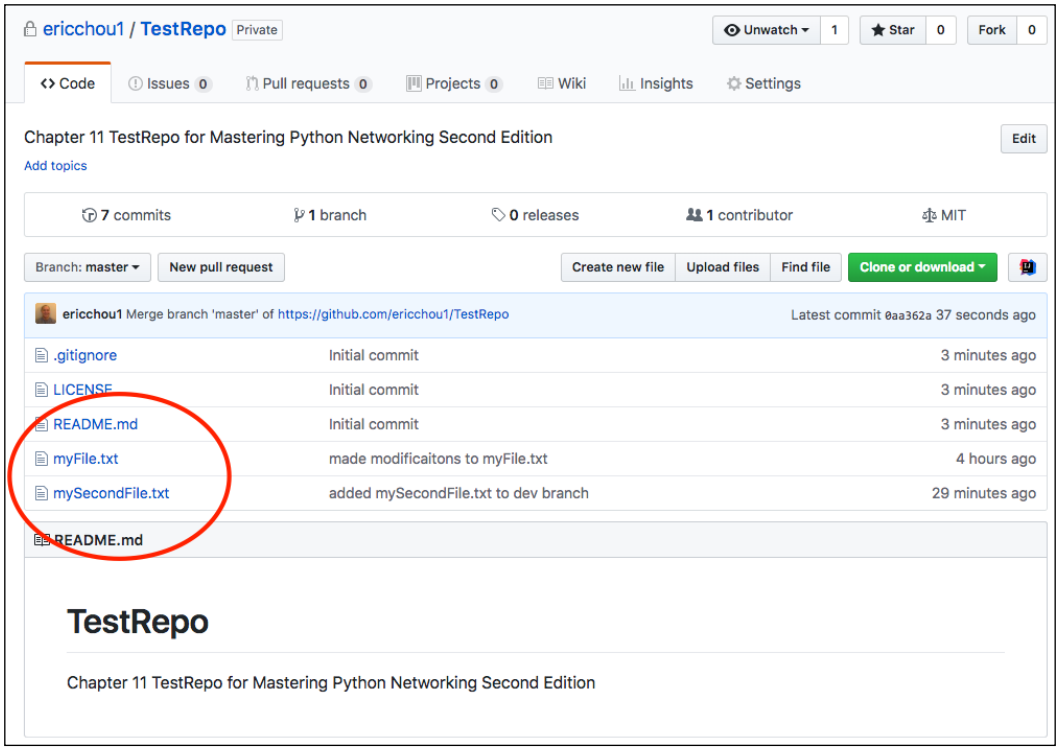


Figure 14.3: GitHub repository

Now another user can simply make a copy, or `clone`, of the repository:

```
[This is operated from another host]
$ cd /tmp
$ git clone https://github.com/ericchou1/TestRepo.git
Cloning into 'TestRepo'...
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (13/13), done.
```

```
remote: Total 20 (delta 2), reused 15 (delta 1), pack-reused 0
Unpacking objects: 100% (20/20), done.
$ cd TestRepo/
$ ls
LICENSE myFile.txt
README.md mySecondFile.txt
```

This copied repository will be the exact copy of my original repository, including all the commit history:

```
$ git log
commit 0aa362a47782e7714ca946ba852f395083116ce5 (HEAD -> master, origin/
master, origin/HEAD)
Merge: bc078a9 a001b81
Author: Eric Chou <skip>
Date: Fri Jul 20 14:18:58 2018 -0700
    Merge branch 'master' of https://github.com/ericchou1/TestRepo
commit a001b816bb75c63237cbc93067dffcc573c05aa2
Author: Eric Chou <skip>
Date: Fri Jul 20 14:16:30 2018 -0700
    Initial commit
...
```

I can also invite another person as a collaborator for the project under the repository settings:

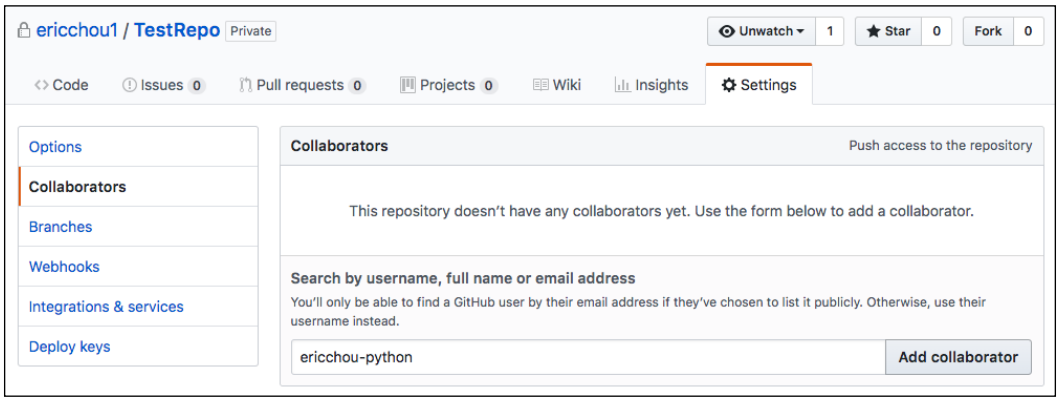


Figure 14.4: Repository invite

In the next example, we will see how we can fork a repository and perform a pull request for a repository we do not maintain.

Collaborating with Pull Requests

As mentioned, Git supports collaboration between developers for a single project. We will look at how it is done when the code is hosted on GitHub.

In this case, we will use the GitHub repository for the second edition of this book from Packt's GitHub public repository. I will use a different GitHub handle, so I appear as a non-administrative user. I will click on the **Fork** button to make a copy of the repository in my account:

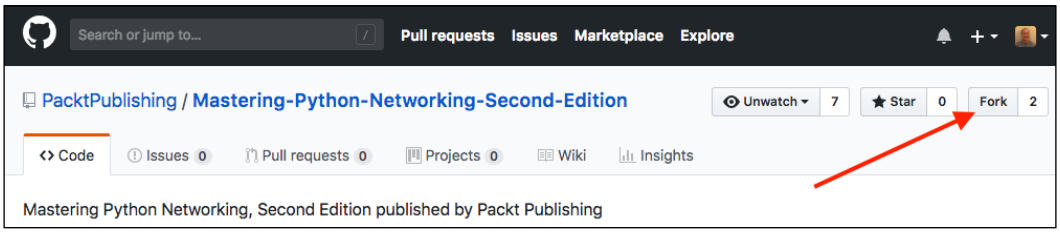


Figure 14.5: Git Fork button

It will take a few seconds to make a copy:

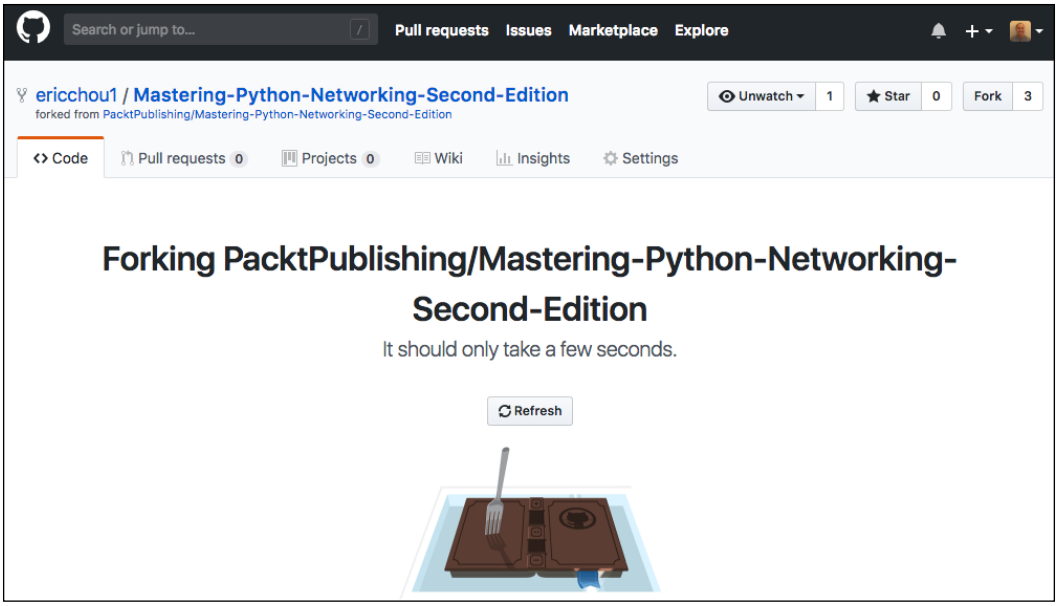


Figure 14.6: Git Fork in progress

After it is forked, we will have a copy of the repository in our account:

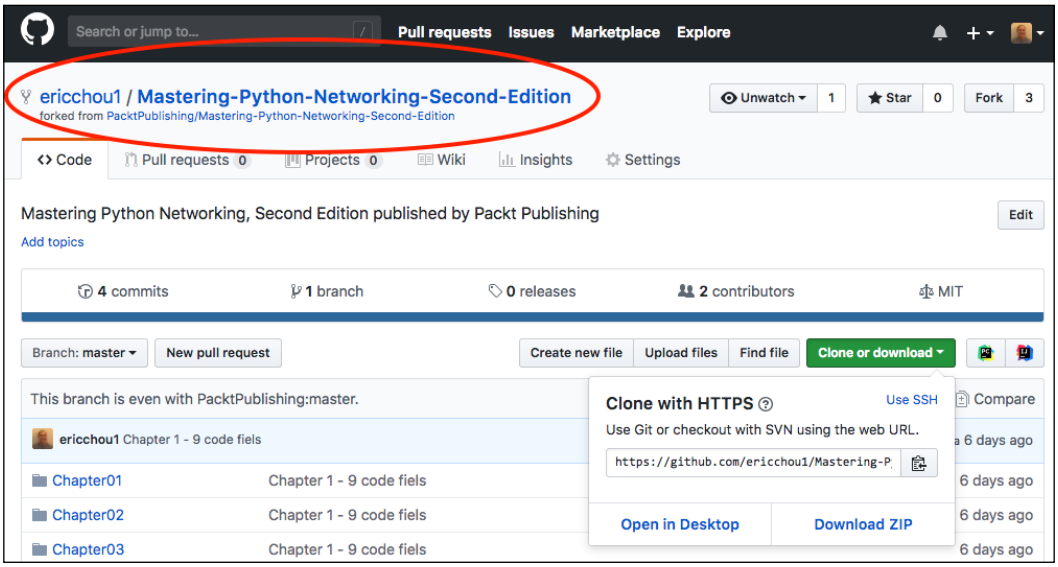


Figure 14.7: Git Fork

We can follow the same steps we used to modify the files. In this case, I will make some changes to the `README.md` file. After the change is made, I can click on the **New pull request** button to create a pull request:

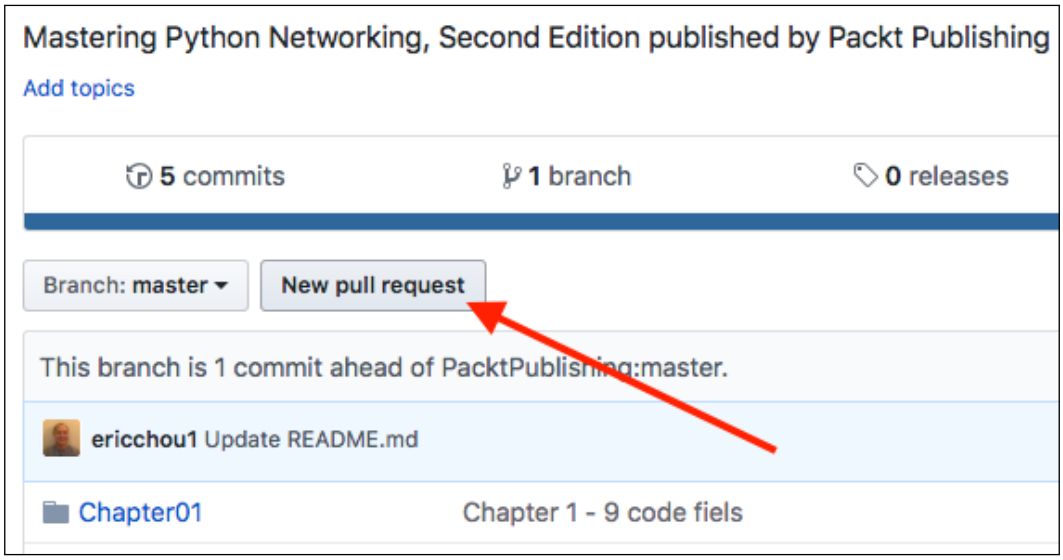


Figure 14.8: Pull request

When making a pull request, we should fill in as much information as possible to provide justifications for making the change:

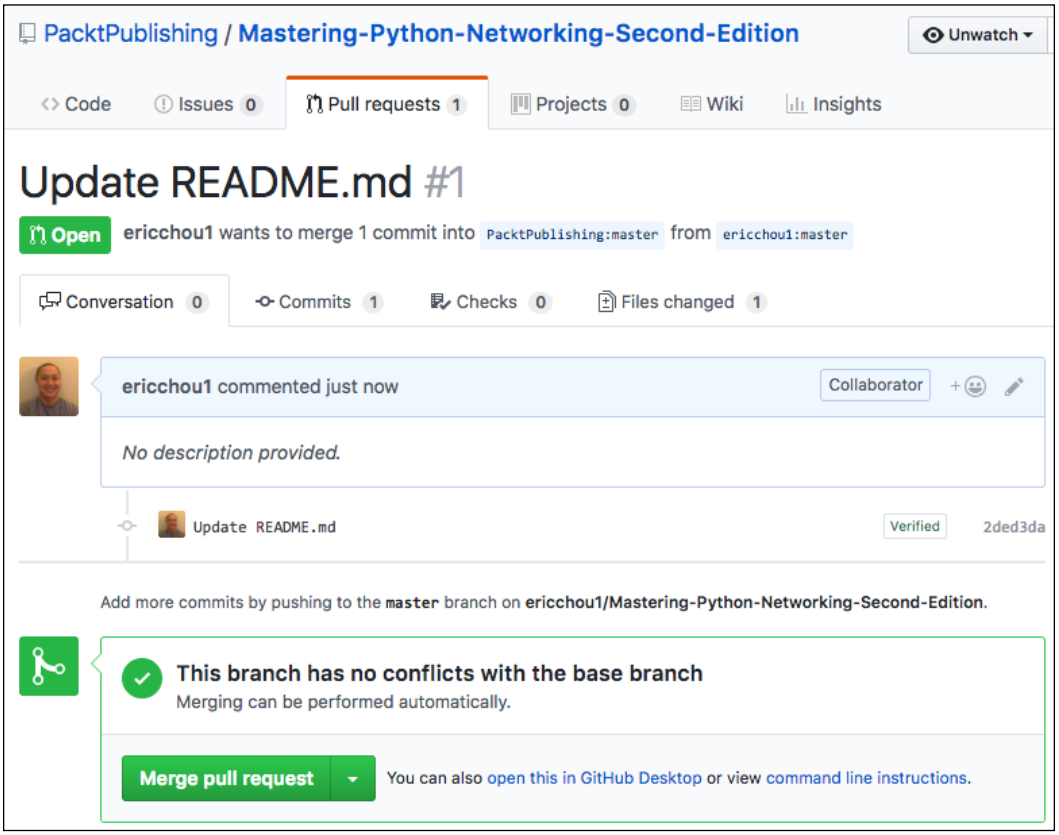


Figure 14.9: Pull request details

The repository maintainer will receive a notification of the pull request; if accepted, the change will make its way to the original repository:

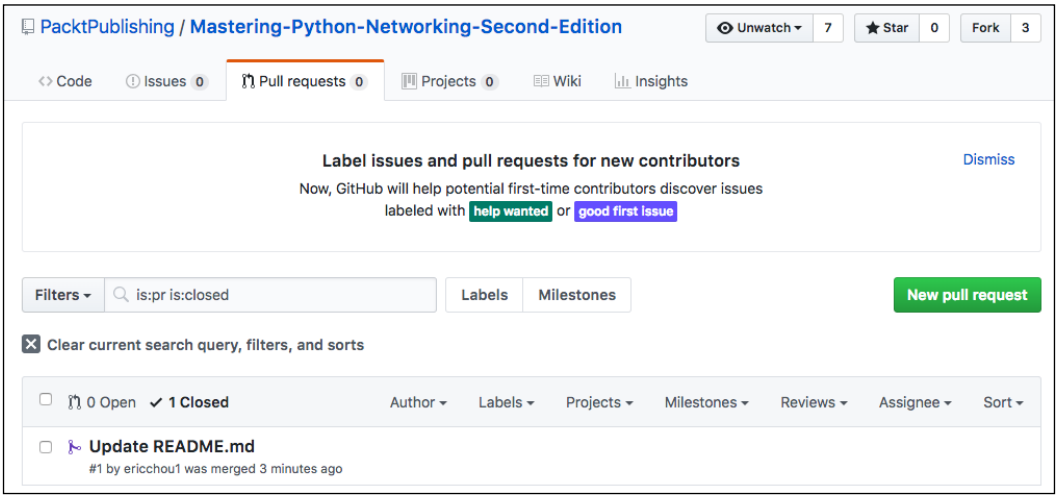


Figure 14.10: Pull request record

GitHub provides an excellent platform for collaboration with other developers; this is quickly becoming the de facto development choice for many large, open-source projects. Since Git and GitHub are used extensively in many projects, a natural next step would be to automate the processes we have seen in this section. In the following section, let's look at how we can use Git with Python.

Git with Python

There are some Python packages that we can use with Git and GitHub. In this section, we will look at the GitPython and PyGitHub libraries.

GitPython

We can use the GitPython package, `https://gitpython.readthedocs.io/en/stable/index.html`, to work with our Git repository. We will install the package and use the Python shell to construct a `Repo` object. From there, we can list all the commits in the repository:

```
$ pip install gitpython
$ python
>>> from git import Repo
>>> repo = Repo('/home/echou/Mastering_Python_Networking_third_edition/
Chapter13/TestRepo-1')
```

```
>>> for commits in list(repo.iter_commits('master')):
... print(commits)
...
1b24b4e95eb0c01cc9a7124dc6ac1ea37d44d51a
169a2034fb9844889f5130f0e42bf9c9b7c08b05
a537bdcc1648458ce88120ae607b4ddea7fa9637
ff7dc1a40e5603fed552a3403be97addefddc4e9
5d7c1c8543c8342b689c66f1ac1fa888090ffa34
```

We can also look at the index entries in the `repo` object:

```
>>> for (path, stage), entry in repo.index.entries.items():
... print(path, stage, entry)
...
myFile.txt 0 100644 69e7d4728965c885180315c0d4c206637b3f6bad 0 myFile.txt
mySecondFile.txt 0 100644 75d6370ae31008f683cf18ed086098d05bf0e4dc 0
mySecondFile.txt
```

GitPython offers good integration with all the Git functions. However, it might not be the easiest library to work with for beginners. We need to understand the terms and structure of Git to take full advantage of GitPython, and it is always good to keep it in mind if we need it for other projects.

PyGitHub

Let's look at using the PyGithub library, `http://pygithub.readthedocs.io/en/latest/`, to interact with GitHub reappearing around GitHub API v3, `https://developer.github.com/v3/`:

```
$ pip install PyGithub
```

Let's use the Python shell to print out the user's current repository:

```
$ python
>>> from github import Github
>>> g = Github("<username>", "<password>")
>>> for repo in g.get_user().get_repos():
... print(repo.name)
...
Mastering-Python-Networking-Second-Edition
Mastering-Python-Networking-Third-Edition
```

For more programmatic access, we can also create more granular control using an access token. GitHub allows a token to be associated with the selected rights:

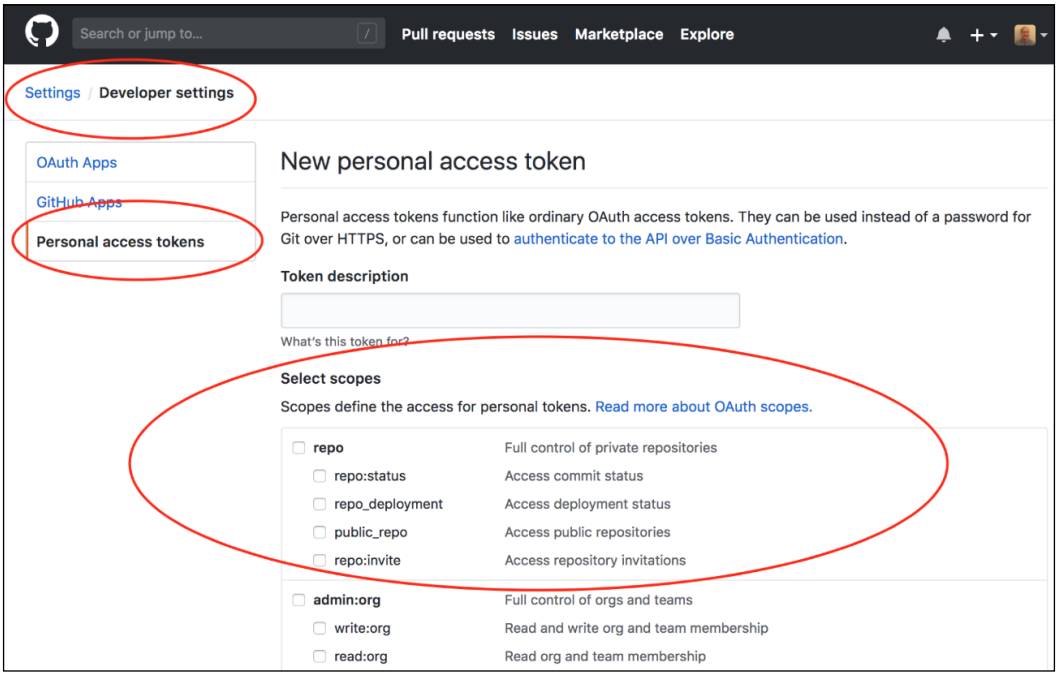


Figure 14.11: GitHub token generation

The output is a bit different if you use the access token as the authentication mechanism:

```
>>> from github import Github
>>> g = Github("<token>")
>>> for repo in g.get_user().get_repos():
... print(repo)
...
Repository(full_name="oreillymedia/distributed_denial_of_service_ddos")
Repository(full_name="PacktPublishing/-Hands-on-Network-    Programming-
with- Python")
Repository(full_name="PacktPublishing/Mastering-Python-Networking")
Repository(full_name="PacktPublishing/Mastering-Python-Networking-Second-
Edition")
...
```

Now that we are familiar with Git, GitHub, and some Python packages, we can use them to work with the technology. We will look at some practical examples in the upcoming section.

Automating Configuration Backup

In this example, we will use PyGithub to back up a directory containing our router configurations. We have seen how we can retrieve the information from our devices with Python or Ansible; we can now check them into GitHub.

We have a subdirectory, named `config`, with our router configs in text format:

```
$ ls configs/
iosv-1 iosv-2
$ cat configs/iosv-1
Building configuration...
Current configuration : 4573 bytes
!
! Last configuration change at 02:50:05 UTC Sat Jun 2 2018 by cisco
!
version 15.6
service timestamps debug datetime msec
...
```

We can use the following script, `Chapter14_1.py`, to retrieve the latest index from our GitHub repository, build the content that we need to commit, and automatically commit the configuration:

```
#!/usr/bin/env python3
# reference: https://stackoverflow.com/questions/38594717/how-do-i-push-
new-files-to-github
from github import Github, InputGitTreeElement
import os
github_token = '<token>'
configs_dir = 'configs'
github_repo = 'TestRepo'
# Retrieve the list of files in configs directory
file_list = []
for dirpath, dirname, filenames in os.walk(configs_dir):
    for f in filenames:
        file_list.append(configs_dir + "/" + f)
g = Github(github_token)
```

```
repo = g.get_user().get_repo(github_repo)
commit_message = 'add configs'
master_ref = repo.get_git_ref('heads/master')
master_sha = master_ref.object.sha
base_tree = repo.get_git_tree(master_sha)
element_list = list()
for entry in file_list:
    with open(entry, 'r') as input_file:
        data = input_file.read()
    element = InputGitTreeElement(entry, '100644', 'blob', data)
    element_list.append(element)
# Create tree and commit
tree = repo.create_git_tree(element_list, base_tree)
parent = repo.get_git_commit(master_sha)
commit = repo.create_git_commit(commit_message, tree, [parent])
master_ref.edit(commit.sha)
```

We can see the `configs` directory in the GitHub repository:

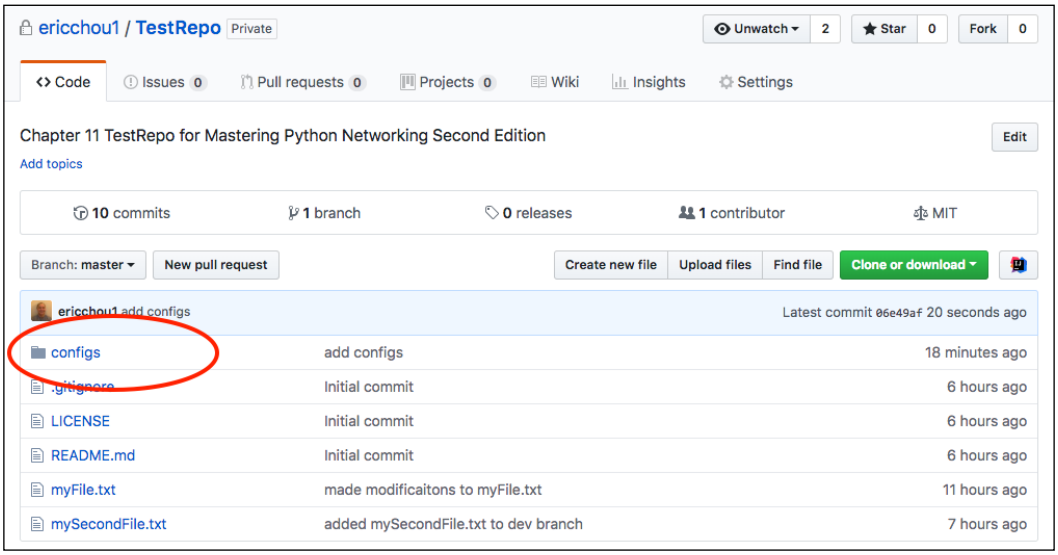


Figure 14.12: Configs directory

The commit history shows the commit from our script:

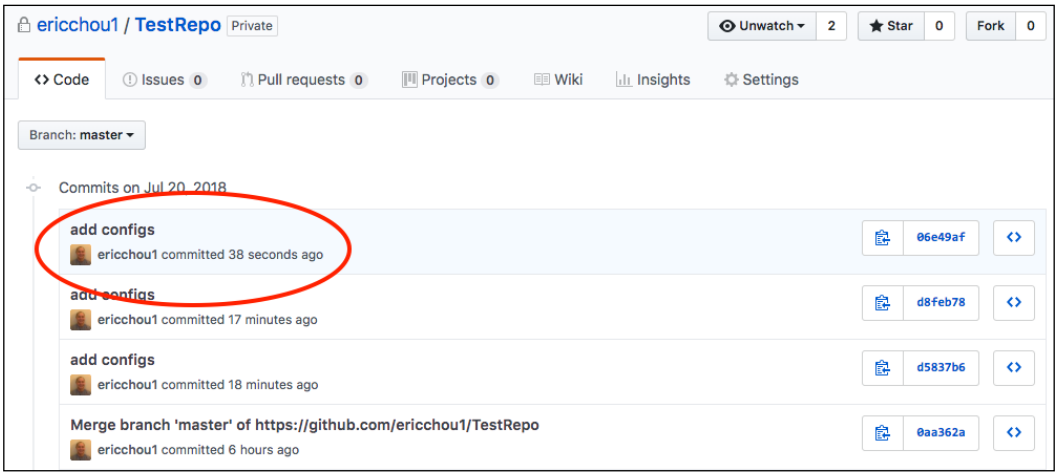


Figure 14.13: Commit history

In the GitHub example section, we saw how we could collaborate with other developers by forking the repository and making pull requests. Let's look at how we can further collaborate with Git.

Collaborating with Git

Git is an awesome collaboration technology, and GitHub is an incredibly effective way to develop projects together. GitHub provides a place for anyone in the world with internet access to share their thoughts and code for free. We know how to use Git and some basic collaboration steps using GitHub, but how do we join and contribute to a project?

Sure, we would like to give back to these open-source projects that have given us so much, but how do we get started?

In this section, we'll look at some of the things to know about software development collaboration using Git and GitHub:

- **Start small**: One of the most important things to understand is the role we can play within a team. We might be awesome at network engineering, but mediocre at Python development. There are plenty of things we can do that don't involve being a highly skilled developer. Don't be afraid to start small; documentation and testing are two good ways to get your foot in the door as a contributor.

- **Learn the ecosystem**: With any project, large or small, there is a set of conventions and a culture that has been established. We are all drawn to Python for its easy-to-read syntax and beginner-friendly culture; it also has a development guide centered around that ideology (`https://devguide.python.org/`). The Ansible project, on the other hand, also has an extensive community guide (`https://docs.ansible.com/ansible/latest/community/index.html`). It includes the code of conduct, the pull request process, how to report bugs, and the release process. Read these guides and learn the ecosystem for the project of interest.
- **Make a branch**: I made the mistake of forking a project and making a pull request for the main branch. The main branch should be left alone for the core contributors to make changes. We should create a separate branch for our contribution and allow the branch to be merged later.
- **Keep the forked repository synchronized**: Once you have forked a project, no rule forces the cloned repository to sync with the main repository. We should make a point to regularly do `git pull` (get the code and merge locally) or `git fetch` (get the code with any change locally) to ensure we have the latest copy of the main repository.
- **Be friendly**: Just as in the real world, the virtual world has no place for hostility. When discussing an issue, be civil and friendly, even in disagreements.

Git and GitHub provide a way for any motivated individual to make a difference by making it easy to collaborate on projects. We are all empowered to contribute to any open source or private projects that interest us.

Summary

In this chapter, we looked at the version-control system known as Git and its close sibling, GitHub. Git was developed by Linus Torvolds in 2005 to help develop the Linux kernel and was later adopted by other open-source projects as their source-control system. Git is a fast, distributed, and scalable system. GitHub provides a centralized location to host Git repositories on the internet that allows anybody with an internet connection to collaborate.

We looked at how to use Git in the command line and its various operations and how they are applied in GitHub. We also studied two popular Python libraries for working with Git: GitPython and PyGithub. We ended this chapter with a configuration backup example and notes about project collaboration.

In *Chapter 15*, *Continuous Integration with GitLab*, we will look at another popular open-source tool used for continuous integration and deployment: GitLab.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



15

Continuous Integration with GitLab

The network touches every part of the technology stack; in all the environments I have worked in, the network is always a Tier-Zero service. It is a foundation service that other services rely on for their services to work. In the minds of other engineers, business managers, operators, and support staff, the network should just work. It should always be accessible and function correctly—a good network is a network that nobody hears about.

Of course, as network engineers, we know the network is as complex as any other technology stack. Due to its complexity, the constructs that make up a running network can be fragile. Sometimes, I look at a network and wonder how it can work at all, let alone how it's been running for months and years without any business impact.

Part of the reason we are interested in network automation is to find ways to repeat our network-change process reliably and consistently. By using Python scripts or the Ansible framework, we can make sure the changes that we make will stay consistent and be reliably applied. As we saw in the last chapter, we can use Git and GitHub to store components of the process, such as templates, scripts, requirements, and files, reliably. The code that makes up the infrastructure is version-controlled, collaborated, and accountable for changes. But how do we tie all the pieces together? In this chapter, we will look at a popular repository that can optimize the network-management pipeline called GitLab.



GitLab's open core is published under an MIT open-source license. The rest is source-available, `https://about.gitlab.com/solutions/open-source/`.

In this chapter, we'll cover the following topics:

- Challenges with the traditional change management process
- An introduction to continuous integration and GitLab
- GitLab installation and examples
- GitLab with Python
- Continuous integration for network engineering

We'll begin by looking at the traditional change management process. As any battle-tested network engineer could tell you, the traditional change management process typically involves much manual labor and human judgment. As we will see, it is not consistent and is difficult to streamline.

The traditional change management process

Engineers who have worked in a large network environment know that the impact of a network change gone wrong can be big. We can make hundreds of changes without any issues, but all it takes is one bad change that can cause the network to harm the whole business.



There is no shortage of war stories about network outages causing business pain. One of the most visible and large-scale AWS EC2 outages in 2011 was caused by a network change that was part of the normal AWS scaling activities in the AWS US-East region. The change occurred at 00:47 PDT and caused a brown-out for various services for over 12 hours, losing millions of dollars for Amazon. More importantly, the reputation of the relatively young service took a serious hit. IT decision-makers pointed to the outage as a reason NOT to migrate to the young AWS cloud. It took many years to rebuild its reputation. You can read more about the incident report at `https://aws.amazon.com/message/65648/`.

Due to the potential impact and complexity, in many environments, the **change-advisory board (CAB)** process is implemented for networks. The typical CAB process is as follows:

1. The network engineer will design the change and write out the detailed steps required for the change. These can include the reason for the change, the devices involved, the commands that will be applied or deleted, how to verify the output, and the expected outcome for each step.

2. The network engineer is typically required to ask for a technical review from a peer first. Depending on the nature of the change, there can be different levels of peer review. Simple changes can require a single-peer technical review; more complex changes might require a senior designated engineer for approval.
3. The CAB meeting is generally scheduled for set times with emergency ad hoc meetings available.
4. The engineer will present the change to the board. The board will ask the necessary questions, assess the impact, and either approve or deny the change request.
5. The change will be carried out, either by the original engineer or another engineer, during the scheduled change window.

This process sounds reasonable and inclusive but proves to have a few challenges in practice:

- **Write-ups are time-consuming**: It typically takes a long time for the design engineer to write up the document, and sometimes the writing process takes longer than the time to apply the change. This is generally due to the fact that all network changes are potentially impactful and we need to document the process for both technical and non-technical CAB members.
- **Engineer expertise**: High-level engineer expertise is a limited resource. There are different levels of engineering expertise; some are more experienced, and they are typically the most sought-after resources. We should reserve their time for tackling the most complex network issues, not reviewing basic network changes.
- **Meetings are time-consuming**: It takes a lot of effort to put together meetings and have each member show up. What happens if a required approval person is on vacation or sick? What if you need the network change to be made prior to the scheduled CAB time?

These are just some of the bigger challenges of the human-based CAB process. Personally, I hate the CAB process with a passion. I do not dispute the need for peer review and prioritization; however, I think we need to minimize the potential overhead involved. For the remainder of this chapter, let's look at a potentially suitable replacement pipeline for CAB, and change management in general, that has been adopted in the software engineering world.

Introduction to continuous integration

Continuous Integration (**CI**) in software development is a way to publish small changes to the code base quickly, with built-in code tests and validation. The key is to classify the changes to be CI-compatible, that is, not overly complex and small enough to be applied so that they can be backed out of easily. The tests and validation process are built in an automated way to gain a baseline of confidence that changes will be applied without breaking the whole system.

Before CI, changes to software were often made in large batches and often required a long validation process (does that sound familiar?). It could be months before developers saw their changes in production, received feedback loops, and corrected bugs. In short, the CI process aims to shorten the process from idea to change.

The general workflow typically involves the following steps:

1. The first engineer takes a current copy of the code base and works on the change.
2. The first engineer submits the change to the repository.
3. The repository can notify the necessary parties of a change in the repository to a group of engineers who can review the change. They can either approve or reject the change.
4. The CI system can continuously pull the repository for changes, or the repository can send a notification to the CI system when changes happen. Either way, the CI system will pull the latest version of the code.
5. The CI system will run automated tests to try to catch any breakage.
6. If there are no faults found, the CI system can choose to merge the change into the main code and optionally deploy it to the production system.

This is a generalized list of steps. The process can be different for each organization. For example, automated tests can be run as soon as the delta code is checked instead of after code review. Sometimes, the organization might choose to have a human engineer involved for sanity checks in between the steps.

In the next section, we will illustrate the instructions to install GitLab on an Ubuntu 22.04 LTS system.

Installing GitLab

GitLab is a powerful, all-in-one tool to handle the end-to-end DevOps collaboration tool. As we will see in a minute, it hosts the code repository and handles the code testing, deployment, and verification. It is one of the most popular DevOps tools used in the field today.



The company behind the technology, GitLab Inc., had its successful initial public offering on NASDAQ (ticket GTLB) in late 2021, `https://techcrunch.com/2021/09/17/inside-gitlabs-ipo-filing/`. The company's success shows the strength of and sustainability of the technology.

We will only need a small set of its features to get up and running with a test lab. The objective is to familiarize ourselves with the overall flow of the steps. I encourage you to look at the GitLab documentation at `https://docs.gitlab.com/` to get a sense of its features.

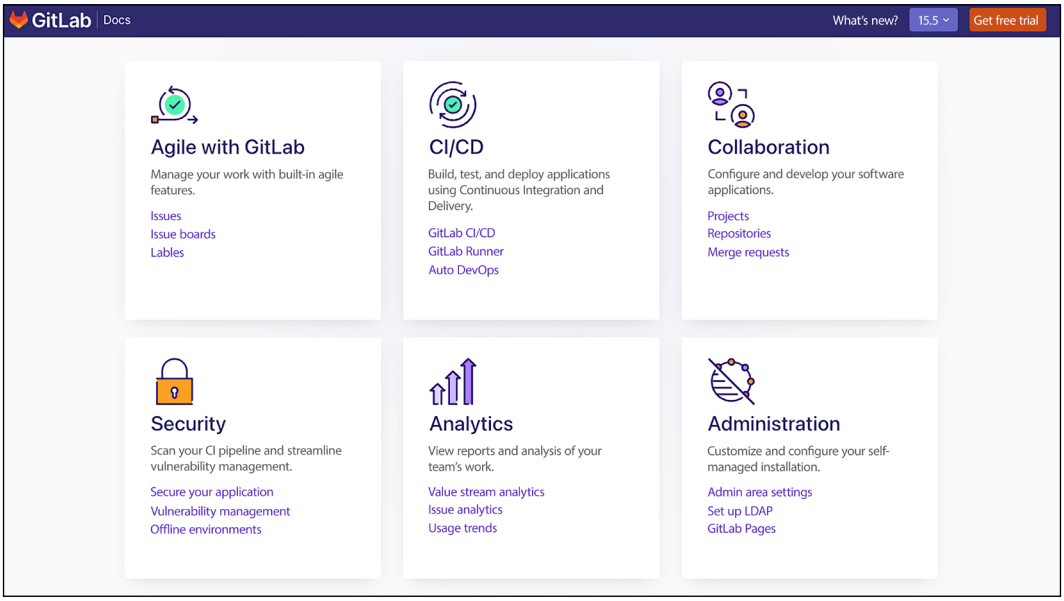


Figure 15.1: GitLab Documentation

For our network lab, we will use the same lab topology that we have been using for the last few chapters.

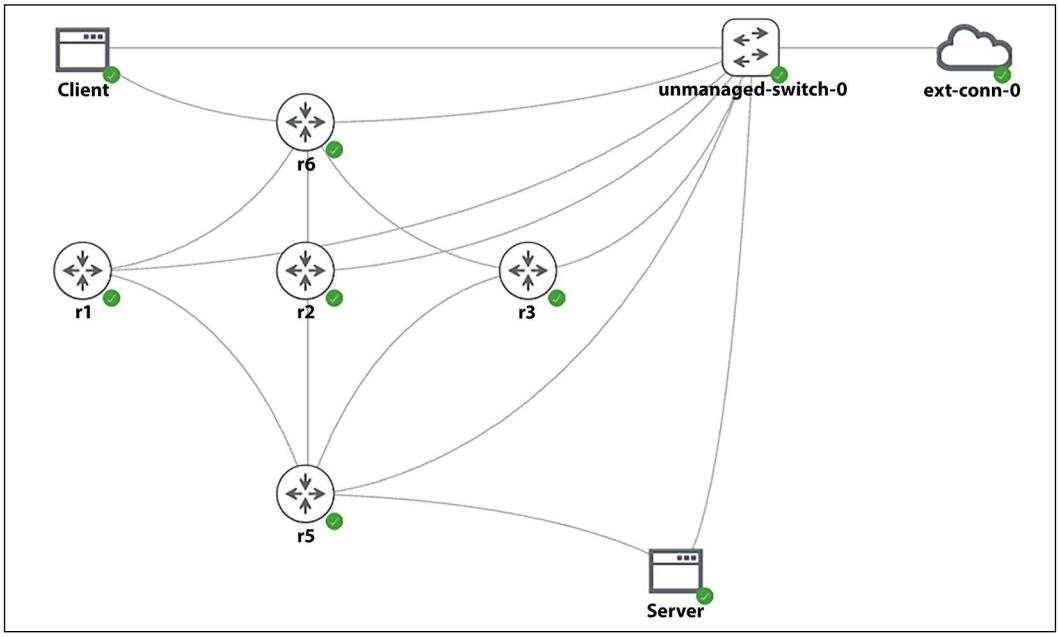


Figure 15.2: Lab Topology

While it is tempting to run GitLab as a Docker image, the GitLab runners (components executing the steps) are Docker images themselves, and running Docker-in-Docker introduces more complexity in our lab. Therefore, in this chapter, we will install GitLab on a VM with the runners running in containers. The installation system requirements can be found here, `https://docs.gitlab.com/ee/install/requirements.html`.

We will install Docker Engine, docker-compose, then the GitLab packages. Let's get Docker ready first:

```
# Installing Docker Engine
$ sudo apt-get install ca-certificates curl gnupg lsb-release
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/
keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/
ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/
docker.list > /dev/null
$ sudo apt-get update
```

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
# Run Docker as user
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
# Install Docker-Compose
$ sudo curl -L "https://github.com/docker/compose/releases/
download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/
docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose --version
docker-compose version 1.29.2, build 5becea4c
```

For GitLab, we will install the self-managed GitLab with the official steps: `https://docs.gitlab.com/omnibus/index.html#installation-and-configuration-using-omnibus-package`. Note that the steps require port forwarding to the host on an externally accessible URL:

```
$ sudo apt update
$ sudo apt-get install -y curl openssh-server ca-certificates tzdata perl
$ sudo apt-get install -y postfix
$ curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/
script.deb.sh | sudo bash
$ sudo EXTERNAL_URL="http://gitlab.networkautomationnerds.com:9090" apt-
get install gitlab-ee
```

We should see the success message once installed:

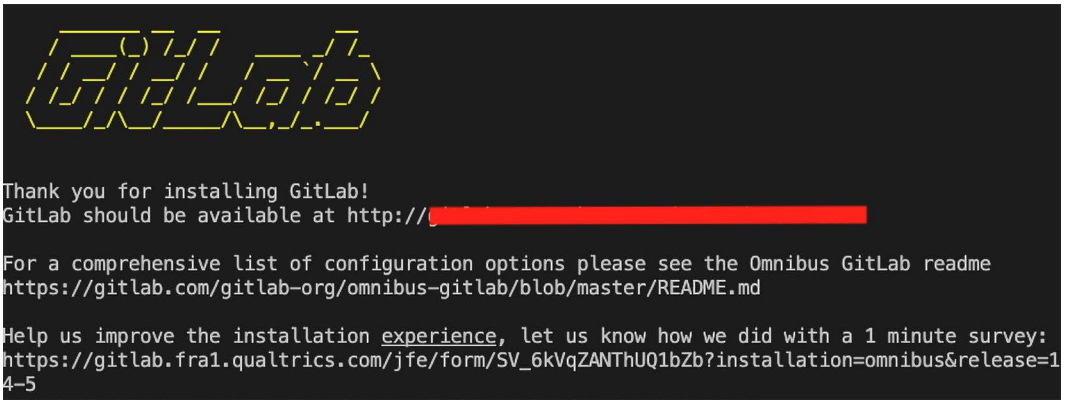


Figure 15.3: GitLab Installation

We will use the initial password to log in and then reset it (`https://docs.gitlab.com/ee/security/reset_user_password.html#reset-your-root-password`):

```
$ sudo cat /etc/gitlab/initial_root_password
…
Password: <random password>
$ sudo gitlab-rake "gitlab:password:reset"
```

We should be able to see the dashboard under '**Menu -> Admin**' once everything is configured:

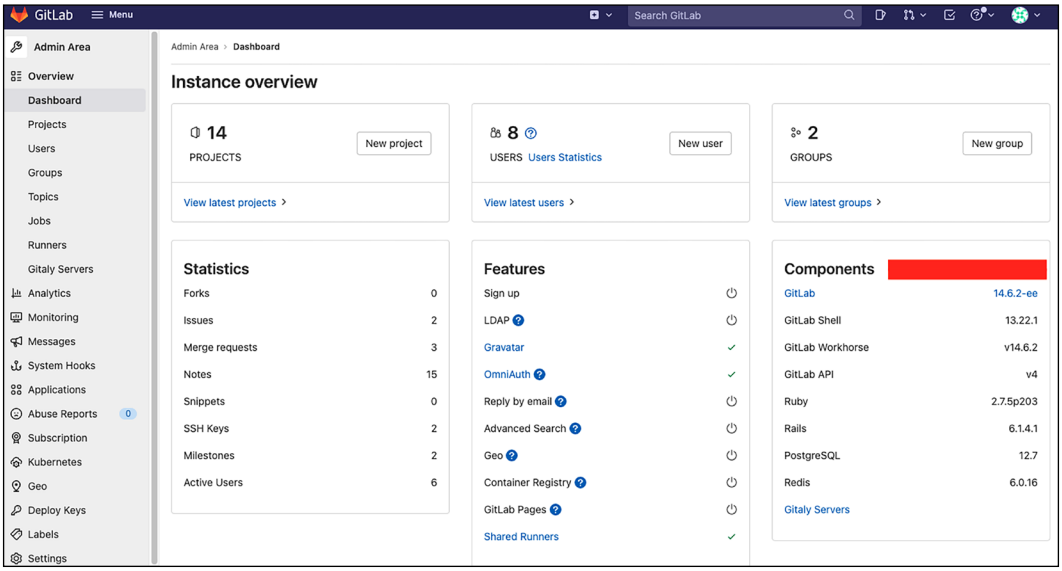


Figure 15.4: GitLab Dashboard

As an optional step, we can enable SMTP settings under `/etc/gitlab/gitlab.rb`. This would allow us to receive emails for the important messages on GitLab (`https://docs.gitlab.com/omnibus/settings/smtp.html`):

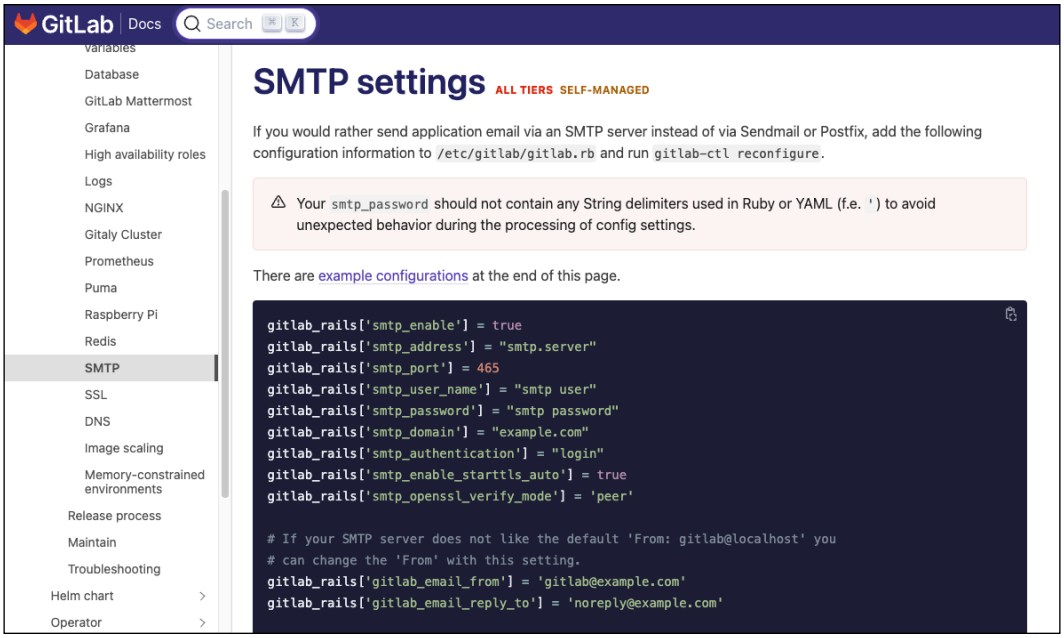


Figure 15.5: GitLab SMTP Settings

Let's talk about GitLab runners.

GitLab runners

GitLab uses the concept of a runner. A runner is a process that picks up and executes **Continuous Integration/Continuous Deployment (CI/CD)** jobs for GitLab. The runner can be run in a Docker container on the host itself, `https://docs.gitlab.com/runner/install/docker.html`:

```
$ docker run --rm -t -i gitlab/gitlab-runner –help
Unable to find image 'gitlab/gitlab-runner:latest' locally
latest: Pulling from gitlab/gitlab-runner
7b1a6ab2e44d: Pull complete
5580ef77ebbe: Pull complete
d7b21acbe607: Pull complete
Digest:
sha256:d2db6b687e9cf5baf96009e43cc3eaebf180f634306cdc74e2400315d35f0dab
Status: Downloaded newer image for gitlab/gitlab-runner:latest
```

```
…

$    docker run -d --name gitlab-runner --restart always \
>      -v /srv/gitlab-runner/config:/etc/gitlab-runner \
>      -v /var/run/docker.sock:/var/run/docker.sock \
>      gitlab/gitlab-runner:latest
617b94e5e4c5c72d33610b2eef5eb7027f579f4e069558cbf61f884375812306
```

We can go ahead and register the runner with the host, `https://docs.gitlab.com/runner/register/index.html#docker`, under **Admin Area -> Runners -> Register** an instance runner. We will take note of the token:

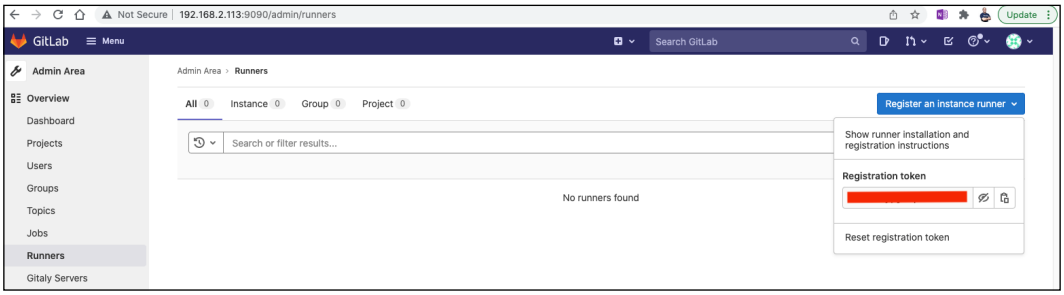


Figure 15.6: GitLab Runner Registration

We can then use the token to pull and register a runner with a base image:

```
(venv) echou@gitlab:~$ docker run --rm -it -v /srv/gitlab-runner/config:/
etc/gitlab-runner gitlab/gitlab-runner register
Runtime platform                                    arch=amd64 os=linux
pid=8 revision=5316d4ac version=14.6.0
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
http://<ip>:<port>
Enter the registration token:
<token>
Enter a description for the runner:
[fef6fb5a91dd]: local-runner
```

```
Enter tags for the runner (comma-separated): << Leave empty unless we want
matching tag to run the runners jobs

Registering runner... succeeded                     runner=64eCJ5yp
Enter an executor: virtualbox, docker-ssh+machine, kubernetes, custom,
docker-ssh, parallels, docker+machine, docker, shell, ssh:
docker
Enter the default Docker image (for example, ruby:2.6):
docker pull ubuntu:latest
Runner registered successfully. Feel free to start it, but if it's running
already the config should be automatically reloaded!
```

We are now ready to handle our first job!

First GitLab example

We can begin by creating a separate user under **Menu -> Admin Area -> Users** (under Overview) and login via that user:

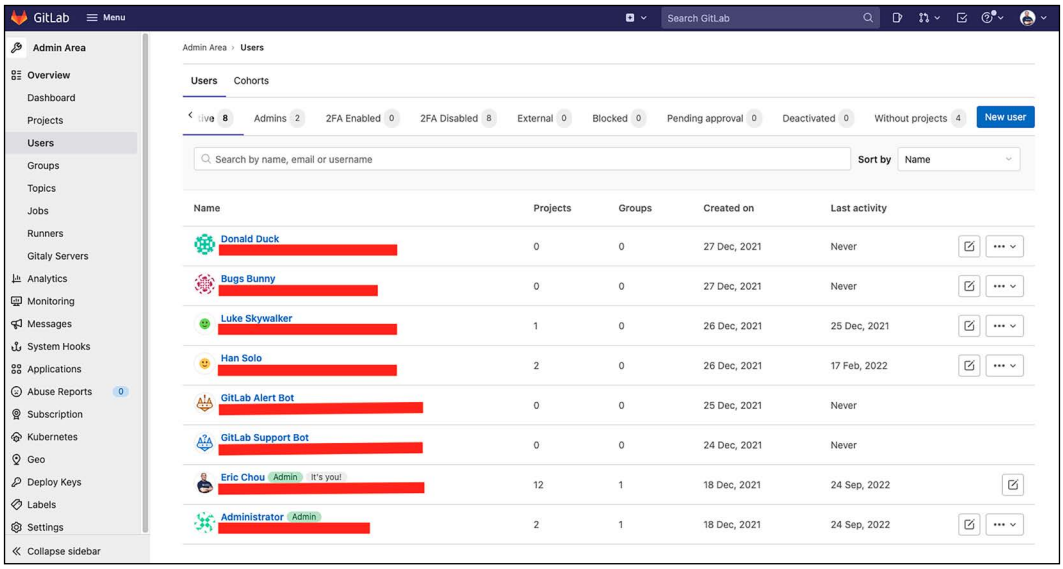


Figure 15.7: GitLab Users

To push or pull from repositories, we will also add our SSH key. This can be done via the settings section within the user profile:

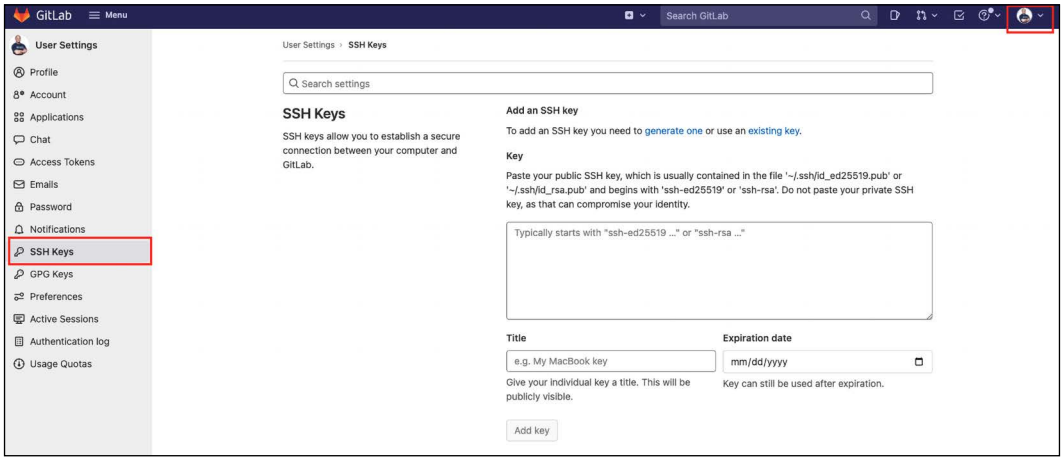


Figure 15.8: User SSH Key

We can now create a new project under **Menu -> Projects -> Create New Project**:

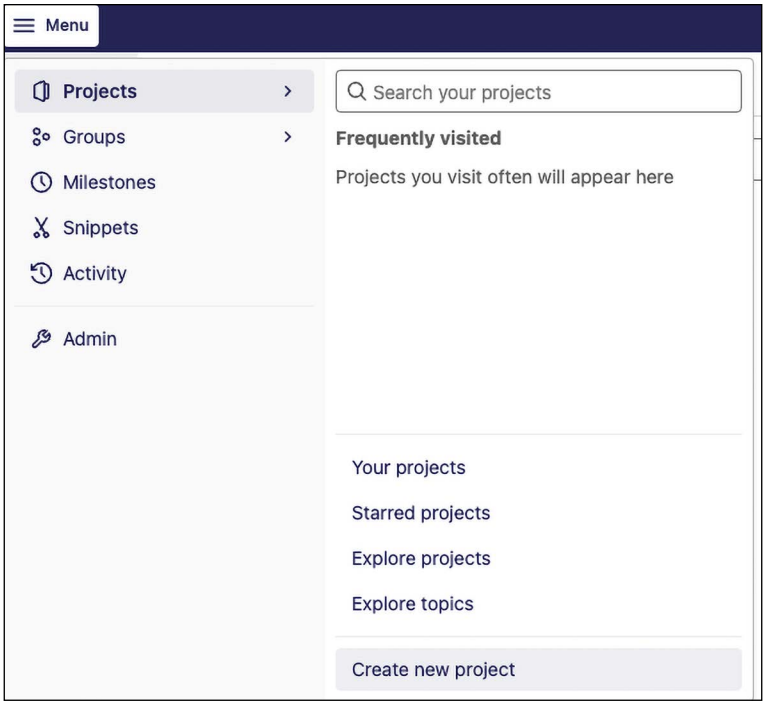


Figure 15.9: Create New Project

We will name this project `chapter15_example1`:

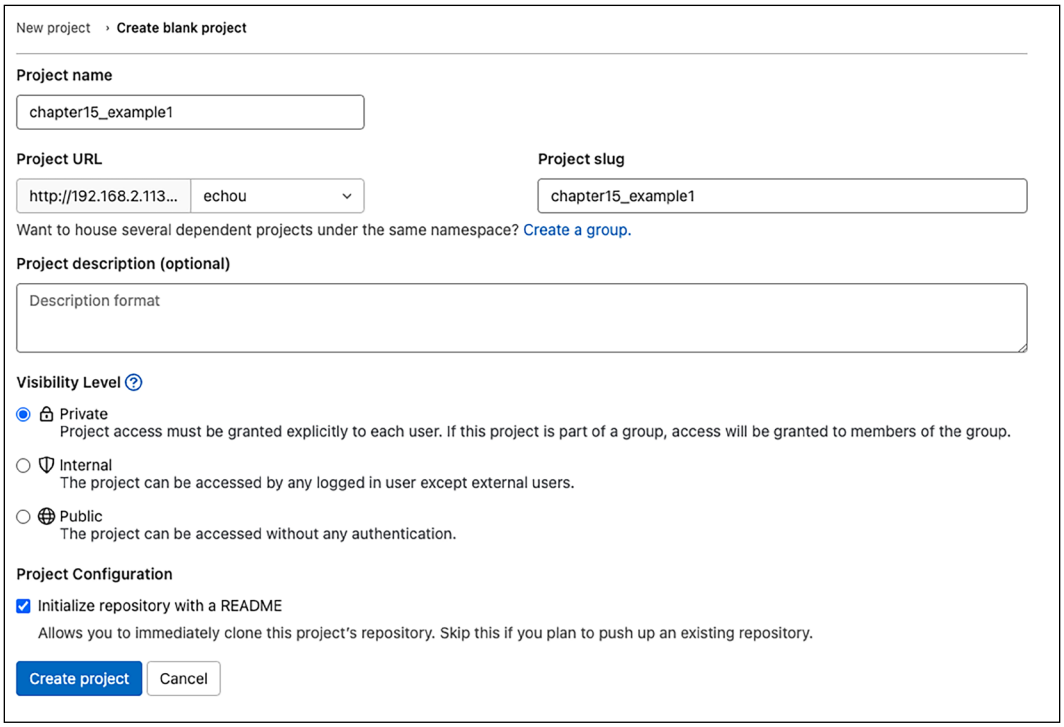


Figure 15.10: New Project Settings

We can leave the rest of the settings as we see fit. As a precaution, I typically leave the project visibility on private, but we can always change that later.

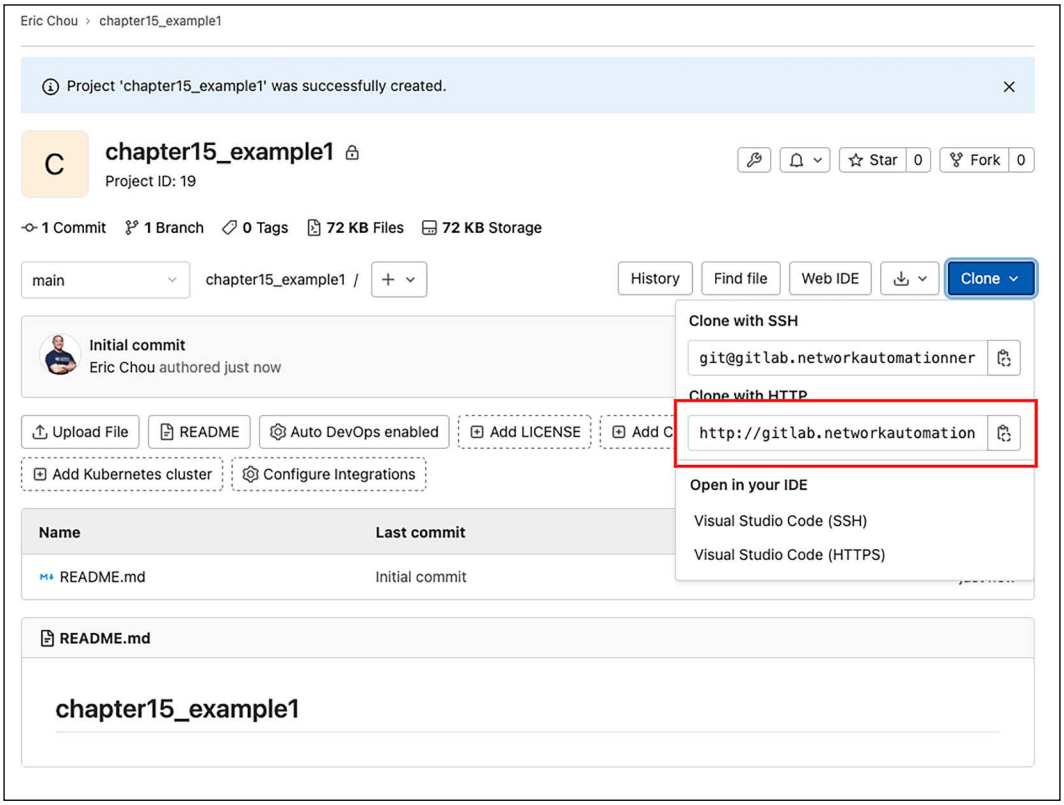


Figure 15.11: Project Clone URL

We can grab the URL for the project and clone that project on our management station:

```
$ git clone http://gitlab.<url>/echou/chapter15_example1.git
Cloning into 'chapter15_example1'...
Username for 'http://gitlab.<url>': <user>
Password for 'http://<user>@<url>':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
$ cd chapter15_example1/
$ ls
README.md
```

We will create a special file `.gitlab-ci.yml` that is recognized by GitLab as a CI/CD instruction:

```
# define stages
stages:
    - build
    - test
    - deploy

# define the job
deploy our network:
    image: "ubuntu:20.04"
    stage: build
    script:
        - mkdir new_network
        - cd new_network
        - touch bom.txt
        - echo "this is our build" >> bom.txt
    artifacts:
        paths:
          - new_network/

test our network:
    stage: test
    image: "ubuntu:20.04"
    script:
        - pwd
        - ls
        - test -f new_network/bom.txt

deploy to prod:
    stage: deploy
    image: "ubuntu:20.04"
    script:
        - echo "deploy to production"
    when: manual
```

We will check in, commit, and push the file to our GitLab repository:

```
$ git add .gitlab-ci.yml
$ git commit -m "initial commit"
$ git push origin main
Username for 'http://<url>': <username>
Password for 'http://<url>': <password>
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 512 bytes | 512.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To http://<url> /echou/chapter15_example1.git
   c0b232d..5552a10  main -> main
```

The `.gitlab-ci.yml` file contains instructions for a GitLab CI/CD pipeline in a YAML format. It contains two main sections, stage and job definition:

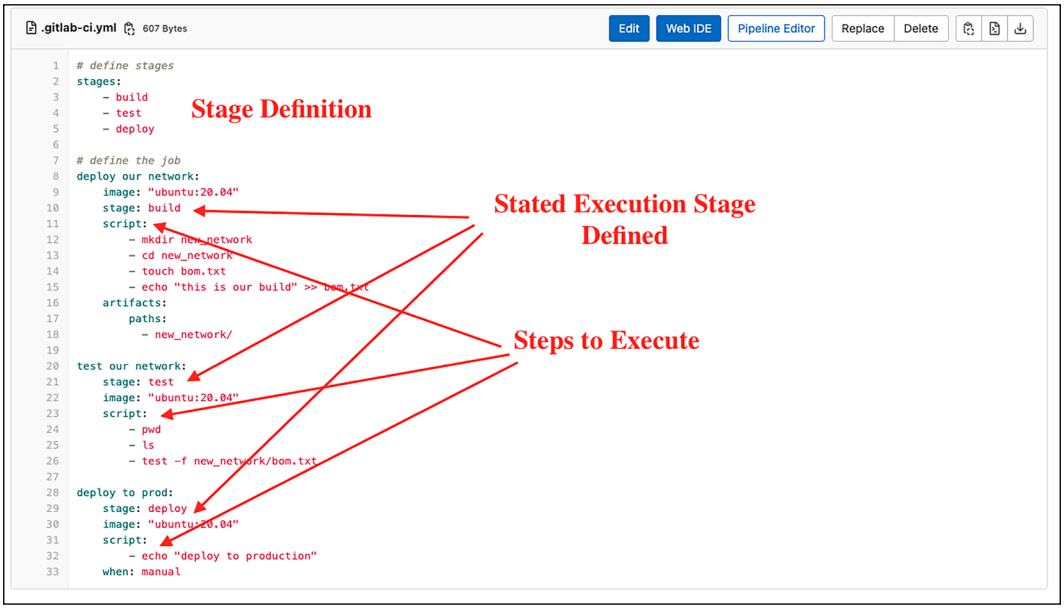


Figure 15.12: GitLab CI File

In our file, we defined three stages with the keyword `stages`. In the execution section, we define the Docker base image to be pulled, the name of the job to be done, the stage it corresponds to, as well as the steps to be executed under `script`. There can be optional instructions such as `artifacts` under `build` and `when` under `deploy`.

If we move back to the project, we can click on **CI/CD -> Pipelines** to show us the status of the job:

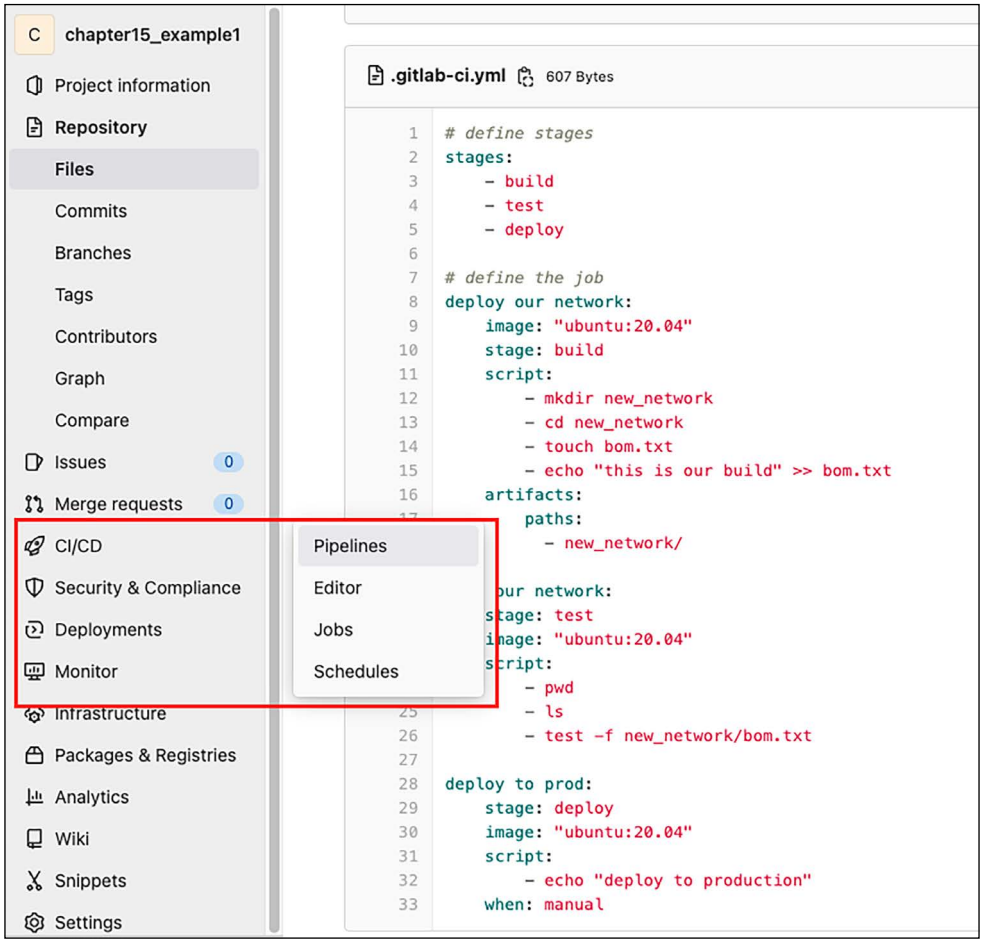


Figure 15.13: CI/CD Pipeline

There are three circles, each representing a stage.

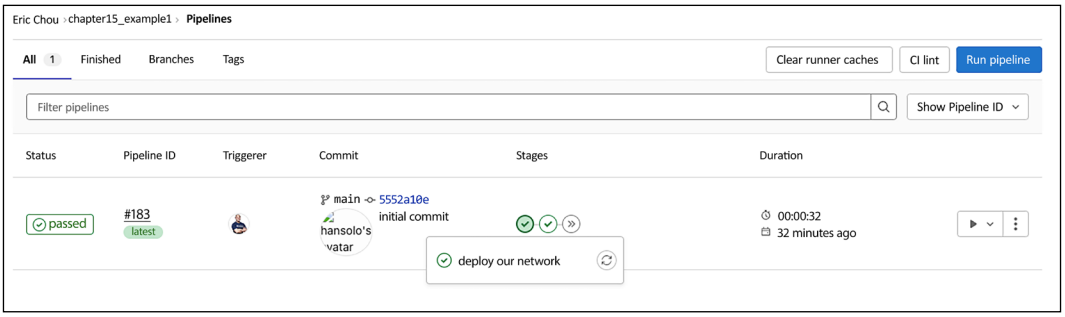


Figure 15.14: Pipeline Output

We can click on the circles and see the container output:

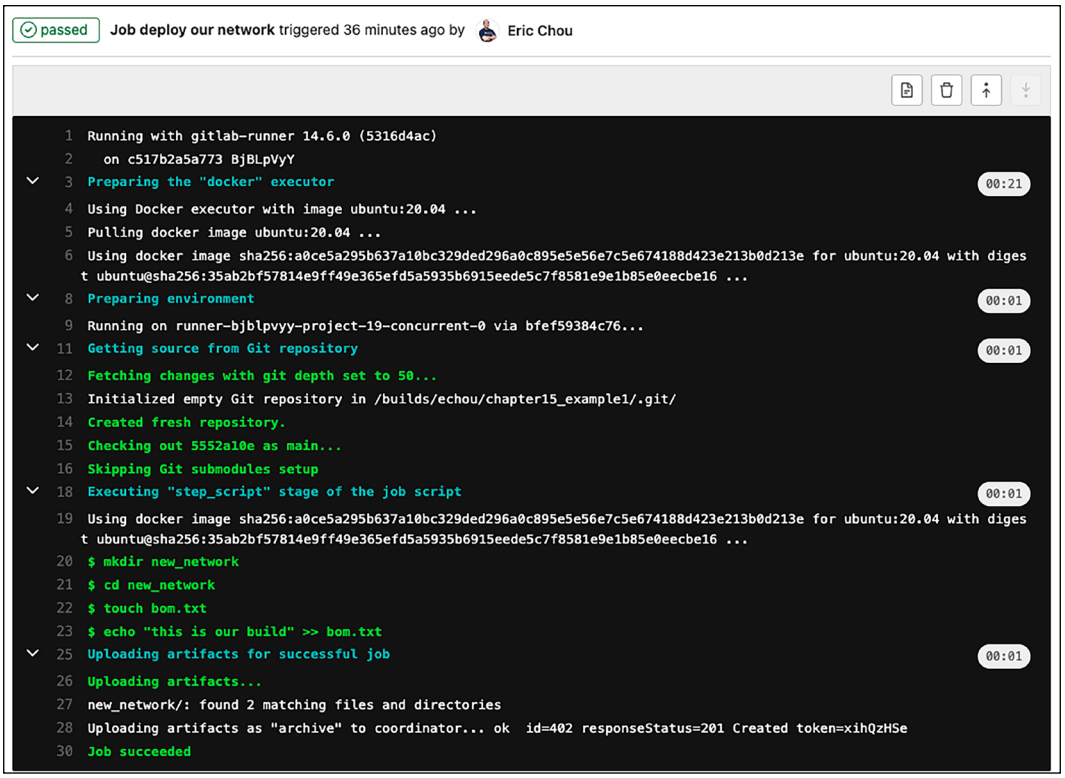


Figure 15.15: Execution Output

Remember we have optional steps under `build` and `deploy`? The artifacts give us something to download:



Figure 15.16: Artifacts

The `when` keyword allows us to manually push out the step instead of having GitLab automatically execute for us:

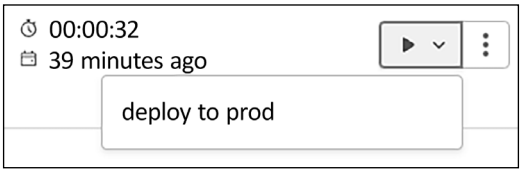


Figure 15.17: Manual Push

Is this awesome, or what? We now have some workers automatically executing jobs for us. We can also leverage many of the features of Git for collaboration, such as inviting colleagues for code review. Let's see another example.

GitLab network example

We will go ahead and create another project named `chapter15_example2` on the GitLab server. On the local machine, we will clone the remote repository:

```
$ git clone http://<url>/echou/chapter15_example2.git
$ cd chapter15_example2/
```

In this example, we will integrate the Nornir library to see how we can execute `show version` on two of the IOSv devices. We will begin by defining the `hosts.yaml` file:

```
---
r1:
    hostname: '192.168.2.218'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'

r2:
    hostname: '192.168.2.219'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'
```

We can then construct the Python script for execution:

```
#!/usr/bin/env python

from nornir import InitNornir
from nornir_utils.plugins.functions import print_result
from nornir_netmiko import netmiko_send_command

nr = InitNornir()

result = nr.run(
    task=netmiko_send_command,
    command_string="show version"
)

print_result(result)
```

We will define a `requirements.txt` file to specify the packages to be installed:

```
$ cat requirements.txt
…
flake8==4.0.1
…
netmiko==3.4.0
nornir==3.2.0
nornir-netmiko==0.1.2
nornir-utils==0.1.2
paramiko==2.9.2
…
```

We will also define the `.gitlab-ci.yml` file to define the stages and the scripts. Notice in the file that we specify another `before_script` step to be executed before any of the stages:

```
stages:
  - Test
  - QA

before_script:
  - python --version
  - pip3 install -r requirements.txt

Test-Job:
  stage: Test
  script:
    - python3 show_version.py

flake8:
  stage: QA
  script:
    - flake8 show_version.py
```

Once the files are checked in and pushed to the repository, we can go to the CI/CD section to look at the outputs. The steps will take longer this time because of the package download time. We can click on the step and examine the execution in real time.

```
125 You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
126 $ python3 show_version.py
127 netmiko_send_command************************************************************
128 * r1 ** changed : False ***********************************************************
129 vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO
130 Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.8(3)M2, RELEASE SOFTWARE (fc2)
131 Technical Support: http://www.cisco.com/techsupport
132 Copyright (c) 1986-2019 by Cisco Systems, Inc.
133 Compiled Thu 28-Mar-19 14:06 by prod_rel_team
134 ROM: Bootstrap program is IOSv
135 r1 uptime is 3 hours, 19 minutes
136 System returned to ROM by reload
137 System image file is "flash0:/vios-adventerprisek9-m"
138 Last reload reason: Unknown reason
139 This product contains cryptographic features and is subject to United
140 States and local country laws governing import, export, transfer and
141 use. Delivery of Cisco cryptographic products does not imply
142 third-party authority to import, export, distribute or use encryption.
143 Importers, exporters, distributors and users are responsible for
144 compliance with U.S. and local country laws. By using this product you
145 agree to comply with applicable laws and regulations. If you are unable
146 to comply with U.S. and local laws, return this product immediately.
147 A summary of U.S. laws governing Cisco cryptographic products may be found at:
148 http://www.cisco.com/wwl/export/crypto/tool/stqrg.html
149 If you require further assistance please contact us by sending email to
150 export@cisco.com.
151 Cisco IOSv (revision 1.0) with  with 472289K/50176K bytes of memory.
152 Processor board ID 9GVZK4GBUY8E6QOSSRDUM
153 3 Gigabit Ethernet interfaces
154 DRAM configuration is 72 bits wide with parity disabled.
155 256K bytes of non-volatile configuration memory.
```

Figure 15.18: Nornir CI/CD Step Execution

We should be able to see the pipeline successfully executed.

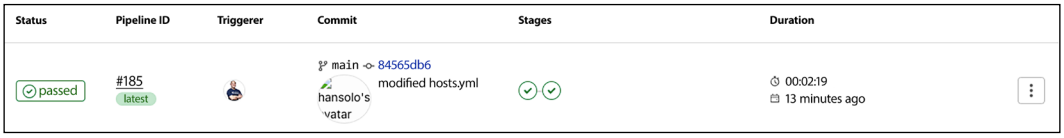


Figure 15.19: CI/CD Result

Using GitLab CI/CD is a terrific way of automating our network operation steps. The pipeline might take longer to set up, but once it is completed, it will save us much time and allow us to preserve our energy to focus on more interesting work. For more information, check out `https://docs.gitlab.com/ee/ci/`.

Summary

In this chapter, we examined the traditional change management process and why it is not a good fit for today's rapidly changing environment. The network needs to evolve with the business to become more agile and adapt to change quickly and reliably.

We looked at the concept of continuous integration, in particular, the open-source GitLab system. GitLab is a full-featured, expandable, continuous integration system widely used in software development. We can adapt the same system for our network operations. We saw two examples using the GitLab Git repository and runners to execute our operations automatically.

In *Chapter 16*, *Test-Driven Development for Networks*, we will look at test-driven development with Python.

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`



16

Test-Driven Development for Networks

In the previous chapters, we used Python to communicate with network devices, monitor and secure a network, automate processes, and extend an on-premises network to public cloud providers. We have come a long way from having to exclusively use a terminal window and manage the network with a CLI. When working together, the services we have built function like a well-oiled machine that gives us a beautiful, automated, programmable network. However, the network is never static and is constantly changing to meet the demands of the business. What happens when the services we build are not working optimally? As we have done with monitoring and source control systems, we are actively trying to detect faults.

In this chapter, we are extending the active detection concept with **test-driven development (TDD)**. We will cover the following topics:

- An overview of TDD
- Topology as code
- Writing tests for networking
- pyATS and Genie

We'll begin this chapter with an overview of TDD before diving into its applications within networks. We will look at examples of using Python with TDD and gradually move from specific tests to larger network-based tests.

Test-driven development overview

The idea of TDD has been around for a while. American software engineer Kent Beck, among others, is credited with leading the TDD movement, along with agile software development. Agile software development requires very short build-test-deploy development cycles; all software requirements are turned into test cases. These test cases are usually written before the code is written, and the software code is only accepted when the test passes.

The same idea can be drawn in parallel with network engineering. For example, when we face the challenge of designing a modern network, we can break the process down into the following steps, from high-level design requirements to the network tests that we can deploy:

1. We start with the overall requirement for the new network. Why do we need to design a new network or part of a new network? Maybe it is for new server hardware, a new storage network, or new microservice software architecture.
2. The new requirements are broken down into smaller, more specific requirements. This could be evaluating a new switch platform, testing a possibly more efficient routing protocol, or a new network topology (for example, fat-tree). Each of the smaller requirements can be broken down into the categories of **required** or **optional**.
3. We draw out the test plan and evaluate it against the potential candidates for solutions.
4. The test plan will work in reverse order; we will start by testing the features, then integrate the new feature into a bigger topology. Finally, we will try to run our test close to a production environment.

What I am trying to get at is that, even without realizing it, we might already be adopting some of the TDD methodologies in the normal network engineering process. This was part of my revelation when I was studying the TDD mindset. We are already implicitly following this best practice without formalizing the method.

By gradually moving parts of the network to code, we can use TDD for the network even more. If our network topology is described in a hierarchical format in XML or JSON, each component can be correctly mapped and expressed in the desired state, which some might call "the source of truth." This is the desired state that we can write test cases against to test production deviation from this state. For example, if our desired state calls for a full mesh of iBGP neighbors, we can always write a test case to check against our production devices for the number of iBGP neighbors it has.

The sequence of TDD is loosely based on the following six steps:

1. Write a test with the result in mind
2. Run all tests and see whether the new test fails
3. Write the code
4. Run the test again
5. Make the necessary changes if the test fails
6. Repeat

As with any process, how closely we follow the guideline is a judgment call. I prefer to treat these guidelines as goals and follow them somewhat loosely. For example, the TDD process calls for writing test cases before writing any code or, in our instance, before any network components are built. As a personal preference, I always like to see a working version of the network or code before writing test cases. It gives me a higher level of confidence, so if anybody is judging my TDD process, I might just get a big fat "F." I also like to jump around between different levels of testing; sometimes, I test a small portion of the network, Other times, I conduct a system-level end-to-end test, such as a ping or traceroute test.

The point is I do not believe there is a one-size-fits-all approach to testing. It depends on personal preference and the scope of the project. This is true for most of the engineers I have worked with. It is a good idea to keep the framework in mind so we have a working blueprint to follow, but you are the best judge of your problem-solving style.

Before we delve further into TDD, let's cover some of the most common terminologies in the following section so that we have a good conceptual grounding before getting into more details.

Test definitions

Let's look at some of the terms commonly used in TDD:

- **Unit test**: Checks a small piece of code. This is a test that is run against a single function or class.
- **Integration test**: Checks multiple components of a code base; multiple units are combined and tested as a group. This can be a test that checks against a Python module or multiple modules.

- **System test**: Checks from end to end. This is a test that runs as close to what an end user would see as possible.
- **Functional test**: Checks against a single function.
- **Test coverage**: A term defined as the determination of whether our test cases cover the application code. This is typically done by examining how much code is exercised when we run the test cases.
- **Test fixtures**: A fixed state that forms a baseline for running our tests. The purpose of a test fixture is to ensure there is a well-known and fixed environment in which tests are run so they are repeatable.
- **Setup and teardown**: All the prerequisite steps are added in the setup and cleaned up in the teardown.

The terms might seem very software development-centric, and some might not be relevant to network engineering. Remember that the terms are a way for us to communicate a concept or step. We will be using these terms in the rest of this chapter. As we use the terms more in the network engineering context, they might become clearer. With that covered, let's dive into treating network topology as code.

Topology as code

When we discuss topology as code, an engineer might jump up and declare: "The network is too complex. It is impossible to summarize it into code!" From personal experience, this has happened in some of the meetings I have been in. In the meeting, we would have a group of software engineers who want to treat infrastructure as code, but the traditional network engineers in the room would declare that it was impossible. Before you do the same and yell at me across the pages of this book, let's keep an open mind. Would it help if I told you we have been using code to describe our topology in this book already?

If you take a look at any of the lab topology files that we have been using in this book, they are simply YAML files that include a description of the relationship between nodes. For example, in this chapter, we will use same topology we have been using for the last few chapters:

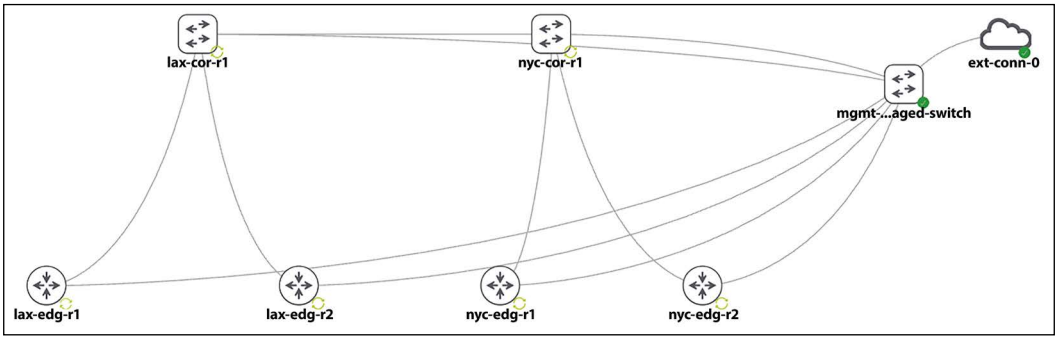


Figure 16.1: Lab Topology

If we open up the topology file, `chapter16_topology.yaml`, with a text editor, we will see that the file is a YAML file describing the node and the links between the nodes:

```
lab:
  description: Imported from 2_DC_Topology.virl
  notes: |-
    ## Import Progress
    - processing node /lax-edg-r1 (iosv)
    - processing node /lax-edg-r2 (iosv)
    - processing node /nyc-edg-r1 (iosv)
    - processing node /nyc-edg-r2 (iosv)
    - processing node /lax-cor-r1 (nxosv)
    - processing node /nyc-cor-r1 (nxosv)
    - link GigabitEthernet0/1.lax-edg-r1 -> Ethernet2/1.lax-cor-r1
    - link GigabitEthernet0/1.lax-edg-r2 -> Ethernet2/2.lax-cor-r1
    - link GigabitEthernet0/1.nyc-edg-r1 -> Ethernet2/1.nyc-cor-r1
    - link GigabitEthernet0/1.nyc-edg-r2 -> Ethernet2/2.nyc-cor-r1
    - link Ethernet2/3.lax-cor-r1 -> Ethernet2/3.nyc-cor-r1
  timestamp: 1615749425.6802542
  title: 2_DC_Topology.yaml
  version: 0.0.4
```

The node section include each of the node's id, label, definition, and configurations:

```
nodes:
  - id: n0
    label: lax-edg-r1
    node_definition: iosv
    x: -100
    y: 200
    configuration: |-
      !
      ! Last configuration change at 02:26:08 UTC Fri Apr 17 2020 by cisco
      !
      version 15.6
      service timestamps debug datetime msec
      service timestamps log datetime msec
      no service password-encryption
      !
      hostname lax-edg-r1
      !
      boot-start-marker
      boot-end-marker
      !
      !
      vrf definition Mgmt-intf
       !
       address-family ipv4
       exit-address-family
       !
<skip>
```

If we open up the previous chapter's lab topology file, which contain Linux nodes, we can see the Linux host nodes can be described the same way as the network nodes:

```
  - id: n5
    label: Client
    node_definition: server
    x: 0
    y: 0
    configuration: |-
```

```
      # converted cloud-config
      hostname Client
      ifconfig eth1 up 10.0.0.9 netmask 255.255.255.252
      route add -net 10.0.0.0/8 gw 10.0.0.10 dev eth1
      route add -net 192.168.0.0/28 gw 10.0.0.10 dev eth1

      # original cloud-config
      # #cloud-config
      # bootcmd:
      # - ln -s -t /etc/rc.d /etc/rc.local
      # hostname: Client
      # manage_etc_hosts: true
      # runcmd:
      # - start ttyS0
      # - systemctl start getty@ttyS0.service
      # - systemctl start rc-local
      # - sed -i '/^\s*PasswordAuthentication\s\+no/d' /etc/ssh/sshd_
config
      # - echo "UseDNS no" >> /etc/ssh/sshd_config
      # - service ssh restart
      # - service sshd restart
```

By expressing the network as code, we can declare a source of truth for our network. We can write test code to compare the actual production value against this blueprint. We will use this topology file as the base and compare the production network value against it.

XML parsing example

Besides YAML, another popular way to express topology as code is XML. In fact, that was the format that the predecessor of CML 2, Cisco VIRL used. From previous editions of the book, I have provided an example of a two-host, two-network node example file named `chapter15_topology.virl` for our parsing example.

To work with XML files, we can use Python to extract the element from this topology file and store it as a Python data type so we can work with it. In `chapter16_1_xml.py`, we will use `ElementTree` to parse the `virl` topology file and construct a dictionary consisting of the information of our devices:

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

```
with open('chapter15_topology.virl', 'rt') as f:
    tree = ET.parse(f)
devices = {}
for node in tree.findall('./{http://www.cisco.com/VIRL}node'):
    name = node.attrib.get('name')
    devices[name] = {}
    for attr_name, attr_value in sorted(node.attrib.items()):
        devices[name][attr_name] = attr_value
# Custom attributes
devices['iosv-1']['os'] = '15.6(3)M2'
devices['nx-osv-1']['os'] = '7.3(0)D1(1)'
devices['host1']['os'] = '16.04'
devices['host2']['os'] = '16.04'
pprint.pprint(devices)
```

The result is a Python dictionary that consists of the devices according to our topology file.

We can also add customary items to the dictionary:

```
(venv) $ python chapter16_1_xml.py
{'host1': {'location': '117,58',
           'name': 'host1',
           'os': '16.04',
           'subtype': 'server',
           'type': 'SIMPLE'},
 'host2': {'location': '347,66',
           'name': 'host2',
           'os': '16.04',
           'subtype': 'server',
           'type': 'SIMPLE'},
 'iosv-1': {'ipv4': '192.168.0.3',
            'location': '182,162',
            'name': 'iosv-1',
            'os': '15.6(3)M2',
            'subtype': 'IOSv',
            'type': 'SIMPLE'},
 'nx-osv-1': {'ipv4': '192.168.0.1',
              'location': '281,161',
              'name': 'nx-osv-1',
```

```
                'os': '7.3(0)D1(1)',
                'subtype': 'NX-OSv',
               'type': 'SIMPLE'}}
```

If we want to compare this "source of truth" to the production device version, we can use our script from *Chapter 3*, *APIs and Intent-Driven Networking*, `cisco_nxapi_2.py`, to retrieve the production NX-OSv device's software version. We can then compare the value we received from our topology file with the production device's information. Later, we can use Python's built-in `unittest` module to write test cases.

We will discuss the `unittest` module in just a bit. Feel free to skip ahead and come back to this example if you'd like.

Here is the relevant `unittest` code in `chapter16_2_validation.py`:

```
import unittest
<skip>
# Unittest Test case
class TestNXOSVersion(unittest.TestCase):
    def test_version(self):
        self.assertEqual(nxos_version, devices['nx-osv-1']['os'])
if __name__ == '__main__':
    unittest.main()
```

When we run the validation test, we can see that the test passes because the software version in production matches what we expected:

```
$ python chapter16_2_validation.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

If we manually change the expected NX-OSv version value to introduce a failure case, we will see the following failed output:

```
$ python chapter16_3_test_fail.py
F
======================================================================
FAIL: test_version (__main__.TestNXOSVersion)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "chapter15_3_test_fail.py", line 50, in test_version
    self.assertEqual(nxos_version, devices['nx-osv-1']['os'])
AssertionError: '7.3(0)D1(1)' != '7.4(0)D1(1)'
- 7.3(0)D1(1)
?   ^
+ 7.4(0)D1(1)
?   ^
----------------------------------------------------------------------
Ran 1 test in 0.001s
FAILED (failures=1)
```

We can see that the test case result returned as failed; the reason for the failure was the version mismatch between the two values. As we saw in the last example, the Python `unittest` module is a great way to test our existing code based on our expected results. Let's take a deeper look at the module.

Python's unittest module

The Python standard library includes a module named `unittest`, which handles test cases where we can compare two values to determine whether a test passes. In the previous example, we saw how to use the `assertEqual()` method to compare two values to return either `True` or `False`. Here is an example, `chapter16_4_unittest.py`, that uses the built-in `unittest` module to compare two values:

```
#!/usr/bin/env python3
import unittest
class SimpleTest(unittest.TestCase):
    def test(self):
        one = 'a'
        two = 'a'
        self.assertEqual(one, two)
```

Using the `python3` command-line interface, the `unittest` module can automatically discover the test cases in the script:

```
$ python -m unittest chapter16_4_unittest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Besides comparing two values, here are more examples of testing whether the expected value is `True` or `False`. We can also generate custom failure messages when a failure occurs:

```
#!/usr/bin/env python3
# Examples from https://pymotw.com/3/unittest/index.html#module-unittest
import unittest
class Output(unittest.TestCase):
    def testPass(self):
        return
    def testFail(self):
        self.assertFalse(True, 'this is a failed message')
    def testError(self):
        raise RuntimeError('Test error!')
    def testAssesrtTrue(self):
        self.assertTrue(True)
    def testAssertFalse(self):
        self.assertFalse(False)
```

We can use `-v` for the option to display a more detailed output:

```
$ python -m unittest -v chapter16_5_more_unittest
testAssertFalse (chapter16_5_more_unittest.Output) ... ok
testAssesrtTrue (chapter16_5_more_unittest.Output) ... ok
testError (chapter16_5_more_unittest.Output) ... ERROR
testFail (chapter16_5_more_unittest.Output) ... FAIL
testPass (chapter16_5_more_unittest.Output) ... ok

======================================================================
ERROR: testError (chapter16_5_more_unittest.Output)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16/
chapter16_5_more_unittest.py", line 14, in testError
    raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: testFail (chapter16_5_more_unittest.Output)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16/
chapter16_5_more_unittest.py", line 11, in testFail
    self.assertFalse(True, 'this is a failed message')
AssertionError: True is not false : this is a failed message

----------------------------------------------------------------------
Ran 5 tests in 0.001s

FAILED (failures=1, errors=1)
```

Starting from Python 3.3, the `unittest` module includes the `mock` object library by default (`https://docs.python.org/3/library/unittest.mock.html`). This is a very useful module that you can use to make a fake HTTP API call to a remote resource without actually making the call. For example, we have seen using NX-API to retrieve the NX-OS version number. What if we want to run our test, but we do not have an NX-OS device available? We can use the `unittest` mock object.

In `chapter16_5_more_unittest_mocks.py`, we created a class with a method to make HTTP API calls and expect a JSON response:

```
# Our class making API Call using requests
class MyClass:
    def fetch_json(self, url):
        response = requests.get(url)
        return response.json()
```

We also created a function that mocks two URL calls:

```
# This method will be used by the mock to replace requests.get
def mocked_requests_get(*args, **kwargs):
    class MockResponse:
        def __init__(self, json_data, status_code):
            self.json_data = json_data
            self.status_code = status_code
        def json(self):
            return self.json_data
    if args[0] == 'http://url-1.com/test.json':
        return MockResponse({"key1": "value1"}, 200)
    elif args[0] == 'http://url-2.com/test.json':
```

```
        return MockResponse({"key2": "value2"}, 200)
    return MockResponse(None, 404)
```

Finally, we make the API call to the two URLs in our test case. However, we are using the `mock.patch` decorator to intercept the API calls:

```
# Our test case class
class MyClassTestCase(unittest.TestCase):
    # We patch 'requests.get' with our own method. The mock object is
passed in to our test case method.
    @mock.patch('requests.get', side_effect=mocked_requests_get)
    def test_fetch(self, mock_get):
        # Assert requests.get calls
        my_class = MyClass()
        # call to url-1
        json_data = my_class.fetch_json('http://url-1.com/test.json')
        self.assertEqual(json_data, {"key1": "value1"})
        # call to url-2
        json_data = my_class.fetch_json('http://url-2.com/test.json')
        self.assertEqual(json_data, {"key2": "value2"})
        # call to url-3 that we did not mock
        json_data = my_class.fetch_json('http://url-3.com/test.json')
        self.assertIsNone(json_data)
if __name__ == '__main__':
    unittest.main()
```

When we run the test, we will see that the test passes without needing to make an actual API call to the remote endpoint. Neat, huh?

```
$ python chapter16_5_more_unittest_mocks.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

For more information on the `unittest` module, Doug Hellmann's Python module of the week (`https://pymotw.com/3/unittest/index.html#module-unittest`) is an excellent source of short and precise examples on the `unittest` module. As always, the Python documentation is a good source of information as well: `https://docs.python.org/3/library/unittest.html`.

More on Python testing

In addition to the built-in `unittest` library, there are lots of other testing frameworks from the Python community. `pytest` is one of the most robust, intuitive Python testing frameworks and is worth a look. `pytest` can be used for all types and levels of software testing. It can be used by developers, QA engineers, individuals practicing TDD, and open source projects.

Many large-scale open source projects have switched from `unittest` or `nose` (another Python test framework) to `pytest`, including Mozilla and Dropbox. The attractive features of `pytest` include the third-party plugin model, a simple fixture model, and assert rewriting.

If you want to learn more about the `pytest` framework, I highly recommend *Python Testing with pytest* by Brian Okken (ISBN 978-1-68050-240-4). Another great source is the pytest documentation: `https://docs.pytest.org/en/latest/`.

`pytest` is command line-driven; it can find the tests we have written automatically and run them by appending the `test` prefix in our function. We will need to install `pytest` before we can use it:

```
$ pip install pytest
$ python
Python 3.10.6 (main, Aug 10 2022, 11:40:04) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pytest
>>> pytest.__version__
'7.1.3'
>>>
```

Let's look at some examples using `pytest`.

pytest examples

The first `pytest` example, `chapter16_6_pytest_1.py`, will be a simple assert for two values:

```
#!/usr/bin/env python3
def test_passing():
    assert(1, 2, 3) == (1, 2, 3)
def test_failing():
    assert(1, 2, 3) == (3, 2, 1)
```

When we run `pytest` with the `-v` option, `pytest` will give us a pretty robust answer for a reason for the failure. The verbose output is one of the reasons people like `pytest`:

```
$ pytest -v chapter16_6_pytest_1.py
================================ test session starts ====================
============
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0 -- /home/
echou/Mastering_Python_Networking_Fourth_Edition/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16
collected 2 items

chapter16_6_pytest_1.py::test_passing PASSED
[ 50%]
chapter16_6_pytest_1.py::test_failing FAILED
[100%]

====================================== FAILURES ==========================
============
_____________________________________ test_failing ______________________
___________

    def test_failing():
>       assert(1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Full diff:
E         - (3, 2, 1)
E         ?  ^     ^
E         + (1, 2, 3)
E         ?  ^     ^

chapter16_6_pytest_1.py:7: AssertionError
============================== short test summary info
==============================
FAILED chapter16_6_pytest_1.py::test_failing - assert (1, 2, 3) == (3, 2,
1)
============================ 1 failed, 1 passed in 0.03s
============================
```

In the second pytest example, `chapter16_7_pytest_2.py`, we will create a `router` object. The `router` object will be initiated with some values of `None` and some with default values. We will use `pytest` to test one instance with the default and one instance without:

```
#!/usr/bin/env python3
class router(object):
    def __init__(self, hostname=None, os=None, device_type='cisco_ios'):
        self.hostname = hostname
        self.os = os
        self.device_type = device_type
        self.interfaces = 24
def test_defaults():
    r1 = router()
    assert r1.hostname == None
    assert r1.os == None
    assert r1.device_type == 'cisco_ios'
    assert r1.interfaces == 24
def test_non_defaults():
    r2 = router(hostname='lax-r2', os='nxos', device_type='cisco_nxos')
    assert r2.hostname == 'lax-r2'
    assert r2.os == 'nxos'
    assert r2.device_type == 'cisco_nxos'
    assert r2.interfaces == 24
```

When we run the test, we will see whether the instance was accurately applied with the default values:

```
$ pytest chapter16_7_pytest_2.py
============================== test session starts ======================
============
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16
collected 2 items

chapter16_7_pytest_2.py ..
[100%]

=============================== 2 passed in 0.01s =======================
============
```

If we were to replace the previous `unittest` example with `pytest`, in `chapter16_8_pytest_3.py`, we can see the syntax with `pytest` is simpler:

```
# pytest test case
def test_version():
    assert devices['nx-osv-1']['os'] ==  nxos_version
```

Then we run the test with the `pytest` command line:

```
$ pytest chapter16_8_pytest_3.py
================================ test session starts ====================
============
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16
collected 1 item

chapter16_8_pytest_3.py .
[100%]

================================= 1 passed in 3.80s =====================
============
```

Between `unittest` and `pytest`, I find `pytest` more intuitive to use. However, since `unittest` is included in the standard library, many teams might prefer using the `unittest` module for their testing.

Besides doing tests on code, we can also write tests to test our network as a whole. After all, users care more about their services and applications functioning properly and less about individual pieces. We will look at writing tests for the network in the next section.

Writing tests for networking

So far, we have been mostly writing tests for our Python code. We have used both the `unittest` and `pytest` libraries to assert `True/False` and `equal/non-equal` values. We were also able to write mocks to intercept our API calls when we do not have an actual API-capable device but still want to run our tests.

In this section, let's look at how we can write tests relevant to the networking world. There is no shortage of commercial products regarding network monitoring and testing. Over the years, I have come across many of them. However, in this section, I prefer to use simple, open source tools for my tests.

Testing for reachability

Often, the first step of troubleshooting is to conduct a small reachability test. For network engineers, `ping` is our best friend when it comes to network reachability tests. It is a way to test the reachability of a host on an IP network by sending a small package across the network to the destination.

We can automate the `ping` test via the `OS` module or the `subprocess` module:

```
>>> import os
>>> host_list = ['www.cisco.com', 'www.google.com']
>>> for host in host_list:
...     os.system('ping -c 1 ' + host)
...
PING www.cisco.com(2001:559:19:289b::b33 (2001:559:19:289b::b33)) 56 data
bytes
64 bytes from 2001:559:19:289b::b33 (2001:559:19:289b::b33): icmp_seq=1
ttl=60 time=11.3 ms
--- www.cisco.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 11.399/11.399/11.399/0.000 ms
0
PING www.google.com(sea15s11-in-x04.1e100.net (2607:f8b0:400a:808::2004))
56 data bytes
64 bytes from sea15s11-in-x04.1e100.net (2607:f8b0:400a:808::2004): icmp_
seq=1 ttl=54 time=10.8 ms
--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 10.858/10.858/10.858/0.000 ms
0
```

The `subprocess` module offers the additional benefit of catching the output:

```
>>> import subprocess
>>> for host in host_list:
...     print('host: ' + host)
...     p = subprocess.Popen(['ping', '-c', '1', host], stdout=subprocess.
PIPE)
...
host: www.cisco.com
```

```
host: www.google.com
>>> print(p.communicate())
(b'PING www.google.com(sea15s11-in-x04.1e100.net
(2607:f8b0:400a:808::2004)) 56 data bytes\n64 bytes from sea15s11-in-
x04.1e100.net (2607:f8b0:400a:808::2004): icmp_seq=1 ttl=54 time=16.9
ms\n\n--- www.google.com ping statistics ---\n1 packets transmitted,
1 received, 0% packet loss, time 0ms\nrtt min/avg/max/mdev =
16.913/16.913/16.913/0.000 ms\n', None)
>>>
```

These two modules prove to be very useful in many situations. Any command we can execute in the Linux and Unix environments can be executed via the `OS` or `subprocess` module.

Testing for network latency

The topic of network latency can sometimes be subjective. Working as network engineers, we are often faced with the user saying that the network is slow. However, "slow" is a very subjective term.

If we could construct tests that turn subjective terms into objective values, it would be very helpful. We should do this consistently to compare the values over a time series of data.

This can sometimes be difficult since the network is stateless by design. Just because one packet is successful does not guarantee success for the next packet. The best approach I have seen over the years is to use `ping` across many hosts frequently and log the data, conducting a ping-mesh graph. We can leverage the same tools we used in the previous example, catch the return-result time, and keep a record. We do this in `chapter16_10_ping.py`:

```
#!/usr/bin/env python3
import subprocess
host_list = ['www.cisco.com', 'www.google.com']
ping_time = []
for host in host_list:
    p = subprocess.Popen(['ping', '-c', '1', host], stdout=subprocess.
PIPE)
    result = p.communicate()[0]
    host = result.split()[1]
    time = result.split()[13]
    ping_time.append((host, time))
print(ping_time)
```

In this case, the result is kept in a `tuple` and put into a `list`:

```
$ python chapter16_10_ping.py
[(b'e2867.dsca.akamaiedge.net', b'ttl=54'), (b'www.google.com',
b'ttl=58')]
```

This is not perfect and is merely a starting point for monitoring and troubleshooting. However, in the absence of other tools, this offers some baseline of objective values.

Testing for security

We saw one of the best tools for security testing in *Chapter 6*, *Network Security with Python*, which was Scapy. There are lots of open source tools for security, but none offer the flexibility that comes with constructing our packets.

Another great tool for network security testing is `hping3` (`https://docs.python-cerberus.org/en/stable/`). It offers a simple way to generate a lot of packets at once. For example, you can use the following one-liner to generate a TCP SYN flood:

```
# DON'T DO THIS IN PRODUCTION #
echou@ubuntu:/var/log$ sudo hping3 -S -p 80 --flood 192.168.1.202
HPING 192.168.1.202 (eth0 192.168.1.202): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.1.202 hping statistic ---
2281304 packets transmitted, 0 packets received, 100% packet loss round-
trip min/avg/max = 0.0/0.0/0.0 ms
echou@ubuntu:/var/log$
```

Again, since this is a command-line tool, we can use the `subprocess` module to automate any `hping3` tests that we want.

Testing for transactions

The network is a crucial part of the infrastructure, but it is only a part of it. What the users care about is often the service that runs on top of the network. If the user is trying to watch a YouTube video or listen to a podcast but cannot, in their opinion, the service is broken. We might know that the network transport is not at fault, but that doesn't comfort the user.

For this reason, we should implement tests that are as similar to the user's experience as possible. In the example of a YouTube video, we might not be able to duplicate the YouTube experience 100% (unless you work for Google).

Still, we can implement a layer-7 service as close to the network edge as possible. We can then simulate the transaction from a client at a regular interval as a transactional test.

The Python `HTTP` standard library module is a module that I often use when I need to test layer-7 reachability on a web service quickly. We already saw how to use it when we were performing network monitoring in *Chapter 4*, *The Python Automation Framework – Ansible*, but it's worth seeing again:

```
$ python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 ...
127.0.0.1 - - [25/Jul/2018 10:15:23] "GET / HTTP/1.1" 200 -
```

If we can simulate a full transaction for the expected service, that is even better. But Python's simple `HTTP` server module in the standard library is always a great one for running some ad hoc web service tests.

Testing for network configuration

In my opinion, the best test for network configuration is using standardized templates to generate the configuration and back up the production configuration often. We have seen how we can use the Jinja2 template to standardize our configuration per device type or role. This will eliminate many of the mistakes caused by human error, such as copy and paste.

Once the configuration is generated, we can write tests against the configuration for known characteristics that we would expect before we push the configuration to production devices. For example, there should be no overlap of IP addresses in all of the network when it comes to loopback IP, so we can write a test to see whether the new configuration contains a loopback IP that is unique across our devices.

Testing for Ansible

For the time I have been using Ansible, I cannot recall using a `unittest`-like tool to test a playbook. For the most part, the playbooks use modules that were tested by module developers.



If you want a lightweight data validation tool, please check out Cerberus (`https://docs.python-cerberus.org/en/stable/`).

Ansible provides unit tests for their library of modules. Unit tests in Ansible are currently the only way to drive tests from Python within Ansible's continuous-integration process. The unit tests that are run today can be found under `/test/units` (`https://github.com/ansible/ansible/tree/devel/test/units`).

The Ansible testing strategy can be found in the following documents:

- **Testing Ansible**: `https://docs.ansible.com/ansible/latest/dev_guide/testing.html`
- **Unit tests**: `https://docs.ansible.com/ansible/latest/dev_guide/testing_units.html`
- **Unit testing Ansible modules**: `https://docs.ansible.com/ansible/latest/dev_guide/testing_units_modules.html`

One of the interesting Ansible testing frameworks is Molecule (`https://pypi.org/project/molecule/`). It intends to aid in the development and testing of Ansible roles. Molecule supports testing with multiple instances, operating systems, and distributions. I have not used this tool, but it is where I would start if I wanted to perform more testing on my Ansible roles.

We should now know how to write tests for our network, whether testing for reachability, latency, security, transaction, or network configuration.

In the next section, we will take a look at an extensive testing framework developed by Cisco (and recently released as open source) called `pyATS`. Much to their credit, releasing such an extensive framework as open source for the benefit of the community was a great gesture by Cisco.

pyATS and Genie

pyATS (`https://developer.cisco.com/pyats/`) is an end-to-end testing ecosystem originally developed by Cisco and made available to the public in late 2017. The pyATS library was formerly known as Genie; they will often be referred to in the same context. Because of its roots, the framework is very focused on network testing.

pyATS, and the pyATS library (also known as Genie), was the winner of the 2018 Cisco Pioneer Award. We should all applaud Cisco for making the framework open source and available to the public. Good job, Cisco DevNet!

The framework is available on PyPI:

```
$ pip install pyats
```

To get started, we can look at some of the example scripts on the GitHub repository, `https://github.com/CiscoDevNet/pyats-sample-scripts`. The tests start with creating a testbed file in YAML format. We will create a simple `chapter16_pyats_testbed_1.yml` testbed file for our `lax-edge-r1-edg-r1` device. The file should look similar to the Ansible inventory file that we have seen before:

```
testbed:
    name: Chapter_16_pyATS
    tacacs:
      username: cisco
    passwords:
      tacacs: cisco
      enable: cisco

devices:
   lax-edg-r1:
       alias: lax-edg-r1
       type: ios
       connections:
         defaults:
           class: unicon.Unicon
         management:
           ip: 192.168.2.51
           protocol: ssh

topology:
    lax-edg-r1:
        interfaces:
            GigabitEthernet0/1:
                ipv4: 10.0.0.1/30
                link: link-1
                type: ethernet
            Loopback0:
                ipv4: 192.168.0.10/32
                link: iosv-1_Loopback0
                type: loopback
```

In our first script, `chapter16_11_pyats_1.py`, we will load the testbed file, connect to the device, issue a `show version` command, then disconnect from the device:

```
#!/usr/bin/env python3
#
# derived from https://devnet-pubhub-site.s3.amazonaws.com/media/pyats/
docs/getting_started/index.html
#
from pyats.topology import loader

# load testbed
testbed = loader.load('chapter16_pyats_testbed_1.yml')

# access the device
testbed.devices
lax_edg_r1 = testbed.devices['lax-edg-r1']

# establish connectivity
lax_edg_r1.connect()

# issue command
print(lax_edg_r1.execute('show version'))

# disconnect
lax_edg_r1.disconnect()
```

When we execute the command, we can see the output is a mixture of the pyATS setup as well as the actual output of the device. This is similar to the Paramiko scripts we have seen before but note that pyATS took care of the underlying connection for us:

```
$ python chapter16_11_pyats_1.py
/home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16/pyATS/
chapter16_11_pyats_1.py:8: DeprecationWarning: 'tacacs.username' is
deprecated in the testbed YAML.  This key has been moved to 'credentials'.
  testbed = loader.load('chapter16_pyats_testbed_1.yml')
/home/echou/Mastering_Python_Networking_Fourth_Edition/Chapter16/pyATS/
chapter16_11_pyats_1.py:8: DeprecationWarning: 'passwords.tacacs' is
deprecated in the testbed YAML.  Use 'credentials' instead.
  testbed = loader.load('chapter16_pyats_testbed_1.yml')
```

```
device's os is not provided, unicon may not use correct plugins

2022-09-25 17:03:08,615: %UNICON-INFO: +++ lax-edg-r1 logfile /tmp/lax-
edg-r1-cli-20220925T170308615.log +++
<skip>
2022-09-25 17:03:09,275: %UNICON-INFO: +++ connection to spawn: ssh -l
cisco 192.168.2.51, id: 140685765498848 +++

2022-09-25 17:03:09,276: %UNICON-INFO: connection to lax-edg-r1
cisco@192.168.2.51's password:

**************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS  *
* education. IOSv is provided as-is and is not supported by Cisco's      *
* Technical Advisory Center. Any use or disclosure, in whole or in part, *
* of the IOSv Software or Documentation to any third party for any       *
* purposes is expressly prohibited except as otherwise authorized by     *
* Cisco in writing.                                                      *
**************************************************************************
lax-edg-r1#

2022-09-25 17:03:09,364: %UNICON-INFO: +++ initializing handle +++

2022-09-25 17:03:09,427: %UNICON-INFO: +++ lax-edg-r1 with via
'management': executing command 'term length 0' +++
term length 0
lax-edg-r1#

2022-09-25 17:03:09,617: %UNICON-INFO: +++ lax-edg-r1 with via
'management': executing command 'term width 0' +++
term width 0
lax-edg-r1#

2022-09-25 17:03:09,821: %UNICON-INFO: +++ lax-edg-r1 with via
'management': executing command 'show version' +++
show version
Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version
15.8(3)M2, RELEASE SOFTWARE (fc2)
```

```
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2019 by Cisco Systems, Inc.
Compiled Thu 28-Mar-19 14:06 by prod_rel_team
```

In the second example, we will see a full example of connection setup, test cases, then connection teardown. First, we will add the `lax-cor-r1` device to our testbed in `chapter16_pyats_testbed_2.yml`. The additional device is needed as the connected device to `iosv-1` for our ping test:

```
testbed:
    name: Chapter_16_pyATS
    tacacs:
      username: cisco
    passwords:
      tacacs: cisco
      enable: cisco

devices:
    lax-edg-r1:
       alias: iosv-1
       type: ios
       connections:
         defaults:
           class: unicon.Unicon
         vty:
           ip: 192.168.2.50
           protocol: ssh

    lax-cor-r1:
        alias: nxosv-1
        type: ios
        connections:
          defaults:
            class: unicon.Unicon
          vty:
            ip: 192.168.2.51
            protocol: ssh

topology:
```

```
    lax-edg-r1:
        interfaces:
            GigabitEthernet0/1:
                ipv4: 10.0.0.1/30
                link: link-1
                type: ethernet
            Loopback0:
                ipv4: 192.168.0.10/32
                link: lax-edg-r1_Loopback0
                type: loopback
    lax-cor-r1:
        interfaces:
            Eth2/1:
                ipv4: 10.0.0.2/30
                link: link-1
                type: ethernet
            Loopback0:
                ipv4: 192.168.0.100/32
                link: lax-cor-r1_Loopback0
                type: loopback
```

In `chapter16_12_pyats_2.py`, we will use the `aest` module from pyATS with various decorators. Besides setup and cleanup, the `ping` test is in the `PingTestCase` class:

```
@aetest.loop(device = ('ios1',))
class PingTestcase(aetest.Testcase):

    @aetest.test.loop(destination = ('10.0.0.1', '10.0.0.2'))
    def ping(self, device, destination):
        try:
            result = self.parameters[device].ping(destination)

        except Exception as e:
            self.failed('Ping {} from device {} failed with error: {}'.
format(
                                destination,
                                device,
```

```
                                str(e),
                            ),
                        goto = ['exit'])
        else:
            match = re.search(r'Success rate is (?P<rate>\d+) percent',
result)
            success_rate = match.group('rate')
```

It is best practice to reference the testbed file at the command line during runtime:

```
$ python chapter16_12_pyats_2.py --testbed chapter16_pyats_testbed_2.yml
```

The output is similar to our first example, with the additions of `STEPS Report` and `Detailed Results` with each test case.

```
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: |                            Starting
common setup                               |
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: |                       Starting
subsection check_topology                       |
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: The result of subsection check_topology
is => PASSED
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: |                  Starting subsection
establish_connections                    |
2022-09-25T17:14:13: %AETEST-INFO: +------------------------------------
--------------------------------------+
2022-09-25T17:14:13: %AETEST-INFO: +....................................
....................................+
2022-09-25T17:14:13: %AETEST-INFO: :                  Starting STEP 1:
Connecting to lax-edg-r1                    :
2022-09-25T17:14:13: %AETEST-INFO: +....................................
....................................+
```

```
2022-09-25T17:14:13: %UNICON-WARNING: device's os is not provided, unicon
may not use correct plugins
```

The output also indicates the log filename that is written to the `/tmp` directory:

```
$ ls /tmp/lax*
/tmp/lax-edg-r1-cli-20220925T170012042.log
/tmp/lax-edg-r1-cli-20220925T170030754.log
/tmp/lax-edg-r1-cli-20220925T170308615.log
/tmp/lax-edg-r1-cli-20220925T171145090.log
/tmp/lax-edg-r1-cli-20220925T171413444.log
$ head -20 /tmp/lax-edg-r1-cli-20220925T170012042.log

2022-09-25 17:00:12,043: %UNICON-INFO: +++ lax-edg-r1 logfile /tmp/lax-
edg-r1-cli-20220925T170012042.log +++

2022-09-25 17:00:12,043: %UNICON-INFO: +++ Unicon plugin generic (unicon.
plugins.generic) +++

******************************************************************************
* IOSv is strictly limited to use for evaluation, demonstration and IOS      *
* education. IOSv is provided as-is and is not supported by Cisco's          *
* Technical Advisory Center. Any use or disclosure, in whole or in part,     *
* of the IOSv Software or Documentation to any third party for any           *
* purposes is expressly prohibited except as otherwise authorized by         *
* Cisco in writing.                                                          *
******************************************************************************

2022-09-25 17:00:12,705: %UNICON-INFO: +++ connection to spawn: ssh -l
cisco 192.168.2.51, id: 140482828326976 +++

2022-09-25 17:00:12,706: %UNICON-INFO: connection to lax-edg-r1
cisco@192.168.2.51's password:

******************************************************************************
```

The pyATS framework is a great framework for automated testing. However, because of its origin, the support for vendors outside of Cisco is a bit lacking.

One open source tool for network validation to take note of is Batfish, `https://github.com/batfish/batfish`, from the folks at Intentionet. A primary use case for Batfish is to validate configuration changes before deployment. Another open source project is Suzieq (`https://suzieq.readthedocs.io/en/latest/`). Suzieq is the first open source, multi-vendor network observability platform application.

There is a bit of a learning curve involved with `pyATS`; it basically has its own way of performing tests that takes some getting used to. Understandably, it is also heavily focused on Cisco platforms in its current iteration. The `pyATS` core is closed sourced and released in a binary form. The packages that are developed to be used with `pyATS`, such as the parser libraries, YANG connectors, and various plugins are open sourced. For the open source portion, we are all encouraged to make contributions if we would like to add additional vendor support or make syntax or process changes.

We are near the end of the chapter, so let's go over what we have done in this chapter.

Summary

In this chapter, we looked at TDD and how it can be applied to network engineering. We started with an overview of TDD; then, we looked at examples using the `unittest` and `pytest` Python modules. Python and simple Linux command-line tools can be used to construct tests for network reachability, configuration, and security.

pyATS is a tool that Cisco released. It is a network-centric automated testing framework that we can leverage.

Simply put, if it is not tested, it is not trusted. Everything in our network should be programmatically tested as much as possible. As with many software concepts, TDD is a never-ending service wheel. We strive to have as much test coverage as possible, but even at 100% test coverage, we can always find new ways and test cases to implement. This is especially true in networking, where the network is often the internet, and 100% test coverage of the internet is impossible.

We are at the end of the book. I hope you have found the book as much a joy to read as it was a joy for me to write. I want to say a sincere "thank you" for taking the time to read this book. I wish you success and happiness on your Python network journey!

Join our book community

To join our community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/networkautomationcommunity`





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.

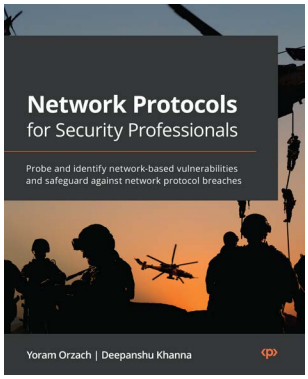
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning 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

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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

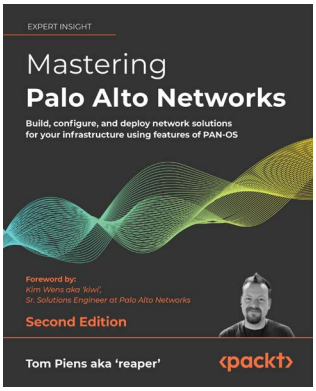


Network Protocols for Security Professionals

Yoram Orzach

ISBN: 9781789953480

- Understand security breaches, weaknesses, and protection techniques
- Attack and defend wired as well as wireless networks
- Discover how to attack and defend LAN-, IP-, and TCP/UDP-based vulnerabilities
- Focus on encryption, authorization, and authentication principles
- Gain insights into implementing security protocols the right way
- Use tools and scripts to perform attacks on network devices
- Wield Python, PyShark, and other scripting tools for packet analysis
- Identify attacks on web servers to secure web and email services

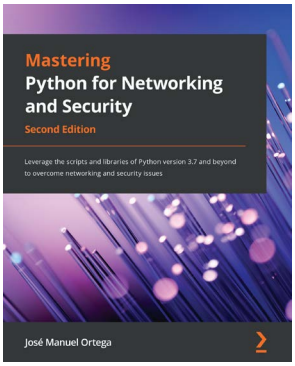


Mastering Palo Alto Networks – Second Edition

Tom Piens

ISBN: 9781803241418

- Explore your way around the web interface and command line
- Discover the core technologies and see how to maximize your potential in your network
- Identify best practices and important considerations when configuring a security policy
- Connect to a freshly booted appliance or VM via a web interface or command-line interface
- Get your firewall up and running with a rudimentary but rigid configuration
- Gain insight into encrypted sessions by setting up SSL decryption
- Troubleshoot common issues, and deep-dive into flow analytics
- Configure the GlobalProtect VPN for remote workers as well as site-to-site VPN



Mastering Python for Networking and Security – Second Edition

José Manuel Ortega

ISBN: 9781839217166

- Create scripts in Python to automate security and pen testing tasks
- Explore Python programming tools that are used in network security processes
- Automate tasks such as analyzing and extracting information from servers
- Understand how to detect server vulnerabilities and analyze security modules
- Discover ways to connect to and get information from the Tor network
- Focus on how to extract information with Python forensics tools

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Mastering Python Networking, Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

access lists 198, 199
implementing, with Ansible 199, 201

Advanced Research Projects Agency Network (ARPANET) 10

Amazon Elastic Compute Cloud (EC2)
reference link 353

Amazon GuardDuty
reference link 373

Amazon Resource Names (ARNs)
reference link 353

Amazon VPC
reference link 353

Amazon Web Services (AWS)
setup 340-342
URL 339

Ansible 112-118
access lists, implementing 199-203
advantages 125, 199
architecture 119, 120
control node installation 120, 121
example 119
lab topology 121, 122
MAC access lists 203, 204
reference link, for versioning split 119
versions 118

Ansible Content Collections 127, 128
reference link 127

Ansible, features
agentless 125, 126
extensible 127
idempotence 126
simplicity 127

Ansible loops 137
looping, over dictionaries 139-141
standard loops 137-139

Ansible network examples 128
Ansible conditionals 131-133
Ansible loops 137
Ansible network facts 135-137
configuration changes 133-135
inventory nesting 129-131
templates 142, 144

Ansible testing 535
reference link 536

API structured output
versus screen scraping 72-75

application and network load balancer, comparison
reference link 372

Application Gateway
reference link 414

Application Load Balancer 372

application program interface (API) 33

Arista eAPI management 101
examples 104, 105
preparation 101-104

Arista Networks
reference link 100

Arista Pyeapi library
examples 106-110

installation 106, 107
Arista Python API 100, 101
Arista vEOS
reference link 42
asynchronous operations 309-312
overview 322
Python multiprocessing 323, 324
Python multithreading 324-326
Availability Zones (AZs) 345, 388
AWS CLI 342-344
installation link 342
AWS CloudFormation
reference link 360
AWS CloudFront 352
AWS console 341
URL 341
AWS edge locations 352
AWS network
overview 345-353
AWS network services 373
AWS OpenSearch
reference link 429
AWS Python SDK 345
AWS Shield 374
reference link 374
AWS transit centers 352
AWS Transit VPC
reference link 373
AWS WAF 373
reference link 373
Azure
global infrastructure 388, 389
reference link, for latest offerings 378
setting up 378-380
Azure Administration 381
Azure, and AWS
network service comparison 376, 377
Azure APIs 381
Azure Cloud Shell 381-383
Azure ExpressRoute 412
advantages and disadvantages 413
reference link 414
Azure, for Python Developers page
reference link 384
Azure network load balancers 414, 415
Azure network services
container networking 416
Distributed Denial of Service Protection 416
DNS services 416
VNet TAP 416
Azure RBAC
reference link 386
Azure regions 389
Azure SDK, for Python page
reference link 384
Azure service principal 384-387
Azure virtual networks (VNets) 389-392
internet access 392-396
network resource creation 396-398
network security groups 407-409
peering 399-402
routing 402-406
service endpoints 398, 399
Azure VPN devices
reference link 412
Azure VPN gateways
reference link 409
Azure VPNs 409-412

B

Beats 421
using, for data ingestion 440-446
Border Gateway Protocol (BGP) 369
Boto3 VPC API documentation
reference link 358
branch 463
Bring Your Own Device (BYOD) 196
built-in types, of Python language
mapping 23, 24
None type 20
numeric 20
sequence 20-23
set 24, 25

C

Cacti 238
installation 238-240
Python script as input source 240-242
reference link 238, 239
Canvas
reference link 458
Cerberus
reference link 534, 535
change-advisory board (CAB)
implementing, for networks 492
checkout 463
Cisco ACI examples 85-88
Cisco IOS-XE scripts 89
Cisco API examples 77
Cisco ACI examples 85-89
Cisco NX-API 78
Cisco YANG Model 84
Cisco Certified Internetwork Expert (CCIE) 37
Cisco DevNet 40
reference link 37
Cisco DevNet Sandbox
reference link 41
Cisco Meraki controller 89-91
Cisco Modeling Labs (CML) 37
advantages 38
reference link 37
tips 39, 40
Cisco NX-API 78
examples 79-84
lab preparation 79
Cisco SNMP Object Navigator
reference link 218, 219
Cisco YANG Model 84
classic attacks
reference link 197
client-server model 10
cloud computing 339
cloud computing service models
references 339
cloud data centers 6, 7
CloudFormation 360-364
CloudFront 373
CloudFront CDN services 373
command line interface (CLI) 342
challenges 34
commit 463
configuration backup
automating 485, 486
container-based world
challenges 170
Containerlab 171-175
reference link 42

container networking 166, 167
container host network 168
custom bridge network 169
options 170
reference link 416
containers
Flask, running 315-319
content delivery network (CDN) 373
content management
consideration 460
Continuous Integration (CI) 494
workflow 494
control node 119
control node installation, Ansible 120, 121

D

data centers 5
cloud data centers 6, 7
edge data centers 7, 8
enterprise data centers 5
data ingestion
with Beats 440-446
with Logstash 437, 438, 439
data input methods
reference link 240
data modeling
for IaC 76
data visualization
Python 225
with Kibana 452-458
Dense Wavelength Division Multiplexing (DWDM) 4
device ID API 305, 306
devices API 303-305
dictionary 23
direct connect 370, 371
Distributed Denial of Service (DDoS) 374
Distributed Denial of Service Protection
reference link 416
Django
URL 287
Django database documentation
reference link 287
DNS services
reference link 416
Docker
advantages 153, 154
application, building 157-161
commands 155
hello world, building 156
installing 154, 155
overview 153
Python applications, building 154
using, with Kubernetes 175
Docker CLI references
URL 155
Docker Community Edition (Docker-CE) 153
Docker-compose
container orchestration 163-166
reference link 163
Docker container environment, components
building and development 153
Container Orchestration 153
Docker Engine 153
Docker Desktop 153
installation link 154
Docker Engine 153
Docker Enterprise Edition (Docker-EE) 153
Dockerfile 157
reference link 157, 162

Docker Hub 153
Docker images
sharing 161-163
Domain Name System (DNS) 13
DOT
reference link 247

E

edge data centers 7, 8
Elastic Cloud
reference link 428
Elastic Compute Cloud (EC2) service 340
Elastic IP (EIP) 367, 368
characteristics 367
Elastic Load Balancing (ELB) 372
Elasticsearch
reference link 447-452
using, with Python client 435, 436
Elastic Stack 420, 421
reference link 420
using, as service 428, 429
Emulated Virtual Environment Next Generation (Eve-NG)
reference link 42
end-to-end example 430-435
enterprise data centers 5
Equinix Cloud Exchange Fabric
reference link 371
Equinix FABRIC
reference link 414
extensible markup language (XML) 91

F

fetch 463
Filebeat 440
Flask 287, 289
examples 289, 290
HTTPie client 290-292
jsonify return 296, 297
reference link 289
running, in containers 315-319
setup 288, 289
URL 287
URL generation 295, 296
URL routing 293, 294
URL variables 294, 295
versions 289
Flask API endpoint 319
Flask-Login
reference link 315
Flask's httpauth
reference link 312
Flask-SQLAlchemy 297, 299
reference link 297
floor division 25
flow-based monitoring 263
IPFIX 263
NetFlow 263
sFlow 263
functional test 518

G

Genie 536- 544
GitHub 463
collaborating, with Pull requests 479-482
example 473-478

gitignore file 465
reference link 465
GitLab
example 501-509
installing 495-499
GitLab network
example 509-512
GitLab runner 499-501
GitPython library
reference link 482
Global Information Tracker (Git) 461, 463
benefits 461
branch 471, 473
collaborating 487
reference link 463
setting up 464
terminology 462, 463
usage examples 466- 470
using, with Python packages 482
Global Interpreter Lock (GIL) 324
Global Server Load Balancing 372
GNS3 42
Graphviz 246, 247
examples 249-251
installation 248
lab setup 247, 248
LLDP neighbor graphing 252-255
Python wrapper 251, 252
Grok
reference link 438

H

hosts 4
HTTPBin
URL 291
HTTPie
URL 290
HTTPie client 290- 292

I

IAM service
reference link 353
idempotence
advantages 126
reference link 126
Infrastructure-as-a-Service (IaaS) 339, 376
Infrastructure-as-Code (IaC) 70, 71
data modeling 76
intent-driven networking 71
screen scraping, API structured output 72-75
INI format
reference link 122
integration test 517
intent-based networking (IBN) 71
intent-driven networking (IDN) 71
Intermediate Distribution Frame (IDF) 5
International Organization for Standardization (ISO) 9
internet
cloud data centers 6
data centers 5
hosts components 4
network components 4
overview 3
servers components 4
Internet Assigned Numbers Authority (IANA) 12
Internet Control Message Protocol (ICMP) 188

Internet of Things (IoT) 4

Internet Protocol (IP) 4, 14, 15
network address translation (NAT) 15
network security 15
routing concepts 15

Internet Service Provider (ISP) 3

Internetwork Operating System (IOS) 35

IPFIX 263

J

Jinja
reference link 142

Jinja template
conditional statement 147, 148
loops 146
variables 145, 146

jsonify return 296, 297

Juniper NETCONF
device preparation 92
examples 93, 94, 96

Juniper Networks
Python API 91

Juniper PyEZ
examples 98, 99
for developers 96
installation and preparation 96-98

Juniper vMX
reference link 42

JunOS Olive 92

K

Kibana
using, in data visualization 452-458

Kubernetes
Docker, using with 175
reference link 153, 175

L

lab setup 178-185, 288, 289

lab topology, Ansible 121, 122
playbook 123-125
variable files 122

large data centers 101

lightweight data shippers
reference link 440

line graph
reference link 233

Link Layer Discovery Protocol (LLDP) 247

Linux containers (LXC) 152

Linux, post-installation
reference link 154

LLDP neighbor graphing 252-255
information retrieval 255, 256
playbook, testing 261, 262
Python parser script 256-261

Local Area Network (LAN) 3

Logstash
reference link 440
using, for data ingestion 437-439

looping, over dictionaries 139-142

M

Main Distribution Frame (MDF) 5

managed node 120

management host IP 318

management information base (MIB) 216

mapping 23, 24

Matplotlib 225
example 226-228

installation 225
resources 232
URL 225
used, for graphing SNMP results 228- 232
Matplotlib 3.6.2 documentation
reference link 233
matplotlib.dates
reference link 228
merge action 463
Molecule
reference link 536
MRTG Done Right 238
Multi-Protocol Label Switching (MPLS) 371
Multi Router Traffic Grapher (MRTG)
reference link 238

N

NAT gateways 368, 369
NETCONF 77
NetFlow 263
parsing, with Python 264, 265
Netmiko library 62, 64
reference link 112
network
Ansible, testing 535
configuration, testing 535
latency, testing 533, 534
reachability, testing 532, 533
security, testing 534
tests, writing 531
transactions, testing 534
network access control lists (ACLs)
versus security groups 366
Network Address Translation (NAT) 339
network automation 116
reference link 120
network content API 299- 303
network dynamic operations 306-309
asynchronous operations 309-312
network engineering field
containers 170
network lab topology 421-428
Network Load Balancer 372
network operating system (NOS) 171
network protocol suites 11
Internet Protocol (IP) 14, 15
Transmission Control Protocol (TCP) 12
User Datagram Protocol (UDP) 13
network resource API 297
device ID API 305, 306
devices API 303, 304, 305
Flask-SQLAlchemy 297, 299
network content API 299-303
network scaling services 371
CloudFront CDN services 373
Elastic Load Balancing (ELB) 372
Route 53 DNS service 372, 373
network security 15
Network Security Group (NSG) 404-409
Nokia SR-Linux
reference link 42
None type 20
Nornir framework 64, 65, 66
reference link 64
Novells IPX/SPX protocol 11
ntop
Python extension 272-276
traffic monitoring 269-272
numeric 20

NumPy
reference link 225

O

object identifier (OID) 216

object-oriented programming (OOP) 29

OpenSearch product
reference link 429

Open Systems Interconnection (OSI) model 2, 8-10

organizationally unique identifier (OUI) 203

OS-level virtualization 151

P

P2S VPN connection, Azure
reference link 412

Personal Access Token (PAT) 476

physical devices
advantages and disadvantages 36

Platform-as-a-Service (PaaS) 339, 376

playbook 123
executing 124

port address translation (PAT) 368

private VLAN
Community (C) port 209
Isolated (I) port 209
Promiscuous (P) port 209

pull 463

Pull Requests
collaborating with 479-482

pxssh subclass 53

pyATS 536-544
reference link 536

Pygal 233
example 233, 235
installation 233
resources 237
URL 233
used, for graphing SNMP results 235-237

PyGitHub library 483, 485
reference link 483

PySNMP 219-225

pytest documentation
reference link 528

pytest examples 528-531

Python 285
for Cacti 238
for data visualization 225
socket module 265-268
struct module 265-268
testing 528
Uncomplicated Firewall (UFW), using with 209, 210
unittest module 524-527
versus PowerShell 387

Python asyncio module 326-331

Python language
built-in types 19
classes 29, 30
control flow tools 26-28
functions 28, 29
modules and packages 30, 31
operating system 17
operators 25, 26
overview 16, 17
program, running 18, 19
versions 17

Python libraries
Matplotlib 225
Pygal 233

Python multiprocessing 323, 324
Python multithreading 324-326
Python Paramiko library 54
examples 61, 62
features 59
installation 55
overview 55, 57
program 58
servers 59, 60
Python Paramiko library, limitations 66
bad automation 67
idempotent network device interaction 67
Python pexpect library 44
example 53, 54
features 51, 52
installation 44
limitations 66
overview 45-50
program 50, 51
reference link 44
SSH connection 53
Python script
as input source 240, 242
Python virtual environment
reference link 43
Python web frameworks
comparing 285, 286, 287
reference link 285

R

ref 462
regular expression module
searching with 206-208
virtual local area networks (VLANs) 208, 209
remote procedure calls (RPCs) 91
repository 462
Requests
URL 79
round-robin database tool (RRDtool) 238
reference link 238
Route 53 DNS service 372, 373

S

S2S connection, Azure
reference link 412
Scalable Vector Graphics (SVG) 233
Scapy
advantage 185
attacks 197
installing 186, 187
interactive examples 188-190
ping collection 196, 197
reference link 185
resources 198
TCP port scan 192-195
used, for packet captures 190, 191
Scrapli project 331
async example 334-337
example 331- 334
reference link 112
screen scraping
versus API structured output 72-75
secure shell (SSH) 52
security groups 365
versus network ACLs 366
sequence 20-23
servers 4
service-level agreement (SLA) 370
set 24, 25
setup and teardown 518
sFlow 263, 276

sFlow-RT
using, with Python 277-281
SFlowtool
using, with Python 277- 281
Simple Network Management Protocol (SNMP) 215-217
PySNMP 219-225
setup 217-219
SNMP results
graphing, with Matplotlib 228-232
graphing, with Pygal 235-237
Software-as-a-Service (SaaS) 339, 376
Software-Defined Networking (SDN) 1, 339
Software-Defined Wide-Area-Networks (SD-WANs) 8
SQLAlchemy 297
SR Linux (Service Router Linux) 171
standard loops 137-139
Suzieq
reference link 544
Syslog search 205, 206
system test 518

T

tag 463
TCP/IP guide
reference link 13
Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T) 9
templates, Ansible 142, 144
Jinja template conditional 147-149
Jinja template loops 146
Jinja template variables 145, 146
ternary content-addressable memory (TCAM) 198
test coverage 518
test-driven development (TDD)
overview 516, 517
steps 517
terms, defining 517
test fixtures 518
tests
writing, for network 531
Timelion
reference link 458
Tiny Core Linux
reference link 215
Tool Command Language (TCL) 44
topology, as code 518-521
pytest example 528-531
Python's unittest module 524-527
Python testing 528
XML parsing example 521-524
traditional change management process 492
challenges 493
Transmission Control Protocol (TCP) 2, 12
functions and characteristics 12
messages and data transfer 12

U

Ubuntu Server 22.04 LTS 393
Uncomplicated Firewall (UFW)
using, with Python 209, 210
unit testing Ansible modules
reference link 536
unit testing framework
reference link 527

unit tests 517
reference link 536
URL generation 295, 296
URL routing 293, 294
URL variables 294, 295
user authentication 312-315
user authorization 312-315
User Datagram Protocol (UDP) 2, 13
reference link 14

V

virtual devices 36, 37
advantages and disadvantages 36
virtual environment 288
virtual Ethernet (veth) interface 167
Virtual Internet Routing Lab (VIRL)
reference link 37
Virtual lab
Cisco DevNet 40
Cisco Modeling Labs (CML) 37, 38
constructing 35
GNS3 42
physical devices 36
virtual devices 36, 37
virtual local area networks (VLANs) 208
VirtualNetworkServiceEndpoint 405
virtual network (VNet) 375
virtual private cloud (VPC) 352-358
automation, with CloudFormation 360-364
Elastic IP (EIP) 367, 368
NAT gateways 368, 369
network ACLs 364-366
route tables 358-360
route targets 358-360
security groups 364, 365, 366
virtual private gateway (VPG) 370
VNetGlobalPeering 405
VNet TAP
reference link 416
VNet traffic routing document
reference link 405
VPC peering
reference link 364
VPN gateways 369
VyOS 111, 112
download link 111
reference link 111

X

XML parsing example 521-524

Y

Yet Another Next Generation (YANG) 77

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781803234618`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly