

Zaawansowana inżynieria sieci w Pythonie

Automatyzacja, monitorowanie
i zarządzanie chmurą

Wydanie IV



<packt>

Eric Chou

Tytuł oryginału: Mastering Python Networking: Utilize Python packages and frameworks for network automation, monitoring, cloud, and management, 4th Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-0280-0

Copyright © Packt Publishing 2023. First published in the English language under the title 'Mastering Python Networking - Fourth Edition – (9781803234618)

Polish edition copyright © 2024 by Helion S.A.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/zains4>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/zains4.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści |

O autorze	13
O recenzencie	14
Przedmowa.....	15
Wstęp	17
ROZDZIAŁ 1	
Prezentacja zestawu protokołów TCP/IP i Pythona	23
Przedstawienie internetu	25
Serwery, hosty i komponenty sieci	26
Powstanie centrów danych	27
Model OSI	31
Model klient-serwer	34
Zestawy protokołów sieciowych	34
Protokół sterowania transmisją — TCP	35
Protokół pakietów użytkownika — UDP	36
Protokół internetowy — IP	37
Przegląd języka Python	39
System operacyjny	41
Uruchamianie programów w Pythonie	42
Wbudowane typy danych Pythona	43
Operatory w Pythonie	49
Narzędzia do sterowania przepływem	50
Funkcje w Pythonie	52
Klasy w Pythonie	52
Moduły i pakiety	53
Podsumowanie	55

ROZDZIAŁ 2**Niskopoziomowe interakcje z urządzeniami sieciowymi56**

Wyzwania stosowania CLI	57
Konstruowanie wirtualnego laboratorium	59
Urządzenia fizyczne	59
Urządzenia wirtualne	59
Laboratoria modelowania Cisco	61
Cisco DevNet	64
GNS3 i inne laboratoria	64
Wirtualne środowisko Pythona	66
Biblioteka Pexpect	67
Instalacja modułu Pexpect	67
Przegląd biblioteki Pexpect	68
Pierwszy program używający Pexpect	73
Więcej możliwości Pexpect	74
Pexpect i SSH	75
Kompletny przykład użycia biblioteki Pexpect	76
Biblioteka Paramiko	77
Instalacja biblioteki Paramiko	77
Prezentacja biblioteki Paramiko	77
Pierwszy program z użyciem Paramiko	80
Więcej możliwości biblioteki Paramiko	81
Więcej przykładów użycia biblioteki Paramiko	83
Biblioteka Netmiko	84
Framework Nornir	86
Wady Pexpect i Paramiko w porównaniu z innymi narzędziami	88
Podsumowanie	89

ROZDZIAŁ 3**API i sieci intuicyjne90**

Infrastruktura jako kod (IaC)	91
Sieci intuicyjne	92
Analiza wyświetlanych wyników a strukturalne wyniki API	93
Modelowanie danych na potrzeby IaC	96
YANG i NETCONF	98
Przykłady wykorzystania API urządzeń Cisco	98
Cisco NX-API	99
Model YANG firmy Cisco	104
Przykłady Cisco ACI	105
Kontroler Cisco Meraki	109
API Pythona dla urządzeń Juniper Networks	110
Juniper i NETCONF	111
Biblioteka Juniper PyEZ dla programistów	115

API Pythona dla urządzeń Arista	119
Zarządzanie przy użyciu Arista eAPI	120
Biblioteka Pyeapi firmy Arista	124
Przykład VyOS	128
Inne biblioteki	129
Podsumowanie	130

ROZDZIAŁ 4

Framework automatyzacyjny Pythona — Ansible	131
Ansible — framework bardziej deklaracyjny	132
Wersje Ansible	134
Pierwszy sieciowy przykład użycia Ansible	135
Instalacja Ansible na węźle kontrolnym	136
Topologia laboratorium	137
Zalety Ansible	141
Brak agentów	141
Idempotentność	142
Prostota i rozszerzalność	143
Ansible Content Collection	143
Kolejne sieciowe przykłady użycia Ansible	144
Zagnieżdżanie ewidencji	145
Wyrażenia warunkowe w Ansible	147
Zmiany konfiguracji	149
Fakty sieciowe Ansible	150
Pętle Ansible	152
Szablony	156
Podsumowanie	162

ROZDZIAŁ 5

Kontenery Dockera dla inżynierów sieciowych	163
Prezentacja Dockera	164
Zalety Dockera	165
Tworzenie aplikacji Pythona w Dockerze	166
Instalowanie Dockera	166
Przydatne polecenia Dockera	167
Budowanie aplikacji hello-world	168
Budowanie własnej aplikacji	169
Udostępnianie obrazów Dockera	172
Orkiestracja kontenerów przy użyciu docker-compose	175
Sieci w kontenerach	177
Sieć hosta kontenerów	178
Własne sieci mostkowe	179
Inne opcje sieciowe kontenerów	180

Kontenery w zastosowaniach inżynierii sieciowej	181
Containerlab	181
Docker i Kubernetes	185
Podsumowanie	186

ROZDZIAŁ 6

Bezpieczeństwo sieci w Pythonie 187

Przygotowanie laboratorium	188
Program Scapy	194
Instalowanie Scapy	194
Interaktywne przykłady korzystania ze Scapy	196
Przechwytywanie pakietów z użyciem Scapy	198
Skanowanie portów TCP	199
Zestaw narzędzi ping	203
Popularne ataki	204
Zasoby dotyczące Scapy	205
Listy dostępu	205
Implementacja list dostępu przy użyciu Ansible	206
Listy dostępu adresów MAC	210
Przeszukiwanie dzienników Syslog	211
Przeszukiwanie z użyciem modułu obsługi wyrażeń regularnych	212
Inne narzędzia	214
Prywatne wirtualne sieci lokalne	214
Obsługa UFW w Pythonie	215
Dalsza lektura	216
Podsumowanie	216

ROZDZIAŁ 7

Monitorowanie sieci przy użyciu Pythona — część 1. 218

Konfiguracja laboratorium	219
SNMP	221
Konfiguracja	222
PySNMP	225
Stosowanie Pythona do wizualizacji danych	230
Matplotlib	230
Pygal	238
Stosowanie Cacti	242
Instalacja	243
Skrypt Pythona jako źródło danych wejściowych	245
Podsumowanie	247

ROZDZIAŁ 8**Monitorowanie sieci przy użyciu Pythona — część 2.249**

Graphviz	250
Konfiguracja laboratorium	251
Instalacja	252
Przykłady Graphviz	253
Przykłady użycia Graphviza w Pythonie	255
Użycie LLDP do prezentowania sąsiadujących węzłów sieci	256
Monitorowanie oparte na przepływach	265
Parsewanie danych NetFlow z użyciem Pythona	266
Monitorowanie ruchu sieciowego przy użyciu ntop	270
Rozszerzanie ntop przy użyciu Pythona	273
sFlow	277
Podsumowanie	282

ROZDZIAŁ 9**Tworzenie sieciowych usług webowych przy użyciu Pythona283**

Porównanie frameworków webowych Pythona	285
Konfiguracja Flaska i laboratorium sieciowego	288
Wprowadzenie do Flaska	289
Wersje Flaska	289
Przykłady stosowania Flaska	290
Klient HTTPie	291
Obsługa adresów URL	293
Zmienne URL	294
Generowanie adresów URL	295
Zwracanie wyników w formacie JSON	296
API zasobu sieciowego	297
Flask-SQLAlchemy	297
API zawartości sieci	299
API urządzeń	302
API konkretnego urządzenia	304
Dynamiczne operacje sieciowe	305
Operacje asynchroniczne	308
Uwierzelnianie i autoryzacja	310
Uruchamianie Flaska w kontenerach	313
Podsumowanie	317

ROZDZIAŁ 10**Wprowadzenie do asynchronicznych operacji wejścia-wyjścia318**

Przegląd operacji asynchronicznych	319
Wieloprocusowość w Pythonie	320
Wielowątkowość w Pythonie	321

Moduł asyncio Pythona	323
Projekt Scrapli	327
Przykład zastosowania Scrapli	327
Asynchroniczny przykład korzystający ze Scrapli	329
Podsumowanie	332

ROZDZIAŁ 11

Sieci w chmurze AWS334

Konfiguracja AWS	335
AWS CLI i Python SDK	337
Przegląd zagadnień sieciowych w chmurze AWS	340
Wirtualna chmura prywatna	346
Tabele tras i cele tras	351
Automatyzacja przy użyciu CloudFormation	353
Grupy bezpieczeństwa i sieciowe listy ACL	356
Elastic IP	358
Bramy NAT	360
Direct Connect i VPN	361
Bramy VPN	361
Direct Connect	362
Usługi skalowania sieci	364
Elastic Load Balancing	364
Usługa DNS Route 53	365
Usługi CDN CloudFront	365
Inne sieciowe usługi AWS	366
Podsumowanie	366

ROZDZIAŁ 12

Sieci w chmurze Azure368

Porównanie usług sieciowych Azure i AWS	369
Konfiguracja Azure	371
Administracja i API Azure	372
Jednostki usług Azure	376
Python a PowerShell	379
Globalna infrastruktura Azure	380
Sieci wirtualne na platformie Azure	381
Dostęp do internetu	384
Tworzenie zasobów sieciowych	387
Punkty końcowe usługi VNet	389
Peering sieci VNet	390
Routing w sieciach wirtualnych	392
Grupy bezpieczeństwa sieci	397
Wirtualne sieci prywatne na platformie Azure	399

ExpressRoute	402
Równoważenie obciążenia w Azure	404
Inne usługi sieciowe Azure	405
Podsumowanie	406

ROZDZIAŁ 13

Analiza danych sieciowych z użyciem Elastic Stack 407

Czym jest Elastic Stack?	408
Topologia laboratorium	409
Elastic Stack jako usługa	415
Pierwszy kompleksowy przykład	417
Obsługa Elasticsearch przy użyciu klienta napisanego w Pythonie	422
Pozyskiwanie danych za pomocą Logstash	424
Pozyskiwanie danych za pomocą Beats	426
Wyszukiwanie przy użyciu Elasticsearch	432
Wizualizacja danych za pomocą Kibany	436
Podsumowanie	442

ROZDZIAŁ 14

Korzystanie z systemu Git 443

Rozważania dotyczące zarządzania treścią i system Git	444
Wprowadzenie do systemu Git	445
Zalety Gita	446
Terminologia Gita	447
Git i GitHub	448
Konfiguracja Gita	449
Plik gitignore	450
Przykłady stosowania systemu Git	451
Gałęzie	455
Przykład GitHub	457
Stosowanie Gita z Pythonem	465
Pakiet GitPython	465
Biblioteka PyGithub	466
Automatyzacja tworzenia kopii zapasowych konfiguracji	467
Współdziałanie przy użyciu Gita	470
Podsumowanie	471

ROZDZIAŁ 15

Ciągła integracja z użyciem GitLab 472

Tradycyjny proces zarządzania zmianą	473
Wprowadzenie do ciągłej integracji	474
Instalowanie GitLab 476	476

Runnery GitLaba	480
Pierwszy przykład stosowania GitLaba	481
Sieciowy przykład użycia GitLaba	489
Podsumowanie	491

ROZDZIAŁ 16

Programowanie w oparciu o testy

na potrzeby programowania sieciowego 493

Prezentacja programowania w oparciu o testy	494
Definicje testów	495
Topologia jako kod	496
Przykład parsowania kodu XML	499
Moduł unittest Pythona	501
Więcej o testowaniu w Pythonie	504
Przykłady stosowania modułu pytest	505
Pisanie testów na potrzeby sieci	507
Testowanie dostępności	508
Testowanie opóźnień sieci	509
Testowanie bezpieczeństwa	510
Testowanie transakcji	510
Testowanie konfiguracji sieciowych	511
Testowanie Ansible	511
pyATS i Genie	512
Podsumowanie	518

Skorowidz 520

Framework automatyzacyjny Pythona — Ansible

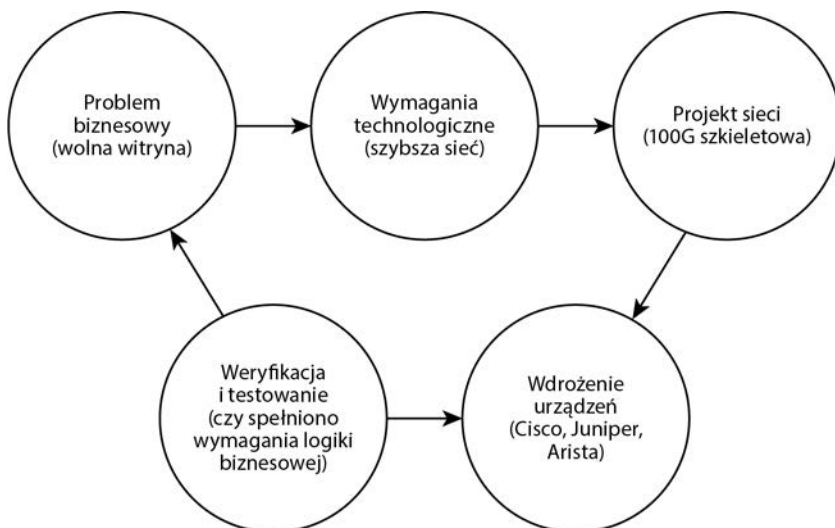
Rozdział

4

Poprzednie dwa rozdziały stanowiły stopniowe wprowadzenie w różne sposoby interakcji z urządzeniami sieciowymi. W rozdziale 2., pt. „Niskopoziomowe interakcje z urządzeniami sieciowymi”, przedstawiłem wykorzystanie bibliotek Pexpect i Paramiko, które w celu kontrolowania interakcji zarządzają interaktywną sesją. W rozdziale 3., pt. „API i sieci intuicyjne”, zaczęliśmy myśleć o naszej sieci w kategoriach API i intencji. Przedstawiłem w nim różne interfejsy API zawierające dobrze zdefiniowaną strukturę poleceń i zapewniające strukturyzowany sposób uzyskiwania informacji zwrotnych z urządzenia. Przechodząc od rozdziału 2. do 3., zaczęliśmy myśleć o naszych zamierzeniach dotyczących sieci. Stopniowo zaczęliśmy też wyrażać naszą sieć w formie kodu.

W tym rozdziale rozwinieśmy ideę przekształcania naszych zamierzeń na wymagania stawiane sieci. Jeśli pracowałeś nad projektami sieci, prawdopodobnie najtrudniejszą częścią procesu nie będą dla Ciebie różne elementy sprzętu sieciowego, ale raczej określanie wymagań biznesowych i przekładanie ich na rzeczywisty projekt sieci. Projekt sieci musi bowiem rozwiązywać problemy biznesowe. Na przykład możesz pracować w zajmującym się infrastrukturą większym zespole obsługującym dobrze prosperującą witrynę poświęconą handlowi elektronicznemu, która w godzinach szczytu wykazuje wolny czas reakcji. Jak ustalić, czy przyczyną tych problemów jest sieć? Jeśli długi czas reakcji witryny faktycznie był spowodowany przeciążeniem sieci, to którą część sieci należy zmodernizować? Czy reszta systemu może skorzystać na większej szybkości i przepustowości sieci?

Diagram przedstawiony na rysunku 4.1 ilustruje prosty proces, który możemy wykonać, by przełożyć wymagania biznesowe na projekt sieci.



Rysunek 4.1. Przekładanie logiki biznesowej na wdrożenie sieciowe

Moim zdaniem automatyzacja sieci to nie tylko możliwość szybszego zmieniania konfiguracji. Powinna ona również rozwiązywać problemy biznesowe, a jednocześnie dokładnie i niezawodnie przekładać nasze intencje na zachowanie urządzeń. Są to cele, o których powinniśmy pamiętać podczas naszej podróży w kierunku automatyzacji sieci. W tym rozdziale przyjrzymy się napisanemu w Pythonie frameworkowi o nazwie **Ansible**, który pozwala nam deklarować intencje dotyczące sieci i jeszcze bardziej uniezależniać się od konkretnych interfejsów API oraz poleceń CLI.

W tym rozdziale przedstawiłem następujące zagadnienia:

- prezentacja Ansible,
- zalety Ansible,
- architektura Ansible,
- zaawansowane zagadnienia stosowania Ansible.

Zacniemy od wprowadzenia do frameworku Ansible.

Ansible — framework bardziej deklaratywny

Wyobraź sobie hipotetyczną sytuację: pewnego ranka budzisz się zły zimnym potem z powodu koszmara o potencjalnym naruszeniu bezpieczeństwa sieci. Zdajesz sobie sprawę, że Twoja sieć zawiera cenne zasoby cyfrowe, które powinny być chronione.

Dobrze wykonywałeś swoją pracę jako administrator sieci, więc jest ona dość bezpieczna, jednak dla pewności chciałbyś jeszcze bardziej zabezpieczyć swoje urządzenia sieciowe.

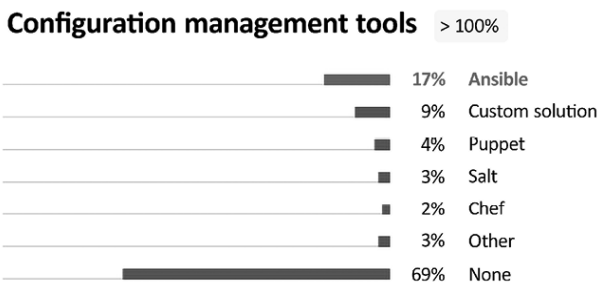
Na samym początku prac dzielisz cel na dwa główne kierunki działań:

- Aktualizację urządzeń do najnowszej wersji oprogramowania. Ten kierunek obejmuje następujące czynności:
 1. Przesłanie obrazu do urządzenia.
 2. Nakazanie urządzeniu, by użyło nowego obrazu.
 3. Ponowne uruchomienie urządzenia.
 4. Sprawdzenie, czy urządzenie działa z nowym obrazem oprogramowania.
- Skonfigurowanie odpowiedniej listy kontroli dostępu na urządzeniach sieciowych. Ten kierunek obejmuje następujące czynności:
 1. Utworzenie listy dostępu na urządzeniu.
 2. Skonfigurowanie listy dostępu na interfejsie sieciowym w sekcji konfiguracji interfejsu.

Jako inżynier sieciowy koncentrujący się na automatyzacji chcesz pisać skrypty, aby dzięki nim niezawodnie konfigurować urządzenia i otrzymywać informacje zwrotne o wykonywanych operacjach. Zaczynasz badać niezbędne polecenia i interfejsy API dla wszystkich wykonywanych działań, sprawdzasz je w laboratorium, a na koniec wdrażasz w środowisku produkcyjnym. Po wykonaniu sporej ilości pracy związanej z aktualizacją systemu operacyjnego i wdrożeniem list kontroli dostępu (ang. *access control list*, w skrócie: ACL) masz nadzieję, że skrypty będzie można przenieść na urządzenia nowej generacji.

Czy nie byłoby miło, gdyby istniało narzędzie pozwalające na skrócenie cyklu projektowanie-implementacja-wdrożenie? W tym rozdziale do automatyzacji będziemy używać frameworku o nazwie Ansible, udostępnianego jako oprogramowanie typu *open source*. Jest to framework, który może uprościć proces przechodzenia od logiki biznesowej do wykonania zadania bez konieczności posługiwania się konkretnymi poleceniami sieciowymi. Ansible pozwala konfigurować systemy, wdrażać oprogramowanie i organizować realizację zadań.

Framework Ansible został napisany w Pythonie i stał się jednym z wiodących narzędzi do automatyzacji dla programistów używających tego języka programowania. Ansible to także jeden z frameworków automatyzacji, które są najczęściej obsługiwane przez producentów urządzeń sieciowych. W badaniu „Python Developers Survey 2020” przeprowadzonym przez JetBrains Ansible znalazł się na pierwszym miejscu wśród narzędzi do zarządzania konfiguracją (patrz rysunek 4.2).



Rysunek 4.2. Wyniki badania Python Developers Survey 2020
(źródło: <https://www.jetbrains.com/lp/python-developers-survey-2020/>)

Zaczynając od wersji 2.10 frameworku, twórcy Ansible rozdzielili harmonogramy wydawania pakietów **ansible-core** i pakietów przygotowywanych przez społeczność. Jest to nieco mylące, więc przyjrzymy się różnicom.

Wersje Ansible

Przed wersją 2.9 framework Ansible miał dość prosty system wersji, były to kolejno: 2.5, 2.6, 2.7 itd. (https://docs.ansible.com/ansible/latest/roadmap/old_roadmap_index.html). Jednak od wersji 2.10 nastąpił przeskok z projektu Ansible 2.10 do 3.0, 4.0 i tak dalej (https://docs.ansible.com/ansible/latest/roadmap/ansible_roadmap_index.html#ansible-roadmap). Jaka jest przyczyna tej zmiany? Otóż zespół Ansible chce oddzielić podstawowy silnik, moduły i wtyczki od szerszych modułów i wtyczek rozwijanych przez społeczność. Pozwala to głównemu zespołowi twórców Ansible na szybszą pracę nad podstawowymi możliwościami frameworku, co daje jednocześnie społeczności czas na nadrobienie zaległości w utrzymaniu ich kodu.

Kiedy mówimy o Ansible, mamy na myśli zbiór pakietów społeczności na pewnym poziomie, powiedzmy, w wersji 3.0. W tej wersji zostanie określona konkretna wymagana wersja pakietu **ansible-core** (początkowo nazywanego **ansible-base**). Na przykład Ansible 3.0 wymaga **ansible-core** 2.10 i nowszych, podczas gdy Ansible 4.0 wymaga **ansible-core** 2.11+. Dzięki zastosowaniu takiej struktury, jeśli zajdzie taka konieczność, możemy zaktualizować **ansible-core** do najnowszej wersji, zachowując jednocześnie starsze wydania pakietów społeczności.

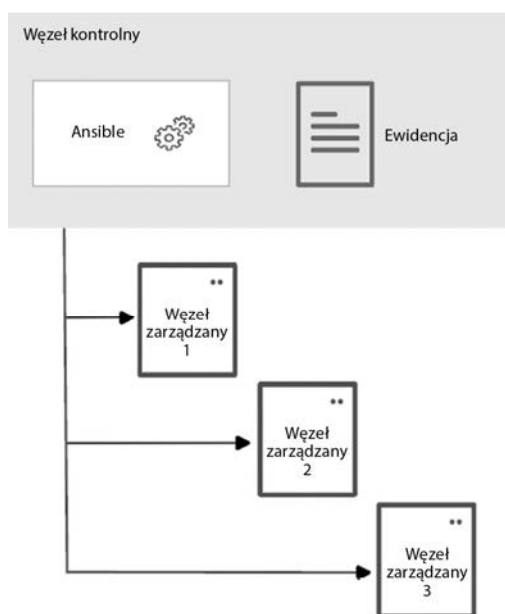
Wskazówka

Jeśli chcesz dowiedzieć się czegoś więcej na temat tego rozdzielania wersji, Ansible udostępnia przydatną stronę z pytaniami i odpowiedziami, która po raz pierwszy pojawiła się w Ansible 3.0: <https://www.ansible.com/blog/ansible-3.0.0-qa>.

A teraz przejdźmy dalej i przyjrzymy się przykładom użycia Ansible.

Pierwszy sieciowy przykład użycia Ansible

Ansible to narzędzie służące do automatyzacji. Jego głównymi cechami są: prostota i łatwość użycia przy minimalnej liczbie zmiennych elementów. Ansible zarządza maszynami bez korzystania z agentów (wyjaśnię to dokładniej w dalszej części rozdziału), a do uruchamiania swojego kodu używa istniejących danych uwierzytelniających systemu operacyjnego i zdalnego oprogramowania Pythona. Ansible jest instalowany na scentralizowanej maszynie zwanej węzłem kontrolnym (ang. *control node*) i wykonywany na maszynie, którą chce kontrolować, zwanej węzłem zarządzanym (ang. *managed node*), patrz rysunek 4.3.



Rysunek 4.3. Architektura Ansible

(źródło: https://docs.ansible.com/ansible/latest/getting_started/index.html)

Podobnie jak w przypadku większości rozwiązań związanych z automatyzacją infrastruktury IT, Ansible rozpoczął od zarządzania serwerami. Większość serwerów ma zainstalowany język Python lub jest w stanie uruchamiać kod napisany w tym języku; Ansible mógłby wykorzystać tę możliwość, by przysyłać kod do zarządzanego węzła i uruchamiać go na nim lokalnie. Jednak jak wiadomo, większość urządzeń sieciowych nie jest w stanie uruchomić kodu napisanego w Pythonie; dlatego też jeśli chodzi o automatyzację sieci, konfiguracja Ansible najpierw jest uruchamiana lokalnie, a dopiero potem zmiany są wprowadzane na urządzeniach zdalnych.

Uwaga

Więcej informacji na temat różnic w automatyzacji sieci można znaleźć w opracowanym przez Ansible dokumencie opublikowanym na stronie https://docs.ansible.com/ansible/latest/network/getting_started/network_differences.html.

Zacznijmy zatem od zainstalowania frameworku Ansible na węźle kontrolnym.

Instalacja Ansible na węźle kontrolnym

W naszym laboratorium zainstalujemy Ansible na hoście z systemem Ubuntu. Jedyne wymagania dotyczące węzła kontrolnego to zainstalowanie na nim języka Python 3.8 lub nowszego, a także systemu zarządzania pakietami Pythona — pip.

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

Zainstalowaną wersję Ansible, jak również inne informacje dotyczące pakietu, możemy sprawdzić przy użyciu przełącznika `--version`:

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

Wskazówka

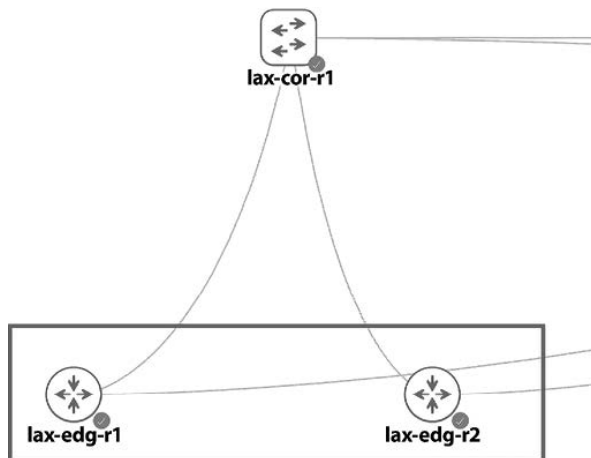
Jeśli jesteś zainteresowany instalacją Ansible na określonych systemach operacyjnych przy użyciu odpowiednich systemów zarządzania pakietami, to informacje na ten temat znajdziesz w dokumentacji Ansible na stronie https://docs.ansible.com/ansible/latest/installation_guide/installation_distros.html.

Wyniki tego polecenia zawierają kilka ważnych informacji. Najważniejszą z nich jest wersja jądra Ansible (2.13.3) i nazwa pliku konfiguracyjnego (na razie go nie ma). To wszystko, czego potrzebujemy, aby rozpocząć korzystanie z Ansible, możemy już zatem zacząć konstruować nasze pierwsze zadanie automatyzacji.

Topologia laboratorium

Framework Ansible jest znany z udostępniania wielu różnych sposobów wykonywania tego samego zadania. Na przykład pliki konfiguracyjne Ansible możemy definiować w różnych lokalizacjach. Oprócz tego zmienne dotyczące konkretnego hosta możemy określać w różnych miejscach, takich jak ewidencja (ang. *inventory*), w playbookach, w rolach, jak również z poziomu wiersza poleceń. To może być zbyt mylące dla osób, które dopiero zaczynają korzystać z Ansible. W tym rozdziale użyję tylko jednego sposobu wykonania konkretnej operacji, który uważam za najbardziej sensowny. Gdy nauczysz się już podstaw, zawsze możesz zapoznać się z dokumentacją, aby znaleźć inne sposoby wykonania tego samego zadania.

W pierwszym przykładzie użyjemy tej samej topologii laboratorium co wcześniej i uruchomimy zadanie na dwóch urządzeniach IOSv, `lax-edg-r1` i `lax-edg-r2` (patrz rysunek 4.4).



Rysunek 4.4. Topologia laboratorium

Pierwszą rzeczą, o której musimy pomyśleć, jest zdefiniowanie hostów, którymi chcemy zarządzać. W Ansible hosty, którymi chcemy zarządzać, definiuje się w pliku ewidencji. Utwórzmy zatem plik o nazwie `hosts` i zapiszmy w nim następujący tekst:

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

Pliki tego typu są zapisywane w formacie INI (<https://pl.wikipedia.org/wiki/INI>). Powyższy plik informuje, że mamy grupę urządzeń o nazwie `ios_devices`, do której należą dwa urządzenia: `iosv-1` i `iosv-2`.

Teraz musimy określić konkretne zmienne odnoszące się do każdego z tych hostów.

Pliki zmiennych

Istnieje wiele miejsc, w których możemy umieścić zmienne dotyczące hosta. My utworzymy katalog o nazwie `host_vars`, a w nim dwa pliki o nazwach identycznych z nazwami hostów, które określiliśmy w pliku ewidencji. Katalog i nazwy plików są ważne, ponieważ w ten sposób Ansible dopasowuje zmienne do hosta. Poniżej przedstawiłem strukturę katalogu z dwoma plikami:

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

W każdym z tych plików umieścimy niezbędne informacje dotyczące odpowiedniego hosta. Na przykład możemy określić adres IP hosta, nazwę użytkownika, hasło, jak również inne informacje. Poniżej przedstawiłem zawartość pliku `iosv-1` używanego w naszym laboratorium:

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

Ten plik jest zapisany w formacie YAML (https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html). Symbol `---` oznacza początek dokumentu. Poniżej symbolu początku dokumentu można zapisać dowolnie wiele par klucz-wartość. Wszystkie klucze zaczynają się od słowa `ansible`, a wartość jest oddzielona od klucza znakiem dwukropka. Wartości skojarzone z kluczami `ansible_host`, `ansible_user` i `ansible_ssh_pass` powinieneś zmienić na wartości odpowiadające Twojemu laboratorium. Skąd mamy znać te nazwy kluczy? Otóż w tym przypadku Twoim najlepszym przyjacielem okaże się dokumentacja Ansible. Ansible wykorzystuje standardowy sposób nazywania parametrów wymieniony w dokumentacji: https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html.

Uwaga

Przed Ansible 2.8 moduły sieciowe nie miały standardowego sposobu nazywania parametrów, co było bardzo mylące. Zaczynając od wersji 2.8, standaryzacja parametrów w modułach sieciowych i reszcie modułów Ansible stała się znacznie lepsza.

Po zdefiniowaniu odpowiednich pól dla zmiennych hosta jesteśmy gotowi, by skonstruować nasz pierwszy playbook Ansible.

Pierwszy playbook

W Ansible playbook to plan opisujący, co chcemy zrobić z zarządzanymi węzłami przy użyciu modułów. Jako operatorzy sieci korzystający z frameworka Ansible większość czasu będziemy spędzać właśnie na pracy z playbookami. A czym są moduły? W pewnym uproszczeniu moduły są gotowymi kodami, których możemy używać do wykonania określonych zadań. Podobnie jak w przypadku modułów Pythona, kod modułu może być dostarczany wraz z domyślną instalacją Ansible, bądź też możemy go instalować osobno.

Gdybyśmy chcieli opisać korzystanie z Ansible, posługując się analogią budowania domku na drzewie, to playbook byłby instrukcją obsługi, moduły używanymi narzędziami, a ewidencja komponentami, nad którymi pracujemy.

Playbooki zostały zaprojektowane tak, aby były czytelne dla człowieka, dlatego zapisujemy je w formacie YAML (https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html). Poniżej przedstawiłem zawartość naszego pierwszego playbooka o nazwie `ios_config_backup.yml`:

```
- name: Kopia zapasowa konfiguracji urządzenia IOS
  hosts: all
  gather_facts: false
  tasks:
    - name: backup
      ios_config:
        backup: yes
```

Zwróć uwagę na znak minusa (-) przed `name` — w formacie YAML określa on element listy. Wszystko w tym samym elemencie listy powinno mieć takie samo wcięcie. Wartość `gather_facts` ustawiliśmy na `false`, ponieważ większość zadań sieciowych jest wykonywana lokalnie przed wprowadzeniem zmian na zdalnym urządzeniu. Moduł Ansible `gather_facts` był używany głównie wtedy, gdy zarządzanymi węzłami były serwery, i służył do zebrania informacji o serwerze przed wykonaniem jakichkolwiek zadań.

W elemencie listy znajdują się dwie pary klucz-wartość, `hosts` i `tasks`. Zmienna `hosts` z przypisaną wartością `all` określa, że będziemy operować na wszystkich hostach wymienionych w pliku ewidencji. Wartością zmiennej `tasks` jest inny element listy, który korzysta z modułu `ios_config` (https://docs.ansible.com/ansible/latest/collections/cisco/ios/ios_config_module.html#ansible-collections-cisco-ios-ios-config-module). `ios_config` jest jednym z elementów kolekcji modułów instalowanych wraz z Ansible.

Posiada on również sporo różnych argumentów. W powyższym przykładzie użyliśmy argumentu `backup`, któremu przypisaliśmy wartość `yes`, aby wskazać, że należy wykonać kopię zapasową bieżącej konfiguracji urządzenia (`running-config`).

Następnym zadaniem, które wykonamy, jest użycie nowej wtyczki połączenia LibSSH dla Ansible. Domyślnie sieciowe połączenia SSH nawiązywane przez Ansible korzystają z biblioteki Paramiko. Jednak biblioteka Paramiko nie gwarantuje zgodności z wymaganiami FIPS¹ i jest dość wolna w sytuacjach, gdy konieczne jest nawiązywanie połączeń z wieloma urządzeniami. Wtyczkę LibSSH można zainstalować w następujący sposób:

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

Informacje określające sposób jej użycia zapiszemy w nowym pliku: `ansible.cfg`. Utwórz ten plik w tym samym katalogu, w którym znajduje się playbook, a następnie zapisz w nim następującą zawartość. W tym pliku konfiguracyjnym ustawimy zmienną `host_key_checking` na `false`, aby zapobiec błędowi, który będzie się pojawiał, jeśli host nie znajdzie się początkowo na liście `known_hosts` w pliku konfiguracyjnym `ssh`:

```
[defaults]
host_key_checking = False

[persistent_connection]
ssh_type = libssh
```

Na koniec możemy wykonać playbook, używając w tym celu polecenia `ansible-playbook` z przełącznikiem `-i`, aby wskazać plik ewidencji:

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

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

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

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

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

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

Jeśli spojrzymy na nasz katalog roboczy, w którym wykonywany jest playbook, to zobaczymy, że jak za dotknięciem czarodziejskiej różdżki pojawi się w nim katalog o nazwie

¹ FIPS — *Federal Information Processing Standard*; to opracowany przez Narodowy Instytut Standaryzacji i Technologii (NIST) USA zestaw standardów bezpieczeństwa — *przyp. tłum.*

backup, a w nim opatrzone znacznikami czasu pliki bieżących konfiguracji dwóch urządzeń z naszego laboratorium! Teraz możemy zaplanować wykonywanie tego polecenia w cronie, aby uruchamiać je co noc w celu tworzenia kopii zapasowej konfiguracji wszystkich naszych urządzeń.

Gratuluję wykonania pierwszego playbooka Ansible! Nawet przy tak prostym playbooku jak ten udało się nam przygotować bardzo przydatne zadanie automatyzacji; co więcej, udało się nam to zrobić w całkiem krótkim czasie. Już niebawem nieco rozwiniemy ten playbook, ale najpierw wyjaśnię, dlaczego Ansible jest dobrym narzędziem do zarządzania siecią. Pamiętaj, że moduły Ansible są napisane w Pythonie, co jest wielką zaletą dla inżyniera sieci pragnącego używać Pythona do automatyzacji swoich zadań, nieprawdaz?

Zalety Ansible

Oprócz Ansible istnieje wiele innych frameworków przeznaczonych do automatyzacji infrastruktury, takich jak: Chef, Puppet i SaltStack. Każdy z nich oferuje swoje unikalne możliwości; nie można wskazać jednego, który spełniałby unikalne wymagania wszystkich organizacji. W tym podrozdziale przedstawię niektóre zalety frameworku Ansible i wyjaśnię, dlaczego uważam, że jest to dobre narzędzie do automatyzacji sieci.

Aby nie wywoływać niepotrzebnych kłótni, opisując zalety Ansible, będę je pobieżnie porównywał z innymi frameworkami. Te inne frameworki mogą przyjmować niektóre z tych samych filozofii lub niektóre możliwości Ansible, jednak rzadko kiedy udostępniają wszystkie możliwości, o których wspomnę w dalszej części rozdziału. To właśnie połączenie wszystkich tych cech i filozofii sprawia, że Ansible jest idealnym frameworkiem do automatyzacji sieci.

Brak agentów

W przeciwieństwie do niektórych swoich odpowiedników Ansible nie wymaga ścisłego modelu *master-client*. Na kliencie nie trzeba instalować żadnego oprogramowania ani agenta, który komunikowałby się z serwerem. Poza interpreterem Pythona, dostępnym domyślnie na wielu platformach, nie jest potrzebne żadne dodatkowe oprogramowanie.

W przypadku modułów automatyzacji sieci, zamiast polegać na agentach zdalnego hosta, Ansible używa wywołań SSH lub API do przesyłania wymaganych zmian do zdalnego hosta. To dodatkowo zmniejsza potrzebę korzystania z interpretera Pythona. Ma to ogromne znaczenie dla zarządzania urządzeniami sieciowymi, ponieważ ich producenci zazwyczaj niechętnie umieszczają na swoich platformach oprogramowanie innych firm. Z drugiej strony protokół SSH jest obsługiwany przez urządzenia sieciowe.

Jak widzieliśmy w rozdziale 3., pt. „API i rozwiązania IDN”, nowsze urządzenia sieciowe zapewniają również warstwę API, która także może być wykorzystywana przez Ansible.

Ponieważ na zdalnym hoście nie ma agenta, Ansible nie korzysta z modelu ściągania (ang. *pull*), w którym to agent pobiera informacje z serwera głównego, a zamiast tego używa modelu wypychania (ang. *push*) do przesyłania zmian na zdalne urządzenia. Model wypychania jest bardziej deterministyczny, ponieważ wszystko pochodzi z maszyny sterującej. W modelu ściągania czas obsługi może się różnić w zależności od klienta, co może powodować rozbieżności czasowe.

Chciałbym jeszcze raz podkreślić, jak niezwykle ważne jest znaczenie działania bez korzystania z agentów podczas pracy z już istniejącym sprzętem sieciowym. Jest to zwykle jeden z głównych powodów tego, że operatorzy sieci i dostawcy stosują Ansible.

Idempotentność

Według Wikipedii w matematyce i informatyce idempotentność to właściwość niektórych operacji, które stosowane wielokrotnie dają taki sam wynik jak w przypadku użycia jednokrotnego (<https://pl.wikipedia.org/wiki/Idempotentność>). Mówiąc bardziej potocznie, oznacza to, że wielokrotne uruchamianie tej samej procedury nie zmienia systemu inaczej, niż gdy został on zmieniony po jej pierwszym uruchomieniu. Ansible dąży do bycia idempotentnym, co jest dobre dla operacji sieciowych, które wymagają określonej kolejności operacji. W naszym pierwszym przykładowym playbooku podczas jego wykonywania pojawia się wartość "changed"; wartość ta będzie „fałszywa”, jeśli na zdalnym urządzeniu nie wprowadzono żadnych zmian.

Zaletę idempotentności najlepiej porównać z napisanymi wcześniej skryptami korzystającymi z bibliotek Pexpect i Paramiko. Pamiętaj, że te skrypty zostały napisane w celu przesyłania poleceń tak, jakby inżynier siedział przy terminalu i wykonywał je na zdalnym urządzeniu. Jeśli miałbyś wykonać taki skrypt 10 razy, skrypt 10 razy wprowadziłby te same zmiany. Jeśli napiszemy to samo zadanie w formie playbooka Ansible, najpierw zostanie sprawdzona istniejąca konfiguracja urządzenia, a playbook zostanie wykonany tylko wtedy, gdy wprowadzane zmiany nie będą istnieć. Jeśli wykonamy playbook 10 razy, zmiana zostanie zastosowana tylko podczas pierwszego uruchomienia, a podczas kolejnych 9 uruchomień zostanie ona pominięta.

Dzięki idempotentności możemy wielokrotnie wykonywać playbook bez obawy, że zostaną wprowadzone niepotrzebne zmiany. Jest to ważne, ponieważ musimy automatycznie sprawdzać spójność stanu bez dodatkowych kosztów.

Prostota i rozszerzalność

Ansible jest napisany w Pythonie, a do pisania playbooków używa formatu YAML. Zarówno sam Python, jak i format YAML są stosunkowo łatwe do opanowania. A pamiętasz składnię poleceń Cisco IOS? Jest to **język dziedzinowy** (ang. *domain-specific language*, w skrócie **DSL**), który ma zastosowanie tylko w przypadku zarządzania urządzeniami Cisco IOS lub innym sprzętem o podobnej strukturze; nie jest to język ogólnego przeznaczenia i nadaje się do stosowania tylko w ściśle ograniczonym zakresie. Na szczęście, w przeciwieństwie do niektórych innych narzędzi do automatyzacji, nie istnieje żaden **język dziedzinowy**, którego trzeba by się nauczyć, by móc używać Ansible, ponieważ zarówno YAML, jak i Python są szeroko stosowane jako języki ogólnego przeznaczenia.

Ansible jest frameworkiem rozszerzalnym. Jak pokazałem w poprzednim przykładzie, Ansible początkowo służył do automatyzacji działania serwerów (głównie linuksowych). Następnie zaczął także zapewniać możliwość zarządzania maszynami z systemem Windows, a używał do tego celu PowerShella. Ponieważ coraz więcej osób w branży sieciowej zaczęło wdrażać Ansible, automatyzacja sieci jest obecnie głównym zagadnieniem w grupach roboczych Ansible.

Ta prostota i rozszerzalność dobrze rokują na przyszłość. Świat technologii szybko ewoluje, a my nieustannie staramy się do niego dostosowywać. Czy nie byłoby wspaniale nauczyć się technologii raz i móc korzystać z niej w przyszłości, niezależnie od najnowszych trendów? Osiągnięcia Ansible dobrze rokują pod względem adaptacji technologii w przyszłości.

Skoro już przedstawiłem niektóre zalety Ansible, spróbujmy zastosować to, czego już się dowiedziałeś.

Ansible Content Collection

Zacznę od przedstawienia wszystkich modułów dostępnych w domyślnej instalacji Ansible. Są one zorganizowane w formie kolekcji zawartości — Content Collections (<https://www.ansible.com/products/content-collections>), czasami określanych skrótowo „kolekcjami”. Listę tych kolekcji można wyświetlić za pomocą polecenia `ansible-galaxy collection list`. Poniżej przedstawiłem niektóre z najważniejszych kolekcji sieciowych:

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

# /home/echou/Mastering_Python_Networking_Fourth_Edition/venv/lib/
python3.10/site-packages/ansible_collections
Collection                                Version
-----
```

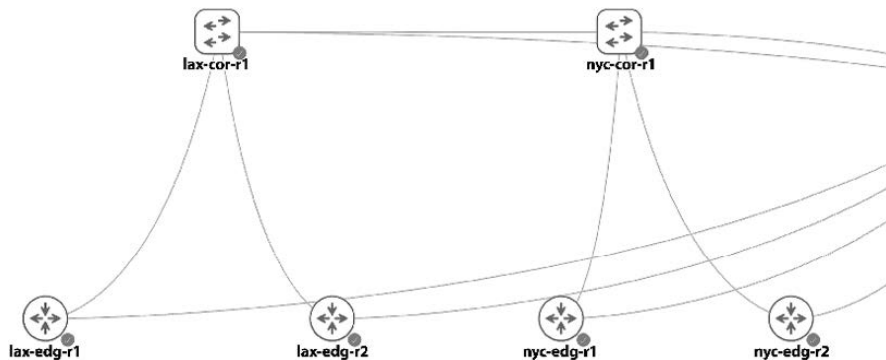
```
...
ansible.netcommon      3.1.0
arista.eos              5.0.1
cisco.aci               2.2.0
cisco.asa               3.1.0
cisco.dnac              6.5.3
cisco.intersight       1.0.19
cisco.ios               3.3.0
cisco.iosxr             3.3.0
cisco.ise               2.5.0
cisco.meraki            2.10.1
cisco.mso               2.0.0
cisco.nso               1.0.3
cisco.nxos              3.1.0
cisco.ucs               1.8.0
community.ciscosmb      1.0.5
community.fortios       1.0.0
community.network       4.0.1
dellemc.enterprise_sonic 1.1.1
f5networks.f5_modules   1.19.0
fortinet.fortimanager   2.1.5
fortinet.fortios        2.1.7
mellanox.onyx           1.0.0
openstack.cloud         1.8.0
openvswitch.openvswitch 2.1.0
vyos.vyos               3.0.1
```

Jak wynika z powyższej listy, nawet przy domyślnej instalacji Ansible dostępna jest duża kolekcja modułów związanych z siecią, z których można korzystać. Obejmują one zarówno oprogramowanie korporacyjne, jak i projekty typu *open source*. Doskonałym punktem wyjścia będzie przejrzanie tej listy i zapoznanie się z tymi modułami, które mogą się przydać w Twoim środowisku produkcyjnym. Dokumentacja Ansible zawiera również pełną listę wszystkich dostępnych kolekcji, a można ją znaleźć na stronie <https://docs.ansible.com/ansible/latest/collections/index.html>. Kolekcje można również rozszerzyć przy użyciu polecenia `ansible-galaxy install`. Więcej informacji na ten temat można znaleźć na stronie https://docs.ansible.com/ansible/latest/user_guide/collections_using.html.

Kolejne sieciowe przykłady użycia Ansible

Pierwszy sieciowy przykład użycia Ansible sprawił, że choć byłeś zupełnie początkującym użytkownikiem tego frameworku, udało Ci się uruchomić pierwsze użyteczne zadanie związane z automatyzacją sieci. Opierając się na tych solidnych podstawach, spróbujemy teraz poznać więcej możliwości Ansible.

Zacniemy od sprawdzenia, jak możemy zbudować plik ewidencji, który obejmuje wszystkie nasze urządzenia sieciowe. Jeśli pamiętasz, mamy dwa centra danych, każde z urządzeniami rdzenia i brzegowymi (patrz rysunek 4.5).



Rysunek 4.5. Pełna topologia laboratorium

W tym przykładzie w pliku ewidencji uwzględnimy wszystkie nasze urządzenia sieciowe.

Zagnieżdżanie ewidencji

Możemy przygotować plik ewidencji zawierający zagnieżdżenia. Na przykład możemy utworzyć plik z listą hostów o nazwie *hosts_full*, a w nim grupy, które będą zawierać elementy z innych grup:

```
[lax_cor_devices]
lax-cor-r1
```

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

```
[nyc_cor_devices]
nyc-cor-r1
```

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

```
[lax_dc:children]
lax_cor_devices
lax_edg_devices
```

```
[nyc_dc:children]
nyc_cor_devices
nyc_edg_devices
```

```

[ios_devices:children]
lax_edg_devices
nyc_edg_devices

[nxos_devices:children]
nyc_cor_devices
lax_cor_devices

```

W tym pliku grupujemy urządzenia zarówno według ról, jak i funkcji, używając w tym celu zapisu o postaci [*nazwa*:children]. Aby pracować z tym nowym plikiem ewidencji, będziemy musieli zaktualizować katalog *host_vars*, tak by zawierał odpowiednie nazwy urządzeń:

```

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

```

Będziemy również musieli odpowiednio zmienić wartości *ansible_host* i *ansible_network_os*; poniżej pokazałem, jak to zrobić na przykładzie urządzenia *lax-cor-r1*:

```

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

```

Teraz możemy użyć nazwy grupy nadrzędnej, aby uwzględnić jej elementy podrzędne. Na przykład w playbooku *nxos_config_backup.yml* określiłem tylko grupę nadrzędną *nxos_devices* zamiast *all*:

```

- name: Kopia zapasowa konfiguracji urządzeń NX-OS
  hosts: nxos_devices
  gather_facts: false
  tasks:
    - name: backup
      nxos_config:
        backup: yes

```

Podczas wykonywania tego playbooka automatycznie dołączone zostaną elementy podrzędne wskazanej grupy, czyli: *lax_cor_devices* i *nyc_cor_devices*. Zwróć także uwagę, że by umożliwić uwzględnienie nowego typu urządzeń, użyliśmy oddzielnego modułu *nxos_config* (https://docs.ansible.com/ansible/latest/collections/cisco/nxos/nxos_config_module.html#ansible-collections-cisco-nxos-nxos-config-module).

Wyrażenia warunkowe w Ansible

Wyrażenia warunkowe w Ansible są podobne do instrukcji warunkowych w językach programowania. Ansible używa słów kluczowych związanych z warunkami, aby uruchamiać zadanie tylko wtedy, gdy dany warunek jest spełniony. W wielu przypadkach wykonanie całych grup zadań (ang. *play*) lub samych zadań może zależeć od wartości faktu (ang. *fact*), zmiennej lub wyniku poprzedniego zadania. Na przykład dysponując grupą zadań służącą do aktualizacji obrazów routerów, zapewne będziesz chciał dołączyć krok, który pozwoli Ci się upewnić, że nowy obraz routera znajduje się na urządzeniu, zanim przejdziesz do następnej grupy zadań służącej do ponownego uruchomienia tego routera.

W tym przykładzie przyjrzymy się klauzuli `when`, która jest obsługiwana przez wszystkie moduły. Klauzula ta przydaje się, gdy chcemy sprawdzić wartość zmiennej lub wynik wykonania grupy zadań i odpowiednio zareagować. Poniżej przedstawiłem niektóre z dostępnych warunków:

- równy (`eq`),
- różny (`neq`),
- większy niż (`gt`),
- większy lub równy (`ge`),
- mniejszy niż (`lt`),
- mniejszy lub równy (`le`),
- zawiera.

Przyjrzyjmy się poniższemu playbookowi o nazwie `ios_conditional.yml`:

```
---
- name: Wyniki polecenia IOS dla klauzuli when
  hosts: ios_devices
  gather_facts: false
  tasks:
    - name: wyświetl nazwę hosta
      ios_command:
        commands:
          - show run | i hostname
      register: output

    - name: pokaż wyniki, używając warunku when
      when: output.stdout == ["hostname nyc-edg-r2"]
      debug:
        msg: '{{ output }}'
```

W playbooku znajdują się dwa zadania. W pierwszym z nich używamy modułu `register`, aby zapisać dane wyjściowe polecenia `show run | i hostname` w zmiennej `output`.

Zmienna `output` będzie zatem zawierać listę `stdout` z danymi wyjściowymi. W drugim zadaniu używamy klauzuli `when`, aby wyświetlić dane wyjściowe tylko wtedy, gdy nazwą hosta jest `nyc-edg-r2`. Spróbujmy teraz wykonać ten playbook:

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

PLAY [Wyniki polecenia IOS dla klauzuli when] *****
*****

TASK [wyświetl nazwę hosta] *****
*****

ok: [lax-edg-r1]
ok: [nyc-edg-r2]
ok: [lax-edg-r2]
ok: [nyc-edg-r1]

TASK [pokaż wyniki, używając warunku when] *****
*****

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

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

Jak widać, dane wyjściowe z urządzeń `lax-edg-r1`, `lax-edg-r2` i `nyc-edg-r1` zostały pominięte, ponieważ nie spełniały zadanego warunku. Co więcej, możemy zobaczyć wynik `changed=0` dla wszystkich urządzeń. Jest to zgodne z cechą idempotentności Ansible.

Zmiany konfiguracji

Możemy połączyć zastosowanie warunku z wprowadzaniem zmian w konfiguracji; przykład takiego rozwiązania przedstawiłem w poniższym playbooku o nazwie *ios_conditional_config.yml*:

```
---
- name: Wyniki polecenia IOS dla klauzuli when
  hosts: ios_devices
  gather_facts: false
  tasks:
    - name: wyświetl nazwę hosta
      ios_command:
        commands:
          - show run | i hostname
      register: output

    - name: pokaż wyniki, używając warunku with
      when: output.stdout == ["hostname nyc-edg-r2"]
      ios_config:
        lines:
          - logging buffered 30000
```

Bufor rejestrowania będzie zmieniany tylko wtedy, gdy warunek zostanie spełniony. Poniżej przedstawiłem wyniki uzyskane po pierwszym uruchomieniu playbooka:

```
(venv) $ ansible-playbook -i hosts_full ios_conditional_config.yml
<pominięte>
TASK [pokaż wyniki, używając warunku with] *****
*****
skipping: [lax-edg-r1]
skipping: [lax-edg-r2]
skipping: [nyc-edg-r1]
[WARNING]: To ensure idempotency and correct diff the input configuration
lines should be similar to how they appear if
present in the running configuration on device
changed: [nyc-edg-r2]
PLAY RECAP *****
*****
lax-edg-r1           : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
lax-edg-r2           : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r1           : ok=1    changed=0    unreachable=0
failed=0    skipped=1    rescued=0    ignored=0
nyc-edg-r2           : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Konsola urządzenia nyc-edg-r2 pokaże, że konfiguracja została zmieniona:

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

Jednak gdy uruchomimy ten playbook po raz drugi, ta sama zmiana **nie** zostanie zastosowana ponownie, ponieważ została wprowadzona już wcześniej:

```
<pominięte>
TASK [pokaż wyniki, używając warunku with] *****
*****
skipping: [lax-edg-r1]
skipping: [lax-edg-r2]
skipping: [nyc-edg-r1]
ok: [nyc-edg-r2]
```

Czyż to nie fajne? Za pomocą prostego playbooka możemy bezpiecznie wprowadzić zmianę konfiguracji tylko na wybranych urządzeniach, a co więcej, możemy zrobić to z zachowaniem idempotentności.

Fakty sieciowe Ansible

Przed udostępnieniem wersji 2.5 Ansible narzędzia sieciowe frameworku zawierały wiele modułów faktów charakterystycznych dla konkretnych producentów sprzętu. W efekcie zarówno nazewnictwo poszczególnych faktów, jak i ich wykorzystanie różniły się w zależności od producenta. Począwszy od wersji 2.5, Ansible zaczęło standaryzować swoje moduły faktów sieciowych. Moduły faktów sieciowych Ansible zbierają informacje z systemu i przechowują wyniki w faktach, których nazwy rozpoczynają się od prefiksu `ansible_net_`. Dane zebrane przez te moduły są opisane w sekcjach *return values* w dokumentacji modułów. Jest to bardzo przydatne, gdyż możemy gromadzić fakty sieciowe i wykonywać zadania tylko na ich podstawie.

W poniższym playbooku `ios_facts_playbook` przedstawiłem przykład zastosowania modułu `ios_facts`:

```
---
- name: Fakty sieciowe IOS
  connection: network_cli
  gather_facts: false
  hosts: ios_devices
  tasks:
    - name: Gromadzenie faktów przy użyciu modułu ios_facts
      ios_facts:
        when: ansible_network_os == 'ios'

    - name: Wyświetlenie wybranych faktów
      debug:
```

```
msg: "Na hoście {{ ansible_net_hostname }} działa Ansible {{
ansible_net_version }}"
```

- name: Wyświetlenie wszystkich faktów dla hostów
- debug:
- var: hostvars

W tym playbooku wprowadziłem pojęcie zmiennych. Podwójne nawiasy klamrowe {{ }} wskazują, że jest to zmienna, a wartość zmiennej powinna zostać wyświetlona w generowanych wynikach.

Poniżej przedstawiłem wybrane fragmenty wyników generowanych podczas wykonywania tego playbooka:

```
(venv) $ ansible-playbook -i hosts_full ios_facts_playbook.yml
...
TASK [Display certain facts] *****
*****
ok: [lax-edg-r1] => {
  "msg": "Na hoście lax-edg-r1 działa Ansible 15.8(3)M2"
}
ok: [lax-edg-r2] => {
  "msg": "Na hoście lax-edg-r2 działa Ansible 15.8(3)M2"
}
ok: [nyc-edg-r1] => {
  "msg": "Na hoście nyc-edg-r1 działa Ansible 15.8(3)M2"
}
ok: [nyc-edg-r2] => {
  "msg": "Na hoście nyc-edg-r2 działa Ansible 15.8(3)M2"
}
...
TASK [Wyświetlenie wszystkich faktów dla hostów] *****
*****
ok: [lax-edg-r1] => {
  "hostvars": {
    "lax-cor-r1": {
      ...
      "ansible_facts": {
        "net_api": "cliconf",
        "net_gather_network_resources": [],
        "net_gather_subset": [
          "default"
        ],
        "net_hostname": "lax-edg-r1",
        "net_image": "flash0:/vios-adventerprisek9-m",
        "net_iostype": "IOS",
        "net_model": "IOSv",
        "net_python_version": "3.10.4",
        "net_serialnum": "98U40DKV403INH1ULHYHB",
        "net_system": "ios",
```

```

        "net_version": "15.8(3)M2",
        "network_resources": {}
    },
    ...

```

Możemy teraz wykorzystać fakty i łączyć je z klauzulą warunkową, aby dostosowywać sposoby działania wykonywanych operacji.

Pętle Ansible

Ansible udostępnia szereg pętli, których można używać w playbookach, a są to: standardowe pętle, pętle operujące na plikach, podelementy, pętle `do-until` i wiele innych. W tym punkcie przyjrzymy się dwóm najczęściej używanym rodzajom pętli: pętlom standardowym i pętlom operującym na słownikach.

Pętle standardowe

Pętle standardowe w playbookach są często używane do łatwego wielokrotnego wykonywania podobnych zadań. Składnia tych pętli jest bardzo prosta: zmienna `{{ item }}` jest nazwą zastępczą, której są przypisywane kolejne elementy listy `loop`. W następnym przykładzie, zapisanym w pliku *standard_loop.yml*, użyjemy polecenia `echo`, by dla hosta `localhost` wyświetlić w pętli wszystkie elementy listy `loop`.

```

- name: Wyświetlenie elementów listy w pętli
  hosts: "localhost"
  gather_facts: false
  tasks:
    - name: Wyświetlenie elementów listy w pętli
      command: echo "{{ item }}"
      loop:
        - 'r1'
        - 'r2'
        - 'r3'
        - 'r4'
        - 'r5'

```

A teraz spróbujmy wykonać tego playbooka:

```

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

PLAY [Wyświetlenie elementów listy w pętli] *****
*****

TASK [Wyświetlenie elementów listy w pętli] *****
*****
changed: [localhost] => (item=r1)
changed: [localhost] => (item=r2)
changed: [localhost] => (item=r3)

```

```

changed: [localhost] => (item=r4)
changed: [localhost] => (item=r5)

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

```

W analogiczny sposób możemy systematycznie dodawać do naszych urządzeń sieci VLAN. Kolejny playbook, zapisany w pliku *standard_loop_vlan_example.yml*, pokazuje, w jaki sposób można dodać do urządzenia trzy sieci VLAN:

```

- name: Dodawanie kilku sieci VLAN
  hosts: "nyc-cor-r1"
  gather_facts: false
  connection: network_cli
  vars:
    vlan_numbers: [100, 200, 300]
  tasks:
    - name: Dodanie sieci VLAN
      nxos_config:
        lines:
          - vlan {{ item }}
      loop: "{{ vlan_numbers }}"
      register: output

```

Poniżej przedstawiłem wyniki wykonania tego playbooka:

```

(venv) $ ansible-playbook -i hosts_full standard_loop_vlan_example.yml

PLAY [Dodawanie kilku sieci VLAN] *****
*****

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

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

```

Jak widać w tym playbooku, lista, na której operuje pętla, może być wczytywana ze zmiennej, co zwiększa elastyczność struktury playbooka:

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

Pętla standardowa to świetne rozwiązanie pozwalające oszczędzić czas w przypadkach, gdy w playbooku musimy wykonywać powtarzające się zadania. W następnym podpunkcie rozdziału przedstawię pętlę operującą na słowniku.

Pętle operujące na słownikach

Kiedy musimy wygenerować konfigurację, często musimy posługiwać się encją zawierającą więcej niż jeden atrybut. Jeśli przypomnisz sobie przykład tworzenia sieci VLAN przedstawiony w poprzednim podpunkcie, to zapewne zwrócisz uwagę, że każda sieć VLAN ma kilka unikalnych atrybutów, takich jak opis, adres IP bramy i ewentualnie inne. Często do reprezentowania encji, w których chcemy umieszczać wiele atrybutów, możemy używać słownika.

Rozszerzmy poprzedni przykład poprzez zastosowanie w nim zmiennej słownikowej; zmodyfikowany kod przykładu znajdziesz w pliku *standard_loop_vlan_example_2.yml*. Zdefiniowaliśmy w nim słownik z informacjami o trzech sieciach VLAN, przy czym jego elementami są zagnieżdżone słowniki zawierające opis sieci oraz jej adres IP:

```
---
- name: Dodawanie kilku sieci VLAN
  hosts: "nyc-cor-r1"
  gather_facts: false
  connection: network_cli
  vars:
    vlans: {
      "100": {"description": "floor_1", "ip": "192.168.10.1"},
      "200": {"description": "floor_2", "ip": "192.168.20.1"},
      "300": {"description": "floor_3", "ip": "192.168.30.1"}
    }
  tasks:
    - name: Dodanie sieci VLAN
      nxos_config:
        lines:
          - vlan {{ item.key }}
        with_dict: "{{ vlans }}"
    - name: Skonfigurowanie sieci VLAN
      nxos_config:
        lines:
          - description {{ item.value.description }}
          - ip address {{ item.value.ip }}/24
        parents: interface vlan {{ item.key }}
        with_dict: "{{ vlans }}"
```

Pierwszym zadaniem określonym w tym playbooku jest dodanie sieci VLAN przy użyciu kluczy elementów. Z kolei drugie zadanie polega na skonfigurowaniu interfejsów VLAN przy użyciu wartości zapisanych w każdym z elementów słownika. Zauważ, że używamy parametru `parent`, aby jednoznacznie zidentyfikować sekcję, w której polecenia powinny być sprawdzane. Wynika to z faktu, że zarówno opis, jak i adres IP są konfigurowane w podsekcji `interface vlan <numer>` konfiguracji.

Przed wykonaniem polecenia musimy się upewnić, że na urządzeniu `nyc-cor-r1` została włączona funkcja interfejsu warstwy 3:

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

Następnie możemy uruchomić playbook tak, jak zrobiliśmy to wcześniej. Wygenerowane wyniki pokażą, że operacje zostały wykonane w pętli dla każdego elementu słownika:

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

PLAY [Dodawanie kilku sieci VLAN] *****
*****

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

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

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

Końcowy efekt możemy sprawdzić na samym urządzeniu:

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

interface Vlan200
  description floor_2
  ip address 192.168.20.1/24

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

Informacje o innych rodzajach pętli dostępnych w frameworku Ansible można znaleźć w jego dokumentacji na stronie https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html.

Stosowanie pętli operujących na słownikach wymaga pewnej praktyki i uwagi podczas pierwszych kilku zastosowań. Ale podobnie jak pętle standardowe, także i pętle operujące na słownikach będą nieocenionym narzędziem w naszym przyborniku. Pętle Ansible są narzędziem, które może zaoszczędzić nam wiele czasu i znacząco poprawić czytelność playbooków. W następnym punkcie rozdziału przyjrzymy się szablonom Ansible, które pozwalają nam wprowadzać systematyczne zmiany w plikach tekstowych powszechnie używanych do konfigurowania urządzeń sieciowych.

Szablony

Odkąd zacząłem pracować jako inżynier sieci, zawsze korzystałem z jakiegoś systemu szablonów. Z mojego doświadczenia wynika, że konfiguracje sieciowe wielu urządzeń zawierają sekcje, które są identyczne, zwłaszcza jeśli te urządzenia pełnią w sieci tę samą rolę.

W większości przypadków, gdy musimy przygotować do użycia nowe urządzenie, używamy tej samej konfiguracji w formie szablonu, zastępujemy wartości niezbędnych pól i kopiujemy plik na nowe urządzenie. Dzięki szablonom Ansible (https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_templating.html) wszystkie te operacje można zautomatyzować.

Ansible w celu zapewnienia obsługi dynamicznych wyrażeń oraz dostępu do zmiennych i faktów używa silnika szablonów Jinja (<https://jinja.palletsprojects.com/en/3.1.x/>). Jinja ma własną składnię i metodę wykonywania pętli i warunków; na szczęście do naszych celów musimy znać tylko jego podstawy. Moduł szablonów Ansible jest ważnym narzędziem, którego będziemy używać w naszych codziennych zadaniach, dlatego większość

tego punktu rozdziału poświęcę na jego dokładniejsze przedstawienie. Składnię szablonów Ansible będę wyjaśniał stopniowo, rozbudowując początkowo prosty playbook o coraz bardziej złożone zadania.

Podstawowa składnia użycia szablonu jest bardzo prosta; musimy tylko określić plik źródłowy i lokalizację docelową, do której należy go skopiować.

Zacniemy od utworzenia nowego katalogu o nazwie *Templates*, w którym będziemy tworzyć nasze playbooks. Na razie utwórz pusty plik:

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

Następnie użyjemy następującego playbooka o nazwie *template_1.yml*, aby skopiować plik *file1* do pliku *file2*. Pamiętaj, że playbook jest wykonywany tylko na maszynie kontrolnej:

```
---
- name: Podstawy szablonów
  hosts: localhost
  tasks:
    - name: Skopiuj plik i zapisz go pod inną nazwą
      template:
        src=/home/echou/Mastering_Python_Networking_Fourth_Edition/
        Chapter04/Templates/file1
        dest=/home/echou/Mastering_Python_Networking_Fourth_Edition/
        Chapter04/Templates/file2
```

Wykonanie tego playbooka spowoduje utworzenie nowego pliku:

```
(venv) $ ansible-playbook -i hosts template_1.yml
PLAY [Podstawy szablonów] *****
*****

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

TASK [Skopiuj plik i zapisz go pod inną nazwą]
*****
*****
changed: [localhost]

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

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

W naszych szablonach pliki źródłowe mogą mieć dowolne rozszerzenie, ale ponieważ są one przetwarzane przez silnik szablonów Jinja2, jako źródło szablonu utworzymy plik tekstowy o nazwie *nxos.j2*. Szablon będzie zgodny z konwencją silnika Jinja polegającą na zapisywaniu zmiennych w podwójnych nawiasach klamrowych, a także używaniu nawiasów klamrowych i znaku procenta do określania poleceń:

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

feature telnet
feature ospf
feature bgp
feature interface-vlan

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

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

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

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

Teraz przygotujemy playbook do tworzenia szablonów konfiguracji bazujących na pliku *nxos.j2*.

Zmienne szablonów Jinja

Przedstawiony poniżej playbook *template_2.yml* to zmodyfikowana wersja playbooka z poprzedniego przykładu, zawierająca następujące rozszerzenia:

- Używa pliku źródłowego *nxos.j2*.
- Nazwa docelowa jest teraz zmienną pobieraną ze zmiennej *nexus_devices* zdefiniowanej w playbooku.
- Każde z urządzeń w grupie *nexus_devices* zawiera zmienne, które w szablonie zostaną podstawione lub użyte w pętli.

Ten playbook może wyglądać na bardziej złożony niż poprzedni, ale gdyby usunąć z niego część definiującą zmienne, okaże się, że oba playbooki są do siebie bardzo podobne:

```

---
- name: Pętle w szablonach
  hosts: localhost
  vars:
    nexus_devices: {
      "nx-osv-1": {
        "hostname": "nx-osv-1",
        "username": "cisco",
        "password": "cisco",
        "vlans": [100, 200, 300],
        "l3_vlan_interfaces": True,
        "vlan_interfaces": [
          {"int_num": "100", "ip": "192.168.10.1"},
          {"int_num": "200", "ip": "192.168.20.1"},
          {"int_num": "300", "ip": "192.168.30.1"}
        ],
        "netflow_enable": True
      },
      "nx-osv-2": {
        "hostname": "nx-osv-2",
        "username": "cisco",
        "password": "cisco",
        "vlans": [100, 200, 300],
        "l3_vlan_interfaces": False,
        "netflow_enable": False
      }
    }
  tasks:
    - name: Tworzymy pliki konfiguracyjne routera
      template:
        src=/home/echou/Mastering_Python_Networking_Fourth_Edition/
        Chapter04/Templates/nxos.j2
        dest=/home/echou/Mastering_Python_Networking_Fourth_Edition/
        Chapter04/Templates/{{ item.key }}.conf
        with_dict: "{{ nexus_devices }}"

```

Nie wykonuj na razie tego playbooka; musimy jeszcze przyjrzeć się instrukcjom warunkowym `if` i pętlom `for` zapisanym w szablonach Jinja2 wewnątrz symboli `{% %}`.

Pętle w szablonach Jinja

W przedstawionym wcześniej szablonie *nxos.j2* zastosowaliśmy dwie pętle `for`; pierwsza z nich operuje na sieciach VLAN, a druga na interfejsach VLAN:

```

{% for vlan_num in item.value.vlans %}
vlan {{ vlan_num }}
{% endfor %}
{% if item.value.l3_vlan_interfaces %}
{% for vlan_interface in item.value.vlan_interfaces %}
interface {{ vlan_interface.int_num }}

```

```

    ip address {{ vlan_interface.ip }}/24
{% endfor %}
{% endif %}

```

Jak pamiętasz, w szablonach Jinja pętle mogą operować zarówno na listach, jak i na słownikach. W naszym przykładzie zmienna `vlangs` jest listą, podczas gdy zmienna `vlan_interfaces` jest listą słowników.

Pętla operująca na liście `vlan_interfaces` jest zagnieżdżona wewnątrz instrukcji warunkowej. Jest to ostatnia rzecz, którą dodamy do naszego playbooka przed jego wykonaniem.

Instrukcje warunkowe w szablonach Jinja

Jinja obsługuje instrukcję warunkową `if`. W naszym przykładowym szablonie `nxos.j2` dodaliśmy ją w dwóch miejscach: w pierwszej instrukcji używamy zmiennej `netflow`, a w drugiej zmiennej `l3_vlan_interfaces`. Blok instrukcji warunkowej zostanie wykonany tylko wtedy, gdy jej warunek przyjmie wartość `True`:

```

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

```

W playbooku zmiennej `netflow_enable` przypisaliśmy wartość `True` dla urządzenia `nx-os-v1`, a dla urządzenia `nx-os-v2` przypisaliśmy wartość `False`.

```

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

```

I w końcu możemy wykonać playbook:

```

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

PLAY [Pętle w szablonach] *****
*****

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

```

```
ok: [localhost]

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

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

Czy pamiętasz, że nazwy plików docelowych są określone na podstawie wzorca `{{ item.key }}.conf`? A zatem utworzone zostały dwa pliki o nazwach odpowiadających nazwom urządzeń:

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

Sprawdźmy podobieństwa i różnice obu tych plików konfiguracyjnych, aby upewnić się, że zostały w nich odpowiednio wprowadzone wszystkie zamierzone zmiany. Oba pliki powinny zawierać elementy statyczne, takie jak `feature ospf`, nazwy hostów i inne zmienne powinny zostać odpowiednio zastąpione, natomiast tylko plik `nx-osv-1.conf` powinien mieć włączony `netflow`, a także zawierać konfigurację interfejsu `vlan` warstwy 3.:

```
$ cat nx-osv-1.conf
hostname nx-osv-1
feature telnet
feature ospf
feature bgp
feature interface-vlan
feature netflow
username cisco password cisco role network-operator
vlan 100
vlan 200
vlan 300
interface 100
    ip address 192.168.10.1/24
interface 200
    ip address 192.168.20.1/24
interface 300
    ip address 192.168.30.1/24
```

Poniżej przedstawiłem zawartość pliku *nx-osv-2.conf*:

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

Fajne, prawda? Zastosowanie szablonów z pewnością może nam zaoszczędzić mnóstwo czasu na wykonanie pracy, która wcześniej wymagała wielokrotnego kopiowania i wklejania. Jeśli o mnie chodzi, to moduł szablonów był dla mnie dużym przełomem. Kilka lat temu już ten jeden moduł wystarczył, aby zmotywować mnie do nauki i rozpoczęcia korzystania z Ansible.

Podsumowanie

W tym rozdziale przedstawiłem framework automatyzacji Ansible udostępniany jako oprogramowanie typu *open source*. W przeciwieństwie do skryptów automatyzacji sieci opartych na bibliotece Pexpect i API Ansible do automatyzacji operacji na urządzeniach sieciowych udostępnia warstwę o wyższym poziomie abstrakcji, nazywaną playbookiem.

Ansible to framework do automatyzacji o ogromnych możliwościach, zdolny do zarządzania dużymi infrastrukturami. W tej książce koncentrujemy się na zarządzaniu urządzeniami sieciowymi, ale Ansible jest w stanie zarządzać serwerami, bazami danych, infrastrukturą chmurową i nie tylko. W tym rozdziale przedstawiłem jedynie niewielki ułamek jego możliwości. Jeśli uważasz, że Ansible jest narzędziem, o którym chciałbyś dowiedzieć się więcej, to doskonałym punktem wyjścia będzie jego dokumentacja. A gdybyś chciał się zaangażować, to społeczność Ansible jest przyjazna i chętnie wita nowych członków.

W rozdziale 5, pt. „Kontenery Dockera dla inżynierów sieci”, przedstawię Dockera i zacznę wprowadzać Cię w świat kontenerów.

| Skorowidz

A

ACI, Application Centric Infrastructure, 98, 105

ACL, access control list, 133, 356

adresy

IP

elastyczne, 358

maszyny wirtualnej, 386

MAC, 210

URL, 293

dynamiczne zmienne, 294

generowanie, 295

analiza

danych

LLDP, 259

sieciowych, 407

datagramu NetFlow, 266

wyników, 93

Ansible, 131

brak agentów, 141

Content Collection, 143

fakty sieciowe, 150

idempotentność, 142

implementacja list dostępu, 206

instalacja, 136

pętle operujące na słownikach, 154

pętle standardowe, 152

playbook, 139

rozszerzalność, 143

szablony, 156

użycie, 135

wersje frameworka, 134

wyrażenia warunkowe, 147

zagnieżdżanie ewidencji, 145

zarządzanie hostami, 137

zmienianie konfiguracji, 149

API, application programming interface, 56, 90

Azure, 372

konkretnego urządzenia, 304

oparte na Flasku, 289

strukturalne wyniki, 93

typu RESTful, 313

urządzeń, 302

Arista, 119

Cisco, 98

Juniper Networks, 110

zasobu sieciowego, 297

zawartości sieci, 299

Arista

eAPI, 120

Pyeapi, 124

ARNs, Amazon Resource Names, 346

asynchroniczne operacje

wejścia-wyjścia, 318

atak

typu Blokada usług, 204

typu DDoS, 366

typu Ping of Death, 204

typu exploit, 366

autoryzacja, 310

AWS, Amazon Web Services, 334

CLI, 337

CloudFormation, 353

CloudFront, 345

CloudFront CDN, 345

dostępność usług, 341

grupy bezpieczeństwa, 356

IAM, 337

dodawanie użytkownika, 338

uwierzytelnianie użytkownika, 339

użytkownicy, 338

konfiguracja, 335

konsola, 336

regiony, 341, 343, 344

sieciowe listy ACL, 356

strefy dostępności, 340, 344

usługi sieciowe, 369

AZ, availability zones, 340

Azure

administracja, 372

Cloud Shell, 374

- ExpressRoute, 402
- globalna infrastruktura, 380
- grupy zabezpieczeń, 399
- interfejs sieciowy, 385
- jednostki usług, 376
- konfiguracja, 371
- Load Balancer, 404
- maszyny wirtualne, 384–386
- peering sieci VNet, 390
- Python SDK, 387
- routing, 392
- równoważenie obciążenia, 404
- sieci VPN, 399
- sieci wirtualne VNet, 381
 - punkty końcowe, 389
 - tworzenie, 383
- subskrypcje, 373
- topologia sieci, 393
- usługi sieciowe, 369, 370, 372
- zasoby sieciowe, 387

B

- baza informacji zarządzania, MIB, 221
- Beats
 - pozyskiwanie danych, 426
- bezpieczeństwo
 - sieci, 38, 187
 - sieci wirtualnej, 397
 - VPC, 357
- BGP, Border Gateway Protocol, 362
- biblioteka
 - GitPython, 465
 - jsonrpclib, 122
 - Matplotlib, 230
 - matplotlib.dates, 233
 - ncclient, 99, 102
 - Netmiko, 84
 - Paramiko, 77
 - Pexpect, 67, 305
 - Pyeapi, 124
 - PyEZ, 115
 - Pygal, 238
 - PyGithub, 466
 - PySNMP, 225
 - Requests, 100, 422
 - Scrapli, 327
 - unittest, 504
 - Werkzeug, 310
- blokada GIL, 321

- brama
 - NAT, 360
 - działanie, 360
 - tworzenie, 361
 - Site-to-Site VPN, 401
 - VPN, 361, 399
 - VPN klienta, 401

C

- Cacti, 242
 - instalacja, 243
- CDN, content delivery network, 365
- centra danych, data centers, 27
 - brzegowe, 30
 - klasy korporacyjnej, 27
 - w chmurze, 28
- chmura
 - AWS, 334
 - Azure, 368
- ciągła
 - integracja, CI, 472, 474
 - integracja/ciągłe wdrażanie, CI/CD, 480
- Cisco
 - ACI, 105
 - ACI Tenants, 107
 - DevNet, 64
 - DevNet Sandbox, 64
 - IOS-XE, 108
 - Meraki, 109
 - NX-API, 99
- CLI, 56
 - stosowanie, 57
- Cloud Shell, 374
- CloudFormation
 - tworzenie VPC, 353
- CloudFront, 365
- CML, Cisco Modeling Labs, 61, 62
- Containerlab, 181

D

- DevOps, 181
- Direct Connect, 362
- Django, 287
- DNS, Domain Name System, 37
- Docker, 164
 - Engine, 477
 - instalowanie, 166

Docker
 opcje sieciowe, 177, 180
 polecenia, 167
 tworzenie aplikacji, 166, 168
 udostępnianie obrazu kontenera, 172
 użycie docker-compose, 175
 zalety, 165
drzewo ifEntry, 228
DSL, domain-specific language, 143
dynamiczne operacje sieciowe, 305
dziennik Syslog, 211

E

eAPI, 120
 przygotowanie, 120
 zastosowanie, 122
EC2, Elastic Compute Cloud, 335, 346
elastic adres IP, EIP, 358
Elastic
 Filebeat
 dziennik Cisco, 429
 NetFlow
 dane wejściowe, 430
 Stack, 408
 analiza danych sieciowych, 407
 Beats, 427
 jako usługa, 415
 widok danych, 420
 zarządzanie indeksem, 420
 zarządzanie stosem, 419
 zastosowanie, 417
 Elasticsearch, 412, 416
 generowanie indeksu, 418
 obsługa w Pythonie, 422
 wyszukiwanie, 432
 ELB, Elastic Load Balancing, 364
 ELK, Elasticsearch, Logstash, Kibana,
 Patrz Elastic Stack
etykieta, tag, 447
ExpressRoute, 402

F

Filebeat, 427
 instalacja, 427
Flask, 287, 289
 klient HTTPie, 291
 konfiguracja frameworka, 288
 obsługa adresów URL, 293
 operacje asynchroniczne, 308

 rozszerzenie httpauth, 310
 SQLAlchemy, 297
 tworzenie webowych API, 289
 uruchamianie w kontenerach, 313
 użycie formatu JSON, 296
 wersje, 289
 zastosowanie, 290
format JSON, 296
framework
 Ansible, 131
 Django, 287
 Flask, 287
 Nornir, 86
 pyATS, 512, 518
 pytest, 504
frameworki webowe, 285
funkcje, 52

G

gałąź, branch, 447, 455
Genie, 512
geografie, geographies, 381
GIL, global interpreter lock, 321
Git, 444, 445
 gałęzie, 455
 konfiguracja, 449
 terminologia, 447
 współdziałanie, 470
 zalety, 446
 zastosowanie, 451
GitHub, 448
 adres URL repozytorium, 458
 kopiowanie repozytorium, 462
 prośba o pobranie, 463
 przycisk Fork, 462
 repozytorium, 460
 synchronizacja repozytorium, 457
GitLab
 ciągła integracja, 472
 instalowanie, 476, 478
 pulpit, 479
 runnery, 480
 sieciowy przykład użycia, 489
 ustawienia SMTP, 479
 zastosowanie, 481
GitPython, 465
GNS3, 64
graf, 253
 nieskierowany, 254
 skierowany, 254

Graphviz, 250
 instalacja, 252
 tworzenie grafów, 253
grupy
 bezpieczeństwa AWS, 356, 398
 bezpieczeństwa sieci, NSG, 397

H

hipernadzorca, hypervisor, 65
hosty, hosts, 26
HTTPIe, 291

I

IaaS, Infrastructure-as-a-Service, 334, 369
IaC, 91
 modelowanie danych, 96
IAM, Identity and Access Management, 337
IBN, intent-based networking, 92
idempotentność, idempotency, 88, 142
identyfikator
 obiektu, OID, 221
 UUID, 308
IDF, Intermediate Distribution Frame, 28
IDN, intent-driven networking, 92
indeks
 Elastic Filebeat, 429
 Syslog, 425
infrastruktura
 jako kod, IaC, 91
 jako usługa, IaaS, 334, 369
instrukcje warunkowe, 160
 elif, 50
 else, 50
 if, 50
interaktywna powłoka, interactive shell, 43
interfejs
 API ACI, 106
 API XML, 116
 eAPI, 120
 NX-API, 99
 programowania aplikacji, API, 56, 90
 wiersza poleceń, CLI, 56
internet, 25
internet rzeczy, Internet of Things, 26
IP, Internet Protocol, 26, 37
 nagłówek IPv4, 38
 nagłówek IPv6, 38
IPFIX, 265

J

język
 DSL, 143
 Python, 23, 39
Juniper, 111
 NETCONF, 113
 PyEZ, 115

K

Kibana, 412, 414
 metryki, 441
 narzędzia dla programistów, 433
 tabela, 442, 431
 tworzenie indeksu, 419
 wizualizacja danych, 431, 436
 wykres kołowy, 437, 440
klasy, 52
komponenty sieciowe, 27
konfiguracje początkowe testów, test fixtures, 496
konsola
 AWS, 336
 CML, 62
kontenery
 Dockera, 163
 orkiestracja, 175
 uruchamianie Flaska, 313
 zastosowania w sieciach, 181
kontroler
 Cisco ACI, 107
 Cisco Meraki, 109
kopie zapasowe
 automatyczne tworzenie, 467
Kubernetes, 185
KVM, 65

L

laboratoria modelowania Cisco, 61
laboratorium sieciowe, 59
LAN, local area network, 25
listy dostępu, 205, 356
 adresów MAC, 210
 implementacja, 206
LLDP, Link Layer Discovery Protocol, 251
 obsługa protokołu, 256
 prezentowanie węzłów sieci, 256

Logstash, 415
konfiguracja, 417, 425
pozyskiwanie danych, 424

M

mapy, 47
maszyny wirtualne Azure, 385
adresy IP, 386
tworzenie, 384
Matplotlib, 230, 237
instalacja, 231
prezentacja wyników SNMP, 233
tworzenie wykresu, 231, 234, 237
MDF, Main Distribution Frame, 27
MIB, management information base, 221
model
danych, 96
klient-serwer, 34
NETCONF, 111
OSI, 31
YANG, 104
moduł, 53
asyncio, 323
pytest, 505
socket, 267
struct, 267
unittest, 501
monitorowanie sieci, 218, 249
Graphviz, 250
NetFlow, 266
oparte na przepływach, 265
przy użyciu ntop, 270
stosowanie Cacti, 242
użycie LLDP, 256
użycie SNMP, 221
wizualizacja danych, 230
MPLS, Multi-Protocol Label Switching,
110, 363

N

narzędzie
Azure AZ, 375
Batfish, 518
Cacti, 242
curl, 419
docker-compose, 175
Filebeat, 427
Graphviz, 250
HTTPIe, 291

Inspector, 436
MRTG, 242
NfSen, 271
ntop, 270
ping, 203
pip, 77, 100, 115
PIP, 337
RRDtool, 242
Scapy, 194
tcpdump, 196
NAT, 38, 334, 360
NETCONF, 98, 111
NetFlow, 265
Netmiko, 84
Nornir, 86
ntop, 270
instalacja, 271
monitorowanie ruchu, 273
rozszerzenia, 273
NumPy, 231
NX-API, 95, 99
NX-API Developer Sandbox, 101

O

obrazy kontenerów Docker, 172
obsługa
adresów URL, 293
zdarzeń, 326
OID, object identifier, 221
operacje
asynchroniczne, 308, 318, 331
synchroniczne, 331
operatory, 49
oprogramowanie jako usługa, SaaS, 334,
369
orkiestracja kontenerów, 175
osobisty żeton dostępu, PAT, 459

P

PaaS, Platform-as-a-Service, 334, 369
pakiet AWS CLI, 337
pakiety, 53
przechwytywanie, 198
Paramiko
instalacja, 77
interakcje idempotentne, 88
przegląd biblioteki, 77
zarządzanie serwerami, 81
zastosowanie, 80, 83

- parsowanie
 - danych NetFlow, 266
 - kodu XML, 499
- PAT, Personal Access Token, 360, 459
- peering
 - sieci VNet, 390
 - VPC, 356
- Pexpect, 305
 - instalacja, 67
 - przegląd biblioteki, 68
 - użycie biblioteki, 72–76
 - wady, 88
- pętla
 - for, 51
 - while, 51
- pętla Ansible, 152
- piaskownice
 - NX-API, 100, 101
 - Cisco DevNet, 106
- platforma jako usługa, PaaS, 334, 369
- playbook, 139, 258
 - topologia sieci, 263
- plik gitignore, 450
- pobranie, fetch, 447
- pokrycie testami, test coverage, 496
- polecenie
 - access-group, 92
 - access-list, 92
 - agalaxy install, 144
 - ansible-galaxy collection list, 143
 - ansible-playbook, 140, 171
 - apt, 387
 - aws configure, 339
 - aws ec2 describe-regions, 343
 - az login, 376
 - az network vnet list, 390
 - build, 169
 - checkout, 455
 - commit -m, 452
 - configuration, 85, 126
 - containerlab deploy, 183
 - containerlab destroy, 185
 - curl, 419
 - deactivate, 66
 - docker, 165
 - compose, 176
 - container ls, 167
 - exec, 167
 - image, 167
 - network, 167
 - network ls, 180
 - network rm, 180
 - rmi, 170
 - run, 167
 - dot, 252
 - echo, 152
 - enable, 80
 - filter-list, 92
 - git
 - config -l, 451
 - fetch, 471
 - log, 453
 - pull, 459
 - rm, 456
 - hostname, 189
 - ifconfig, 189
 - ip link, 189
 - merge, 456
 - netplan apply, 190
 - new_connection.recv(), 79
 - ping, 193
 - plt.show(), 231
 - Python3, 42
 - rank, 262
 - revert, 454
 - route, 192
 - service Filebeat, 428
 - sflowtool, 279
 - show, 83, 85, 125
 - ip interface brief, 93, 95
 - interface em1, 117
 - lldp neighbors, 260
 - lldp neighbors2, 258
 - management api http-commands, 120
 - running-config, 126, 128
 - version, 70, 100, 113, 328, 489
 - sudo, 202
 - terminal length 0, 80
 - time.sleep(), 81
 - traceroute, 193
- PowerShell, 374, 379
- program, *Patrz* narzędzie
- programowanie
 - oparte na testach, TDD, 493
 - zorientowane obiektowo, OOP, 52
- programowo definiowane sieci rozległe, 31
- protokoły
 - internetowe, 33
 - sieciowe, 34

- protokół
 - BGP, 362
 - IP, 23, 37
 - LLDP, 251
 - NETCONF, 98, 104, 111
 - SNMP, 221
 - SSHv2, 81
 - TCP, 23, 35
 - UDP, 36
- przechwytywanie pakietów, 198
- przepływ, flow, 265
 - próbkowany, 277
- przywracanie, checkout, 447
- pyATS, 512, 518
- Pyeapi, 124
 - instalacja, 124
 - zastosowanie, 125
- PyEZ, 115
 - instalacja, 115
 - zastosowanie, 117
- Pygal, 238, 242
 - instalacja, 238
 - prezentowanie wyników SNMP, 239
 - tworzenie wykresów SVG, 238
- PyGithub, 466
 - tworzenie kopii zapasowych, 467
- PySNMP, 225
- Python, 23, 39
 - analizowanie datagramów NetFlow, 266
 - bezpieczeństwo sieci, 187
 - blokada GIL, 321
 - framework Ansible, 131
 - frameworki webowe, 285
 - funkcje, 52
 - klasy, 52
 - moduł, 53
 - asyncio, 323
 - pytest, 505
 - socket, 267
 - struct, 267
 - unittest, 501
 - monitorowanie sieci, 218, 249
 - obsługa Elasticsearch, 422
 - obsługa UFW, 215
 - operatory, 49
 - pakiety, 53
 - rozszerzanie ntop, 273
 - SDK, 337
 - sterowanie przepływem, 50
 - środowisko wirtualne, 66

- testowanie, 504
- tworzenie usług webowych, 283
- typy danych, 43
- uruchamianie programów, 42
- wizualizacja danych, 230

R

- referencja, ref, 447
- repozytorium, repository, 447
 - Docker Hub, 173
- REST, representational state transfer, 283
- RESTful, 283
- routing, 349, 351, 392, 396
 - IP, 39
 - w sieciach wirtualnych, 392
- RPC, Remote Procedure Call, 111
- runner, 480

S

- SaaS, Software-as-a-Service, 334, 369
- scalenie, merge, 447
- Scapy, 194
 - instalowanie, 194
 - przechwytywanie pakietów, 198
 - przykłady użycia, 196
- Scrapli, 327
 - operacje asynchroniczne, 329
 - przykład zastosowania, 327
 - sterowniki, 329
- SDK, software development kit, 91
- SDN, Software-Defined Networking, 334
- sekwencje, 44
- sFlow, sampled flow, 265, 277
 - RT, 278, 280
- Sflowtool, 278
- sieci
 - intuicyjne, 92
 - w chmurze AWS, 334
 - w chmurze Azure, 368
 - wirtualne Azure, VNets, 381
 - grupy zabezpieczeń, 397
 - punkty końcowe, 389
 - tablice routingu, 396
 - trasowanie, 392
 - tworzenie, 383
- sieć
 - Clos, 29
 - dostarczania treści, CDN, 365

- hosta kontenerów, 178
- lokalna, 25
- mostkowa, 179
- skanowanie portów TCP, 199
- SLA, service-level agreement, 362
- słownik, dictionary, 47
- słowo kluczowe
 - async, 324
 - await, 324
 - class, 52
 - def, 52
- SNMP, 221
 - działanie protokołu, 221
 - implementacja silnika, 225
 - konfigurowanie opcji, 223
 - prezentacja wyników
 - użycie Matplotlib, 233
 - prezentowanie wyników
 - użycie Pygal, 239
- SQLAlchemy, 297
- SSH, 75
- strefy dostępności, AZ, 340
- SVG, Scalable Vector Graphics, 238
- Syslog, 211
- system
 - operacyjny
 - NOS, 182
 - VyOS, 128
 - zarządzania treścią, 444
- szablony
 - Ansible, 156
 - Jinja
 - instrukcje warunkowe, 160
 - pętle, 159
 - zmienne, 158

Ś

- ściągnięcie, pull, 447

T

- tablice
 - asocjacyjne, associated arrays, 48
 - mieszające, hashing table, 48
 - routingu, 349, 351, 393, 396
- TCL, Tool Command Language, 67
- TCP, Transmission Control Protocol, 24, 35
 - cechy protokołu, 35
 - funkcje, 35

- komunikaty, 36
- nagłówki, 35
- transfer danych, 36
- TDD, test-driven development, 493
- test
 - funkcjonalny, functional test, 496
 - integracyjny, integration test, 496
 - jednostkowy, unit test, 495
 - systemowy, system test, 496
- testowanie
 - Ansible, 511
 - bezpieczeństwa, 510
 - dostępności, 508
 - konfiguracji sieciowych, 511
 - opóźnień sieci, 509
 - transakcji, 510
- topologia
 - jako kod, 496
 - laboratorium, 68, 99, 137, 145, 189, 220, 252, 288, 410, 477, 497
 - sieci, 261, 263, 264
 - sieci Azure, 393
- transfer danych, 36
- translacja adresów
 - portów, PAT, 360
 - sieciowych, NAT, 38, 334, 360
- trasowanie, *Patrz* routing
- tworzenie
 - aplikacji w Dockerze, 166
 - bramy NAT, 361
 - grafów, 253
 - kopii zapasowych, 467
 - maszyny wirtualnej, 384
 - modelu danych, 97
 - sieci wirtualnej, 381, 383
 - usług webowych, 283
 - VPC, 348, 353
 - wykresów, 231, 234, 237
 - 2D, 230
 - SVG, 238
 - zasobów sieciowych, 387
- typ None, 44
- typy liczbowe, 44

U

- UDP, User Datagram Protocol, 25, 36
 - nagłówki, 37
- UFW, Uncomplicated Firewall, 187, 211
 - obsługa w Pythonie, 215

urządzenia

- fizyczne, 59
- wirtualne, 59

usługa

AWS

- Amazon GuardDuty, 366
- CDN, 365
- CloudFront, 365
- EC2, 335, 342
- Shield, 366
- Transit VPC, 366
- WAF, 366

Azure

- DDoS Protection, 405
- DNS, 405
- sieciowa usługa kontenerowa, 405
- VNet TAP, 405
- DNS Route 53, 365
- ELB, 364
- Filebeat, 428
- IAM, 337, 346
- VNet

- punkty końcowe, 389

usługi

- sieciowe Azure, 370, 372
- webowe, 283

uwierzytelnianie, 310, 378

V

VIRL, Virtual Internet Routing Lab, 61

VLAN, virtual local area network, 214

VNet, 382

- peering sieci, 390

VNets, Azure virtual networks, 381

VPC, virtual private cloud, 345, 346

- bezpieczeństwo, 357
- stosowanie peeringu, 355
- tablica tras, 349
- tworzenie, 348, 353

VPG, virtual private gateway, 362

VPN, virtual private network, 361, 399

- połączenie z VPC, 362

VyOS, 128

W

warstwy modelu OSI, 32

wieloprocusowość, 320

wielowątkowość, 321

wirtualna

- brama prywatna, VPG, 362
- chmura prywatna, *Patrz* VPC

wirtualne

- laboratorium, 59
- sieci lokalne, VLAN, 214
- sieci prywatne, VPN, 361, 399
- środowisko Pythona, 66

wizualizacja danych, 230

wtyczka

- asynssh, 329
- LibSSH, 140

wykres

- kołowy, 237, 241
- kołowy w Kibanie, 437, 440
- liniowy, 232

wykresy SVG, 238

wyrażenia

- regularne, 212
- warunkowe w Ansible, 147

wywołanie NX-API, 95

X

XML, Extensible Markup Language, 111

Y

YANG, 98, 104

Z

zapora sieciowa AWS WAF, 366

zarządzanie

- treścią, *Patrz* Git
- zmianą, 473

zatwierdzenie, commit, 447

zbiory, 48

zmienne, 158

- URL, 294

Ż

źródło danych wejściowych, 245

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

Chcesz mieć sieć najnowszej generacji? Python jest dla Ciebie!

Programiści cenią Pythona za wyrazistość i zwięzłość kodu, a także za bogatą kolekcję narzędzi i bibliotek. Z tych zalet mogą korzystać również inżynierowie sieci. Programistyczne zarządzanie siecią stanowi odpowiedź na rozwój technologii — tradycyjny interfejs wiersza poleceń i pionowo zintegrowane metody kontroli sieci nie są już najlepszymi sposobami zarządzania współczesnymi sieciami.

Oto uzupełnione i zaktualizowane wydanie bestsellerowego przewodnika dla inżynierów sieci. Dzięki niemu przejdziesz trudną (ale ekscytującą!) drogę od tradycyjnej platformy do platformy sieciowej opartej na najlepszych praktykach programistycznych. Zaczyniesz od zagadnień podstawowych, aby następnie zagłębić się w tajniki stosowania bibliotek Pexpect, Paramiko czy Netmiko do komunikacji z urządzeniami sieciowymi. W kolejnych rozdziałach znajdziesz solidny przegląd różnych narzędzi wraz ze sposobami ich użycia: Cisco NX-API, Meraki, Juniper PyEZ, Ansible, Scapy, PySNMP, Flask, Elastic Stack i wielu innych. Rozeznasz się również w kwestiach związanych z kontenerami Dockera, a także usługami sieciowymi chmur AWS i Azure. Lektura tej książki pozwoli Ci się w pełni przygotować na następną generację sieci!

W książce między innymi:

- interakcja Pythona z urządzeniami sieciowymi
- uzyskiwanie informacji o sieci i analiza danych sieciowych
- tworzenie wysokopoziomowych API
- korzystanie z biblioteki AsyncIO
- paradygmat programowania sterowanego testami w Pythonie
- zastosowanie GitLab w praktykach DevOps w kontekście zagadnień sieciowych

Eric Chou od ponad 20 lat zajmuje się inżynierią sieci. Pracował nad sieciami takich firm jak Amazon i Microsoft. Jest również autorem bestsellerów poświęconych bezpieczeństwu, przetwarzaniu danych i programowaniu. Posiada wiele amerykańskich patentów w dziedzinie telefonii IP i sieci.

Helion	KOD KORZYŚCI Śięgnij po więcej! ▶	
 helion.pl	ISBN 978-83-289-0280-0	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788328 902800	
Cena: 129,00 zł		

<packt>