EXPERT INSIGHT

Mastering Python Networking

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

Foreword by:
Michael Kennedy
Founder, Talk Python

Third Edition

Eric Chou

Packt>

Mastering Python Networking

Third Edition

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

Eric Chou

BIRMINGHAM - MUMBAI

Mastering Python Networking

Third Edition

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

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

Production reference: 1280120

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

ISBN 978-1-83921-467-7

www.packt.com

For my wife Joanna and children, Mikaelyn and Esmie.
For my parents, who lit up the fire in me many years ago.

packt.com

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

Why subscribe?

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

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

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

Foreword

This book you are holding in your hand or are reading on your screen has a power that can be yours if you take time to study it. Much like Thor's hammer or Iron Man's suit, programming is a superpower that amplifies your existing knowledge and skill set.

Many people feel, or are told, that 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 Python in the context of network configuration, administration, monitoring, and more.

If you are tired of logging in and typing a bunch of commands to configure your network, Python is for you. If you need to be certain that 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.

First, as Eric will demonstrate throughout this book, there are many Python libraries (sometimes called packages) designed specifically on network engineering. A focused search over at `https://pypi.org` for the network topic turns up over 500 different libraries for network automation and monitoring. With libraries such as Ansible, you can create complex network and server configurations declaratively 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.

Second, 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.com`.

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, these languages are not beginner friendly. You have to learn about pointers, compilers, linkers, headers, classes, accessibility (public/private), and so on just to get started.

Python lives in both realms. On one hand it is incredibly easy to be productive with just a few lines of code and simple programming concepts. On the other, it is increasingly the language of choice for some of the world's most significant software; such as that behind YouTube, Instagram, Reddit, and others. Microsoft chose Python as their language to implement the **command line interface** (**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 language. Additionally, Python has many highly polished libraries for working with networks in many facets. This book, *Mastering Python Networking*, *Third Edition*, combines all of these and will change the way you think about networking. Enjoy the journey.

Michael Kennedy
Portland, OR
Founder, Talk Python

Introduction

In 2014, I taught the first Coding 101 workshop on Python and REST APIs for network engineers in the DevNet Zone at Cisco Live. The room was full of prominent network engineers and architects, and many of them made their first API call in this workshop. Since then, I have been honored to work with network engineers from around the world who have decided to add coding to their skill set.

IT and Operations teams are changing. I believe the new normal will be network engineers and software developers working side by side in the same team. The scalability, complexity, and security that modern application deployments demand from the network require automation to make network management repeatable, reliable, and agile at scale.

Network engineers are problem solvers with deep expertise. Adding Python, network automation, and API skills to the network engineering tool set creates a powerful combination. With these added layers, engineers can approach problems in new ways and tackle new types of challenges. *Mastering Python Networking, Third Edition* is a valuable resource both for network engineers who want to learn coding skills and for software engineers who want to take advantage of new programmable infrastructure opportunities.

One of the questions that I often hear from engineers is, "Where do I start?" My advice is: start simple. Look for challenges that your team faces, which are "read-only," and focus on using automation to troubleshoot and gather information. You can then use automation to flow the information that is gathered into ticketing systems or chat applications, and soon you have started to build a workflow. This safe-start, read-only evolution helps teams build confidence with automation and familiarity with the tools.

At the beginning, focus on learning the core coding skills that will help you in every project including Python coding and RESTful APIs and on using tools like Git and GitHub to manage your source code and collaborate with others. Take the time to set up your development environment. Try different code editors and tools such as Postman and curl for exploring APIs. Build a solid understanding of how to work with JSON and XML. Start exploring software development methodologies such as **test-driven development** (**TDD**), and the core principles of DevOps.

Mastering Python Networking, *Third Edition* is a great resource for working on these skills because it helps you learn these topics within the context of networking. The book starts with using Python for basic network device interactions using CLI and API, then moves up the stack with a general-purpose automation framework – all from the perspective of network engineers. Along the way, Eric provides Python examples on network security, monitoring, and constructing your own API with the Flask framework. Eric also introduces networking in the cloud with both AWS and Azure, as well as with common DevOps tools such as Git, Jenkins, and TDD. Throughout the book, Eric explains topics using a pragmatic, practitioner-based approach that helps you bring these concepts into your work.

DevOps and Cloud are transforming our industry. Development, operations, security, and networking teams are connecting in new ways with shared goals and responsibilities to deliver business outcomes. Network engineers who learn software skills will help define this transformation.

So, find your first project and work on learning these skills in the scope of that project. Pick something you do every day that you want to repeat reliably and try to automate it. Get hands on and write code early and often. Build or find a development lab where you can work. The more you can experiment, the faster you will learn.

The innovations that are going to be created over the next five years by network engineers and software engineers working together are going to be game-changing.

Take your first step, and don't forget to celebrate when you get your first `200 OK` successful API response.

Happy Coding!

Mandy Whaley
Senior Director of Developer Experience, Cisco DevNet

Contributors

About the author

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

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

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

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

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

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

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

About the reviewer

Rickard Körkkö is a NetDevOps consultant at SDNit (`https://sdnit.se`), where he's part of a group of experienced technical specialists with a great interest in, and focus on, emerging network technologies.

He's a self-taught programmer with a primary focus on Python. His daily work includes working with orchestration tools such as Ansible to manage network devices.

He has also served as a technical reviewer for the book *A Practical Guide to Linux Commands, Editors, and Shell Programming, Third Edition,* by Mark G. Sobell.

Table of Contents

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 no doubt experiencing a similar time with the rapid changes in the field of network engineering. As software development becomes more integrated into all aspects of networking, the traditional command line interface and vertically integrated, network stack 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 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 and second editions of this book, I have been able to have interesting and meaningful conversations with many of the readers of the book. I am humbled by the success of the first two editions and took to heart the feedback I was given. In this third edition, I tried to incorporate many of the newer libraries, update existing examples with the latest software and newer hardware platforms, and added two more chapters that I think are important for today's network engineers.

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 on 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 that make up internet communication today, from the OSI and client-server model to the TCP, UDP, and IP protocol 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 for the examples.

Chapter 3, *APIs and Intent-Driven Networking*, discusses the newer 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 Basics*, discusses the basics of Ansible, an open source, Python-based automation framework. Ansible 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 Cisco, Juniper, and Arista devices.

Chapter 5, *The Python Automation Framework – Beyond Basics*, builds on the knowledge in the previous chapter and covers the more advanced Ansible topics. We will cover conditionals, loops, templates, variables, Ansible Vault, and roles. It will also cover the basics of writing custom modules.

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. An overview of Elasticsearch and how it can be used for network monitoring will also be covered.

Chapter 9, *Building Network Web Services with Python*, shows you how to use the Python Flask web framework to create our own API 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, *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 11, *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 12, *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 13, *Working with Git*, is where we will illustrate 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 14, *Continuous Integration with Jenkins*, uses Jenkins to automatically create operations pipelines that can save us time and increase reliability.

Chapter 15, *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. We will also see how we can integrate the tests into continuous integration tools, such as Jenkins.

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, with the exceptions of chapters 4 and 5, which should be read in sequence. 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 own network lab.

Download the example code files

You can download the example code files for this book from your account at `http://www.packt.com`. If you purchased this book elsewhere, you can visit `http://www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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://static.packt-cdn.com/downloads/9781839214677_ColorImages.pdf`.

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:

```
$ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also 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.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@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 would report this to us. Please visit, `http://www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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@packt.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`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Review of TCP/IP Protocol Suite and Python

Welcome to the new exciting age of network engineering! When I started working as a network engineer at the turn of the millennium, 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:

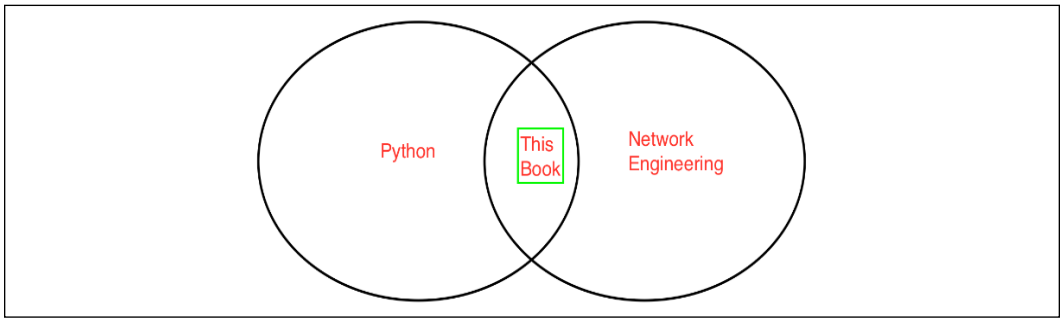


Figure 1: The intersection between Python and network engineering

There are many books that have already been 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 that you have some hands-on experience of managing networks, as well as a basic understanding of network protocols. It's helpful if you're already familiar with Python as a language, but we will cover some basics later in the chapter. You do not need to be an expert in Python or network engineering to get the most out of this book. This book intends to build on the basic foundations of network engineering and Python to help readers to learn and practice various applications that can make their lives easier.

In this chapter, we will do a general review some of the 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 Python tutorials 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. 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.

We will also look at a high-level review of the Python language; just enough for those readers who do not code in Python on a daily basis 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 do check out the references for further information if required.

As network engineers, we are typically challenged by the scale and complexity of the network 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 that span the globe. The biggest 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 your background. The internet means different things to different people; the young, the old, students, teachers, business people, 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 are able to do this by aggregating small networks into bigger networks that they maintain. Your ISP network often consists of edge nodes that aggregate the traffic to their core network. The core network's function is to interconnect these edge networks via a higher speed network.

At special edge nodes, 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 of these networks back to your device, while the source and destination remain the same.

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

Servers, hosts, and network components

Hosts are end nodes on the network that communicate to 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 that we use in agriculture, farming, automobiles, and more. With the explosion of the number of hosts connected to the internet, all of them 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 so on. 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 special supernodes on the network that provide additional capabilities to their peers. We will look at servers later on in 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 OSI model that we will look at in a bit, these network components are layer one to three devices that sometimes venture into layer four as well. They are layer two and 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.

Enterprise data centers

In a typical enterprise, the company generally has the business needs for internal tools such as emailing, document storage, sales tracking, ordering, HR tools, and a knowledge-sharing intranet. These services translate into file and mail servers, database servers, and web servers. Unlike user computers, these are generally high-end computers that require a lot of power, cooling, and network connections. A byproduct of the hardware is also the amount of noise they make, which is not suitable for a normal work space. 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 a location closer to the user, which is 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.

Enterprise data centers generally follow the network design of three layers. These layers are access layers, distribution layers, and a 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 really 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, Microsoft, and Google without the user having to directly manage the resources.

Because of the number of servers they need to house, the 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 provider data centers for many years, every time I visit a cloud provider data center, I am still amazed at the scale of them. Just to give examples of the sheer scale of them, 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, have built their 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 of the 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, consists of 1.2 million square feet of facility on 200 acres of land and required the city to spend an estimated $ 65 million in public infrastructure upgrades.

Figure 2: Utah data center (source: https://en.wikipedia.org/wiki/Utah_Data_Center)

At the cloud provider scale, the services that 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 different racks, to provide redundancy and flexibility for service owners.

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

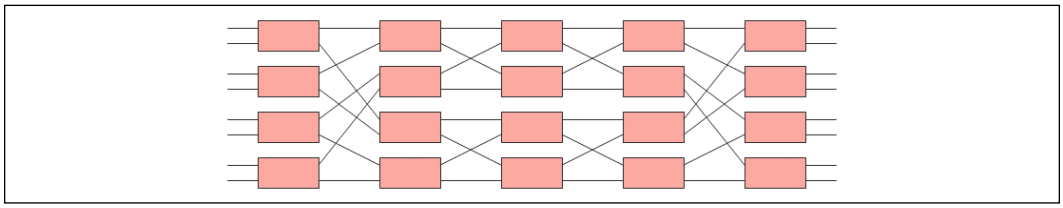


Figure 3: Clos network

In a way, cloud data centers are where network automation becomes a necessity for speed, flexibility, and reliability. If we follow the traditional way of managing network devices via a Terminal and command line interface, the number of engineering hours required would not allow the service to be available in a reasonable amount of time. This is 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 configuration to accommodate rapidly changing business needs.

Personally, cloud data centers are where I started my path of network automation with Python a number of 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 anywhere else but at these data centers? All the connections from clients around the world can be routed back to the data center servers providing the service, and we can call it a day, right? The answer is, of course, it depends on the use case. The biggest 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, large 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 has to traverse through multiple networks, sometimes through an undersea cable, slow satellite links, 3G or 4G cellular links, or Wi-Fi connections.

What if we need to reduce the network latency? One solution would be to reduce the number of networks the end user requests traverse through. Be as closely connected to the end user as possible at the edge where the user enters your network and place enough resources at the edge location 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 service. In order 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 I can connect to, to reduce the hop count. All the connections would be with as much bandwidth as needed to decrease latency during peak hours. 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.

We will cover security (*Chapter 6*, *Network Security with Python*) and visibility (*Chapter 7*, *Network Monitoring with Python – Part 1*, and *Chapter 8*, *Network Monitoring with Python – Part 2*) in later chapters of this book. As with many complex subjects that break the complexity by dividing the subject into smaller, digestive pieces, networking is based on the concept of layers. Over the years, there are different networking models that have been developed. We will take a 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 conceptional 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 types of 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 mutiple back-and-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 4: 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 period as the OSI model development, the internet was taking shape. The reference model the original designer used 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 is the model 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:

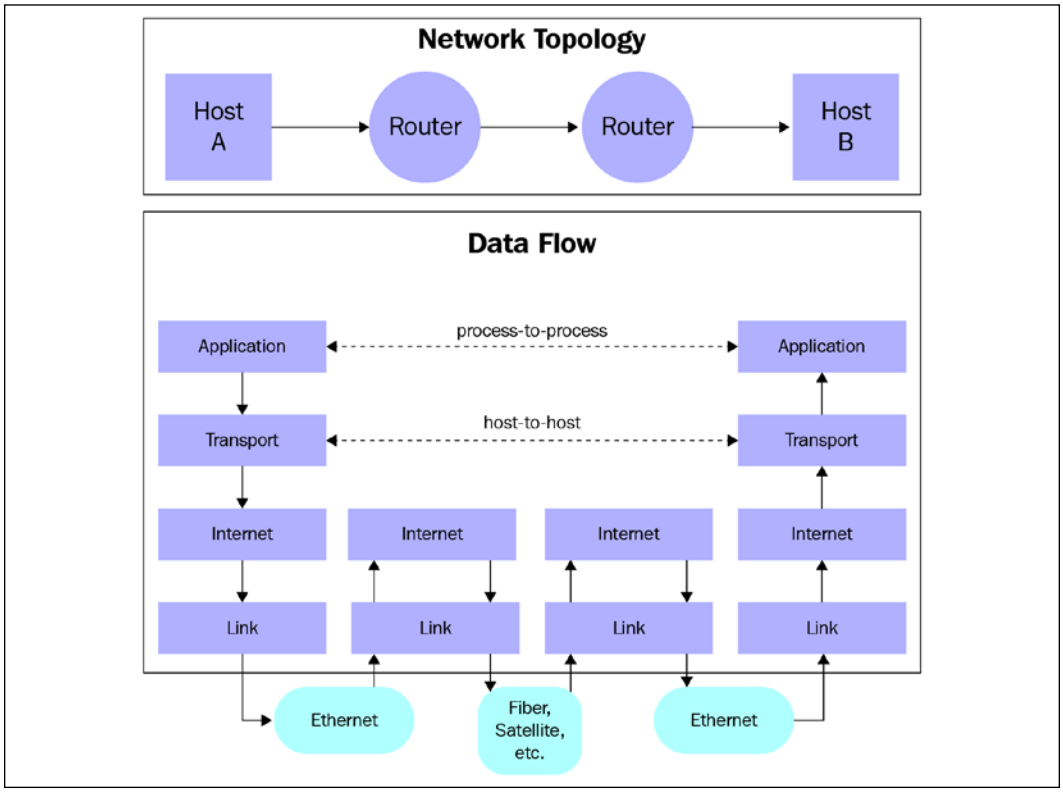


Figure 5: Internet protocol suite

Both the OSI and TCP/IP models are useful for providing standards for end-to-end data communication. However, for the most part, we will refer to the TCP/IP model more in this book, since that is what the internet was originally built on. We will refer to the OSI 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 take a look at the client-server model in the next section.

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 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 really 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 networks 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 who 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 an internet translation device, such as a router, is used to translate from one protocol to another. For example, in order 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 take a look at each of the protocol suites.

The transmission control protocol

The TCP is one of the main protocols used on 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 others, source and destination ports, a sequence number, an acknowledgment number, control flags, and a checksum:

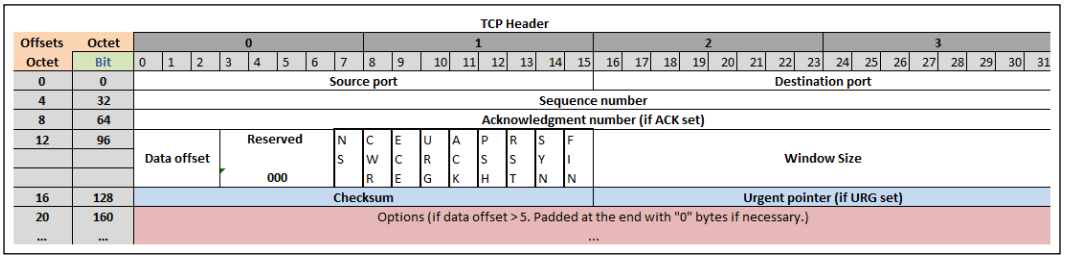


Figure 6: TCP header

Functions and characteristics of TCP

TCP uses datagram sockets or ports to establish a host-to-host communication. The standard body, called **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 by the socket that represents the local endpoint for connection.

The protocol operation consists of a state machine, where 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`.

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 internet protocol suite. 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 extra 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:

UDP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port														
4	32	Length																Checksum														

Figure 7: 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 calling, 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 where the tradeoff between accuracy and latency usually tips to the side of small 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.

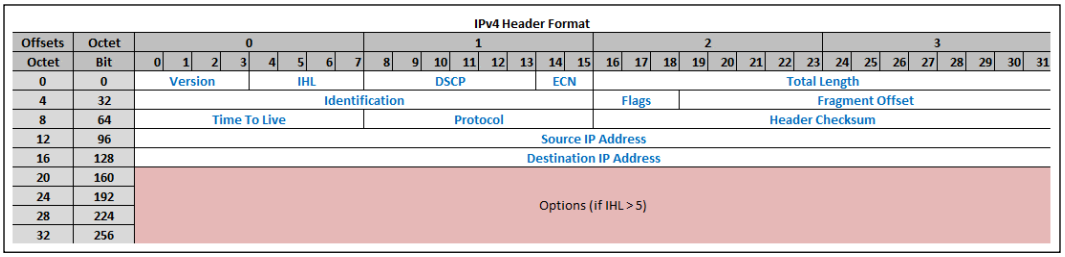
Again, this section does not do justice to the topic of UDP, and the reader is encouraged to explore the topic through various resources if you are interested in learning more about UDP.

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 (`/24`):

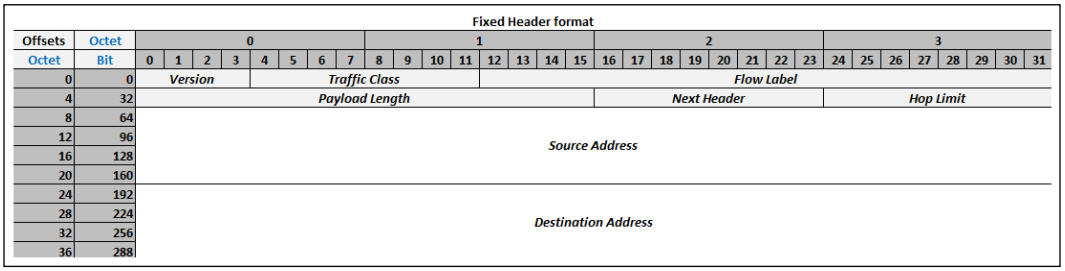


IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Figure 8: IPv4 header

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



Fixed Header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Version*				*Traffic Class*												*Flow Label*															
4	32	*Payload Length*																*Next Header*								*Hop Limit*							
8	64	*Source Address*																															
12	96																																
16	128																																
20	160																																
24	192	*Destination Address*																															
28	224																																
32	256																																
36	288																																

Figure 9: 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. As much as the protocol designer would like to move from IPv4 to IPv6, the internet today is still pretty much addressed with IPv4, with some of the service provider networks addressed with IPv6 internally.

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 to 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 6 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 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 via 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 do 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 taken a look 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, dynamic semantics mentioned previously probably do not mean much to you. But I think we can all agree that rapid application development and simple, easy-to-learn syntax sounds like a good thing. Python, as an interpreted language, means there is little to no compilation process required before execution, so the time to write, test, and edit Python programs is greatly reduced. For simple scripts, if your script fails, a print statement is usually all you need to debug what was going on.

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

The functions, modules, and packages encourages 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 that for common tasks, you need not download any additional packages outside of the Python language itself. In order 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 full production system is very handy 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 in size is invaluable. You might be surprised to see a language that was deemed as a scripting language by many is being used for full production systems by many cutting-edge companies (organizations using Python; `https://wiki.python.org/moin/OrganizationsUsingPython`).

If you have ever worked in an environment where you have to 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 for a bit of a refresher. If you are already familiar with the basics, feel free to quickly scan through it or skip the rest of the chapter.

Python versions

As many readers are already aware, Python has been going through a transition from Python 2 to Python 3 for the last few years. Python 3 was released back in 2008, over 10 years ago, with active development with the most recent release of 3.7. Unfortunately, Python 3 is not backward compatible with Python 2.

At the time of writing the third edition of this book, in late 2019, the Python community has largely moved over to Python 3. In fact, Python 2 will officially be end-of-life beginning January 1st, 2020 (`https://pythonclock.org/`). The latest Python 2.x release, 2.7, was released over six years ago in mid-2010. Fortunately, both versions can coexist on the same machine. Given that Python 2 is end-of-life and not maintained by the time you read this passage, we should all switch to Python 3. More information is given in the next section about invoking the Python interpreter, but here is an example of invoking Python 2 and Python 3 on an Ubuntu Linux machine:

```
$ python2
Python 2.7.15+ (default, Jul 9 2019, 16:51:35)
[GCC 7.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
$ python3.7
Python 3.7.4 (default, Sep 2 2019, 20:47:34)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

With the 2.7 release being end-of-life, most Python frameworks now support Python 3. Python 3 also has lots of good features, such as asynchronous I/O that can be taken advantage of when we need to optimize our code. This book will use Python 3 for its code examples unless otherwise stated. We will also try to point out the Python 2 and Python 3 differences when applicable.

If a particular library or framework is better suited for Python 2, such as Ansible (see the following information), it will be pointed out, and we will use Python 2 instead. We should aim at using Python 3 as the default option and only use Python 2 when it is absolutely necessary.

At the time of writing, Ansible 2.8 and above have support for Python 3. Prior to 2.5, Python 3 support was considered a tech preview. Given the relatively new supportability, many of the community modules are still in the process of migrating to Python 3. For more information on Ansible and Python 3, please see `https://docs.ansible.com/ansible/2.5/dev_guide/developing_python_3.html`.

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 compatibility, such as taking care of the subtle differences between backslashes in Windows filenames and activating a virtual environment on different platforms. Since this book is 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 a Linux Ubuntu 18.04 LTS machine. I will also try my best to make sure the code runs the same on the Windows and the macOS platform.

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

```
$ uname -a

Linux network-dev-2 4.18.0-25-generic #26~18.04.1-Ubuntu SMP Thu Jun 27
07:28:31 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

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 and results are displayed. 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 in use today, CPython. Whenever we mention Python in this book, we are referring 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 `Python` keyword:

```
$ python3.7
Python 3.7.4 (default, Sep 2 2019, 20:47:34)
[GCC 7.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
```

In Python 3, the print statement is a function; therefore, it requires parentheses. In Python 2, you can omit the parentheses.

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 this 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 in 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 over and over again like you have to do in the interactive shell. Python files are just regular text files that are 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 following 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
$
```

Python built-in types

Python implements dynamic-typing, or duck typing, and tries to automatically determine the object's type as you declare them. 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

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. The `None` type is also used in function arguments to error out if the caller does not pass in an actual value.

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` are 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. In this and the next few sections, we will use the interactive interpreter to illustrate the different types.

Please feel free to type along on your own computer.

Sometimes, it surprises people that `string` is actually 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 have to match, and 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 similar to 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 as well as slicing:

```
>>> 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 actually 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 the minimum and maximum values:

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

It will come as no surprise that there are various methods that apply 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, for example, lists and dictionaries, can be changed after they have been created, and an immutable object, for example, strings, cannot. If you want to use the new value, 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 list data is great for storing data, but it is a bit tricky at times to keep track of data if we need to reference them by location. We will take a look at Python mapping types next.

Mapping

Python provides one mapping type, called the **dictionary**. The dictionary 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 the *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 that this is a powerful type, because you can refer to the object with a human-readable key. This key, instead of just a list of items, will make more sense for 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 and were troubleshooting 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 can use curly braces to create a dictionary:

```
>>> 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 my network scripts. There are other data containers that can come in handy, 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 in 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. However, the built-in set type would allow you to eliminate the duplicate entries with just one line of code. To be honest, I do not use the Python set data type that much, 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 taken a look at different data types, we will take a tour of Python operators next.

Python operators

Python has some *numeric operators* that you would expect, 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 and returns the integer value. 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 # modular operator
1
```

There are also *comparison operators*. Note the double equals sign for comparison and a single equals sign 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 see 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 take a 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 the blocks. As one would expect, 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 taken a look at Python data types, operators, and control flows, we are ready to group them together into reusable code pieces called functions.

Python functions

Most of the time, when you find yourself copy and pasting some pieces of code, 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 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, or, 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, so here is a quick example. In the following examples, we use positional parameters, so the first element is always referred to as the first variable in the function. Another way of referring to parameters are as 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 such a class. The class serves as a blueprint for subsequent instances.

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

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

Once defined, you are able to create as many instances of that class as you'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, in fact, a Python file is a module and any functions and classes you define in that source file can be reused. 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 you 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:

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 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`.

Packages allow a collection of modules to be grouped together. This further organizes Python modules for more namespace protection to further reusability. A package is defined by creating a directory with a name you want to use as the namespace, then you can place the module source file under that directory.

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

```
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ mkdir math_
stuff
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ touch math_
stuff/__init__.py
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$ tree
.
├── helloworld.py
└── math_stuff
    ├── __init__.py
    └── subtract.py
1 directory, 3 files
echou@pythonicNeteng:~/Master_Python_Networking/Chapter1$
```

The way you will now refer to the module will need to include the package name using the 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.

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 program 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. In 2017, Kirk and others teamed up with David Barroso from the NAPALM project to create the Nornir framework to provide a pure Python network automation framework.

For the most part, the libraries can be used concurrently, for example, Ansible (covered in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, and *Chapter 5*, *The Python Automation Framework – Beyond Basics*) uses both Paramiko and Ansible-NAPALM 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 command line interface. It has proven to be ineffective in network management with moderate size networks. This chapter will introduce Python libraries that can work with that limitation. First, let us discuss some of the challenges with CLI in more detail.

The challenges of the CLI

At the Interop expo in Las Vegas in 2014, Big Switch Networks' CEO Douglas Murray displayed the following slide to illustrate what had changed in **data center networking** (**DCN**) in the 20 years between 1993 to 2013:

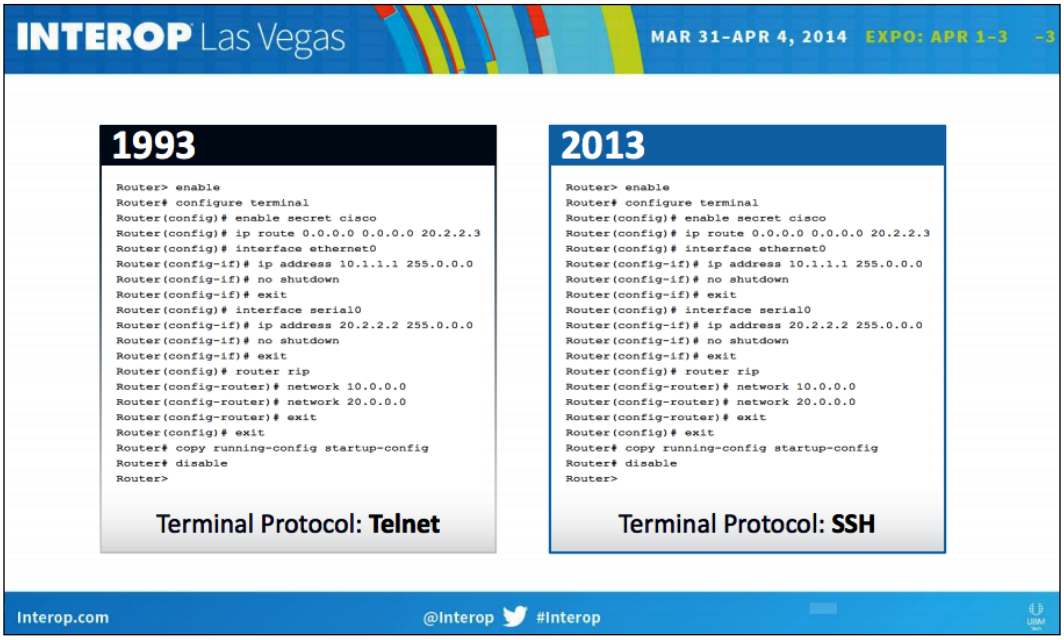


Figure 1: Data center networking changes (source: https://www.bigswitch.com/sites/default/files/presentations/murraydouglasstartuphotseatpanel.pdf)

His point was obvious: not much had changed in those 20 years in the way we manage network devices. While he might have been negatively biased toward the incumbent vendors when displaying this slide, his point is well taken. In his opinion, the only thing that had changed about managing routers and switches in 20 years was the protocol changing from the less secure Telnet to the more secure SSH.

It was right around the same time in 2014 that 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 initial proof of concepts, and deploying the topology for the first time. However, once we've moved beyond the initial deployment, the network management requirements are usually changed to consistently make the same changes reliably across network devices, to make the changes error-free, and to repeat them over and over again without the engineer being distracted or feeling tired. This requirement sounds like an ideal job for computers and our favorite language, Python.

Referring back to the slide, if the network devices can only be managed with the command line, the main challenge becomes replicating the interactions previously between the router and the administrator 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 at 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 in the first try. It is easy enough to install Python and the necessary packages for the management host, but what about those routers and switches that we want to simulate?

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 consisting of 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:

- **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. In short, the comfort level with physical devices is extremely high because of familiarity.
- **Disadvantages**: It is relatively expensive to pay for a device that is only used in the lab. Physical devices require engineering hours to rack and stack and are not very flexible once constructed.

Virtual devices

These are emulations or simulations of actual network devices. They are either provided by the vendors or by the open source community:

- **Advantages**: Virtual devices are easier to set up, relatively cheap, and can make changes to the topology quickly.
- **Disadvantages**: They are usually a scaled-down version of their physical counterpart. 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 the lab and production. In some of the environments I have worked on, 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 quicker because they are usually 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.

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

As such, in previous editions, readers have been able to use many of the popular virtual network labs such as GNS3, Eve-NG, and other virtual machines.

For the examples in this book, I am using virtual machines from various vendors such as Juniper and Arista. At the time of writing, Arista vEOS can be downloaded for free from the Arista site. Juniper JunOS Olive, which I use, is not an official supported platform, but Juniper offers a free trial license for vMX that can be substituted. I am also using a network lab program from Cisco called **Virtual Internet Routing Lab** (**VIRL**), `https://learningnetworkstore.cisco.com/virtual-internet-routing-lab-virl/cisco-personal-edition-pe-20-nodes-virl-20`. It is a paid program, but in the following sections I will explain why I think it is a good option for virtual network labs.

Again, I want to point out that the use of the VIRL program is entirely optional; you can use the free alternatives if you'd like. It is strongly recommended that you have some lab equipment to follow along with the examples in this book.

Cisco VIRL

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 at a discount, each router and switch cost hundreds of US dollars, so 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, it was a pain to connect all the cables, and to introduce link failure, I would literally 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 I tried to learn a new concept. All you need is the IOS images from Cisco, a few carefully constructed topology files, and you can easily build a virtual network that you can test your knowledge on. I had a whole folder of network topologies, pre-saved configurations, and different version of images, as called for by different scenarios. The addition of a GNS3 frontend gives 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 thing that was lacking was the tool not being officially blessed by the vendor, that is Cisco, and the perceived lack of credibility because of it.

In 2015, the Cisco community decided to fulfill this need by releasing the Cisco VIRL. This is my preferred method of developing and trying out much of the Python code, both for this book and my own production use.

As of November 14, 2019, the personal edition 20-Node license is available for purchase for only USD $199.99 per year.

Even at a monetary cost, in my opinion, the VIRL platform offers a few advantages over other alternatives:

- **Ease of use**: As mentioned, all the images for IOSv, IOS-XRv, CSR1000v, NX-OSv, and ASAv are included in a single download.
- **Official** (**kind of**): Although support is community-driven, it is a widely used tool internally at Cisco. Because of its popularity, bugs get fixed quickly, new features are carefully documented, and useful knowledge is widely shared among its users.
- **The cloud migration path**: The project offers a logical migration path when your emulation grows out of the hardware power you have, such as Cisco dCloud (`https://dcloud.cisco.com/`), VIRL on Packet (`http://virl.cisco.com/cloud/`), and Cisco DevNet (`https://developer.cisco.com/`). This is an important feature that sometimes gets overlooked.
- **The link and control-plane simulation**: The tool can simulate latency, jitter, and packet loss on a per-link basis for real-world link characteristics. There is also a control-plane traffic generator for external route injection.
- **Others**: The tool offers some nice features, such as VM Maestro topology design and simulation control, AutoNetKit for automatic config generation, and user workspace management if the server is shared. There are also open source projects such as virlutils (`https://github.com/CiscoDevNet/virlutils`), which are actively worked on by the community to enhance the workability of the tool.

We will not use all of the features in VIRL in this book. But since this is a relatively new tool that is worth your consideration, if you do decide this is the tool you would like to use, I want to offer some of the setups I used.

Again, I want to stress the importance of having a lab to follow along for the book examples. It does not need to be the Cisco VIRL lab. The code examples provided in this book should work across any lab device, as long as it runs the same software type and version.

VIRL tips

The VIRL website (`http://virl.cisco.com/`) offers lots of guidance, preparation, and documentation. I also find that the VIRL user community generally offers quick and accurate help. I will not repeat information already offered in those two places; however, some of the setups I use for the lab in this book follow.

My VIRL lab uses two virtual Ethernet interfaces for connections. The first interface is set up as **network address translation** (**NAT**) for the host machine's internet connection, and the second is used for local management interface connectivity (VMnet2 in the following example). I use a separate virtual machine with a similar network setup in order to run my Python code, with the first primary Ethernet used for internet connectivity and the second Ethernet connection to VMnet2 for lab device management network:

Figure 2: VIRL Ethernet adapter 1 changed to NAT

1. VMnet2 is a custom network created to connect the Ubuntu host with the VIRL virtual machine:

Figure 3: VIRL Ethernet adapter 2 connects to VMNet2

In the **Topology** design option, I set the **Management Network** option to **Shared flat network** in order to use VMnet2 as the management network on the virtual routers:

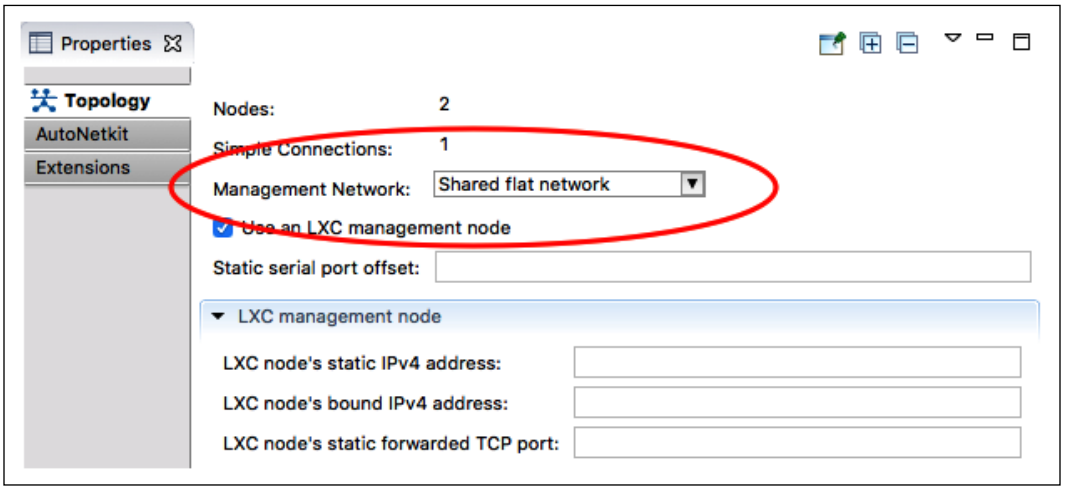


Figure 4: Use Shared flat network for the VIRL management network

2. Under the **Node** configuration, you have the option to statically configure the management IP. I try to statically set the management IP addresses instead of having them dynamically assigned by the software. This allows for more deterministic accessibility:

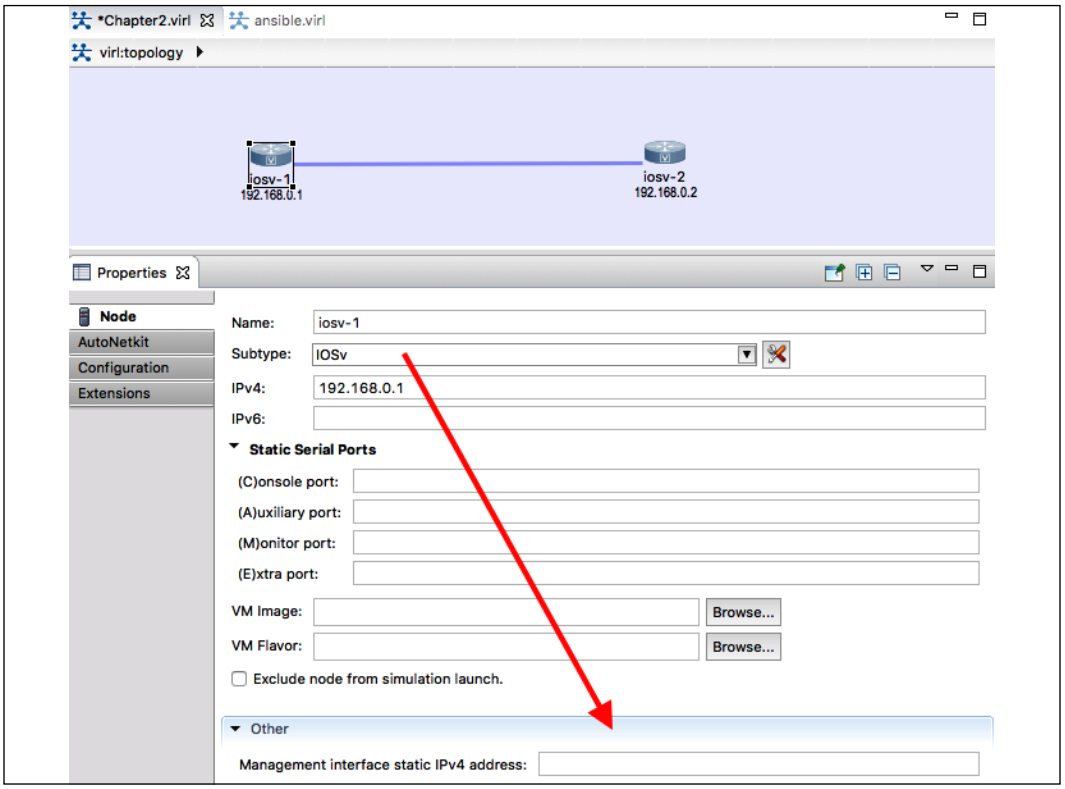


Figure 5: IOSv1 with static management IP

The VIRL lab topology will be provided with each chapter's code examples.

Cisco DevNet and dCloud

Cisco provides two other excellent, at the time of writing, free methods for practicing network automation with various Cisco gear. Both of the tools require a **Cisco Connection Online** (CCO) login. They are both really good, especially for the price point (they are free!).

The first tool is the Cisco DevNet (`https://developer.cisco.com/`) sandbox, which includes guided learning tracks, complete documentation, and sandbox remote labs, among other benefits. Some of the labs are always on, while others you need to reserve. The lab availability will depend on usage. It is a great alternative if you do not already have a lab at your own disposal. DevNet is certainly a tool that you should take full advantage of, regardless of whether you have a locally run VIRL host or not:

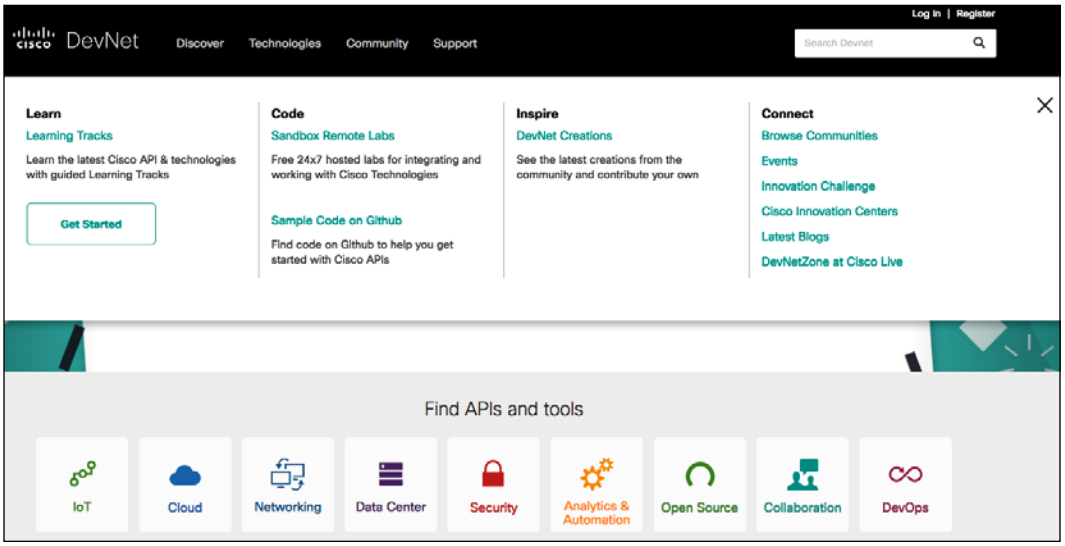


Figure 6: Cisco DevNet

Since its inception, Cisco DevNet has become the defacto destination for all things related to network programmability and automation at Cisco. In fact, in June 2019, Cisco announced many new tracks of DevNet certifications, `https://developers.cisco.com/certification/`

Another free online lab option for Cisco is `https://dcloud.cisco.com/`. You can think of dCloud as running VIRL on other people's servers without having to manage or pay for those resources. It seems that Cisco is treating dCloud as both a standalone product as well as an extension to VIRL. For example, in the use case of when you are unable to run more than a few IOS-XR or NX-OS instances locally, you can use dCloud to extend your local lab.

It is a relatively new tool, but it is definitely worth a look:

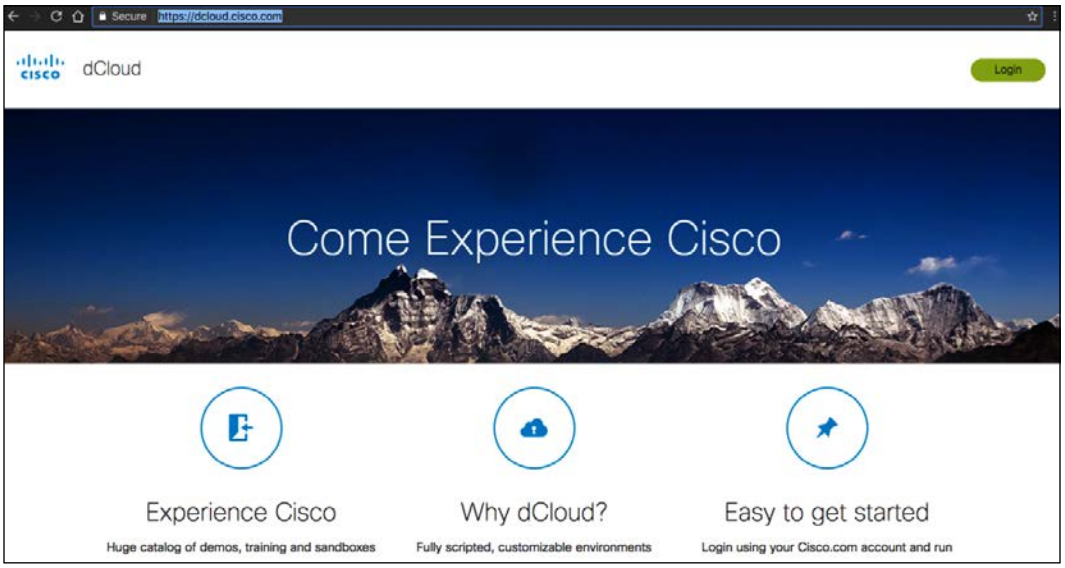


Figure 7: Cisco dCloud

GNS3

There are a few other virtual labs that I have used for other projects. The GNS3 tool is one of them:

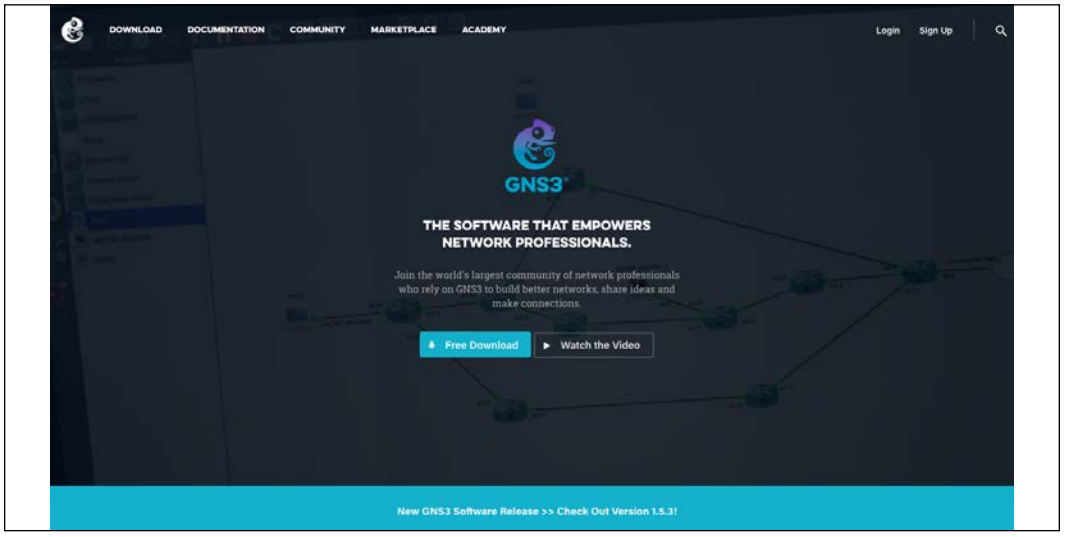


Figure 8: GNS3 Website

As mentioned previously, GNS3 is what a lot of us used 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. One of the downsides for Cisco-sponsored tools, such as VIRL, DevNet, and dCloud, is that they only contain Cisco technologies. Even though they provide ways for virtual lab devices to communicate with the outside world and to communicate with other vendor equipment, the steps are not very intuitive. GNS3 is vendor-neutral and can include a multi-vendor virtualized platform directly in the 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 Juniper Olive emulation). GNS3 is useful when there is a need to incorporate multi-vendor technologies into the same lab.

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 much experience with the tool, but many of my colleagues and friends in the industry use it for their network labs.

There are also other virtualized platforms, such as Arista vEOS (`https://eos.arista.com/tag/veos/`), Juniper vMX (`http://www.juniper.net/us/en/products-services/routing/mx-series/vmx/`), and vSRX (`http://www.juniper.net/us/en/products-services/security/srx-series/vsrx/`), which you can use as a standalone virtual appliance during testing. They are great complementary tools for testing platform-specific features, such as the differences between the API versions on the platform. Many of them are offered as paid products on public cloud provider marketplaces for easier access. They often offer the identical feature as their physical counterpart.

Now that we have built our network lab, we can start to experiment with the Python libraries that can help with management and automation. We will begin with 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 your script to spawn a child application and control it as if a human were typing commands.

Read the Pexpect docs at `https://pexpect.readthedocs.io/en/stable/index.html`

Let's take a look at the Python Pexpect library. Similar to the original **Tool Command Language** (**TCL**) `Expect` module by Don Libe, 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.

Python virtual environment

Let us start by using the Python virtual environment, which allows us to manage separate package installations for different projects. This is accomplished by creating a "virtual" isolated Python installation and installing packages into that virtual installation and we would not need to worry about breaking the packages installed globally or from other virtual environments. We will start by installing the Python `pip` tool, then create the virtual environment:

```
$ sudo apt update
$ sudo apt install python3-pip
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
(venv) $ which python
/home/echou/venv/bin/python
(venv) $ deactivate
```

As you can see from the output, we use the `venv` package from the Python 3 standard library, create the directory containing our environment, 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`.

Python 2's usage and installation of a virtual environment is slightly different. You can find a number of tutorials online for Python 2 virtual environments.

Pexpect installation

Pexpect installation process is pretty straight forward:

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

If you are installing Python packages in the global environment, you will need to use root privilege, such as `sudo pip install pexpect`.

Do a quick to test to make sure the package is usable, make sure we start the Python interactive shell from the virtual environment:

```
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] 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']
```

Pexpect overview

For our first lab, we will construct a simple network with two IOSv devices connected back to back:

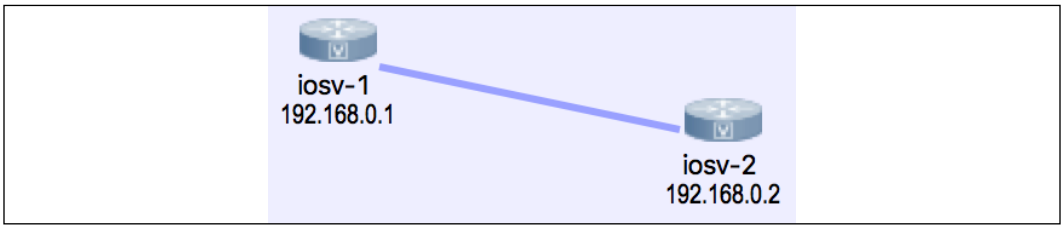


Figure 9: Lab topology

The devices will each have a loopback address in the `192.16.0.x/24` range and the management IP will be in the `172.16.1.x/24` range. The VIRL topology file is included in the accompanying downloadable files as well as on the GitHub repository (`https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition`) for the book. You can import the topology to your own VIRL software. If you do not have VIRL, you can also view the necessary information by opening the topology file with a text editor. The file is simply an XML file with each node's information under the node element:

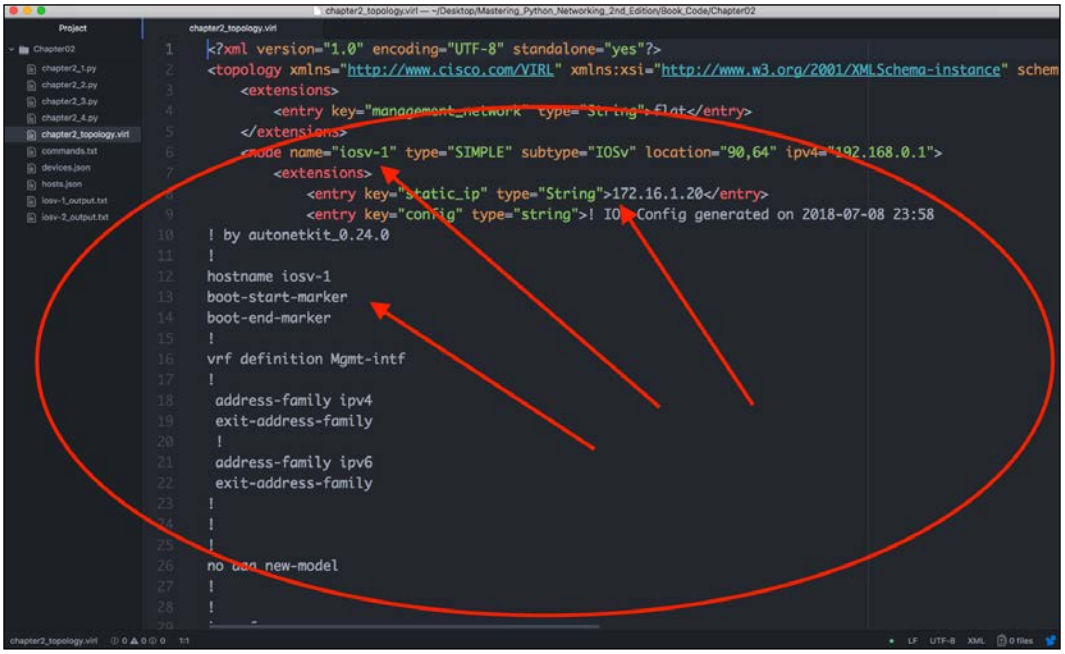


Figure 10: Lab node information

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 172.16.1.20
Trying 172.16.1.20...
Connected to 172.16.1.20.
Escape character is '^]'.
<skip>
User Access Verification

Username: cisco
Password:
```

I used VIRL AutoNetKit to automatically generate the initial configuration of the routers, which generated the default username `cisco`, and the password `cisco`. Notice that the user is already in privileged mode because of the privilege assigned in the configuration:

```
iosv-1#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 172.16.1.20')
>>> child.expect('Username')
0
>>> child.sendline('cisco')
6
>>> child.expect('Password')
0
>>> child.sendline('cisco')
6
>>> child.expect('iosv-1#')
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 that the string in the Pexpect process looks for is an indicator 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 (`iosv-1#`) 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 your 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 (`iosv-1#`), including the `show version` command. The `after` text is the router hostname prompt:

```
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
```

```
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/3.7/library/stdtypes.html`)

What would happen if you expected the wrong term? For example, if you typed in `username` with the lowercase "u" instead of `Username` after spawning the child application, then 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 you 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 Python regular expressions, go to: `https://docs.python.org/3.7/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, you 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 you can fit the different hostname in a regular expression, whereas use the possible options if you need to catch completely different responses from the router, such as a password rejection. For example, if you use several different passwords for your login, you 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, you 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('iosv-1')
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 iosv-1`:

```
iosv-1#sh run | i hostname
hostname iosv-1
```

Taking a closer look at the expected string will reveal the mistake. In this case, we were missing the hash (#) sign behind the `iosv-1` 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('iosv-1#')
0
>>> child.before
b'#show run | i hostname\r\nhostname iosv-1\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': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
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 use a nested dictionary in line 5:

```
devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
```

The nested dictionary allows us to refer to the same device (such as `iosv-1`) 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:

```
(venv) $ python chapter2_1.py
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'
b'show version | i V\r\nCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(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('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrn'
>>> child.interact()
show version | i V
Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version
15.6(3)M2, RELEASE SOFTWARE (fc2)
Processor board ID 9Y0KJ2ZL98EQQVUED5T2Q
iosv-1#sh run | i hostname
hostname iosv-1
iosv-1#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 0x7f95f25ff780>\ncommand: /usr/
bin/telnet\nargs: ['/usr/bin/telnet', '172.16.1.20']\nbuffer (last 100
chars): b''\nbefore (last 100 chars): b'                        *\\r\\n*****
**********************************************************
**********\\r\\n'\nafter: b'iosv-1#'\nmatch: <_sre.SRE_Match object;
span=(612, 619), match=b'iosv-1#'>\nmatch_index: 0\nexitstatus: 1\
nflag_eof: False\npid: 5676\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 172.16.1.20')
>>> child.logfile = open('debug', 'wb')
```

Use `child.logfile = open('debug', 'w')` for Python 2. Python 3 uses byte strings by default. 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 leave our communication in clear text during the session. In modern networks, we typically use **secure shells** (**SSH**) for management. In the next section, we will take a look at Pexpect with SSH.

Pexpect and SSH

If you try to use the previous Telnet example and plug it into an SSH session instead, you might find yourself pretty frustrated with the experience. You always have to include the username in the session, answering the `ssh` new key question, and much more mundane tasks. There are many ways to make SSH sessions work, but luckily, 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.

Let's generate the `ssh-key` for `iosv-1` for `ssh`:

```
iosv-1(config)#crypto key generate rsa general-keys
The name for the keys will be: iosv-1.virl.info
Choose the size of the key modulus in the range of 360 to 4096 for your
General Purpose Keys. Choosing a key modulus greater than 512 may take a
few minutes


How many bits in the modulus [512]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 2 seconds)
```

The procedures are mostly the same, with the exception of `login()` and `logout()`:

```
>>> from pexpect import pxssh
>>> child = pxssh.pxssh()
>>> child.login('172.16.1.20', 'cisco', 'cisco', auto_prompt_reset=False)
True
>>> child.sendline('show version | i V')
19
>>> child.expect('iosv-1#')
0
>>> child.before
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2) Processor
board ID 9MM4BI7B0DSWK40KV1IIRrn'
>>> 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.

Putting things together for Pexpect

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`.

You can download the script from the book GitHub repository, `https://github.com/PacktPublishing/Mastering-Python-Networking-Third-Edition`.

If `ssh` key has not been generated on the other routers, `iosv-2`, we should do so now:

```
iosv-2(config)#crypto key generate rsa general-keys
The name for the keys will be: iosv-2.virl.info
Choose the size of the key modulus in the range of 360 to 4096 for your
General Purpose Keys.Choosing a key modulus greater than 512 may take a
few minutes

How many bits in the modulus [512]: 2048
% Generating 2048 bit RSA keys, keys will be non-exportable...
[OK] (elapsed time was 2 seconds)
```

Refer to the following code:

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

import getpass
from pexpect import pxssh

devices = {'iosv-1': {'prompt': 'iosv-1#', 'ip': '172.16.1.20'},
           'iosv-2': {'prompt': 'iosv-2#', 'ip': '172.16.1.21'}}
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
- It writes the output in two files, `iosv-1_output.txt` and `ios-2_output.txt`, to be further analyzed

For Python 2, use `raw_input()` instead of `input()` for the username prompt. Also, use `w` for the file mode instead of `wb`.

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 Basics* and *Chapter 5*, *The Python Automation Framework – Beyond Basics*. 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, Mac, and other flavors of Linux can be found at: `https://cryptography.io/en/latest/installation/`

We will show the Paramiko installation of our Ubuntu 18.04 virtual machine in the following output. The following output shows the installation steps, as well as Paramiko successfully imported into the Python interactive prompt:

```
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 3.6.8 (default, Aug 20 2019, 17:12:48)
[GCC 8.3.0] 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('172.16.1.20', 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\**********************************************************
**********
***rn* IOSv is strictly limited to use for evaluation, demonstration
and IOS *rn* education. IOSv is provided as-is and is not supported by
Cisco's
```

```
*rn* Technical Advisory Center. Any use or disclosure, in whole or in
part,
*rn* of the IOSv Software or Documentation to any third party for any
*rn* purposes is expressly prohibited except as otherwise authorized by
*rn* Cisco in writing.
*rn************************************************************
**********
**rniosv-1#"
>>> new_connection.send("show version | i V\n")
19
>>> time.sleep(3)
>>> output = new_connection.recv(5000)
>>> print(output)
b'show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#'
>>> new_connection.close()
>>>
```

The `time.sleep()` function inserts a time delay to ensure that 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 you are seeing the Paramiko operation for the first time, the beauty of Python and its clear syntax means that you 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('172.16.1.20', 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, `iosv-1` is not present 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 the additional login session from Paramiko:

```
iosv-1#who
Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
579 vty 1 cisco idle 00:01:30 172.16.1.173
Interface User Mode Idle Peer Address
iosv-1#
```

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('172.16.1.20', 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 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor board ID
9MM4BI7B0DSWK40KV1IIRrn'
>>>
```

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:

```
iosv-1#who
Line User Host(s) Idle Location
*578 vty 0 cisco idle 00:00:00 172.16.1.1
Interface User Mode Idle Peer Address
iosv-1#
```

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):
```

```
File "<stdin>", line 1, in <module>

File "/usr/local/lib/python3.5/dist-packages/paramiko/client.py", line
435, in exec_command

chan = self._transport.open_session(timeout=timeout)

File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line
711, in open_session

timeout=timeout)

File "/usr/local/lib/python3.5/dist-packages/paramiko/transport.py", line
795, in open_channel

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 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor

board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#show version | i VrnCisco IOS
Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.6(2)T,
RELEASE SOFTWARE (fc2)rnProcessor board ID 9MM4BI7B0DSWK40KV1IIRrniosv-
1#show version | i VrnCisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(2)T, RELEASE SOFTWARE (fc2)rnProcessor
board ID 9MM4BI7B0DSWK40KV1IIRrniosv-1#'

>>>
```

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

Our 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-Third-Edition`. I will list the notable differences here:

```
devices = {'iosv-1': {'ip': '172.16.1.20'}, 'iosv-2': {'ip':
'172.16.1.21'}}
```

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 some 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, during the time between the execution of commands, we will wait five seconds. 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 Basics*, 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 VIRL 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 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/echoun'
>>> 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. Some network devices, such as Cumulus and Vyatta switches, also support key-based authentication.

Putting things together for Paramiko

We are almost at the end of the chapter. In this last 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 of the script. Hardcoding the host and command has a higher chance of making mistakes. Besides, if you were to pass on the script to colleagues, they might not feel comfortable working in Python, Paramiko, or Linux.

By making both the host and command files be read in as parameters for the script, we can eliminate some of these concerns. Users (and a future you) 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
{
"iosv-1": {"ip": "172.16.1.20"},
"iosv-2": {"ip": "172.16.1.21"}
}
```

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:

```
(venv) $ python chapter2_4.py
Username: cisco
Password:
```

```
b'terminal length 0\r\niosv-1#config t\r\nEnter configuration commands,
one per line.  End with CNTL/Z.\r\niosv-1(config)#'
b'logging buffered 30000\r\niosv-1(config)#'
b'end\r\niosv-1#'
<skip>
```

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

```
iosv-1#sh run | i logging
logging buffered 30000
iosv-1#sh start | i logging
logging buffered 30000

iosv-2#sh run | i logging
logging buffered 30000
iosv-2#sh start | i logging
logging buffered 30000
```

The Paramiko library is a general-purpose library that is intended for working with interactive command line programs. For network management, there is another library, Netmiko, that is 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 if you have noticed from previous examples, we are repeating many of the same steps between `iosv-1` and `isov-2` for device login and execution. Once we start to develop more automation commands, we also start to repeat ourselves to capture outputs and format 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
>>> ios_v1 = {'device_type': 'cisco_ios', 'host': '172.16.1.20',
'username': 'cisco', 'password': 'cisco'}
>>> net_connect = ConnectHandler(**ios_v1)
```

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()
'iosv-1#'
>>> output = net_connect.send_command('show ip int brief')
>>> print(output)
Interface                  IP-Address      OK? Method Status
Protocol
GigabitEthernet0/0         172.16.1.20     YES NVRAM  up
up
GigabitEthernet0/1         10.0.0.5        YES NVRAM  up
up
Loopback0                  192.168.0.1     YES NVRAM  up
up
```

Let's see another example for the second Cisco IOS device in our lab, but this time we will define the `iosv-2` parameter when we initiate the `ConnectHandler` object 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='172.16.1.21', username='cisco', password='cisco')
>>> output = net_connect_2.send_config_set(['logging buffered 19999'])
>>> print(output)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
iosv-2(config)#logging buffered 19999
iosv-2(config)#end
iosv-2#
>>> exit()
```

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 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 discuss another automation framework written in Python, Ansible, in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, and *Chapter 5*, *The Python Automation Framework – Beyond Basics*. I wanted to introduce the framework in this chapter as a way to demonstrate another way to automate devices with low-level interaction. However, if you are just starting out on the automation journey, the framework might make more sense after you have finished reading *Chapter 5*, *The Python Automation Framework – Beyond Basics*. Please feel free to scan through the example and come back to it later.

Of course, we will start with installing `nornir` in our environment:

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

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:

```
---
iosv-1:
    hostname: '172.16.1.20'
    port: 22
    username: 'cisco'
    password: 'cisco'
    platform: 'cisco_ios'

iosv-2:
    hostname: '172.16.1.21'
    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:

```
from nornir import InitNornir
```

```
from nornir.plugins.tasks.networking import netmiko_send_command
from nornir.plugins.functions.text import print_result

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************************************************************
* iosv-1 ** changed : False ***********************************************
*******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.0.5               -   fa16.3e0e.a3a3  ARPA
GigabitEthernet0/1
Internet  10.0.0.6              40   fa16.3ed7.1041  ARPA
GigabitEthernet0/1
^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^
* iosv-2 ** changed : False ***********************************************
*******
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvvv INFO
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.0.5              40   fa16.3e0e.a3a3  ARPA
GigabitEthernet0/1
Internet  10.0.0.6               -   fa16.3ed7.1041  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'll discuss another automation framework, pyATS and Genie, in *Chapter 15*, *Test-Driven Development for Networks*, when we discuss testing and verification.

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 method for automating CLI-only devices so far 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.

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

Bad automation speeds bad things up

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 us 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 procedures. The internet is full of horror stories of when someone pressed the *Enter* key and immediately regretted it.

We need to make sure the chances of bad automation scripts screwing things up are as small as possible. 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 two 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.

3
APIs and Intent-Driven Networking

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

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

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

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

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

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

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

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

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

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

Infrastructure-as-code

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

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

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

Intent-driven networking

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

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

Screen scraping versus API structured output

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Data modeling for infrastructure-as-code

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

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

The data modeling process is illustrated in the following diagram:

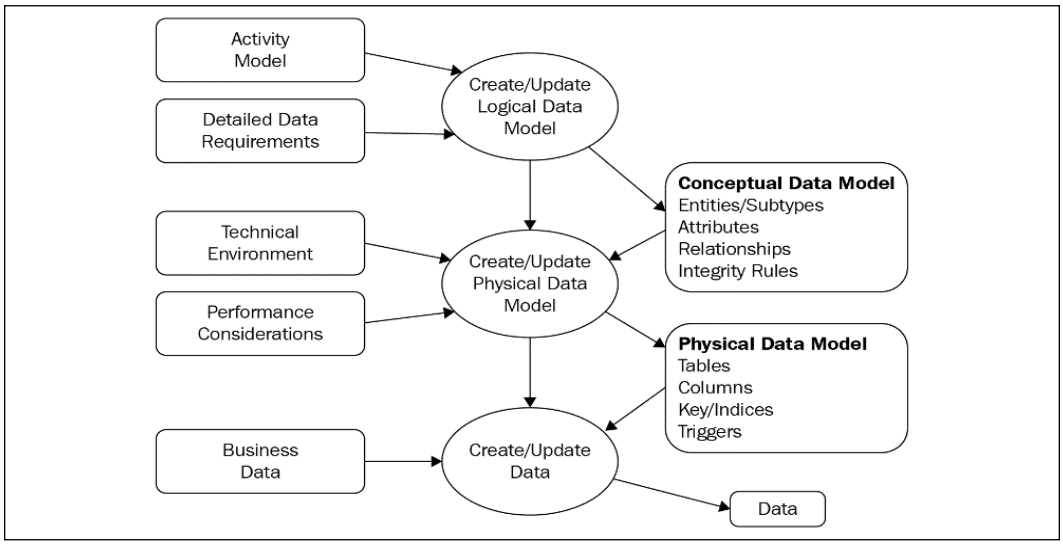


Figure 1: Data modeling process

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

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

YANG and NETCONF

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

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

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

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

The Cisco API and ACI

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

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

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

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

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

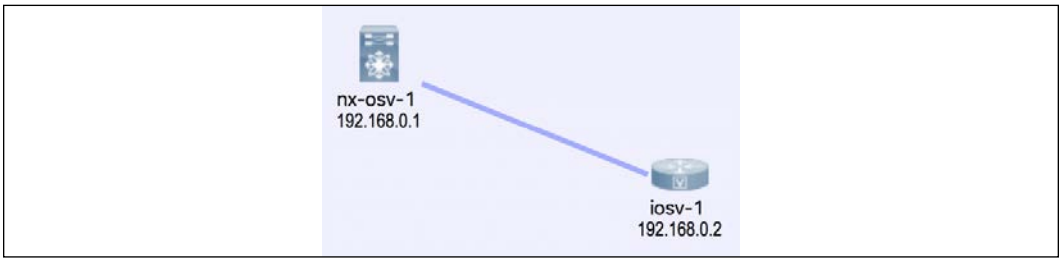


Figure 2: Lab topology

Let's take a look at NX-API.

Cisco NX-API

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

Lab software installation and device preparation

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

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

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

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

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

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

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

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

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

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

NX-API examples

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

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

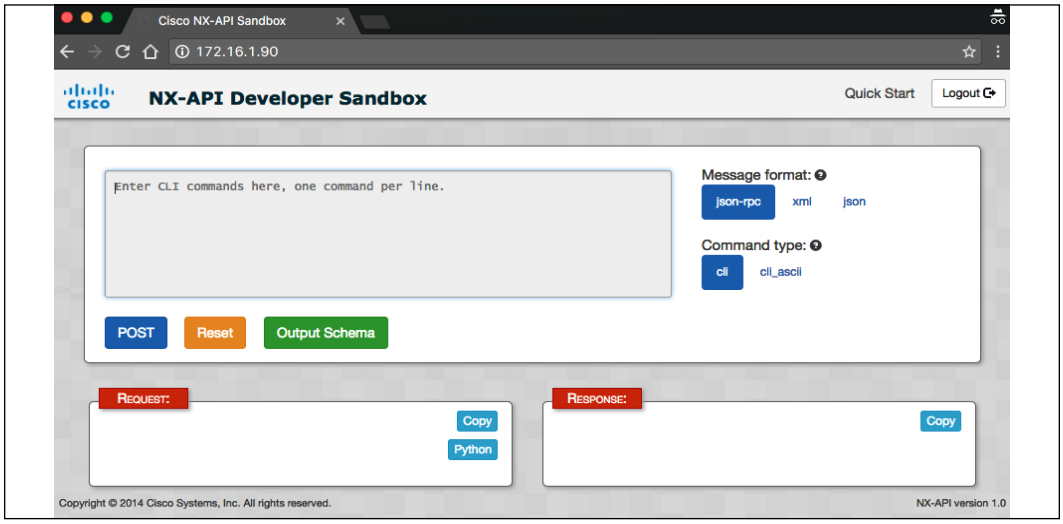


Figure 3: NX-API Developer Sandbox

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

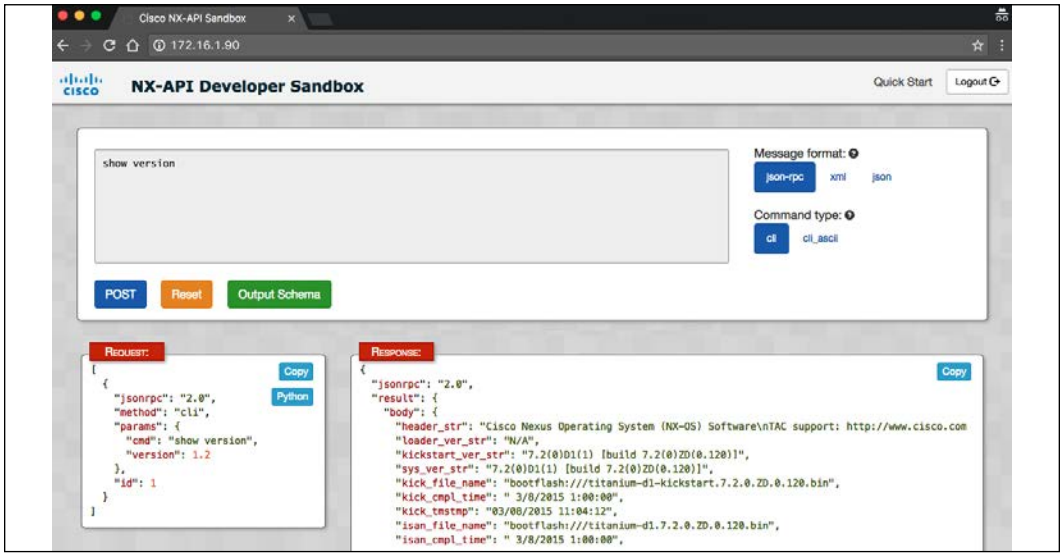


Figure 4: Cisco NX-API Developer Sandbox command output

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

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

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

from ncclient import manager

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

for value in conn.server_capabilities:
    print(value)

conn.close_session()
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    "id": 4
    }
]
```

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

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

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

The Cisco YANG model

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

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

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

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

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

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

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

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

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

<omitted>
```

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

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

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

The Cisco ACI and APIC-EM

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

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

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

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

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

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

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

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

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

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

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

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

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

Cisco IOS-XE

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

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

Cisco Meraki controller

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

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

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

Let us execute the preceding script:

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

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

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

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

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

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

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

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

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

The Python API for Juniper Networks

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

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

Juniper and NETCONF

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

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

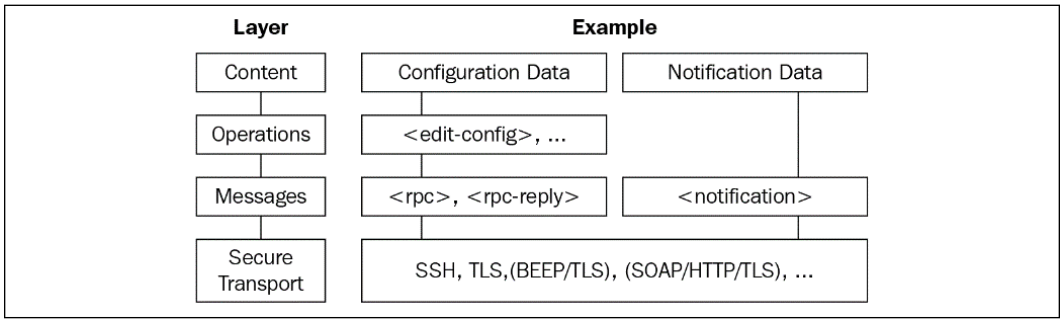


Figure 5: NETCONF model

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

Device preparation

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

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

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

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

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

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

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

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

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

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

Juniper NETCONF examples

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

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

```
from ncclient import manager

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

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

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

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

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

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

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

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

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

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

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

We will lock the configuration and make configuration changes:

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

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

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

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

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

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

# unlock config
conn.unlock()

# close session
conn.close_session()
```

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

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

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

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

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

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

Juniper PyEZ for developers

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

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

Installation and preparation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We can also try to use the SSH key authentication:

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

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

PyEZ examples

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

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

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

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

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

dev.close()
```

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

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

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

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

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

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

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

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

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

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

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

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

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

dev.close()
```

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

The Arista Python API

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

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

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

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

Arista eAPI management

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

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

eAPI preparation

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

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

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

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

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

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


URLs

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


arista1#
```

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

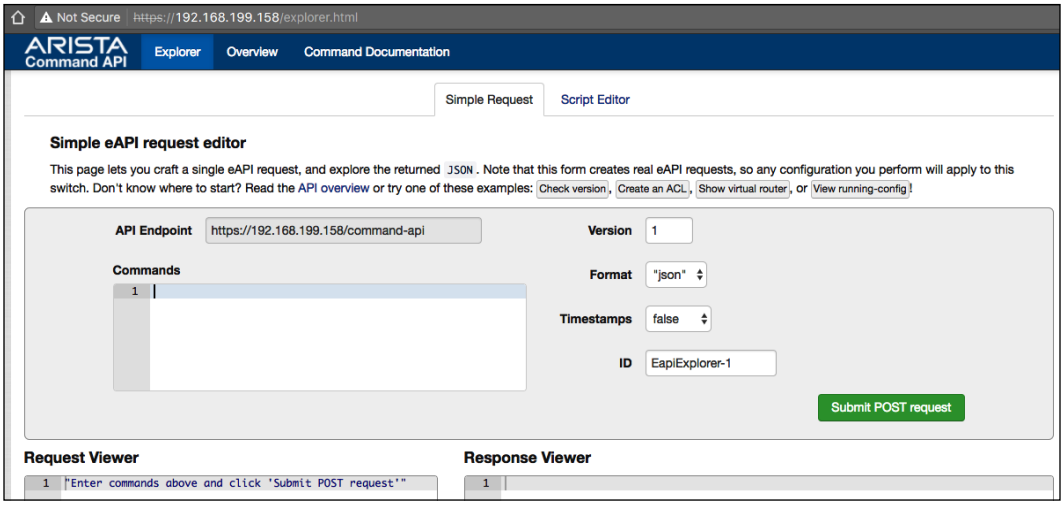


Figure 6: Arista EOS explorer

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

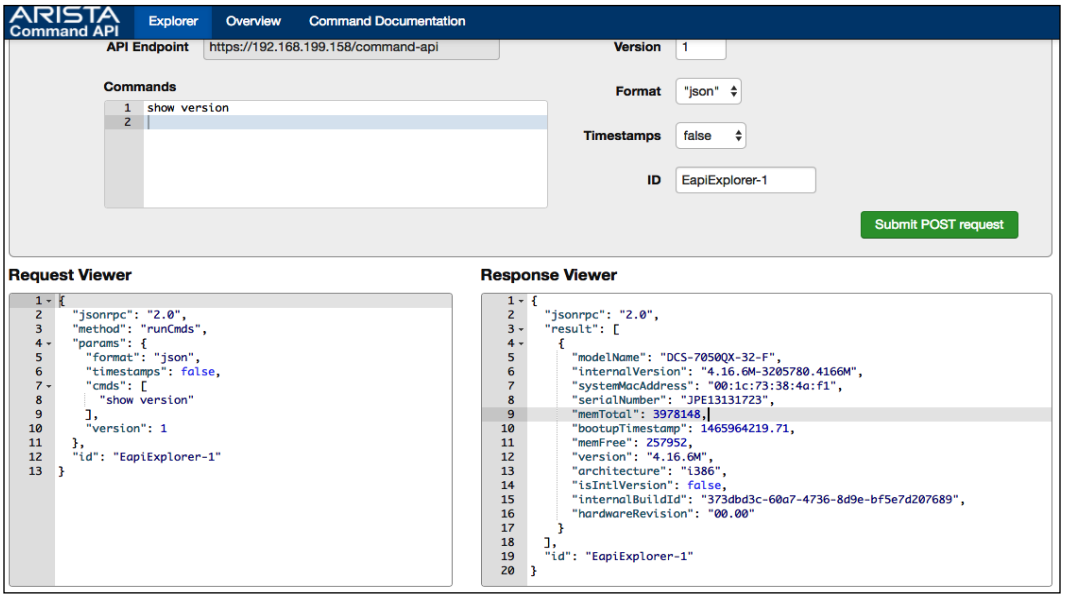


Figure 7: Arista EOS explorer viewer

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

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

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

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

eAPI examples

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

```
#!/usr/bin/python2

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

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

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

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

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

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

The output will contain only the serial number:

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

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

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

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

```
#!/usr/bin/python2

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

ssl._create_default_https_context = ssl._create_unverified_context

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


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

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

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

Here is the output of the command's execution:

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

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

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

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

The Arista Pyeapi library

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

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

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

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

Pyeapi installation

Installation is straightforward with `pip`:

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

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

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

Here is the file I am using in the lab:

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

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

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

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

Pyeapi examples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

import pyeapi
```

```
class my_switch():

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

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

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

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

We have now taken a look at Python scripts for three of the top vendors in networking: Cisco Systems, Juniper Networks, and Arista Networks. In the next section, we will take a look at an open source network operating system that is gaining some momentum in the same space.

VyOS example

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

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

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

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

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

import vymgmt

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

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

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

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

Other libraries

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

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

Summary

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

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

4

The Python Automation Framework – Ansible Basics

The previous two chapters incrementally introduced different ways to interact with network devices. In *Chapter 2*, *Low-Level Network Device Interactions*, we discussed Pexpect and Paramiko libraries, which manage an interactive session to control the 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 and gradually expressed our network in terms of 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:

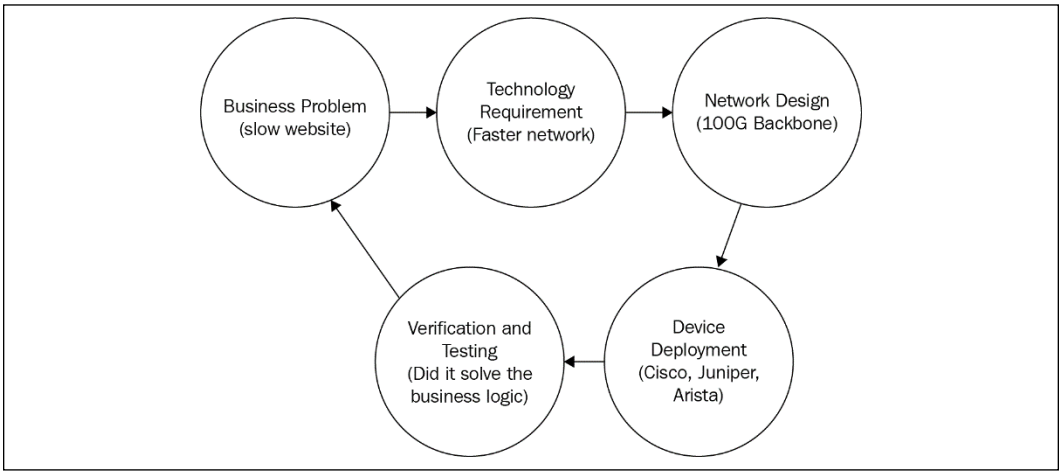


Figure 1: Business logic to network deployment

In my opinion, network automation is not just about faster configuration. It should also be about solving business problems, and accurately and reliably translating our intention into device behavior. These are the goals that we should keep in mind as we march on the network automation journey. In this chapter, we will start to 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
- A quick Ansible example
- The advantages of Ansible
- The Ansible architecture
- Ansible Cisco modules and examples
- Ansible Juniper modules and examples
- Ansible Arista modules and examples

Ansible – a more declarative framework

Let us imagine ourselves 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 realized that 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, which requires:
    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, which in most cases is under the interface configuration section so that it can be applied to the interfaces.

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 of the steps, 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 and in *Chapter 5*, *The Python Automation Framework – Beyond Basics*, we will work with an open source automation tool called Ansible. It is a framework that can simplify the process of going from business logic to 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 as well as one of the most supported by network equipment vendors. In the 'Python Developers Survey 2018' sponsored by the Python Software Foundation, Ansible is ranked at #1 for configuration management:

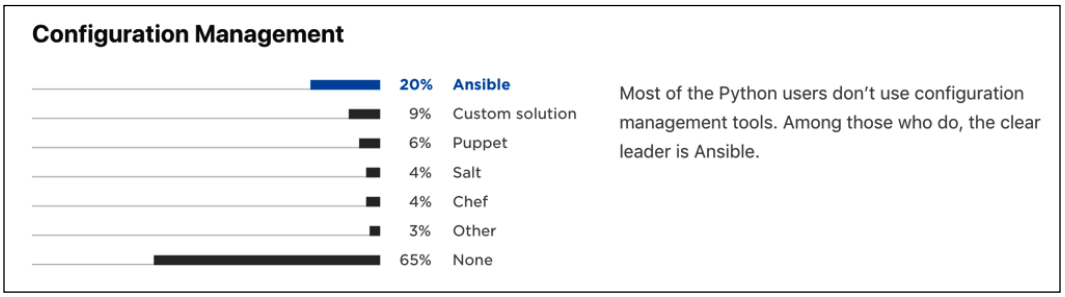


Figure 2: Python Software Foundation survey result for Configuration Management (source: https://www.jetbrains.com/research/python-developers-survey-2018/)

At the time of writing this third edition, Ansible release 2.8 can be run from any machine with Python 2 (version 2.7) or Python 3 (Python 3.5 and higher). Just like Python, many of the useful features of Ansible come from the community-driven extension modules. Even with Ansible core module supportability with Python 3, many of the extension modules and production deployments are still in Python 2 mode. It will take some time to bring all the extension modules up from Python 2 to Python 3. Due to this reason, for the rest of this book, we will use Python 2.7 with Ansible 2.8.

Ansible 2.5, released in March 2018, marks an important release for network automation. Starting from version 2.5 and beyond, Ansible starts to offer many new network module features with new connection methods, playbook syntax, and best practices. Given that this was relatively recent, many production deployments are still pre-2.5 release. As a way to maintain backward compatibility, some of the examples will contain pre-2.5 format first, then we will also see examples with the latest release. The differences will be pointed out as we move along in the chapter. It also makes a good learning experience to go from the older style to the newer one, the reasoning behind the newer changes will make more sense.

For the latest information on Ansible Python 3 support, check out: `http://docs.ansible.com/ansible/python_3_support.html`.

As one can tell from the previous chapters, I am a believer in learning by example. Just like the underlying Python code for Ansible, the syntax for Ansible constructs is easy enough to understand, even if you have not worked with Ansible before. If you have some experience with YAML or Jinja2, you will quickly draw a correlation between the syntax and the intended procedure. Let's take a look at an example first.

A quick Ansible example

As with other automation tools, Ansible started out by managing servers before expanding its ability to manage networking equipment. For the most part, the modules and what Ansible refers to as the 'playbooks' are similar between server modules and network modules, with subtle differences. In this chapter, we will look at a server task example first and draw comparisons later on with network modules.

The control node installation

First, let's clarify the terminology we will use in the context of Ansible. We will refer to the virtual machine with Ansible installed as the control machine or control node, and the machines being managed as the target machines or managed nodes. Ansible can be installed on most of the Unix systems, with the only dependency of Python 2.7 or Python 3.5+. Currently, the Windows operating system is not officially supported as the control machine. Windows hosts can still be managed by Ansible; they are just not supported as the control machine.

As Windows 10 starts to adopt the Windows subsystem for Linux, Ansible might soon be ready to run on Windows as well. For more information, please check the Ansible documentation for Windows (`https://docs.ansible.com/ansible/latest/user_guide/windows_faq.html`).

On the managed node requirements, you may notice some documentation mentioning that Python 2.7 or later is a requirement. This is true for managing target nodes with operating systems such as Linux, but obviously not all network equipment supports Python. We will see how this requirement is bypassed for networking modules via local execution on the control node.

For Windows, Ansible modules are implemented in PowerShell. Windows modules in the core and extra repository live in a Windows subdirectory if you would like to take a look.

We will be installing Ansible on our Ubuntu virtual machine. For instructions on installation on other operating systems, check out the installation documentation (`http://docs.ansible.com/ansible/intro_installation.html`). In the following code block, you will see the steps for installing the software packages:

```
$ sudo apt update
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get install ansible
```

We can also use pip to install Ansible: `pip install ansible`. My personal preference is to use the operating system's package management system, such as Apt on Ubuntu.

We can now do a quick verification as follows:

```
$ ansible --version
ansible 2.8.5
  config file = /etc/ansible/ansible.cfg
```

Now, let's see how we can run different versions of Ansible on the same control node. This is a useful feature to adopt if you'd like to try out the latest development features without permanent installation. We can also use this method if we intend on running Ansible on a control node for which we do not have root permissions.

Running different versions of Ansible from source

You can run Ansible from a source code checkout (we will look at Git as a version control mechanism in *Chapter 13*, *Working with Git*):

```
$ git clone https://github.com/ansible/ansible.git --recursive
$ cd ansible/
$ source ./hacking/env-setup
...
Setting up Ansible to run out of checkout...
$ ansible --version
ansible 2.10.0.dev0
  config file = /etc/ansible/ansible.cfg
...
```

As illustrated, we are now running ansible `2.10.0.dev0` in the shell, which is different from the system version of 2.8.5. To run another version, we can simply use `git checkout` for the different branch or tag and perform the environment setup again:

```
$ git branch -a
$ git tag --list
$ git checkout v2.5.6
...
HEAD is now at 0c985fee8a New release v2.5.6
$ source ./hacking/env-setup
$ ansible --version
ansible 2.5.6 (detached HEAD 0c985fee8a) last updated 2019/09/23 07:05:28
(GMT -700)
  config file = /etc/ansible/ansible.cfg
```

If the Git commands seem a bit strange to you, we will cover Git in more detail in *Chapter 13, Working with Git*.

Let us switch to Ansible version 2.2 and run the update for the core module:

```
$ git checkout v2.2.3.0-1
HEAD is now at f5be18f409 New release v2.2.3.0-1
$ source ./hacking/env-setup
$ ansible --version
ansible 2.2.3.0 (detached HEAD f5be18f409) last updated 2019/09/23
07:09:11 (GMT -700)
```

Git allows the maintainer to include other Git repositories called submodules in a repository. As recommended by Ansible for running from source, `https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#running-from-source`, update the submodule to synchronize with the current release:

```
$ git submodule update --init --recursive Submodule 'lib/ansible/modules/
core'
(https://github.com/ansible/ansible-modules-core) registered for path
'lib/ansible/modules/core'
```

Let's take a look at the lab topology we will use in this chapter and the next one.

Lab setup

In this chapter and in *Chapter 5*, *The Python Automation Framework – Beyond Basics*, our lab will have an Ubuntu 18.04 control node machine with Ansible installed. This control machine will have reachability for the management network for our VIRL devices, which consist of IOSv and NX-OSv devices. We will also have a separate Ubuntu virtual machine for our playbook example when the target machine is a Linux host:

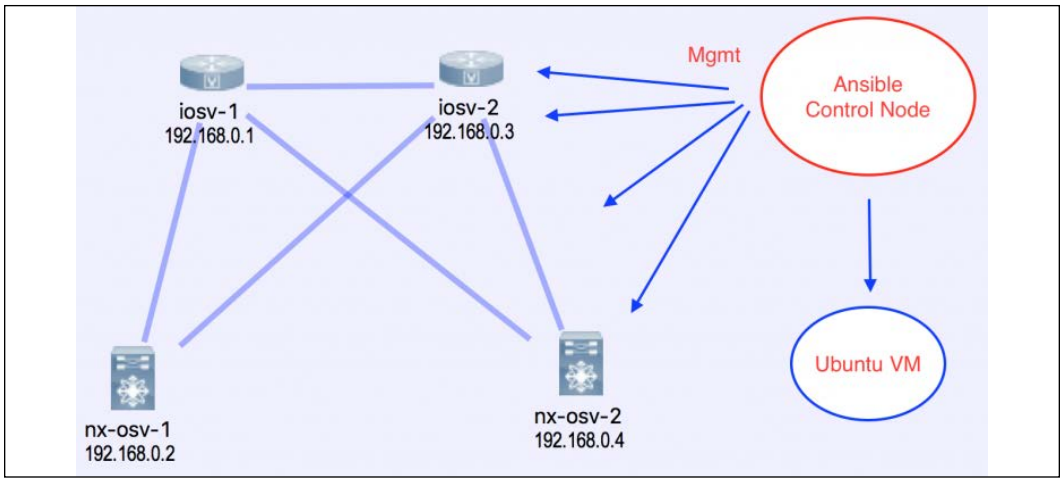


Figure 3: Lab topology

Now, we are ready to see our first Ansible playbook example.

Your first Ansible playbook

Our first playbook will be used between the control node and a remote Ubuntu host. We will take the following steps:

1. Make sure the control node can use key-based authorization
2. Create an inventory file
3. Create a playbook
4. Execute and test it

The public key authorization

The first thing to do is copy your SSH public key from your control machine to the target machine. A full public key infrastructure tutorial is outside the scope of this book, but here is a quick walk-through on the control node:

```
$ ssh-keygen -t rsa <<<< generates public-private key pair on the host
machine if you have not done so already
$ cat ~/.ssh/id_rsa.pub <<<< copy the content of the output and paste it
to the ~/.ssh/authorized_keys file on the target host for the same user,
create the file with a text editor such as VI or Emac if the file does
not exist.
```

You can read more about PKI at: `https://en.wikipedia.org/wiki/Public_key_infrastructure`.

Because we are using key-based authentication, we can turn off password-based authentication on the remote node and be more secure. You will now be able to use SSH to connect from the control node to the remote node using the private key without being prompted for a password.

Can you automate the initial public key copying? It is possible, but is highly dependent on your use case, regulation, and environment. It is comparable to the initial console setup for network gear to establish initial IP reachability. Do you automate this? Why or why not?

In the next section, let us take a look at how we can indicate the target machines to be managed by Ansible.

The inventory file

We wouldn't need Ansible if we had no remote target to manage, right? Everything starts with the fact that we need to perform some task on a remote host. In Ansible, the way we specify the potential remote target is with an inventory file. We can have this inventory file as the `/etc/ansible/hosts` file or use the `-i` option to specify the file during playbook runtime. Personally, I prefer to have this file in the same directory as where my playbook is and use the `-i` option.

Technically, this file can be named anything you like as long as it is in a valid format. However, the convention is to name this file `hosts`. You can potentially save yourself and your colleagues some headaches in the future by following this convention.

The inventory file is a simple, plaintext INI-style (`https://en.wikipedia.org/wiki/INI_file`) file that states your target. By default, the target can either be a DNS FQDN or an IP address:

```
$ cat hosts
192.168.2.122
```

In this case, `192.168.2.122` is the IP address of a Linux machine that is reachable from the Ansible control host. We can now use the command line option to test Ansible and the `hosts` file:

```
$ ansible -i hosts 192.168.2.122 -m ping
192.168.2.122 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

By default, Ansible assumes that the same user executing the playbook exists on the remote host. For example, I am executing the playbook as `echou` locally; the same user also exists on my remote host. If you want to execute as a different user, you can use the `-u` option when executing, that is, `-u REMOTE_USER`.

The previous command line execution shown reads in the host file as the inventory file and executes the ping module on the host with the IP address of `192.168.2.122`. Ping (`http://docs.ansible.com/ansible/ping_module.html`) is a trivial test module that connects to the remote host, verifies a usable Python installation, and returns the output `pong` upon success.

You may take a look at the ever-expanding module list (`http://docs.ansible.com/ansible/list_of_all_modules.html`) if you have any questions about the use of existing modules that were shipped with Ansible.

If the host's key is not in the control node's `~/.ssh/known_hosts` file, you will get a prompt. Answer '`yes`' to add the host. You can disable this by checking on `/etc/ansible/ansible.cfg or ~/.ansible.cfg` with the following code:

```
[defaults]
host_key_checking = False
```

Now that we have validated the inventory file and the Ansible package, we can make our first playbook.

Our first playbook

Playbooks are Ansible's blueprint to describe what you would like to do to the hosts, using modules. This is where we will be spending the majority of our time as operators when working with Ansible. If we use an analogy of building a tree house with Ansible, the playbook will be your manual, the modules will be your tools, while the inventory will be the components that you will be working on when using the tools.

The playbook is designed to be human-readable, and is in YAML format. We will look at the common syntax used in the Ansible architecture section. For now, our focus is to run an example playbook to get the look and feel of Ansible.

Originally, YAML was said to mean Yet Another Markup Language, but now `http://yaml.org/` has repurposed the acronym to be YAML Ain't Markup Language.

Let's look at this simple six-line playbook, `df_playbook.yml`:

```
---
- hosts: 192.168.2.122

  tasks:
      - name: check disk usage
        shell: df > df_temp.txt
```

In a playbook, there can be one or more plays. In this case, we have one play (lines two to six). In any play, we can have one or more tasks. In our example play, we have just one task (lines four to six). The `name` field specifies the purpose of the task in a human-readable format and the `shell` module was used. The module takes one argument of `df`. The `shell` module reads in the command in the argument and executes it on the remote host. In this case, we execute the `df` command to check the disk usage and copy the output to a file named `df_temp.txt`.

We can execute the playbook via the following code:

```
$ ansible-playbook -i hosts df_playbook.yml
PLAY [192.168.2.122] ***********************************************
*******
TASK [setup] *******************************************************
*******
ok: [192.168.2.122]
TASK [check disk usage] ********************************************
```

```
*******

changed: [192.168.2.122]

PLAY RECAP *********************************************************************
*******

192.168.2.122              : ok=2    changed=1    unreachable=0
failed=0
```

If you log into the managed host (`192.168.2.122`, in this example), you will see that the `df_temp.txt` file contains the output of the `df` command. Neat, huh?

You may have noticed that there were actually two tasks executed in our output, even though we only specified one task in the playbook; the setup module is automatically added by default. It is executed by Ansible to gather information about the remote host, which can be used later on in the playbook. For example, one of the facts that the setup module gathers is the host's operating system type. What is the purpose of gathering facts about the remote target? You can use this information as a conditional for additional tasks in the same playbook. For example, the playbook can contain additional tasks to install packages. By knowing the operating system type, Ansible can install packages with `apt` for Debian-based hosts and `yum` for Red Hat-based hosts.

If you are curious about the output of a setup module, you can find out what information Ansible gathers via `$ ansible -i hosts <host> -m setup`.

Underneath the hood, there are actually a few things that have happened in relation to our simple task. When the playbook was executed, the control node copied the Python module to the remote host, executed the module, copied the module output to a temporary file, then captured the output and deleted the temporary file. For now, we can probably safely ignore these underlying details until we need them.

It is important that we fully understand the simple process that we have just gone through because we will be referring back to these elements later in this chapter. I purposely chose a server example to be presented here, because this will make more sense as we dive into the networking modules when we need to deviate from them (remember we mentioned the Python interpreter is most likely not available on the network equipment we want to manage).

Congratulations on executing your first Ansible playbook! We will look more into the Ansible architecture, but for now 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 own unique features and models; there is no one right framework that fits all organizations. In this section, I would like to list some of the advantages of Ansible over other frameworks and why I think this is a good tool for network automation.

I will list the advantages of Ansible without comparing them to other frameworks. 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. I believe it is the combination of all the following features and philosophy 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 the 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. This mentality has changed a bit in the last few years, but overall, SSH is the common denominator for all network equipment while configuration management agent support is not. As you will remember 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, in my opinion, 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 it comes to 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 over and over again does not change the system after the first time. Ansible aims to be idempotent, which is good for network operations that require a certain order of operations.

The advantage of idempotence is best compared to the Pexpect and Paramiko scripts that we have written. Remember that these scripts were written to push out commands as if an engineer was sitting at the terminal. If you were to execute the script 10 times, the script will make 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 means 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 any extra overhead.

Simple and extensible

Ansible is written in Python and uses YAML for the playbook language, both of which are considered 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 or DSL to learn for Ansible because YAML and Python are both widely used as general-purpose languages.

As you can see from the previous example, even if you have not seen YAML before, it is easy to accurately guess what the playbook is trying to do. Ansible also uses Jinja2 as a template engine, which is a common tool used by Python web frameworks such as Django and Flask, so the knowledge is transferable.

I cannot stress enough about the extensibility of Ansible. As illustrated by the preceding example, Ansible starts out 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 industry started to adapt Ansible, the network became a topic that started to get more attention.

The right people and team were hired at Ansible, network professionals started to get involved, and customers started to demand vendors for support. Starting with Ansible 2.0, network automation has become a first-class citizen alongside server management. The ecosystem is alive and well, with continuous improvement in each of the releases.

Just like the Python community, the Ansible community is friendly, and its attitude is inclusive of new members and ideas. I have first-hand experience of being a noob and trying to make sense of contribution procedures and wishing to write modules to be merged upstream. I can testify to the fact that I felt welcomed and respected for my opinions at all times.

The simplicity and extensibility really speak well for future-proofing. The technology world is evolving fast, and we are constantly trying to adapt to it. Wouldn't it be great to learn a technology once and continue to use it, regardless of the latest trend? Obviously, nobody has a crystal ball to accurately predict the future, but Ansible's track record speaks well for future technology adaptation.

Network vendor support

Let's face it, we don't live in a vacuum. There is a running joke in the industry that the OSI layer should include a layer 8 (money) and 9 (politics). Every day, we need to work with network equipment made by various vendors.

Take API integration as an example. We saw the difference between the Pexpect and API approach in previous chapters. API clearly has an upper hand in terms of network automation. However, the API interface does not come cheap for the vendors. Each vendor needs to invest time, money, and engineering resources to make the integration happen. The willingness of the vendor to support a technology matters greatly in our world. Luckily, all the major vendors support Ansible, as clearly indicated by the ever-increasing available network modules (`http://docs.ansible.com/ansible/list_of_network_modules.html`).

Why do vendors support Ansible more than other automation tools? Being agentless certainly helps, since having SSH as the only dependency greatly lowers the bar of entry. Engineers who have been on the vendor side know that the feature request process is usually months long and many hurdles have to be jumped over. Any time a new feature is added, it means more time spent on regression testing, compatibility checking, integration reviews, and much more. Lowering the bar of entry is usually the first step in getting vendor support.

The fact that Ansible is based on Python, a language liked by many networking professionals, is another great propeller for vendor support. For vendors such as Juniper and Arista who have already made investments in PyEZ and Pyeapi, they can easily leverage the existing Python modules and quickly integrate their features into Ansible. As you will see in *Chapter 5*, *The Python Automation Framework – Beyond Basics*, we can use our existing Python knowledge to easily write our own modules.

Ansible already had a large number of community-driven modules before it focused on networking. The contribution process is somewhat baked and established, or as baked as an open source project can be. The core Ansible team is familiar with working with the community for submissions and contributions.

Another reason for the increased network vendor support also has to do with Ansible's ability to give vendors the ability to express their own strengths in the module context. We will see in the coming section that, besides SSH, the Ansible module can also be executed locally and communicate with these devices by using an API. This ensures that vendors can express their latest and greatest features as soon as they make them available through the API. In terms of network professionals, this means that you can use the cutting-edge features to select the vendors when you are using Ansible as an automation platform.

We have spent a relatively large portion of space discussing vendor support because I feel that this is often an overlooked part in the Ansible story. Having vendors willing to put their weight behind the tool means you, the network engineer, can sleep at night knowing that the next big thing in networking will have a high chance of Ansible support, and you are not locked into your current vendor as your network needs grow.

Now that we've covered the advantages of Ansible, let's explore its architecture.

The Ansible architecture

The Ansible architecture consists of playbooks, plays, and tasks. Take a look at `df_playbook.yml`, which we used previously:

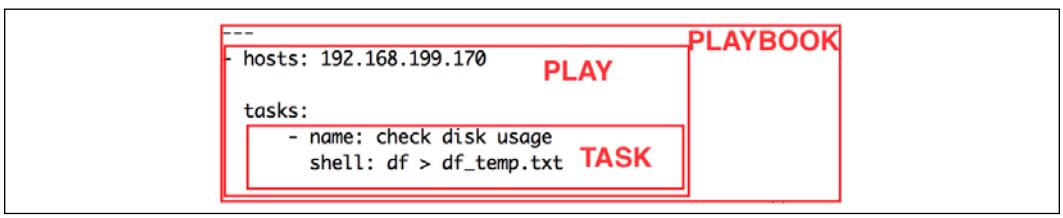


Figure 4: An Ansible playbook

The whole file is called a playbook, which contains one or more plays. Each play can consist of one or more tasks. In our simple example, we only have one play, which contains a single task. In this section, we will take a look at the following components and terms related to Ansible, some of which we have already seen:

- **YAML**: This format is extensively used in Ansible to express playbooks and variables.
- **Inventory**: The inventory is where you can specify and group hosts in your infrastructure. You can also optionally specify host and group variables in the inventory file.
- **Variables**: Each network device is different. It has a different hostname, IP, neighbor relations, and so on. Variables allow for a standard set of plays while still accommodating these differences.
- **Templates**: Templates are nothing new in networking. In fact, you are probably using one without thinking of it as a template. What do we typically do when we need to provision a new device or replace a **return merchandise authorization** (**RMA**)? We copy the old configuration over and replace the differences such as the hostname and the loopback IP addresses. Ansible standardizes the template formatting with Jinja2, which we will dive deeper into later on.

In *Chapter 5*, *The Python Automation Framework – Beyond Basics*, we will cover some more advanced topics such as conditionals, loops, blocks, handlers, playbook roles, and how they can be included with network management.

YAML

YAML is the syntax used for Ansible playbooks and some other files. The official YAML documentation contains the full specifications of the syntax. Here is a compact version as it pertains to the most common usage for Ansible:

- A YAML file starts with three dashes (---)
- Whitespace indentation is used to denote structures when they are lined up, just like Python
- Comments begin with the hash (#) sign
- List members are denoted by a leading hyphen (-), with one member per line
- Lists can also be denoted by square brackets ([]), with elements separated by a comma (,)
- Dictionaries are denoted by key: value pairs, with a colon for separation

- Dictionaries can be denoted by curly braces, with elements separated by a comma (,)
- Strings can be unquoted, but can also be enclosed in double or single quotes

As you can see, YAML maps well into JSON and Python datatypes. If I were to rewrite `df_playbook.yml` in `df_playbook.json`, this is what it would look like:

```
[
  {
    "hosts": "192.168.199.170",
    "tasks": [
      {"name": "check disk usage"},
      {"shell": "df > df_temp.txt"}
    ]
  }
]
```

This is obviously not a valid playbook, but serves as an aid in helping to understand the YAML formats while using the JSON format as a comparison. Most of the time, comments (`#`), lists (`-`), and dictionaries (`key: value`) are what you will see in a playbook.

Inventories

By default, Ansible looks at the `/etc/ansible/hosts` file for hosts specified in your playbook. As mentioned previously, I find it more expressive to specify the host file via the `-i` option. This is what we have been doing up to this point. To expand on our previous example, we can write our inventory host file as follows:

```
[ubuntu]
192.168.2.122
[nexus]
172.16.1.142
172.16.1.143

[nexus:vars]
username=cisco
password=cisco

[nexus_by_name]
```

```
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143
```

As you may have guessed, the square bracket headings specify group names, so later on in the playbook we can point to this group. For example, in `cisco_1.yml` and `cisco_2.yml`, I can act on all of the hosts specified under the `nexus` group to the group name of `nexus`:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local
  <skip>
```

A host can exist in more than one group. The group can also be nested as `children`:

```
[cisco]
router1
router2

[arista]
switch1
switch2

[datacenter:children]
cisco
arista
```

In the previous example, the datacenter group includes both the `cisco` and `arista` members with four total devices.

We will discuss variables in the next section. There are a few places you can declare your variables, in fact, you have already seen some usage of it. In our first inventory file example, we have declared variables both for hosts and groups in the inventory file. [`nexus:vars`] specifies variables for the whole nexus group. The `ansible_host` variable declares variables for each of the hosts on the same line.

For more information on the inventory file, check out the official documentation (`http://docs.ansible.com/ansible/intro_inventory.html`).

Variables

We discussed variables a bit in the previous section. Why do we need variables? Because our managed nodes are not exactly alike, we need to accommodate the differences via variables. Variable names should be letters, numbers, and underscores, and should always start with a letter. Variables are commonly defined in three locations:

- The playbook
- The inventory file
- Separate files to be included in files and roles

Let's look at an example of defining variables in a playbook, `cisco_1.yml`:

```
---
- name: Configure SNMP Contact
  hosts: "nexus"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ inventory_hostname }}"
      username: cisco
      password: cisco
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
        var: output
```

In the playbook, you can see the `cli` variable declared under the `vars` section, which is being referenced using the double curly bracket ("{{ `cli` }}") in the task of `nxos_snmp_contact`.

For more information on the `nxos_snmp_contact` module, check out the online documentation (`http://docs.ansible.com/ansible/nxos_snmp_contact_module.html`).

To reference a variable, you can use the Jinja2 templating system convention of a double curly bracket. You don't need to put quotes around the curly bracket unless you are starting a value with it, but I typically find it easier to just always put quotes around the curly brackets so I do not need to think about the differences.

You may have also noticed the {{ `inventory_hostname` }} reference, which is not declared in the playbook. It is one of the default variables that Ansible provides for you automatically. It references to the IP address or a DNS-resolvable hostname in the inventory file. Sometimes in the documentation these variables are referred to as 'magic' variables.

There are not many magic variables, and you can find the list in the documentation (`http://docs.ansible.com/ansible/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts`).

We have declared variables in an inventory file in the previous section:

```
[nexus:vars]
username=cisco
password=cisco

[nexus_by_name]
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143
```

To use the variables in the inventory file instead of declaring them in the playbook, let's add the group variables for [`nexus_by_name`] in the host file:

```
[nexus_by_name]
switch1 ansible_host=172.16.1.142
switch2 ansible_host=172.16.1.143

[nexus_by_name:vars]
username=cisco
password=cisco
```

Then, modify the playbook to match what we can see here in `cisco_2.yml`, to reference the variables:

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
        var: output
```

Notice that in this example, we are referring to the [`nexus_by_name`] group in the inventory file, the `ansible_host` host variable, and the `username` and `password` group variables.

By offloading the username and password in a separate file, we can make the file write-protected with a better security posture.

To see more examples of variables, check out the Ansible documentation (`http://docs.ansible.com/ansible/playbooks_variables.html`).

To access complex variable data that's provided in a nested data structure, you can use two different notations. Noted in the `nxos_snmp_contact` task, we registered the output in a variable and displayed it using the debug module.

You will see something like the following during playbook execution:

```
$ ansible-playbook -i hosts cisco_2.yml
TASK [show output] ******************************************************
*******
ok: [switch1] => {
    "output": {
        "changed": false,
        "end_state": {
            "contact": "TEST_1"
        },
        "existing": {
            "contact": "TEST_1"
        },
        "proposed": {
            "contact": "TEST_1"
        },
        "updates": []
    }
}
```

In order to access the nested data, we can use the following notation, as specified in `cisco_3.yml`:

```
    tasks:
      - name: configure snmp contact
        nxos_snmp_contact:
          contact: TEST_1
          state: present
          provider: "{{ cli }}"

        register: output

      - name: show output in output["end_state"]["contact"]
        debug:
          msg: '{{ output["end_state"]["contact"] }}'

      - name: show output in output.end_state.contact
        debug:
          msg: '{{ output.end_state.contact }}'
```

You will receive just the value indicated:

```
$ ansible-playbook -i hosts cisco_3.yml
TASK [show output in output["end_state"]["contact"]]
*************************
ok: [switch1] => {
    "msg": "TEST_1"
}
ok: [switch2] => {
    "msg": "TEST_1"
}

TASK [show output in output.end_state.contact] **********************
**********
ok: [switch1] => {
    "msg": "TEST_1"
}
ok: [switch2] => {
    "msg": "TEST_1"
}
```

Lastly, we mentioned variables can also be stored in a separate file. To see how we can use variables in a role or included file, we should get a few more examples under our belt, because they are a bit complicated to start with. We will see more examples of roles in *Chapter 5*, *The Python Automation Framework – Beyond Basics*.

Templates with Jinja2

In the previous section, we used variables with the Jinja2 syntax of {{ `variable` }}. While you can do a lot of complex things in Jinja2, luckily, we only need some of the basic things to get started with Ansible templates.

Jinja2 (`http://jinja.pocoo.org/`) is a full-featured, powerful template engine that originated in the Python community. It is widely used in Python web frameworks such as Django and Flask.

For now, it is enough to just keep in mind that Ansible utilizes Jinja2 as the template engine. We will revisit the topics of Jinja2 filters, tests, and lookups as the situations call for them.

You can find more information on the Ansible Jinja2 template here: `http://docs.ansible.com/ansible/playbooks_templating.html`.

This concludes our rundown of Ansible's architecture. In the next section, we'll look at Ansible networking modules where most of the network tasks we will encounter will be handled in Ansible.

Ansible networking modules

Ansible was originally made for managing nodes with full operating systems such as Linux and Windows before it was extended to support network equipment. You may have already noticed the subtle differences in playbooks that we have used so far for network devices, such as the lines of `gather_facts: false` and `connection: local`; we will take a closer look at the differences in the following sections.

Ansible provides nicely written documentation on 'How Network Automation is Different': `https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html`.

Local connections and facts

Ansible modules are Python code that's executed on the remote host by default. Because of the fact that most network equipment does not expose Python directly, or it simply does not contain Python, we are almost always executing the playbook locally on the control node. This means that the playbook is interpreted locally first and commands or configurations are pushed out later, as needed.

Recall that in our server example, the remote host facts were gathered via the setup module, which was added by default. Since we are executing the playbook locally, the setup module will gather the facts on the localhost instead of the remote host. This is certainly not needed, therefore when the connection is set to local, we can reduce this unnecessary step by setting the fact gathering to no or false. Starting from release 2.5, there are fact-gathering modules specific to each platform. You can take a look at the `fact-demo.yml` example: `https://docs.ansible.com/ansible/latest/network/user_guide/network_best_practices_2.5.html#step-2-creating-the-playbook`.

Because network modules are executed locally, for those modules that offer a backup option, the files are backed up locally on the control node as well.

One of the most important changes introduced in Ansible 2.5 was the introduction of different network communication protocols (`https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html#multiple-communication-protocols`).

The connection method now includes `network_cli`, `netconf`, `httpapi`, and `local`. If the network device uses CLI over SSH, you indicate the connection method as `network_cli` in one of the device variables. It is good to be aware of both of the pre-2.5 as well as the post-2.5 connection syntax. In general, you will find the post-2.5 syntax more streamlined and concise.

Provider arguments

As we have seen from *Chapter 2*, *Low-Level Network Device Interactions*, and *Chapter 3*, *APIs and Intent-Driven Networking*, network equipment can be connected via both SSH and API, depending on the platform and software release. All core networking modules implement a `provider` argument, which is a collection of arguments used to define how to connect to the network device. Some modules only support `cli` while some support other values, for example, Arista supports eAPI and Cisco supports NX-API on the Nexus platform.

Starting with Ansible 2.5, the recommended way to specify the transport method is by using the `connection` variable. You will start to see the provider parameter being gradually phased out from future Ansible releases. Using the `ios_command` module as an example, `https://docs.ansible.com/ansible/latest/modules/ios_command_module.html#ios-command-module`, the provider parameter still works, but is being labeled as deprecated. We will see an example of this later in this chapter.

Some of the basic arguments supported by the `provider` transport are as follows:

- **host**: This defines the remote host
- **port**: This defines the port to connect to
- **username**: This is the username to be authenticated
- **password**: This is the password to be authenticated
- **transport**: This is the type of transport for the connection
- **authorize**: This enables privilege escalation for devices that require it
- **auth_pass**: This defines the privilege escalation password

As you can see, not all arguments need to be specified in the `provider` variable. For example, for our previous playbooks, our user always has the admin privilege when logged in, therefore we do not need to specify the `authorize` or the `auth_pass` arguments.

These arguments are just variables, so they follow the same rules for variable precedence. For example, let's say I change `cisco_3.yml` to `cisco_4.yml` and observe the following precedence:

```
---
- name: Configure SNMP Contact
  hosts: "nexus_by_name"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli

  tasks:
    - name: configure snmp contact
      nxos_snmp_contact:
        contact: TEST_1
        state: present
        username: cisco123 #new
        password: cisco123 #new
        provider: "{{ cli }}"

      register: output

    - name: show output in output["end_state"]["contact"]
      debug:
        msg: '{{ output["end_state"]["contact"] }}'

    - name: show output in output.end_state.contact
      debug:
        msg: '{{ output.end_state.contact }}'
```

The username and password defined at the task level will override the username and password at the playbook level. I will receive the following error when trying to connect because the user does not exist on the device:

```
PLAY [Configure SNMP Contact] ***************************************
*******


TASK [configure snmp contact] ***************************************
*******
fatal: [switch2]: FAILED! => {"changed": false, "failed": true, "msg":
"failed to connect to 172.16.1.143:22"}
fatal: [switch1]: FAILED! => {"changed": false, "failed": true, "msg":
```

```
"failed to connect to 172.16.1.142:22"}
to retry, use: --limit @/home/echou/Mastering_Python_Networking_third_
edition/Chapter04/cisco_4.retry


PLAY RECAP *********************************************************
*******
switch1    : ok=0    changed=0    unreachable=0    failed=1
switch2    : ok=0    changed=0    unreachable=0    failed=1
```

In the next section, we will dive deeper into examples for managing Cisco devices.

The Ansible Cisco example

Cisco's support in Ansible is categorized by the operating systems IOS, IOS-XR, and NX-OS. We have already seen a number of NX-OS examples, so in this section let's try to manage IOS-based devices.

Our host file will consist of two hosts, `ios-r1` and `ios-r2`:

```
[ios-devices]
ios-r1 ansible_host=172.16.1.134
ios-r2 ansible_host=172.16.1.135


[ios-devices:vars]
username=cisco
password=cisco
```

Our playbook, `cisco_5.yml`, will use the `ios_command` module to execute arbitrary `show commands`:

```
---
- name: IOS Show Commands
  hosts: "ios-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      transport: cli
```

```
    tasks:
      - name: ios show commands
        ios_command:
          commands:
            - show version | i IOS
            - show run | i hostname
          provider: "{{ cli }}"

        register: output

      - name: show output in output["end_state"]["contact"]
        debug:
            var: output
```

The result is what we would expect as the `show version` and `show run` output:

```
$ ansible-playbook -i hosts cisco_5.yml

PLAY [IOS Show Commands] ********************************************
*******

TASK [ios show commands] *********************************************
*******
ok: [ios-r1]
ok: [ios-r2]

TASK [show output in output["end_state"]["contact"]]
**************************
ok: [ios-r1] => {
    "output": {
        "changed": false,
        "stdout": [
            "Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\nROM: Bootstrap program is
IOSv\nCisco IOSv (revision 1.0) with  with 460033K/62464K bytes of
memory.",
            "hostname iosv-1"
        ],
        "stdout_lines": [
            [
                "Cisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)",
                "ROM: Bootstrap program is IOSv",
                "Cisco IOSv (revision 1.0) with  with 460033K/62464K
bytes of memory."
            ],
```

```
            [
                "hostname iosv-1"
            ]
        ],
        "warnings": []
    }
}
ok: [ios-r2] => {
    "output": {
        "changed": false,
        "stdout": [
            "Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M),
Version 15.6(3)M2, RELEASE SOFTWARE (fc2)\nROM: Bootstrap program is
IOSv\nCisco IOSv (revision 1.0) with  with 460033K/62464K bytes of
memory.",
            "hostname iosv-2"
        ],
        "stdout_lines": [
            [
                "Cisco IOS Software, IOSv Software (VIOS-
ADVENTERPRISEK9-M), Version 15.6(3)M2, RELEASE SOFTWARE (fc2)",
                "ROM: Bootstrap program is IOSv",
                "Cisco IOSv (revision 1.0) with  with 460033K/62464K
bytes of memory."
            ],
            [
                "hostname iosv-2"
            ]
        ],
        "warnings": []
    }
}

PLAY RECAP *********************************************************
*******
ios-r1     : ok=2    changed=0    unreachable=0    failed=0
ios-r2     : ok=2    changed=0    unreachable=0    failed=0
```

I want to point out a few things illustrated by this example:

- The playbook between NX-OS and IOS is largely identical
- The syntax `nxos_snmp_contact` and `ios_command` modules follow the same pattern, with the only difference being the argument for the modules
- The IOS version of the devices is pretty old with no understanding of API, but the modules still have the same look and feel

As you can see from the preceding example, once we have the basic syntax down for the playbooks, the subtle difference relies on the different modules for the task we would like to perform.

Ansible 2.8 playbook example

We have briefly talked about the addition of network connection changes in Ansible playbooks, starting with version 2.5 and beyond. Along with the changes, Ansible also released a network best practices document: `https://docs.ansible.com/ansible/latest/network/user_guide/network_best_practices_2.5.html`. Let's build an example based on the best practices guide. Since there are multiple files involved in this example, the files are grouped into a subdirectory named `ansible_2-8_example` with the course code files.

Either use the system installed version or switch back to Ansible version 2.8 using the Git source code as illustrated before:

```
$ ansible --version
ansible 2.8.5
```

In our previous examples, we mainly used just the inventory host file to contain both the inventory information as well as the associated variables. In this example, we will offload the variables to a separate directory named `host_vars`:

```
$ tree.
.
├── hosts
├── host_vars
│   ├── iosv-1
│   └── iosv-2
└── my_playbook.yml
1 directory, 4 files
```

Our inventory file is reduced to the group and the name of the hosts:

```
$ cat hosts
[ios-devices]
iosv-1
iosv-2
```

In the `host_vars` directory there are two files. Each corresponds to the name specified in the inventory file:

```
$ ls host_vars/
iosv-1
iosv-2
```

The variable file for the hosts contains what was previously included in the `CLI` variable. The additional variable of `ansible_connection` specifies `network_cli` as the transport:

```
$ cat host_vars/iosv-1
---
ansible_host: 172.16.1.134
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


$ cat host_vars/iosv-2
---
ansible_host: 172.16.1.135
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
```

Our playbook will use the `ios_config` module with the backup option enabled. Notice the use of the `when` condition in this example, so that if there are other hosts with a different operating system, this task will not be applied:

```
$ cat ansible2-8_playbook.yml
---
- name: Chapter 4 Ansible 2.8 Best Practice Demonstration
```

```
  connection: network_cli
  gather_facts: false
  hosts: all
  tasks:
    - name: backup
      ios_config:
        backup: yes
      register: backup_ios_location
      when: ansible_network_os == 'ios'
```

When the playbook is run, a new backup folder will be created with the configuration backed up for each of the hosts:

```
$ ansible-playbook -i hosts ansible2-8_playbook.yml
PLAY [Chapter 4 Ansible 2.8 Best Practice Demonstration] ***************
*****************
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
```

We can see the newly created backup directory with two files inside:

```
$ tree
.
├── ansible2-8_playbook.yml
├── backup
│   ├── iosv-1_config.2019-09-24@10:40:36
│   └── iosv-2_config.2019-09-24@10:40:36
├── hosts
└── host_vars
    ├── iosv-1
    └── iosv-2
2 directories, 6 files
```

```
$ head -20 backup/iosv-1_config.2019-09-24@10\:40\:36
Building configuration...

Current configuration : 4598 bytes
!
! Last configuration change at 17:02:29 UTC Sun Sep 22 2019
!
version 15.6
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
!
hostname iosv-1
!
boot-start-marker
boot-end-marker
!
!
vrf definition Mgmt-intf
 !
```

This example illustrates the `network_connection` variable and the recommended structure based on Ansible network best practices. We will look at offloading variables into the `host_vars` directory and conditionals in *Chapter 5*, *The Python Automation Framework – Beyond Basics*. This structure can also be used for the Juniper and Arista examples in this chapter. For the different devices, we will just use different values for `network_connection` as we will see from the Juniper examples in the next section.

The Ansible Juniper example

The Ansible Juniper module requires the Juniper PyEZ package and NETCONF. If you have been following the API example in *Chapter 3*, *APIs and Intent-Driven Networking*, you are good to go. If not, refer back to that section for installation instructions as well as some test script to make sure PyEZ works. The Python package called `jxmlease` is also required:

```
(venv) $ pip install jxmlease
```

In the host file, we will specify the device and connection variables:

```
[junos_devices]
J1 ansible_host=192.168.24.252


[junos_devices:vars]
username=juniper
password=juniper!
```

In our Juniper playbook, we will use the `junos_facts` module to gather basic facts for the device. This module is equivalent to the `setup` module and will come in handy if we need to take action depending on the returned value. Note the different values of `transport` and `port` in the example here:

```
---
- name: Get Juniper Device Facts
  hosts: "junos_devices"
  gather_facts: false
  connection: local

  vars:
    netconf:
      host: "{{ ansible_host }}"
      username: "{{ username }}"
      password: "{{ password }}"
      port: 830
      transport: netconf

  tasks:
    - name: collect default set of facts
      junos_facts:
        provider: "{{ netconf }}"

      register: output

    - name: show output
      debug:
          var: output
```

When executed, you will receive this output from the `Juniper` device:

```
PLAY [Get Juniper Device Facts]
*********************************************
```

```
TASK [collect default set of facts]
****************************************** ok: [J1]

TASK [show output]
************************************************************
ok: [J1] "
<skip>

PLAY RECAP
********************************************************************* J1
: ok=2 changed=0 unreachable=0 failed=0
```

Now that we have seen examples of managing Cisco and Juniper devices, let us compare them with some examples where the target machines are Arista devices.

The Ansible Arista example

The final playbook example we will look at will be the Arista command module. At this point, we are quite familiar with our playbook syntax and structure. The Arista device can be configured to use transport using `cli` or `eapi`, so, in this example, we will use `cli`.

This is the host file:

```
[eos-devices]
arista1 ansible_host=192.168.199.158
```

The playbook is also similar to what we have seen previously:

```
---
- name: EOS Show Commands
  hosts: "eos_devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "arista"
      password: "arista"
      authorize: true
      transport: cli
```

```
tasks:
  - name: eos show commands
    eos_command:
      commands:
        - show version | i Arista
      provider: "{{ cli }}"

    register: output

  - name: show output
    debug:
        var: output
```

As you can see from the Arista example, it is not that different from Cisco or Juniper in terms of structure. This speaks to the strength of using Ansible; even with a new vendor that we have never worked with before, using Ansible can provide a layer of structure to be followed.

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 was originally constructed to manage servers and was later extended to network devices; therefore, we took a look at a server example. Then, we compared and contrasted the differences when it came to network management playbooks. Later, we looked at the example playbooks for Cisco IOS, Juniper JUNOS, and Arista EOS devices. We also looked at the best practices recommended by Ansible if you are using the latest Ansible release, version 2.8.

In *Chapter 5*, *The Python Automation Framework – Beyond Basics*, we will leverage the knowledge we gained in this chapter and start to look at some of the more advanced features of Ansible, such as group variables, templates, and conditional statements.

5
The Python Automation Framework – Beyond Basics

In *Chapter 4*, *The Python Automation Framework – Ansible Basics*, we looked at some of the basic structures to get Ansible up and running. We worked with Ansible inventory files, variables, and playbooks. We also looked at some examples of using network modules for Cisco, Juniper, and Arista devices.

In this chapter, we will further build on the knowledge we have gained from the previous chapters and dive deeper into the more advanced topics of Ansible. Many books have been written about Ansible, and there is more to Ansible than we can cover in two chapters. The goal here is to introduce the majority of the features and functions of Ansible that I believe you will need as a network engineer, and shorten the learning curve as much as possible.

It is important to point out that if you were not clear on some of the points made in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, now is a good time to go back and review them as they are a prerequisite for this chapter.

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

- Ansible conditionals
- Ansible loops
- Templates
- Group and host variables
- The Ansible Vault
- Ansible roles
- Writing your own module

We have a lot of ground to cover, so let's get started!

Lab preparation

For this chapter, we will continue to use the same lab topology we used in *Chapter 4*, *The Python Automation Framework – Ansible Basics*. We will also follow the best practice suggested with offloading the host variables in a `host_vars` directory. We will also use a few options on `ansible.cfg` such as disabling `host_key_checking` and Python interpreter discovery.

For more information on the `ansible.cfg` options: `host_key_checking`: `https://docs.ansible.com/ansible/latest/user_guide/intro_getting_started.html#host-key-checking`; and Python interpreter discovery: `https://docs.ansible.com/ansible/latest/reference_appendices/config.html`.

We will follow the Ansible version and file structure as illustrated below:

```
$ ansible --version
ansible 2.8.5
$ tree host_vars/
host_vars/
├── ios-r1
└── ios-r2
<skip>

$ cat ansible.cfg
[defaults]
host_key_checking=False
interpreter_python=auto
```

Ansible conditionals

Ansible conditionals are similar to conditional statements in programming languages. In *Chapter 1*, *Review of TCP/IP Protocol Suite and Python*, we saw that Python uses conditional statements to execute only a section of the code by using `if`, `then`, or `while` statements. In Ansible, it 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 section, we will discuss the `when` clause, which is supported for all modules, as well as unique conditional states that are supported in Ansible networking command modules. 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 `when` clause in action.

The when clause

The `when` clause is useful when you need to check the output of a variable or a play execution result and act accordingly. We saw a quick example of the `when` clause in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, when we looked at the Ansible 2.8 best practices structure. If you recall, the task only ran when the network operating system of the device was the Cisco IOS. Let's look at another example of its use in `chapter5_1.yml`:

```
---
- name: IOS Command Output
  hosts: "ios-devices"
  gather_facts: false
  connection: network_cli

  tasks:
    - name: show hostname
      ios_command:
        commands:
            - show run | i hostname

      register: output

    - name: show output with when conditions
      when: '"iosv-2" in "{{ output.stdout }}"'
      debug:
        msg: '{{ output }}'
```

We have seen all the elements in this playbook before in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, up to the end of the first task. For the second task in the play, we are using the `when` clause to check whether the output contains the `iosv-2` keyword. If true, we will proceed to the task, which is using the debug module to display the output. When the playbook is run, we will see the following output:

```
$ ansible-playbook -i hosts chapter5_1.yml
<skip>
TASK [show output with when conditions] ********************************
*****************************************
skipping: [ios-r1]
ok: [ios-r2] => {
    "msg": {
        <skip>
        "failed": false,
        "stdout": [
            "hostname iosv-2"
        ],
        "stdout_lines": [
            [
                "hostname iosv-2"
            ]
        ]
    }
}
```

We can see that the `iosv-r1` device is skipped from the output because the clause did not pass. We can further expand this example in `chapter5_2.yml` to only apply certain configuration changes when the condition is met:

```
   <skip>
     tasks:
       - name: show hostname
         ios_command:
           commands:
               - show run | i hostname

         register: output

       - name: config example
```

```
        when: '"iosv-2" in "{{ output.stdout }}"'
        ios_config:
          lines:
               - logging buffered 30000
```

We can see the execution output here:

```
$ ansible-playbook -i hosts chapter5_2.yml
<skip>
TASK [config example] ***********************************************
*************************
skipping: [ios-r1]


changed: [ios-r2]


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

Again, note in the execution output that `ios-r2` was the only change applied while `ios-r1` was skipped. In this case, the logging buffer size was only changed on `ios-r2`:

```
iosv-2#sh run | i logging
logging buffered 30000
```

The `when` clause is also very useful in situations when the setup or facts module is used – you can act based on some of the `facts` that were gathered initially. For example, the following statement will ensure that only the Ubuntu host with major release `16` or greater will be acted upon by placing a conditional statement in the clause:

```
when: ansible_os_family == "Debian" and ansible_lsb.major_release|int >=
16
```

For more conditionals, check out the Ansible conditionals documentation (`http://docs.ansible.com/ansible/playbooks_conditionals.html`).

In the next section, we will take a look at how Ansible gathers network device facts and uses them in the context of network playbooks.

Ansible network facts

Prior to 2.5, Ansible networking shipped with a number of vendor-specific fact modules. The network fact modules exist prior to version 2.5, but the naming and usage was different between vendors. Starting with version 2.5, Ansible started to standardize its network fact modules. The Ansible network fact modules gather information from the 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 a pretty big milestone for Ansible networking modules, as it does a lot of the heavy lifting for you to abstract the fact-gathering process by default.

Let's use the same structure we saw in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, Ansible 2.8 best practices, but expand upon it to see how the `ios_facts` module was used to gather facts. As a review, our inventory file contains two IOS hosts with the host variables residing in the `host_vars` directory:

```
$ cat hosts
[ios-devices]
iosv-1
iosv-2
$ cat host_vars/iosv-1
---
ansible_host: 172.16.1.134
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
```

Our playbook will have three tasks. The first task will use the `ios_facts` module to gather facts for both of our network devices. The second task will display certain facts gathered and stored for each of the two devices. You will see that the facts we displayed were the default `ansible_net` facts, as opposed to a registered variable from the first task.

The third task will display all the facts we collected for the `iosv-1` host:

```
$ cat ios_facts_playbook.yml
---
- name: Chapter 5 Ansible 2.8 network facts
  connection: network_cli
  gather_facts: false
  hosts: all
  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 a host
      debug:
        var: hostvars['iosv-1']
```

When we run the playbook, you can see that the results for the first two tasks were what we would have expected:

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

PLAY [Chapter 5 Ansible 2.8 network facts] ********************************
*********************************************

TASK [Gathering facts via ios_facts module] *******************************
*********************************************

ok: [iosv-2]

ok: [iosv-1]

TASK [Display certain facts] **********************************************
************************************************

ok: [iosv-1] => {
    "msg": "The hostname is iosv-1 running 15.6(3)M2"
}
ok: [iosv-2] => {
    "msg": "The hostname is iosv-2 running 15.6(3)M2"
}
```

The third task will display all the network device facts gathered for IOS devices. There is a ton of information that has been gathered for IOS devices that can help with your networking automation needs; we can see them in the third task:

```
TASK [Display all facts for a host] ************************************
*************************************************
ok: [iosv-1] => {
    "hostvars['iosv-1']": {
        "ansbile_become": true,
        "ansible_become_method": "enable",
        "ansible_become_pass": "cisco",
        "ansible_check_mode": false,
        "ansible_connection": "network_cli",
        "ansible_diff_mode": false,
        "ansible_facts": {
            "discovered_interpreter_python": "/usr/bin/python",
            "net_all_ipv4_addresses": [
                "10.0.0.13",
                "10.0.0.5",
                "10.0.0.17",
                "172.16.1.134",
                "192.168.0.1"
            ],
            "net_all_ipv6_addresses": [],
            "net_api": "cliconf",
            "net_filesystems": [
                "flash0:"
            ],
<skip>
```

The network facts module starting with Ansible 2.5 was a big step forward in streamlining your workflow and brought it on par with other server modules.

Network module conditional

Let's take a look at another network device conditional example by using the comparison keyword we saw at the beginning of this chapter. We can take advantage of the fact that both IOSv and Arista EOS provide the outputs in JSON format for the `show` commands. For example, we can check the status of the interface:

```
veos01#sh int eth 1 | json
{
    "interfaces": {
        "Ethernet1": {
            "lastStatusChangeTimestamp": 1569573423.6540787,
            "name": "Ethernet1",
            "interfaceStatus": "disabled",
            "autoNegotiate": "off",
            "loopbackMode": "loopbackNone",
            "interfaceStatistics": {
<skip>
```

Let us pretend we have an operation that we only wish to perform when `Ethernet1` is disabled. We can use the following tasks in the `chapter5_3.yml` playbook to check whether the condition is met before we proceed. It uses the `eos_command` module to gather the interface state output, and checks the interface status using the `wait_for` and `eq` keywords before proceeding to the next task:

```
<skip>
  tasks:
    - name: "sh int ethernet 1 | json"
      eos_command:
        commands:
            - "show interface ethernet 1 | json"
        wait_for:
          - "result[0].interfaces.Ethernet1.interfaceStatus eq
disabled"

      register: output
    - name: show output
      debug:
        msg: "Interface Disabled, Safe to Proceed"
```

Upon the condition being met, the second task will be executed:

```
$ ansible-playbook -i hosts chapter5_3.yml
<skip>
TASK [sh int ethernet 1 | json] ************************************
**********************************
ok: [arista1]
TASK [show output] *************************************************
*********************
```

```
ok: [arista1] => {
    "msg": "Interface Disabled, Safe to Proceed"
}
```

If the interface is active, an error will be given as illustrated by the output below. After the execution of the first task, the output lets us know that the change did not take place due to the condition not being met:

```
TASK [sh int ethernet 1 | json] ************************************
*********************************


fatal: [arista1]: FAILED! => {<skip>"changed": false, "failed_
conditions": ["result[0].interfaces.Ethernet1.interfaceStatus eq
disabled"], "msg": "One or more conditional statements have not been
satisfied"}


PLAY RECAP *********************************************************
**************


arista1                    : ok=0    changed=0    unreachable=0
failed=1    skipped=0    rescued=0    ignored=0
```

Check out the other conditional statements such as `contains`, `greater than`, and `less than`, as they fit into your situation. The conditional statements allow the playbook to be smart and executed based on the state of the device. In the next section, we will take a look at Ansible loops and how they can help us automatically execute plays to multiple items with only a few lines.

Ansible loops

Ansible provides a number of loops in the playbook, such as 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 `with_items` list. In our next example, `chapter5_4.yml`, we will loop over the items in the `with_items` list with the echo command from our localhost.

```
$ cat chapter5_4.yml
---
- name: Echo Loop Items
  hosts: "localhost"
  gather_facts: false

  tasks:
    - name: echo loop items
      command: echo "{{ item }}"
      with_items:
        - 'r1'
        - 'r2'
        - 'r3'
        - 'r4'
        - 'r5'
```

We will copy and paste our own public key under `~/.ssh/id_rsa.pub` to `~/.ssh/authorized_keys` and execute the playbook:

```
$ ansible-playbook -i hosts chapter5_4.yml
<skip>
TASK [echo loop items] *********************************************************
*************************


changed: [localhost] => (item=r1)
changed: [localhost] => (item=r2)
changed: [localhost] => (item=r3)
changed: [localhost] => (item=r4)
changed: [localhost] => (item=r5)
```

We will combine the standard loop with the network command module in the `chapter5_5.yml` playbook to add multiple VLANs to the device:

```
---
- name: Add Multiple Vlans
  hosts: "nxos-r1"
  gather_facts: false
  connection: network_cli

  vars:
    vlan_numbers: [100, 200, 300]
```

```
  tasks:
    - name: add vlans
      nxos_config:
        lines:
            - vlan {{ item }}
      with_items: "{{ vlan_numbers }}"
      register: output
```

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

```
     vars:
       vlan_numbers: [100, 200, 300]
```

We can execute the playbook and verify on `nxos-r1` that the VLANs were properly added:

```
$ ansible-playbook -i hosts chapter5_5.yml
<skip>
TASK [add vlans] ***********************************************************
*******************
changed: [nxos-r1] => (item=100)
changed: [nxos-r1] => (item=200)
changed: [nxos-r1] => (item=300)
PLAY RECAP *****************************************************************
**************
nxos-r1                    : ok=1    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0


nx-osv-1# sh vlan

VLAN Name                              Status    Ports
---- --------------------------------- --------- -------------------------
------
1    default                           active
100  VLAN0100                          active
200  VLAN0200                          active
300  VLAN0300                          active
```

The standard loop is a great time saver when it comes to performing redundant tasks in a playbook. It also makes the playbook more readable by reducing the lines required for the task.

Starting in Ansible 2.5, the `loop` keyword was added to replace most of the `with_<lookup>` loops, `https://docs.ansible.com/ansible/2.8/user_guide/playbooks_loops.html`. The `with_<lookup>` keyword is still valid for the foreseeable future due to its popularity. However, we should be aware of the new keyword and change.

In the next section, we will take a look at looping over dictionaries.

Looping over dictionaries

Looping over a simple list is nice. However, 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 attached to it, such as description, `vlan` gateway IP address, and possibly others. Oftentimes, we can use a dictionary to represent the entity to incorporate multiple attributes to it.

Let's expand on the `vlan` example in `chapter5_5.yml` for a dictionary example in `chapter5_6.yml`. We defined the dictionary values for three `vlans`, each with a nested dictionary for the description and the IP address:

```
---
- name: Add Multiple Vlans
  hosts: "nxos-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 proceeded 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 Nexus device:

```
nx-osv-1# sh run | i interface-vlan
feature interface-vlan
```

Upon execution, you will see the dictionary being looped through:

```
$ ansible-playbook -i hosts chapter5_6.yml
<skip>
TASK [add vlans] ***************************************************
**************************************************
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.30.1',
u'description': u'floor_3'}, 'key': u'300'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.20.1',
u'description': u'floor_2'}, 'key': u'200'})
changed: [nxos-r1] => (item={'value': {u'ip': u'192.168.10.1',
u'description': u'floor_1'}, 'key': u'100'})

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

```
u'description': u'floor_1'}, 'key': u'100'})
<skip>
```

Let's check if the intended configuration is applied to the device:

```
nx-osv-1# sh run int vlan 100

!Command: show running-config interface Vlan100
!Time: Fri Sep 27 18:00:24 2019

version 7.3(0)D1(1)

interface Vlan100
  description floor_1
  ip address 192.168.10.1/24
```

For more loop types of Ansible, feel free to check out the corresponding documentation (`http://docs.ansible.com/ansible/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 your tool belt. Ansible loop is a tool that can save us time and make the playbook more readable. In the next section, we will take a look at an Ansible template that allows 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 template module (`http://docs.ansible.com/ansible/template_module.html`).

The base template file we are using utilizes the Jinja2 template language (`http://jinja.pocoo.org/docs/`). We briefly discussed the Jinja2 templating language in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, and we will look at it a bit more here. Just like Ansible, Jinja2 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 is an important tool that we will be using in our daily task, and we will spend more of this section exploring it. We will learn the syntax by gradually building up our playbook from simple to more complex.

The basic syntax for template usage is very simple; you just need to specify the source file and the destination location that you 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:

```
$ mkdir Templates
$ cd Templates/
$ touch file1
```

Then, we will use the following playbook, `chapter5_7.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_third_edition/
Chapter05/Templates/file1
        dest=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/Templates/file2
```

Executing the playbook will create a new file:

```
$ ansible-playbook -i hosts chapter5_7.yml
TASK [copy one file to another] ****************************************
**********************************


changed: [localhost]


$ ls file*
file1
file2
```

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 Jinja2 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 Jinja2 template variables

The `chapter5_8.yml` playbook expands on the previous template example with the following additions:

1. The source file is `nxos.j2`.
2. The destination filename is now a variable in itself taken from the `nexus_devices` variable defined in the playbook.
3. Each of the devices within `nexus_devices` contain the variables that would be substituted or loop 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_third_edition/
Chapter05/Templates/nxos.j2
        dest=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/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.

Jinja2 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 both a list as well as a dictionary in Jinja2. 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.

The Jinja2 conditional

Jinja2 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:

```
$ ansible-playbook -i hosts chapter5_8.yml

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

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

ok: [localhost]

TASK [create router configuration files] ***************************
************************************************

changed: [localhost] => (item={'value': {u'username': u'cisco',
u'hostname': u'nx-osv-2', u'l3_vlan_interfaces': False, u'vlans': [100,
200, 300], u'password': u'cisco', u'netflow_enable': False}, 'key': u'nx-
osv-2'})

changed: [localhost] => (item={'value': {u'username': u'cisco', u'vlan_
interfaces': [{u'int_num': u'100', u'ip': u'192.168.10.1'}, {u'int_
num': u'200', u'ip': u'192.168.20.1'}, {u'int_num': u'300', u'ip':
u'192.168.30.1'}], u'hostname': u'nx-osv-1', u'l3_vlan_interfaces': True,
u'vlans': [100, 200, 300], u'password': u'cisco', u'netflow_enable':
True}, 'key': u'nx-osv-1'})


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 in 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


$ 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 required repeated copy and paste before. 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.

Our playbook is getting kind of long. In the next section, we will see how we can optimize the playbook by offloading the variable files into groups and directories.

Group and host variables

Note that, in the previous playbook, `chapter5_8.yml`, we have repeated ourselves in the username and password variables for the two devices under the `nexus_devices` variable:

```
    vars:
      nexus_devices: {
          "nx-osv-1": {
              "hostname": "nx-osv-1",
              "username": "cisco",
              "password": "cisco",
<skip>
          "nx-osv-2": {
              "hostname": "nx-osv-2",
              "username": "cisco",
              "password": "cisco",
<skip>
```

This is not ideal. If we ever need to update the username and password values, we will need to remember to update at both locations. This increases the management burden as well as the chances of making mistakes. As best practice, Ansible suggests that we use the `group_vars` and `host_vars` directories to separate out the variables in our playbook.

For more Ansible best practices, check out `http://docs.ansible.com/ansible/playbooks_best_practices.html`.

To look at the usage for grouping variables, we will create a new directory called `group_host_vars` in our working directory to house the code in the next section. We will maintain what we have already written for `chapter5_8.yml` and use that as a base for the next example. Let's start by copying over the `chapter5_8.yml` playbook to this new directory and renaming it to `chapter5_9.yml` and see how we can add variables to the playbook.

Group variables

By default, Ansible will look for group variables in the same directory as the playbook called `group_vars` for variables that can be applied to the group. By default, it will look for the filename that matches the group name in the inventory file. For example, if we have a group called [`nexus-devices`] in the inventory file, we can have a file under `group_vars` named `nexus-devices` to house all the variables that can be applied to the group.

We can also use a special file named `all` to include variables applied to all the groups.

We will utilize this feature for our username and password variables. First, we will create the `group_vars` directory:

```
$ mkdir group_vars
```

Then, we can create a YAML file called `all` to include the username and password:

```
$ cat group_vars/all
---
username: cisco
password: cisco
```

In the `chapter5_9.yml` playbook, we can now use the group variables for the playbook:

```
    "nexus_devices":
      "nx-osv-1":
        "hostname": "nx-osv-1"
        "username": "{{ username }}"
        "password": "{{ password }}"
    <skip>
      "nx-osv-2":
        "hostname": "nx-osv-2"
        "username": "{{ username }}"
        "password": "{{ password }}"
    <skip>
```

Using group variables works when we have the same value for multiple devices. In our example, `nx-osv-1` and `nx-osv-2` have the same username and password. In the next example, we will take a look at how we can use host-specific variables if they are different.

Host variables

We can further separate out the host variables in the same format as the group variables. This was how we were able to apply the variables in the Ansible 2.8 playbook examples in *Chapter 4*, *The Python Automation Framework – Ansible Basics*, and earlier in this chapter:

```
$ mkdir host_vars
```

In our case, we execute the commands on the localhost, and so the file under `host_vars` should be named accordingly, `host_vars/localhost`. In our `host_vars/localhost` file, we can also keep the variables declared in `group_vars`:

```
$ cat host_vars/localhost
---
"nexus_devices":
  "nx-osv-1":
    "hostname": "nx-osv-1"
    "username": "{{ username }}"
    "password": "{{ password }}"
    "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": "{{ username }}"
    "password": "{{ password }}"
    "vlans": [100, 200, 300]
    "l3_vlan_interfaces": False
    "netflow_enable": False
```

After we separate out the variables, the playbook now becomes very lightweight and only consists of the logic of our operation:

```
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
```

```
        template:
          src=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/Group_Host_Vars/nxos.j2
          dest=/home/echou/Mastering_Python_Networking_third_edition/
Chapter05/Group_Host_Vars/{{ item.key }}.conf
        with_dict: "{{ nexus_devices }}"
```

The `group_vars` and `host_vars` directories not only decrease our operations overhead, they can also help with consolidating sensitive information into a few files and securing the files with Ansible Vault, which we will look at next.

The Ansible Vault

As you can see from the previous section, in most cases, the Ansible variable provides sensitive information, such as a username and password. It would be a good idea to put some security measures around the variables so that we can safeguard against them. The Ansible Vault (`https://docs.ansible.com/ansible/2.8/user_guide/vault.html`) provides encryption for files so they do not appear in plaintext.

All Ansible Vault functions start with the `ansible-vault` command. You can manually create an encrypted file via the `create` option. You will be asked to enter a password. If you try to view the file, you will find that the file is not in clear text. If you have downloaded the book example, the password I used was just the word `password`:

```
$ ansible-vault create secret.yml
Vault password: <password>
$ cat secret.yml
$ANSIBLE_VAULT;1.1;AES256 33656462646237396232663532636132363932363535363
0646665656430353261383737623
<skip>65353733383738386363653035646462303233343238613930333566326 2 3962
```

To edit or view an encrypted file, we will use the `edit` option to edit or view the file via the `view` option:

```
$ ansible-vault edit secret.yml
Vault password: <password>
```

Let's now encrypt the `group_vars/all` and `host_vars/localhost` variable files:

```
$ ansible-vault encrypt group_vars/all host_vars/localhost
New Vault password:
Confirm New Vault password:
```

Now, when we run the playbook, we will get a decryption failed error message:

```
$ ansible-playbook chapter5_10.yml
PLAY [Ansible Group and Host Varibles] ***********************************
*****************************************
ERROR! Attempting to decrypt but no vault secrets found
```

We will need to use the `--ask-vault-pass` option when we run the playbook:

```
$ ansible-playbook chapter5_10.yml --ask-vault-pass
Vault password:
<skip>
```

The decryption will happen in memory for any vault-encrypted files that are accessed.

Prior to Ansible 2.4, Ansible Vault required all the files to be encrypted with the same password. Since Ansible 2.4 and later, you can use vault ID to supply a different password file, `https://docs.ansible.com/ansible/2.8/user_guide/vault.html`.

We can also save the password in a file and make sure that the specific file has restricted permission:

```
$ chmod 400 ~/.vault_password.txt
$ ls -lia ~/.vault_password.txt
809496 -r-------- 1 echou echou 9 Feb 18 12:17
/home/echou/.vault_password.txt
```

We can then execute the playbook with the `--vault-password-file` option:

```
$ ansible-playbook chapter5_10.yml --vault-password-file
~/.vault_password.txt
```

We can also encrypt just a string and embed the encrypted string inside the playbook by using the `encrypt_string` option (`https://docs.ansible.com/ansible/2.8/user_guide/vault.html#use-encrypt-string-to-create-encrypted-variables-to-embed-in-yaml`):

```
$ ansible-vault encrypt_string New
New Vault password:
Confirm New Vault password:
!vault |
          $ANSIBLE_VAULT;1.1;AES256
```

```
          3631313961616434386163393733633466343533643431313237346333633435303836623065383
9

          3365636263643138643738366536373465376130663134610a3637376539656134323333335626432

          6465343738373032373764626636356165343136633464626537616130313734316434353863376
5

          3663303232626338310a336238323364383338353835613162653763663438636137653865646261

          3234

Encryption successful
```

The string can then be placed in the playbook file as a variable. In the next section, we will optimize our playbook even further with `include` and `roles`.

The Ansible include and roles

The best way to handle complex tasks is to break them down into smaller pieces. Of course, this approach is common in both Python and network engineering. In Python, we break complicated code into functions, classes, modules, and packages. In networking, we also break large networks into sections such as racks, rows, clusters, and data centers. In Ansible, we can use `roles` and `includes` to segment and organize a large playbook into multiple files. Breaking up a large Ansible playbook simplifies the structure as each of the files focuses on fewer tasks. It also allows sections of the playbook to be reused.

The Ansible include statement

As the playbook grows in size, it will eventually become obvious that many of the tasks and plays can be shared across different playbooks. The Ansible `include` statement is similar to many Linux configuration files in that it just tells the machine to extend the file the same way as if the file was directly written in. We can use an `include` statement for both playbooks and tasks. Here, we will look at a simple example of extending our task.

Let's assume that we want to show outputs for two different playbooks. We can make a separate YAML file called `show_output.yml` as an additional task:

```
---
- name: show output
  debug:
    var: output
```

Then, we can reuse this task in multiple playbooks, such as in `chapter5_11_1.yml`, which looks largely identical to the last playbook, with the exception of registering the output and the `include` statement at the end:

```
---
- name: Ansible Group and Host Varibles
  hosts: localhost

  tasks:
    - name: create router configuration files
      template:
        src=./nxos.j2
        dest=./{{ item.key }}.conf
      with_dict: "{{ nexus_devices }}"
      register: output

    - include: show_output.yml
```

As an example of playbook reuse with include, another playbook, `chapter5_11_2.yml`, can reuse `show_output.yml` in the same way:

```
---
- name: show users
  hosts: localhost

  tasks:
    - name: show local users
      command: who
      register: output

    - include: show_output.yml
```

Note that both playbooks use the same variable name, output, because in `show_output.yml`, we hardcoded the variable name for simplicity. You can also pass variables into the included file if that is not desired.

Ansible roles

Ansible roles separate the logical function with a physical host to fit your network better. For example, you can construct roles such as spines, leafs, and core, as well as Cisco, Juniper, and Arista. The same physical host can belong to multiple roles; for example, a device can belong to both Juniper and the core. This flexibility allows us to perform operations, such as upgrading all Juniper devices, without worrying about the device's location in the layer of the network.

Ansible roles can automatically load certain variables, tasks, and handlers based on a known file infrastructure. The key is that this is a known file structure that we automatically include. In fact, you can think of roles as pre-made `include` statements by Ansible.

The Ansible playbook role documentation (`http://docs.ansible.com/ansible/playbooks_roles.html#roles`) describes a list of role directories that we can configure, such as tasks, handlers, files, templates, vars, defaults, and meta. We do not need to use all of them. In our example, we will only modify the `tasks` and the `vars` folders. However, it is good to know all of the available options in the Ansible role directory structure.

The following is what we will use as an example for our roles:

```
$ tree .
.
├── chapter5_12.yml
├── chapter5_13.yml
├── hosts
└── roles
    ├── cisco_nexus
    │   ├── tasks
    │   │   └── main.yml
    │   └── vars
    │       └── main.yml
    └── spines
        ├── tasks
        │   └── main.yml
        └── vars
            └── main.yml

7 directories, 7 files
```

We can see that the hosts file and the playbooks are located at the top level. We also have a folder named `roles`. Inside the roles folder, there are two roles folders named: `cisco_nexus` and `spines`. As you can see, we are only using the `tasks` and `vars` folders, not all the available folders for roles. There is a file named `main.yml` inside each of them. This is the default behavior: the `main.yml` file is our entry point that is automatically included in the playbook when we specify the role in the playbook. If we need to break out additional files, you can use the `include` statement in the `main.yml` file.

Here is our scenario:

- We have two Cisco Nexus devices, `nxos-r1` and `nxos-r2`. We will configure the logging server as well as the log link status for all of them, utilizing the `cisco_nexus` role for them.
- In addition, `nxos-r1` is also a spine device, where we will want to configure more verbose logging, perhaps because spines are at a more critical position within our network.

For our `cisco_nexus` role, we have the following variables in:

`roles/cisco_nexus/vars/main.yml`:

```
---
cli:
  host: "{{ ansible_host }}"
  username: cisco
  password: cisco
  transport: cli
```

We have the following configuration tasks in:

`roles/cisco_nexus/tasks/main.yml`:

```
---
- name: configure logging parameters
  nxos_config:
    lines:
      - logging server 191.168.1.100
      - logging event link-status default
    provider: "{{ cli }}"
```

Our `chapter5_12.yml` playbook is extremely simple, as it just needs to specify the hosts that we would like to configure according to `cisco_nexus role`:

```
---
- name: playbook for cisco_nexus role
  hosts: "cisco_nexus"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
```

When you run the playbook, the playbook will include the tasks and variables defined in the `cisco_nexus` role and configure the devices accordingly:

```
$ ansible-playbook -i hosts chapter5_12.yml
<skip>
TASK [cisco_nexus : configure logging parameters] ***********************
**************************************************
changed: [nxos-r1]
changed: [nxos-r2]
```

For our `spine` role, we will have an additional task of more verbose logging in:

`roles/spines/tasks/main.yml`:

```
---
- name: change logging level
  nxos_config:
    lines:
      - logging level local7 7
    provider: "{{ cli }}"
```

In our playbook, `chapter5_13.yml`, we can specify the `cisco_nexus` and `spines` roles:

```
---
- name: playbook for spine role
  hosts: "spines"
  gather_facts: false
  connection: local

  roles:
    - cisco_nexus
    - spines
```

When we include both roles in this order, the `cisco_nexus` role will be executed, followed by the `spines` role:

```
$ ansible-playbook -i hosts chapter5_13.yml
<skip>
TASK [cisco_nexus : configure logging parameters] ***********************
**************************************************
changed: [nxos-r1]


TASK [spines : change logging level] ************************************
****************************************
changed: [nxos-r1]
<skip>
```

Ansible roles are flexible and scalable – just like Python functions and classes. Once your code grows beyond a certain level, it is almost always a good idea to break it into smaller pieces for maintainability.

You can find more examples of roles in the Ansible examples Git repository at `https://github.com/ansible/ansible-examples`.

Ansible Galaxy (`https://docs.ansible.com/ansible/latest/reference_appendices/galaxy.html`) is a free community site for finding, sharing, and collaborating on roles. You can see an example of the Juniper networks supplied by the Ansible role on Ansible Galaxy:

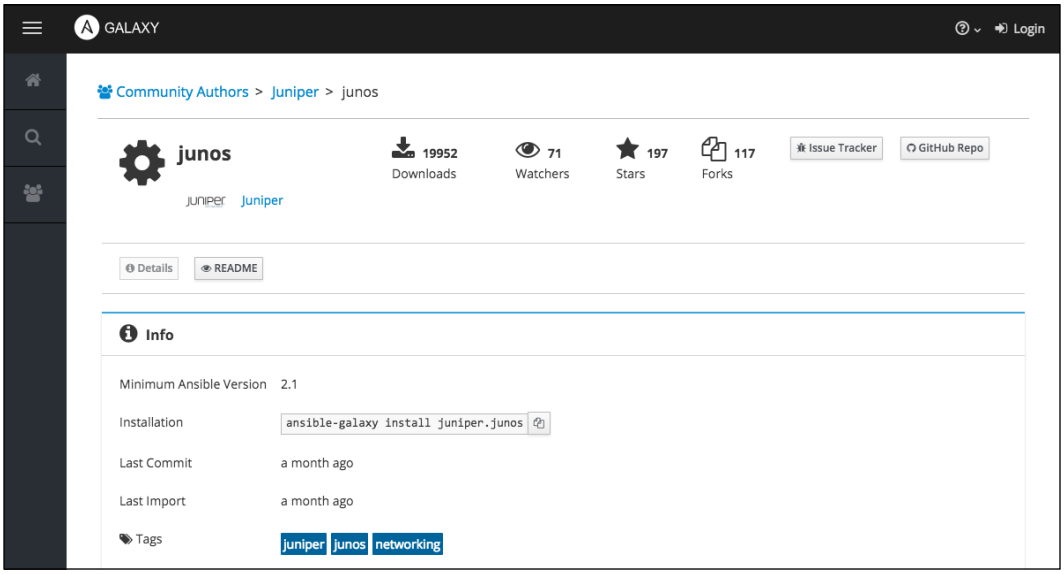


Figure 1: JUNOS role on Ansible Galaxy (https://galaxy.ansible.com/Juniper/junos)

In the next section, we will take a look at how to write our own custom Ansible module.

Writing your own custom module

By now, you may get the feeling that network management in Ansible is largely dependent on finding the right module for your task. There is certainly a lot of truth in that logic.

Modules provide a way to abstract the interaction between the managed host and the control machine; they allow us to focus on the logic of our operations. Up to this point, we have seen the major vendors providing a wide range of modules for Cisco, Juniper, and Arista.

Using the Cisco Nexus modules as an example, besides specific tasks such as managing the BGP neighbor (`nxos_bgp`) and the aaa server (`nxos_aaa_server`), most vendors also provide ways to run arbitrary show (`nxos_command`) and configuration commands (`nxos_config`). This generally covers most of our use cases.

Starting with Ansible 2.5, there is also the streamline naming and usage of network facts modules.

What if the device you are using does not currently have the module for the task that you are looking for? In this section, we will look at how to remedy this situation by writing our own custom module.

The first custom module

Writing a custom module does not need to be complicated; in fact, it doesn't even need to be in Python. But since we are already familiar with Python, we will use Python for our custom modules. We are assuming that the module is what we will be using ourselves and our team without submitting back to Ansible. Therefore, we will ignore some of the documentation and formatting for the time being.

If you are interested in developing modules that can be submitted upstream to Ansible, please consult the developing modules guide from Ansible (`https://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html`).

By default, if we create a folder named `library` in the same directory as the playbook, Ansible will include the directory in the module search path. Therefore, we can put our custom module in the directory, and we will be able to use it in our playbook. The requirement for the custom module is very simple: all the module needs is to return a JSON output to the playbook.

Recall that in *Chapter 3*, *APIs and Intent-Driven Networking*, we used the following NXAPI Python script, `cisco_nxapi_2.py`, to communicate with the NX-OS device:

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

import requests
```

```
import json

url='http://172.16.1.90/ins'
switchuser='cisco'
switchpassword='cisco'

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=myheade
rs,auth=(switchuser,switchpassword)).json()

print(response['result']['body']['sys_ver_str'])
```

When we executed it, we simply received the system version. We can modify the last line to be a JSON output, as shown in the following code:

```
version = response['result']['body']['sys_ver_str']
print json.dumps({"version": version})
```

If you did not enable the `nxapi` feature on the remote device in *Chapter 3, APIs and Intent-Driven Networking*, you need to configure it in order for the custom module script to work:

nx-osv-1(config)# feature nxapi

nx-osv-1(config)# nxapi http port 80

nx-osv-1(config)# nxapi sandbox

We will place this file under the `library` folder:

```
$ ls -a library/
.... custom_module_1.py
```

In our playbook, we can then use the action plugin (`https://docs.ansible.com/ansible/dev_guide/developing_plugins.html`), `chapter5_14.yml`, to call this custom module:

```
---
- name: Your First Custom Module
  hosts: localhost
  gather_facts: false
  connection: local

  tasks:
    - name: Show Version
      action: custom_module_1
      register: output

    - debug:
        var: output
```

Note that, just like the `ssh` connection, we are executing the module locally, with the module making API calls outbound. When you execute this playbook, you will get the following output:

```
$ ansible-playbook chapter5_14.yml
PLAY [Your First Custom Module] ************************************************
****************************


TASK [Show Version] ************************************************************
****************
ok: [localhost]


TASK [debug] *******************************************************************
************************************************
ok: [localhost] => {
    "output": {
        "changed": false,
        "failed": false,
        "version": "7.3(0)D1(1)"
    }
}
<skip>
```

As you can see, you can write any module that is supported by any device API, and Ansible will happily take any returned JSON output.

The second custom module

Building upon the last module, let's utilize the common module boilerplate code from Ansible that's stated in the module development documentation (`http://docs.ansible.com/ansible/dev_guide/developing_modules_general.html`). We will modify the last custom module and create `custom_module_2.py` to ingest inputs from the playbook.

First, we will import the boilerplate code from `ansible.module_utils.basic`:

```
from ansible.module_utils.basic import AnsibleModule
if __name__ == '__main__':
    main()
```

From there, we can define the main function where we will house our code. `AnsibleModule`, which we have already imported, provides lots of common code for handling returns and parsing arguments. In the following example, we will parse three arguments for host, username, and password, and make them required fields:

```
def main():
    module = AnsibleModule(
      argument_spec = dict(
        host = dict(required=True),
        username = dict(required=True),
        password = dict(required=True)
      )
    )
```

The values can then be retrieved and used in our code:

```
    device = module.params.get('host')
    username = module.params.get('username')
    password = module.params.get('password')
    url='http://' + host + '/ins'
    switchuser=username
    switchpassword=password
```

Finally, we will follow the exit code and return the value:

```
    module.exit_json(changed=False, msg=str(data))
```

Our new playbook, `chapter5_15.yml`, will look identical to the last playbook, except now we can pass values for different devices to the playbook:

```
    tasks:
      - name: Show Version
        action: custom_module_2 host="172.16.1.142" username="cisco"
password="cisco"
        register: output
```

When executed, this playbook will produce the exact same output as the last playbook. However, because we are using arguments in the custom module, the custom module can now be passed around for other people to use without them knowing the details of our module. They can write in their own username, password, and host IP in the playbook.

Of course, this is a functional but incomplete module. For one, we did not perform any error checking, nor did we provide any documentation for usage. However, it is a good demonstration of how easy it is to build a custom module. The additional benefit is that we saw how we can use an existing script that we already made and turn it into a custom Ansible module.

Summary

In this chapter, we covered a lot of ground. Building from our previous knowledge of Ansible, we expanded into more advanced topics such as conditionals, loops, and templates. We looked at how to make our playbook more scalable with host variables, group variables, include statements, and roles. We also looked at how to secure our playbook with the Ansible Vault. Finally, we used Python to make our own custom modules.

Ansible is a very flexible Python framework that can be used for network automation. It provides another abstraction layer separated from the likes of the Pexpect and API-based scripts. It is declarative in nature in that it is more expressive in terms of matching our intent. Depending on your needs and network environment, it might be the ideal framework that you can use to save time and energy.

In *Chapter 6*, *Network Security with Python*, we will look at network security with Python.

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 with a holistic approach by 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

The lab setup

The devices being used in this chapter are a bit different from the previous chapters. In the previous chapters, we were isolating a particular device to focus on the topic at hand. For this chapter, we will use a few more devices in our lab to illustrate the function of the tools we will be using. The connectivity and operating system information are important as they have ramifications regarding the security tools that 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 which direction the client is making their connections from. 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 be using the same Cisco VIRL tool with four nodes: two hosts and two network devices. If you need a refresher on Cisco VIRL, feel free to go back to *Chapter 2*, *Low-Level Network Device Interactions*, where we first introduced the tool:

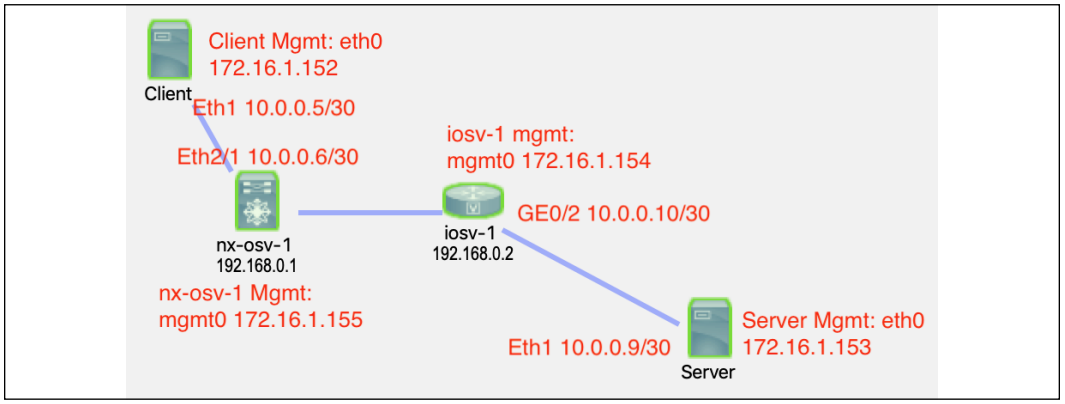


Figure 1: Lab topology

The IP addresses listed might be different in your own lab. They are listed here for easy reference to the remainder of the chapter code examples.

As illustrated, we will rename the host on the top as the client, and the bottom host as the server. This is analogous to an internet client trying to access a corporate server within our network. We will again use the **Shared flat network** option for the management network to access the devices for the out-of-band management:

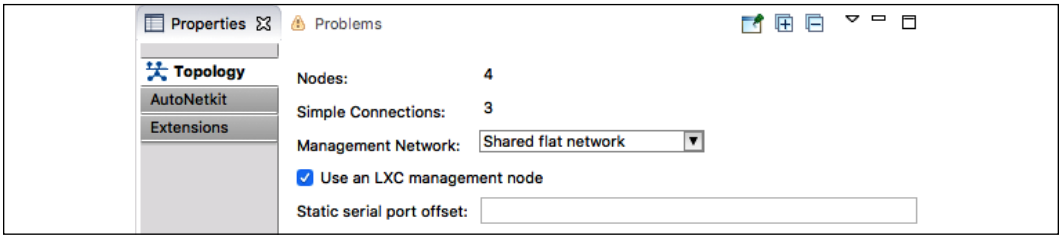


Figure 2: Management network option for lab

The client host will need to have external access via VMnet2 in this example. For my lab, I have an external USB upstream connected to my ESXi host that provides connectivity. You may also need to add external DNS servers under `/etc/resolveconf/resolv.conf.d/base`:

```
cisco@Client:~$ cat /etc/resolvconf/resolv.conf.d/base
nameserver 8.8.8.8
nameserver 8.8.4.4
```

For the two switches, **Open Shortest Path First** (**OSPF**) is run as the `IGP` and both of the devices are in area `0`. By default, `BGP` is turned on and both the devices are using AS 1.

From the configuration autogeneration, the interfaces connected to the Ubuntu hosts are put into OSPF area `1`, so they will show up as inter-area routes. The NX-OSv configurations are shown here and the IOSv configuration and output are similar:

```
interface Ethernet2/1
  description to Client
  no switchport
  mac-address fa16.3e00.0001
  ip address 10.0.0.6/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
!
interface Ethernet2/2
  description to iosv-1
  no switchport
  mac-address fa16.3e00.0002
  ip address 10.0.0.14/30
  ip router ospf 1 area 0.0.0.0
  no shutdown
```

```
!
nx-osv-1# sh ip route
<skip>
10.0.0.8/30, ubest/mbest: 1/0
    *via 10.0.0.13, Eth2/2, [110/41], 14:10:28, ospf-1, intra
192.168.0.2/32, ubest/mbest: 1/0
    *via 10.0.0.13, Eth2/2, [110/41], 14:10:28, ospf-1, intra
<skip>
```

The OSPF neighbor and the BGP output for NX-OSv are shown here, and the IOSv output is similar:

```
nx-osv-1# sh ip ospf neighbors
 OSPF Process ID 1 VRF default
 Total number of neighbors: 1
 Neighbor ID     Pri State            Up Time  Address         Interface
 192.168.0.2       1 FULL/DR          14:12:31 10.0.0.13       Eth2/2
!
nx-osv-1# sh ip bgp summary
BGP summary information for VRF default, address family IPv4 Unicast
BGP router identifier 192.168.0.1, local AS number 1
BGP table version is 5, IPv4 Unicast config peers 1, capable peers 1
2 network entries and 2 paths using 288 bytes of memory
BGP attribute entries [2/288], BGP AS path entries [0/0]
BGP community entries [0/0], BGP clusterlist entries [0/0]

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down
State/PfxRcd
192.168.0.2     4     1     936     857        5    0    0 14:12:33 1
```

The hosts in our network are running Ubuntu 16.04, similar to Ubuntu VM 18.04, which we have been using up to this point:

```
cisco@Server:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 16.04.3 LTS
Release: 16.04
Codename: xenial
```

On both of the Ubuntu hosts, there are two network interfaces, `eth0` and `eth1`. `eth0` connects to the management network (`172.16.1.0/24`), while `eth1` connects to the network devices (`10.0.0.x/30`). The routes to the device loopback are directly connected to the network block, and the remote host networks are statically routed to `eth1` with the default route going toward the management network:

```
cisco@Client:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask          Flags Metric Ref    Use
Iface
0.0.0.0         172.16.1.254    0.0.0.0          UG    0      0        0
eth0
10.0.0.0        10.0.0.6        255.0.0.0        UG    0      0        0
eth1
10.0.0.4        0.0.0.0         255.255.255.252  U     0      0        0
eth1
172.16.1.0      0.0.0.0         255.255.255.0    U     0      0        0
eth0
192.168.0.0     10.0.0.6        255.255.255.248  UG    0      0        0
eth1
```

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:

```
# Server Eth1 IP is 10.0.0.9/30
cisco@Server:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr fa:16:3e:68:bb:ce
          inet addr:10.0.0.9  Bcast:10.0.0.11  Mask:255.255.255.252
<skip>

# From Client Eth1 IP 10.0.0.5/30 Ping to Server
cisco@Client:~$ ifconfig eth1
eth1      Link encap:Ethernet  HWaddr fa:16:3e:7c:75:ec
          inet addr:10.0.0.5  Bcast:10.0.0.7  Mask:255.255.255.252
<skip>

cisco@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=7.00 ms
```

```
--- 10.0.0.9 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.007/7.007/7.007/0.000 ms

# traceroute from client to server
cisco@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)  5.694 ms  10.632 ms  10.599 ms
 2  10.0.0.13 (10.0.0.13)  13.078 ms  19.132 ms  19.101 ms
 3  10.0.0.9 (10.0.0.9)  14.929 ms  19.026 ms  19.004 ms
```

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 you to craft your own packet from the 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 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 Python3 support per Scapy release:

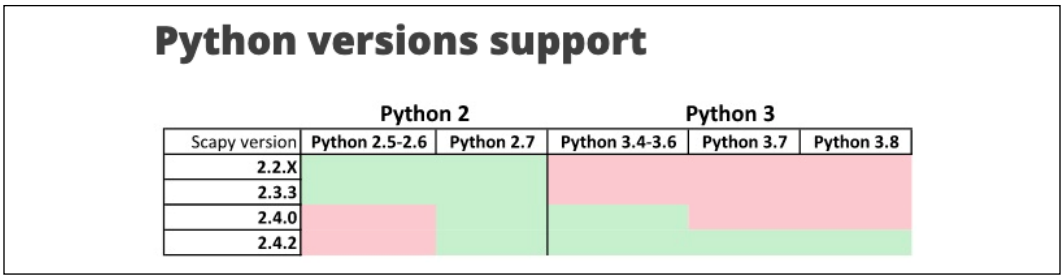


Figure: 3 Python version support (source: https://scapy.net/download/)

The long story for Python 3 support in Scapy is that there was an independent fork of Scapy from version 2.2.0 in 2015, aimed at supporting only Python 3. The project was named `Scapy3k`. The fork diverged from the main Scapy code base. If you read the first edition of this book, that was the information provided at the time of writing. There was confusion surrounding `python3-scapy` on PyPI and the official support of the Scapy code base. Our main purpose is to learn about Scapy in this chapter. Therefore, I have made the choice to use an older, Python 2-based, version of Scapy.

In our lab, since we are crafting packet sources from the client to the destination server, Scapy needs to be installed on the client:

```
cisco@Client:~$ git clone https://github.com/secdev/scapy.git
cisco@Client:~$ cd scapy/
cisco@Client:~/scapy$ sudo python3 setup.py install
```

Following installation, we can launch the Scapy interactive shell by typing in **scapy** in the command prompt:

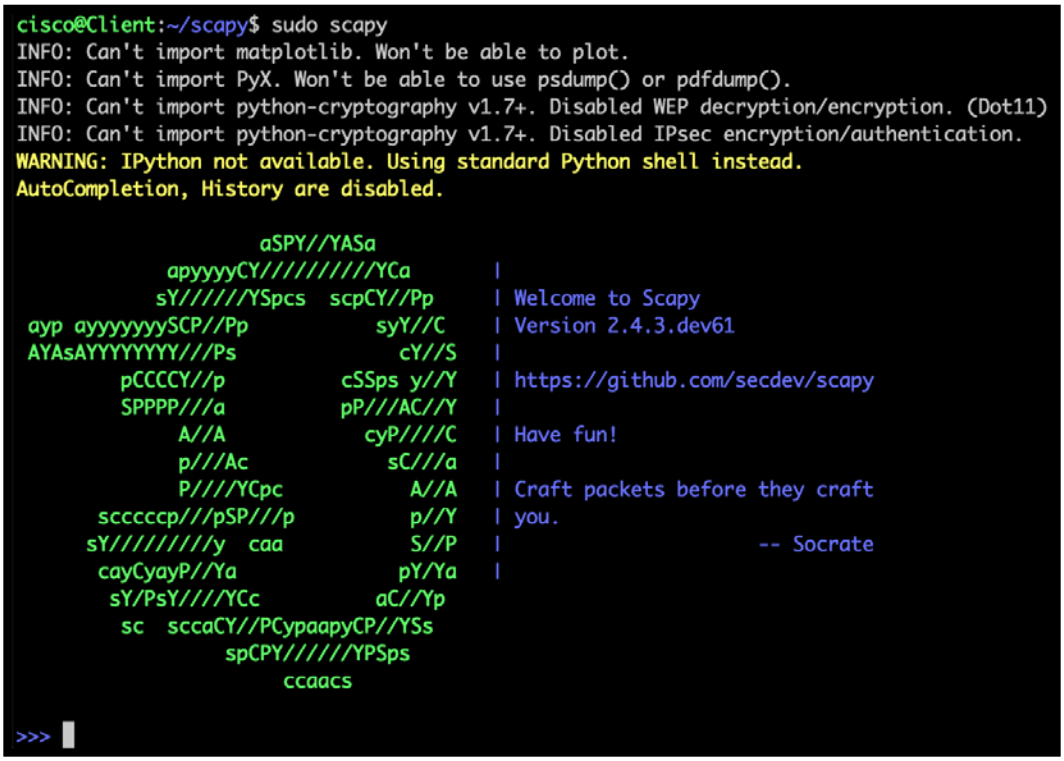


Figure 4: Python Scapy testing

Here is a quick test to make sure we can access the `Scapy` library from Python 3:

```
cisco@Client:~$ python3
Python 3.5.2 (default, Jul 10 2019, 11:58:48)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> exit()
```

Great! 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
cisco@Client:~/scapy$ sudo scapy
>>> send(IP(dst="10.0.0.9")/ICMP())
.
Sent 1 packets.


# Server side
cisco@Server:~$ sudo tcpdump -i eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
17:19:24.812184 IP 10.0.0.5 > 10.0.0.9: ICMP echo request, id 0, seq 0,
length 8
17:19:24.812205 IP 10.0.0.9 > 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=44710 flags= frag=0 ttl=62
proto=icmp chksum=0xba2d src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echo-
reply code=0 chksum=0xffff id=0x0 seq=0x0 |>>
```

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 can see that it is another tuple containing the packet that we have sent as well as the returned packet:

```
>>> for i in ans:
...     print(type(i))
...
<class 'tuple'>
>>>
>>>
>>> for i in ans:
...     print(i)
...
(<IP  frag=0 proto=icmp dst=10.0.0.9 |<ICMP  |>>, <IP  version=4 ihl=5
tos=0x0 len=28 id=19027 flags= frag=0 ttl=62 proto=icmp chksum=0x1e81
src=10.0.0.9 dst=10.0.0.5 |<ICMP  type=echo-reply code=0 chksum=0xffff
id=0x0 seq=0x0 |>>)
```

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 17 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x20 len=76 id=17713 flags= frag=0 ttl=121
proto=udp chksum=0x28c5 src=8.8.8.8 dst=192.168.2.211 |<UDP  sport=domain
dport=domain len=56 chksum=0xa9db |<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=274 rdlen=None
rdata=216.58.217.36 |> 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 192.168.2.211 > 8.8.8.8 echo-request 0 / Raw
0001 Ether / IP / ICMP 8.8.8.8 > 192.168.2.211 echo-reply 0 / Raw
0002 Ether / IP / ICMP 192.168.2.211 > 8.8.8.8 echo-request 0 / Raw
0003 Ether / IP / ICMP 8.8.8.8 > 192.168.2.211 echo-reply 0 / Raw
0004 Ether / IP / ICMP 192.168.2.211 > 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= 70:4f:57:94:7f:86
```

```
  src= 5e:00:00:02:00:00
  type= IPv4
###[ IP ]###
     version= 4
     ihl= 5
     tos= 0x0
     len= 84
     id= 1856
     flags= DF
     frag= 0
     ttl= 64
     proto= icmp
     chksum= 0x5fde
     src= 192.168.2.211
     dst= 8.8.8.8
     \options\
###[ ICMP ]###
        type= echo-request
        code= 0
        chksum= 0x4616
        id= 0x4a84
        seq= 0x1
###[ Raw ]###
           load= 'k\x9a\x8f]\x00\x00\x00\x00\xac\x99\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, so that they can then focus their efforts on the attack. Of course, we need to open certain ports in order to service our customer; that is part of the risk we need to accept. However, we should also 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"))
>>> p = sr1(IP(dst.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)
...
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ftp_data
flags=S |>>, <IP  version=4 ihl=5 tos=0x0 len=40 id=59720 flags=DF
frag=0 ttl=62 proto=tcp chksum=0x3f7a 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 |>>)
(<IP  frag=0 proto=tcp dst=10.0.0.9 |<TCP  sport=666 dport=ftp
flags=S |>>, <IP  version=4 ihl=5 tos=0x0 len=40 id=59721 flags=DF
frag=0 ttl=62 proto=tcp chksum=0x3f79 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 |>>)
```

```
(<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=3932520059 ack=1 dataofs=6 reserved=0 flags=SA window=29200
chksum=0xa666 urgptr=0 options=[('MSS', 1460)] |>>)
>>>
```

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,fla
gs="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):

```
cisco@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.

cisco@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 own packet with Scapy. We tested out 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 own 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 own packet, you can be in charge of the type of operations and tests that you would like to run. Along the same line of thought of constructing our own packets using Scapy, we can construct our own 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 of the 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 a similar commercial software. With Scapy, you can conduct the test while maintaining full control as well as adding more tests in the future.

The first attack basically 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 16.04 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 own network and host without having to wait 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 to be more and more common on the internet. Scapy is definitely 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 quite a bit of 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 with me that Scapy is a great tool to keep in your toolset as a network engineer. 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 test 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 on the inbound direction on the external facing network interface(s). In our lab scenario, this means we will place an inbound access list at Ethernet2/1 on nx-osv-1, 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 port `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 the last two chapters, but it is worth repeating the advantages of using Ansible in this scenario:

- **Easier management**: For a long access list, we are able to 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 a regular interval 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. For simplicity, we are putting the host variables directly in the inventory file:

```
[nxosv-devices]
nx-osv-1 ansible_host=172.16.1.155 ansible_username=cisco ansible_
password=cisco

[iosv-devices]
iosv-1 ansible_host=172.16.1.154 ansible_username=cisco ansible_
password=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 }}"
      transport: cli
```

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 choose 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
        provider: "{{ cli }}"
    - nxos_acl:
        name: border_inbound
```

```
            seq: 30
            action: deny
            proto: tcp
            src: 192.168.0.0/16
            dest: any
            state: present
            log: enable
            provider: "{{ cli }}"
    <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 2/1
          nxos_acl_interface:
            name: border_inbound
            interface: Ethernet2/1
            direction: ingress
            state: present
            provider: "{{ cli }}"
```

The access list on VIRL NX-OSv is only supported on the management interface. You will see this warning: **Warning: ACL may not behave as expected since only one management interface is supported if you configure this ACL via the CLI**. This warning is okay, as our purpose is just to demonstrate the configuration automation of the access list.

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 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: [nx-osv-1]

<skip>

TASK [apply ingress acl to Ethernet 2/1] ***************************************
*****************************************

changed: [nx-osv-1]

We should log in to the nx-osv-1 devices to verify the changes:

nx-osv-1# 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 2/1
!
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 take a look at how to restrict device access in a layer 2 environment.

MAC access lists

In the case where you have an 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 change 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
        provider: "{{ cli }}"
    - name: Apply filter on bridge group 1
      ios_config:
        lines:
          - bridge-group 1
          - bridge-group 1 input-address-list 700
        parents:
          - interface GigabitEthernet0/1
        provider: "{{ cli }}"
```

We can execute the playbook and verify the application of it on `iosv-1`:

```
$ ansible-playbook -i hosts access_list_mac_iosv.yml
TASK [Deny Hosts with vendor id fa16.3e00.0000] ************************
**************************************************
changed: [iosv-1]

TASK [Apply filter on bridge group 1] **********************************
**************************************************
changed: [iosv-1]
```

As we have done before, let's log in to the device to verify our change:

```
iosv-1#sh run int gig 0/1
!
interface GigabitEthernet0/1
 description to nx-osv-1
 <skip>
 bridge-group 1
 bridge-group 1 input-address-list 700
end
```

As more virtual networks become popular, the 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 gear a bit but stay within the security context and take a 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 of time. In these slow breaches, quite often, we 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 that 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 iptables, 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 accommodated 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. In fact, Python's interpretative nature might actually 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 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 a number of other powerful tools that can make this process easier and more effective. In the last 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

A **virtual local area networks** (**VLANs**) have been around for a long time. They are essentially a broadcast domain where all hosts can be connected to a single switch, but are partitioned out to different domains, so we can separate the hosts out according to which host 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 own 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 both 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 that was 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 router.
- **The Isolated (I) port**: This port is only allowed to communicate with P ports, and they are 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 as well as with P ports.

We can again use Ansible or any of the 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 would come in handy at times when you need to isolate ports even further in an layer 2 VLAN.

UFW with Python

We briefly mentioned UFW as the frontend for iptables 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 `http://docs.ansible.com/ansible/ufw_module.html`. Because Ansible is written in Python, we can go further and examine what is inside the Python module source code. More information is available at `https://github.com/ansible/ansible/blob/devel/lib/ansible/modules/system/ufw.py`.
- 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 the existing Python knowledge if you ever 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 pentesters, by Justin Seitz (ISBN-10: 1593275900)

I have personally used Python extensively in our 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 VIRL tool to set up our lab with both 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 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 as well as a MAC access list to protect our network. They are usually the first line of defense in our network protection. Using Ansible, we are able to 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.

7
Network Monitoring with Python – Part 1

Imagine you get a call from your company's network operations center at 2:00 a.m. in the morning. 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 this 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? Chances are you would check your monitoring tool and see if any of the key metrics changed in the last few hours. Better yet, you 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, with the goal of keeping 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 together, constantly passing traffic back and forth.

There are lots of moving parts that can cause your 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 rather a question 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 of the 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 information the equipment is capable of transmitting. Second, we need to identify what useful information we can interpret from it.

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 is similar to the one in *Chapter 6*, *Network Security with Python*, but with this difference: both of the network devices are IOSv devices. Here's an illustration of this:

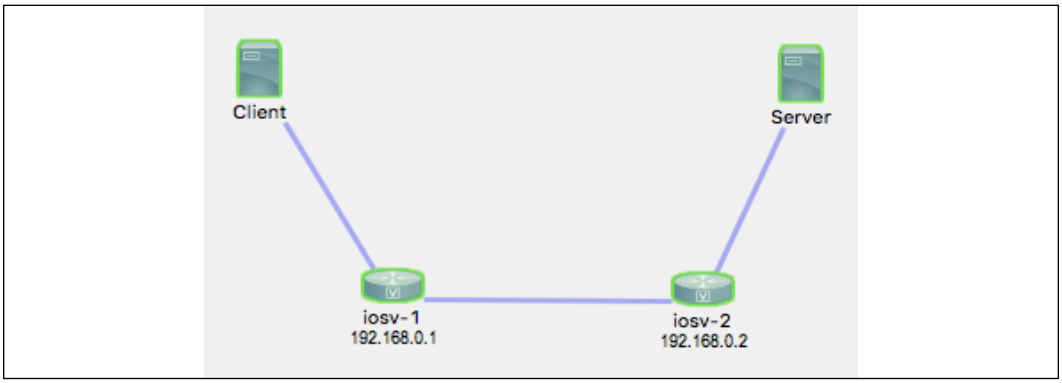


Figure 1: Lab topology

The two Ubuntu hosts will be used to generate traffic across the network so that we can look at some non-zero counters.

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 a bit uneasy. SNMP version 3 has added cryptographic security and new concepts and terminology to the protocol, but the way 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, which is 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:

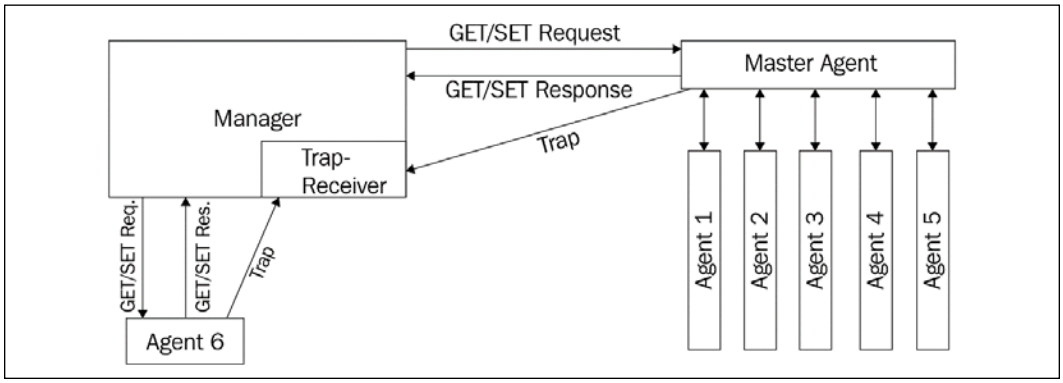


Figure 2: SNMP operations

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 really talking about using the management station to query the specific OID that represents the information we are after. There is a common OID structure, such as a systems and interfaces OID, that is shared among vendors. Besides common OID, each vendor can also supply an enterprise-level OID that is specific to them.

As an operator, you are required to put some effort into consolidating information into an OID structure in your environment to retrieve useful information. This can sometimes be a tedious process of finding one OID at a time. For example, you might be making a request to 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 is a 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 in order to query the data.

Since SNMP has been around for a while, my assumption is that you have some experience with it already. Let's jump directly into package installation and our first SNMP example.

Setup

First, let's make sure 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:

Please refer to *Chapter 6*, *Network Security with Python*, for external access in the VIRL host if needed.

```
$ sudo apt-get update
$ sudo apt-get install snmp
```

The next step is to turn on and configure the SNMP options on the network devices, `iosv-1` and `iosv-2`. 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 as well as 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 172.16.1.123 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 IOS MIB Locator (`http://tools.cisco.com/ITDIT/MIBS/servlet/index`) 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 172.16.1.189 .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 172.16.1.189 .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 172.16.1.189 .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 nor messages; it simply stated that the OID does not exist. This can be pretty frustrating at times.

The last thing to check would be 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 29 16:39:19.857: %SEC-6-IPACCESSLOGNP: list permit_snmp permitted 0
172.16.1.123 -> 0.0.0.0, 1 packet
```

As you can see, the biggest challenge in setting up SNMP is finding the right OID. Some of the OIDs are defined in standardized MIB-2; others are under the enterprise portion of the tree. Vendor documentation is the best bet, though. There are a number of tools that can help, such as an 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) $ git checkout 0.2.3
(venv) $ python 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) $ python 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. The steps we will take are slightly modified versions from the PySNMP example at: `http://pysnmp.sourceforge.net/faq/response-values-mib-resolution.html`. 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(('172.16.1.189', 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(('172.16.1.189', 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 out 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`:

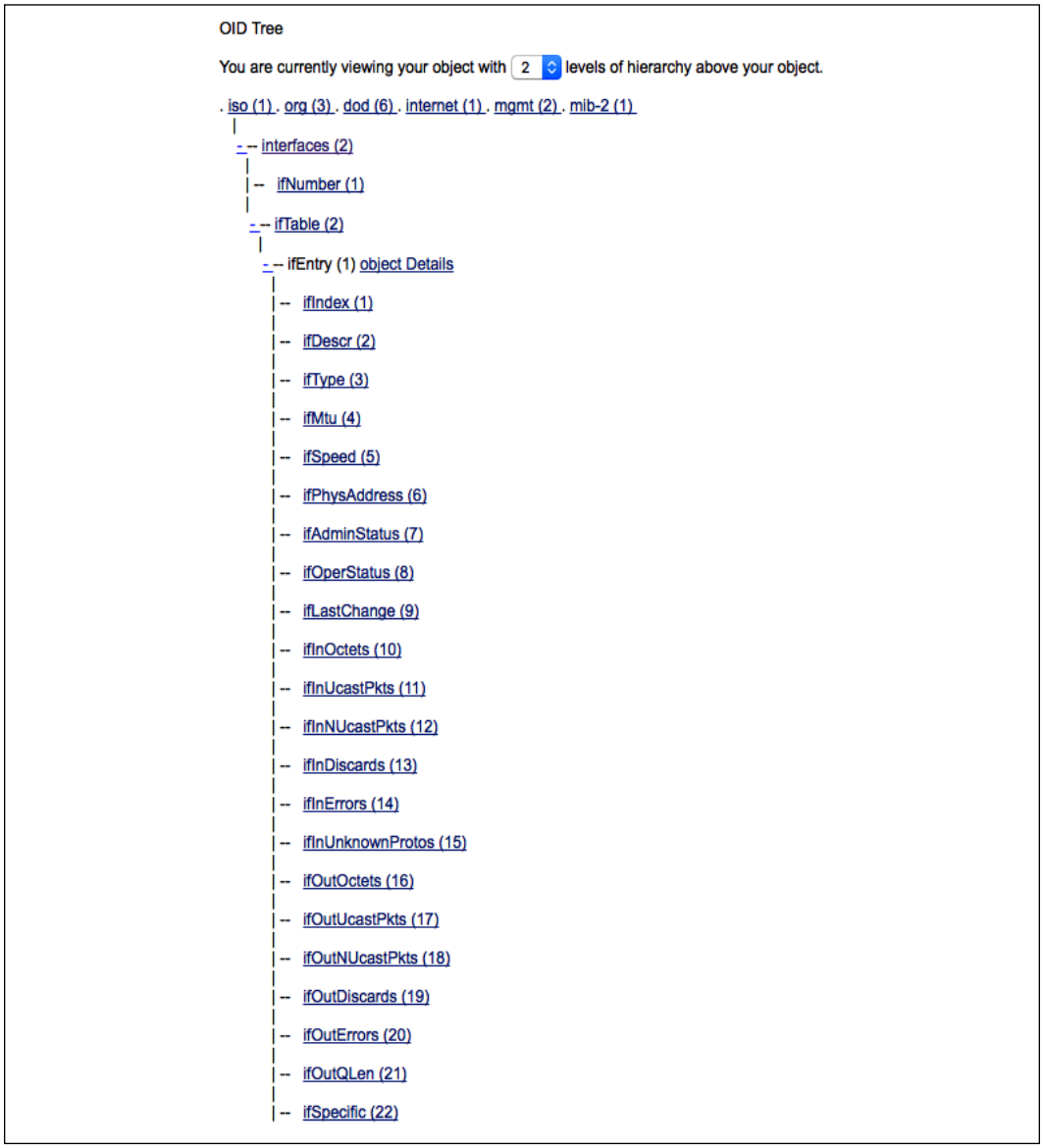


Figure 3: SNMP ifEntry OID tree

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 exactly the same since there might be some traffic on the wire between the time of the CLI output and the SNMP query time:

```
iosv-1#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 172.16.1.189 .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 the `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_third_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 for the purpose of gaining 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:

```
(venv) $ pip install matplotlib
```

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 you download it (as you will see soon). Note that you will need to set the `$DISPLAY` variable in some of the graphs that we will produce in this section.

The following is a screenshot of the Ubuntu desktop 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:

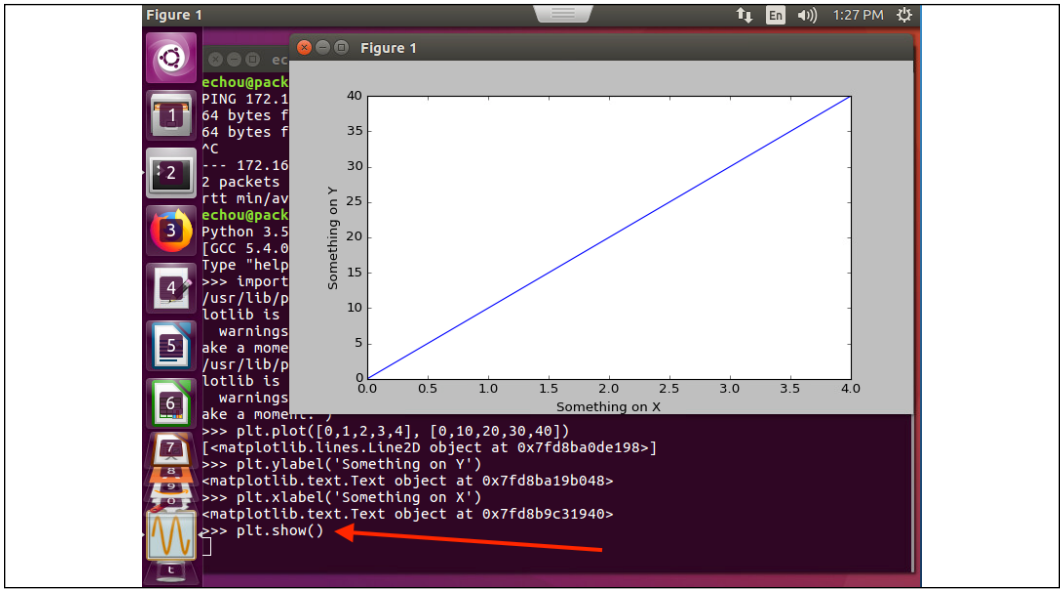


Figure 4: Matplotlib visualization with the Ubuntu desktop

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 show up as a line graph:

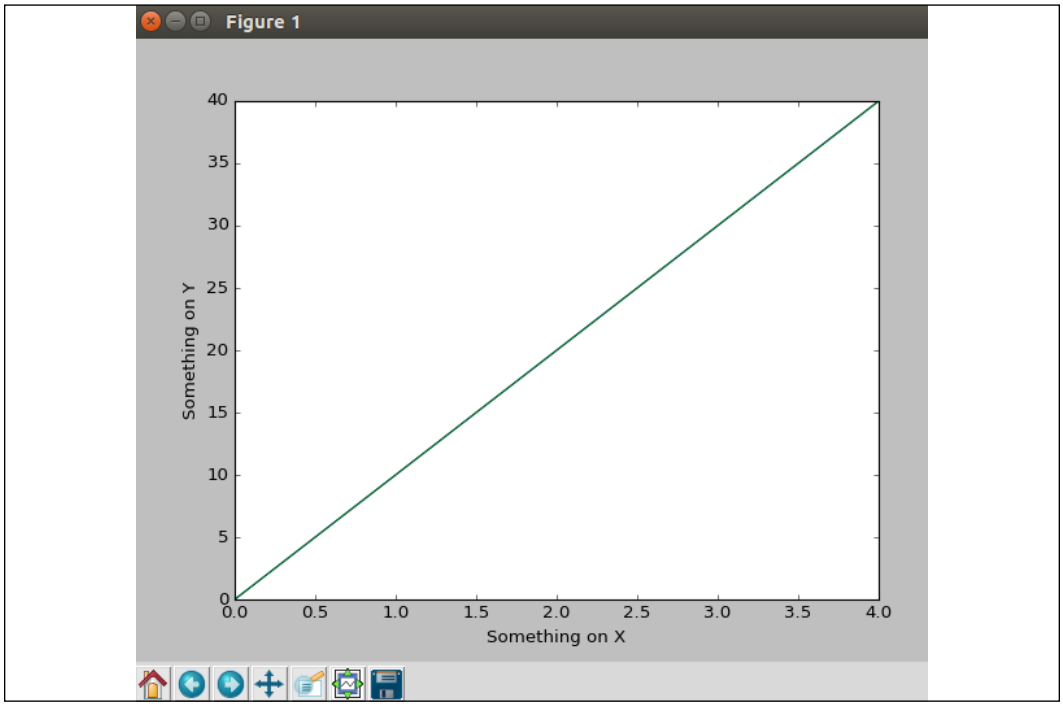


Figure 5: Matplotlib line graph

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, namely `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: `http://matplotlib.org/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 rotating 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:

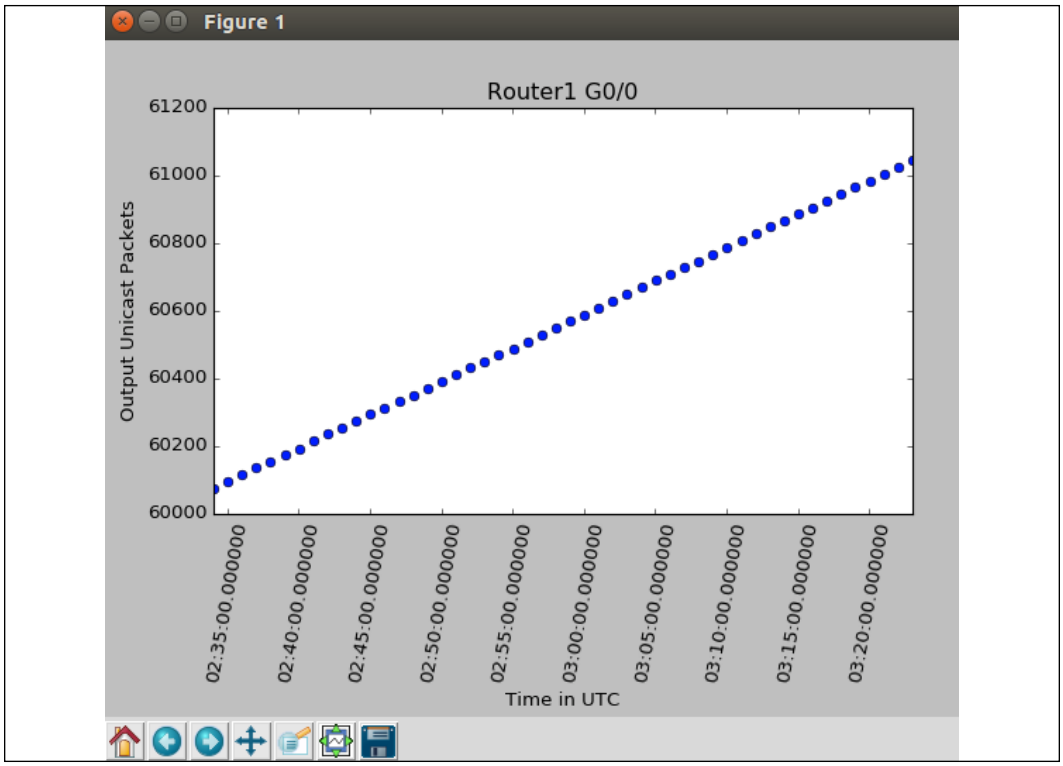


Figure 6: Router1 Matplotlib graph

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 in order to see the values:

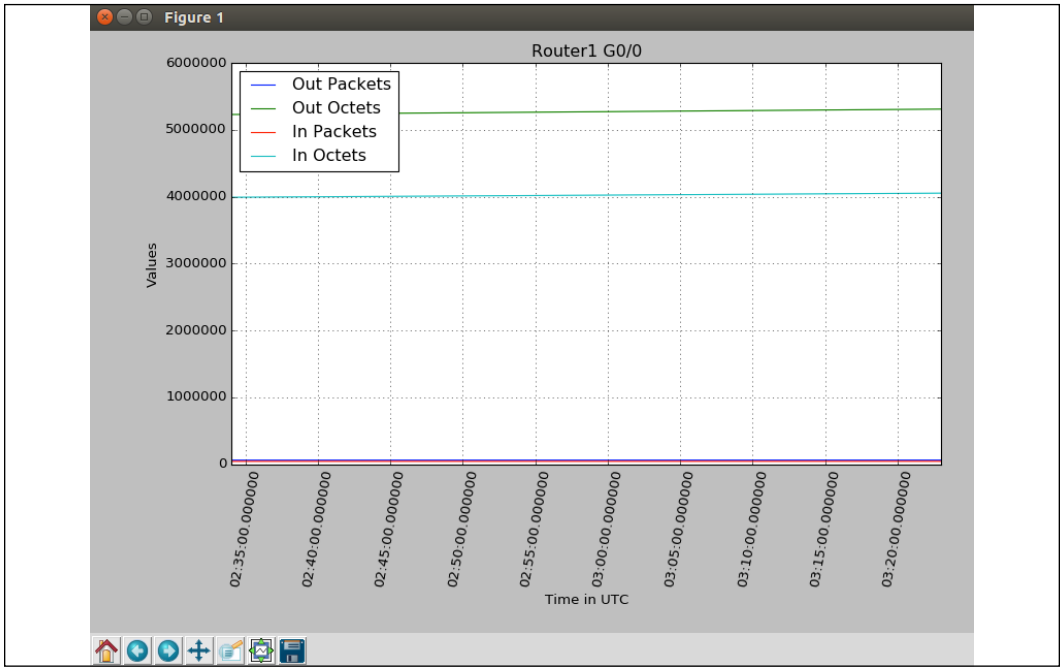


Figure 7: Router1 – Matplotlib multiline graph

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()`:

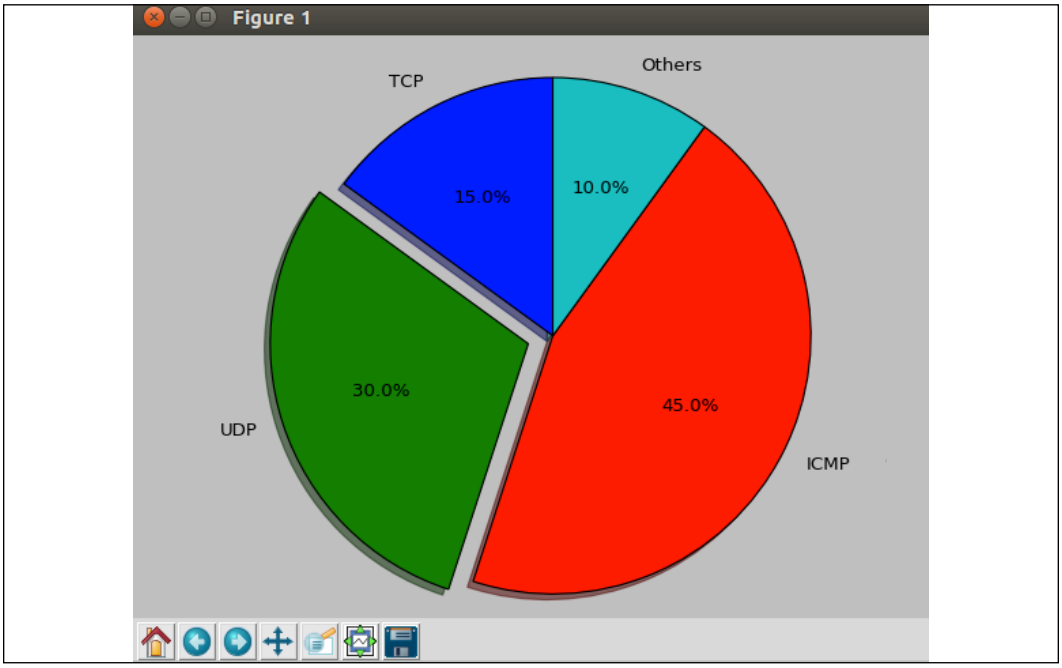


Figure 8: Matplotlib pie chart

In this section, we have used Matplotlib to graph our network data into more visually appealing graphs to help 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, there are many additional Matplotlib resources that 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, its aim is 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 `http://matplotlib.org/2.0.0/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 (`http://www.pygal.org/`) is a dynamic SVG charting library written in Python. The biggest advantage of Pygal, in my opinion, is that it produces **Scalable Vector Graphics** (**SVG**) format graphs easily and natively. There are many advantages of SVG over other graph formats, but 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. In years 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:

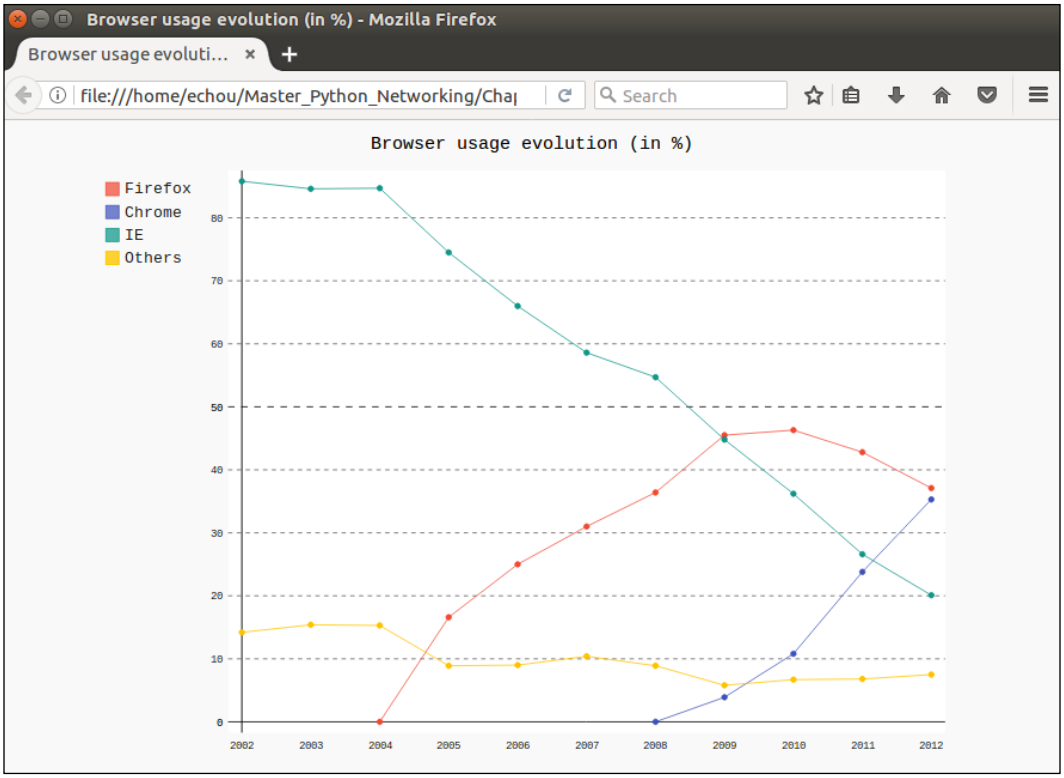


Figure 9: Pygal sample graph

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:

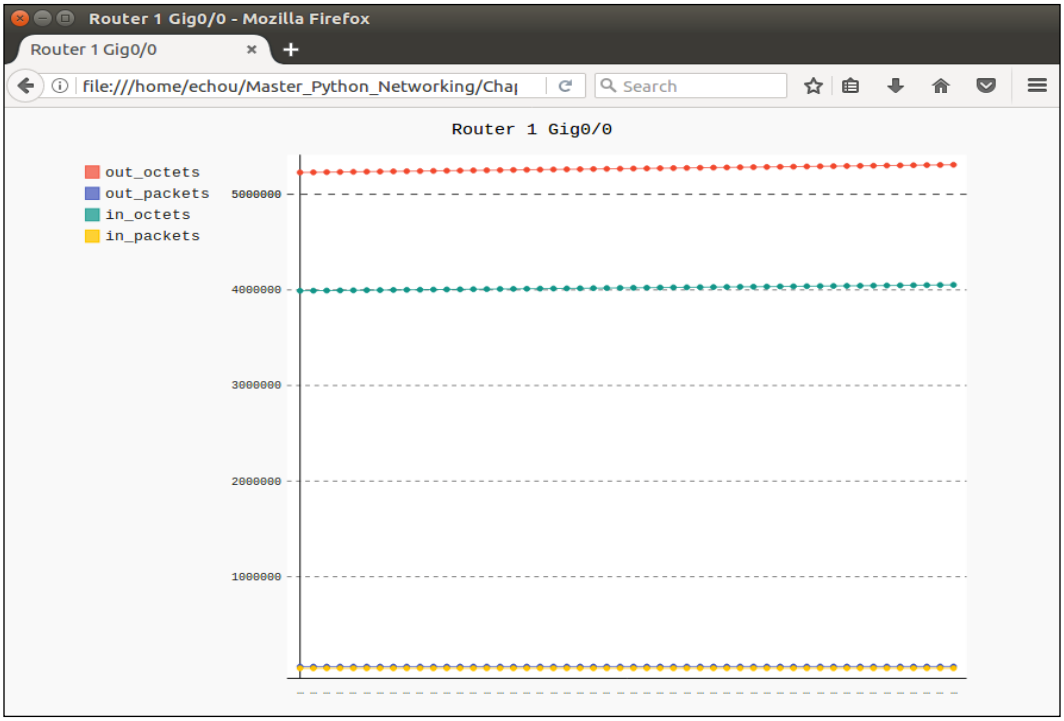


Figure 10: Router 1 — Pygal multiline graph

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:

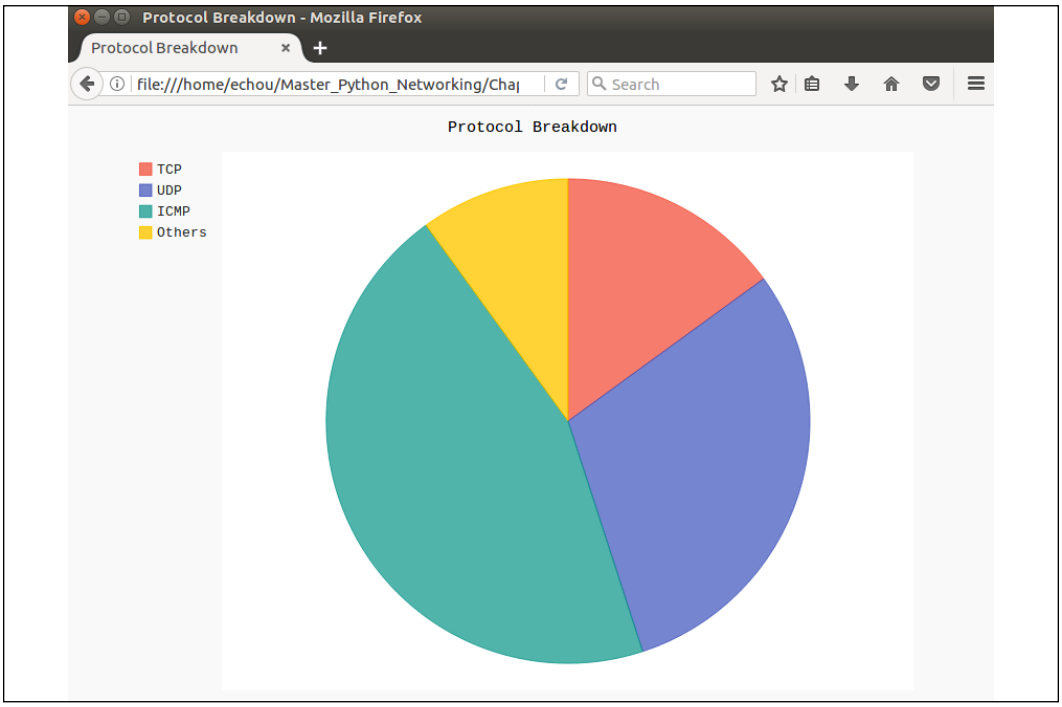


Figure 11: Pygal pie chart

Pygal is a great tool when it comes to generating production-ready SVG graphs. If this is the type of graph that is required, look no further than the Pygal library. In this section, we have looked at 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 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 working 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 really 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 itself. However, Cacti offers a custom data query feature that we can use Python for. 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

Due to the fact that 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 in our lab. 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 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 you are logged in, you 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 on: `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:

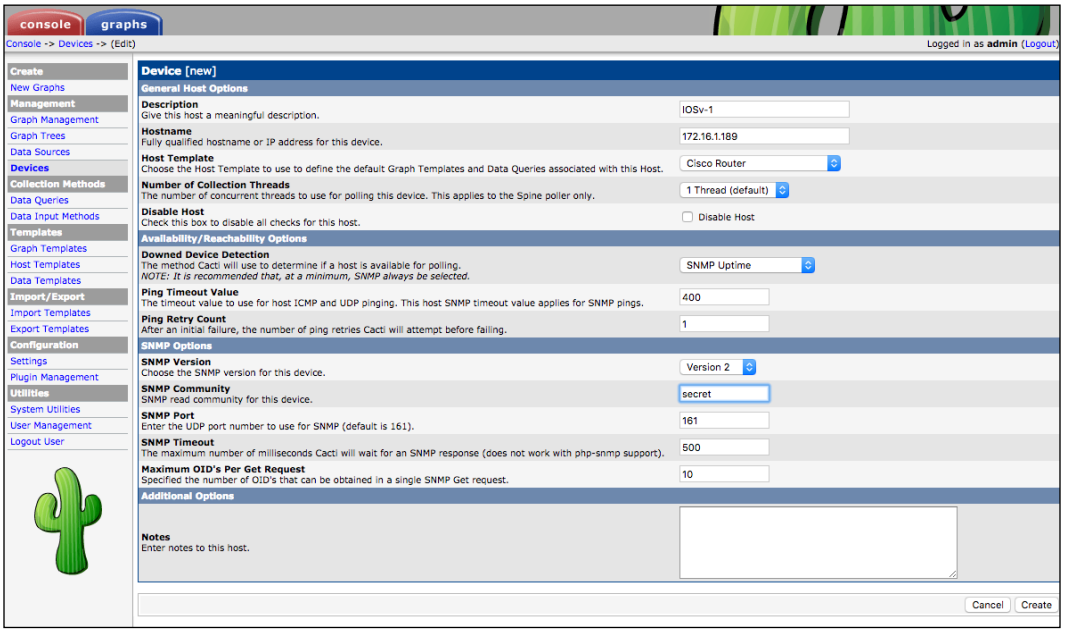


Figure 12: Cacti device edit page

A sign indicating the SNMP communication is working is when you can see the device uptime:

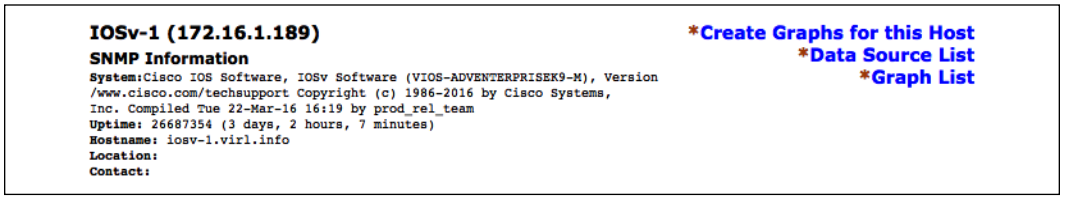


Figure 13: Device edit result page

You can add graphs to the device for interface traffic and other statistics:

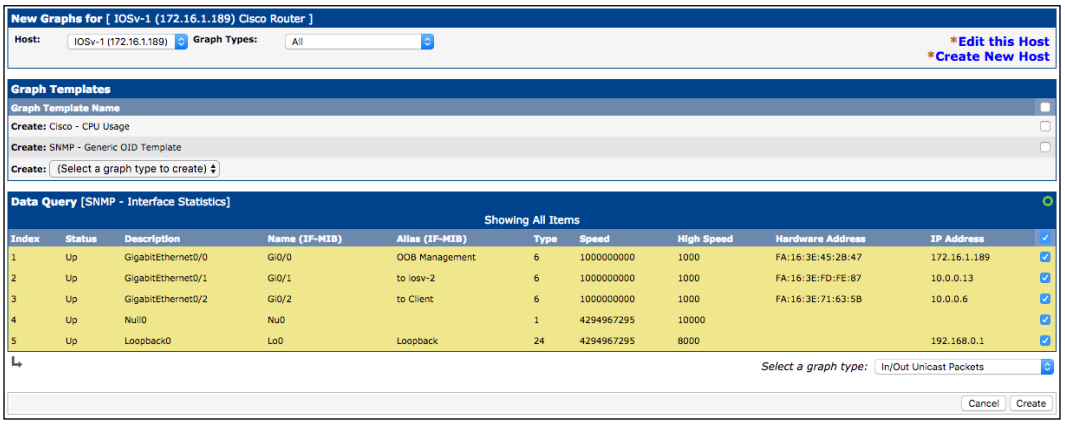


Figure 14: New graphs for device

After some time, you will start seeing traffic, as shown here:

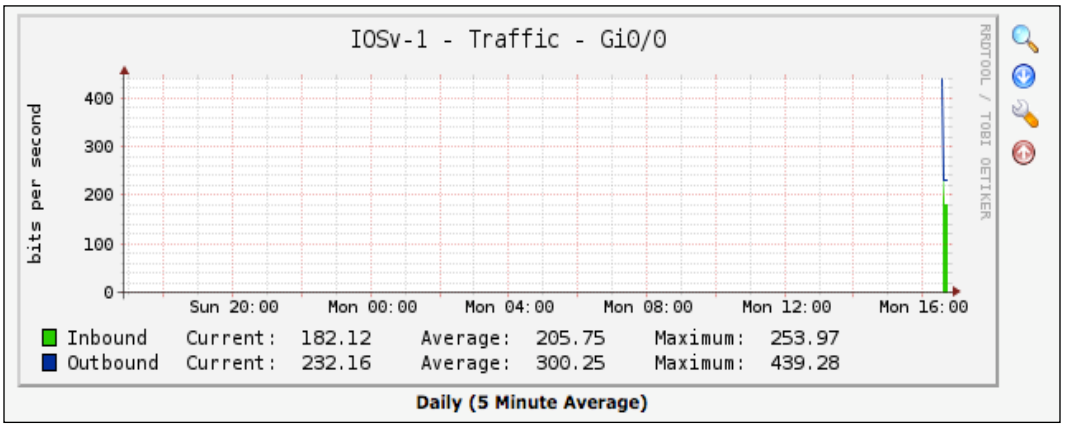


Figure 15: 5-minute average graph

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. 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, chances are that 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 to `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 172.16.1.173')
    child.expect(device_prompt)
    output = child.before
```

The output in its raw form will appear as follows:

```
b'sh ip access-lists permit_snmp | i 172.16.1.173rn 10 permit
172.16.1.173 log (6428 matches)rn'
```

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:

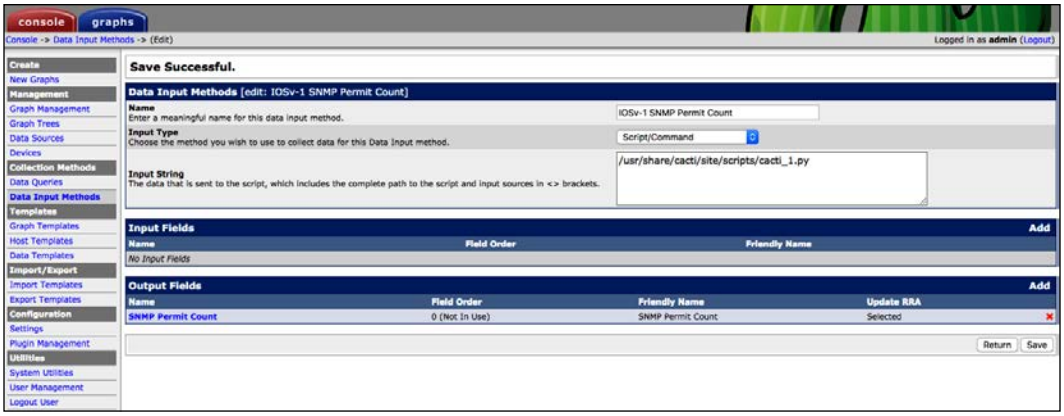


Figure 16: Data input method results page

SNMP is a common way to provide network monitoring services to devices. RRDtool with Cacti as the frontend provides a good platform to be used for all network devices via SNMP. We can also use Python scripts as a way 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 that is widely used in data science projects. Pygal can natively generate SVG format graphs that are flexible and web-friendly. We saw how we can generate line and pie graphs that are relevant for 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 can 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.

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 by 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. It's something we can graph 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 are constantly asking the device for a particular answer. This particular 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 over and over again, 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 there are 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 version 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 really 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 information

Let's start by looking at how to use Graphviz as a tool to monitor network topology changes.

Graphviz

Graphviz is an open source graph visualization software. Imagine if 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 for us. 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 constructing the lab we will be using.

Lab setup

We will use VIRL to construct our lab. As in the previous chapters, we will put together a lab with multiple routers, a server, and a client.

We will use five IOSv network nodes along with two Ubuntu hosts:

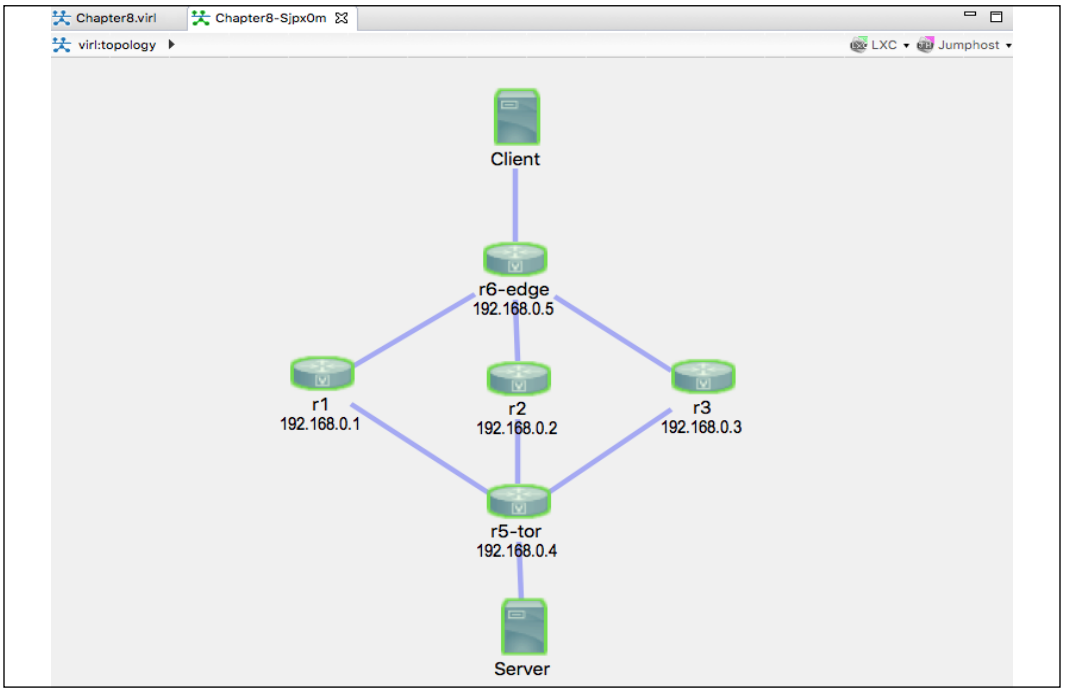


Figure 1: Lab topology

If you are wondering about our choice of IOSv as opposed to NX-OS or IOS-XR and the number of devices, here are a few points for you to consider when you build your own lab:

- Nodes virtualized by NX-OS and IOS-XR are much more memory-intensive than IOS.
- The VIRL virtual manager I am using has 16 GB of RAM, which seems enough to sustain 9 nodes but could be a bit unstable (with nodes changing from reachable to unreachable at random).
- If you wish to use NX-OS, consider using NX-API or other API calls that will return structured data.

For our example, we are going to use LLDP as the protocol for link-layer neighbor discovery because it is vendor-neutral. Note that VIRL provides an option to automatically enable CDP, which can save you some time and is similar to LLDP in functionality; however, it is a Cisco proprietary technology so we will disable it for our lab:

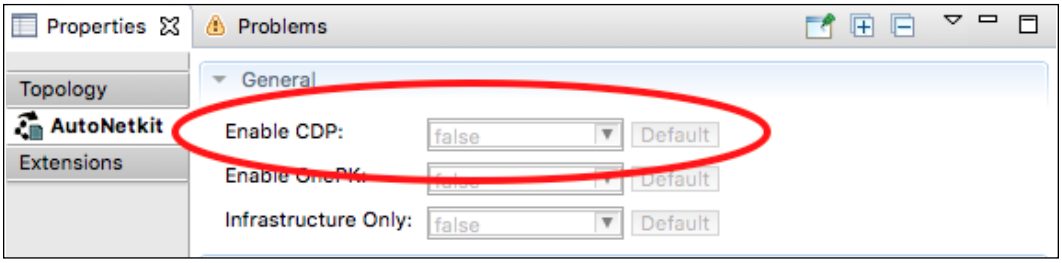


Figure 2: CDP option for VIRL

Once the lab is up and running, 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 2.40.1 (20161225.0304)
```

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__
'0.13'
>>> 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:

```
$ dot -Tpng chapter8_gv_1.gv -o output/chapter8_gv_1.png
```

The resultant graph can be viewed from the following output folder:

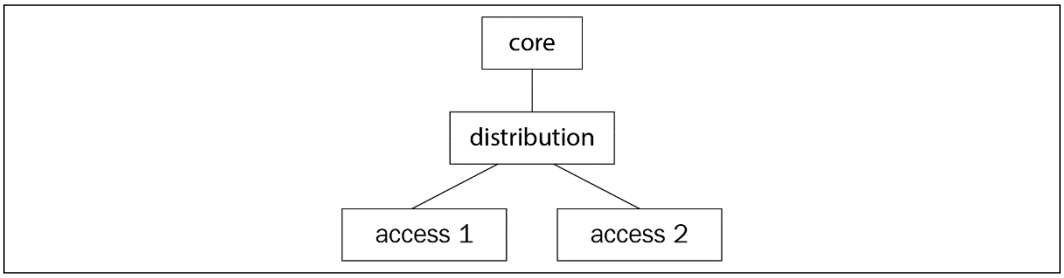


Figure 3: Graphviz undirected dot graph example

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 as well as 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, and so on. 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:

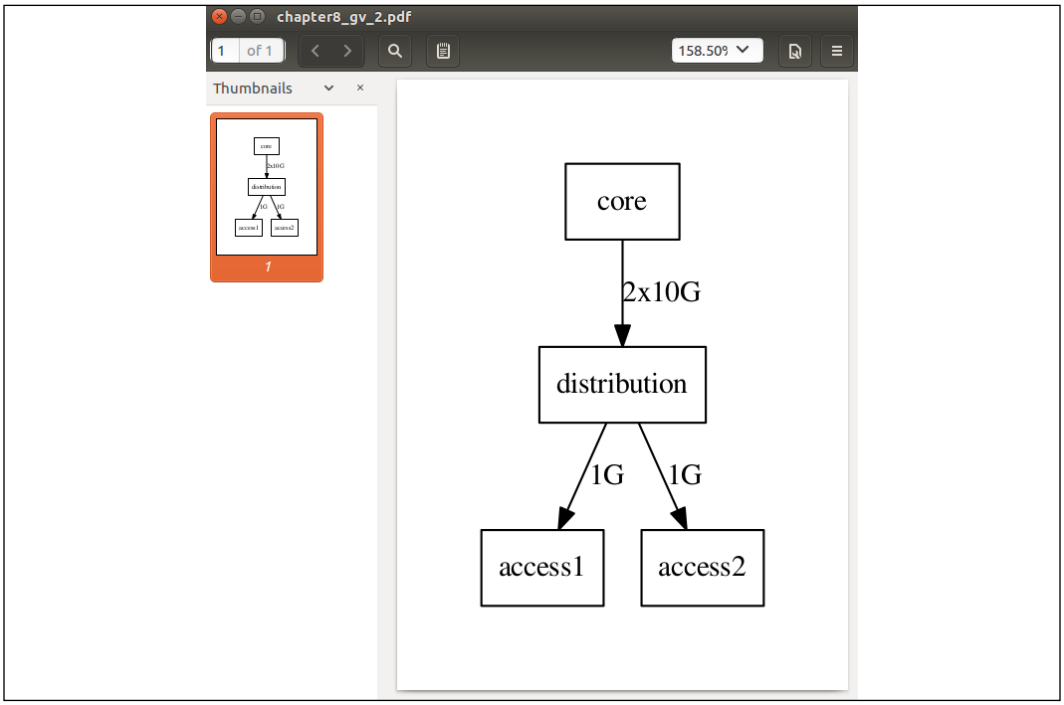


Figure 4: Network graph with directional arrow and line descriptions

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 presented with 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: 172.16.1.218
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
r2.virl.info        Gi0/0          120        R               Gi0/0
r3.virl.info        Gi0/0          120        R               Gi0/0
r5.virl.info        Gi0/2          120        R               Gi0/1
r5.virl.info        Gi0/0          120        R               Gi0/0
r6.virl.info        Gi0/0          120        R               Gi0/0

Device ID           Local Intf     Hold-time  Capability      Port ID
r6.virl.info        Gi0/1          120        R               Gi0/1

Total entries displayed: 6
```

In the output, you will see that `G0/0` is configured as the MGMT interface; therefore, you will see LLDP peers as if they are on a flat management network. What we really care about is the `G0/1` and `G0/2` interfaces connected to other peers. This knowledge 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:

```
(venv) $ ls -l tmp
total 100
-rw-rw-r-- 1 echou echou   772 Oct  1 17:17 r1_lldp_output.txt
-rw-rw-r-- 1 echou echou   772 Oct  1 17:17 r2_lldp_output.txt
-rw-rw-r-- 1 echou echou   772 Oct  1 17:17 r3_lldp_output.txt
-rw-rw-r-- 1 echou echou   843 Oct  1 17:17 r5-tor_lldp_output.txt
-rw-rw-r-- 1 echou echou   843 Oct  1 17:17 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", "veos01              Gi0/0          120
B               Ethernet1", "r2.virl.info        Gi0/0          120
R               Gi0/0", "r3.virl.info        Gi0/0          120
R               Gi0/0", "r5.virl.info        Gi0/2          120
R               Gi0/1", "r5.virl.info        Gi0/0          120
R               Gi0/0", "r6.virl.info        Gi0/0          120
R               Gi0/0", "r6.virl.info        Gi0/1          120        R
Gi0/1", "", "Total entries displayed: 7"]]
```

So far, we have worked on retrieving information from the 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 of the LLDP neighbors have disappeared due to link failure or other issues. Let's take a 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'`):

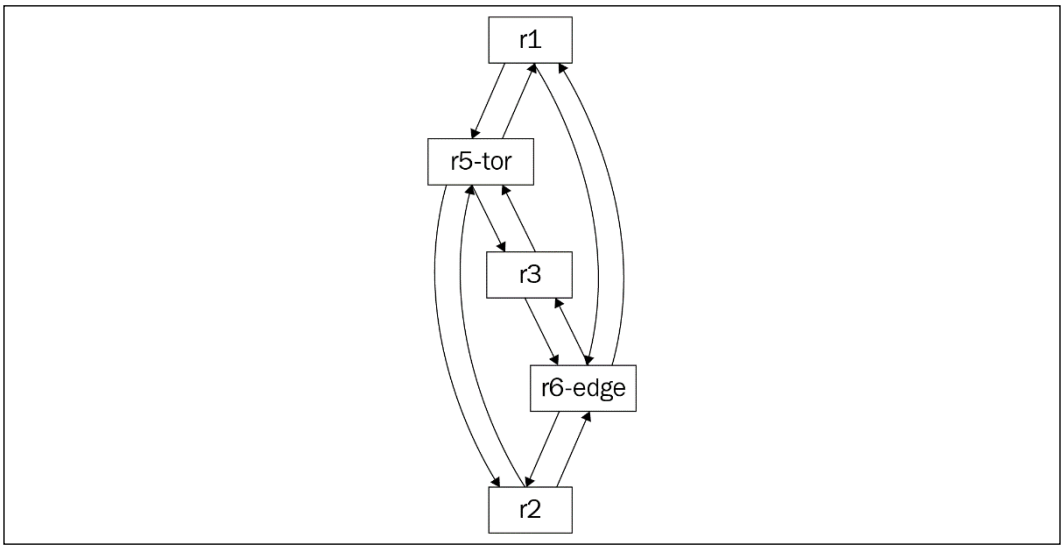


Figure 5: Topology graph 1

The `neato` layout represents an attempt to draw undirected graphs with even less hierarchy:

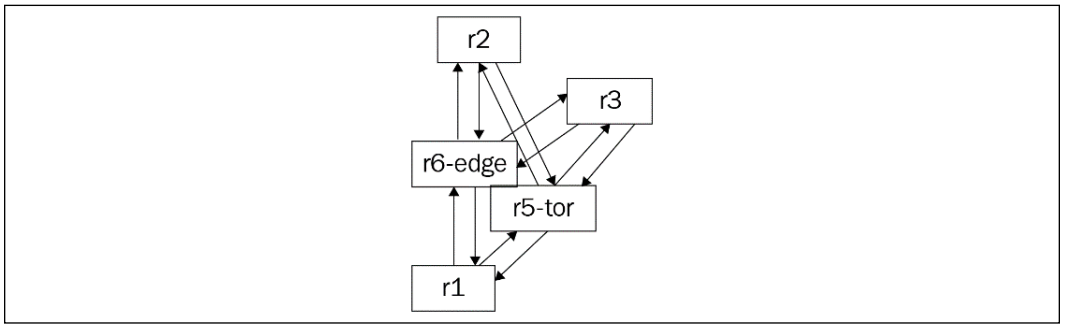


Figure 6: Topology graph 2

Sometimes, the default layout presented by the tool is just fine, especially if your goal is to detect faults as opposed to making it 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:

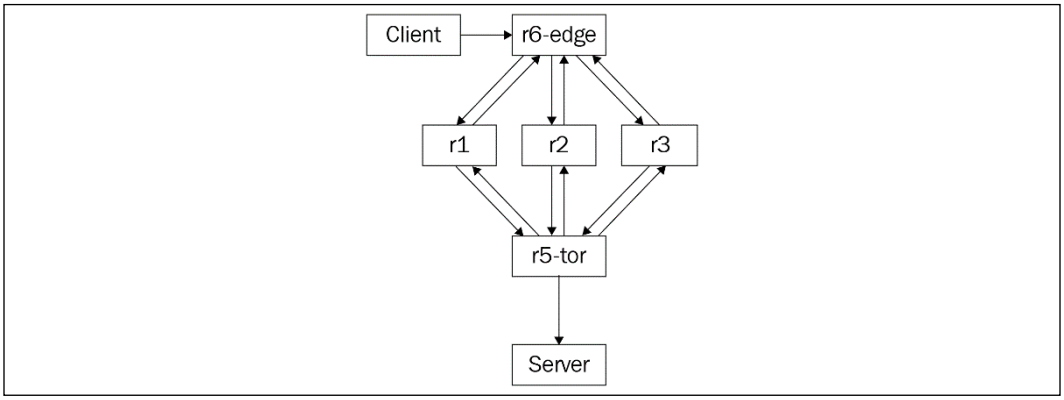


Figure 7: Topology graph 3

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 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:

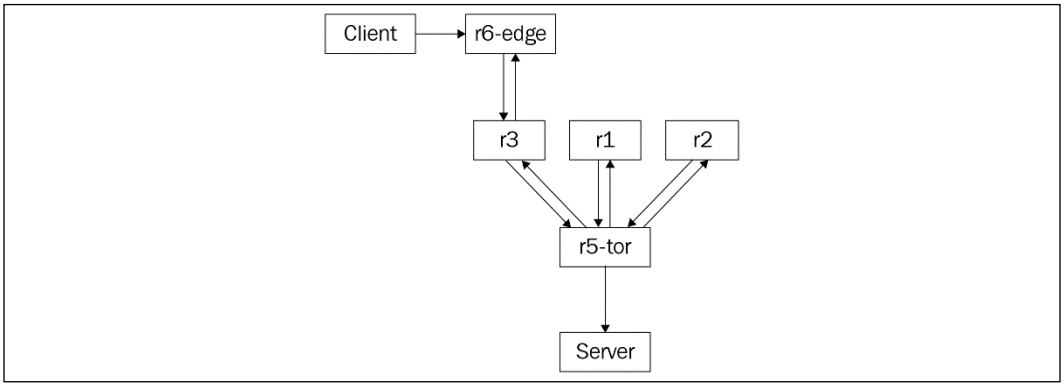


Figure 8: Toplogy graph 4

This is a relatively long example to demonstrate 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 make the argument 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 choose to 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 a number of 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 aggregate packets by matching the set of tuples (source interface, source IP/port, destination IP/port, and so on). Here, once a flow is completed, the network device exports the flow characteristics, including total bytes and packet counts in the flow, 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 will be able to parse IPFIX as long as you match the template definition.

- **sFlow**: sFlow actually 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 being transported on the wire. This gives us a way to look at the NetFlow packet in detail as well as to troubleshoot any NetFlow issues when it is not working as expected.

First, let's generate some traffic between the client and server across the VIRL 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 172.16.1.123 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 take a 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 as well as 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 are applicable 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/3.7/library/socket.html` and `https://docs.python.org/3.7/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 quick 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 9: 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 size of the record 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, there are other tools that 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

Just 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 include 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 a number of 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 a number of comparison studies for different tools.

Each one of them has its own strengths and weaknesses; which one to use is really a matter of preference, platform, and our appetite for customization. I would recommend choosing a tool that would support both v5 and v9, and potentially sFlow as well. 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 that 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 would recommend installing ntop on a separate host than the management station in production.

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 172.16.1.123 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 172.16.1.219 255.255.255.0
 ip flow ingress
 ip flow egress
<skip>
!
ip flow-export version 5
ip flow-export destination 172.16.1.123 5556 vrf Mgmt-intf
!
```

Once everything is set up, you can check the **ntop** web interface for local IP traffic:

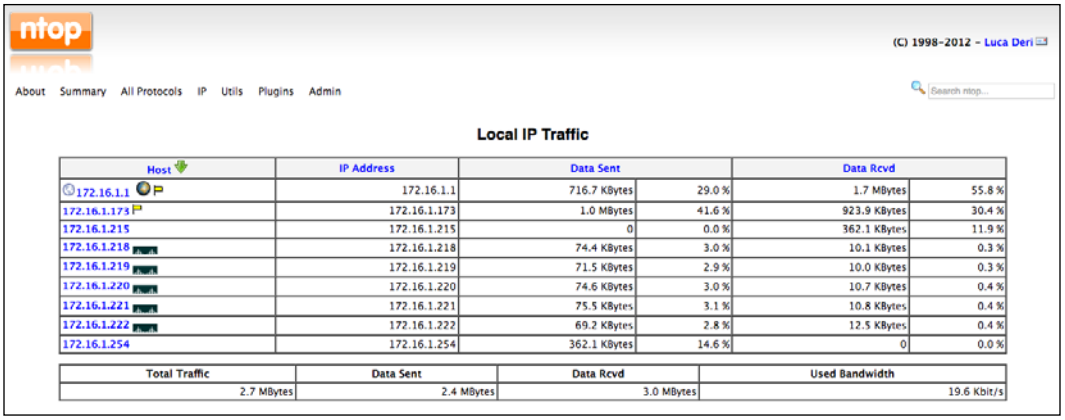
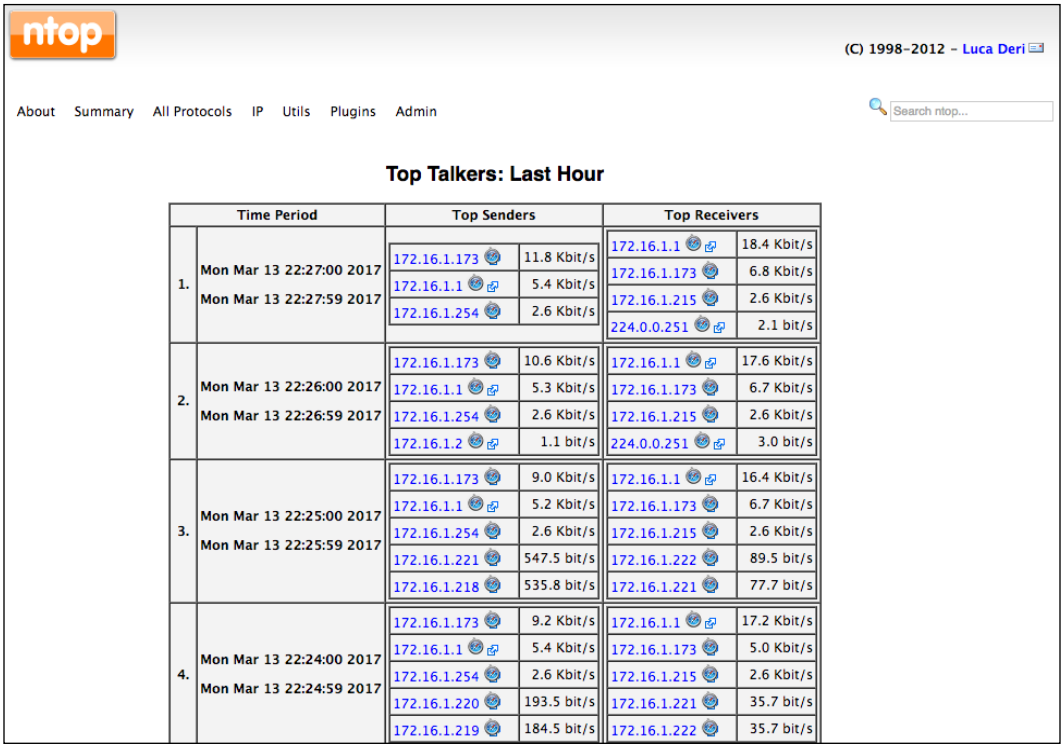


Figure 10: Ntop Local IP Traffic

One of the most often used features of ntop is using it to look at the Top Talkers graph:



	Time Period	Top Senders		Top Receivers	
1.	Mon Mar 13 22:27:00 2017 Mon Mar 13 22:27:59 2017	172.16.1.173	11.8 Kbit/s	172.16.1.1	18.4 Kbit/s
		172.16.1.1	5.4 Kbit/s	172.16.1.173	6.8 Kbit/s
		172.16.1.254	2.6 Kbit/s	172.16.1.215	2.6 Kbit/s
				224.0.0.251	2.1 bit/s
2.	Mon Mar 13 22:26:00 2017 Mon Mar 13 22:26:59 2017	172.16.1.173	10.6 Kbit/s	172.16.1.1	17.6 Kbit/s
		172.16.1.1	5.3 Kbit/s	172.16.1.173	6.7 Kbit/s
		172.16.1.254	2.6 Kbit/s	172.16.1.215	2.6 Kbit/s
		172.16.1.2	1.1 bit/s	224.0.0.251	3.0 bit/s
3.	Mon Mar 13 22:25:00 2017 Mon Mar 13 22:25:59 2017	172.16.1.173	9.0 Kbit/s	172.16.1.1	16.4 Kbit/s
		172.16.1.1	5.2 Kbit/s	172.16.1.173	6.7 Kbit/s
		172.16.1.254	2.6 Kbit/s	172.16.1.215	2.6 Kbit/s
		172.16.1.221	547.5 bit/s	172.16.1.222	89.5 bit/s
		172.16.1.218	535.8 bit/s	172.16.1.221	77.7 bit/s
4.	Mon Mar 13 22:24:00 2017 Mon Mar 13 22:24:59 2017	172.16.1.173	9.2 Kbit/s	172.16.1.1	17.2 Kbit/s
		172.16.1.1	5.4 Kbit/s	172.16.1.173	5.0 Kbit/s
		172.16.1.254	2.6 Kbit/s	172.16.1.215	2.6 Kbit/s
		172.16.1.220	193.5 bit/s	172.16.1.221	35.7 bit/s
		172.16.1.219	184.5 bit/s	172.16.1.222	35.7 bit/s

Figure 11: 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 in order 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 perform 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.

There are several resources that 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 12: 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 13: Plugin directories

Under **About | Online Documentation | Python ntop Engine**, there are links for the Python API as well as the tutorial:

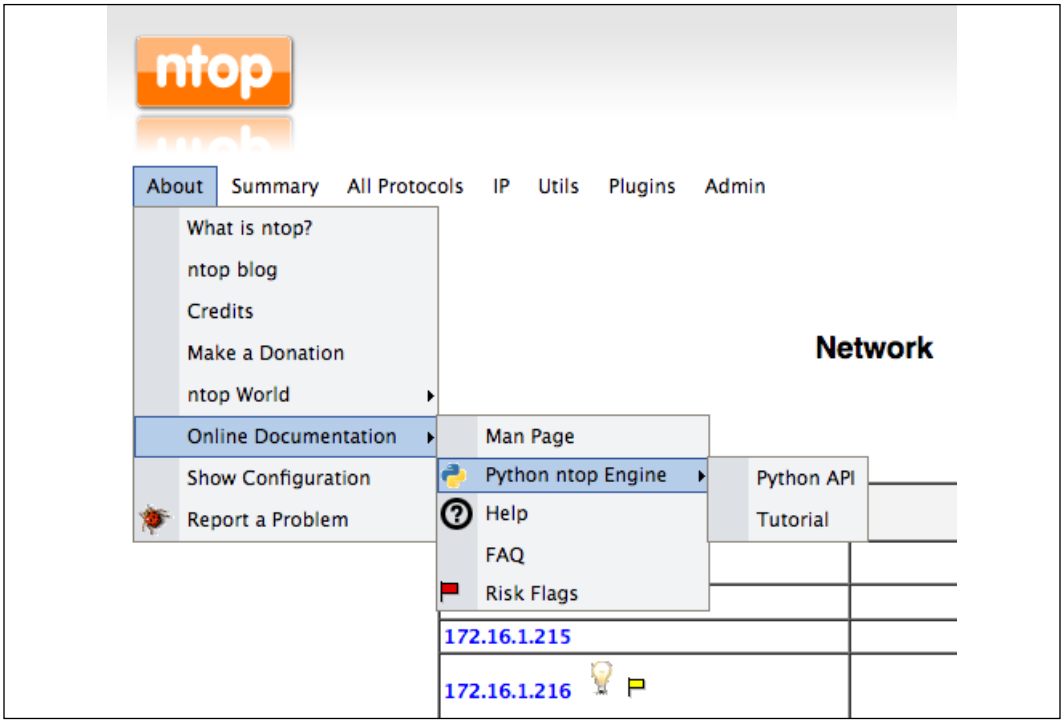


Figure 14: Python ntop documentation

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 is used to drill 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 use 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:

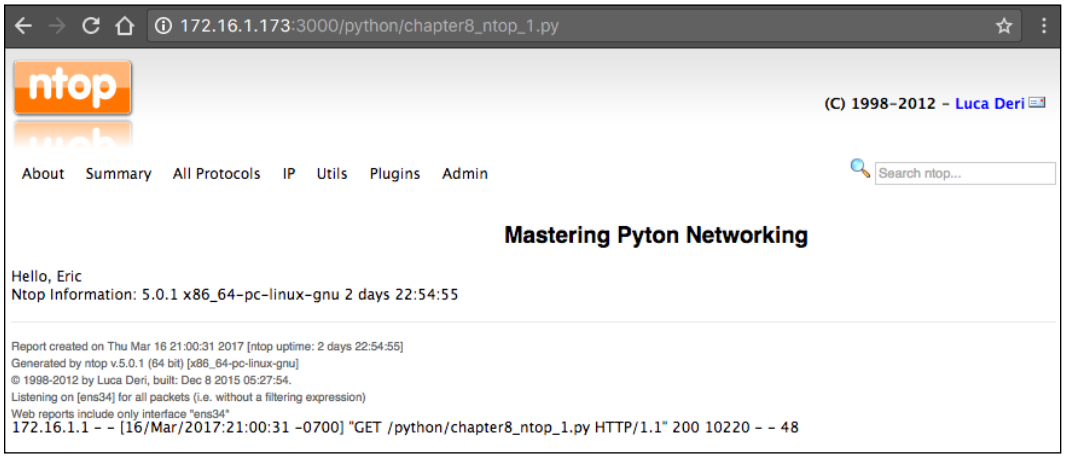


Figure 15: Ntop script result

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 16: 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 derive an estimate of 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 that our VIRL lab devices do not support (not even with the NX-OSv virtual switches). You can either 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 happen to have access to a Cisco Nexus 3048 switch running 7.0 (3), which I will be using 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:

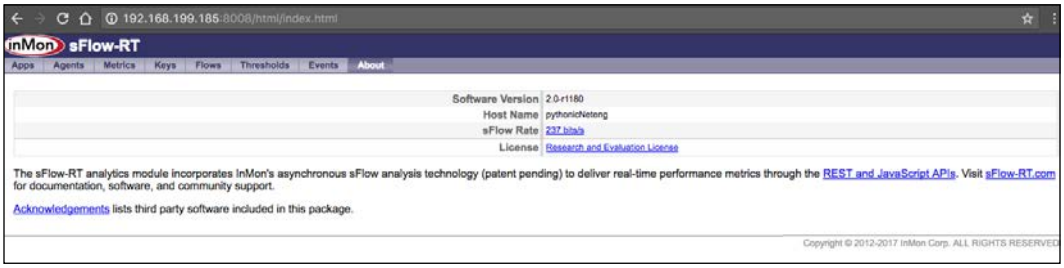


Figure 17: sFlow-RT version

As soon as sFlow-RT receives any sFlow packets, the agents and other metrics will appear:

Figure 18: 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.

If you have read the previous editions of this book, you will see the *ELK Stack* section has been removed from this chapter. In this edition, we will dedicate a complete chapter to Elastic Stack.

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 definitely 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 take a 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 by 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.

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 router 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, 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 information exchange.

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 to a RESTful API.

On the provider side, one of the advantages of providing RESTful services to users is the ability to hide the 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 in the form of 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 then open up the endpoints only as needed.

As the master of our own 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 having to know 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 own 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 get started 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 of the Python web frameworks. There is even an annual conference held for Django, one of the most popular Python frameworks, called DjangoCon. It attracts hundreds of attendees every year. As of the writing of this book in late 2019, the first-ever Flask Conference took place in Brazil in 2018 and a more general PyConWeb took place in the spring of 2019. Did I mention Python has a thriving web development community?

If you sort the Python web frameworks on: `https://hotframeworks.com/languages/python`, you can see that there is no shortage of choices when it comes to Python and web frameworks:

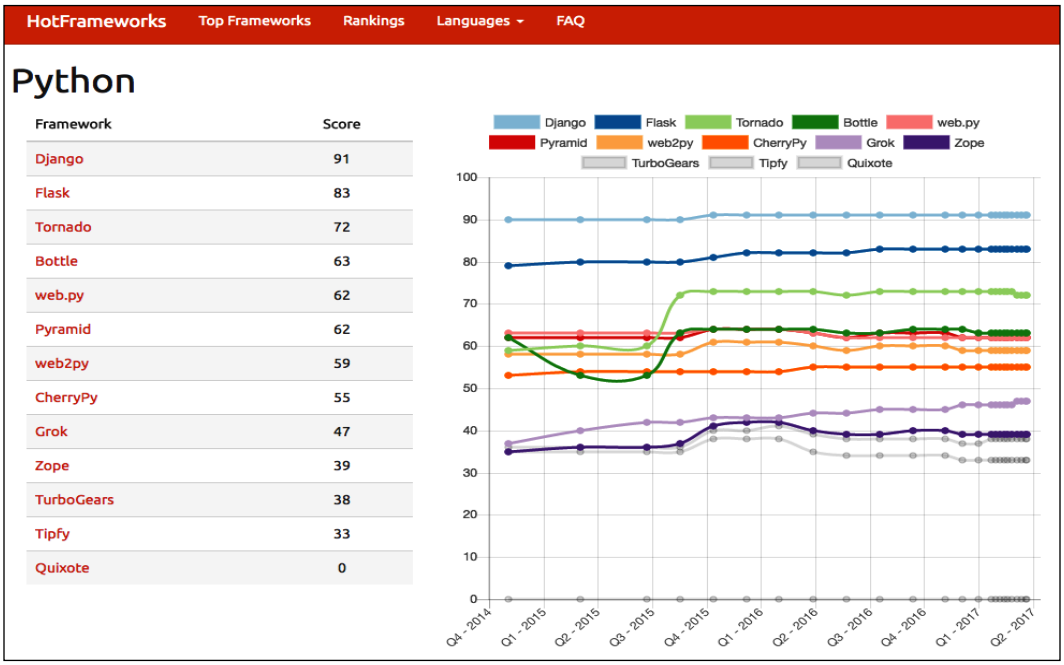


Figure 1: Python web framework rankings

With so many options to choose from, which framework should we pick? Clearly, trying all the frameworks out yourself would be really 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 have my own bias toward programming languages (Python!) and web frameworks (Flask!). In this section, I hope to convey to you my reasoning behind choosing one over the other. 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 being easy to extend when needed. The "micro" in microframework does not mean that Flask is lacking in functionality, nor does it mean it cannot work in a production environment.

Personally, 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/2.2/ref/databases/`), you will notice that the framework supports a number of different SQL databases. However, they are all variants of a SQL database such as MySQL, PostgreSQL, SQLite, and others.

What if you want to use a NoSQL database such as MongoDB or CouchDB? It might be possible but could be leaving you in your own hands. Being an opinionated framework is certainly not a bad thing, it is just a matter of opinion (no pun intended).

The idea of keeping the core code small and extending it when needed is very appealing when you 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 your own extensions, such as a decorator, is pretty 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 out of the box.

As you can see, besides Django, Flask is the second most popular Python framework by some measure. The popularity that comes with community contribution, support, and quick development helps it further expand its reach.

For the preceding reasons, I feel that 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. If you prefer, you can start a new virtual environment, or you can continue to use the existing virtual environment that we have been using up to this point.

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 at the end of the process:

```
(venv) $ 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

(venv) $ pip install -r requirements.txt
```

For our network topology, we will use a simple four-node network, as shown here:

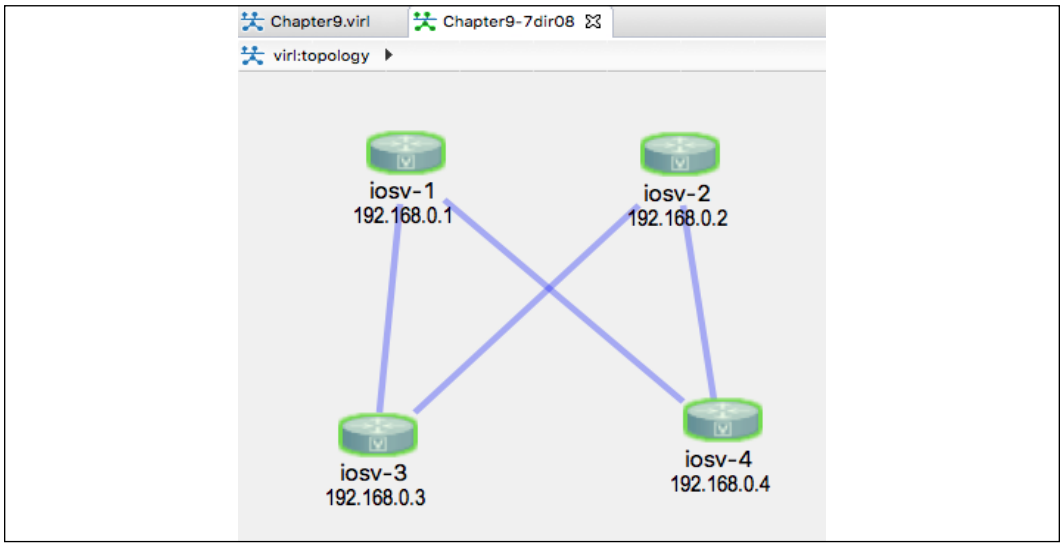


Figure 2: Lab topology

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/1.1.x/`. If you'd like to dig deeper into Flask, the project documentation would be a great place to start.

I would also 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. In fact, Miguel's class, *Building Web APIs with Flask*, inspired me to write this chapter. You can take a look at his published code on GitHub: `https://github.com/miguelgrinberg/oreilly-flask-apis-video`.

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 will almost be your design pattern for Flask in the following examples. 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 (`https://docs.python.org/3.7/library/__main__.html`). We also add the host and debug options, which allow more verbose output and allow us to listen on all the interfaces of the host (by default, it only listens on loopback). We can run this application using the development server:

```
(venv) $ python chapter9_1.py
 * Serving Flask app "chapter9_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 141-973-077
```

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 reading 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 that 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:

```
(venv) $ http http://192.168.2.123:5000
HTTP/1.0 200 OK
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Tue, 08 Oct 2019 19:06:23 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

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://192.168.2.123:5000`.

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 HTTP Bin (`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:

```
$ http GET https://httpbin.org/user-agent
<skip>
{
    "user-agent": "HTTPie/1.0.3"
}
```

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`:

```
$ http POST https://httpbin.org/post name=eric twitter=at_ericchou
married:=true
<skip>
Content-Type: application/json
<skip>

{
    <skip>
    "headers": {
        "Accept": "application/json, */*",
        <skip>
        "User-Agent": "HTTPie/1.0.3"
    },
    "json": {
        "married": true,
        "name": "eric",
        "twitter": "at_ericchou"
    },
    <skip>
    "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.org/doc#usage`.

Getting back to our Flask program, a large part of API building is based on the flow of URL routing. Let's take a deeper look 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://192.168.2.123:5000
<skip>


You are at index()


$ http http://192.168.2.123:5000/routers/
<skip>


You are at routers()
```

As the requests are made from the client side, the server screen will see the requests coming in:

```
(venv) $ python chapter9_2.py
<skip>
```

```
192.168.2.123 - - [08/Oct/2019 12:43:08] "GET / HTTP/1.1" 200 -
192.168.2.123 - - [08/Oct/2019 12:43:18] "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
(venv) $ python chapter9_3.py

(venv) # Client Side
$ http http://192.168.2.123:5000/routers/host1
HTTP/1.0 200 OK
<skip>

You are at host1

(venv) $ http http://192.168.2.123:5000/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://192.168.2.123:5000/routers/host1/interface/one
HTTP/1.0 404 NOT FOUND
<skip>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//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
(venv) $ python chapter9_4.py
<skip>
/r1/list_interfaces
/r2/list_interfaces
/r3/list_interfaces

# client side
(venv) $ http http://192.168.2.123:5000/r1/list_interfaces
<skip>

Listing interfaces for r1

(venv) $ http http://192.168.2.123:5000/r2/list_interfaces
<skip>

Listing interfaces for r2

# bad request
(venv) $ http http://192.168.2.123:5000/r1000/list_interfaces
<skip>

Invalid hostname
```

For now, you can think of `app.text_request_context()` as a dummy request object that is necessary for demonstration purposes. If you are interested in the local context, feel free to take a look at `http://werkzeug.pocoo.org/docs/0.14/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:

```
(venv) $ http http://192.168.2.123:5000/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 a database abstraction and object-relational mapper, respectively. It's a fancy way of saying 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 take a 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:

```
(venv) $ python chapter9_db_1.py
(venv) $ ls -l network.db
-rw-r--r-- 1 echou echou 3072 Oct 8 15:38 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:

```
(venv) $ 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, `iosv-1` and `iosv-2`, are in the network role of the spine. The other two devices, `iosv-3` and `iosv-4`, are in our network service as leaves. These are obviously 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:

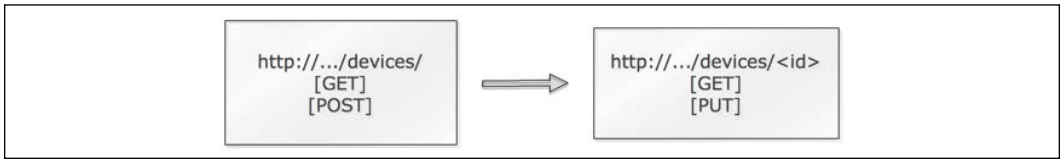


Figure 3: Network content API

The first API will be our `http://172.16.1.123/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 to have 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://172.16.1.123/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 end result, I am going to jump ahead and show the end result quickly before we take a look at the code. If you want to follow along with 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:

```
(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-1'
'loopback'='192.168.0.1'
'mgmt_ip'='172.16.1.225'
'role'='spine'
'vendor'='Cisco'
'os'='15.6'

HTTP/1.0 201 CREATED
Content-Length: 3
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:15:31 GMT
Location: http://172.16.1.123:5000/devices/1
Server: Werkzeug/0.16.0 Python/3.6.8
{}
```

I can repeat the preceding step for the three additional devices:

```
(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-2'
'loopback'='192.168.0.2'
'mgmt_ip'='172.16.1.226'
'role'='spine'
'vendor'='Cisco'
'os'='15.6'

(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-3',
```

```
'loopback'='192.168.0.3'
'mgmt_ip'='172.16.1.227'
'role'='leaf'
'vendor'='Cisco'
'os'='15.6'

(venv) $ http POST http://172.16.1.123:5000/devices/
'hostname'='iosv-4',
'loopback'='192.168.0.4'
'mgmt_ip'='172.16.1.228'
'role'='leaf'
'vendor'='Cisco'
'os'='15.6'
```

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:

```
(venv) $ http GET http://172.16.1.123:5000/devices/
HTTP/1.0 200 OK
Content-Length: 192
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:21:12 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "device": [
        "http://172.16.1.123:5000/devices/1",
        "http://172.16.1.123:5000/devices/2",
        "http://172.16.1.123:5000/devices/3",
        "http://172.16.1.123:5000/devices/4"
    ]
}
```

Similarly, using the `GET` request for `/devices/<id>` will return specific information related to the device:

```
(venv) echou@network-dev-2:~$ http GET http://172.16.1.123:5000/devices/1
<skip>
{
    "hostname": "iosv-1",
```

```
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "15.6",
    "role": "spine",
    "self_url": "http://172.16.1.123:5000/devices/1",
    "vendor": "Cisco"
}
```

Let's pretend we have downgraded the `r1` operating system from `15.6` to `14.6`. We can use the `PUT` request to update the device record:

```
(venv) $ http PUT http://172.16.1.123:5000/devices/1
'hostname'='iosv-1'
'loopback'='192.168.0.1'
'mgmt_ip'='172.16.1.225'
'role'='spine'
'vendor'='Cisco'
'os'='14.6'
HTTP/1.0 200 OK
# Verification
(venv) $ http GET http://172.16.1.123:5000/devices/1HTTP/1.0 200 OK
<skip>

{
    "hostname": "iosv-1",
    "loopback": "192.168.0.1",
    "mgmt_ip": "172.16.1.225",
    "os": "14.6",
    "role": "spine",
    "self_url": "http://172.16.1.123:5000/devices/1",
    "vendor": "Cisco"
}
```

Now, let's take a 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 out, 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 as well as 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://172.16.1.173:5000/devices/4
Server: Werkzeug/0.9.6 Python/3.5.2
```

Let's look at the API that queries and returns information pertaining to 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, which is why 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 that 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('iosv-1', 'iosv-1#', '172.16.1.225', 'cisco',
'cisco'))
('iosv-1', 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 9Z1DS4YEJWHZGVUM73HWA\r\n')
```

Make sure that your Pexpect script works before you proceed. The following code assumes that you 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:

```
(venv) $ http GET http://172.16.1.123:5000/devices/1/version
HTTP/1.0 200 OK
Content-Length: 212
Content-Type: application/json
Date: Tue, 08 Oct 2019 23:53:49 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "version": "('iosv-1', 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 9Z1DS4YEJWHZGVUM73HWA\\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:

```
(venv) $ http GET http://172.16.1.123:5000/devices/spine/version
HTTP/1.0 200 OK
<skip>
{
    "iosv-1": "('iosv-1', 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 9Z1DS4YEJWHZGVUM73HWA\\r\\n')",
    "iosv-2": "('iosv-2', 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 9BPSF4VEH068CWL8YVZGT\\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 on GitHub. 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:

```
(venv) $ http GET http://172.16.1.123:5000/devices/spine/version
HTTP/1.0 202 ACCEPTED
Content-Length: 3
Content-Type: application/json
```

```
Date: Tue, 08 Oct 2019 23:58:57 GMT
Location: http://172.16.1.123:5000/status/057c895371b448d2aad30525c31e
1c51
Server: Werkzeug/0.16.0 Python/3.6.8
{}
```

We can then make a second request to the location to retrieve the result:

```
(venv) $ http GET http://172.16.1.123:5000/status/057c895371b448d2aad3052
5c31e1c51
HTTP/1.0 200 OK
<skip>
{
    "iosv-1": "('iosv-1', 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 9Z1DS4YEJWHZGVUM73HWA\\r\\n')",
    "iosv-2": "('iosv-2', 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 9BPSF4VEH068CWL8YVZGT\\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://172.16.1.123:5000'
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(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(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`:

```
(venv) $ python chapter9_request_1.py
Status: 202 Resource: http://172.16.1.123:5000/status/6108048c6e9b40fbab5a5b53c5817e7c
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 to us, 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` 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:

```
(venv) $ http GET http://172.16.1.123:5000/devices/
HTTP/1.0 401 UNAUTHORIZED
<skip>
WWW-Authenticate: Basic realm="Authentication Required"

{
    "error": "unauthorized",
    "message": "please authenticate",
    "status": 401
}
```

We will now need to provide the authentication header for our requests:

```
(venv) $ http --auth eric:secret GET http://172.16.1.123:5000/devices/
HTTP/1.0 200 OK
Content-Length: 192
Content-Type: application/json
Date: Wed, 09 Oct 2019 00:31:41 GMT
Server: Werkzeug/0.16.0 Python/3.6.8

{
    "device": [
        "http://172.16.1.123:5000/devices/1",
        "http://172.16.1.123:5000/devices/2",
        "http://172.16.1.123:5000/devices/3",
        "http://172.16.1.123:5000/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 an individual device or a group of devices. We also added basic security measures to ensure that only the users we created are able to 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 would 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 own RESTful API. By providing the abstraction, we are free to use what is required in the backend, such as Pexpect, while still providing a uniform frontend to our requester. We can even take a step forward and replace the underlying network device without impacting the users who are 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. An in-depth discussion of containers is beyond the scope of this book. 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 18.04 machines (`https://www.digitalocean.com/community/tutorials/how-to-build-and-deploy-a-flask-application-using-docker-on-ubuntu-18-04`). 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 19.03.2, build 6a30dfc
```

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 Netowrking!"
<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, as well as creating 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  49.15kB
Step 1/7 : FROM tiangolo/uwsgi-nginx-flask:python3.7-alpine3.7
python3.7-alpine3.7: Pulling from tiangolo/uwsgi-nginx-flask
48ecbb6b270e: Pulling fs layer
692f29ee68fa: 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

ac5384e6b007        docker.test         "/entrypoint.sh /sta…"   55
minutes ago      Up 46 minutes       443/tcp, 0.0.0.0:8000->80/tcp
docker.test
```

We can see the **management host IP** displayed in the address bar as follows:

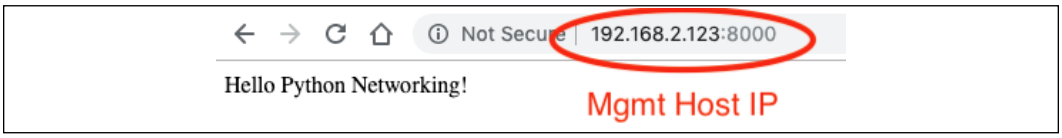


Figure 4: Management host IP

We can see the **Flask API endpoint** as follows:

Figure 5: Flask 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 provides us with even more flexibility and the option to deploy our own API abstraction in production. Containers, of course, offer their own 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 take a look at what we have done so far before looking forward to the next chapters.

Summary

In this chapter, we started to move onto the path of building RESTful APIs for our network. We looked at different popular Python web frameworks, namely Django and Flask, and compared and contrasted the two. By choosing Flask, we are able to 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 four nodes, two of which we have designated as spine routers while the other two are designated 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 as well as user authentication to our API. We also looked at how to run our Flask API application in a Docker container.

In *Chapter 10, AWS Cloud Networking*, we will shift gear to look at cloud networking using **Amazon Web Services** (**AWS**).

10
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 a person who has not learned about, used, or built services based in the cloud. Cloud computing is here, and we 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 Infrastructure-as-a-Service 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 2019. 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, **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 and simple; 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 we are just getting started is that they offer a number of 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 be using 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 instance 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:

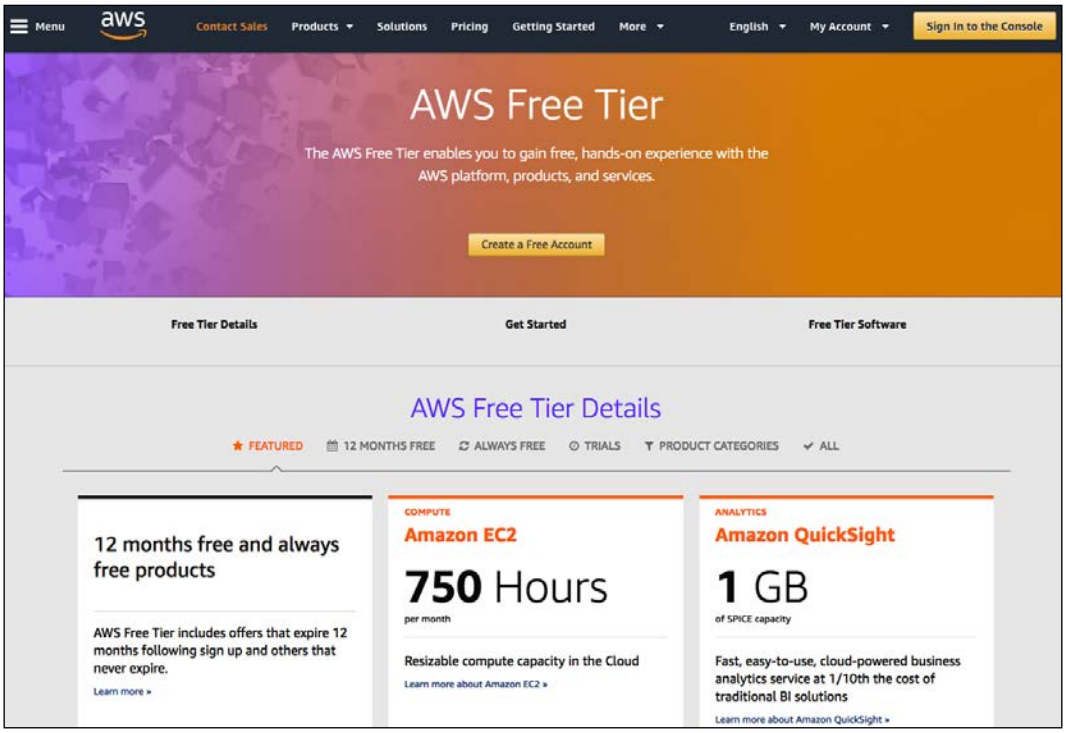


Figure 1: AWS free tier

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 console is where we can configure all the services and look at our monthly bills:

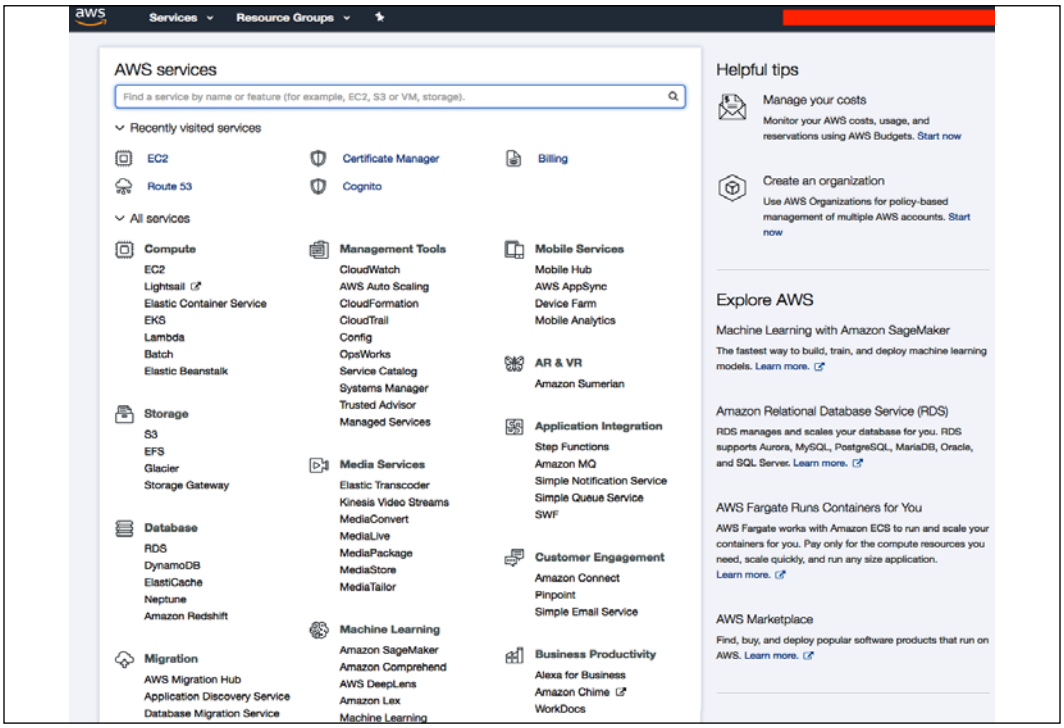


Figure 2: The AWS console

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:

```
(venv) $ pip install awscli
(venv) $ aws --version
aws-cli/1.16.259 Python/3.6.8 Linux/5.0.0-27-generic botocore/1.12.249
```

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:

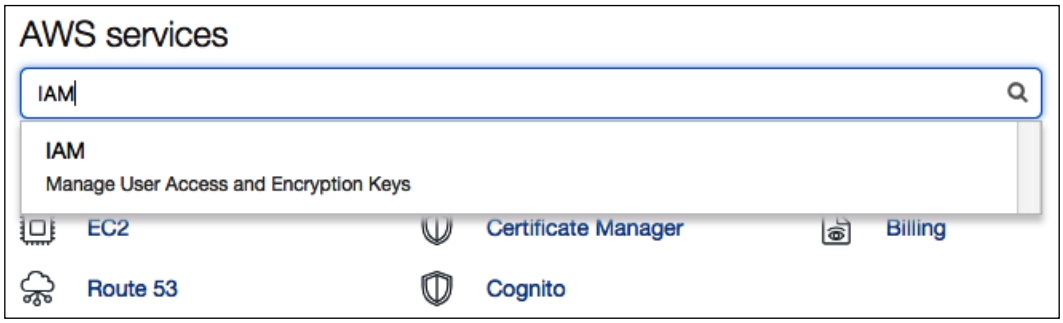


Figure 3: AWS IAM

We can choose **Users** on the left panel to create a user:

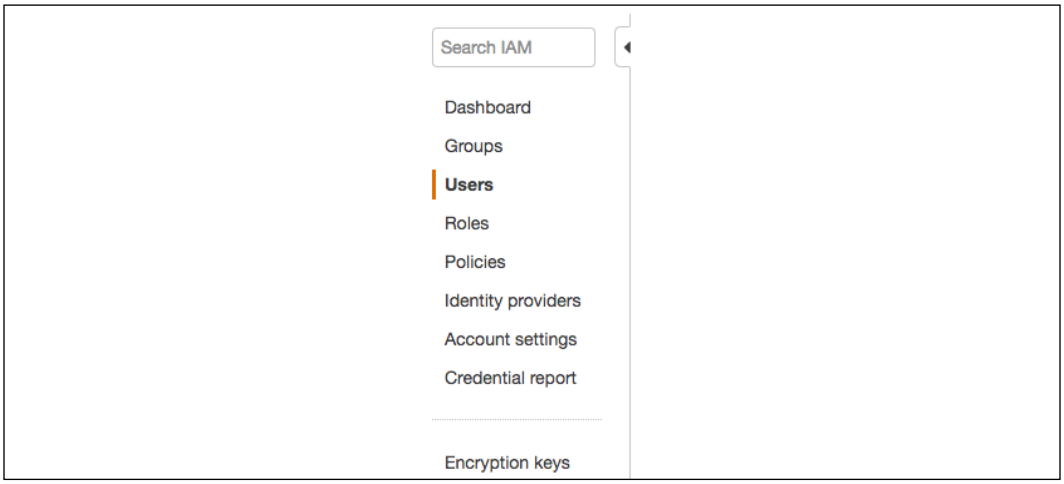


Figure 4: AWS IAM users

Select **Programmatic access** and assign the user to the default administrator group:

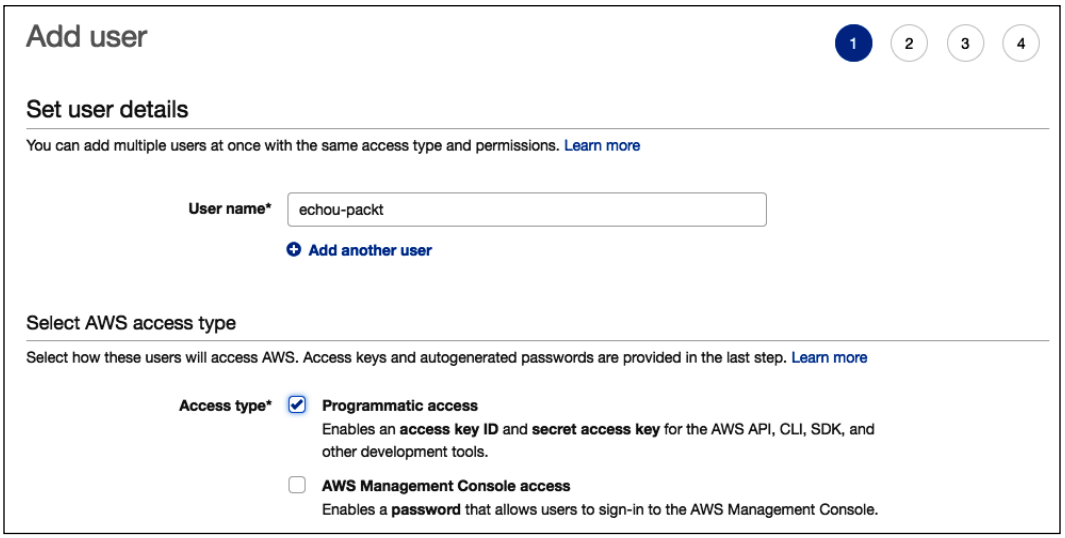


Figure 5: AWS IAM add user

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:

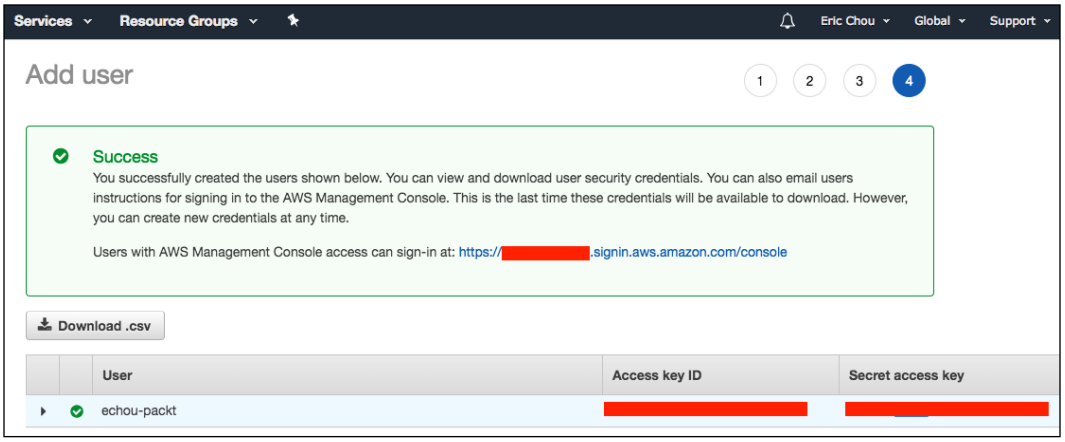


Figure 6: AWS IAM user security credentials

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
# verification
(venv) $ python
Python 3.6.8 (default, Oct 7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import boto3
>>> 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 listed 22 geographic regions and 69 **AZs** around the world. 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://www.infrastructure.aws/`.

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, and so on. 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:

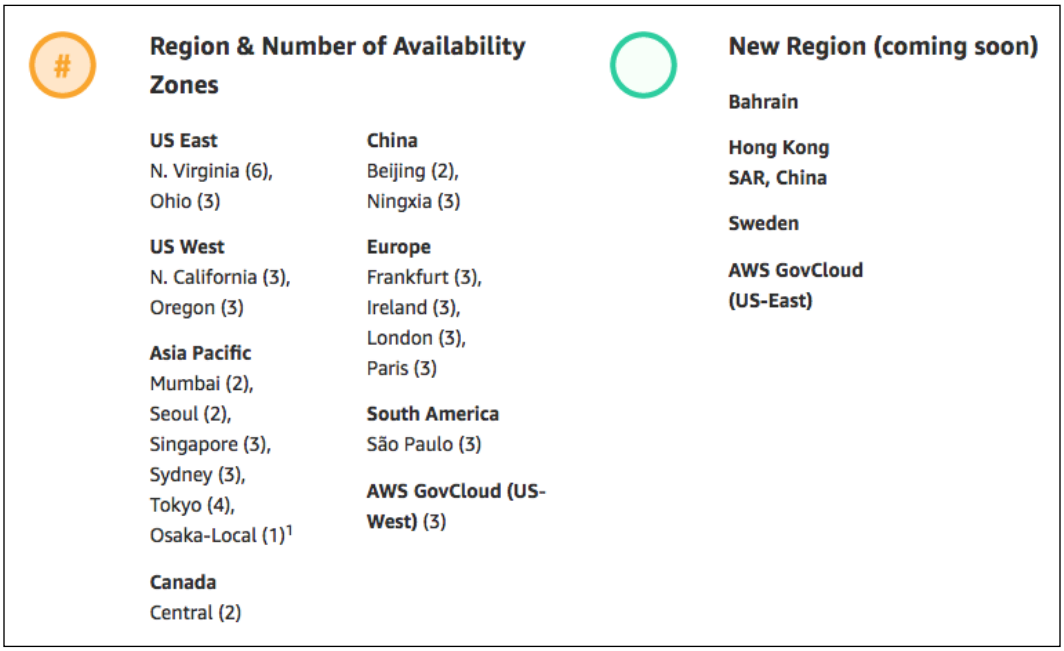
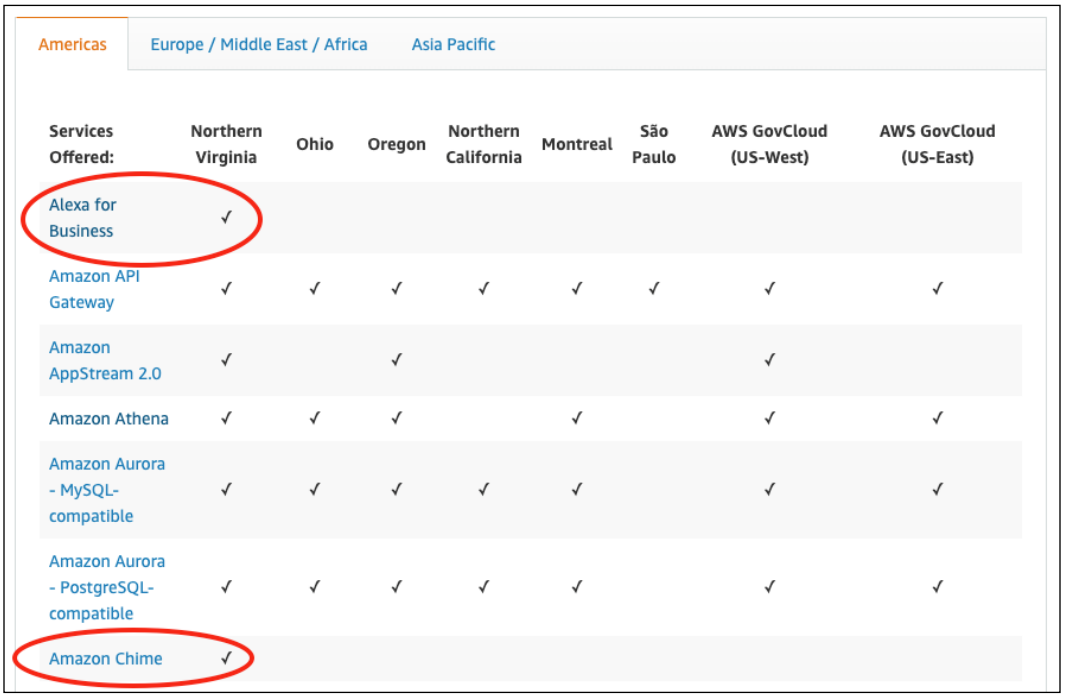


Figure 7: AWS Regions

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:

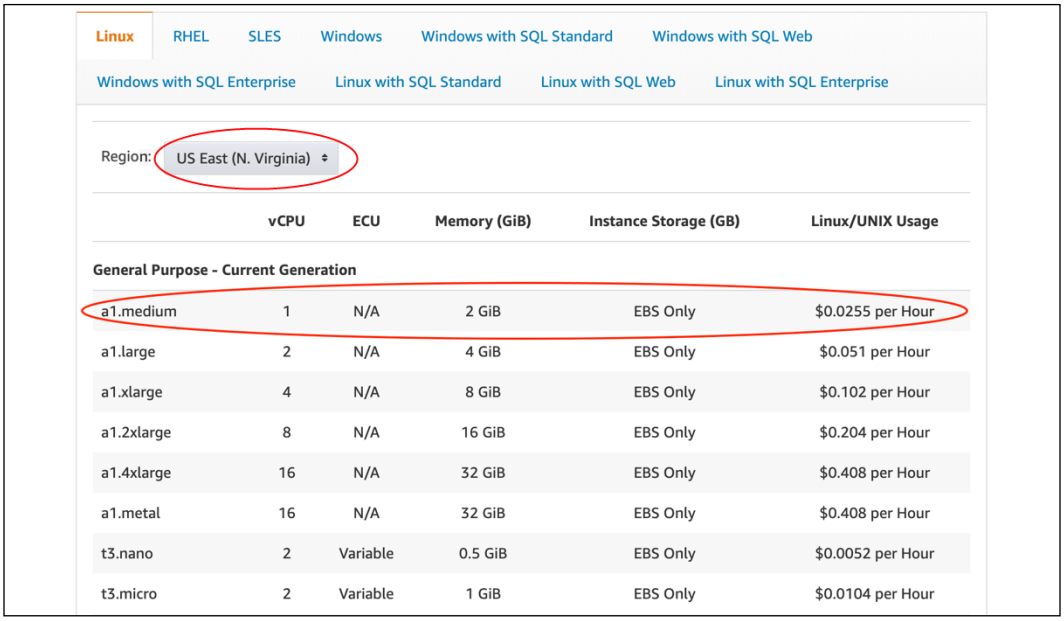


Americas | Europe / Middle East / Africa | Asia Pacific

Services Offered:	Northern Virginia	Ohio	Oregon	Northern California	Montreal	São Paulo	AWS GovCloud (US-West)	AWS GovCloud (US-East)
Alexa for Business	✓							
Amazon API Gateway	✓	✓	✓	✓	✓	✓	✓	✓
Amazon AppStream 2.0	✓		✓				✓	
Amazon Athena	✓	✓	✓		✓		✓	✓
Amazon Aurora - MySQL-compatible	✓	✓	✓	✓	✓		✓	✓
Amazon Aurora - PostgreSQL-compatible	✓	✓	✓	✓	✓		✓	✓
Amazon Chime	✓							

Figure 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)**:

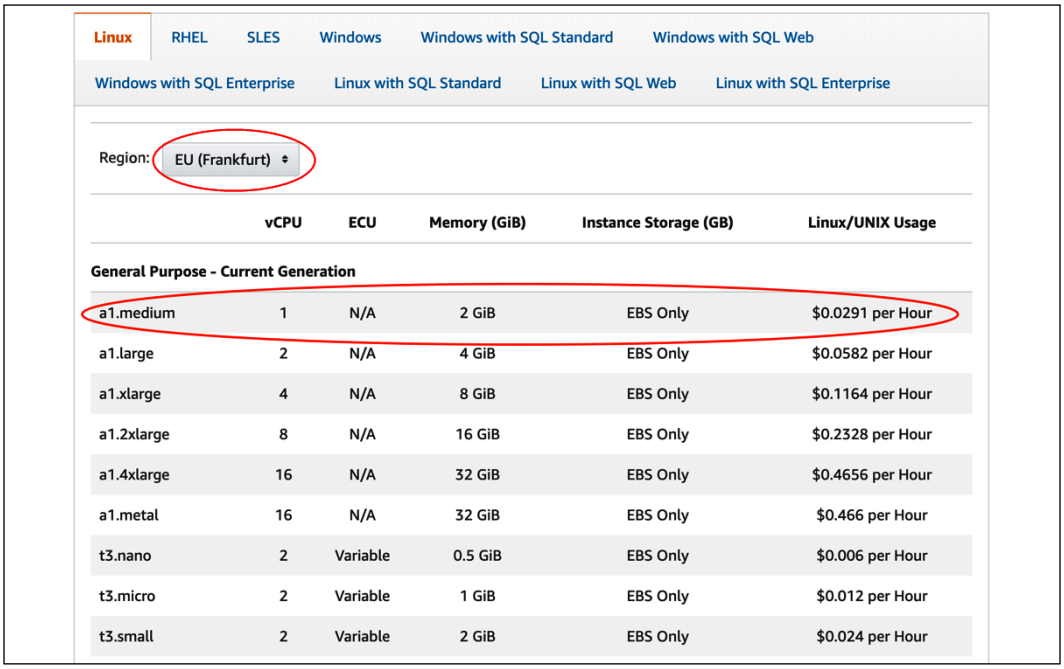


Linux | RHEL | SLES | Windows | Windows with SQL Standard | Windows with SQL Web | Windows with SQL Enterprise | Linux with SQL Standard | Linux with SQL Web | Linux with SQL Enterprise

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 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 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 if we have multiple Regions offering the same service, say **US-East** and **US-West**, and need the services to back up each other, 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:

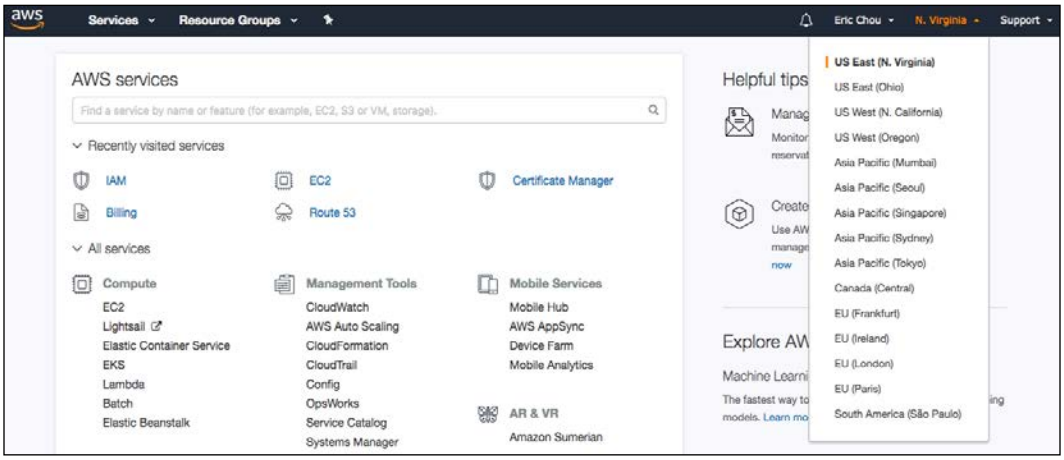


Figure 11: AWS Regions

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 we select the US West Region, none of our EC2 instances will show up. I have made this mistake a few times and wondered where all of my instances went!

The number after the Regions in the preceding **AWS Regions** screenshot represents the number of AZs in 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. Each AZ is in its own 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:

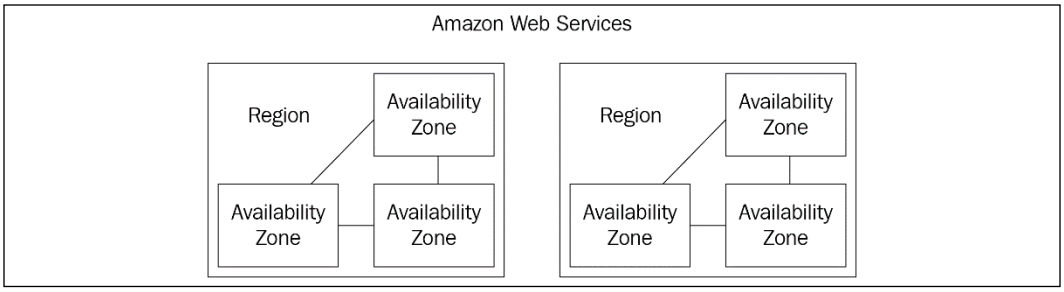


Figure 12: AWS Regions and availability zones

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 availability zone, `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, needs to 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:

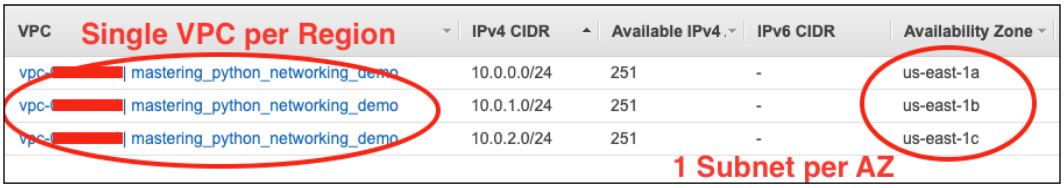


Figure 13: VPCs and AZs per Region

AWS edge locations are part of the **AWS CloudFront** content delivery network in 73 cities across 33 countries as of October 2019. 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 only, 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 (`https://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 that 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 who 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 take a look at the most import concept (in my opinion) in AWS networking: VPC.

Virtual private cloud

Amazon VPC 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 own 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 not available, 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**:

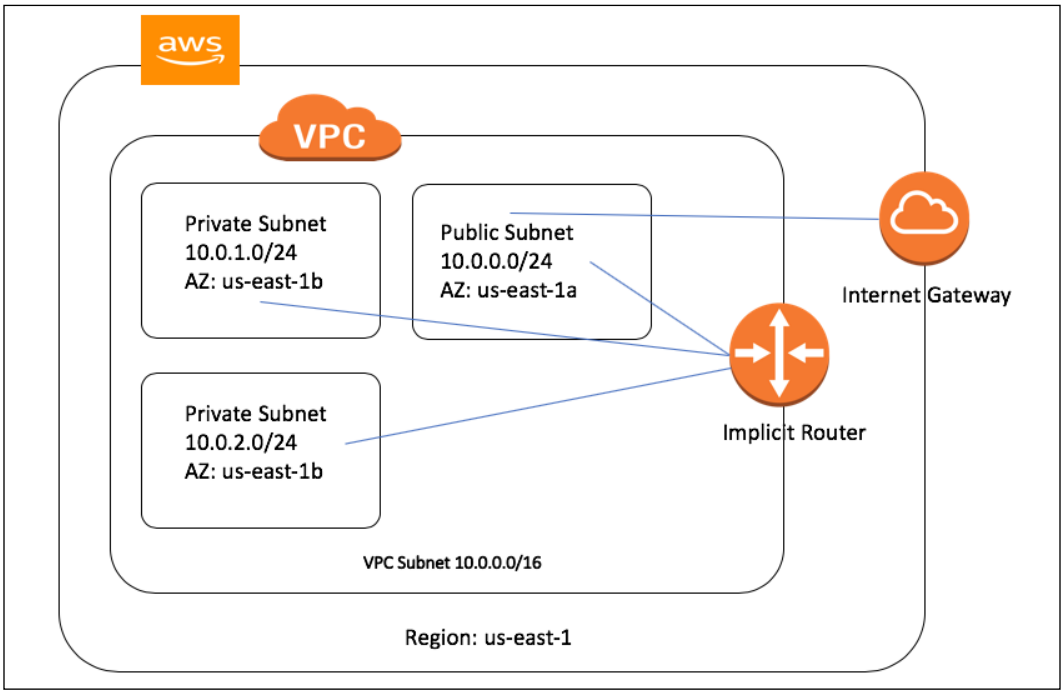


Figure 14: Our first VPC in US-East-1

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 a number of good tutorials online. I have listed the steps with the associated locations of each on the VPC dashboard:

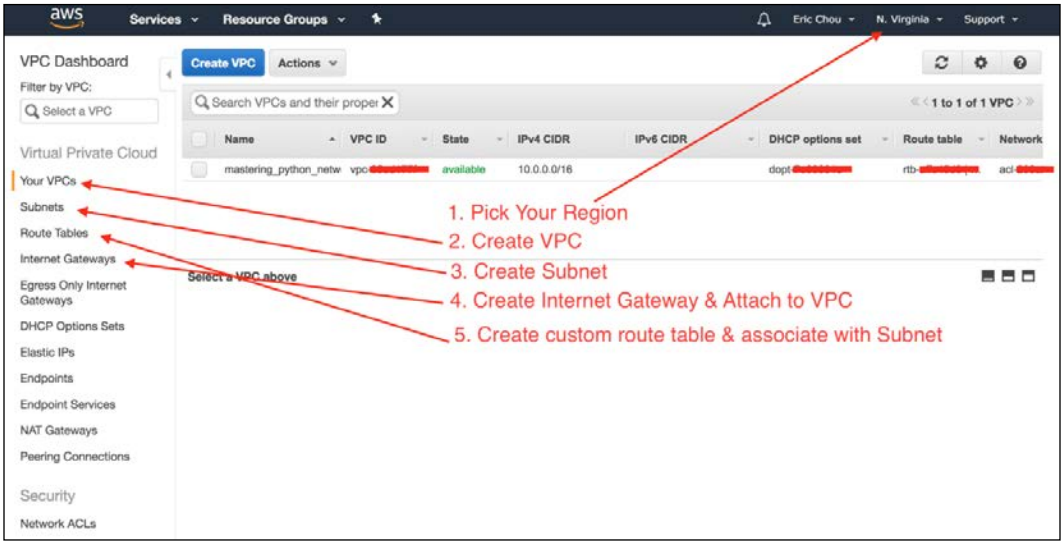


Figure 15: Steps for creating the VPC, subnet, and other features

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:

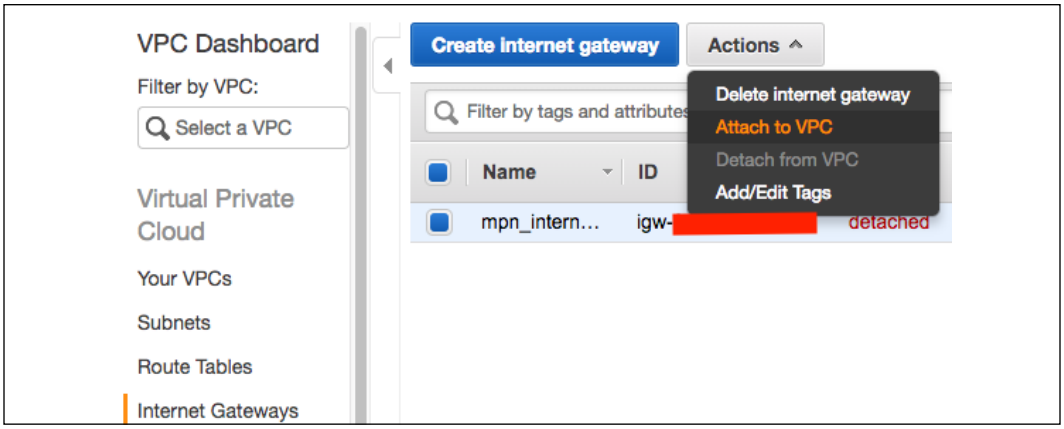


Figure 16: AWS internet gateway-to-VPC assignment

We can then create a custom route table with a default route pointing to the internet gateway, which will allow for 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:

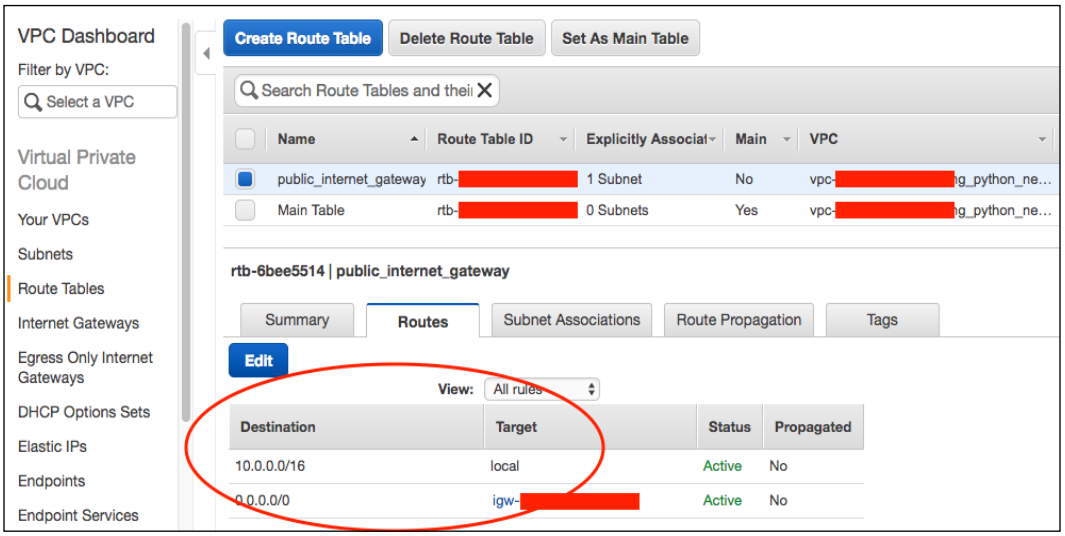


Figure 17: Route table

Let's use the Boto3 Python SDK to see what we have created; I used the tag `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 programmatically query the Region for the VPC we created:

```
(venv) $ python Chapter10_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 a 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 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:

```
#!/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 Chapter10_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 subnets that are private, `us-east-1b` and `us-east-1c`, following the same steps. The end 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 CloudFomation (`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:

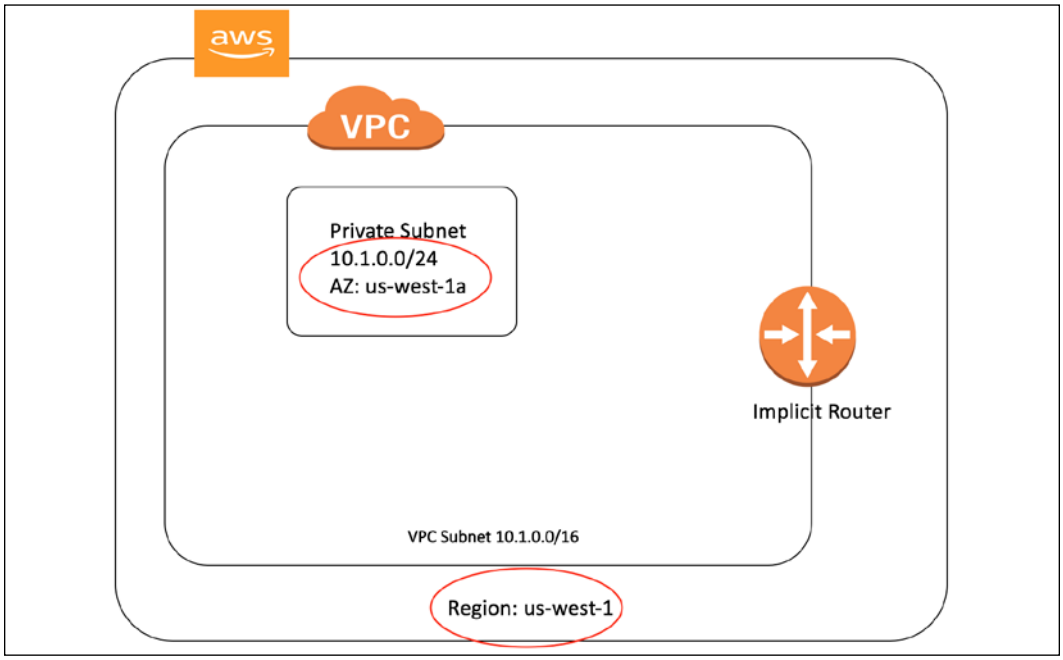


Figure 18: VPC for us-west-1

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://Chapter10_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, `Chapter10_4_cloud_formation_full.yml`, to create both the VPC as well as 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://Chapter10_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:

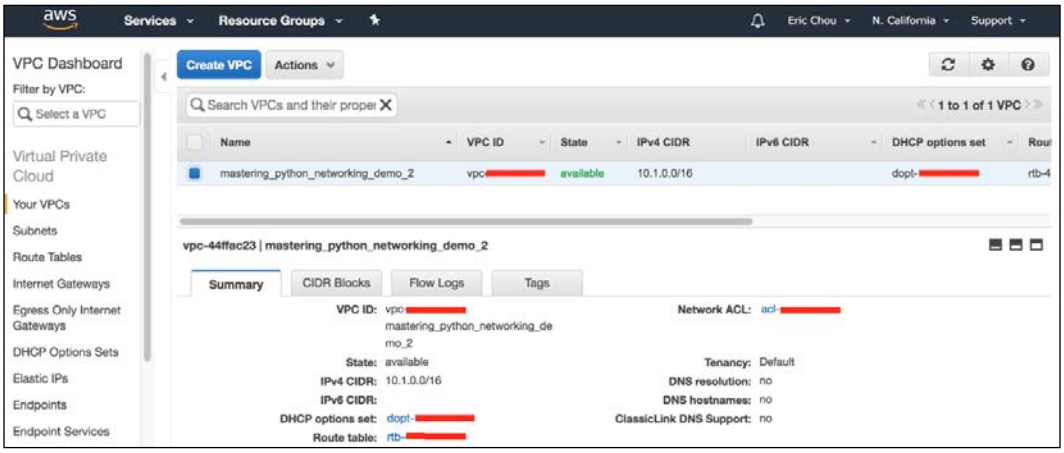


Figure 19: VPC in us-west-1

We can also take a look at the subnet:

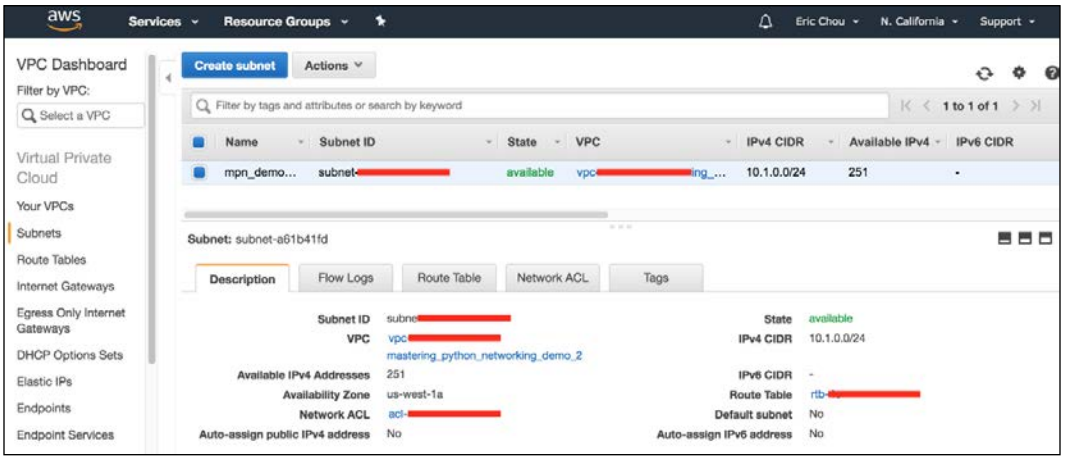


Figure 20: Subnet in us-west-1

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 you would like the two VPCs to be connected, you 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, DNS name) are taken care of.

In the coming section, we will take a look at VPC security groups and network access control lists.

Security groups and network ACLs

AWS **Security Groups** and **Network ACLs** can be found under the **Security** section of your VPC:

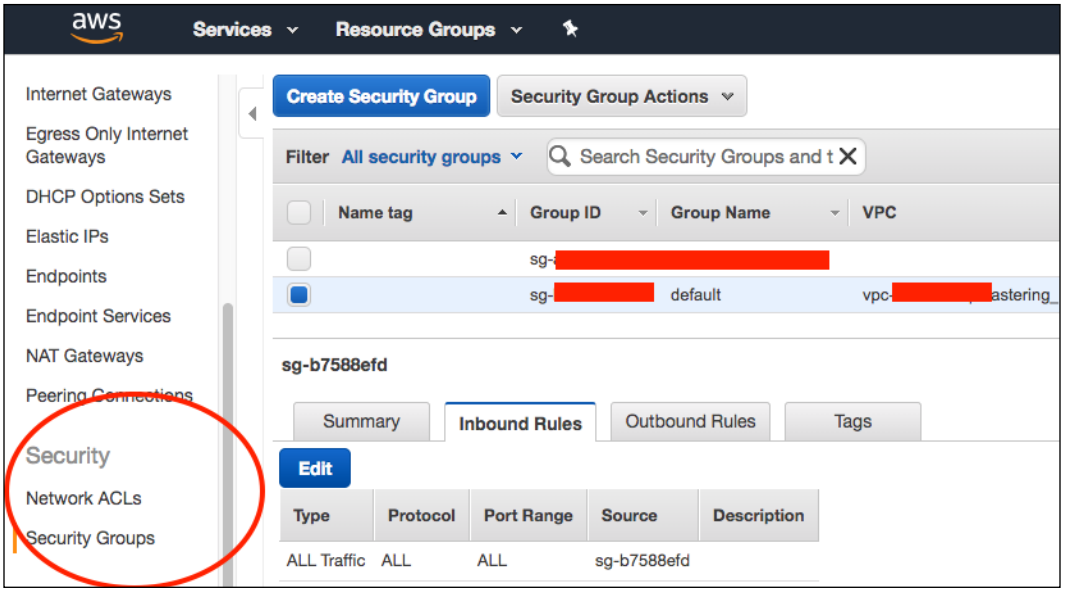


Figure 21: VPC security

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 as a way 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, `Chapter10_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 Chapter10_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 **access control lists** (**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 needs to be specifically allowed.

Let's take a 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 2019.

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 assignment, you can associate it with the desired resources:

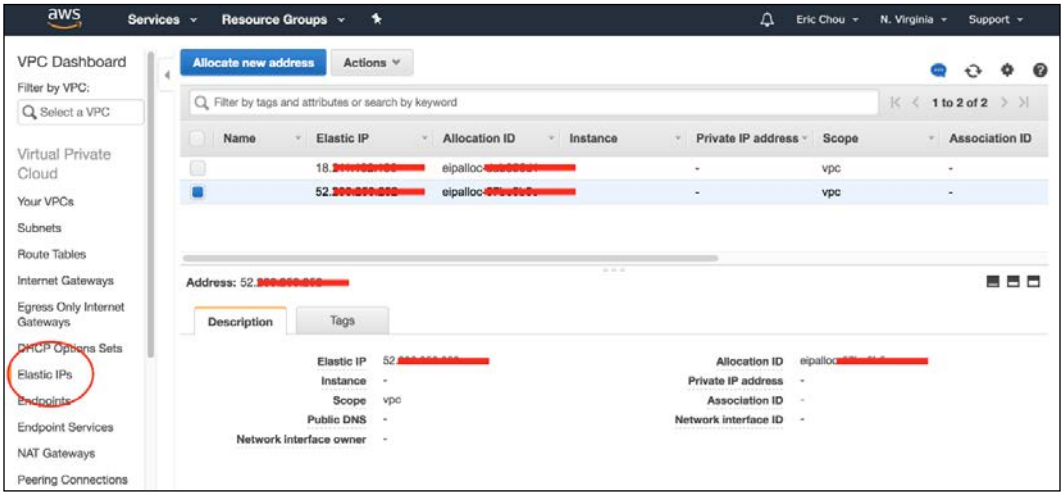


Figure 22: Elastic IP

Unfortunately, EIPs are limited by default 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 coming 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 this book, 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.

This is where 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 an NAT gateway, we can perform the following steps:

1. Create an 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:

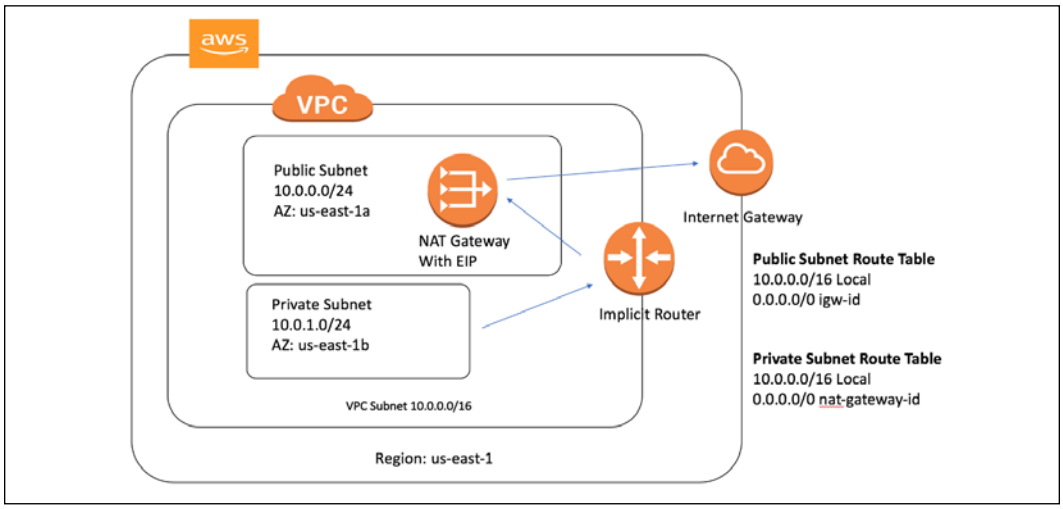


Figure 23: NAT gateway operations

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:

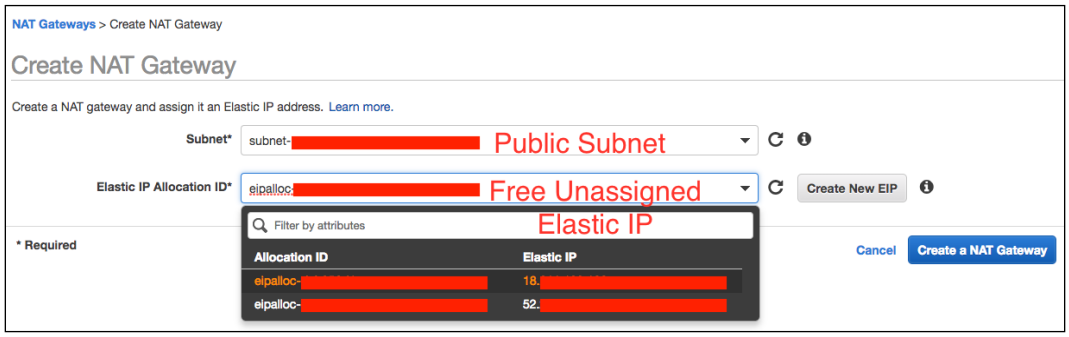


Figure 24: NAT gateway creation

In the coming section, we will take a 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 of 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 two ways in which 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-premise network to VPC is with traditional IPSec VPN connections. We will need a publicly accessible device that can 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 over. Currently, AWS recommends using 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:

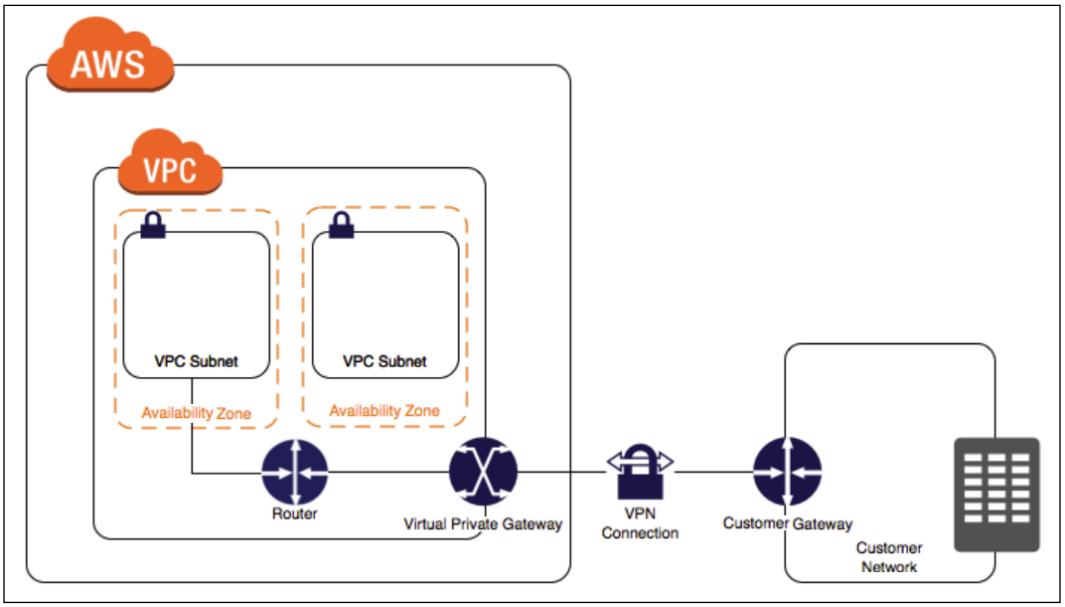


Figure 25: VPC VPN connection (source: https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_VPN.html)

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-premise 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 the reliability of it. 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 allows customers to 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 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:

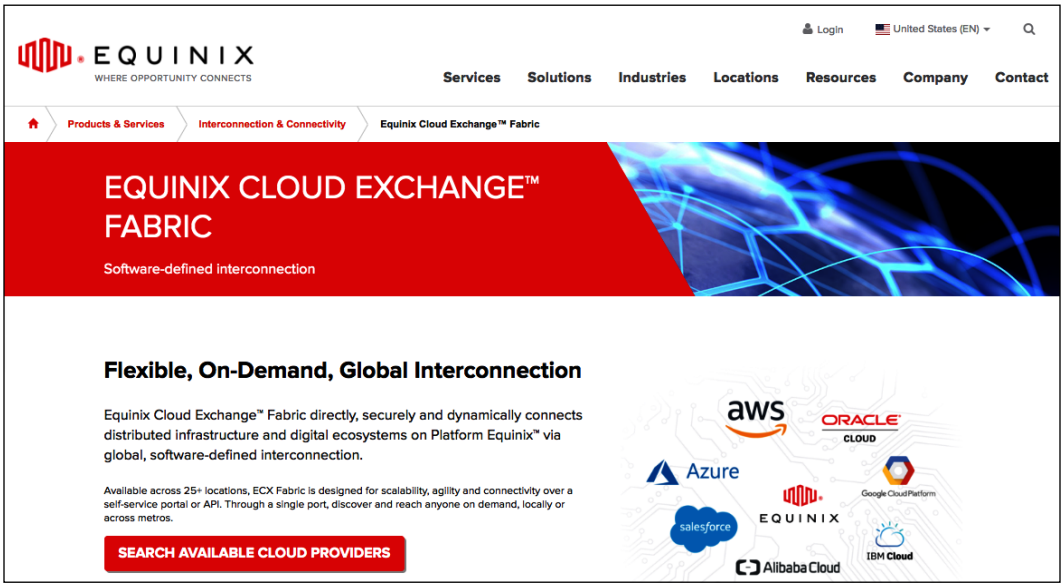


Figure 26: Equinix Cloud Exchange (source: https://www.equinix.com/services/interconnection-connectivity/cloud-exchange/)

In the upcoming section, we will take a look at some of the network scaling services AWS offers.

Network scaling services

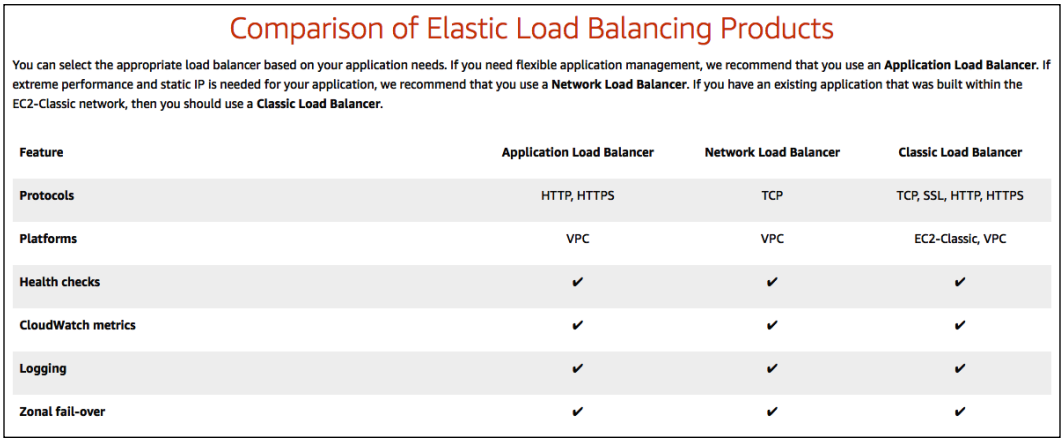
In this section, we will take a look at some of the network services AWS offers. Many of these services do not have direct network implications, such as DNS and content distribution networks. They are relevant in our discussion due to their close relationship with the network and application's performance.

Elastic Load Balancing

Elastic Load Balancing (**ELB**) allows incoming traffic from the internet to be automatically distributed across multiple EC2 instances. Just 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 Application Load Balancer handles web traffic via **HTTP** and **HTTPS**; the Network 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 balancer 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 needs. 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**.

Feature	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 fail-over	✔	✔	✔

Figure 27: ELB comparison (source: https://aws.amazon.com/elasticloadbalancing/details/)

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 referred to as 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 turn our attention to 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 but are also in many other locations around the globe. The high-level operation of CloudFront is as follows:

- Users access your website for one or more objects
- DNS routes the request to the Amazon CloudFront edge location closest to the user's request
- 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 important ones 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 went over the AWS network definitions of Region, Availability Zone, edge locations, and Transit Center. By understanding the overall AWS network, this 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, as well as CloudFormation to automate some of the tasks.

We covered AWS Virtual Private Cloud 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 in relation to 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 Elastic Load Balancing, Route 53 DNS, and CloudFront.

In the next chapter, we will take a look at the networking services offered by another public cloud provider, Microsoft Azure.

11
Azure Cloud Networking

As we saw in *Chapter 10, 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, with a focus 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 it was renamed to "Microsoft Azure" in 2014. Since AWS released its first product, S3, in 2006, they 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 own 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 when it comes to 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 to have 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, making comparisons between 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 in that 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 get started 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 Services 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 own infrastructure in the Azure cloud. As the undisputed leader in the space, much of the target audience has already had 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 as 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 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, there are also differences that 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 have learned about 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 big promotions on Artificial Intelligence and Kubernetes services with many amazing products being always free or free for the first twelve months:

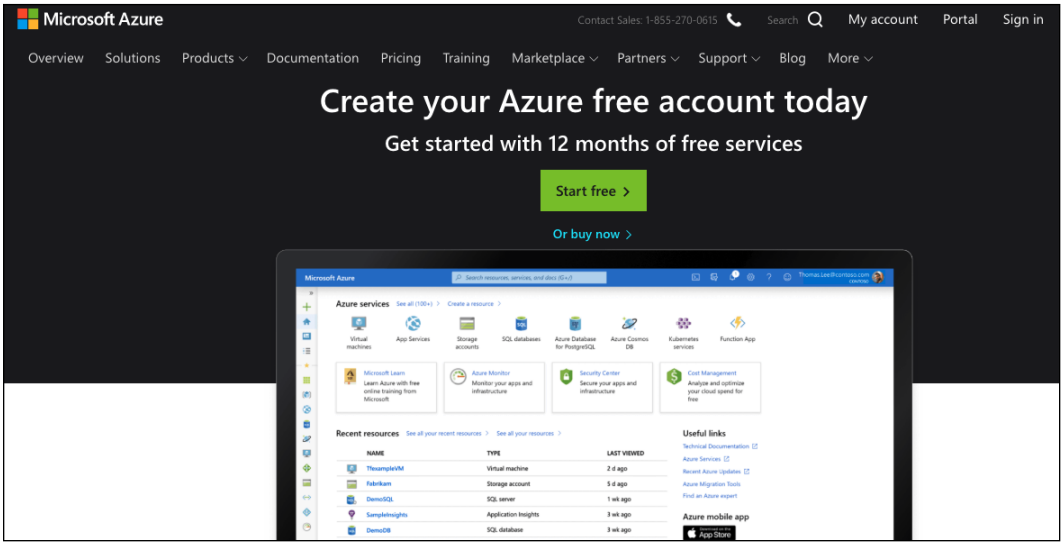


Figure 2: Azure portal (source: https://azure.microsoft.com/en-us/free/)

After the account is created, we can see the services available on the Azure portal at `https://portal.azure.com`:

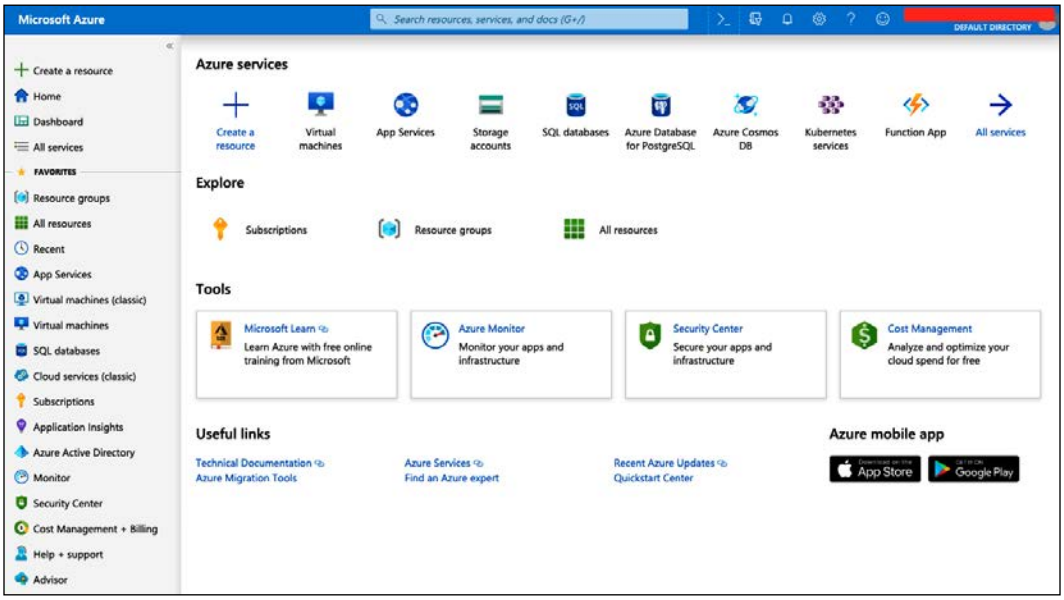


Figure 3: Azure services

Before any service can actually be launched, however, we will need to provide a payment method. This is done by adding a subscription service:

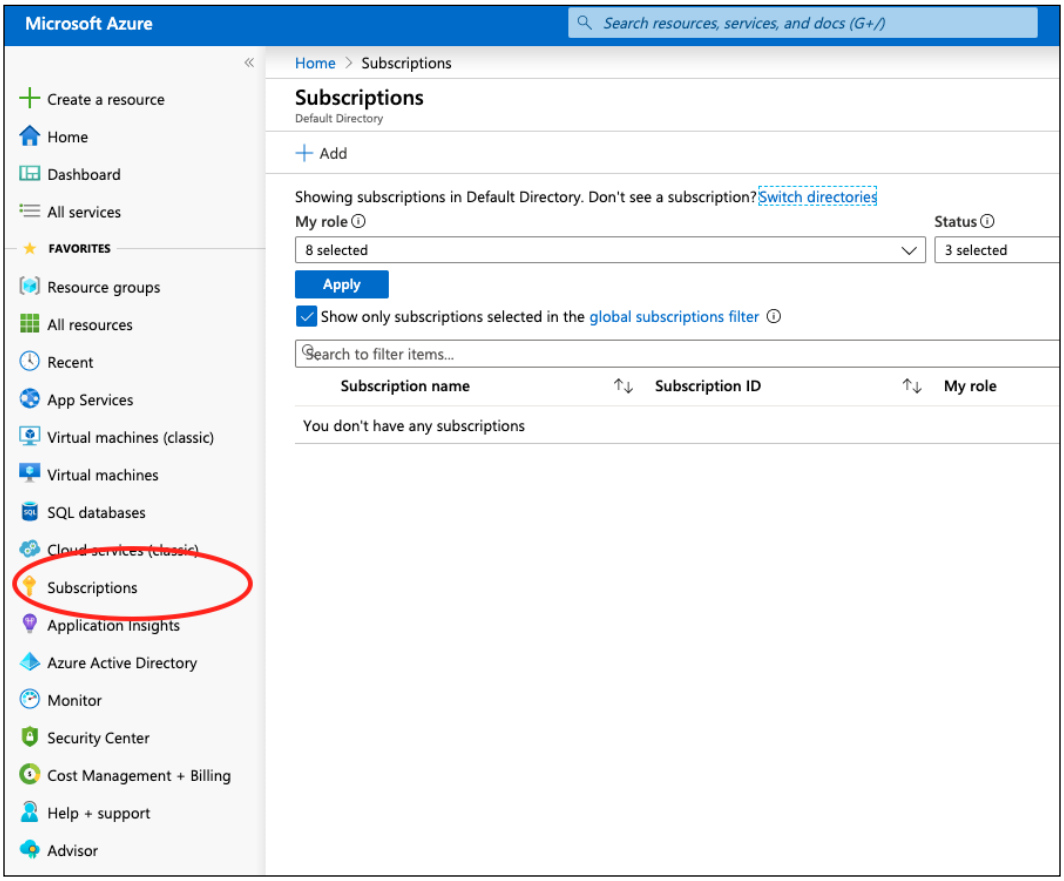


Figure 4: Azure subscriptions

I would recommend adding a Pay-As-You-Go plan, which has no up-front costs and no long-term commitment, but you 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 most sleek and modern looking 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 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:

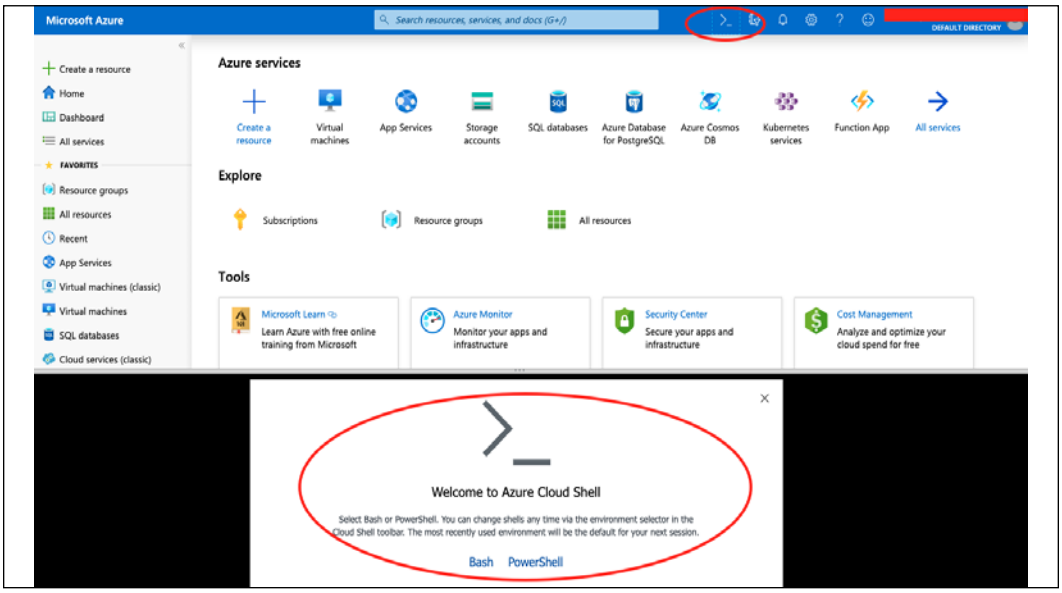


Figure 6: Azure Cloud Shell

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 be run simultaneously:

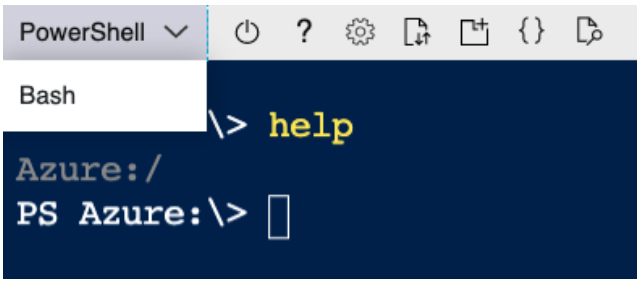


Figure 7: Azure Cloud Shell with PowerShell

My personal preference is the **Bash** shell, which allows me to use the pre-installed Azure CLI and Python SDK:

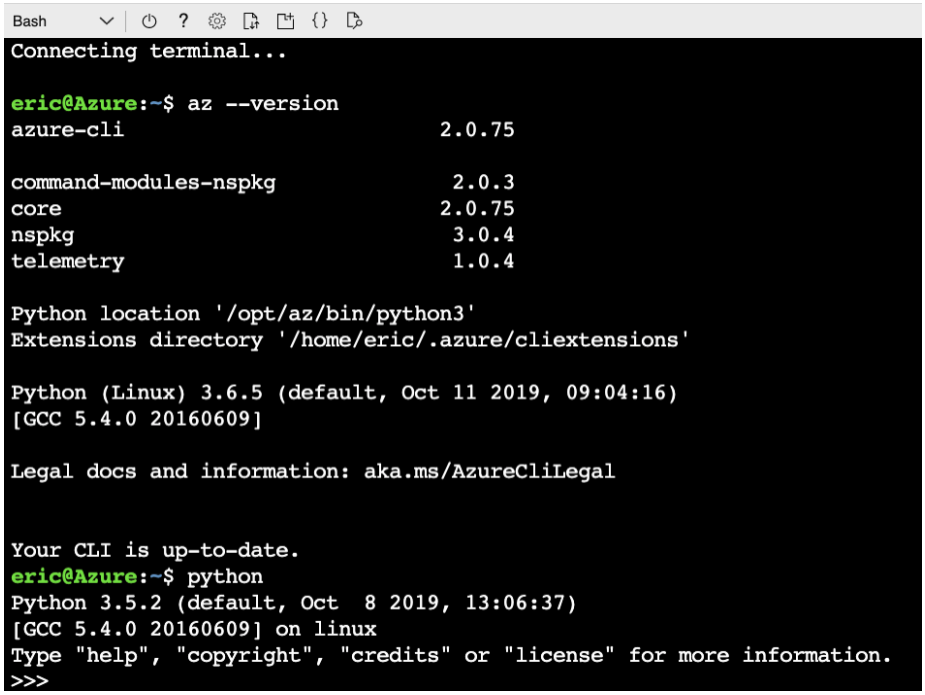


Figure 8: Azure AZ tool

The 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                         2.0.75

command-modules-nspkg              2.0.3
core                              2.0.75
nspkg                              3.0.4
telemetry                          1.0.4
```

Let's also install the Azure Python SDK on our management host:

```
(venv) $ pip install azure
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import azure
>>> exit()
```

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.

We are now ready to take a look at some of the service principles of Azure and launch our own Azure services.

Azure service principal

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. 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
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:

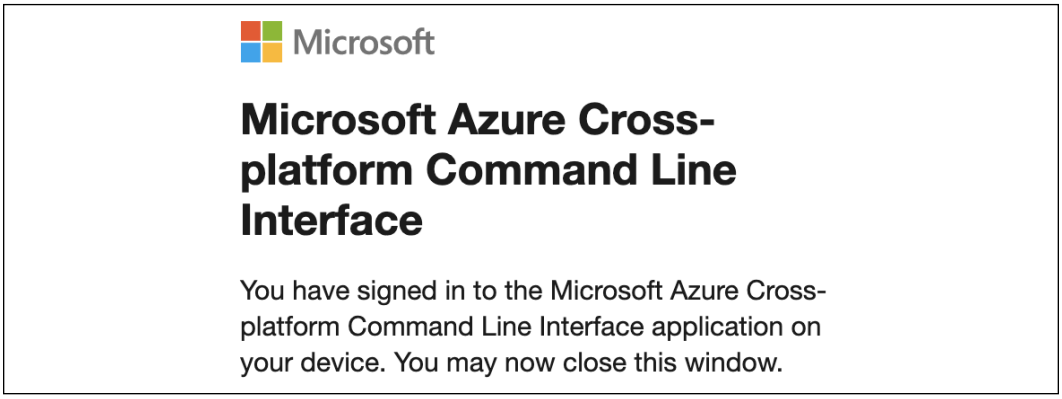


Figure 9: Azure Cross-platform Command Line Interface

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
```

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:

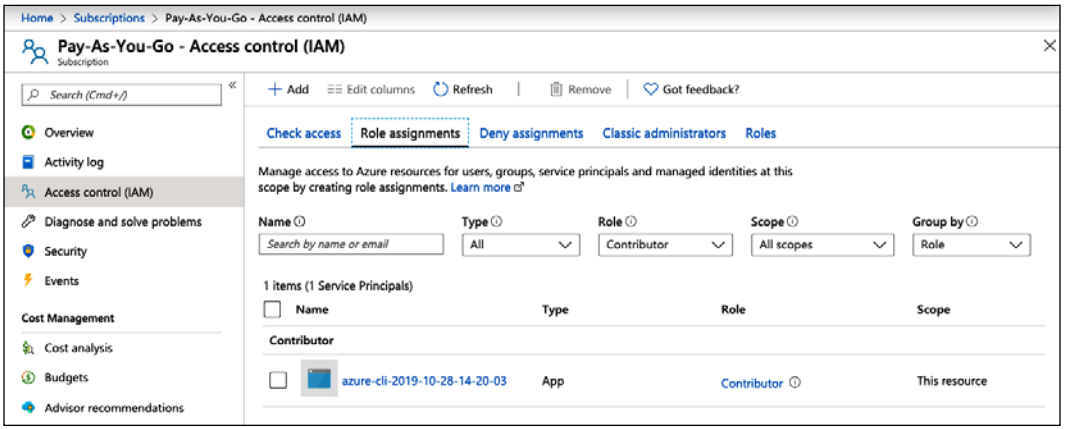


Figure 10: Azure Pay-As-You-Go IAM

We will use a simple Python script, `Chapter11_1_auth.py`, to import the library for client authentication and network management:

```
from azure.common.client_factory import get_client_from_auth_file
from azure.mgmt.network import NetworkManagementClient

client = get_client_from_auth_file(NetworkManagementClient)
print("Network Management Client API Version: " + client.DEFAULT_API_
VERSION)
```

If the file executes without an error, we have successfully authenticated with the Python SDK client:

```
(venv) $ python Chapter11_1_auth.py
Network Management Client API Version: 2018-12-01
```

While reading the Azure documentation, you may have noticed the examples are primarily given in PowerShell code. 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 when it comes to Azure. In much of the Azure documentation, you will find direct references to PowerShell examples. There are often opinionated discussions around the web forums on which tool, Python or PowerShell, is better suited to managing Azure resources.

As of July 2019, we can run 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`. However, given the beta nature of the release, it is expected to have some bugs and feature gaps.

We will not get into a debate on language superiority. I do not mind using PowerShell when required – personally, I find it easy and intuitive – and I agree that sometimes the Python SDK lags behind PowerShell on 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.

In the beginning, 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 54 regions and more than 150 edge node locations, as illustrated on the product page (`https://azure.microsoft.com/en-us/global-infrastructure/`):

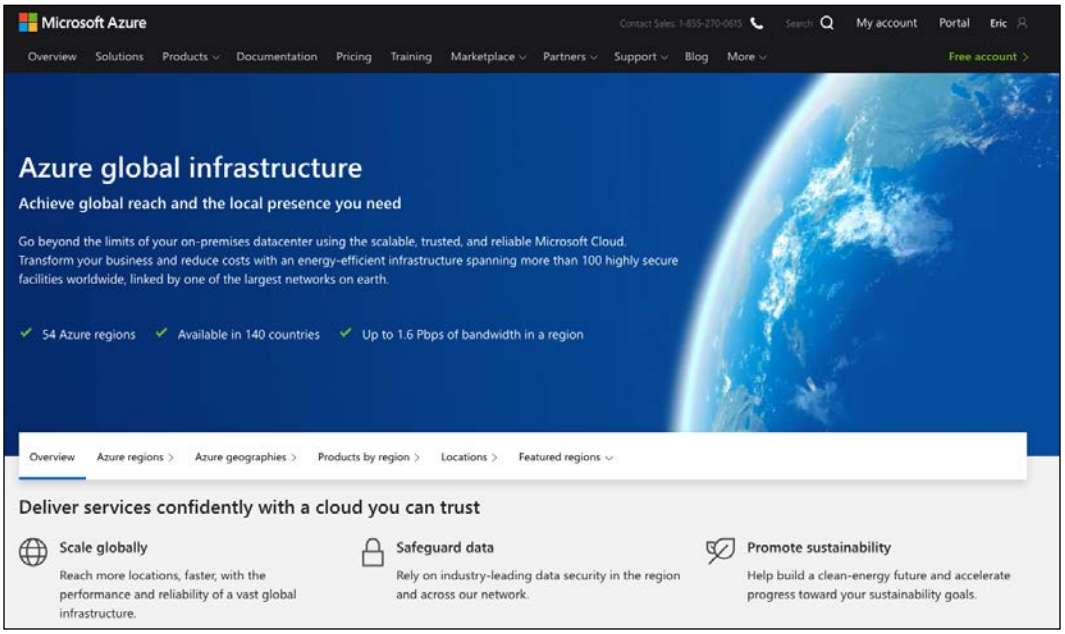


Figure 11: Azure global infrastructure (source: https://azure.microsoft.com/en-us/global-infrastructure/)

Like AWS, Azure products are offered via regions, so we will 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 definitely something to be aware of when picking our region. I would recommend picking regions with AZs such as West US 2, Central US, and East US 1.

If we need to build in a region without Availability Zones, 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 two 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 will spend most of our time. Similar to a traditional network 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 virtual machines to communicate with each other, with the internet, and with our on-premises network through a VPN or ExpressRoute.

Let's begin by building our first VNet using the portal. We will start by browsing to the **virtual network page** via **Create a Resource -> Networking -> Virtual network**:

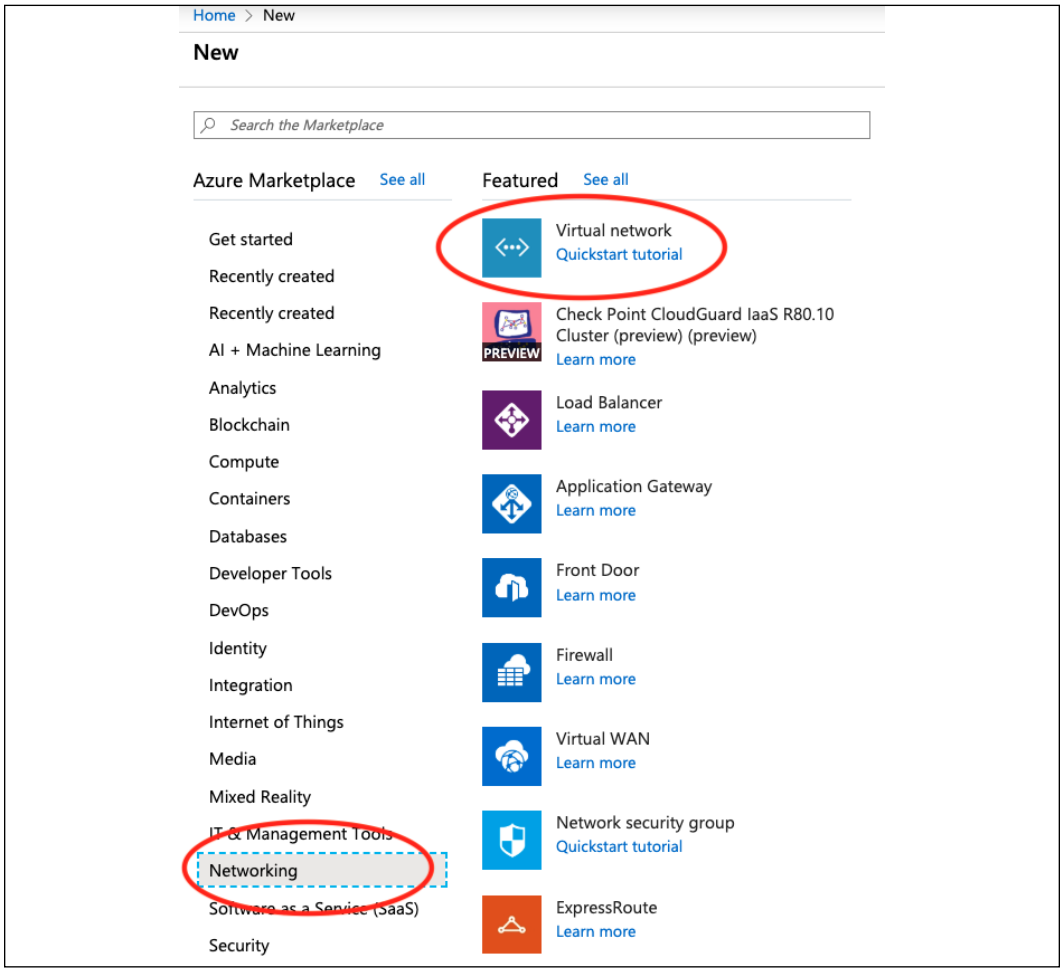


Figure 12: Azure VNet

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:

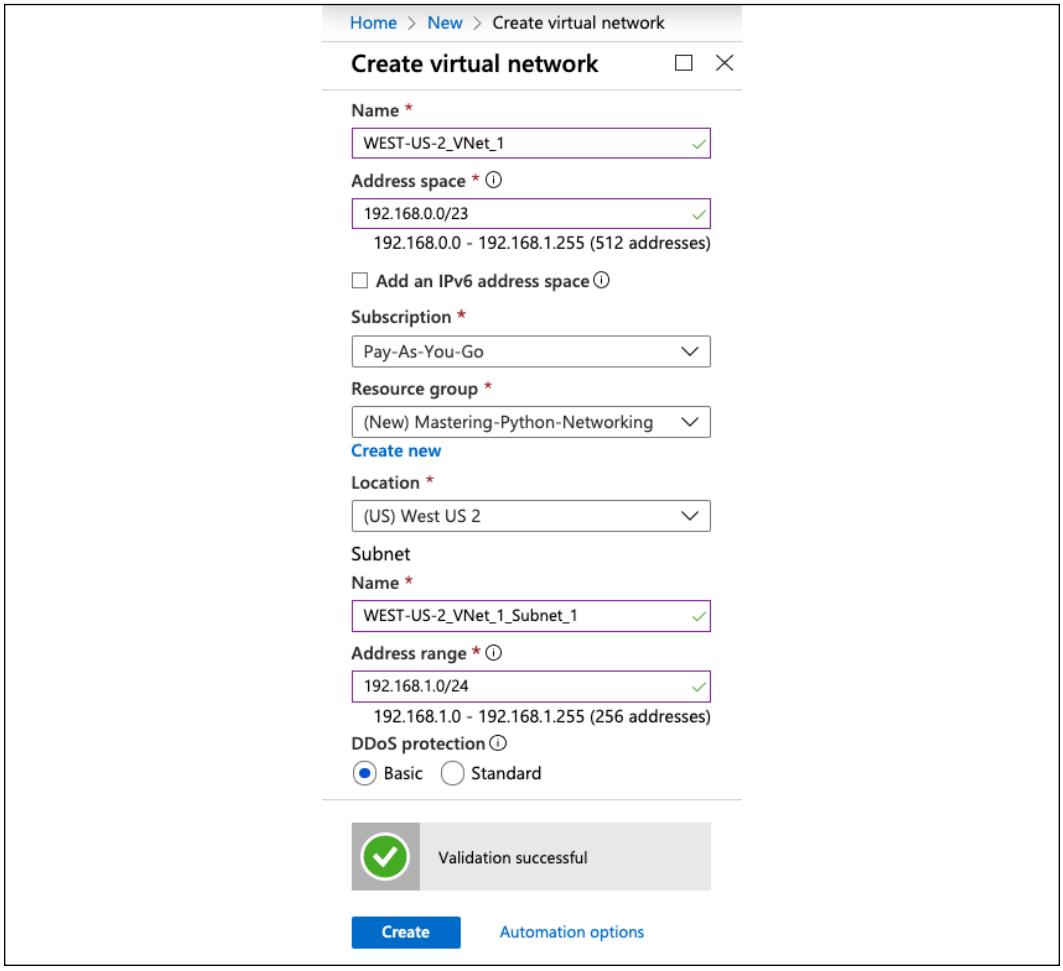


Figure 13: Azure VNet creation

Once the resource is created, we can navigate to it via **Home -> Resource groups -> Mastering-Python-Networking**:

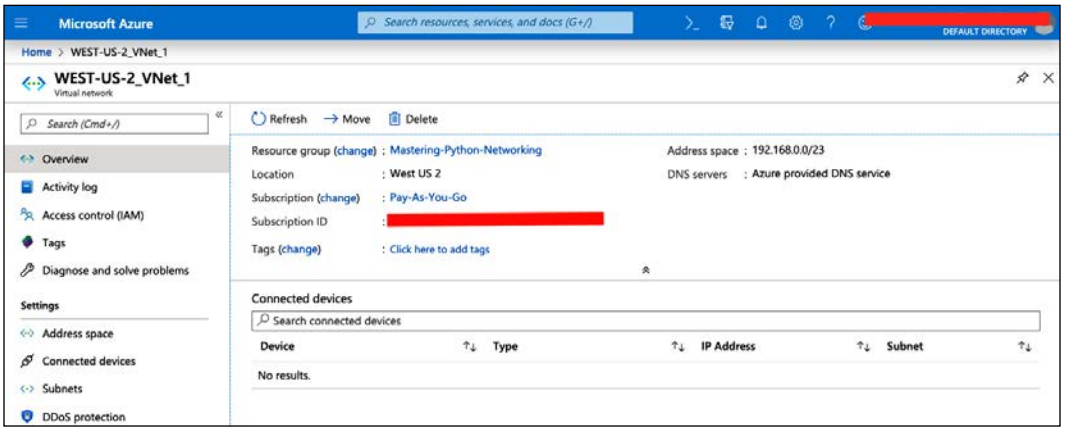


Figure 14: Azure VNet overview

Congratulations, we just created our first VNet in the Azure cloud! Of course, our network needs to communicate with the outside world to be useful. We will take a 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 virtual machines within our network.

We can create our first virtual machine from **Home -> Resource groups -> Mastering-Python-Networking -> New -> Create a virtual machine**:

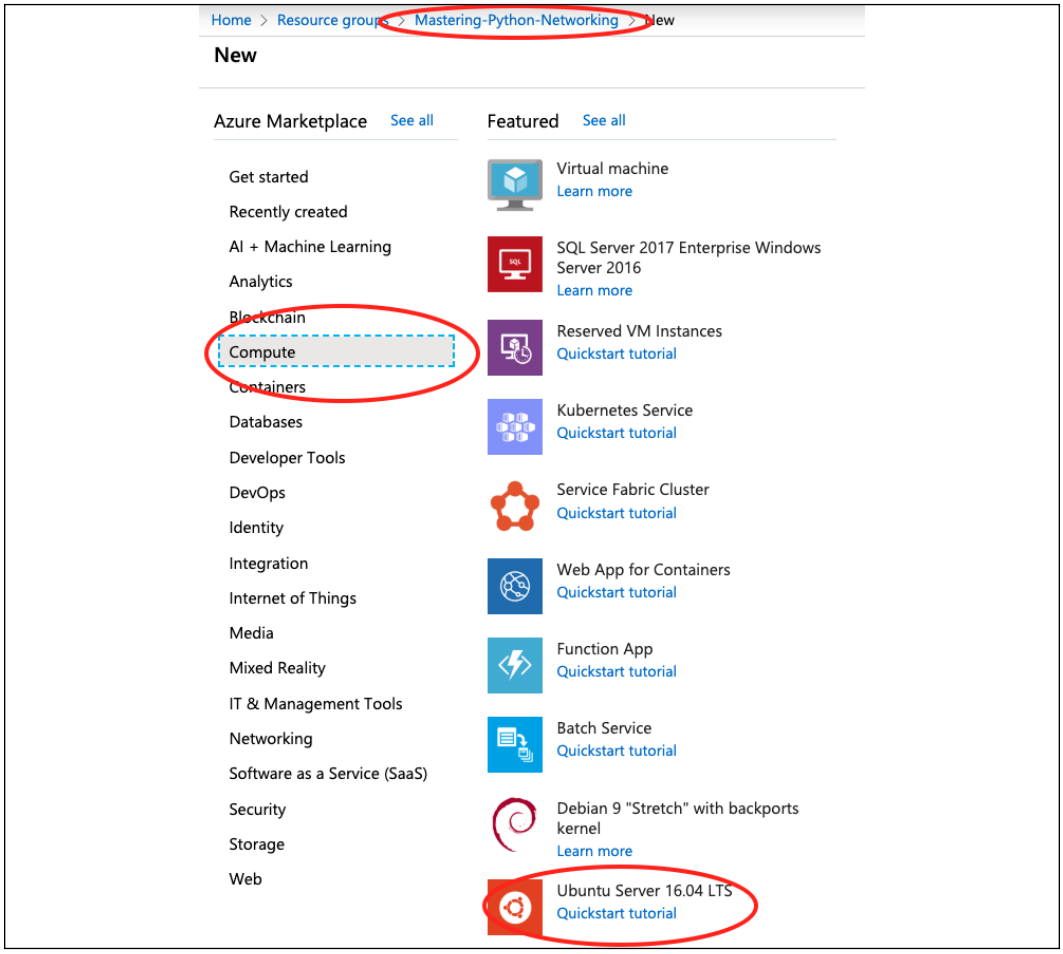


Figure 15: Azure create VM

I will pick **Ubuntu Server 16.04 LTS** as the virtual machine, and use the name `myMPN-VM1` when prompted. I will pick the region `West US 2`, as well as choosing a password as the authentication method and allowing an SSH inbound connection.

We can leave the other options as their default settings. We will put the VM into the subnet that we created, as well as assigning a new public IP:

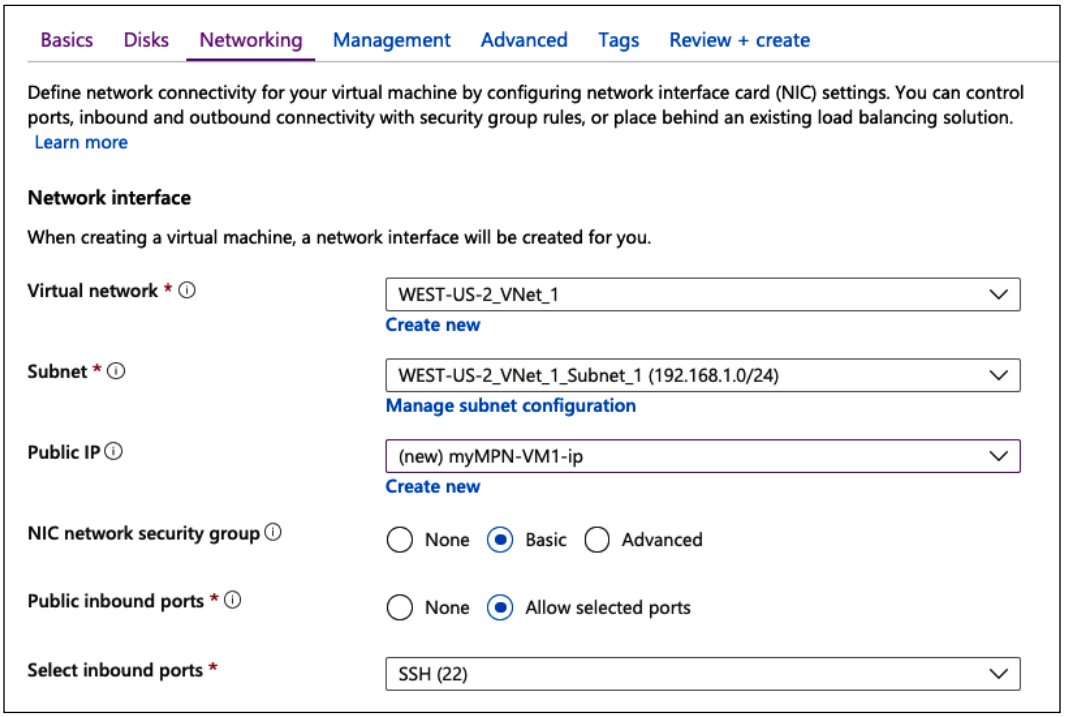


Figure 16: Azure network interface

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:~$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0d:3a:6e:14:f3
          inet addr:192.168.1.4  Bcast:192.168.1.255  Mask:255.255.255.0
<skip>
echou@myMPN-VM1:~$ ping -c 1 www.google.com
PING www.google.com (172.217.14.228) 56(84) bytes of data.
64 bytes from sea30s02-in-f4.1e100.net (172.217.14.228): icmp_seq=1
ttl=51 time=4.88 ms


--- www.google.com ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.888/4.888/4.888/0.000 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:

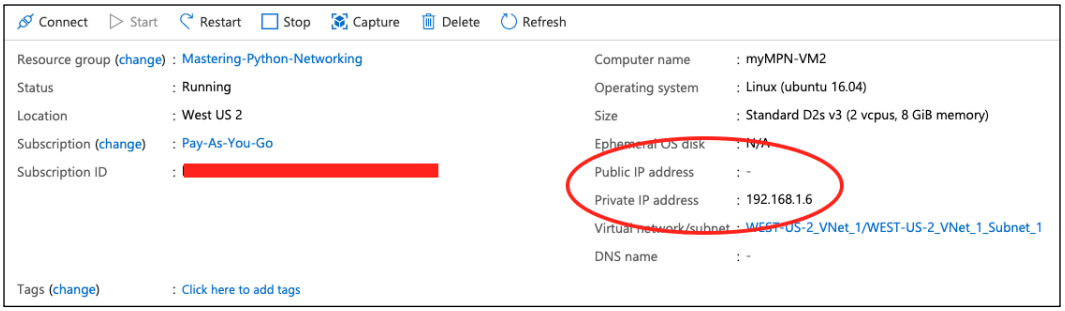


Figure 17: Azure VM IP addresses

After the VM creation, we can `ssh` to `myMPN-VM2` from `myMPN-VM1` with the private IP:

```
echou@myMPN-VM1:~$ ssh echou@192.168.1.6
echou@192.168.1.6's password:
<skip>
0 updates are security updates.Last login: Tue Oct 29 01:05:44 2019 from
192.168.1.4
echou@myMPN-VM2:~$
```

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 xenial InRelease
Get:2 http://azure.archive.ubuntu.com/ubuntu xenial-updates InRelease
[109 kB]
Get:3 http://azure.archive.ubuntu.com/ubuntu xenial-backports InRelease
[107 kB]
Hit:4 http://security.ubuntu.com/ubuntu xenial-security InRelease
Fetched 216 kB in 0s (720 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, `Chapter11_2_network_resources.py`, we will use the `NetworkManagementClient` class (`https://docs.microsoft.com/en-us/python/api/azure-mgmt-network/azure.mgmt.network.networkmanagementclient?view=azure-python`) we defined in the previous example and use the `subnet.create_or_update` API to create a new `192.168.0.128/25` subnet in the VNet:

```
GROUP_NAME = 'Mastering-Python-Networking'
LOCATION = 'westus2'

def create_subnet(network_client):
    subnet_params = {
        'address_prefix': '192.168.0.128/25'
    }
    creation_result = network_client.subnets.create_or_update(
        GROUP_NAME,
        'WEST-US-2_VNet_1',
        'WEST-US-2_VNet_1_Subnet_2',
        subnet_params
    )

    return creation_result.result()

creation_result = create_subnet(network_client)
```

We will receive the following creation result message when we execute the script:

```
(venv) $ python3 Chapter11_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:

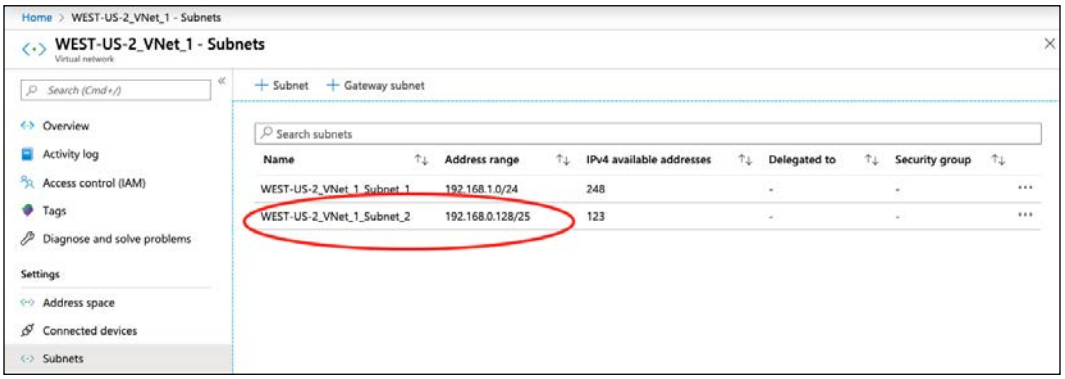


Figure 18: Azure VNet subnets

For more examples on using the Python SDK, check out `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/python` for a Windows VM example and `https://github.com/Azure-Samples/virtual-machines-python-manage/blob/master/example.py` for a Linux Ubuntu example.

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 that 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 endpoint

VNet service endpoints can extend the VNet to other Azure services over a direct connection. This allows traffic from the VNet to the particular 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:

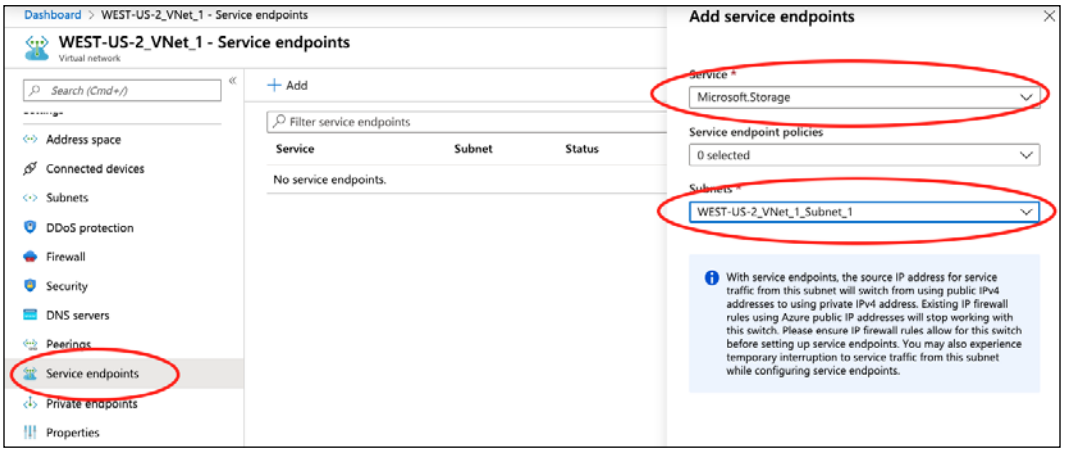


Figure 19: Azure service endpoints

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 through 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 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 `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:

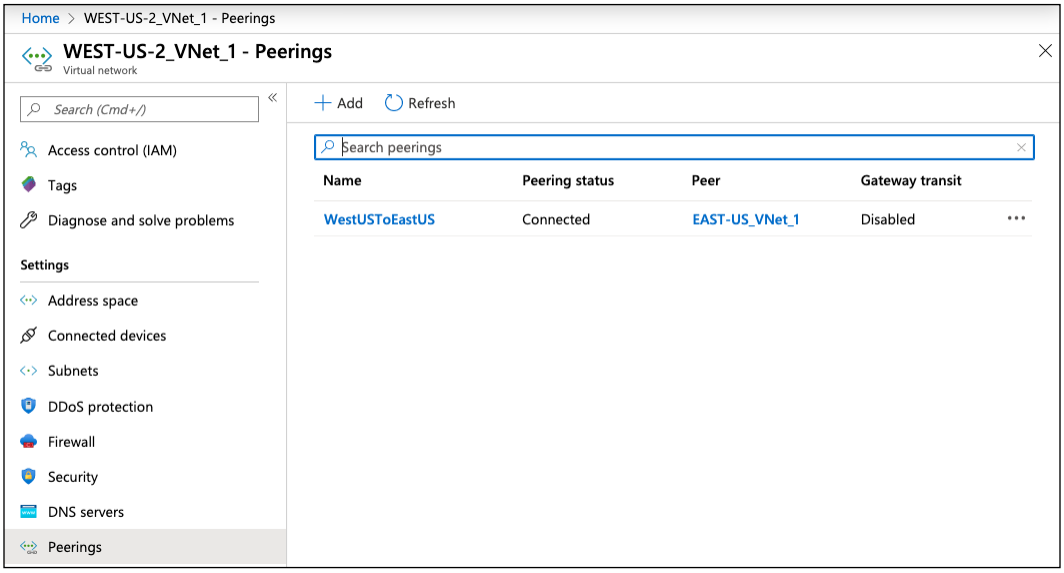


Figure 20: Azure VNet peering

Now that we have several hosts, subnets, VNets, and VNet peering in our setup, we should take a 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 be complex at times, but every packet and route is accounted for. For virtual networks in the cloud, obviously, 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**:

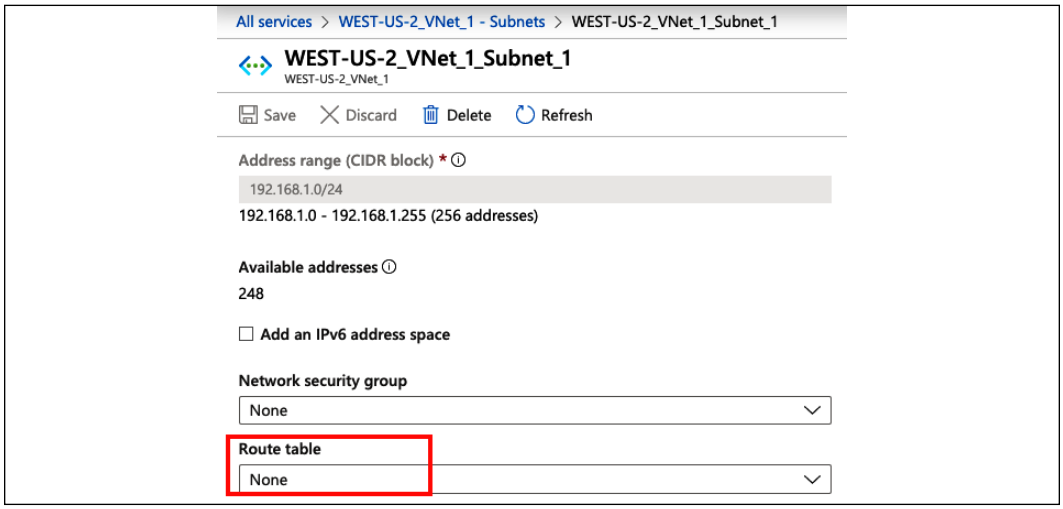


Figure 21: Azure subnet routing table

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? It turns out the routing has been implemented on the host and NIC level. We can see it via **All Services -> Virtual Machines -> myNPM-VM1 -> Networking (left panel) -> Topology (top panel)**:

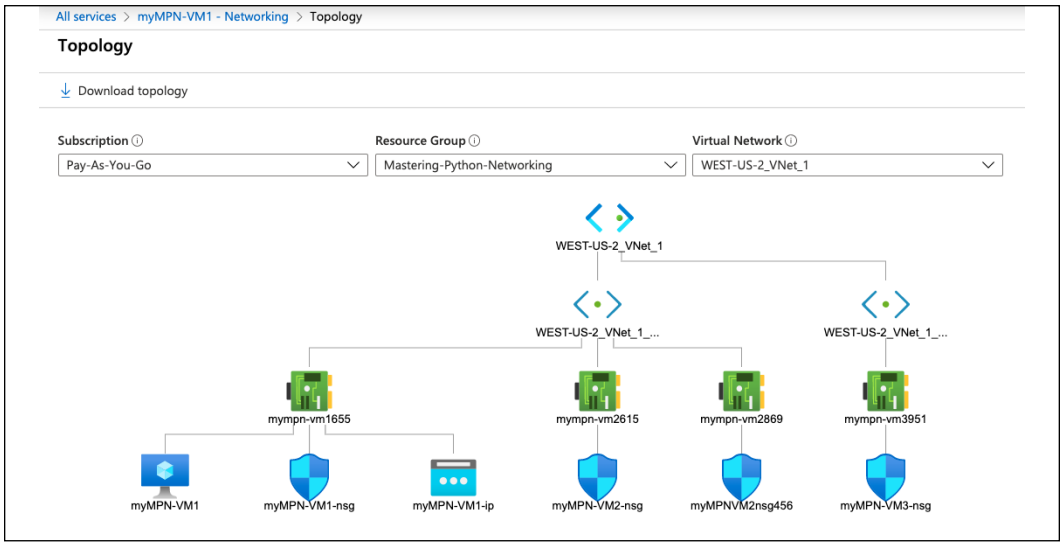


Figure 22: Azure network topology

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 Network Security Groups 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:

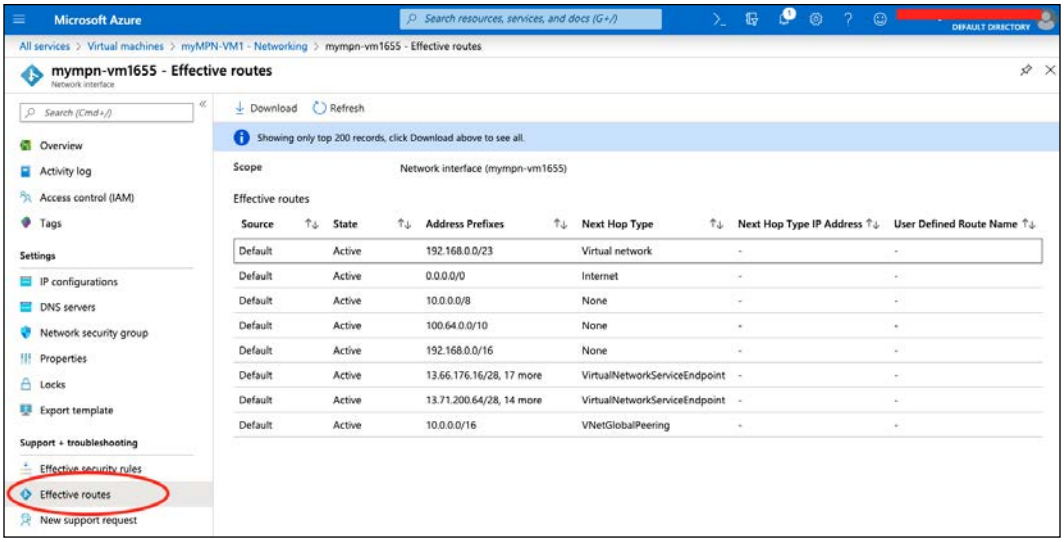


Figure 23: Azure VNet effective routes

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 Site-to-Site VPN or ExpressRoute)
- System route

We can create a route table in the **Networking** section:

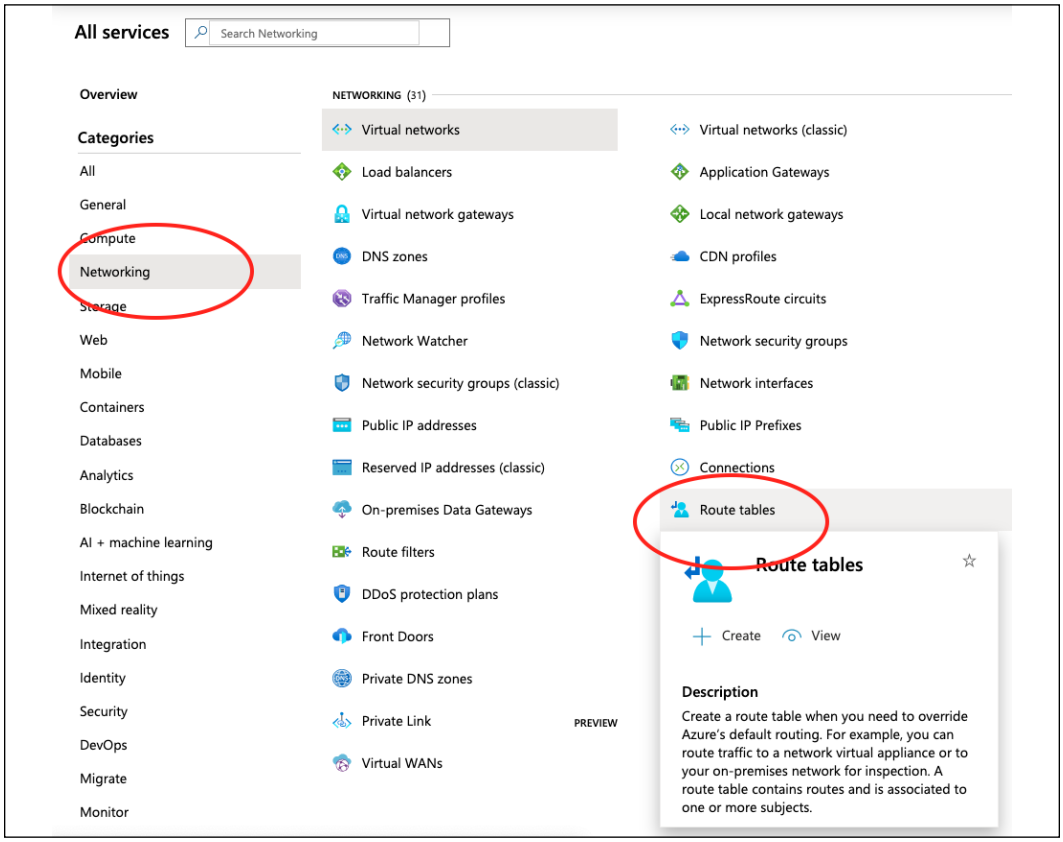


Figure 24: Azure VNet route tables

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 a 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, a 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**:

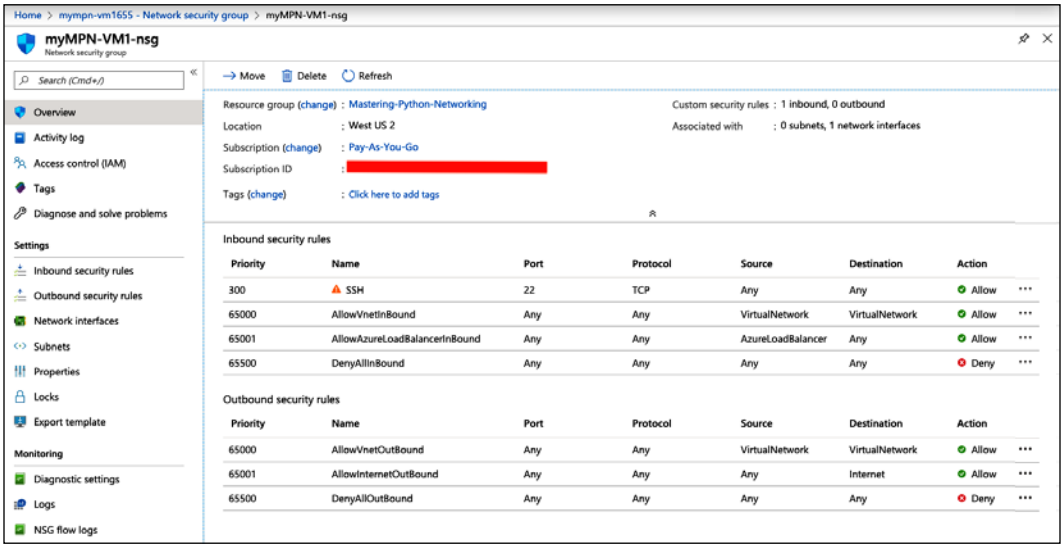


Figure 25: Azure VNet NSG

There are a few things worth pointing out:

- The priority level of system-implemented rules are 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:

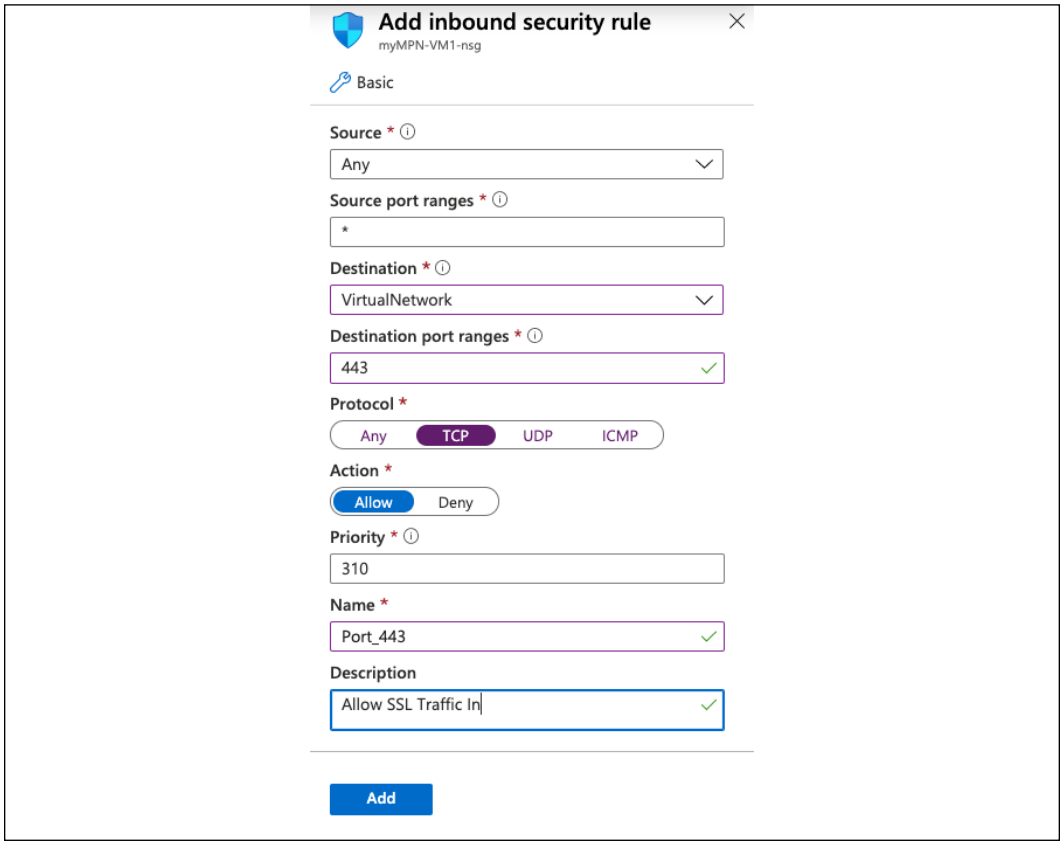


Figure 26: Azure security rule

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:

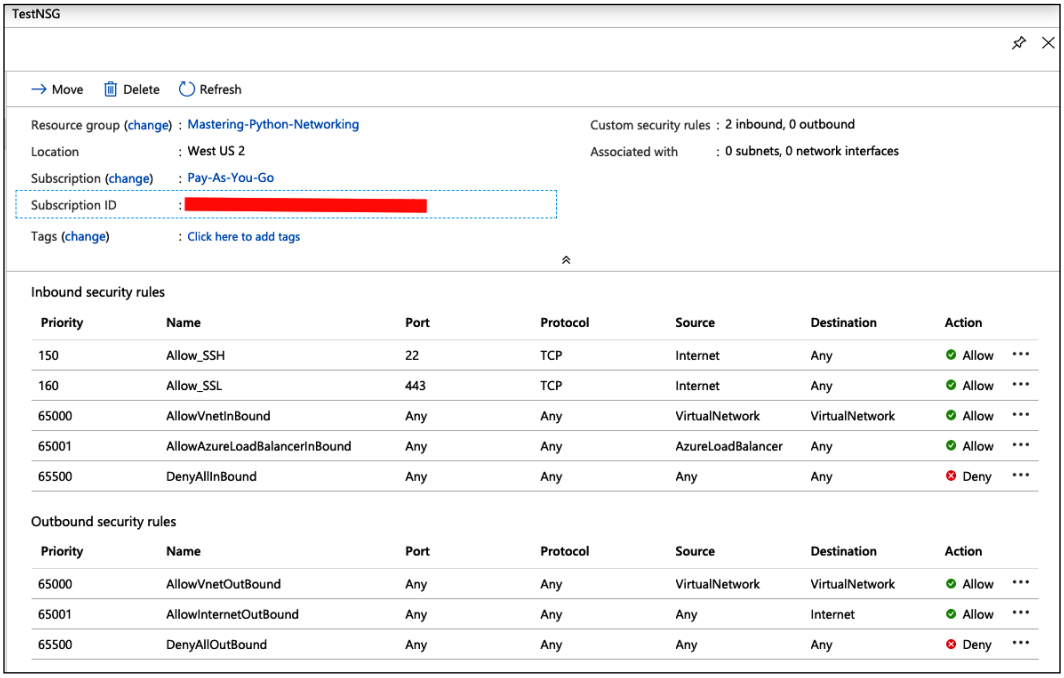


Figure 27: Azure security rules

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 virtual machines 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 28: 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:

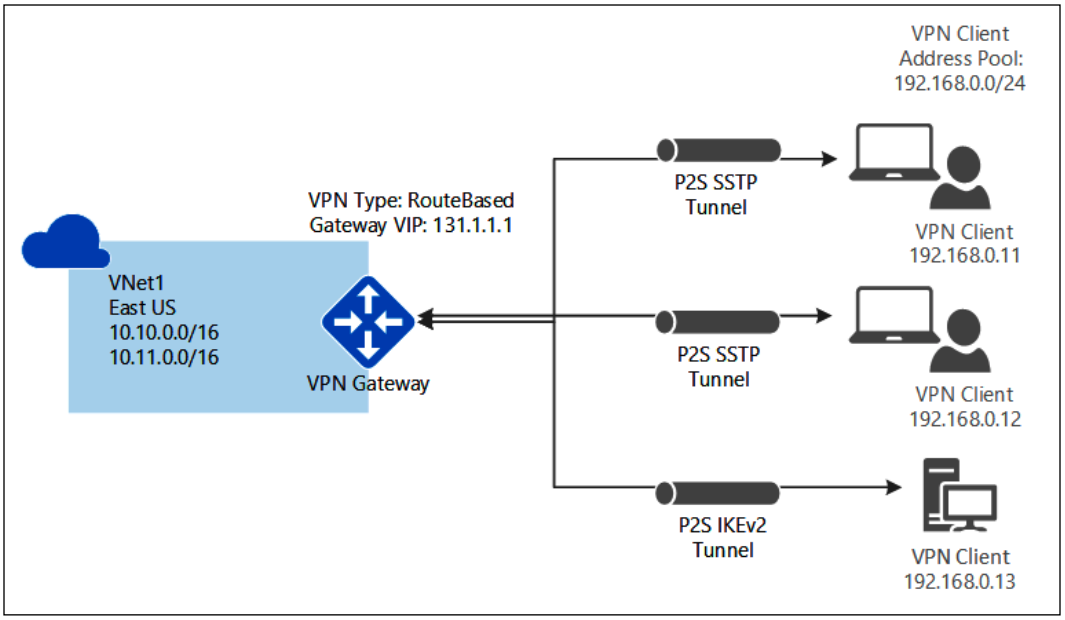


Figure 29: Azure Site-to-Site VPN gateway
(source: https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways)

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:

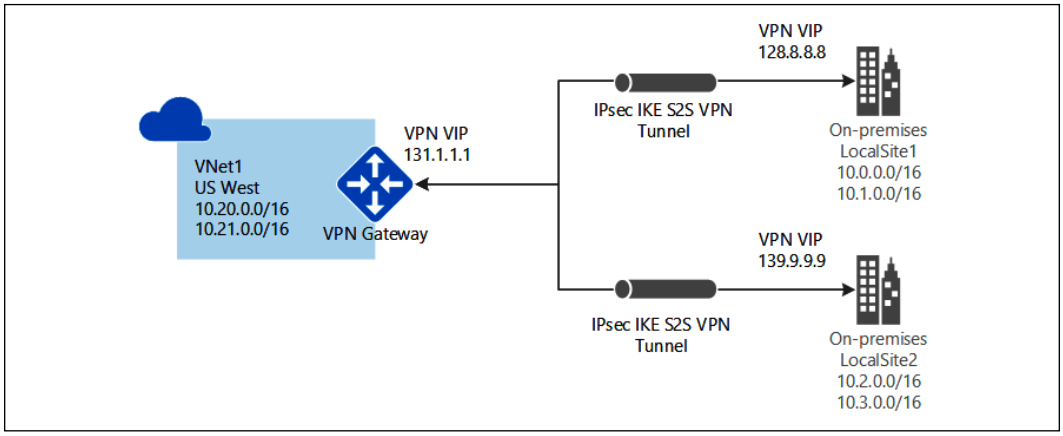


Figure 30: Azure client VPN gateway
(source: https://docs.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-about-vpngateways)

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-point-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:

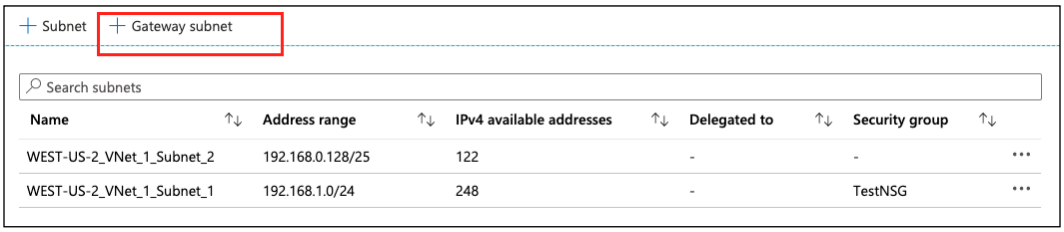


Figure 31: Azure VPN gateway subnet

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:

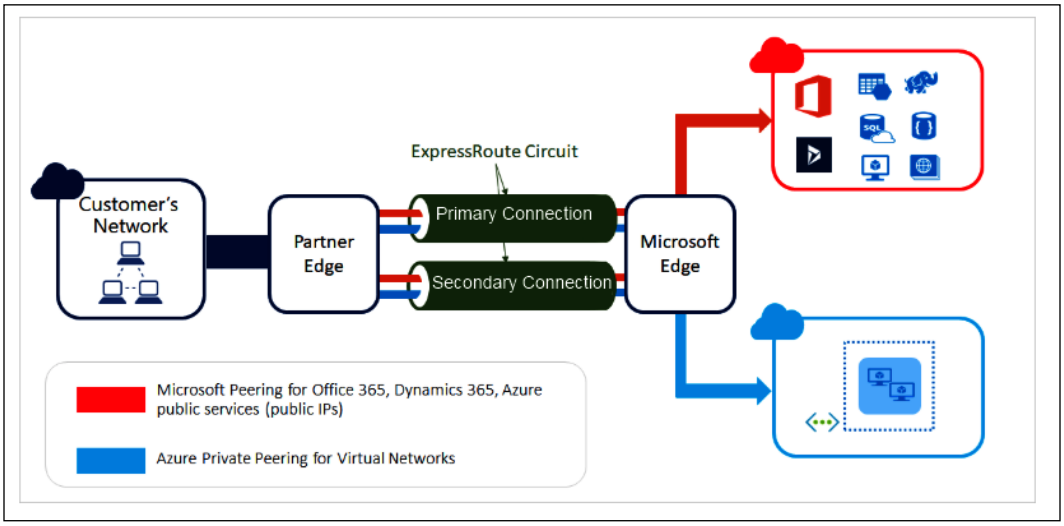


Figure 32: Azure Express Route Circuits
(source: https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction)

The advantages of ExpressRoute include:

- More reliability, 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 the 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 a 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 10*, *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 CloudExchange (`https://www.equinix.com/resources/data-sheets/equinix-cloud-exchange-for-NSP/`).

In the next section, we will take a 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:

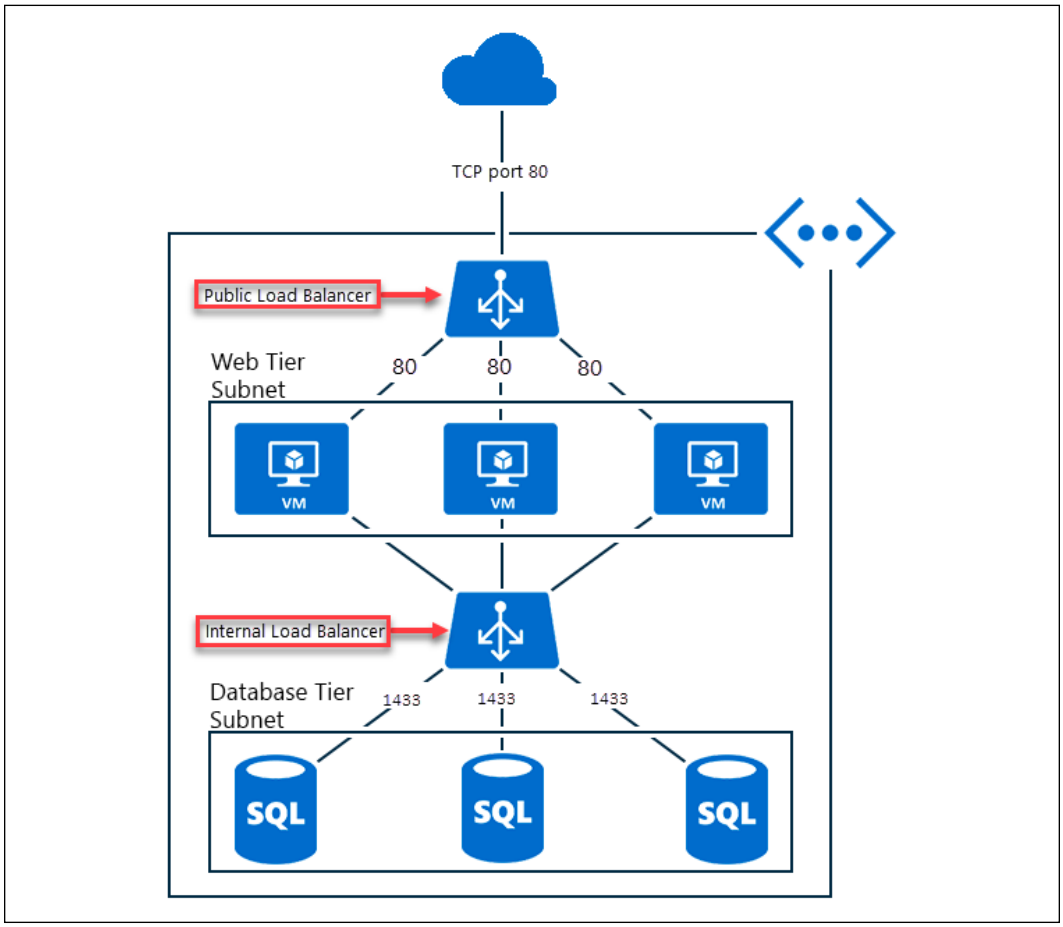


Figure 33: Azure Load Balancer
(source: https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview)

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 virtual machine 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: Elastic Stack.

12
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 in which we can 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 in order to decide what the data means. Most of the time, the analyzed results are displayed in a graph, whether that be a line graph, bar graph, or a pie chart. We can use individual tools such as PySNMP, Matplotlib, and Pygal for each of the steps, or we can leverage all-in-one tools such as Cacti or Ntop for monitoring. The tools introduced in those two chapters allowed us to have 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 10*, *AWS Cloud Networking*, and *Chapter 11*, *Azure Cloud Networking*, we extended our on-premises network to the cloud by way of AWS and Azure. We have covered a lot of 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 own 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.*

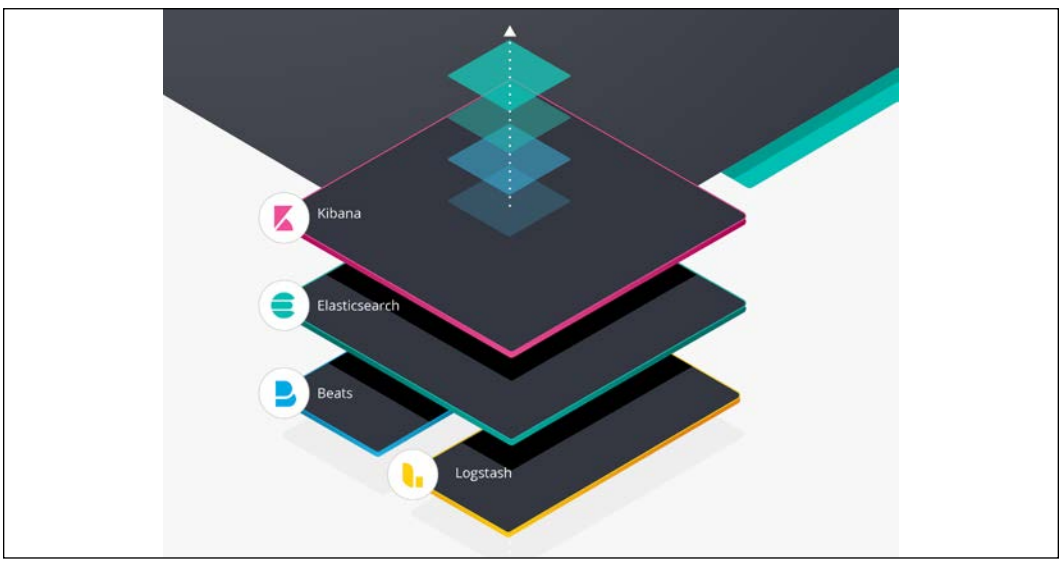


Figure 1: Elastic Stack (source: https://www.elastic.co/what-is/elk-stack)

As we can see from the statement, the Elastic Stack is really 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 do not like 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, which is 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, but the stack has many different use cases including risk management, e-commerce personalization, security analysis, fraud detection, and more. They are being used by a range of organizations; from web companies such as Cisco, Box, and Adobe, to government agencies such as NASA JPL, the United States Census Bureau, and more (`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 `172.16.1.0/24` 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? One option is to install the ELK Stack on the management station we have been using up to this point. Another option is to install it on a separate **virtual machine** (**VM**) besides the management station with two NICs, one connected to the management network and the other connected to the outside network. My personal preference is to separate the monitoring server from the management server and pick the latter option. The reason for this is that the monitoring server typically has different hardware and software requirements than other servers, as you will see in later sections in this chapter. Another reason for separation is that this setup is more in line with what we will typically see in production; it allows us to separate the management and monitoring between the two servers. Following is a graphical representation of our lab topology:

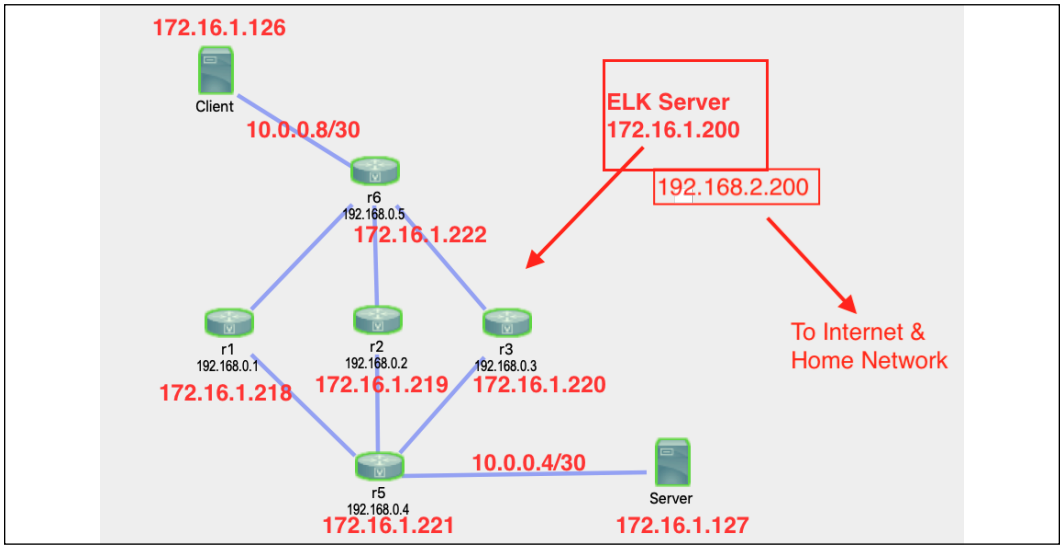


Figure 2: Lab topology

The ELK Stack will be installed on a new Ubuntu 18.04 server with two NICs, with the first NIC's IP address located in the same management network, `172.16.1.200`. The VM will have a second NIC with an IP address of `192.168.2.200` that connects my home network with internet access.

In production, we typically want our ELK cluster to have at least a 3-node system with master and data nodes. In terms of functions, ELK master nodes can control the cluster and index the data while data nodes can perform data-retrieval operations. The 3-node system is recommended for redundancy: 1 node is the active master, while the 2 other nodes are master-eligible if the master node goes down. All three nodes will also be data nodes. We do not need to worry about that for the lab; instead, we will install a 1-node system with a node that is both the master and data node without redundancy.

The hardware requirements for the ELK Stack largely depend on the amount of data we want to put in the system. Since this is a lab, we will not need a lot of horsepower behind the hardware since we will not have a ton of data.

For more information on setups, check out the Elastic Stack and production documentation at `https://www.elastic.co/guide/index.html`.

In general, Elasticsearch is more memory-intensive but not as demanding about CPU and storage. We will create a separate VM with the following specifications:

- CPU: 1 vCPU
- Memory: 4 GB (more if possible)
- Disk: 20 GB
- Network: 1 NIC in the lab management network, an (optional) additional NIC for internet access

Elasticsearch is built using Java; each distribution includes a bundle version of OpenJDK. We can download the latest Elasticsearch version from Elastic.co:

```
echou@elk-stack-mpn:~$ wget https://artifacts.elastic.co/downloads/
elasticsearch/elasticsearch-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~$ tar -xvzf elasticsearch-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~$ cd elasticsearch-7.4.2/
```

We will need to tweak the default virtual memory settings on the node (`https://www.elastic.co/guide/en/elasticsearch/reference/current/vm-max-map-count.html`):

```
echou@elk-stack-mpn:~$ sudo sysctl -w vm.max_map_count=262144
```

Elasticsearch nodes, by default, will try to discover and form a cluster with other nodes. It is best practice to change the node name and cluster-related items before launching. Let's configure the settings in the `elasticsearch.yml` file:

```
echou@elk-stack-mpn:~/elasticsearch-7.4.2$ vim config/elasticsearch.yml
# change the following settings
node.name: mpn-node-1
network.host: <change to your host IP>
http.port: 9200
discovery.seed_hosts: ["mpn-node-1"]
cluster.initial_master_nodes: ["mpn-node-1"]
```

We can now run Elasticsearch in the background:

```
echou@elk-stack-mpn:~/elasticsearch-7.4.2$ ./bin/elasticsearch &
```

We can test the result by performing an `HTTP GET` request on the host running Elasticsearch this can be done on either the management host or locally on the monitoring host:

```
(venv) $ curl 192.168.2.200:9200
{
  "name" : "mpn-node-1",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "9hTywXc-S9eg3jMi6__XSQ",
  "version" : {
    "number" : "7.4.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "2f90bbf7b93631e52bafb59b3b049cb44ec25e96",
    "build_date" : "2019-10-28T20:40:44.881551Z",
    "build_snapshot" : false,
    "lucene_version" : "8.2.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Let's repeat this process for the installation of Kibana, our visualization tool:

```
echou@elk-stack-mpn:~/ $ wget https://artifacts.elastic.co/downloads/
kibana/kibana-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~/ $ tar -xvzf kibana-7.4.2-linux-x86_64.tar.gz
echou@elk-stack-mpn:~/ $ cd kibana-7.4.2-linux-x86_64/
```

We'll make some configuration changes in the configuration file as well:

```
echou@elk-stack-mpn:~/ kibana-7.4.2-linux-x86_64$ vim config/kibana.yml
server.port: 5601
server.host: "192.168.2.200"
server.name: "mastering-python-networking"
elasticsearch.hosts: ["http://192.168.2.200:9200"]
```

We can start the Kibana process in the background:

```
echou@elk-stack-mpn:~/kibana-7.4.2-linux-x86_64$ ./bin/kibana &
```

Once the process has finished launching, we can point our browser to `http://<ip address>:5601`:

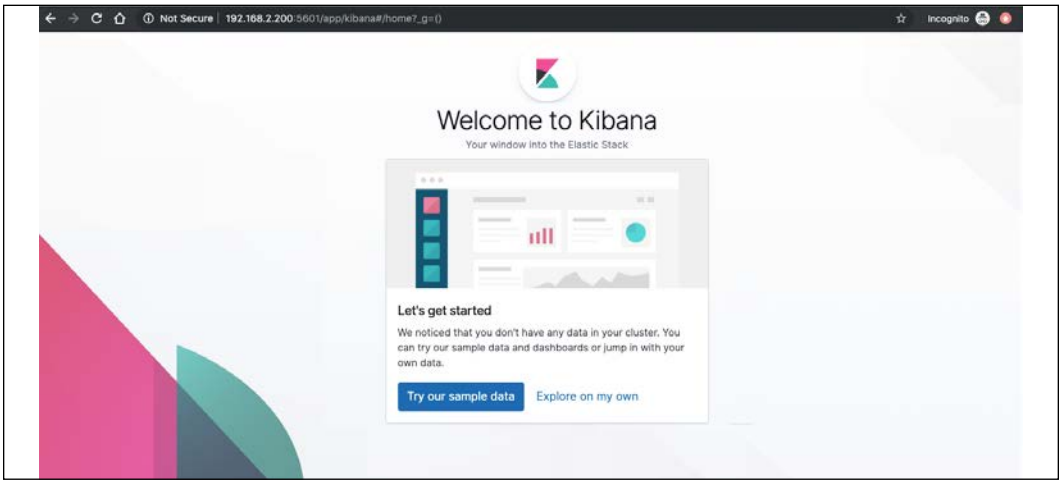


Figure 3: Kibana landing page

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:

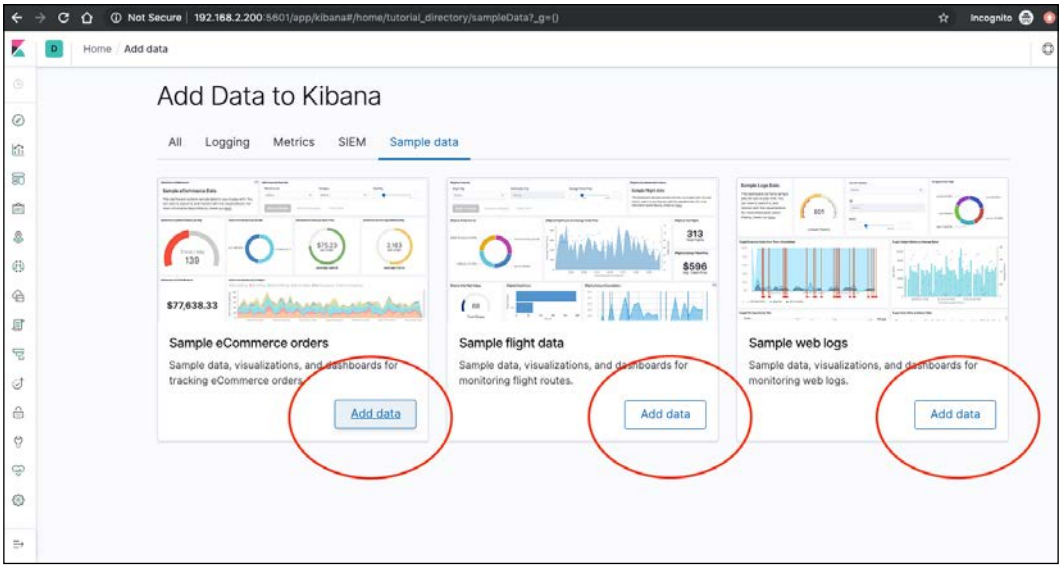


Figure 4: Add Data to Kibana

Great! We are almost done. The last piece of the puzzle is Logstash. Unlike Elasticsearch, the Logstash bundle does not include Java out of the box. We will need to install Java 8 or Java 11 first:

```
echou@elk-stack-mpn:~$ sudo apt install openjdk-11-jre-headless
echou@elk-stack-mpn:~$ java --version
openjdk 11.0.4 2019-07-16
OpenJDK Runtime Environment (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3)
OpenJDK 64-Bit Server VM (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3,
mixed mode, sharing)
```

We can download, extract, and configure Logstash in the same way that we did for Elasticsearch and Kibana:

```
echou@elk-stack-mpn:~$ wget https://artifacts.elastic.co/downloads/
logstash/logstash-7.4.2.tar.gz
echou@elk-stack-mpn:~$ tar -xvzf logstash-7.4.2.tar.gz
echou@elk-stack-mpn:~$ cd logstash-7.4.2/
echou@elk-stack-mpn:~/logstash-7.4.2$ vim config/logstash.yml
node.name: mastering-python-networking
http.host: "192.168.2.200"
http.port: 9600-9700
```

We will not start Logstash at this time. We'll wait until we have installed the network-related plugins and created the necessary configuration file later in the chapter in order 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 that is available as a hosted option by both Elastic.co and AWS. 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 the cloud provider, such as AWS:

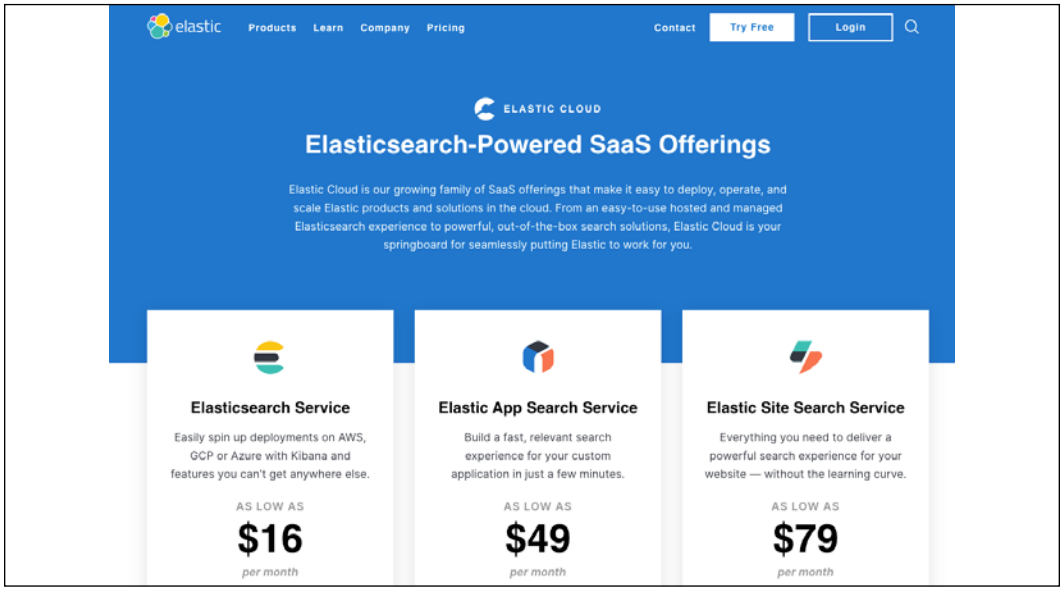


Figure 5: Elastic Cloud offerings

AWS offers a hosted Elasticsearch product (`https://aws.amazon.com/elasticsearch-service/`) that is tightly integrated with the existing AWS offerings. For example, AWS CloudWatch Logs can be streamed directly to the AWS Elasticsearch instance (`https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html`):

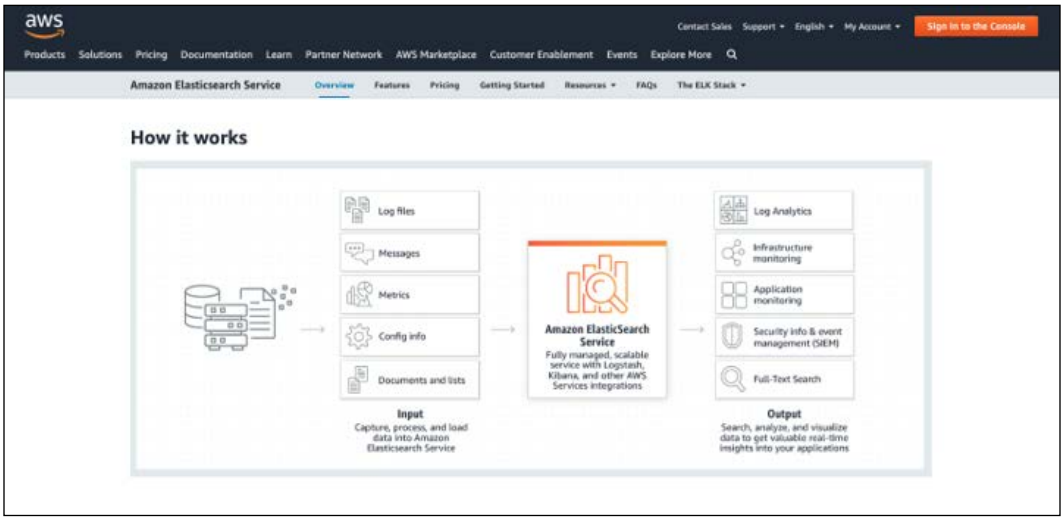


Figure 6: AWS Elasticsearch service

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 Elasticsearch instance saves you the time and effort required to maintain a separate cluster.

Let's take a 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 in order 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.

Before we dig deeper into the different components of the Elastic Stack, it is helpful to look at an example that spans across 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 will be able to 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 172.16.1.200 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 Elasticsearch host:

```
echou@elk-stack-mpn:~$ cd logstash-7.4.2/
echou@elk-stack-mpn:~/logstash-7.4.2$ mkdir network_configs
echou@elk-stack-mpn:~/logstash-7.4.2$ touch network_configs/simple_
config.cfg

echou@elk-stack-mpn:~/logstash-7.4.2$ cat network_configs/simple_config.
cfg
input {
  udp {
    port => 5144
    type => "syslog-ios"
  }
}

output {
  elasticsearch {
    hosts => ["http://192.168.2.200:9200"]
    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:

```
echou@elk-stack-mpn:~/logstash-7.4.2$ sudo bin/logstash -f network_
configs/simple_config.cfg
[2019-11-03T09:54:37,201][INFO ][logstash.inputs.udp      ][main]
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:

```
[2019-11-03T10:01:09,029][INFO ][o.e.c.m.MetaDataCreateIndexService]
[mpn-node-1] [cisco-syslog-2019.11.03] creating index, cause [auto(bulk
api)], templates [], shards [1]/[1], mappings []
[2019-11-03T10:01:09,130][INFO ][o.e.c.m.MetaDataMappingService] [mpn-
node-1] [cisco-syslog-2019.11.03/00NRNwGlRx2OTf_b-qt9SQ] create_mapping
[_doc]
```

At this point, we can do a quick `curl` to see the index created on Elasticsearch:

```
(venv) $ curl http://192.168.2.200:9200/_cat/indices/cisco*
yellow open cisco-syslog-2019.11.03 00NRNwGlRx2OTf_b-qt9SQ 1 1 7 0 20.2kb
20.2kb
```

We can now use Kibana to create the index by going to **Settings -> Kibana -> Index Patterns**:

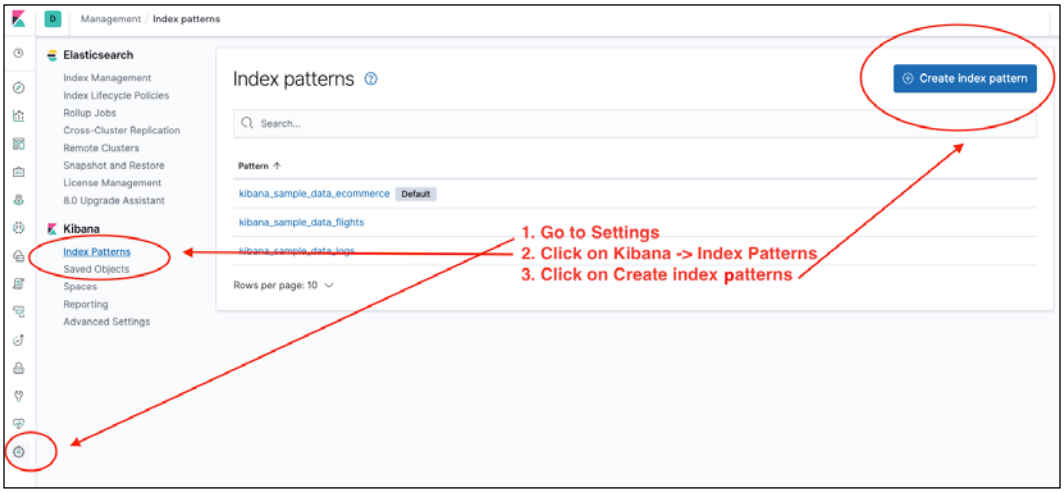


Figure 7: Elasticsearch add Index Patterns

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 the word **Cisco**:

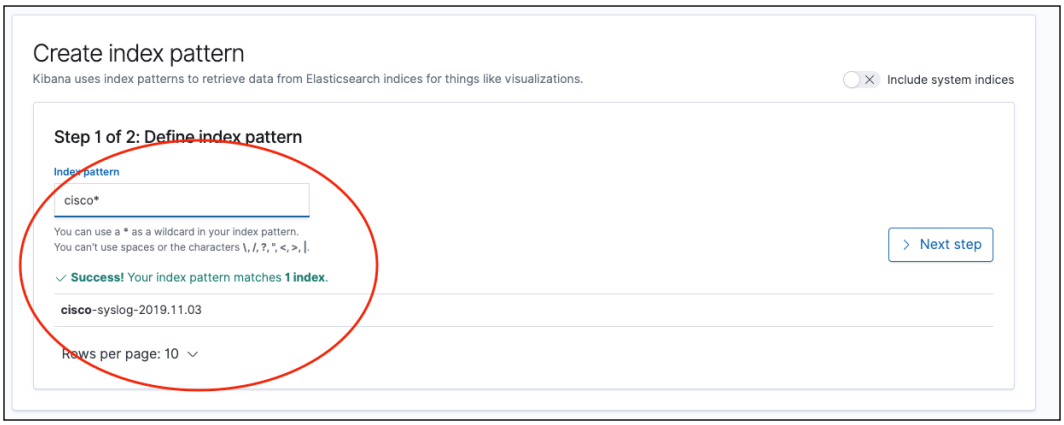


Figure 8: Elasticsearch define index pattern

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:

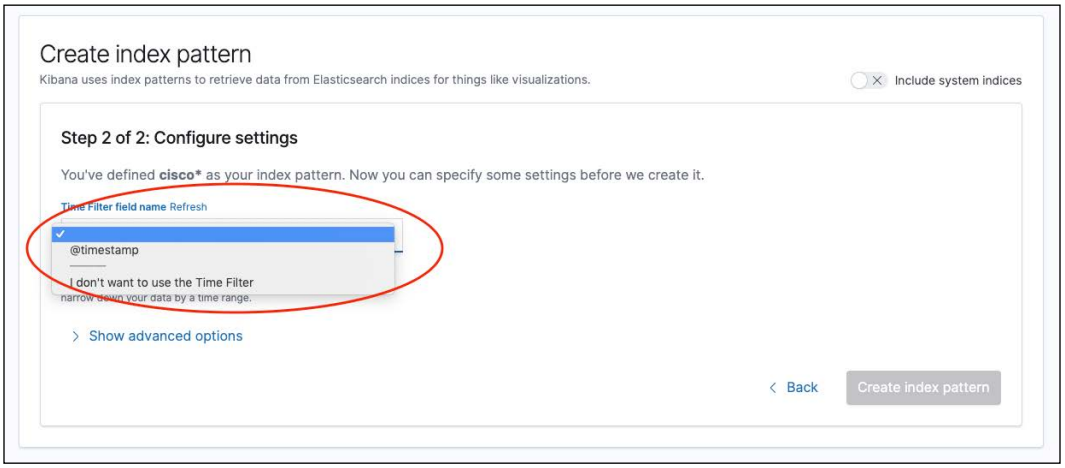


Figure 9: Elasticsearch configure index pattern timestamp

After the index pattern is created, we can use the Kibana **Discover** tab to look at the entries visually:

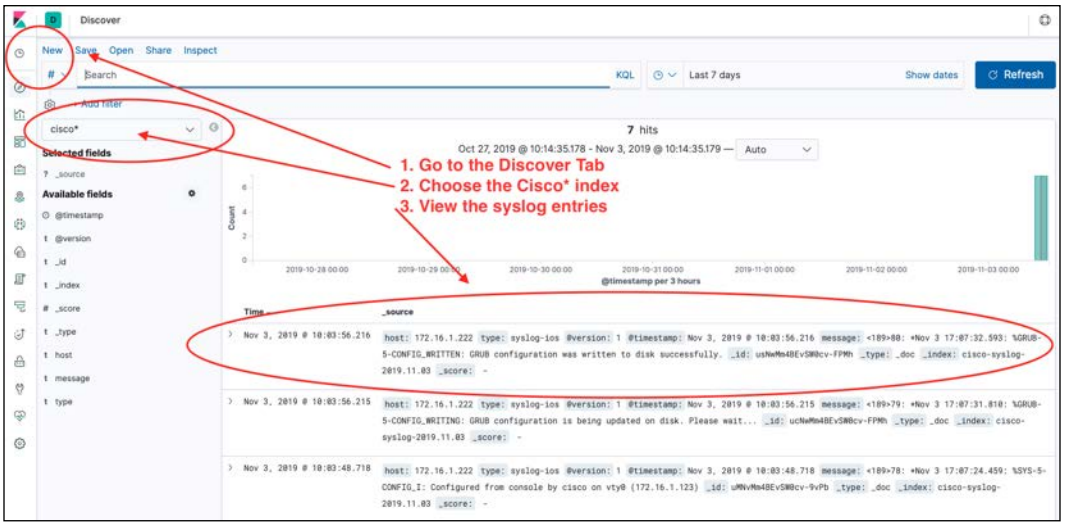


Figure 10: Elasticsearch Index document discovery

After we have collected some more log information, we can stop the Logstash process by using *Ctrl* + *C* on the Elastic Stack server. This first example shows how we can leverage the Elastic Stack from data ingestion, to storage, to 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 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 analyzation needs.

In the next section, we will take a 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`:

```
(venv) $ curl http://192.168.2.200:9200/_cat/indices/kibana*
green open kibana_sample_data_ecommerce Pg5I-1d8SIu-LbpUtn67mA 1 0  4675
0    5mb    5mb
green open kibana_sample_data_logs      3Z2JMdk2T5OPEXnke9l5YQ 1 0 14074
0 11.2mb 11.2mb
green open kibana_sample_data_flights   sjIzh4FeQT2icLmXXhkDvA 1 0 13059
0  6.2mb  6.2mb
```

We can use the `requests` library to make a similar function in a Python script, `Chapter12_1.py`:

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

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)
```

```
        for line in response.text.split('\n'):
            if line:
                current_indices.append(line.split()[2])
        return current_indices

    if __name__ == "__main__":
        es_host = 'http://192.168.2.200:9200'
        indices_list = current_indices_list(es_host, 'kibana')
        print(indices_list)
```

Executing the script will give us a list of indices starting with `kibana`:

```
(venv) $ python Chapter12_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, `Chapter12_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("http://192.168.2.200/")

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:

```
(venv) $ python Chapter12_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 need to create a more complex search operation such as a scroll where we need to use the return 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}" ]
    }
  }
}
<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 172.16.1.200 vrf Mgmt-intf transport udp port 5145
```

When we start the Logstash server, we will see that both ports are now listening:

```
echou@elk-stack-mpn:~/logstash-7.4.2$ sudo bin/logstash -f network_
configs/config_2.cfg
<skip>
[2019-11-03T15:31:35,480][INFO ][logstash.inputs.udp       ][main]
Starting UDP listener {:address=>"0.0.0.0:5145"}
[2019-11-03T15:31:35,493][INFO ][logstash.inputs.udp       ][main]
Starting UDP listener {:address=>"0.0.0.0:5144"}
<skip>
```

By separating out the entries using different types, we can specifically search for the types in the Kibana `Discover` dashboard:

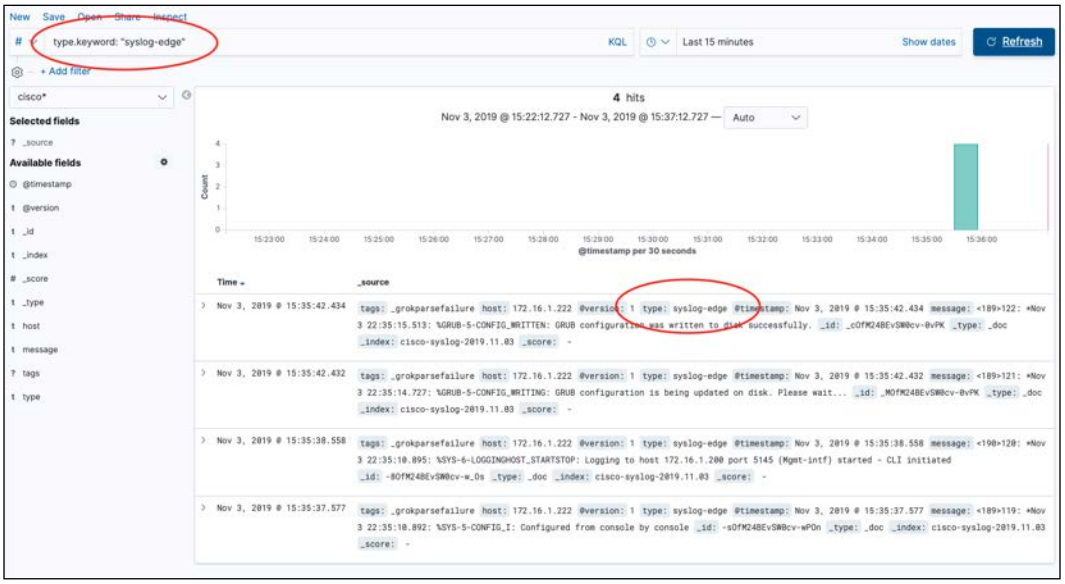


Figure 11: Syslog Index

If we expand on the entry with the `syslog-edge` type, we can see the new field that we added:

Figure 12: 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 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: `https://www.elastic.co/guide/en/logstash/7.4/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 to Logstash for further processing.

There are literally 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 process of data ingestion 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.

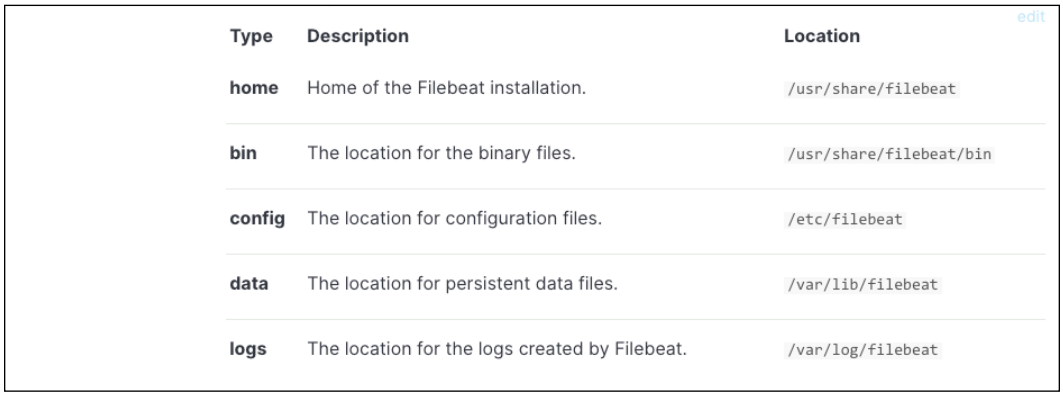
While we cannot completely get away from aggregation and the complexity of data ingestion, the current trend is to move toward a more lightweight, single-purpose agent that sits as close to the data source as possible. For example, we can have a data collection agent installed directly on our Apache 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 the 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 take a 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:

```
echou@elk-stack-mpn:~$ curl -L -O https://artifacts.elastic.co/downloads/
beats/filebeat/filebeat-7.4.2-amd64.deb
echou@elk-stack-mpn:~$ sudo dpkg -i filebeat-7.4.2-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.	`/usr/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: Elastic Filebeat file locations
(source: https://www.elastic.co/guide/en/beats/filebeat/7.4/directory-layout.html)

We will make a few changes to the configuration file, `/etc/filebeat/filebeat.yml`, for the location of Elasticsearch and Kibana:

```
setup.kibana:
  host: "192.168.2.200:5601"
output.elasticsearch:
  hosts: ["192.168.2.200:9200"]
```

Filebeat can be used to set up the index templates and example Kibana dashboards:

```
echou@elk-stack-mpn:~$ sudo filebeat setup --index-management
-E output.logstash.enabled=false -E 'output.elasticsearch.
hosts=["192.168.2.200:9200"]'
echou@elk-stack-mpn:~$ sudo filebeat setup -dashboards
```

Let's enable the `Cisco` module for Filebeat:

```
echou@elk-stack-mpn:~$ sudo filebeat modules enable 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]`:

```
echou@elk-stack-mpn:~$ sudo service filebeat start
```

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:

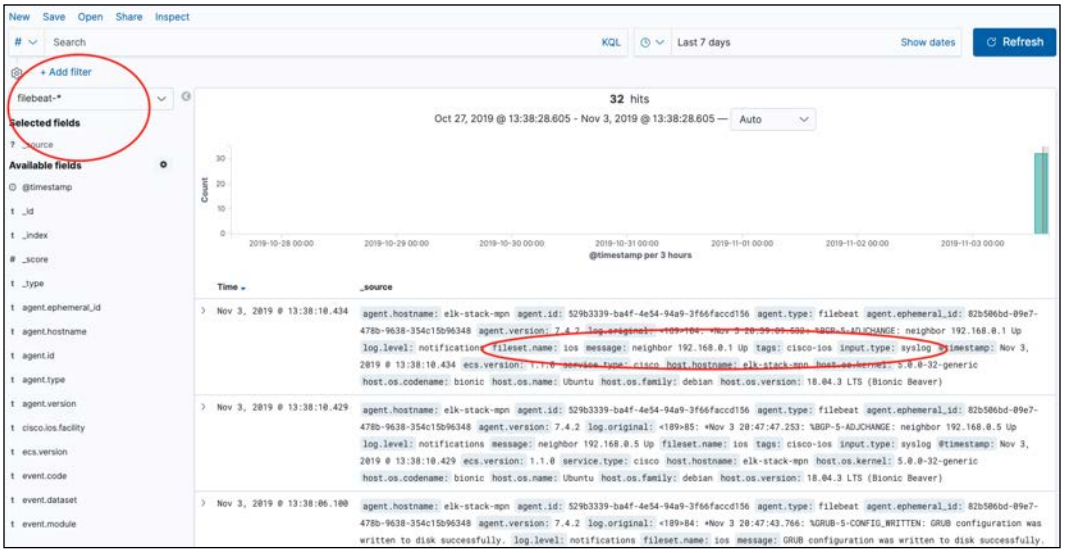


Figure 14: Elastic Filebeat index

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`:

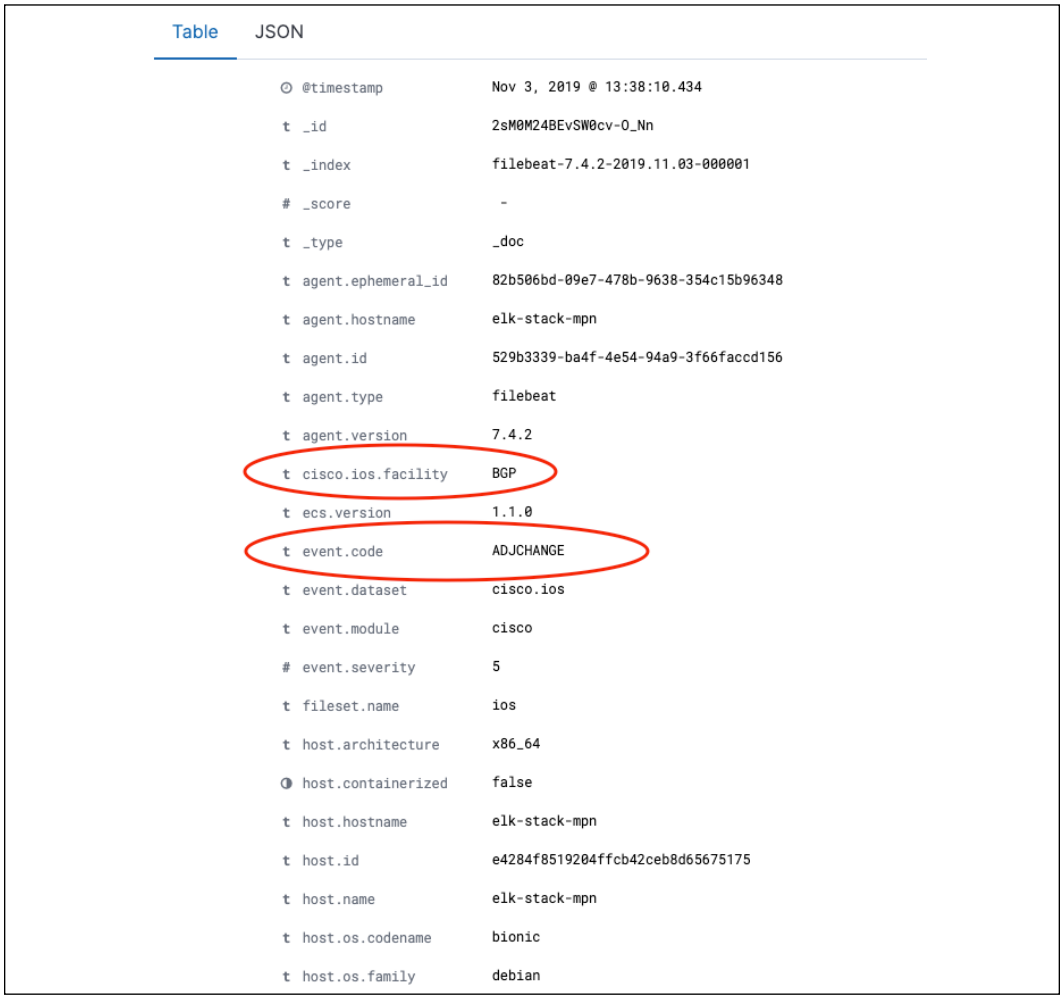


Figure 15: Elastic Filebeat Cisco log

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/7.4/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:

```
echou@elk-stack-mpn:~$ sudo filebeat modules enable netflow
echou@elk-stack-mpn:~$ 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:

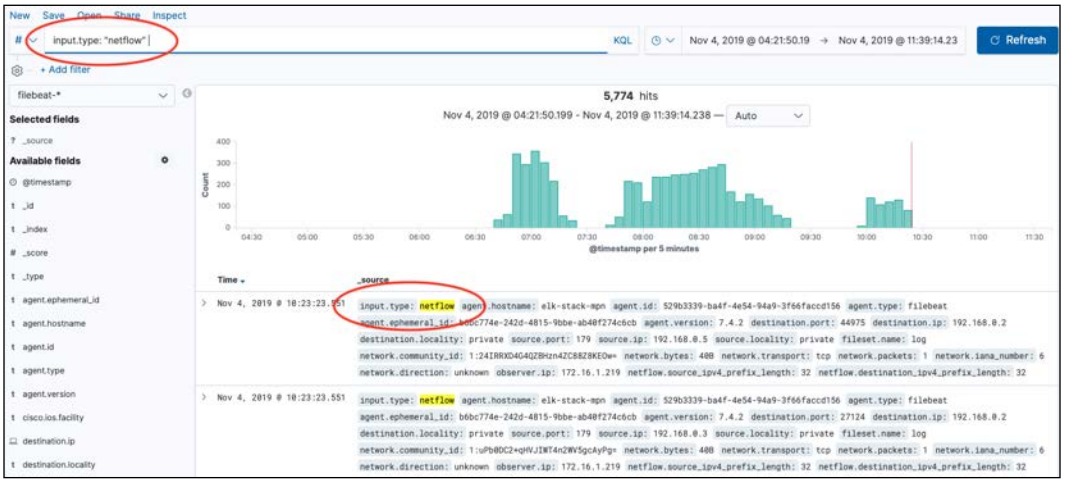


Figure 16: Elastic NetFlow input

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:

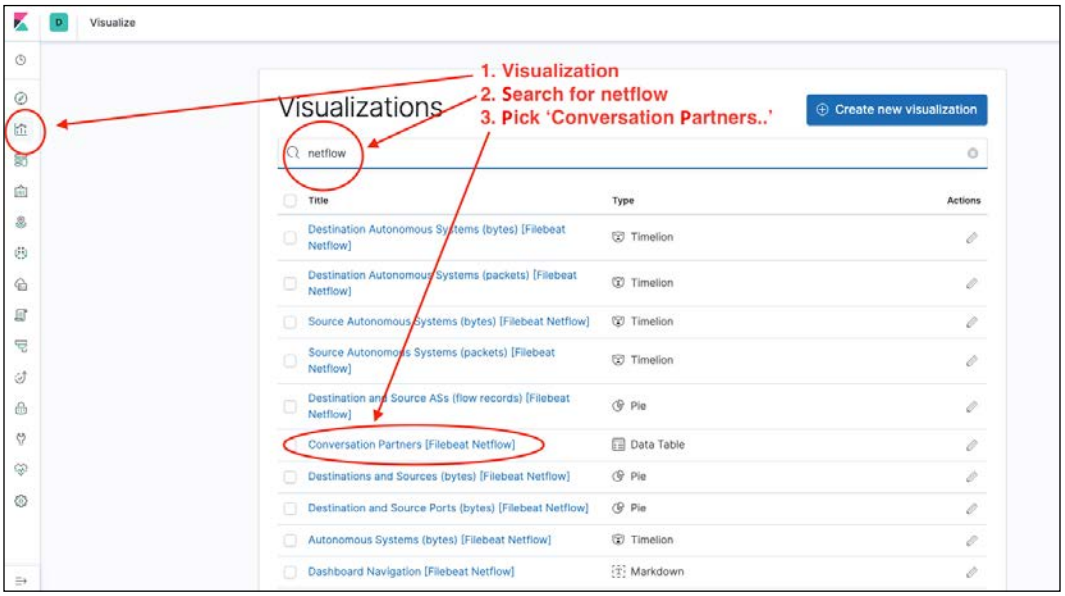


Figure 17: Kibana Visualization

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:

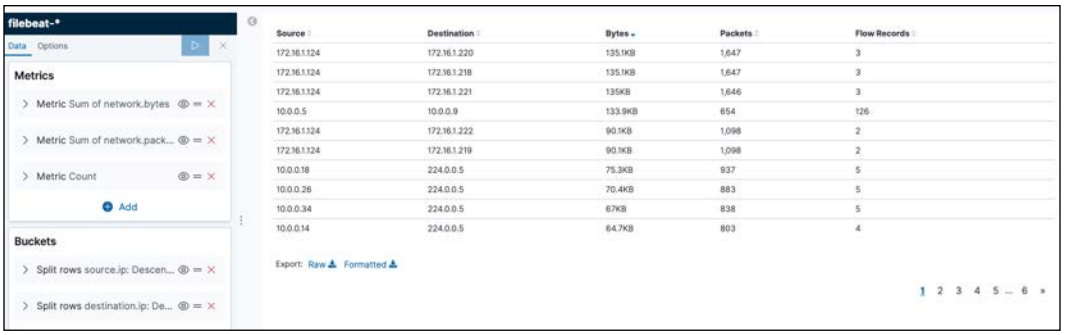


Figure 18: Kibana table

If you are interested in using the ELK Stack for NetFlow monitoring, please also check out the ElastiFlow project: `https://github.com/robcowart/elastiflow`.

In the next section, we will direct our attention to 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 `Chapter12_2.py` script example, when we did the search, 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, `Chapter12_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 = Elasticsearch(['http://192.168.2.200:9200'])
```

```
    # 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:

```
(venv) $ python3 Chapter12_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


(venv) $ python3 Chapter12_3.py -q query_body_1.json -i "cisco*"
50
(venv) $ python3 Chapter12_3.py -q query_body_1.json -i "filebeat*"
10000
```

When we are 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. 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:

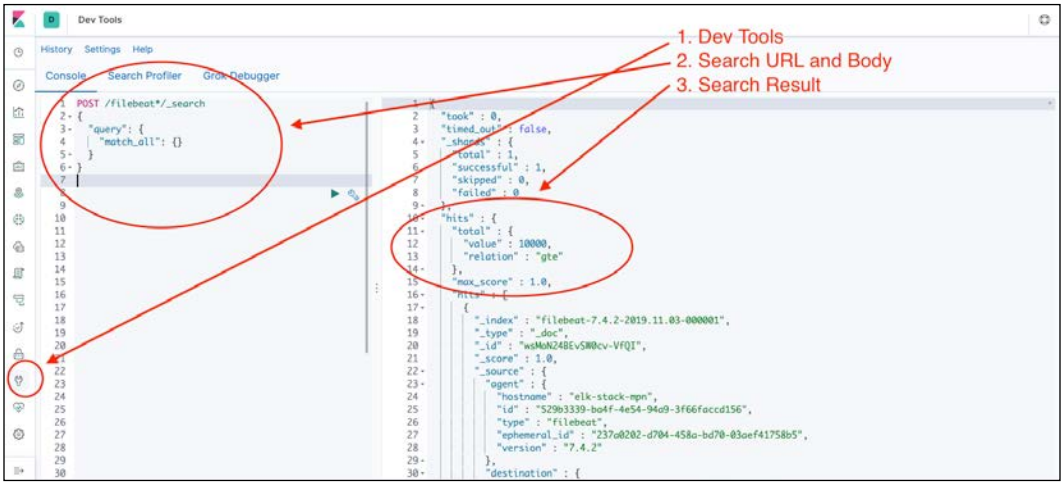


Figure 19: Kibana Dev Tools

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 `Chapter12_3.py` script to `Chapter12_4.py` and modify the output to grab the number of hits as well as 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 `68` hits in the last 10 minutes:

```
(venv) $ python3 Chapter12_4.py -i "filebeat*" -q query_body_2.json
Total hits: 68
```

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:

```
(venv) $ python3 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, `Chapter12_5.py`. The current result is about 1 MB for me when I run the script consecutively:

```
(venv) $ python3 Chapter12_5.py -i "filebeat*" -q query_body_4.json
1089.0
(venv) $ python3 Chapter12_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:

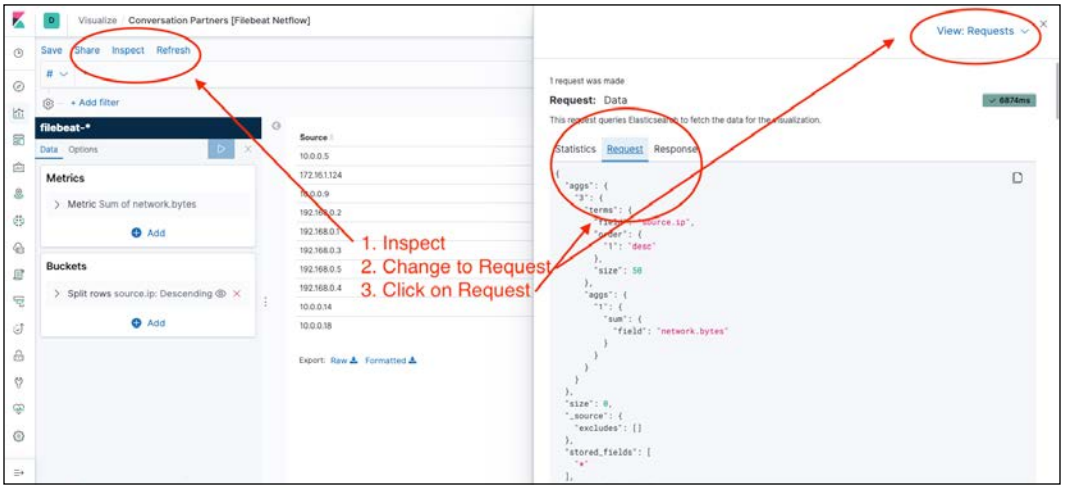


Figure 20: Kibana request

We can put that into a query JSON file, `query_body_5.json`, and execute it with the `Chapter12_6.py` file. We will receive the raw data that the graph was based on:

```
(venv) $ python3 Chapter12_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 the 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 **Visualization -> New Visualization -> Pie**:

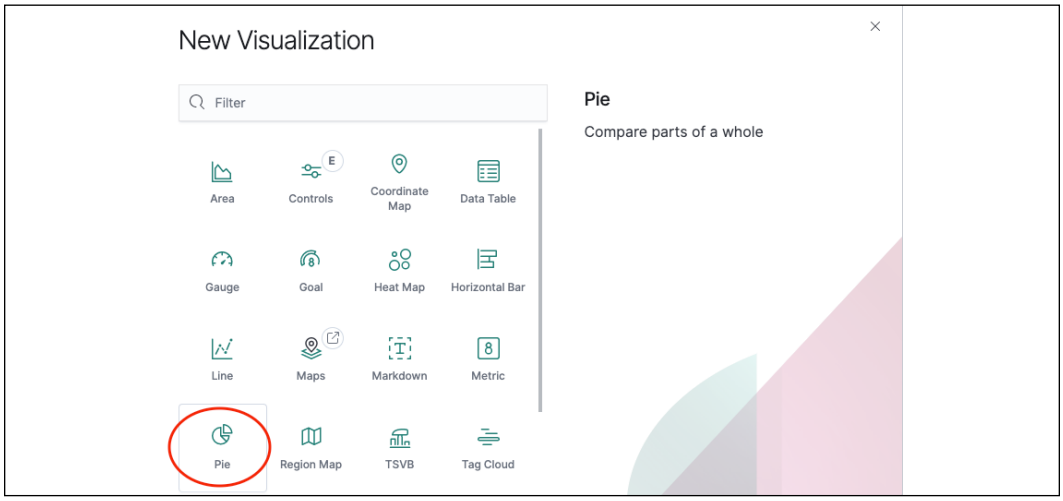


Figure 21: Kibana pie chart

Then we will type **netflow** in the search bar to pick our **[Filebeat NetFlow]** indices:

Figure 22: 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 23: Kibana time range

We can assign a custom label for the graph:

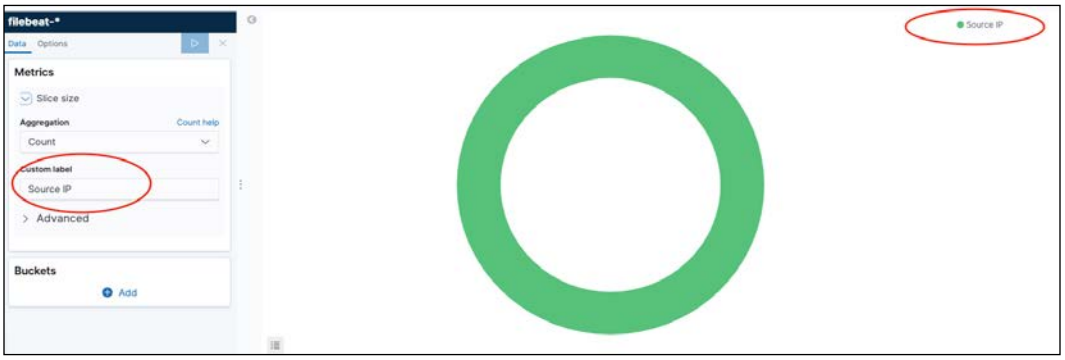


Figure 24: Kibana chart label

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 option for **descending** but increase the **size** to **10**.

The change will only be applied when you click on 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:

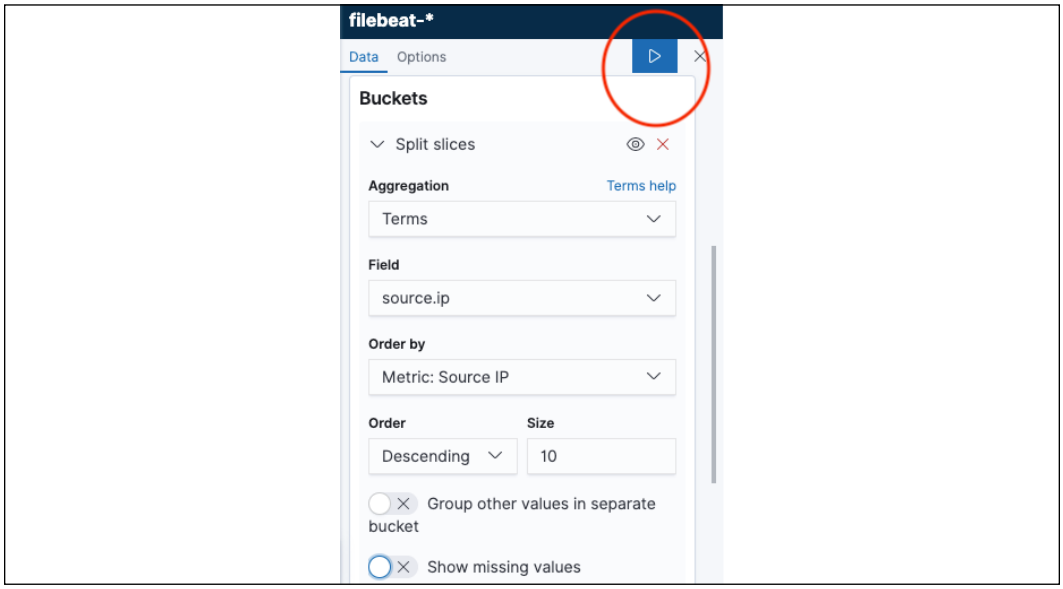


Figure 25: Kibana play button

We can click on the **Options** link at the top to turn off **Donut** and turn on **Show labels**:

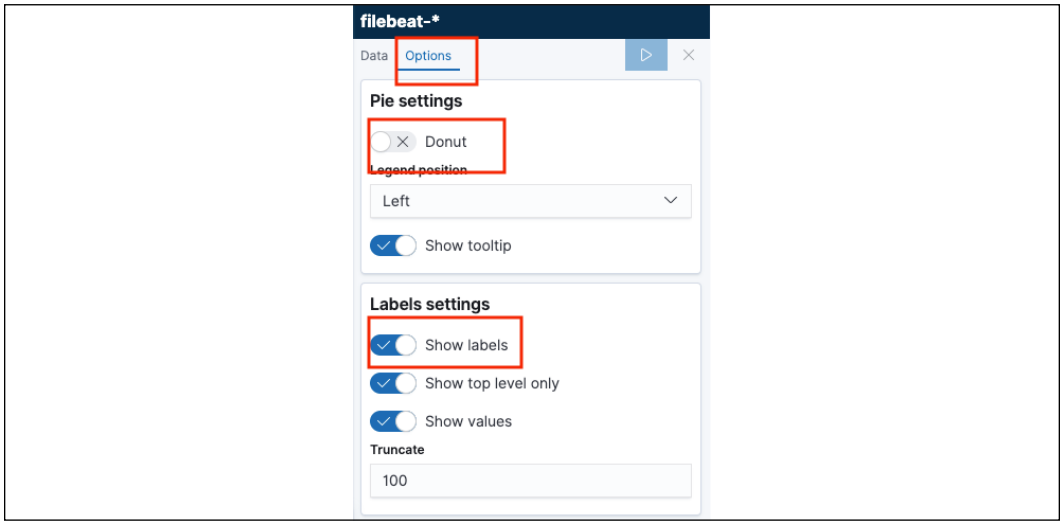


Figure 26: Kibana chart options

The final graph is a nice pie chart showing the top IP sources based on the number of document counts:

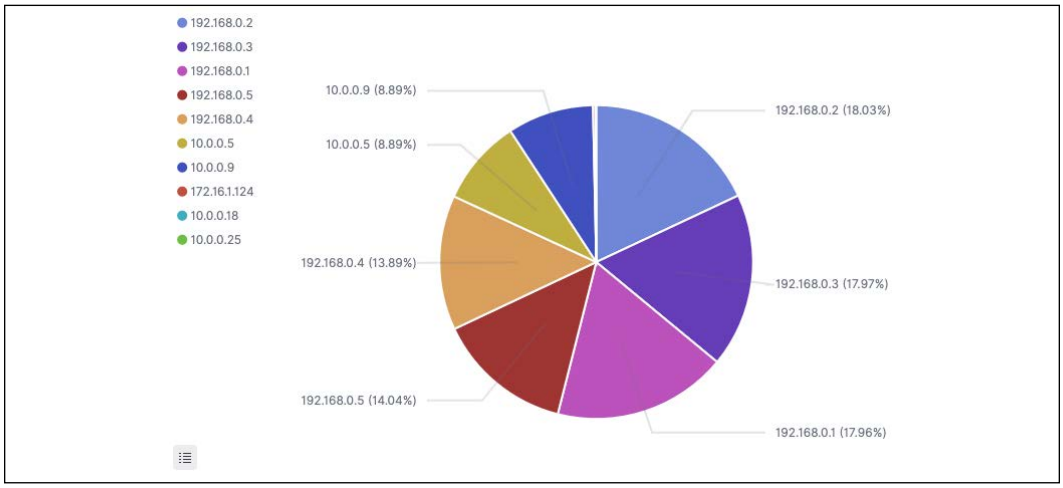


Figure 27: Kibana pie chart

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:

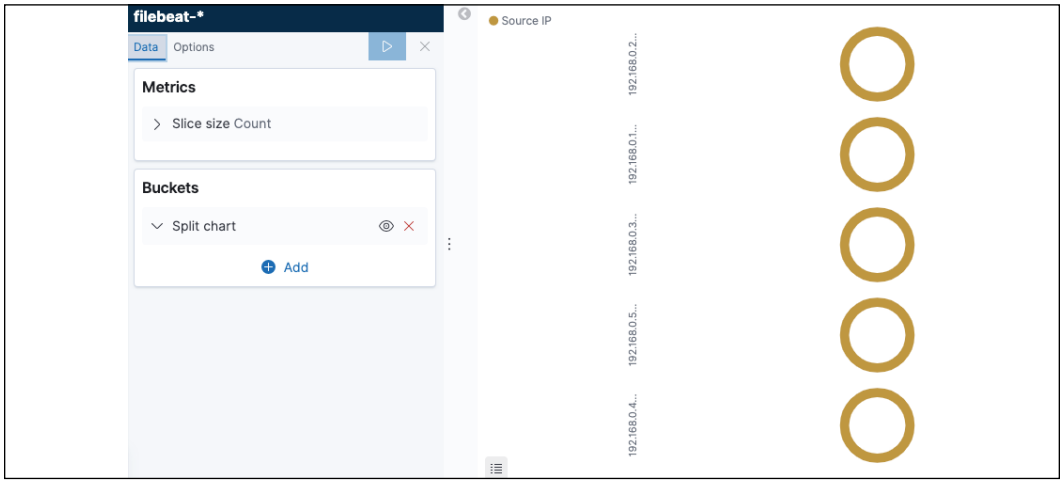


Figure 28: Kibana split chart

Let's stick to splitting things into slices on the same pie chart and change the time range to the **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:

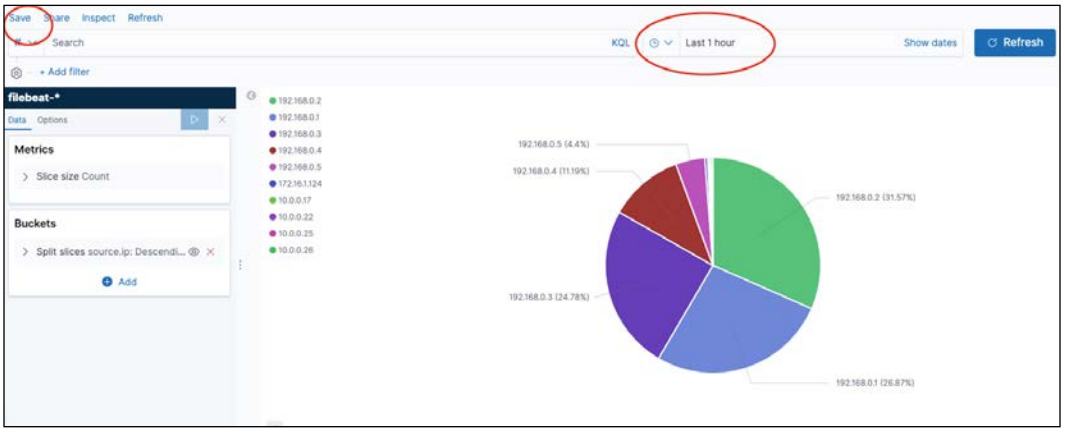


Figure 29: Kibana save chart

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:

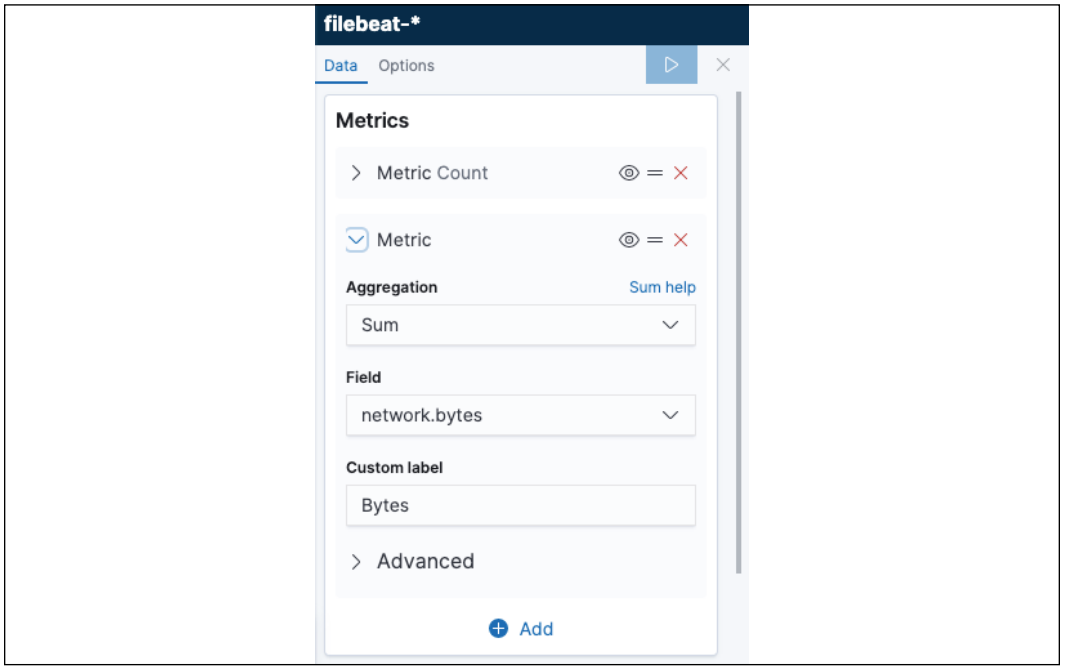


Figure 30: Kibana metrics

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:

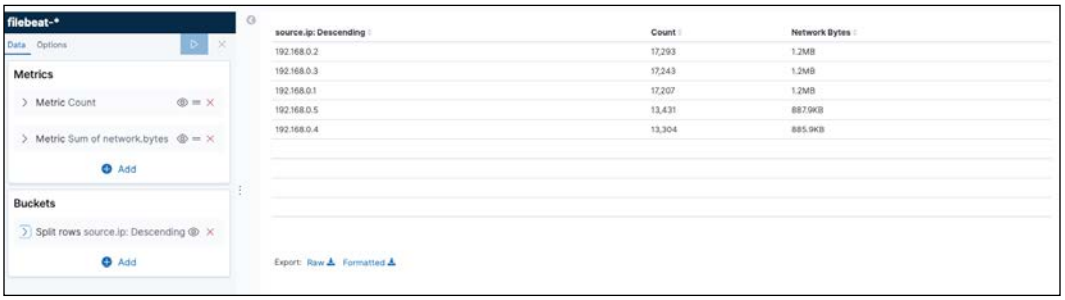


Figure 31: Kibana tables

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/7.4/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 in a meaningful way. 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 span of time 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 use 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.

13
Working with Git

We have worked on various aspects of network automation with Python, Ansible, and many other tools. If you have been following along with the examples, in the first twelve 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 set of scripts and tools, we are now ready to go out and conquer our network tasks, right? Well, not so fast, my fellow network ninjas.

There are a number of 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 the collaboration process between our team members. If we work with other network engineers, we will most likely need to work collectively on the files. The files can be the Python scripts, Ansible Playbook, Jinja2 templates, INI-style configuration files, and many others. The point is any kind of text-based file should be tracked with multiple inputs that everybody in 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 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 up my Microsoft Word program, the file constantly saves itself, and I can go back in time to revisit the changes or rollback 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 of the more 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 was 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 take a 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 on which explanation you like better.

The project came together really quickly. About ten 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 developer's inherent inertia in switching source-control systems. After many years of hosting the Python code from Mercurial at `https://hg.python.org/`, the project was switched to Git on GitHub in February of 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 of the 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 were 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 all together 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 take a look at some of the terms used in Git, before we go any deeper into it.

Git terminology

Here are some Git terms we should be familiar with:

- **Ref**: The name that begins with `refs` that 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 to store a new snapshot into 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 in a point in time in a repository that is significant. In *Chapter 4*, *The Python Automation Framework – Ansible Basics*, we saw tags were used to specify the release points, `v2.5.0a1`.

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 thing. 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. Often, we just 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 that copy as their centralized 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 besides command line; this makes Git more user-friendly.

Now that we've differentiated Git and GitHub, we can properly get started! First, let's talk about setting up Git.

Setting up Git

So far, we have been using Git to just 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 am going to use the same Ubuntu 18.04 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 at the right set of instructions.

If you have not done so already, install Git via the `apt` package-management tool:

```
(venv) $ sudo apt update
(venv) $ sudo apt install -y git
(venv) $ git --version
git version 2.17.1
```

Once `git` is installed, we need to configure a few things so 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 in to 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 making a commit. 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 you create a brand new repository. In fact, GitHub provides an option to do just that when you create a repository on their 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-Third-Edition/blob/master/.gitignore`

I see the `.gitignore` file as a file that should be created at the same time as any new repository. That is why this concept is introduced as early as possible. We will take a look at some of the Git usage examples in the next section.

Git usage examples

In my experience, most of the time when we work with Git, we will likely be using 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 it for the normal branching and commits. We can look at Git's command-line option by using the help option:

```
(venv) $ 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:

```
(venv) $ mkdir TestRepo-1
(venv) $ cd TestRepo-1/
(venv) $ git init
Initialized empty Git repository in /home/echou/Mastering_Python_
Networking_third_edition/Chapter13/TestRepo-1/.git/
(venv) $ 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:

```
(venv) $ ls -a
.  ..  .git  myFile.txt
(venv) $ ls .git/
branches  config  description  HEAD  hooks  info  objects  refs
```

There are several locations Git receives its configurations in a hierarchy format. The files are read from `system`, `global`, and `repository` by default. The more specific the location to the repository, 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 will be able to 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 a `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 ever wonder about the `SHA-1` value being accidentally or purposely modified to overlap, there is an interesting article on GitHub blog about detecting this `SHA-1` 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:

```
(venv) $ 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`. `revert` changes all the file for a specific commit back to its state before the commit:

```
(venv) $ git revert ff7dc1a40e5603fed552a3403be97addefddc4e9
[master 75921be] Revert "made modifications to myFile.txt"
 1 file changed, 1 insertion(+), 1 deletion(-)

(venv) $ 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:

```
(venv) $ 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:

```
(venv) $ git reset --hard ff7dc1a40e5603fed552a3403be97addefddc4e9
HEAD is now at ff7dc1a made modifications to myFile.txt

(venv) $ 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
```

Personally, I like to keep all the history, including any rollbacks that I have done. Therefore, when I need to rollback 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 take a look at how we can work with a collection of files that is group 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. There are many reasons for branching; there are no hard-set rules about when to branch or when to work on just the master branch. 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:

```
(venv) $ git branch dev
(venv) $ git branch
  dev
* master
```

Notice we need to specifically move into the `dev branch` after creation. We do that with `checkout`:

```
(venv) $ git checkout dev
Switched to branch 'dev'
(venv) $ git branch
* dev
  master
```

Let's add a second file to the `dev` branch:

```
(venv) $ echo "my second file" > mySecondFile.txt
(venv) $ git add mySecondFile.txt
(venv) $ 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:

```
(venv) $ git branch
* dev
  master
(venv) $ git checkout master
Switched to branch 'master'
(venv) $ ls
myFile.txt
(venv) $ git checkout dev
Switched to branch 'dev'
(venv) $ 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:

```
(venv) $ git branch
* dev
  master
(venv) $ git checkout master
Switched to branch 'master'
(venv) $ git merge dev master
Updating ff7dc1a..a537bdc
Fast-forward
 mySecondFile.txt | 1 +
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 mySecondFile.txt
(venv) $ git branch
  dev
* master
(venv) $ 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:

```
(venv) $ touch myThirdFile.txt
(venv) $ git add myThirdFile.txt
(venv) $ git commit -m "adding myThirdFile.txt"
[master 169a203] adding myThirdFile.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 myThirdFile.txt
(venv) $ ls
myFile.txt  mySecondFile.txt  myThirdFile.txt
(venv) $ git rm myThirdFile.txt
rm 'myThirdFile.txt'
(venv) $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    myThirdFile.txt
(venv) $ 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:

```
(venv) $ 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 take a 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 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, they also offer unlimited free private repositories. In this case, we will create a private repository and add the license and `.gitignore` file:

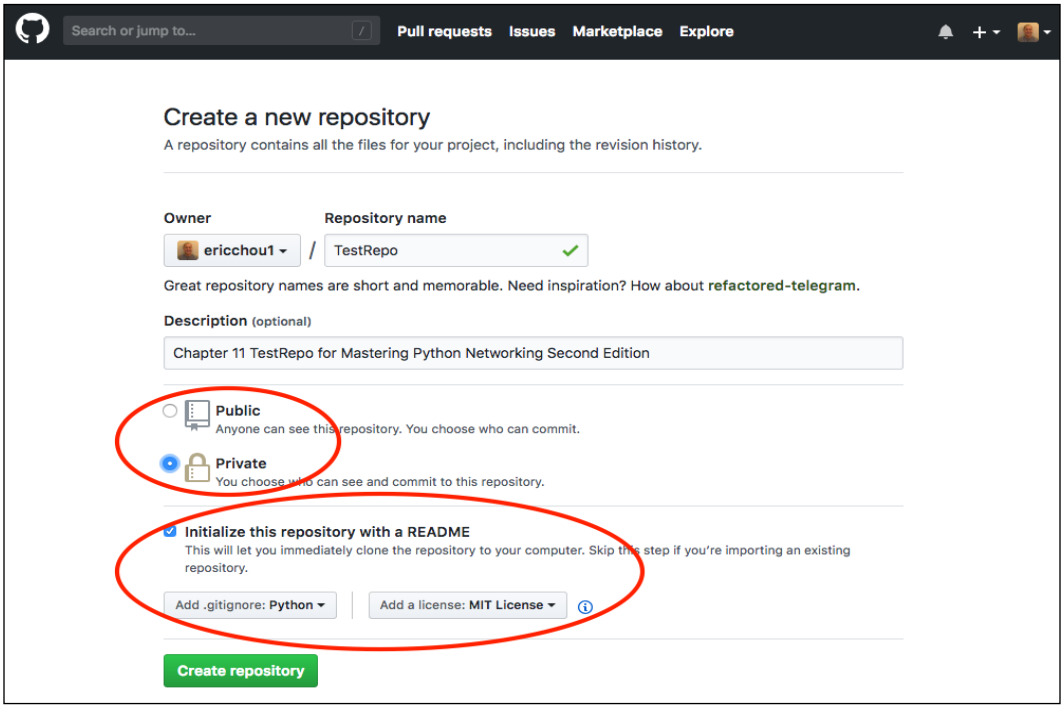


Figure 1: Creating a private repository in GitHub

Once the repository is created, we can find the URL for this repository:

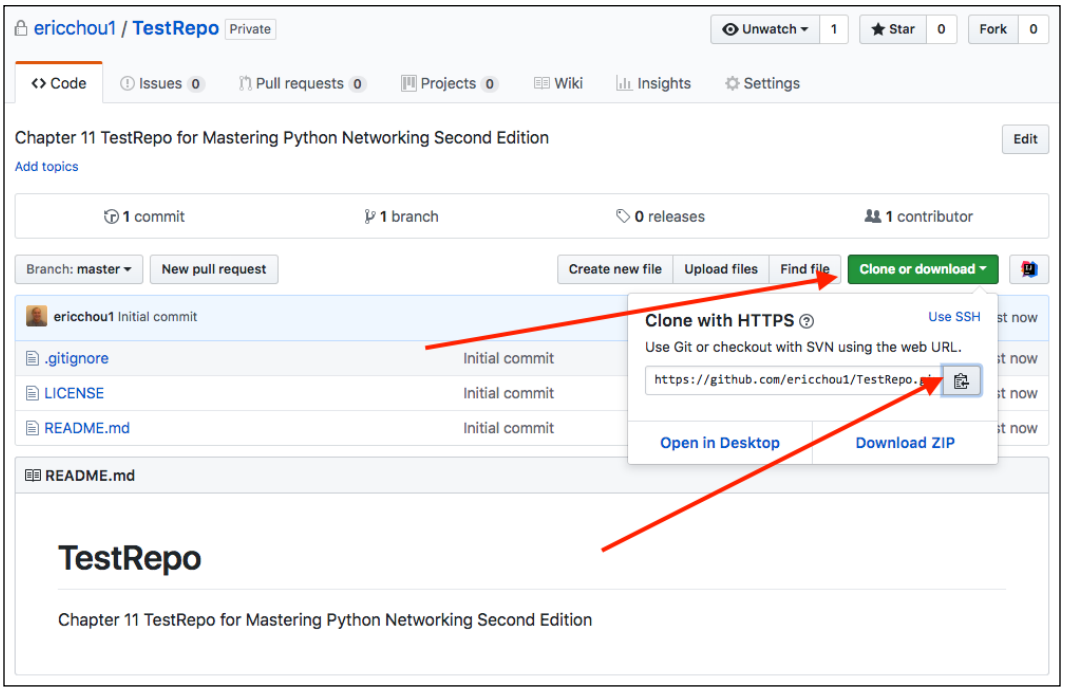


Figure 2: GitHub repository URL

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`:

```
(venv) $ git remote add gitHubRepo https://github.com/ericchou1/TestRepo.
git
(venv) $ git remote -v
gitHubRepo    https://github.com/ericchou1/TestRepo.git (fetch)
gitHubRepo    https://github.com/ericchou1/TestRepo.git (push)
```

Since we chose to create a `README.md` and `LICENSE` file during creation, the remote repository and local repository are not the same.

If we were to push local changes to the GitHub repository, we would receive the following error:

```
(venv) $ git push gitHubRepo master
Username for 'https://github.com': <skip>
Password for 'https://echou@yahoo.com@github.com': <skip>
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:

```
(venv) $ git pull gitHubRepo master
Username for 'https://github.com': <skip>
Password for 'https://<username>@github.com': <skip>
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':
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:

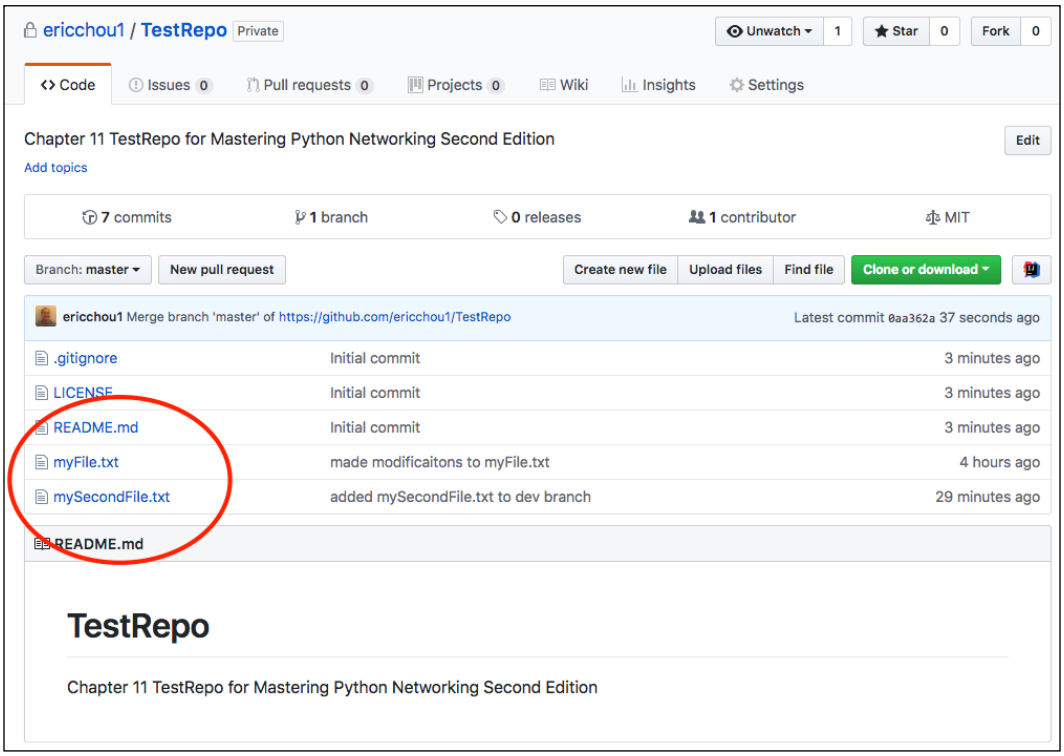


Figure 3: GitHub repository

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:

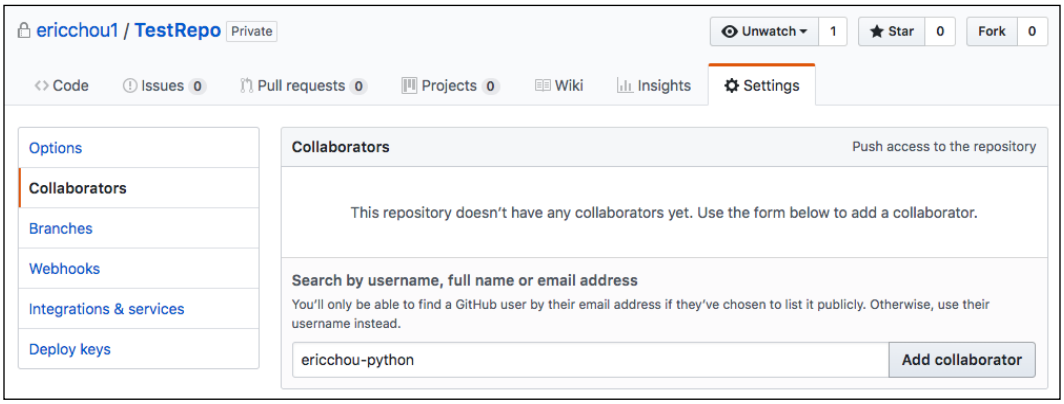


Figure 4: Repository invite

In the next example, we will see how we can fork a repository and perform a pull request for a repository that we do not maintain.

Collaborating with pull requests

As mentioned, Git supports collaboration between developers for a single project. We will take a 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 am going to 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 personal account:

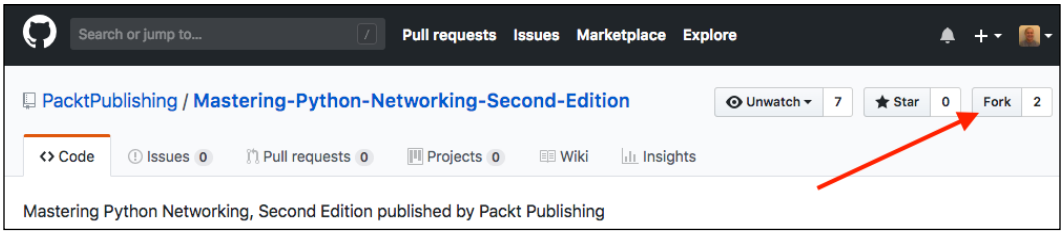


Figure 5: Git Fork button

It will take a few seconds to make a copy:

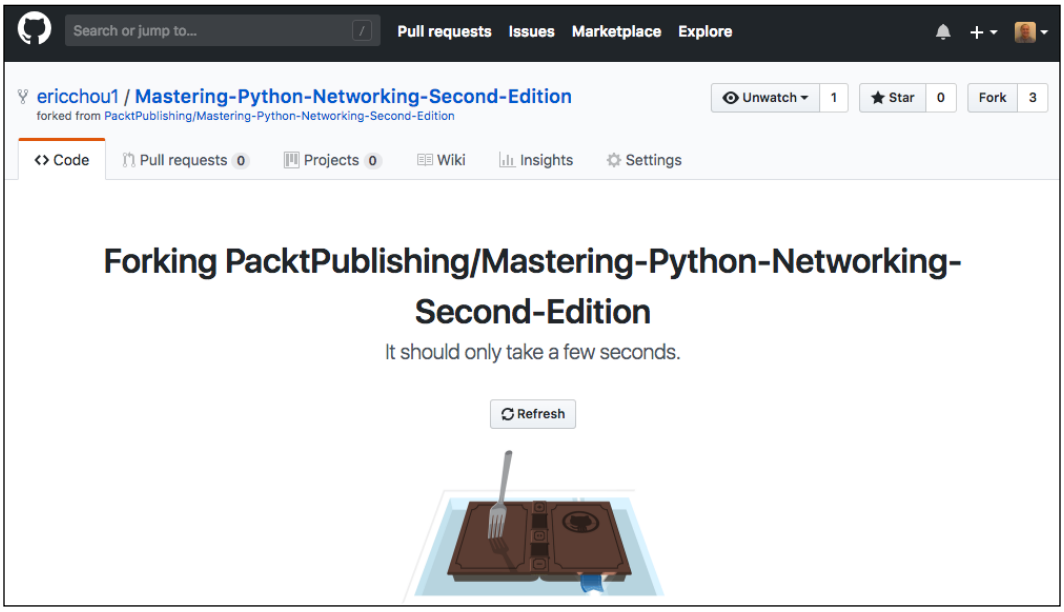


Figure 6: Git Fork in progress

After it is forked, we will have a copy of the repository in our personal account:

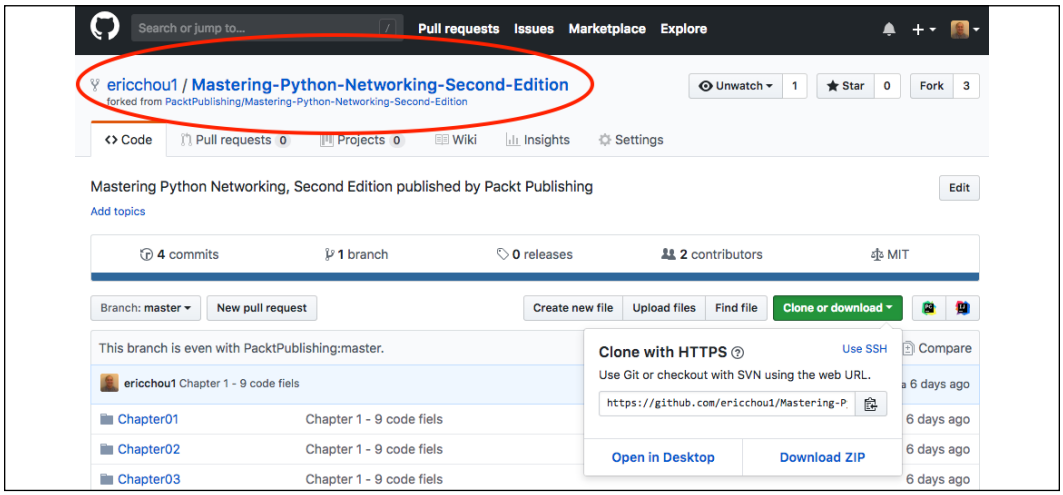


Figure 7: Git Fork

We can follow the same steps we have used before to make some modifications to 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:

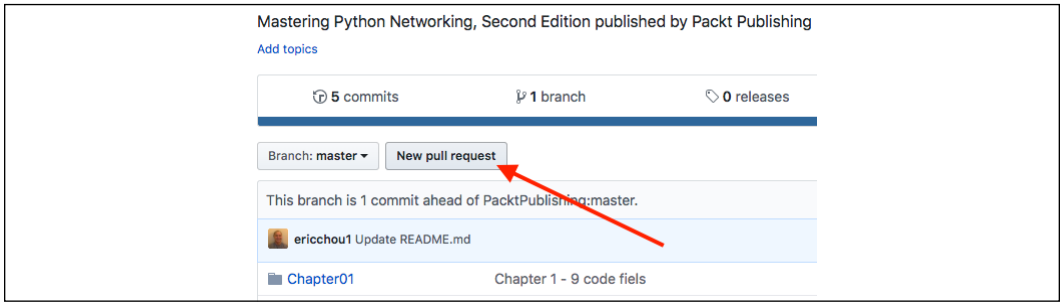


Figure 8: Pull request

When making a pull request, we should fill in as much information as possible to provide justifications for making the change:

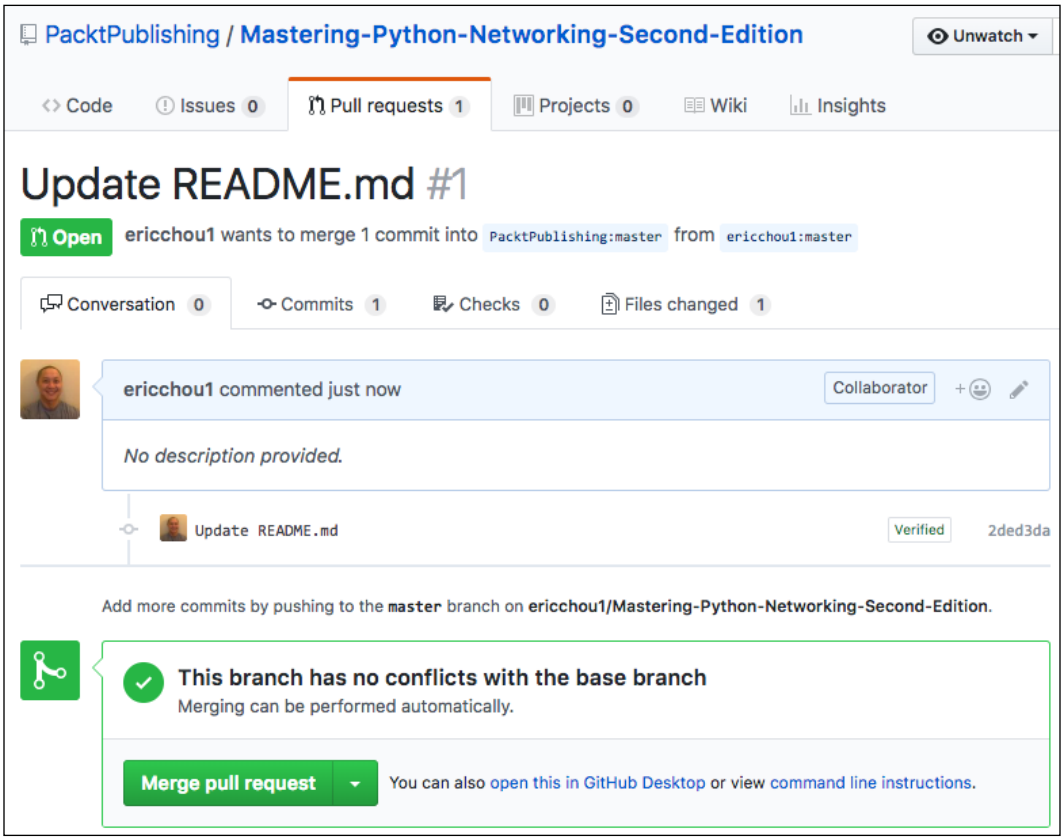


Figure 9: Pull request details

The repository maintainer will receive a notification of the pull request; if accepted, the change will make its way to the original repository:

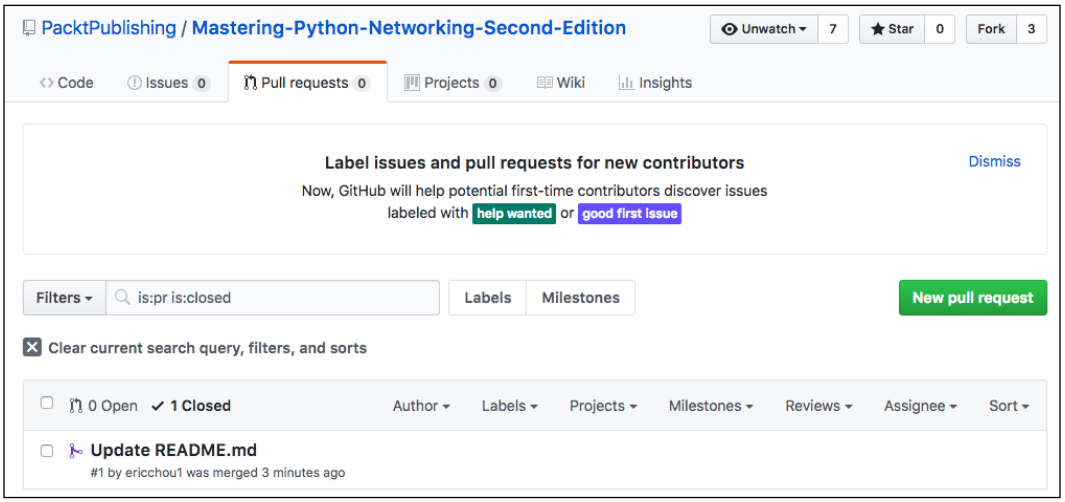


Figure 10: Pull request record

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 take a 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 take a 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:

```
(venv) $ pip install gitpython
(venv) $ 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. But it is always good to keep it in mind in case 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 repositories. The package is a wrapper around GitHub APIv3, `https://developer.github.com/v3/`:

```
(venv) $ pip install PyGithub
```

Let's use the Python shell to print out the user's current repository:

```
(venv) $ 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:

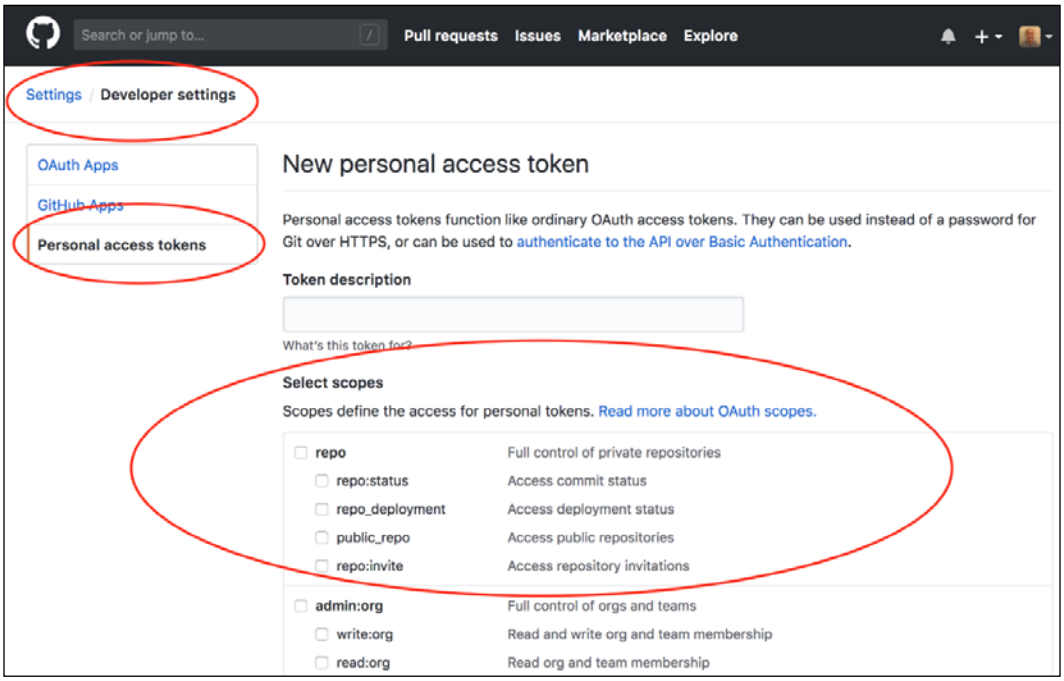


Figure 11: GitHub token generation

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 of the Python packages, we can use them to work with the technology. We will take a look at some practical examples in the coming 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, `Chapter13_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:

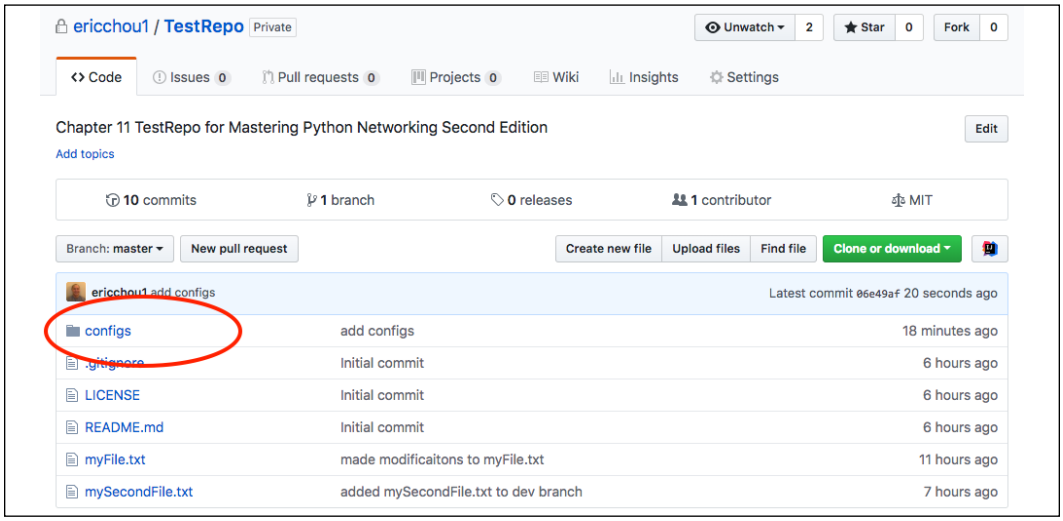


Figure 12: Configs directory

The commit history shows the commit from our script:

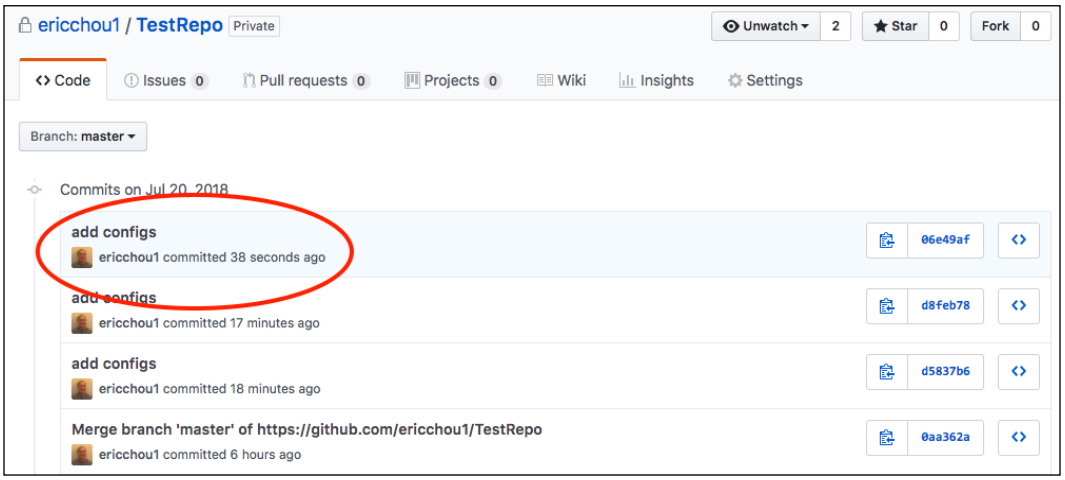


Figure 13: Commit history

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 of the 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 Python developers. 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; they also have a development guide that is 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 have 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 to. We should create a separate branch for our contribution and allow the branch to be merged at a later date.
- **Keep the forked repository synchronized**: Once you have forked a project, there is no rule that 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 make sure 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 we find interesting.

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 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 allow 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 of the 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 14*, *Continuous Integration with Jenkins*, we will look at another popular open source tool used for continuous integration and deployment: Jenkins.

14
Continuous Integration with Jenkins

The network touches every part of the technology stack; in all of 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 at times. 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 open source tool that can optimize the network-management pipeline, called Jenkins.

In this chapter, we'll cover the following topics:

- Challenges with the traditional change management process
- An introduction to continuous integration and Jenkins
- Jenkins installation and examples

- Jenkins 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 a lot of manual labor and human judgment. As we will see, they are not consistent and are difficult to streamline.

The traditional change management process

For engineers who have worked in a large network environment, they 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 have a negative impact on 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 in the process. More importantly, the reputation of the relatively young service took a serious hit. IT decision-makers pointed to the outage as reasons to `NOT` 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**) 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 of the steps.
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.

An 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 the 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 any 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 their 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 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 Jenkins on an Ubuntu 18.04 system.

Installing Jenkins

For the examples we will use in this chapter, we can install Jenkins on the management host or a separate machine. As indicated in previous chapters, my personal preference is to install it on a separate virtual machine. The virtual machine will have a similar network set up as the management host up to this point, with one interface for the internet connection and another interface for the VMNet2 connection to the VIRL management network.

The Jenkins image and installation instructions per operating system can be found at `https://jenkins.io/download/` and `https://jenkins.io/doc/book/installing/`.

The following are the instructions I used to install Jenkins on the Ubuntu host. Jenkins does not require a lot of hardware power; a single vCPU and 2 GB of RAM is what I have used in the lab. It also requires either Java 8 or 11 to be installed. We will use OpenJDK-11 for our server:

```
$ sudo apt install openjdk-11-jre-headless
$ java --version
openjdk 11.0.4 2019-07-16
OpenJDK Runtime Environment (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3)
OpenJDK 64-Bit Server VM (build 11.0.4+11-post-Ubuntu-1ubuntu218.04.3,
mixed mode, sharing)


$ wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-
key add -
$ sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /
etc/apt/sources.list.d/jenkins.list'
$ sudo apt-get update
$ sudo apt-get install Jenkins
$ sudo /etc/init.d/jenkins start
Correct java version found
[ ok ] Starting jenkins (via systemctl): jenkins.service.
```

Jenkins is a bit picky when it comes to Java version dependencies. As of the writing of this chapter, it only supports Java 8 and 11. Java versions 9, 10, and 12 are not supported (`https://jenkins.io/doc/administration/requirements/java/`).

Once Jenkins is installed, we can point the browser to the IP at port `8080` to continue the process:

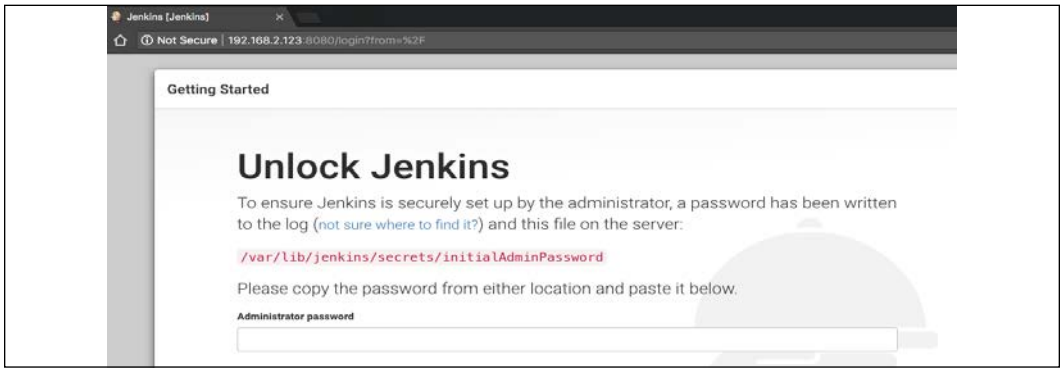


Figure 1: Unlock Jenkins screen

As stated on the screen, get the admin password from `/var/lib/jenkins/secrets/initialAdminPassword` and paste the output on the screen:

```
$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword

<one time admin password>
```

Once the password is pasted in, the next screen will ask us how to set up Jenkins. We will choose the **Install suggested plugins** option:

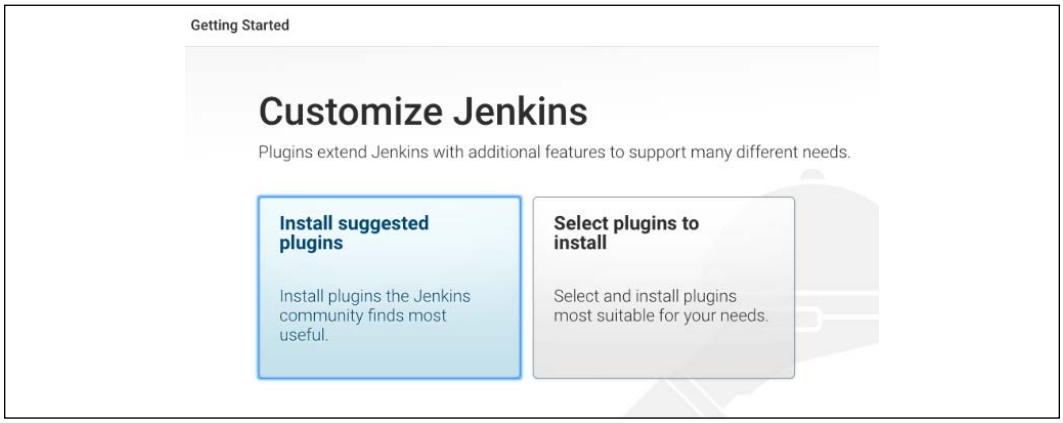


Figure 2: Install suggested plugins

You will be redirected to create the admin user; once created, Jenkins will be ready. If you see the Jenkins dashboard, the installation was successful:

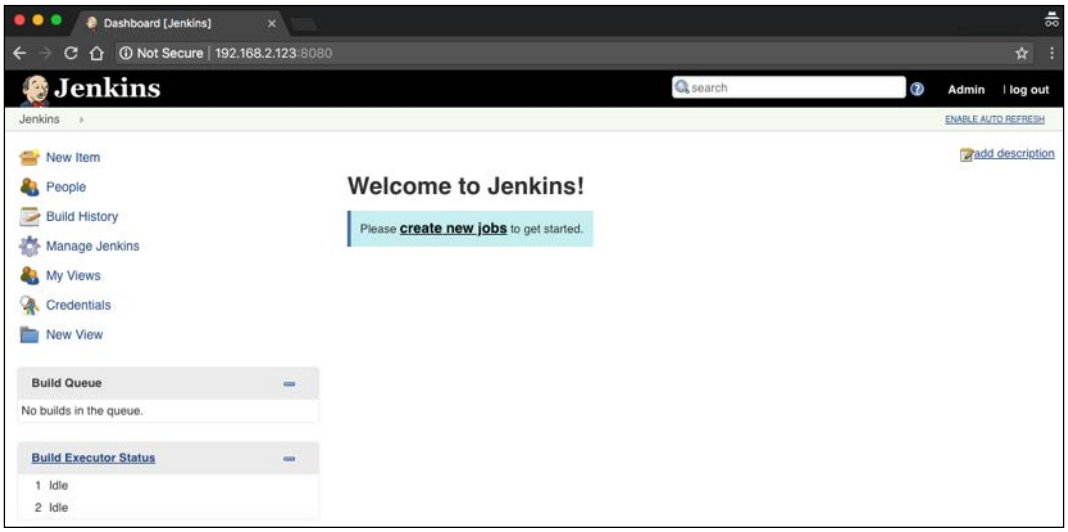


Figure 3: Jenkins dashboard

We are now ready to use Jenkins to schedule our first job.

Jenkins example

In this section, we will take a look at a few Jenkins examples and how they tie into the various technologies we have covered in this book. The reason Jenkins is covered in one of the last chapters of this book is because it leverages many of the other tools we have covered, such as our Python script, Ansible, Git, and GitHub. Feel free to refer back to any of the previous chapters for different topics if needed.

In the examples, we will use the Jenkins master to execute our jobs. In production, it is recommended to add Jenkins agent nodes to handle the execution of jobs.

For our lab, we will use a simple two-node topology with IOSv devices:

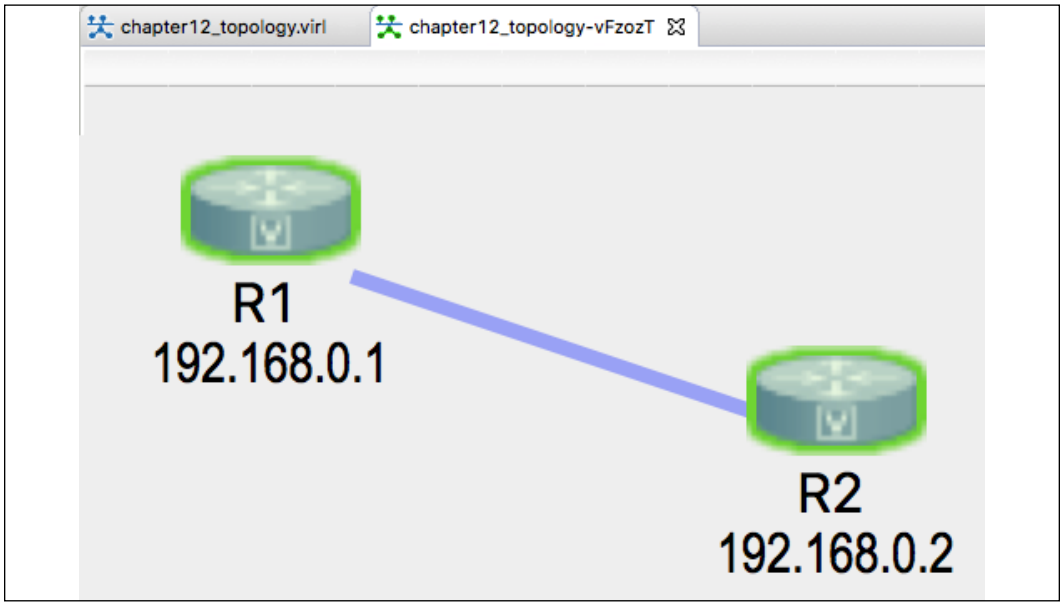


Figure 4: Lab topology

Let's build our first job.

The first job for the Python script

For our first job, let's use the `Paramiko` script that we built in *Chapter 2*, *Low-Level Network Device Interactions*, `chapter2_3.py`. If you recall, this is a script that uses `Paramiko` to `SSH` to remote devices and grabs the show run and show version output of the devices. Before we create a Jenkins job, we should always check to make sure the scripts work as expected on the machine first:

```
$ ls chapter14_1.py
chapter14_1.py
$ python3 chapter14_1.py
$ ls ios*
iosv-1_output.txt  iosv-2_output.txt
```

We will use the **create new item** link to create the job and pick the **Freestyle project** option:

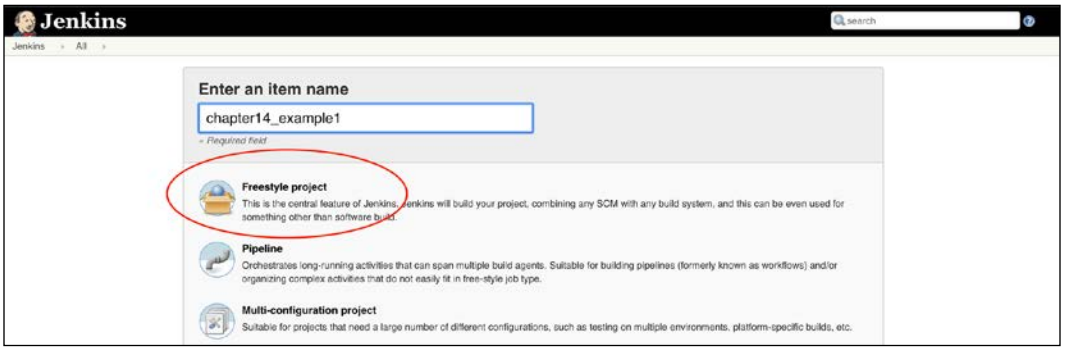


Figure 5: Jenkins item

We will enter our own description and leave everything as default and unchecked. Scroll down the page and select **Execute shell** as the build option:

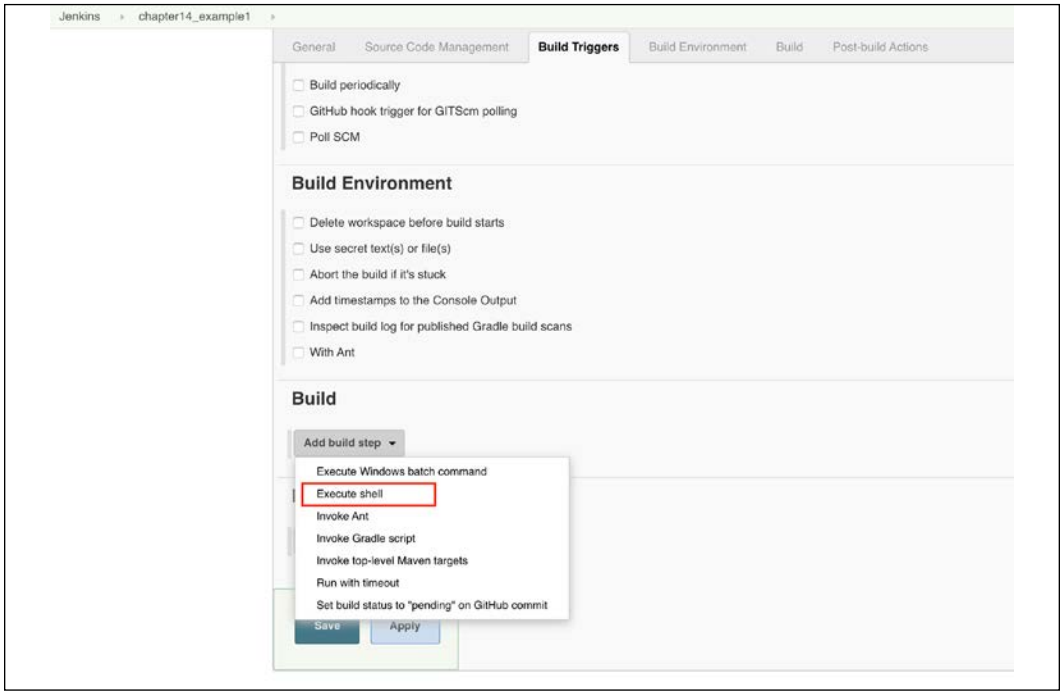


Figure 6: Jenkins build triggers

When the prompt appears, we will enter in the exact commands we use in the shell:

Figure 7: Jenkins execute shell

Once we save the job configuration, we will be redirected to the project dashboard. We can choose the **Build Now** option, and the job will appear under **Build History**:

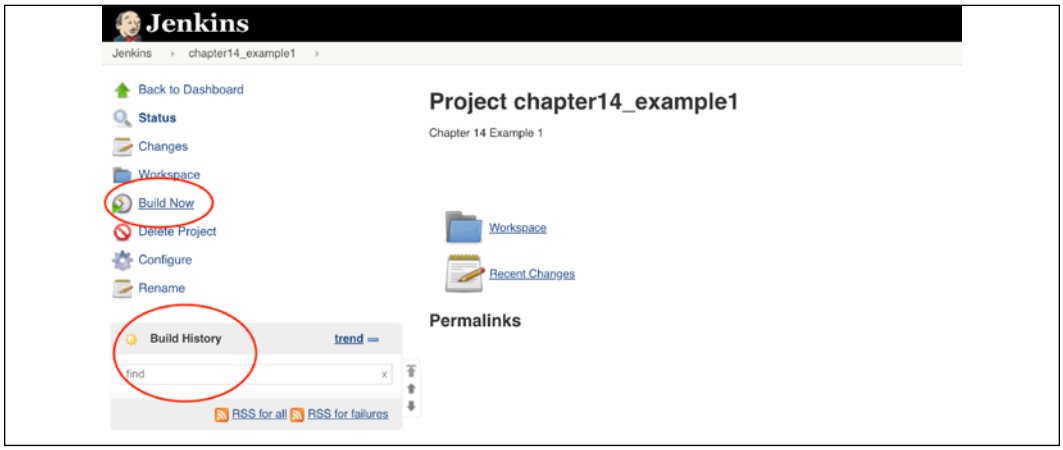


Figure 8: Jenkins build history

You can check the status of the build by clicking on it and choosing the **Console Output** on the left-hand panel:

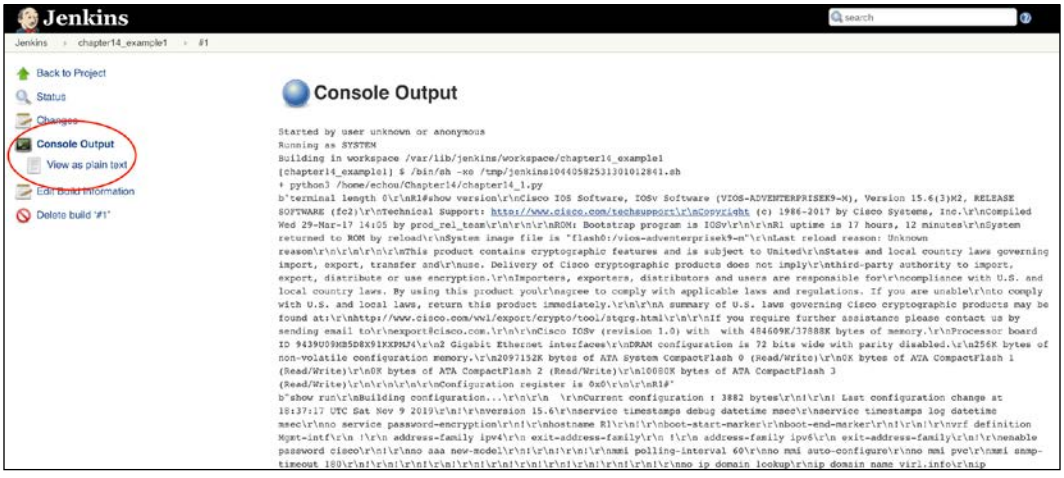


Figure 9: Jenkins console output

As an optional step, we can schedule this job at regular intervals, much like `cron` would do for us. We can navigate back to the **job** menu and choose **configure**. The job can be scheduled under **Build Triggers**. Choose to **Build periodically** and enter the `cron`-like schedule. In this example, the script will run at 02:00 and 20:00 each day:

Figure 10: Build trigger

We can also configure the SMTP server on Jenkins to allow notification of the build results. First, we will need to configure the SMTP server settings under **Manage Jenkins | Configure System** from the main menu:

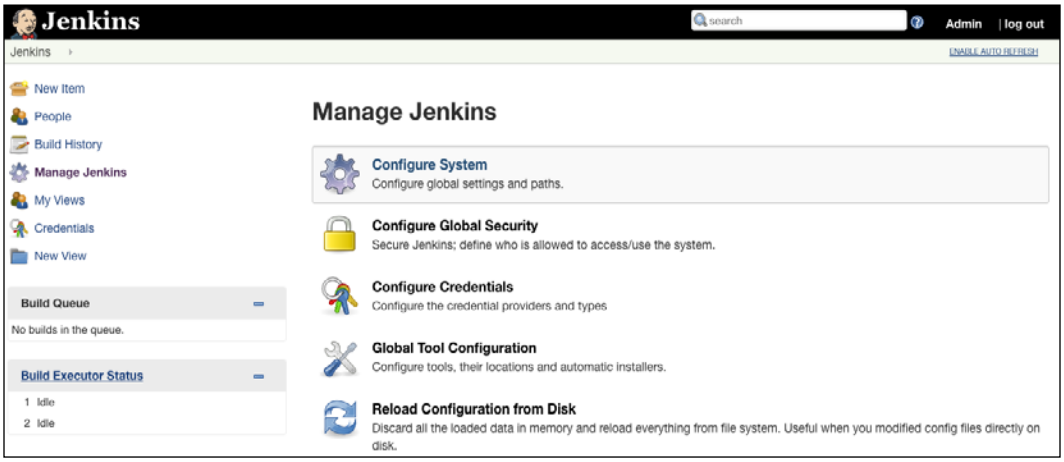


Figure 11: Configure System

We will see the SMTP server settings toward the bottom of the page. Click on **Advanced settings** to configure the SMTP server settings, as well as to send out a test email:

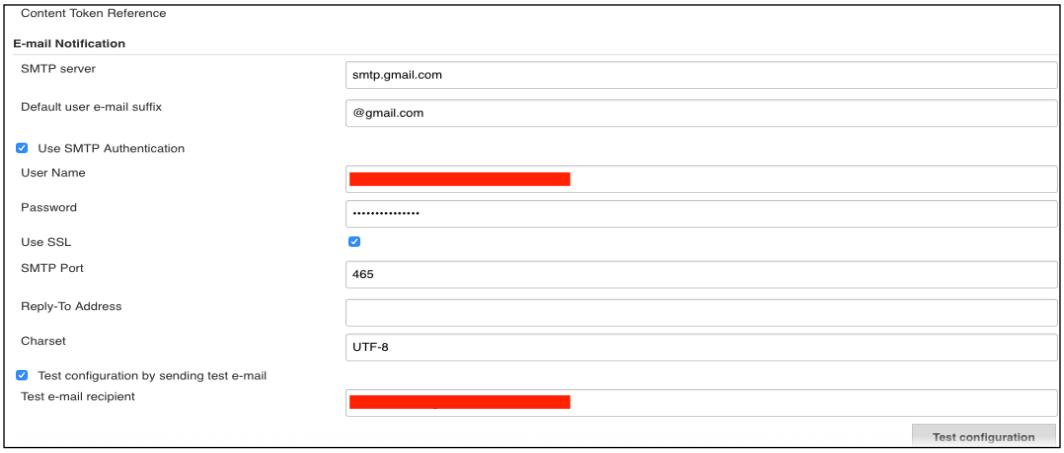


Figure 12: Configure SMTP

We will be able to configure email notifications as part of the post-build actions for our job:

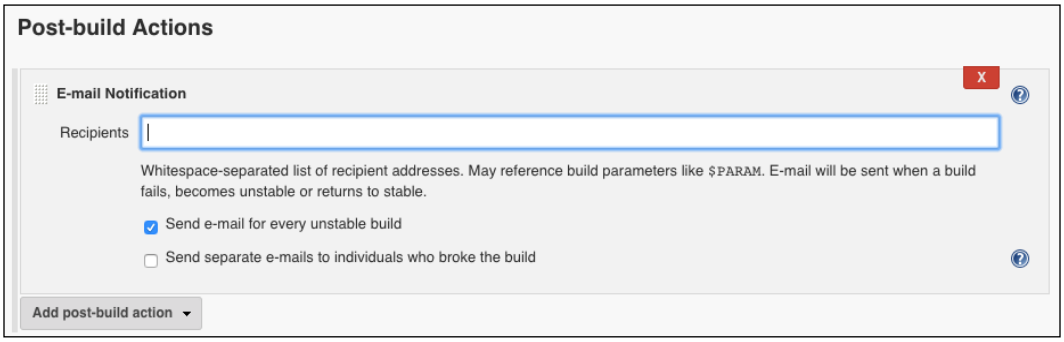


Figure 13: Email notification

Congratulations! We have just used Jenkins to create our first job. Functionally, this has not done anything more than what we could have achieved with our management host.

However, there are several advantages of using Jenkins:

- We can utilize Jenkins' various database-authentication integrations, such as LDAP, to allow existing users to execute our script.
- We can use Jenkins' role-based authorization to limit users. For example, some users can only execute jobs without modification access while others can have full administrative access.
- Jenkins provides a web-based graphical interface that allows users to access scripts easily.
- We can use the Jenkins email and logging services to centralize our jobs and be notified of the results.

Jenkins is a great tool by itself. Just like Python, it has a big third-party plugin ecosystem that can be used to expand its features and functionalities. We'll take a look at this ecosystem in the following section.

Jenkins plugins

We will install a simple schedule plugin as an example illustrating the plugin-installation process. The plugins are managed under **Manage Jenkins** | **Manage Plugins**:

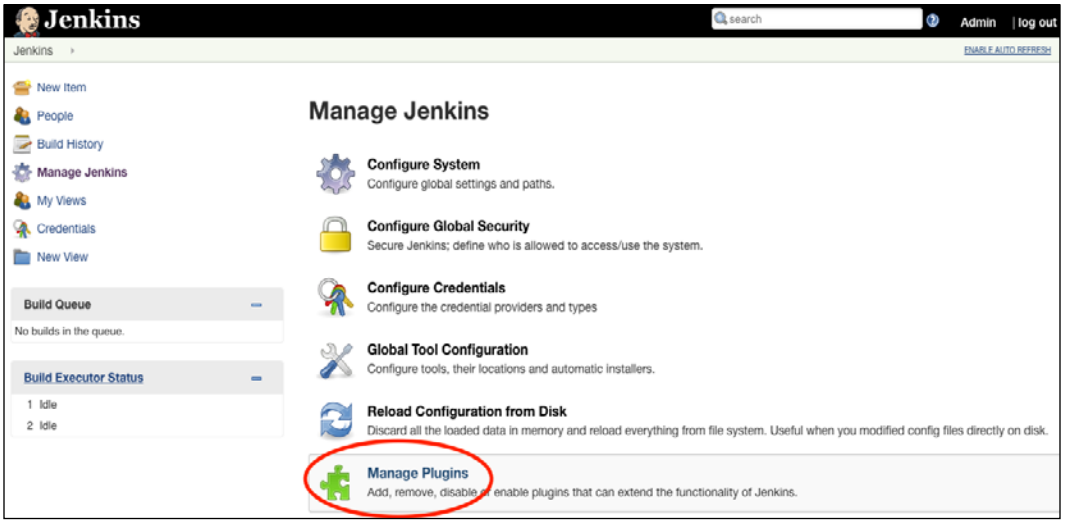


Figure 14: Jenkins plugin

We can use the search function to look for the **Schedule Build** plugin under the **Available** tab:

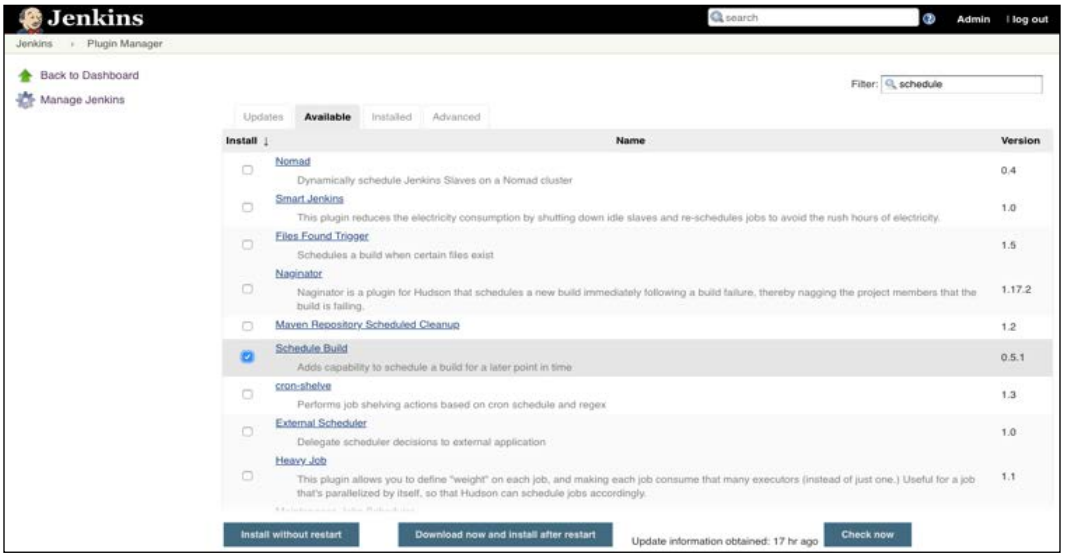


Figure 15: Jenkins plugin search

From there, we will just click on **Install without restart**, and we will be able to check the installation progress on the following page:

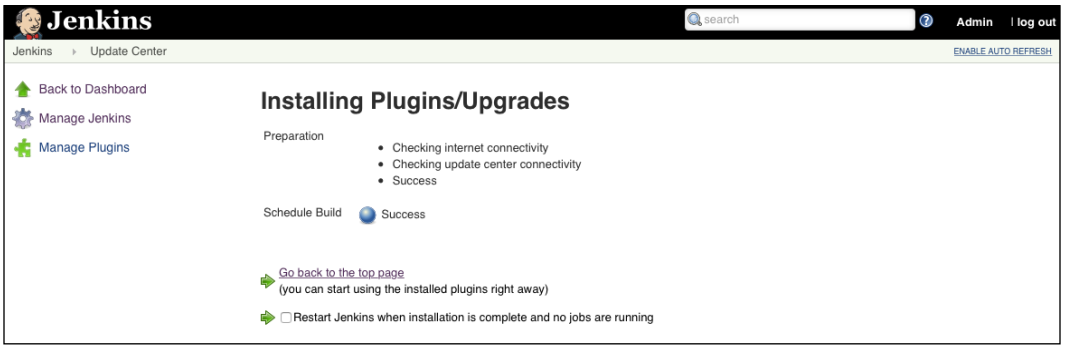


Figure 16: Jenkins plugin installation

After the installation is completed, we will be able to see a new icon that allows us to schedule jobs more intuitively:

Figure 17: Jenkins plugin result

It is one of the strengths of a popular open source project to have the ability to grow over time. For Jenkins, plugins provide a way to customize the tool for different customer needs. In the coming section, we will look at how to integrate version control and the approval process into our workflow.

Network continuous integration example

In this section, let's integrate our GitHub repository with Jenkins. By integrating the GitHub repository, we can take advantage of the GitHub code review and collaboration tools.

First, we will create a new GitHub repository. I will call this repository `chapter14_example2`. We can clone this repository locally and add the files we want to the repository. In this case, I am adding an Ansible playbook that copies the output of the `show version` command to a file:

```
---
- name: show version
  hosts: "ios-devices"
  gather_facts: false
  connection: local

  vars:
    cli:
      host: "{{ ansible_host }}"
      username: "{{ ansible_user }}"
      password: "{{ ansible_password }}"

  tasks:
    - name: show version
      ios_command:
        commands: show version
        provider: "{{ cli }}"

      register: output

    - name: show output
      debug:
```

```
            var: output.stdout

        - name: copy output to file
          copy: content="{{ output }}" dest=./output/{{ inventory_hostname
    }}.txt
```

By now, we should be pretty familiar with running an Ansible playbook. I will skip the output of `host_vars` and the inventory file. However, the most important thing is to verify that it runs on the local machine before committing to the GitHub repository:

```
$ ansible-playbook -i hosts chapter14_playbook.yml

PLAY [show version]
*****************************************************************

TASK [show version]
*****************************************************************
ok: [iosv-1]
ok: [iosv-2]
...
TASK [copy output to file]
*****************************************************
changed: [iosv-1]
changed: [iosv-2]

PLAY RECAP
*********************************************************************
iosv-1 : ok=3 changed=1 unreachable=0 failed=0
iosv-2 : ok=3 changed=1 unreachable=0 failed=0
```

We can now push the playbook and associated files to our GitHub repository:

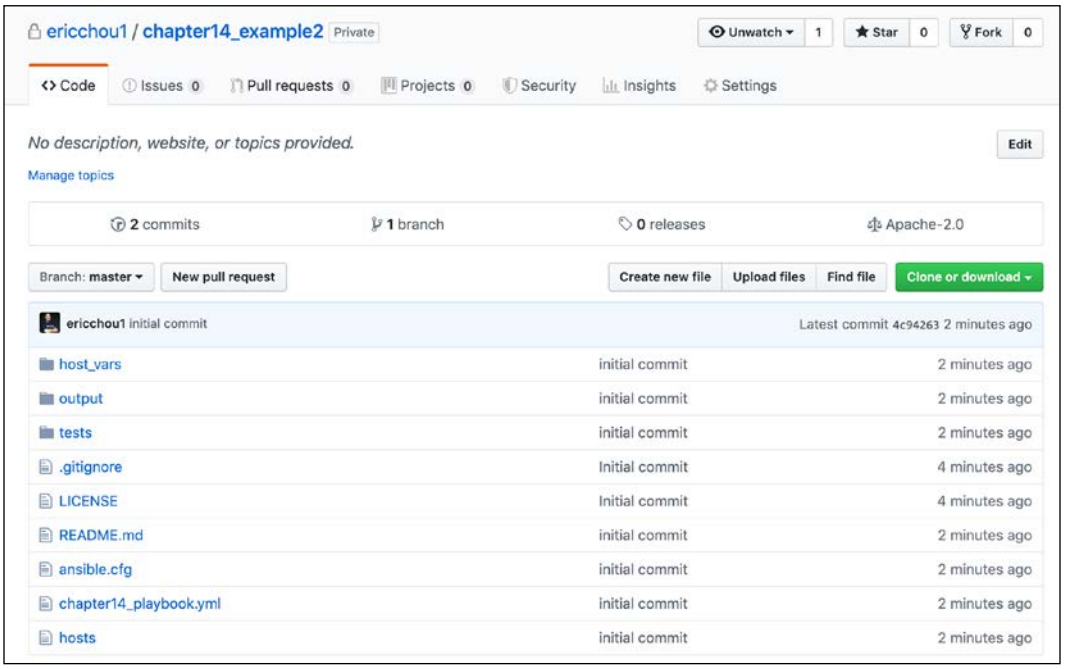


Figure 18: Jenkins GitHub example repository

Let's log back into the Jenkins host to install Git and Ansible:

```
$ sudo apt-get install software-properties-common
$ sudo apt-get update
$ sudo apt-get install ansible
$ sudo apt-get install git
```

As a point of reference, some of the tools can be installed under **Global Tool Configuration**; Git is one of them. However, since we are installing Ansible, we can install Git in the same command prompt:

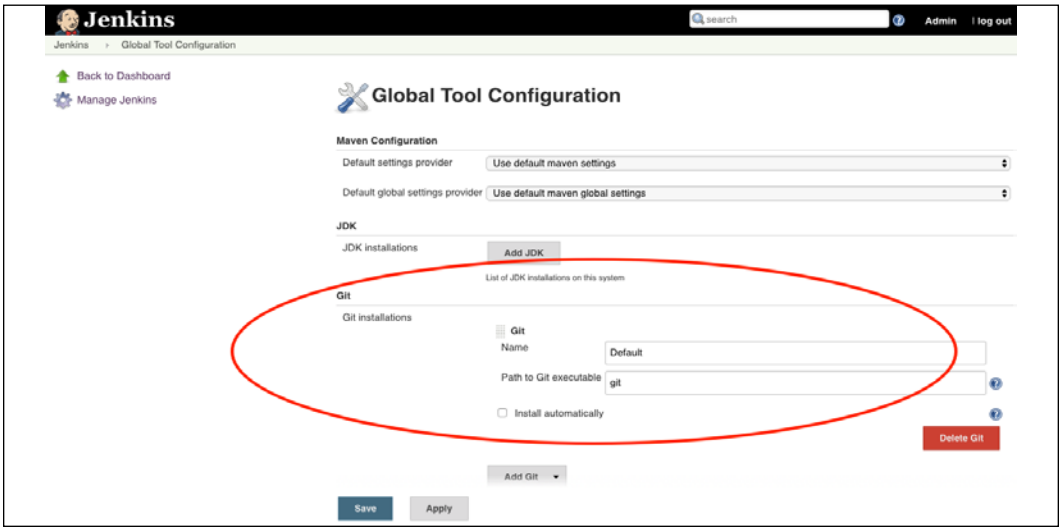


Figure 19: Global tools configuration

We can create a new freestyle project named `chapter14_example2`. Under **Source Code Management**, we will specify the GitHub repository as the source:

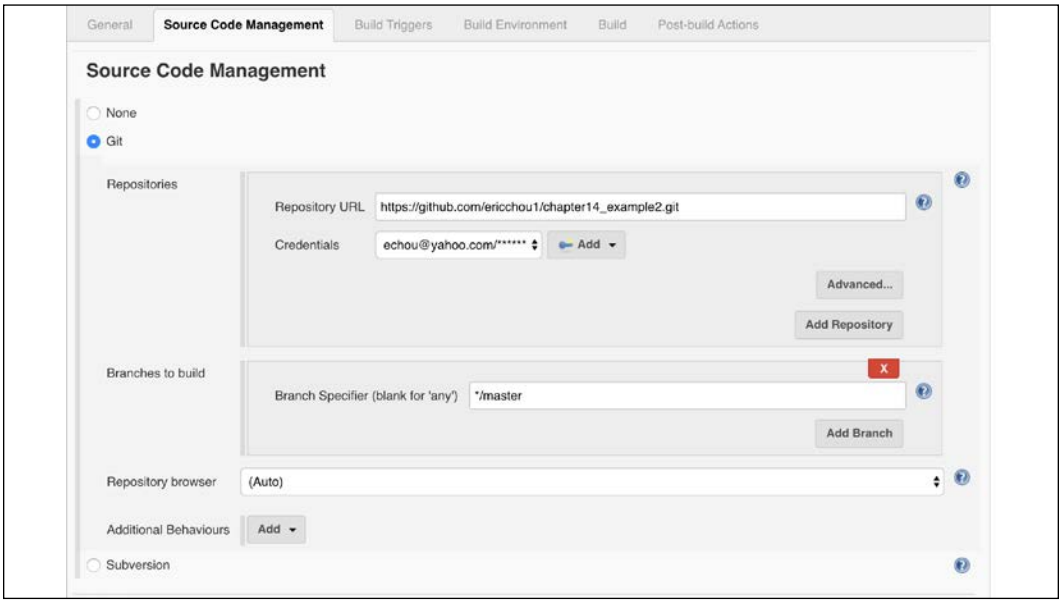


Figure 20: Jenkins source code management

Before we move on to the next step, let's save the project and run a build. In the build **Console Output**, we should be able to see the repository being cloned and the index value should match what we see on GitHub:

Figure 21: Another look at the Jenkins console output

We can now add the Ansible playbook command in the **Build** section:

Figure 22: Jenkins execute shell

If we run the build again, we can see from the console output that Jenkins will fetch the code from GitHub before executing the Ansible playbook:

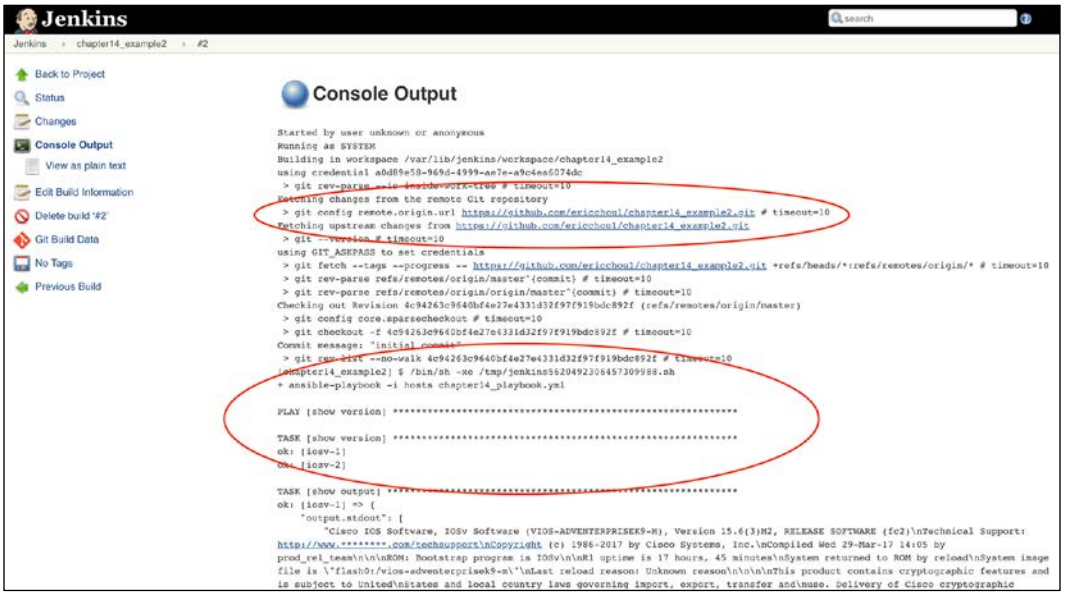


Figure 23: Further exploring the Jenkins console output

One of the benefits of integrating GitHub with Jenkins is that we can see all the Git information on the same screen:

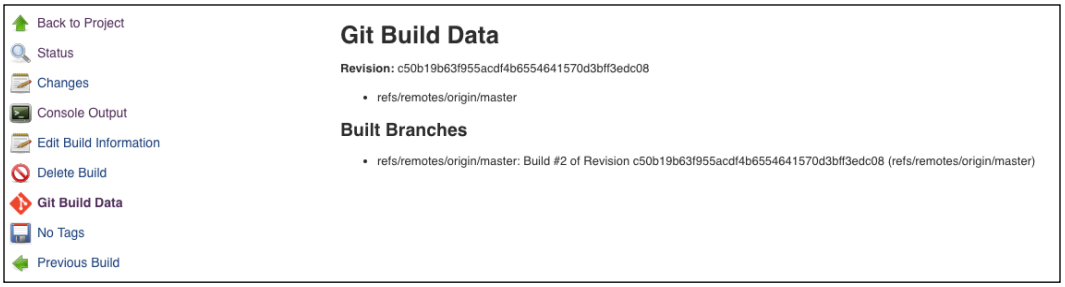


Figure 24: Git build data

The results of the project, such as the output of the Ansible playbook, can be seen in the **Workspace** folder:

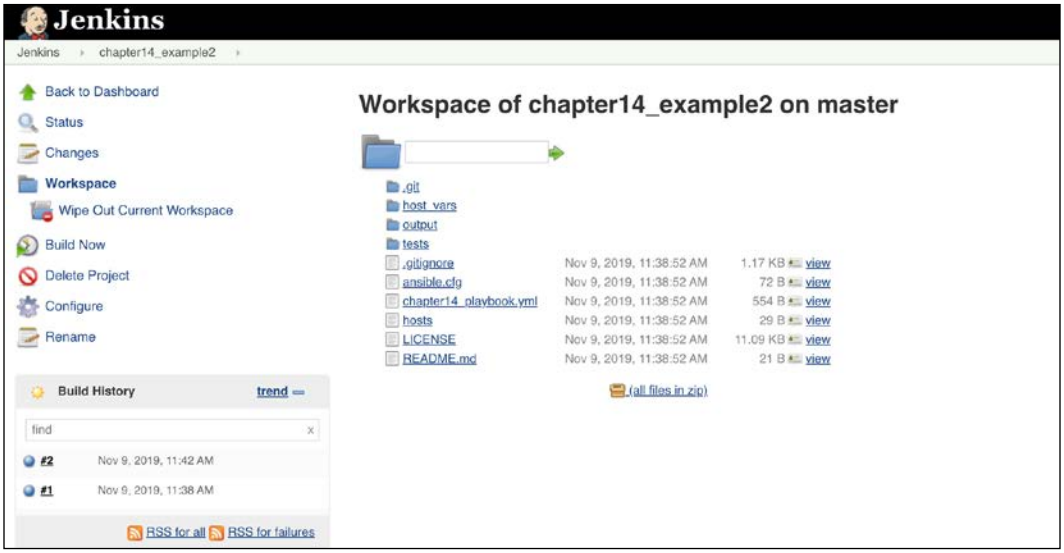


Figure 25: Jenkins workspace

At this point, we can follow the same step as before to use **periodic build** as the build trigger. If the Jenkins host is publicly accessible, we can also use GitHub's Jenkins plugin to notify Jenkins as a trigger for the build. This is a two-step process. The first step would be to enable webhooks on the GitHub repository:

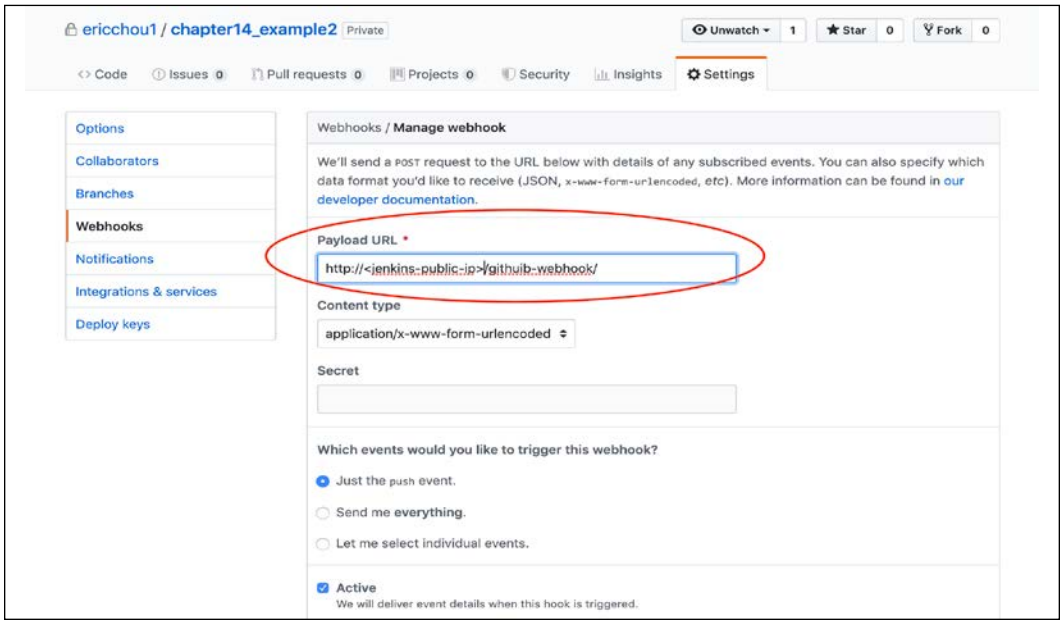


Figure 26: GitHub webhooks

The second step would be to install the necessary plugins and enable GitHub as a build trigger for our project. We should already have the Git plugin installed. Go ahead and install the GitHub plugin if not:

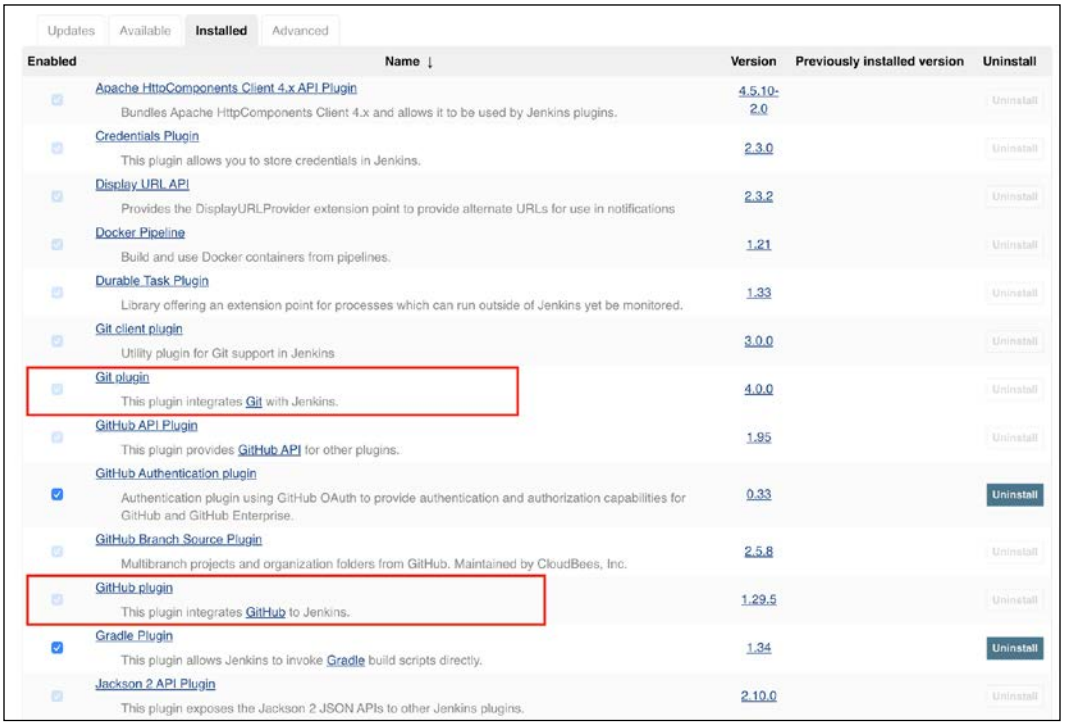


Figure 27: Jenkins Git and GitHub plugins

The GitHub plugin provides bi-directional integration with GitHub which allows a service hook that will hit the Jenkins instance every time a change is pushed to GitHub. We will enable the GitHub hook as the build trigger for our project:

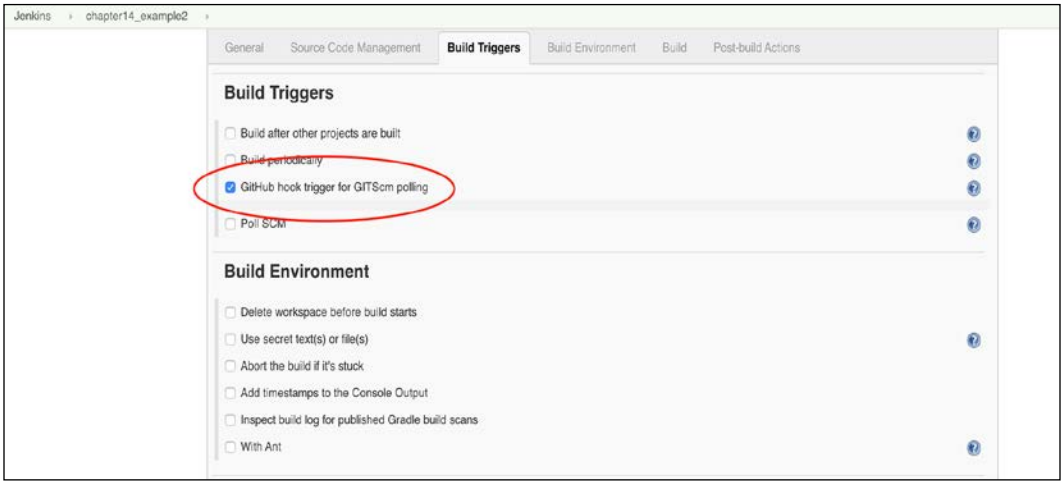


Figure 28: Jenkins GitHub hook trigger

Having the GitHub repository as the source allows for a brand-new set of possibilities of treating infrastructure as code. We can now use GitHub's fork, pull requests, issue tracking, and project management tools to work together efficiently. Once the code is ready, Jenkins can automatically pull the code down and execute it on our behalf.

You will notice we did not mention anything about automated testing. We will go over testing in *Chapter 15*, *Test-Driven Development for Networks*.

Jenkins is a full-featured system that can become complex. We have just scratched the surface of it with the two examples presented in this chapter. The Jenkins pipeline, environmental setup, multi-branch pipeline, and so on are all useful features that can accommodate the most complex automation projects. Hopefully, this chapter will serve as an interesting introduction for you to further explore the Jenkins tool.

So far in this chapter, we have been working with Jenkins using the web interface. In the next section, we will see how we can use Python libraries to work with Jenkins.

Jenkins with Python

Jenkins provides a full set of RESTful APIs for its functionalities: `https://wiki.jenkins.io/display/JENKINS/Remote+access+API`. There are also a number of Python wrappers that make interaction even easier. Let's take a look at the `python-jenkins` package:

```
(venv) $ pip install python-jenkins
```

We can test out the package with the following interactive prompt shell:

```
>>> import jenkins
>>> server = jenkins.Jenkins('http://192.168.2.124:8080',
username='<user>', password='<pass>')
>>> user = server.get_whoami()
>>> version = server.get_version()
>>> print('Hello %s from Jenkins %s' % (user['fullName'], version))
Hello Admin from Jenkins 2.121.2
```

We can work with the management of the server, such as `plugins`:

```
>>> plugin = server.get_plugins_info()
>>> plugin
[{'active': True, 'backupVersion': None, 'bundled': False, 'deleted':
False, 'dependencies': [{'optional': False, 'shortName': 'workflow-scm-
step', 'version': '2.9'}, {'optional': False, 'shortName': 'workflow-
step-api', 'version': '2.20'}, {'optional': False, 'shortName':
'credentials', 'version': '2.3.0'}, {'optional': False, 'shortName':
'git-client', 'version': '3.0.0'}, {'optional': False, 'shortName':
'mailer', 'version': '1.23'}, {'optional': False, 'shortName': 'scm-api',
<skip>
```

We can also manage the Jenkins jobs:

```
>>> job = server.get_job_config('chapter14_example1')
>>> import pprint
>>> pprint.pprint(job)
("<?xml version='1.1' encoding='UTF-8'?>\n"
 '<project>\n'
 '  <actions/>\n'
```

```
 '  <description>Chapter 14 Example 1</description>\n'
 '  <keepDependencies>false</keepDependencies>\n'
 '  <properties/>\n'
 '  <scm class="hudson.scm.NullSCM"/>\n'
 '  <canRoam>true</canRoam>\n'
 '  <disabled>false</disabled>\n'
<skip>
```

Using Python-Jenkins allows us to have a way to interact with Jenkins in a programmatic way. In the next section, let's discuss using continuous integration and Jenkins in our network engineering workflow.

Continuous integration for networking

Continuous integration has been adopted in the software development world for a while, but it is relatively new to network engineering. We are admittedly a bit behind in terms of using continuous integration in our network infrastructure. It is no doubt a bit of a challenge to think of our network in terms of code when we are still struggling to figure out how to stop using the CLI to manage our devices.

There are a number of good examples of using Jenkins for network automation. One is by Tim Fairweather and Shea Stewart at the AnsibleFest 2017 Network Track: `https://www.ansible.com/ansible-for-networks-beyond-static-config-templates`. Another use case was shared by Carlos Vicente from Dyn at NANOG 63: `https://www.nanog.org/sites/default/files/monday_general_autobuild_vicente_63.28.pdf`.

Even though continuous integration might be an advanced topic for network engineers who are just beginning to learn coding and the toolsets, in my opinion, it is worth the effort to start learning and using continuous integration in production today. Even at a basic level, the experience will trigger more innovative ways for network automation that will no doubt help the industry move forward.

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 Jenkins system. Jenkins is a full-featured, expandable, continuous integration system that is widely used in software development. We installed and used Jenkins to execute our Python script based on `Paramiko` at periodic intervals with email notifications. We also saw how we can install plugins for Jenkins to expand its features.

We looked at how we can use Jenkins to integrate with our GitHub repository and trigger builds based on code-checking. By integrating Jenkins with GitHub, we can utilize the GitHub process of collaboration.

In *Chapter 15*, *Test-Driven Development for Networks*, we will look at test-driven development with Python.

15
Test-Driven Development for Networks

In the previous chapters, we were able to use 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 undergoing changes 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 test-driven development
- Topology as code
- Writing tests for networking
- `pytest` integration with Jenkins
- 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 typically credited with leading the TDD movement, along with agile software development. Agile software development requires very short build-test-deploy development cycles; all of the 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 a 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 as close to a production environment as possible.

What I am trying to get at is, even without realizing, we might already be adopting some of the TDD methodology 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 of the components 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. Personally, 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 components of the network are built. As a matter of 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 when it comes 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 style of problem-solving.

Before we delve further into TDD, let's cover some of the most common terminology 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. Keep in mind 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 tell you we have been using code to describe our topology in this book already?

If you take a look at any of the VIRL topology files that we have been using in this book, they are simply XML files that include a description of the relationship between nodes. For example, in this chapter, we will use the following topology for our lab:

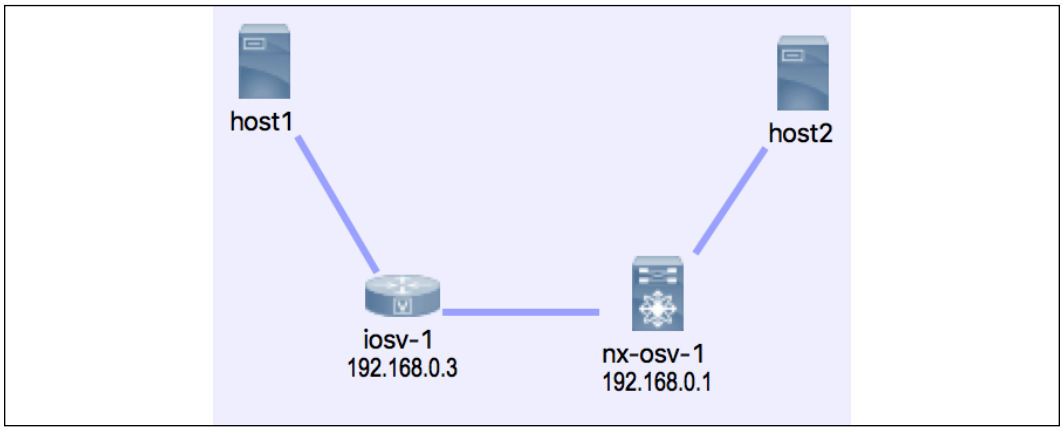


Figure 1: The topology graph for our lab

If we open up the topology file, `chapter15_topology.virl`, with a text editor, we will see that the file is an XML file describing the node and the relationship between the nodes. At the top, or root, level is the `<topology>` node with child nodes of `<node>`. Each of the child nodes consists of various extensions and entries:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<topology xmlns="http://www.cisco.com/VIRL" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" schemaVersion="0.95"
xsi:schemaLocation="http://www.cisco.com/VIRL https://raw.github.com/
CiscoVIRL/schema/v0.95/virl.xsd">
<extensions>
<entry key="management_network" type="String">flat</entry>
</extensions>
```

The child node attributes are embedded with attributes such as name, type, and location. We can also see the configuration of each node in the text value of the `<entry key="config">` element:

```
<node name="iosv-1" type="SIMPLE" subtype="IOSv" location="182,162"
ipv4="192.168.0.3">
<extensions>
<entry key="static_ip" type="String">172.16.1.20</entry>
<entry key="config" type="string">
! IOS Config generated on 2018-07-24 00:23
! by autonetkit_0.24.0
!
hostname iosv-1
boot-start-marker
boot-end-marker
!
...
</node>
<node name="nx-osv-1" type="SIMPLE" subtype="NX-OSv"
location="281,161" ipv4="192.168.0.1">
    <extensions>
        <entry key="static_ip" type="String">172.16.1.21</entry>
        <entry key="config" type="string">! NX-OSv Config generated on
2018-07-24 00:23
! by autonetkit_0.24.0
!
version 6.2(1)
license grace-period
!
hostname nx-osv-1
```

Even though the node is a host, we can also represent it in an XML element in the same file:

```
...
<node name="host2" type="SIMPLE" subtype="server" location="347,66">
    <extensions>
         <entry key="static_ip" type="String">172.16.1.23</entry>
         <entry key="config" type="string">#cloud-config

bootcmd:
ln -s -t /etc/rc.d /etc/rc.local
hostname: host2
manage_etc_hosts: true
runcmd:
start ttyS0
```

```
systemctl start getty@ttyS0.service
systemctl start rc-local
<annotations/>
<connection dst="/virl:topology/virl:node[1]/virl:interface[1]" src="/
virl:topology/virl:node[3]/virl:interface[1]"/>
<connection dst="/virl:topology/virl:node[2]/virl:interface[1]" src="/
virl:topology/virl:node[1]/virl:interface[2]"/>
<connection dst="/virl:topology/virl:node[4]/virl:interface[1]" src="/
virl:topology/virl:node[2]/virl:interface[2]"/>
</topology>
```

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.

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 `chapter15_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 chapter15_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 wanted 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 `chapter15_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:

```
(venv) $ python chapter15_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:

```
(venv) $ python chapter15_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 was returned as failed; the reason for 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 result. 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 or not. 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, `chapter15_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:

```
(venv) $ python -m unittest chapter15_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:

```
(venv) $ python -m unittest -v chapter15_5_more_unittest.py
testAssertFalse (chapter15_5_more_unittest.Output) ... ok
testAssesrtTrue (chapter15_5_more_unittest.Output) ... ok
testError (chapter15_5_more_unittest.Output) ... ERROR
testFail (chapter15_5_more_unittest.Output) ... FAIL
testPass (chapter15_5_more_unittest.Output) ... ok

======================================================================
ERROR: testError (chapter15_5_more_unittest.Output)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_third_edition/Chapter15/
chapter15_5_more_unittest.py", line 14, in testError
    raise RuntimeError('Test error!')
```

```
RuntimeError: Test error!


======================================================================
FAIL: testFail (chapter15_5_more_unittest.Output)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/echou/Mastering_Python_Networking_third_edition/Chapter15/
chapter15_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 the example of 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 `chapter15_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?

```
(venv) $ python chapter15_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:

```
(venv) $ pip install pytest
(venv) $ python
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pytest
>>> pytest.__version__
'5.2.2'
```

Let's look at some examples using `pytest`.

pytest examples

The first `pytest` example, `chapter15_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 the reason for the failure. The verbose output is one of the reasons people like `pytest`:

```
(venv) $ pytest -v chapter15_6_pytest_1.py
================================== test session starts ================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0 --
/home/echou/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 2 items


chapter15_6_pytest_1.py::test_passing PASSED
[ 50%]
chapter15_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         - (1, 2, 3)
```

```
E         ?  ^     ^
E         + (3, 2, 1)
E         ?  ^     ^


chapter15_6_pytest_1.py:7: AssertionError
============================== 1 failed, 1 passed in 0.03s
==============================
```

In the second `pytest` example, `chapter15_7_pytest_2.py`, we will create a `router` object. The `router` object will be initiated with some values in `None` and some values 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:

```
(venv) $ pytest chapter15_7_pytest_2.py
================================== test session starts ================
==================
```

```
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 2 items


chapter15_7_pytest_2.py ..
[100%]


================================== 2 passed in 0.01s ===================
==================
```

If we were to replace the previous `unittest` example with `pytest`, in `chapter15_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:

```
(venv) $ pytest chapter15_8_pytest_3.py
================================== test session starts =================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item


chapter15_8_pytest_3.py .
[100%]


================================== 1 passed in 0.09s ===================
==================
```

Between `unittest` and `pytest`, I find `pytest` more intuitive to use. However, since `unittest` is included in the standard library, many teams might have a preference for 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 take a 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.

A few years ago, Matt Oswalt announced the **Testing On Demand**: **Distributed** (**ToDD**) validation tool for network changes. It is an open source framework aimed at testing network connectivity and distributed capacity. You can find more information about the project on its GitHub page: `https://github.com/toddproject/todd`. Oswalt also talked about the project on this Packet Pushers Priority Queue 81, Network Testing with ToDD: `https://packetpushers.net/podcast/podcasts/pq-show- 81-network-testing-todd/`.

In this section, let's look at how we can write tests that are relevant to the networking world. There is no shortage of commercial products when it comes to 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 a network engineer, 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 so that we can compare the values over a time series of data.

This can sometimes be difficult to do 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 just 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 `chapter15_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`:

```
(venv) $ python chapter15_10_ping.py
[(b'www.cisco.com(2001:559:19:289b::b33', b'time=16.0'), (b'www.google.
com(sea15s11-in-x04.1e100.net', b'time=11.4')]
```

This is by no means 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 offers the flexibility that comes with constructing our packets.

Another great tool for network security testing is `hping3` (`http://www.hping.org/`). 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 are part of Google), but we can implement a layer-seven 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 quickly test layer-seven reachability on a web service. We already saw how to use it when we were performing network monitoring in *Chapter 5*, *The Python Automation Framework – Beyond Basics*, but it's worth seeing again:

```
# Python 3
(venv) $ 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/2.5/dev_guide/testing.html`
- **Unit tests**: `https://docs.ansible.com/ansible/2.5/dev_guide/testing_units.html`
- **Unit testing Ansible modules**: `https://docs.ansible.com/ansible/2.5/dev_guide/testing_units_modules.html`

One of the interesting Ansible testing frameworks is Molecule (`https://pypi.org/ project/molecule/2.16.0/`). It intends to aid in the development and testing of Ansible roles. Molecule provides support for 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. Can we integrate testing with a source control tool such as Jenkins? The answer is yes. We will take a look at how to do so in the next section.

pytest Integration with Jenkins

Continuous Integration (**CI**) systems, such as Jenkins, are frequently used to launch tests after each of the code commits. This is one of the major benefits of using a CI system.

Imagine that there is an invisible engineer who is always watching for any changes in the network; upon detecting a change, the engineer will faithfully test a bunch of functions to make sure that nothing breaks. Who wouldn't want that?

Let's look at an example of integrating `pytest` into Jenkins tasks.

Jenkins integration

Before we can insert the test cases into our CI, let's install some of the plugins that can help us visualize the operation. The two plugins we will install are **build-name-setter** and **Test Results Analyzer**:

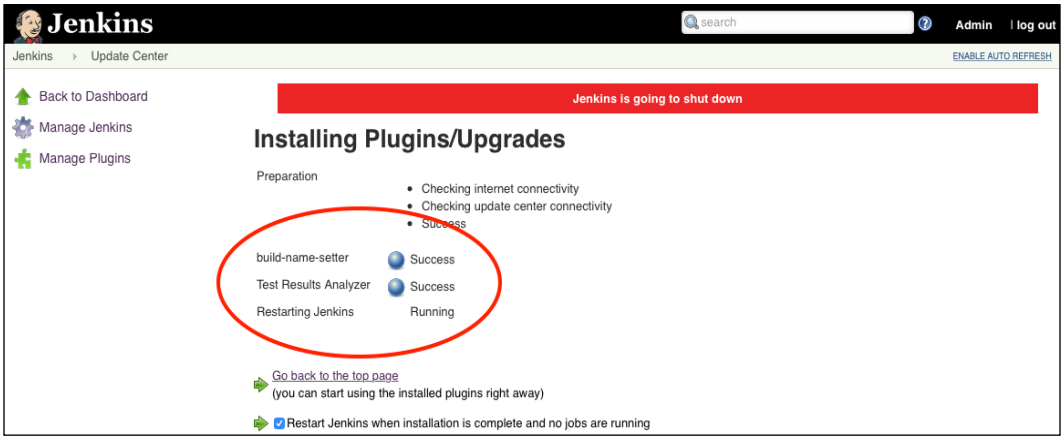


Figure 2: Jenkins plugin installation

The test we will run will reach out to the NX-OS device and retrieve the operating system version number. This will ensure that we have API reachability to the Nexus device. The full script content can be read in `chapter15_9_pytest_4.py`. The relevant `pytest` portion and result are as follows:

```
def test_transaction():
assert nxos_version != False
```

```
(venv) $ pytest chapter15_9_pytest_4.py
================================== test session starts ==================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item

chapter15_9_pytest_4.py .
[100%]

=================================== 1 passed in 0.10s ===================
==================
```

We will use the `--junit-xml=results.xml` option to produce the file Jenkins needs:

```
(venv) $ pytest --junit-xml=result.xml chapter15_9_pytest_4.py
================================== test session starts ==================
==================
platform linux -- Python 3.6.8, pytest-5.2.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/echou/Mastering_Python_Networking_third_edition/Chapter15
collected 1 item

chapter15_9_pytest_4.py .
[100%]

- generated xml file: /home/echou/Mastering_Python_Networking_third_
edition/Chapter15/result.xml -
=================================== 1 passed in 0.10s ===================
==================
```

The next step is to check this script into the GitHub repository. I prefer to put the test in its own directory. Therefore, I created a **/tests** directory and put the test file there:

Figure 3: Project repository

We will create a new project named `chapter15_example1`:

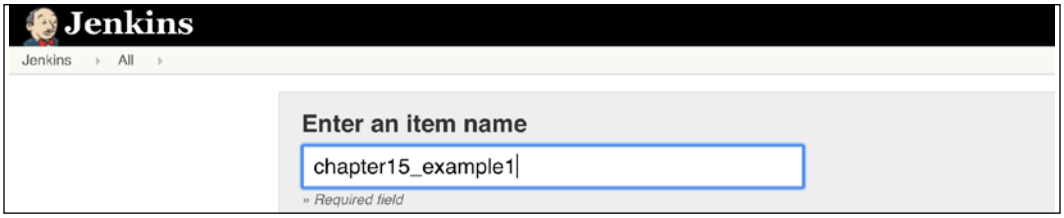


Figure 4: Name your project in Jenkins

We can copy over the previous task, so we do not need to repeat all the steps:

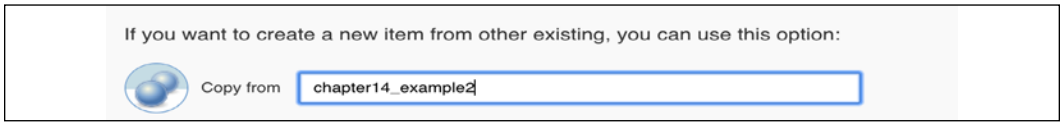


Figure 5: Use the copy from function in Jenkins

In the execute shell section, we will add the `pytest` step:

Figure 6: The execute shell

We will add a post-build step of **Publish JUnit test result report**:

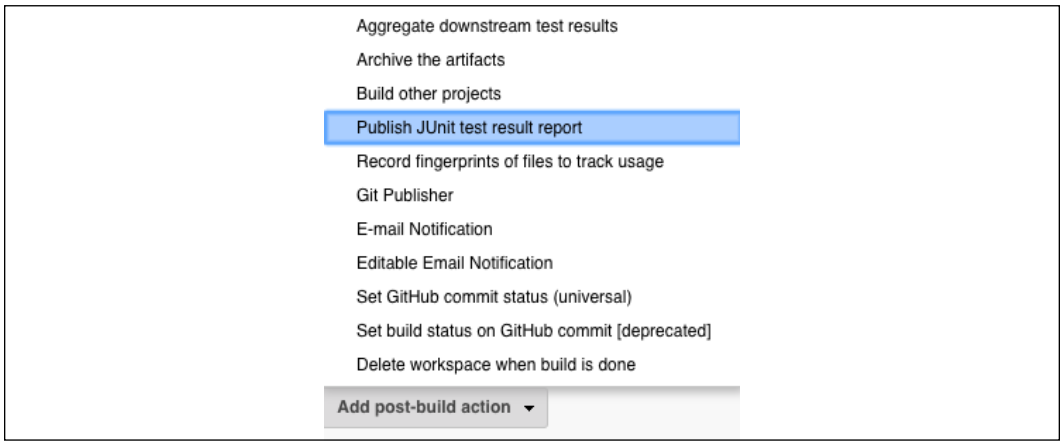


Figure 7: Post-build step

We will specify the **results.xml** file as the JUnit result file:

Figure 8: Test report XML location

After we run the build a few times, we will be able to see the **Test Results Analyzer** graph:

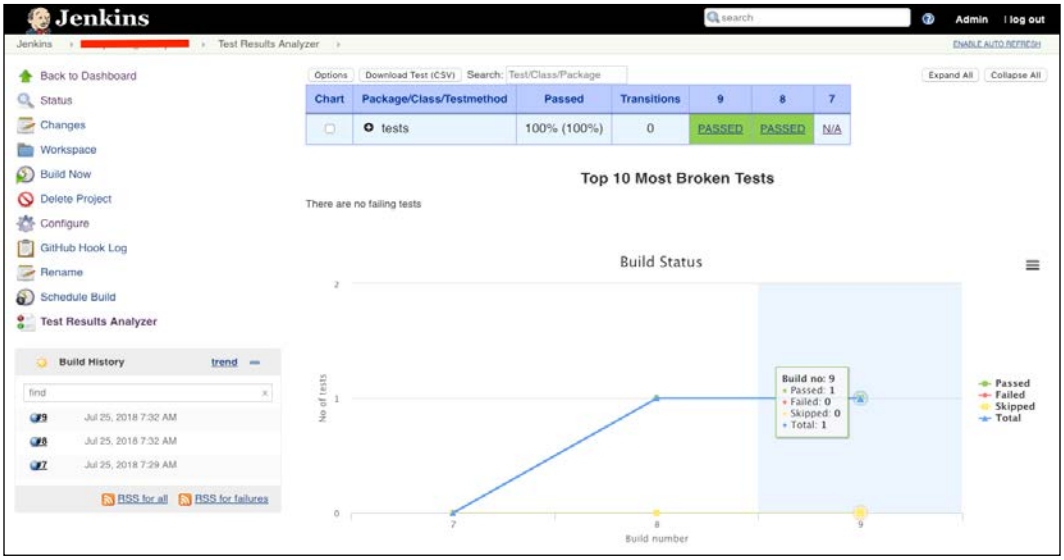


Figure 9: Test Results Analyzer graph in Jenkins

The test result can also be seen on the project home page. Let's introduce a test failure by shutting down the management interface of the Nexus device. If there is a test failure, we will be able to see it right away on the **Test Result Trend** graph on the project dashboard:

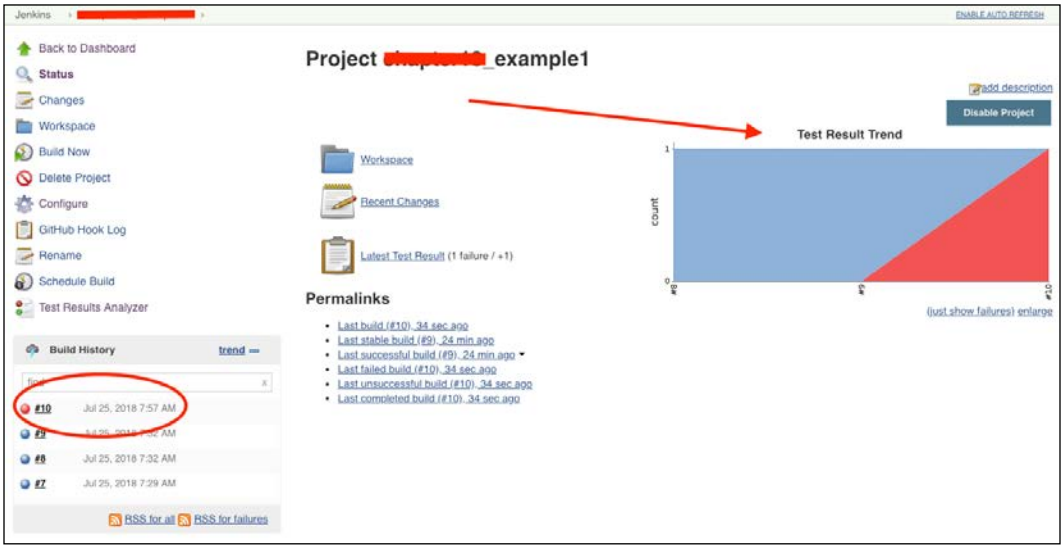


Figure 10: Test Result Trend graph in Jenkins

This is a simple but complete example. We can use the same pattern to build other integrated tests in Jenkins.

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 open source, end-to-end testing ecosystem originally developed by Cisco and made available to public in late 2017. The pyATS library was formerly known as Genie; many times they will 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:

```
(venv) echou@network-dev-2:~$ pip install pyats
```

To get started, we can take a 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 `chapter15_pyats_testbed_1.yml` testbed file for our `iovs-1` device. The file should look similar to the Ansible inventory file that we have seen before:

```
testbed:
    name: Chapter_15_pyATS
    tacacs:
      username: cisco
    passwords:
      tacacs: cisco
      enable: cisco

devices:
   iosv-1:
       alias: iosv-1
       type: ios
       connections:
```

```
            defaults:
              class: unicon.Unicon
            management:
              ip: 172.16.1.20
              protocol: ssh

    topology:
        iosv-1:
            interfaces:
                GigabitEthernet0/2:
                    ipv4: 10.0.0.5/30
                    link: link-1
                    type: ethernet
                Loopback0:
                    ipv4: 192.168.0.3/32
                    link: iosv-1_Loopback0
                    type: loopback
```

In our first script, `chapter15_11_pyats_1.py`, we will load the testbed file, connect to the device, issue a `show version` command, then disconnect from the device:

```
from pyats.topology import loader

testbed = loader.load('chapter15_pyats_testbed_1.yml')

testbed.devices
ios_1 = testbed.devices['iosv-1']

ios_1.connect()

print(ios_1.execute('show version'))

ios_1.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:

```
(venv) $ python chapter15_11_pyats_1.py
[2019-11-10 08:11:55,901] +++ iosv-1 logfile /tmp/iosv-1-default-
20191110T081155900.log +++
[2019-11-10 08:11:55,901] +++ Unicon plugin generic +++
<skip>
[2019-11-10 08:11:56,249] +++ connection to spawn: ssh -l cisco
172.16.1.20, id: 140357742103464 +++
```

```
[2019-11-10 08:11:56,250] connection to iosv-1
[2019-11-10 08:11:56,314] +++ initializing handle +++
[2019-11-10 08:11:56,315] +++ iosv-1: executing command 'term length 0'
+++
term length 0
iosv-1#
[2019-11-10 08:11:56,354] +++ iosv-1: executing command 'term width 0'
+++
term width 0
iosv-1#
[2019-11-10 08:11:56,386] +++ iosv-1: executing command 'show version'
+++
show version
<skip>
```

In the second example, we will see a full example of connection setup, test cases, then connection teardown. First, we will add the `nxosv-1` device to our testbed in `chapter15_pyats_testbed_2.yml`. The additional device is needed as the connected device to iosv-1 for our ping test:

```
    nxosv-1:
        alias: nxosv-1
        type: ios
        connections:
          defaults:
            class: unicon.Unicon
          vty:
            ip: 172.16.1.21
            protocol: ssh
```

In `chapter15_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 = ('10.0.0.5', '10.0.0.6'))
    def ping(self, device, destination):
        try:
            result = self.parameters[device].ping(destination)
```

It is best practice to reference the testbed file at the command line during runtime:

```
(venv) $ python chapter15_12_pyats_2.py --testbed chapter15_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. The output also indicates the log filename that is written to the `/tmp` directory:

```
2019-11-10T08:23:08: %AETEST-INFO: Starting common setup
<skip>
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
--------------------+
2019-11-10T08:23:22: %AETEST-INFO: |                        STEPS Report
|
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
--------------------+
<skip>
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
2019-11-10T08:23:22: %AETEST-INFO: |
Detailed Results                                |
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
2019-11-10T08:23:22: %AETEST-INFO:  SECTIONS/TESTCASES  RESULT
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
2019-11-10T08:23:22: %AETEST-INFO: |                    Summary|
2019-11-10T08:23:22: %AETEST-INFO: +-----------------------------------
----------------------------------------+
<skip>
2019-11-10T08:23:22: %AETEST-INFO:  Number of PASSED
3
```

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.

Another open source tool for network validation is Batfish, `https://github.com/batfish/batfish`, from the folks at IntentionNet. A primary use case for Batfish is to validate configuration changes before deployment.

There is a bit of a learning curve involved with `pytest`; it basically has its own way of performing tests that takes some getting used to. Understandably so, it is also heavily focused on Cisco platforms in its current iteration. But since this is now an open source project, we are all encouraged to make contributions if we would like to add other 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 test-driven development and how it can be applied to network engineering. We started with an overview of TDD; then we looked at examples of using the `unittest` and `pytest` Python modules. Python and simple Linux command line tools can be used to construct various tests for network reachability, configuration, and security.

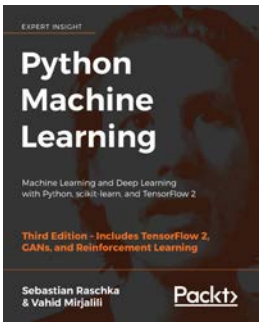
We also looked at how we can utilize testing in Jenkins, a CI tool. By integrating tests into our CI tool, we can gain more confidence in the sanity of our changes. At the very least, we hope to catch any errors before our users do. pyATS is an open source tool that Cisco recently released. It is a network-centric 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 just not possible.

We are at the end of the book, I hope you have found the book as a joy to read as it was a joy for me to write. I want to say a sincere 'Thank You' for spending time with this book. I wish you success and happiness on your Python network journey!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

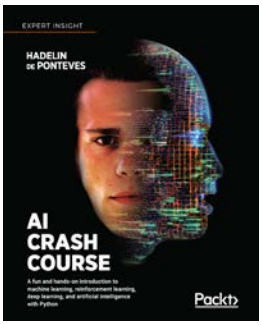


Python Machine Learning - Third Edition

Sebastian Raschka, Vahid Mirjalili

ISBN: 978-1-78995-575-0

- Master the frameworks, models, and techniques that enable machines to 'learn' from data
- Use scikit-learn for machine learning and TensorFlow for deep learning
- Apply machine learning to image classification, sentiment analysis, intelligent web applications, and more
- Build and train neural networks, GANs, and other models
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis



AI Crash Course

Hadelin de Ponteves

ISBN: 978-1-83864-535-9

- Master the key skills of deep learning, reinforcement learning, and deep reinforcement learning
- Understand Q-learning and deep Q-learning
- Learn from friendly, plain English explanations and practical activities
- Build fun projects, including a virtual-self-driving car
- Use AI to solve real-world business problems and win classic video games
- Build an intelligent, virtual robot warehouse worker

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y