

SECOND EDITION

<packt>

Network Automation Cookbook

Over 100 recipes to effectively configure and manage network infrastructure with Ansible

Christian Adell | Jeffrey Kala | Karim Okasha



Network Automation Cookbook

Second Edition

Over 100 recipes to effectively configure and manage
network infrastructure with Ansible

Christian Adell

Jeffrey Kala

Karim Okasha



Network Automation Cookbook

Second Edition

Copyright © 2025 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 authors, 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.

Portfolio Director: Kartikey Pandey

Relationship Lead: Prachi Rana

Project Manager: Sonam Pandey

Content Engineer: Sarada Biswas

Technical Editor: Simran Ali

Copy Editor: Safis Editing

Indexer: Rekha Nair

Proofreader: Sarada Biswas

Production Designer: Ponraj Dhandapani

Growth Lead: Amit Ramadas

First published: April 2020

Second edition: August 2025

Production reference: 2200825

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-798-1

www.packtpub.com

I want to dedicate this book to all my colleagues at Network to Code. They gave me the environment to try and develop the skills needed to write this book.

– Christian Adell

To my incredible wife, Jacqueline, and my two amazing sons, Jack and Jude. Thank you for your unwavering support, endless encouragement, and for always being my biggest cheerleaders. This book would not have been possible without your love.

And to my co-author, Christian, I extend my deepest gratitude. Thank you for pushing me out of my comfort zone and inspiring me to take on this journey. Your guidance was invaluable, and I am a better writer because of it.

– Jeff Kala

Contributors

About the authors

Christian Adell is a principal architect at NTC. He is focused on building network automation solutions for diverse use cases, with great emphasis on open source software. He is passionate about learning and helping others to be happier, and also has more hobbies than hours in the day, so working remotely from Barcelona gives him the time and the space to achieve his dreams. Christian is a co-author of O'Reilly's "Network Programmability and Automation" book and a co-author of the "Modern Network Observability" and "Network Automation with Nautobot" books by Packt.

Jeffrey Kala is a principal architect at Network to Code and holds a BSc in network communication management. He has transformed his career from a traditional network engineer into a network automation architect. His journey started in network engineering, working for a large cellular provider. As the network he helped maintain grew in complexity and size, he explored and implemented automation techniques that resulted in a renewed career trajectory in network automation.

Karim Okasha is a network consultant with over 15 years of experience in the ICT industry. He specializes in the design and operation of large telecom and service provider networks and has lots of experience in network automation. Karim has a bachelor's degree in telecommunications and holds several expert-level certifications, such as CCIE, JNCIE, and RHCE. He is currently working at Red Hat as a network automation consultant, helping large telecom and service providers design and implement innovative network automation solutions. Prior to joining Red Hat, he worked for Saudi Telecom Company, as well as Cisco and Orange S.A.

About the reviewers

Tony Dubiel is a principal specialist solution architect and team lead for Ansible Networking at Red Hat, focusing on network automation for the North American public sector. Living in the Raleigh-Durham-Chapel Hill area, this U.S. Air Force veteran brings over 26 years of industry experience to his role. His career is dedicated to helping customers onboard network use cases into automation using Ansible best practices.

At Cisco, he served in many roles over 12 years and most recently was a DevNet Automation Advocate for Cisco solutions. His deep expertise in large-scale automated deployments using Ansible is validated by premier certifications, including a triple CCIE (#10844) and several Red Hat Certified Specialist in Ansible Automation certifications.

Josh VanDeraa is a 20 year networking veteran who has been doing Network automation for the past 10 years he has worked in a large enterprise retail, travel, managed services, and most recently professional services industries. He has worked on networks of all sizes bringing multiple different network automation solutions to the table to drive real value with Python, Ansible, and the Python web frameworks.

*Thank you to my family for allowing me to put the time into reviewing the content for the book.
Without their support I would not be able to help the community.*

I would also like to thank Jeff and Christian, for taking the time to share their experiences about Ansible into the community. This is not an easy subject to tackle, but they have brought real, practical experiences into writing for you to consume.

Table of Contents

Preface	xv
Your Book Comes with Exclusive Perks – Here’s How to Unlock Them	xx
 Part 1: Getting Started with Ansible	 1
Chapter 1: Why Ansible for Network Automation?	3
Understanding network automation	4
From SDN to NetDevOps • 4	
Network automation drivers • 6	
The building blocks for network automation • 7	
Discovering Ansible	10
Ansible’s characteristics • 11	
The role of Ansible in a network automation strategy • 12	
Summary	13
 Chapter 2: Building Blocks of Ansible	 15
Technical requirements	16
Installing Ansible	18
Building the Ansible inventory	20
Controlling Ansible options	23
Defining Ansible variables	25
Building a playbook	29
Using Ansible conditionals	31
Using Ansible loops	33
Securing secrets with Ansible Vault	37
Using Jinja2 templates with Ansible	39

Using Ansible network-related filters	42
Using Ansible tags	45
Summary	48
Chapter 3: Advanced Ansible Concepts	49
Using Ansible roles	50
Using Ansible collections	53
Running Ansible in check mode	57
Using advanced YAML syntax	60
YAML multiline strings • 60	
YAML anchors and aliases • 64	
Controlling execution strategies in Ansible	65
The free strategy • 67	
The host_pinned strategy • 68	
The serial keyword • 70	
Configuring fact caching in Ansible	75
Creating custom Python filters for Ansible	79
Summary	84
Part 2: Managing Network Infrastructure with Ansible	85
Chapter 4: Building Data Center Networks with Arista	87
Augmenting the Ansible network inventory	89
Connecting to Arista devices	92
Migrating connections from SSH to HTTP(S) by utilizing the Arista eAPI	95
Creating and managing users and authentication options	98
Configuring generic system options on Arista devices	101
Configuring Arista interfaces using Ansible	103

Advanced fabric configuration for Arista Data Center	111
Generating the underlay configuration • 111	
Generating the overlay configuration • 117	
Generating the remaining VXLAN configuration • 119	
Finalizing BGP configuration to the core	123
Pushing configuration to Arista devices	125
Collecting Arista device facts using Ansible	128
Running arbitrary commands on Arista devices	130
Summary	131
 Chapter 5: Automating a Juniper Network Core Using Ansible	 133
Augmenting the Ansible network inventory	135
Connecting to Juniper devices	136
Migrating from the CLI to NETCONF	140
Creating and managing users and authentication options	144
Configuring generic system options on Juniper devices	147
Configuring Juniper interfaces	149
Advanced routing configuration using the template module	153
Pushing configuration to Juniper devices	158
Collecting Juniper device facts using Ansible	160
Running arbitrary commands on Juniper devices	162
Parsing Junos commands using parse_xml	163
Summary	167
 Chapter 6: Managing Nokia SR Devices Using Ansible	 169
Augmenting the Ansible network inventory	171
Connecting to Nokia devices	173
Creating and managing users and authentication options	176
Configuring generic system options on Nokia devices	179
Configuring Nokia interfaces	181
Advanced routing configuration using template module	190

Generating the overlay configuration	195
Generating the remaining VXLAN configuration	199
Finalizing BGP configuration to the core	204
Pushing the configuration to Nokia devices	206
Collecting Nokia device facts using Ansible	208
Running arbitrary commands on Nokia devices	210
Summary	213

Chapter 7: Administering a Multi-Vendor Network with NAPALM and Ansible **215**

Installing NAPALM with Ansible	217
Retrieving network facts with NAPALM	222
Retrieving BGP neighbor information from network devices	226
Configuring network devices with NAPALM	228
Validating and auditing a network with NAPALM	232
Building reports with NAPALM and Ansible	234
Summary	236

Part 3: Managing Cloud Networking with Ansible **239**

Chapter 8: Deploying and Operating AWS Networking **241**

Installing the AWS Python SDK	243
Building an Ansible inventory	244
Authenticating to your AWS account	246
Deploying VPCs using Ansible	250
Deploying Subnets using Ansible	254
Deploying IGWs using Ansible	256
Controlling routing within a VPC using Ansible	258
Deploying NACLs using Ansible	262

Deployment validation using Ansible	264
Decommissioning resources on AWS using Ansible	266
Summary	269
Chapter 9: Deploying and Operating Azure Networking	271
Installing the Azure Python SDK	273
Building an Ansible inventory	274
Authenticating to your Azure account	276
Creating a Resource Group	280
Creating virtual networks	282
Creating subnets	285
Building user-defined routes	286
Deploying network security groups	290
Deployment validation	294
Decommissioning Azure resources	297
Summary	299
Chapter 10: Deploying and Operating GCP Networking	301
Installing the GCP Python SDK	303
Building the Ansible inventory	305
Authenticating to your GCP account	306
Creating GCP VPC networks	310
Creating subnets	315
Deploying firewall rules in GCP	317
Deploying VMs in GCP	321
Adjusting routing within a GCP VPC	328
Validating a GCP deployment	330
Decommissioning GCP resources	333
Summary	337

Part 4: Integrating Ansible 339

Chapter 11: Network Validation with Batfish 341

Installing Batfish	344
Integrating Batfish with Ansible	346
Pulling in the network config	348
Initializing the network snapshot with Ansible	350
Collecting network facts from Batfish	353
Validating the network with Batfish assertions	356
Summary	362

Chapter 12: Building a Network Inventory with Nautobot 365

Installing Nautobot	369
Integrating Nautobot with Ansible	371
Populating locations in Nautobot	377
Populating devices in Nautobot	380
Populating connections in Nautobot	384
Populating IP prefixes in Nautobot	388
Populating IP addresses in Nautobot	390
Populating cloud networks in Nautobot	394
Using Nautobot as an Ansible dynamic inventory	402
Generating configurations using Nautobot	406
Summary	408

Chapter 13: Simplifying Automation with AWX 411

Installing AWX	413
Managing users and teams in AWX	417
Creating a network inventory on AWX	421
Managing network credentials on AWX	426
Creating projects in AWX	430

Managing the execution environment in AWX	433
Creating templates in AWX	435
Creating workflow templates in AWX	440
Running automation tasks using the AWX API	444
Summary	446
Chapter 14: Event-Driven Automation with Ansible	449
Installing EDA-Server	452
Creating projects in EDA-Server	455
Creating a Rulebook	456
Creating a decision environment	459
Activating a Rulebook in EDA-Server	461
Preparing AWX, Nautobot, and EDA-Server	463
Executing an event-driven workflow	473
Summary	477
Chapter 15: Terraform and Ansible Dancing Together	479
Deploying dynamic infrastructure with Terraform	481
Integrating Terraform outputs with Ansible	488
Building an Ansible inventory from the Terraform state • 488	
Executing an Ansible playbook from within Terraform • 489	
Using an AWS EC2 Ansible dynamic inventory plugin	492
Automating firewall rule provisioning with Ansible	494
Integrating Terraform with AWX	498
Cleaning up dynamic infrastructure with Terraform	507
Summary	508
Chapter 16: Appendix - The Lab	511
The lab environment setup	512
Hardware Requirements • 513	
Software Requirements • 514	

Step 0. Forking the Lab Environment from Git Repository • 515	
<i>Why Fork?</i> • 515	
Step 1. Building the Lab Environment • 516	
<i>Manual Server Provisioning</i> • 517	
<i>Automated Server Provisioning</i> • 524	
Step 2. Removing the Lab Environment • 527	
Summary	527
Index	529

Other Books You May Enjoy	538

Preface

When the first edition of *Network Automation Cookbook* was published, the networking industry was at a turning point. Teams around the world were beginning to shift from manual, error-prone configurations toward automation-driven operations. Ansible, with its simple and human-readable approach, stood out as a tool that could bridge the gap between traditional network engineers and modern infrastructure-as-code practices.

Since then, the landscape has evolved dramatically.

In just a few years, Ansible has matured into a cornerstone of network automation. Vendors have expanded support, the community has grown, and new tools have emerged to extend its reach from dynamic inventory management and multi-vendor abstraction layers to event-driven automation and cloud-native networking. At the same time, automation practices have moved beyond one-off tasks and scripts to integrated, production-grade workflows that span physical, virtual, and cloud infrastructure.

This second edition is a reflection of that evolution.

We've completely refreshed the book to go beyond just automating CLI commands or pushing configurations. This edition is designed to guide you through building scalable, maintainable, and vendor-agnostic automation solutions using Ansible as the foundation. You'll find practical recipes across on-premises data center automation, public cloud networking, source-of-truth integration, network validation, event-driven workflows, and the increasingly common need to combine Ansible with Terraform for full stack infrastructure automation.

Each chapter is hands-on, actionable, and focused on solving real-world problems. Whether you're rendering configurations for Arista, automating a Juniper core, managing Nokia SR Linux devices, or deploying networking in AWS, Azure, or GCP, this book provides a clear and tested approach.

New additions such as Ansible EDA to explore event-driven Ansible for reactive workflows, Nautobot (similar to what you could do with NetBox or Infracore) for dynamic inventory and source-of-truth-driven automation, and how Terraform and Ansible can coexist to deliver end-to-end automation in heterogeneous environments—take this edition far beyond the basics.

If you're a network engineer looking to adopt automation, an infrastructure architect aiming to scale network operations, or a developer integrating network changes into CI/CD pipelines, this cookbook is for you.

We hope this second edition helps you move confidently from manual operations to modern, automated network engineering. The recipes in this book are based on real-world experience, field-tested tools, and a passion for making networking more reliable, repeatable, and programmable.

Let's build the future of networking—one playbook at a time.

Who this book is for

This book is for network and DevOps engineers seeking to enhance their network operations via network automation, leveraging Ansible and other tools that complement it. Basic experience with YAML, Python, and network automation concepts is recommended to get the most out of this book.

What this book covers

Chapter 1, Why Ansible for Network Automation?, details the different ways of installing Terraform manually, with scripts, or by using a Docker container, and it also details the Terraform migration configuration process.

Chapter 2, Building Blocks of Ansible, concerns the writing of Terraform configurations for a provider, variables, outputs, built-in functions, condition expressions, Yaml file manipulation, and pre and post conditions.

Chapter 3, Advanced Ansible Concepts, shows you how to build dynamic environments by going further with Terraform configuration writing using loops, maps, and collections.

Chapter 4, Building Data Center Networks with Arista, explores how to use Terraform with external data and local files, and how to execute local programs and scripts with Terraform.

Chapter 5, Automating a Juniper Network Core Using Ansible, explains Terraform State management, including reading, moving, deleting, and importing resources into the Terraform State.

Chapter 6, Managing Nokia SR Devices Using Ansible, explains the use of Terraform's CLI to validate the configuration, use outputs, destroy resources provisioned by Terraform, use workspaces, generate dependency graphs, and debug the execution of Terraform.

Chapter 7, Administering a Multi-Vendor Network with NAPALM and Ansible, covers the creation, use, and sharing of Terraform modules, and shows testing module practices.

Chapter 8, Deploying and Operating AWS Networking, illustrates the use of Terraform in a practical scenario with the cloud provider Azure. It covers topics such as authentication, remote backends, ARM templates, Azure CLI execution, and Terraform configuration generation for an existing infrastructure.

Chapter 9, Deploying and Operating Azure Networking, provides a starting point for provisioning AWS and GCP infrastructure using Terraform, and includes details on these providers, authentication, and remote backend storage.

Chapter 10, Deploying and Operating GCP Networking, explains how to use Terraform to create Docker containers and deploy Kubernetes resources.

Chapter 11, Network Validation with Batfish, details Terraform configuration testing practices using several tools like Tfsec, OPA, terraform-compliance, and Pester.

Chapter 12, Building a Network Inventory with Nautobot, discusses topics that go further with Terraform, such as the execution of Terraform configuration tests, zero-downtime deployment, Terraform wrappers with Terragrunt, checking configuration using Git-Hook, and using the Terraform CDK as a developer.

Chapter 13, Simplifying Automation with AWX, explores local Terraform automation processes, and implementing a CI/CD pipeline to apply Terraform configuration automatically.

Chapter 14, Event-Driven Automation with Ansible, explains how to use Terraform Cloud to run Terraform in a team with the sharing of Terraform modules in a private registry, the use of remote backends for Terraform State, migrating Terraform State, running Terraform remotely, and integrating cost estimation.

Chapter 15, Terraform and Ansible Dancing Together, lists several Terraform errors and explains how to resolve them.

Chapter 16, Appendix - The Lab, contains a Terraform Cheat Sheet command line and my Terraform resources list.

To get the most out of this book

Start setting up your lab environment as described in *Chapter 16*; all the recipes can be reproduced from it. It is still possible to follow the recipes without using the lab.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Run the `pb_arista_facts.yml` playbook to validate the operational state of our fabric.”

A block of code is set as follows:

```
---
- name: "CREATE OUTPUT CONFIGS DIRECTORY"
  ansible.builtin.file:
    path: "{{ config_output_root_dir }}"
    state: "directory"
  run_once: yes
```

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

```
ansible-playbook pb_validate_facts.yml
PLAY [Collect and Validate Arista DC Fabric Facts] *****
TASK [Collect Arista Device Facts] *****
ok: [ceos-01]
ok: [ceos-02]
ok: [ceos-03]
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Create a new organization, Network, for all network teams—as shown in the following screenshot—by selecting **Access Management** and **Organizations** from the left panel and clicking the **Create Organization** button.”



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 or have any general feedback, please email us at customercare@packt.com and mention the book’s title in the subject of your message.

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

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

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

Your Book Comes with Exclusive Perks – Here's How to Unlock Them

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name. Ensure it's the correct edition.

Note: Keep your purchase invoice ready before you start.

UNLOCK NOW



Enhanced reading experience with our Next-gen Reader:

 **Multi-device progress sync:** Learn from any device with seamless progress sync.

 **Highlighting and notetaking:** Turn your reading into lasting knowledge.

 **Bookmarking:** Revisit your most important learnings anytime.

 **Dark mode:** Focus with minimal eye strain by switching to dark or sepia mode.

Learn smarter using our AI assistant (Beta):

 **Summarize it:** Summarize key sections or an entire chapter.

✦ **AI code explainers:** In the next-gen Packt Reader, click the **Explain** button above each code block for AI-powered code explanations.



Note: The AI assistant is part of next-gen Packt Reader and is still in beta.

Learn anytime, anywhere:

 Access your content offline with DRM-free PDF and ePub versions—compatible with your favorite e-readers.

Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:



Next-gen Packt Reader



AI assistant (beta)



DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

How to unlock these benefits in three easy steps

Step 1

Keep your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.



Note: Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

Step 2

Scan this QR code or go to packtpub.com/unlock.



On the page that opens (which will look similar to *Figure 1* if you're on desktop), search for this book by name. Make sure you select the correct edition.



Subscription



0



Explore Products

Best Sellers

New Releases

Books

Videos

Audiobooks

Learning Hub

Newsletter Hub

Free Learning

Discover and unlock your book's exclusive benefits

Bought a Packt book? Your purchase may come with free bonus benefits designed to maximise your learning. Discover and unlock them here

Discover Benefits

Sign Up/In

Upload Invoice

[Need Help?](#)

1. Discover your book's exclusive benefits

CONTINUE TO STEP 2

2. Login or sign up for free



3. Upload your invoice and unlock



Figure 1: Packt unlock landing page on desktop

Step 3

Sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



Note: If you are still facing issues, reach out to customercare@packt.com.

Share your thoughts

Once you've read *Network Automation Cookbook, Second Edition*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

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

Part 1

Getting Started with Ansible

The first part of the book lays the groundwork for everything that follows by introducing Ansible as a powerful and flexible automation tool, not just for networking but for infrastructure and systems in general.

You'll start by understanding what makes Ansible different, how it fits into modern automation workflows, and why its simplicity, readability, and agentless architecture have made it a widely adopted solution across IT teams.

We will then dive into the core building blocks of Ansible: inventories, variables, playbooks, templates, and Ansible Vault for secrets management. You'll also learn about key concepts such as conditionals, loops, and Jinja2 templating, which are critical tools for building dynamic, reusable automation.

Finally, we will explore more advanced features that enable production-grade usage: roles and collections for modularity, execution strategies for performance, and techniques for extending Ansible with custom filters and plugins.

Whether you're new to Ansible or looking to reinforce your fundamentals, this part provides the essential skills and understanding needed to confidently automate complex environments before applying those skills to real-world network automation in the chapters ahead.

This part of the book includes the following chapters:

- *Chapter 1, Why Ansible for Network Automation?*
- *Chapter 2, Building Blocks of Ansible*
- *Chapter 3, Advanced Ansible Concepts*

1

Why Ansible for Network Automation?

Before getting started with looking at how to use **Ansible** for network automation, it's important to understand why Ansible has become one of the most popular tools in the network automation space. It hasn't been an abrupt change, but a progressive adoption of more than one decade has positioned Ansible as one of the favored ways to manage networks nowadays.

To explain this process, the chapter is divided into two sections:

- *Understanding network automation*: Provides a historical view of how network automation exploded after the **software-defined networking (SDN)** hype in the late 2010s, and which benefits it provides compared with traditional network management
- *Discovering Ansible*: Explains the key characteristics of Ansible and how it compares with other alternatives

This is the only chapter in the book that doesn't include technical recipes, but we strongly encourage you not to skip it. Gaining a solid understanding of the context will enhance your grasp of the technical topics covered in the following chapters.

So, before going all-in with Ansible, let's recap how network automation has become the holy grail of network operation nowadays.

Understanding network automation

There is no official definition of network automation. Even asking your favorite large language model-based system (such as ChatGPT) might yield only a vague description. Why? Because network automation is a broad term encompassing many facets. We define it as the *streamlining of all network-related tasks*.

These network-related tasks span a wide scope, including *configuration, management, monitoring, troubleshooting, capacity planning, testing*, and more, across all types of network environments—from physical and virtual devices to cloud networking services. This marks a departure from traditional network operations, which, from the 1980s until recently, relied heavily on human-driven **command-line interfaces (CLIs)** that required constant manual intervention.

While this manual approach worked reasonably well for a time, the early 2000s brought increased network scale and complexity. This spurred network operators to seek alternatives that could support programmable interactions with network devices, enabling a more automated approach.

Note



A milestone reference here is RFC 3535

(<https://datatracker.ietf.org/doc/html/rfc3535>), *Overview of the 2002 IAB Network Management Workshop* (which was the origin of the **Network Configuration Protocol (NETCONF)** interface).

Yet limitations extended beyond network management. There was little flexibility to influence packet forwarding independently of control protocols (such as **Open Shortest Path First (OSPF)** or **Border Gateway Protocol (BGP)**). In this context—and with the emergence of the first hyper-scalers, such as **Amazon Web Services (AWS)** in 2006—the need to evolve how we interact with and program networks became clear. This led to the advent of SDN, a movement that can be seen as a foundational step toward modern network automation.

From SDN to NetDevOps

Let's clarify from the start: SDN is not the same as network automation. SDN focuses on programming the network's forwarding plane—that is, changing how packet movement decisions are made. To achieve this, a programmable interface to communicate with network devices is essential.

The OpenFlow protocol was at the forefront of SDN, and although it ultimately faced challenges that limited its long-term success, it generated substantial excitement and acted as a catalyst for the network industry. OpenFlow highlighted significant limitations, particularly the need for robust **application programming interfaces (APIs)**, a key factor for effective network automation.



Note

To get a more detailed understanding of the evolution of network programmability and how it influences network automation, we recommend *Network Programmability and Automation, 2nd edition*, by Oswalt, M., Adell, C., Lowe, S. S., and Edelman, J..

Simultaneously, the exponential growth of networks to support global connectivity—driven by the rise of cloud computing and the expansion of the internet—made the limitations of human-only network operation increasingly evident, signaling the need for a paradigm shift.

Fortunately, this push for operational improvement wasn't confined to networking. Across IT, a new approach emerged: the DevOps movement. As the name suggests, DevOps combines development and operations to streamline application delivery, bridging gaps and enhancing traditional practices on both sides.

From an operations perspective, DevOps introduced transformative concepts for managing **infrastructure as code (IaC)**, along with software development best practices such as version control and **continuous integration/continuous deployment (CI/CD)**. This approach redefined how IT infrastructure—computing, storage, and networking—could be managed, enabling unattended interaction. However, not all infrastructure was ready for seamless programmatic integration. While cloud-based **Infrastructure as a Service (IaaS)** offered a more accessible interface, behind the scenes, or in many IT environments, the OS shell or CLI remained the primary interaction method.

To support this new automated approach, three popular configuration management tools emerged in quick succession:

- **Puppet**, released in 2005
- **Chef**, launched in 2009
- **Ansible**, released in 2012

In the *Ansible's characteristics* section, we'll delve into the distinctions among these tools; however, to preview, Ansible proved the most successful for network automation (Puppet/Chef's reliance on agents was a non-starter for most network hardware).

With both the motivation and the tools in place, the stage was set for the rise and consolidation of the NetDevOps movement as a specialized branch of DevOps. NetDevOps brings DevOps principles into network engineering, enabling faster, more reliable, and more efficient network provisioning and management.



Note

In this book, we will share with you not only Ansible's best practices but also how they connect to NetDevOps principles.

You may already have inferred some of the benefits of network automation and the NetDevOps approach, but delving a bit deeper into the main goals should solidify your understanding.

Network automation drivers

Network automation extends beyond simply running tasks unattended (as discussed in the *The building blocks for network automation* section). It is a comprehensive approach to managing network infrastructure with minimal human intervention. When implemented effectively, it enables IT infrastructure to meet today's demands for scalability, reliability, and efficiency. Let's recap the primary drivers behind the adoption of network automation.

From a high-level perspective, we can group the main drivers behind network automation into four areas:

- **Higher reliability and operational efficiency:** The capability to perform numerous unattended operations enables the scaling of best practices, ensuring reliability through reproducible tasks that operate independently of the person initiating them. Programmatic executions are deterministic, providing consistent results every time (ideally, idempotently, yielding the same result in every execution). For example, network automation allows us to implement a standardized provisioning process across thousands of devices simultaneously. After deployment, automated validation processes can collect and analyze data to confirm successful execution. This same approach is invaluable during unexpected issues, where automated data gathering and analysis offer critical insights, aiding operations engineers in troubleshooting. Additionally, advanced network automation architectures, such as closed-loop automation, introduce self-healing mechanisms that adapt to anticipated scenarios without human intervention, further enhancing network service availability.

- **Agility and scalability:** Streamlining network operations enables faster, more reliable delivery of network services than manual processes, increasing agility in network management. This accelerated delivery also allows larger network environments to be managed with equal or even less human oversight. For instance, provisioning a new network service, such as a VPN connection, can be fully automated—from user request to configuration and validation—creating a seamless, self-service experience. Additionally, because automation has to be data-driven, it promotes standardization, as non-standard networks complicate automation. Another example is orchestrating software upgrades across an entire network fleet due to a critical security vulnerability. With a consistent, automated process, these upgrades can be deployed swiftly, requiring limited engineering resources to monitor.
- **Support for ubiquitous networks:** Today's IP networks are more heterogeneous than ever, encompassing various domains such as campus, data centers, wireless, internet, and cloud networks. These environments utilize different control technologies that must be leveraged to ensure end-to-end connectivity. This diversity introduces a range of interfaces and options, often presented in user-friendly formats such as CLIs or **user interfaces (UIs)**. To meet current expectations, it is essential to support all these interfaces from a holistic perspective. A robust network automation solution can integrate with any type of network from any vendor, delivering comprehensive solutions and the ability to interchange vendors' equipment with limited engineering overhead.
- **Risk reduction via policy enforcement:** Nearly every business depends on IT to support its operations, and networks play a critical role in ensuring **business continuity (BC)** by mitigating risks. They enable controlled operations under compliance regulations while also enforcing necessary network security policies whenever and wherever needed.

None of these goals can be achieved without a structured approach. The following section introduces a network automation architecture to help organize the features required.

The building blocks for network automation

So far, we've explored network automation in broad terms without delving into specific concepts. While it may appear to be a single unified approach—an impression some vendors may reinforce—the reality is that network automation encompasses various functionalities. We highly recommend beginning with a structured plan: identify required functions first, then select the tools. This approach clarifies each tool's purpose and facilitates future reassessments as requirements evolve.

To offer a more concrete framework, we'll introduce a network automation architecture from *Network Programmability and Automation, 2nd edition*. The following diagram from the book illustrates the primary building blocks of this architecture.

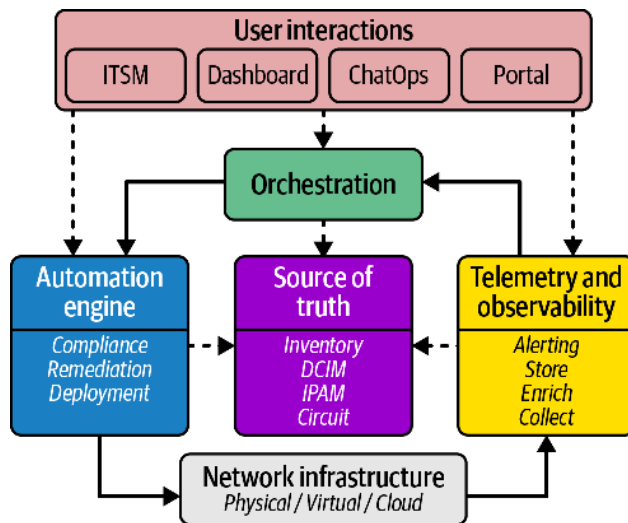


Figure 1.1 – Network automation architecture

This architecture has the following components:

- **Network infrastructure:** This component encompasses traditional physical devices and modern trends, such as cloud and controller-based services. Thus, it involves various management interfaces that need to be integrated properly across the entire network infrastructure.
- **User interactions:** Even in an automated system, human interactions are crucial for its success. Different interfaces are used based on the context, such as Git for GitOps workflows or ticketing systems for self-service options for non-technical users. Properly selecting and designing these interfaces is essential, as a poorly designed interface can hinder adoption. Effective user interactions may take various forms, including service portals, dashboards, and messaging applications.

- **Source of truth (SoT):** An implicit requirement of automation is having consistently structured data. This idea is represented by the SoT concept, as it contains the data that represents the desired state of the network. While simple automation tasks may rely on ad hoc data, more complex solutions require a persistent reference to the intended state of the network. This state includes critical information such as the *network inventory*, *IP addressing*, *VLANs*, *routing protocols*, and **access control lists (ACLs)**. The SoT doesn't need to be a unique system but has to provide a consistent view of the network intent, integrating different data sources, such as YAML and Jinja2 files in Git or a database (of any flavor).
- **Automation engine:** The automation engine is responsible for executing tasks that change the state of the network infrastructure. It takes input data from user interactions and the SoT to interact with the network via the appropriate interfaces. This component encompasses configuration management and operational tasks, such as rebooting devices. The automation engine integrates various tools and scripts to ensure efficient operation.
- **Telemetry and observability:** This component focuses on retrieving and analyzing the network's operational state in a read-only manner. Telemetry and observability involve collecting, storing, and visualizing network data to provide insights into its performance. By monitoring the actual state of the network against the intended state defined in the SoT, automation tasks can be triggered when discrepancies arise.
- **Orchestration:** Orchestration serves as the connective tissue between the other components, defining the workflows that chain tasks together. Most automation solutions are not linear; they have multiple execution paths depending on the outcomes of tasks, such as success, failure, or approval. For instance, orchestration can initiate a task in the automation engine in response to an alert from the observability component, enabling closed-loop automation solutions that enhance operational efficiency.

In reality, one tool may cover different blocks. For example, Ansible is an automation engine that also offers a CLI for users to interact with directly. Understanding the required functionalities and how they are implemented by the tools guides in choosing the right ones.

Other tools offer similar functionality. For example, **Nornir** is a Python framework that enables automation through pure Python code, giving developers greater control over execution. This provides more flexibility for building customized solutions, but it can present a steeper learning curve for traditional network engineers compared to more declarative tools.

From the myriad of tools available for this book, both open source and proprietary ones, we chose Ansible and related solutions to illustrate how to implement a network automation stack.

Discovering Ansible

Ansible is the most popular tool for configuration management and has become a standout solution for network automation due to several unique features, which we'll explore in the following sections. Before diving into these specifics, let's take a quick look at Ansible's journey, from its initial release to today.

Ansible was created by Michael DeHaan in 2012 as an open source, agentless automation tool focused on simplifying configuration management, application deployment, and task automation. DeHaan aimed to provide a tool that was easier to use than other solutions available at the time, such as Puppet and Chef. This vision led to Ansible's simple *YAML syntax*, *agentless architecture*, and *reliance on SSH*, which made secure communication straightforward and accessible.

In 2013, Ansible 1.0 was released, introducing several core features that are still widely used today, including the inventory system, playbooks, and modules. Shortly after, dynamic inventories were added, along with enhanced error-handling and reporting capabilities.

In 2015, Ansible was acquired by Red Hat, and Ansible 2.0 was released in 2016. It brought a significant overhaul of the code base, introducing a modular plugin architecture and new capabilities such as roles, which improved playbook organization and code reusability. This release also saw the launch of **Ansible Galaxy**, a repository for sharing and discovering reusable Ansible content. The 2.x series continued to evolve through 2020, adding support for new modules, Docker orchestration, version control integration, asynchronous task execution, and other improvements. By version 2.9, Ansible introduced **Content Collections**, allowing users to package and distribute modules, plugins, and roles in a structured way, enhancing Ansible's extensibility and organization.

Since Ansible 2.10, an important reorganization of Ansible has introduced two separate tracks:

- **Ansible Core:** It only includes the language, runtime, and built-in plugins; any extension needs to be installed. It continues with the classic Ansible versioning: starting from 2.10, and increasing to 2.11, 2.12, and so on (it doesn't use semantic versioning). Being a lean base to build from, it's used when embedding Ansible into custom tooling or in an environment with tightly controlled dependencies.
- **Ansible community package:** It comes with `ansible-core` and a curated set of community-maintained collections (modules, plugins, roles). It uses a new versioning schema following semantic versioning rules, starting from 2.10, followed by 3.0.0, 4.0.0, and so on. Coming with all batteries included, it's a convenient way to get started on using Ansible for broad automation needs.

To get all the updates on Ansible release and maintenance rules, please refer to https://docs.ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html.



Note

As an informational reference, at the time of this writing, the version for `ansible-core` is 2.17, and the Ansible community package is at 10.x.

At this point, you may still be wondering what makes Ansible special. We believe that, before getting hands-on with Ansible, it's important to understand its different traits.

Ansible's characteristics

To understand why Ansible has become the leading tool for network automation, it's essential to examine its key characteristics:

- **Agentless operation with agent-based option:** Ansible can function without an agent, meaning it doesn't require pre-installed software on remote devices to operate. This is crucial in environments where installing an agent, as on legacy network devices or cloud infrastructure, is impractical or impossible. This flexibility was a major advantage over agent-based tools such as Chef or Puppet.
- **Imperative and declarative approaches:** Ansible allows an imperative approach, where each step is defined sequentially. But, also, some modules offer a declarative style, allowing users to specify the desired end state without detailing each action necessary to achieve it. This flexibility makes Ansible adaptable to various network automation scenarios.
- **Idempotency for consistent, repeatable automation:** Ansible supports idempotent operations, meaning tasks can be run repeatedly without unintended side effects if the system is already in the target state. This ensures predictable automation outcomes. However, idempotency relies on using declarative modules, which may not always be available for certain network automation tasks.
- **Configuration management and infrastructure provisioning:** While Ansible excels in configuration management, it also supports infrastructure provisioning. For provisioning needs, it integrates smoothly with dedicated tools such as Terraform, allowing users to leverage each tool's strengths.

- **Extensive built-in and community functionality:** Ansible's robust set of built-in modules, broad community integrations, and extensibility via custom plugins make it easy to get started quickly. Additionally, it supports diverse vendors and solutions, making it highly adaptable to various environments.
- **YAML syntax for ease of use:** Ansible uses YAML for configuration, offering a user-friendly and accessible interface that reduces the learning curve compared to tools requiring a full programming language, such as Python.

Hopefully, the preceding list will help you understand the reasons behind Ansible's success. However, within a network automation architecture, such as the one presented in the *The building blocks for network automation* section, Ansible may play different roles, which we will explain next.

The role of Ansible in a network automation strategy

We already mentioned that the main role of Ansible is as an automation engine. It executes automated tasks on the network infrastructure. However, Ansible can fulfill other roles:

- **User integrations:** Ansible offers a CLI to run playbooks and an editor to update the YAML files that describe the tasks or the data (we will use them extensively in this book). Both interfaces have been easily adopted by traditional network engineers due to their being more friendly to them than other low-level interfaces. Also, other components of the Ansible ecosystem, such as the AWX platform (which we will cover in *Chapter 13*), provide a **graphical UI (GUI)** to interact with it.
- **Source of Truth:** Even though Ansible offers a dynamic inventory integration (as we explain in *Chapter 12*), it provides a static inventory and variable organization (which you will discover in *Chapter 2*) that can be good enough for some small and static environments.
- **Observability:** Ansible is not a platform for collecting telemetry or logs, but it can support some observability tasks, such as getting data to validate network changes or supporting troubleshooting operations via the execution of tasks in read-only mode. In all the chapters in *Part 2* and *Part 3*, there are examples of collecting data with Ansible.
- **Orchestration:** In *Chapter 13*, we will explain the AWX platform as an orchestrator (the open source version of **Ansible Automation Platform**, or **AAP** for short) to concatenate multiple Ansible playbooks together.

Hopefully, this first chapter has motivated you to get deep into Ansible understanding for the rest of this book.

Summary

This chapter introduced the foundational context of network automation, detailing its evolution and the key challenges it addresses in modern network environments. It also explored Ansible's journey to becoming a central tool in network automation, emphasizing its unique characteristics, such as agentless operation, YAML-based simplicity, and extensive community support, which set it apart in the landscape of configuration management tools. Additionally, it underscored the importance of having a structured network automation plan, choosing tools based on specific functions, and understanding architectural components. This overview highlighted why Ansible gained such a prominent role in automating diverse network environments efficiently and reliably.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



2

Building Blocks of Ansible

Ansible is an enormously popular automation framework that has been used to automate IT operations for a long time. It simplifies the management of IT infrastructure and translates business logic into well-defined implementation procedures.

Ansible is written in **Python** and communicates with infrastructure with **APIs**, including **SSH**. It started support for networking devices beginning with Ansible 1.9, and since Ansible 2.9, the support for network devices has grown extensively.

In this chapter, we will introduce the main components of Ansible and outline the different features and options that Ansible supports. The following are the recipes that will be covered:

- Installing Ansible
- Building the Ansible inventory
- Controlling Ansible's options
- Defining Ansible variables
- Building a playbook
- Using Ansible conditionals
- Using Ansible loops
- Securing secrets with Ansible Vault
- Using Jinja2 templates with Ansible
- Using Ansible network-related filters
- Using Ansible tags

This chapter aims to provide a basic understanding of the Ansible components you will utilize throughout this book to interact with the IT network infrastructure. Consequently, not all the examples in this chapter focus on managing networking devices. Instead, we will focus on understanding the components of Ansible so that we can use them effectively in the next chapters.

Technical requirements

All the examples in the book can be reproduced by you. For this chapter, you only need a Linux system with internet connectivity.

If you'd like to **do it yourself (DIY)**, the requirements for installing Ansible include the following:

- A **Linux virtual machine (VM)** with either of the following distributions:
 - **Ubuntu 22.04** or higher
- Internet connectivity for the VM

Alternatively, *Chapter 16, The Lab*, serves as a controlled hands-on environment to facilitate a consistent reproduction of all the steps and recipes included in the book. The packaged VM the lab spins up has the following installed and ready to go:

- Ansible (Ansible Core and Ansible community)
- Access to network lab equipment is installed, and a topology is provided.
 - The topology includes Arista, Nokia, and Juniper containerized network operating systems that we will use throughout the book. Ansible AWX is shown here and will be fully utilized later in the book, but to whet the appetite, AWX provides a web-based **user interface (UI)**, REST API, and task engine built on top of Ansible.

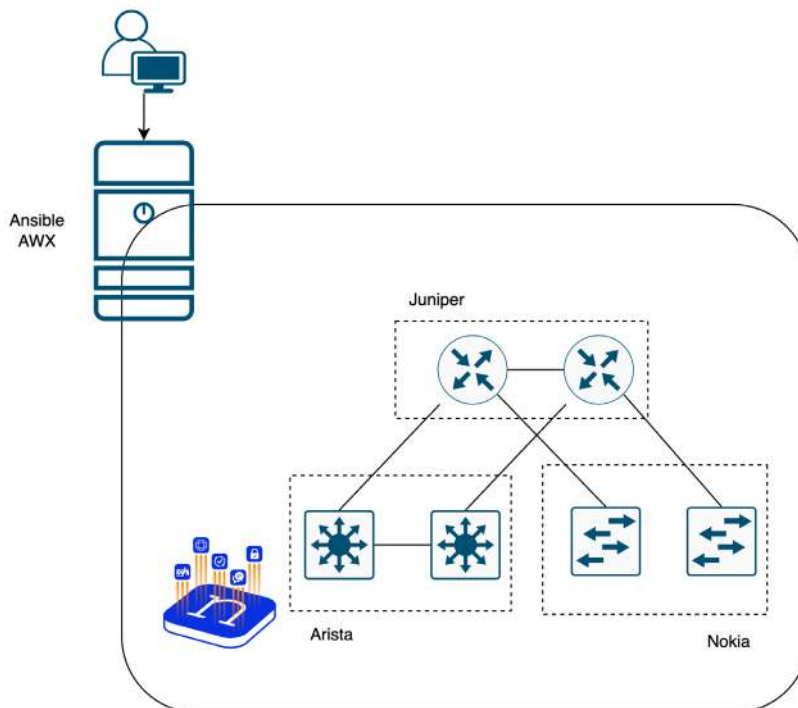


Figure 2.1 – Lab environment setup

 Quick tip: Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.





Note

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch02.

The chapter builds upon itself, so cloning the repository is not recommended.

First things first, we have to get Ansible installed.

Installing Ansible

As mentioned already, if you are using the lab environment provided in the book, you can skip this step. But if you want to install Ansible on your own, these are the steps to get it done.

We should install Ansible on a Unix-like machine (this is known as the Ansible control machine). In this recipe, we will outline how to install Ansible on an Ubuntu Linux machine (specifically, with **version 22.04**).

Ansible is written in Python, so you must have Python installed on your system. Most OSs nowadays come with Python pre-installed, but follow the next steps to validate and finalize the Python install. Lastly, you will install `python3-venv`, which allows you to create multiple Python environments and keep different dependencies installed (a best practice):

1. Install Python 3 and Python 3 virtualenv:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# sudo apt install  
python3.11 python3.11-venv
```

2. Validate the Python 3 installation:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# python3.11 --version  
Python 3.11.10
```

3. Create a Python virtual environment to install Ansible into:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# python3.11 -m venv  
~/virtualenvs/ansible-cookbook-venv
```

4. Activate the virtual environment you just created. All the installed packages from now on will be restricted to this environment and will not mess with the default Python environment in the system:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# source
~/virtualenvs/ansible-cookbook-venv/bin/activate
```

5. Install Ansible into the virtual environment. As mentioned in *Chapter 1*, you could choose to install only the core functionalities or include the community collections. For this book, we recommend the latter to get a more complete environment that doesn't require you to install the collections needed in the next chapters:

1. Install Ansible Core only (does not come with community collections installed):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# pip install
ansible-core
```

2. Install Ansible community version (comes with collections installed):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# pip install
ansible
```

6. Validate the Ansible installation:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible --version
ansible [core 2.17.6]
... output omitted ...
```

7. Quickly validate that the collections are installed (when done via the community install):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-galaxy
collection list
/root/miniconda/lib/python3.12/site-packages/ansible_collections
Collection                                Version
-----
amazon.aws                               8.2.1
ansible.netcommon                        6.1.3
ansible.posix                            1.6.2
... output omitted ...
```

Note



You can execute the following command to get a full list of the collections installed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-
galaxy collection list
```


Once Ansible is installed, the next, and important, step is to create your Ansible inventory. In the next section, we will analyze the steps to create an inventory.

Building the Ansible inventory

One of Ansible's two fundamental building blocks is the **inventory**. It defines the nodes in scope, their attributes, connection types, variables, and more. Ansible also supports both static and dynamic inventories. In this book, we use the static inventory in all but *Chapter 12*, where a dynamic inventory integration is explained.

You will learn how to create a static inventory in two formats: the INI format and the YAML format. Let's take a look at how to build an INI-style inventory file specifically for our lab topology.

The inventory file supports organization into groups to simplify how to target different components of the infrastructure. In our lab, we have four types of devices: **Linux servers**, **Arista cEOS**, **SR Linux**, and **Juniper cRPD** network devices. Thus, we group them accordingly. In the following inventory file, you will create node definitions and group them based on the types of devices:

```
root@cookbook-lab:~/cookbook-lab# vi chapters/ch02/hosts
[linux]
client1 ansible_host=198.51.100.2
client2 ansible_host=198.51.100.3
[arista_ceos]
ceos-01 ansible_host=198.51.100.11
ceos-02 ansible_host=198.51.100.12
[juniper_crpd]
crpd-01 ansible_host=198.51.100.21
crpd-02 ansible_host=198.51.100.22
[nokia_srlinux]
srl-01 ansible_host=198.51.100.31
srl-02 ansible_host=198.51.100.32
```

Now, let's look at what this identical static inventory file looks like in the *Ansible YAML* inventory format. This code block shows an identical inventory that is formatted as YAML:

```
root@cookbook-lab:~/cookbook-lab# vi chapters/ch02/hosts.yml
---
linux:
  hosts:
    client1:
```

```
    ansible_host: 198.51.100.2
  client2:
    ansible_host: 198.51.100.3
arista_ceos:
  hosts:
    ceos-01:
      ansible_host: 198.51.100.11
    ceos-02:
      ansible_host: 198.51.100.12
juniper_crpd:
  hosts:
    crpd-01:
      ansible_host: 198.51.100.21
    crpd-02:
      ansible_host: 198.51.100.22
nokia_srlinux: (1)
  hosts:
    srl-01: (2)
      ansible_host: 198.51.100.31 (3)
    srl-02:
      ansible_host: 198.51.100.32
```

Notice that in both formats, there are three main parts (in bold in the preceding code snippet). This code snippet reference style is going to be used throughout the book. Code snippets within this code block will contain numbered references, denoted as (#). You should consult the numbered list to obtain a comprehensive explanation of each referenced element:

- (1) **Group name:** `nokia_srlinux`
- (2) **Component/hostname:** `srl-01`
- (3) **Variables (name and value):** `ansible_host: 198.51.100.31`

The Ansible engine will attempt to resolve hostnames into an IP address (using hostname resolution, such as DNS). If your Ansible control host cannot resolve the hostname into an IP address, then set the `ansible_host` variable to explicitly specify it.

Ansible inventories are flexible; they support grouping hosts into “groups,” setting parent/child relationships, and even support the ability to use regexes to define many hosts that follow a similar hostname naming convention easily.

Suppose our inventory had 10 Nokia SR Linux devices instead of 2. We could define the hosts with a regex pattern, as seen next. All the variables under these hosts will have the same variables. Let's examine an inventory that utilizes a regex pattern to define multiple hosts in a single line:

```
nokia_srlinux:
  hosts:
    srl-[01:10]:
```

Finally, there may be a reason to further group the inventory by using parent-child groups. Perhaps you want a group that encompasses all network equipment. To achieve that scenario, you will group three device type groups under a parent group, as seen next:

```
root@cookbook-lab:~/cookbook-lab# vi chapters/ch02/hosts.yaml
... omitted output ...
network_equipment:
  children:
    arista_ceos:
    juniper_crpd:
    nokia_srlinux:
```



Note

To learn more about the flexibility of the Ansible inventory, check out the Ansible Core documentation: https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html#how-to-build-your-inventory.

Throughout this chapter, we focus on the YAML inventory format, but in the book, we will use it interchangeably with the INI format to allow you to familiarize yourself with both.

To validate that your static inventory files are being parsed and generated appropriately, use the following code to effectively visualize your inventory content. It supports both graph and list views. The graph format is shown next:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-inventory -i
hosts.yaml --graph
@all:
|--@ungrouped:
|--@network_equipment:
| |--@arista_ceos:
```

```
| | |--ceos-01
| | |--ceos-02
| |--@juniper_crpd:
| | |--crpd-01
| | |--crpd-02
| |--@nokia_srlinux:
| | |--sr1-01
| | |--sr1-02
|--@linux:
| |--client1
| |--client2
```

In the previous output, you can see how we have nested groups: all -> network_equipment -> arista_ceos. The all group is created by default, and it includes all the hosts.

Controlling Ansible options

Ansible is a customizable framework. It can be controlled by tuning its configuration via a **configuration file**. Options such as how Ansible looks during playbook execution, where Ansible can find extra details (e.g., local collections or roles), or even how Ansible connects to devices.

There is a precedence order in how Ansible configuration options are evaluated. These are fully documented in the Ansible docs (https://docs.ansible.com/ansible/latest/reference_appendices/config.html#ansible-configuration-settings); however, a quick ordered list is provided next:

1. The `ANSIBLE_CONFIG` environment variable
2. An `ansible.cfg` file in the local project directory
3. A `.ansible.cfg` file in the executing user's home directory (for example, `~/.ansible.cfg`)
4. The system-wide configuration file: `/etc/ansible/ansible.cfg`



Note

This is a first-match rule; as soon as it finds a config, it uses that.

With the configuration file being a first-match rule, to quickly confirm the configuration file that will be used, run the `ansible --version` command:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible --version
ansible [core 2.17.6]
  config file = None
```

As you can see, the config file is None since we have not created it yet.

Ansible has many configuration options, as you can check in the previous documentation link; this recipe will focus on a few of the most common options you may need, specific to network automation, using a file in the chapter folder. The next code block shows how to define these options and create a new Ansible configuration file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi ansible.cfg
[defaults] (1)
inventory=hosts.yml (2)
gathering=explicit (3)
host_key_checking=False (4)
```

The preceding Ansible configuration file configures a few common Ansible options that are explained further next:

- (1) `[defaults]`: This is the default grouping that holds the default settings for Ansible as a whole.
- (2) `inventory`: This option modifies the default inventory file that Ansible searches to find its inventory (by default, this is `/etc/ansible/hosts`). We adjust this option to let Ansible use our inventory file and stop using the `-i` operator to specify our inventory during each playbook run.
- (3) `gathering`: By default, Ansible runs a setup module to gather facts regarding managed nodes while the playbook is running. This setup module is not compatible with all network nodes since this module requires a Python interpreter on the managed nodes. By setting fact gathering to `explicit`, we disable this default behavior.
- (4) `host_key_checking`: By default, Ansible has host key checking on by default; this is a security measure and helps prevent some common concerns. It's common for SSH keys to change on network devices, and because of that, the `known_hosts` file on the control node can be incorrect. To avoid constantly cleaning up the `known_hosts` file, you can set this to `False`.

**Note**

Now that you've added your default configuration and pointed it to the `hosts.yml` file, all future examples will not pass the `-i` trigger into the playbook executions.

Once the configuration, inventory hosts, and groups are in place, the next natural step is to enrich the hosts with related variables.

Defining Ansible variables

Ansible uses **variables** to store data attributes; they can be at the *node*, *group*, or *default* level. You will define the two most common variable types in this section: host and group variables. It's important to know that, by default, Ansible expects files in specific directories. In the root of the project, these folders are called `host_vars` and `group_vars`, respectively.

Let's demonstrate how to use `host_vars` and `group_vars` to show how you might flexibly store NTP configuration data.

In this recipe, you will enrich the Ansible inventory with the following NTP variable data:

1. Set two NTP servers for all hosts to `192.168.1.10` and `192.168.2.10`.
2. Set just the `client2` Linux host to use `192.168.2.10` and `192.168.3.10`.
3. Set the `nokia_srlinux` group to use `192.168.1.10` and `192.168.3.10`.

To properly set up the variables, you must first create files and a directory structure. The following tree output shows the `host_vars` and `group_vars` directories and the files that need to be created. Ansible expects the host and group YAML files to be named the same as the group and hostnames in the inventory file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# tree .
.
├── ansible.cfg
├── group_vars
│   ├── all.yml
│   └── nokia_srlinux.yml
├── host_vars
│   └── client2.yml
├── hosts
└── hosts.yml
```

The output demonstrates what the Ansible best practices look like. `group_vars` and `host_vars` are folders that hold specific YAML files. Next, you will explore each of these further vars files in more detail:

1. Let's first analyze a special group variable file called `all.yml`. This is a group file that gets applied to all groups. Notice we're setting our NTP servers and order for all groups:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi group_vars/all.
yml
---
ntp:
  - 192.168.1.10
  - 192.168.2.10
```

2. Our requirements dictate that `nokia_srlinux` devices should have a different NTP server, so we have a group variable file that will override the `all.yml` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi group_vars/nokia_
srlinux.yml
---
ntp:
  - 192.168.1.10
  - 192.168.3.10
```

3. Finally, you do the most specific variable definition using the host variable file for `client2`. In this file, you specify the NTP servers to use for this *single host*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi host_vars/
client2.yml
---
ntp:
  - 192.168.2.10
  - 192.168.3.10
```

4. Once the data is populated in Ansible, you may want to see the rendered/merged variable data. Simply add the `--vars` command-line switch to the `ansible-inventory` command. This is helpful to visualize which data is going to be used when the playbook targets a specific host:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-inventory -i
hosts.yml --graph --vars
```

```
@all:
  |--@ungrouped:
  |--@network_equipment:
  | |--@arista_ceos:
  | | |--ceos-01
  | | | |--{ansible_host = 198.51.100.11}
  | | | |--{ntp = ['192.168.1.10', '192.168.2.10']}
  ... omitted output ...
  |--@nokia_srlinux:
  | |--sr1-01
  | | |--{ansible_host = 198.51.100.31}
  | | | |--{ntp = ['192.168.1.10', '192.168.3.10']}
  ... omitted output ...
  |--@linux:
  | |--client1
  | | |--{ansible_host = 198.51.100.2}
  | | |--{ntp = ['192.168.1.10', '192.168.2.10']}
  | |--client2
  | | |--{ansible_host = 198.51.100.3}
  | | |--{ntp = ['192.168.2.10', '192.168.3.10']}
  |--{ntp = ['192.168.1.10', '192.168.2.10']}
```

There are other techniques to define variables, including the following:

- Using the `vars` keyword within the play. (A play in Ansible groups multiple tasks together and is the unit of execution that defines the nodes to execute against.) This is just a snippet of a playbook we will build in the coming recipes:

```
---
- name: Initial Playbook
  hosts: all
  gather_facts: no
  vars:
    ntp_source_interface: "loopback0"
  ... output omitted ...
```


- Using `vars_files` to define variables in a file and having Ansible read them:

- Create a file named `network_config_vars`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi network_
config_vars
ntp_source_interface="loopback0"
```

- Point to `vars_files` in the play:

```
---
- name: Initial Playbook
  hosts: all
  gather_facts: no
  vars_files:
  - network_config_vars
  ... output omitted ...
```

- Specifying variables at the command line using the `--e` option. An example command is shown next:

```
ansible-playbook example_playbook.yml --e ntp_source_
interface=loopback0
```

In addition to user-defined variables, Ansible has some default variables that it builds dynamically for its inventory. Frequently used variables include the following:

- `inventory_hostname`: The name of the hosts as defined in the inventory (for example, `sr1-01` and `client1`)
- `play_hosts`: A list of all hosts included in the play
- `group_names`: A list of all groups that a specific host is a part of (for example, for `sr1-01`, this will be `[network_equipment and nokia_srlinux]`)



Note

Ansible merges variables based on precedence orders. A good rule of thumb is to use `group_vars` and `host_vars`, and use command-line arguments as the most preferred overrides. For the full precedence ordering, see https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html#ansible-variable-precedence.

Now that you have a grasp on setting up an Ansible inventory, configuring Ansible, and defining variable data in an **infrastructure as code (IaC)** format, next, you can create and execute your first playbook!

Building a playbook

An Ansible playbook is the other fundamental element in Ansible that declares what actions you would like to perform on the target hosts (taken from the inventory). The format of the Ansible playbook is YAML, and it describes *what to do* in an ordered and repeatable manner, becoming the central tool for managing the infrastructure as code.

In this recipe, you will learn how to write an Ansible playbook and define the hosts that will be targeted. Thus, you will create a YAML file, determine the play's general details (for example, the hosts to execute toward), and define a list of tasks.

The command to execute the playbook is `ansible-playbook`. It can be explicit and point to a specific inventory, via `-i hosts.yml`, or take the default inventory from the Ansible configuration. Finally, it takes the playbook name to execute (`pb_debug_vars.yml`):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi pb_debug_vars.yml
---
- name: Initial Playbook (1)
  hosts: all (2)
  gather_facts: no (3)
  Tasks: (4)
    - name: Display Hostname (5)
      ansible.builtin.debug: (6)
        msg: "Hostname is {{ inventory_hostname }}" (7)
    - name: Display the NTP Variables
      ansible.builtin.debug:
        msg: "{{ inventory_hostname }} has NTP servers {{ ntp }}"
```

Analyzing the main concepts of the previous playbook, you should notice the following:

- (1) The name of the playbook.
- (2) Target hosts (or group). In this case, `all` includes all the hosts from the inventory.
- (3) **Do not gather** device-specific facts from the device. Gathering facts runs multiple commands on the node in order to provide metadata for the node.
- (4) A list of inner tasks that the playbook is composed of.

- (5) The name of the inner task.
- (6) The Ansible module to execute. In this case, it is a built-in one.
- (7) Attribute taken by the module that renders a text message with Jinja2 (notice the variable syntax: `{{ inventory_hostname }}`). These variables are the ones rendered for each target host, either from the host or group variables.

When the playbook is run, Ansible starts from the top of the file and executes it in order. The first section of the playbook is the *play* details, and then, there is a list of tasks with the modules we want to execute; in this case, `ansible.builtin.debug`.



Note

`ansible.builtin.debug` is a Fully Qualified Collection Name (FQCN), where the format is `namespace.collection.module` (explained in *Chapter 3*), and finally, any arguments to pass into the module (e.g., `msg` in this example).

An Ansible playbook is structured as a list of plays, and each play targets a specific group of hosts (from the inventory). Each play defines one or more tasks to execute against the hosts in this play. Each task runs a specific Ansible module that has a number of parameters. The general structure of the playbook is outlined next:

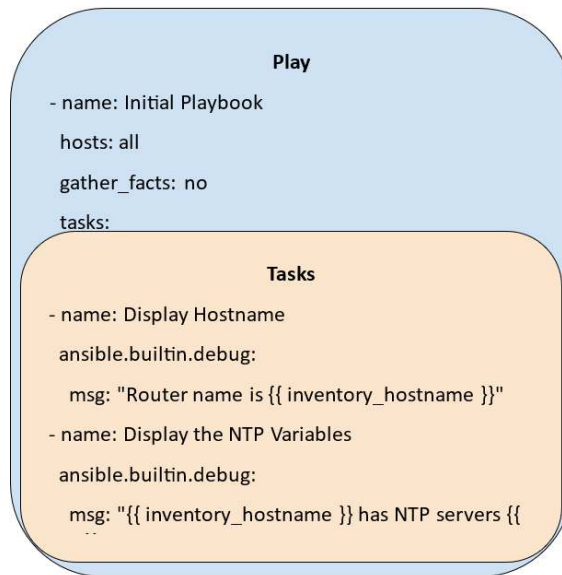


Figure 2.2 – Playbook structure

The playbook run is shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook pb_debug_
vars.yml
PLAY [Initial Playbook] *****
TASK [Display Hostname] *****
ok: [client1] => {
    "msg": "Router name is client1"
}
... omitted output ...

TASK [Display the NTP Variables] *****
ok: [client1] => {
    "msg": "client1 has NTP servers ['192.168.1.10', '192.168.2.10']"
}
ok: [client2] => {
    "msg": "client2 has NTP servers ['192.168.2.10', '192.168.3.10']"
}
... omitted output ...
```

The debug module properly debugged the output per host and crafted the message as expected, with the corresponding variables in each case.



Tip

This playbook uses the `all` keyword to specify all hosts within the inventory. This is a default, well-known group name that Ansible dynamically constructs for all hosts within the inventory.

What happens, though, if you only want to run tasks when certain criteria are met? We will discover Ansible conditionals next.

Using Ansible conditionals

Conditionals are a powerful functionality of all programming languages. Ansible has the ability to use conditional logic to run tasks/plays on a given host based on a conditional test. In this recipe, we will demonstrate how to use conditional task execution.

Here, we will adapt the previous playbook and add two conditionals:

- The first task runs only when the host is in the `linux` Ansible group
- The second task runs when the host is in the `network_equipment` parent group

Very intuitively, the fundamental conditional statement in Ansible is the `when` condition: a task will only execute *when* the conditional is true. To meet the new requirements, the original playbook has been reused and extended:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cp pb_debug_vars.yml pb_
debug_conditionals.yml
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi pb_debug_conditionals.
yml
---
- name: Debug with Conditionals
... omitted output ...
  - name: Display Hostname
    ansible.builtin.debug:
      msg: "Router name is {{ inventory_hostname }}"
    when: inventory_hostname in groups["linux"]
  - name: Display the NTP Variables
    ansible.builtin.debug:
      msg: "{{ inventory_hostname }} has NTP servers {{ ntp }}"
    when: inventory_hostname in groups["network_equipment"]
```

In this updated playbook, you're using a native Ansible variable called `groups` with the `in` operator to describe that the `inventory_hostname` variable is in the group specified (e.g., `network_equipment`).

In the playbook output, Ansible marks the specific task as *skipped* when the conditional check results in a false output (in the first run, only the Linux hosts are included and the rest skipped, and in the second, the opposite). An example output of the playbook is shown next:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook pb_debug_
conditionals.yml
PLAY [Debug with Conditionals] *****
TASK [Display Hostname] *****
ok: [client1] => {
  "msg": "Router name is client1"
}
```

```

ok: [client2] => {
  "msg": "Router name is client2"
}
skipping: [ceos-01]
... omitted output ...
TASK [Display the NTP Variables] *****
skipping: [client1]
skipping: [client2]
ok: [ceos-01] => {
  "msg": "ceos-01 has NTP servers ['192.168.1.10', '192.168.2.10']"
}
... omitted output ...

```

Now that you've learned how to use one programming concept (conditionals), another important programming concept will be covered next: how to loop over objects.

Using Ansible loops

Often, you may need to run a task inside an Ansible playbook to *loop over* some data. Ansible **loops** allow you to iterate over a variable (a dictionary or a list) multiple times to achieve this behavior. In this recipe, we will outline how to use Ansible loops and how to customize how they work.

The first step will be to create a `group_vars` file called `arista_ceos.yml` (matching the group name used in the inventory). This is where we will define the specific NTP server variable data for this group:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi group_vars/arista_ceos.
yml
---
ntp:
  - 192.168.3.10
  - 192.168.5.10

```

Next, we construct an Ansible playbook (limited to Arista devices via the `hosts` configuration) that will execute the only task as many times as items are contained in the `ntp` variable:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi pb_loops.yml
---
- name: LOOP THROUGH A LIST AND PRINT
  hosts: arista_ceos

```

```
gather_facts: no
tasks:
  - name: DISPLAY ALL NTP SERVERS
ansible.builtin.debug:
  msg: "Router {{ inventory_hostname }} with ntp server {{ item }}"
loop: "{{ ntp }}"
```

Ansible is going to execute the playbook only on hosts that are in the `arista_ceos` inventory group. The variables are taken from the `group_vars/arista_ceos.yml` file (more specific than the `all.yml` file), and each host will have the `ntp` variable, which is a list of IP addresses.

The playbook task steps through the `ntp` server list and prints a message for each:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook pb_loops.yml
PLAY [LOOP THROUGH A LIST AND PRINT]*****
TASK [DISPLAY ALL NTP SERVERS] *****
ok: [ceos-01] => (item=192.168.3.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.3.10"
}
ok: [ceos-02] => (item=192.168.3.10) => {
  "msg": "Router ceos-02 with ntp server 192.168.3.10"
}
ok: [ceos-01] => (item=192.168.5.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.5.10"
}
ok: [ceos-02] => (item=192.168.5.10) => {
  "msg": "Router ceos-02 with ntp server 192.168.5.10"
}
... omitted output ...
```

Notice that the task order is not per host but per item. So, the task took the first item in the list and rendered for `ceos-01` and `ceos-02` first, and in the second iteration, took the second item and did the same for both devices.

By default, Ansible's loop variable is named `item` (as you can see in the previous output), but it can be customized by `loop_control` with the `loop_var` key, in the task settings:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cp pb_loops.yml pb_loop_control.yml
```

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi pb_loop_control.yml
... omitted output ...
- name: DISPLAY ALL NTP SERVERS LOOP_VAR
  ansible.builtin.debug:
    msg: "Router {{ inventory_hostname }} with ntp server {{
      ntp_server }}"
loop: "{{ ntp }}"
loop_control:
  loop_var: "ntp_server"

```

Notice that instead of using `item`, we're now calling the `ntp_server` loop variable. As seen next, the results are the same:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook pb_loop_
control.yml
TASK [DISPLAY ALL NTP SERVERS] *****
ok: [ceos-01] => (item=192.168.3.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.3.10"
}
ok: [ceos-02] => (item=192.168.3.10) => {
  "msg": "Router ceos-02 with ntp server 192.168.3.10"
}
ok: [ceos-01] => (item=192.168.5.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.5.10"
}
ok: [ceos-02] => (item=192.168.5.10) => {
  "msg": "Router ceos-02 with ntp server 192.168.5.10"
}

```



Note

Older versions of Ansible supported multiple looping mechanisms via the `with_*` formatting; these have been replaced (deprecated but not removed yet) with the `loop` keyword.

Lastly, it's common to need the iterable index, which is available via the `index_var` option in `loop_control`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cp pb_loop_control.yml pb_
loop_index.yml

root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi pb_loop_index.yml
... omitted output ...
- name: DISPLAY ALL NTP SERVERS LOOP_VAR and LOOP_INDEX
  ansible.builtin.debug:
    msg: "Router {{ inventory_
hostname }} with ntp server {{ item }} at index {{ loop_index }}"
    loop: "{{ ntp }}"
    loop_control:
      index_var: "loop_index"
```

Notice that instead of using `loop.index`, we're now calling the `loop_index` loop index variable. As seen next, the results display how that is used by referencing the new variable name:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook pb_loop_
index.yml
TASK [DISPLAY ALL NTP SERVERS LOOP_VAR and LOOP_INDEX] *****
ok: [ceos-01] => (item=192.168.3.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.3.10 at index 0"
}
ok: [ceos-01] => (item=192.168.5.10) => {
  "msg": "Router ceos-01 with ntp server 192.168.5.10 at index 1"
}
... omitted output ...
```



Note

For more customizable options for Ansible loops, visit the Ansible docs at https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html#loops.

As your playbook variable data gets more sophisticated, secret variable data is a natural next step. In the next section, you will learn how to secure secrets with Ansible Vault.

Securing secrets with Ansible Vault

Some of the data that you need to use in an Ansible playbook can be sensitive, such as credentials to access the target hosts. Obviously, this information can't be stored as data variables in plain text. There is a functionality, **Ansible Vault**, that provides a method to encrypt data and, therefore, can be safely stored and decrypted when needed while the playbook is running. In this recipe, you will outline how to use Ansible Vault in order to secure sensitive information in Ansible. Let's get started!

1. Create a new file called `.ansible_vault_passwd` that contains a password to encrypt and decrypt the data. This password should never be shared in plain text or pushed to a shared repository:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# echo 'strong_
password' > ~/.ansible_vault_passwd
```

2. The `ansible-vault` command creates a new file that is encrypted using a key/password from the file (which is called `.ansible_vault_passwd` in our example), specified by `--vault-id`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-vault create
--vault-id=~/.ansible_vault_passwd secrets
```



Note

The `ansible-vault create` command will create the file but also open that file in a text editor to provide the data before it is encrypted.

3. Add the following variables to this new secrets file (the previous command will open a `vi` editor). You can add as many secret variables as you need:

```
ospf_password: ospf_P@ssw0rD
```

4. Once you exit the text editor and save the file, the contents of the secrets file are encrypted. If you try to read it without decryption, it is hidden:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cat secrets
$ANSIBLE_VAULT;1.1;AES256
65383466336265383330393631663566653933656138313937333836646435383662
346431616436
... omitted output ...
```

5. Create a new playbook called `ch2_pb_vault.yml`, referencing the variable file in the playbook using `vars_files`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi ch2_pb_vault.yml
---
- name: USING ANSIBLE VAULT
  hosts: all
  gather_facts: no
  vars_files:
    - secrets
  tasks:
    - name: DISPLAY SECURE DATA
      ansible.builtin.debug:
        msg: "OSPF Password for Router {{ inventory_
hostname }} is {{          ospf_password }}"
        when: inventory_hostname == 'ceos-01'
```

6. Run the playbook as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
--vault-id=~/.ansible_vault_passwd ch2_pb_vault.yml
```

For Ansible to decrypt this file, you must supply the decryption password using the `--vault-id` option. When you run `ansible-playbook`, you must supply this decryption password; otherwise, `ansible-playbook` fails, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
ch2_pb_vault.yml
ERROR! Attempting to decrypt but no vault secrets found
```

Alternatively, if the file doesn't exist or security requirements disallow a vault file, the `--ask-vault-pass` option can be used, which will prompt for the password. Next, you can use this command-line option to prompt for the decryption password:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
--ask-vault-pass ch2_pb_vault.yml
Vault password:****
PLAY [USING ANSIBLE VAULT] *****
***
TASK [DISPLAY SECURE DATA] *****
***
```

```
skipping: [client1]
skipping: [client2]
ok: [ceos-01] => {
    "msg": "OSPF Password for Router ceos-01 is ospf_P@ssw0rd"
}
... omitted output ...
```



Note

Ansible Vault can be used to encrypt full playbooks, variable files, and much more. As a bonus tip, you can even use `ansible-vault` in conjunction with other files, such as a Python script.

The next section is going to build upon what you’ve done with Ansible variables and teach you how to use those variables to generate configuration snippets.

Using Jinja2 templates with Ansible

Jinja2 is a powerful templating engine for Python and is supported natively by Ansible. It can also be used to generate any text-based files, such as HTML, CSV, or YAML. In this recipe, you will utilize Jinja2 with Ansible variables in order to generate custom CLI configuration files for network devices. It is important to become familiar with Jinja2 as it has become the de facto templating language for network automation.



Note

Jinja2 is a robust templating language that is heavily used in Ansible; however, this book doesn’t intend to teach Jinja2 in deep detail. For more information on Jinja2, visit the documentation: <https://jinja.palletsprojects.com/en/stable/>.

In the *Using Ansible loops* section, we defined the `ntp` variable. Now, let’s use this data and create a configuration snippet for NTP configuration for two **Arista cEOS** devices.

1. Create a `templates/` directory that will contain a template file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# mkdir templates
```

2. Create an `arista_ceos.j2` file that contains Jinja2 templating content:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi templates/arista_
ceos.j2
hostname {{ inventory_hostname }}
!
{% for server in ntp %}
ntp {{ server }}
{% endfor %}
```

3. Create a new playbook called `ch2_pb_jinja2.yml` that creates a new directory called `configs` (which groups all the generated configurations), and then a task to generate one file per network device:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi ch2_pb_jinja2.yml
---
- name: GENERATE NTP CONFIGS
  hosts: arista_ceos
  gather_facts: no
  tasks:
    - name: Create Configs Directory
      ansible.builtin.file:
        path: configs
        state: directory
      delegate_to: localhost (1)
      run_once: true (2)
    - name: Generate Arista Config
      ansible.builtin.template:
        src: templates/arista_ceos.j2 (3)
        dest: configs/{{ inventory_hostname }}.cfg (4)
      delegate_to: localhost
```

Let's deconstruct what's going on in this recipe. The first task is used to ensure idempotently that an output directory (called `configs`) exists, using the `ansible.builtin.file` Ansible built-in module. Let's explore how to use `delegate_to` and `run_once` to achieve this:

(1) `delegate_to: localhost`: By default, the template modules create an output file on remotely managed nodes. However, this is not possible in our case since the managed devices are network nodes. We use this option in order to run this task locally on the Ansible control machine.

(2) `run_once`: Runs the tasks only once. This works well in this case because we're using it in conjunction with `delegate_to`.

To clarify further, `delegate_to` makes the task run on the control node, but without `run_once`, it would still run for each host in the play (e.g., it would try to create a `configs` directory twice for the two Arista devices). Adding `run_once` ensures it truly runs only a single time.

In the second task, we use an Ansible built-in module called `template` that takes two main parameters:

(3) `src`: This references the Jinja2 template file.

(4) `dest`: This specifies the output file that will be generated, using the Jinja2 rendering functionality to personalize the filename using the `inventory_hostname` host.

The final element that isn't directly provided via a parameter is *context* (variable) data, which comes natively from the Ansible framework via `vars`. That is, all the related host variables will be available within the Jinja2 template rendering; for example, `ntp`.

1. Run the playbook. Ansible will create a `configs` directory and create files per host:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
ch2_pb_jinja2.yml
PLAY [GENERATE NTP CONFIGS] *****
***

TASK [Create Configs Directory] *****
***
changed: [ceos-01 -> localhost]
ok: [ceos-02 -> localhost]
TASK [Generate Arista Config] *****
***
changed: [ceos-01 -> localhost]
changed: [ceos-02 -> localhost]
... omitted output ...
```

2. Lastly, once the template is rendered in this recipe, the resulting configuration file looks like the following:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cat configs/ceos-01.cfg
hostname ceos-01
!
ntp 192.168.3.10
ntp 192.168.5.10
```

Note



The Ansible built-in template module is an important module to learn; it allows Ansible to be a powerful configuration management solution for any size environment. See the module documentation for full details:

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html.

Now that Jinja2 has been introduced and you've generated a configuration snippet from a template, next, you will learn how you can use Ansible filters to transform data into different formats.

Using Ansible network-related filters

Ansible's **filters** are mainly derived from **Jinja2 filters**, and all Ansible filters are used to transform and manipulate data (Ansible's variables). In addition to Jinja2 filters, Ansible implements its own filters to augment Jinja2 filters, while also allowing users to define their own custom filters. In this recipe, we will outline how to configure and use Ansible filters to manipulate our input data using network-related utilities.

We're going to build on top of the last recipe by adding our management interface configuration to our Arista template. The first thing that needs to be done is to add some more data to our `host_vars` directory. Since we don't have `ceos-01.yml` or `ceos-02.yml` files in our `host_vars` directory, let's create them and populate them with the data needed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi host_vars/ceos-01.yml
---
interfaces:
  mgmt:
    Management0:
```

```

    ip_addr: "198.51.100.11/24"
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi host_vars/ceos-02.yml
---
interfaces:
  mgmt:
    Management0:
      ip_addr: "198.51.100.12/24"

```

We're going to reuse the playbook from the *Using Jinja2 templates with Ansible* section. Just update the template to use the new variable data (interfaces) and pass it through an Ansible filter to get the expected interface configuration:

1. Update the Arista template file with the new code:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi templates/arista_
ceos.j2
hostname {{ inventory_hostname }}
!
{% for server in ntp %}
ntp {{ server }}
{% endfor %}
!
{% for mgmt_int, int_details in interfaces['mgmt'].items() %} (1)
interface {{ mgmt_int }}
    ip address {{ int_details['ip_addr'] }} (2)
{% endfor %}

```

(1) In this line, you will use the Python dictionary `.items()` method call to loop through any interfaces nested under the `mgmt` key dictionary.

(2) Now that you're in the loop, you can access `mgmt_int`, which is the key of the dictionary, and you can capture `ip_addr` directly from `int_details`, which is the value of the dictionary key lookup.

2. When the playbook is run again, the resulting configurations include the new configuration:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
ch2_pb_jinja2.yml
... omitted output ...
root@cookbook-lab:~/cookbook-lab/chapters/ch02# cat configs/ceos-01.

```



```
cfg
hostname ceos-01
!
ntp 192.168.3.10
ntp 192.168.5.10
!
interface Management0
    ip address 198.51.100.11/24
```

Now, you have an IP address configuration line in the format you specified in your variable file, but what if the network vendor OS requires the data to be in a different format (for example, a full CIDR mask)?

Let's look at one available filter in the `ansible.utils` collection that uses the `netaddr` backend Python library:

3. If the `ansible.utils` collection was not installed (it is in the lab), you first need to install the Ansible collection:

```
root@cookbook-lab/chapters/ch02# ansible-galaxy collection install
ansible.utils (1)
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Downloading
... omitted output ...
ansible.utils:5.1.2 was installed successfully
```

(1) The `ansible.utils` collection is an Ansible collection that holds Ansible content. The Ansible public repository for collections is Ansible Galaxy. Collections will be explained further in the next chapter.

4. Then, you have to install the actual Python library required by the `ansible.utils.ipaddr` filter (it does not come installed with the Ansible collection):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# pip install netaddr
... omitted output ...
Successfully installed netaddr-1.3.0
```

5. Next, let's update the template to render the IP address in the full CIDR format: `198.51.100.11 255.255.255.0`.

You have to update the template to pass the available variable through the `ipaddr` Jinja2 filter so that we can accomplish this. The template turns into this format:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi templates/arista_
ceos.j2
... omitted output ...
{% for mgmt_int, int_details in interfaces['mgmt'].items() %}
interface {{ mgmt_int }}
    ip address \
{{ int_details['ip_addr'] | ansible.utils.ipaddr('address') }} \ (1)
{{ int_details['ip_addr'] | ansible.utils.ipaddr('netmask') }} (2)
{% endfor %}
```

In this template, you'll notice a new format that hasn't been explored yet. The pipe (`|`) is how you pass data through a filter. The data on the left-hand side of the pipe gets fed into the filter function as the first argument; further arguments are passed within the parentheses of the filter. Here are some examples:

- (1) In Python, this would look like `ipaddr('198.51.100.1', 'address')`.
- (2) This is how you would get the subnet mask: `ipaddr('198.51.100.1', 'netmask')`.



Note

You will learn more about Ansible filters and how you can use, expose, and create them in *Chapter 3*.

Finally, in the last section of this chapter, you will discover how to isolate playbook task executions by utilizing tags.

Using Ansible tags

Ansible tags are a powerful functionality that allows you to tag specific tasks within an Ansible playbook and provides the flexibility to choose which tasks to run within a given playbook based on the tags you specify. In this recipe, we will outline how to configure and use Ansible tags:

1. Let's get started by creating a new playbook called `ch2_pb_tags.yml` with the contents shown next:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# vi ch2_pb_tags.yml
---
```

```

- name: Using Ansible Tags
  hosts: arista_ceos
  gather_facts: no
  tasks:
    - name: Print OSPF
      ansible.builtin.debug:
        msg: "Router {{ inventory_hostname }} will Run OSPF"
      tags: [ospf, routing] (1)
    - name: Print BGP
      ansible.builtin.debug:
        msg: "Router {{ inventory_hostname }} will Run BGP"
      tags: (2)
        - bgp
        - routing
    - name: Print NTP
      ansible.builtin.debug:
        msg: "Router {{ inventory_hostname }} will run NTP"
      tags: ntp (3)

```

Upon executing the playbook, as seen in the preceding code snippet, Ansible will run all three tasks; however, if you pass in the `--tags` command-line switch, it filters the tasks to run based on that. Also, notice the different format of the tags in (1), (2), and (3). All are valid formats.

2. If you run the playbook with the routing tag, you will see how only tasks (1) and (2) are run:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-
playbook ch2_pb_tags.yml --tags routing
PLAY [Using Ansible Tags] *****
TASK [Print OSPF] *****
ok: [ceos-01] => {
    "msg": "Router ceos-01 will Run OSPF"
}
... omitted output ...
TASK [Print BGP] *****
ok: [ceos-01] => {
    "msg": "Router ceos-01 will Run BGP"
}

```

```
... omitted output ...
```

3. Now, if you run the playbook with the `ospf` tag, only task (1) runs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-
playbook -i hosts.yml ch2_pb_tags.yml --tags ospf
PLAY [Using Ansible Tags] *****
TASK [Print OSPF] *****
ok: [ceos-01] => {
    "msg": "Router ceos-01 will Run OSPF"
}
... omitted output ...
```

4. Finally, running the playbook with the `ntp` tag, it will only run task (3):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
ch2_pb_tags.yml --tags ntp
PLAY [Using Ansible Tags] *****
***
TASK [Print NTP] *****
***
ok: [ceos-01] => {
    "msg": "Router ceos-01 will run NTP"
}
... omitted output ...
```

Tags are a powerful feature of Ansible; they allow playbooks to be flexible and meet multiple requirements from one playbook. If you need to know what the available tags to filter on are, don't forget to use the `--list-tags` option!

```
root@cookbook-lab:~/cookbook-lab/chapters/ch02# ansible-playbook
ch2_pb_tags.yml --list-tags
playbook: ch2_pb_tags.yml
  play #1 (arista_ceos): Using Ansible Tags TAGS: []
    TASK TAGS: [bgp, ntp, ospf, routing]
```

Now that you've learned how to use Ansible tags to run specific tasks from a playbook, you have the basic building blocks in your repertoire to start automating some tasks!

Summary

Ansible is a framework you can get up to speed with quickly; once Ansible is installed and an inventory is created, you can start automating tasks. Ansible's YAML-based **domain-specific language (DSL)** is human-readable and comes with features such as variable interpolation, loops, and templating. As you continue into the next chapter, you will learn how to install and use collections, modify execution patterns, run playbooks in check mode, and many more advanced techniques.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



3

Advanced Ansible Concepts

As you continue to hone your skills on your way to becoming a power user of Ansible, it becomes increasingly important to know how to tune **Ansible's advanced settings** to achieve advanced outcomes. In order to manage infrastructure at any scale, you must feel confident using different execution strategies. These strategies can dictate how many services/network devices are being touched at the same time, what configuration Ansible is going to change, and how to handle variations of syntax. All of these questions and more will be answered in this chapter.

The main recipes covered in this chapter are the following:

- Using Ansible roles
- Using Ansible collections
- Running Ansible in check mode
- Using advanced YAML syntax
- Controlling execution strategies in Ansible
- Configuring fact caching in Ansible
- Creating custom Python filters for Ansible

This chapter builds on top of what you did in *Chapter 2*; the technical requirements described in that chapter will be required to successfully complete the recipes in this chapter.



Note

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch03.

To start this chapter, you will learn how to create and use Ansible roles. Automation focuses on creating reusable elements of work. In Ansible, one of the methods to organize code for reuse is called roles. You will discover these in the first section.

Using Ansible roles

Ansible roles promote code reusability and provide a simple method for packaging Ansible code in a simple way that can be shared and consumed. An **Ansible role** is a grouping of all the required Ansible tasks, variables, handlers, and Jinja2 templates that are packaged together in a specific structure. A role should be designed in order to deliver a specific function/task to facilitate its reusability. In this recipe, we will outline how to create an Ansible role and how to use it in your playbooks:

Let's start out by creating a new directory in the `ch03` folder, called `roles`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# mkdir roles && cd roles
```

Next, you will use the `ansible-galaxy` utility function to generate your role boilerplate code (this is a great way to get started!):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03/roles# ansible-galaxy init
basic_config
- Role basic_config was created successfully
```



Note

Ansible Galaxy is essentially a public repository where users can share roles and collections with the user community. More information can be found at <https://galaxy.ansible.com/ui/>.

After the `ansible-galaxy init` command has been executed, you will notice that an entire structure of boilerplate code is now in the `basic_config` role directory, with the added benefit of following Ansible's best practices for role layout:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03/roles# tree .
.
├── basic_config
│   ├── README.md
│   ├── defaults
│   │   └── main.yml
│   └── files
```

```
├─ handlers (1)
│   └─ main.yml
├─ meta
│   └─ main.yml
├─ tasks (2)
│   └─ main.yml
├─ templates (3)
├─ tests
│   └─ inventory
│       └─ test.yml
└─ vars (4)
    └─ main.yml
9 directories, 8 files
```

Each of the numbered references from the output is explained next:

- (1) The `handlers` folder contains the `main.yml` file, which includes all the handlers that should run as part of this role. A handler is a way in Ansible to only execute a task *if* something was changed in a prior step. Imagine a network device; perhaps you only want to run the `copy` configuration command if the configuration change resulted in an actual change on the device. This is a great use case for a handler.
- (2) The `tasks` folder contains the `main.yml` file, which lists all tasks (i.e., the playbook) that should be executed when we use this role.
- (3) The `template` folder contains all the Jinja2 templates that we will use as part of this role.
- (4) The `vars` folder contains all the variables that we want to define and that we will use in our role. The variables inside the `vars` folder have very high precedence when evaluating the variables while running the playbook.

Now that you've initialized the role and understand some of the crucial elements of creating a role, let's write the code to complete our role. This involves the following steps:

You will start by defining a *simple variable* in `vars`; here, you are simply assigning a directory name for the configurations you will generate:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi roles/basic_config/vars/main.yml
```

```
---
config_dir: basic_config
```


Now, you will create *reusable tasks* that you want this role to expose, using the variable from *step 1*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi roles/basic_config/
tasks/main.yml
---
- name: CREATE CONFIGS DIRECTORY (1)
  ansible.builtin.file:
    path: "{{ config_dir }}"
    state: directory
    run_once: yes
- name: GENERATE ARISTA BASIC CONFIG (2)
  ansible.builtin.template:
    src: "{{ ansible_network_os | split('.') | last }}.j2" (3)
    dest: "{{ config_dir }}/{{ inventory_hostname }}.cfg"
```

The numbered references from the output are explained next:

- (1) Ensures the `config_dir` directory you have set in vars exists; if it does not, then it creates it.
- (2) It then uses the Ansible `template` module to take a template and render the output into a destination directory.
- (3) This is a simple Jinja2 filter to take `network_os` `arista.eos.eos`, split it into a list of elements, and grab the last element, which is `eos`.

To supplement the task you just created, you need to create *Jinja2 templates* that will build the configurations. In this example, we will create just an Arista EOS NTP template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi
roles/basic_config/templates/eos.j2
hostname {{ inventory_hostname }}
!
{% for server in ntp_servers %}
ntp {{ server }}
{% endfor %}
```

Create a playbook called `pb_ansible_role.yml` in the `ch03` root directory:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03/# vi pb_ansible_role.yml
```

```
---
- name: BUILD BASIC CONFIG USING ROLES
```

```
hosts: arista_ceos
connection: local
vars:
  ntp_servers:
    - 8.8.8.8
    - 8.8.4.4
roles:
  - basic_config
```

Now, execute the playbook, which in turn looks at the `basic_config` role and by default runs the tasks defined in `roles/basic_config/tasks/main.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
ansible_role.yml
PLAY [BUILD BASIC CONFIG USING ROLES] *****
TASK [basic_config : CREATE CONFIGS DIRECTORY] *****
ok: [ceos-01]
TASK [basic_config : GENERATE ARISTA BASIC CONFIG] *****
changed: [ceos-01]
changed: [ceos-02]
```

This proves that the playbook to run the role didn't require any changes to the role itself. This exposes the power of roles and reusable code that can be packaged and shared, and does not need any source code changes to accomplish a goal.

As the Ansible ecosystem continues to grow, Ansible collections are used to share extensions among internal teams and/or community members. You will discover what collections are and how to use them next.

Using Ansible collections

Ansible Content Collections are pluggable packages that are easy to distribute, manage, and share. They allow an easy way to extend the Ansible framework. Ansible hasn't always had collections, though.

Ansible grew in popularity, support, and functionality, becoming burdensome to maintain the project as a monorepo. Thus, back in early 2021, with the release of Ansible major version 3.0, the project split into multiple libraries, as commented in *Chapter 1*. The main difference was the philosophy on installation, that is, the installation separation of core and community packages:

- **Ansible Core:** This core library offers extensibility features that allow Ansible to be extended. It comes with *no* collections pre-installed.
- **Community package:** Installs `ansible-core`, along with a long list of collections that have gone through a certification process and have been approved to be included as a community package.



Note

Both installation variants were covered in *Chapter 2*; all remaining steps are using the community package installation, which came with the lab VM provided.

Now that you understand when and why collections were introduced and how the project split impacts Ansible, let's talk about how to install and use an Ansible collection.

In this recipe, you will first install a collection via two popular methods. Secondly, you will write a playbook that uses a module from that collection by calling the **Fully Qualified Collection Name (FQCN)**. You can think of an FQCN like a file directory path; it simply explains the path within the Ansible collection to the module in use. The FQCN is the recommended naming approach for collections, but you could also see a simplified approach, only with the module name, for built-in modules.

For example, the commonly used debug module can be specified in two formats:

- `debug`: Module name only
- `ansible.builtin.debug`: FQCN to the module name

The first collection you will install is the `nokia.srlinux` collection manually. This is not included in the community package by default:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-galaxy collection
install nokia.srlinux
... omitted output ...
Installing 'nokia.srlinux:0.5.1' to '/root/.ansible/collections/ansible_
collections/nokia/srlinux'
nokia.srlinux:0.5.1 was installed successfully
... omitted output ...
```

Next, you install other collections by utilizing a `requirements.yml` file. This approach helps with dependency management, where all the requirements can be provided and installed by pointing to a file. First, you define a file with the dependencies:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi requirements.yml (1)
```

```
---
collections: (2)
  - "nokia.srlinux"
  - "junipernetworks.junos"
  - "arista.eos"
  - "networktocode.nautobot"
# roles: (3)
# - batfish.base
```

The numbered references from the output are explained next:

(1) `requirements.yml` is a best practice name for the file.

This is followed by a list of collections to install.

Optionally, a role key can be used to install roles from Ansible Galaxy.

With the dependency file defined, you install it with the `-r` option:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-galaxy collection
install -r requirements.yml
... omitted output ...
'nokia.srlinux:0.5.1' is already installed, skipping. (1)
'junipernetworks.junos:8.0.0' is already installed, skipping.
'arista.eos:9.0.0' is already installed, skipping.
'ansible.netcommon:6.1.3' is already installed, skipping.
'ansible.utils:4.1.0' is already installed, skipping.
Installing 'networktocode.nautobot:5.3.1' to '/root/.ansible/collections/
ansible_collections/networktocode/nautobot'
networktocode.nautobot:5.3.1 was installed successfully (2)
```

The numbered references from the output are explained next:

(1) Because the Nokia collection was installed in the previous step, it is not installed again.

(2) The `networktocode.nautobot` collection is not in the community packages for Ansible at the time of writing this book, so it gets installed like the Nokia one in the previous step.

After the collections have been installed, you can create a simple playbook to interact with one of the Arista cEOS devices that are in the topology. At this point, the collections for all the different vendors are installed, and you will arbitrarily use Arista at this time. This will demonstrate how

to call a module from a playbook using the FQCN. Notice we're calling the FQCN of `arista.eos.eos_command` and running a `show version` CLI command:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_collection_test.yml
```

```
---
- name: "VALIDATE COLLECTION IS WORKING"
  gather_facts: false
  connection: "ansible.netcommon.network_cli" (1)
  hosts: "ceos-01"
  tasks:
    - name: "RUN A COMMAND ON ARISTA EOS"
      arista.eos.eos_command:
        commands: "show version"
```

The numbered reference from the output is explained next:

(1) This `network_cli` connection plugin provides a connection over SSH. The plugin is typically used by network devices for sending and receiving CLI commands.

Finally, execute the playbook *and* pass command-line arguments to prompt for a password because no other authentication mechanism has been defined:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook -i hosts.
yml pb_collection_test.yml -u admin -k (1)
SSH password: ****
PLAY [VALIDATE COLLECTION IS WORKING]*****
TASK [RUN A COMMAND ON ARISTA EOS] *****
[WARNING]: ansible-pylibssh not installed, falling back to paramiko (2)
ok: [ceos-01]
PLAY RECAP *****
*
ceos-01  : ok=1  changed=0  unreachable=0  failed=0  skipped=0
          rescued=0  ignored=0
```

The numbered references from the output are explained next:

(1) The `-u` command argument can be used to pass a username. The `-k` argument is used to prompt for the SSH password. Ansible has a huge number of available arguments; use `ansible-playbook --help` to list them.

(2) This was used to demonstrate a quick way to test a playbook and pass the credentials; however, going forward, these will be in `*_vars` files or in a vault for protection.

You will learn how to fix this warning in the next section, but for now, this signals that the default library to connect with the device is not available in the Python environment.

Now that you know how to install and use Ansible collections, the next recipe will explore how to run a module in *check* mode. You will also learn how to determine whether a module supports that functionality.



Note

Not all modules support check mode; it is module-dependent. The recommendation is to review each module's documentation to determine whether check mode is implemented and to what extent.

Running Ansible in check mode

Ansible provides an option called **check mode**. In check mode (or **dry-run**), Ansible provides the user a way to validate the execution of a task without executing it. Thus, the task will be executed as a simulation. No changes will be made by the module; instead, a *report* is provided that shows what would be done. Furthermore, combined with the diff mode, with the addition of `--diff`, it provides further details on what will happen beyond just a summary line. However, not all modules support check and diff modes, but the ones that do offer the ability to validate changes before deploying them can be really useful.

One way to determine whether a module supports check mode is by looking at the module documentation. You can do this using the Ansible Galaxy portal, or alternatively, by using the command-line tooling, `ansible-doc`, that comes natively with Ansible (and the collections installed). For example, from the command line, you can run the following:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-doc ansible.
builtin.pip
... omitted ...
ATTRIBUTES:
  check_mode: (1)
  description: Can run in check_mode and return changed status prediction
               without modifying target, if not supported the action will be skipped.
  support: full (2)
... omitted ...
```

The numbered references from the output are explained next:

- (1) The documentation shows that `check_mode` is a supported attribute for the module.
- (2) `check_mode` has different levels of support; in this case, the built-in `pip` module completely supports check mode.

In this recipe, you will learn how to use the `ansible.builtin.pip` module to install `ansible-pylibssh`. This is the library that raised a warning in the previous playbook.

To address that warning, it seems obvious that you have to install the library. Here, we will demonstrate how to use check mode to validate what the `pip` module will do beforehand:

Create a new playbook called `pb_check_mode.yml`. This will target `localhost` and tell the `pip` built-in module to install the missing `ansible-pylibssh` library:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_check_mode.yml
---
- name: "INSTALL MISSING PYLIBSSH PLAY"
  gather_facts: false
  connection: "local"
  hosts: "localhost"
  tasks:
    - name: "PIP INSTALL ANSIBLE_PYLIBSSH"
      ansible.builtin.pip:
        name: "ansible-pylibssh"
```

However, when running the playbook, you add the `--check` command-line argument that summons the check mode. The output shows how `localhost` will be changed due to the installation of the new library:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook -check
pb_check_mode.yml -vvv (1)
... output omitted ...
PLAYBOOK: pb_check_mode.yml *****
1 plays in pb_check_mode.yml
PLAY [INSTALL MISSING PYLIBSSH PLAY] *****
TASK [PIP INSTALL ANSIBLE_PYLIBSSH] *****
task path: /root/cookbook-lab/chapters/ch03/pb_check_mode.yml:7
... output omitted ...
changed: [localhost] => {
```

```

    "changed": true,
    ... output omitted ...
    "stdout_lines": [
        "anaconda-anon-usage==0.4.4",
        "ansible==11.0.0",
        "ansible-core==2.18.0",
    ... output omitted ...
        "Pygments==2.18.0",
        (2)
        "PyNaCl==1.5.0",
    ... output omitted ...
    ]
}

```

The numbered references from the output are explained next:

- (1) -v is a command-line trigger to get more verbose output; *the more v's, the more verbosity* (up to six at the time of writing).
- (2) Notice that `pylibssh` is not shown in the check output, which tells the module to install it and set the `changed` keyword to `true`.

After you've validated what would happen, you can rerun the playbook without check mode to install the missing library with greater confidence. You can notice that the output is exactly the same as before, as you expected, but now, the changes have actually been applied:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_check_
mode.yml
PLAY [INSTALL MISSING PYLIBSSH PLAY] *****
TASK [PIP INSTALL ANSIBLE_PYLIBSSH] *****
changed: [localhost]

```

To validate that the library has been installed, you can rerun this playbook, and because it is idempotent, you don't need to do anything. You could also run the previous recipe and notice how the warning is no longer present:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_check_
mode.yml
PLAY [INSTALL MISSING PYLIBSSH PLAY] *****
TASK [PIP INSTALL ANSIBLE_PYLIBSSH] *****
ok: [localhost] (1)

```



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

The numbered reference from the output is explained next:

(1) The re-execution of the same playbook shows ok but has not changed, indicating that idempotency was already achieved.

As you can see, this recipe installs a missing Python library using Ansible. You were able to check the required changes before executing the playbook to commit the changes. Check mode provides a great way to pre-validate configuration changes before they happen.



Note

Adding the `pylibssh` library was a change that was required on our Ansible control host; therefore, `hosts: localhost` was used, along with the connection: `local` setting. Both of these tune Ansible to run against the local control host.

Now that you’ve seen the power of the Ansible dry-run feature, let’s quickly look at ways you can make your YAML files more readable by applying the **Don’t Repeat Yourself (DRY)** pattern.

Using advanced YAML syntax

As we mentioned throughout the first few chapters, Ansible relies heavily on YAML. As long as you keep creating more complex playbooks, you need to incorporate more complex variables, data structures, and arguments. Using advanced YAML syntax can help solve these complexities.

YAML multiline strings

In this recipe, you will learn how to use **YAML multiline strings**. The YAML documentation describes several different enumerations of handling multiline strings, but for Ansible, you will learn about **literal** and **block** formatting. The following are the steps involved:

Start by creating a new playbook called `pb_yaml_multiline.yml`. You will use the built-in debug method to output a text with more than one line. The **literal** block is used for a **literal block**. A literal block will keep newline characters, and extra levels of indentation are also preserved:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_yaml_multiline.yml
---
- name: "Explore YAML Multiline Strings"
```

```

hosts: "localhost"
tasks:
  - name: "Debug Literal Block"
    ansible.builtin.debug:
      msg: |
        banner login $
        *****
        This is an example login banner.
        *****
        $

```

When executing the playbook, you will notice it keeps the formatting while making the playbook more readable:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_yaml_
multiline.yml
PLAY [Explore YAML Multiline Strings]*****

TASK [Debug Literal Block]*****
ok: [localhost] => {
  "msg":
    "banner login $
    *****
    This is an example login banner.
    *****"
}

```

The second multiline type is the **folded block**. You can add a new task to the same playbook, but instead of using `|`, you now use `>`. The `>` syntax is used for a folded block. It replaces newline characters with a single space; however, if a nested indentation block is encountered, it preserves newline characters:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_yaml_multiline.yml
... omitted ...
- name: "Debug Folded Style"
  ansible.builtin.debug:
    msg: >
    banner login $
    *****

```

```

    This is an example login banner.
    *****

    $

```

By executing the playbook, you will notice that it also keeps the formatting:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_yaml_
multiline.yml

PLAY [Explore Advanced YAML Mechanisms]*****
... omitted ...
TASK [Debug Folded Style] *****
ok: [localhost] => {
    "msg":
        "banner login $
        *****

        This is an example login banner.
        *****

        $"
    }

```

The literal and folded multilines also support a *stripping* option that follows the same principles as the two previous examples; however, it strips the trailing blank lines:

In the same playbook, you can add two new tasks using the `|-` and `>-` syntax that perform the same functions as before, but now with stripping:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_yaml_multiline.yml
---
... omitted ...
- name: "Debug Literal Block with Striping"
  ansible.builtin.debug:
    msg: |-
      banner login $
      *****

      This is an example login banner.
      *****

      $
- name: "Debug Folded Style with Striping"
  ansible.builtin.debug:

```

```

msg: >-
banner login $
*****

    This is an example login banner.
*****

$

```

By executing the playbook, you will notice that both options strip the trailing blank/newline:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_yaml_
multiline.yml
PLAY [Explore Advanced YAML Mechanisms]*****
... omitted ...

TASK [Debug Literal Block with Striping] *****
ok: [localhost] => {
  "msg":
    "banner login $
    *****

    This is an example login banner.
    *****

    $"
}
TASK [Debug Folded Style with Striping] *****
ok: [localhost] => {
  "msg":
    "banner login $
    *****

    This is an example login banner.
    *****

    $"
}

```

The second advanced YAML syntax is the anchors and aliases that are used together.



Note

More information on multiline YAML syntax can be found at:

https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html.

YAML anchors and aliases

To close out this section, you will learn how to use **YAML anchors**. Anchors and aliases can greatly help with keeping your YAML files simpler, avoiding repeating the same data by allowing the reuse of variable data. In YAML, an anchor (&) marks a point in the document for later reference, allowing you to reuse that data elsewhere using an alias (*).

In this recipe, you will learn how to share a generic variable using an anchor. This will be showcased later in the book, for example, in *Chapter 9*, where you will reuse multiple Azure arguments using an anchor:

Start by creating a new playbook called `pb_yaml_anchors.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_yaml_anchors.yml
---
- name: "Explore YAML Anchors"
  hosts: "localhost"
  vars:
    &example_anchor (1)
    msg: "Demonstrating reusing a variable as an anchor."
    verbosity: 0
  tasks:
- name: "Default Anchor Message and Verbosity"
  ansible.builtin.debug:
    <<: *example_anchor (2)
- name: "Overloaded Anchor Message with Default Verbosity"
  ansible.builtin.debug:
    <<: *example_anchor (3)
    msg: "Demonstrating OVERLOADED msg argument from anchor."
```

The numbered references from the output are explained next:

- (1) The & sign defines the anchor that we can reuse as needed. The variable name to use for the anchor is `example_anchor`.
- (2) Calling the debug module, and instead of having to define required module arguments, we're passing them in from the anchor and passing them through the * unpacking operation.
- (3) The same as task (2), but anchors allow for the merging of data, which will overload the `msg` argument with a new value (`msg`) just for this task.



Note

<< is used as a **merge key**, where data from multiple mappings can be merged together. A more detailed definition can be found in the YAML spec: <https://yaml.org/type/merge.html>.

Executing the playbook shows the power of being able to define some defaults for module arguments, with the flexibility to overload individual arguments as needed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_yaml_anchors.yml
PLAY [Explore YAML Anchors] *****

TASK [Debug Using Default Anchor Message and Verbosity] *****
ok: [localhost] => {
    "msg": "Demonstrating reusing a variable as an anchor."
}
TASK [Debug Using Overloaded Anchor Message with Default Verbosity]
*****
ok: [localhost] => {
    "msg": "Demonstrating OVERLOADED msg argument from anchor."
}
... omitted ...
```

With advanced YAML syntax covered, let's continue to another concept of Ansible that can be used to control how Ansible executes against the inventory of devices. This feature in Ansible is called a **strategy**.

Controlling execution strategies in Ansible

Adjusting Ansible playbook execution strategies is a common task among Ansible users. Different changes may require a certain order of operations. The default strategy Ansible uses is called **linear**, and it works great a majority of the time, but what if you're automating a task that needs to be done in a specific order or way?

Strategies can be used to define *how* a playbook should be executed. Some basic strategies that can be used to control parallelism are the following:

- **linear (default)**: The playbook runs in a linear execution, each task is run in order, and the next task is not started until all hosts are finished with the current task

- `debug`: It allows the user to run the playbook interactively for troubleshooting purposes
- `free`: The playbook is executed linearly, but each host runs the tasks at its own pace and has no requirement for all hosts to finish a task before continuing to the next task
- `host_pinned`: Operating like the `free` strategy, `host_pinned` will run tasks as fast as it can on the number of hosts specified in the `serial` keyword, immediately starting a host as soon as one ends

To see available strategies, use the `ansible-doc -t strategy -l` command.

Strategies can be set per `PLAY` instance or under the `[defaults]` group (in the `ansible.cfg` file).

Ansible also provides some extra strategies that can be used to further tune how a playbook executes:

- `forks`: This defines the number of simultaneous connections a task can use to perform work. The `forks` strategy can be tuned based on the processing power available. The default is set to 5. If processing capacity on the Ansible control machine is not a concern, this can be modified. This allows for faster playbook execution.
- `run_once`: This option is exactly like its name suggests: a task that should be run only once. It is great for operations that need to be done only once for the group, not once per host. A good example is creating a directory on the control machine for storing configuration files. You do *not* need every host to create a directory; this needs to be completed only once.
- `serial`: The number of hosts that Ansible should be running side by side. It is great for playbooks that include redundant devices that should not be worked on at the same time or for when there are possible race conditions.
- `throttle`: This can be used in the task definition to limit the number of hosts the task is executed on. It is similar to `fork`, but at a task level.
- `order`: This specifies how the hosts for a given play are going to be executed. By default, the order comes from the inventory file, but other options exist that can provide flexibility if inventory hostnames provide insights into a given environment. Here are some examples:
- `reverse_inventory`: This is the opposite of the default mode explained previously. It reverses the order from the inventory file.
- `shuffle`: This follows a random order.
- `sorted`: This follows alphabetical order.
- `reverse_sorted`: This follows reverse alphabetical order.

In this recipe, you will use some different strategies and settings (linear is the default strategy and will not be demonstrated in this recipe). The playbooks used will be dramatically simplified to allow the focus to be on strategies in use rather than the content of the playbooks.

The free strategy

In this recipe, you will explore the free strategy:

Let's define a new playbook called `pb_strategy_free.yml`. This playbook will demonstrate how Ansible's default strategy works:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_strategy_free.yml
---
- name: "DEMONSTRATE FREE STRATEGY"
  hosts: "all"
  gather_facts: False
  strategy: "free"
  tasks:
  - name: "GET DEVICE NAMES WITH NETWORK OS"
    ansible.builtin.debug:
      msg: "Device name {{ inventory_hostname }} => {{ ansible_network_
os }}"
```

Then, when the playbook is executed, the first thing you notice is that each task isn't completed before the next task is executed. The free strategy allows Ansible to run each host independently and execute tasks as quickly as it can. It is a great option when tasks aren't dependent on one another or the hosts themselves aren't dependent on one another. A good example might be collecting operation state data after a change:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
strategy_free.yml
PLAY [DEBUG INVENTORY]*****

TASK [GET DEVICE NAMES]*****
ok: [client1] => {
    "msg": "client1"
}
ok: [ceos-01] => {
    "msg": "ceos-01"
}
```



```

ok: [crpd-01] => {
    "msg": "crpd-01"
}
... output omitted ...
TASK [GET DEVICE OS]*****
ok: [client1] => {
    "msg": "linux"
}
ok: [client2] => {
    "msg": "linux"
}
ok: [ceos-01] => {
    "msg": "arista.eos.eos"
}
... output omitted ...

```

The host_pinned strategy

Next, you will test the `host_pinned` strategy:

The `host_pinned` strategy is a bit trickier, and in the backend, it uses the `serial` argument to determine the batch number used. If the `serial` argument is not set, it defaults to `all`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_strategy_hostpin.yml
---
- name: "DEMONSTRATE HOSTPIN STRATEGY"
  hosts: "all"
  gather_facts: False
  strategy: "host_pinned"
  serial: 2
  tasks:
... same tasks as the previous example ...

```

Pay close attention to each iteration. When `serial` is used, it loops over `PLAY` itself with the number specified in the `serial` keyword (i.e., 2). In this playbook output, `PLAY` was executed four times, each time running through the tasks with two hosts:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
strategy_hostpin.yml
PLAY [DEMONSTRATE HOSTPIN STRATEGY]*****

TASK [GET DEVICE NAMES]*****
ok: [client1] => {
    "msg": "client1"
}
ok: [client2] => {
    "msg": "client2"
}
TASK [GET DEVICE OS]*****
ok: [client1] => {
    "msg": "linux"
}
ok: [client2] => {
    "msg": "linux"
}
PLAY [DEMONSTRATE HOSTPIN STRATEGY]*****

TASK [GET DEVICE NAMES]*****
ok: [ceos-01] => {
    "msg": "ceos-01"
}
ok: [ceos-02] => {
    "msg": "ceos-02"
}

TASK [GET DEVICE OS]*****
ok: [ceos-01] => {
    "msg": "arista.eos.eos"
}
ok: [ceos-02] => {
    "msg": "arista.eos.eos"
}
... output omitted ...
```

Now that the main strategies available have been demonstrated, the next recipe will focus on the `serial` keyword.

The serial keyword

Using `serial` without combining it with a strategy allows for further flexibility. Running playbooks in batches defined by the administrator can help validate that a playbook works as expected before executing it against a larger batch of hosts. Since the previous example showed what `serial` does, the following shows a few options when specifying the `serial` option:

In this first `serial` example, we define a list of values that will determine the number of hosts per execution:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_serial_numbers.yml
---
- name: "DEMONSTRATE SERIAL NUMBERS"
  hosts: "all"
  gather_facts: False
  serial:
    - 1
    - 2
    - 5
  tasks:
    ... same tasks as the previous example ...
```

Notice that when this playbook is executed, `PLAY` is first executed against a batch of 1 device, then 2, then 5, based on our definition. You will see this in the following playbook execution:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
serial_numbers.yml
PLAY [DEMONSTRATE SERIAL NUMBERS] *****

TASK [GET DEVICE NAMES] *****
ok: [client1] => {
  "msg": "client1"
}

TASK [GET DEVICE OS]*****
ok: [client1] => {
  "msg": "linux"
```

```
}

PLAY [DEMONSTRATE SERIAL NUMBERS] *****

TASK [GET DEVICE NAMES] *****
ok: [client2] => {
    "msg": "client2"
}
ok: [ceos-01] => {
    "msg": "ceos-01"
}

TASK [GET DEVICE OS]*****
ok: [client2] => {
    "msg": "linux"
}
ok: [ceos-01] => {
    "msg": "arista.eos.eos"
}

PLAY [DEMONSTRATE SERIAL NUMBERS] *****
TASK [GET DEVICE NAMES]*****
ok: [ceos-02] => {
    "msg": "ceos-02"
}
ok: [crpd-01] => {
    "msg": "crpd-01"
}
ok: [crpd-02] => {
    "msg": "crpd-02"
}
ok: [sr1-02] => {
    "msg": "sr1-02"
}
ok: [sr1-01] => {
    "msg": "sr1-01"
}
```

```

TASK [GET DEVICE OS]*****
ok: [ceos-02] => {
    "msg": "arista.eos.eos"
}
ok: [crpd-01] => {
    "msg": "junipernetworks.junos.junos"
}
ok: [srl-01] => {
    "msg": "nokia.srl"
}
ok: [crpd-02] => {
    "msg": "junipernetworks.junos.junos"
}
ok: [srl-02] => {
    "msg": "nokia.srl"
}
... omitted output ...

```

Finally, you will try this same playbook again, but replacing the list of serial options, you will provide a percentage of hosts to run against:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_serial_percentage.
yaml
---
- name: "DEMONSTRATE SERIAL PERCENTAGE"
  hosts: "all"
  gather_facts: False
  strategy: "host_pinned"
  serial: 30%
  tasks:
... same tasks as the previous example ...

```

Notice that each PLAY instance is run against 30% of the hosts at a time:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
serial_percentage.yaml
PLAY [DEMONSTRATE SERIAL PERCENTAGE] *****

```

```
TASK [GET DEVICE NAMES] *****
ok: [client1] => {
    "msg": "client1"
}
ok: [client2] => {
    "msg": "client2"
}

TASK [GET DEVICE OS] *****
ok: [client1] => {
    "msg": "linux"
}
ok: [client2] => {
    "msg": "linux"
}

PLAY [DEMONSTRATE SERIAL PERCENTAGE] *****

TASK [GET DEVICE NAMES] *****
ok: [ceos-01] => {
    "msg": "ceos-01"
}
ok: [ceos-02] => {
    "msg": "ceos-02"
}

TASK [GET DEVICE OS]*****
ok: [ceos-01] => {
    "msg": "arista.eos.eos"
}
ok: [ceos-02] => {
    "msg": "arista.eos.eos"
}

PLAY [DEMONSTRATE SERIAL PERCENTAGE] *****

TASK [GET DEVICE NAMES] *****
```

```
ok: [crpd-01] => {
    "msg": "crpd-01"
}
ok: [crpd-02] => {
    "msg": "crpd-02"
}

TASK [GET DEVICE OS]*****
ok: [crpd-01] => {
    "msg": "junipernetworks.junos.junos"
}
ok: [crpd-02] => {
    "msg": "junipernetworks.junos.junos"
}

PLAY [DEMONSTRATE SERIAL PERCENTAGE] *****

TASK [GET DEVICE NAMES] *****
ok: [sr1-01] => {
    "msg": "sr1-01"
}
ok: [sr1-02] => {
    "msg": "sr1-02"
}

TASK [GET DEVICE OS] *****
ok: [sr1-01] => {
    "msg": "nokia.sr1"
}
ok: [sr1-02] => {
    "msg": "nokia.sr1"
}
... omitted output ...
```

There are a number of other options, for example, throttled and ordered. The important takeaway from this recipe is that Ansible is incredibly flexible and the framework offers support for many execution strategies, not to mention a pluggable way to write your own!

As you continue into the next section, you will learn how to use Ansible to collect facts on inventory hosts, with the added flexibility to store the resulting data using a caching solution.

Configuring fact caching in Ansible

Fact caching is an important feature that can help you optimize and speed up the execution time of your playbooks when you require facts to be collected, for example, from your infrastructure.

Ansible is a powerful tool to collect information about the operational state of our infrastructure, and we can use this information to generate configurations, build reports, and also validate the state of our infrastructure. In cases where the state of our infrastructure is highly stable, we might not need to collect network facts from our devices during every playbook run. In such cases, we might opt to use fact caching to speed up the execution of our playbooks. Once facts have been gathered, the next time we read the facts (network state) of our devices, we will get them from a stored location on the Ansible control node, instead of connecting to the devices and collecting information from the live network.

Aside from the speed savings, there are also traffic savings that can be relevant in some **out-of-band (OOB)** environments. Let's get started with getting our recipe set up to explore fact caching:

First, to enable fact caching, you update the `ansible.cfg` file and set the output directory for the facts to be stored. You also set the backend type that you will use to store the fact data. There are multiple options, ranging from YAML or JSON files to storing this data in Redis or Memcached databases. In our example, for simplicity, we will use local YAML files to store the facts collected from the devices. We will also specify the folder location to store this information (`./fact_cache`):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi ansible.cfg
[defaults]
... omitted ...
fact_caching=yaml
fact_caching_connection=./fact_cache
```

1. Create a new playbook, `pb_get_facts.yml`, to collect facts from the network using different approaches:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_get_facts.yml
---
- name: "Collect Network Facts"
  hosts: "arista_ceos"
  tasks:
```



```
- name: "Collect Facts from EOS"
  arista.eos.eos_facts:
    gather_subset: "all" (1)
- name: "Set and Cache Custom Fact"
  ansible.builtin.set_fact:
    site: "Egypt" (2)
    cacheable: yes (3)
```

The numbered references from the output are explained next:

- (1) Most modules implement a `gather_facts` module; within these, the collection author can support gathering subsets of the facts instead of gathering all details. In this recipe, we want to gather all the facts the module can return.
- (2) Ansible's built-in `set_fact` module provides a mechanism to add a variable (fact) into (2) Ansible's variable store. Here, you have a variable with a key named `site` with a value of `Egypt`.
- (3) The `cacheable` argument is set to either allow or disallow the ability for Ansible to cache the fact.

Run the new Ansible playbook on a single node from our inventory using the `-l` option:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_get_
facts.yml -l ceos-01
PLAY [Collect Network Facts] *****

TASK [Collect Facts from EOS] *****
ok: [ceos-01]
TASK [Set and Cache Custom Fact] *****
ok: [ceos-01]
```

Next, you can check that the new folder has been created and that a new YAML file for each node within our inventory is stored in this location. All the facts collected by these modules are saved in these YAML files, as shown in the following output:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# tree fact_cache/
fact_cache/
├── ceos-01
```

With the directory structure understood, let's visualize some of the output from the file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# tail -n 5 fact_cache/
ceos-01
```

```

ansible_net_serialnum: 91D0D5FE4238E93E1626790BC6EEA561
ansible_net_system: eos
ansible_net_version: 4.30.1F-32308478.4301F (engineering build)
ansible_network_resources: {}
site: Egypt

```

As shown in the cached facts file, you have many facts cached from the `eos.eos.get_facts` module, and also the `site: Egypt` cached entry that we set via a manual `set_facts` module.

Now, you can write another playbook to use the variables that we cached in the previous playbook, to examine what data is cached; explore the `fact_cache/` directory based on the caching configuration in `ansible.cfg`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_validate_from_cache.
yaml
---
- name: "Validate Cache Data"
  connection: "local"
  hosts: "all"
  tasks:
- name: "Validate all Interfaces"
  ansible.builtin.assert:
    that:
      - item.value.lineprotocol == "up"
  loop: "{{ ansible_net_interfaces | dict2items }}"
- name: "Validate Custom Fact"
  ansible.builtin.assert:
    that:
      - site == "Egypt"

```

Upon executing the playbook, you see that you've been able to run an **Ansible test (assert)** that the cached data is present, and secondly, that all the interfaces are up on the `ceos-01` device:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_
validate_from_cache.yaml -l ceos-01
PLAY [Validate Cache Data] *****

TASK [Validate all Interfaces] *****
... other interfaces omitted ...
ok: [ceos-

```

```

01] => (item={'key': 'Management0', 'value': {'bandwidth': 1000000000,
'masklen': 24}, 'lineprotocol': 'up', 'macaddress': '02:42:c6:33:64:0b',
'mtu': 1500, 'operstatus': 'connected', 'type': 'routed'}}) => {
  "ansible_loop_var": "item",
  "changed": false,
  "item": {
    "key": "Management0",
    "value": {
      "bandwidth": 1000000000,
      "description": "",
      "duplex": "duplexFull",
      "ipv4": {
        "address": "198.51.100.11",
        "masklen": 24
      },
      "lineprotocol": "up",
      "macaddress": "02:42:c6:33:64:0b",
      "mtu": 1500,
      "operstatus": "connected",
      "type": "routed"
    }
  },
  "msg": "All assertions passed"
}

TASK [Validate Custom Fact] *****
ok: [ceos-01] => {
  "changed": false,
  "msg": "All assertions passed"
}

```

In this playbook, you are utilizing the variables collected from the cache (`ansible_net_interfaces`, in this example) and running the tasks against the devices in the inventory. By default, the entries in the cache are valid only for a specific amount of time, controlled by the `timeout` value for our cache, to ensure that any outdated state in our cache will not be considered. This value is controlled by the `fact_caching_timeout` option, which can be set in the `ansible.cfg` file, and by default is 86400 seconds.

**Note**

For more information on Ansible's fact caching functionalities, see:

<https://docs.ansible.com/ansible/latest/plugins/cache.html>.

The final section will teach you how to write and use custom Python filters inside your playbooks. The filters are Jinja2 filters that allow you to accomplish more sophisticated data transformations.

Creating custom Python filters for Ansible

Ansible provides a rich set of filters out of the box. You used some of these in *Chapter 2*, but network automation can expose some challenging situations that the out-of-the-box filters cannot directly solve. As is true with most features of Ansible, you can extend the framework to create your own filters to use within Ansible. This exposes the ability to solve complex data problems by passing data directly to a Python function and getting back data exactly how you need it for your goal.

**Note**

This recipe requires basic Python knowledge.

In this recipe, you will create a custom filter. Ansible exposes a standard directory structure where filters are automatically loaded into the environment. One of these common locations is a folder called `filter_plugins` in the root of the directory where the playbook sits:

To get started, create a `filter_plugins` directory in the project root directory. Then, within the directory, you can create your custom filter. Once again, Ansible provides a mechanism to streamline the creation of a custom filter with a new file in the `filter_plugins/` directory, called `filter.py`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# touch filter_plugins/
filter.py
```

In `filter.py`, you define a new Python class with some expected methods:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03/filter_plugins# vi filter.
py
class FilterModule(object): (1)
    """ACL State Filter."""
    def filters(self): (2)
        """Default filters method to overload."""
```

```

    return {"acl_state": self.acl_state} (3)
def acl_state(self, acl_def): (4)
    """ACL State filter logic."""
    for acl_name, acl_rules in acl_def.items():
        for rule in acl_rules:
            rule["state"] = rule["state"].upper()
    return acl_def (5)

```

The numbered references from the output are explained next:

- (1) This is a Python class definition using the Ansible expected class name, `FilterModule`.
- (2) The filter method of the class is used to register filter names into the Ansible/Jinja2 environment.
- (3) This example registers a filter called `acl_state` (the dictionary key), and when that filter is executed, the `acl_state` class method (the dictionary value) is called.
- (4) The filter method is a standard Python method; the `acl_state` method accepts one argument called `acl_def`.
- (5) The return value of the method call is the resulting data structure that will be returned to the Ansible filter call.

Now that you know how to create a custom filter and the required elements that the Ansible framework expects, let's explore how to actually use the custom filter within an Ansible playbook:

Create a new playbook called `pb_test_custom_filter.yml` that uses the new filter, `acl_state`, to manipulate a variable:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_test_custom_filter.
yml
---
- name: "Test Custom Filter"
  hosts: "all"
  connection: "local"
  tasks:
- name: "Read ACL data"
  ansible.builtin.include_vars: "ACLs.yml"
  run_once: yes
  delegate_to: "localhost"
- name: "Apply Our Custom Filter"

```

```
ansible.builtin.set_fact:
  standard_acl: "{{ ACLs | acl_state }}"
run_once: yes
delegate_to: "localhost"
- name: "Display Output After Filter"
  ansible.builtin.debug: "var=standard_acl"
```

Create an extra vars file called `ACLs.yml` with the ACLs' definition. Notice that this is not part of `host_vars` or `group_vars`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi ACLs.yml
---
ACLs: (1)
  INFRA_ACL: (2)
- src: "10.1.1.0/24" (3)
  dst: "any"
  dport: "ssh"
  state: "present"
- src: "10.2.1.0/24"
  dst: "any"
  app: "udp"
  dport: "snmp"
  state: "present"
- src: "10.3.2.0/24"
  dst: "10.2.2.0/24"
  dport: "dns"
  app: "udp"
  state: "present"
```

The numbered references from the output are explained next:

- (1) This creates a variable data structure called `ACLs`.
- (2) An additional nested dictionary key is called `INFRA_ACL`.
- (3) The first list index stores a standard ACL entry.

Then, when you execute the playbook, you will see how the state key value is converted into uppercase, as defined in the custom filter:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# ansible-playbook pb_test_
custom_filter.yml
PLAY [Test Custom Filter] *****

TASK [Read ACL data] *****
ok: [client1 -> localhost] (1)
TASK [Apply Our Custom Filter] *****
ok: [client1 -> localhost]
TASK [Display Output After Filter] *****
ok: [srl-02] => {
  "standard_acl": { (2)
    "INFRA_ACL": [
      {
        "dport": "ssh",
        "dst": "any",
        "src": "10.1.1.0/24",
        "state": "PRESENT"
      },
      {
        "app": "udp",
        "dport": "snmp",
        "dst": "any",
        "src": "10.2.1.0/24",
        "state": "PRESENT"
      },
      {
        "app": "udp",
        "dport": "dns",
        "dst": "10.2.2.0/24",
        "src": "10.3.2.0/24",
        "state": "PRESENT"
      }
    ]
  }
}
... omitted output ...
```

The numbered references from the output are explained next:

(1) The first two tasks executed are showing `[client 1 -> localhost]`. This is due to the use of the `delegate_to: localhost` definition on these tasks. These not only need to be executed once (defined with `run_once`), but it also makes sense to use the control host to do these instead of relying on remote host resources.

(2) The debug shows the rendered `standard_acl` instance that took our variable data, ran it through our custom Ansible filter, and returned the properly formatted data (with uppercase states).

Finally, what if you need to pass in more than a single variable into a filter? Well, good news—you simply extend your method definition to include additional arguments:

Create a new filter called `custom_acl`, where we expect an argument called `acl_def` and an optional keyword argument named `field`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi filter_plugins/filter.py
... omitted ...
def custom_acl(self, acl_def, field=None):
    for acl_name, acl_rules in acl_def.items():
        for rule in acl_rules:
            if field and field in rule.keys():
                rule[field] = rule[field].upper()
    return acl_def
def filters(self):
    return {
        'acl_state': self.acl_state,
        'custom_acl': self.custom_acl
    }
```

Notice that we simply add the new method, `custom_acl`, into the same filter method so that we properly register it to be used.

To avoid demonstrating a nearly identical playbook, you would call this filter that has multiple `args/kwarg`s by adding this task to your playbook, where ACLs become the first argument, and additional `args/kwarg`s are added inside the filter's parentheses, just like a *Python callable*:


```
root@cookbook-lab:~/cookbook-lab/chapters/ch03# vi pb_test_custom_filter.
.yml
... output omitted ...
- name: Apply Our Custom Filter
  ansible.builtin.set_fact:
    standard_acl: "{{ ACLs | acl_state }}"
    final_acl: "{{ ACLs | custom_acl('dport') }}"
```

As you close out this chapter, you’ve learned how to use custom Ansible filters to manipulate data to achieve more complex data structures. Custom filters open the doors for programmers to add complex data transformation logic using a more familiar programming language; there is no shortage of valuable use cases for custom filters.



Note

A great source of examples is in the Ansible core repository in the `core.py` file: <https://github.com/ansible/ansible/blob/devel/lib/ansible/plugins/filter/core.py>.

Summary

As you built on top of the Ansible basics you learned in *Chapter 2*, in this chapter, you discovered multiple advanced Ansible techniques. Throughout the rest of the book, you will see these advanced concepts used on many occasions: you will be installing more collections, executing playbooks in check mode, and using multiple custom Python filters to achieve the network automation design that we will be introducing in the next few chapters.

With these Ansible fundamentals, you are ready to explore the other parts of this book, where you will learn how to manage your network infrastructure with real examples.

Unlock this book’s exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



Part 2

Managing Network Infrastructure with Ansible

With the foundational Ansible concepts in place, this part of the book shifts focus to real-world network automation, demonstrating how to apply Ansible to manage and operate network infrastructure across multiple vendors.

Through hands-on examples, you'll learn how to automate end-to-end workflows—from generating and pushing configurations to retrieving operational state and validating intent. Each chapter targets a specific platform or approach, giving you practical experience with tools and techniques used in production environments.

You'll start by building an Arista EOS data center fabric, then move on to automating a Juniper CRPD-based core, followed by managing Nokia SR Linux devices in a campus environment—all using Ansible as the central automation engine. Finally, you'll explore how to simplify multi-vendor automation using NAPALM, a powerful abstraction layer that brings consistency to diverse device environments.

Whether you're working in a homogeneous environment or navigating the complexity of multi-vendor networks, this part will equip you with the skills and patterns needed to scale network automation with confidence and clarity.

This part of the book includes the following chapters:

- *Chapter 4, Building Data Center Networks with Arista*
- *Chapter 5, Automating a Juniper Network Core Using Ansible*
- *Chapter 6, Managing Nokia SR Devices Using Ansible*
- *Chapter 7, Administering a Multi-Vendor Network with NAPALM and Ansible*

4

Building Data Center Networks with Arista

Now that you've learned the basic and advanced concepts of Ansible, it's time to start *automating networks*! In this book, we want to offer you realistic network scenarios with common network vendors but without overcomplicating the network architecture so that you can focus on the automation part. The first scenario in *Part 2* is a data center fabric using **Arista EOS**. This chapter outlines how to manage Arista EOS devices using Ansible modules and illustrates how Ansible connects to Arista devices and helps you configure and collect operational data.

In this chapter, we will start building the Arista configurations for our topology that was described in *Chapter 2*, but you could apply the learning to any type of network scenario or network vendor (leveraging the right Ansible collections). Remember that the lab is available by following the instructions in *Chapter 16*. The lab topology you will be creating over the next three chapters is shown in the following figure for reference. In this chapter, you will be focused on the Arista Data Center Fabric.

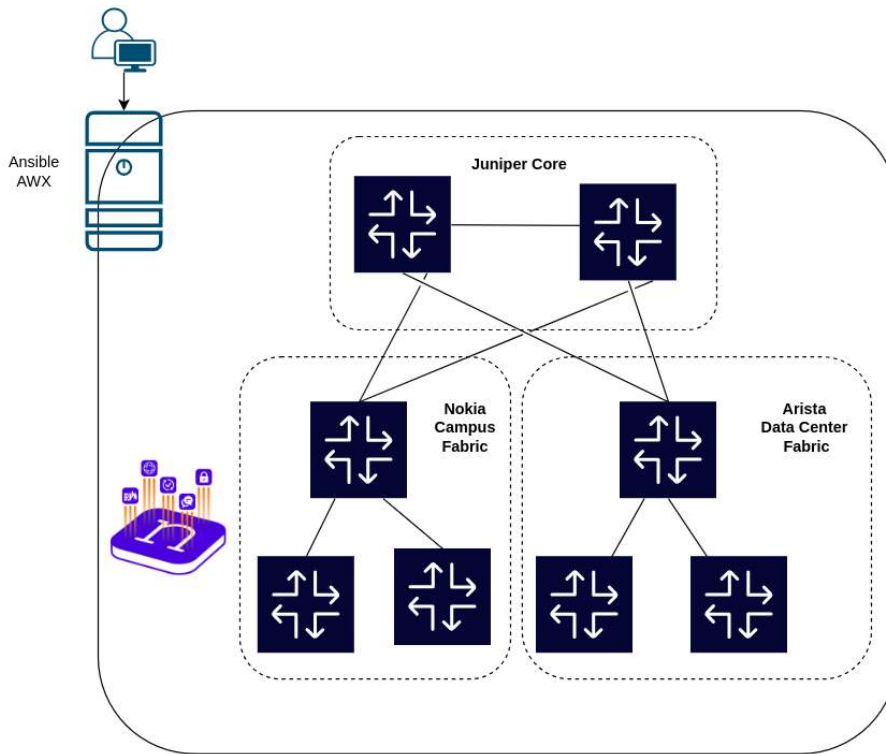


Figure 4.1 – Lab environment setup

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



**Note**

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch04.

The recipes covered in this chapter are as follows:

- Augmenting the Ansible network inventory
- Connecting to Arista devices
- Migrating connections from SSH to HTTP(S) by utilizing the Arista eAPI
- Creating and managing users and authentication options
- Configuring generic system options on Arista devices
- Configuring Arista interfaces using Ansible
- Advanced fabric configuration for Arista Data Center
- Finalizing BGP configuration to the core
- Pushing configuration to Arista devices
- Collecting Arista device facts using Ansible
- Running arbitrary commands on Arista devices

Let's get started by getting our inventory prepared to automate our data center fabric.

Augmenting the Ansible network inventory

In *Chapter 2*, you learned how to create and manage the key elements of an Ansible inventory. In this chapter, you will focus on interacting with **Arista cEOS devices** to deploy a very simple **data center fabric** with *one spine* and *two leaves*. In the first recipe, you will extend the existing Ansible inventory from *Chapter 2* to create two new groups:

- `spines` will hold your single spine device
 - `leafs` will hold both leaf devices
1. First, create the `ch04` directory under the `chapters` folder, and reuse the `hosts.yml` file from the inventory that was covered in *Chapters 2* and *3*, adding `ceos-03`, which is the second leaf in our topology:

```
root@cookbook-lab:~/cookbook-lab/chapters# mkdir ch04 && cp ch03/
```

```
hosts.yml ch04/hosts.yml && cd ch04/ && vi hosts.yml
---
network_equipment:
  children:
    arista_ceos:
arista_ceos:
  hosts:
    ceos-01:
      ansible_host: 198.51.100.11
    ceos-02:
      ansible_host: 198.51.100.12
    ceos-03:
      ansible_host: 198.51.100.13
```

 Quick tip: Enhance your coding experience with the AI Code Explainer and Quick Copy features. Open this book in the next-gen Packt Reader. Click the Copy button

(1) to quickly copy code into your coding environment, or click the Explain button

(2) to get the AI assistant to explain a block of code to you.

```
function calculate(a, b) {
  return {sum: a + b};
};
```

Copy

Explain

1

2



The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



2. Next, you create a new group called `spines` and assign `ceos-01` to this group:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi hosts.yml
---
network_equipment:
  children:
    arista_ceos:
arista_ceos:
  hosts:
    ... omitted output ...
  children: (1)
    spines:
spines: (2)
  hosts:
    ceos-01: (3)
```

The numbered references from the output are explained here:

- (1) Create a children group under the `arista_ceos` parent with a child group called `spines`. Adding additional child groupings allows for more flexibility while creating configurations based on the role of the device. This can be Ansible plays targeting just the spine switches, and/or provides the ability within configuration templates to add elements that are only required for spine devices.
 - (2) Define the `spines` group in the inventory. This name is what you will specify under the children grouping.
 - (3) Assign the `hosts` value that will be part of this group.
3. Validate that `ceos-01` is assigned to the `spines` group:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-inventory -i
hosts.yml --graph spine
@spines:
|--ceos-01
```


4. Finally, create another new group, called `leafs`, and assign `ceos-02` and `ceos-03` to this group:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi hosts.yml
... omitted output ...
  children:
    spines:
    leafs:
  leafs:
    hosts:
      ceos-02:
      ceos-03:
```

5. Similarly, validate that the `leafs` group has the two new nodes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-
inventory -i hosts.yml --graph leafs
@leafs:
|--ceos-02
|--ceos-03
```

Now that the inventory groups have been created and nodes have been assigned, the next section will set up the remaining configuration and connection options.

Connecting to Arista devices

In this recipe, we will outline how to connect to Arista devices from Ansible via **Secure Shell (SSH)** in order to start managing the devices from Ansible. SSH is the traditional way to connect and is a comfortable method for network engineers who are starting their automation journey:

1. Create the `ansible.cfg` file and set some default values:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi ansible.cfg
[defaults]
inventory=hosts.yml
gathering = explicit
host_key_checking = False
```

2. Create the default connection options (`network_cli`), which are going to be stored in `group_vars`, just like they were in *Chapter 3*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# mkdir group_vars &&
cd group_vars/ && vi arista_ceos.yml
---
ansible_network_os: "arista.eos.eos"
ansible_connection: "ansible.netcommon.network_cli"
ansible_user: "ansible"
ansible_password: "ansible123"
```



Note

`ansible.netcommon.network_cli` is used for SSH connectivity. In the next section, you will use the eAPI as an alternative.

In this recipe, you leverage the Ansible inventory file defined in the previous recipe. *Network reachability* (i.e., *IP and port accessibility*) between the Ansible control machine and all the devices in the network must be implemented, and finally, there is some bootstrap configuration that must be carried out on the device before Ansible can manage it (i.e., *management access*).

3. Log in to the Arista devices. The Arista nodes are accessible from the lab VM using the `ssh` command. The default credentials for cEOS are `admin/admin`. Enter into CLI configuration mode with `configure terminal`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ssh admin@ceos-01
Warning: Permanently added 'ceos-01' (ED25519) to the list of known
hosts.
admin@ceos-01's password: admin
Last login: Tue Jan  7 04:23:07 2025 from 198.51.100.1
ceos-01# configure terminal
```

- Next, create a user named `ansible`, and set the authorization according to the username and password defined in `group_vars`. An example for `ceos-01` is provided as follows, but it must be done for all Arista devices:

```
ceos-01#configure
ceos-01(config)#username ansible privilege
15 role network-admin secret sha512
$6$7t1S6EfBw1CH6SAK$MIZFvd7I1FtfeKXuTW1sXYWo8sN0E4n0DyRYpFSW/
cxR00hP2t0VwAhEs2i1pF.p/ncX0z9HYUQxRs7htener1
ceos-01(config)#aaa authorization exec default local
```

- Still within the CLI session, enable SSH management-level configuration:

```
ceos-01(config)#management ssh
ceos-01(config-mgmt-ssh)# idle-timeout 300
ceos-01(config-mgmt-ssh)# authentication mode password
ceos-01(config-mgmt-ssh)# login timeout 300
```

- Finally, validate that the MGMT **Virtual Routing Function (VRF)** was already configured on the `Management0` interface:

```
ceos-01#show run section interface management
interface Management0
    vrf MGMT (1)
    ip address 198.51.100.11/24
```

The numbered reference from the output is explained here:

(1) The MGMT VRF was configured by default during the Containerlab deployment by setting `CLAB_MGMT_VRF: MGMT` in the kind definition.

- Now that the basic prerequisites are configured, the connection options are set in `group_vars`, and the username and password are set in the `ansible.cfg` file, in order to test that SSH is working and that the credentials are working, you will execute the Ansible ping module via the CLI against all three Arista devices that are part of the same `arista_ceos` group:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible -m ansible.
builtin.ping arista_ceos
ceos-01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
```

```
}  
... omitted output ...
```

The Ansible ping module does much more than the typical ping network utility. In this case, it's testing the following:

- Reachability to the nodes' IP management address (taken from the `ansible_host` Ansible variable) on port TCP/22 (SSH). This is not an ICMP ping.
- Authentication is successful using the variables/credentials stored in `arista_ceos.yml`, which become available in the play.



Note

By implementing a **Zero-Touch Provisioning (ZTP)** solution, the preceding manual configuration steps can be omitted in favor of using ZTP configuration bootstrapping.

Now that you've updated the inventory file to add a few other groups and validated that the default SSH connectivity is working for our `arista_ceos` devices, the next section will introduce a more robust connectivity option for Arista devices that use HTTP(S) instead of SSH. You will use SSH in order to automate the alternate connectivity option.

Migrating connections from SSH to HTTP(S) by utilizing the Arista eAPI

In this recipe, we will outline how to enable the eAPI on Arista devices. The **eAPI** is a **Representational State Transfer (REST)** API (transport) using JSON-RPC. It provides a programmatic way to define actions in a remote system on Arista devices, which abstracts the CLI by providing a consistent API to manage them. This task is critical since we will use the eAPI in all future recipes to manage the Arista devices, but the same recipes could also be run over SSH.

Before you can use the eAPI for future configuration changes, you must first configure the initial eAPI setup via a specific Ansible playbook. This will be accomplished by using the `network_cli` SSH connectivity option, which was explained in the previous section. However, this could also be done manually, the same as what you did to enable the basic SSH access before:

1. Create an `all.yml` file inside the `group_vars` folder, with the following management VRF data, which will be used to enable the eAPI under the correct management VRF. This

means that all the network devices, Arista or others, will use the MGMT VRF for management purposes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi group_vars/all.
yaml
---
global:
  mgmt_vrf: MGMT
```

Create a new playbook called `pb_eos_enable_eapi.yml`. In this playbook, you are temporarily overriding the connection type to `network_cli` for just this playbook, because the eAPI isn't configured yet. After this playbook executes, all subsequent plays will utilize the new `httpapi` default from our `group_vars` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_eos_enable_
eapi.yml
---
- name: "Enable eAPI on Arista Switches"
  hosts: arista_eos
  vars:
    ansible_connection: ansible.netcommon.network_cli (1)
  tasks:
    - name: "Enable eAPI"
      arista.eos.eos_eapi: (2)
        https_port: 443
        https: yes
        state: started
        vrf: "{{ global['mgmt_vrf'] }}" (3)
```

The numbered references from the output are explained here:

- (1) Force the use of SSH (`network_cli`) for just this task. This will allow this playbook to continue to function even once you've changed to the eAPI, by utilizing Ansible's built-in precedence ordering.
- (2) The `eos_eapi` module is a dedicated module for managing eAPI configuration.
- (3) Enables additional network isolation by implementing the eAPI under the MGMT VRF.

2. Next, you will update the connection options to use the eAPI, changing `ansible_connection` and introducing two new *eAPI-specific variables*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi group_vars/
arista_ceos.yml
---
ansible_network_os: "arista.eos.eos"
ansible_connection: "ansible.netcommon.httpapi" (1)
ansible_httpapi_use_ssl: "yes" (2)
ansible_httpapi_validate_certs: "no" (3)
ansible_user: "ansible"
ansible_password: "ansible123"
ansible_httpapi_ciphers: AES256-SHA:DHE-RSA-AES256-SHA:AES128-
SHA:DHE-RSA-AES128-SHA (4)
```

The numbered references from the output are explained here:

- (1) Moves the connection from `network_cli` to `httpapi`. This is because the eAPI is a RESTful API that is connected over HTTP(S).
 - (2) Set to yes to force the use of HTTPS and not HTTP.
 - (3) Set to no to disable certificate validations (since we are using the default certificate on the Arista devices, which is not signed by a trusted **Certificate Authority (CA)**).
 - (4) Some versions of EOS default cipher suites can cause an [SSL: SSLV3_ALERT_HANDSHAKE_FAILURE] sslv3 alert handshake failure (_ssl.c:1000) error. Specify the ciphers to avoid potentially encountering this error later.
3. Finally, execute the playbook to turn on and configure the eAPI:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_eos_enable_eapi.yml
PLAY [Enable eAPI on Arista Switches] *****
***
TASK [Enable eAPI] *****
***
changed: [ceos-01]
ok: [ceos-02]
ok: [ceos-03]
... omitted output ...
```

In the execution of the playbook, you can see that enable eAPI is set to changed, indicating that the eAPI was successfully configured.

You can validate that you are using the correct connection setting and that Ansible is able to communicate with the Arista devices using the eAPI with the following command. Remember that adding more `-v` increases the *verbosity level*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible all -m
ansible.builtin.ping -l ceos-01 -vvvv
... omitted output ...
<198.51.100.11> attempting to start connection
<198.51.100.11> using connection plugin ansible.netcommon.httpapi
<198.51.100.11> local domain socket listeners started successfully
```

Now that you have validated that Ansible can connect to the Arista devices using the HTTPAPI connection option, the rest of this chapter will use the eAPI to configure and validate operations. You will start by learning how to create and manage users and authentication options using the `arista.eos.eos_user` module.

Creating and managing users and authentication options

Managing users and authentication options is a common audit requirement. Tracking the local users and how they can authenticate is a great task that Ansible can manage. In this section, you will learn how to use the `eos_user` module in order to manage users and their authentication methods. Both password and SSH key authentication will be demonstrated.

In this recipe, you will create a new read-only user named `ansible-ro`. This user will use a standard password for authentication, and you will add public key authentication to that user using the following steps:

1. Create a new playbook called `pb_users.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_users.yml
---
- name: "Create New User with Ansible"
  hosts: "arista_ceos"
  tasks:
    - name: "Create the ansible-ro user with password auth"
      arista.eos.eos_user:
        name: "ansible-ro"
```

```

    configured_password: "ansiblero123" (1)
    role: "network-operator"
    state: "present"
- name: "Generate an OpenSSH rsa keypair with 2048 bits"
  community.crypto.openssh_keypair: (2)
    path: "~/ssh/id_rsa"
    size: 2048
  run_once: yes
- name: "Update the ansible-ro user adding public key auth"
  arista.eos.eos_user:
    name: "ansible-ro"
    sshkey: "{{ lookup('file', '~/ssh/id_rsa.pub') }}" (3)
    role: "network-operator" (4)
    state: present

```

The numbered references from the output are explained here:

- (1) This module argument takes the cleartext password to configure.
- (2) This Ansible module can generate SSH keys.
- (3) The Ansible file lookup allows a module to load a file's data and pass it in as a parameter.
- (4) "network-operator" in Arista defaults to a read-only account.

Create a new playbook called `pb_readonly_user.yml`.



Note

To keep the playbook simple, the users and arguments are not variablized and instead have been hardcoded. In a production environment, variables should be used and the password should be secured in a vault.

2. Run the playbook to configure the new user, generate the SSH key, and update the user:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_users.yml
PLAY [Create New Users with Ansible] *****
***
TASK [Create the ansible-ro user with password auth]
*****
changed: [ceos-01]

```



```

changed: [ceos-02]
changed: [ceos-03]
TASK [Generate an OpenSSH rsa keypair with 2048 bits]
*****

changed: [ceos-01] (1)
TASK [Update the ansible-ro user adding public key auth]
*****

changed: [ceos-01]
changed: [ceos-02]
changed: [ceos-03]
... omitted output ...

```

The numbered reference from the output is explained here:

(1) Only runs once as the Ansible control node SSH key only needs to be generated once.

Now that you've configured two new users, `ansible` (which was done manually) and `ansible-ro` (using the preceding steps), it's important to know how you can use these from an Ansible perspective. To use the `ansible-ro` user, simply update `group_vars/arista_ceos.yml` with the new values:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi group_vars/
arista_ceos.yml
---
... omitted output ...
ansible_user: "ansible-ro"
ansible_password: "ansiblero123"

```

It's important to notice that these two variables, `ansible_user` and `ansible_password`, are magic variables within Ansible. They're automatically used for connection credentials when specified.



Note

Visit https://docs.ansible.com/ansible/latest/inventory_guide/connection_details.html#setting-up-ssh-keys for the current best practices for setting up SSH keys.

With our multiple users (`ansible` and `ansible-ro`) configured and authenticated, let's move on to configuring some generic system options.

Configuring generic system options on Arista devices

In this recipe, you will learn how to configure some basic system options, such as **hostname** and **Domain Name System (DNS)** servers. You will understand how to set up all these system-level parameters using various Ansible modules and learn the different ways to manage these parameters:

1. The first update you need to make is to the `groups_vars/all.yml` file. Here, the global defaults need to be specified so that you can access them programmatically, all while managing them from a single spot within the project:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi group_vars/all.
yml
---
global:
  mgmt_vrf: MGMT
  dns:
    - 8.8.8.8
    - 8.8.4.4
```

In this case, you are adding a new dictionary key named `dns`, which is nested under the `global` root key. This is a simple list of IP addresses to define the DNS servers to be used by the network devices. Here, you will use the two common public Google DNS servers for testing.

2. Create a new playbook, `pb_basic_config.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_basic_config.
yml
---
- name: "Configure Hostname and DNS"
  hosts: arista_ceos
  tasks:
    - name: "Configure Basic Configuration on Arista Fabric"
      arista.eos.eos_system:
        hostname: "{{ inventory_hostname }}" (1)
        name_servers: "{{ global['dns'] }}" (2)
        state: present
```

The numbered references from the output are explained here:

(1) Here, you are setting the hostname of the Arista device to equal the hostname from the inventory, using the magic variable exposed by Ansible (`inventory_hostname`).

(2) Use the variable defined in the `group_vars/all.yml` file to create the DNS name servers.

3. Execute the playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_basic_config.yml
PLAY [Configure Hostname and DNS] *****
***
TASK [Configure Basic Configuration on Arista Fabric]
*****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...
```

4. If you log in to one of the devices, you can see that the configuration changes were successful:

```
ceos-01#show run | section name-server
ip name-server vrf default 8.8.4.4
ip name-server vrf default 8.8.8.8
```

This section demonstrates the power of a declarative Ansible module, in this case, `eos_system`. Declarative modules allow an operator to define the end state they're looking to achieve. Most network OS vendors have certain parameters of the configuration that have dedicated modules; however, they might not cover all the parameters that you need to set up a network device (e.g., lack of mature implementation). To have more control and flexibility to configure the system-level parameters, you can use Jinja2 templates along with the `template` Ansible module to generate the specific system-level configuration needed for your deployment.

In the next section, you will configure the interfaces by starting your journey in creating a reusable role utilizing the aforementioned modules to generate configurations.

Configuring Arista interfaces using Ansible

In this recipe, you will configure different interface parameters on the Arista devices, such as the *interface description* and *IP address information*. You will *not* use the declarative `eos_interface` module, because you will be building out a configuration that does not have declarative modules for all the different configuration elements. You will start by building a new Ansible role that you will further build upon over the next few chapters (because of this, you will create it outside of the `ch04` directory). This is to promote the idea of code reuse. This new role will be very similar to the role you created in *Chapter 3*, but much more flexible.



Note

The code for the Ansible role will be stored in https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/roles.

Let's get started by creating the role that we will be using throughout Part 2 of the book:

1. Let's start by creating a new directory in the `chapters` folder called `roles`:

```
root@cookbook-lab:~/cookbook-lab/chapters# mkdir roles && cd roles
```

2. Next, use the `ansible-galaxy` utility function to generate the role's boilerplate code (this is a great way to get started!):

```
root@cookbook-lab:~/cookbook-lab/chapters/roles# ansible-galaxy init
config_generation
- Role config_generation was created successfully
```

Now that we've initialized the role and understand some of the crucial elements of creating a role, let's write the code to complete our role:

3. Start by defining a simple variable in `defaults`. Here, you are simply assigning a directory name for the configurations you will generate:

```
root@cookbook-lab:~/cookbook-lab/chapters/roles# vi config_
generation/defaults/main.yml
---
config_template_root_dir: "templates/{{ ansible_network_os |
split('.') | last }}" (1)
```

```
config_output_root_dir: "{{ global_output_root_dir |
default('output/config') }}" (2)
```

The numbered references from the output are explained here:

- (1) This chained filter will take `ansible_network_os` (`arista.eos.eos`) and split it on dot (`.`), which renders into a list `["arista", "eos", "eos"]`. Finally, it will pass it through the last filter to grab the last index of the list, resulting in `eos`.
 - (2) `global_output_root_dir` is an *optional* variable that can be set to overwrite the default root directory, which is set to `output/config`. It is optional because the `| default` filter executes a variable lookup and sets a default value if the lookup isn't successful.
4. Now, you will create the reusable tasks that you want this role to expose, using the variable from the preceding defaults, which were defined in `/defaults/main.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters# vim roles/config_
generation/tasks/main.yml
---
- name: "CREATE OUTPUT CONFIGS DIRECTORY"
  ansible.builtin.file: (1)
    path: "{{ config_output_root_dir }}"
    state: "directory"
  run_once: yes
- name: "GENERATE CONFIGURATIONS FROM JINJA2 TEMPLATE"
  ansible.builtin.template: (2)
    src: "{{ config_template_root_dir }}.j2"
    dest: "{{ config_output_root_dir }}/{{ inventory_hostname
    }}.cfg"
  lstrip_blocks: true (3)
```

The numbered references from the output are explained here:

- (1) An Ansible module that handles file and directory tasks. In this case, the module is making sure the `config_output_root_dir` directory exists.
- (2) An Ansible module capable of taking in a source Jinja2 file and rendering the results to the destination file defined.

(3) Determine when leading spaces and tabs should be stripped. In the Ansible template module, this defaults to `false`. Set it to `true` to avoid having to track this manually using the `-%` formatting. Jinja2 manages whitespace within a template with a few defaults. A single trailing newline is stripped if present; other whitespace (spaces, tabs, newlines, etc.) is returned unchanged by modifying config variables such as `lstrip_blocks`. The defaults can be changed.



Note

The Jinja2 whitespace control is described in

<https://jinja.palletsprojects.com/en/stable/templates/#whitespace-control>.

With the role created and the initial defaults and tasks written, the next step is to create the data model for the variables required to generate the interface configurations.



Note

With the generation of the role completed, take some time to research `ansible-lint` (<https://ansible.readthedocs.io/projects/lint/#about-ansible-lint>). `ansible-lint` can validate roles and other Ansible constructs for best practices, errors, and style.

To set the stage, the following diagram shows the interface. Then, it is followed by a table with the IP address schema that we will be building out.

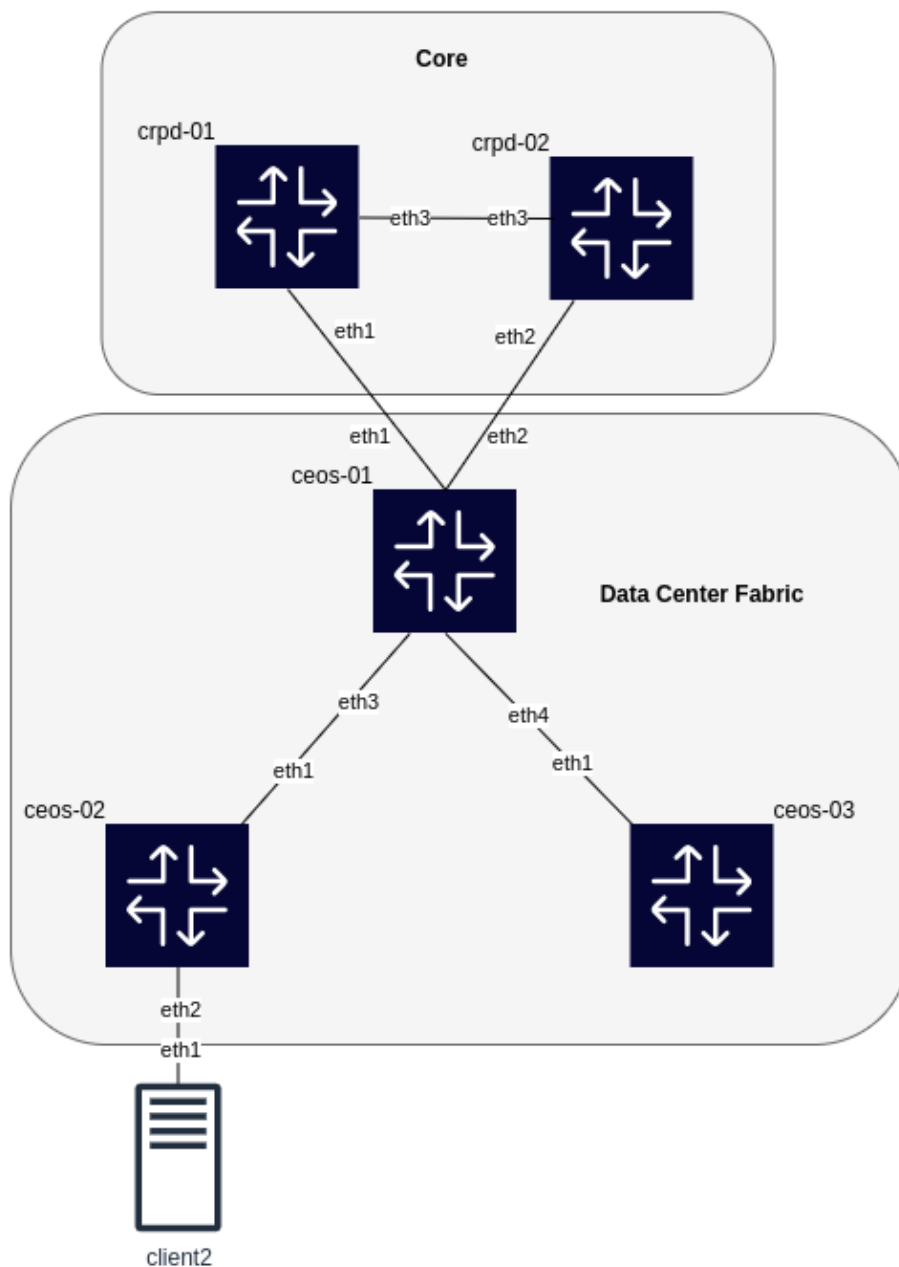


Figure 4.2 – Arista fabric

With the topology defined, in the following table, the interfaces in the diagram have been linked with the IP addresses that will be configured.

Device	Role	Interface	IP Address
ceos-01	Spine switch	Management0	198.51.100.11/24
ceos-01	Spine switch	Ethernet1	172.31.1.4/31
ceos-01	Spine switch	Ethernet2	172.31.1.6/31
ceos-01	Spine switch	Ethernet3	172.31.1.0/31
ceos-01	Spine switch	Ethernet4	172.31.1.2/31
ceos-02	Leaf switch	Management0	198.51.100.12/24
ceos-02	Leaf switch	Ethernet1	172.31.1.1/31
ceos-02	Leaf switch	Ethernet2	—
ceos-02	Leaf switch	Vlan10 (SVI)	192.168.20.254/24
ceos-03	Leaf switch	Management0	198.51.100.13/24
ceos-03	Leaf switch	Ethernet1	172.31.1.3/31
ceos-03	Leaf switch	Vlan10 (SVI)	192.168.20.254/24

Table 4.1 – Interface IP address scheme

**Note**

The management IPs are defined by our Containerlab topology.

Each host will have a dedicated vars file that is stored in `host_vars`, using the expected `host_vars/{{ inventory_hostname }}` pattern, which will auto-load the variables into the execution.

1. Create `host_vars/ceos-01.yml` (now, again, in the `ch04` folder) and create the following data model:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi host_vars/ceos-01.yml
---
interfaces:
  Ethernet1:
    ip_address: "172.31.1.4/31"
    description: "SPINE-TO-CORE-CRPD-01-ETHERNET2"
  Ethernet2:
```



```

    ip_address: "172.31.1.6/31"
    description: "SPINE-TO-CORE-CRPD-02-ETHERNET2"
Ethernet3:
    ip_address: "172.31.1.0/31"
    description: "SPINE-TO-LEAF-CEOS-02-ETHERNET1"
Ethernet4:
    ip_address: "172.31.1.2/31"
    description: "SPINE-TO-LEAF-CEOS-03-ETHERNET1"
Management0:
    ip_address: "198.51.100.11/24"
    description: "clab mgmt network"

```

To keep the data model simple, we're using Classless Inter-Domain Routing (CIDR) slash notation and a simple description.

2. Now that the data model is defined for the interfaces and IP addresses, you must write a template to use this data and transform the variables into Arista configuration:

```

root@cookbook-lab:~/cookbook-lab/chapters/
ch04# cd templates && vi interfaces.j2
{% for name, int_details in interfaces.items() %} (1)
interface {{ name }}
    description {{ int_details['description'] }} (2)
    {% if int_details['ip_address'] is defined %}
    {% if 'Vlan' in name %}
    ip address virtual {{ int_details['ip_address'] }} (3)
    {% else %}
    {% if int_details['ip_address'] is defined %}
    {% if 'Management' not in name %} (4)
    no switchport
    {% endif %}
    ip address {{ int_details['ip_address'] }}
    {% endif %}
    {% if int_details['vlan'] is defined %}
    switchport access vlan {{ int_details['vlan'] }}
    {% endif %}
    {% endif %}
!
{% endfor %}

```

The numbered references from the output are explained here:

- (1) Loop through the dictionary you defined previously in `host_vars`. Use the `(.items())` method, which allows access to the key-value pairs of the dictionary.
 - (2) The value of `int_details` is an inner dictionary; therefore, our subsequent lookups are dictionary key lookups to get the value.
 - (3) VLAN interfaces (SVIs) will use virtual IP addresses that are shared between both leaves, which serve as the VXLAN default gateway.
 - (4) A management port is not capable of being a switchport; here, you do a conditional check to ensure that Management is *not* included in the current name variable.
3. This step begs the question, how do you call multiple Jinja2 templates from a single file? The Ansible role you created takes in a source file of `eos.j2`, but you just created a new file called `interfaces.j2`, so how do you use a single entry point Jinja2 file and inject other Jinja2 files? The answer is by using the Jinja2 `include` statement. Let's update `eos.j2` in order to include the `interfaces.j2` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/eos.j2
! Comment line {{ inventory_hostname }} config
{% include "./interfaces.j2" %} (1)
```

The numbered reference from the output is explained here:

- (1) The Jinja2 `include` statement accepts absolute or relative file path types. In this case, you're saying to look in the current directory and include the file named `interfaces.j2`.
4. Create the playbook named `pb_advanced_configs.yml`, which will call our role:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_advanced_configs.yml
---
- name: "BUILD CONFIGS FOR ARISTA"
  hosts: "arista_ceos"
  connection: "local"
  roles:
    - "config_generation" (1)
```

The numbered reference from the output is explained here:

- (1) Execute the `config_generation` role you created previously. This is how roles are reused.
5. Finally, execute the playbook. The role will run and save the interface configuration output after it has been rendered by Jinja2:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
changed: [ceos-01]
]TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [ceos-01]
changed: [ceos-02]
changed: [ceos-03]
```

6. View the file located at `output/config/ceos-01.cfg` to view the rendered template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# cat output/config/
ceos-01.cfg
! Comment line ceos-01 config
interface Ethernet1
    description SPINE-TO-CORE-CRPD-01-ETHERNET2
    no switchport
    ip address 172.31.1.4/31
!
... omitted output ...
interface Management0
    description clab mgmt network
    ip address 198.51.100.11/24
!
```

Now that we've generated the interface configuration for our data center devices, we're going to continue to build on top of these fundamentals to build out the other configuration snippets to finalize the full fabric configuration.

In the next section, you will generate the rest of the EVPN/VXLAN configuration, which will prepare you for deploying that configuration in a later section.

Advanced fabric configuration for Arista Data Center

Our data center topology has one spine switch and two leaf switches. An **EVPN/VXLAN fabric** consists of an underlay network that dynamically provides the routing for the VXLAN (VTEPS) to establish and an overlay network that provides the routing information for the host devices to be able to communicate. Simply put, VXLAN allows layer 2 tunnels over a layer 3 network. In this small data center fabric example, you will use BGP as our routing protocol for both the underlay and overlay.



Note

For more information on VXLAN configurations on Arista, see <https://www.arista.com/en/um-eos/eos-vxlan-configuration>.

Generating the underlay configuration

The underlay in our topology is using eBGP (i.e., different autonomous systems). The following table outlines the BGP **Autonomous System Number (ASN)** for each of the devices in our fabric.

Node	Role	BGP ASN
ceos-01	Spine switch	65100
ceos-02	Leaf switch	65001
ceos-03	Leaf switch	65002

Table 4.2 – BGP ASN scheme

Next, we will begin translating our configuration data into variables that will be used in our templates:

1. Update the `host_vars` files for each device to add the `Loopback0` interface, which is used as the `router-id` value in our topology:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi host_vars/ceos-01.yml
---
interfaces:
... omitted output ...

Loopback0:
  ip_address: "10.100.1.254/32"
```

```

    description: "LOOPBACK FOR ROUTER ID"
  Management0:
... omitted output ...

```

2. Add new variable data in `host_vars` that will help you generate the necessary underlay configuration according to the design:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi host_vars/ceos-01.yml
... omitted output ...
bgp:
  asn: 65100
  peer_groups:
    underlay: (1)
    - peer: ceos-02
      ip_address: 172.31.1.1 (2)
      remote_as: 65001 (3)
    - peer: ceos-03
      ip_address: 172.31.1.3
      remote_as: 65002

```

The numbered references from the output are explained here:

- (1) This provides the first peer group for our BGP configuration. To simplify some of the conditional logic in the upcoming templates, `underlay` makes the most sense.
 - (2) For each BGP neighbor to create, you will need the corresponding neighbor IP address.
 - (3) A required element to create a BGP neighbor is also the remote ASN.
3. Create a new Jinja2 file named `underlay_policy.j2`, with the following template data. This template also includes the `prefix-list` that we will use to control BGP advertisements in our fabric:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/underlay_policy.j2
!
route-map connected permit 10
  match ip address prefix-list loopback
{% if inventory_hostname in groups['leafs'] %}
route-map connected permit 20

```

```
    match ip address prefix-list svls
{% endif %}
!
{% if inventory_hostname in groups['spines'] %} (1)
!
ip prefix-list loopback
{% for leaf_switch in groups['leafs'] %}
    seq {{loop.index + 10 }} permit {{ hostvars[leaf_switch]
['interfaces']['Loopback0']['ip_address'] }} (2)
{% endfor %}
    seq 100 permit {{ interfaces['Loopback0']['ip_address'] }}
!
{% else %}
!
    seq 10 permit {{ interfaces['Loopback0']['ip_address'] }}
!
{% endif %}
... omitted output ...
```

The numbered references from the output are explained here:

- (1) Ansible exposes a magic variable, `groups`, that provides a simple mechanism to be able to loop through hosts that are part of a specific group.
- (2) `loop.index` is a built-in variable that counts the iteration number during the loop. This provides a great way to increment a sequence number.



Note

An Access Control List (ACL) sequence number is an entry in an ACL with a defined numeric value that represents the order in which it should be evaluated. It also provides a mechanism to allow inserting statements after the fact into the proper placement.

4. Next, create a new Jinja2 file called `bgp.j2`. This template will be the first iteration of the BGP configuration section. It will be updated to add overlay components in the next section:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/bgp.j2
!
router bgp {{ bgp['asn'] }} (1)
  router-id {{ interfaces['Loopback0']['ip_address'] | split('/') |
    first }} (2)
  maximum-paths 2
  bgp bestpath tie-break router-id
  {% for group_name in bgp['peer_groups'] %} (3)
    neighbor {{ group_name | upper }} peer group
    neighbor {{ group_name | upper }} description "Peer Group for {{
    group_name | title }}"
    neighbor {{ group_name | upper }} graceful-restart-helper
    neighbor {{ group_name | upper }} send-community extended
    neighbor {{ group_name | upper }} maximum-routes 100000
    warning-only
  {% endfor %}
  {% for group_name, peer_group_peers in bgp['peer_groups'].
items() %}
    {% for bgp_peer in peer_group_peers %} (4)
      Neighbor {{ bgp_peer['ip_address'] }} peer group {{ group_name |
      upper }}
      neighbor {{ bgp_peer['ip_address'] }} remote-as {{ bgp_
peer['remote_as'] }}
    {% endfor %}
  {% endfor %}
  redistribute connected route-map connected
!
  {% if bgp['peer_groups']['underlay'] is defined %} (5)
    address-family ipv4
    neighbor UNDERLAY activate
    neighbor UNDERLAY route-map connected out
  {% endif %}

```

The numbered references from the output are explained here:

- (1) From the current node's `host_vars`, interpolate with the BGP ASN.
 - (2) `router-id` will be using the `Loopback0` interface IP address.
 - (3) Loop through the `peer_groups` defined in `host_vars`. Each peer group will create the peer group configuration snippets.
 - (4) Now that the peer group has been created with the required configuration data, each peer has to be created.
 - (5) Finally, under `ipv4 address-family`, activate the peer group, and add the route-map created in the previous template to implement the security policy.
5. Before running the playbook to generate the configuration, we will update our `eos.j2` entry point template; here, include the two new files you just created:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/eos.j2
! Comment line {{ inventory_hostname }} config

{% include "./interfaces.j2" %}
{% include "./underlay_policy.j2" %}
{% include "./bgp.j2" %}
```

6. Executing the playbook will generate the full configuration that we've templated so far:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****

TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...
```


7. By looking at ceos-01 generated configuration, you can see how the variable data and the templates created our underlay network configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# cat output/config/
ceos-01.cfg
interface Loopback0
  description LOOPBACK FOR ROUTER ID
  no switchport
  ip address 10.100.1.254/32
... omitted output ...
route-map connected permit 10
  match ip address prefix-list loopback
!
ip prefix-list loopback
... omitted output ...
  seq 100 permit 10.100.1.254/32
!
router bgp 65100
  router-id 10.100.1.254
  maximum-paths 2
  bgp bestpath tie-break router-id
  neighbor UNDERLAY peer group
... omitted output ...
  neighbor 172.31.1.1 peer group UNDERLAY
  neighbor 172.31.1.1 remote-as 65001
... omitted output ...
  redistribute connected route-map connected
!
address-family ipv4
  neighbor UNDERLAY activate
  neighbor UNDERLAY route-map connected out
!
```

Let's continue updating our templates and variable data to create our overlay peer group.

Generating the overlay configuration

In this recipe, you will generate the configuration for the **overlay BGP EVPN** as the control plane for VXLAN tunnels across our leaf spine DC fabric in our sample topology, using Ansible:

1. Update the `bgp.j2` template to extend its flexibility to conditionally check for underlay versus overlay peer group naming. Update the template as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/bgp.j2
router bgp {{ bgp['asn'] }}
  router-id {{ interfaces['Loopback0']['ip_address'] | split('/') |
  first }}
  maximum-paths 2
  bgp bestpath tie-break router-id
  {% for group_name in bgp['peer_groups'] %}
  neighbor {{ group_name | upper }} peer group
  neighbor {{ group_name | upper }} description "Peer Group for {{
  group_name | title }}"
  neighbor {{ group_name | upper }} graceful-restart-helper
  neighbor {{ group_name | upper }} send-community extended
  neighbor {{ group_name | upper }} maximum-routes 100000
  warning-only
  {% if 'overlay' in group_name %} (1)
  neighbor {{ group_name | upper }} update-source Loopback0
  neighbor {{ group_name | upper }} ebgp-multihop 2
  {% endif %}
  {% endfor %}
  {% for group_name, peer_group_peers in bgp['peer_groups'].items()
  %}
  {% for bgp_peer in peer_group_peers %}
  neighbor {{ bgp_peer['ip_address'] }} peer group {{ group_name |
  upper }}
  neighbor {{ bgp_peer['ip_address'] }} remote-as {{
  bgp_peer['remote_as'] }}
  {% endfor %}
  {% endfor %}
  redistribute connected route-map connected
  !
  {% if bgp['peer_groups']['overlay'] is defined %} (2)
```

```

address-family evpn
    neighbor OVERLAY activate
!
{% endif %}
{% if bgp['peer_groups']['underlay'] is defined %}
address-family ipv4
    neighbor UNDERLAY activate
    neighbor UNDERLAY route-map connected out
{% endif %}

```

The numbered references from the output are explained here:

- (1) Add the two extra configuration parameters to our peer group when the peer group is Overlay.
 - (2) Activate the overlay peer group, which is done under the EVPN address family.
2. Rerun the playbook, which will update the generated configuration file and change only what is needed:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...

```

3. Once again, looking at the ceos-01 configuration, notice the changes to the BGP configuration, which adds all the overlay components:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# cat output/config/
ceos-01.cfg
router bgp 65100
    router-id 10.100.1.254
    maximum-paths 2
    bgp bestpath tie-break router-id

```

```

.. omitted output ...
neighbor OVERLAY peer group
neighbor OVERLAY update-source Loopback0
neighbor OVERLAY description "Peer Group for Overlay"
neighbor OVERLAY ebgp-multihop 2
neighbor OVERLAY graceful-restart-helper
neighbor OVERLAY send-community extended
neighbor OVERLAY maximum-routes 100000 warning-only
... omitted output ...
neighbor 10.100.1.1 peer group OVERLAY
neighbor 10.100.1.1 remote-as 65001
neighbor 10.100.1.2 peer group OVERLAY
neighbor 10.100.1.2 remote-as 65002
redistribute connected route-map connected
!
address-family evpn
    neighbor OVERLAY activate
!
... omitted output...

```

Once the configurations for both the underlay and overlay are completed, there are just a few outstanding configurations that are needed for hosts and VXLAN routing. The next section will add these components.

Generating the remaining VXLAN configuration

In this final recipe, you will add the remaining configurations to the templates. The remaining configuration components include the VXLAN interface and fabric-specific BGP configuration:

1. Create a new template called `vxlan_interface.j2`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/vxlan_
interface.j2
interface Vxlan1 (1)
    vxlan source-interface Loopback0 (2)
    vxlan udp-port 4789 (3)
    {% for leaf_switch in groups['leaves'] %}
    {%     for int_data in hostvars[leaf_switch]['interfaces'].values()
    {% }
    {%     if int_data['vlan'] is defined %} (4)

```

```

vxlan vlan {{ int_data['vlan'] }} vni 10{{ int_data['vlan'] }} (5)
{%     endif %}
{%   endfor %}
{% endfor %}

```

The numbered references from the output are explained here:

- (1) Vxlan1 is the default interface for Arista; therefore, we hardcode it.
 - (2) Using the Loopback0 source interface, which is our VTEP, this is the IP address that traffic will be sourced from.
 - (3) The default UDP port, 4789, is used for encapsulating layer 2 frames within a UDP packet.
 - (4) Use the values of our interfaces dictionary. If the vlan key exists, assume this is a switch port and generate the required VXLAN/EVPN configurations.
 - (5) Map our VLAN IDs to VNI id of 1000 + VLAN ID (e.g VLAN 10 = VNI 1010).
 - A VNI is a VXLAN network identifier, which is a virtual network interface that represents a subnet.
2. Update the entry point eos.j2 template by adding an include statement to add the VXLAN interface only when the device is a leaf switch:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/eos.j2
... omitted output...
{% if inventory_hostname in groups['leafs'] %} (1)
{% include "./vxlan_interface.j2" %}
{% endif %}
... omitted output...

```

The numbered reference from the output is explained here:

- (1) Since **VXLAN Tunnel Endpoints (VTEPs)** are only configured on leaf switches, you will only include this template when the host is a leaf switch.
3. Next, create bgp_vlan.j2 for the EVPN BGP configuration. This snippet will be included in the correct section of the BGP config within the bgp.j2 template. Notice how include allows flexible composition of the templates:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/bgp_
vlan.j2

```

```

    {% for leaf_switch in groups['leafs'] %}
        {% for int_data in hostvars[leaf_switch]['interfaces']
            .values() %}
            {% if int_data['vlan'] is defined %} (1)
            vlan {{ int_data['vlan'] }}
            rd {{ hostvars[leaf_switch]['interfaces']['Loopback0']['ip_
            address'] | split('/') | first }}:10{{ int_data['vlan'] }}
            route-target both 10{{ int_data['vlan'] }}:10{{ int_data['vlan'] }}

        learned
            {% endif %}
        {% endfor %}
    {% endfor %}

```

The numbered reference from the output is explained here:

(1) This template will look through all the interfaces and determine whether the `vlan` key is defined. If so, it will create the EVPN VLAN BGP configuration and add the VLAN-to-VNI mapping.

4. Update `bgp.j2` to add the BGP VLAN configuration. This runs only for leaf switches:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/bgp.j2
... omitted output...
    redistribute connected route-map loopback
    !
    {% if inventory_hostname in groups['leafs'] %}
    {% include './bgp_vlan.j2' %}
    {% endif %}
    !
    {% if bgp['peer_groups']['overlay'] is defined %}
... omitted output...

```

5. Let's execute the playbook again to regenerate the configuration based on our new templates:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****

```

```
ok: [ceos-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...
```

6. Finally, validate that one of the leaf (ceos-02) configurations is fully accurate:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# cat output/config/
ceos-02.cfg
! Comment line ceos-02 config
... omitted output ...
interface Vxlan1 (1)
    vxlan source-interface Loopback0
    vxlan udp-port 4789
    vxlan vlan 10 vni 1010
router bgp 65001
    router-id 10.100.1.1
    ... omitted output ...
    !
    vlan 10 (2)
        rd 10.100.1.1:1010
        route-target both 1010:1010
        vxlan vlan 10 vni 1010 (3)
    !
    ... omitted output ...
```

The numbered references from the output are explained here:

- (1) The leaf switch properly generated the Vxlan1 interface.
- (2) The leaf switch generated the subcommands for VLAN under the BGP routing section.
- (3) VLAN ID 10 has been properly assigned to VNI ID 1010.

At this point, our fabric devices have all the advanced configuration snippets generated and they have been validated by scanning through the rendered files. The last piece of configuration you need to generate is the BGP configuration for the peers that are facing the core, which will be configured in *Chapter 5*.

Finalizing BGP configuration to the core

In this recipe, you will generate the **BGP configuration** that is needed to establish the BGP neighborhood with our core network, which is running on Juniper devices. You will update the BGP Jinja2 templates from the last section to add the neighbors that face the Juniper core devices.

The following table outlines the BGP ASN for each of the devices we're configuring in this section:

Node	Role	BGP ASN
ceos-01	Spine switch	65100
crpd-01	Core router	65400
crpd-02	Core router	65400

Table 4.3 – Core BGP ASN scheme

Let's start by updating our Ansible variable data with the core BGP details:

1. Update the variable data to add a new **peer group** and provide the **neighbor details**:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi host_vars/ceos-01.yml
... omitted output ...
bgp:
  asn: 65100
  peer_groups:
    ... omitted output ...
core: (1)
  - peer: crpd-01
    ip_address: 172.31.1.5
    remote_as: 65400
  - peer: crpd-02
    ip_address: 172.31.1.7
    remote_as: 65400
```

The numbered reference from the output is explained here:

- (1) Add another peer group named **core** to define neighbors to both Juniper core routers.

2. Update `bgp.j2` by activating the new `peer_group` under the IPv4 address family. Note that there are no updates to the rest of the template as our existing loop will create the peer group that is needed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi templates/bgp.j2
router bgp {{ bgp['asn'] }}
... omitted output ...
address-family ipv4
{% if bgp['peer_groups']['core'] is defined %} (1)
neighbor CORE activate
{% endif %}
{% if bgp['peer_groups']['underlay'] is defined %}
neighbor UNDERLAY activate
neighbor UNDERLAY route-map loopback out
{% endif %}
```

The numbered reference from the output is explained here:

- (1) If the core peer group is defined in `host_vars`, then activate that group.
3. Now that you've updated the template to add the additional activate peer group conditional statement, you can run the playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...
```

4. Finally, validate that the spine (`ceos-01`) configuration has the `CORE` peer group and that it is being activated:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# cat output/config/
ceos-01.cfg
! Comment line ceos-01 config
```

```
... omitted output ...
router bgp 65100
  ... omitted output ...
  neighbor CORE peer group
  neighbor CORE description "Peer Group for Core"
  neighbor CORE graceful-restart-helper
  neighbor CORE send-community extended
  neighbor CORE maximum-routes 100000 warning-only

  neighbor 172.31.1.5 peer group CORE
  neighbor 172.31.1.5 remote-as 65400
  neighbor 172.31.1.7 peer group CORE
  neighbor 172.31.1.7 remote-as 65400
  !
  address-family ipv4
    neighbor CORE activate
    neighbor UNDERLAY activate
  ... omitted output ...
```



Note

The BGP peers facing the core will remain down until *Chapter 5*, where you will configure the remote side of the BGP neighbor.

With the final BGP configuration completed, the last step is to deploy the configurations to the devices.

Pushing configuration to Arista devices

In this recipe, you will push the configuration you have generated in the previous recipes to the Arista devices. You will learn how to interact with the Arista configuration using the suitable Ansible module, in order to correctly provision the devices as per the intended network fabric design:

1. Update `pb_advanced_configs.yml` with a tasks section. The tasks will use the `eos_config` module. This module accepts a configuration file and merges the file's contents with the running configuration. That is exactly what we need!

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_advanced_configs.yml
```

```

---
- name: "BUILD CONFIGS FOR ARISTA"
  hosts: "arista_ceos"
  connection: "local"
  roles:
    - "config_generation"
  tasks: (1)
    - name: "DEPLOY ADVANCED CONFIG"
      arista.eos.eos_config: (2)
        src: "output/config/{ inventory_hostname }.cfg" (3)
        save_when: changed (4)

```

The numbered references from the output are explained here:

- (1) Add the tasks keyword and add a new task to deploy the configuration files that you've generated.
 - (2) Use the eos_config module to be able to pass in a source file with configuration changes to be applied to the device.
 - (3) Point the src argument to the generated file that was built during the config_generation role.
 - (4) The default for the eos_config module is to never save. Here, you will change the default to save running-config to startup-config when a change is made to persist the changes.
2. Run the playbook to run the config_generation role, and execute the new task you just created to deploy those configurations:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]TASK [config_generation : GENERATE CONFIGURATIONS FROM
JINJA2 TEMPLATE] *****
*****
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
TASK [DEPLOY ADVANCED CONFIG] *****

```

```
changed: [ceos-01]
changed: [ceos-03]
changed: [ceos-02]
... omitted output ...
```

3. Validate idempotence and rerun the playbook. Notice that no changes were needed on any task that was executed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR ARISTA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
ok: [ceos-01]
ok: [ceos-03]
ok: [ceos-02]
TASK [DEPLOY ADVANCED CONFIG] *****
ok: [ceos-01]
ok: [ceos-03]
ok: [ceos-02]
... omitted output ...
```



Note

For more information on the `eos_config` module for Arista, see https://docs.ansible.com/ansible/latest/collections/arista/eos/eos_config_module.html.

That was a lot of progress you just made, starting with the creation of our `config_generation` role, moving on to the many different configuration elements to get our small data center fabric up and running, and finally, deploying that configuration. The next step is to validate that the configuration you pushed resulted in the expected end state. You will do this by retrieving device facts and running arbitrary `show` commands on the Arista devices.

Collecting Arista device facts using Ansible

In this recipe, you will learn how to retrieve the basic system facts collected by Ansible for an Arista device running the Arista EOS software. These basic system facts provide us with a basic health check regarding our Arista devices, which we can use to validate their operational state:

1. Create a new playbook, `pb_validate_facts.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_validate_facts.yml
---
- name: "Collect and Validate Arista DC Fabric Facts"
  hosts: "arista_ceos"
  tasks:
- name: "Collect Arista Device Facts"
  arista.eos.eos_facts: (1)
    gather_subset: all
- name: "Validate all DC Fabric Interface are Operational"
  ansible.builtin.assert:
    that:
      - ansible_net_interfaces[item].lineprotocol == 'up' (2)
    fail_msg: "Interface {{item}} is not Operational "
  loop: "{{ interfaces.keys() | list }}" (3)
- name: "Validate all DC Fabric Interface a have the Correct IP"
  ansible.builtin.assert:
    that:
      - ansible_net_interfaces[item].ipv4.address ==
        interfaces[item]['ip_address'].split('/')[0] (4)
    fail_msg: "Interface {{item}} has Wrong IP Address"
  when: interfaces[item]['ip_address'] is defined
  loop: "{{ interfaces.keys() | list }}"
```

The numbered references from the output are explained here:

- (1) Use the `eos_facts` module to collect the device facts for all our Arista devices. This module returned the basic facts collected by Ansible for each device in multiple variables.

- (2) The `eos_facts` module creates the `ansible_net_interfaces` variable, which holds the operational state of all the interfaces on the device in order to validate that all the fabric interfaces are operational.
 - (3) Provide the list of interfaces from our `host_vars` that need to be validated.
 - (4) We used the `ansible_net_interfaces` variable to validate that all the fabric interfaces have the correct IP address assigned to them as per our design.
2. Run the `pb_arista_facts.yml` playbook to validate the operational state of our fabric:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_validate_facts.yml
PLAY [Collect and Validate Arista DC Fabric Facts] *****
TASK [Collect Arista Device Facts] *****
ok: [ceos-01]
ok: [ceos-02]
ok: [ceos-03]TASK [Validate all DC Fabric Interface are Operational]
*****
ok: [ceos-01] => (item=Ethernet1) => {
    "ansible_loop_var": "item",
    "changed": false,
    "item": "Ethernet1",
    "msg": "All assertions passed"
}
... omitted output ...
TASK [Validate all DC Fabric Interface a have the Correct IP] ***
ok: [ceos-02] => (item=Ethernet1) => {
    "ansible_loop_var": "item",
    "changed": false,
    "item": "Ethernet1",
    "msg": "All assertions passed"
}
... omitted output ...
```

Note



For more information on the `eos_facts` module for Arista, see

https://docs.ansible.com/ansible/latest/collections/arista/eos/eos_facts_module.html.

In this recipe, you used structured data gathered from the `eos_facts` module to process some operational checks. In the final section of this chapter, you will learn how to run ad hoc CLI commands against the devices.

Running arbitrary commands on Arista devices

In this recipe, you will execute operational commands on Arista devices and use the output to validate the state of the devices:

1. Create a new playbook called `pb_get_vlans.yml` and populate it to execute the `show vlan` command on all leaf switches and store the output in a variable:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# vi pb_get_vlans.yml
---
- name: "Retrieve All VLANs from Leaf Switches"
  hosts: "leafs" (1)
  tasks:
- name: "Get All VLANs"
  arista.eos.eos_command:
    commands: show vlan | json (2)
  register: show_vlan
- name: "Validate VLANs are Present"
  ansible.builtin.assert:
    that: (interfaces[item]['vlan'] | string) in
          show_vlan.stdout[0].vlans.keys() (3)
    fail_msg: "VLAN:{{ interfaces[item]['vlan'] }} is NOT
              configured"
    success_msg: "VLAN:{{ interfaces[item]['vlan'] }} is configured"
  when: interfaces[item]['vlan'] is defined
  loop: "{{ interfaces.keys() | list }}"
  delegate_to: localhost
```

The numbered references from the output are explained here:

- (1) Delegate this play to only run on hosts in the `leafs` inventory group.
- (2) Use the `eos_command` module to run an arbitrary command. Run `show vlan | json`. Passing the native `| json` switch will return native JSON data.
- (3) Using the `assert` module, you will validate that for any interface in our design, as defined in `host_vars`, that has a VLAN attribute, the VLAN exists in the switches configuration.

2. Execute the playbook to validate that the VLANs are configured properly on the leaf switches:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch04# ansible-playbook
pb_get_vlans.yml
PLAY [Retrieve All VLANs from Leaf Switches] *****
TASK [Get All VLANs]*****
ok: [ceos-03]
ok: [ceos-02]
TASK [Validate VLANs are Present] *****
ok: [ceos-02 -> localhost] => (item=Ethernet2) => {
    "ansible_loop_var": "item",
    "changed": false,
    "item": "Ethernet2",
    "msg": "VLAN:10 is configured"
}
... omitted output ...
```

The playbook execution validates that VLAN 10 is configured on the leafs, as it is expected to be, since Ethernet2 on ceos-02 has an access VLAN of 10 assigned to the interface.

Now that you know two methods for getting data off an Arista device and using the assert module to validate the state, you're armed with a key skill to be able to validate a configuration change.

Summary

In this chapter, you have set up an Arista device to prepare for Ansible automating configurations. You also created an Ansible role that will be reused across multiple chapters to generate configurations for multiple network OSs. Next, you pushed configurations with native Arista Ansible declarative modules, as well as generating configurations using the Ansible templates module, followed by pushing those configurations onto the devices. Finally, a configuration change can't be considered complete without validating the operational status after the change, so you also learned two different ways to validate the operational state of the devices.

Next, you will tackle the core of a network scenario built with Juniper devices.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



5

Automating a Juniper Network Core Using Ansible

Now that you've configured the data center fabric using **Arista devices**, the next area to consider is the core network using Juniper cRPD devices. By the end of this chapter, the previously configured **Border Gateway Protocol (BGP)** peers to the Arista data center fabric will also be up!

As was the case with the previous chapter, the network architecture isn't intended to be production-ready. The focus is on the content of the playbooks more than the network architecture/engineering. In this chapter, you will build the **Juniper core network** for the topology that was described in *Chapter 2*. Remember that the lab is available using the instructions in *Chapter 16*. The following figure depicts the lab environment setup.

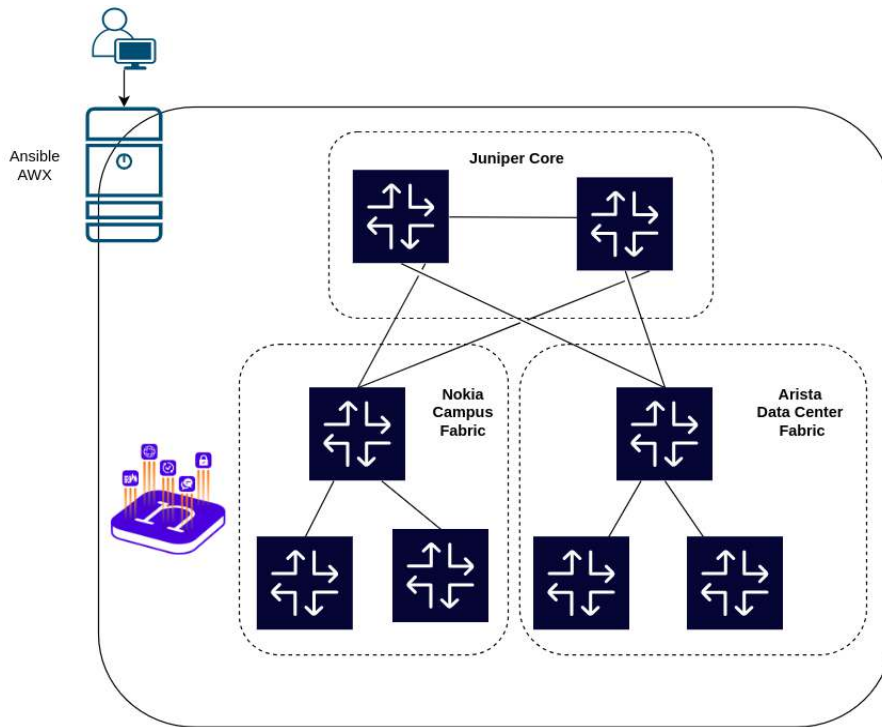


Figure 5.1 - Lab environment setup



Info

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch05.

The recipes covered in this chapter are as follows:

- Augmenting the Ansible network inventory
- Connecting to Juniper devices
- Migrating from the CLI to NETCONF
- Creating and managing users and authentication options
- Configuring generic system options on Juniper devices
- Configuring Juniper interfaces
- Advanced routing configuration using the template module

- Pushing configuration to Juniper devices
- Collecting Juniper device facts using Ansible
- Running arbitrary commands on Juniper devices
- Parsing Junos commands using `parse_xml`

The first step to automating our network core is to set up our Ansible inventory accordingly.

Augmenting the Ansible network inventory

The Juniper core has only two nodes. Let's start by getting our core network inventory ready:

1. First, create the `ch05` directory, and then create your `hosts.yml` file. Let's copy over the inventory that we augmented in *Chapter 4*:

```
root@cookbook-lab:~/cookbook-lab/chapters# mkdir ch05 && cp ch04/
hosts.yml ch05/hosts.yml && cd ch05/
```

2. Update the `hosts.yml` file. Since you copied the inventory file from *Chapter 4*, you simply need to enhance it with the `vars` section:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi hosts.yml
---
network_equipment:
  children:
    arista_ceos:
    juniper_crpd:
    nokia_srlinux:
... omitted output ...
juniper_crpd:
  vars:
    ansible_network_os: "junipernetworks.junos.junos" (1)
  hosts:
    crpd-01:
      ansible_host: 198.51.100.21
    crpd-02:
      ansible_host: 198.51.100.22
... omitted output ...
```

(1) Instead of setting `network_os` in `host_vars` or `group_vars`, this time, you will set it in the `vars` section of the inventory file. As you learned in *Chapters 2 and 3*, Ansible allows the same variables to be set in multiple places. This demonstrates its flexibility.

3. Before moving on, validate that the `juniper_crpd` group has the two nodes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-inventory -i
hosts.yml --graph juniper_crpd
@juniper_crpd:
  |--crpd-01
  |--crpd-02
```

4. Create the `ansible.cfg` file. This will use the same default options as in *Chapter 4*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# cp ../ch04/ansible.
cfg . && cat ansible.cfg
[defaults]
inventory=hosts.yml
gathering = explicit
host_key_checking = False
roles_path = ~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/
roles:/root/cookbook-lab/chapters/roles
```

Notice that if you include `roles_path`, this will allow your playbook to access the `config_generation` role that was created in *Chapter 4*, which we will be extending to add support for Juniper in the coming sections (the power of reusability!).

In the next section, you will set up the remaining configuration and connection options. Juniper devices expose multiple different ways to connect, such as via **Secure Shell (SSH)** or **Network Configuration (NETCONF)**.

Connecting to Juniper devices

In this recipe, you will connect to Juniper devices via SSH in order to start managing the devices from Ansible. SSH is a common connection point, with the added benefit that network engineers are familiar with it.

1. Create **private and public SSH keys** in the `ch05` working directory:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh-keygen -t rsa -b
2048 -f Ansible_SSH_key (1)
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in Ansible_SSH_key
Your public key has been saved in Ansible_SSH_key.pub
```

```
The key fingerprint is:
SHA256:2fQreD37u5d5IVmciHu1Lh8vLQZmFaU2FLNmbFvCAMk
... omitted output ...
```

The numbered reference from the preceding code is explained as follows:

- (1) `ssh-keygen` is executed to generate the private and public keys:

We specify the encryption algorithm with the `-t` option. Here, it is set to `rsa`.

We specify the size of the encryption key using the `-b` option. Here, we set the size to 2,048 bits.

We specify the location to save the private and public keys using the `-f` option. Here, we specified the name for the public and private keys that will be generated, which is `Ansible_SSH_key`.

2. Validate that both keys, public and private, were created:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ls -la
... omitted output ...
-rw----- 1 root root 1831 Jan  9 03:21 Ansible_SSH_key (1)
-rw-r--r-- 1 root root 407 Jan  9 03:21 Ansible_SSH_key.pub (2)
```

The numbered references from the preceding code are explained as follows:

- (1) The file on the filesystem that stores the private key
 - (2) The corresponding file that stores the public key
3. Capture the public key that was created. The public key is configured in the Juniper routers in the upcoming tasks. This public key (held on the server) is used to authenticate into our Juniper router:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# cat Ansible_SSH_key.
pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACgshvHNY3pgVzjU808k9aQd-
7Nt+20gUza10rPNE4Vt2Im5ZJZd0ixien+bPBNJU0FF2nh3v0MupmNLg1n1ym0HrN-
r2zHXOut5F4zWS1sXwD5GmWkKWuu0B6yuHVR6snKJaEbUJTgICL6icWREBKHgI-
HVzYicm6RoGXewlwAXr0G1PwJYSmELSchqOyqjEBN1VTE8JLK912R9R0splJG7/
pNXAeyaJs1GQkTNARUIu15kanLo8AwcHde2w8ZN1xUzy6RLX17I3ctIOq-
fRgowHibe+N3HQjv7kwIed1ElRennpOIazdeBT2NQr/bY3m7bZE90mV/hTiUxdYznX-
8aSnxH root@cookbook-lab
```

4. To prep the Juniper devices for automation, log in using SSH. The default credentials for cRPD devices in this lab are root/clab123:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh root@crpd-01
(root@crpd-01) Password:
===>
      Containerized Routing Protocols Daemon (CRPD)
Copyright (C) 2020-2023, Juniper Networks, Inc. All rights reserved.
<===
root@crpd-01:~# (1)
```

(1) cRPD drops you into the Linux shell for the cRPD device, not the network operating system CLI directly.



Note

This behavior is specific to the containerized version of Junos (cRPD); a physical Juniper device would typically log a user directly into the Junos CLI.

5. Switch into the CLI by entering the cli command:

```
root@crpd-01:~# cli
root@crpd-01> (1)
```

(1) You are now within the Junos CLI and you can complete the manual configuration changes.

6. On the Juniper devices, add a new superuser called ansible with the associated public SSH key to allow the router to authenticate future SSH sessions using the private key. The superuser declaration in Juniper will give full access to the system:

```
root@crpd-01> configure (1)
root@crpd-01# set system login user ansible uid 2001 class
super-user authentication ssh-rsa "ssh-rsa AAAAB3NzaC1y-
c2EAAAADAQABAAQACgshvHNY3pgVzjU808k9aQd7Nt+20gUza10rP-
NE4Vt2Im5ZJZd0ixien+bPBNJU0FF2nh3v0MupmNLg1n1ym0HrNr-
2zHXOut5F4zW51sXwD5GmwKkWuu0B6yuHVR6snKJaEbUJTgICL6icWREBKHgIH-
VzYicm6RoGxewlwAXrOGlPwJYSmELSchq0yqjEBN1VTE8JLK912R9R0sp1JG7/
pNXAeyaJs1GQkTNARUIu15kanLo8AWcHde2w8ZN1xUzy6R1X17I3ctIOq-
fRgowHibe+N3HQjv7kwIed1ElRennpOIazdeBT2NQr/bY3m7bZE90mV/hTiUxdYznX-
8aSnxH root@cookbook-lab" (2)
```

The numbered references from the preceding code are explained as follows:

- (1) Enter Juniper configuration mode.
 - (2) Configure the `ansible` user to use SSH keys for authentication. Copy the public SSH key that was created on the Ansible machine to the device.
7. Commit the configuration change to make the changes take effect:

```
root@crpd-01# commit and-quit
commit complete
Exiting configuration mode
root@crpd-01>
```

8. Now you can test that connectivity, and, more specifically, the authentication, using the SSH keys we just created and configured. You will use the **Ansible ping module**, like in *Chapter 4*, but using all command-line arguments this time:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible juniper_crpd
-m ansible.builtin.ping \
-u ansible --private-key Ansible_SSH_key \ (1)
-c network_cli (2)
crpd-01 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
crpd-02 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

The numbered references from the preceding code are explained as follows:

- (1) You specify the username to connect to the devices using the `-u` option and the private SSH key using the `--private-key` option.
- (2) Finally, you use the `-c` option in order to specify the connection plugin used to connect to the managed devices. In this case, we are using the `network_cli` connection plugin to open an SSH session with the managed Juniper devices.

9. You can also try to manually log in to one of the devices from the **Ansible control host**:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh ansible@crpd-01
-i Ansible_SSH_key
Warning: Permanently added 'crpd-01' (ED25519) to the list of known
hosts.
ansible@crpd-01> (1)
```

The numbered reference from the preceding code is explained as follows:

- (1) Notice that by manually logging in to the device using SSH and using `-i` with the identity file, you log in without a password prompt, as expected because the SSH private key is used to authenticate the user.
10. Now that you've confirmed that you can connect to the Juniper devices, let's do some preparation for the next section. Create `group_vars/juniper_crpd.yml` and move the command-line options into a permanent place:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# mkdir group_vars &&
vi group_vars/juniper_crpd.yml
---
ansible_user: ansible
ansible_ssh_private_key_file: Ansible_SSH_key
```



Note

`ansible.netcommon.network_cli` is used for SSH connectivity. Using the command-line option is temporary until NETCONF is configured in the next section.

Now that you've updated the inventory file and validated that the default SSH connectivity is working for your `juniper_crpd` devices, the next section will introduce a more robust configuration option for Juniper devices that use NETCONF over SSH.

Migrating from the CLI to NETCONF

In this recipe, you will learn how to enable the **NETCONF protocol** on **Junos devices**. This task is critical since you will use NETCONF in all future recipes to manage the Juniper devices. NETCONF provides several advantages compared to the traditional CLI configuration; for example, configuration management using YANG models allows for a more standardized approach to configur-

ing network devices. Another advantage is the ability to manage configurations using different datastores (e.g., startup, running, and candidate). The candidate datastore allows changes to be prepped ahead of time!

Note



Although not used here, a lot of modern network equipment also supports a RESTCONF interface. RESTCONF supports similar configuration datastores, but simplifies the available operations (via HTTP methods) compared to NETCONF. Also, RESTCONF supports both XML and JSON, versus NETCONF, which supports only XML.

Before you can use NETCONF, you must first enable it via a specific Ansible playbook. This will be accomplished by using the `network_cli` SSH connectivity option that was enabled before.

1. Create a new playbook called `pb_junos_enable_netconf.yml` and start with a task to enable NETCONF:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_junos_enable_
netconf.yml
---
- name: "Prepare Juniper for Automation"
  hosts: "juniper_crpd"
  tasks:
    - name: "Enable NETCONF on Default Port"
      junipernetworks.junos.junos_netconf: (1)
        listens_on: "830" (2)
        state: "present"
      vars:
        ansible_connection: "ansible.builtin.network_cli" (3)
```

The numbered references from the preceding code are explained as follows:

- (1) A Juniper declarative module for configuring NETCONF on a device.
- (2) Set NETCONF to listen on port 830, which is the default port for NETCONF.
- (3) In *Chapter 4*, when we configured the eAPI for Arista, we used vars that were defined at the **play** level. For Juniper, we will use **task**-level vars. This demonstrates the Ansible precedence ordering for variables, which allows you to set this variable only for this task. Since once this task is completed and NETCONF is configured, we will no longer use this Ansible connection type.

2. Next, you will execute the playbook to configure NETCONF on both devices:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_junos_enable_netconf.yml
PLAY [Prepare Juniper for Automation] *****
TASK [Enable NETCONF on Default Port] *****
changed: [crpd-01]
changed: [crpd-02]
... omitted output ...
```

Note



If the playbook fails with [WARNING]: ansible-pylibssh not installed, falling back to paramiko, install it by running `pip install ansible-pylibssh`. You did this in previous chapters, though, so this step should not be necessary.

3. Finally, check that the configuration was successfully updated by accessing the devices via the CLI:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh ansible@crpd-01
-i Ansible_SSH_key
ansible@crpd-01> show configuration | display set | match netconf
set system services netconf ssh port 830 (1)
```

- (1) In the configuration snippet, you can see that the NETCONF service was enabled on port 830, as our playbook declared.

You can validate that you are using the correct connection setting and that Ansible is able to communicate with the Juniper devices using NETCONF with the following command, which specifies the connection type:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible all -m ansible.
builtin.ping -l crpd-01 -vvvv -c ansible.netcommon.netconf
... omitted output ...
<198.51.100.21> local domain socket does not exist, starting it
<198.51.100.21> loaded netconf plugin
... omitted output ...
<198.51.100.21> local domain socket path is /root/.ansible/pc/dbc128289f
```

Now that you have validated that Ansible can connect to the Juniper devices using the NETCONF connection plugin, let's update the inventory with the connection type in preparation for the next section:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi hosts.yml
---
network_equipment:
  children:
    arista_ceos:
    juniper_crpd:
    nokia_srlinux:
  juniper_crpd:
    vars:
      ansible_network_os: "junipernetworks.junos.junos"
      ansible_connection: "ansible.netcommon.netconf" (1)
    hosts:
      crpd-01:
        ansible_host: 198.51.100.21
      crpd-02:
        ansible_host: 198.51.100.22
... omitted output ...
```

The numbered reference from the preceding code is explained as follows:

- (1) Add the additional variable, `ansible_connection`, to the `juniper_crpd` Ansible group. This is added within the Juniper variables to properly set the connection type to use NETCONF. This is just one of the many places you can configure Ansible variables. Others will be highlighted throughout the book.



Note

The Juniper Networks collection supports multiple connection plugins. For more information on the best practices, see: https://docs.ansible.com/ansible/latest/network/user_guide/platform_junos.html.

With all the configurations completed to set up NETCONF, let's explore another declarative module available in the Juniper Networks collections to manage a read-only `ansible-ro` user, similar to what we did in the last chapter.

Creating and managing users and authentication options

In the last section, you created the ansible **superuser** manually to initiate an account to execute automation. An account with that level of privilege isn't suitable for all tasks.

In this recipe, you will create a new **read-only user** named `ansible-ro`. This user will be authenticated using a password and won't be able to authenticate using a public key, to provide some variety:

1. First, you will use `ansible-vault` to encrypt the password for the `ansible-ro` user:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-vault en-
crypt_string
New Vault password: *** (1)
Confirm New Vault password: ***
Reading plaintext input from stdin. (ctrl-d to end input, twice if
your content does not already have a newline)
ansible123 (2)
Encryption successful
!vault | $ANSIBLE_VAULT;1.1;AES256
32356
... omitted output ...
```

The numbered references from the preceding code are explained as follows:

(1) Define a vault password for this specific encryption. In this case, you will use `vault_pass`. It will prompt you to confirm this password, so enter it once again.

(2) Once the vault password is set, you enter the string to encrypt, and enter the password for `ansible-ro`, which, in this case, is `ansible123`.

2. Next, create a file called `decrypt_passwd` and put the vault password from the previous step (`vault_pass`) in it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi decrypt_passwd
text_used_to_encrypt
```

3. Update `ansible.cfg` and add the option pointing to the vault decryption file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi ansible.cfg
[defaults]
... omitted output ...
vault_password_file= ./decrypt_passwd
```

4. Unlike *Chapter 4*, where you used *hardcoded user values*, in this recipe, you will define the users to create using variables in the `group_vars/juniper_crpd.yml` file (thus, reusing the same credentials for all the cRPD devices):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi group_vars/juniper_crpd.yml
---
... omitted output ...
global:
  users:
    - name: "ansible-ro"
      password: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          32356
          ... omitted output ...
      role: "read-only"
```

5. Create a new playbook called `pb_users.yml` with the task to create a new user:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_users.yml
---
- name: "Create New User with Ansible"
  hosts: "juniper_crpd"
  tasks:
    - name: "Create the ansible-ro user with password auth"
      junipernetworks.junos.junos_user:
        name: "{{ item['name'] }}" (1)
        role: "{{ item['role'] }}" (2)
        encrypted_password: "{{ item['password'] |
            password_hash('sha512') }}" (3)
        state: present
      loop: "{{ global['users'] }}" (4)
      loop_control:
        label: "{{ item['name'] }}" (5)
```

The numbered references from the preceding code are explained as follows:

- (1) For each loop of the `global['users']` list from `group_vars`, set the username.
- (2) The role to assign the user as defined in `group_vars`.

- (3) Grabbing the password from `group_vars`, it will be decrypted using the `vault_password_file` you set in `ansible.cfg`, followed by sending the cleartext password through a **password hashing filter**. This Ansible filter has a Python dependency that will be covered in the next step.
 - (4) Look up `group_vars` for the host and get the nested list of users from the `global` dictionary variable.
 - (5) The loop label modifies what gets printed out to the playbook logs for each item. This limits the output to only the username, instead of the full dictionary.
6. Before you can run the playbook to configure the user, install two Python requirements required by the `NETCONF` connection and the `password_hash` Ansible filter, respectively:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# pip install ncclient
passlib (1)
... omitted output ...
Installing collected packages: lxml, ncclient, passlib
Successfully installed lxml-5.3.0 ncclient-0.6.16 passlib-1.7.4
```

The numbered reference from the preceding code is explained as follows:

- (1) `ncclient` is used by the Juniper collection, specifically, the `NETCONF` connection. `passlib` is used to encrypt the password via `| password_hash` Ansible filter before sending it as an argument to the Ansible module.
7. Finally, run the playbook to create the `ansible-ro` user on both Juniper routers:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_users.yml
PLAY [Create New User with Ansible] *****
TASK [Create the ansible-ro user with password auth] *****
changed: [crpd-02] => (item=ansible-ro)
changed: [crpd-01] => (item=ansible-ro)
... omitted output ...
```

8. To validate that the user was properly configured, try to SSH to one of the Juniper devices with the credentials defined:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh ansible-ro@crpd-
01
(ansible-ro@crpd-01) Password: ***
ansible-ro@crpd-01>
```

With our users configured and authentication validated, let's configure some of the generic system options on the core routers.

Configuring generic system options on Juniper devices

In this recipe, you will learn how to configure some basic system options, such as **hostname** and **Domain Name System (DNS) servers**. You will understand how to set up all these *system-level parameters* using various Ansible modules, and learn the different ways to manage these parameters:

1. The first update you need to make is to the `groups_vars/juniper_crpd.yml` file. Here, the global defaults need to be specified so that you can access them programmatically, all while managing them from a single spot within the project:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi group_vars/juniper_crpd.yml
---
global:
... omitted output ...
  dns:
    - 8.8.8.8
    - 8.8.4.4
```

In this case, you are adding a new dictionary key named `dns`, which is nested under the `global` root key, which already exists with our user variables from the last section. This is a list of IP addresses to define the DNS servers to be used by the network devices. Here, you will use the two common public Google DNS servers for testing.

2. Create a new playbook, `pb_basic_config.yml`. This playbook should look familiar to you as it is identical to the Arista playbook from *Chapter 4*, except for the module name!

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_basic_config.yml
---
- name: "Configure Hostname and DNS"
  hosts: "juniper_crpd"
  tasks:
    - name: "Configure Basic Configuration on Juniper Core"
      junipernetworks.junos.junos_system:
        hostname: "{{ inventory_hostname }}" (1)
        name_servers: "{{ global['dns'] }}" (2)
        state: "present"
```


The numbered references from the preceding code are explained as follows:

- (1) Here, you are setting the hostname of the Juniper device to equal the hostname from the inventory, using the magic variable exposed by Ansible (`inventory_hostname`). Magic variables are used by Ansible to reflect internal state and cannot be set directly by the user. This makes the variables usable without having to worry about them being previously modified.
- (2) Use the variable defined in the `group_vars/juniper_crpd.yml` file to create the DNS name servers. Notice that this attribute takes a list instead of item by item.
3. Execute the `pb_basic_config.yml` playbook to configure the DNS servers and the host-name:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_basic_config.yml
PLAY [Configure Hostname and DNS] *****
***
TASK [Configure Basic Configuration on Juniper Core]
*****
changed: [crpd-01]
changed: [crpd-02]
... omitted output ...
```

4. If you log in to one of the devices, you can see the configuration changes were successful:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh ansible@crpd-01
-i Ansible_SSH_key
ansible@crpd-01> show configuration | display set | grep name-server
set system name-server 8.8.8.8
set system name-server 8.8.4.4
```

This section demonstrated how many declarative Ansible modules across vendors can share the same module schema (but this is just a coincidence!). Here, you used `junos_system`. This playbook is identical to the module that was used for `eos_system` to configure the generic system options on Arista. This common schema can be an added benefit to using Ansible declarative modules across a multi-vendor environment.

In the next section, you will configure the interfaces by reusing them on top of the `config_generation` role that was created in *Chapter 4*.

Configuring Juniper interfaces

In this recipe, we will configure the **interfaces** on our *Juniper core routers*. The attributes that will be configured include the **IP address** and **interface description information**. The `junos_interface` declarative module will not be used in this section; instead, we will add the necessary Juniper templates to our `config_generation` role. This will demonstrate how developing a role with reusability in mind can vastly speed up a network automation strategy.



Reminder

The code for the Ansible role can be found in https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/roles.

To set the stage, the following diagram shows the interface followed by a table with the IP address schema that you will be building out. The diagram provides a logical network diagram for our core network topology, including interface names and logical endpoint connections.

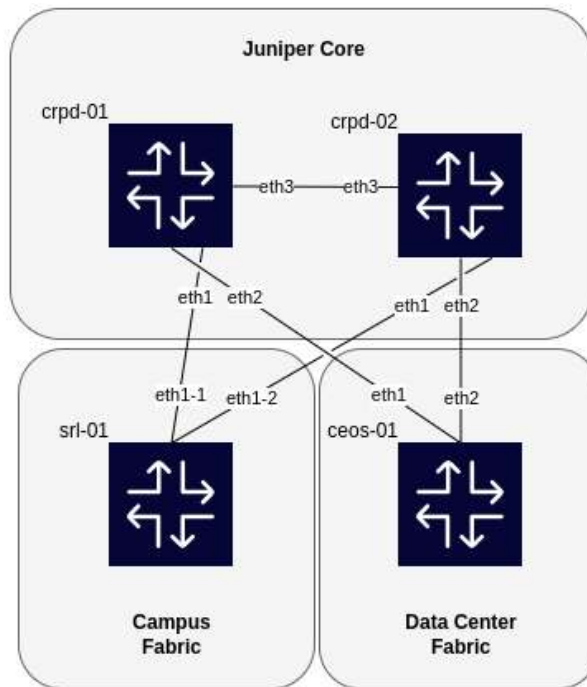


Figure 5.2 - Juniper core

The preceding diagram does not depict the entire lab topology. It has been condensed to focus on the nodes you will be working with in this chapter. In the following table, you can see how the interfaces in the diagram will be configured with IP addresses.

Device	Interface	IP Address
crpd-01	eth0 (Management)	198.51.100.21/24
crpd-01	eth1	172.31.2.5/31
crpd-01	eth2	172.31.1.5/31
crpd-01	eth3	10.0.0.0/31
crpd-01	Loopback0	10.200.1.1/32
crpd-02	eth0 (Management)	198.51.100.22/24
crpd-02	eth1	172.31.2.7/31
crpd-02	eth2	172.31.1.7/31
crpd-02	eth3	10.0.0.1/31
crpd-02	Loopback0	10.200.1.2/32

Table 5.1 Interface IP address scheme



Note

The management IPs are defined by the Containerlab topology.

Each host will have a dedicated vars file that is stored in `host_vars`, using the expected pattern. If you need a refresher on this pattern, it was explained in the *Defining Ansible variables* section in *Chapter 2*. `host_vars/{{ inventory_hostname }}.yaml` will auto-load the variables into the execution (and repeat the equivalent for `crpd-02`). Let's get started with defining our host variables:

1. Create `host_vars/crpd-01.yaml` and then create the following interface data:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi host_vars/crpd-01.yaml
---
interfaces:
  Eth0:
    ip_address: "198.51.100.21/24"
    description: "CLAB-MGMT-NETWORK"
```

```

Eth1:
  ip_address: "172.31.2.5/31"
  description: "CORE-TO-CAMPUS-SRL-01-ETH1/1"
Eth2:
  ip_address: "172.31.1.5/31"
  description: "CORE-TO-DC-CEOS-01-ETH1"
Eth3:
  ip_address: "10.0.0.0/31"
  description: "CORE-TO-CORE-CRPD-02-ETH3"
Lo0:
  ip_address: "10.200.1.1/32"
  description: "LOOPBACK-FOR-ROUTER-ID"

```

2. Now that the data model is defined for the interfaces and IP addresses, let's write the template to use this data and transform the variables into Juniper configuration. We will reuse the role that was created in the last chapter, so we will follow a similar pattern by creating the templates directory and putting the templates in there:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch05# mkdir templates &&
cd templates && vi interfaces.j2
{% for name, int_details in interfaces.items() %} (1)
set interfaces {{ name | lower }} unit 0 family inet address {{ int_
details['ip_address'] }}
set interfaces {{ name | lower }} unit 0 description {{ int_de
tails['description'] | upper }}
{% endfor %}

```

(1) Loop through the dictionary you defined previously in `host_vars`. Using the `.items()` method, which allows access to the key-value pairs of the dictionary. The value of the `int_details` variable is an inner dictionary; therefore, our subsequent lookups are dictionary key lookups to get the value.

3. The `config_generation` role expects the entry point file called `junos.j2`. Remember that these entry point files allow the Ansible role to call a single Jinja2 file to render all the different templates you need, resulting in a full configuration. Create this file and use the Jinja2 include statement to include the `interfaces.j2` file:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi templates/junos.
j2
{% include "../interfaces.j2" %} (1)

```

(1) Use a relative path to include the file named `interfaces.j2`.

4. Create a new playbook named `pb_advanced_configs.yml`, which will call our role:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_advanced_configs.yml
---
- name: "BUILD CONFIGS FOR JUNIPER"
  hosts: "juniper_crpd"
  connection: "local"
  roles:
    - "config_generation"
```

5. Finally, execute the playbook. The role will run and save the interface configurations output after they've been rendered by Jinja2. The Ansible role was configured to store these configurations in the `output/config` directory in the current working directory:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR JUNIPER] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
changed: [crpd-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2 TEMPLATE] *****
changed: [crpd-01]
changed: [crpd-02]
```

6. Check the file located at `output/config/crpd-01.cfg` to view the rendered template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# cat output/config/crpd-01.cfg
# Comment line crpd-01 config
set interfaces eth0 unit 0 family inet address 198.51.100.21/24
set interfaces eth0 unit 0 description CLAB-MGMT-NETWORK
... omitted output ...
set interfaces lo0 unit 0 family inet address 10.200.1.1/32
set interfaces lo0 unit 0 description LOOPBACK-FOR-ROUTER-ID
```

Now that we've generated the interface configuration for our core devices, we're going to build out the other configuration snippets to finalize the full configuration.

In the next section, you will generate the routing configuration, including BGP and **Open Shortest Path First (OSPF)**.

Advanced routing configuration using the template module

In this recipe, you will create configuration templates for BGP and OSPF. BGP is used to connect to both the campus fabric and the data center fabric that was created in the last chapter. OSPF will be used as the core network **interior gateway protocol (IGP)** to transport all the routes throughout the core. The following logical diagram provides more details. The details specifically show what links will be using BGP, and where and how the OSPF area will be configured.

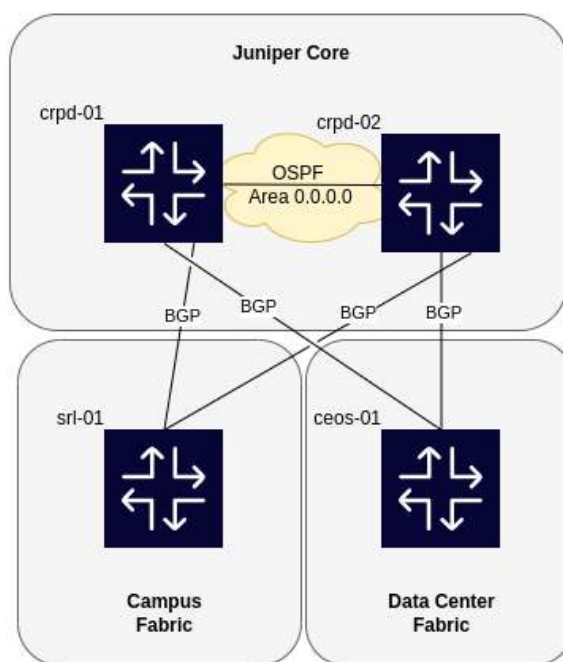


Figure 5.3 - Core routing protocols

Note



OSPF is used as the IGP between the data center and campus networks, for sharing routes between the different infrastructures. **RFC 7938** advocates for BGP in large-scale data centers by highlighting its advantages in scalability, convergence, and flexibility, which are critical in those environments and where traditional IGPs such as OSPF might fall short. However, that level of complexity is not necessary in our lab topology.

The following table outlines the BGP **autonomous system number (ASN)** for each of the devices we're configuring in this section.

Node	Role	BGP ASN
crpd-01	Core router	65400
crpd-02	Core router	65400
ceos-01	Data center spine	65100
srl-01	Campus spine	65200

Table 5.2 – BGP ASN mapping

With the topology and logic routing designs covered, let's start defining our data model in order to set up our configuration generation:

1. Update the host variable data in `crpd-01.yml` to add the BGP configuration and provide the neighbor details:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi host_vars/crpd-01.yml
... omitted output ...
bgp:
  asn: 65400
  peer_groups:
    ebgp-peer-dc: (1)
      - peer: ceos-01
        ip_address: 172.31.1.4 (2)
        local_address: 172.31.1.5 (3)
        remote_as: 65100 (4)
    ebgp-peer-campus: (5)
      - peer: srl-01
        ip_address: 172.31.2.4
        local_address: 172.31.2.5
        remote_as: 65200
```

The numbered references from the preceding code are explained as follows:

- (1) Add the BGP peer group named `ebgp-peer-dc` to define neighbors to the data center spine router.
- (2) For each BGP neighbor to create, you will need the corresponding neighbor IP address.

- (3) Juniper configurations also require the local address for the BGP neighborhood.
 - (4) A required element to create a BGP neighbor is also the remote ASN.
 - (5) The second peer group for the neighbor facing the campus fabric (to be configured in *Chapter 6*.
2. Create `bgp.j2` by activating the new `peer_group` under the IPv4 address family. Note that there are no updates to the rest of the template as our existing loop will create the peer group that is needed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi templates/bgpe.j2
set routing-options autonomous-system {{ bgp['asn'] }} (1)
set routing-options router-id {{ interfaces['Lo0']['ip_address'].
split('/')[0] }} (2)
{% for group_name, peer_group_peers in bgp['peer_groups'].items() %}
(3)
{% for bgp_peer in peer_group_peers %} (4)
set protocols bgp group {{ group_name }} type external
set protocols bgp group {{ group_name }} peer-as {{ bgp_peer['re-
mote_as'] }}
set protocols bgp group {{ group_name }} neighbor {{ bgp_peer['ip_
address'].split('/')[0] }}
set protocols bgp group {{ group_name }} local-address {{ bgp_
peer['local_address'].split('/')[0] }}
{% endfor %}
{% endfor %}
```

The numbered references from the preceding code are explained as follows:

- (1) Populate the BGP ASN from the `host_vars` definition.
- (2) `router-id` will be using the `Lo0` interface IP address.
- (3) Loop through the `peer_groups` defined in `host_vars`. Each peer group will create the peer group configuration snippets.
- (4) Now that that peer group has been created with the required configuration data, each peer has to be created.

3. Before running the playbook to generate the configuration again, to get the new configuration, you need to include `bgp.j2` in the `junos.j2` entry point template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi templates/junos.j2

{% include "./interfaces.j2" %}
{% include "./bgp.j2" %}
```

4. Now that you've updated the template, you can run the playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR JUNIPER] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [crpd-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2 TEM-
PLATE] *****
changed: [crpd-01]
changed: [crpd-02]
... omitted output ...
```

5. If you view the `crpd-01` generated configuration again, you will see the rendered BGP configuration for the core routers appended:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi output/config/
crpd-01.cfg
... omitted output ...
set routing-options autonomous-system 65400
set routing-options router-id 10.200.1.1
set protocols bgp group ebgp-peer-dc type external
set protocols bgp group ebgp-peer-dc peer-as 65100
set protocols bgp group ebgp-peer-dc neighbor 172.31.1.4
set protocols bgp group ebgp-peer-dc local-address 172.31.1.5
set protocols bgp group ebgp-peer-campus type external
set protocols bgp group ebgp-peer-campus peer-as 65200
set protocols bgp group ebgp-peer-campus neighbor 172.31.2.4
set protocols bgp group ebgp-peer-campus local-address 172.31.2.5
... omitted output ...
```

Notice that both BGP peer groups and the corresponding neighbor statements have been created.



Note

The BGP peers facing the campus will remain down until *Chapter 6*, where you will configure the campus side of the BGP neighbor.

Next, you will create the OSPF configuration for the area 0.0.0.0. There will only be one neighbor for our topology. OSPF will be able to share routes from the data center to the campus and vice versa. This is important to allow our lab topology to have end-to-end reachability between our two clients. You will validate this reachability in later chapters as we learn about Batfish. Let's explore the OSPF template first:

1. Create the ospf.j2 template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi templates/ospf.j2
set protocols ospf area 0.0.0.0 interface eth3 interface-type p2p
(1)
set protocols ospf area 0.0.0.0 interface lo.0 interface-type nbma
```

(1) Set eth3, which is the interface between the two cRPD core devices, as a point-to-point link. Specifying interface-type helps the routers create the proper OSPF link state database route entries.

2. Execute the playbook once again to update our rendering configuration files with the OSPF configuration lines:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR JUNIPER] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [crpd-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2 TEM-
PLATE] *****
changed: [crpd-01]
changed: [crpd-02]
... omitted output ...
```

With the routing configuration completed, the last step is deploying the configurations to the devices.

Pushing configuration to Juniper devices

In the previous recipes, you generated different sections of the configuration for Juniper devices such as interfaces, OSPF, and BGP. You used the template module to generate a single configuration file. This file is stored in the `output/config` folder for each device. You will use the `junos_config` module to push this configuration file that you have generated. You can use the `update` parameter to control how the configuration that you want to push will be merged with the existing configuration on the device. It supports the following options:

- `merge`: This causes the configuration from our file to be merged with the configuration on the device (the candidate configuration). This is the default option that is used.
- `override/update`: This causes the configuration from our file to override the complete configuration on the managed device.

With the configuration options explained, we can move on to creating our playbooks:

1. Create `pb_deploy_configs.yml` with one task. The `junos_config` module accepts a configuration file and merges the files' contents into the running configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_deploy_configs.yml
---
- name: "DEPLOY CONFIGS FOR JUNIPER"
  hosts: "juniper_crpdpd"
  tasks:
    - name: "DEPLOY ADVANCED CONFIG"
      junipernetworks.junos.junos_config:
        src: "output/config/{{ inventory_hostname }}.cfg" (1)
        update: "merge" (2)
```

The numbered references from the preceding code are explained as follows:

- (1) Point the `src` argument to the generated file that was built during the `config_generation` role.
- (2) `merge` combines the data in the loaded configuration with the candidate configuration. `merge` is the default for the module, but defining it explicitly makes it more apparent.

2. Run the playbook to run the `config_generation` role, and execute the new task you just created to deploy those configurations:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_deploy_configs.yml
PLAY [DEPLOY CONFIGS FOR JUNIPER] *****
TASK [DEPLOY ADVANCED CONFIG] *****
changed: [crpd-02]
changed: [crpd-01]
```

3. Validate idempotence and rerun the playbook. Notice that no changes were needed on any task that was executed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_deploy_configs.yml
PLAY [DEPLOY CONFIGS FOR JUNIPER] *****
TASK [DEPLOY ADVANCED CONFIG] *****
ok: [crpd-02]
ok: [crpd-01]
```

4. Quickly looking at the configurations, you'll see that the correct configuration was pushed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ssh ansible@crpd-01
-i Ansible_SSH_key
ansible@crpd-01> show configuration | display set | grep protocols
set protocols bgp group ebgp-peer-dc type external
set protocols bgp group ebgp-peer-dc local-address 172.31.1.5
set protocols bgp group ebgp-peer-dc peer-as 65100
set protocols bgp group ebgp-peer-dc neighbor 172.31.1.4
set protocols bgp group ebgp-peer-campus type external
set protocols bgp group ebgp-peer-campus local-address 172.31.2.5
set protocols bgp group ebgp-peer-campus peer-as 65200
set protocols bgp group ebgp-peer-campus neighbor 172.31.2.4
set protocols ospf area 0.0.0.0 interface eth3 interface-type p2p
set protocols ospf area 0.0.0.0 interface lo.0 interface-type nbma
```



Info

For more information on the `junos_config` module for Juniper, see https://docs.ansible.com/ansible/latest/collections/junipernetworks/junos/junos_config_module.html.

Now that all the configuration has been deployed to the core network, let's explore how to validate the operational state of the network after the deployment is processed.

Collecting Juniper device facts using Ansible

This recipe will provide you with a playbook to validate BGP on the Juniper core devices that you configured. You will retrieve operational state data using the `junos_rpc` module and then validate the data retrieved. Let's get started by creating the playbook to validate our device facts:

1. Create a new playbook called `pb_validate_facts.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_validate_facts.yml
---
- name: "Collect and Validate Juniper Core Facts"
  hosts: "juniper_crpd"
  tasks:
    - name: "Collect BGP Facts Using RPC"
      junipernetworks.junos.junos_rpc: (1)
        rpc: get-bgp-summary-information
      register: bgp_information
    - name: "Set Fact for the response XML data"
      ansible.builtin.set_fact: (2)
        parsed_xml_data: "{{ bgp_information['output'][0] |
          ansible.utils.from_xml }}" (3)
    - name: "Validate BGP Neighbor Exists"
      ansible.builtin.assert:
        that:
          - parsed_xml_data['rpc-reply']['bgp-information']
            ['bgp-peer'] | selectattr("peer-address", "equalto",
              item['ip_address']) (4)
      quiet: true
```

```
fail_msg: "{{ item }}" peering is not configured."
loop: "{{ bgp['peer_groups']['ebgp-peer-dc'] }}"
loop_control:
  label: "{{ item['ip_address'] }}" BGP Neighbor Exist"
```

The numbered references from the preceding code are explained as follows:

- (1) The `junos_rpc` module runs a **remote procedure call (RPC)** against the Juniper devices to collect the specified device facts. In this case, the `get-bgp-summary-information` RPC returns the basic BGP operational state data.
- (2) The `set_fact` module is used to set a variable to be reused later.
- (3) The `from_xml` filter is also used to turn the returned RPC XML data into a Python data structure.
- (4) Evaluate the BGP `ip_address` from `host_vars` and assert that the `peer-address` attribute that is configured on the device is the same value.



Note

The `junos_facts` module for Juniper does exist and could be another option; however, it's not being used here as many of the commands that are run are not supported on the `crpd` platform.

2. Run the `pb_validate_facts.yml` playbook to validate the operational state of our fabric:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_validate_facts.yml
PLAY [Collect and Validate Juniper Core Facts] *****
TASK [Collect BGP Facts Using RPC] *****
ok: [crpd-02]
ok: [crpd-01]
TASK [Set Fact for the response XML data] *****
ok: [crpd-02]
ok: [crpd-01]
TASK [Validate BGP Neighbor Exists] *****
ok: [crpd-01] => (item=172.31.1.4 BGP Neighbor Exist)
ok: [crpd-02] => (item=172.31.1.6 BGP Neighbor Exist)
... omitted output ...
```

In this recipe, you used structured data gathered from the `juniper_rpc` module to process some operational checks. In the next section of this chapter, you will learn how to run ad hoc CLI commands against the devices.

Running arbitrary commands on Juniper devices

In this recipe, you will execute operational commands on Juniper devices. Although we just explored executing RPCs, which are similar, the goal of this section is to execute show commands and use the output to validate the state of the devices.

1. Create a new playbook called `pb_get_ospf.yml` and populate it to execute the `show ospf neighbor` command and store the output in a variable:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_get_ospf.yml
---
- name: "Collect and Validate OSPF"
  hosts: "juniper_crpd"
  tasks:
    - name: "Collect OSPF Neighbors"
      junipernetworks.junos.junos_command:
        commands: show ospf neighbor | display json (1)
        register: ospf_information
    - name: "Validate All OSPF Peers are in Full State"
      ansible.builtin.assert:
        that:
          item['ospf-neighbor-state'][0]['data'] == 'Full' (2)
        quiet: true (3)
      loop: "{{ ospf_information['stdout'][0]
                ['ospf-neighbor-information'][0]['ospf-neighbor'] }}"
      loop_control:
        label: "OSPF Neighbor {{ item['neighbor-address'][0]['data']
                }}"
```

The numbered references from the preceding code are explained as follows:

- (1) Use the `junos_command` module to run an arbitrary command. Run `show ospf neighbor | display json`. Passing the native (`| display json`) switch will return native JSON data.
- (2) Using the `assert` module, you will validate that for any OSPF neighbor found, the OSPF neighborship is in the **full** state. If this state is unfamiliar to you, the full state signifies a successful neighbor adjacency where routers are fully synchronized.

- (3) The `quiet` argument simply turns verbose outputs off. It avoids printing the entire contents of the item.
2. Execute the playbook to validate that the OSPF neighbors are in the *full neighbor state*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_get_ospf.yml
PLAY [Collect and Validate OSPF] *****
TASK [Collect OSPF Neighbors] *****
ok: [crpd-01]
ok: [crpd-02]
TASK [Validate All OSPF Peers are in Full State] *****
ok: [crpd-01] => (item=OSPF Neighbor 10.0.0.1)
ok: [crpd-02] => (item=OSPF Neighbor 10.0.0.0)
... omitted output ...
```

Now that you know two methods for retrieving data from a Juniper device, the final section of this chapter covers using the `parse_xml` filter to work with standard XML data that is returned from the NETCONF session.

Parsing Junos commands using `parse_xml`

In this recipe, you will learn how to parse XML data into JSON. Unlike previous examples, where we used `from_xml` to translate the data into native Python data structures, this recipe will show an alternative way of parsing XML into JSON, with the added flexibility of providing your own specification file.

1. Create a playbook called `pb_get_bgp.yml` to get the BGP neighbor state:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_get_bgp.yml
---
- name: "Collect BGP and Parse XML with parse_xml"
  hosts: "juniper_crpd"
  tasks:
    - name: "Collect BGP Neighbors"
      junipernetworks.junos.junos_command:
        commands: "show bgp neighbor"
        display: "xml" (1)
        register: "output"
    - name: "Debug XML Output"
```



```
ansible.builtin.debug:
  msg: "{{ output['stdout'][0] }}" (2)
```

The numbered references from the preceding code are explained as follows:

- (1) Dictate to the module the format the response should come back in, in this case, XML.
 - (2) Debug message to display the XML data (unparsed) to the screen, which is helpful during step 4, where the spec creation will occur.
2. Next, you can run the playbook to see the XML response data that gets outputted:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook
pb_get_bgp.yml
PLAY [Collect BGP and Parse XML with parse_xml] *****
TASK [Collect BGP Neighbors] *****
ok: [crpd-01]
ok: [crpd-02]
TASK [Debug XML Output] *****
ok: [crpd-01] => {
  "msg": "<rpc-reply message-id=\"urn:uuid:de33b4bb-d0f4-4043-a560-
48044321ceb2\"><bgp-information><bgp-peer style=\"detail\"><peer-ad-
dress>172.31.1.4+179</peer-address>
... omitted output ...
```

3. The output of the XML when formatted looks like the following:

```
<rpc-reply message-id="urn:uuid:426b5c75-f902-4162-a28c-
5257a0ae00aa">
  <bgp-information>
    <bgp-peer style="detail">
      <peer-address>172.31.1.6+179</peer-address>
      <peer-state>Established</peer-state>
... omitted output ...
```

4. Now, let's explore how to define your own XML parsing data model to get out the information that you care about. You will also create a specs/ directory and a spec file called show_bgp_spec.yml:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# mk-
dir specs && vi specs/show_bgp_spec.yml
---
```

```
vars:
  bgp_peers: (1)
    address: "{{ item.address }}"
    peer_state: "{{ item.peer_state }}"
keys:
  result: (2)
    value: "{{ bgp_peers }}" (3)
    top: "bgp-information/bgp-peer" (4)
    items: (5)
      address: "peer-address" (6)
      peer_state: "peer-state" (7)
```

The numbered references from the preceding code are explained as follows:

- (1) A variable definition is to be used in our parsed output creation.
 - (2) The dictionary key for the output to be represented inside.
 - (3) The data that will be assigned to the key provided before. In this example, result is the dictionary key, and the value of that key becomes the bgp_peers variable.
 - (4) The XPath from the root of the XML data where the parsing tree should begin.
 - (5) Definition of items on a per XML element (index).
 - (6) From the top XPath definition how to extract the given data attribute for the BGP peer address.
 - (7) Same as the previous point, except for retrieving the BGP peer state.
5. Update pb_get_bgp.yml, adding another debug task to call the parse_xml filter:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch05# vi pb_get_bgp.yml
---
- name: "Collect BGP and Parse XML with parse_xml"
  hosts: "juniper_crp"
  tasks:
... omitted output ...
- name: "Use parse_xml to transform XML to JSON"
  ansible.builtin.debug:
    msg: "{{ output['stdout'][0] | ansible.netcommon.parse_xml(
      'specs/show_bgp_spec.yml') }}"
```

6. Execute the playbook a final time to see the parsed data:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch05# ansible-playbook pb_
get_bgp.yml
PLAY [Collect BGP and Parse XML with parse_xml] *****
TASK [Collect BGP Neighbors] *****
ok: [crpd-01]
ok: [crpd-02]
... omitted output ...
TASK [Use parse_xml to transform XML to JSON] *****
ok: [crpd-01] => {
  "msg": {
    "result": [ (1)
      { (2)
        "address": "172.31.1.4",
        "peer_state": "Established" (3)
      },
      {
        "address": "172.31.2.4",
        "peer_state": "Idle" (4)
      }
    ]
  }
}
ok: [crpd-02] => {
  "msg": {
    "result": [
      {
        "address": "172.31.1.6",
        "peer_state": "Established" (3)
      },
      {
        "address": "172.31.2.6",
        "peer_state": "Idle" (4)
      }
    ]
  }
}
... omitted output ...

```

The numbered references from the preceding code are explained as follows:

- (1) The `result` dictionary key that we specified in the spec we just created.
- (2) A list of dictionaries that have the keys that we defined in the spec. The values are the exact data attributes that were pulled out of the XML-encoded data.
- (3) The two BGP neighbors facing the Arista data center are up (`Established`).
- (4) The two BGP neighbors facing the campus are down (`Idle`) because we have not configured the campus side of the BGP neighbor yet (we will do this in *Chapter 6*)!

Now that you've explored how to transform XML output into a data model specification that meets your needs, you have a robust way to validate operational data that is returned from NETCONF using XML encoding.

Summary

In this chapter, you have set up Juniper devices to prepare for Ansible automating configurations. You also extended the Ansible role that generates configurations. Next, you pushed configurations and validated the operational status after the change. So, you learned two different ways of validating the operational state of the devices, as well as how to parse XML data.

In the next chapter, you will configure the campus fabric of the network scenario built with Nokia devices.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



6

Managing Nokia SR Devices Using Ansible

Now that you've automated the data center fabric using Arista, and the core network built on top of Juniper, it's time to manage **Nokia SR Linux devices**, building a **Campus Fabric**. A network fabric is the underlying infrastructure of interconnected network devices, such as switches and routers, designed to provide high-bandwidth, low-latency connectivity for efficient data transfer, and has largely replaced older network three-tier designs. In this chapter, you will use the **Nokia Ansible modules** to connect to SR Linux devices and configure and collect operational data from the devices.

According to the Nokia website, the Nokia **Service Router Linux (SR Linux)** is a truly open, extensible, and resilient **network operating system (NOS)** that makes your IP and data center networks more scalable, flexible, and simple to operate. The platform supports multiple methodologies to configure and manage the system. The Ansible module utilizes the JSON-RPC interface, but SR Linux also has a CLI, as well as a gNMI interface.

As mentioned, in this chapter, you will complete the lab scenario described in *Chapter 2* by implementing the campus fabric. The lab is available using *Chapter 16* instructions.

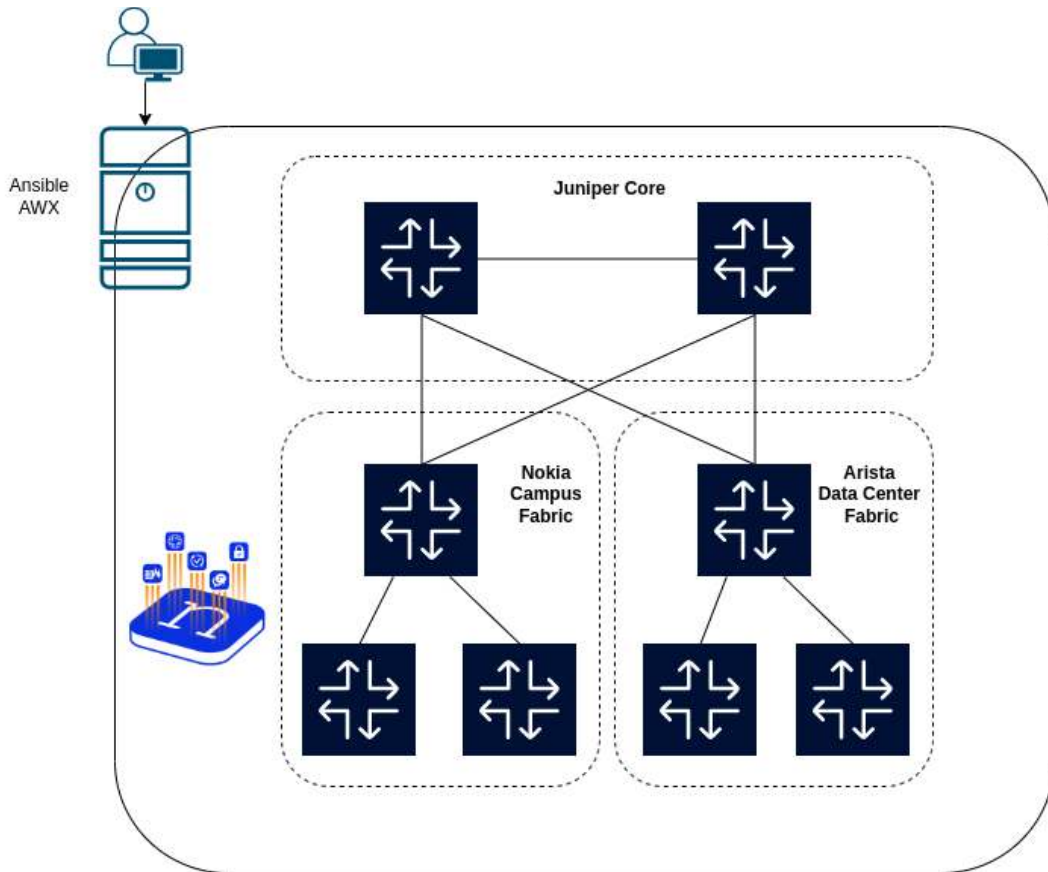


Figure 6.1 – Lab environment setup



Note

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch06

The recipes covered in this chapter are as follows:

- Augmenting the Ansible network inventory
- Connecting to Nokia devices
- Creating and managing users and authentication options
- Configuring generic system options on Nokia devices

- Configuring Nokia interfaces
- Advanced routing configuration using template modules
- Finalizing BGP configuration to the core
- Pushing the configuration to Nokia devices
- Collecting Nokia device facts using Ansible
- Running arbitrary commands on Nokia devices

Let's get started by setting up the inventory to prepare for configuring our campus fabric.

Augmenting the Ansible network inventory

The campus fabric inventory has *three nodes, one spine, and two leafs*. Spines serve as the backbone devices and create connectivity to all leaf nodes, whereas leafs connect to endpoints (servers, hosts, etc.). The devices for each of these roles (e.g., leafs and spines) will be added to the Ansible groups previously created, called **leafs** and **spines**. To get started, let's get our variable files created and our data model developed:

First, create the `ch06` directory and create your `hosts.yml` file. Let's copy over the inventory that we augmented in *Chapter 5*:

```
root@cookbook-lab:~/cookbook-lab/chapters# mkdir ch06 && cp ch05/hosts.yml ch06/hosts.yml
```

1. Update the `hosts.yml` file. Since you copied the previous inventory file that already contained the `nokia_srlinux` group and hosts, you simply need to enhance it with the `vars` section:

```
---
network_equipment:
  children:
    arista_ceos:
    juniper_crpd:
    nokia_srlinux:
    ... omitted output ...
    nokia_srlinux:
      vars:
        ansible_network_os: "nokia.srlinux.srlinux" (1)
        ansible_connection: "ansible.netcommon.httpapi" (2)
      hosts:
        srl-01:
```



```
    ansible_host: 198.51.100.31
srl-02:
    ansible_host: 198.51.100.32
srl-03:
    ansible_host: 198.51.100.33
... omitted output ...
```

The numbered references from the output are explained here:

- (1) Set `network_os` in the vars section of the inventory file.
 - (2) The `nokia.srlinux` collection uses a JSON-RPC interface connection by utilizing the `httpapi` connection plugin. This should look familiar from *Chapter 4*, as you used the same connection plugin for Arista eAPI.
2. For added flexibility, add `srl-01` to the group called `spines` and assign `srl-02` and `srl-03` to the `leafs` group. These groups already exist with the Arista fabric devices in them.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi hosts.yml
-
... omitted output ...
nokia_srlinux:
  hosts:
  ... omitted output ...
  children: (1)
spines:
leafs:
spines: (2)
  hosts:
ceos-01:
srl-01:
leafs: (3)
  hosts:
ceos-02:
ceos-03:
srl-0[2:3]:
```

The numbered references from the output are explained here:

- (1) Set the two child groups under the `nokia_srlinux` parent called `spines` and `leafs`. The groupings make it easier to create configurations based on the role of the device.

- (2) Update the spines group in the inventory, and assign `sr1-01` to the group.
- (3) Update the leafs group in the inventory, and assign `sr1-0[2:3]` to that group using the inventory pattern syntax that was covered in *Chapter 2*. You want to add the devices to their group, as it will make conditional configuration logic in our templates easier to manage.
3. Finally, validate that the `nokia_srlinux` group has the three nodes and the two groups (leafs/spines) now have both Arista and Nokia devices:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-inventory -i
hosts.yml --graph nokia_srlinux
@nokia_srlinux:
  |--sr1-01
  |--sr1-02
  |--sr1-03
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-inventory -i
hosts.yml --graph spines
@spines:
  |--ceos-01
  |--sr1-01
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-inventory -i
hosts.yml --graph leafs
@leafs:
  |--ceos-02
  |--ceos-03
  |--sr1-02
  |--sr1-03
```

Now that the inventory groups have been created and nodes have been assigned to each of the groups, the next section will set up the remaining configuration and connection options.

Connecting to Nokia devices

In this recipe, you will connect to Nokia devices via the JSON-RPC interface in order to start managing the devices from Ansible:

1. Copy the `ansible.cfg` file from the last chapter, since the file is identical:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# cp ../ch05/ansible.
cfg .
```

2. In the previous chapters, there was some manual configuration that was required on the device, but in the case of **Nokia SR Linux on Containerlab**, this management config comes natively on the devices. To validate that the JSON-RPC interface is enabled, log in to the device. The default credentials are (admin/NokiaSr11!):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ssh admin@srl-01
.....
:           Welcome to Nokia SR Linux!           :
:           Open Network OS for the NetOps era.     :
... omitted output ...
(admin@srl-01) Password:
--{ + running }--[ ]--
A:srl-01#
```

3. Run the info command from the SR Linux device to ensure the JSON-RPC configuration is present and accurate:

```
A:srl-01# info /system json-rpc-server
system {
  json-rpc-server (1) {
    admin-state enable
    network-instance mgmt {
      http (2) {
        admin-state enable
      }
      https (3) {
        admin-state enable
        tls-profile clab-profile
      }
    }
  }
}
--{ + running }--[ ]--
```

The numbered references from the output are explained here:

(1) The json-rpc-server section of the config is enabled and set to use the mgmt network instance.

(2) The configuration for the JSON-RPC server on HTTP (**port 80**) is enabled.

(3) The configuration for the JSON-RPC HTTPS (**port 443**) is enabled.



Note

To learn more about the Nokia SR Linux JSON-RPC interface, visit <https://learn.srlinux.dev/tutorials/programmability/json-rpc/basics/#json-rpc-basics>

4. Create a file named `decrypt_password` where you will store the decryption password you will use in the next step:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi decrypt_password
supers3cret!
```

5. Now, with `ansible-vault`, you can persist the required credentials in a YAML file. (Note: the Ansible Vault password is taken from `ansible.cfg`.)

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-vault create
srl_secret.yml
... an editor is opened ...
---
ansible_username: "admin"
ansible_password: "NokiaSrl!"
```

6. Quickly validate that the vaulted file was created and encrypted:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi srl_secret.yml
$ANSIBLE_VAULT;1.1;AES256
66306538643061306463663362
... omitted output ...
```

7. Now you can test that connectivity. Instead of using the `ping` module from the command line, you will create a playbook to do the testing. Create the `pb_ping.yml` playbook, which you will use to validate reachability via ICMP:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_ping.yml
---
- name: "Check Connectivity"
  hosts: "nokia_srlinux"
  vars_files:
```

```
- srl_secret.yml (1)
  tasks:
  - name: "Check reachability with Ansible Ping"
    ansible.builtin.ping:
```

(1) Provide the credentials to connect to the devices by passing in our encrypted variable file that was created previously. In the playbook, `vars_file` will be retrieved. Since the full contents of the file are encrypted, Ansible will grab the decryption password from the file you specified in the `ansible.cfg` file under `vault_password_file` and decrypt the contents of the file.

8. Execute the playbook to validate connectivity to the devices:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_ping.yml
PLAY [Check Connectivity] *****
TASK [Check reachability with Ansible Ping] *****
ok: [srl-01]
ok: [srl-03]
ok: [srl-02]
```

Great! Now that the inventory has been updated and connectivity is validated, the next step is to manage users and authentication options.

Creating and managing users and authentication options

You just executed a playbook that validated reachability to the devices. In this recipe, you will create the same two users we've been using throughout this part of the book. The `ansible-ro` user will be configured as a read-only user, and the **read/write user** will be named `ansible-rw`:

1. Edit the encrypted vars file (`srl_secrets.yml`) that was created in the last section:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-vault edit
srl_secret.yml
... an editor is opened ...
---
... omitted output ...
ansible_rw_password: "ansible123"
ansible_ro_password: "ansiblero123"
```

2. Create a new playbook called `pb_users.yml` that will take care of provisioning the users:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_users.yml
---
- name: "Create New User with Ansible"
  hosts: "nokia_srlinux"
  vars_files:
- srl_secret.yml
  tasks:
- name: "Create Users on Nokia Devices."
  nokia_srlinux.config:
    replace: (1)
    - path: /system/aaa/authentication/
      user[username={{ item }}] (2)
        value:
          password: "{{ lookup('vars') }}_password" (3)
          superuser: "{{ % if 'rw' in item %}true{% else %}
false{% endif %}}" (4)
        save_when: "changed"
    loop: (5)
      - "ansible_rw"
      - "ansible_ro"
```

The numbered references from the output are explained here:

(1) Since the SR Linux device is modeled in YANG (a data modeling language used to model configuration, state, RPCs, and more for network management protocols), the JSON-RPC management interface has standard parameters defined as follows:

`replace`: Replaces the entire configuration within a specific context with the supplied configuration; equivalent to a delete/update.

`update`: Updates a leaf or container with the specified value.

`delete`: Deletes a leaf or container. All children under the parent are removed from the system.

(2) This YANG path is to create the users. The JSON-RPC management interface is modeled on top of the gMNI path convention. This is similar to a configuration stanza or child hierarchy in other vendors. The device supports a vendor-specific YANG model, as well as the OpenConfig YANG models. You will be using the vendor-specific model in this chapter.



Note

The SR Linux YANG explorer can be used to find the specific information. Navigate to <https://yang.srlinux.dev/v24.10.3> and search user to get the details of the model.

(3) The password is set in the `srl_secret.yml` file. You will do a variable lookup to return the password to set for each username.

(4) The superuser status will be determined based on the username. In this case, if `rw` is in the username, the `superuser` tag will be set.

(5) In the Ansible loop list data, you will set both usernames to create, and the variable lookups will return the password that is hashed in `vars_files`.



Note

For more information on the SR Linux YANG data model, check <https://yang.srlinux.dev/>.

3. Next, run the playbook to configure the two users on all SR Linux devices:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_users.yml
PLAY [Create New User with Ansible] *****
TASK [Create Users on Nokia Devices.] *****
changed: [srl-01] => (item=ansible_rw)
changed: [srl-01] => (item=ansible_ro)
... omitted output ...
```

4. Let's ensure the two new users (as shown by (1) and (2) in the following output) were created appropriately by manually logging into the device and viewing the configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ssh admin@srl-01
... omitted output ...
(admin@srl-01) Password:
--{ running }--[ ]--
A:srl-01# info system aaa authentication
... omitted output ...
```

```
user ansible_ro { (1)
    password $y$j9T$d0e306feKXjmx.f.NN7
    superuser false
}
user ansible_rw { (2)
    password $y$j9T$45cee99bdhpf1NYNH/
    superuser true
}
```

Now that you've learned how to manage users for a Nokia SR Linux device using YANG paths, let's continue with our next recipe, which will configure more system options on the devices.

Configuring generic system options on Nokia devices

In this recipe, you will configure the **hostname** and **Domain Name System (DNS)** servers. This follows the same patterns from the other chapters in this section of the book.

This section will demonstrate the usability of YANG data models and how to execute configuration changes with them. This is a change from Arista and Juniper, where you used declarative modules to perform these recipes. In the case of Nokia, they don't offer a declarative module as the YANG paths offer similar flexibility. Let's get started by defining our group variables:

1. The first update you need to make is to create the `groups_vars/all.yml` file. Here, you will set the global defaults for the DNS servers. Even though our inventory only uses Nokia, setting these variables in `all.yml` makes sense as they will be used for all infrastructure:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi group_vars/all.
yml
---
global:
  dns:
    - 8.8.8.8
    - 8.8.4.4
```

This dictionary is the same as in previous chapters. The dictionary key named `dns`, which is nested under the `global` root key, has a list of IP addresses to define the DNS servers to be used by the network devices.

2. Create a new playbook, `pb_basic_config.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_basic_config.
yml
---
- name: "Configure Hostname and DNS"
  hosts: "nokia_srlinux"
  vars_files:
- srl_secret.yml
  tasks:
- name: "Configure the Hostname and DNS Servers"
  nokia_srlinux.config:
    update:
    - path: /system/name/host-name (1)
      value: "{{ inventory_hostname }}" (2)
    - path: /system/dns/server-list (3)
      value: "{{ global['dns'] }}" (4)
    save_when: "changed"
```

The numbered references from the output are explained here:

- (1) The YANG path that points to the hostname of the device. Refer to the previous step to explore this model in the SR Linux YANG explorer.
 - (2) Specify that the value of the hostname should be the name from the inventory.
 - (3) The YANG path to update the DNS server list.
 - (4) The DNS servers that should be configured on the routers. It is important to note that this YANG leaf is expecting the data in a list/array format. This is what `host_vars` are formatted in already, so no post-processing is required.
3. Up until this point, you've been using the default credentials for the Nokia devices to manage the devices. Let's use the new `ansible-rw` credentials you just created for all the remaining configuration changes in this chapter:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-vault edit
srl_secret.yml
... an editor is opened ...
---
... omitted output ...
```

```
ansible_user: "ansible_rw"
ansible_password: "{{ ansible_rw_password }}"
ansible_rw_password: "ansible123"
ansible_ro_password: "ansiblero123"
```

4. Execute the playbook to configure the hostname and DNS servers:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_basic_config.yml
PLAY [Configure Hostname and DNS] *****
TASK [Configure the Hostname and DNS Servers] *****
changed: [srl-01]
changed: [srl-03]
changed: [srl-02]
... omitted output ...
```

5. If you log in to one of the devices, you can see that the configuration changes were successful:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ssh ansible_rw@srl-
01
(ansible_rw@srl-01) Password:
A:srl-01# info system dns server-list
system {
    dns {
        server-list [
            8.8.4.4
            8.8.8.8
        ]
    }
}
--{ running }--[ ]--
```

In the next section, you will configure the interfaces by building on top of the `config_generation` role that you created in *Chapter 4*.

Configuring Nokia interfaces

In this recipe, you will configure the interface description and layer 3 information on our campus fabric routers. You will add the necessary templates to the `config_generation` role. This continues to demonstrate how an Ansible role can speed up a network automation strategy.



Important

The code for the custom Ansible role is stored in https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/roles.

The following diagram and table provide the design that you will be implementing.

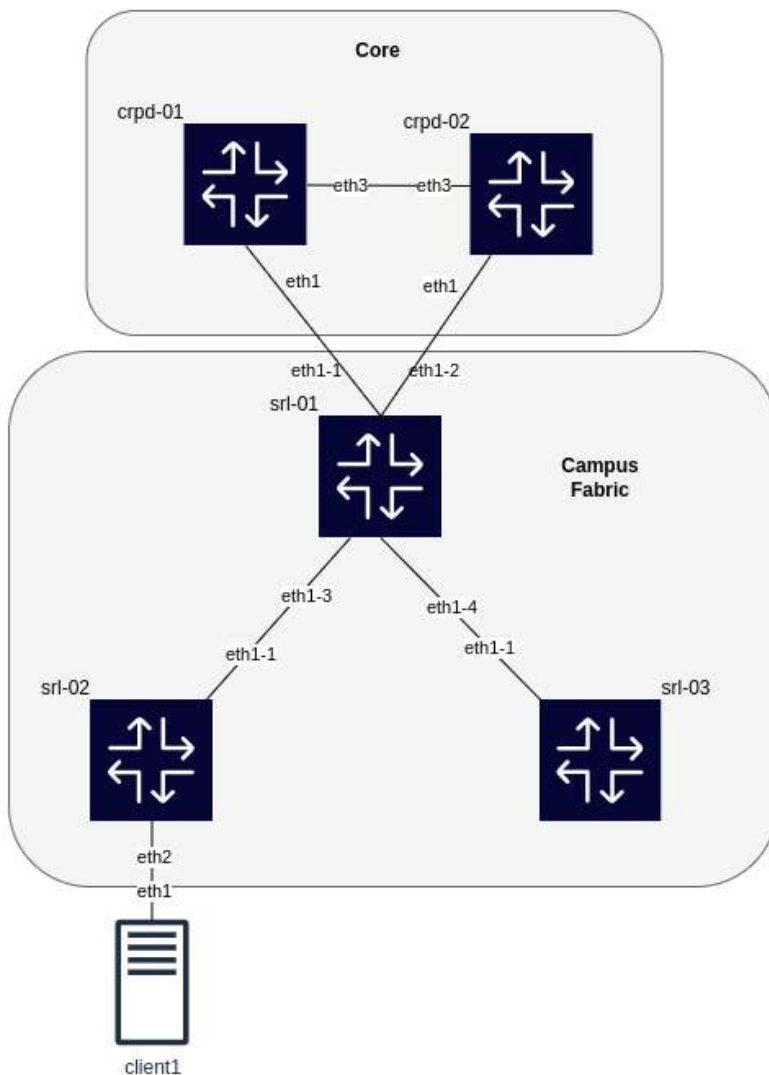


Figure 6.2 – SR Linux campus fabric



 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Note

The campus fabric is going to be based on the L2 EVPN with SR Linux lab exposed at <https://learn.srlinux.dev/tutorials/l2evpn/fabric/>. This guide is a fantastic resource and provides additional details.

Device	Role	Interface	IP Address
srl-01	Spine Switch	mgmt0.0	198.51.100.31/24
srl-01	Spine Switch	ethernet1/1.0	172.31.2.4/31
srl-01	Spine Switch	ethernet1/2.0	172.31.2.6/31
srl-01	Spine Switch	ethernet1/3.0	192.168.11.2/30
srl-01	Spine Switch	ethernet1/4.0	192.168.12.2/30
srl-02	Leaf Switch	mgmt0.0	198.51.100.32/24
srl-02	Leaf Switch	ethernet1/1.0	192.168.11.1/30
srl-02	Leaf Switch	ethernet1/2.0	–
srl-02	Leaf Switch	irb0.111	192.168.0.254/24 (anycast)

Device	Role	Interface	IP Address
srl-03	Leaf Switch	mgmt0.0	198.51.100.33/24
srl-03	Leaf Switch	ethernet1/1.0	192.168.12.1/30
srl-03	Leaf Switch	irb0.111	192.168.0.254/24 (anycast)

Table 6.1 – Interface details



Info

The management IPs are defined by our **Containerlab** topology. The remaining IPs are based on the topology you’ve been building across the last few chapters.

Let’s get started by creating the `host_vars` files to define our interface details per device:

1. Create `host_vars/srl-01.yml` and create the interface data that follows:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/srl-01.yml
---
interfaces:
  ethernet1/1:
    unit: 0 (1)
    ip_address: "172.31.2.4/31"
    description: "SPINE-TO-CORE-CRPD-01-ETH1"
    vrf: "default" (2)
  ethernet1/2:
    unit: 0
    ip_address: "172.31.2.6/31"
    description: "SPINE-TO-CORE-CRPD-02-ETH1"
    vrf: "default"
  ethernet1/3:
    unit: 0
    ip_address: "192.168.11.2/30"
    description: "SPINE-TO-LEAF-SRL-02-e1/1.0"
    vrf: "default"
  ethernet1/4:
```

```

unit: 0
ip_address: "192.168.12.2/30"
description: "SPINE-TO-LEAF-SRL-03-e1/1.0"
vrf: "default"

```

The numbered references from the output are explained here:

- (1) Nokia SR Linux has similar concepts to Juniper units. In Nokia, they're called *subinterfaces* or *indexes*. To key a similar data model as Juniper, you will use the unit key.
 - (2) VRFs in SR Linux are called network instances. Similar to the previous bullet point, keeping the data model the same as other vendors makes templates more maintainable and easy to understand.
2. Now that the data model is defined for the interfaces and IP addresses, let's write the template to use this data and transform the variables into Nokia configuration. As a reminder, the role that you are reusing is expecting the templates directory to exist, and the role will use these templates to generate the configurations:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# mkdir templates &&
cd templates && vi interfaces.json.j2
{
  "interface": [
{% for name, int_details in interfaces.items() %} (1)
  {
    "name": "{{ name }}",
    "description": "{{ int_details['description'] }}",
    "admin-state": "enable",
    "subinterface": [
      {
        "index": "{{ int_details['unit'] | int }}",
        "admin-state": "enable",
        "ipv4": {
          "address": [
            {
              "ip-prefix": "{{ int_details['ip_
address'] | default('') }}"
              {% if 'irb' in name %} (2)
              ,"anycast-gw": true
              {% endif %}

```

```

        }
    ]
    {% if 'irb' in name %} (3)
    {% include "../irb-interfaces.json.j2" %} (4)
    {% endif %}
    }
}
]
}{{ "," if not loop.last else "" }} (5)
{% endfor %}
]
}

```

For this recipe, you will write the configuration templates in JSON format since that is what Nokia accepts:

- (1) Loop through the dictionary you defined previously in `host_vars`, using the `(.items())` method, which allows access to the key-value pairs of the dictionary. The value of `int_details` is an inner dictionary; therefore, our subsequent lookups are dictionary key lookups to get the value.
- (2) If the interface is an **Integrated Routing Bridge (IRB)**, an IRB allows a network device to simultaneously perform Layer 2 bridging and Layer 3 routing, enabling communication between bridged and routed network segments. If an IRB exists, then `anycast true` should be set for the anycast IP address. The *anycast IP* is shared across leafs, and also serves as the default gateway for the servers.
- (3) If the interface is an IRB, then it also requires multiple other ARP-based settings for EVPN and anycast.
- (4) Here, you are including an additional template file specifically for the IRB interfaces. This is a common way to abstract templates, and it can lead to more readable templates, and manageable templates. As a reminder, the Jinja2 `include` statement injects another template at the specified location before rendering.

(5) To make this proper JSON, you need to omit the trailing comma on the last loop iteration.



Note

JSON syntax does not permit a trailing comma on the final item in a list or object. This Jinja2 conditional logic is a standard pattern to ensure we generate valid JSON.

The included `irb-interfaces.json.j2` template is shown here for clarity:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/irb-
interfaces.json.j2
, "arp": { (1)
    "learn-unsolicited": true,
    "proxy-arp": true,
    "host-route": {
        "populate": [
            {
                "route-type": "dynamic"
            }
        ]
    },
    "evpn": {
        "advertise": [
            {
                "route-type": "dynamic"
            }
        ]
    }
},
"anycast-gw": {
    "anycast-gw-mac": "00:00:5E:00:53:00"
```

(1) The `,` before `arp` here is to keep the JSON concatenating into properly encoded JSON.

1. The `config_generation` role expects the entry point file called `srlinux.j2`. Create this file and use the Jinja2 `include` statement to include the `interfaces.json.j2` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
srlinux.j2
{% include "./interfaces.json.j2" %}
```

2. Next, to establish routing, you need to assign routed interfaces to a network instance, also known as a VRF. Create a new file called `network-instance.json.j2`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi network-instance.
json.j2
"network-instance": [
{
  "name": "default",
  "interface": [
{% for name, int_details in interfaces.items() %}
{%   if int_details['vrf'] is defined %}
{%     if int_details['vrf'] == 'default' %}
    {
      "name": "{{ name }}"
    }{{ "," if not loop.last else "" }}
{%   endif %}
{% endif %}
{% endfor %}
  ]
}
```

3. Include the new template in `srlinux.j2`. The comma separation between the two included templates is required to keep the JSON data encoded correctly:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
srlinux.j2
{
{% include "./interfaces.json.j2" %}
,
{% include "./network-instance.json.j2" %}
}
```

4. Create a playbook named `pb_advanced_configs.yml` that will call our role to configure the interface and the network instance using the role and the custom template:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_advanced_configs.yml
---
- name: "BUILD CONFIGS FOR NOKIA"
  hosts: "nokia_srlinux"
  connection: "local"
  roles:
  - "config_generation"
```

5. Finally, execute the playbook. The role will run and save the interface and network-instance configurations output after they've been rendered by Jinja2. Remember that the `config_generation` role dictates `run_once` on the `CREATE OUTPUT CONFIGS DIRECTORY`.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
changed: [sr1-01]
]TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2 TEMPLATE] *****
changed: [sr1-01]
changed: [sr1-02]
changed: [sr1-03]
```

6. View the file located at `output/config/sr1-01.cfg` to view the rendered template. This rendered file is encoded as a JSON payload, which aligns with the YANG data model defined by the vendor. The JSON-RPC management interface will take this payload in as JSON and map it into the YANG configuration model:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# cat output/config/sr1-01.cfg
{
  "interface": [
  {
    "name": "ethernet-1/1",
    "description": "SPINE-TO-CORE-CRPD-01/ETH1",
    "admin-state": "enable",
```

```
    "subinterface": [
      {
        "index": "0",
        "admin-state": "enable",
        "ipv4": {
          "admin-state": "enable",
          "address": [
            {
              "ip-prefix": "172.31.2.4/31"
            }
          ]
        }
      }
    ]
  }
  ... omitted output ...
  "network-instance": [
    {
      "name": "default",
      "interface": [
        {
          "name": "ethernet1/1"
        }
      ]
    }
  ]
  ... omitted output ...
```

Now that you've generated the interface and network instance JSON payload for our campus devices and formatted them according to the Nokia vendor-specific YANG model, into JSON, you're going to build out the other configuration snippets to finalize the full configuration.

Advanced routing configuration using template module

Our campus topology has **one spine switch and two leaf switches**. In *Chapter 4*, you learned that an EVPN/VXLAN fabric consists of an underlay network and an overlay network. In this Nokia fabric example, you will generate the routing configuration, including eBGP for the underlay and iBGP for the overlay.

The following table outlines the BGP **Autonomous System Number (ASN)** for each of our devices in our fabric.

Node	Role	BGP ASN
srl-01	Spine Switch	65200
srl-02	Leaf Switch	65101
srl-03	Leaf Switch	65102

Table 6.2 – BGP ASN details

Now that the topology and BGP routing details have been shown, let's start defining our variables:

1. Update the `host_vars` files for each device to add a loopback, which on SR Linux is defined on the `system0` interface, which is used as the `router-id` in our topology:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/srl-01.
yml
---
interfaces:
... omitted output ...
  system0:
unit: 0
ip_address: "10.20.1.1/32"
description: "LOOPBACK FOR ROUTER ID"
vrf: "default"
```

2. Add the BGP variable data in `host_vars`, which will help you generate the necessary underlay configuration according to the design:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/srl-01.
yml
... omitted output ...
bgp:
  asn: 65200
  peer_groups:
underlay: (1)
  - peer: srl-02
    ip_address: 192.168.11.1 (2)
    remote_as: 65101 (3)
  - peer: srl-03
    ip_address: 192.168.12.1
    remote_as: 65102
```

The numbered references from the output are explained here:

- (1) To make the BGP modeling easier, you will separate the different BGP configuration sections by nesting the data under a peer group. The first peer group for our BGP configuration is specific to our BGP configuration for our underlay network; therefore, the name underlay makes the most sense.
 - (2) For each BGP neighbor that needs to be created, you will need the corresponding neighbor IP address.
 - (3) A required element to create a BGP neighbor is also the remote ASN.
3. Create a new Jinja2 file named `underlay_policy.json.j2`, with the following template data. This template will be used to control BGP advertisements in our fabric. Again, this is coming from the **OpenConfig Routing Policy module**:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
underlay_policy.json.j2
    "routing-policy": {
    "policy": [
        {
            "name": "all",
            "default-action": {
                "policy-result": "accept"
            }
        }
    ]
}
```

4. Next, you will create a new Jinja2 file called `bgp.json.j2`. This template will be the first iteration of the BGP configuration section. It will be updated to add overlay components in the next section:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/bgp.
json.j2
{% for group_name in bgp['peer_groups'] %} (1)
    "autonomous-system": {{ bgp['asn'] }}, (2)
    "router-id": "{{ interfaces['system0'][ 'ip_
address' ] | split('/') | first }}", (3)
    "afi-safi": [
        {
```

```

        "afi-safi-name": "ipv4-unicast",
        "admin-state": "enable"
    }
],
"group": [
    {
        "group-name": "{{ group_name }}", (4)
        "export-policy": [
            "all"
        ],
        "import-policy": [
            "all"
        ]
    }
],
"neighbor": [
    {% for group_name, peer_group_peers in bgp['peer_groups'].
items() %}
    {% for bgp_peer in peer_group_peers %} (5)
    {
        "peer-address": "{{ bgp_peer['ip_address'] }}",
        "peer-as": {{ bgp_peer['remote_as'] }},
        "peer-group": "{{ group_name }}"
    }{{ "," if not loop.last else "" }}
    {% endfor %}
    {% endfor %}
{% endfor %}
]

```

The numbered references from the output are explained here:

- (1) Loop through the peer groups defined in `host_vars`. Each peer group will create the peer group configuration snippets.
- (2) From the current node's `host_vars`, interpolate with the BGP ASN.
- (3) The `router-id` will be using the `system0` interface IP address.
- (4) Assign the `underlay_policy` to the BGP peer group.
- (5) Now that peer group has been created with the required configuration data, each peer has to be created.

5. Before running the playbook to generate the configuration, you have to update the entry point template, `srlinux.j2`. Here, you will include the `underlay_policy.j2` file you created:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
srlinux.j2
{
{% include "./interfaces.json.j2" %}
,
{% include "./network-instance.json.j2" %}
,
{% include "./underlay_policy.json.j2" %}
}
```

6. Next, you will update the `network-instance.json.j2` file to include the BGP template because the BGP configuration is part of the network instance module in the **OpenConfig** model:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
network-instance.json.j2
  "network-instance": [
{
  "name": "default",
  "interface": [
... omitted output ...
  "protocols": {
    "bgp": {
      {% include "./bgp.json.j2" %}
    }
  }
... omitted output ...
```

7. Execute the playbook to generate the full configuration that you've templated out so far:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [sr1-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
```

```
changed: [sr1-01]
changed: [sr1-02]
changed: [sr1-03]
... omitted output ...
```

8. Looking at `sr1-01.cfg` generated configuration, you will see how the variable data and the templates created our underlay network configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi output/config/
sr1-01.cfg
  "protocols": {
    "bgp": {
      "autonomous-system": 65200,
      "router-id": "10.20.1.1",
... omitted output ...
    "neighbor": [
      {
        "peer-address": "192.168.11.1",
        "peer-as": 65101,
        "peer-group": "underlay"
      },
      {
        "peer-address": "192.168.12.1",
        "peer-as": 65102,
        "peer-group": "underlay"
      }
    ]
... omitted output ...
```

Let's continue updating our templates and variable data to create our overlay peer group.

Generating the overlay configuration

In this recipe, you will generate the configuration for the **overlay iBGP EVPN** as the **control plane for VXLAN tunnels** across our leaf-spine fabric in our topology. The iBGP EVPN overlay network uses **internal Border Gateway Protocol (iBGP)** to distribute EVPN control plane information, enabling Layer 2 and Layer 3 connectivity across a Layer 3 underlay, which was established in the previous section.

With the underlay data model defined, the next peer group to define is the overlay peer group:

1. Update `host_vars` and add overlay peers for leafs *only* (e.g., `srl-01` and `srl-03`), because these are the ones the end devices are connected to. For this implementation of a **fabric**, no route-reflector is used; therefore, no overlay BGP peers are configured:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/srl-02.
yml
bgp:
  asn: 65101
  peer_groups:
underlay:
... omitted output ...
overlay: (1)
  - peer: srl-03
    ip_address: 10.20.0.2
    remote_as: 100
```

- (1) Add an additional peer group to the leafs `host_vars` file to specify the iBGP peers between the two leafs.



Note

If you're unfamiliar with BGP route reflectors, you can visit https://infocenter.nokia.com/public/7750SR222R1A/index.jsp?topic=%2Fcom.nokia.Unicast_Guide%2Fbgp_router_refl-ai9exj5yk8.html to get a good overview, specific to Nokia SR Linux.

2. Update the `bgp.json.j2` template to extend its flexibility to conditionally check for underlay versus overlay peer-group naming. Update the template as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/bgp.
json.j2
... omitted output ...
{% if 'overlay' in group_name %} (1)
{
  "group-name": "{{ group_name }}",
  "peer-as": 100,
```

```

    "export-policy": [
        "all"
    ],
    "import-policy": [
        "all"
    ],
    "afi-safi": [
        {
            "afi-safi-name": "evpn", (2)
            "admin-state": "enable"
        },
        {
            "afi-safi-name": "ipv4-unicast",
            "admin-state": "disable"
        }
    ],
    "local-as": {
        "as-number": 100
    },
    "timers": {
        "minimum-advertisement-interval": 1
    }
},
{% endif %}
... omitted output ...

```

The numbered references from the output are explained here:

- (1) If 'overlay' is in the BGP peer group name, add additional configuration.
 - (2) Activate the EVPN address family for the iBGP peers.
3. Rerun the playbook, which will update the generated configuration file and change only what is needed:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [srl-01]

```

```
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [sr1-01]
changed: [sr1-02]
changed: [sr1-03]
... omitted output ...
```

4. Finally, look at the sr1-02 configuration. Notice the changes to the BGP configuration, which adds all the overlay components:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# cat output/config/
sr1-02.cfg
... omitted output ...
  "protocols": {
    "bgp": {
... omitted output ...
      "group": [
        {
          "group-name": "overlay",
          "peer-as": 100,
... omitted output...
        "neighbor": [
          {
            "peer-address": "192.168.11.2",
            "peer-as": 65200,
            "peer-group": "underlay"
          }
          {
            "peer-address": "10.20.0.2",
            "peer-as": 100,
            "peer-group": "overlay"
          }
        ]
      ]
    }
  }
```

At this point, the configurations for both the underlay and overlay are completed. The next section will add the remaining VXLAN components.

Generating the remaining VXLAN configuration

In this final recipe, you will add the remaining configuration to the templates:

1. First, you will generate the configuration for the single interface between `sr1-02`(eth1-2) and `client1` (eth1). These same configuration snippets would be added for any host devices. In our case, you only have one host in this example topology:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/sr1-02.
yml
---
interfaces:
... omitted output ...
  ethernet-1/2:
    (1)
  unit: 0
  description: "LEAF-TO-CLIENT1-ETH1"
```

(1) For this interface, no `ip_address` attribute is used, which will allow you to conditionally check for and apply bridged configuration versus routed configuration.

2. Let's revisit the `interfaces.json.j2` template and add the conditional checks you need to properly determine if routed or bridged configuration should be used:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
interfaces.json.j2
... omitted output ...
{% if int_details['ip_address'] is not defined %} (1)
  "vlan-tagging": true,
{% endif %}
... omitted output ...
{% if int_details['ip_address'] is defined %} (2)
  "ipv4": {
... omitted output ...
  {% else %} (3)
    "type": "bridged",
    "admin-state": "enable",
    "vlan": {
      "encap": {
        "untagged": {
... omitted output ...
```

The numbered references from the output are explained here:

- (1) If `ip_address` is not defined in the interface attributes, assign the `vlan-tagging` option to `true`.
 - (2) If the `ip_address` attribute is defined on the interface, then generate the IPv4 configuration snippet.
 - (3) If it is not defined, then configure the interface as a bridged interface with no layer 3 (IP) configuration.
3. Next, create a new template called `vxlan_interface.j2` to model all the related data for this data model:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/vxlan_
interface.json.j2
  "tunnel-interface": [
  {
    "name": "vxlan1", (1)
    "vxlan-interface": [
      {
        "index": 1,
        "type": "bridged",
        "ingress": {
          "vni": 1 (2)
        }
      }
    ]
  }
]
```

The numbered references from the output are explained here:

- (1) `vxlan1` is the default interface for Nokia; therefore, we hardcode it. As you have seen before, when the data is not variable, it's simpler to manage to embed it directly in the payload.
- (2) Statically configure the **VxLAN Network Identifier (VNI)** to 1 as you're using untagged mode.

4. Update the entry point `srlinux.j2` template, adding an include statement to add the `vxlan` interface only when the device is a leaf switch:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
srlinux.j2
... omitted output...
{% if inventory_hostname in groups['leafs'] %} (1)
,
{% include "./vxlan-interface.json.j2" %}
{% endif %}
}
... omitted output...
```

(1) Since **VXLAN Tunnel Endpoints (VTEPs)** are only configured on leaf switches, you will only include this template when the host is a leaf switch.

5. Next, you will map the MAC-VRF network instance to the interfaces we just created. You will update the `network-instance.json.j2` template by adding the MAC-VRF network instances. Similar to the previous step, this configuration is only for leaf switches (as shown in (1)):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
network-instance.json.j2
"protocols": {
... omitted output ...
}{% if inventory_hostname in groups['leafs'] %}, (1)
{
  "name": "vrf-1",
  "type": "mac-vrf",
  "admin-state": "enable",
  "interface": [
    {
      "name": "ethernet-1/2.0"
    },
    {
      "name": "irb0.111"
    },
  ],
  "vxlan-interface": [
```

```

        {
            "name": "vxlan1.1"
        }
    ]
}
{% endif %}
]

```

Include this snippet when the host is a leaf switch.



Note

To avoid further changes to our interface data stored in `host_vars`, the interfaces here are hardcoded for brevity.

1. Update `network-instance.json.j2` to add the final VPN/EVPN BGP configuration. This runs only for leaf switches, and this configuration allows leafs to advertise the locally learned MACs to the remote leafs:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi templates/
network-instance.json.j2
... omitted output...
}{% if inventory_hostname in groups['leafs'] %},
{
    "name": "vrf-1",
    ... omitted output...
    "protocols": { (1)
        "bgp-evpn": { (2)
            "bgp-instance": [
                {
                    "id": 1,
                    "admin-state": "enable",
                    "vxlan-interface": "vxlan1.1",
                    "evi": 111
                }
            ]
        },
        "bgp-vpn": { (3)

```

```

    "bgp-instance": [
      {
        "id": 1,
        "route-target": {
          "export-rt": "target:100:111",
          "import-rt": "target:100:111"
        }
      }
    ]
    ... omitted output ...

```

The numbered references from the output are explained here:

- (1) Under the `mac-vrf` (`vrf-1`) that was created previously, add the `protocols` subsection and configure the BGP elements.
 - (2) Provides the configuration required to enable the MAC addresses to be shared over the EVPN address family.
 - (3) BGP-VPN is set up to be able to dictate which route-targets are allowed to be imported and which should be exported.
2. Let's execute the playbook again to regenerate the configuration based on our new templates.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [sr1-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [sr1-01]
changed: [sr1-02]
changed: [sr1-03]
... omitted output ...

```

3. Finally, validate that one of the leaf (`sr1-02`) configurations is fully accurate:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# cat output/config/
sr1-02.cfg
    "name": "vrf-1", (1)
    "type": "mac-vrf",
    "vxlan-interface": [

```



```

    {
      "name": "vxlan1.1"
    }
... omitted output ...
    "tunnel-interface": [
    {
      "name": "vxlan1",
      "vxlan-interface": [
... omitted output ...
      "name": "ethernet-1/2",
      "subinterface": [
        {
          "type": "bridged", (3)
... omitted output ...

```

The numbered references from the output are explained here:

- (1) The leaf switch configuration was generated with `vrf-1` as expected.
- (2) The leaf switch properly generated the `Vxlan1` interface.
- (3) The `ethernet-1/2` interface is properly assigned as a bridge interface.

The last piece of configuration you need to generate is the BGP configuration for the peers that are facing the core, which was configured in *Chapter 5*.

Finalizing BGP configuration to the core

In this recipe, you will generate the BGP configuration that is needed to establish the BGP neighborhood with our core network. You will update the BGP Jinja2 templates from the last section to add the neighbors that face the Juniper core devices.

The following table outlines the BGP **Autonomous System Number (ASN)** for each of the devices you're configuring in this section.

Node	Role	BGP ASN
srl-01	Spine Switch	65200
crpd-01	Core Router	65400
crpd-02	Core Router	65400

Table 6.3 – BGP ASN details

Now that you know the BGP ASN details for our network configuration, you can start defining the variables:

1. Update the variable data to add a new peer group and provide the neighbor details:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi host_vars/sr1-01.yml
... omitted output ...
bgp:
  asn: 65100
  peer_groups:
    ... omitted output ...
core: (1)
  - peer: crpd-01
    ip_address: 172.31.2.5
    remote_as: 65400
  - peer: crpd-02
    ip_address: 172.31.2.7
    remote_as: 65400
```

- (i) Add another peer group named **core** to define neighbors to both Juniper core routers.
2. Now that you've updated the **host_vars** to add the additional **core peer group**, you can run the playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation: CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [sr1-01]
TASK [config_generation : GENERATE CONFIGURATIONS FROM JINJA2
TEMPLATE] *****
changed: [sr1-01]
ok: [sr1-02]
ok: [sr1-03]
... omitted output ...
```

3. Finally, validate that the spine (srl-01) configuration has the core peer group and that it is being activated:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# cat output/config/
srl-01.cfg
... omitted output ...
    {
      "peer-address": "172.31.2.5",
      "peer-as": 65400,
      "peer-group": "core"
    },
    {
      "peer-address": "172.31.2.7",
      "peer-as": 65400,
      "peer-group": "core"
    }
  }
... omitted output ...
```



Note

The BGP peers facing the core will come up now! In *Chapter 5*, you configured the remote side of the BGP neighbor.

Now that this complex configuration scenario is completed, notice how *code reusability* and *reproducibility* were gained by using the Ansible role that was created in *Chapter 4*. This allowed you to focus on only the network configuration as opposed to the Ansible tasks to generate the JSON configuration.

With the final BGP configuration completed, the last step is to deploy the configurations to the devices.

Pushing the configuration to Nokia devices

In this recipe, you will push the configuration you have generated in the previous recipes to the Nokia devices, all in one, in order to correctly provision the devices as per the intended network fabric design:

1. Update `pb_advanced_configs.yml` with a `tasks` section. The tasks will use the `nokia.srlinux.config` module. This module accepts a configuration file encoded in JSON and updates/replaces specific YANG trees based on the data provided:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_advanced_
configs.yml
---
- name: "BUILD CONFIGS FOR NOKIA"
  hosts: "nokia_srlinux"
  connection: "local"
  roles:
- "config_generation"
  vars_files:
- srl_secret.yml
  tasks:
- name: "DEPLOY ADVANCED CONFIG"
  nokia.srlinux.config:
    update: (1)
    - path: / (2)
      value: "{{ lookup('ansible.builtin.template', '{{playbook_
dir}}/templates/srlinux.j2') }}" (3)

```

The numbered references from the output are explained here:

- (1) The update action updates the paths specified without removing the other paths not mentioned, providing an incremental configuration. The SR Linux JSON-RPC interface utilizes the gNMI path convention as explained earlier in this chapter.
 - (2) The Nokia SR Linux config module takes a path, which depicts where the provided (value) data should be placed into the configuration tree. In this example, you will provide all the configuration based on the root path (/) of the configuration data model.
 - (3) The module does not provide an src module argument like the other config modules you've used so far in this book. Instead, you will use a built-in file lookup to provide the generated configuration file to the module.
2. Run the playbook to run the config_generation role, and execute the new task you just created to deploy those configurations.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]TASK [config_generation : GENERATE CONFIGURATIONS FROM

```

```
JINJA2 TEMPLATE] *****
changed: [sr1-01]
changed: [sr1-03]
changed: [sr1-02]
TASK [DEPLOY ADVANCED CONFIG] *****
changed: [sr1-01]
changed: [sr1-03]
changed: [sr1-02]
... omitted output ...
```

3. Validate idempotence and rerun the playbook. Notice no changes were needed on any task that was executed.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_advanced_configs.yml
PLAY [BUILD CONFIGS FOR NOKIA] *****
TASK [config_generation : CREATE OUTPUT CONFIGS DIRECTORY] *****
ok: [ceos-01]TASK [config_generation : GENERATE CONFIGURATIONS FROM
JINJA2 TEMPLATE] *****
ok: [sr1-01]
ok: [sr1-03]
ok: [sr1-02]
TASK [DEPLOY ADVANCED CONFIG] *****
ok: [sr1-01]
ok: [sr1-03]
ok: [sr1-02]
... omitted output ...
```

Now that all the different elements of the configuration have been templated and deployed. The next step is to validate that the configuration you pushed resulted in the expected end state. You will do this by retrieving device facts and running arbitrary show commands on the Nokia devices.

Collecting Nokia device facts using Ansible

In this recipe, you will learn how to use the Nokia get module to retrieve and validate that the intended state aligns with the expected results, for both configuration data and state data:

1. Create a new playbook, `pb_validate_facts.yml`. In this playbook, you will validate that all BGPs are established:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_validate_facts.yml
---
- name: "Collect and Validate Nokia Facts"
  hosts: "nokia_srlinux"
  vars_files:
- srl_secret.yml
  tasks:
- name: "Collect BGP Facts"
  nokia_srlinux.get:
    paths:
    - path: "/network-instance[name=default]/protocols/bgp/
neighbor"
      datastore: state
      yang_models: srl
      register: bgp_information
- ansible.builtin.set_fact:
  bgp_neighbors: "{{ bgp_
information['result'] | map(attribute='neighbor') | flatten }}" (1)
  ansible.builtin.assert:
    that: "{{ item['session-state'] == 'established' }}"
    quiet: true
  loop: "{{ bgp_neighbors }}"
  loop_control:
    label: "{{ item['peer-address'] }} is established"

```

(1) Here, you will set a **fact** to be reused in the next task. In this task, you will take the BGP information result from the facts module and pull out all the neighbors from the output. The `bgp_information['result']` is a list of dictionaries data structure. By utilizing the `map` filter, you can pull out a specific key (`neighbor`) for each dictionary object in the list. Lastly, the `flatten` filter can take a list of lists and remove unnecessary nesting from the object – for example, `[[1],[2]]` becomes `[1,2]`.

(2) Then, run the `pb_validate_facts.yml` playbook to validate the operational state of our fabric:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_validate_facts.yml
PLAY [Collect and Validate Nokia Facts] *****
TASK [Collect BGP Facts] *****

```

```

ok: [sr1-01]
ok: [sr1-02]
ok: [sr1-03]
TASK [ansible.builtin.set_fact] *****
ok: [sr1-02]
ok: [sr1-01]
ok: [sr1-03]
TASK [Validate BGP Neighbor Exists] *****
ok: [sr1-02] => (item=10.20.0.2 is established)
... omitted output ...
ok: [sr1-03] => (item=192.168.12.2 is established)

```

This execution asserts that the BGP neighbors are all established. If you have not completed *Chapters 4 and 5* before this step, some of the BGP neighbors will remain down (*Idle, Active*).



Info

For more information on the `get` module for Nokia, see <https://learn.srlinux.dev/ansible/collection/#modules>.

In this recipe, you executed commands on the Nokia devices and validated that the operational state of the BGP neighbors was achieved. In the final section of this chapter, you will learn how to run ad hoc CLI commands against the devices.

Running arbitrary commands on Nokia devices

In this recipe, you will execute operational commands on Nokia devices and use the output to validate that the `client1` server is online and that the routes/MAC address are being learned across the fabric. This recipe differs from the previous use of the Nokia `get` module, simply to show flexibility and how to gather information using a more common CLI interface that a network engineer is familiar with:

1. Create a new playbook called `pb_get_mactable.yml` and populate it to execute the `show network-instance vrf-1 bridge-table mac-table all` command on all leaf switches and store the output in a variable:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# vi pb_get_mactable.
yml
---
```

```

- name: "Retrieve Learned MACs from Leaf Switches"
  hosts: "leafs" (1)
  vars_files:
- srl_secret.yml
  tasks:
- name: "Learned Mac-Table for vrf-1"
  nokia.srlinux.cli:
    commands: "show network-instance vrf-1 bridge-table mac-
table all" (2)
    register: show_macs
- ansible.builtin.set_fact:
    mac_table: "{{ show_macs['result'][0]['Network'] }}"
- name: "Assert Mac-Table is Accurate"
  ansible.builtin.assert:
    that: item['Address'] == '00:C1:AB:00:00:01' (3)
    when: mac_table[0]['Mac table'] is defined
    loop: "{{ mac_table[0]['Mac table'] }}"
    delegate_to: localhost

```

The numbered references from the output are explained here:

- (1) Delegate this play to only run on hosts in the leafs inventory group.
 - (2) Use the `nokia.srlinux.cli` module to run an arbitrary command. Run `show network-instance vrf-1 bridge-table mac-table all` and the Nokia CLI module by default returns JSON data.
 - (3) Using the `assert` module, you will validate that the MAC address if `client1` is being learned on both leaf switches.
2. In order to ensure our configuration and routing are working as intended, start by initiating some traffic from `client1` to ensure that an **Address Resolution Protocol (ARP)** occurs and `sr1-02` learns the MAC address. This validates that our overlay routing is sharing routes as expected:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# docker exec -it
client1 ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
From 192.168.0.1 icmp_seq=1 Destination Host Unreachable
From 192.168.0.1 icmp_seq=2 Destination Host Unreachable

```



```

From 192.168.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 192.168.0.2 ping statistics ---
4 packets transmitted, 0 received, +3 errors, 100% packet loss, time
3055ms

```

Important



The PING does not need to succeed; you just need traffic to be initiated. Any traffic that originates from the client device will send an ARP broadcast to find its default gateway for layer 3 traffic to be routed. This ARP process will learn the MAC address of the client, and therefore the MAC address table is populated with the entry.

3. Execute the playbook to validate that the VLANs are configured properly on the leaf switches:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch06# ansible-playbook
pb_get_mactable.yml
PLAY [Retrieve Learned MACs from Leaf Switches] *****
TASK [Learned Mac-Table for vrf-1] *****
ok: [sr1-03]
ok: [sr1-02]
TASK [ansible.builtin.set_fact] *****
ok: [sr1-02]
ok: [sr1-03]
TASK [Assert Mac-Table is Accurate] *****
ok: [sr1-02 -> localhost] => (item={'Address':
... omitted output ...
},
  "msg": "MAC:'00:C1:AB:00:00:01' is learned in the Mac Table"
}
ok: [sr1-03 -> localhost] => (item={'Address':
... omitted output ...
},
  "msg": "MAC:'00:C1:AB:00:00:01' is learned in the Mac Table"
}
... omitted output ...

```

The playbook execution validates that the `client1` MAC address is being learned locally on `sr1-02`, but also that the MAC address is being learned via an EVPN type 2 route on `sr1-03` via the iBGP (overlay) neighbor that was created earlier in this chapter.

Now that you've learned two methods for getting data off Nokia devices and using the `assert` module to validate the state, you're armed with a key skill to be able to validate a configuration change.

Summary

In this chapter, you have set up SR Linux Nokia devices to prepare for Ansible automating configurations. You also extended the Ansible role that generates configurations, this time using JSON data. With the campus fabric now online and connected to the core, the entire network scenario first introduced in *Chapter 2* is complete and operational. You have successfully automated an Arista data center, a Juniper core, and a Nokia campus using a consistent, reusable, and data-driven automation strategy. Next, you pushed configurations and validated the operational status after the change. Finally, you ensured that the client server's MAC address is being learned across the fabric via EVPN routes.

Coming up in the final chapter of this section, you will learn how to use NAPALM to administer a multi-vendor network.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



7

Administering a Multi-Vendor Network with NAPALM and Ansible

Network Automation and Programmability Abstraction Layer with Multivendor support (NAPALM), as the name implies, is an open source, multi-vendor Python library intended for interaction with different network vendor equipment. It provides a consistent method to interact with all these devices, irrespective of the vendor equipment used.



Note

For more information about NAPALM support, check out the following page:
<https://napalm.readthedocs.io/en/latest/support/index.html>.

In the previous chapters, you interacted with different network devices using Ansible modules. As you built the playbooks in those chapters, you saw that the data returned from each vendor-specific module can be different. This is one of the main problems that NAPALM tries to solve. NAPALM attempts to provide an abstracted and consistent API to interact with multiple vendor OSs. The data returned by NAPALM from these different vendor OSs is normalized and consistent.

NAPALM interacts with each device according to the most common interface supported by the node, and the interface that is widely adopted by the community (e.g., CLI, HTTP APIs, etc.). The following diagram outlines how NAPALM interacts with the most common network devices, as well as the libraries used in NAPALM to interact with the APIs on these devices:

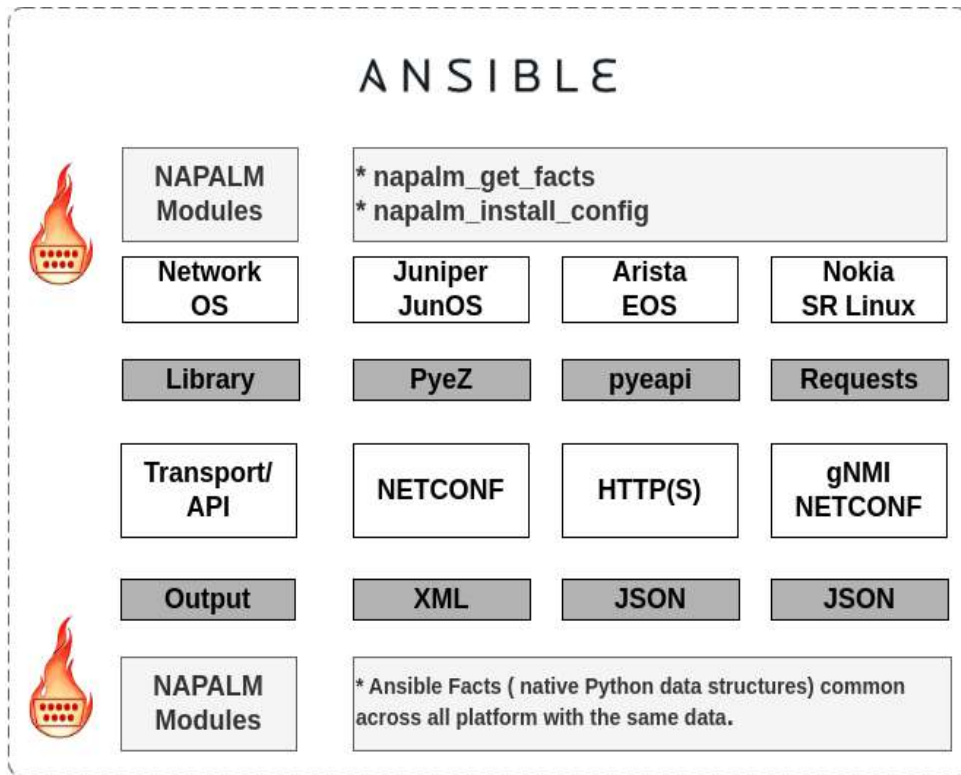


Figure 7.1 – NAPALM Ansible architecture

Since NAPALM tries to provide a consistent method to interact with network equipment, it supports a limited set of vendor devices that have been adapted. NAPALM also supports the most common tasks that are carried out on these devices, such as device configuration, retrieving the operational state for interfaces, implementing **Border Gateway Protocol (BGP)** and **Link Layer Discovery Protocol (LLDP)**, and many others.

Even though NAPALM offers an abstract API that is supported across many different platforms, it's important to understand that abstraction is hard, and keeping up with these community drivers over a long period of time is a real challenge. NAPALM supports many vendors—Cisco,

Arista, Juniper, Nokia, and so on—across many platforms, but it’s not always possible to keep all these open source/community-developed drivers working. It’s always important to test and, if possible, contribute to these projects to keep them successful and working!

In this chapter, you will learn how to automate a multi-vendor network using NAPALM and Ansible. You will manage the configuration of these different vendor OSs, as well as retrieving the operational state from these devices. Throughout the previous chapters, you configured the entire lab topology. To demonstrate the NAPALM configuration module, you will add NTP servers to the configuration. Remember that the lab is available using the instructions in *Chapter 16*.

The recipes covered in this chapter are the following:

- Installing NAPALM with Ansible
- Retrieving network facts with NAPALM
- Retrieving BGP neighbor information from network devices
- Configuring network devices with NAPALM
- Validating and auditing the network with NAPALM
- Building reports with NAPALM and Ansible

Installing NAPALM with Ansible

In this recipe, you will install the NAPALM library and integrate it to work with Ansible. This task is mandatory since NAPALM Ansible modules are not part of the community modules that are shipped with Ansible by default. So, in order to start working with these modules, you need to install NAPALM and all of its Ansible modules. Then, you need to inform Ansible of where to find the NAPALM Python package and start working with the specific modules developed by the NAPALM team for Ansible.

Let’s get started by getting a fresh copy of our inventory file from *Chapter 6* and then proceeding with the NAPALM installation:

1. First, create the `ch07` directory and copy over the inventory that you augmented in *Chapter 6*:

```
root@cookbook-lab:~/cookbook-lab/chapters# mkdir ch07 && cp ch06/
hosts.yml ch07/hosts.yml && cd ch07/
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button

(1) to quickly copy code into your coding environment, or click the **Explain** button

(2) to get the AI assistant to explain a block of code to you.



The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



2. Next, let's start the installation of NAPALM. In order for NAPALM to work, there are two requirements. The first is installing the Python NAPALM library by using pip. Note that although the package is named `napalm-ansible`, it is a Python library and not an Ansible collection/role:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# pip install napalm-  
ansible  
Collecting napalm-ansible  
  Downloading napalm-ansible-1.1.0.tar.gz (13 kB)  
  Preparing metadata (setup.py) ... done  
Collecting napalm>=3.0.1  
... omitted output ...
```

3. Now that the Python dependency has been installed, the second requirement is to install the **NAPALM Ansible collection** from **Ansible Galaxy**:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-galaxy
collection install napalm.napalm
Starting galaxy collection install process
Process install dependency map
Starting collection install process
Downloading ... omitted output ...
Installing 'napalm.napalm:0.9.13' to '/root/.ansible/collections/
ansible_collections/napalm/napalm'
napalm.napalm:0.9.13 was installed successfully
```



Note

To learn more about Ansible Galaxy, revisit *Chapters 2 and 3*.

4. One additional step is to let Ansible know where to find the NAPALM modules that you just installed. You can provide the path in the `ansible.cfg` file. Copy the `ansible.cfg` file from the last chapter and add the two new configuration options to the end:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# cp ../ch06/ansible.
cfg . && vim ansible.cfg
[defaults]
... omitted output ...
library = /root/miniconda/lib/python3.12/site-packages/napalm_
ansible/modules
action_plugins = /root/miniconda/lib/python3.12/site-packages/
napalm_ansible/plugins/action
```

The library and action plugins that NAPALM Ansible exposes are directly configured for Ansible to be able to discover them.



Note

If you need to find where the library is installed outside the lab environment in this book, you can run the following command:

```
pip show napalm-ansible | grep Location
```


1. Now, validate the installation of the previous two steps:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# pip show napalm-ansible
Name: napalm-ansible
Version: 1.1.0
Summary: Network Automation and Programmability Abstraction Layer with Multivendor support
... omitted output ...
```

Next, validate that the NAPALM Ansible collection is installed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-galaxy collection list | grep napalm
napalm.napalm                                0.9.13
```

2. Now that the inventory hosts.yml file has been copied into ch07 and you've validated that NAPALM is installed successfully, let's move on to validating our inventory and connection options for all three vendor OSs.

Note



To summarize, the pip package installs the core NAPALM Python logic, an **Ansible collection** provides the integration modules (`napalm_get_facts`, etc.), and `ansible.cfg` paths are needed to tell Ansible where to find the non-standard modules that don't live in a collection's default path.

NAPALM uses a specific transport mechanism for each supported vendor equipment. It uses the API in order to connect to the device. For our sample topology, we need the following:

- NETCONF for Juniper (configured in *Chapter 5*)
- JSON-RPC for Nokia (configured in *Chapter 6*)
- Instead of using the eAPI (configured in *Chapter 4*) for Arista, you will use SSH

This demonstrates the flexibility of platforms that offer multiple connection types.

As a final step to prepare for executing playbooks, you will update the `hosts.yml` file, adding the required variables:

1. Update the `hosts.yml` file with the Arista connection type and network OS definitions:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim hosts.yml
... omitted output ...
arista_ceos:
  vars: (1)
    ansible_network_os: "arista.eos.eos"
    ansible_connection: "ansible.netcommon.network_cli" (2)
    ansible_user: "ansible" (3)
    ansible_password: "ansible123" (3)
... omitted output ...
```

The numbered references from the output are explained here:

- (1) vars have been created for Arista.
- (2) `ansible_connection` was changed from `httpapi` (eAPI) to `network_cli` (SSH).
- (3) The default credentials were also added.

2. Update the `hosts.yml` file for Juniper vars with our SSH key that was created in *Chapter 5*:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim hosts.yml
... omitted output ...
juniper_crpdp:
  vars:
    ansible_network_os: "junipernetworks.junos.junos"
    ansible_connection: "ansible.netcommon.netconf" (1)
    ansible_user: ansible
    ansible_ssh_private_key_file: ../ch05/Ansible_SSH_key (2)
... omitted output ...
```

The numbered references from the output are explained here:

- (1) Juniper will continue to use NETCONF, which you set up in *Chapter 5*.
- (2) Simply change the `ansible_ssh_private_key_file` variable out of `group_vars` and into the inventory file from the previous chapter to keep everything simple for our current inventory.

3. Next, update the Nokia group with the credentials as well:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim hosts.yml
... omitted output ...
vars:
  ansible_network_os: "nokia.srlinux.srlinux"
  ansible_connection: "ansible.netcommon.httpapi"
  ansible_user: "ansible"
  ansible_password: "ansible123"
... omitted output ...
```

Now, all three of our vendor groups (Arista, Juniper, and Nokia) have their required Ansible variables set in a single place to simplify the management for this chapter!

4. Execute the Ansible ping module to ensure connectivity is established and working:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible network_
equipment -m ansible.builtin.ping
ceos-01 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
crpd-01 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
srl-01 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
... omitted output ...
```

With everything for the inventory and NAPALM installation ready, let's retrieve some operational information from our devices to show NAPALM's power!

Retrieving network facts with NAPALM

In this recipe, you will collect operational data from network devices using the NAPALM fact-gathering module. This can be used to validate the current network state across multi-vendor equipment since NAPALM Ansible's fact-gathering module returns a consistent data structure across all vendor OSs supported by NAPALM.

Here, you will pull interface facts for our Nokia and Arista devices:

1. Create a playbook named `pb_napalm_get_facts.yml` with the following content:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_get_facts.yml
---
- name: "Collect Network Facts using NAPALM"
  hosts: "arista_ceos,nokia_srlinux" (1)
  gather_facts: False
  tasks:
    name: "Collect Interfaces NAPALM Facts"
    napalm_get_facts:
      optional_args: "{{ optional_args }}" (2)
      dev_os: "{{ dev_os }}" (3)
      filter:
        - interfaces (4)
```

The numbered references from the output are explained here:

- (1) How to execute a playbook against multiple different groups in the same playbook.
- (2) Set `optional_args` to the dictionary defined in the `group_vars` of the host. NAPALM uses optional arguments to help the backend library set additional attributes to help determine connection types, public keys, timeouts, and more.
- (3) Set the correct NAPALM device driver name. NAPALM has its own driver name definition, since the project is built on top of community drivers. **NetUtils** provides a mapping to attempt to map all the drivers in a single place.
- (4) Filter the NAPALM getter functionality down to just interface details. If no filter is provided, the module will gather all the facts for the device.

Note



The **NetUtils Python library** has many network automation framework mappings. See https://netutils.readthedocs.io/en/latest/user/lib_use_cases_lib_mapper/#napalm-mapper for a list of available NAPALM driver names.

2. NAPALM manages its own connection attributes using an `optional_arguments` key. NAPALM also uses its own value for network drivers that don't directly align with Ansible network OS definitions. Because of this, you will create `group_vars` for each platform type to define these extra requirements:

Create `group_vars/arista_ceos.yml`. Define the NAPALM `dev_os` and set the optional arguments to use SSH:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim group_vars/
arista_ceos.yml
---
dev_os: "eos"
optional_args: {"transport": "ssh"}
```

Create `group_vars/juniper_crpd.yml`. Define the NAPALM `dev_os` and set the optional arguments to use the SSH private key for authentication:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim group_vars/
juniper_crpd.yml
---
dev_os: "junos"
optional_args: {"use_keys": True, "key_file": "../ch05/Ansible_SSH_
key"}
```

Create `group_vars/nokia_srlinux.yml`. Define the NAPALM `dev_os` and keep the optional arguments empty as this platform doesn't require any special options:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07#
---
dev_os: "sr1"
optional_args: {}
```

3. Now, you can run the playbook and see that the task runs successfully:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-playbook
pb_napalm_get_facts.yml
PLAY [Collect Network Facts using NAPALM] *****
TASK [Collect Interfaces NAPALM Facts] *****
ok: [ceos-01]
ok: [ceos-02]
ok: [sr1-01]
```

```
ok: [sr1-02]
ok: [ceos-03]
ok: [sr1-03]
```

4. Next, you will update the playbook and debug the registered variable. This ensures that the JSON interface data is being returned for all the devices:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_get_facts.yml
---
- name: "Collect Network Facts using NAPALM"
  hosts: "arista_ceos,nokia_srlinux"
  gather_facts: False
  tasks:
    - name: "Collect Interfaces NAPALM Facts"
      napalm_get_facts:
        optional_args: "{{ optional_args }}"
        dev_os: "{{ dev_os }}"
        filter:
          - interfaces
      register: interfaces
    - ansible.builtin.debug:
        var: interfaces
```

With the debug task added, you can now run the playbook again and see the JSON outputs with the interface details.

5. Execute the playbook again and investigate the outputs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-playbook pb_napalm_get_facts.yml
PLAY [Collect Network Facts using NAPALM] *****
TASK [Collect Interfaces NAPALM Facts] *****
ok: [ceos-01]
... omitted output ...
  "mgmt0": { (1)
    "description": "",
    "is_enabled": true,
    "is_up": true,
    "last_flapped": 1741146878.0,
```

```

        "mac_address": "7A:A4:FD:9F:0C:2B",
        "mtu": 1514,
        "speed": 1.0
    },
    ... omitted output ...

```

The numbered reference from the output is explained as follows:

- (1) One of the interfaces from the ceos-01 device, showing the dictionary schema you'd expect to see for every interface on all devices.

Now that you can retrieve interface facts from network devices using NAPALM, you will expand this by creating and running an additional playbook that retrieves BGP neighbor information from the Arista and Juniper devices.

Retrieving BGP neighbor information from network devices

In this recipe, you will retrieve the **BGP neighbor facts**. This time, you will take things a step further by analyzing the returned structured data and asserting that all BGP neighbors are in the established state:

1. Create a new playbook named `pb_napalm_bgp.yml`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_bgp.
yml
---
- name: "Collect Network Facts using NAPALM"
  hosts: "arista_ceos,juniper_crpd" (1)
  gather_facts: False
  tasks:
    - name: "Collect BGP NEIGHBOR NAPALM Facts"
      napalm_get_facts:
        optional_args: "{{ optional_args }}"
        dev_os: "{{ dev_os }}"
        filter:
          - bgp_neighbors (2)
      register: bgp_info
      name: "Assert all BGP Neighbors are established"
      ansible.builtin.assert: (3)

```

```

    that:
      - item.value['is_up'] is true (4)
    quiet: true
    loop: "{{ bgp_info['ansible_facts']['napalm_bgp_neighbors']
['global']['peers'] | dict2items }}" (5)
    loop_control: (6)
    label: "{{item.key}} is_up: {{ item.value['is_up'] }}"

```

The numbered references from the output are explained here:

- (1) Limit the playbook to hosts defined within the comma-separated groups to Arista and Juniper.
 - (2) Filter the NAPALM facts down to only the BGP neighbor information.
 - (3) Use the built-in Ansible assertion module to validate that the returned data conforms to the test case specified.
 - (4) Assert that each neighbor's `is_up` dictionary key is set to true.
 - (5) Loop through the returned BGP neighbor details specifically for the global BGP network instance.
 - (6) Utilize the loop control label to make the outputs more consumable to the playbook executor.
2. As a final step, execute the playbook to see whether all the BGP peers in the global VRF/network instance are up (if you have gone through all the previous chapters and provisioned the network accordingly):

```

root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_bgp.
yml
... omitted output ...
TASK [Assert all BGP Neighbors are established] *****
ok: [ceos-01] => (item=10.100.1.1 is_up: True)
ok: [ceos-02] => (item=10.100.1.254 is_up: True)
ok: [crpd-01] => (item=172.31.1.4 is_up: True)
ok: [ceos-03] => (item=10.100.1.254 is_up: True)
ok: [crpd-02] => (item=172.31.1.6 is_up: True)
... omitted output ...

```


This is a fantastic step forward in verifying the operational state of your network, not to mention the added simplicity of using NAPALM, where the returned structured data follows the same schema across different vendor OSs. This is ultimately one of the largest benefits of using NAPALM in a multi-vendor environment. In the next section, you will learn how to use NAPALM to add configuration to your devices. Later in this chapter, you will also explore how to use a NAPALM module called `validate`, which is a more sophisticated way to validate the state of your network.

Configuring network devices with NAPALM

In this recipe, you will push configurations to different vendor devices using Ansible and NAPALM. NAPALM provides a single Ansible module for configuration management, and this module allows you to use a single common method to push any configuration on any vendor equipment supported by NAPALM, greatly simplifying Ansible playbooks.

Since your topology is already configured from the last three chapters, you will configure an additional system-level configuration here to simplify and learn how to utilize NAPALM to push configuration in a multi-vendor environment. Let's get started and configure time-related services on all three platform types:



Note

In this chapter, you will *not* reuse the configuration generation role created in *Chapter 4*; you will create a simplified playbook that can render configurations. This will ensure you learn multiple ways to accomplish similar tasks.

1. Create `pb_napalm_time.yml` and fill in the following contents, including the NTP variables that you will use for the configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_time.
yml
---
- name: "CONFIGURE NTP USING NAPALM"
  hosts: "network_equipment"
  gather_facts: False
  vars:
    ntp: (1)
    server:
      0.us.pool.ntp.org
```

```

1.us.pool.ntp.org
2.us.pool.ntp.org
tasks:
  name: "CREATE CONFIGS DIRECTORY"
  ansible.builtin.file: (2)
    path: "configs"
    state: "directory"
    run_once: true
- name: "GENERATE NTP CONFIGURATION TO PUSH"
  ansible.builtin.template: (3)
    src: "{{ ansible_network_os | split('.') | last }}.j2"
    dest: "./configs/{{ inventory_hostname }}.cfg"
    name: "INSTALL NTP CONFIG AND SAVE DIFF"
  napalm_install_config: (4)
    optional_args: "{{ optional_args }}"
    dev_os: "{{ dev_os }}"
  (5) config_file: "./configs/{{ inventory_hostname }}.cfg"
    commit_changes: true (6)
    replace_config: false (7)
    get_diffs: false (8)

```

The numbered references from the output are explained here:

- (1) The variables for the NTP servers that you will be configuring in this recipe. They are set in the playbook to demonstrate the flexibility of variable declaration in Ansible.
- (2) Create a directory in `ch07/` called `configs`. This is where the generated configs will be stored once they're rendered by Jinja2. Set `run_once` to make sure you don't unnecessarily run this multiple times.
- (3) Run the Ansible built-in module to render templates. This module expects a source file, which in our case is a Jinja2 template, and a destination that the rendered template is saved into.
- (4) Use the NAPALM `install_config` module to configure all three of the vendor OSs in our topology.
- (5) The generated config to provision the device with.
- (6) Set `commit_changes` to `true`. This argument is used to force the saving of the configuration.

(7) NAPALM modules support both a full replace and a merge-based configuration deployment option. Setting `replace_config` to `false` ensures you only merge the new configuration into the system instead of a full replace, which would override the previous configuration on the device.

(8) Finally, setting `get_diffs` to `false` simply disables the NAPALM configuration diff that gets displayed.

2. Create a templates directory and add a new Jinja2 template named `eos.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# mkdir templates &&
cd templates && vim eos.yml

{% for server in ntp["server"] %}
ntp server {{ server }}
{% endfor %}
clock timezone America/Denver

Arista's NTP configuration follows a familiar pattern, similar to
Cisco's syntax. You will loop through the three NTP servers defined
in our playbook vars, and also set the timezone to America/Denver.
Next, create a new template named junos.yml:
root@cookbook-lab:~/cookbook-lab/chapters/ch07/templates# vim junos.
yml

set system time-zone America/Denver
```

For Juniper, you will *only* set the timezone; the NTP servers will not be configured because the Juniper CRPD image doesn't support configuring NTP servers.

Note



If instead of the cRPD network OS you were using a standard Junos OS, you could use the following configuration command:

```
set system ntp server {{ server }}
```

1. The final Jinja2 template you will create is for the Nokia devices. Create a file named `srlinux.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07/templates# vim
srlinux.yml
```

```
{% for server in ntp["server"] %}
set system ntp admin-state enable network-instance default server {{
server }}
{% endfor %}
set system clock timezone America/Denver
```

For Nokia, you will use the Nokia set formatting instead of the JSON formatting you used in *Chapter 6*. This shows the flexibility of the Nokia system. First, you configure each of the three NTP servers, followed by the timezone.

2. Finally, you can execute the playbook and see all the tasks get executed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07/templates# cd .. &&
ansible-playbook pb_napalm_time.yml

PLAY [CONFIGURE NTP USING NAPALM] *****
TASK [CREATE CONFIGS DIRECTORY] *****
changed: [ceos-01]
TASK [GENERATE NTP CONFIGURATION TO PUSH] *****
changed: [crpd-01]
changed: [ceos-03]
... omitted output ...
changed: [sr1-02]
TASK [INSTALL NTP CONFIG AND SAVE DIFF] *****
changed: [crpd-01]
changed: [ceos-03]
changed: [sr1-02]
... omitted output ...
```

Notice that the playbook ran the directory creation only once, followed by creating the configuration file for every device in our topology. Finally, the configuration is pushed via the merge mechanism into the devices using NAPALM.

Now that you've pushed the configuration using a single Ansible module to three different vendor OSs, you can see the power of NAPALM. Managing multi-vendor environments is difficult; being able to accomplish configuration changes without needing to install, configure, and use vendor-specific modules streamlines this process.

In the next section, you will use another great NAPALM module, named `validate`. This module can get data from network devices and validate the contents against a golden state defined in a YAML file.

Validating and auditing a network with NAPALM

In this recipe, you will utilize NAPALM to validate the rest of the BGP configuration. As mentioned in the previous section on retrieving BGP neighbor information, the NAPALM module called `validate` has a more streamlined way to validate facts from a device. It retrieves facts from devices on demand using the NAPALM getter functionality and compares them against an intended-state YAML file.

For this recipe, you will validate the BGP information for the Arista and Juniper devices, but the same pattern could be followed for Nokia or any other vendor OS that has a NAPALM driver:

1. Create a new playbook called `pb_napalm_validation.yml`, and add the following content:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim pb_napalm_validation.yml
---
- name: "VALIDATING NETWORK STATE VIA NAPALM"
  hosts: "arista_ceos,juniper_crpd"
  tasks:
    - name: "VALIDATION WITH NAPALM"
      napalm_validate:
        optional_args: "{{ optional_args }}"
        dev_os: "{{ dev_os }}"
        validation_file: "napalm_validate/
                        {{ inventory_hostname }}.yml" (1)
      ignore_errors: false
```

The numbered reference from the output is explained here:

- (1) Points the `napalm_validate` module to the specific YAML file. This YAML file tells NAPALM what elements should be checked.
2. Create a new directory called `napalm_validate`. This directory will store the golden/intended state data that we want to compare against:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# mkdir napalm_validate
```

3. Create a YAML validation file for every Juniper and Arista device following the pattern specified in the following `ceos-01.yml` file. Each file in this directory should be named `{ { inventory_hostname } }.yml`. This file will be used to validate the operational state against the intended state:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# vim napalm_validate/
ceos-01.yml
---
- get_bgp_neighbors:
    global:
        router_id: 10.100.1.254
    peers:
        10.100.1.1:
            is_enabled: true
            is_up: true
            local_as: 65100
            remote_as: 65001
            remote_id: 10.100.1.1
... omitted output ...
```

This file will be populated with every BGP peer. The module will then validate each peer's state, local ASN, remote ASN, and remote ID (router ID).

4. Run the validation playbook and notice that all our BGP information is accurate compared to the validation file we created. This ensures the actual/operational state matches the intended state defined in our validation files:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-playbook
pb_napalm_validation.yml
ansible-playbook pb_napalm_validation.yml
PLAY [VALIDATING NETWORK STATE VIA NAPALM] *****
TASK [VALIDATION WITH NAPALM] *****
ok: [ceos-01]
ok: [ceos-02]
ok: [crpd-02]
ok: [crpd-01]
ok: [ceos-03]
... omitted output ...
```

Now that the BGP operational data is fully validated, in the last section, you will learn how to create reports based on the data.

Building reports with NAPALM and Ansible

In this recipe, you will create reports based on the data you can retrieve using NAPALM. It is common practice in enterprises to enforce the requirement to have pre-checks and post-checks in a change window. This is especially true with configuration changes for network devices. This recipe provides a simplified way to provide this.

You will reuse the playbook that was created in the previous section as a starting point:

1. Copy the `pb_napalm_validation.yml` playbook to `pb_napalm_compliance.yml` and add the remaining tasks to create the folders and reports:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# cp pb_napalm_
validation.yml pb_napalm_compliance.yml && vim pb_napalm_compliance.
yml
... omitted output ...
  register: net_validate
- name: "CREATE COMPLIANCE REPORT FOLDER"
  ansible.builtin.file: (1)
    path: "compliance_folder"
    state: "directory"
- name: "CLEAN LAST COMPLIANCE REPORT"
  ansible.builtin.file: (2)
    path: "compliance_folder/{{inventory_hostname}}.txt"
    state: "absent"
- name: "CREATE COMPLIANCE REPORT"
  ansible.builtin.copy: (3)
    content: "{{ net_validate.compliance_report | to_nice_
yaml }}" (3)
    (4) dest: "compliance_folder/{{ inventory_hostname }}.txt"
```

The numbered references from the output are explained here:

- (1) Use the built-in module named `file` to ensure the new directory named `compliance_folder` exists.
- (2) This is another task using the `file` module, this time to empty the content of the directory, removing the old reports from the folder. This ensures only the last report is available.
- (3) Use the `copy` module to copy the data content into the destination file.

2. Run the playbook to see the new folders get created alongside the compliance reports:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# ansible-playbook
pb_napalm_compliance.yml
PLAY [VALIDATING NETWORK STATE VIA NAPALM] *****
TASK [VALIDATION WITH NAPALM] *****
ok: [ceos-01]
... omitted output ...
TASK [CREATE COMPLIANCE REPORT FOLDER] *****
changed: [ceos-01]
TASK [CLEAN LAST COMPLIANCE REPORT] *****
ok: [ceos-01]
ok: [crpd-02]
... omitted output ...
TASK [CREATE COMPLIANCE REPORT] *****
changed: [ceos-02]
changed: [crpd-02]
... omitted output ...
```

Great, the playbook executed successfully! The new compliance folder was created, all the existing files were deleted, and new files were created per host with all the compliance details included.

3. Let's take a look at one of the compliance reports that were generated:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# cat compliance_
folder/ceos-01.txt
complies: true
get_bgp_neighbors:
  complies: true
  extra: []
  missing: []
  present:
  global:
    complies: true
    nested: true
skipped: []
```


The compliance results show compliance for each NAPALM fact you are validating against. It is also capable of providing lines that are extra and missing. For example, if you modify one of the BGP peers and set it to false for `is_enabled` and `is_up`, the output of the compliance report will appear as follows:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch07# cat compliance_
folder/ceos-01.txt
172.31.1.7:
    present:
        is_enabled:
            actual_value: true
            complies: false
            expected_value: false
            nested: false
        is_up:
            actual_value: true
            complies: false
            expected_value: false
            nested: false
```

With the reports generated and explained, this brings us to the end of our chapter on NAPALM. The modules that were used throughout the chapter show the great power of NAPALM, but it's important to mention that NAPALM is built and maintained by a community of developers, so not all NAPALM drivers have the same level of support. If you decide to use NAPALM in a production multi-vendor environment, make sure that you test and validate the operations supported by each NAPALM driver.

Summary

In this chapter, you learned the power of NAPALM, from executing getters to go out and retrieve operational state data from multiple devices, to checking routing information, to pushing configuration, and, finally, validating state data and creating compliance reports. As this is the last chapter in this section of the book, it's important to realize that you've now configured an end-to-end topology in Containerlab. All the configurations were automated and pushed using Ansible. Finally, you created an Ansible role that focused on code reuse and helped illustrate the power of reusing code in projects.

Coming up in the next section of the book, you will learn how to take your Ansible journey into the cloud and discover how to use Ansible to manage public cloud infrastructure.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



Part 3

Managing Cloud Networking with Ansible

As organizations increasingly adopt hybrid and multi-cloud strategies, automating cloud networking has become essential for agility, scalability, and consistency. In this part of the book, we explore how Ansible extends beyond traditional network infrastructure to automate networking services across the three major public cloud providers.

Through detailed, lifecycle-based examples, you'll learn how to use Ansible to deploy, validate, and decommission networking resources in AWS, Azure, and Google Cloud Platform (GCP). Each chapter focuses on a specific cloud environment, showing how to automate tasks such as provisioning VPCs, managing routing and firewalls, and validating network state—all within an infrastructure-as-code workflow.

By the end of this part, you'll have the tools and patterns needed to bring the same level of automation maturity to your cloud networks as you would in on-premises or hybrid environments, streamlining operations while reducing risk and complexity.

This part of the book includes the following chapters:

- *Chapter 8, Deploying and Operating AWS Networking*
- *Chapter 9, Deploying and Operating Azure Networking*
- *Chapter 10, Deploying and Operating GCP Networking*

8

Deploying and Operating AWS Networking

Cloud technology is transforming numerous industries, fundamentally reshaping IT infrastructure, application deployment, and the way applications are architected for cloud environments. Among the leading cloud providers, AWS stands out, offering an extensive range of networking resources and services designed to support scalable, highly available network architectures that host applications in the AWS cloud.

Automation is a key pillar of cloud adoption, enabling the rapid deployment of workloads. Each cloud provider has unique automation tools; for example, AWS uses CloudFormation, a service that allows users to define AWS infrastructure through **Infrastructure as Code (IaC)** and deploy it seamlessly. Ansible, however, provides a significant advantage over provider-specific tools such as CloudFormation by enabling resource deployment across multiple cloud providers. This cross-platform capability allows Ansible to serve as a consistent tool for managing workloads in multi-cloud environments.

Ansible offers a variety of modules to interact with AWS, enabling the provisioning and management of various resources. This chapter will guide you through deploying foundational AWS network services using Ansible, enabling the construction of a scalable network design within AWS. We'll use a sample AWS network design as a reference, demonstrating step by step how to build it using Ansible.

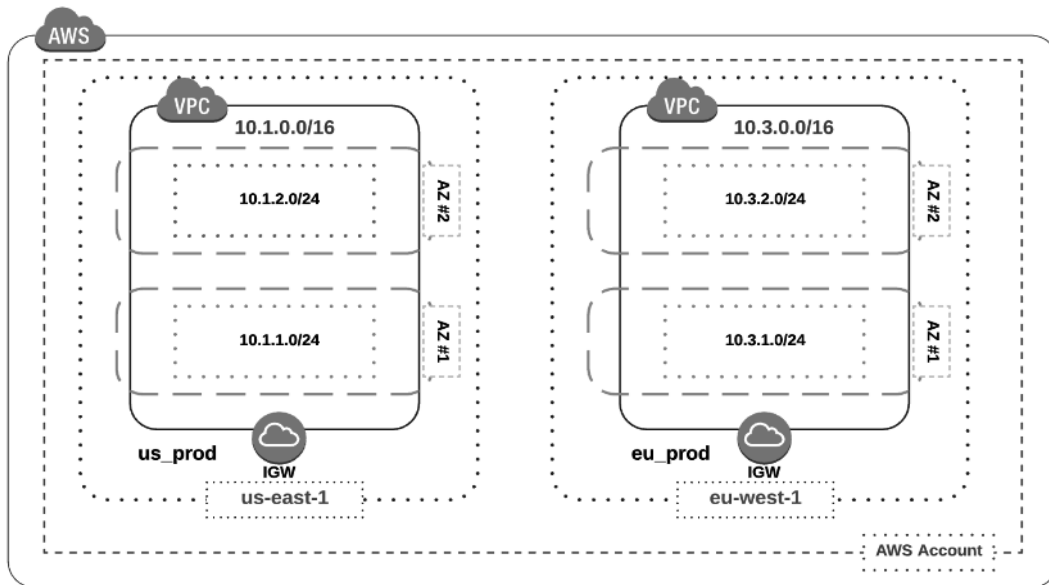


Figure 8.1 – AWS networking scenario

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



The main recipes covered in this chapter are as follows:

- Installing the AWS Python SDK
- Building an Ansible inventory
- Authenticating to your AWS account

- Deploying VPCs using Ansible
- Deploying subnets using Ansible
- Deploying IGWs using Ansible
- Controlling routing within a VPC using Ansible
- Deploying network ACLs using Ansible
- Deployment validation using Ansible
- Decommissioning AWS resources using Ansible



We will use the same lab scenario as used in the rest of the chapters. The code used in this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch08.

But before diving into the AWS recipes, we have to prepare by installing the required libraries.

Installing the AWS Python SDK

In this recipe, we'll cover the steps to install the necessary Python libraries for Ansible to interact with AWS. Installing these libraries (botocore and boto3) on the same Python environment Ansible uses is essential because they abstract the interaction with AWS' API. These packages must be installed on the Ansible Python environment, as all Ansible AWS modules depend on one of these libraries to function.

By following these steps, you will make sure that your Ansible execution environment is ready to use:

1. Check that the `amazon.aws` Ansible collection is installed:

```
root@cookbook-lab:~/cookbook-lab# ansible-galaxy collection list
amazon.aws
# /usr/local/lib/python3.10/dist-packages/ansible_collections
Collection Version
-----
amazon.aws 8.2.1
```

2. We can verify that the Ansible AWS module needs the botocore and boto3 Python packages to run:

```
root@cookbook-lab:~/cookbook-lab# ansible localhost -m amazon.aws.
aws_az_info
localhost | FAILED! => {
```



```

    "changed": false,
    "msg": "Failed to import the required Python library (
           botocore and boto3) on cookbook-lab's Python
           /usr/bin/python3. ... omitted output ..."
  }

```

3. As root, install the botocore and boto3 packages:

```

root@cookbook-lab:~/cookbook-lab# pip install botocore boto3
... omitted output ...
Successfully installed boto3-1.35.50 botocore-1.35.50 python-
dateutil-2.9.0.post0 s3transfer-0.10.3

```

4. Rerun `amazon.aws.aws_az_info` and you'll see that the module fails due to a lack of configuration:

```

root@cookbook-lab:~/cookbook-lab# ansible localhost -m amazon.aws.
aws_az_info
localhost | FAILED! => {
    "changed": false,
    "msg": "The amazon.aws.aws_az_info module requires a region and
           none was found in configuration, environment variables or
           module parameters"
}

```



Note

In this chapter, we will use the official AWS Ansible collection. For further information, check out the documentation at: <https://docs.ansible.com/ansible/latest/collections/amazon/aws/index.html>.

After sorting out the execution dependencies, it's time to create the necessary inventory configuration.

Building an Ansible inventory

Ansible needs a description of the infrastructure network setup that we will build across the AWS public cloud. In this recipe, we create the **host inventory file** in Ansible, declaring our **Virtual Private Clouds (VPCs)** as nodes in the inventory, similar to defining a network node. The main difference is that VPCs don't have a management IP address, so we skip the `ansible_host` argument for them.

Think of these inventory hosts not as machines to SSH into but as logical objects that your playbook will manage. The `transport=local` setting in `ansible.cfg` tells Ansible to run all tasks from the control node and interact directly with the AWS API.

In the inventory file, we organize VPCs into the following groups:

- `us_prod_vpcs`: Grouping all VPCs in the US
- `eu_prod_vpcs`: Grouping all VPCs in Europe
- `prod_vpcs`: Grouping both the US and Europe VPCs together (equivalent to all VPCs, in this case)

Additionally, we configure the `ansible.cfg` file with the settings used in previous recipes, specifying the vault password file that stores the encryption password for securing sensitive information.

1. Create a new folder called `ch08` within `chapters`, and create a `hosts` file inside it:

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch08
root@cookbook-lab:~/cookbook-lab# vi chapters/ch08/hosts
[us]
    us_prod_vpc
[eu]
    eu_prod_vpc
[prod_vpcs]
    us_prod_vpc
    eu_prod_vpc
```

2. Also inside the `chapters/ch08` folder, create `ansible.cfg`, with the same configuration used in previous chapters (refer to *Chapter 2*):

```
root@cookbook-lab:~/cookbook-lab# vi chapters/ch08/ansible.cfg
[defaults]
    inventory=hosts
    vault_password_file=~/.ansible_vault_passwd
    gathering=explicit
    transport=local
    retry_files_enabled=False
    action_warnings=False
```

**Note**

In this chapter, we use a static inventory file that works well for simple scenarios. However, at scale, for cloud-native scenarios, we recommend using the AWS dynamic inventory plugin, which would help you manage your existing resources (e.g., VPCs) without typing them all into a file. You will learn how to use it in *Chapter 15*.

After preparing the Ansible environment, next, we have to get the right access to the AWS cloud platform to allow interaction with it programmatically.

Authenticating to your AWS account

As you may guess, in order to interact with AWS, you are required to *authenticate* yourself and also get the right permissions level to perform the operations. The AWS service in charge of this is **AWS Identity and Access Management (IAM)**, and it provides fine-grained access control to all AWS services. However, to make it simpler, we will use the default **AdministratorAccess** policy, which grants the administrator user access to all AWS services, for example, to create VPCs or subnets (in production environments, you should carefully select the right permissions level).

**Note**

If you don't have an AWS account, you can get free tier access at aws.amazon.com/free.

In this recipe, we create the IAM programmatic credentials and store them in our Ansible environment to allow the playbooks to use them:

1. Create a new user in the AWS console using AWS IAM, by going to **IAM | Users | Create user**.

IAM > Users > Create user

Step 1
Specify user details

Step 2
Set permissions

Step 3
Review and create

Specify user details

User details

User name

ansible

The user name can have up to 64 characters. Valid characters: A-Z, a-z, 0-9, and + = . @ _ - (hyphen)

☐ Provide user access to the AWS Management Console - *optional*
If you're providing console access to a person, it's a [best practice](#) to manage their access in IAM Identity Center.

If you are creating programmatic access through access keys or service-specific credentials for AWS CodeCommit or Amazon Keyspaces, you can generate them after you create this IAM user.
[Learn more](#)

Cancel

Next

2. Assign the **AdministratorAccess** IAM permission policy.

IAM > Users > Create user

Step 1
Specify user details

Step 2
Set permissions

Step 3
Review and create

Set permissions

Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. [Learn more](#)

Permissions options

☐ Add user to group
Add user to an existing group, or create a new group. We recommend using groups to manage user permissions by job function.

☐ Copy permissions
Copy all group memberships, attached managed policies, and inline policies from an existing user.

☒ Attach policies directly
Attach a managed policy directly to a user. As a best practice, we recommend attaching policies to a group instead. Then, add the user to the appropriate group.

Permissions policies (1/1247)

Choose one or more policies to attach to your new user.

Filter by Type

administrator

All types

15 matches

Policy name	Type	Attached entities
☒ AdministratorAccess	AWS managed - job function	1

3. Review the details and create the user.

The screenshot shows the 'Review and create' step in the AWS IAM console. The breadcrumb trail is 'IAM > Users > Create user'. The left sidebar shows three steps: 'Step 1: Specify user details' (active), 'Step 2: Set permissions', and 'Step 3: Review and create'. The main content area is titled 'Review and create' and includes a sub-header 'Review your choices. After you create the user, you can view and download the autogenerated password, if enabled.'

User details

User name ansible	Console password type None	Require password reset No

Permissions summary

Name	Type	Used as
AdministratorAccess	AWS managed - job function	Permissions policy

Tags - optional

Tags are key-value pairs you can add to AWS resources to help identify, organize, or search for resources. Choose any tags you want to associate with this user.

No tags associated with the resource.

[Add new tag](#)

You can add up to 50 more tags.

At the bottom right, there are three buttons: 'Cancel', 'Previous', and 'Create user'.

4. On the new Ansible user page, go to the **Security Credentials** tab, then **Access keys**, and select **Create access key**. Acknowledge the best practice recommendation, and continue to the last step, **Create access key**:

The screenshot shows the 'Access key best practices & alternatives' page in the AWS IAM console. The breadcrumb trail is 'IAM > Users > ansible > Create access key'. The left sidebar shows three steps: 'Step 1: Access key best practices & alternatives' (active), 'Step 2 - optional: Set description tag', and 'Step 3: Retrieve access keys'.

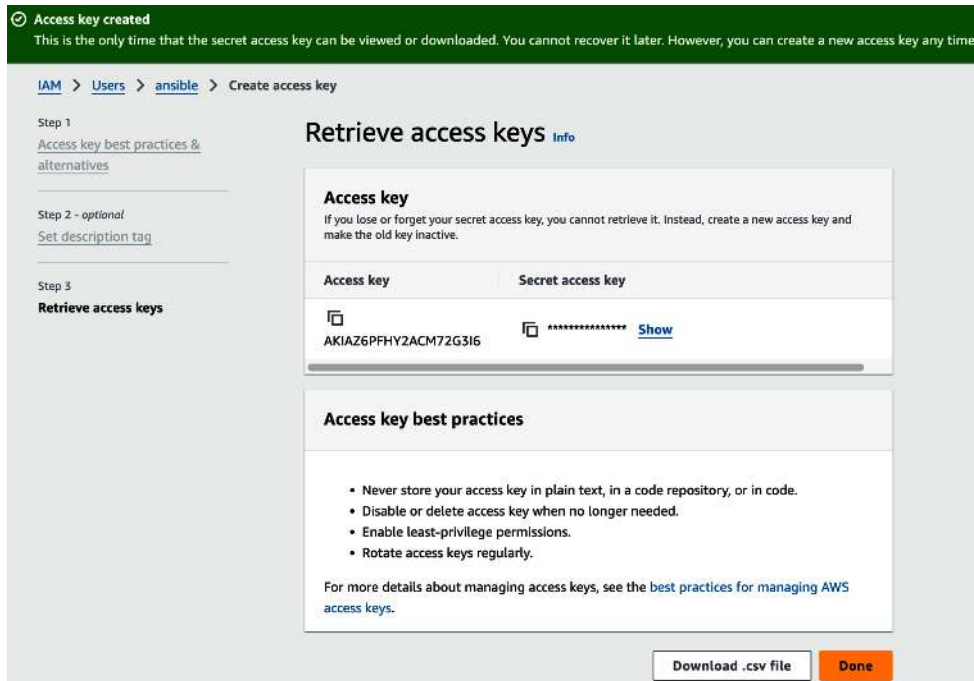
Access key best practices & alternatives

Avoid using long-term credentials like access keys to improve your security. Consider the following use cases and alternatives.

Use case

- ☐ **Command Line Interface (CLI)**
You plan to use this access key to enable the AWS CLI to access your AWS account.
- ☒ **Local code**
You plan to use this access key to enable application code in a local development environment to access your AWS account.

- On the final page, you have access to the access key ID and its secret access key. You can also download it as a CSV file. These two passwords are only visible and available upon creation. If you lose either of them, you should create a new one.



- Now, we encrypt the credentials using Ansible Vault because, as the best practices recommend, we should never store them in plain text. `ansible-vault` provides an encrypted output using the vault password, and the same password will be used to decrypt the content when needed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-vault
encrypt_string <ACCESS_KEY_ID> --name aws_access_key
Encryption successful
aws_access_key: !vault |
          $ANSIBLE_VAULT;1.1;AES256
... omitted output ...
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-
vault encrypt_string <SECRET_ACCESS_KEY> --name aws_secret_key_id
Encryption successful
```

```
aws_secret_key_id: !vault |
    $ANSIBLE_VAULT;1.1;AES256
... omitted output ...
```

7. Finally, you create the `group_vars` folder inside `ch08` and create the `all.yml` file within it. In the `all.yml` file, you use the encrypted output of the previous step, so you will be able to leverage them when creating the AWS resources. Notice that you could overwrite these variables (i.e., the credentials) with the more concrete `group_vars`, for example, for production and development environments (i.e., accounts):

```
ansible_connection: local

aws_access_key: !vault |
    $ANSIBLE_VAULT;1.1;AES256
... omitted output ...
aws_secret_key_id: !vault |
    $ANSIBLE_VAULT;1.1;AES256
... omitted output ...
```



Note

The AWS console steps may differ for different console versions, but the idea remains the same.

Everything is ready to start automating AWS infrastructure. The first step will be to deploy the AWS VPC because it will contain all the other resources.

Deploying VPCs using Ansible

AWS VPCs form the foundational network structure within AWS, essentially acting as virtual data centers within each AWS account. Before any other infrastructure-related services can be established, a VPC must first be in place. We'll outline how to define the necessary VPC configurations and automate their deployment using Ansible.



Note

Most of the steps in this chapter have to be done incrementally. For example, you can't create a subnet without a VPC. We recommend you follow the proposed order.

AWS divides its global infrastructure into **Regions**, each serving as an *isolated fault domain* with distinct orchestration and management systems. When creating a VPC, specifying the deployment Region is essential. In our example, we assign the `us-east-1` Region for VPCs in the US and the `eu-west-1` Region for VPCs in the EU. These values are specified in Ansible under the `aws_region` variable within the `eu.yml` and `us.yml` files, located in the `group_vars` directory.

Most AWS services are Region-specific, and all networking constructs we build will reflect this setup. To ensure Ansible modules connect to the appropriate AWS Region's API endpoint, the Region must be specified, as each Region's API endpoint has a unique FQDN. Refer to the AWS documentation (<https://docs.aws.amazon.com/general/latest/gr/rande.html>) for a comprehensive list of AWS service endpoints by Region.

Each VPC's specific variables—such as name, IP prefix, and tags—will be defined in separate YAML files under the `host_vars` directory. Finally, the Ansible playbook for provisioning the VPCs leverages the `ec2_vpc_net` Ansible module and the `environment` option to temporarily set environment variables (`AWS_ACCESS_KEY` and `AWS_SECRET_KEY`) during execution, drawing their values from `aws_access_key` and `aws_secret_key_id` in `group_vars/all.yml`. This provides the necessary credentials for AWS modules to authenticate API requests during the playbook run.

1. Create the `us.yml` and `eu.yml` files under the `group_vars` directory and populate these files with the AWS Region name definitions, as follows:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi group_vars/eu.yml
aws_region: eu-west-1
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi group_vars/us.yml
aws_region: us-east-1
```

2. Create the `host_vars` folder, and populate `eu_prod_vpc.yml` and `us_prod_vpc.yml` with the VPC parameters. Check the repository for the US file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# mkdir host_vars
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/eu_
prod_vpc.yml
vpc_name: EU_Prod_Public_VPC
vpc_cidr: 10.3.0.0/16
vpc_tags:
  role: prod
  region: EU
```


3. Create the new playbook, `pb_aws_net_build.yml`, to create the VPCs, and register the output in the `create_vpc` variable (there is an extra step, `debug`, to output the registered variable):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.
yml
- name: Create all AWS Networks
  hosts: prod_vpcs
  environment:
    AWS_ACCESS_KEY: "{{ aws_access_key }}"
    AWS_SECRET_KEY: "{{ aws_secret_key_id }}"
  tasks:
    - name: Create New VPC
      amazon.aws.ec2_vpc_net:
        cidr_block: "{{ vpc_cidr }}"
        region: "{{ aws_region }}"
        name: "{{ vpc_name }}"
        state: "{{ vpc_state | default('present') }}"
        tags: "{{ vpc_tags }}"
      register: create_vpc
    - name: Display the VPC output
      ansible.builtin.debug:
        var: create_vpc
```

4. Run the playbook created in the previous step:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_build.yml
PLAY [Create all AWS Networks] *****
TASK [Create New VPC] *****
changed: [us_prod_vpc]
changed: [eu_prod_vpc]
TASK [Display the VPC output] *****
ok: [us_prod_vpc] => {
  "create_vpc": {
    ... output omitted for brevity ...
    "vpc": {
      "cidr_block": "10.1.0.0/16",
      "id": "vpc-0cdf9079b52542fa8",
      "name": "US_Prod_Public_VPC",
```

```

    "owner_id": "<your_aws_account>",
    "state": "available",
    "tags": {
      "Name": "US_Prod_Public_VPC",
      "region": "US",
      "role": "prod"
    }
    ... omitted output ...
  },
}
}
ok: [eu_prod_vpc] => {
  ... omitted output ...

```

5. If you check the AWS console, selecting the Regions where the VPCs were created, you will see the new VPCs showing up. The following screenshot is from the us-west-1 Region (notice that there is another VPC that comes by default with AWS' initial setup):

Your VPCs (1/2) info Last updated 1 minute ago Actions

Search

	Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR
☒	US_Prod_Public_VPC	vpc-0cdf9079b52542fa8	Available	10.1.0.0/16	—
☐	—	vpc-5e4e8638	Available	172.31.0.0/16	—

vpc-0cdf9079b52542fa8 / US_Prod_Public_VPC

Details | Resource map | CIDRs | Flow logs | Tags | Integrations

Details

VPC ID vpc-0cdf9079b52542fa8	State Available	DNS hostnames Enabled	DNS resolution Enabled
Tenancy Default	DHCP option set dopt-0dd1116a	Main route table rtb-0453d54919ceeeda9	Main network ACL acl-07173b108a0d9194d



Note

Ansible's value proposition lies in its readability and simplicity. To appreciate this, we encourage you to explore how the same VPC definition would look using other tools—for example, AWS CloudFormation—as a point of comparison.

After the VPC creation, AWS allows the network domains to be broken down using subnets.

Deploying Subnets using Ansible

Subnets are a fundamental networking construct in AWS, enabling enhanced resiliency for applications deployed on the cloud. By mapping subnets to different Availability Zones, we can ensure high availability, distributing resources across zones to mitigate the impact of any single-zone failure. This approach strengthens the deployment's reliability and availability.

Within an AWS Region, where we have deployed the VPC, AWS offers Availability Zones to provide resiliency for physical infrastructure, and efficient use involves distributing infrastructure across them within a VPC, which belongs to a Region. This is achieved using AWS subnets.

In our example, we deploy two subnets across two Availability Zones to ensure high availability. Each subnet is defined with its own CIDR block (which must be a subset of the VPC CIDR), an Availability Zone, and relevant tags. Availability Zones are named by AWS Region plus a suffix (e.g., a, b, etc.).

To create subnets, we associate each with its parent VPC by specifying the `vpc-id` parameter, which AWS assigns to a VPC upon creation. We retrieve this identifier from the VPC creation task output, storing it in the `vpc_create` variable. Using `set_fact`, we map this ID to the `vpc-id` variable, and then create the subnets via the `ec2_vpc_subnet` module, iterating over the `vpc_subnets` data structure to provision each required subnet.

1. Update the `host_vars/eu_prod_vpc.yml` and `host_vars/us_prod_vpc.yml` files with the subnet's data, as shown here (check the repository for the US example):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/us_
prod_vpc.yml
... omitted output ...
vpc_subnets:
  eu-prod-public-a:
    cidr: 10.3.1.0/24
    az: "{{ aws_region }}a"
    tags: "{{ vpc_tags }}"
    public: true
  eu-prod-public-b:
    cidr: 10.3.2.0/24
    az: "{{ aws_region }}b"
    tags: "{{ vpc_tags }}"
    public: true
```

2. Update the previous `pb_aws_net_build.yml` playbook with the retrieval of the VPC ID (stored in the `create_vpc` variable) and the creation of the two subnets:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.yml
... omitted output ...
  - name: Set VPC ID
    set_fact:
      vpc_id: "{{ create_vpc.vpc.id }}"
  - name: Create VPC subnets
    amazon.aws.ec2_vpc_subnet:
      region: "{{ aws_region }}"
      vpc_id: "{{ vpc_id }}"
      cidr: "{{ item.value.cidr }}"
      az: "{{ item.value.az }}"
      tags: "{{ item.value.tags | combine({ 'Name': item.key }) }}"
    with_dict: "{{ vpc_subnets }}"
    register: create_vpc_subnets
```

3. Now, you can rerun the playbook update. The AWS Ansible collection provides a declarative approach, so as the VPC was already created, it's not changed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook pb_aws_net_build.yml
... output omitted ...
TASK [Create VPC subnets] *****
changed: [us_prod_vpc] => (item={'key': 'us-prod-public-a', 'value': {'cidr': '10.1.1.0/24', 'az': 'us-east-1a', 'tags': {'role': 'prod', 'region': 'US'}, 'public': True}})
changed: [eu_prod_vpc] => (item={'key': 'eu-prod-public-a', 'value': {'cidr': '10.3.1.0/24', 'az': 'eu-west-1a', 'tags': {'role': 'prod', 'region': 'EU'}, 'public': True}})
changed: [us_prod_vpc] => (item={'key': 'us-prod-public-b', 'value': {'cidr': '10.1.2.0/24', 'az': 'us-east-1b', 'tags': {'role': 'prod', 'region': 'US'}, 'public': True}})
changed: [eu_prod_vpc] => (item={'key': 'eu-prod-public-b', 'value': {'cidr': '10.3.2.0/24', 'az': 'eu-west-1b', 'tags': {'role': 'prod', 'region': 'EU'}, 'public': True}})
```

4. After the playbook runs, you can double-check the new AWS subnets created in the AWS console. All the predefined data is taken into account, including the tags.



Subnets (1/8) Info Last updated 1 minute ago Actions Create subnet

Find resources by attribute or tag

	Name	Subnet ID	State	VPC	IPv4 CIDR
☒	us-prod-public-a	subnet-06d36c2d863a101df	Available	vpc-0cdf9079b52542fa8 US_P...	10.1.1.0/24
☐	us-prod-public-b	subnet-09f668f4dd8915892	Available	vpc-0cdf9079b52542fa8 US_P...	10.1.2.0/24

The next networking service to add is the **Internet Gateway (IGW)**, which provides internet connectivity.

Deploying IGWs using Ansible

In this recipe, we'll outline how to deploy IGWs to AWS VPCs using Ansible. An IGW serves as the exit point from a VPC to the internet, allowing public-facing resources within the VPC, such as EC2 or RDS instances, to connect to external networks. For a public-facing service, IGWs are essential to provide internet reachability.

To create an IGW, we must specify the VPC it will attach to, which requires the VPC's unique ID. This ID is obtained during VPC creation and can be saved as a variable for reference in subsequent tasks. We can then use the `ec2_vpc_igw` module to create the IGW, specifying the target AWS Region, the VPC ID, and any tags we want to assign. While tags are optional, they are highly recommended for automation, as they facilitate resource identification for deployment validation and fact collection in later tasks.

Once deployed, the `ec2_vpc_igw` module will return various parameters about the IGW, including its unique `igw-id`, which is critical for future operations involving the IGW.

1. As before, we continue updating the EU and US `host_vars` files:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/us_
prod_vpc.yml
... omitted output ...
igw_name: us_prod_igw
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/eu_
prod_vpc.yml
... omitted output ...
igw_name: eu_prod_igw
```

2. Then, we keep extending the `pb_aws_net_build.yml` playbook with the new task to build the IGW nodes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.
yml
... omitted output ...
- name: Create IGW
  amazon.aws.ec2_vpc_igw:
    region: "{{ aws_region }}"
    vpc_id: "{{ vpc_id }}"
    state: present
    tags: "{{ vpc_tags | combine({'Name': igw_name}) }}"
    register: vpc_igw_create
- name: Extract VPC IGW ID
  set_fact:
    igw_id: "{{ vpc_igw_create.gateway_id }}"
```

3. Like before, we run the updated playbook to create the IGW:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_build.yml
... output omitted ...
TASK [Create IGW] *****
changed: [us_prod_vpc]
changed: [eu_prod_vpc]
TASK [Display the IGW output] *****
ok: [us_prod_vpc] => {
  "vpc_igw_create": {
    "changed": true,
    "failed": false,
    "gateway_id": "igw-0f5a57b45b16aa372",
    "tags": {
      "Name": "us_prod_igw",
      "region": "US",
      "role": "prod"
    },
    "vpc_id": "vpc-0cdf9079b52542fa8"
  }
}
```

```
... omitted output ...
TASK [Extract VPC IGW ID] *****
ok: [eu_prod_vpc]
ok: [us_prod_vpc]
```

- 4. As previously, in the AWS console, you can verify that the new IGW is ready to use.

Internet gateways (1/2) Info

Q Search

☐	Name	Internet gateway ID	State
☒	us_prod_igw	igw-0f5a57b45b16aa372	Attached
☐	-	igw-49089f2e	Attached

igw-0f5a57b45b16aa372 / us_prod_igw

Details

Tags

Details

Internet gateway ID  igw-0f5a57b45b16aa372	State Attached	VPC ID vpc-0cdf9079b52542fa8 US_Prod_Public_VPC

The IGW serves as a routing instance to access the internet, but as your networking knowledge tells you, to use a router, first, you need a route to forward the packets to. This is the next step.

Controlling routing within a VPC using Ansible

This recipe covers how to configure **routing** within an AWS VPC to manage traffic flow across its subnets and to external destinations. By adjusting routing within the VPC, we can control how traffic is directed both internally and when exiting to the internet. So far, we’ve set up the VPC, its subnets, and an IGW for internet connectivity. However, without updates to the VPC’s routing table to point to the IGW, no internal traffic will route through it, making routing adjustments essential to enabling internet-bound traffic.



Note

AWS VPCs include a default route table, automatically assigned to the VPC and any subnet lacking a specific route table. As a result, all subnets within the VPC are initially associated with this default route table.

Let's go through the process from the data definition to the playbook execution:

1. Again, the first step is updating `host_vars/eu_prod_vpc.yml` and the US file with the data that defines the subnets whose route table has to be updated, and the route to add (with the destination and the next *hop*):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/eu_
prod_vpc.yml
... omitted output ...
route_table:
  tags:
    name: eu_prod_public_rt
  routes:
    - dest: 0.0.0.0/0
      gateway_id: "{{ igw_id }}"
  subnets:
    - eu-prod-public-a
    - eu-prod-public-b
```

2. Then, you have to add the new tasks to the `pb_aws_net_build.yml` playbook to get the default VPC route table:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.
yml
... omitted output ...
- name: Get Default VPC Route Table
  amazon.aws.ec2_vpc_route_table_info:
    region: "{{ aws_region }}"
    filters:
      vpc-id: "{{ vpc_id }}"
  register: vpc_route_table_facts
- name: Extract Route Table IDs
  set_fact:
    rt_id: "{{vpc_route_table_facts.route_tables[0].id }}"
```

3. With the default VPC route table, we update it by assigning the new routes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.
yml
... omitted output ...
- name: Update Default VPC Route Table
```



```
ec2_vpc_route_table :
  region: "{{ aws_region }}"
  vpc_id: "{{ vpc_id }}"
  route_table_id: "{{ rt_id }}"
  routes: "{{ route_table.routes }}"
  subnets: "{{ route_table.subnets }}"
  lookup: id
  state: present
  tags: "{{ vpc_tags | combine(route_table.tags) }}"
```

4. Then, we run the playbook to update the route table. In the **Display the Route Table output**, you can observe the default routing table for `us_prod_vpc` before changing it, with the local route for the whole VPC (i.e., `10.1.0.0/16`):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_build.yml
... output omitted ...
TASK [Get Default VPC Route Table] *****
ok: [us_prod_vpc]
ok: [eu_prod_vpc]
TASK [Display the Route Table output] *****
ok: [us_prod_vpc] => {
  "vpc_route_table_facts": {
    ... omitted output ...
    "route_tables": [
      {
        "associations": [
          {
            "association_state": {
              "state": "associated"
            },
            "main": true,
            "route_table_association_id": "rtbassoc-
01888c955331efb32",
            "route_table_id": "rtb-0453d54919ceeeda9"
          }
        ],
        "id": "rtb-0453d54919ceeeda9",
```

```
      "routes": [
        {
          "destination_cidr_block": "10.1.0.0/16",
          "gateway_id": "local",
          ... omitted output ...
        }
      ],
      "vpc_id": "vpc-0cdf9079b52542fa8"
    }
  ]
}
ok: [eu_prod_vpc] => {
  ... omitted output ...
TASK [Extract Route Table IDs] *****
ok: [us_prod_vpc]
ok: [eu_prod_vpc]
```

- 5. In the following screenshot, you can observe how the `us_prod_public_rt` route table has the default `0.0.0.0/0` destination route through the IGW, and the route table is associated with the two subnets.

Route tables (1/2) Info

Last updated less than a minute ago

Actions

Create

Find resources by attribute or tag

Name	Route table ID	Explicit subnet associ...	Edge...	Main	VPC
☒ us_prod_public_rt	rtb-0453d54919ceeeda9	2 subnets	-	Yes	vpc-0cdf9079b5
☐ -	rtb-21858047	-	-	Yes	vpc-5e4e8638

rtb-0453d54919ceeeda9 / us_prod_public_rt

Details Routes Subnet associations Edge associations Route propagation Tags

Routes (2)

Filter routes

Destination	Target	Status	Propagated
0.0.0.0/0	igw-0f5a57b45b16aa372	Active	No
10.1.0.0/16	local	Active	No

Finally, we will bring a network security functionality into AWS networking.

Deploying NACLs using Ansible

AWS **Network Access Control Lists (NACLs)** are stateless ACLs that regulate IP traffic at the subnet level, allowing or denying access based on Layer 3 and Layer 4 IP details. Applied to subnets, NACLs protect all resources within the subnet by controlling both ingress (incoming) and egress (outgoing) traffic. Rules in an NACL are processed by rule number, with the first matching rule determining the traffic's fate. Complementing NACLs, but not covered here, AWS security groups provide an additional layer of protection at the instance level, using stateful filtering to control inbound and outbound traffic per resource, which adds granularity to security within the VPC.

Let's reproduce the same workflow, from the data to the deployment:

1. We start adding the NACLs' data, with a data structure that represents a traditional ACL, also matching the expected data format by AWS. You can define ingress and egress rules according to the traffic direction. Then, it includes a *rule identifier* (for ordering), the *type of traffic*, and the *IP and port destination*, plus the *action* (allow/deny):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi host_vars/eu_
prod_vpc.yml
... omitted output ...
network_acls:
  - name: eu_prod_acls
    subnets: "{{ vpc_subnets.keys() | list }}"
    ingress_rules:
      - [100, 'tcp', 'allow', '0.0.0.0/0', null, null, 80, 80]
      - [200, 'tcp', 'allow', '0.0.0.0/0', null, null, 443, 443]
    egress_rules:
      - [100, 'all', 'allow', '0.0.0.0/0', null, null, null, null]
```

2. We update the `pb_aws_net_build.yml` playbook with the task to create the NACLs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# vi pb_aws_net_build.yml
... omitted output ...
- name: Create Network ACLs
  amazon.aws.ec2_vpc_nacl:
    region: "{{ aws_region }}"
    vpc_id: "{{ vpc_id }}"
    name: "{{ item.name }}"
    subnets: "{{ item.subnets }}"
    ingress: "{{ item.ingress_rules }}"
    egress: "{{ item.egress_rules }}"
    tags: "{{ vpc_tags | combine({'Name':item.name}) }}"
    loop: "{{ network_acls }}"
```

3. Finally, we can run the playbook again to update the NACL of each VPC:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook pb_aws_net_build.yml
... output omitted ...
TASK [Create Network ACLs] *****
changed: [us_prod_vpc] => (item={'name': 'us_prod_acls', 'subnets': ['us-prod-public-a', 'us-prod-public-b'], 'ingress_rules': [[100, 'tcp', 'allow', '0.0.0.0/0', None, None, 80, 80], [200, 'tcp', 'allow', '0.0.0.0/0', None, None, 443, 443]], 'egress_rules': [[100, 'all', 'allow', '0.0.0.0/0', None, None, None, None]]})
changed: [eu_prod_vpc] => (item={'name': 'eu_prod_acls', 'subnets': ['eu-prod-public-a', 'eu-prod-public-b'], 'ingress_rules': [[100, 'tcp', 'allow', '0.0.0.0/0', None, None, 80, 80], [200, 'tcp', 'allow', '0.0.0.0/0', None, None, 443, 443]], 'egress_rules': [[100, 'all', 'allow', '0.0.0.0/0', None, None, None, None]]})
```

4. In the AWS console, you can see the new NACL with the inbound and outbound rules created.

The screenshot shows the AWS Management Console interface for Network ACLs. At the top, there's a header 'Network ACLs (1/3)' with a search bar and a 'Create network ACL' button. Below this is a table listing Network ACLs:

Name	Network ACL ID	Associated with	Default	VPC ID
-	acl-07173b108a0d9194d	-	Yes	vpc-0cdf9079b52542fa8 / US_Prod_Pu...
☒ us_prod_acls	acl-071e14b438a7f6fa5	2 Subnets	No	vpc-0cdf9079b52542fa8 / US_Prod_Pu...
☐ -	acl-b9f86cdf	6 Subnets	Yes	vpc-5e4e8638

Below the table, the selected NACL 'acl-071e14b438a7f6fa5 / us_prod_acls' is shown with tabs for 'Details', 'Inbound rules', 'Outbound rules', 'Subnet associations', and 'Tags'. The 'Inbound rules' tab is active, showing a table of inbound rules:

Rule number	Type	Protocol	Port range	Source	Allow/Deny
100	HTTP (80)	TCP (6)	80	0.0.0.0/0	Allow
200	HTTPS (443)	TCP (6)	443	0.0.0.0/0	Allow
*	All traffic	All	All	0.0.0.0/0	Deny

With this, we have completed the recipes to create AWS resources. The same approach can be used to create other types of resources. Something interesting to notice is the need to store the outputs of previous tasks (e.g., identifiers) to connect the resource creation and association.

Now, we will focus on creating a playbook that can collect the operational state and validate that it is being deployed as per the design, that is, the data.

Deployment validation using Ansible

The AWS collection supports modules to collect information, for example, `ec2_vpc_net_info` and `ec2_vpc_subnet_info`. This is very useful to retrieve the actual operational information and compare it with the intended data, from host and group vars.

In this recipe, we will use a new playbook, `pb_aws_net_validate.yml`, adding VPC and subnet validation:

1. Create the `pb_aws_net_validate.yml` playbook, with the playbook definition and the VPC validation tasks. It checks the CIDR block assigned and the name, according to the data in `host_vars`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_validate.yml
- name: Validate VPC resources
  hosts: all
```

```
gather_facts: no
environment:
  AWS_ACCESS_KEY: "{{ aws_access_key }}"
  AWS_SECRET_KEY: "{{ aws_secret_key_id }}"
  AWS_REGION: "{{ aws_region }}"
tasks:
  - name: Get VPC info
    amazon.aws.ec2_vpc_net_info:
      filters:
        "tag:Name": "{{ vpc_name }}"
      register: vpc_facts
  - name: Validate VPC Info
    assert:
      that:
        - vpc_facts.vpcs[0].cidr_block == vpc_cidr
        - vpc_facts.vpcs[0].tags.Name == vpc_name
```

2. Then, you can complete the playbook with the validation of the subnet information. In this case, it validates the CIDR assigned and whether the subnet is in the right Availability Zone:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_validate.yml
... omitted output ...
  - block:
    - name: Extract VPC ID
      set_fact:
        vpc_id: "{{ vpc_facts.vpcs[0].id }}"
    - name: Get Subnet info
      ec2_vpc_subnet_info:
        filters:
          vpc-id: "{{ vpc_id }}"
        register: vpc_subnet_facts
    - name: Validate VPC Subnets Info
      assert:
        that:
          - vpc_subnet_facts.subnets |
              selectattr('tags.Name','equalto',item.key) |
              map(attribute='cidr_block') |
              list | first == item.value.cidr
```

```

- vpc_subnet_facts.subnets |
  selectattr('tags.Name','equalto',item.key) |
  map(attribute='availability_zone') |
  list | first == item.value.az
  with_dict: "{{ vpc_subnets }}"
when: vpc_facts.vpcs != []

```

3. When the playbook is executed, you can observe whether or not the assertions are passing:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_validate.yml
PLAY [Validate VPC resources] *****
TASK [Get VPC info] *****
ok: [us_prod_vpc]
ok: [eu_prod_vpc]
TASK [Validate VPC Info] *****
ok: [us_prod_vpc] => {
    "changed": false,
    "msg": "All assertions passed"
}
... output omitted ...

```

Before closing this chapter, it's necessary to cover the other side of the dynamic infrastructure creation, the decommissioning of resources.

Decommissioning resources on AWS using Ansible

In this recipe, we will outline how to **decommission** a complete network within AWS with all the associated network resources; that is, how we can easily build and tear down resources on the cloud with a simple playbook execution using Ansible. The starting point is the same, the intended data defined and used in the Ansible inventory, but this time using the collections to delete resources.

It's important to notice the order. We have to undo it in the reverse order of creation, considering the AWS resource dependencies. Thus, the VPC deletion is the last one.

Here are the steps:

1. Let's create a new `pb_delete_vpc.yml` playbook that implements all the tasks, starting with the VPC ID collection, which is needed in all the subsequent tasks:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_delete.yml
- name: Delete all VPC resources
  hosts: all
  gather_facts: no
  environment:
    AWS_ACCESS_KEY: "{{ aws_access_key }}"
    AWS_SECRET_KEY: "{{ aws_secret_key_id }}"
    AWS_REGION: "{{ aws_region }}"
  tasks:
    - name: Get VPC facts
      amazon.aws.ec2_vpc_net_info:
        filters:
          "tag:Name": "{{ vpc_name }}"
        register: vpc_facts
    - name: Extract VPC ID
      set_fact:
        vpc_id: "{{ vpc_facts.vpcs[0].id }}"
```

2. Then, we define a block where, for each VPC, all the related resources are deleted using `state: absent`. We leave the VPC deletion to the end:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_delete.yml
... omitted output ...
- name: Start Delete VPC Resources
  block:
    - name: Delete Subnets
      amazon.aws.ec2_vpc_subnet:
        cidr: "{{ item.value.cidr }}"
        vpc_id: "{{ vpc_id }}"
        state: absent
        with_dict: "{{ vpc_subnets }}"
    - name: Delete NACLs
```



```

amazon.aws.ec2_vpc_nacl:
  name: "{{ item.name }}"
  vpc_id: "{{ vpc_id }}"
  state: absent
loop: "{{ network_acls }}"
- name: Delete IGW
  amazon.aws.ec2_vpc_igw:
    vpc_id: "{{ vpc_id }}"
    state: absent
- name: Delete VPC
  amazon.aws.ec2_vpc_net:
    cidr_block: "{{ vpc_cidr }}"
    name: "{{ vpc_name }}"
    state: absent
when: vpc_id is defined

```

3. The last step is to run the `pb_aws_net_delete.yml` playbook, which cleans up all the AWS resources created through this chapter. Notice that the playbook recap notifies us of the four tasks that have changed the target infrastructure—subnets, NACLs, IGWs, and VPCs:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch08# ansible-playbook
pb_aws_net_delete.yml
... omitted output ...
TASK [Delete VPC] *****
changed: [us_prod_vpc]
changed: [eu_prod_vpc]
PLAY RECAP *****
eu_prod_vpc           : ok=6    changed=4    unreach-
able=0    failed=0    skipped=0    rescued=0    ignored=0
us_prod_vpc           : ok=6    changed=4    unreach-
able=0    failed=0    skipped=0    rescued=0    ignored=0

```

With the AWS infrastructure now cleaned up—a crucial step when working with dynamic environments—let’s move on to the chapter summary.

Summary

Ansible collections, such as the AWS collection, offer modules that support managing dynamic infrastructure with an idempotent approach. This ensures playbooks can be run repeatedly while producing consistent results, only adjusting resources when necessary to meet the desired state.

Unlike static infrastructure, where resources are predefined, dynamic infrastructure management often requires a different approach. For example, in *Part 2*, each network device was explicitly defined in the inventory. However, in this case, the inventory targets Regions, and the resources themselves are dynamically generated as the tasks are executed, as there's no fixed reference for them.

Throughout the remaining chapters of *Part 3*, we'll continue exploring additional dynamic infrastructure providers to build a well-rounded perspective and highlight comparable functionalities across providers.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



9

Deploying and Operating Azure Networking

In the previous chapter, we covered provisioning network resources in AWS and orchestrating those resources with Ansible. Now, we turn our attention to another leading cloud provider, **Microsoft**, and its **Azure cloud platform**. Azure offers a comprehensive networking service suite to support highly scalable, resilient cloud solutions. Leveraging Ansible's robust set of Azure-specific modules, you can automate and streamline deployments in the Azure environment, just as you did with AWS.

In this chapter, we'll explore Azure's fundamental networking constructs and demonstrate how to use key Ansible modules to efficiently build, configure, and validate a foundational network architecture within Azure, using the following scenario as our reference setup:

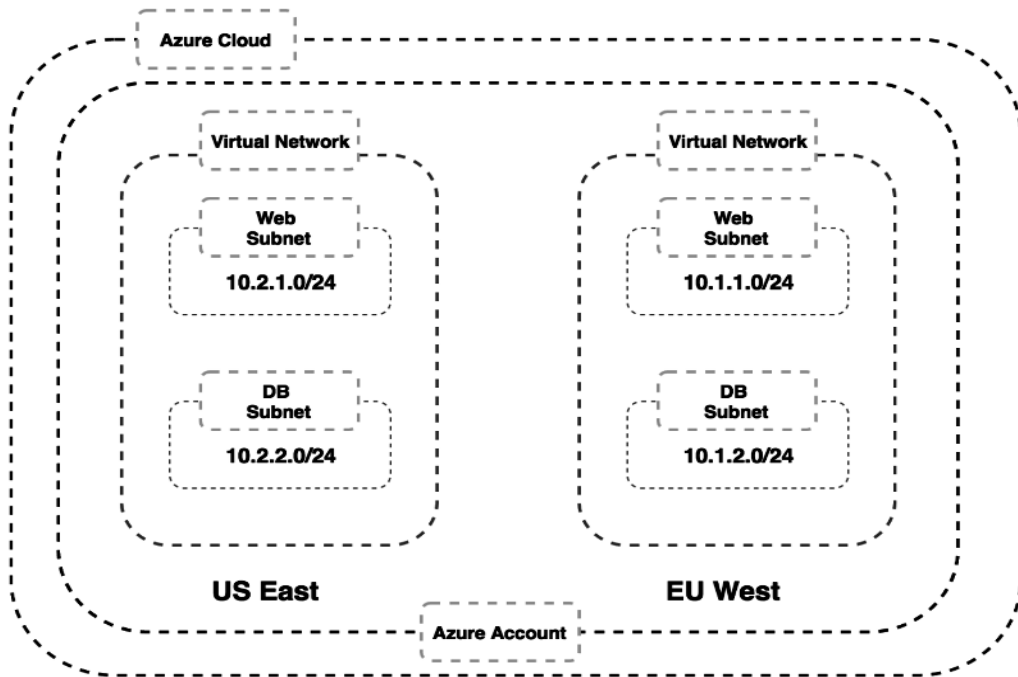


Figure 9.1 – Azure networking scenario

The main recipes covered in this chapter are as follows:

- Installing the Azure Python SDK
- Building an Ansible inventory
- Authenticating to your Azure account
- Creating a resource group
- Creating virtual networks
- Creating subnets
- Building user-defined routes
- Deploying network security groups
- Deployment validation
- Decommissioning Azure resources



As in all the other chapters, we will use the same lab scenario, and the code used in the chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch09

Similarly to the previous chapter, before getting started with Azure Ansible modules, you have to install the required dependencies.

Installing the Azure Python SDK

To interact with Azure with Ansible, you need to use Azure's Ansible collection and the dependent Azure Python libraries.

Following the next steps, you will ensure your Ansible Python environment is ready to go:

1. Check that the `azure.azcollection` Ansible collection is installed:

```
root@cookbook-lab:~/cookbook-lab# ansible-galaxy collection list
azure.azcollection
# /usr/local/lib/python3.10/dist-packages/ansible_collections
Collection          Version
-----
azure.azcollection  2.7.0
```

2. In this case, we suggest upgrading to the *latest version* (3.0.0) to match the Azure SDK requirements:

```
root@cookbook-lab:~/cookbook-lab# ansible-galaxy collection install
-U azure.azcollection
... output omitted ...
azure.azcollection:3.0.0 was installed successfully
```

3. You can verify that the Ansible Azure module needs `ansible[azure]` Python packages to run:

```
root@cookbook-lab:~/cookbook-lab# ansible localhost -m azure.
azcollection.azure_rm_resourcegroup_info
localhost | FAILED! => {
  "changed": false,
  "msg": "Failed to import the required Python library (ansible[
azure] (azure >= 2.0.0)) on cookbook-lab's Python
/usr/bin/python3. ... omitted output ..."
}
```

4. As root, you should install the `ansible[azure]` package. At the time of writing, a conflict exists between two required libraries, so we are pinning them to specific, compatible versions in the `cchapters/ch09/requirements.txt` file:

```
root@cookbook-lab:~/cookbook-lab# pip install -r requirements.txt
packaging
requests[security]
xmltodict
msgpack==1.6.0
azure-cli-core==2.64.0
... omitted output ...
```

5. Rerun `ansible.azcollection.azure_rm_resourcegroup_info` and check that the module fails due to a lack of configuration:

```
root@cookbook-lab:~/cookbook-lab# ansible localhost -m azure.
azcollection.azure_rm_resourcegroup_info
localhost | FAILED! => {
  "changed": false,
  "msg": "Failed to load CLI profile [Errno 2] No such file or
        directory: '/root/.azure/azureProfile.json'."
}
```

Note



In this chapter, we will use the official Azure Ansible collection. For more documentation, check out <https://docs.ansible.com/ansible/latest/collections/azure/azcollection/index.html#plugins-in-azure-azcollection>.

The module configuration will come after configuring the Azure credentials.

Building an Ansible inventory

Again, as in the AWS chapter, you have to describe the network services to deploy in the Azure cloud. In this recipe, we'll create another host inventory file in Ansible, declaring the virtual networks as nodes in the inventory, an equivalent of the AWS VPCs. You will still notice that the virtual network doesn't have a management IP address, so the `ansible_host` argument is not needed.

In the inventory file, we organize the network into the following groups:

- `us_az_net`: Grouping the virtual networks in the United States
- `eu_az_net`: Grouping the virtual networks in Europe
- `az_net`: Grouping both virtual networks



Note

Similarly to *Chapter 8*, we'll use a static inventory file. But there is an Azure dynamic inventory plugin to pull existing Azure resources dynamically:

https://docs.ansible.com/ansible/latest/collections/azure/azcollection/azure_rm_inventory.html

In this chapter, we use a static inventory file that works well for simple scenarios. However, at scale, for cloud-native scenarios, we recommend using the AWS dynamic inventory plugin, which will help you manage your existing resources (e.g., VPCs) without typing them all into a file. You will learn how to use it in *Chapter 15*.

You can reuse the `ansible.cfg` file with settings used in previous recipes (for example, *Chapter 8*), specifying the vault password file that stores the encryption password for securing sensitive information:

1. Create a new folder, `ch09`, within `chapters`, and create a `hosts` file inside it:

```
root@cookbook-lab:~/cookbook-lab# vi chapters/ch09/hosts
[us]
  us_az_net
[eu]
  eu_az_net
[az_net]
  us_az_net
  eu_az_net
```

After setting up the Ansible environment, the next step is to obtain proper access to the Azure cloud platform to enable programmatic interaction.

Authenticating to your Azure account

As with any other service, you have to create an account in Azure to obtain the credentials required to programmatically authenticate and interact with the Azure service via Ansible. You will need to create a construct known as a **service principal** in your Azure account. A service principal functions similarly to a user but is specifically designed for API-based interactions.

Begin by creating a service principal and assigning it the **Contributor** role through Azure's access management system. This role provides the necessary permissions to create and manage resources in your Azure account. To authenticate Ansible with the Azure API using the service principal, you must supply the following four key components:

- Client ID
- Tenant ID
- Subscription ID
- Service principal password

By completing these steps, you'll establish the foundation for secure and automated interaction with Azure using Ansible.



Note

If you don't have an Azure account, you can get free tier access at:

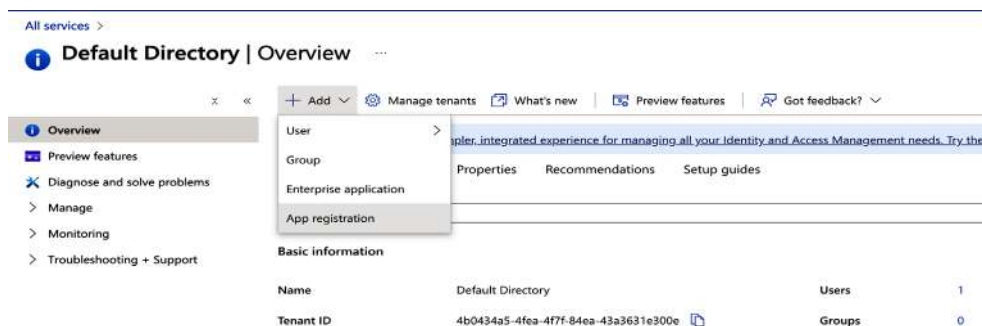
<https://azure.microsoft.com/en-au/free/>.

In this recipe, you will create the IAM programmatic credentials and store them in the Ansible environment to allow the playbooks to use them:

1. Log in to the Azure UI portal (<https://portal.azure.com/>) with an account that has administrative rights, and go to **All services** to pick **Azure Entra ID service** (previously, Azure Active Directory).



2. In the **Add** tab, select **App registration**:



3. On the app registration page, define the application name, which, in this case, is **ansible**, and click on the **Register** button.

All services > Default Directory | Overview >

Register an application

* Name

The user-facing display name for this application (this can be changed later).

ansible

Supported account types

Who can use this application or access this API?

☒ Accounts in this organizational directory only (Default Directory only - Single tenant)

4. After the application is created, you get some information required to enable programmatic access, with **Application (client) ID** and **Directory (tenant) ID**. On this page, select **Add a certificate or secret**.

All services > Default Directory | Overview >

ansible

Search

Overview

Quickstart

Integration assistant

Diagnose and solve problems

Manage

Support + Troubleshooting

Essentials

Display name	: ansible	Client credentials	: Add a certificate or secret
Application (client) ID	: 1f955c1c-6b97-4162-a529-1bb0fe2ffda	Redirect URIs	: Add a Redirect URI
Object ID	: 84723858-13f2-49ee-829a-a65606d07530	Application ID URI	: Add an Application ID URI
Directory (tenant) ID	: 4b0434a5-4fea-477f-84ea-43a3631e300e	Managed application in L...	: ansible
Supported account types	: My organization only		

5. In the **Client secrets** tab, select **New client secret**.

Certificates (0)

Client secrets (0)

Federated credentials (0)

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret

Description	New client secret	Expires	Value ⓘ	Secret ID
No client secrets have been created for this application.				

6. In the client secret form, you add the description/name for this secret and its expiry date, and click on the **Add** button:

Add a client secret

×

Description

ansible_password

Expires

Recommended: 180 days (6 months) ▾

7. The result provides you with the secret value (the only time that you will be able to see this in plain text), which is the preceding service principal password reference.

Certificates (0)

Client secrets (1)

Federated credentials (0)

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

+ New client secret

Description	Expires	Value ⓘ	Secret ID
ansible_password	5/29/2025	lpn8Q~.yJz5pVe7ZotAZ~... ⓘ	ac75c6f8-d4a0-42c5-b7f... ⓘ ⓘ

8. Next, you have to assign the required permissions to manage the services used in this chapter. You go back to **All Services** and select **Subscriptions**. There, you click your subscription name – **Azure subscription 1** in this case.

All services > Subscriptions >

Subscriptions ⓘ ...

Default Directory

+ Add ⓘ

Advanced options ▾

Global administrators can manage all subscriptions in this list by updating their policy setting [here](#).

View list of subscriptions for which you have role-based access control (RBAC) permissions to manage Azure resources. To view subscriptions for which you have billing access, [click here](#)

Showing subscriptions in Default Directory directory. Don't see a subscription? [Switch directories](#)

🔍 Search for any fi...

Subscriptions: Filtered (1 of 1)

My role == all

Status == all

⚙ Add filter

Subscription name ↑↓	Subscription ID ↑↓	My role ↑↓	Current cost	Secure Score ↑↓	Parent management group ↑↓	Status ↑↓
Azure subscription 1	a79d4c6a-8001-41fb-80dc-928450887dcf	Owner	0.00	-	Tenant Root Group	Active

- In the subscription details, record the **Subscription ID** string (also needed for authentication), and then click on **Access control (IAM)**.

Azure subscription 1 ☆ ...

Subscription

Search

Cancel subscription Rename → Change directory Feedback

Overview

- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Security
- Events
- Cost Management

Essentials

Subscription ID: a79d4c6a-8001-41fb-80dc-92f450887dcf Copy to clipboard

Directory: Default Directory (chadellgmail.onmicrosoft.com)

Status: Active

Parent management group: 4b0434a5-4fea-4f7f-84ea-43a3631e300e

Subscription name: [Azure subscription 1](#)

My role: Owner

Plan: Azure Plan

Secure Score: [Not available](#)

Warning: Your remaining \$200.00 of free credit expires in 30 days. Upgrade to keep going with your account. →

- Click on **Add role assignment**, and select and assign the **Contributor** role (in the privileges administrator roles) to the member Ansible application created before.

All services > Azure subscription 1 | Access control (IAM) >

Add role assignment

Role **Members** Conditions Review + assign

Selected role Contributor

Assign access to

☒ User, group, or service principal

☐ Managed identity

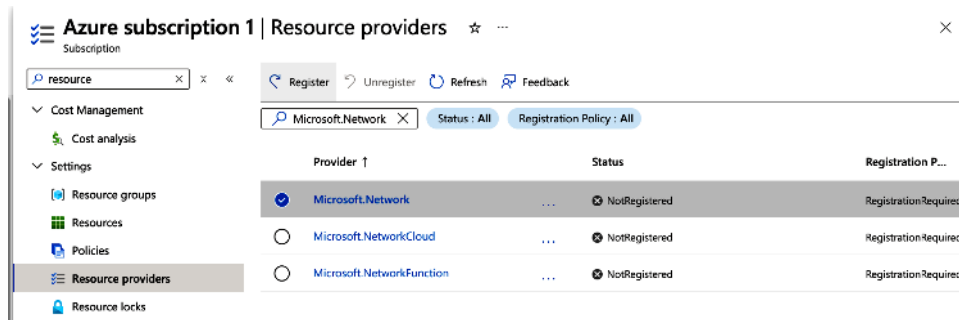
Members [+ Select members](#)

Name	Object ID	Type
ansible	ed854047-6220-447e-a5af-71621c4d5e...	App

- Finally, with `ansible-vault`, you can persist the required credentials in a YAML file. (Note: the Ansible Vault password is taken from `ansible.cfg`.)

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-vault create
azure_secret.yml
... an editor is opened ...
---
tenant_id: <your-tenant-id>
client_id: <your-client-id>
subscription_id: <your-subscription-id>
secret: <your-secret-value>
```

12. Last, but not least, to enable network resource creation, you have to register your subscription to the **Microsoft.Network** resource provider.



Note

The Azure portal services and steps may differ over time, but the idea remains the same.

At this point, you have your environment ready to manage your Azure infrastructure via Ansible. In this case, we will start deploying a resource group.

Creating a Resource Group

Resource groups are a core component of the **Azure Resource Manager (ARM) deployment model**, which is the recommended approach for provisioning and managing resources in the Azure cloud.

A resource group acts as a logical container, allowing you to organize and manage related resources that share the same lifecycle. For example, resources such as a **virtual machine (VM)**, its associated **network interface card (NIC)**, and an IP address can all be grouped together within a single resource group. This structure simplifies resource management, enabling you to efficiently deploy, monitor, and decommission related resources as a cohesive unit.

Throughout this recipe, you will focus on creating and deploying resource groups as the foundational step for provisioning Azure resources. This approach ensures that all resources are systematically organized and aligned with best practices for Azure infrastructure management:

1. The first step is to create the `group_vars` folder, define the `resource_group_name` variable for all groups, using Jinja2 rendering, and append `rg_` to the inventory hostname. In this chapter, you only have one host per group; therefore, you can concentrate all the variables in `group_vars`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch09# mkdir group_vars &
vi group_vars/all.yml
---
resource_group_name: "rg_{{ inventory_hostname }}"

```

2. Like AWS Regions, Azure organizes resources around global infrastructure locations. Add a region name for each group. This region name is a keyword for Azure to locate the resource in its regions (more info at <https://learn.microsoft.com/en-us/dotnet/api/microsoft.azure.documents.locationnames>). In this case, we use the westeurop and useast regions:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/eu.yml
---
region: "westeurope"
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/us.yml
---
region: "useast"

```

3. Create a new playbook, `pb_build_azure_net.yml`, using the `azure.azcollection.azure_rm_resourcegroup` module, which leverages the secrets taken from the `azure_secrets.yml` file, that allow using encrypted variables, such as `client_id`:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_build_azure_net.yml
---
- name: Build Azure Network Infrastructure
  hosts: all
  connection: local
  gather_facts: no
  vars_files:
    - azure_secrets.yml
  tasks:
    - name: Create Resource Group
      azure.azcollection.azure_rm_resourcegroup:
        tenant: "{{ tenant_id }}"
        client_id: "{{ client_id }}"
        secret: "{{ secret }}"
        location: "{{ region }}"
        subscription_id: "{{ subscription_id }}"

```

```
name: "{{ resource_group_name }}"
state: "{{ state | default('present') }}"
```

4. Run the playbook created in the previous step:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_build.yml
PLAY [Build Azure Network Infrastructure] *****
TASK [Create Resource Group] *****
changed: [us_az_net]
changed: [eu_az_net]
... omitted output ...
```

5. Go back to **All Services** in the Azure portal and select **Resource groups** to check that both groups have been created, for each location.

All services >

Resource groups ...

Default Directory

+ Create ⚙️ Manage view ▾ 🔄 Refresh ⬇️ Export to CSV 🔍 Open query | 🏷️ Assign tags

🕒 You are viewing a new version of Browse experience. Some features may be missing. [Click here to access the old experience.](#)

🔽 Filter for any field... Subscription equals all Location equals all ✕ + Add filter

☐	Name ↑		Subscription	Location
☐	🔗 rg_eu_az_net	...	Azure subscription 1	West Europe
☐	🔗 rg_us_az_net	...	Azure subscription 1	East US

The resource group will facilitate the decommissioning of all the next resources, as you will notice in the last section of the chapter. Now, let's start creating resources within the recently created resource group.

Creating virtual networks

Azure Virtual Networks (VNETs) serve as virtual data centers, providing a secure and isolated environment to group and manage resources such as virtual machines and databases. VNETs are comparable to **AWS Virtual Private Clouds (VPCs)**, both offering IP address management, subnets, and networking security. VNETs can span multiple regions (different than AWS VPCs), enabling use cases such as disaster recovery, regional load balancing, and multi-region deployments.

In this recipe, we'll guide you through defining and provisioning a VNet in Azure. This is a critical step in building a secure and scalable foundation for deploying your Azure resources:

1. Update the `us.yml` and `eu.yml` files under the `group_vars` directory with the virtual CIDR address (according to the network topology depicted in *Figure 9.1*), and the templated VNet name in `all.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/eu.yml
vnet_cidr: 10.1.0.0/16
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/us.yml
vnet_cidr: 10.2.0.0/16
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/all.
yml
vnet_name: "vn_{{ inventory_hostname }}"
```

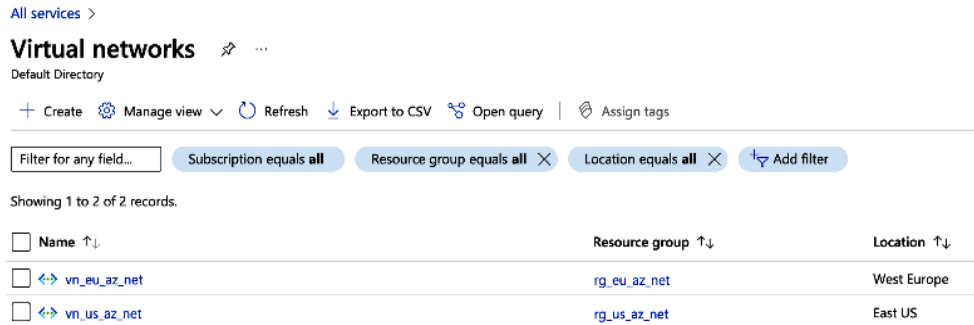
2. Update the `pb_azure_net_build.yml` playbook with the new task to create the virtual network per region using the `azure.azcollection.azure_rm_virtualnetwork` module:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_
build.yml
... omitted output ...
- name: Create Azure Virtual Networks
  azure.azcollection.azure_rm_virtualnetwork:
    tenant: "{{ tenant_id }}"
    client_id: "{{ client_id }}"
    secret: "{{ secret }}"
    location: "{{ region }}"
    subscription_id: "{{ subscription_id }}"
    resource_group: "{{ resource_group_name }}"
    name: "{{ vnet_name }}"
    address_prefixes_cidr: "{{ vnet_cidr }}"
    state: "{{ state | default('present') }}"
```

3. Run the playbook created in the previous step to create the resource groups, one per region:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_build.yml
... omitted output ...
TASK [Create Azure Virtual Networks] *****
changed: [us_az_net]
changed: [eu_az_net]
... omitted output ...
```


4. As a result, in the **Virtual networks** service in the Azure portal, you can check that both VNets have been created, one for each location.



If you check the module's required variables, it needs the same four variables for authentication. This is consistent across all the modules, so writing them every time doesn't seem the most effective way. An idea to improve this is to use YAML anchors and aliases to reuse data more effectively:

1. Create an alias in the vars section, with the content of the four variables defining the authentication for Azure:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_build.yml
... omitted output ...
vars:
  credentials: &azure_credentials
  tenant: "{{ tenant_id }}"
  client_id: "{{ client_id }}"
  secret: "{{ secret }}"
  subscription_id: "{{ subscription_id }}"
```

2. Use the data in the module via a **YAML anchor**. This simplifies the data reusing variables:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_build.yml
... omitted output ...
- name: Create Azure Virtual Networks
  azure.azcollection.azure_rm_virtualnetwork:
    <<: *azure_credentials
    location: "{{ region }}"
    resource_group: "{{ resource_group_name }}"
```

```
name: "{{ vnet_name }}"
address_prefixes_cidr: "{{ vnet_cidr }}"
state: "{{ state | default('present') }}"
```

After the VNets are created, you can move on to provisioning the inner subnets.

Creating subnets

A **subnet** in Azure is a key networking component that segments a virtual network into smaller, manageable sections. It allows us to define distinct routing and security boundaries within the network, providing fine-grained control over traffic flow and access to resources. Each subnet can have its own routing policies, **network security groups (NSGs)**, and other configurations tailored to specific application needs.

In this recipe, we will walk through the steps to define and provision subnets in Azure, ensuring proper network segmentation and security for your resources.

To create subnets, you have to associate each subnet with its parent VNet by specifying the `virtual_network_name` parameter:

1. Update the `group_vars/eu.yml` and `group_vars/us.yml` files with the subnets' data, as shown here for the EU region (check the repository for the US example):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/eu.yml
... omitted output ...
subnets:
- name: web_tier
  cidr: 10.1.1.0/24
- name: db_tier
  cidr: 10.1.2.0/24
```

2. Update the previous `pb_azure_net_build.yml` playbook with the retrieval of the VNet name, and the creation of the two subnets, using the `azure.azcollection.azure_rm_subnet` module:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_build.yml
... omitted output ...
- name: Create Azure Subnets
  azure.azcollection.azure_rm_subnet:
    <<: *azure_credentials
```

```

resource_group: "{{ resource_group_name }}"
name: "{{ item.name }}"
virtual_network_name: "{{ vnet_name }}"
address_prefix_cidr: "{{ item.cidr }}"
state: "{{ state | default('present') }}"
loop: "{{ subnets }}"
loop_control:
  label: "{{ item.name }}"

```

3. Then, you run the playbook to roll out the new updates, without changing the previous object via an idempotent approach:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_build.yml
... output omitted ...
TASK [Create Azure Subnets] *****
changed: [us_az_net] => (item=web_tier)
changed: [eu_az_net] => (item=web_tier)
changed: [us_az_net] => (item=db_tier)
changed: [eu_az_net] => (item=db_tier)

```

4. Finally, you can verify the new Azure subnets created in the Azure portal, selecting each virtual network with the new subnets.

Name	IPv4	IPv6	Available IPs
web_tier	10.1.1.0/24	-	251
db_tier	10.1.2.0/24	-	251

Next, we will tailor the subnets' routing by creating user-defined routes.

Building user-defined routes

In our setup, we have two subnets—**web** and **DB**—and we need to apply different routing treatments to the DB subnet, ensuring it doesn't have public internet access. To achieve this, we will create a custom route table and configure a default route with next-hop set to **none**, effectively blocking all traffic destined for the internet.

You'll define the custom route table in the `route_tables` variable and deploy it within each region's configuration. This will allow us to control the routing behavior and ensure the DB subnet has restricted access while the web subnet remains publicly accessible.

In this recipe, we will walk through how to use **user-defined routes (UDRs)** to control traffic routing within a subnet. These UDRs can be associated with specific subnets in Azure, allowing us to fine-tune routing and adjust forwarding behaviors as needed:

1. Because the route table definition is the same for both regions, you can define it in `group_vars/all.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/all.
yml
... omitted output ...
route_tables:
  - name: db_tier_rt
    subnet: db_tier
    routes:
      - name: Default Route
        prefix: 0.0.0.0/0
        nh: none
```

2. Then, you extend the `pb_azure_net_build.yml` playbook with the new task to build the custom route tables:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_
build.yml
... omitted output ...
- name: Create Azure Custom Route Table
  azure.azcollection.azure_rm_routetable:
    <<: *azure_credentials
    resource_group: "{{ resource_group_name }}"
    name: "{{ item.name }}"
    state: "{{ state | default('present') }}"
    loop: "{{ route_tables }}"
```

3. Still in the same playbook, you continue adding the next task to provision the routes within the previously created route table:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_
build.yml
... output omitted ...
- name: Provision Routes
  azure.azcollection.azure_rm_route:
    <<: *azure_credentials
    resource_group: "{{ resource_group_name }}"
    route_table_name: "{{ item.0.name }}"
    name: "{{ item.1.name }}"
    address_prefix: "{{ item.1.prefix }}"
    next_hop_type: "{{ item.1.nh }}"
    state: "{{ state | default('present') }}"
  with_subelements:
    - "{{ route_tables }}"
    - routes
```

4. Finally, in the playbook, the last component is to associate the custom route with the subnet using the same `azure.azcollection.azure_rm_subnet` module:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_
build.yml
... output omitted ...
- name: Attach Route Table to Subnet
  azure.azcollection.azure_rm_subnet:
    <<: *azure_credentials
    resource_group: "{{ resource_group_name }}"
    name: "{{ item.subnet }}"
    virtual_network_name: "{{ vnet_name }}"
    route_table: "{{ item.name }}"
    state: "{{ state | default('present') }}"
  loop: "{{ route_tables }}"
  loop_control:
    label: "{{ item.name }}"
```

- When you run the playbook, the three tasks are executed and the route tables and routes are created and finally associated with the subnets:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_build.yml
... output omitted ...
TASK [Create Azure Custom Route Table] *****
changed: [us_az_net] => (item={'name': 'db_tier_rt',
'subnet': 'db_tier', 'routes': [{ 'name': 'Default Route',
'prefix': '0.0.0.0/0', 'nh': 'none' } ]})
changed: [eu_az_net] => (item={'name': 'db_tier_rt',
'subnet': 'db_tier'}, { 'name':
'Default Route', 'prefix': '0.0.0.0/0', 'nh': 'none' }])
TASK [Provision Routes] *****
changed: [us_az_net] => (item=[{ 'name': 'db_tier_rt', 'subnet': 'db_
tier'}, { 'name': 'Default Route',
'prefix': '0.0.0.0/0', 'nh': 'none' }])
changed: [eu_az_net] => (item=[{ 'name': 'db_tier_rt', 'subnet': 'db_
tier'}, { 'name': 'Default Route',
'prefix': '0.0.0.0/0', 'nh': 'none' }])
TASK [Attach Route Table to Subnet] *****
changed: [eu_az_net] => (item=db_tier_rt)
changed: [us_az_net] => (item=db_tier_rt)
```

- You can check **Route tables** in **All services** to verify the new route tables.

[All services >](#)

Route tables

Default Directory

 Create  Manage view  Refresh  Export to CSV  Open query  Assign tags

Filter for any field... Subscription equals all Resource group equals all X Location equals all X  Add filter

Showing 1 to 2 of 2 records.

☐ Name ↑↓	Resource group ↑↓	Location ↑↓
☐  db_tier_rt	rg_eu_az_net	West Europe
☐  db_tier_rt	rg_us_az_net	East US

In the next screenshot, for each route table, you can check the details where `0.0.0.0/0` is dropping all the traffic because no next hop is assigned to it.

The screenshot shows the Azure portal interface for a route table named 'db_tier_rt'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Monitoring, Automation, and Help. The main content area is divided into 'Essentials' and 'Routes' sections.

Essentials

- Resource group: [\(move\)](#) [rg_eu_az_net](#)
- Location: West Europe
- Subscription: [\(move\)](#) [Azure subscription 1](#)
- Subscription ID: a79d4c6a-8001-41fb-80dc-92f450887dcf
- Tags: [\(edit\)](#) [Add tags](#)
- Associations: 1 subnet associations

Routes

Search routes

Name	Address prefix	Next hop type	Next hop IP address
Default Route	0.0.0.0/0	None	-

Subnets

Search subnets

Name	Address range	Virtual network	Security group
db_tier	10.1.2.0/24	vn_eu_az_net	-

Finally, you will create the security constructs to filter the traffic between the Azure cloud resources.

Deploying network security groups

Security in cloud environments is essential, and Azure offers various tools and services to help create a secure infrastructure for applications. One such tool is **Network Security Groups (NSGs)**, which act as stateful firewalls that can be attached to VMs or subnets to control traffic flow.

Azure provides default NSGs, which are automatically applied to subnets, offering basic security controls for resources deployed within those subnets. By default, it allows traffic between virtual network CIDR ranges (enabling inter-subnet communication), plus inbound traffic from Azure load balancers and outbound traffic to the internet, finally denying all other traffic.

While these default rules provide basic security, *custom NSGs* can be used to further refine security policies. Custom NSGs are evaluated based on the priority of each rule (lower values are evaluated first). Once a rule is matched, it is applied to traffic traversing the subnet.

For instance, when deploying a web application in the `web_tier` subnet, we need to allow inbound HTTP and HTTPS traffic. This can be achieved by defining a custom NSG with rules that allow these specific types of inbound traffic. In this recipe, you will create a custom NSG and configure the necessary rules to secure the `web_tier` subnet:

1. In each region, you create the NSG definition, updating `group_vars/eu.yml` and `group_vars/us.yml`. In this example, you can see the EU data, and the corresponding US data can be found in the chapter repository:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi group_vars/eu.yml
... omitted output ...
acls:
  - name: Inbound_Web_Tier
    subnet: web_tier
    rules:
      - name: Allow_HTTP_Internet
        destination_address_prefix: 10.1.1.0/24
        direction: Inbound
        access: Allow
        protocol: Tcp
        destination_port_range:
          - 80
          - 443
        priority: 101
```

2. Then, you add a new task to the `pb_aws_net_build.yml` playbook to create the NSG using the `azure.azcollection.azure_rm_securitygroup` module:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_build.yml
... omitted output ...
  - name: Create Network Security Group
    azure.azcollection.azure_rm_securitygroup:
      <<: *azure_credentials
      resource_group: "{{ resource_group_name }}"
      name: "{{ item.name }}"
      purge_rules: yes
      rules: "{{ item.rules }}"
      loop: "{{ acls }}"
```


3. Then, the network security group has to be attached to the target subnet (`web_tier`):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_aws_net_build.
yaml
... omitted output ...
- name: Attach Network Security Group to Subnet
  azure.azcollection.azure_rm_subnet:
    <<: *azure_credentials
    resource_group: "{{ resource_group_name }}"
    name: "{{ item.subnet }}"
    virtual_network_name: "{{ vnet_name }}"
    security_group: "{{ item.name }}"
    state: "{{ state | default('present') }}"
    loop: "{{ acs }}"
```

4. Finally, you run the playbook to create and associate the NSG to the `web_tier` subnet:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_build.yaml
... output omitted ...
TASK [Create Network Security Group] *****
changed: [us_az_net] => (item={'name': 'Inbound_Web_
Tier', 'subnet': 'web_tier', 'rules': [{'name': 'Allow_HTTP_
Internet', 'destination_address_prefix': '10.2.1.0/24', 'direction':
'Inbound', 'access': 'Allow', 'protocol': 'Tcp', 'destination_port_
range': [80, 443], 'priority': 101}]})
changed: [eu_az_net] => (item={'name': 'Inbound_Web_
Tier', 'subnet': 'web_tier', 'rules': [{'name': 'Allow_
HTTP_Internet', 'destination_address_
prefix': '10.1.1.0/24', 'direction':
'Inbound', 'access': 'Allow', 'protocol': 'Tcp', 'destination_port_
range': [80, 443], 'priority': 101}]})
TASK [Attach Network Security Group to Subnet] *****
changed: [us_az_net] => (item={'name': 'Inbound_Web_
Tier', 'subnet': 'web_tier', 'rules': [{'name': 'Allow_HTTP_
Internet', 'destination_address__prefix': '10.2.1.0/24',
'direction': 'Inbound', 'access': 'Allow',
'protocol': 'Tcp', 'destination_port_
range': [80, 443], 'priority': 101}]})
changed: [eu_az_net] => (item={'name': 'Inbound_Web_
```

```
Tier', 'subnet': 'web_tier', 'rules': [{ 'name': 'Allow_HTTP_
Internet', 'destination_address_prefix': '10.1.1.0/24',
'direction': 'Inbound', 'access': 'Allow',
'protocol': 'Tcp', 'destination_port_
range': [80, 443], 'priority': 101}]])
```

5. In the **Network security groups** service (in **All services**), you can review the created groups:

Network security groups

Default Directory

+ Create Manage view Refresh Export to CSV Open query Assign tags

Filter for any field... Subscription equals all Resource group equals all Location equals all Add filter

Showing 1 to 2 of 2 records.

Name	Resource group	Location	Subscription
Inbound_Web_Tier	rg_eu_az_net	West Europe	Azure subscription 1
Inbound_Web_Tier	rg_us_az_net	East US	Azure subscription 1

In the following screenshot, you can check the resulting NSG rules defined (both custom and default) for inbound and outbound directions:

Inbound_Web_Tier

Network security group

Search Move Delete Refresh Give feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems Settings Monitoring Automation Help

Essentials

Resource group (moved) : rg_eu_az_net Location : West Europe Subscription (moved) : Azure subscription 1 Subscription ID : a79d4cfa-8001-41b1-80d6-92f453887dcd Tags (edit) : Add tags

Custom security rules : 1 inbound, 0 outbound Associated with : 1 subnets, 0 network interfaces

Filter by name Port == all Protocol == all Source == all Destination == all Action == all

Priority	Name	Port	Protocol	Source	Destination	Action
Inbound Security Rules						
101	Allow_HTTP_Internet	80,443	Tcp	Any	10.1.1.0/24	Allow
65000	AllowVnetInbound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowAzureLoadBalancerInbound	Any	Any	AzureLoadBalancer	Any	Allow
65503	DenyAllInbound	Any	Any	Any	Any	Deny
Outbound Security Rules						
65000	AllowVnetOutbound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowInternetOutbound	Any	Any	Any	Internet	Allow
65503	DenyAllOutbound	Any	Any	Any	Any	Deny

After creating all these services, the next step is to validate that the actual operational state in Azure is the expected one.

Deployment validation

Ansible offers a variety of modules to gather the operational state of resources deployed in Azure, providing a programmatic method to validate the current status of network components. This eliminates the need to log into the Azure portal manually to check the status of different infrastructure components. In this recipe, we will demonstrate how to use several of these modules to validate the resource groups and virtual networks deployed in Azure.

Additionally, we will cover an alternative authentication method to connect to Azure. Instead of specifying authentication parameters (such as `tenant_id`, `client_id`, etc.) directly in the Ansible playbook, we will store these details in the `~/ .azure/credentials` file. With this approach, Ansible modules can automatically authenticate to the Azure API without needing explicit credentials in each task.

To validate the deployment, Ansible provides fact-gathering modules, such as `azure_rm_resourcegroup_info` and `azure_rm_virtualnetwork_info`, which collect detailed information about resource groups and virtual networks. These modules register the data as Ansible facts, meaning there's no need to manually store the retrieved information in custom variables. This makes the process of gathering and utilizing cloud resource data more streamlined and efficient.

The next steps show you how to define and validate the deployment, using a convention file to store Azure secrets, avoiding specifying them directly in the playbook:

1. Create a new file, `~/ .Azure/credentials`, to store environmental secrets:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi ~/.Azure/credentials
[default]
subscription_id=XXX-XXXX-XXXX
client_id=XXX-XXXX-XXXX
secret=XXX-XXXX-XXXX
tenant=XXX-XXXX-XXXX
```

2. Create a new playbook, `pb_azure_net_validate.yml`, to validate your deployment and include the following tasks to collect resource group info and validate it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_validate.yml
---
- name: Validate Azure Network Infrastructure
  hosts: all
```

```
connection: local
gather_facts: no
tasks:
  - name: Get Resource Group Facts
    azure.azcollection.azure_rm_resourcegroup_info:
      name: "{{ resource_group_name }}"
      register: azure_resourcegroups
  - name: Validate Resource Group is Deployed
    ansible.builtin.assert:
      that:
        - rg.name == resource_group_name
        - rg.properties.provisioning_state == 'Succeeded'
        - rg.location == region
      loop: "{{ azure_resourcegroups.resourcegroups }}"
      loop_control:
        loop_var: rg
```

3. Update the `pb_azure_net_validate.yml` playbook to include the tasks that will collect the virtual network's facts and validate its state:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_validate.yml
... omitted output ...
  - name: Get Virtual Network Facts
    azure.azcollection.azure_rm_virtualnetwork_info:
      resource_group: "{{ resource_group_name }}"
      register: azure_virtualnetworks

  - name: Validate Virtual Networks are Deployed
    ansible.builtin.assert:
      that:
        - vnet.name == vnet_name
        - vnet.provisioning_state == 'Succeeded'
        - vnet.address_prefixes | length == 1
        - vnet.address_prefixes[0] == vnet_cidr
      loop: "{{ azure_virtualnetworks.virtualnetworks }}"
      loop_control:
        loop_var: vnet
```

4. In the AWS console, you can see the new NACL with the inbound and outbound rules created:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook
pb_azure_net_validate.yml
PLAY [Validate Azure Network Infrastructure] *****
TASK [Get Resource Group Facts] *****
ok: [us_az_net]
ok: [eu_az_net]
TASK [Validate Resource Group is Deployed] *****
ok: [us_az_net] => (item={'id': '/subscriptions/
a79d4c6a-8001-41fb-80dc-92f450887dcf/resourceGroups/
rg_us_az_net', 'name': 'rg_us_az_net', 'type': 'Microsoft.
Resources/resourceGroups', 'properties': {'provisioning_
state': 'Succeeded'}, 'location': 'eastus'}) => {
    "ansible_loop_var": "rg",
    "changed": false,
    "msg": "All assertions passed",
    "rg": {
        "id": "/subscriptions/a79d4c6a-8001-41fb-80dc-92f450887dcf/
resourceGroups/rg_us_az_net",
        "location": "eastus",
        "name": "rg_us_az_net",
        "properties": {
            "provisioning_state": "Succeeded"
        },
        "type": "Microsoft.Resources/resourceGroups"
    }
}
ok: [eu_az_net] => ... omitted output ...
TASK [Get Virtual Network Facts] *****
ok: [us_az_net]
ok: [eu_az_net]
TASK [Validate Virtual Networks are Deployed] *****
ok: [us_az_net] => ... omitted output ...
ok: [eu_az_net] => ... omitted output ...

```

**Note**

When working with data structures registered by modules, the format may change from version to version. For example, in `vnet.address_prefixes[0]`, we are using a dictionary (`vnet`) in which the `address_prefixes` key contains a list, and the first item is picked. To understand data structures, we recommend you use the `debug` function.

As you already know, when managing dynamic infrastructure, the deletion of resources and services that are no longer needed is just as important as their creation. This is going to be the last recipe of the chapter.

Decommissioning Azure resources

Just as automation allows us to provision resources at scale, it also enables us to efficiently decommission them when no longer needed. With Azure and Ansible, this process is simplified—by making a single API call with the appropriate parameters, we can delete all resources within a resource group. In this recipe, we will demonstrate how to destroy the resources you’ve provisioned by decommissioning the entire resource group. (Remember we mentioned that grouping resources under a resource group is a best practice? Here comes the *why* it is so.)

To accomplish this, we can use the `azure_rm_resourcegroup` Ansible module, which allows you to delete both the resources within a resource group and the resource group itself. Two key parameters must be specified to ensure a successful deletion:

- Set `state` to **absent** to indicate that the resource group should be deleted
- Set the `force_delete_nonempty` parameter to **yes** to ensure that even non-empty resource groups are removed, deleting all contained resources

This approach allows for a clean and efficient way to tear down resources, ensuring we can manage both provisioning and decommissioning seamlessly.

Follow the next steps to define and run a playbook to clean up the dynamic infrastructure:

1. Create a new `pb_azure_net_destroy.yml` playbook and add the following task to delete all the resource groups:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# vi pb_azure_net_
delete.yml
---
```

```
- name: Decommission Azure Infrastructure
  hosts: all
  connection: local
  gather_facts: no
  vars_files:
    - azure_secrets.yml
  tasks:
    - name: Delete Resource group
      azure.azcollection.azure_rm_resourcegroup:
        tenant: "{{ tenant_id }}"
        client_id: "{{ client_id }}"
        secret: "{{ secret }}"
        location: "{{ region }}"
        subscription_id: "{{ subscription_id }}"
        name: "{{ resource_group_name }}"
        force_delete_nonempty: yes
        state: absent
```

2. With a simple Ansible run, all the infrastructure is removed from your Azure account:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch09# ansible-playbook pb_
azure_net_destroy.yml
PLAY [Decommission Azure Infrastructure] *****
TASK [Delete Resource group] *****
changed: [us_az_net]
changed: [eu_az_net]
```

Note



Notice that while in AWS you had to delete each resource individually in reverse order, in Azure, you can delete the single resource group and all contained resources are removed automatically. This highlights a key difference in the resource management models of these two cloud providers.

Now, if you check your Azure portal, you will not find any of the network resources created before.

Summary

Similar to what you saw in *Chapter 8*, the Ansible Azure collection provides modules that facilitate managing dynamic infrastructure with an idempotent approach. This ensures that playbooks can be executed repeatedly while delivering consistent results, only modifying resources when necessary to align with the desired state.

All the chapters in *Part 3* have common storytelling to facilitate you comparing how to interact with them, with some hints about each one, such as using resource groups in Azure to facilitate managing related infrastructure.

In the next chapter, you will learn how to automate the other big cloud provider, **Google Cloud Platform (GCP)**.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



10

Deploying and Operating GCP Networking

Building on the approach we used in the previous chapters to manage network resources in AWS (*Chapter 8*) and Azure (*Chapter 9*), this chapter shifts focus to **Google Cloud Platform (GCP)**. You will continue with a similar methodology, demonstrating how to automate resource provisioning in GCP using Ansible. Additionally, you will explore the various Ansible modules available for orchestrating the creation and management of virtual networks within the GCP cloud.

You will use the simple network setup illustrated in the following diagram, depicting a traditional web architecture with different layers:

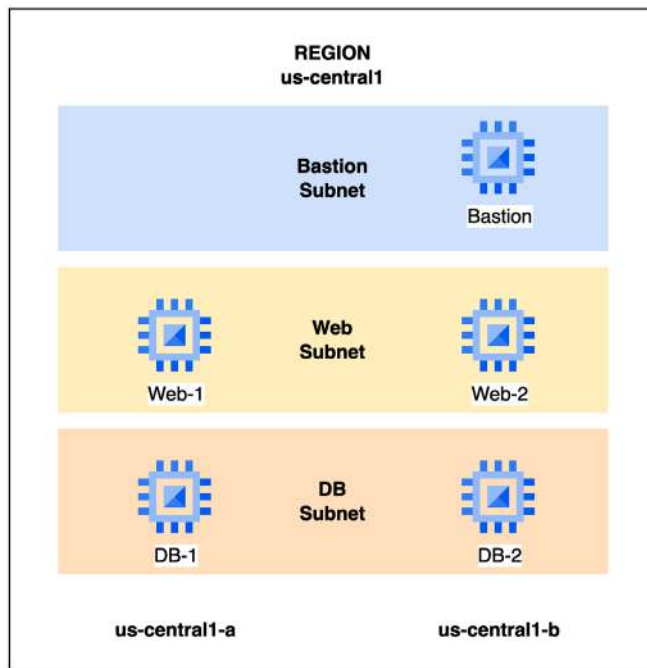


Figure 10.1 – GCP networking scenario

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



This chapter will cover the following recipes:

- Installing the GCP Python SDK
- Building an Ansible inventory
- Authenticating to your GCP account
- Creating GCP VPC networks
- Creating subnets
- Deploying firewall rules in GCP
- Deploying VMs in GCP
- Adjusting routing within a GCP VPC
- Validating a GCP deployment
- Decommissioning GCP resources



Info

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch10.

Firstly, you will prepare the environment to run the Ansible playbooks.

Installing the GCP Python SDK

According to the Ansible Community documentation (<https://docs.ansible.com/ansible/latest/index.html>), the Ansible GCP modules require both the `requests` and `google-auth` Python libraries to be installed. The first is a popular package to interact with REST APIs (e.g., the Google orchestration system), and the second is the Google package to authenticate against their respective APIs.

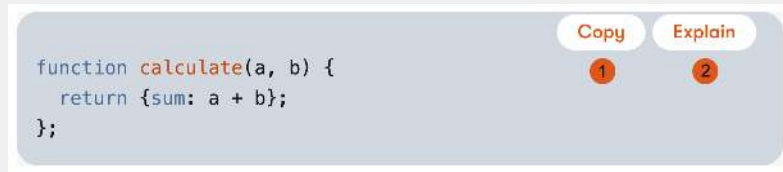
1. First, let's check that the `google.cloud` Ansible collection is installed. This step is important because, at the time of writing, there is an open issue that will remove this collection from the Ansible package from Ansible 12 (because of a requirements violation). If it's not installed, you can install it with `ansible-galaxy collection install google.cloud`:

```
root@cookbook-lab:~/cookbook-lab# ansible-galaxy collection list
google.cloud
# /usr/local/lib/python3.10/dist-packages/ansible_collections
Collection  Version
-----
google.cloud 1.4.1
```

💡 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button

(1) to quickly copy code into your coding environment, or click the **Explain** button

(2) to get the AI assistant to explain a block of code to you.



The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



2. As root, install the requests and google-auth packages:

```
root@cookbook-lab:~/cookbook-lab# pip install requests google-auth
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (2.32.3)
... omitted output ...
Successfully installed cachetools-5.5.0 google-auth-2.37.0 rsa-4.9
```

3. Finally, let's create the folder to host all the code for this chapter, called chapters/ch10:

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch10
```

Note

In this chapter, we will use the official Google Cloud Ansible collection. For more information, check out: <https://docs.ansible.com/ansible/latest/collections/google/cloud/index.html#plugins-in-google-cloud>.

If, at some point, the collection is removed from the Ansible package, it will still be available on the public Galaxy repository at: <https://galaxy.ansible.com/ui/repo/published/google/cloud/>, and you can install it via `ansible-galaxy collection install google.cloud`.

Once the execution dependencies are resolved, the next step is to set up the required inventory configuration.

Building the Ansible inventory

In this recipe, you will create a simple group to include the virtual network (VPC) and leverage Ansible roles to complement it to achieve the expected results.

Note

Like in *Chapters 8 and 9*, we use a static inventory, but a dynamic approach using a GCP dynamic inventory could be used to manage existing infrastructure.

You can reuse the settings in the `ansible.cfg` file used in previous recipes (for example, *Chapters 8 and 9*), specifying the vault password file that stores the encryption password for securing sensitive information.

1. Create a hosts files inside the `ch10` folder with the following data:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi hosts
[gcp_vpc]
    demo_gcp_vpc
```

2. Reuse the `ansible.cfg` file from other chapters (for example, *Chapter 9*):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi ansible.cfg
[defaults]
    inventory=hosts
    vault_password_file=~/.ansible_vault_passwd
```

```
gathering=explicit
transport=local
retry_files_enabled=False
action_warnings=False
```

Next, you have to get programmatic access to GCP.

Authenticating to your GCP account

You have to create a **service account** within your GCP project to enable programmatic access to the GCP API—which is how Ansible interacts with the GCP cloud to provision resources. A service account functions similarly to a user but is exclusively designed for API access. In our setup, we created a service account named *Ansible* and assigned it the **Project Owner** role, granting it *full privileges* to create and manage resources in the project. However, in production environments, assigning a more restrictive role to limit its permissions is recommended.

For authentication, GCP provides a JSON file containing the credentials and identification information for the service account. This file is essential for authenticating to the GCP API and includes key details such as the following:

- The private key associated with the service account
- The account type
- `project_id`
- `client_id`
- `client_email`

This JSON file serves as the bridge for securely connecting to GCP and enabling automated resource management with Ansible. We will save this JSON file and copy it to our directory (not committing it to the repository), as it will be referenced in all our playbooks to provision resources on the GCP cloud. However, as you may guess, this information shouldn't be leaked, so you will use Ansible Vault to secure this file and access it encrypted.

The final step is to enable the API within our GCP project. Specifically, you need to enable the GCP Compute Engine API to start interacting with the GCP cloud, as API access is disabled by default within the project.

**Note**

If you don't have a GCP account, you can get free credit at: <https://console.cloud.google.com/freetrial/signup/tos>.

Let's get started:

1. Log in to the GCP console (<https://console.cloud.google.com/>) with an administrative account.
2. From the main console, search for **Manage Resources** (part of **IAM & admin**), and then click on **Create Project**, which creates the environment that will house all the infrastructure you will build in GCP.

New Project



You have 10 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name *

gcp-automation-demo



Project ID: gcp-automation-demo. It cannot be changed later. [EDIT](#)

Location *

 No organisation

[BROWSE](#)

Parent organisation or folder

3. Now, search for **Service accounts** (also part of **IAM & admin**) and make sure the new **gcp-automation-demo** project is selected (you may have other projects). Click on **Create Service Account** for the new ansible user, enter the data, and select **CREATE AND CONTINUE** to move on to the role assignment:

[←](#) Create service account

1

Service account details

Service account name

ansible

Display name for this service account

Service account ID *

ansible

X C

Email address: ansible@gcp-automation-demo.iam.gserviceaccount.com

📧

Service account description

Service Account for Ansible Automation

Describe what this service account will do

CREATE AND CONTINUE

2

Grant this service account access to the project (optional)

3

Grant users access to this service account (optional)

DONE

CANCEL

4. In the second step, assign the appropriate role (**Owner**) to this new service account so it will be able to create, edit, and delete resources in this GCP project. Click on the **DONE** button, because the next, optional step can be skipped as you are not managing users here.

✓

Service account details

2

Grant this service account access to the project (optional)

Grant this service account access to gcp-automation-demo so that it has permission to complete specific actions on the resources in your project. [Learn more](#)

Role

Owner

Full access to most Google Cloud resources See the list of included permissions.

IAM condition (optional) ?

+ ADD IAM CONDITION

🗑

+ ADD ANOTHER ROLE

CONTINUE

3

Grant users access to this service account (optional)

DONE

CANCEL

- Back to the **Service accounts** panel, in the **Actions** column, select the three vertical dots and then **Manage keys**. Within the **Keys** section, click on **Add Key** and then **Create new key**. When asked about the key type, select **JSON** and then **Create**. This will automatically save the JSON file on your computer. The service account information can't be retrieved after this. If you lose it, you have to remove the key and create a new one.

Create private key for 'ansible'

Downloads a file that contains the private key. Store the file securely because this key cannot be recovered if lost.

Key type

☒ JSON

Recommended

☐ P12

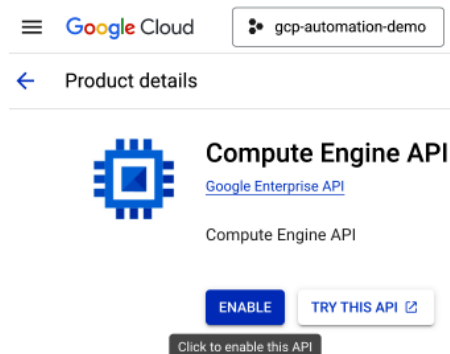
For backward compatibility with code using the P12 format

CANCEL CREATE

- In the JSON file, you have the necessary information to authenticate the GCP API, such as the private key and `client_id`. Obviously, this is sensitive information that can't be left in plain text. To allow Ansible to access the JSON file, copy the JSON file into the `ch10` folder with the name `gcp-ansible-secret.json`, and encrypt it using Ansible Vault:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-vault
encrypt gcp-ansible-secret.json
Encryption successful
```

- Finally, in the GCP console, look for/select **APIs & Services** and choose **Enable APIs and Services**. Then, search for **Compute Engine API** to enable it in the current project.



**Note**

The GCP console services and steps may differ over time, but the concepts can be reused by adapting to the new service parameters.

With the GCP project defined and the services account with API access, you are ready to start creating GCP resources. The first one is the base network, the VPC (this term is the same as in AWS, but there are others that differ, as you will see next).

Creating GCP VPC networks

In GCP, VPCs serve as the foundational networking construct, functioning as a virtual data center in the cloud. They provide a centralized framework to group and organize all cloud resources, enabling seamless communication and connectivity. Defining and configuring VPCs is a crucial step in preparing your cloud environment to host applications securely and efficiently.

In this recipe, you will define and provision a VPC in GCP, setting the stage for deploying your cloud infrastructure:

1. Create a new YAML file called `gcp_account_info.yml` and include the following data according to your GCP authentication-related parameters (it will be used for every API call made by the GCP modules). Notice that there is a reference to a JSON file with the credentials (but not the encrypted one!) and other relevant information to determine, within GCP, which environment you are targeting:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi gcp_account_info.
yml
---
service_account_file: gcp-ansible-secret-decrypted.json (1)
project: gcp-automation-demo (2)
auth_kind: serviceaccount (3)
```

The numbered references from the output are explained here:

- (1) The JSON file with the decrypted authentication content. This decrypted file does not exist on the disk permanently. It's created by the playbook in memory for the modules to use, and then a handler ensures it's deleted, so the plaintext credentials are never stored.
- (2) The name of the GCP project created.
- (3) It specifies the type of authentication, which, in this case, is `serviceaccount`.

2. Create the `group_vars` folder and, within it, the `gcp_vpc.yml` file with the data for the VPC:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
---
vpc_name: ansible-demo-vpc
```

3. Then, create a new Ansible role called `gcp_net_build` that will contain all the related Ansible artifacts using the built-in `ansible-galaxy` initialization, which takes care of all the boilerplate within a folder with the `gcp_net_build` name (in this chapter, we only use the tasks part). This role will be used from the main playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# cd roles && ansible-galaxy init gcp_net_build
- Role gcp_net_build was created successfully
```

4. In the role, update the already-existing `tasks/main.yml` file with the following task to create the VPC:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10/roles# vi gcp_net_build/tasks/main.yml
---
# tasks file for gcp_net_build
- name: Create a New GCP VPC
  google.cloud.gcp_compute_network:
    name: "{{ vpc_name | regex_replace('_', '-') }}" (1)
    routing_config:
      routing_mode: "REGIONAL" (2)
    auto_create_subnetworks: no (3)
    state: present
    auth_kind: "{{ auth_kind }}"
    project: "{{ project }}"
    service_account_file: "{{ service_account_file }}"
  register: gcp_vpc
```

The numbered references from the output are explained here:

(1) The `regex_replace` Ansible filter makes sure that the VPC name doesn't contain the underscore character (`_`), since it is not a valid character in the VPC name. We use this filter to replace any occurrence of the underscore with a dash (`-`).

- (2) Set the routing mode to `REGIONAL` to prevent route propagation between subnets across regions. It's important to note that VPCs in GCP have a *global scope*, meaning they span all regions within the GCP cloud. In contrast, subnets are region-specific. Since custom VPCs do not include subnets by default, we have complete control over where to define them. This behavior contrasts with AWS, where VPCs are region-bound.
 - (3) Set the autocreation of subnets to `no` to create a custom VPC rather than an auto-mode VPC.
5. Create the main `pb_gcp_env_build.yml` playbook, starting with a task that reads the encrypted file and decrypts it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_build.
yml
---
- name: Build GCP Environment
  hosts: all
  connection: local
  gather_facts: no
  force_handlers: True (1)
  vars_files:
    - gcp_account_info.yml
  tasks:
    - name: Read the Vault Encrypted JSON File
      copy: (2)
        content: "{{ lookup('file','gcp-ansible-secret.json') }}"
        dest: "{{ service_account_file }}"
      notify: Clean tmp Decrypted Files (3)
```

The numbered references from the output are explained here:

(1) The `force_handlers` directive at the play level guarantees that all handler tasks run, even if other tasks fail. This will ensure the plaintext file containing sensitive credentials is always deleted.

(2) Since you used Ansible Vault to encrypt the contents of the JSON file containing the authentication information, the file must be decrypted during playbook execution to access its data. However, instead of directly referencing the JSON file, you will leverage the `service_account_file` parameter in all GCP modules. To handle this securely, you can create a task to decrypt the file using the `lookup` module and store the plaintext data in a

temporary file. This temporary JSON file is then used as input for the `service_account_file` parameter.

(3) Notify the Ansible handler to run when a state changes.

6. In the same playbook, add a new task to create the required VPCs using the role:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_build.
yaml
... omitted output ...
  - name: Build GCP Network
    ansible.builtin.import_role:
      name: gcp_net_build
```

7. Then, to ensure secure cleanup, employ an Ansible handler task to delete the temporary file at the end of the play:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_build.
yaml
... omitted output ...
  handlers:
    - name: Clean tmp Decrypted Files
      ansible.builtin.file:
        path: "{{ service_account_file }}"
        state: absent (1)
```

The numbered reference from the output is explained here:

(1) The `absent` state signals that the file has to be removed.

(2) Execute the created playbook to create the new VPC:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook pb_
gcp_env_build.yaml
PLAY [Build GCP Environment] *****
TASK [Read the Vault Encrypted JSON File] *****
changed: [demo_gcp_vpc]
TASK [gcp_net_build : Create a New GCP VPC] *****
changed: [demo_gcp_vpc]
RUNNING HANDLER [Clean tmp Decrypted Files] *****
changed: [demo_gcp_vpc]
```

- 8. (Optional) You can use the `ansible.builtin.debug` module to check the registered output (the `gcp_vpc` variable) of the VPC creation. You can see the relevant information to use in the following tasks when creating subnets:

```
TASK [gcp_net_build : Check GCP VPC] *****
ok: [demo_gcp_vpc] => {
  "gcp_vpc": {
    "autoCreateSubnetworks": false,
    "changed": true,
    "creationTimestamp": "2024-12-15T06:43:50.813-08:00",
    "failed": false,
    "id": "5512786008677242313",
    "kind": "compute#network",
    "name": "ansible-demo-vpc",
    "networkFirewallPolicyEnforcementOrder": "AFTER_CLASSIC_
FIREWALL",
    "routingConfig": {
      "bgpBestPathSelectionMode": "LEGACY",
      "routingMode": "REGIONAL"
    },
    "selfLink": "https://www.googleapis.com/compute/v1/projects/
gcp-automation-demo/global/networks/ansible-demo-vpc",
    ... omitted output ...
  }
}
```

- 9. Finally, in the GCP console, you can see the brand-new VPC network created (`ansible-demo-vpc`).

VPC networks

 Filter	Enter property name or value					
Name ↑	Subnets	MTU ?	Mode	IPv6 ULA range	Gateways	Firewall rules
ansible-demo-vpc	0	1460	Custom			0
default	40	1460	Auto			4

**Note**

By default, there is a **default** VPC network, but we recommend not using it and instead owning your resource definition.

With the VPC created, you can move on to creating the inner subnets.

Creating subnets

GCP VPCs are organized with **subnets**, which enable the deployment of compute workloads in specific regions. Subnets also allow us to segment our virtual network into distinct routing and security domains, giving us fine-grained control over routing and security policies. This approach ensures that each subnet can have tailored behavior to meet specific requirements.

In this recipe, you will define and provision subnets within the GCP cloud using the `google.cloud.gcp_compute_subnet` module:

1. Update the `group_vars/gcp_vpc.yml` file with the subnets' data, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
region: us-central1
subnets:
  - name: usc-web
    cidr: 192.0.1.0/24
  - name: usc-db
    cidr: 192.0.2.0/24
  - name: usc-bastion
    cidr: 192.0.3.0/24
```

2. Update the previous `roles/gcp_net_build/tasks/main.yml` module to create the subnets according to the previous data:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi roles/gcp_net_build/tasks/main.yml
... omitted output ...
- name: Create Subnets
  google.cloud.gcp_compute_subnetwork:
    name: "{{ subnet.name }}"
```



```

    ip_cidr_range: "{{ subnet.cidr }}"
    network: "{{ gcp_vpc }}" (1)
    region: "{{ region }}"
    state: present
    auth_kind: "{{ auth_kind }}"
    project: "{{ project }}"
    service_account_file: "{{ service_account_file }}"
  loop: "{{ subnets }}"
  loop_control:
    loop_var: subnet
    label: "{{ subnet.cidr }}"
  register: gcp_subnets

```

The numbered reference from the output is explained here:

(1) The reference for our VPC that you want this subnet to be part of. You get this parameter from the output when creating the VPC. You supply the `gcp_vpc` variable, the registered variable from our VPC creation task.

3. Then, you run the playbook to create the GPC subnets:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_azure_net_build.yml
... output omitted ...
TASK [gcp_net_build : Create Subnets] *****
changed: [demo_gcp_vpc] => (item=192.0.1.0/24)
changed: [demo_gcp_vpc] => (item=192.0.2.0/24)
changed: [demo_gcp_vpc] => (item=192.0.3.0/24)
... output omitted ...

```

4. Finally, let's check that the three subnets have been created within the VPC in the region specified (`us-central1`).

ansible-demo-vpc

OVERVIEW

SUBNETS

STATIC INTERNAL IP ADDRESSES

Subnets

 ADD SUBNET MANAGE FLOW LOGS ▾ Filter Enter property name or value

☐	Name ↑	Region	Stack type	Primary IPv4 range
☐	usc-bastion	us-central1	IPv4 (single-stack)	192.0.3.0/24
☐	usc-db	us-central1	IPv4 (single-stack)	192.0.2.0/24
☐	usc-web	us-central1	IPv4 (single-stack)	192.0.1.0/24

Next, you will change your focus to defining the firewall rules applied to the VPC communication flows.

Deploying firewall rules in GCP

GCP offers a variety of tools to enforce security within customer environments. Among these, firewall rules are one of the fundamental security mechanisms provided by GCP, serving as the first line of defense for all workloads within a VPC. In this recipe, you will walk through defining and provisioning firewall rules in the GCP cloud.

In GCP, **firewall rules** are stateful, meaning they automatically track the state of network connections and apply rules accordingly. These rules are applied to hosts within a VPC and can operate in either the ingress or egress direction. By default, GCP creates the following firewall rules for any new custom VPC:

- **Ingress direction:** A default rule denies all incoming traffic destined for any host within the VPC
- **Egress direction:** A default rule allows all outgoing traffic from any host within the VPC

With these default rules, and since GCP's firewall rules are stateful, any traffic initiated by a host within the VPC to the outside world is allowed. However, any incoming traffic initiated from outside the VPC is rejected unless explicitly permitted.

GCP firewall rules can match traffic based on several criteria:

- Source and destination IPv4 ranges
- IP protocol numbers
- TCP/UDP port numbers
- Network tags

Most of these criteria are straightforward, except for network tags. **Network tags** are metadata that can be assigned to any host within a VPC to identify and group them. These tags allow you to apply firewall rules selectively, ensuring the rules are enforced only for specific subsets of hosts.

Using this information, you will implement the following security policy for our sample network:

- Allow SQL traffic exclusively from web servers to database servers
- Allow HTTP/HTTPS traffic only to web servers
- Restrict SSH access from the outside world to bastion hosts only
- Limit SSH access to web and database servers to bastion hosts

To enforce these policies, you will define the firewall rules in a data structure called `fw_rules`, which contains a list of all the rules that need to be applied to the VPC. Network tags were used throughout these policies to ensure each rule was applied to the correct set of hosts within the VPC.

1. Create the `fw_rules` data structure in `group_vars/gcp_vpc.yml` and add the rule to secure traffic between the web and database layers:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
fw_rules:
  - name: allow_sql_from_usc-web_to_usc-db
    type: allow
    direction: ingress
    priority: 10 (1)
    apply_to: usc-db
    src_tag: usc-web
    dest_tag:
    protocol: tcp
    port: 3389
    state: present
```

The numbered reference from the output is explained here:

- (1) The priority will be used to determine the precedence. By default, rules have a value of 1000. The lower the value, the more precedence it has when processing.
2. Similarly, continue updating `group_vars/gcp_vpc.yml`, now with the firewall rules to secure traffic toward the web layer:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
- name: allow_internet_to-usc-web
  type: allow
  direction: ingress
  priority: 10
  src: 0.0.0.0/0
  apply_to: usc-web
  protocol: tcp
  port: 80,443
  state: present
```

3. To complete the firewall rule, update `group_vars/gcp_vpc.yml` with the rules to allow SSH only to bastion hosts:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
- name: allow_ssh_to_usc-bastion
  type: allow
  direction: ingress
  priority: 10
  src: 0.0.0.0/0
  apply_to: usc-bastion
  protocol: tcp
  port: 22
  state: present
- name: allow_ssh_from_bastion_only
  type: allow
  direction: ingress
  priority: 10
```

```
src_tag: usc-bastion
apply_to: usc-web,usc-db
protocol: tcp
port: 22
state: present
```

4. Next, update the `roles/gcp_net_build/tasks.main.yml` file with the following task to create all the required firewall rules:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
roles/gcp_net_build/tasks.main.yml
... output omitted ...
- name: Create Allow Firewall Rules
  google.cloud.gcp_compute_firewall:
    name: "{{ rule.name | regex_replace('_', '-') }}"
    network: {selfLink: "{{ gcp_vpc.selfLink }}}"} (1)
    priority: "{{ rule.priority | default(omit) }}" (2)
    direction: "{{ rule.direction | upper | mandatory }}"
    allowed:
      - ip_protocol: "{{ rule.protocol }}"
        ports: "{{ (rule.port|string).split(',') }}"
        source_ranges: "{{ rule.src | default(omit) }}"
        source_tags: "{{ omit if rule.src_tag is not defined else rule.
src_tag.split(',') }}"
        destination_ranges: "{{ rule.dest | default(omit) }}"
        target_tags: "{{ omit if rule.apply_to is not defined else rule.
apply_to.split(',') }}"
        auth_kind: "{{ auth_kind }}"
        project: "{{ project }}"
        service_account_file: "{{ service_account_file }}"
    loop: "{{ fw_
rules | selectattr('type','equalto','allow') | list }}" (3)
    loop_control:
      loop_var: rule
      label: "{{ rule.name }}"
```

The numbered references from the output are explained here:

- (1) The network is defined with the `selfLink` contained in the `gcp_vpc` variable.

- (2) The `default` Ansible filter allows you to remove this parameter (and use the GCP default) when it is not defined.
- (3) The `selectattr` Ansible filter allows the selection of the item in the loop only when the type is `allow`.
5. Finally, run the playbook to create the firewall rules according to the definition:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_gcp_env_build.yml
... output omitted ...
TASK [gcp_net_build : Create Allow Firewall Rules] *****
changed: [demo_gcp_vpc] => (item=allow_sql_from_usc-web_to_usc-db)
changed: [demo_gcp_vpc] => (item=allow_internet_to-usc-web)
changed: [demo_gcp_vpc] => (item=allow_ssh_to_usc-bastion)
changed: [demo_gcp_vpc] => (item=allow_ssh_from_bastion_only)
```

6. In the GCP console, in the **VPC** section, under the **FIREWALLS** tab, you can observe the recently created ingress firewall rules (without egress rules, everything is allowed in one direction).

ansible-demo-vpc

OVERVIEW	SUBNETS	STATIC INTERNAL IP ADDRESSES	FIREWALLS	FIREWALL ENDPOINTS	ROUTES	VPC NETWORK PEERING	PRIVATE SERVICES ACCESS	DNS
ADD FIREWALL RULE		DELETE						
Filter Enter property name or value								
☐	Name	Enforcement order	Type ↑	Rule priority	Targets	Source	Protocols and ports	Action
	▼ vpc-firewall-rules	1	VPC firewall rules					
☐	allow-ssh-to-usc-bastion		Ingress firewall rule	10	Tags: usc-bastion	IPv4 ranges: 0.0.0.0/0	tcp:22	Allow
☐	allow-sql-from-usc-web-to-usc-db		Ingress firewall rule	10	Tags: usc-db	Tags: usc-web	tcp:3389	Allow
☐	allow-ssh-from-bastion-only		Ingress firewall rule	10	Tags: usc-web, usc-db	Tags: usc-bastion	tcp:22	Allow
☐	allow-internet-to-usc-web		Ingress firewall rule	10	Tags: usc-web	IPv4 ranges: 0.0.0.0/0	tcp:80, 443	Allow

Next, you will create the virtual servers in each subnet.

Deploying VMs in GCP

According to our example network design, we will deploy two web servers and two database servers across two different availability zones for high availability. Additionally, we will provision a **bastion host** in a single availability zone, as it is solely used for management purposes.

To streamline provisioning, you will define all the required machines in the `compute_nodes` variable. For each machine, some parameters (such as the name, network, zone, and access to the internet) will be used during the deployment process.

1. You start by updating the `group_vars/gcp_vpc.yml` file to include the information to describe the flavor and OS that will define all your VMs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
compute_node_flavor: f1-micro (1)
compute_node_images: projects/ubuntu-os-cloud/global/images/family/ubuntu-2204-lts (2)
compute_node_image_size: 10 # This number is in Gigabyte
```

The numbered references from the output are explained here:

- (1) Google Cloud (like the other cloud providers) offers different types of compute instances. More information can be found at:

<https://cloud.google.com/compute/docs/machine-resource>.

- (2) You can find all the available OS images at:

<https://cloud.google.com/compute/docs/images/os-details>.

2. Still in the same file (`group_vars/gcp_vpc.yml`), include the information to describe the four nodes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
... omitted output ...
compute_nodes:
  - name: web-server-1 (1)
    network: usc-web (2)
    has_internet: yes (3)
    zone: us-central1-a (4)
  ... another web-server omitted for brevity ...
  - name: db-server-1
    network: usc-db
    has_internet: no
    zone: us-central1-a
```

```
... another db-server omitted for brevity ...
- name: bastion-host
  network: usc-bastion
  ip: 192.0.3.253
  has_internet: yes
  ip_forwarding: yes
  zone: us-central1-a
```

The numbered references from the output are explained here:

- (1) Name of the VM.
 - (2) Specifies the subnet in which the VM is deployed and is also used for the network tag.
 - (3) Specifies whether the VM should get a public IP address.
 - (4) Specifies the zone in which the VM is deployed.
3. Then, create a new Ansible role (`gcp_vm_build.yml`) to deploy the VMs on GCP:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# cd roles && ansible-
galaxy init gcp_vm_build
- Role gcp_vm_build was created successfully
```

4. In this new role, update the existing `gcp_vm_build/tasks/main.yml` file with the following task to create the disks for the VMs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi roles/gcp_vm_
build/tasks/main.yml
- name: create a disk for {{ node.name }}
  google.cloud.gcp_compute_disk:
    name: "{{ node.name | regex_replace('_', '-') }}-disk"
    size_gb: "{{ compute_node_image_size }}" (1)
    source_image: "{{ compute_node_images }}" (2)
    zone: "{{ node.zone }}" (3)
    auth_kind: "{{ auth_kind }}"
    project: "{{ project }}"
    service_account_file: "{{ service_account_file }}"
    state: "{{ status | default('present') }}"
  register: gcp_vm_disk (4)
```


The numbered references from the output are explained here:

- (1) The size, in GB, of the disk to be deployed.
 - (2) The OS the machine will run; in the example, this is Ubuntu.
 - (3) The availability zone where the disk will be created.
 - (4) The `gcp_vm_disk` variable is registered to be used in the next tasks.
5. Still in the same file (`gcp_vm_build/tasks/main.yml`), you add the task to create a VM without a public IP address:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi roles/gcp_vm_
build/tasks/main.yml
... omitted output ...
- name: create a {{ node.name }} instance with no Internet
  google.cloud.gcp_compute_instance:
    name: "{{ node.name | regex_replace('_', '-') }}"
    machine_type: "{{ compute_node_flavor }}" (1)
    disks:
      - source: "{{ gcp_vm_disk }}" (2)
        boot: 'true'
    network_interfaces: (3)
      - network: "{{ gcp_vpc }}"
        subnetwork: "{{ gcp_subnets.results |
          selectattr('name', 'equalto', node.network) |
          list | first }}"
    metadata:
      tier: "{{ node.name.split('-')[0] }}"
    tags: (4)
      items: "{{ node.network }}"
    zone: "{{ node.zone }}" (5)
    auth_kind: "{{ auth_kind }}"
    project: "{{ project }}"
    service_account_file: "{{ service_account_file }}"
    state: "{{ status | default('present') }}"
  when: not node.has_internet (6)
```

The numbered references from the output are explained here:

- (1) Specifies the instance type from the cloud offerings (check the following note for more context).
 - (2) Reference to the disk created in the previous task.
 - (3) A dictionary that specifies which VPC and subnet the VM should be deployed in. It uses the already-existing `gcp_vpc` variable and the value of the `subnets`' output.
 - (4) Specifies the network tasks that will match the firewall rules to create a mapping of rules to compute nodes.
 - (5) Specifies the availability zone where the VMs are deployed.
 - (6) Conditional to select when to run the task, in this case, when the node has no access to the internet.
6. Again, continuing with the same file (`gcp_vm_build/tasks/main.yml`), and completing the role tasks, create a similar task but to be used when the node requires internet access:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi roles/gcp_vm_
build/tasks/main.yml
... omitted output ...
- name: create an {{ node.name }} instance with Internet
  gcp_compute_instance:
    name: "{{ node.name | regex_replace('_', '-') }}"
    machine_type: "{{ compute_node_flavor }}"
    can_ip_forward: "{{ node.ip_forwarding if node.ip_
forwarding is defined else 'no' }}"
    disks:
      - source: "{{ gcp_vm_disk }}"
        boot: 'true'
    network_interfaces:
      - network: "{{ gcp_vpc }}"
        network_ip: "{{ node.ip if node.ip is defined else omit }}"
        subnetwork: "{{ gcp_subnets.results |
                        selectattr('name', 'equalto', node.network) |
                        list | first }}"
    access_configs: (1)
      - name: External NAT
```

```

    type: ONE_TO_ONE_NAT
  metadata:
    tier: "{{ node.name.split('-')[0] }}"
    zone: "{{ node.zone }}"
  tags:
    items: "{{ node.network }}"
  auth_kind: "{{ auth_kind }}"
  project: "{{ project }}"
  service_account_file: "{{ service_account_file }}"
  state: present
  register: vm_data
  when: node.has_internet

```

The numbered reference from the output is explained here:

- (1) When the instance has access to the internet, it's necessary to define the external NAT access, in this case, ONE_TO_ONE_NAT.
7. Finally, in the main playbook (pb_gcp_env_build.yml), you include a new task to create all the required VMs using the role defined:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_build.
yml
... omitted output ...
- name: Build VM Instances
  ansible.builtin.include_role:
    name: gcp_vm_build
  loop: "{{ compute_nodes }}"
  loop_control:
    loop_var: node

```

8. When you run the playbook again with all the recently added VM deployment configuration to extend the previous environment, you get:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_gcp_env_build.yml
... output omitted ...
TASK [Build VM Instances] *****
included: gcp_vm_build for demo_gcp_vpc => (item={'name': 'web-serv-
er-1', 'network': 'usc-web', 'has_internet': True, 'zone': 'us-cen-
tral1-a'})

```

```
included: gcp_vm_build for demo_gcp_vpc => (item={'name': 'web-server-2', 'network': 'usc-web', 'has_internet': True, 'zone': 'us-central1-b'})
... omitted output ...
included: gcp_vm_build for demo_gcp_vpc => (item={'name': 'bastion-host', 'network': 'usc-bastion', 'ip': '192.0.3.253', 'has_internet': True, 'ip_forwarding': True, 'zone': 'us-central1-a'})
TASK [gcp_vm_build : create a disk for web-server-1] *****
changed: [demo_gcp_vpc]
TASK [gcp_vm_build : create a web-server-1 instance with no Internet] *****
skipping: [demo_gcp_vpc] (1)
TASK [gcp_vm_build : create an web-server-1 instance with Internet] *****
changed: [demo_gcp_vpc]
... output omitted ...
```

The numbered reference from the output is explained here:

(1) Depending on whether or not there is access to the internet, the task is either run or skipped.

- 9. After the execution, if you go to the GCP console and then **Compute Engine | VM Instances**, you will find all the VMs provisioned:

VM instances

 **Filter** Enter property name or value

☐	Status	Name ↑	Zone	Internal IP	External IP	Connect
☐	✓	bastion-host	us-central1-a	192.0.3.253 (nic0)	34.56.236.13 (nic0)	SSH ▾
☐	✓	db-server-1	us-central1-a	192.0.2.2 (nic0)		SSH ▾
☐	✓	db-server-2	us-central1-b	192.0.2.3 (nic0)		SSH ▾
☐	✓	web-server-1	us-central1-a	192.0.1.2 (nic0)	34.56.232.155 (nic0)	SSH ▾
☐	✓	web-server-2	us-central1-b	192.0.1.3 (nic0)	104.197.186.84 (nic0)	SSH ▾

- 10. Also, because the VMs were created with the correct network tags, you can verify that the firewall rules are applied only on those VMs that require them by looking into the **Network Security | Firewall Policies | allow-internet-to-usc-web** rule.

 Filter Filter by instance name, project or subnetwork

Name ↑	Subnetwork	Internal IP ranges	External IP ranges	Tags	Service accounts	Project
web-server-1	usc-web	192.0.1.2	34.56.232.155	usc-web	None	gcp-automation-demo
web-server-2	usc-web	192.0.1.3	104.197.186.84	usc-web	None	gcp-automation-demo

Note



In many cloud platforms, resource availability can vary between regions. In Google Cloud, this availability is dynamic and often depends on the type of resource. For instance, when working on this example, we initially targeted the Australia Southeast 1 region. However, at the time of writing, the cheapest VM types were unavailable there, so we switched to the US Central 1 region instead. This flexibility highlights one of the greatest advantages of dynamic infrastructure—adaptability and ease of change.

To complete the cloud networking setup, you need to adjust the routing to determine the proper network flows.

Adjusting routing within a GCP VPC

In the example setup, the current routing and firewall rules prevent our database servers from accessing the internet. However, to install software or apply patches, these servers need occasional internet connectivity. To achieve this, you will configure the bastion hosts as NAT instances to provide internet access to the database servers. This requires updating the routing configuration for the database servers within the VPC.

By default, GCP creates a route in the VPC pointing to the internet gateway, which is applied to all hosts in the network. However, since the database servers lack external public IP addresses and are restricted by firewall rules, they cannot access the internet through this default route. To resolve this, you will modify the routing for the database servers, directing their internet-bound traffic to the bastion hosts (acting as NAT instances). At the same time, you must retain the original default route, as it is used by the web and bastion hosts to access the internet.

1. Update the `group_vars/gcp_vpc.yml` file to include the routing data that defines the routing policy:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi group_vars/gcp_vpc.yml
```

```
... output omitted ...
custom_routes:
  - name: default-route-db
    description: Default Route for DB nodes
    dest: '0.0.0.0/0'
    nh: '192.0.3.253'
    apply_to: usc-db
```

2. Update the `pb_gcp_env_build.yml` playbook with the task to create the routes in GCP:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_build.
yml
... output omitted ...
  - name: Create the Route
    google.cloud.gcp_compute_route:
      name: "{{ route.name }}"
      description: "{{ route.description | default(omit) }}"
      dest_range: "{{ route.dest }}"
      network: "{{ gcp_vpc }}"
      next_hop_ip: "{{ route.nh }}" (1)
      tags: "{{ route.apply_to.split(',') | default(omit) }}"
      state: present
      auth_kind: "{{ auth_kind }}"
      project: "{{ project }}"
      service_account_file: "{{ service_account_file }}"
    loop: "{{ custom_routes }}"
    loop_control:
      loop_var: route
    when:
      - custom_routes is defined
      - route.nh is defined
```

The numbered reference from the output is explained here:

(1) The next hop could be either an IP address or an instance identification.

3. Finally, when you run the `pb_gcp_env_build.yml` playbook, the route is added:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_gcp_env_build.yml
... output omitted ...
```

```
TASK [Create the Route] *****
changed: [demo_gcp_vpc] => (item={'name': 'default-route-db', 'description': 'Default Route for DB nodes', 'dest': '0.0.0.0/0', 'nh': '192.0.3.253', 'apply_to': 'usc-db'})
```

- 4. In the GCP console, within the **VPC network** details, under the **ROUTES** tab, you can see the new route added, with the next hop. Notice that you have to select the region to see this.

ansible-demo-vpc

OVERVIEW

SUBNETS

STATIC INTERNAL IP ADDRESSES

FIREWALLS

FIREWALL ENDPOINTS

ROUTES

VPC NETWORK PEERING

Select the region for which you want to view routes. To manage your routes, go to [Route management](#).

Region *

us-central1 (Iowa)

VIEW

REFRESH

Filter

Enter property name or value

Status	Name ↑	Type	Destination IP range	Scope limits ?	Next hop
✓	default-route-3fe466c1a0ad5fa6	Static	0.0.0.0/0	—	Default Internet gateway
✓	default-route-db	Static	0.0.0.0/0	Instance tags: usc-db	IP address 192.0.3.253 (instance bastion-host)
✓	default-route-r-2f03607e02c59622	Subnet	192.0.1.0/24	—	Network ansible-demo-vpc
✓	default-route-r-32f2e802c177cd25	Subnet	192.0.3.0/24	—	Network ansible-demo-vpc
✓	default-route-r-73ac7ac754e7e33a	Subnet	192.0.2.0/24	—	Network ansible-demo-vpc



Note

You can notice that the route task is directly in the main playbook rather than in a role definition. This is because the next hop has to exist before the route is created.

At this point, you should have the chapter’s scenario up and running, so it’s time to validate that it works as expected.

Validating a GCP deployment

Ansible offers various modules to retrieve the operational state of resources in GCP. These modules enable us to programmatically validate the current state of our network resources, eliminating the need to manually log in to the GCP portal and use the **Graphical User Interface (GUI)** to check the status of deployed components. In this recipe, you will learn how to use specific Ansible modules to validate the network subnets you have provisioned in GCP:

1. Create a new `pb_gcp_net_validate.yml` playbook with a task to validate the VPC subnet information, using a similar skeleton as in the previous one, using the GCP secrets JSON file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_net_validate.yml
... output omitted ...
- name: Get Subnet Facts
  google.cloud.gcp_compute_subnetwork_info:
    region: "{{ region }}"
    auth_kind: "{{ auth_kind }}"
    project: "{{ project }}"
    service_account_file: "{{ service_account_file }}"
    loop: "{{ subnets }}"
    register: gcp_vpc_subnets (1)
... output omitted ...
```

The numbered reference from the output is explained here:

- (1) The result of the task is stored to be validated/asserted in the next task.
2. Complete the playbook with a task to validate that the IP prefix provisioned matches the one from the subnet definition:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_net_validate.yml
... output omitted ...
- name: Validate all Subnets are Deployed
  ansible.builtin.assert:
    that:
      - gcp_vpc_subnets.results |
        selectattr('item.cidr','==',item.cidr) | (1)
        map(attribute='item') | (2)
        list (3)
    loop: "{{ subnets }}"
```


The numbered references from the output are explained here:

(1) `selectattr('item.cidr', '==', item.cidr):`

Filters `gcp_vpc_subnets.results` to find objects where `item.cidr` matches the current `item.cidr` from the loop. It's a pure coincidence that both are named the same.

(2) `map(attribute='item'):` Extracts the `item` field from the filtered results.

(3) `list:` Converts the result into a list for further processing.

3. When you run the playbook, it will iterate over the subnets to check and look for the mapping in the subnet definition:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_gcp_net_validate.yml
... omitted output ...
TASK [Get Subnet Facts] *****
ok: [demo_gcp_vpc] => (item={'name': 'usc-
web', 'cidr': '192.0.1.0/24'})
ok: [demo_gcp_vpc] => (item={'name': 'usc-
db', 'cidr': '192.0.2.0/24'})
ok: [demo_gcp_vpc] => (item={'name': 'usc-
bastion', 'cidr': '192.0.3.0/24'})
TASK [Validate all Subnets are Deployed] *****
ok: [demo_gcp_vpc] => (item={'name': 'usc-
web', 'cidr': '192.0.1.0/24'}) => {
    "ansible_loop_var": "item",
    "changed": false,
    "item": {
        "cidr": "192.0.1.0/24",
        "name": "usc-web"
    },
    "msg": "All assertions passed"
}
... omitted output ...
```

Next, let's revert all the changes in the dynamic infrastructure to leave it clean (and save us costs!).

Decommissioning GCP resources

All the tasks to decommission GCP will be contained in one single playbook, where the most relevant difference versus the build playbook is the usage of state: `absent` (instead of `present`). Because of the resource dependency, it's important to define the tasks in the proper order, inside out.

1. Create a new `pb_gcp_env_destroy.yml` playbook with the same structure as the other playbooks, with the read of the vault encrypted JSON file and the final handler to clean it up. But now, let's use a new setup using well-known environmental variables for the GCP library that saves some credential specification in every task:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
environment:
  GCP_SERVICE_ACCOUNT_FILE: "{{ service_account_file }}"
  GCP_AUTH_KIND: 'serviceaccount'
vars_files:
  - gcp_account_info.yml
```

2. Then, let's add the first task that will collect VPC information and save it for the next tasks that need it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Get VPC Info
  google.cloud.gcp_compute_network_info:
    project: "{{ project }}"
  register: gcp_vpc
```

3. Next, you start adding tasks to clean up resources, starting in the reverse order from creation, with the VMs:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Delete Instances
  google.cloud.gcp_compute_instance:
    name: "{{ node.name | regex_replace('_', '-') }}"
```

```

    zone: "{{ node.zone }}"
    project: "{{ project }}"
    state: absent
  loop: "{{ compute_nodes }}"
  loop_control:
    loop_var: node
    label: "{{ node.name }}"

```

4. Continuing in the destroy playbook, delete the disks created for the VMs:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Delete disks for Instances
  google.cloud.gcp_compute_disk:
    name: "{{ node.name | regex_replace('_', '-') }}-disk"
    zone: "{{ node.zone }}"
    project: "{{ project }}"
    state: absent
  loop: "{{ compute_nodes }}"
  loop_control:
    loop_var: node
    label: "{{ node.name }}"

```

5. Update the playbook with the deletion of all the firewall rules:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Delete All Firewall Rules
  google.cloud.gcp_compute_firewall:
    name: "{{ rule.name | regex_replace('_', '-') }}"
    network: "{{ gcp_vpc }}"
    project: "{{ project }}"
    state: absent
  loop: "{{ fw_rules }}"
  loop_control:
    loop_var: rule

```

6. Next, continue deleting the routes created manually:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Delete all Routes
  google.cloud.gcp_compute_route:
    name: "{{ route.name }}"
    dest_range: "{{ route.dest }}"
    network: "{{ gcp_vpc }}"
    project: "{{ project }}"
    state: absent
  loop: "{{ custom_routes }}"
  loop_control:
    loop_var: route
  when:
    - custom_routes is defined
```

7. Finally, delete the GCP subnets and the VPC:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch10# vi pb_gcp_env_
destroy.yml
... omitted output ...
- name: Delete GCP Subnets
  google.cloud.gcp_compute_subnetwork:
    name: "{{ subnet.name }}"
    ip_cidr_range: "{{ subnet.cidr }}"
    network: "{{ gcp_vpc }}"
    region: "{{ region }}"
    project: "{{ project }}"
    state: absent
  loop: "{{ subnets }}"
  loop_control:
    loop_var: subnet
- name: Delete GCP VPC
  google.cloud.gcp_compute_network:
    name: "{{ vpc_name | regex_replace('_', '-') }}"
    project: "{{ project }}"
    state: absent
```

8. When you run the playbook, after a few minutes, the GCP project is clean as it was at the beginning:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch10# ansible-playbook
pb_gcp_env_destroy.yml
... omitted output ...
TASK [Get VPC Facts] *****
ok: [demo_gcp_vpc]
TASK [Delete Instances] *****
changed: [demo_gcp_vpc] => (item=web-server-1)
... omitted output ...
TASK [Delete disks for Instances] *****
changed: [demo_gcp_vpc] => (item=web-server-1)
... omitted output ...
TASK [Delete All Firewall Rules] *****
changed: [demo_gcp_vpc] => (item={'name': 'allow_sql_from_usc-
web_to_usc-db', 'type': 'allow', 'direction': 'ingress', 'pri-
ority': 10, 'apply_to': 'usc-db', 'src_tag': 'usc-web', 'dest_
tag': None, 'protocol': 'tcp', 'port': 3389, 'state': 'present'})
... omitted output ...
TASK [Delete all Routes] *****
changed: [demo_gcp_vpc] => (item={'name': 'de-
fault-route-db', 'description': 'De-
fault Route for DB nodes', 'dest': '0.0.0.0/0', 'nh': '192.0.3.253',
'apply_to': 'usc-db'})
TASK [Delete GCP Subnets] *****
changed: [demo_gcp_vpc] => (item={'name': 'usc-web', 'cidr':
'192.0.1.0/24'})
... omitted output ...
TASK [Delete GCP VPC] *****
changed: [demo_gcp_vpc]

```

Clean up done! Remember, when working with dynamic infrastructure, destroy it when you no longer use it to avoid paying per use.

Summary

In this chapter, you have gone through all the basic steps required to create, validate, and destroy a network setup in the GCP cloud, including the provisioning of VMs. This chapter also introduced the role usage to containerize Ansible tasks (and other resources) for easier reusability.

This is the last chapter in *Part 3*, where you have learned how to automate the most popular cloud providers: AWS, Azure, and GCP. In *Part 4*, you will learn about advanced network scenarios with Ansible.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



Part 4

Integrating Ansible

In this final part of the book, we go beyond device configuration and service deployment to focus on building robust, scalable, and intelligent automation ecosystems. These chapters introduce tools and techniques that extend the power of Ansible and help you move from task automation to full lifecycle orchestration.

You'll begin with network validation using Batfish—a powerful way to catch misconfigurations before they go live. Then, you'll learn how to use Nautobot as a dynamic source of truth to drive the Ansible inventory and enrich your automation with contextual awareness.

Next, we introduce AWX (the upstream project for Red Hat Ansible Automation Platform), which brings enterprise features such as a graphical interface, RBAC, credential and inventory management, and centralized logging to your automation workflows.

From there, you'll dive into event-driven automation, where Ansible reacts to real-time triggers and builds dynamic workflows based on defined rules. Finally, you'll explore how to integrate Ansible with Terraform to combine configuration management with infrastructure provisioning, unlocking end-to-end automation across diverse environments.

Together, these chapters provide a roadmap for evolving your automation practice from simple playbooks to a modern, integrated, and intelligent automation platform.

This part of the book includes the following chapters:

- *Chapter 11, Network Validation with Batfish*
- *Chapter 12, Building a Network Inventory with Nautobot*
- *Chapter 13, Simplifying Automation with AWX*
- *Chapter 14, Event-Driven Automation with Ansible*
- *Chapter 15, Terraform and Ansible Dancing Together*

11

Network Validation with Batfish

In previous chapters of this book, you worked through multiple recipes that outline how to perform network validation using Ansible—you did this by utilizing the different modules available within Ansible. In all of those cases, you performed network validation after pushing the configuration onto the network devices. You then collected the network state and validated that it was aligned with our intended state. However, you may want to validate the network state prior to pushing the configuration onto the devices. As well as this, it might be necessary to validate that the intended network state is as desired without even touching our network. But how do you do this?

Batfish (<https://batfish.org/>) is an open source project that targets this use case. Its main aim is to provide an offline network validation tool to validate multiple aspects of the network configuration. Batfish can provide a validation and correctness guarantee for *security*, *compliance*, and *traffic forwarding* for the network. It uses the device configuration from our network devices to build a neutral analytical data model and forwarding tree for our network, which you can then use to validate the network state and validate the correct traffic forwarding within our network.



Note

Batfish was developed by researchers from UCLA and USC around 2015, later evolving into Intentionet, which Amazon acquired in 2021 to enhance its network verification capabilities.

The following diagram outlines the high-level architecture of Batfish and how it works:

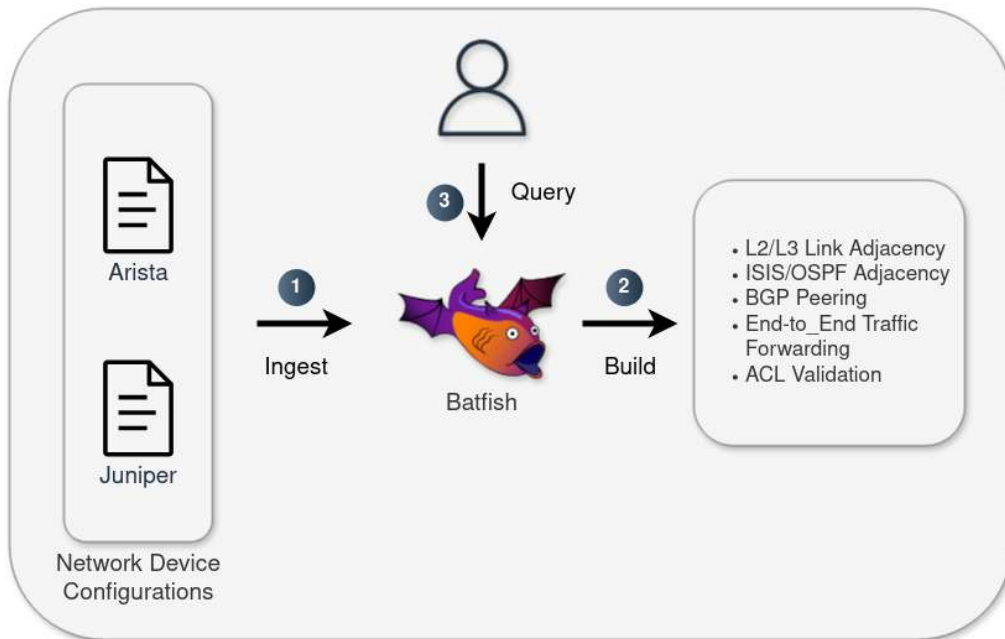


Figure 11.1 – Batfish workflow

Batfish uses a client/server model. With this, you run a Batfish server instance (written in Java), and you communicate with the server API using a client-side **software development kit (SDK)** called **pybatfish** (written in Python). You then initialize a snapshot of our network using our network configuration files, and, based on this network snapshot, the Batfish server computes a data model for our network. Using the client, we can start to ask questions to validate our network using this vendor-neutral data model.



Info

For more information on Batfish, visit <https://github.com/batfish/batfish> and <https://pybatfish.readthedocs.io/en/latest/>.

The Batfish team has developed multiple Ansible modules that wrap around the pybatfish client library to retrieve the data model generated by the Batfish server. The modules allow us to perform different queries against this network model to validate our network state. The diagram that appears next outlines the interaction between Ansible, pybatfish, and the Batfish server.

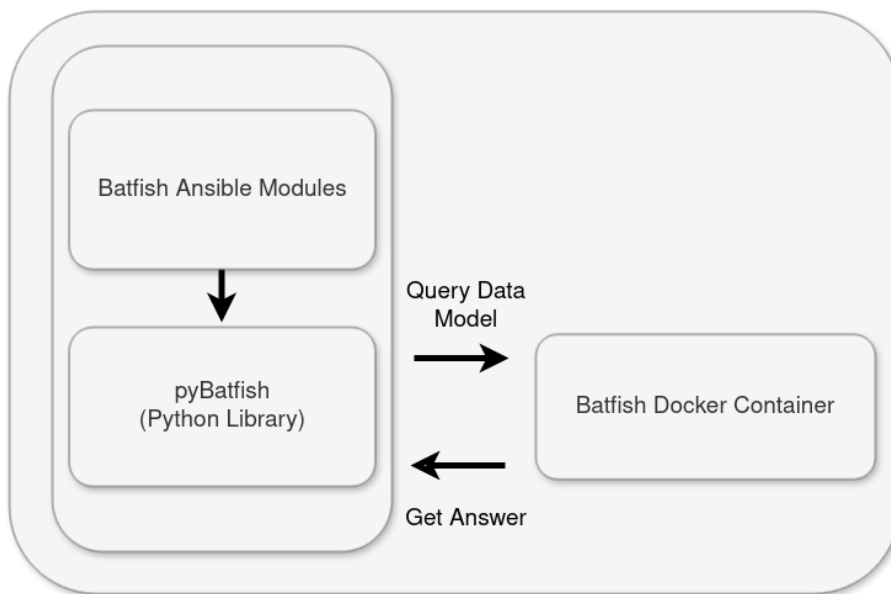


Figure 11.2 – Batfish SDK interaction

In this chapter, you will learn how to install Batfish and integrate it with Ansible in order to start using it to validate the network state, prior to pushing the configuration to our devices. *Batfish running as a Docker container* is the easiest and most supported method of running the application.

This combination is very powerful and can easily be extended to build complete **continuous integration/continuous deployment (CI/CD)** pipelines for network configuration changes. Batfish can be an integral part of providing pre-validation before pushing the configuration to network devices in production.



Warning

Batfish transforms configuration artifacts into an analytical model, requiring dedicated parsing and translation for each network operating system. As a result, achieving 100% coverage of all network functionalities is unlikely.

You are going to reuse the network configurations created in *Chapter 4* to outline how you can validate this sample leaf-spine network topology using Ansible and Batfish.

In order to discover how to use offline configuration validation using Batfish, this chapter will focus on the following recipes:

- Installing Batfish
- Integrating Batfish with Ansible
- Pulling in the network config
- Initializing the network snapshot with Ansible
- Collecting network facts from Batfish
- Validating the network with Batfish assertions



Note

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch11.

Even though it's not the main goal of this chapter, you will take care of installing Batfish in your development environment, using the publicly available Docker container. Since this is a complex lab and setup, it is highly recommended to follow the installation and setup step by step, paying close attention to the notes throughout.

Installing Batfish

Batfish provides multiple options for installing and running the Batfish server. You will use the simplest and most recommended method by running a Docker container that houses the Batfish server.

In this recipe, you will install the server components for Batfish, which come packaged in the publicly available Docker container (to facilitate deploying it in a container-based environment, like in this lab). This is a mandatory foundation step in order to start validating our network using Batfish.

Let's get started by pulling down the Docker container image and running it:

1. Validate that **Docker** is running. (Docker was included in the lab VM prerequisites; therefore, installing Docker is not necessary.)

```
root@cookbook-lab:~# sudo systemctl status docker.service
docker.service - Docker Application Container Engine
Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor
```

```

preset: enabled)
  Active: active (running) since Sun 2025-03-23 14:42:01 UTC; 1 day
  4h ago (1)

```

The numbered reference from the preceding code is explained as follows:

- (1) Make sure the status is active (running). If this is not the case, restart it by executing `sudo systemctl restart docker.service`.
2. Download the Batfish container. By default, Docker will pull from Docker Hub, which is where the Batfish container is hosted.

```

root@cookbook-lab:~# docker pull batfish/batfish (1)
Using default tag: latest
latest: Pulling from batfish/batfish
... omitted output ...
b09cb628243f: Pull complete
Digest: sha256:8cbad6acbb16c70ea4dc3951196b2d1fb75d78836d1e4f21aec
8069d9c982868

```

The numbered reference from the preceding code is explained as follows:

- (1) Batfish provides two Docker containers: `batfish/batfish` and `batfish/allinone`. The `batfish/allinone` container has the Batfish server and the `pybatfish` client library. It also has the Jupyter Notebook Python library installed, along with some sample notebooks, to start interacting with the Batfish server.
3. Start the Batfish container. This can be done by using the following `docker run` command and mapping it on the Linux machine, using the `-p` directive.

```

root@cookbook-lab:~# docker run -d -p 9996:9996 -h batfish --name
batfish batfish/batfish
fa7474d168d7534dae1c8f258ee6d6bea368b51ec04dde4d0928f7cf32f7d048

```

4. Now you can validate that the container is up and running.

```

root@cookbook-lab:~# docker ps
CONTAINER ID   IMAGE             PORTS              COMMAND
CREATED        STATUS           NAMES
fa7474d168d7   batfish/batfish   0.0.0.0:9996->9996/tcp, [::]:9996->9996/tcp   "java
-XX:-UseCompre..." 25 seconds ago    Up 25 seconds
batfish

```

The numbered reference from the preceding code is explained as follows:

- (1) The Docker container is running. The Batfish service exposes a REST API service available on port 9996. This port is exposed to the host system as well so it's accessible to interact with.



Note

Batfish's documentation shows two ports (9996 and 9997). 9997 was the first version of the API, and 9996 was the second version of the API. Since you're using the most up-to-date version of pybatfish, 9996 is the only port that will be used.

Now that Batfish is running in a Docker container and the proper ports are exposed, let's explore how to integrate with Batfish.

Integrating Batfish with Ansible

To integrate Batfish with Ansible, you need to install the required Python packages. Doing so will allow Ansible to communicate with the Batfish server. You will also need to install the Ansible role that exposes the Batfish modules.

In this recipe, you will install these Python packages, followed by installing the required Ansible roles needed to run the required Batfish Ansible modules. This is accomplished through two steps:

1. In the Ansible controller, you need to install the `pybatfish` Python library, which is the Batfish client-side SDK that interacts with the Batfish server. This package is required by the Ansible modules. These will be used to interact with the Batfish server in our playbooks.

```
root@cookbook-lab:~# pip install pybatfish==2023.12.16.1270 (1)
Collecting pybatfish
... omitted output ...
Installing collected packages: pytz, tzdata, simplejson, python-
dateutil, orderly-set, numpy, attrs, requests-toolbelt, pandas,
deepdiff, pybatfish
```

The numbered reference from the preceding code is explained as follows:

- (1) This uses a specific pinning of `pybatfish` version because of the Batfish Ansible role requirements.

2. Secondly, you install the batfish roles, which were developed in order to interact with the Batfish server and validate network device configuration. This Ansible role contains all the required Python scripts to run the custom Ansible modules for Batfish. In order to install this role on the Ansible control machine, you will be using `ansible-galaxy`, which has been covered multiple times throughout the book.

```
root@cookbook-lab:~# ansible-galaxy role install batfish.base (1)
Starting galaxy role install process
- downloading role 'base', owned by batfish
- downloading role from https://github.com/batfish/ansible/archive/
master.tar.gz
- extracting batfish.base to /root/.ansible/roles/batfish.base
- batfish.base (master) was installed successfully
```

The numbered reference from the preceding code is explained as follows:

- (1) A **role** is a self-contained unit of automation that groups related tasks, variables, handlers, and other assets into a reusable package. You've mainly been working with collections in this book, which are packages that can contain multiple roles, modules, plugins, and playbooks, acting as a distribution format for Ansible content.



Warning

Although the Ansible role for Batfish is marked as archived in GitHub, the role continues to be available in Ansible Galaxy. The Batfish developer team marked the repository as archived because they saw the project as feature complete, where no new features and/or bug fixes would be implemented.

Let's validate that the Python library and Ansible role were installed successfully.

1. Let's validate that `pybatfish` is installed in our Ansible control host.

```
root@cookbook-lab:~# pip freeze | grep pybatfish
pybatfish==2024.7.22.1569
```

2. Validate the Batfish Ansible roles were installed properly by listing the roles.

```
root@cookbook-lab:~# ansible-galaxy role list
# /root/.ansible/roles
- batfish.base, master
```


3. Finally, to expose the Python source code for the role, you can look into the `library` folder to get an understanding of what the role comes packaged with:

```
root@cookbook-lab:~# tree /root/.ansible/roles/batfish.base/library/  
/root/.ansible/roles/batfish.base/library/  
├─ bf_assert.py  
├─ bf_extract_facts.py  
├─ bf_init_snapshot.py  
├─ bf_session.py  
├─ bf_set_snapshot.py  
├─ bf_upload_diagnostics.py  
└─ bf_validate_facts.py
```

A few important files to notice are the `bf_session`, `bf_init_snapshot`, and `bf_assert` files. These files expose helpers to make interacting with Batfish from the Ansible playbooks much more straightforward.

With all the setup and prerequisites installed and configured, let's start using Batfish to analyze some of our previously generated configurations.

Pulling in the network config

To start the analysis and validation with Batfish, you need to load the network device configurations to the Batfish server. Batfish is an offline network validation tool. It will ingest network configurations and generate a snapshot of the network that can be used for validation steps.

In this recipe, you will learn how to point Batfish to configuration files using Ansible. Throughout *Part 2* of this book, configuration templates were generated. You even created a config generation Ansible role to help us reuse code. To avoid covering the identical content in this chapter, you will pull in the configurations that were generated in *Chapter 4* and use those directly (if you've not done so, please go through that chapter first).

Let's start by creating the folder structure and a playbook to copy our Arista configurations over for analysis.

1. Create a new folder named `ch11`.

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch11 && cd  
chapters/ch11
```

2. Create a new playbook named `pb_prep_batfish_files.yml`. This playbook will be used to copy the configurations from the `ch04/output/config` directory into the `ch11/configs` directory.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# vi pb_prep_
batfish_files.yml
---
- name: "PREP CONFIGS FOR BATFISH"
  gather_facts: false
  hosts: "localhost"
  tasks:
    - name: "CREATE REQUIRED CONFIGS DIRECTORY"
      ansible.builtin.file:
        path: "configs" (1)
        state: "directory" (2)
    - name: "COPY CH04 GENERATED CONFIGS INTO CH11 CONFIGS
      DIRECTORY"
      ansible.builtin.copy:
        src: "/root/cookbook-lab/chapters/ch04/output/config/" (3)
        dest: "/root/cookbook-lab/chapters/ch11/configs/" (4)

```

The numbered references from the preceding code are explained as follows:

- (1) Create a directory named `configs`.
 - (2) The `ansible.builtin.file` module will create a directory and the directory will be named based on the `path` argument.
 - (3) The source directory to grab all the files from.
 - (4) The destination directory where all the source files should be copied to.
3. Run the playbook to copy over the configuration files from *Chapter 4*, and make sure that the files are in the proper structure.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook
pb_prep_batfish_files.yml
PLAY [PREP CONFIGS FOR BATFISH] *****
**

TASK [CREATE REQUIRED CONFIGS DIRECTORY]
*****
changed: [localhost]

```

```
TASK [COPY CH04 GENERATED CONFIGS INTO CH11 CONFIGS DIRECTORY]
*****
changed: [localhost]
... omitted output ...
```

4. Finally, validate that our Arista configurations have been copied over successfully.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# tree configs/
configs/
├── ceos-01.cfg
├── ceos-02.cfg
└── ceos-03.cfg
```

Note



The configs directory was created based on the Batfish documentation, in which it is a requirement. For the full details, visit the pybatfish documentation: <https://pybatfish.readthedocs.io/en/latest/formats.html#device-configuration-formats>

With the Arista configurations for our data center topology copied over into our new folder (ch11/configs), you can now move on to create your first network snapshot with Batfish.

Initializing the network snapshot with Ansible

To allow Batfish to analyze the network using the devices' configuration files, these files need to be structured in a specific folder, as mentioned in the previous section. In this recipe, you will pass the configuration files to Batfish and initialize your first Batfish snapshot.

Let's get started by creating the playbook to initialize the snapshot.

1. Create a new playbook named `pb_generate_snapshot.yml` and add the following tasks to connect to Batfish and initialize the network snapshot.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# vi pb_generate_
snapshot.yml
---
- name: "CREATE SESSION AND INITIALIZE SNAPSHOT"
  gather_facts: false
  hosts: "localhost"
```

```
roles:
- batfish.base (1)
tasks:
- name: "SETUP CONNECTION TO BATFISH SERVICE"
  bf_session: (2)
    host: "localhost" (3)
    name: "local_batfish"
    run_once: "yes"
    register: "bf_session"
- name: "INITIALIZE THE NETWORK SNAPSHOT"
  bf_init_snapshot: (4)
    network: "arista_dc_network"
    snapshot: "arista_dc_snapshot"
    snapshot_data: "."
    overwrite: true
    run_once: "yes"
    register: bf_snapshot
- name: "PRINT OUT DEBUG OF THE SNAPSHOT"
  ansible.builtin.debug:
    var: bf_snapshot
```

The numbered references from the preceding code are explained as follows:

- (1) Load the Batfish role that you installed before for use within the playbook.
- (2) Call the Ansible module from the Batfish role to establish a connection from the Ansible host to the Batfish container API. Notice also that you are not using the Fully Qualified Collection Name formatting. This is because we're using it as a role and you imported the Batfish rule in (1).
- (3) The **hostname** of the Batfish server. In this example, you are using `localhost`. This is because the Ansible control host and the Batfish server (container) are running on the same VM. Notice that you did not have to set the port to use; this is because `pybatfish` natively assumes you will be connecting to port 9996 by default.

Create a Batfish snapshot with the configurations stored in the location directory via the use of (.). Batfish will automatically take the snapshot_data directory and append (configs/) to it in order to find the configurations to load into the snapshot. The snapshot itself will be initialized inside a locally significant network (a way to bundle certain configurations together with a named reference). You will name this one arista_dc_network, and the snapshot itself will be named arista_dc_snapshot, which is how you can load/use the snapshot in the future. Finally, the snapshot will be overwritten if a snapshot with an identical name already exists.

(4) Debug the snapshot variable to visualize any messages and/or warnings during the creation of the snapshot.

2. Run the playbook and make sure that the Batfish session to the server from the client is established and that the snapshot was created and printed out to validate that the snapshot initialized successfully.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook
pb_generate_snapshot.yml
PLAY [CREATE SESSION AND INITIALIZE SNAPSHOT]*****
TASK [SETUP CONNECTION TO BATFISH SERVICE]*****
changed: [localhost]
TASK [INITIALIZE THE NETWORK SNAPSHOT]*****
changed: [localhost]
TASK [PRINT OUT DEBUG OF THE SNAPSHOT]*****
ok: [localhost] => {
  ... omitted output ...
  "result": {
    "network": "arista_dc_network",
    "snapshot": "arista_dc_snapshot"
  },
  "summary": "Snapshot 'arista_dc_snapshot' created in network
              'arista_dc_network'",
  "warnings": [
    "Your snapshot was successfully initialized but Batfish
      failed to fully recognized some lines in one or more input
      files.
  ... omitted output ...
```



Important

The **warnings** from the snapshot creation can be safely ignored since they will not impact our analysis, but they are related to the limited coverage of Batfish when parsing configuration. Not everything is included in the analytical model.

With our playbook successfully establishing a connection to the Batfish container and passing in our configurations to generate a Batfish snapshot, you're in a great spot to start seeing the power that Batfish has to offer by collecting some of the network facts from Batfish for our Arista devices.

Collecting network facts from Batfish

Batfish can generate a vendor-neutral data model that represents the critical facts discovered from the configuration files supplied to Batfish. In this recipe, you will learn how to collect the facts discovered by Batfish and output them into a directory for analysis.

The network configuration has already been packaged into a snapshot within the Batfish server; this was done in the last playbook execution. Here, you will build a new playbook that utilizes the Batfish snapshot (`arista_dc_snapshot`) alongside the Batfish network (`arista_dc_network`) you created in the last section.

1. Create a new playbook named `pb_get_facts.yml` with the following task to collect facts generated by Batfish.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# vi pb_get_facts.yml
---
- name: "ANALYZE BATFISH SNAPSHOT FOR ARISTA DC CONFIGS"
  gather_facts: false
  hosts: "localhost"
  roles:
- batfish.base
  tasks:
- name: "SETUP CONNECTION TO BATFISH SERVICE"
  bf_session:
    host: "localhost"
    name: "local_batfish"
  run_once: "yes"
  register: "bf_session" (1)
```

```
- name: "RETRIEVE BATFISH FACTS"
  bf_extract_facts:
    output_directory: "analysis_folder/bf_facts" (2)
    network: "arista_dc_network" (3)
    snapshot: "arista_dc_snapshot" (4)
  run_once: yes
  register: bf_facts
```

The numbered references from the preceding code are explained as follows:

- (1) Establish the Batfish API session. This is the same task you used in the previous playbook.
 - (2) The directory in which Batfish should save the results of the fact-gathering operation.
 - (3) Set the Batfish network. In this case, you are setting it to the network that you created in the previous playbook, where the snapshot was created and related to this network name.
 - (4) The name of the snapshot that was also previously initialized in the previous playbook.
2. Next, execute the playbook, which will load the previously created network and snapshot, and query Batfish for the results of the facts engine. Finally, the facts will be stored in the directory set in the `output_directory` argument.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook
pb_get_facts.yml
PLAY [ANALYZE BATFISH SNAPSHOT FOR ARISTA DC CONFIGS] *****
TASK [SETUP CONNECTION TO BATFISH SERVICE] *****
changed: [localhost]
TASK [RETRIEVE BATFISH FACTS] *****
changed: [localhost]
... omitted output ...
```

3. Let's take a look at the directory that was created by executing the playbook. You will see a YAML file per host was created.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# tree analysis_
folder/
analysis_folder/
├── bf_facts
├── ceos-01.yml
├── ceos-02.yml
└── ceos-03.yml
```

4. Finally, let's take a look at some of the data that the `bf_extract_facts` module produces.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# less analysis_
folder/bf_facts/ceos-01.yml
nodes:
  ceos-01:
AS_Path_Access_Lists: []
BGP:
  Multipath_EBGP: true
  Multipath_IBGP: false
  Multipath_Match_Mode: EXACT_PATH
  Neighbors:
    10.100.1.1:
      ... output omitted ...
      Local_AS: 65100
      Remote_AS: '65001'
      Remote_IP: 10.100.1.1
      Route_Reflector_Client: false
      Send_Community: false
      VRF: default
      ... output omitted ...

```

Now that the extraction and saving of the facts have been completed, you may want to validate that the operational state matches the intended state. Since you have learned about the Ansible built-in `assert` module multiple times in this book, the concepts will not be directly covered here again.

A simple example of this concept is shown here:

```

- name: "Read yaml file"
  ansible.builtin.shell: "cat analysis_folder/bf_facts/ceos-01.yml"
  register: result
- name: "Parse yaml into variable"
  set_fact:
my_loaded_bf_file: "{{ result.stdout | from_yaml }}"
- name: Assert Batfish Data
  ansible.builtin.assert:
that:
  - '10.100.1.1' in {{ my_loaded_bf_file.BGP.Neighbors }}
quiet: true

```


Finally, in the last section of this chapter, you will learn about some of the Batfish-specific custom assertions that come packaged with the Ansible Batfish role.

Validating the network with Batfish assertions

Let's look into the unique set of custom assertions that Batfish exposes. These assertions allow you to load configurations, generate a network model, and assert that your network and configuration are following best practices. It can assert that traffic, routing protocols, and even configuration elements are running as expected.



Note

The available assertions that Batfish developed can be found in the *Assertion* documentation in the repository: <https://github.com/batfish/ansible/blob/master/docs/assertions.rst>

In this section, you will explore two Batfish assertions that have self-explanatory names:

- `assert_no_undefined_references`: Finds configurations that have references to named structures (e.g., ACLs) that are not defined
- `assert_filter_has_no_unreachable_lines`: Finds all lines in the specified filters that will not match any packet, either because of being shadowed by prior lines or because of the match condition being empty

Let's get started by making sure our configurations don't include references that don't exist. Many vendor configurations allow configurations to exist (e.g., Access List) without ever being utilized, but they also allow configurations to point to these references when they may or may not be defined. The Batfish Undefined References assertion checks for these exact scenarios. For example, what if you had an inbound ACL set on an interface, and that ACL actually doesn't exist in the configuration at all? That would be an important configuration validation to confirm.



Warning

An extra step that you will need to execute because our VM and dependencies are packaged with the VM using Python 3.12 is that you will need to monkey patch a source file from the Batfish Ansible role. This is due to a deprecated import that hasn't been fixed since the Batfish Ansible repository is now archived, as explained earlier. Modify the `bf_assertion_util.py` file from the role and change the import for Mappings

. This can be done using the following one-liner:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# sed -i 's/from collections
import Mapping/from collections.abc import Mapping/g' /root/.ansible/
roles/batfish.base/module_utils/bf_assertion_util.py
```

The preceding one-liner command uses the Linux program called **sed (Stream Editor)** to inline modify a source code file. The `s/` syntax is a string replacement utility that allows for quick editing when a simple replacement is needed.

In order to simplify this section, you will create very small configuration snippets with only the important elements to get a fundamental understanding of what is being checked.

Let's ensure that no such *misconfigurations* exist in our Arista topology.

1. Create a new directory called `undefined_references`, where you will create the Batfish required directory named `configs`. This will allow us to store configurations for this specific example.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# mkdir -p undefined_
references/configs && cd undefined_references/configs
```

2. Create the following simplistic configuration. This configuration snippet is used as a way to provide a clear and concise example where we set `access-list 10` on the interface without defining the access list itself.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11/undefined_references/
configs# vi router.cfg
hostname router
!
interface Ethernet1
  no switchport
```

```

    ip address 172.31.1.4/31
    ip access-group 10 in
    !
end

```

3. Create a new playbook named `pb_assert_undefined_references.yml`. Since you will be using an ad hoc configuration snippet, you will create the connection, initialize a new snapshot, and run the assertion test. The play definition and the Batfish session creation task will be omitted throughout these examples.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# vi pb_assert_undefined_references.yml
---
... omitted output ...
- name: "INITIALIZE THE NETWORK SNAPSHOT"
  bf_init_snapshot:
    network: "ch11_network"
    snapshot: "ch11_snapshot"
    snapshot_data: "./undefined_references"
    overwrite: true
    run_once: "yes"
    register: bf_snapshot
- name: "VALIDATE NO UNDEFINED REFERENCES EXIST"
  bf_assert:
    assertions:
      - type: assert_no_undefined_references
        name: Confirm we have no undefined references

```

4. Let's run the playbook and see what will happen. It should have been able to pick out from the configuration snippet that you're assigning an ACL named **10** to an interface, without ever defining that ACL within the configuration.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook pb_assert_undefined_references.yml
... omitted output ...
TASK [INITIALIZE THE NETWORK SNAPSHOT]
*****

changed: [localhost]
TASK [VALIDATE NO UNDEFINED REFERENCES EXIST] *****

```

```
fatal: [localhost]: FAILED! => {"changed": false, "msg": "1 of
1 assertions failed", "result": [{"details": "Found undefined
reference(s), when none were expected\n[{'File_Name': 'configs/
router.cfg', 'Struct_Type': 'ipv4 acl', 'Ref_Name': '10',
'Context': 'interface incoming ip access-list', 'Lines':
FileLines(filename='configs/router.cfg', lines=[6])}]", "name":
"Confirm we have no undefined references", "status": "Fail", "type":
"assert_no_undefined_references"}], "summary": "1 of 1 assertions
failed"}
```

Take a look at the **FAILED** result that was generated. The assertion failed because of an undefined reference of type `ipv4 acl`. It specifies all the important metadata such as configuration line number and ACL reference number.

5. Let's fix it and append the definition for access list 10 to our previous configuration.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11/undefined_references/
configs# vi router.cfg
... omitted output ...
access-list 10 permit any
end
```

6. Rerunning the playbook will result in a **passing** assertion.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook
pb_assert_undefined_references.yml
... omitted output ...
TASK [VALIDATE NO UNDEFINED REFERENCES EXIST] *****
ok: [localhost]
```

Secondly, let's take a look at access lists at a deeper level. If you've managed networks at all during your career, you know they can get unruly quickly. How do you confirm you don't have statements in your ACL that are impossible to hit because of shadowing rules? For this, you can use the Batfish **Unreachable Lines** assertion.

7. Create one additional playbook named `pb_assert_unreachable_lines.yml`. The playbook is identical to the previous example except for the last task, which will be the only task shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11# vi pb_assert_
unreachable_lines.yml
... omitted output ...
```

```
- name: "VALIDATE NO UNREACHABLE LINES IN ACLS EXIST"
  bf_assert:
    assertions:
      - type: assert_filter_has_no_unreachable_lines (1)
        name: "ACL VALIDATION"
        parameters:
          filters: router["cookbook_lab_acl"](2)
```

The numbered references from the preceding code are explained as follows:

- (1) The Batfish assertion type that will be executed – in this case, the unreachable lines type.
 - (2) The specific ACL (filter) that should be validated. The format of this parameter is called a filter specifier (more info at <https://pybatfish.readthedocs.io/en/latest/specifiers.html#filter-specifier>).
8. Let's create a small configuration snippet to demonstrate this concept.

```
root@cookbook-lab:~/cookbook-lab/chapters/ch11/unreachable_lines/
configs# vi router.cfg
hostname router
!
interface Ethernet1
... omitted output ...
!
ip access-list cookbook_lab_acl
  10 permit ip 192.168.100.0/24 any
  20 permit ip 192.168.200.0/24 any
  30 deny ip 192.168.0.0/24 any
  40 deny ip 192.168.100.100/24 any (1)
  50 permit ip any any
!
```

The numbered reference from the preceding code is explained as follows:

- (1) Notice that this specific deny line would never be reachable as the permit line at sequence 10 would always be the accepted traffic line. Now, this is somewhat easy to notice in a five-sequence ACL, but what about when it's hundreds or thousands of lines? It becomes nearly impossible.

9. Run the playbook and make sure Batfish identifies the same offending line in the ACL.

```

root@cookbook-lab:~/cookbook-lab/chapters/ch11# ansible-playbook
pb_assert_unreachable_lines.yml
... omitted output ...
TASK [VALIDATE NO UNREACHABLE LINES IN ACLS EXIST] *****
fatal: [localhost]: FAILED! => {"changed": false, "msg":
"1 of 1 assertions failed", "result": [{"details": "Found
unreachable filter line(s), when none were expected\n[{'Sources':
ListWrapper(['router: cookbook_lab_acl']), 'Unreachable_Line': '40
deny ip 192.168.100.100/24 any', 'Unreachable_Line_Action': 'DENY',
'Blocking_Lines': ListWrapper(['10 permit ip 192.168.100.0/24
any']), 'Different_Action': True, 'Reason': 'BLOCKING_LINES',
'Additional_Info': None}]", "name": "ACL VALIDATION", "status":
"Fail", "type": "assert_filter_has_no_unreachable_lines"}],
"summary": "1 of 1 assertions failed"}

```

Great! Batfish found the issue with the ACL. The Ansible assertion failed because Batfish found that sequence 40 would not be reachable; it even provided the exact line (10 permit ip 192.168.100.0/24 any) that would accept the traffic. This is invaluable information when you're trying to secure your network in this ever-threatening cybersecurity world that we live in.

Although you only demonstrated two of the many available assertions that Batfish offers, a few others are listed here. You understand the concepts of how to create the playbooks and how to assert the result you're expecting.

- `assert_all_flows_succeed`: Based on the starting location and destination IP address, port, and protocol, make sure the packets would be routed successfully through the network
- `assert_no_incompatible_bgp_sessions`: Assert that all BGP sessions are configured in a compatible manner
- `assert_no_incompatible_ospf_sessions`: Assert that all OSPF sessions are configured in a compatible manner
- `assert_no_unestablished_bgp_sessions`: Assert that all appropriately configured BGP sessions are established

As already mentioned, there is an important downside of Batfish that must be mentioned here for complete transparency. Batfish is a powerful tool that works the majority of the time, but the application itself is reliant on the backend source code being able to parse configurations, generate a vendor-neutral data model, and calculate all the necessary routing and topology data. For many of the more complex questions, such as BGP and OSPF, the application has fallen behind in supporting the ever-evolving syntax changes and augmentations that vendors support. The BGP peer groups that were used heavily in *Part 2* of the book consistently generated false positives during testing while creating the book, and therefore they had to be removed from examples. The Batfish developers have many documents and even YouTube videos (links in the following note) on how to contribute to this amazing open source project, and we (the authors) strongly suggest documenting issues in GitHub and even contributing to the project to continue its growth. Batfish has multiple different repositories: the main project (Batfish), the Python SDK (pybatfish), and Batfish Ansible Roles. It's important to note that *only* the Batfish Ansible Role repository has been archived; the rest of the project and repositories are still under active development.

Note



The aforementioned developer resources are provided here if you'd like to get involved:

<https://github.com/batfish/batfish/wiki/Developer-notes>

<https://www.youtube.com/watch?v=mJ7LVW0qshk>

Summary

In this chapter, you learned how to install and set up Batfish, initialize a snapshot with the configurations that were generated in *Chapter 4* for the Arista data center topology, and extract the facts that Batfish was aware of from its offline analysis. Finally, you interacted with Batfish custom assertions to make sure your configuration was valid and free of references to unknown configuration elements, you made sure the single ACL named “loopback” didn't have statements that could never be evaluated, and lastly, you learned how to use Batfish offline analysis to validate BGP neighbor compatibility and even that BGP sessions were properly established.

Offline network validation is a powerful tool, allowing an automation engineer to feel confident in network changes before ever touching the live network configurations. There are a few other tools to consider in this space. PyATS is a Cisco project that has multi-vendor support as well as an Ansible collection. It's not necessarily targeted directly at offline analysis, but it can perform some validation. There are other companies, such as Forward Networks, with products offering similar offline network snapshot validation capabilities. Feel free to investigate to see where they fit in your network automation testing pipelines.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



12

Building a Network Inventory with Nautobot

Ansible is more than just an automation engine—it’s the key to streamlining and scaling task execution across your network infrastructure. As you know, every automated task relies on one critical element: *data*. In the previous chapter, we described how to define your network infrastructure using Ansible variables stored in YAML files. While this method works in some cases, it isn’t ideal for organizations aiming to adopt a scalable and sustainable automation strategy.

Enter the **infrastructure as code (IaC)** paradigm, where there’s a fundamental principle: *automation must be data-driven*. The success of your automated processes is only as good as the quality of the data driving them. To achieve consistency, reliability, and scalability, organizations should adopt a robust system that models their network designs—capturing elements such as network inventory, IP addresses, VLANs, cloud networks, tunnels, and more in a centralized and capable data source. This is your **network source of truth (NSoT)**.

Although the term *NSoT* sounds singular, the reality is more complex. Your data might be spread across multiple authoritative sources, and the role of the NSoT is to unify these sources into a *single, consistent view*. This allows you to detect and resolve discrepancies efficiently. Thankfully, several powerful open source solutions exist to fill this role, including **NetBox**, **Nautobot**, and **Infrahub** (in chronological order of release). All the recipes in this chapter can be easily adapted to other tools using the proper Ansible collection.

For this chapter, we’ve chosen Nautobot (<https://docs.nautobot.com/>) as our NSoT. Why Nautobot? It offers out-of-the-box models for both on-premises and cloud environments, making it an excellent fit for hybrid infrastructure needs. While this chapter won’t cover all of Nautobot’s extensive features, we’ll focus on the essentials required to support our examples. If you find areas of your network infrastructure not covered here, we strongly encourage you to explore Nautobot further—it’s highly likely the functionality you need is already included, or the large pluggable app ecosystem has extended it.

Nautobot is an *open source Django-based Python application*, originally forked from NetBox in 2020. It supports popular databases such as **PostgreSQL** and **MySQL** and is designed to do more than just act as a **source of truth (SoT)**. It can also serve as a **single pane of glass (SPOG)** (i.e., a place to coordinate multiple systems) for your entire automation ecosystem. Built with an API-first philosophy, Nautobot provides full capabilities via REST and GraphQL APIs, making it a powerful tool for implementing network automation workflows.

In this chapter, we’ll explore three core use cases for integrating Ansible and Nautobot:

1. **Populating Nautobot with Ansible:** Use Ansible to feed Nautobot with various types of network data, such as locations, devices, **IP address management (IPAM)**, and cloud networks. You will use a common leaf-and-spine architecture to represent a data center and multiple cloud networks from different providers. The data available in Ansible (e.g., group and host variables and the static inventory) is converted into objects in Nautobot.

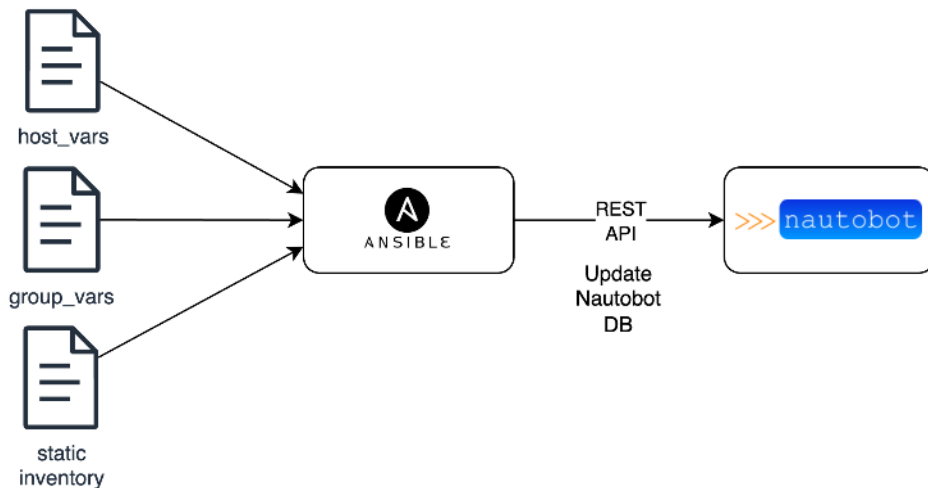


Figure 12.1 – Populating Nautobot with Ansible

2. **Dynamic inventory with Nautobot:** Use Nautobot as a dynamic inventory source for Ansible, enabling Ansible to retrieve and build inventories on the fly. Once Nautobot is populated with data, Ansible can dynamically use it to autogenerate its inventory.

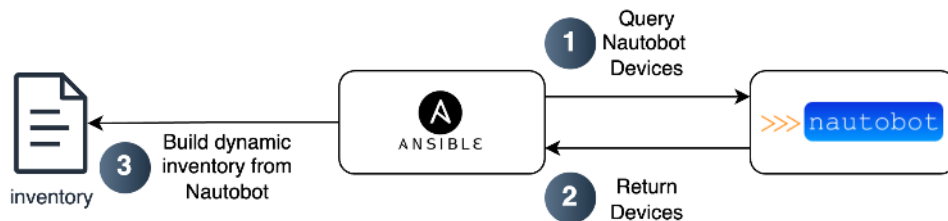


Figure 12.2 – Dynamic inventory with Nautobot

3. **Configuration data sourcing:** Leverage Nautobot as a centralized source of information for generating configuration artifacts—whether CLI configs or API payloads—needed to configure your network devices. While inventory data is fundamental, you may need more data to generate the complete configuration. In this case, Ansible extends its inventory by querying the Nautobot data source.

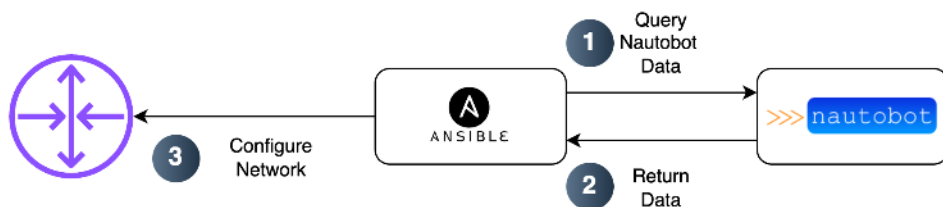


Figure 12.3 – Configuration data sourcing



 Quick tip: Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



You will outline a network scenario resembling those detailed in *Parts 2* and *3* (but not exactly the same). This scenario will feature a leaf-and-spine Arista topology spanning two data centers, complemented by multiple cloud networks distributed across various providers.

This chapter will cover the following recipes:

- Installing Nautobot
- Integrating Nautobot with Ansible
- Populating locations in Nautobot
- Populating devices in Nautobot
- Populating connections in Nautobot
- Populating IP prefixes in Nautobot
- Populating IP addresses in Nautobot
- Populating cloud networks in Nautobot
- Using Nautobot as an Ansible dynamic inventory
- Generating configurations using Nautobot

**Note**

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch12. The lab already includes a Nautobot environment that is up and running, but you will start verifying it.

By the end of this chapter, you'll have the tools and knowledge to integrate Nautobot seamlessly into your Ansible workflows, unlocking the true potential of automated, data-driven network management. Let's dive in!

Installing Nautobot

Nautobot is already installed in the book's lab, so you don't need to install it. However, we think it would be useful to provide you with a high-level overview of how Nautobot is composed, depicting the most relevant components included in *Figure 12.4*:

- **Database:** This is where the SoT lives. The database is where Nautobot stores all of its information about your data models and their relationships to each other.
- **Redis:** This is an open source in-memory database that is used for fast storage and retrieval of key/value pairs. This is used in Nautobot for low-latency caching and as a task queue for Nautobot jobs.
- **Celery worker:** This is the process that executes Nautobot jobs.
- **Celery Beat scheduler:** This is the process that manages the monitoring and dispatching of scheduled jobs in Nautobot.
- **uWSGI:** This is the web server that allows Nautobot Python code to communicate with web clients on the HTTP/HTTPS protocols. It is required to host your web UI, Rest API, and GraphQL API interfaces, and is the core service in the Nautobot architecture.

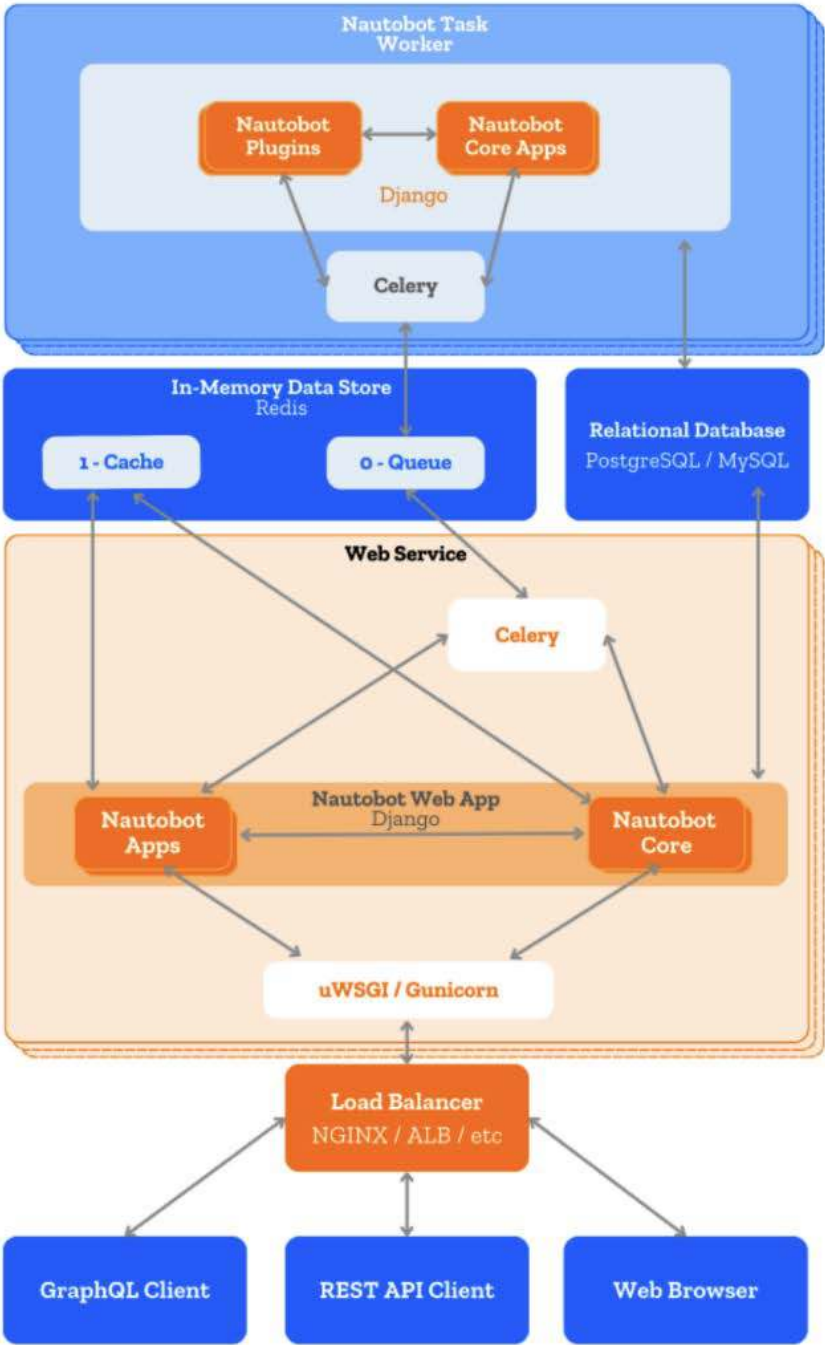


Figure 12.4 – Nautobot application architecture

You can verify the installation by accessing it via `http://<nautobot-server-ip-and-port>/` (in the lab, the server IP and port 8080) and logging in via the credentials (in the lab, by default, the username and password are `admin/nautobot123`). In this example, we used Nautobot version 2.3.13, but the examples will be fully compatible with any minor release in the 2.x track. The following screenshot shows the login page to sign in.

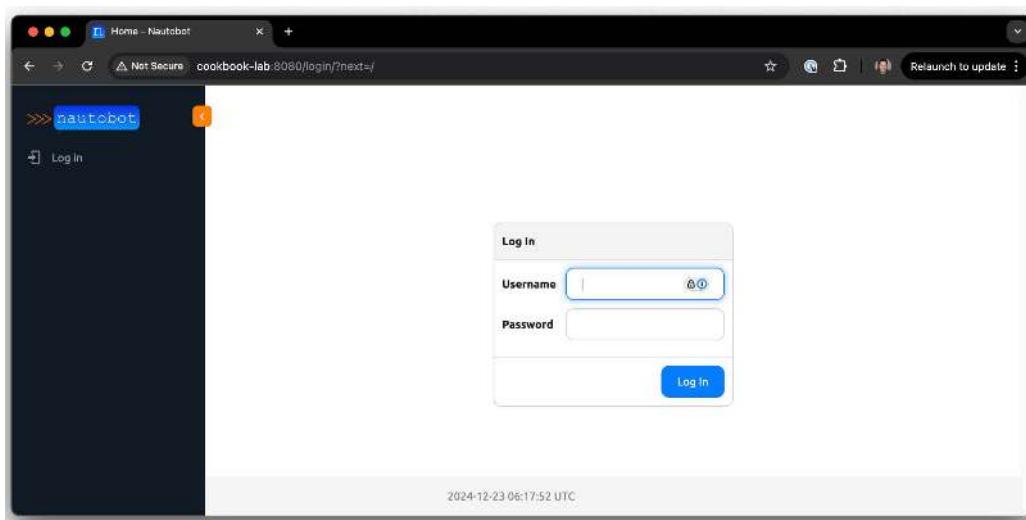


Figure 12.5 – Nautobot login page

After accessing it, you have a clean Nautobot instance ready to be populated.



Note

If you want to get a deeper understanding of Nautobot, we recommend you check out the *Network Automation with Nautobot* book by Packt Publishing.

With Nautobot installed, you can start integrating it with Ansible.

Integrating Nautobot with Ansible

In this recipe, you will get started with the Nautobot Ansible collection and its dependencies, such as `pynautobot`. Also, you will create a service account in Nautobot to allow programmatic access to it with enough permissions:

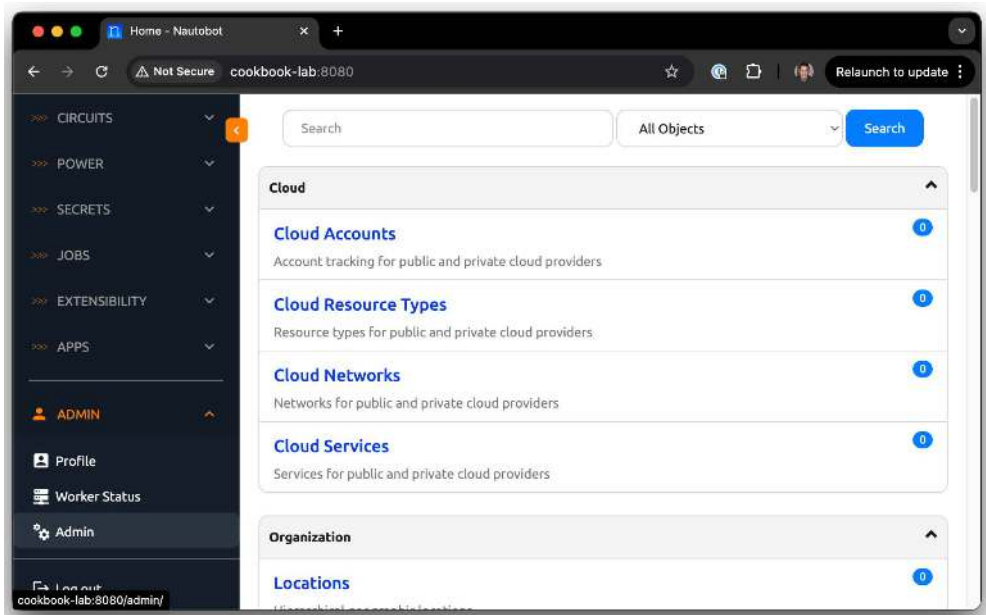
1. On the Ansible control machine, install the Nautobot Ansible collection (more information is available at <https://galaxy.ansible.com/ui/repo/published/networktocode/nautobot>):

```
root@cookbook-lab:~/cookbook-lab# ansible-galaxy collection install
networktocode.nautobot
... omitted output ...
networktocode.nautobot:5.5.0 was installed successfully
```

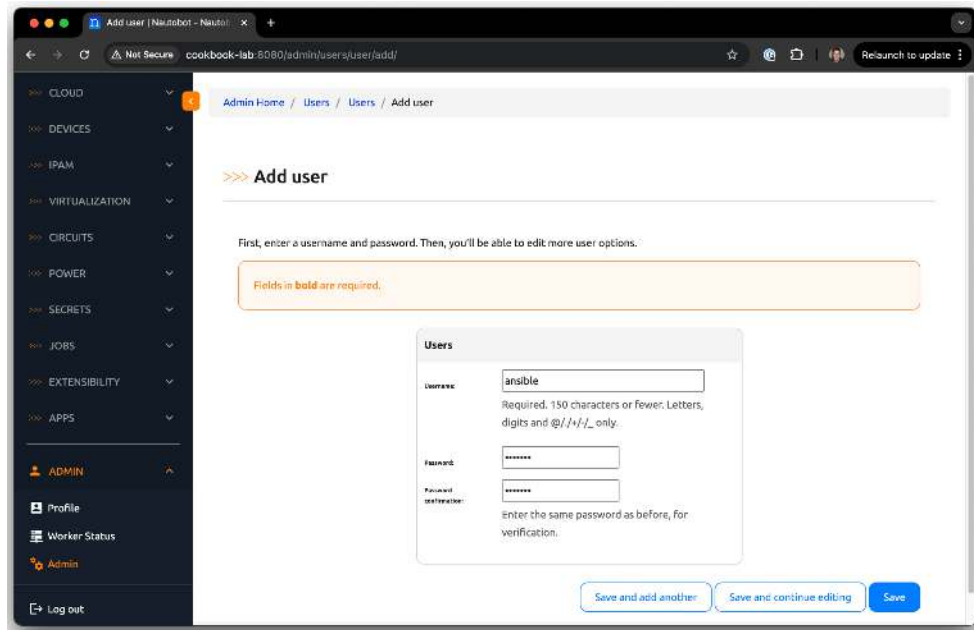
2. Still in the same environment, install the pynautobot Python SDK. It's important to match the version of Nautobot with the Python SDK to have matching capabilities:

```
root@cookbook-lab:~/cookbook-lab# pip install pynautobot
... omitted output ...
Successfully installed packaging-23.2 pynautobot-2.4.2
```

3. Then, log in to the Nautobot server UI using the admin user credentials and go to the **ADMIN | Admin** section to create a service account.

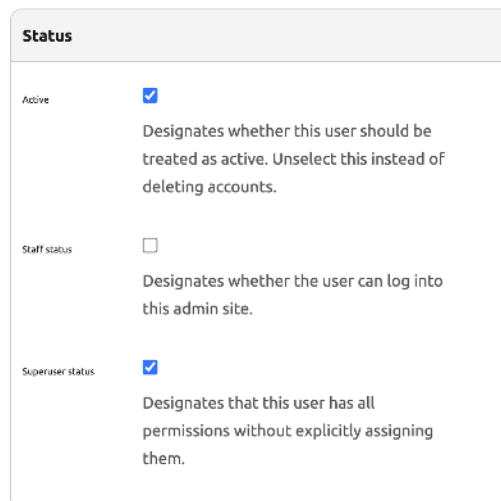


4. On the **Admin** page, create a new user and set its username and password.



The screenshot shows the 'Add user' page in the Nautobot admin interface. The page has a dark sidebar on the left with navigation links for CLOUD, DEVICES, IPAM, VIRTUALIZATION, CIRCUITS, POWER, SECRETS, JOBS, EXTENSIBILITY, and APPS. The 'ADMIN' section is expanded, showing 'Profile', 'Worker Status', 'Admin', and 'Log out'. The main content area is titled 'Add user' and contains a form for adding a new user. The form has three fields: 'Username' (filled with 'ansible'), 'Password', and 'Password confirmation'. A message below the form states: 'First, enter a username and password. Then, you'll be able to edit more user options.' Below this, a yellow box says 'Fields in bold are required.' The 'Username' field is bolded. The 'Password' and 'Password confirmation' fields are also bolded. The 'Password confirmation' field has a note: 'Enter the same password as before, for verification.' At the bottom right, there are three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

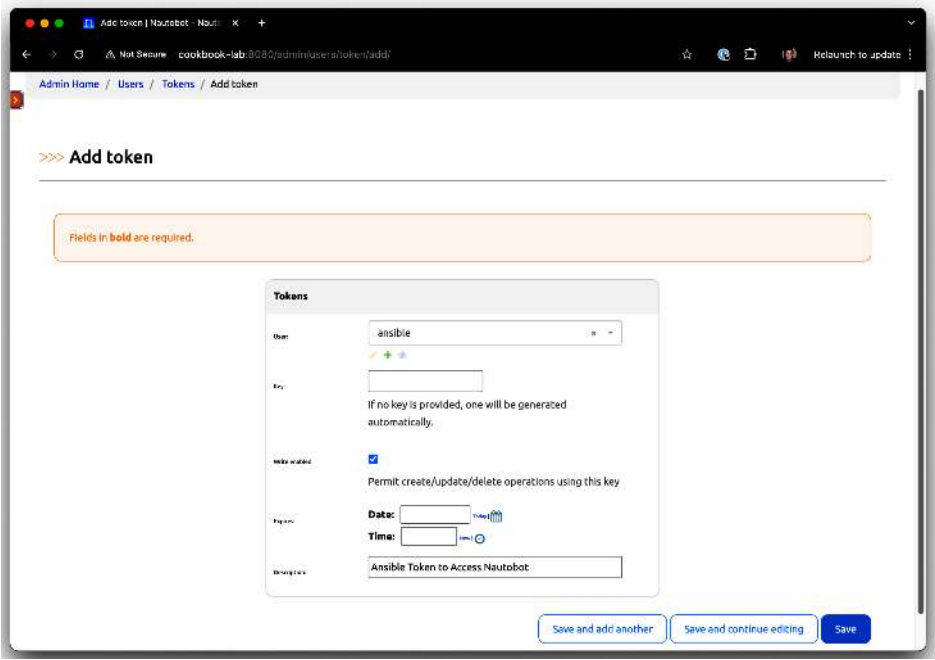
5. When you click **Save** on the **Add User** page, it automatically redirects you to the **User** edit page to assign it to groups and assign permissions, or define it as a superuser to simplify access permissions. In the **Status** section, select **Superuser status** to get full access to the Nautobot database.



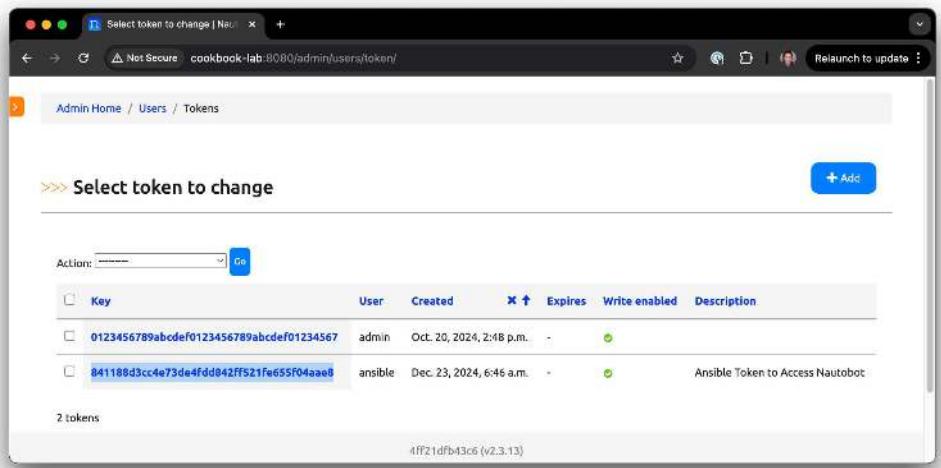
The screenshot shows the 'Status' section in the Nautobot admin interface. It contains three checkboxes with descriptions:

- Active** ☒ Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
- Staff status** ☐ Designates whether the user can log into this admin site.
- Superuser status** ☒ Designates that this user has all permissions without explicitly assigning them.

- 6. Next, in **Admin | Users | Tokens**, let's create a new token to use to identify the service account via an API.



- 7. On the **Tokens** page, you can locate the new token created, associated with the Ansible user:



8. Then, let's prepare the chapter environment by creating a `chapters/ch12` folder and, within it, an `ansible.cfg` file:

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch12 && vi ansible.
cfg
[defaults]
    inventory=hosts
    vault_password_file=~/.ansible_vault_passwd
    gathering=explicit
    transport=local
    retry_files_enabled=False
    action_warnings=False
```

9. To complete the scenario preparation, create a `hosts` Ansible inventory file with the two data center leaf-and-spine topologies that you will populate in the first sections, grouped by data center (`dc1` and `dc2`) and by type (`leaf` and `spine`):

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch12 && cd
chapters/ch12 && vi hosts
[dc1]
dc1-spine01    ansible_host=172.20.1.41
dc1-spine02    ansible_host=172.20.1.42
dc1-leaf01     ansible_host=172.20.1.35
dc1-leaf02     ansible_host=172.20.1.36
[dc2]
dc2-spine01    ansible_host=172.20.2.41
dc2-spine02    ansible_host=172.20.2.42
dc2-leaf01     ansible_host=172.20.2.35
dc2-leaf02     ansible_host=172.20.2.36
[leaf]
dc[1:2]-leaf0[1:2] (1)
[spine]
dc[1:2]-spine0[1:2]
```

The numbered reference from the preceding code is explained as follows:

- (1) The `dc[1:2]-leaf0[1:2]` syntax is equal to `dc1-leaf01`, `dc1-leaf02`, `dc2-leaf01`, and `dc2-leaf02`.

10. Finally, create a `group_vars/all.yml` file to provide the necessary data for the API:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# mkdir group_vars &&
vi group_vars/all.yml
---
nautobot_url: http://127.0.0.1:8080 (1)
nautobot_token: 841188d3cc4e73de4fdd842ff521fe655f04aae8 (2)
```

The numbered references from the preceding code are explained as follows:

- (1) In the book lab, Ansible and Nautobot are running on the same server. This should be adjusted as needed.
- (2) In a real environment, the token should be encrypted with Ansible Vault, but because this is just a lab, we define it in the vars files.

This completes all the basic steps to get Ansible access to Nautobot. Now, you will start creating data objects in Nautobot according to the data center scenario and following the opinionated data model (which can be extended if needed).

Nautobot provides an opinionated data model for representing network infrastructure, enforcing a specific hierarchy of related objects. *Figure 12.6* illustrates the mandatory data model components required to define a Device object—though additional, optional data can also be incorporated, as we do later with IP addresses.

For a deeper understanding of Nautobot's data model, refer to the *Network Automation with Nautobot* book mentioned earlier.

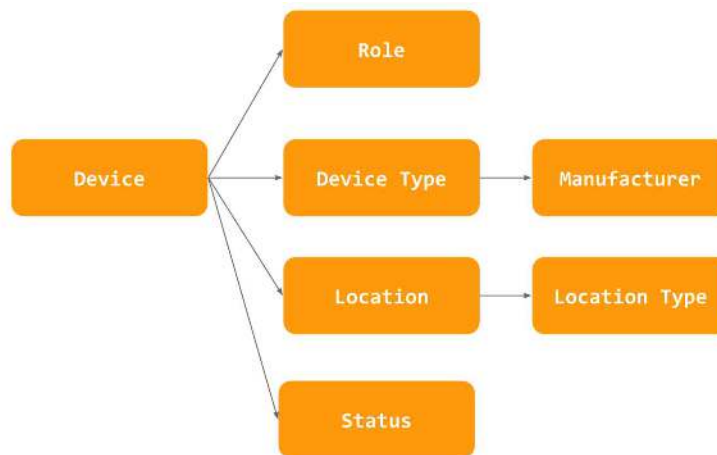


Figure 12.6 – Nautobot device object hierarchy

There is no specific order for the same level, but we start with the locations.

Populating locations in Nautobot

In this recipe, you define **locations** where the network infrastructure will be placed. Locations are a logical construct within Nautobot that allows you to group the infrastructure based on the physical location, which is defined with a location type to represent a site or a building:

1. Update the `group_vars/all.yml` file defining the two data centers:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/all.
yml
... omitted output ...
locations:
  - name: DC1
    description: "Main Data Center in Sydney"
    address: Sydney
    location_type: "datacenter"
  - name: DC2
    description: "Main Data Center in KSA"
    address: Riyadh
    location_type: "datacenter"
```

2. Create a new roles directory under `ch12`, and create a new Ansible role called `build_nautobot_db`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# mkdir roles && cd
roles && ansible-galaxy init build_nautobot_db
- Role build_nautobot_db was created successfully
```

3. Populate the `tasks/main.yml` file to create different location types. In this case, there is only one type of location to represent a data center, but this could be much more specific, such as a room or cabinet:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
---
- name: Create Location Types in Nautobot
  networktocode.nautobot.location_type:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
```

```

    name: "{{ item | lower }}"
    state: "{{ nautobot_state }}"
    content_types: (1)
      - "dcim.device"
      - "ipam.prefix"
    loop: "{{ locations | map(attribute='location_
type') | unique | list }}" (2)
    when: locations is defined
    run_once: yes (3)

```

The numbered references from the preceding code are explained as follows:

- (1) The `content_types` instance defines which objects are suitable to associate with the locations that are assigned to this location type.
 - (2) The location types are embedded within the locations definition. Thus, we extract it from the `map` attribute and get only unique items from it.
 - (3) It's only necessary to run this playbook once, not for all the devices in the inventory.
4. Still in the same file, create locations with a Planned status (this represents that the data centers are not yet ready for use):

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Locations in Nautobot
  networktoencode.nautobot.location:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    name: "{{ item.name | lower }}"
    description: "{{ item.description | default(omit) }}" (1)
    physical_address: "{{ item.address | default(omit) }}" (1)
    state: "{{ nautobot_state }}" (2)
    status: "Planned" (3)
    location_type:
      name: "{{ item.location_type }}" (4)
    loop: "{{ locations }}"
    when: locations is defined
    run_once: yes

```

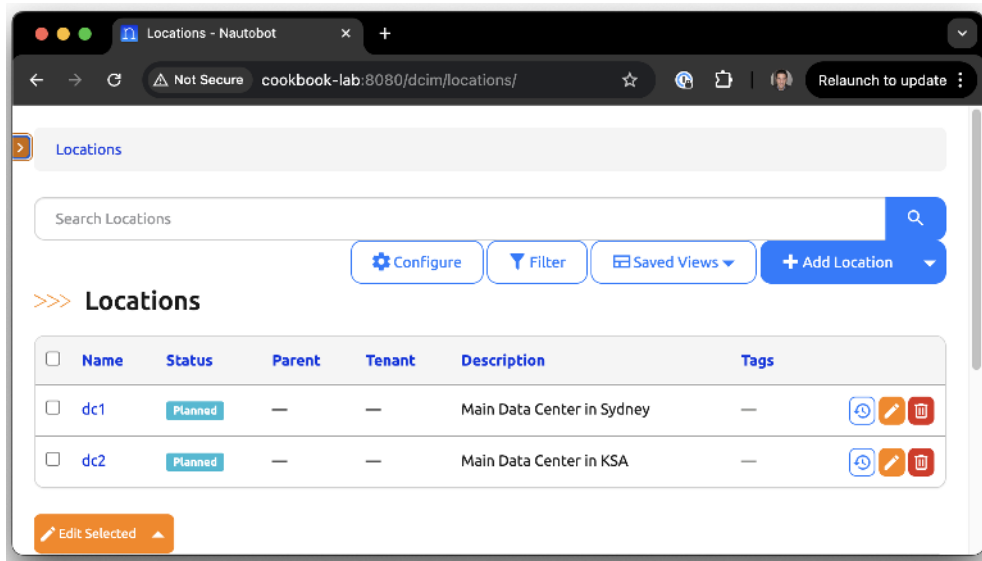
The numbered references from the preceding code are explained as follows:

- (1) The location's `description` and `physical_address` attributes are defined if present in the dictionary.
 - (2) The state module allows two options, `present` or `absent`, to represent the declarative state of the object in the Nautobot database.
 - (3) To define the intended state of the location, you define a `status` attribute. Notice that `Planned` is an already existing status in Nautobot, but you could use one that better represents your intention (creating it first).
 - (4) A location needs a location type, and this references the ones created before.
5. Then, within the `tasks/main.yml` role, include the tasks defined in the `create_datacenter.yml` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/main.yml
---
- name: Create Nautobot Datacenter data
  ansible.builtin.import_tasks: create_datacenter.yml (1)
  tags:
    - datacenter (2)
```

The numbered references from the preceding code are explained as follows:

- (1) The `import_tasks` attribute is a directive used to include tasks from another file into your playbook or role statically. This means tasks are imported and evaluated at the time the playbook is parsed, rather than dynamically during execution.
 - (2) Tagging the task allows all tasks to be selected using filter tags.
6. When you run the playbook with `ansible-playbook pb_build_nautobot_db.yml`, objects are created in Nautobot.



Now that locations have been defined, it's time to populate data center devices in Nautobot.

Populating devices in Nautobot

Network devices, along with other constructs such as virtual networks, form the inventory of your network. In this recipe, you will *create and populate* network devices based on the network design outlined earlier in this chapter using Nautobot. This will serve as the authoritative inventory of your network infrastructure, which you will later leverage to build a dynamic Ansible inventory directly from Nautobot:

1. Start by defining two different *device kinds* to represent both the leaf-and-spine switches, each one with a specific role and type. Update the `group_vars/all.yml` file with the devices' information, as follows:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/all.
yml
... omitted output ...
device_kinds:
  - role:
      name: Leaf_Switch
      color: 'f44336' # red
    type:
      model: 7020SR
      vendor: Arista
```

```
- role:
  name: Spine_Switch
  color: '2196f3' # blue
  type:
    model: 7050CX3
    ru: 2
    vendor: Arista
```

2. Create `group_vars/leaf.yml` and `group_vars/spine.yml` files, then update them with the following information:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/leaf.
yml
---
device_model: 7020SR
device_role: Leaf_Switch
vendor: Arista
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/spine.
yml
---
device_model: 7050CX3
device_role: Spine_Switch
vendor: Arista
```

3. Create a new task to create a manufacturer for all of the devices in our inventory under the `tasks/create_datacenter.yml` file, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Manufacturers within Nautobot
  networktocode.nautobot.manufacturer:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ item | lower }}"
  loop: "{{ device_kinds | map(attribute='type') |
    map(attribute='vendor') | unique | list }}" (1)
  when: device_kinds is defined
  run_once: yes
```

The numbered reference from the preceding code is explained as follows:

- (1) From the `device_kinds` variable, extract all the unique vendors under the type dict.
4. Next, create two role types for the devices: leaf and spine. Add a new task in `tasks/create_datacenter.yml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create roles within Nautobot for Devices
  networktoencode.nautobot.role:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ item.name | lower }}"
    color: "{{ item.color }}"
    content_types: (1)
    - "dcim.device"
  loop: "{{ device_kinds | map(attribute='role') | unique | list }}"
  when: device_kinds is defined
  run_once: yes
```

The numbered reference from the preceding code is explained as follows:

- (1) A role in Nautobot can be applied to different types of objects (similar to locations), so the supported types have to be defined.
5. Then, create different device types, extracting data from the `device_kinds` variable. This is a simplified approach to creating device types. We recommend importing them from <https://github.com/nautobot/devicetype-library/> and importing other data, such as the interface definition:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create device types within Nautobot
  networktoencode.nautobot.device_type:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
```

```

model: "{{ item.model }}"
manufacturer: "{{ item.vendor | lower }}"
u_height: "{{ item.ru | default(omit) }}"
loop: "{{ device_kinds | map(attribute='type') | unique | list }}"
when: device_kinds is defined
run_once: yes

```

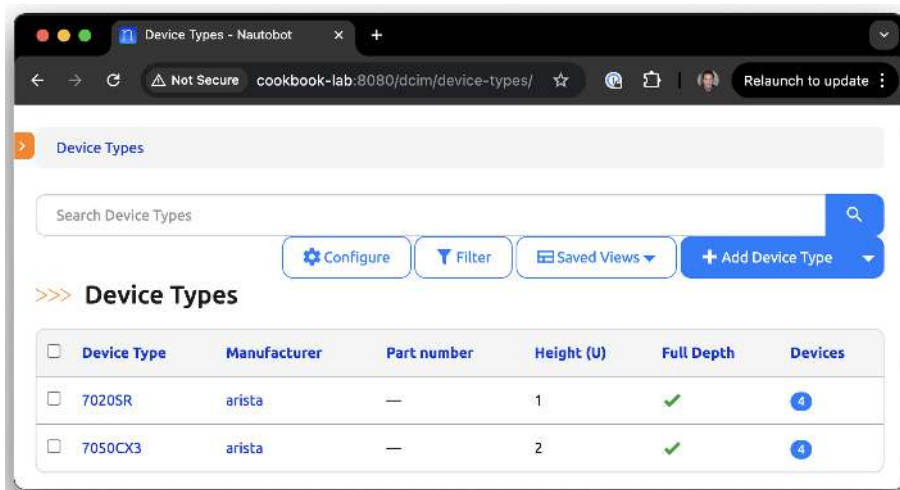
6. Finally, create all eight data center devices, updating the `tasks/create_datacenter.yml` file, as shown here:

```

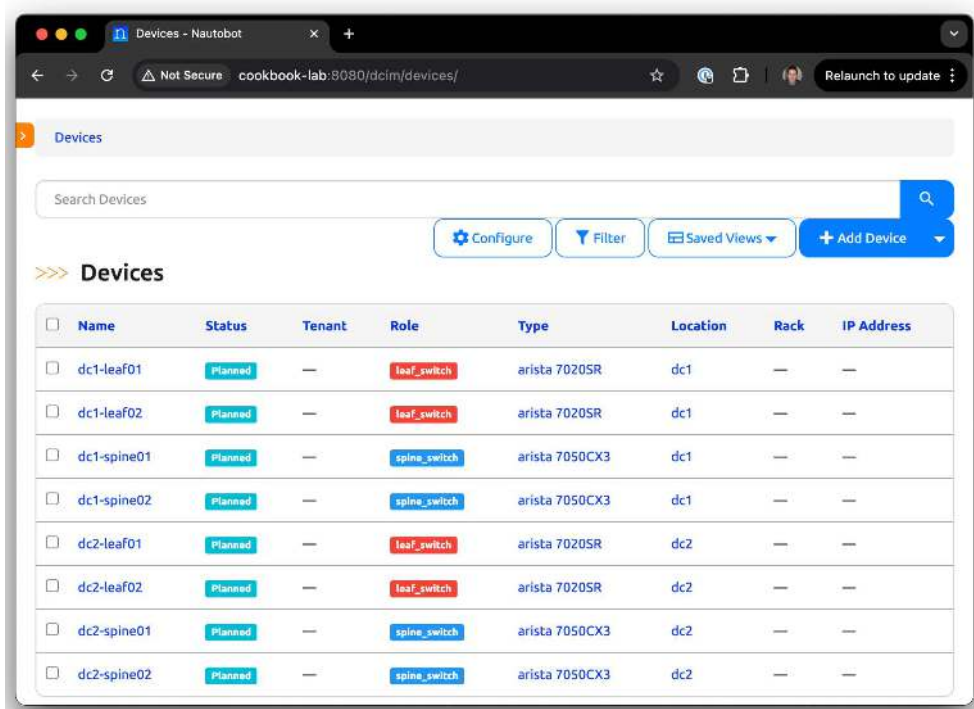
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create devices within Nautobot
  networktocode.nautobot.device:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ inventory_hostname | lower }}"
    device_type: "{{ device_model }}"
    role: "{{ device_role | lower }}"
    location: "{{ inventory_hostname.split('-')[0] }}"
    status: "Planned"

```

7. When executing the playbook, you will get all objects created:



Device Type	Manufacturer	Part number	Height (U)	Full Depth	Devices
☐ 7020SR	arista	—	1	✓	4
☐ 7050CX3	arista	—	2	✓	4



☐	Name	Status	Tenant	Role	Type	Location	Rack	IP Address
☐	dc1-leaf01	Planned	—	leaf_switch	arista 7020SR	dc1	—	—
☐	dc1-leaf02	Planned	—	leaf_switch	arista 7020SR	dc1	—	—
☐	dc1-spine01	Planned	—	spine_switch	arista 7050CX3	dc1	—	—
☐	dc1-spine02	Planned	—	spine_switch	arista 7050CX3	dc1	—	—
☐	dc2-leaf01	Planned	—	leaf_switch	arista 7020SR	dc2	—	—
☐	dc2-leaf02	Planned	—	leaf_switch	arista 7020SR	dc2	—	—
☐	dc2-spine01	Planned	—	spine_switch	arista 7050CX3	dc2	—	—
☐	dc2-spine02	Planned	—	spine_switch	arista 7050CX3	dc2	—	—

Note



Keep in mind that the arguments used in the recipes depend on the data model. Nautobot adheres to strict *semantic versioning*, so you should only expect breaking changes besides major releases.

Having created the devices, you will next create all necessary connections between them.

Populating connections in Nautobot

After creating the devices, you will create all interfaces and necessary connections to establish the fabric topology, plus the management and loopback interface, as follows:

1. Update the `group_vars/create_datacenter.yml` file with point-to-point links within our network fabric in each data center, as follows:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/all.
yml
... omitted output ...
```

```
p2p_ip:
  dc1-leaf01:
    - {port: Ethernet8, ip: 172.30.1.1/31 , peer: dc1-spine01,
      pport: Ethernet1, peer_ip: 172.30.1.0/31}
    - {port: Ethernet9, ip: 172.30.1.5/31 , peer: dc1-spine02,
      pport: Ethernet1, peer_ip: 172.30.1.4/31}
  dc1-leaf02:
    - {port: Ethernet8, ip: 172.30.1.3/31 , peer: dc1-spine01,
      pport: Ethernet2, peer_ip: 172.30.1.2/31}
... omitted output ...
```

2. Create a new task to create interfaces for the fabric in the `tasks/create_datacenter.yml` file using the information contained in the previous variable, `p2p_ip`, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Fabric interfaces within Nautobot
  networktoencode.nautobot.device_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    device: "{{ inventory_hostname }}"
    name: "{{ item.port }}"
    description: "{{ item.type | default('CORE') }}" | "{{ item.
peer }}" | "{{ item.pport }}"
    enabled: true
    mode: "Access"
    status: "Active"
    type: "1000Base-t (1GE)"
    loop: "{{ p2p_ip[inventory_hostname] }}"
    when: p2p_ip is defined
```

3. After the fabric interfaces, create one management interface (using the `mgmt_only` Boolean) in each device, adding new tasks in the `tasks/create_datacenter.yml` file, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
```

```
- name: Create Management Interfaces on Devices
  networktocode.nautobot.device_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    device: "{{ inventory_hostname }}"
    name: "Management1"
    description: "{{ 'Management Interface ' + inventory_hostname }}"
    enabled: true
    mode: "Access"
    mgmt_only: true
    status: "Active"
    type: "1000Base-t (1GE)"
```

4. Also, create a new task to create a loopback interface (with type Virtual) for all of the devices in the inventory under the tasks/create_datacenter.yml file, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Loopback0 Interface on Devices
  networktocode.nautobot.device_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    device: "{{ inventory_hostname }}"
    name: "Loopback0"
    description: "{{ 'Loopback0 Interface ' + inventory_hostname }}"
    enabled: true
    mode: "Access"
    status: "Active"
    type: "Virtual"
```

5. Finally, using the same information, let's connect the interfaces according to the plan, updating the `tasks/create_datacenter.yml` file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create cables within Nautobot
  networktocode.nautobot.cable:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    termination_a_type: dcim.interface
    termination_a:
      device: "{{ inventory_hostname }}"
      name: "{{ item.port }}"
    termination_b_type: dcim.interface
    termination_b:
      device: "{{ item.peer }}"
      name: "{{ item.pport }}"
    status: "Planned"
    loop: "{{ p2p_ip[inventory_hostname] }}"
    when: "p2p_ip is defined and 'spine' in group_names" (1)
    ignore_errors: yes (2)
```

The numbered references from the preceding code are explained as follows:

- (1) Every device in `p2p_ip` has the opposite connection, so to avoid duplicating cables, limit the task to only spine devices.
- (2) Even though the previous filter avoids duplicating the leaf-to-spine connections, between the spines, there is still duplication, and only one cable can be attached to one interface. We could add some more complex logic, but a simple approach is to acknowledge it with `ignore_errors: yes` to continue executing.

- The outcome of these tasks is several interfaces for each device plus the connections/cables between them. If you select one device and navigate to the **Interfaces** tab, you will see something like this:

Name	Status	Enabled	Type	Mode	Description	Cable	Connection	IP Addresses
Ethernet1	Active	✓	1000BASE-T (1GE)	Access	CORE dc1-leaf01 Ethernet8	#e60c08c0-3386-427D-B1e5-c48f08d03c22	dc1-leaf01 > Ethernet8	
Ethernet2	Active	✓	1000BASE-T (1GE)	Access	CORE dc1-leaf02 Ethernet8	#a0f6d15c-d6ef-4fe4-b803-3f5909c10b2d	dc1-leaf02 > Ethernet8	
Ethernet6	Active	✓	1000BASE-T (1GE)	Access	CORE dc1-spine02 Ethernet6	#c9cd0d3c-08e6-4a94-a17e-f5e0db433c0f	dc1-spine02 > Ethernet6	
Loopback0	Active	✓	Virtual	Access	Loopback0 Interface dc1-spine01	—	—	
Management1	Active	✓	1000BASE-T (1GE)	Access	Management Interface dc1-spine01	—	—	

In the previous screenshot, notice that the IP address information is still missing. Our next goal is to provide this.

Populating IP prefixes in Nautobot

IP address and prefix information are the cornerstone of IP networks. An NSoT should offer robust tools for managing this data, a functionality commonly referred to as IPAM. However, an effective NSoT must also integrate this information with other network elements, such as locations and interfaces. In this section, you will define prefixes, and in the following section, you will create the corresponding IP addresses:

Note

Nautobot provides modeling for other IPAM data such as namespaces, **virtual routing and forwarding functions (VRF functions)**, or VLANs, but we decided to keep it simple here.

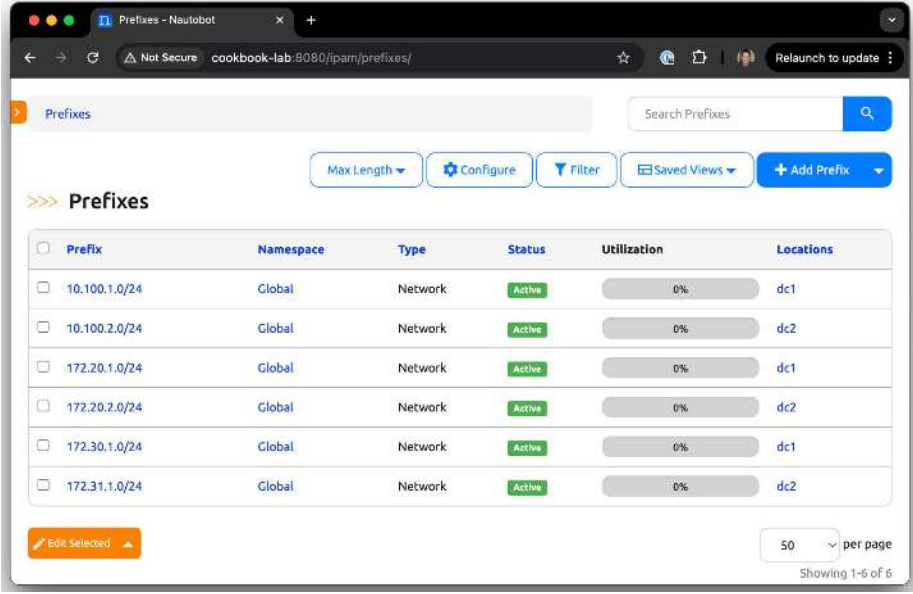
1. Update the `group_vars/all.yml` file with the prefix information under the `subnets` variable:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/all.
yml
... omitted output ...
subnets:
  - prefix: 172.30.1.0/24
    location: dc1
  - prefix: 172.31.1.0/24
    location: dc2
  - prefix: 10.100.1.0/24
    location: dc1
  - prefix: 10.100.2.0/24
    location: dc2
  - prefix: 172.20.1.0/24
    location: dc1
  - prefix: 172.20.2.0/24
    location: dc2
```

2. Create a new task to create prefixes under the `tasks/create_datacenter.yml` file, as shown here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create IP Prefixes within Nautobot
  networktoencode.nautobot.prefix:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    prefix: "{{ item.prefix }}"
    location: "{{ item.location | default(omit) }}"
    status: "Active"
  loop: "{{ subnets }}"
  loop_control:
    label: "{{ item.prefix }}"
  run_once: yes
```

- After running the playbook, you can see the six prefixes used in the data center network.



☐	Prefix	Namespace	Type	Status	Utilization	Locations
☐	10.100.1.0/24	Global	Network	Active	0%	dc1
☐	10.100.2.0/24	Global	Network	Active	0%	dc2
☐	172.20.1.0/24	Global	Network	Active	0%	dc1
☐	172.20.2.0/24	Global	Network	Active	0%	dc2
☐	172.30.1.0/24	Global	Network	Active	0%	dc1
☐	172.31.1.0/24	Global	Network	Active	0%	dc2



Note

In the previous screenshot, you may notice that the utilization is 0%. We encourage you to check it later, when some IP addresses are created, to see how this data changes.

With the prefixes created, you can proceed to create IP addresses (even though this is not a prerequisite).

Populating IP addresses in Nautobot

Finally, to complete the data center setup, we will create and associate IP addresses with the previously created interfaces, providing a complete and detailed view of the scenario:

- The IP addresses for the management interface match the `ansible_host` variable defined in the hosts file. For the loopback interface, you will add a new `loopback_ip` variable in the same file:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi hosts
[dc1]
dc1-spine01      ansible_host=172.20.1.41 loopback_ip=10.100.1.254/32
dc1-spine02      ansible_host=172.20.1.42 loopback_ip=10.100.1.253/32
... omitted output ...
```

2. Update the `create_datacenter.yml` file with a new task to create a loopback IP address (using the previously defined variable) and attach the IP address to the loopback interface:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Loopback IPs within Nautobot
  networktocode.nautobot.ip_address:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    address: "{{ loopback_ip }}"
    status: "Active"
  register: loopback_ip_obj (1)
- name: "Add Loopback IP addresses within Nautobot"
  networktocode.nautobot.ip_address_to_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    ip_address: "{{ loopback_ip_obj.ip_address.id }}" (1)
    interface: (2)
      name: "Loopback0"
      device: "{{ inventory_hostname }}"
```

The numbered references from the preceding code are explained as follows:

- (1) The IP address to attach is retrieved from the output of the IP address creation.
 - (2) The interface is identified with the device and interface names.
3. Repeat the previous step, but this time, for the management interface:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Management IPs within Nautobot
  networktocode.nautobot.ip_address:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    address: "{{ ansible_host }}/32"
```

```

    status: "Active"
  register: management_ip_obj
- name: "Add Management IP addresses within Nautobot"
  networktocode.nautobot.ip_address_to_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    ip_address: "{{ management_ip_obj.ip_address.id }}"
    interface:
      name: "Management1"
      device: "{{ inventory_hostname }}"

```

4. In addition, once the management IP is created, you can update the device, defining it as the primary IP to reach it. This is important as it becomes the Ansible host IP:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Make management IP the primary IP for devices
  networktocode.nautobot.device:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    name: "{{ inventory_hostname | lower }}"
    state: "{{ nautobot_state }}"
    primary_ip4: "{{ management_ip_obj.ip_address.id }}"

```

5. Finally, use the same approach to create IP addresses for all the leaf-and-spine interfaces using the `p2p_ip` variable:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_datacenter.yml
... omitted output ...
- name: Create Fabric IPs within Nautobot
  networktocode.nautobot.ip_address:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    address: "{{ item.ip }}"
    status: "Active"

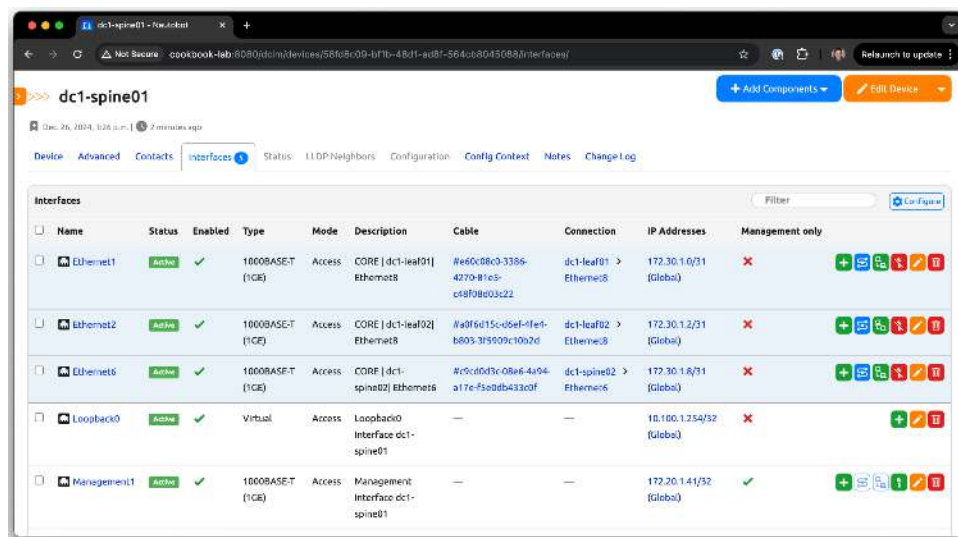
```

```

loop: "{{ p2p_ip[inventory_hostname] }}"
when: p2p_ip is defined
register: fabric_ip_objs
- name: "Add Fabric IP addresses within Nautobot"
  networktocode.nautobot.ip_address_to_interface:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    ip_address: "{{ item.ip_address.id }}"
    interface:
      name: "{{ item.item.port }}"
      device: "{{ inventory_hostname }}"
    loop: "{{ fabric_ip_objs.results }}"

```

- After executing the updated playbook, you can check the **Interfaces** tab for a device again and see the IP addresses included.



Name	Status	Enabled	Type	Mode	Description	Cable	Connection	IP Addresses	Management only
Ethernet1	Active	✓	1000BASE-T (1GbE)	Access	CORE dc1-leaf01 Ethernet8	#e60c8b0-3386-4270-81c5-ct8f0803c22	dc1-leaf01 > Ethernet8	172.30.1.0/31 (Global)	✗
Ethernet2	Active	✓	1000BASE-T (1GbE)	Access	CORE dc1-leaf02 Ethernet8	#a0f6d15c-d5ef-4fe1-b805-3f9909c1092d	dc1-leaf02 > Ethernet8	172.30.1.2/31 (Global)	✗
Ethernet6	Active	✓	1000BASE-T (1GbE)	Access	CORE dc1-spine02 Ethernet6	#9c0cd43c-08a6-4a96-a17e-f50db433c0f	dc1-spine02 > Ethernet6	172.30.1.8/31 (Global)	✗
Loopback0	Active	✓	Virtual	Access	Loopback0 Interface dc1-spine01	—	—	10.100.1.254/32 (Global)	✗
Management1	Active	✓	1000BASE-T (1GbE)	Access	Management Interface dc1-spine01	—	—	172.20.1.41/32 (Global)	✓

With this, you have completed the required information to model the data center topology. You may argue that this is not all the information you need to configure your topology (for example, missing SNMP parameters). We leave this to your exploration to research the other options you have to cover all the data you need in Nautobot.

Now, we change context to create cloud network information in Nautobot.

Populating cloud networks in Nautobot

Cloud networking is a relatively new concept in the field of networking. It introduces ideas such as virtual networks and accounts while building on existing principles such as IPAM. In this recipe, you will configure the foundational data within a new task file for a role, defining two virtual networks for each of the cloud providers covered in this book—AWS, Azure, and GCP:

1. Update the hosts file to include the new virtual networks within each cloud provider group and within a global cloud group to allow filtering with an associated IP prefix:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi hosts
... omitted output ...
[aws]
aws_network_01 prefix=10.200.1.0/24
aws_network_02 prefix=10.200.2.0/24
[azure]
azure_network_01 prefix=10.201.1.0/24
azure_network_02 prefix=10.201.2.0/24
[gcp]
gcp_network_01 prefix=10.202.1.0/24
gcp_network_02 prefix=10.202.2.0/24
[cloud:children] (1)
aws
azure
gcp
```

The numbered reference from the preceding code is explained as follows:

- (1) The cloud group allows you to run the playbook filtering per host with the `-l` option:
`ansible-playbook pb_build_nautobot_db.yml -l cloud.`
2. Update the `main.yml` role file that you will create next, including all cloud-related tasks:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/main.yml
... omitted output ...
- name: Create Nautobot Cloud data
  import_tasks: create_cloud.yml
  when: "'cloud' in group_names" (1)
  tags:
```

```
- cloud
```

The numbered reference from the preceding code is explained as follows:

- (1) In this case, the tasks for the cloud are only relevant to cloud-related hosts.
- 3. Create a `create_cloud.yml` file, and add a task to create cloud providers that are modeled as manufacturers:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
- name: Create cloud providers (Manufacturers) within Nautobot
  networktocode.nautobot.manufacturer:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ group_names | difference(['cloud']) | first }}" (1)
    throttle: 1 (2)
```

The numbered references from the preceding code are explained as follows:

- (1) The provider name is taken directly from the groups that the virtual network belongs to, removing the cloud one.
- (2) The throttle option prevents two virtual networks from the same provider from interfering with the API.
- 4. Similarly to the previous task, now update the `create_cloud.yml` file to create account information. First, include the information in new `group_vars` files, one for each provider:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi group_vars/aws.
yml
---
account_id: "123456789012"
... omitted output for other providers ...
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
... omitted output ...
- name: Create cloud account within Nautobot
  networktocode.nautobot.cloud_account:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
```

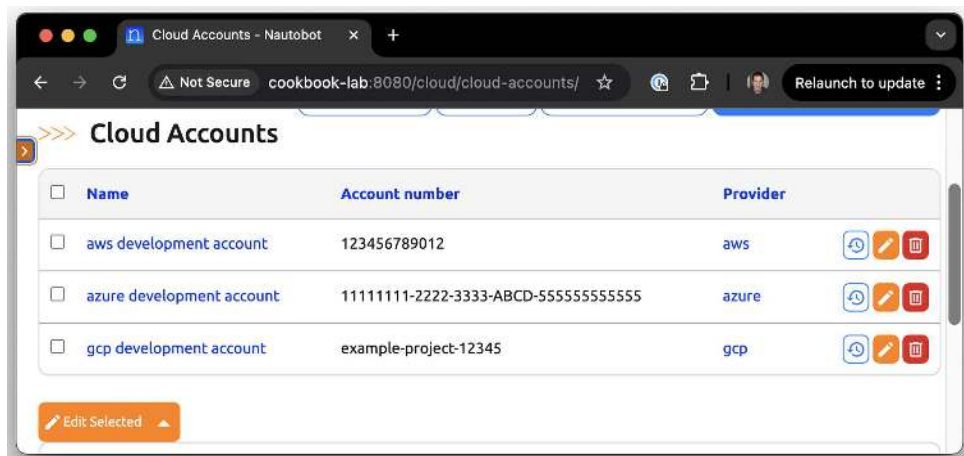


```

state: "{{ nautobot_state }}"
name: "{{ group_names | difference(['cloud']) | first }}
      development account"
provider: "{{ group_names | difference(['cloud']) | first }}"
account_number: "{{ account_id }}"
throttle: 1

```

If you execute the playbook again, as described in *step 1*, you will obtain the following cloud accounts in Nautobot.



5. The next Nautobot construct you need is the cloud resource type. You will define two types, one for the virtual network and another for the subnets that will be created within it:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
... omitted output ...
- name: Create cloud network resource type within Nautobot
  networktoencode.nautobot.cloud_resource_type:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ group_names | difference(['cloud']) | first }}
          network" (1)
    provider: "{{ group_
names | difference(['cloud']) | first }}" (2)
    content_types: (3)
    - "cloud.cloudnetwork"

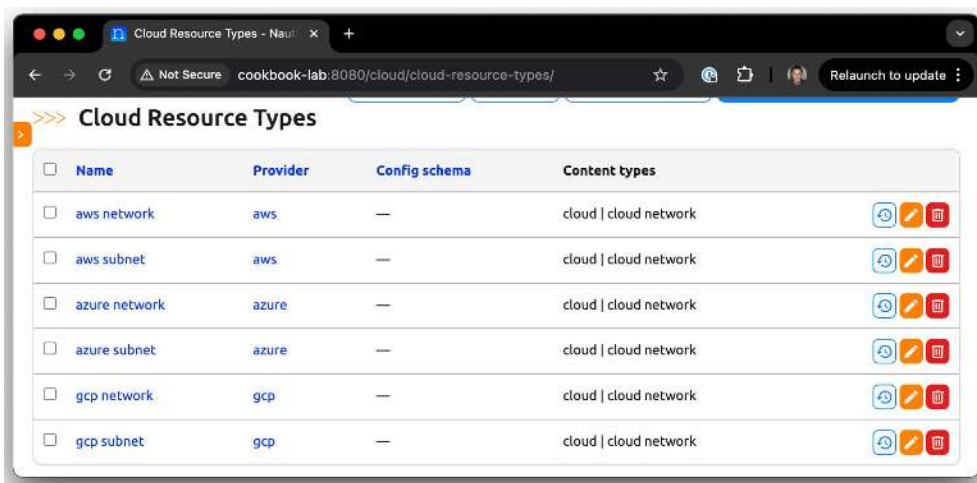
```

```
throttle: 1
... omitted a similar task for the subnet ...
```

The numbered references from the preceding code are explained as follows:

- (1) The difference between the two types is the name, one with network and the other with subnet.
- (2) The cloud resource type belongs to a specific provider.
- (3) This type can be used by cloud networks.

Running the playbook again, these cloud resource types will be populated in Nautobot.



6. With the network and subnet types created, you can continue updating the `create_cloud.yml` files with actual virtual networks and subnet creation:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
... omitted output ...
- name: Create cloud networks within Nautobot
  networktocode.nautobot.cloud_network:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ inventory_hostname }} network"
    cloud_resource_type: "{{ group_names | difference(['cloud']) |
                           first }}" network" (1)
```

```

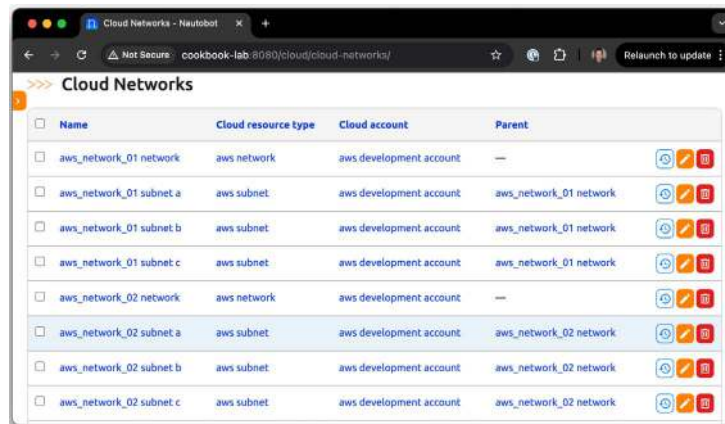
    cloud_account: "{{ account_id }}"
- name: Create cloud subnets within Nautobot
  networkcode.nautobot.cloud_network:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    name: "{{ inventory_hostname }} subnet {{ item }}"
    cloud_resource_type:
      "{{ group_names | difference(['cloud']) | first }} subnet" (1)
    cloud_account: "{{ account_id }}"
    parent: "{{ inventory_hostname }} network" (2)
  loop: (3)
    - a
    - b
    - c

```

The numbered references from the preceding code are explained as follows:

- (1) Each network gets the right cloud resource type.
- (2) With the parent option, you explicitly define that the subnets belong to the parent network.
- (3) Looping over a, b, and c provides an easy way to create three subnets per network.

Executing the Ansible playbook again, the following cloud networks, associated with the cloud resource types and accounts, will be created.



The screenshot shows the Nautobot web interface for Cloud Networks. It displays a table with columns: Name, Cloud resource type, Cloud account, and Parent. The table lists two main networks, each with three associated subnets.

Name	Cloud resource type	Cloud account	Parent
aws_network_01 network	aws network	aws development account	—
aws_network_01 subnet a	aws subnet	aws development account	aws_network_01 network
aws_network_01 subnet b	aws subnet	aws development account	aws_network_01 network
aws_network_01 subnet c	aws subnet	aws development account	aws_network_01 network
aws_network_02 network	aws network	aws development account	—
aws_network_02 subnet a	aws subnet	aws development account	aws_network_02 network
aws_network_02 subnet b	aws subnet	aws development account	aws_network_02 network
aws_network_02 subnet c	aws subnet	aws development account	aws_network_02 network

Next, you can create and assign network prefixes using the prefix per-host variable using the same approach as for the data center task,

1. First, the network prefixes and the related attached are created:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
... omitted output ...
- name: Create Cloud IP Prefixes for networks within Nautobot
  networkcode.nautobot.prefix:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    prefix: "{{ prefix }}"
    status: "Active"
- name: Create a cloud networks to prefix assignment
  networkcode.nautobot.cloud_network_prefix_assignment:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    cloud_network: "{{ inventory_hostname }} network"
    cloud_prefix: "{{ prefix }}"
```

2. After the prefix assignment to the virtual network, it's time to create prefixes for the subnets. However, to avoid defining subnets manually, we propose to create a lookup plugin that calculates the subnetting for us:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# mkdir roles/build_
nautobot_db/lookup_plugins && vi roles/build_nautobot_db/lookup_
plugins/calculate_subnets.py
from ansible.errors import AnsibleError, AnsibleParserError
from ansible.plugins.lookup import LookupBase (1)
import netaddr (2)
class LookupModule(LookupBase): (1)
    """
    This lookup plugin calculates subnets from a given network prefix.
    Example usage:
        "{{ lookup('calculate_subnets', '192.168.0.0/24', 26) }}"
    """
```

```
def run(self, terms, variables=None, **kwargs):
    network = terms[0]
    subnet_size = int(terms[1])
    try:
        ip_network = netaddr.IPNetwork(network) (3)
        subnets = list(ip_network.subnet(subnet_size)) (4)
        return [str(subnet) for subnet in subnets] (5)
    except Exception as e:
        raise AnsibleError(f"Error calculating subnets: {str(e)}")
```

The numbered references from the preceding code are explained as follows:

- (1) Import and define the expected `LookupModule` class that is auto-discovered by Ansible.
 - (2) Import the `netaddr` library that provides helpers to calculate the subnets.
 - (3) Transform the prefix from a string to the `IPNetwork` class.
 - (4) Generate subnets according to the `subnet_size` variable.
 - (5) Return a string representation of the `IPNetwork` objects.
3. Install the `netaddr` Python library, which the lookup plugin requires:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# pip install netaddr
```

4. Update the `create_cloud.yml` tasks with the prefix creation and assignment for subnets:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi roles/build_
nautobot_db/tasks/create_cloud.yml
... omitted output ...
- name: Calculate all subnets within the prefix
  ansible.builtin.set_fact:
    subnets: "{{ query('calculate_subnets', prefix, 26) }}" (1)
- name: Create Cloud IP Prefixes for subnets within Nautobot
  networktocode.nautobot.prefix:
    url: "{{ nautobot_url }}"
    token: "{{ nautobot_token }}"
    state: "{{ nautobot_state }}"
    prefix: "{{ item.0 }}"
    status: "Active"
```

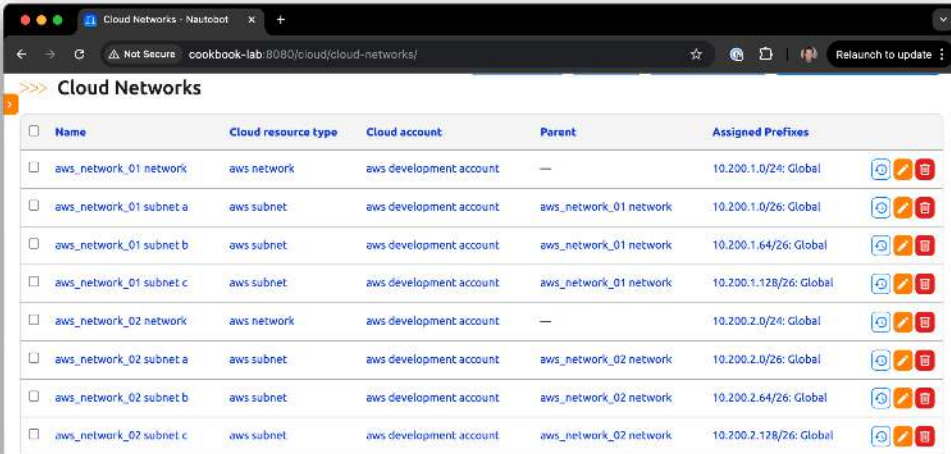
```

loop: "{{{ subnets | zip(['a', 'b', 'c']) | list }}" (2)
register: subnet_prefixes
- name: Create a cloud subnets to prefix assignment
  networktocloud.nautobot.cloud_network_prefix_assignment:
    url: "{{{ nautobot_url }}"
    token: "{{{ nautobot_token }}"
    state: "{{{ nautobot_state }}"
    cloud_network: "{{{ inventory_hostname }}" subnet {{{ item.item[1]
                  }}"
    cloud_prefix: "{{{ item.prefix }}"
loop: "{{{ subnet_prefixes['results'] }}" (3)

```

The numbered references from the preceding code are explained as follows:

- (1) First, using the `calculate_subnets` lookup task, generate subnets with length 26. This means that four prefixes are calculated for each /24. Use `query` instead of `lookup` to obtain a list object instead of a string.
 - (2) To establish a pairing of the subnets calculated with the subnets, use the `zip` function to combine the objects, discarding ones not paired.
 - (3) Finally, register the results of the prefix creation in Nautobot with the assignment of a subnet prefix and name (e.g., a, b, or c).
5. After the network and subnet prefixes' assignment, you get a final representation of the cloud scenario.



Name	Cloud resource type	Cloud account	Parent	Assigned Prefixes
aws_network_01 network	aws network	aws development account	—	10.200.1.0/24: Global
aws_network_01 subnet a	aws subnet	aws development account	aws_network_01 network	10.200.1.0/26: Global
aws_network_01 subnet b	aws subnet	aws development account	aws_network_01 network	10.200.1.64/26: Global
aws_network_01 subnet c	aws subnet	aws development account	aws_network_01 network	10.200.1.128/26: Global
aws_network_02 network	aws network	aws development account	—	10.200.2.0/24: Global
aws_network_02 subnet a	aws subnet	aws development account	aws_network_02 network	10.200.2.0/26: Global
aws_network_02 subnet b	aws subnet	aws development account	aws_network_02 network	10.200.2.64/26: Global
aws_network_02 subnet c	aws subnet	aws development account	aws_network_02 network	10.200.2.128/26: Global

This recipe completes the population of data in Nautobot. Next, we turn it around, and you will retrieve data from Nautobot to create an Ansible inventory dynamically.

Using Nautobot as an Ansible dynamic inventory

Throughout the examples and recipes in this book so far, we have relied on a static inventory file (typically named `hosts`) to define our inventory, which Ansible parses before executing playbooks. In this recipe, we'll explore a different approach: using a dynamic inventory. Instead of relying on a predefined static file, the inventory is generated dynamically at playbook execution time. In this example, all inventory data is maintained in Nautobot, which we'll use as our dynamic inventory source:

1. Create a new `dynamic_inventory` directory and a YAML file, `inventory_source.yml`, to define the dynamic inventory configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12# mkdir dynamic_
inventory && vi dynamic_inventory/inventory_source.yml
---
plugin: networktocode.nautobot.inventory (1)
api_endpoint: http://127.0.0.1:8080
token: 47a72c0b6b7a9c70c07501625a77fd398de59d1c
group_by: (2)
  - device_roles
  - locations
```

The numbered references from the preceding code are explained as follows:

- (1) The inventory instance is another type of Ansible plugin that creates the necessary data to create an Ansible inventory.
 - (2) The inventory plugin supports different options. The `group_by` plugin allows for the creation of groups in Ansible. More information is available at <https://galaxy.ansible.com/ui/repo/published/networktocode/nautobot/content/inventory/inventory/>.
2. Before creating and running a playbook, you can check that the dynamic inventory data can be retrieved:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory#
ansible-inventory --list -i inventory_source.yml
{
  "_meta": {
```

```

    "hostvars": {
      "dc1-leaf01": {
        "ansible_host": "172.20.1.35",
        "device_roles": [ "leaf_switch" ],
        "device_types": [ "7020SR" ],
        "locations": [ "dc1" ],
        "manufacturers": [ "arista" ],
... omitted output ...
      },
... omitted output ...
    "all": {
      "children": [
        "ungrouped",
        "device_roles_leaf_switch",
        "locations_dc1",
        "location_dc1",
        "location_dc2",
        "device_roles_spine_switch",
        "locations_dc2"
      ]
    },
    "device_roles_leaf_switch": {
      "hosts": [
        "dc1-leaf01",
        "dc1-leaf02",
        "dc2-leaf01",
        "dc2-leaf02"
      ]
    },
... omitted output ...

```

3. Create a new playbook, called `pb_create_report.yml`, to build a simple report of the data collected from the inventory:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory# vi
pb_create_report.yml
---
- name: Create Report from Nautobot Data
  hosts: all

```



```

gather_facts: no
connection: local
tasks:
  - name: Build Report
    ansible.builtin.blockinfile:
      block: |
        nautobot_data:
        {% for node in play_hosts %}
          - { node: {{ node }} , type: {{ hostvars[node]
              .device_types[0] }} , mgmt_ip: {{
              hostvars[node].ansible_host }} }
        {% endfor %}
      path: ./report_output.yaml
      create: yes
      delegate_to: localhost
      run_once: yes

```

4. Run the playbook and print to the screen to check the data extracted from Nautobot:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory#
ansible-playbook pb_create_report.yml -I inventory_source.yml &&
cat report_output.yaml
... output omitted ...
# BEGIN ANSIBLE MANAGED BLOCK
nautobot_data:
  - { node: dc1-leaf01 , type: 7020SR , mgmt_ip: 172.20.1.35 }
  - { node: dc1-leaf02 , type: 7020SR , mgmt_ip: 172.20.1.36 }
  - { node: dc2-leaf01 , type: 7020SR , mgmt_ip: 172.20.2.35 }
  - { node: dc2-leaf02 , type: 7020SR , mgmt_ip: 172.20.2.36 }
  - { node: dc1-spine01 , type: 7050CX3 , mgmt_ip: 172.20.1.41 }
  - { node: dc1-spine02 , type: 7050CX3 , mgmt_ip: 172.20.1.42 }
  - { node: dc2-spine01 , type: 7050CX3 , mgmt_ip: 172.20.2.41 }
  - { node: dc2-spine02 , type: 7050CX3 , mgmt_ip: 172.20.2.42 }
# END ANSIBLE MANAGED BLOCK

```

5. Another way to consume a Nautobot dynamic inventory is by using a **Nautobot GraphQL API** rather than a REST API. Let's create another file, `inventory_source_graphql.yml`, and configure it, filtering the data that you require; in this case, the device type and the primary IPv4 address. Also, install the `netutils` Python library as a prerequisite:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory#  
pip install netutils  
root@cookbook-lab:~/cookbook-lab/chapters/ch12# vi dynamic_  
inventory/inventory_source_graphql.yml  
---  
plugin: networktocode.nautobot.gql_inventory  
api_endpoint: http://127.0.0.1:8080  
token: 47a72c0b6b7a9c70c07501625a77fd398de59d1c  
query:  
  devices:  
    device_type: model  
    primary_ip4: host
```

6. Checking the Ansible inventory generated, notice how only the specified data is returned:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory#  
ansible-inventory --list -i inventory_source_graphql.yml  
{  
  "_meta": {  
    "hostvars": {  
      "dc1-leaf01": {  
        "ansible_host": "172.20.1.35",  
        "device_type": {  
          "model": "7020SR"  
        },  
        "name": "dc1-leaf01",  
        "primary_ip4": {  
          "host": "172.20.1.35"  
        }  
      },  
      ... output omitted ...  
    }  
  }
```

Finally, to close the story about the dynamic inventory, you will use the data to perform a task that also collects extra information from Nautobot.

Generating configurations using Nautobot

In this final recipe of the chapter, you will use the dynamic inventory from the previous section and enrich it with more data from Nautobot to render configuration snippets for all data center network devices:

1. Create a new Ansible playbook, `pb_build_config.yml`, with one task to define two variables: `graph_variables` and `query_string`. The first contains the name of the network device, and the second is a GraphQL query string that will retrieve the interface's name, description, IP address, and status to render the configuration:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory# vi pb_build_config.yml
---
- name: Create Interface Configuration Snippets
  hosts: all
  gather_facts: no
  connection: local
  vars:
    nautobot_url: http://127.0.0.1:8080
    nautobot_token: 47a72c0b6b7a9c70c07501625a77fd398de59d1c
  tasks:
    - name: Define the GraphQL query string to get interfaces
      ansible.builtin.set_fact:
        graph_variables:
          device_name: "{{ inventory_hostname }}"
        query_string: |
          query ($device_name:[String]) {
            devices (name: $device_name) {
              name
              interfaces {
                name
                description
                status {
                  name
                }
              }
              ip_addresses {
                address
              }
            }
          }
```

```
    }
  }
}
```

2. Update the same playbook, `pb_build_config.yml`, with the response from the Nautobot GraphQL API using the previous variables plus the Nautobot URL and token:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory# vi
pb_build_config.yml
... output omitted ...
- name: Obtain list of device interfaces from Nautobot
  ansible.builtin.set_fact:
    query_response: "{{ query(
      'networkcode.nautobot.lookup_graphql',
      query_string, graph_variables=graph_variables,
      url=nautobot_url, token=nautobot_token) }}"
```

3. Then, update the playbook with a rendering task:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory# vi
pb_build_config.yml
... output omitted ...
- name: "Render the template inline for CLI config"
  ansible.builtin.copy:
    dest: "./configs/{{ inventory_hostname }}_intf.txt"
    content: | {% for interface in query_response[0]
      .data.devices[0].interfaces %}
      interface {{ interface.name }}
      description {{ interface.description }}
      ip address {{ (interface.ip_addresses | first)['address']
      }}
      {% if interface.status.name == 'Active' %}no {% endif %}
      shutdown
    {% endfor %}
```

4. Finally, let's create a `configs` folder to store the artifacts, run the playbook, and check the output configuration artifacts:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_inventory#
mkdir configs && ansible-playbook pb_build_config.yml -i inventory_
source.yml
```

```
... output omitted ...
root@cookbook-lab:~/cookbook-lab/chapters/ch12/dynamic_
inventory# cat configs/dc1-spine01_intf.txt
interface Ethernet1
  description CORE | dc1-leaf01| Ethernet8
  ip address 172.30.1.0/31
  no shutdown
... other interfaces output omitted ...
```



Note

A GraphQL API is very useful for retrieving hierarchical information in a very effective way, only collecting the attributes required, not whole objects.

By combining the inventory data with follow-up queries, you can gather all the necessary information to generate any type of network configuration artifact—whether it’s a CLI-based configuration or a JSON payload for an API.

Summary

Ansible is a powerful automation engine, but it relies on accurate and well-structured data to drive its workflows. Integrating Ansible with an NSoT, such as Nautobot, not only provides a centralized data source but also enables the creation of sophisticated, dynamic workflows. This combination ensures consistency and scalability by leveraging real-time data to automate tasks such as configuration generation, validation, and deployment. Together, they form a robust framework for modern network automation.

Continuing with enriching the Ansible automation framework, in the next chapter, you will incorporate a new tool, AWX, that incorporates enterprise-grade features around Ansible, such as a web UI, a REST API, **role-based access control (RBAC)**, runtime environments, and job scheduling.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



13

Simplifying Automation with AWX

Ansible is more than just an automation engine; it's a powerful tool for streamlining and scaling tasks across multiple teams. Throughout the previous chapters of this book, you've been using Ansible through its **command-line interface (CLI)**. While this method works well for smaller environments, it's not ideal for meeting the demands of large-scale IT enterprises.

To address these needs, Ansible Engine can be leveraged through automation solutions that offer additional features, designed to manage automation at an enterprise level. These solutions include the following:

- **Graphical User Interface (GUI):** A visual dashboard to execute Ansible playbooks, monitor their status, and display statistics related to different objects within your infrastructure.
- **Role-Based Access Control (RBAC):** This feature provides fine-grained control over objects such as Ansible playbooks, inventories, and machine credentials. It allows you to define user roles and assign privileges based on job functions, enabling design teams to focus on developing playbooks while operations teams handle task delegation.
- **Inventory Management:** This allows you to define static or dynamic inventories and manage hosts and groups, following the folder structure you're already familiar with in Ansible.
- **Credential Management:** A central management system for sensitive credentials, such as passwords and SSH keys. All credentials are encrypted, ensuring that sensitive information is never exposed in plaintext, thus enhancing security. Integrations with other secrets providers such as HashiCorp Vault are also supported.

- **Centralized Logging:** A logging system that tracks all automation tasks run on worker nodes. This enables audit trails, making it possible to track who ran which playbooks on which nodes and review the status of those tasks.
- **RESTful API:** The API allows you to execute automation tasks, integrate with other orchestration and ticketing systems, and retrieve data accessible from the GUI, including inventory information. This simplifies integration in enterprise environments.

These capabilities are available in two forms:

- **Community-based: Ansible Web eXecutable (AWX)**, open source under the Apache 2.0 license (reference: <https://github.com/ansible/awx>)
- **Subscription-based: Ansible Tower**, part of the **Ansible Automation Platform (AAP)**, supported by *Red Hat* (more info at <https://www.redhat.com/es/technologies/management/ansible/>)

The primary distinction between the two is the additional enterprise features, stability, and support offered with the commercial version. While AWX is ideal for small-to-medium environments, it provides many of the same functionalities as Ansible Tower. In this chapter, you will focus on AWX, as most of the concepts are directly applicable to the commercial offering. If you want to give it a try, you can use this link to install it and get free development access: <https://developers.redhat.com/products/ansible/download>.

In short, the main recipes covered in this chapter are as follows:

- Installing AWX
- Managing users and teams in AWX
- Creating a network inventory in AWX
- Managing network credentials in AWX
- Creating projects in AWX
- Managing execution environments in AWX
- Creating templates in AWX
- Creating workflow templates in AWX
- Running automation tasks using the AWX API



Info

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch13.

Even though it's not the main goal of this chapter, you will take care of installing AWX in your development environment.

Installing AWX

Ansible AWX follows a modular architecture designed to manage and execute automation workflows efficiently. At its core, the web service provides a **graphical user interface (GUI)** and REST API (based on the Python Django framework, like Nautobot, explained in *Chapter 12*), enabling users to interact with AWX for job execution, inventory management, and workflow orchestration.

The following components work together to provide a scalable, centralized automation platform:

- The **task engine** handles the execution of automation jobs, distributing tasks to **execution environments**, which are containerized environments that include Ansible and its required dependencies
- **PostgreSQL database** is used to store information about users, inventories, job results, and configurations
- **Redis** serves as a message broker to manage *task queuing* and job execution status

AWX can be deployed in multiple different ways; however, the most convenient way is to deploy it using containers in a container runtime environment, such as Kubernetes. This offers enhanced resiliency for the AWX deployment as a whole. To simplify deploying AWX, a Kubernetes AWX operator has been the preferred way to deploy AWX since version 18.0.

However, setting up a production-ready Kubernetes cluster could be overwhelming, so we propose using a **local emulator** named Kind to run a local Kubernetes cluster using Docker containers as nodes. Kind is designed to test how to interact with Kubernetes, so please limit its usage to local development or Continuous Integration.

The next steps will get the AWX environment up and running. These commands are the ones used in the Ubuntu VM:

1. Install Kind in the VM using the recommended guide (<https://kind.sigs.k8s.io/docs/user/quick-start/>):

```
root@cookbook-lab:~# [ $(uname -m) = x86_64 ] && curl -Lo ./kind
https://kind.sigs.k8s.io/dl/v0.27.0/kind-linux-amd64
root@cookbook-lab:~# chmod +x ./kind
root@cookbook-lab:~# sudo mv ./kind /usr/local/bin/kind
```

💡 Quick tip: Enhance your coding experience with the AI Code Explainer and Quick Copy features. Open this book in the next-gen Packt Reader. Click the Copy button

(1) to quickly copy code into your coding environment, or click the Explain button

(2) to get the AI assistant to explain a block of code to you.



The next-gen Packt Reader is included for free with the purchase of this book. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



2. Install Kubectl, the Kubernetes CLI, to provide visibility of the Kubernetes resources, using the reference guideline (<https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/>):

```
root@cookbook-lab:~# curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
root@cookbook-lab:~# sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

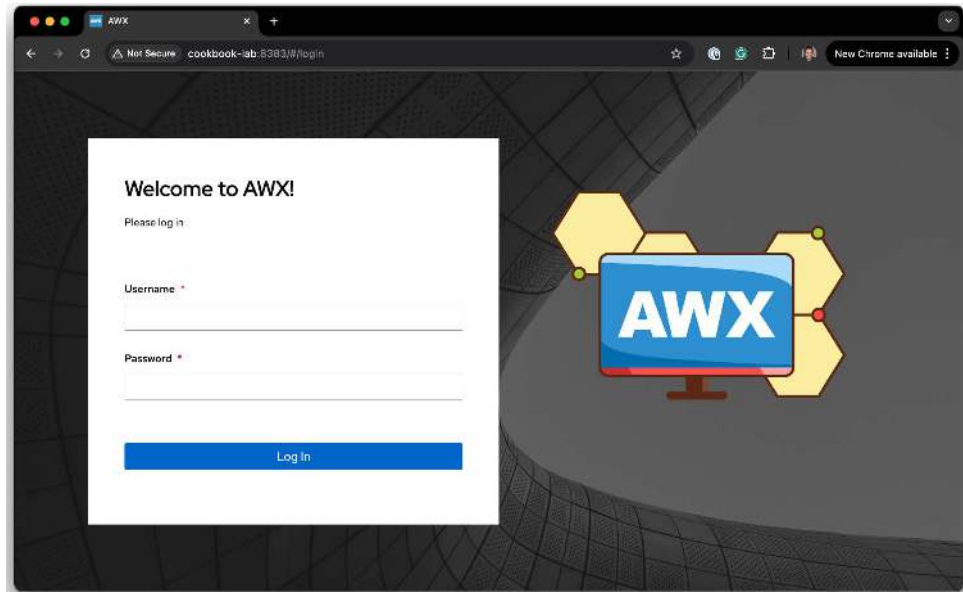
3. Install Helm, a package manager for Kubernetes that simplifies application deployment, management, and versioning using Helm charts. Installation details can be found at <https://helm.sh/docs/intro/install/>:

```
root@cookbook-lab:~# curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
root@cookbook-lab:~# chmod 700 get_helm.sh
root@cookbook-lab:~# ./get_helm.sh
```

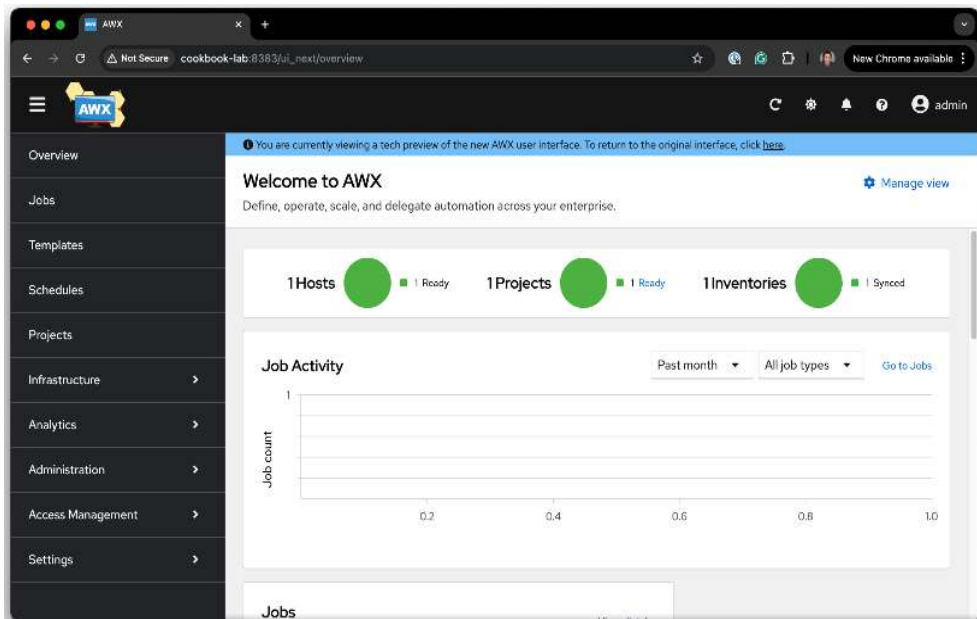
4. To simplify the cluster deployment, you have a *helper script* (https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/blob/main/chapters/ch13/awx_kind_quickstart/quick-start.sh) that sets up the AWX cluster using the AWX Operator embedded in a Helm chart that orchestrates all the Kubernetes actions. Notice the admin credentials that are provided at the end of the output:

```
root@cookbook-lab:~# curl -sSL https://raw.githubusercontent.com/network-automation-cookbook/cookbook-lab/main/chapters/ch13/awx_kind_quickstart/quick-start.sh | bash
Using cluster name: awx-cluster-demo
Using host port: 8383
Deleting existing Kind cluster 'awx-cluster-demo' (if any)...
Deleting cluster "awx-cluster-demo" ...
Deleted nodes: ["awx-cluster-demo-control-plane"]
Creating Kind cluster 'awx-cluster-demo'...
... omitted output ...
AWX Operator installed with Helm Chart version 3.0.0
Applying AWX instance manifest...
awx.awx.ansible.com/awx-demo created
Demo build complete.
Waiting for AWX admin password to be available...
AWX username: admin
AWX password: <password for admin>
```

5. After a few minutes (it takes time to apply all the database migrations), you will be able to access the console via the web at <http://<your-server-ip>:8383>. The following is the login portal you will get.



6. Finally, after logging in to the AWX GUI, you can see the main dashboard with the latest UI, along with all the options available for configuration on the left panel (you can swap between the old and new UI).



7. You can verify the AWX deployment, with the component that you expect, using the `kubectl` CLI. The `awx-operator` takes care of managing the deployment, and `web`, `task`, and `postgres` match the architecture that has just been explained.

```
root@cookbook-lab:~# kubectl get pods -n awx
```

NAME	READY	STATUS
awx-demo-migration-24.6.1-fn7r2	0/1	Completed
0	1h	
awx-demo-postgres-15-0	1/1	Running
0	1h	
awx-demo-task-66597794b4-bj22n	4/4	Running
0	1h	
awx-demo-web-f5fbd968b-gnskd	3/3	Running
0	1h	
awx-operator-controller-manager-799967c78c-n58s2	2/2	Running
0	1h	

Before going forward, to complete the environment, you will set up a simplified **Containerlab** tailored to interact directly with the Docker network set up by the Kind deployment (i.e., reusing the same bridge and network prefix). This is just a convenient adjustment to simplify the connectivity between the AWX platform and the network devices (which have been reduced to only the Arista devices, for simplicity).

```
root@cookbook-lab:~# labcli containerlab deploy ./containerlab/cookbook-awx-lab.yml
```

After AWX is up and running, you can start managing some of the functionalities provided.

Managing users and teams in AWX

One of the key features of AWX is its *robust RBAC*, implemented through various objects within the platform. These objects—primarily *organizations*, *users*, and *teams*—will play a crucial role in managing access and responsibilities in your AWX application.

AWX is designed to function as an enterprise-scale automation framework, enabling multiple teams within an organization to coexist and collaborate. Each team can manage its own devices, maintain its own playbooks, and handle its respective infrastructure. The concept of an **Organization** in AWX allows you to effectively separate and organize these teams and users. For instance, in a network management scenario, you might define a network organization to group all teams and users responsible for the network infrastructure.

Within an organization, users with varying roles and responsibilities will need different levels of access to the application. To streamline the assignment of appropriate roles, AWX provides **Teams**, which allow you to group users with similar privileges. For example, in the same network organization, you could define two teams: `Network_Design` and `Network_Operation`.

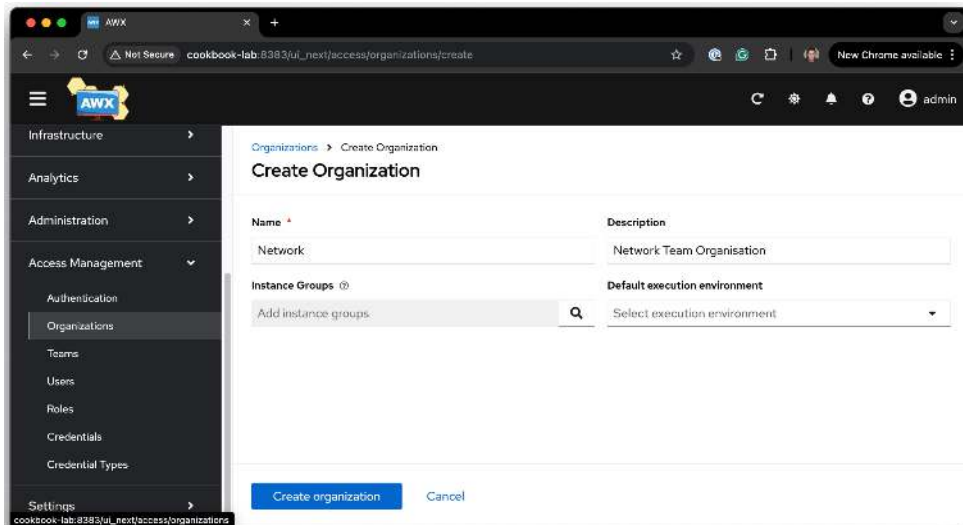
Here's how these teams could operate:

- The `Network_Design` team would be tasked with creating playbooks, setting up network inventories, and managing credentials to access devices
- The `Network_Operation` team would have permission to view the inventories and execute the playbooks developed by the design team

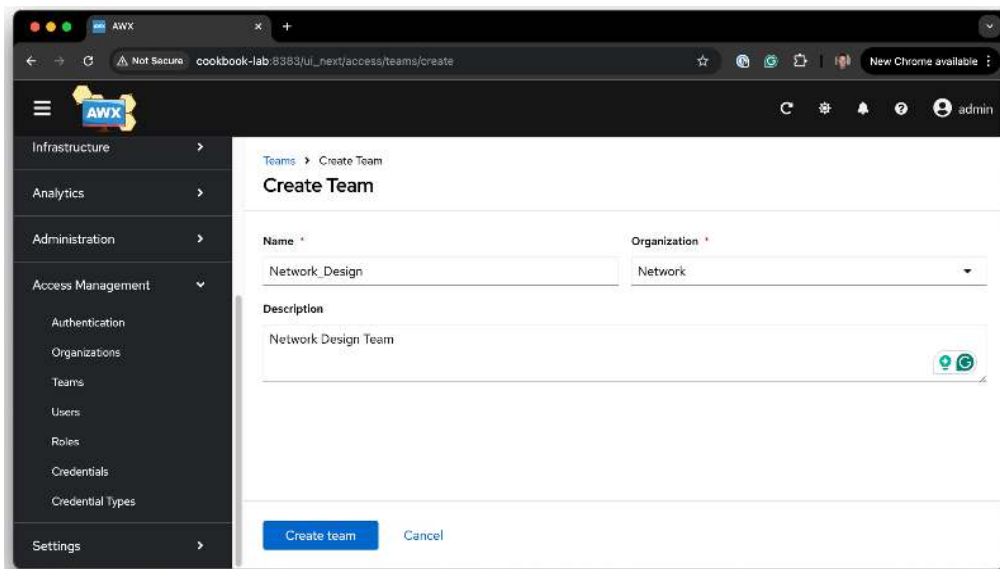
These constructs will work in unison to establish fine-grained RBAC, ensuring each user has the appropriate level of access to AWX resources. By leveraging these features, you can maintain clear boundaries, simplify administration, and ensure security across your automation workflows.

Let's start with the first AWX constructs:

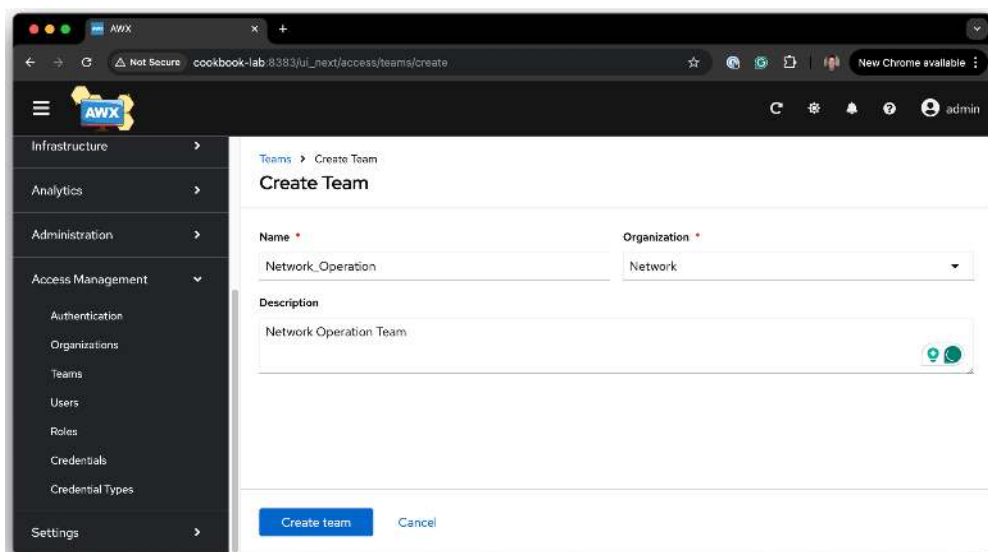
1. Create a new organization, `Network`, for all network teams—as shown in the following screenshot—by selecting **Access Management** and **Organizations** from the left panel and clicking the **Create organization** button.



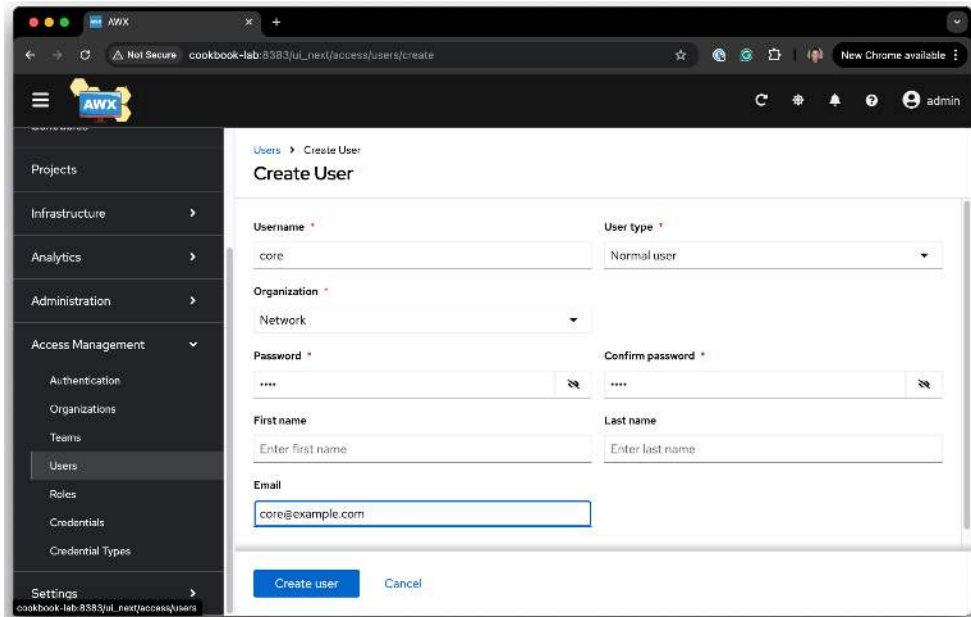
2. Create a new team within the **Network** organization for the **Design** team, by selecting the team from the left panel, and clicking on the **Create team** button.



3. Create another team within the **Network** organization for the **Operation** team, as shown in the following screenshot:

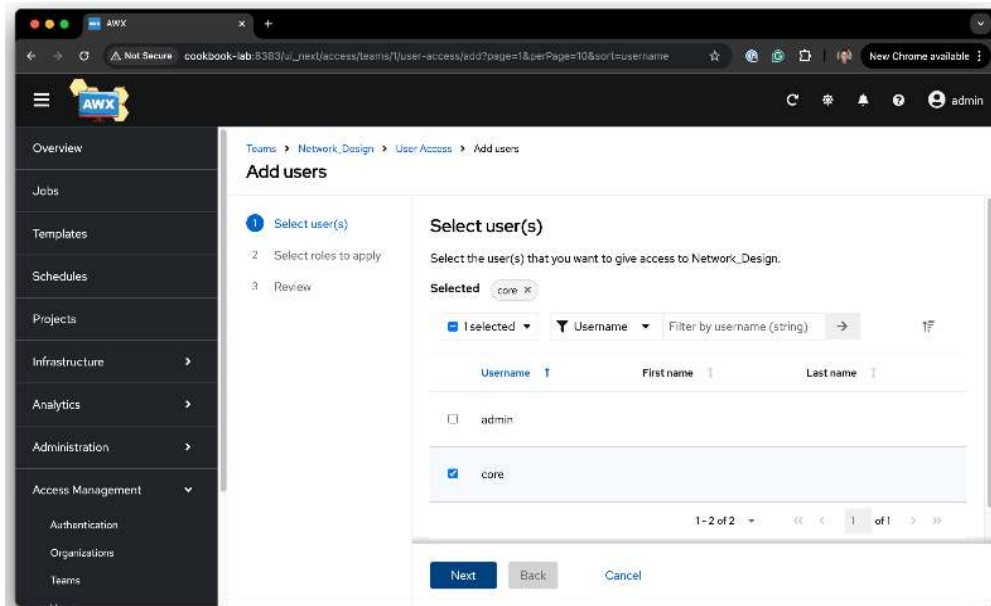


4. Create a core user (users provide a more granular access control) within the **Network** organization by selecting the **Users** button, as shown in the following screenshot:

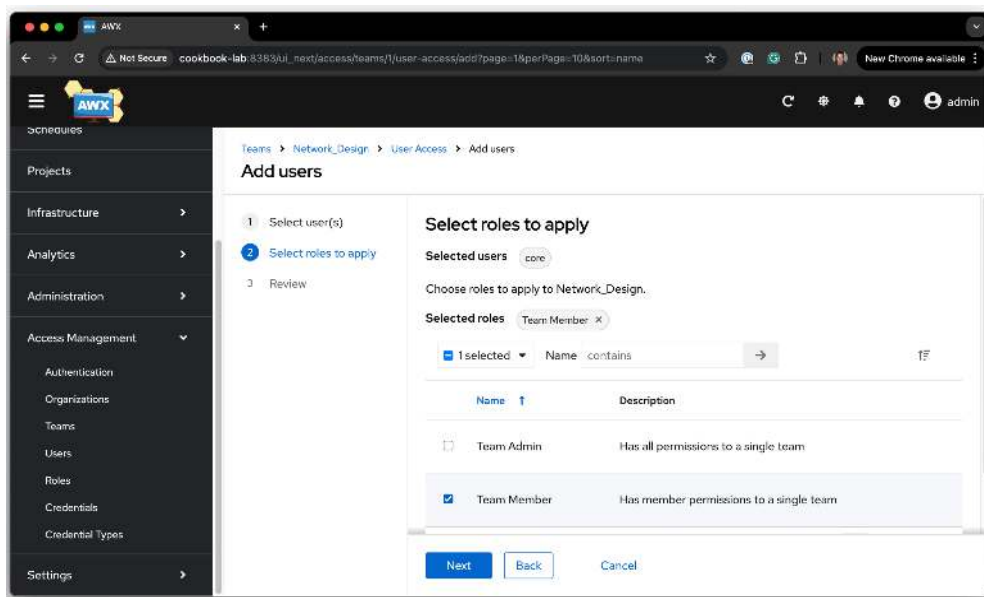


The screenshot shows the AWX web interface with the 'Create User' form. The left sidebar has 'Users' selected under 'Access Management'. The form fields are: Username (core), User type (Normal user), Organization (Network), Password (masked), Confirm password (masked), First name (Enter first name), Last name (Enter last name), and Email (core@example.com). There are 'Create user' and 'Cancel' buttons at the bottom.

5. Assign this new user to the **Network_Design** team – click the **Teams** tab from the left panel, and then select the **Network_Design** team. Click on **Users** and then add the core user to this team, as shown in the following screenshots:



The screenshot shows the AWX web interface with the 'Add users' form for the 'Network_Design' team. The left sidebar has 'Teams' selected under 'Access Management'. The form has three steps: 1. Select user(s), 2. Select roles to apply, and 3. Review. In the 'Select user(s)' step, the 'core' user is selected. The 'admin' user is also listed but not selected. There are 'Next', 'Back', and 'Cancel' buttons at the bottom.



6. Repeat the preceding steps to create a noc user and assign them to the **Network_Operation** team.

After this basic user is configured, you will tackle a common step in AWX, which is defining the inventory.

Creating a network inventory on AWX

In this recipe, you will build the **inventory** for your network in the AWX GUI. This is the exact step of defining an inventory file that you use with all your Ansible playbooks. The following code block shows the **static inventory file** that you normally define when you work with Ansible. You can define the same exact structure using inventories in AWX (in this case, simplifying to Arista and Juniper devices, even though you will only connect to Arista devices in this lab):

```
[arista_ceos]
ceos-01ansible_host=172.20.100.11
ceos-02ansible_host=172.20.100.12
ceos-03ansible_host=172.20.100.13
[juniper_crpd]
crpd-01ansible_host=172.20.100.21
crpd-02ansible_host=172.20.100.22
```

**Note**

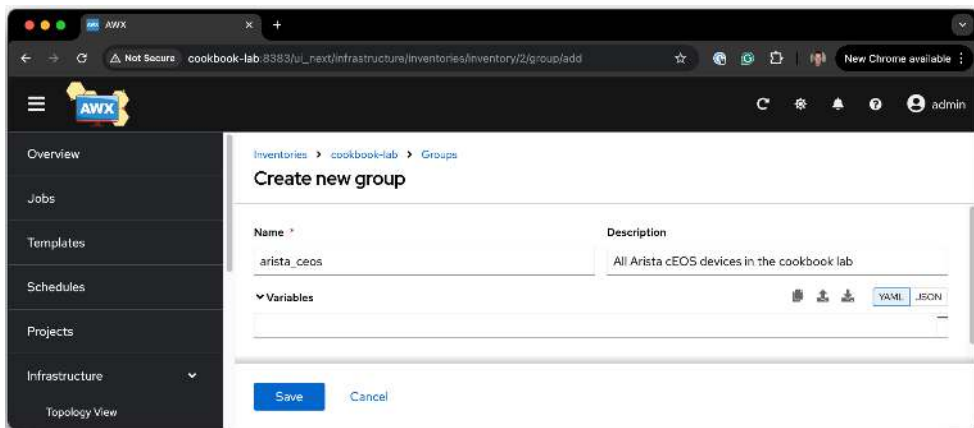
Notice that you are using a new prefix (172.20.0.0/16) that matches the Containerlab topology for this scenario, which is influenced by the Docker network used by the AWX Kind cluster.

Now, you will create your Ansible inventory in AWX:

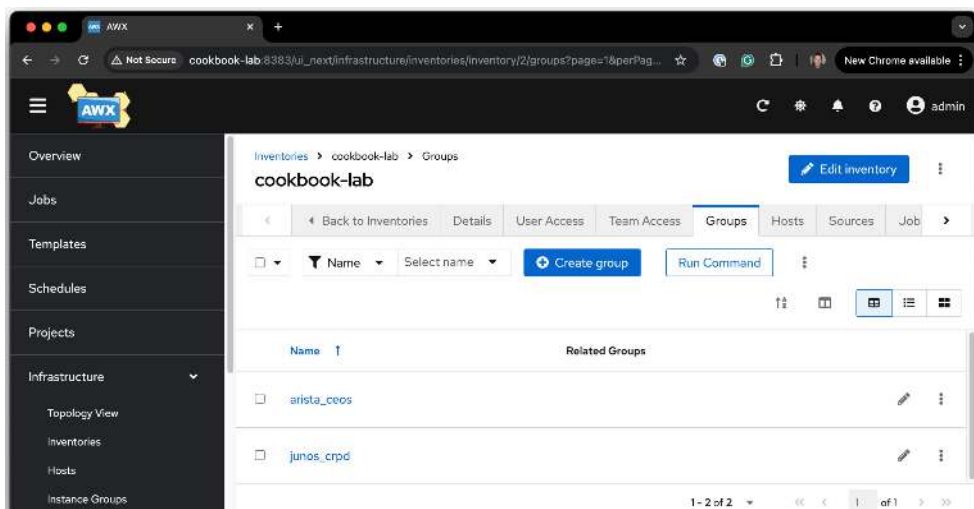
1. Create a new inventory called `cookbook-lab` by navigating to **Infrastructure | Inventories** on the left navigation bar and selecting **Create inventory** to create the new inventory, as shown in the following screenshot:

The screenshot shows the AWX web interface in a browser window. The left-hand navigation menu is expanded, showing the 'Infrastructure' section with sub-items like 'Topology View', 'Inventories', 'Hosts', 'Instance Groups', 'Instances', and 'Execution Environments'. The 'Inventories' item is selected. The main content area displays the 'Create Inventory' form. The form has a title 'Create Inventory' and a breadcrumb 'Inventories > Create Inventory'. Below the title, there are several sections: 'Name' with the value 'cookbook-lab', 'Description' with the value 'Cookbook Lab Network', 'Organization' with a dropdown menu showing 'Network', 'Instance Groups' with a button 'Add instance groups', 'Labels' with a dropdown menu showing 'Select or create labels', 'Variables' with a section header and a button 'Variables', and 'Options' with a checkbox 'Prevent instance group fallback'. At the bottom of the form, there are two buttons: 'Create inventory' (highlighted in blue) and 'Cancel'.

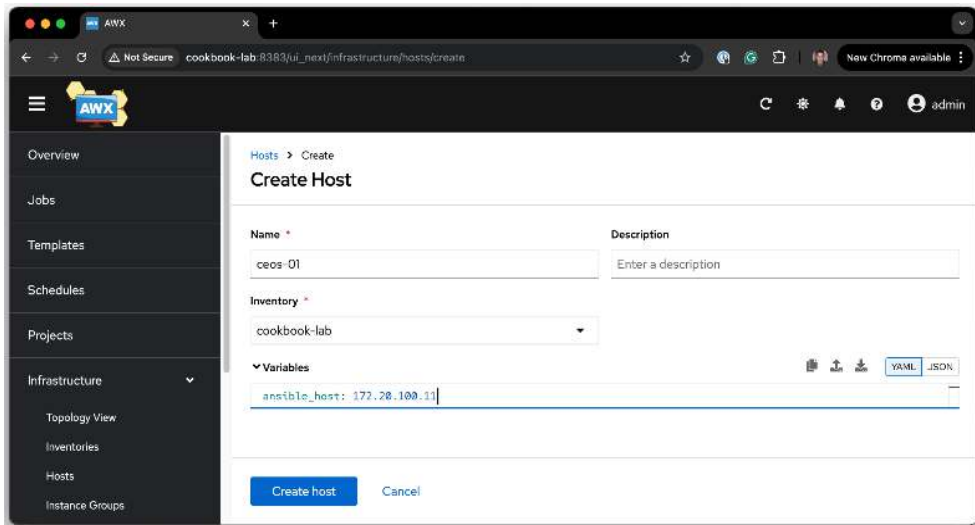
2. Within the inventory, enter the **Groups** tab to create a new group called `arista_ceos` (these groups are equivalent to the groups defined in previous Ansible inventories) as shown in the following screenshot:



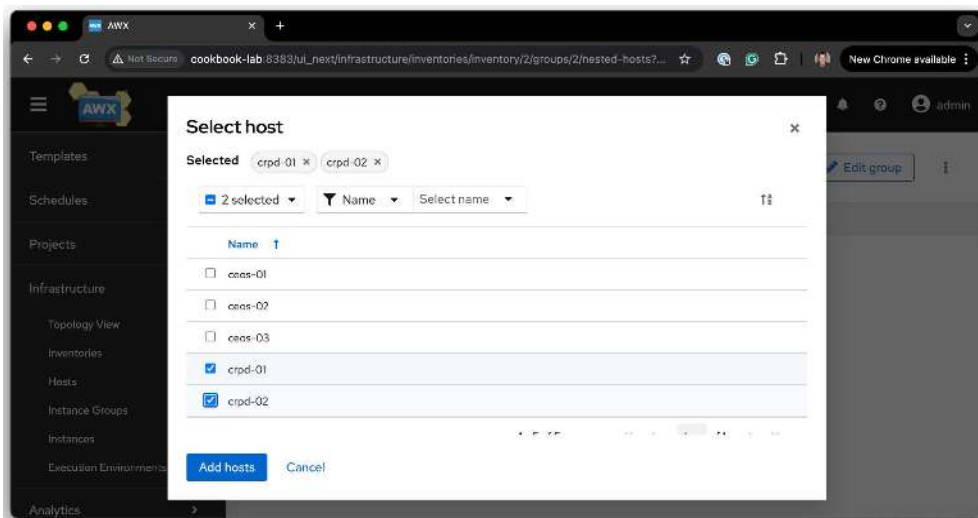
3. Create the `junos_crpd` group, using a similar approach. The final group structure under the `cookbook-lab` inventory should be similar to the one shown in the following screenshot:



4. In the **Hosts** tab of the inventory, create the `ceos-01` host device, and create the `ansible_host` variable under the **Variables** section, as shown in the following screenshot:

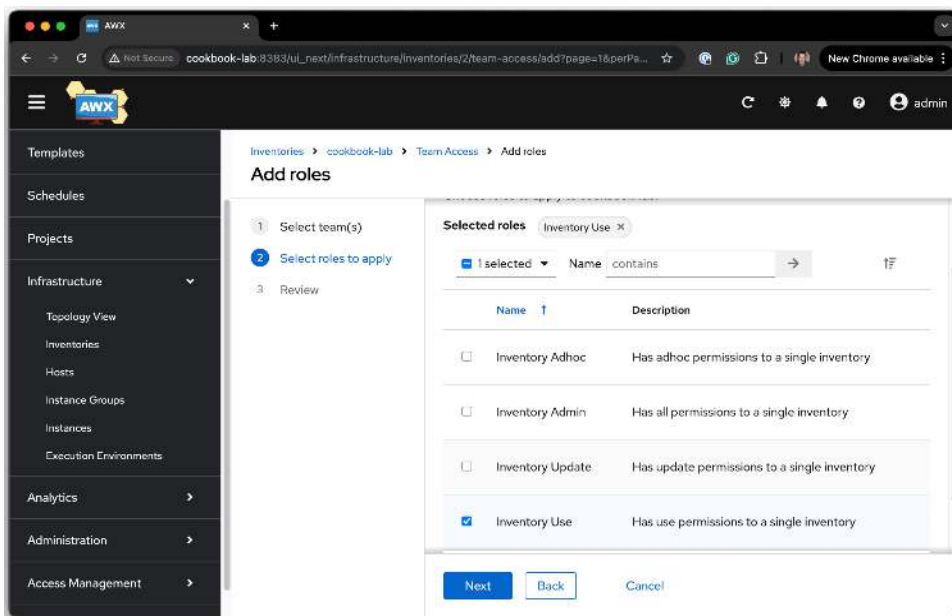
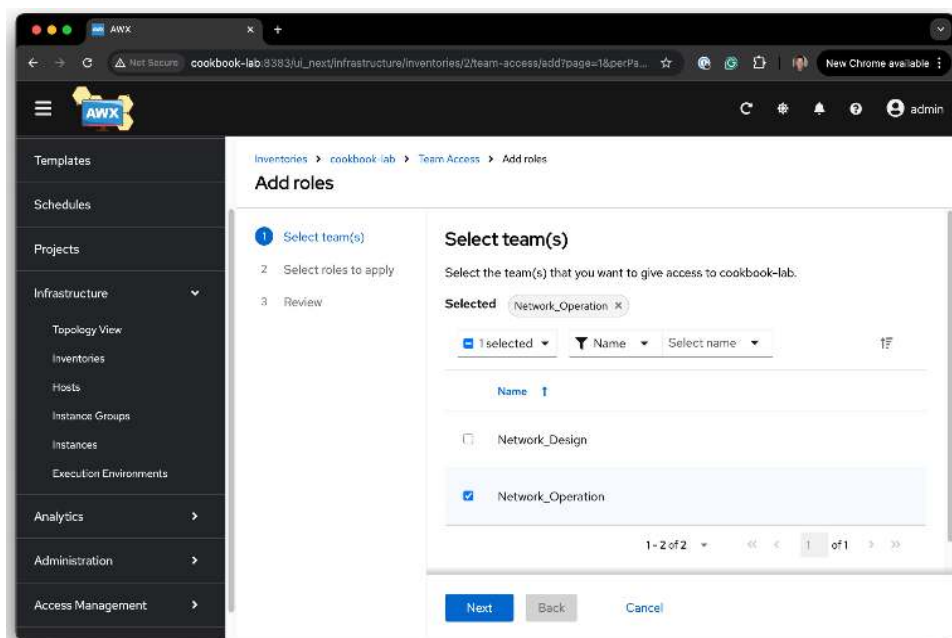


5. Repeat the same process to create the remaining hosts according to the desired inventory.
6. Now, you can group the hosts under different groups. Within the `cookbook-lab` inventory, go to the `junos-crpd` group that you created and add the corresponding hosts, and in the **Hosts** tag within the group, select **Add existing host**, to associate the hosts as shown in the following screenshot:

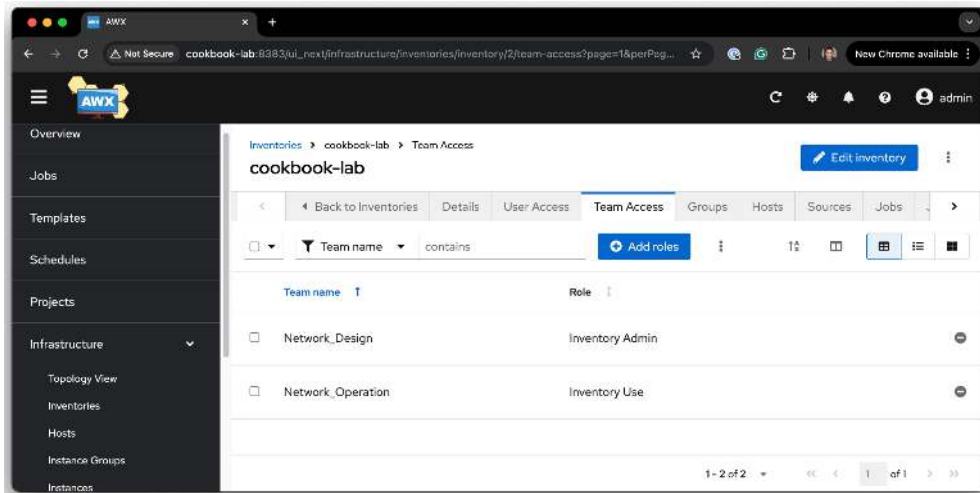


7. Repeat this for the `arista-eos` group with the Arista devices.

8. After creating the `cookbook-lab` inventory, you will grant read access to the **Network_Operation** group for this inventory in the **Team Access** tab and **Add roles**, as shown in the following screenshots:



9. Assign **Inventory Admin** permissions to the `Network_Design` team following the previous approach, and eventually, you will have the following in **Team Access**:



As you can see, the result is pretty much the same as the one you used in previous chapters, but you have used the RBAC component to narrow down the permissions of the inventory.



Note

Notice that you could manage permissions at the user level rather than **Team Access**. It depends on the level of granularity you require.

With the teams and users defined, you will create the credentials that could be used to connect to network devices or pull external repositories from Git.

Managing network credentials on AWX

For AWX to interact with our infrastructure and execute the necessary playbooks, it is essential to define the correct network *credentials* required to access our network devices. In this section, you will go through the process of creating these credentials, enabling AWX to log in to network devices and execute tasks on the managed network inventory.

Additionally, you will explore how AWX's robust RBAC framework simplifies the secure sharing of these sensitive credentials across different teams within the organization. By leveraging RBAC, you can ensure that each team has the appropriate level of access, fostering collaboration while maintaining security and control. Let's start with the credentials to access the network devices:

1. Under **Access Management** in the left navigation bar, select **Credentials** and **Create credential** to create the login credentials required to access the Arista network devices. You will use the **Machine** credential type since you will access the devices using connection modules such as `network_cli`, `NETCONF`, or `httpapi`. Specify the username and password used to log in to the devices in *Chapter 4* (admin/admin).

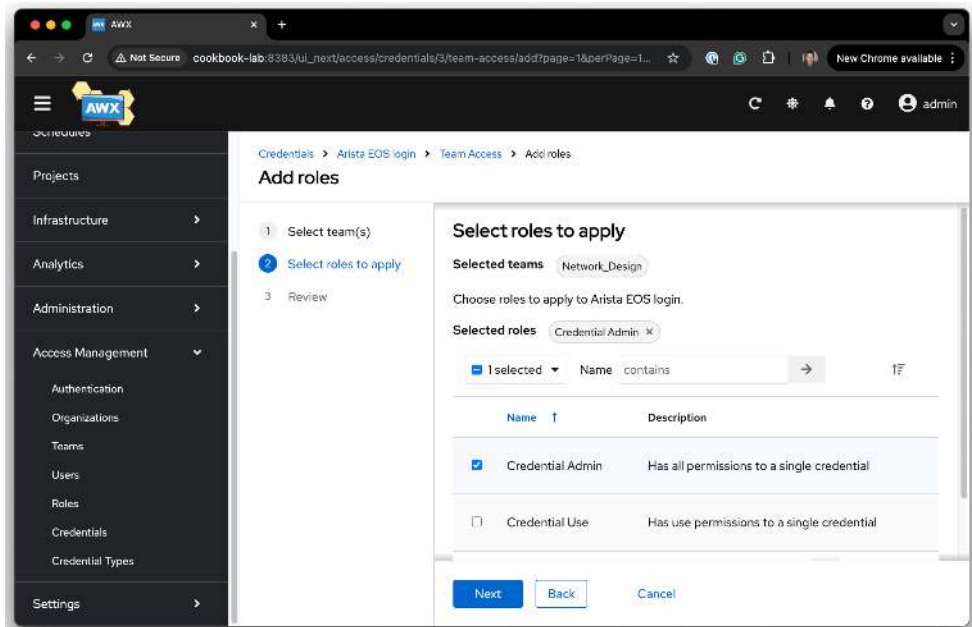
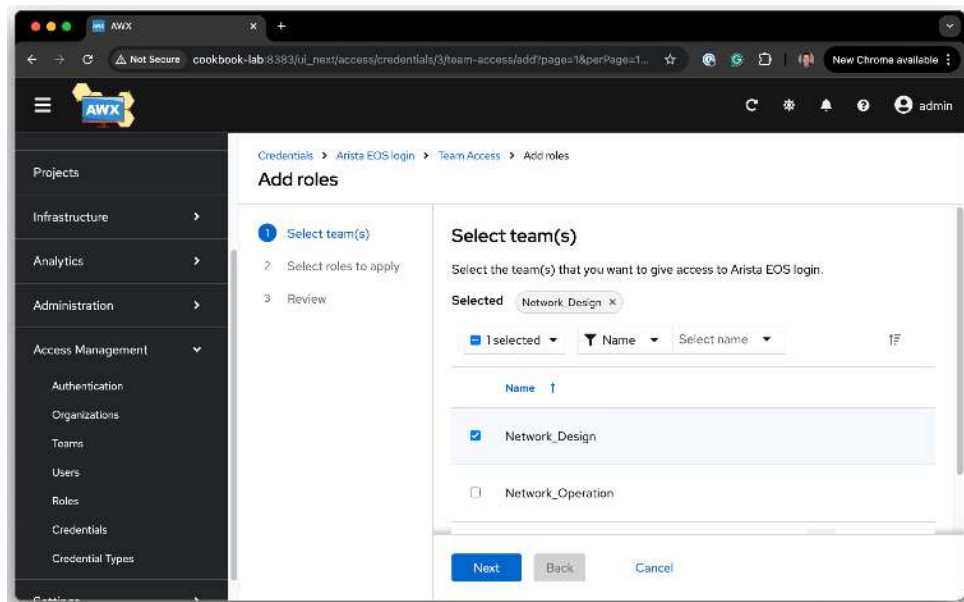
The screenshot shows the AWX web interface for creating a new credential. The left sidebar is expanded to 'Access Management', and 'Credentials' is selected. The 'Create Credential' form is displayed with the following fields:

- Name:** Arista EOS login
- Description:** Enter description
- Organization:** Network
- Credential type:** Machine
- Username:** admin
- Password:** (masked with dots)
- SSH Private Key:** (empty)
- Signed SSH Certificate:** (empty)

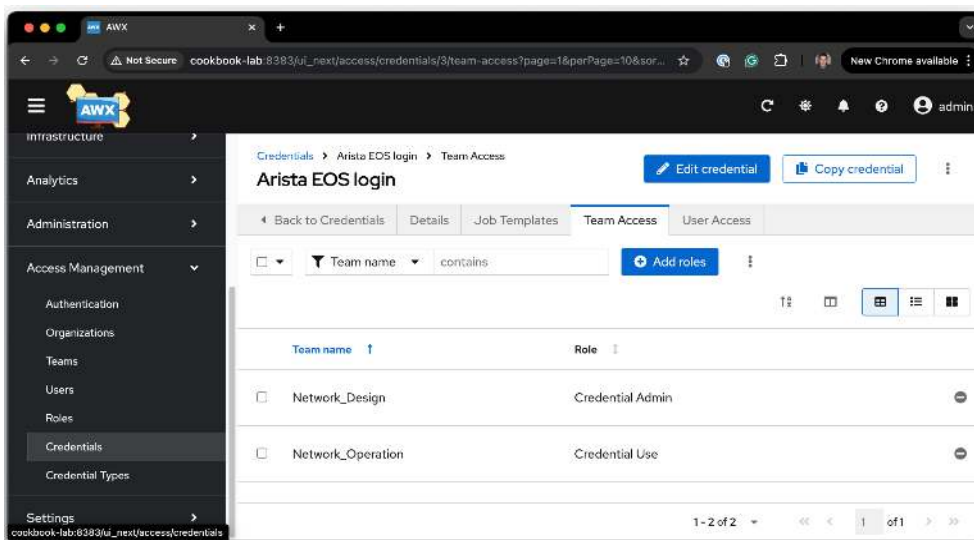
At the bottom of the form, there are two buttons: 'Create credential' and 'Cancel'.

2. Do the same for Juniper cRPD devices, with the credentials used in *Chapter 5* (root/clab123).

3. Then, update the permission for the credentials that you have created so that the Network_Design team is **Credential Admin**.



4. Repeat the process for the `Network_Operations` team with **Credential Use** access. The result should be like the one in the next screenshot:



When you entered the password in the AWX interface, it was encrypted and securely stored in the PostgreSQL database in an unreadable, encrypted format. This ensures enhanced security for password management within the AWX application.

AWX also provides a straightforward way to use sensitive information within an organization. Administrators or authorized users can create and edit credentials while assigning specific permissions to users or teams. These users can utilize the credentials without having the ability to view or modify them, ensuring that sensitive data remains protected.

This approach significantly simplifies password management compared to traditional methods such as managing credentials directly in Ansible or using `ansible-vault`. It streamlines collaboration while maintaining strict security controls over sensitive information.

AWX offers a variety of credential types to securely access different resources, including physical infrastructure, cloud providers, and **version control systems (VCS)**. For this use case, you utilize the **Machine** credential type, as it allows you to connect to our network infrastructure via SSH using a username and password. This ensures secure and streamlined access to devices while maintaining compatibility with the requirements of the underlying infrastructure.

After this initial setup, it's time to start creating automation projects in AWX.

Creating projects in AWX

AWX aims to streamline the execution of Ansible playbooks by offering a centralized automation environment, eliminating the need for isolated playbook executions that lack visibility. The most effective and widely adopted strategy to achieve this is managing Ansible playbooks within AWX projects, which are stored and tracked in Git version control.

This method provides a clear separation between playbook development and execution. Code development for Ansible playbooks is stored, versioned, and managed in Git, ensuring consistent tracking and collaboration. Meanwhile, AWX handles the execution of these playbooks, offering a seamless workflow that enhances organization, version control, and operational efficiency.

The process of organizing an Ansible project involves creating a dedicated folder to house all the necessary files and directories. This typically includes the `group_vars` and `host_vars` folders for defining variables, as well as the playbooks required for the project. These files and folders are then stored in a Git repository, hosted on a Git VCS such as GitHub or GitLab.

To enable AWX to utilize the playbooks you've developed, you begin by creating a new project within AWX. For this project, we select Git as the source control type and provide the URL of the Git repository containing the project. Additional details, such as the branch to use, are specified, and if the repository is private, the necessary credentials are supplied.

Once configured, AWX retrieves the content from the specified Git repository and downloads it to a local directory—by default, `/var/lib/awx/projects`. After the synchronization, all the repository's content will be available on the AWX node and ready to execute playbooks against the targeted network nodes.

The following steps create a new project in your environment, but this could be done anywhere and follow the same importing mechanism:

1. In any folder, create a new folder, `awx_sample_project`, containing the AWX project files. You will use simple playbooks in this chapter because the playbook is not the focus but the actual integration with AWX:

```
root@cookbook-lab:~/cookbook-lab# mkdir ~/awx_sample_project
```

2. Create `ansible.cfg` with basic configuration, specifying `remote_tmp` that is used to define where Ansible stores files during execution (making it compatible with the Kind environment):

```
root@cookbook-lab:~/awx_sample_project# vi ansible.cfg
[defaults]
```

```
remote_tmp = ~/tmp/ansible
gathering = explicit
host_key_checking = False
```

3. Create `group_vars` with DNS parameters and Arista-specific settings:

```
root@cookbook-lab:~/awx_sample_project# mkdir group_vars
root@cookbook-lab:~/awx_sample_project# vi group_vars/all.yml
---
global:
  dns:
    - 8.8.8.8
    - 8.8.4.4
root@cookbook-lab:~/awx_sample_project# vi group_vars/arista_ceos.
yml
---
ansible_network_os: "arista.eos.eos"
ansible_connection: "ansible.netcommon.network_cli"
ansible_become: yes
```

4. Then, create a simple playbook, `pb_basic_config.yml`, to render and configure the DNS data into the network devices:

```
root@cookbook-lab:~/awx_sample_project# vi pb_basic_config.yml
---
- name: "Configure DNS"
  hosts: arista_ceos
  tasks:
    - name: "Configure Basic Configuration on Arista Fabric"
      arista.eos.eos_config:
        lines:
          - "ip name-server {{ item }}"
        loop: "{{ global['dns'] }}"
```

5. On your GitHub account (or whatever Git provider you prefer), create a new repository (for this lab, it's easier to make it public to skip adding Git credentials) – for example, name it `awx_sample_project`.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Repository template

No template ▾

Start your repository with a template repository's contents.

Owner * chadell / **Repository name *** awx_sample_project

awx_sample_project is available.

Great repository names are short and memorable. Need inspiration? How about **solid-palm-tree** ?

Description (optional)

Repository to contain AWX playbooks

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

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

- Inside the `awx_sample_repo` project folder, initialize a Git repository and link it to the GitHub repository that we created in the previous step, as shown in the following code block (these commands are the default ones suggested by GitHub to initialize a repository):

```
root@cookbook-lab:~/awx_sample_project# git init
... omitted output ...
Initialized empty Git repository in /root/awx_sample_project/.git/
root@cookbook-lab:~/awx_sample_project# git add .
root@cookbook-lab:~/awx_sample_project# git commit -m "Initial commit"
[master (root-commit) b278df6] Initial commit
... omitted output ...
root@cookbook-lab:~/awx_sample_project# git branch -M main
root@cookbook-lab:~/awx_sample_project# git remote add origin git@github.com:chadell/awx_sample_project.git
```

```
root@cookbook-lab:~/awx_sample_project# git push -u origin main
... omitted output ...
```

7. Back to the AWX interface, create a new project with **Source Control Type** as **Git** and pointing to the previously created repository and the branch used, as shown in the following screenshot:

The screenshot shows the 'Create Project' form in the AWX interface. The form is titled 'Create Project' and is part of the 'Projects' section. It contains several fields: 'Name' (awx_sample_project), 'Description' (Enter description), 'Organization' (Network), 'Execution environment' (Select execution environment), 'Source Control Type' (Git), 'Content Signature Validation Credential' (Select credential), 'Source Control URL' (https://github.com/chadell/awx_sample_project), 'Source Control Branch/Tag/Commit' (main), 'Source Control Refspec' (empty), and 'Source Control Credential' (Select credential). There are 'Create project' and 'Cancel' buttons at the bottom.

Before moving forward, let's make sure that the **Last Job** status is **Success** to confirm that it's working as expected.

So far, your AWX environment has imported your playbooks and it's ready to run them. Defining the run environment is the goal of the AWX job templates that comes next.

Managing the execution environment in AWX

An AWX job template is a reusable definition that specifies how an Ansible automation task should be executed within AWX. It includes key parameters such as the inventory, project (source of playbooks), playbook to run, credentials, and other settings. One of the key settings is in which environment the playbook will run – the **AWX Execution Environment (EE)**.

AWX introduced EEs to replace Python virtual environments, used previously to isolate dependency management. Notice that this is how you have been running Ansible playbooks in the book so far. However, Python virtual environments complicate maintaining different Ansible versions and dependencies between projects and make it harder to reproduce environments across different systems.

EEs are packaged as containerized environments (this is why AWX works with container-based runtime environments), facilitating sharing and reproducing configurations across different environments.

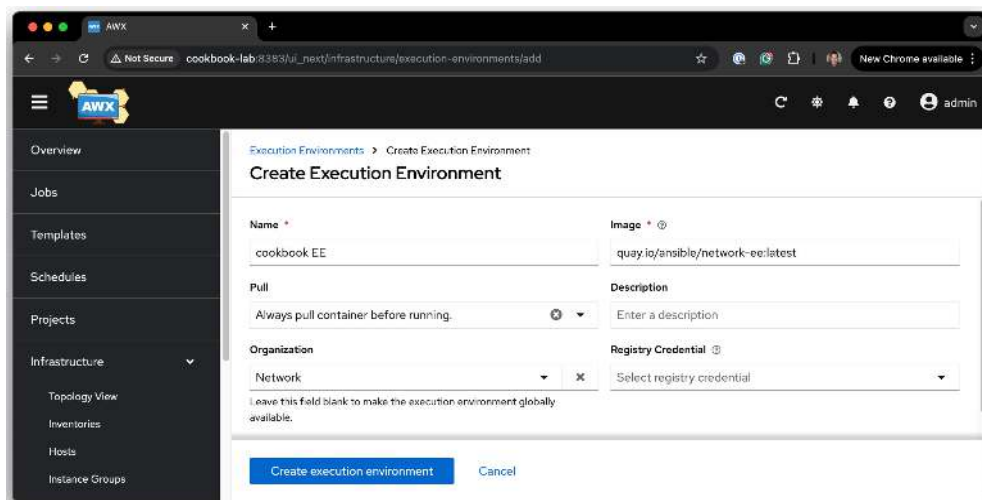
AWX EEs are container images that include the following:

- A specific Ansible version (the `ansible-core` engine)
- The required Ansible collections
- The necessary Python dependencies
- The necessary system-level packages
- Anything else required for execution

You are not expected to create a new AWX EE, but you will learn how to use them within AWX. In this recipe, you will create an EE object by reusing a public AWX EE container. When a job template requires an EE, AWX will pull the image from the container registry (i.e., the image has to be built in advance) and launch the Ansible playbook within the EE container instead of the AWX host system. AWX spins up a new Pod on demand for temporary use during the playbook execution, so in your Kubernetes environment, you will see these Pods get created and torn down.

The next step will set up a new EE, ready to handle the AWX **Job** executions:

1. In the **Infrastructure** section in the left navigation bar, choose **Execution Environments**, and create a new execution environment. The most important parameter is **Image**, which defines which container image to use.

A screenshot of the AWX web interface showing the 'Create Execution Environment' form. The left sidebar contains navigation links: Overview, Jobs, Templates, Schedules, Projects, Infrastructure (selected), Topology View, Inventories, Hosts, and Instance Groups. The main form area has the title 'Create Execution Environment' and the following fields: 'Name' (text input with 'cookbook EE'), 'Image' (text input with 'quay.io/ansible/network-ee:latest'), 'Pull' (dropdown menu with 'Always pull container before running.'), 'Description' (text area with placeholder 'Enter a description'), 'Organization' (dropdown menu with 'Network'), and 'Registry Credential' (dropdown menu with 'Select registry credential'). A note at the bottom of the form states: 'Leave this field blank to make the execution environment globally available.' At the bottom of the form are two buttons: 'Create execution environment' and 'Cancel'.

The `quay.io/ansible/network-ee:latest` image is a Docker image built by the Ansible team from the <https://github.com/ansible/network-ee> repository to include several popular Ansible collections for network management. For example, in `tools/requirements.yaml`, you can find the list of included collections. In this example, you will require the `arista.eos` collection.



Note

In case you need to create a custom EE, we recommend you check out this blog, which explains all the steps: <https://networktocode.com/blog/ansible-builder-runner-ee/>.

With the AWX EE defined, you have all the components to create an AWX job template.

Creating templates in AWX

In this recipe, you will combine all the components you previously configured to execute your playbooks in AWX. AWX uses job templates to create a standardized runtime environment (with different parameters explained shortly), allowing you to run your Ansible playbooks directly from AWX in a customizable and reproducible way.

You will start by creating a job template and assigning it a clear and descriptive name. Then, you will define the necessary parameters to set up the environment for playbook execution:

- Select the inventory against which the playbooks will run
- Provide all the required credentials—whether one or multiple—to ensure secure connections
- Link the project that contains the playbooks you intend to use
- Choose the specific playbook from the project to execute
- Select the EE where the playbook will run

By following these steps, you will have a consistent and reusable setup, enabling efficient and streamlined playbook execution through AWX. The idea is to bring together all the necessary customizations to facilitate rerunning the playbook multiple times under specific circumstances.

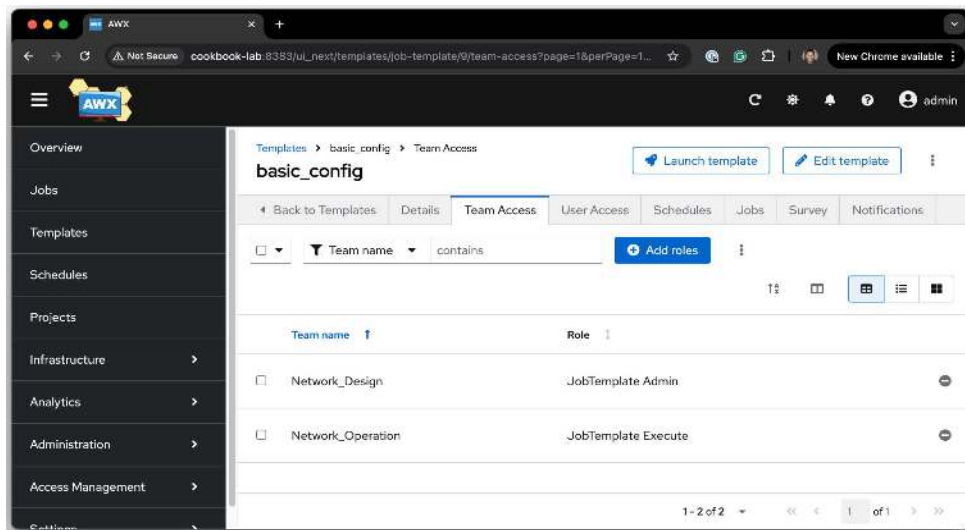
1. Under the **Templates** section in the left navbar, create a new template in AWX called `basic_config`, and assign to it the inventory and credentials that you created. You will use the `awx_sample_project` directory, as shown in the following screenshot:

The screenshot shows the 'Create Job Template' form in the AWX web interface. The left sidebar contains navigation links: Overview, Jobs, Templates, Schedules, Projects, Infrastructure (expanded), Topology View, Inventories, Hosts, Instance Groups, Instances, Execution Environments, Analytics, and Administration. The main content area is titled 'Create Job Template' and contains the following fields:

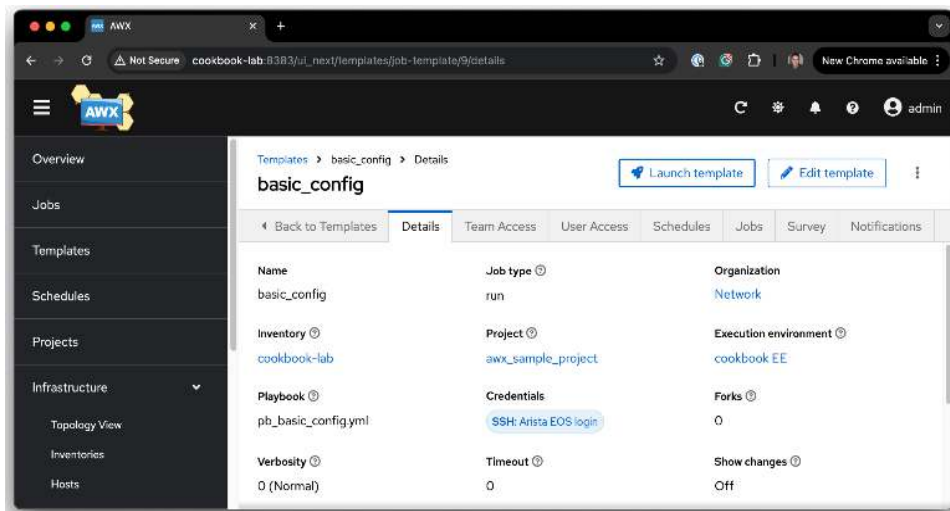
- Name ***: basic_config
- Description**: Add a description for this job template
- Job type ***: Run (with a 'Prompt on launch' checkbox)
- Inventory ***: cookbook-lab (with a 'Prompt on launch' checkbox)
- Project ***: awx_sample_project (with a 'Prompt on launch' checkbox)
- Playbook ***: pb_basic_config.yml (with a 'Prompt on launch' checkbox)
- Execution Environment**: cookbook EE (with a 'Prompt on launch' checkbox)
- Credentials**: Aista EDS login (with a 'Prompt on launch' checkbox)
- Labels**: Select or create labels (with a 'Prompt on launch' checkbox)
- Forks**: 0 (with a 'Prompt on launch' checkbox)
- Limit**: Add a limit to reduce number of hosts. (with a 'Prompt on launch' checkbox)
- Verbosity**: 0 (with a 'Prompt on launch' checkbox)

At the bottom of the form are two buttons: 'Create job template' and 'Cancel'.

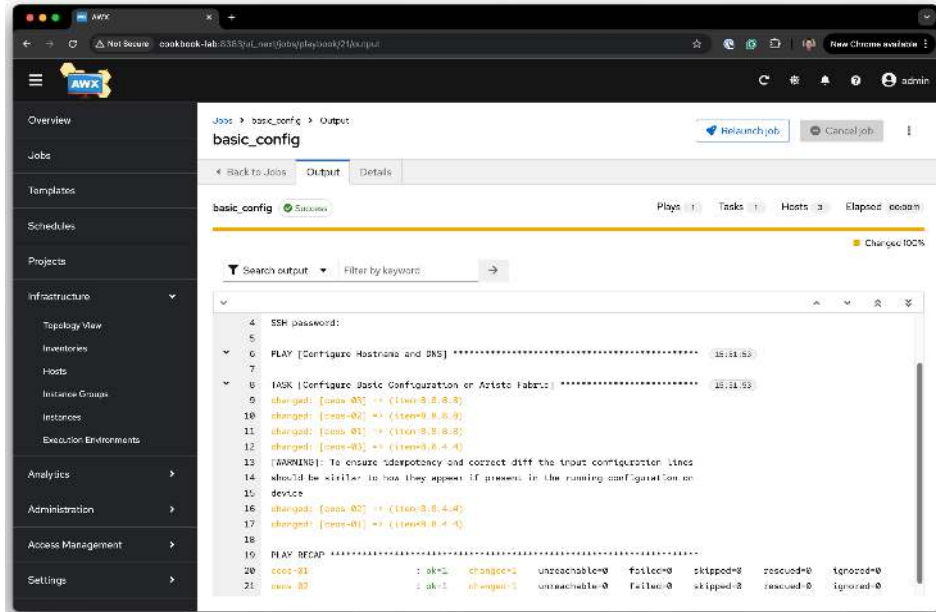
- To be able to execute and manage this template, you have to edit the created template and add team access. The `Network_Design` team will have the `ADMIN` role and also the `EXECUTE` role, as will the `Network_Operations` team, as shown in the following screenshot:



- Once the template is ready, you can launch a job with **Launch template**. This opens a new window that contains the output of the playbook execution:



- The job output looks familiar to you because it's exactly what the Ansible playbook would report on the screen. In this case, you can see how the basic configuration settings have been applied (i.e., changed) to the three Arista devices, as you limited with the hosts playbook definition.



- To double-check the change manually, you can log into one of the Arista devices and check the new name-server config:

```
root@cookbook-lab:~# ssh admin@172.20.100.11
(admin@172.20.100.11) Password:
Last login: Sat Mar  8 06:18:49 2025 from 172.20.0.2
ceos-01>en
ceos-01#show running-config | include ip
ip name-server vrf default 8.8.4.4
ip name-server vrf default 8.8.8.8
```

- To validate the same results within AWX, let's create a new playbook that checks the status. Remember that, to incorporate it into the AWX context, you have to resync the project. Until then, the new playbook won't be available to be assigned to a job template.

```
root@cookbook-lab:~/awx_sample_project# vi pb_validate_basic.yml
---
- name: "Collect and Validate Arista DC Fabric Facts"
```

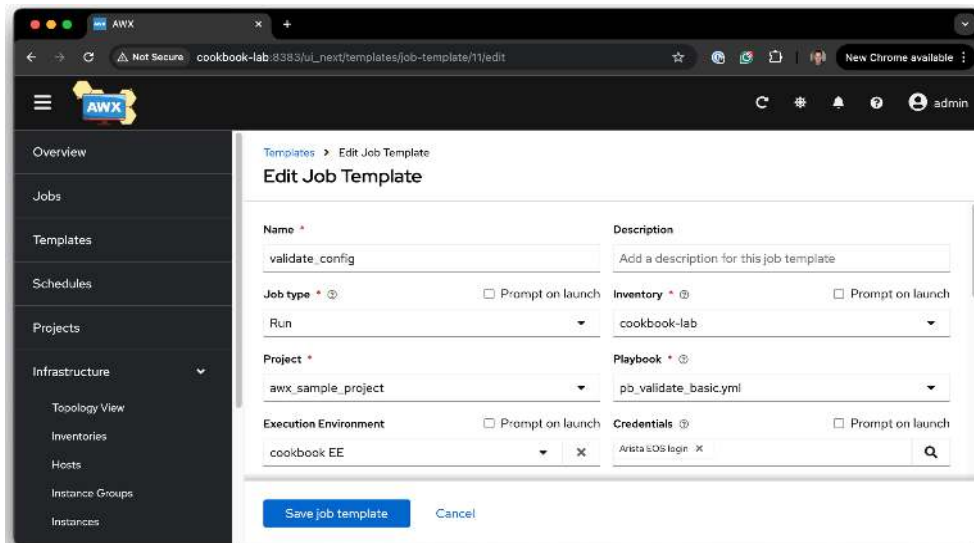
```

hosts: "arista_ceos"
tasks:
  - name: "Collect Arista Device Facts"
    arista.eos.eos_facts:
      gather_subset: all
  - name: "Extract configured DNS servers"
    ansible.builtin.set_fact:
      configured_dns: "{{ ansible_net_config | regex_findall(
        'ip name-server vrf default (.+)') }}" (1)
  - name: "Validate DNS Configuration"
    ansible.builtin.fail:
      msg: "Configured DNS servers {{ configured_dns }} do not
        match expected {{ global['dns'] }}"
      when: configured_dns | sort != global['dns'] | sort (2)

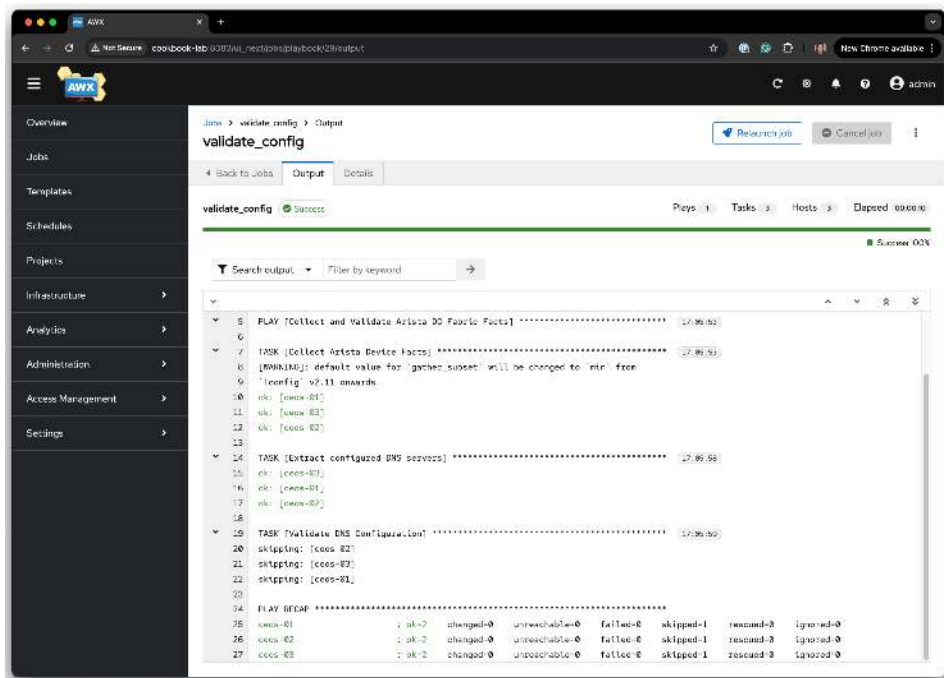
```

The numbered references in the preceding code snippet are explained as follows:

- (1) From the gathered `ansible_net_config` (the running configuration), you extract the IP address list of the name servers configured.
 - (2) You check that the list of DNS servers matches the list defined in the playbook variables.
7. Then, similar to the previous steps, you create a new job template, `validate_config`, which will check it from AWX.



- Finally, you can run this new `validate_config` job to ensure the configuration is in sync with the expected state.



With job templates, AWX offers an effective and reproducible experience to run Ansible playbooks, but the most important feature is the chaining of jobs to create **Workflows**.

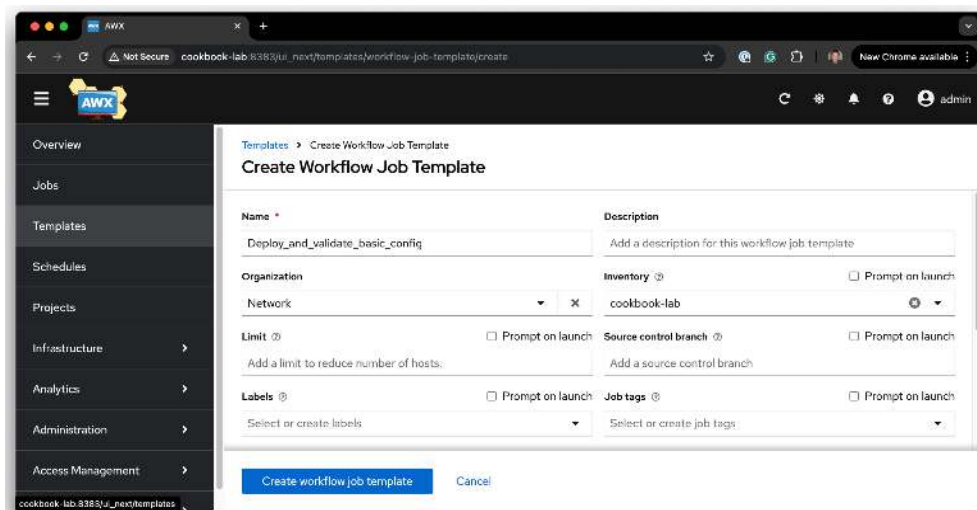
Creating workflow templates in AWX

AWX **workflows** provide a structured approach to orchestrating complex automation processes that require stitching multiple jobs together. This chaining supports conditional execution paths, so the execution can dynamically adjust to job success or failure. This allows you to take into account not only the golden execution path but also to mitigate related issues.

Within the network automation architecture we introduced in *Chapter 1*, AWX was considered an **Orchestrator**. Other features related to the orchestration realm are *scheduling*, *parallel execution*, and *triggering*.

In this recipe, you will create a workflow template on AWX to run multiple playbooks to achieve a global goal: configure and validate the operational status.

- From the **Templates** tab, choose **Create Workflow Job Template**, as shown in the following screenshot:

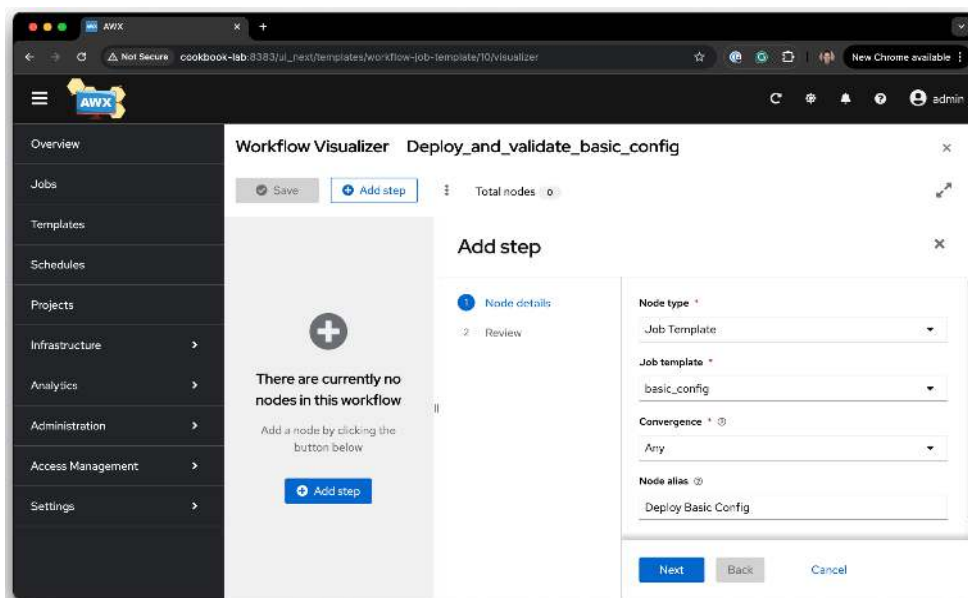


The screenshot shows the 'Create Workflow Job Template' form in the AWX web interface. The left sidebar contains navigation links: Overview, Jobs, Templates, Schedules, Projects, Infrastructure, Analytics, Administration, and Access Management. The main form area has the following fields:

- Name ***: Deploy_and_validate_basic_config
- Description**: Add a description for this workflow job template
- Organization**: Network
- Inventory**: cookbook-lab
- Limit**: Add a limit to reduce number of hosts.
- Source control branch**: Add a source control branch
- Labels**: Select or create labels
- Job tags**: Select or create job tags

There are checkboxes for 'Prompt on launch' for Limit, Source control branch, Labels, and Job tags. At the bottom, there are 'Create workflow job template' and 'Cancel' buttons.

2. Using **Workflow Visualizer**, create the workflow outline starting with **Add step**. A Step can be a job template, another workflow job template (i.e., composability), a manual approval, a project sync, an inventory sync, or a management job. In this case, select **Job Template**, and start with a `basic_config` one.

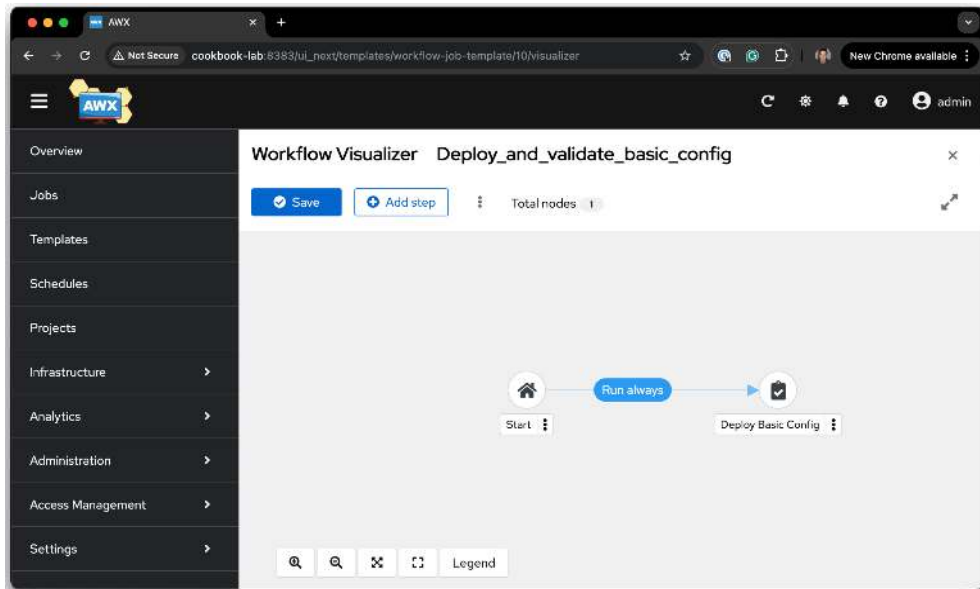


The screenshot shows the 'Workflow Visualizer' interface in the AWX web interface. The left sidebar is the same as the previous screenshot. The main area is titled 'Workflow Visualizer Deploy_and_validate_basic_config'. It has a 'Save' button and an 'Add step' button. Below the 'Add step' button, there is a message: 'There are currently no nodes in this workflow. Add a node by clicking the button below.' Below this message is an 'Add step' button. On the right, there is a 'Node details' panel with the following fields:

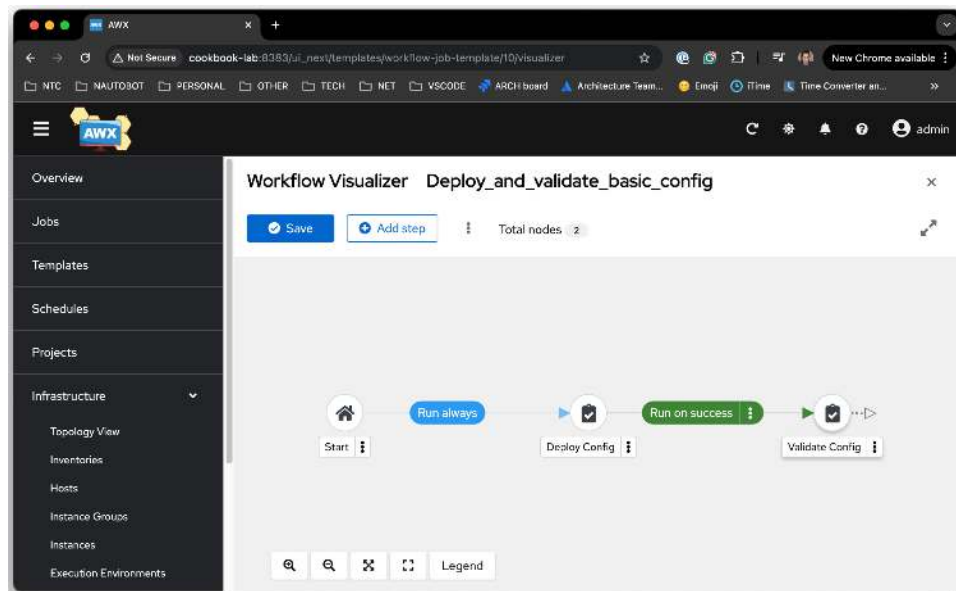
- Node type ***: Job Template
- Job template ***: basic_config
- Convergence ***: Any
- Node alias**: Deploy Basic Config

At the bottom of the 'Node details' panel, there are 'Next', 'Back', and 'Cancel' buttons.

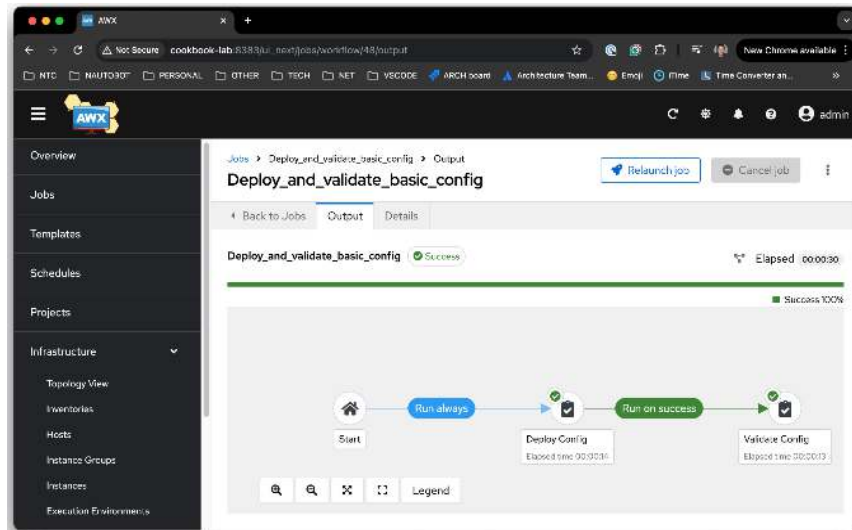
3. By default, the first step is added with the **Run always** logic.



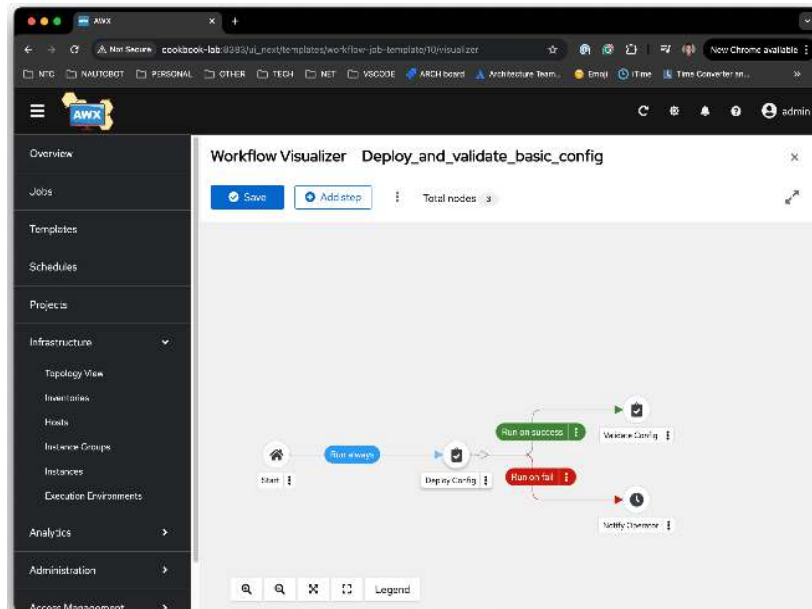
4. Then, add a second step to validate the configuration step. In this case, it only makes sense to validate the config if the configuration step executed successfully, so you can select the **Run on success** logic.



5. You can run it and see how the jobs are being executed, following the flow you defined. You will see how the tasks are progressing, and finally, you can click on the step to get the execution details.



6. Also, to control the failure path, you can add a manual approval step (to check operator awareness) when the deploy configuration step fails using the **Run on fail** logic (i.e., what to do when the task output is a failure).



The orchestration capability is a game-changer in network automation, offering a modular and reusable approach to managing automation tasks. It enables the composition of complex workflows that accurately reflect real-world network operations, streamlining processes such as provisioning, configuration management, and fault remediation. By leveraging orchestration, organizations can improve efficiency, consistency, and scalability in their network automation strategies.

Lastly, because no tool works alone in a network automation solution, we will present the AWX API, which allows for easy integration by third-party systems.

Running automation tasks using the AWX API

This recipe will demonstrate how to initiate jobs on AWX using its powerful **REST API**. AWX provides a comprehensive API that allows users to interact with the system, query objects, execute automation tasks, and manage resources efficiently. Through this API, you can trigger job templates and workflow templates, retrieve the status of running jobs, and access all available resources configured within the AWX interface.

AWX's REST API enables seamless retrieval and inspection of all objects and components within the system. We can use it to launch automation tasks and monitor their execution status. In this recipe, you will explore various ways to interact with the AWX API using the **cURL command-line tool**, but you could use other tools such as Postman or programming languages such as Python or Go to build more sophisticated integrations. Throughout the examples, you will utilize the **jq Linux utility**, a lightweight command-line JSON processor to filter, parse, and manipulate responses for better readability.

1. Start by exploring the AWX API by listing all the resources available through this API, as shown in the following code snippet. As you can see, every endpoint corresponds with one of the pages you have explored in the console in this chapter:

```
$ curl -X GET http://cookbook-lab:8383/api/v2/ -s | jq
{
  "ping": "/api/v2/ping/",
  "users": "/api/v2/users/",
  "execution_environments": "/api/v2/execution_environments/",
  "projects": "/api/v2/projects/",
  "teams": "/api/v2/teams/",
  "credentials": "/api/v2/credentials/",
  "inventory": "/api/v2/inventories/",
  "job_templates": "/api/v2/job_templates/",
```

```
    "jobs": "/api/v2/jobs/",
    "workflow_job_templates": "/api/v2/workflow_job_templates/",
    "workflow_jobs": "/api/v2/workflow_jobs/",
    ... omitted output ...
}
```

2. Collect all the job templates configured on the AWX interface with the following REST API call (using your credentials, as this is an authenticated endpoint), and get the ID and name for each job template. Notice that only the job templates are listed, not the workflow ones. The IDs are relevant because you use them to target operations:

```
$ curl -X GET --user admin:<your password> http://cookbook-lab:8383/
api/v2/job_templates/ -s | jq '.results[] | {id,name}'
{
  "id": 9,
  "name": "basic_config"
}
{
  "id": 11,
  "name": "validate_config"
}
... omitted output ...
```

3. Launch an existing job template with the following REST API call. In this example, you are launching (with **POST** and the launch action) the job template with **ID = 11**, from the previous output. From the return value, you get the ID of the executed job:

```
$ curl -X POST --user admin:<your password> http://cookbook-
lab:8383/api/v2/job_templates/9/launch/ -s | jq '.id'
37
```

4. Get the status of the job launched from the preceding API call with the following call. The endpoint is now jobs, and **ID=37** is retrieved from the previous API call for launching the job template:

```
$ curl -X GET --user admin:<your password> http://cookbook-lab:8383/
api/v2/jobs/37/ -s | jq '.status'
"successful"
```

5. Now, instead of individual jobs, collect all workflow templates configured on the AWX interface with the following API call, and record the ID and name for each one:

```
$ curl -X GET --user admin:<your password> http://cookbook-lab:8383
/api/v2/workflow_job_templates/ -s | jq '.results[] | {id,name}'
{
  "id": 10,
  "name": "Deploy_and_validate_basic_config"
}
```

6. Finally, launch the workflow job template using the ID retrieved from the previous API call, and you will get the ID of the workflow execution, which you will be able to check as you did with the jobs endpoint – in this case, with `workflow_jobs`:

```
$ curl -X POST --user admin:<your password> http://cookbook-
lab:8383/api/v2/workflow_job_templates/10/launch/ -s | jq '.id'
39
```



Info

AWX API docs are available at https://ansible.readthedocs.io/projects/awx/en/latest/rest_api/.

Everything that you can do via the user interface is available in the REST API, so this brings full capacity to operate your AWX platform within a network automation platform.

Summary

This chapter introduced AWX, the open source automation platform that extends Ansible with enterprise-grade orchestration and management capabilities. While previous chapters focused on using Ansible Engine via the CLI, AWX provides a **graphical user interface (GUI)**, **role-based access control (RBAC)**, inventory and credential management, centralized logging, and a RESTful API to support large-scale automation.

The chapter covered key AWX functionalities, including a development installation, user and team management, network inventory creation, credential handling, project and execution environment setup, and automation templates. A dedicated section explored workflow templates, which allow for complex, multi-step automation processes. Additionally, it demonstrated how to use the AWX API to trigger and monitor automation tasks, ensuring seamless integration with enterprise IT ecosystems.

In the next chapter, you'll dive into Ansible EDA—Event-Driven Automation—and explore how to leverage Ansible to respond automatically to real-time events across your infrastructure.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



14

Event-Driven Automation with Ansible

As you must have noticed about the recipes in this book, so far, the focus has been on more traditional on-demand executions. Here, instead of relying on periodic checks or manual triggers, event-driven automation empowers networks to respond in real time to changes (also known as events). Network automation often attempts to solve mundane, repeatable tasks in which a high percentage are the result of an event. Wouldn't it be great to auto-trigger workflows without needing to get involved?

This chapter introduces the new event-driven architecture that Ansible supports and how it can create new network automation solutions. As network automation becomes more sophisticated and information systems improve with more programmable interfaces, event-driven automation offers powerful workflow capabilities that can be triggered based on notifications/events occurring; furthermore, event-driven automation offers the added optimization of asynchronous operations, which allows a program to initiate a task and then continue executing other code without waiting for that task to complete, enabling concurrent execution.

Event-Driven Ansible (EDA) is an Ansible framework that listens to source systems and creates events. These events are then processed by a logic engine before getting passed down to an execution engine that runs actions. These actions include running a playbook, running a job template (in AWX or AAP/Tower), and so on.

EDA can be run in two main forms:

- **Standalone:** This is a custom-built Docker container that is packaged to run `ansible-rulebook`. An Ansible Rulebook is simply the executable to run the program and start the source event listener. This can be synonymous with `ansible-playbook`.
- **EDA-Server:** This is an entire application surrounding the EDA framework. It includes a **user interface (UI)**, job queue (using **Redis**), database (**Postgres**), and workers nodes to execute the rulebook logic.

For this chapter, you will be learning about the EDA-Server approach. EDA is built on top of a mechanism called a **rulebook**. Similar to an Ansible playbook, it holds the logic about when the event is relevant and what will be executed. A rulebook has an event listener task that defines the source of the events that will be listened to. It then has a logic engine and an execution engine to handle conditional evaluation and action to execute.



Note

For more information on EDA, visit:

<https://ansible.readthedocs.io/projects/rulebook/en>.

The recipes covered in this chapter are as follows:

- Installing EDA-Server
- Creating projects in EDA-Server
- Creating a Rulebook
- Creating a decision environment
- Activating a rulebook in EDA-Server
- Preparing AWX, Nautobot, and EDA-Server
- Executing an event-driven workflow



Important

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch14.

In this chapter, you will build an event-driven workflow that includes some tools used previously, together with Ansible EDA-Server. In the following diagram, you will learn about all the inter-working elements that will come together throughout this recipe and chapter.

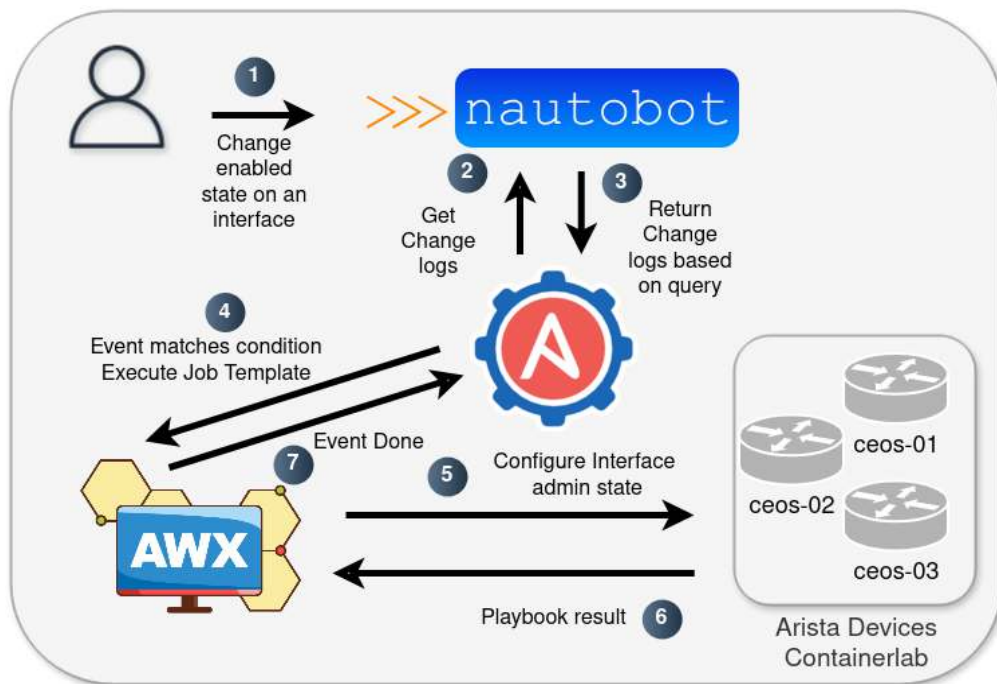


Figure 14.1 - Architecture overview

Let's quickly explain the elements from the diagram in more detail:

- **Nautobot:** The *source of truth* that manages the intended state of the network devices. The event-driven workflow starts here. The change of the enabled state of an interface in Nautobot will automatically create an object change record within Nautobot's database.
- **AWX:** The automation engine exposes an Ansible playbook (via an AWX job template). The job template will be using the Arista EOS declarative interfaces module, which can change the admin state of an interface (e.g., shutdown/no shutdown).
- **EDA-Server:** You will build EDA-Server on top of the Kubernetes kind cluster that was introduced in *Chapter 13*. It is the event-driven logic engine. EDA-Server will run an Ansible rulebook. It will poll the Nautobot API every 5 seconds to take changelog entries and put them into an event queue to be processed. The rulebook will then analyze the events, evaluate a conditional statement, and then trigger the AWX job template.

Let's kick things off by installing the EDA-Server application.



Important

This chapter is designed to be used exclusively with the fully integrated lab environment provided by the book. The manual installation steps are purely informational, and attempting a manual setup should be avoided unless you're an expert user.

Installing EDA-Server

EDA-Server follows a similar architectural design as AWX. Revisit the section on installing AWX in *Chapter 13* if you need a refresher. In this section, you will use the development Kubernetes kind environment that was created in that chapter to install the EDA-Server application:



Note

This recipe assumes that kind, kubectl, and Helm are already installed, as discussed in the last chapter.

1. To simplify the cluster deployment, you have a helper script (https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/blob/main/chapters/ch14/eda_quickstart/quick-start.sh) that sets up the EDA-Server cluster using the EDA-Server Operator (analogous to the AWX Operator in the previous chapter). The admin credentials are provided at the end of the output:

```
root@cookbook-lab:~# curl -sSL https://raw.githubusercontent.com/
network-automation-cookbook/cookbook-lab/main/chapters/ch14/eda_
quickstart/quick-start.sh | bash
Installing/upgrading EDA Server Operator using Kubectl...
namespace/eda-server-operator-system created
... omitted output ...
Applying EDA instance manifest...
eda.eda.ansible.com/eda-demo created
EDA Server username: admin
EDA Server password: <password for admin>
You can access the EDA Server at: http://<your-server-ip>:8384
Note: The EDA Server may take a few minutes to become fully
operational.
```

2. After a few minutes (it takes time to apply all the database migrations), you will be able to access the UI via `http://<your-server-ip>:8384`. This is the login portal you will get:

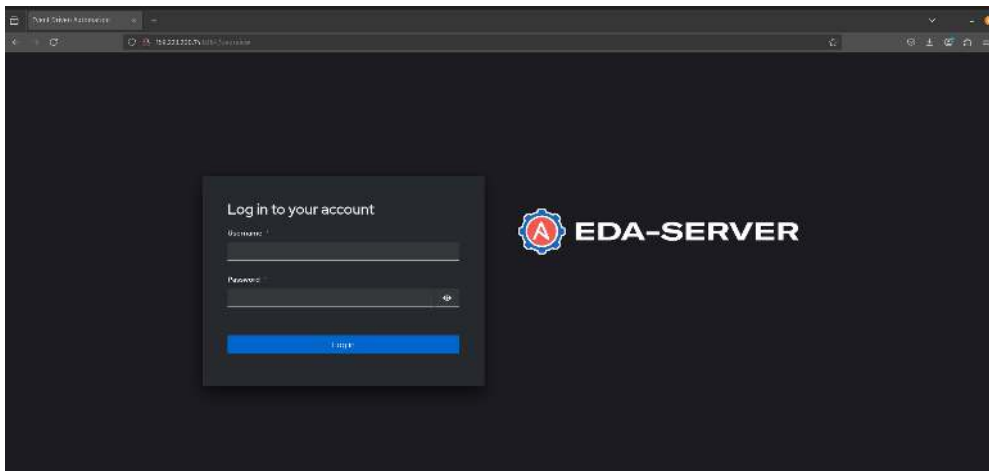


Figure 14.2 – EDA-Server login

3. Finally, after logging in to the EDA-Server GUI, you can see the main dashboard, and you will notice the application itself appears to look very similar to AWX:

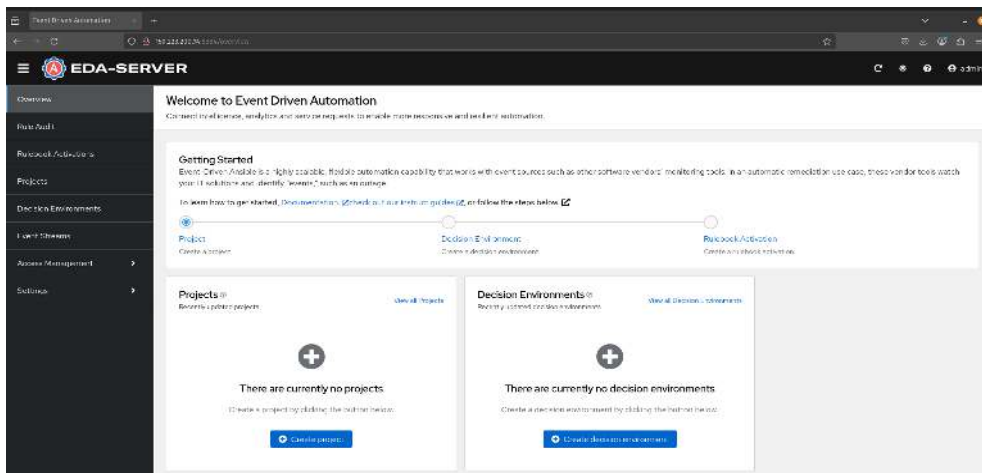


Figure 14.3 – EDA-Server home page

 Quick tip: Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 The next-gen Packt Reader and a free PDF/ePub copy of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



You can verify the EDA-Server deployment, with the component that you expect, using the `kubectl` CLI. `eda-server-operator` takes care of managing the deployment, and the UI, API, and database match the expected architecture explained previously:

```
root@cookbook-lab:~# kubectl get pods -n eda-server-operator-system
NAME                                READY   STATUSRESTARTS   AGE
eda-demo-activation-worker-5cc5864cb9-7lm6j  1/1 Running    0               9m51s
eda-demo-api-68676c5fb7-bdlj6              2/2 Running    0               10m
eda-demo-default-worker-6f9f77c496-rzj5p    1/1 Running    0               9m53s
eda-demo-postgres-13-0                     1/1 Running    0               11m
eda-demo-redis-5d7747879-nvss4              1/1 Running    0               11m
eda-demo-ui-6bd45f867f-rkfss               1/1 Running    0               9m56s
... omitted output ...
```

You will reuse the same Containerlab topology that was created and used in *Chapter 13*; this is just a convenience adjustment to simplify the connectivity between AWX, EDA-Server, and the network devices (which have been reduced to only the Arista devices, for simplicity). This step may not be required if you still have your topology up and running from *Chapter 13*:

```
root@cookbook-lab:~# labcli containerlab deploy --topology ./
containerlab/cookbook-awx-lab.yml
```

Now that EDA-Server is up and running, you can start managing some functionalities.

Creating projects in EDA-Server

Since the EDA server itself follows the same application design as AWX, you will not be creating and managing users, teams, or organizations again (it is exactly the same). The focus is on the minimum setup needed in order to get an Ansible Rulebook up and running utilizing the EDA-Server application. The rulebook itself is the actual unit of work that is running and executing the logic, a key difference from a playbook that executes tasks in sequence until it's finished. A rulebook has `if/else` logic capabilities built in, which makes the event-based approach function.

Since you will not be creating all the different objects in the application, you will use the defaults. The EDA-Server application includes prepackaged organizations, roles, and users, which will all be used during this chapter.

Let's get started by creating a project. To streamline this section, you will build on top of the forked `awx_sample_project` that was created in *Chapter 13*. The same repository will be used and you will simply augment it to support the required updates to add EDA rulebooks:

1. In the EDA-Server interface, create a new project with **Source control type** set as **Git**, and point to the previously created repository and the branch used, as shown in the following screenshot (notice the similarity with AWX):

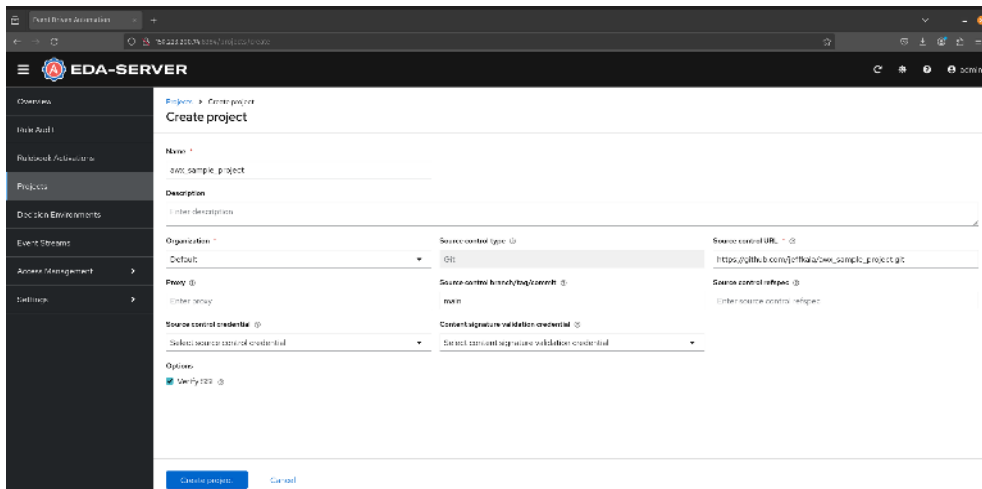


Figure 14.4 – EDA-Server Create project screen

2. Once the project is created, it will be synced in from Git. You will notice an import error is visible in the output. This is expected because you have yet to define any EDA constructs in the repository. By default, the project is searching the repository for one of two locations where rulebooks should exist: in the `rulebooks` directory or the `extensions/eda/rulebooks` directory.

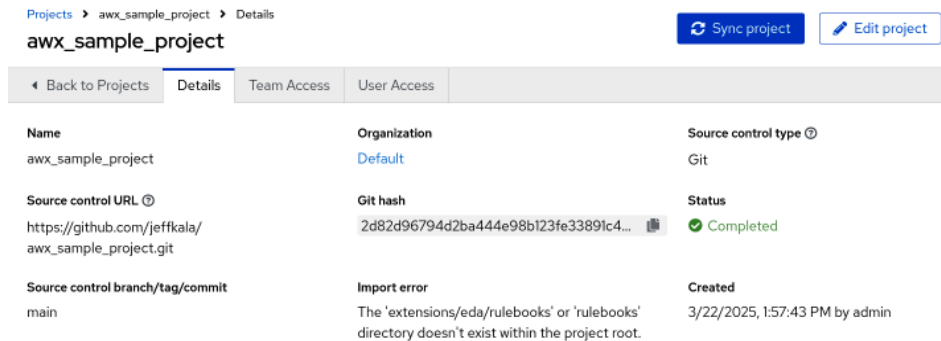


Figure 14.5 – EDA-Server project details

At this time, you have an EDA project synced in successfully but with an import error, which you are going to address by creating your first rulebook!

Creating a Rulebook

A rulebook has an event listener task that defines the source of the events that will be listened to. EDA has many powerful event sources built into the community package, including Kafka topics, AWS SQS queues, Azure Service Bus, and many other generic tools, such as file watch, webhooks, and URL checkers. These fall into two general categories: **push** and **pull**. The push category encompasses the message bus, which is near-real time, whereas webhooks and URL checkers fall into the pull category. These sources provide an interval configuration option to continuously pull in events.

Note

Visit the supported event source list at:

<https://ansible.readthedocs.io/projects/rulebook/en/latest/sources.html>.

Once the source of your events is defined and configured, you will now define actions based upon the events. These actions can run Ansible playbooks, trigger AWX job templates, or simply debug the events.

In this chapter, you will use a *custom event source plugin* that is packaged with the Nautobot Ansible collection. This source plugin queries a Nautobot instance's changelog (where Nautobot annotates every object change such as creation, update, or deletion) and creates events based on changes to the source of truth data.

You will learn how to use these events to monitor the changelog for changes such as a new administrative status of an interface. The source plugin will query Nautobot to find new changes to objects. Upon the receipt of an event, EDA-Server will execute the action of running an AWX playbook. In this playbook, you will use the Arista EOS interfaces declarative module to take the interface name and current interface enabled state and configure that on the router.

Now that the basics of a rulebook have been explained, let's build our first rulebook!

1. Navigate to your forked repository:

```
root@cookbook-lab# cd ~/awx_sample_project
```

2. Create a new directory in the project, named `rulebooks`, and create a new rulebook named `rb_print_events.yml`. The `rulebooks` directory is what the EDA-Server project searches within to find rulebooks that can be linked:

```
root@cookbook-lab:~/awx_sample_
project# mkdir rulebooks && cd rulebooks/ && vim rb_print_events.yml
---
- name: "Watch for new Nautobot changelog entries"
  hosts: "localhost"
  sources: (1)
- networktocode.nautobot.nautobot_changelog: (2)
  instance: "https://demo.nautobot.com" (3)
  token: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
  interval: 5 (4)
  query: "{{ now() | urlencode }}" (5)
  rules: (6)
- name: "New changelog created"
  condition: "event.meta is defined" (7)
  action: (8)
    print_event:
      pretty: true
```

The numbered references in the preceding code snippet are explained as follows:

- (1) The `sources` keyword defines the source event plugin to use to generate events.
- (2) Use the event source plugin that comes packaged with the `nautobot-ansible` collection (this is not yet installed and will be explained in the next section) and define the arguments for the Nautobot instance to connect to.
- (3) Since there are still some prerequisites to get set up and configured in our lab environment, for now, we will use the public Nautobot sandbox to validate that our rulebook is working as intended. The token is available by default.
- (4) The `interval` parameter defines how often EDA checks for changes. Thus, it defines the accuracy of the process and the likelihood of detecting changes sooner with the correspondent resource usage.
- (5) The `query` parameter defines any query parameters we want to pass to the Object Change API. This might be for limiting change types to only creates or updates, but also this is a way to pass a timestamp to be sure you're only going to get events greater than or equal to the current datetime, the current day, and so on. `{{ now() | urlencode }}` is a Jinja2 filter to get the current datetime and URL-encode it for the API.
- (6) `rules` specifies the conditional logic to evaluate followed by the action that conditional logic should execute upon a true condition.
- (7) This condition evaluates the event that gets put into the queue and makes sure the event has an attribute named `meta` (this is always true). You will utilize a more complex condition statement in a later section.
- (8) The action is what gets processed upon a true result on the condition. In this rulebook, you will simply pretty print the event to validate that all the elements of the rulebook are running successfully.



Info

For a full list of native actions, see:

<https://ansible.readthedocs.io/projects/rulebook/en/latest/actions.html>.

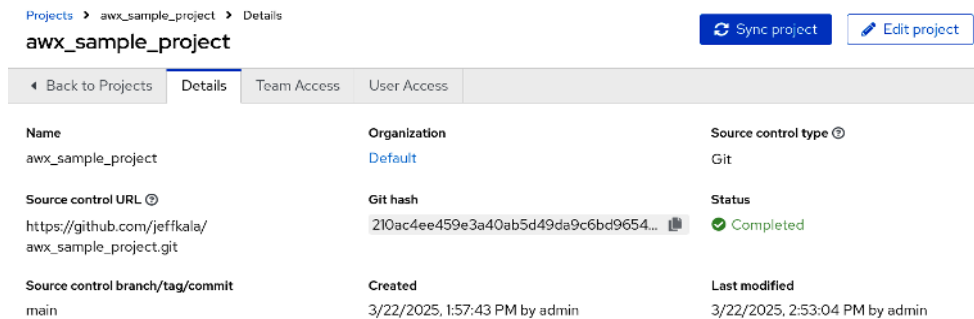
- Next, let's utilize our Git workflow to push these changes up into the remote repository:

```

root@cookbook-lab:~/awx_sample_project# git add rulebooks/rb_print_events.yml
root@cookbook-lab:~/awx_sample_project# git commit -m "add a rulebook"
[main 210ac4e] add a rulebook
1 file changed, 16 insertions(+)
create mode 100644 rulebooks/rb_print_events.yml
root@cookbook-lab:~/awx_sample_project# git push -u origin main
Username for 'https://github.com': <git_username>
Password for 'https://<git_username>@github.com':
... omitted output ...
93c160a..210ac4e  main -> main

```

- Finally, head back over to the EDA-Server UI and navigate to our project. Let's resync it and make sure the import error no longer exists.



Projects > awx_sample_project > Details

awx_sample_project

Sync project Edit project

Back to Projects Details Team Access User Access

Name	Organization	Source control type ⓘ
awx_sample_project	Default	Git
Source control URL ⓘ	Git hash	Status
https://github.com/jeffkala/awx_sample_project.git	210ac4ee459e3a40ab5d49da9c6bd9654...	Completed
Source control branch/tag/commit	Created	Last modified
main	3/22/2025, 1:57:43 PM by admin	3/22/2025, 2:53:04 PM by admin

Figure 14.6 – EDA-Server project sync

Great! Now that our project has rulebooks and it has been resynced successfully, you can move on to the next step: creating a decision environment that can actually run our rulebook.

Creating a decision environment

Similar to **execution environments (EEs)**, which you learned about in *Chapter 13*, Ansible follows suit in the EDA architecture by supporting **decision environments (DEs)**. The concepts of the DE are nearly identical to what was explained about EEs, with the main difference being the Ansible dependencies needed inside the container to run a rulebook versus a playbook. They're intended to be a fully packaged containerized environment that holds all your dependencies to execute a given rulebook, which includes system packages, Python libraries, and Ansible collections.

Unlike in the AWX chapter (*Chapter 13*), where you reused an existing EE, in this chapter, you will use a custom DE that has our dependencies packaged appropriately. The key dependencies are the Python library (pynautobot), the Ansible collection (nautobot-ansible), and the default EDA-specific dependencies (such as the ansible-rulebook executable).

Since the creation of EEs and DEs is out of the scope of this book, the DE has already been created and is available in the GitHub container registry for the book.

Note



Should you be interested in the steps that were used to create the DE for this chapter, review https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/blob/main/chapters/ch14/decision_environment/README.md.

Alternatively, more documentation on DEs can be found at:

https://ansible.readthedocs.io/projects/rulebook/en/v1.1.6/decision_environment.html.

Follow these steps:

1. In the EDA-Server web console, navigate to the **Decision Environments** tab. Click on the **Add** button and fill out the details as shown in the following figure, selecting the DE image build for the book:

The screenshot shows the 'Edit Cookbook Lab DE' form in the EDA-Server web console. The form has the following fields:

- Name ***: Cookbook Lab DE
- Description**: Enter description
- Organization ***: Default
- Image * @**: ghcr.io/jeffkala/awx_sample_project
- Credential @**: Select credential

Figure 14.7 – EDA-Server adding a decision environment

No credentials should be needed since the container registry is public.

That’s all there is for setting up the DE. Next, you will move on to creating an instance of the rulebook via creating a rulebook activation inside EDA-Server.

Activating a Rulebook in EDA-Server

Up until this point in this chapter, you have done a lot of setup and creation of prerequisites. Now, you can finally see the results of all the hard work. Let’s create a *rulebook activation* (which is the EDA-Server application equivalent to the AWX Job, which executes the rulebook) and demonstrate how to see the logs from the UI as well as how to relate that back into our Kubernetes cluster to find the actual Pod that gets spun up to run the rulebooks. Having the logs is invaluable during troubleshooting and also provides the ability to validate that a rulebook is actively running and retrieving events from the source plugin. Follow these steps:

1. In the UI, navigate to **Rulebook Activations** and click on **Add**. Fill out the details for the activation. A majority of the options come directly from the project you synced in previously. The creation includes the following:
 - **Organization:** Default
 - **Project:** awx_sample_project
 - **Rulebook:** rb_print_events.yml
 - **Decision Environment:** Cookbook Lab DE
 - **Log Level:** Info

Rulebook Activations > Create rulebook activation

Create rulebook activation

Name *

Description

Organization *
Default

Project *
awx_sample_project

Rulebook *
rb_print_events.yml

Event streams ⓘ
 ⓘ

Credential ⓘ
 ⓘ

Decision environment * ⓘ
Cookbook Lab DE

Restart policy * ⓘ
On failure

Log level * ⓘ
Info

Service name ⓘ

Figure 14.8 – EDA-Server add rulebook

2. Once you click **Create**, you will see information start displaying in regards to the rulebook activation status.



Figure 14.9 – EDA-Server rulebook status

3. The **Details** tab will start to change states until you see the container **Running** status activation, at which point you can navigate to the **History** tab to see an on-screen logging facility to see the execution logs in real time.

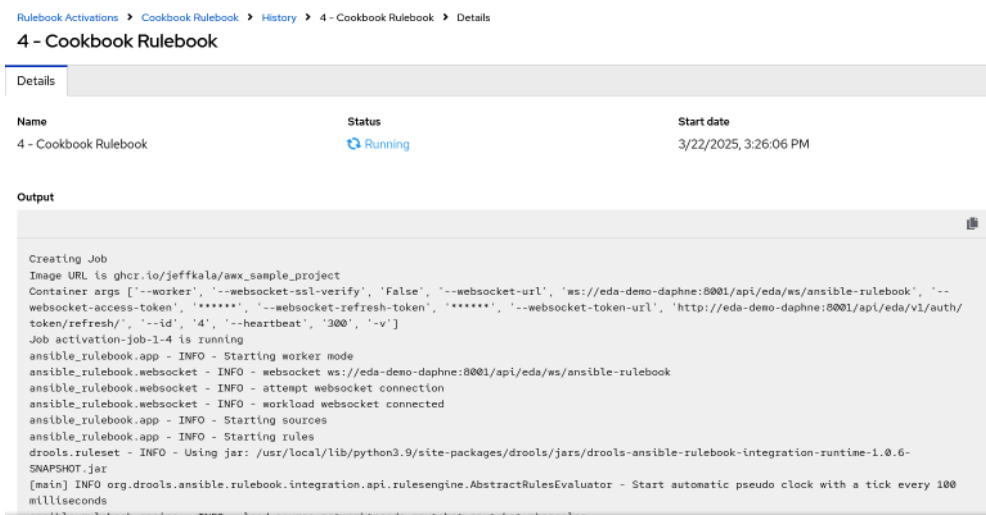


Figure 14.10 – EDA-Server rulebook output

4. Finally, you can scroll through these log outputs to see a few examples of actual events that are getting printed out by our rulebook. Here's an example of an interface enabled status being changed to disabled (i.e., 'enabled': False). The rulebook that is running at this time is grabbing all changelogs (not filtering until a later section), so the events you retrieve may look different. The default logic of the source plugin will grab the last days worth of changelogs if no specific query details are provided:

```
[event] *****
{'bridge': None,
 'created': '2025-03-11T15:38:54.832Z',
 'enabled': False,
```

```
'name': 'GigabitEthernet30',
... omitted output ...
'vrf': None}

*****
```

Since our rulebook action is set to print the event out, that is all you will see in the logs; however, this tells us the following:

- Our DE has all the dependencies needed to execute our rulebook
- Our source event plugin is able to reach out to Nautobot and read changelog records to be put into our EDA message queue
- Finally, it validates our condition, and the action triggers an execution (`print_event`)

Now, let's create an event-driven automated workflow where you create a rulebook to read specific changelogs and execute an AWX job template to toggle a port's admin status based on an external data change.

Preparing AWX, Nautobot, and EDA-Server

In this section, you will prepare AWX, Nautobot, and the EDA-Server application prerequisites in order to execute the end-to-end event-driven workflow described in the introduction to this chapter.

Let's get started by creating a new inventory in AWX named `eda-inventory` and setting it up with a single *Arista CEOS host*. This will ensure that the inventories you created in other chapters in the book don't conflict with the topic at hand:

1. Navigate to **Inventories** in the AWX UI, and create a new inventory called `eda-inventory` within the **Default** organization.
2. In the **Hosts** tab of the inventory, create the `ceos-01` host device, and create the `ansible_host`, `ansible_network_os`, and `ansible_connection` variables under the **Variables** section, as shown in the following screenshot:

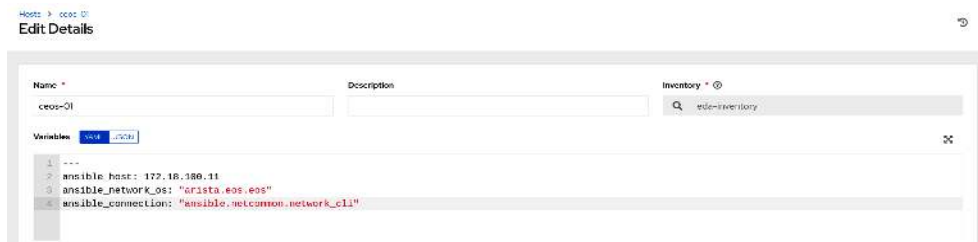


Figure 14.11 – AWX host variables

3. Make sure the credentials still exist from *Chapter 13*. If not, navigate to **Access Management** in the left navigation bar, select **Credentials**, and then **Create credential** to create the login credentials required to access the Arista network devices. Use the **Machine** credential type and specify the username and password used to log in to the devices (admin/admin). Also, set the escalation method to **enable**.

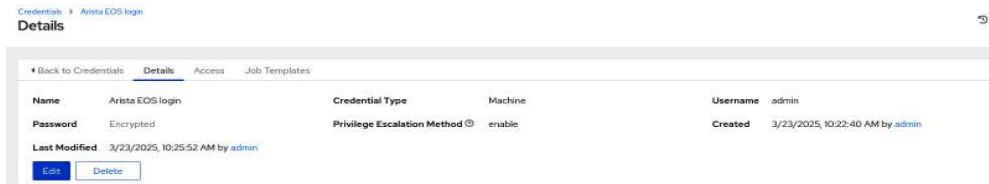


Figure 14.12 – AWX host credentials

4. Next, create the same EE that you used previously, but this time, link it to the **Default** organization (again, to ensure that scenarios built in other chapters don't conflict with our goals here). In the **Infrastructure** section in the left navigation bar, choose **Execution Environments**, and create a new EE. The most important parameter is **Image**, which defines the container image to use:

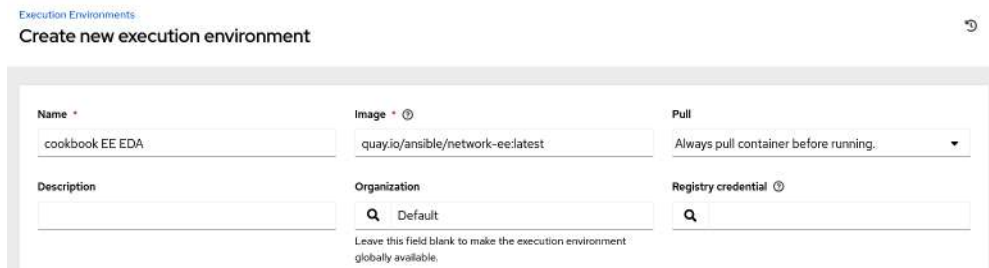


Figure 14.13 – AWX EE creation



Note

It is possible to leave **Organization** blank when creating an EE, which exposes it globally to the app.

Next, before you can execute changes to the interface enabled state in Nautobot to kick off the event-driven workflow, you need to get the Nautobot prerequisite setup for this workflow. In *Chapter 12*, you created devices and their required metadata. For this recipe, you will create a single Arista device with the name `ceos-01-eda`. The steps to create this device will not be covered

again; however, make sure the following exist. It is possible that you already have other devices in your Nautobot instance. It is recommended to create a new device with the details given here:

- A device with the name `ceos-01-eda`. As a reminder, a device requires multiple requirements to be created beforehand: locations, device types, and roles. Any of these can be reused from previous chapters to streamline the creation.
- A device platform (**Arista EOS**) assigned to the device
- It has a primary IP address (**172.18.100.11**) along with a management IP and interface.
- A second physical interface is available for testing. For this, click **Add Components** on the device and create interfaces with the **Ethernet[1-4]** pattern. You will then use Ethernet4 in the testing.

The following screenshots show those details for clarity:

>>> **ceos-01-eda**

📅 March 22, 2025, 10:32 p.m. | ⌚ 0 minutes ago

Device

Advanced

Contacts

Interfaces 5

Status

LLDP Neighbors

Configuration

Device	
Location	EDA Test Site (Site)
Rack	—
Position	—
Tenant	—
Device Family	—
Device Type	Arista CEOS (1U)
Serial Number	—
Asset Tag	—

Figure 14.14 – Nautobot Device details

For the example device, I created an **EDA Test Site** location, but any existing site will suffice as our workflow will be looking for specific interface changes.

Management	
Role	Network
Platform	Arista EOS
Status	Active
Primary IPv4	172.18.100.11/16 (Global)

Figure 14.15 – Nautobot device

Make sure that the created device has the primary IP address assigned, as this will ensure that our entire lab topology works in harmony and connectivity functions properly.

In order to create more interfaces for our testing, navigate to the device you created, click on **Add components**, and select **Interfaces**.

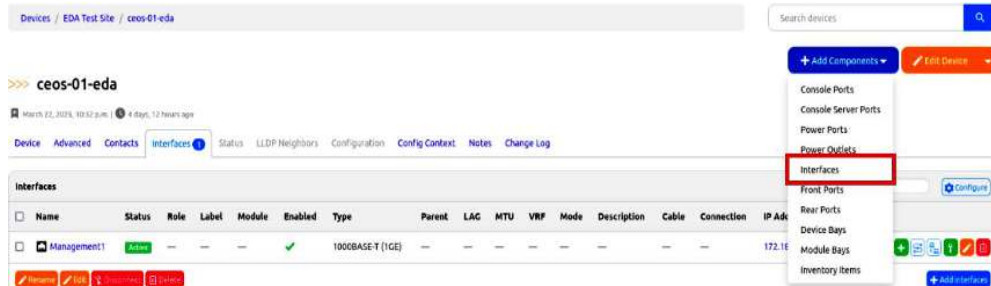


Figure 14.16 – Nautobot adding interfaces

From the **Interface** add component screen, you can use a regex to create multiple interfaces at the same time, as shown in the following figure with **Ethernet[1-4]**:

Interface

Device

ceos-01-eda

x

▼

Module

▼

Name

Ethernet[1-4]

Alphanumeric ranges are supported for bulk creation. Mixed cases and types within a single range are not supported.

Examples:

- [ge,xe]-0/0/[0-9]
- e[0-3][a-d,f]

Label

Label

Alphanumeric ranges are supported. (Must match the number of names being created.)

Status

Active

x

▼

Role

▼

Type

Virtual

x

▼

☒ Enabled

Figure 14.17 – Nautobot add interfaces

This creation will result in the interfaces listed in the following screenshot:

Device

Advanced

Contacts

Interfaces

Status

LLDP Neighbors

Configuration

Config Context

Notes

Change Log

Interfaces																Filter
Name	Status	Role	Label	Module	Enabled	Type	Parent	LAG	MTU	VRF	Mode	Description	Cable	Connection	IP Addresses	
 Ethernet1	Active	—	—	—	✓	1000BASE-T (1GE)	—	—	—	—	—	—	—	—	—	
 Ethernet2	Active	—	—	—	✓	1000BASE-T (1GE)	—	—	—	—	—	—	—	—	—	
 Ethernet3	Active	—	—	—	✓	1000BASE-T (1GE)	—	—	—	—	—	—	—	—	—	
 Ethernet4	Active	—	—	—	✓	1000BASE-T (1GE)	—	—	—	—	—	—	—	—	—	
 Management1	Active	—	—	—	✓	1000BASE-T (1GE)	—	—	—	—	—	—	—	—	172.18.100.11/16 (Global)	

Figure 14.18 – Nautobot Interfaces

Now that the prerequisites in AWX and Nautobot are set up and ready, you will create a new rulebook pointing to the Nautobot instance (instead of the public demo). The rulebook conditional will look specifically in the changelog event for a MAC address key, which is a simplified indicator that the changelog entry is specific to interface changes. Finally, you will create a new playbook that will ensure that the configuration on the device matches the source of truth data:

1. Navigate back into your fork of `awx_sample_project`, and create a new rulebook named `rb_interface_change.yml` inside the `rulebooks` directory:

```
root@cookbook-lab:~/awx_sample_project# vi rulebooks/rb_interface_
change.yml
---
- name: "Watch for new Nautobot changelog entries"
  hosts: "localhost"
  sources:
  - networktocode.nautobot.nautobot_changelog:
      instance: "http://<your Nautobot IP>:8080" (1)
      token: "0123456789abcdef0123456789abcdef01234567" (2)
      interval: 5
      query: ""
  rules:
  - name: "Find Interface based changelogs"
    condition: "event.mac_address is defined" (3)
    action:
      run_job_template: (4)
      name: "Interface Change" (5)
      organization: "Default"
```

The numbered references in the preceding code snippet are explained as follows:

- (1) The IP address where Nautobot is being hosted. In this example, you will point to the HTTP endpoint to avoid having to pass SSL certificates and/or remove SSL verification.
- (2) The Nautobot API token was defined way back during the lab setup and can be found in the `.env` file that you created.
- (3) The condition to evaluate in order to proceed to the action block. In this case, you are looking in the Nautobot changelog event for a key called `mac_address`. This is one of many different ways you could identify an interface-level change, but this one is simple to define.
- (4) The EDA action that the condition should trigger, `run_job_template`, is an action that can trigger an API call to AWX (in our case) or AAP/Tower if that is what is in use in your infrastructure. At this time, EDA doesn't know how to interact with AWX yet. You will configure this inside the EDA-Server application in an upcoming recipe.

- (5) The AWX job template name and organization to execute. The organization is required as AWX uniqueness constraint on a job template is name + organization.
2. Create a new playbook named `pb_interface_change.yml` inside the base fork directory:

```
root@cookbook-lab:~/awx_sample_project# vi pb_interface_change.yml
---
- name: "CONFIGURE PROPER INTERFACE STATUS"
  host: ceos-01 (1)
  tasks:
  - name: "Merge interface admin state"
    arista.eos.eos_interfaces: (2)
    config:
    - name: "{{ ansible_eda.event.name }}" (3)
      enabled: "{{ ansible_eda.event.enabled }}" (4)
      state: merged
```

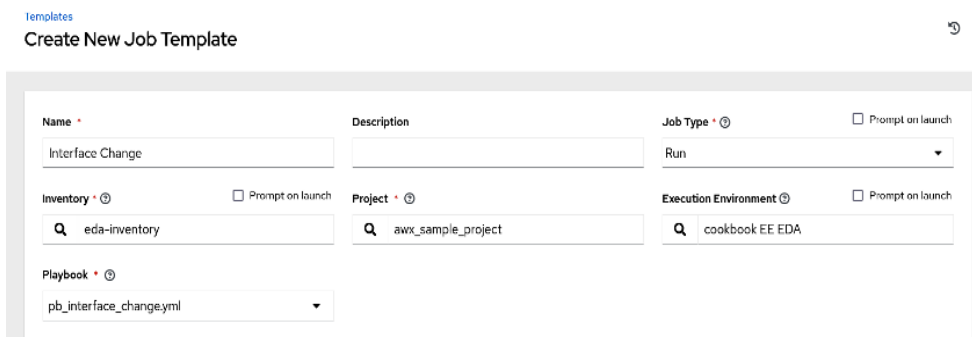
The numbered references in the preceding code snippet are explained as follows:

- (1) For simplicity, set the host argument to `ceos-01`, which is the host in our AWX inventory that you will be executing against.
- (2) The Arista declarative module interacts with basic interface-level configuration, including the admin state of the interface.
- (3) The EDA event that gets passed from EDA-Server into the `run_job_template` action; the event itself is nested inside the `ansible_eda.event` key. Here, the name is the interface name from the Nautobot changelog.
- (4) Since Nautobot is our *source of truth* in this workflow, the network will be configured strictly based on data from the system. This specific attribute is being set based on the object changelog from Nautobot. Nautobot holds the historical change data per object in its database. In this case, the event has the details of the interface and how it changed in Nautobot. The playbook grabs the interface enabled state for the event details and passes it to the `eos_interfaces` configuration module.

3. Commit and push the changes into the remote repository:

```
root@cookbook-lab:~/awx_sample_project# git add .
root@cookbook-lab:~/awx_sample_project# git commit -m "EDA updates"
[main (root-commit) b279ef7] EDA updates
... omitted output ...
root@cookbook-lab:~/awx_sample_project# git push -u origin main
```

4. Head back into both the AWX UI and the EDA-Server UI and sync your projects. This will ensure that the latest committed rulebook and playbook are seen from both applications.
5. In AWX, create a new job template that points to the new playbook that was just created in the repository:



The screenshot shows the 'Create New Job Template' interface in the AWX UI. The form is titled 'Create New Job Template' and includes a 'Templates' link in the top left. The form fields are as follows:

- Name:** Interface Change
- Description:** (empty)
- Job Type:** Run (dropdown menu)
- Inventory:** eda-inventory (dropdown menu)
- Project:** awx_sample_project (dropdown menu)
- Execution Environment:** cookbook EE EDA (dropdown menu)
- Playbook:** pb_interface_change.yml (dropdown menu)

There are three 'Prompt on launch' checkboxes, one for each of the Job Type, Inventory, and Execution Environment fields. The 'Prompt on launch' checkbox for the Job Type field is currently unchecked.

Figure 14.19 – AWX Create New Job Template

Let's look at some key elements from the preceding figure:

- Make sure you set the credentials to the Arista EOS login machine credentials that you created earlier.
- Important: One unique element of trigger job templates from EDA rulebooks is the requirement to get the EDA event-specific variable data passed into the job template. The way this is done is by setting the **Prompt on launch** selection specifically for the **Variables** section. This allows the event variable data to be passed from EDA inside the payload of the API call when EDA triggers the job template execution.

In the next figure, you will set a required variable called **Prompt on launch**:

The screenshot shows the 'Edit Details' page for a job template named 'Interface Change'. The form is organized into several sections: Name, Description, Job Type, Inventory, Project, Execution Environment, Playbook, Credentials, Labels, and Variables. The 'Prompt on launch' checkbox is checked in the Variables section, and this checkbox is highlighted with a red rectangle. The 'Variables' section also shows a 'VARIABLES' button and a 'JSON' button.

Figure 14.20 – AWX Edit Details

1. In order for EDA-Server to be able to trigger job templates via the API, it must have an AWX token to authenticate with. In AWX, you will navigate to **User -> User Details -> Token** and create a new token. Follow the next few steps to create your token:



Figure 14.21 – AWX navigating to Token

- Create a user token; **Application** can be left blank:

The screenshot shows the 'Create user token' page. The form has three main fields: Application, Description, and Scope. The 'Application' field is empty, the 'Description' field contains 'EDA rulebook', and the 'Scope' field is set to 'Write'. There are 'Save' and 'Cancel' buttons at the bottom left of the form.

Figure 14.22 – AWX Create user token

- Click **Create** and save the token for use in the next section:

Token information ×

! This is the only time the token value and associated refresh token value will be shown.

Token ▼ mNrLBXO4TDWosnqdPTif7O2oKcQbpz 📋

mNrLBXO4TDWosnqdPTif7O2oKcQbpz

Expires 7/23/3024, 7:05:47 PM

Figure 14.23 – AWX retrieved user token

2. Connect EDA-Server to AWX by navigating back into the EDA-Server UI and creating a credential definition for the token you just generated in AWX. Go to **Access Management** -> **Credentials** and create a new credential:
 - **Credential type:** Red Hat Ansible Automation Platform (AWX and AAP can be used, as explained in *Chapter 13*)
 - **Red Hat Ansible Automation Platform:** http://<server ip>:8383/api/controller
 - **OAuth Token:** The AWX token you generated in the last step

The following figure shows these options in detail:

EDA-SERVER

Overview
Rule Audit
Rulebook Activations
Projects
Decision Environments
Event Streams
Access Management ▼
Users
Teams
Organizations
Roles
Credentials

Credentials > Create credential

Create credential

Name *
AWX EDA Token

Description
Enter description

Organization * Default **Credential type *** Red Hat Ansible Automation Platform

Type Details

Red Hat Ansible Automation Platform * ⓘ **Username** ⓘ
http://159.223.200.74:8383/api/controller/

OAuth Token ⓘ

Figure 14.24 – EDA-Server Create credential

3. Inside EDA-Server, create a new rulebook activation that links to the new rulebook that was created previously:
 - In **Credentials**, make sure you set the AWX EDA token you just created
 - Set **Log level** to **Info**, which provides more details while you're testing



Figure 14.25 – EDA-Server Create rulebook activation

4. Lastly, ensure that the new rulebook activation was successful by checking that it is running:

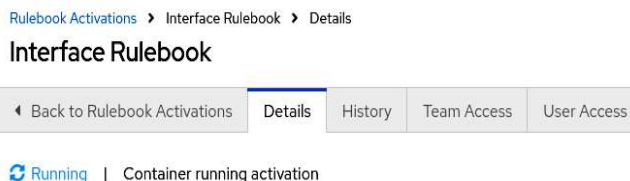


Figure 14.26 – EDA-Server rulebook details

Wow! That was quite a lot of prerequisites to tackle, but now you are finally ready to watch the magic of event-driven automation run. In the final section of this chapter, you will make a change to an interface state in Nautobot and watch all the components work together to configure that state on the Arista network device.

Executing an event-driven workflow

Now that the EDA-Server application, AWX application, and Nautobot infrastructure are deployed and configured, you can execute an event-driven workflow using EDA. This workflow is just the tip of the iceberg as far as event-driven automation possibilities. Imagine executing automated troubleshooting based on events triggered by an observability platform, or even kicking off automated configuration remediation tasks based on a device reporting a configuration compliance failure.

This recipe follows the task execution flow given here:

1. A change in Nautobot to the ceos-01-eda interface Ethernet4 enabled state from disabled (unchecked) to enabled (checked).
2. EDA-Server rulebook activation reads the Nautobot changelog from the REST API and adds events into the EDA event queue. This recurs at a configurable interval in the source event plugin (every 5 seconds by default).
3. The rulebook condition reads the event data and validates that the `mac_address` key is defined.
4. The rulebook action triggers an API call to the AWX job template to kick off the job template with the event details provided as extra variables.
5. AWX runs the job template and configures the interfaces defined in the event with the admin state defined in Nautobot (the source of truth).

Let's kick off our example!

1. Navigate to the ceos-01-eda device in Nautobot and go into the Ethernet4 interface object. Notice that the current **Enabled** state is false:

>>> **ceos-01-eda / Ethernet4**

📅 March 22, 2025, 11:49 p.m. | ⌚ 0 minutes ago

[Interface](#) [Advanced](#) [Contacts](#) [Notes](#) [Change Log](#)

Interface	
Device	ceos-01-eda
Name	Ethernet4
Status	Active
Role	—
Label	—
Type	1000BASE-T (1GE)
Enabled	✗

Figure 14.27 – Nautobot Interface details

2. Log in to the device to validate that the current admin state of the interface is shutdown:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch14# ssh admin@ceos-01
admin@ceos-01's password:
ceos-01>show interfaces ethernet4 status
Port   Name   Status   Vlan Duplex Speed  Type   Flags Encapsulation
Et4                disabled 1    full  1G    EbraTestPhyPort
```

3. Edit the interface in Nautobot and change the **Enabled** status to true (checked). Then, click on the **Update** button to save the changes.

Editing interface Ethernet4

Interface

Device: ceos-01-eda

Name: Ethernet4

Label: Label

Physical label

Status: Active

Role:

Type: 1000BASE-T (1GE)

☒ Enabled

Figure 14.28 – Nautobot enabled interface

4. Head back over to your EDA-Server UI, navigate to the rulebook activation named **Interface Change**, click on the **History** tab, and open up the most recent execution:

[Rulebook Activations](#) > [Interface Rulebook](#) > [History](#) > 22 - Interface Rulebook > Details

22 - Interface Rulebook

Details

Name	Status	Start date
22 - Interface Rulebook	Running	3/23/2025, 9:38:10 AM

Figure 14.29 – EDA-Server rulebook details

5. Scroll to the end of the **Rulebook Activations** console log to see the new event come in and the preceding AWX job template trigger:

```
ansible_rulebook.action.run_job_template - INFO - running job
template: Interface Change, organization: Default
ansible_rulebook.action.run_job_template - INFO - job results url:
http://159.223.200.74:8383/#/jobs/52/details
ansible_rulebook.action.run_job_template - INFO - controller job id:
52
```

You can see in the console logs that the job template was triggered, and it reports that the AWX job ID was 52.

6. Let's go back into AWX and navigate to **Jobs** and look at the details for job ID 52:

Jobs 🔍

☐ Name
1 - 20 of 52 ◀ ▶

Name	Status	Type	Start Time	Finish Time	Actions
52 - Interface Change	✔ Successful	Playbook Run	3/23/2025, 12:13:08 PM	3/23/2025, 12:13:16 PM	

Figure 14.30 – AWX job template

7. Open the job template log to see the status. The logs indicate the configuration was changed, as you would expect:

```

0 [WARNING]: You are running the development version of Ansible. You should only
1 run Ansible from "devel" if you are modifying the Ansible engine, or trying out
2 features under development. This is a rapidly changing source of code and can
3 become unstable at any point.
4 SSH password:
5
6 PLAY [CONFIGURE PROPER INTERFACE STATUS] ***** 12:13:11
7
8 TASK [Merge interface admin state] ***** 12:13:11
9 changed: [ceos-01]
10
11 PLAY RECAP ***** 12:13:15
12 ceos-01 : ok=1 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Figure 14.31 – AWX job template details

8. Finally, head back into the device and validate that the change was successful:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch14# ssh admin@ceos-01
admin@ceos-01's password:
ceos-01>show interfaces ethernet4 status
Port   Name   Status   Vlan Duplex Speed  Type      Flags
Encapsulation
Et4          connected1    full   1G EbraTestPhyPort
```

Fantastic! Our event-driven automation workflow was successful. Our source of truth (Nautobot) drove automation to configure our intended state into the network device. This example, although simple, could offer a huge win for many organizations. Toggling an interface state is a common troubleshooting technique used across various teams. Imagine being able to drive that from simply changing a checkbox in the source of truth.



Note

This chapter utilized the Ansible event source for the Nautobot changelog; however, EDA comes natively with many other event sources, from polling URLs for statuses to message bus event notifications. Learn more about the native event sources at https://galaxy.ansible.com/ui/repo/published/ansible/eda/content/?showing=eda%2Fplugins%2Fevent_source.

Summary

In this chapter, you deployed and configured Event-Driven Ansible and various other components. You went through a multitude of prerequisites that helped solidify previous chapters by using Nautobot and AWX again. Finally, you executed an event-driven automation workflow to take the source of truth data in Nautobot and configure that intended state into the network. This is such a powerful mechanism within the network automation space, and EDA exposes a fantastic entry point into the event-driven space. Consider the endless possibilities that are exposed by this framework: automatic configuration changes triggered by certain conditions or compliance failures, triggering automated troubleshooting based on event notification from your observability platforms when an interface goes down or bandwidth dramatically changes, or finally, the next level where a message bus technology starts driving and stitching together multiple workflows at the same time without any human intervention. These are all exciting possibilities that you can now start implementing to solve real problems in your network automation strategy.

**Hint**

To deep dive into modern network observability, we recommend *Modern Network Observability* by Packt Publishing.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



15

Terraform and Ansible Dancing Together

So far in this book, you've explored how Ansible can automate network operations across various scenarios. However, Ansible is not the only tool available for managing infrastructure. Several other tools take different or complementary approaches, and one of the most widely used is **Terraform** (<https://developer.hashicorp.com/terraform>) by HashiCorp (now part of IBM). Unlike Ansible, which focuses on executing sequential tasks, Terraform defines the *desired final state* of a dynamic infrastructure (i.e., compute, networking, or storage that is provisioned on demand) and ensures it reaches that declarative state.

This chapter will examine approaches to combine Terraform and Ansible through practical examples. While it does not cover Terraform in depth, it will provide essential concepts to highlight the differences between Ansible and Terraform. If you need more details, more documentation is available, including network-related content in *Network Programmability and Automation, 2nd Edition*, which we recommended in *Chapter 1*.

A common question is: *Is there anything Terraform can do that Ansible cannot?* The short answer is *no*—both tools can achieve similar outcomes. However, their underlying approaches differ, which can have a significant impact depending on the scenario.



Note

Terraform is not the only tool for managing dynamic infrastructure, but it is one of the most popular.

Terraform offers several advantages when working with dynamic infrastructure, particularly in cloud environments:

- **Declarative versus procedural approach:** Terraform follows a declarative model, meaning you define the final state of your infrastructure, and Terraform determines how to achieve it. This simplifies dependency management, as relationships between resources are implicitly defined (e.g., a VPC must exist before launching an instance inside it). Ansible, on the other hand, executes tasks in a defined sequence, requiring more manual dependency handling.
- **State management:** Terraform maintains a *state file* that tracks deployed infrastructure resources and their identifiers. This allows for better planning, updates, and deletions, making infrastructure lifecycle management more predictable.
- **Immutable infrastructure:** Terraform encourages an immutable approach, where changes result in resource replacements rather than in-place modifications. This reduces the risk of configuration drift, which can occur when incremental updates are applied over time.
- **Robust dry-run capability:** Terraform provides a global preview of expected changes using `terraform plan`, while Ansible's `--check` mode only validates individual tasks without guaranteeing overall accuracy.

Because of these features, Terraform has gained popularity for *cloud infrastructure provisioning*. While you can provision cloud resources using Ansible (as covered in *Part 3*), many organizations prefer to use *Terraform for infrastructure provisioning* and *Ansible for configuration management*.

In cloud and hybrid networking scenarios, combining Terraform and Ansible can be highly effective. Examples include the following:

- **Infrastructure provisioning with Terraform and configuration management with Ansible:** You can use Terraform to deploy a virtual firewall or router in a cloud environment, and then apply configuration settings using Ansible
- **Dynamic VPN management:** Terraform can provision a VPN connection in the cloud, while Ansible dynamically configures the corresponding data center VPN endpoint based on Terraform's output

Many organizations follow this hybrid approach to leverage the strengths of both tools efficiently. Understanding how to integrate them is the focus of this chapter.

This chapter is structured into the following hands-on recipes:

- Deploying dynamic infrastructure with Terraform
- Integrating Terraform outputs with Ansible
- Automating firewall rule provisioning with Ansible
- Integrating Terraform with AWX
- Cleaning up dynamic infrastructure with Terraform

By the end of this chapter, you will have a solid understanding of how to use Terraform and Ansible together to streamline infrastructure deployment and network automation.



Note

The code files for this chapter can be found here: https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition/tree/main/chapters/ch15.

In this chapter, you will explore a practical use case where Terraform and Ansible complement each other effectively. The scenario consists of deploying an Ubuntu server on AWS using Terraform and then managing its iptables firewall rules with Ansible.

Initially, Terraform provisions the server and applies a baseline firewall configuration. However, since firewall rules must be frequently updated based on a public blacklist of IPs, Ansible will be used to adjust the rules over time dynamically. This will be achieved by periodically executing an Ansible playbook to fetch the latest blacklist and update the firewall accordingly.

This integration showcases how Terraform handles infrastructure provisioning while Ansible manages ongoing configuration changes, making the solution both efficient and maintainable.

The first recipe will cover an introduction to Terraform to understand its basics.

Deploying dynamic infrastructure with Terraform

Terraform is an **infrastructure as code (IaC)** tool that enables the ability to define, provision, and manage infrastructure in a declarative manner. Before getting hands-on, it's important to understand a few key concepts:

- **Configuration plan:** A set of files that define the desired state of the infrastructure using a **domain-specific language (DSL)**—in this case, **HashiCorp Configuration Language (HCL)**.

- **Providers:** Packages that allow Terraform to interact with external systems via APIs. For example, the AWS provider enables Terraform to manage resources in the AWS cloud. Similar to Ansible collections.
- **Resources:** Individual infrastructure components, such as **virtual machines (VMs)**, databases, or networks, that Terraform provisions based on the configuration plan. It is similar to Ansible modules.
- **Data sources:** Existing infrastructure components outside Terraform's control that provide useful data to the configuration plan.
- **Variables and outputs:** Variables make configuration plans flexible and reusable, allowing adjustments based on different inputs. Outputs return useful deployment details, such as a public IP address.
- **State management:** Terraform maintains a state file to track the current infrastructure, enabling effective updates, modifications, and resource management over time.

In Terraform, you start creating a configuration plan file. Similar to a playbook in Ansible, this is where all components are connected.



Note

Like in *Chapter 8*, all the infrastructure used in this chapter is included in the free tier.

However, before getting into your first Terraform configuration plan, you have to start with a few prerequisites: first, install Terraform, and secondly, create (if you don't have one already) an SSH key in the AWS cloud to allow the provision of the Ubuntu server via Ansible after its provisioning. Here are the steps to do that:

1. Install Terraform in your environment. In the Ubuntu server referenced in the book lab, you can use `snap` (a package management system) to streamline the installation:

```
root@cookbook-lab:~/cookbook-lab# snap install terraform --classic
terraform 1.11.1 from Snapcrafters installed
```

2. Create an SSH key in your AWS account (for more details about how to get started, see *Chapter 8*) by logging in to the AWS console and accessing the **EC2** dashboard | **Create key pair**. This automatically downloads the **Privacy Enhanced Mail (PEM)** key that you will leverage in your Ansible access credentials.

us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1

EC2 > Key pairs > Create key pair

Create key pair [Info](#)

Key pair
A key pair, consisting of a private key and a public key, is a set of security credentials that you use to prove your identity when connecting to an instance.

Name
cookbook-key
The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type [Info](#)
☒ RSA
 ☐ ED25519

Private key file format
☒ .pem
For use with OpenSSH
☐ .ppk
For use with PuTTY

Tags - optional
 No tags associated with the resource.
[Add new tag](#)
You can add up to 50 more tags.

[Cancel](#) [Create key pair](#)

CloudShell Feedback Privacy Terms Cookie preferences

- Copy the private key downloaded in the previous step (in the example, into `/tmp/cookbook-key.pem`) to the server and adjust the key permissions:

```
root@cookbook-lab:~/cookbook-lab# chmod 400 /tmp/cookbook-key.pem
```

After these prerequisites are in place, it's time to generate your first Terraform configuration plan:

- Create a new folder for this chapter, `ch15`, which will hold all the content:

```
root@cookbook-lab:~/cookbook-lab# mkdir chapters/ch15 && cd chapters/ch15
```


5. Create a simple Terraform plan to create an Ubuntu server in AWS, starting by defining the networking settings (reusing existing default resources when possible):

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi main.tf
provider "aws" { (1)
    region = "us-east-1" # Change to your preferred region
}

# Get the default VPC
data "aws_vpc" "default" { (2)
    default = true
}

# Get the first available default subnet in the default VPC
data "aws_subnet" "default" { (3)
    filter {
        name     = "vpc-id"
        values   = [data.aws_vpc.default.id]
    }
}

# Security Group allowing SSH access
resource "aws_security_group" "ssh_access" { (4)
    name          = "allow_ssh"
    description   = "Allow SSH access from the internet"
    ingress {
        from_port = 22
        to_port   = 22
        protocol  = "tcp"
        cidr_blocks = ["0.0.0.0/0"] # WARNING: Open to the internet,
                                   restrict in production
    }
    egress {
        from_port = 0
        to_port   = 0
        protocol  = "-1"
        cidr_blocks = ["0.0.0.0/0"]
    }
}

... continues in the next step ...
```

The numbered references in the preceding code are explained as follows:

- (1) The AWS provider section configures the provider and implicitly defines that this provider is required. Providers are not installed by default in Terraform, so defining them will enforce the installation of the necessary dependencies when initializing Terraform in the next step.
 - (2) To skip the creation of a new VPC, the plan uses data from the default VPC that comes out of the box in every AWS account.
 - (3) Analogous to the VPC, the plan also leverages the existing default subnets to connect to the parent VPC, leveraging the VPC data defined in the previous step.
 - (4) Adjust the security group, or AWS instance firewall rules, to allow access from the instance to the internet, and allow external SSH access to it. Obviously, in a production environment, you should restrict access to the specific IP from which you will be managing the infrastructure.
6. Extend the previous Terraform plan with the compute instance:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi main.tf
... continues from the previous step ...
# Fetch the latest Ubuntu AMI
data "aws_ami" "ubuntu" { (1)
    most_recent = true
    owners      = ["099720109477"] # Canonical's AWS account ID
    filter {
        name     = "name"
        values   = ["ubuntu/images/hvm-ssd/ubuntu-*22.04-amd64-server-*"]
    }
}
# EC2 Instance
resource "aws_instance" "ubuntu" { (2)
    ami                = data.aws_ami.ubuntu.id
    instance_type      = "t2.micro"
    subnet_id          = aws_subnet.public_subnet.id
    vpc_security_group_ids = [aws_security_group.ssh_access.id]
    associate_public_ip_address = true
    key_name            = "cookbook-key"
}
```

```
output "public_ip" { (3)
  value = aws_instance.ubuntu.public_ip
}
```

The numbered references in the preceding code are explained as follows:

(1) AWS provides VM images (named **Amazon Machine Images**, or **AMIs**) that are pre-packed with the operating systems, packages, and configurations. In this case, you will use the Ubuntu image provided by Canonical (the company behind Ubuntu) that matches a specific version with its latest release.

(2) Next, an **Elastic Compute Cloud (EC2)** instance (a VM in AWS) is defined with public IP access, using all the previous references (e.g., the AMI, the subnet, and the security group). This dependency connection is what provides Terraform with the right sequence of actions. Until all the references are resolved, you can't create this resource. Also, notice that you are specifying a `key_name` value, which should match the SSH key name that you created in the AWS console.

(3) Finally, the output will print on the screen, which is the resulting public IP associated with the EC2 instance.

7. With the Terraform configuration plan, you have all you need to get started with Terraform, but as mentioned, your system is still missing the AWS provider that is defined in the plan. To solve all the dependencies according to the plan, you use `terraform init`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.90.0...
- Installed hashicorp/aws v5.90.0 (signed by HashiCorp)
... omitted output ...
Terraform has been successfully initialized!
```

8. Before running Terraform, you have to provide Terraform with the necessary credentials to interact with the AWS API. Similar to what you did in *Chapter 8*, create a `~/.aws/credentials` file with access and secret keys:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# mkdir ~/.aws && vi
~/.aws/credentials
[default]
```

```
aws_access_key_id = your-access-key
aws_secret_access_key = your-secret-key
```

9. Finally, you are ready to deploy your first Terraform plan. Terraform allows a dry-run check before actually applying terraform plan, but because it is also presented with the apply mode, you can trigger it directly and check the plan before validating it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform apply
... review and accept the plan ...
aws_security_group.ssh_access: Creating...
aws_security_group.ssh_access: Creation complete after 2s [id=sg-
0dffaa48a4c6c2414]
aws_instance.ubuntu: Creating...
aws_instance.ubuntu: Still creating... [10s elapsed]
aws_instance.ubuntu: Still creating... [20s elapsed]
aws_instance.ubuntu: Still creating... [30s elapsed]
aws_instance.ubuntu: Creation complete after 32s [id=i-
04d7ab6d2da2507e9]
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
Outputs:
public_ip = "54.173.181.113"
```

10. Reading the previous output, it seems that the EC2 instance has been created and is ready to use. Let's verify it by connecting via SSH to the public IP (use the ubuntu user as it's the one configured by default) with the SSH key that was specified:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# ssh
ubuntu@54.173.181.113 -i /tmp/cookbook-key.pem
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1024-aws x86_64)
... output omitted ...
ubuntu@ip-172-31-34-114:~$
```

If you check the ch15 folder, you will notice a new terraform.tfstate file. This is the default option where Terraform persists in the state, and it would be used in subsequent Terraform operations as the reference of the actual state of the infrastructure (for example, it stores the EC2 instance ID). Notice that Terraform assumes that the infrastructure is only managed by itself. No external interaction with the infrastructure should happen, or the state will be invalidated and state remediation will be needed.



Note

While a local state file is sufficient for individual learning or experimentation, using a remote backend—such as an AWS **Simple Storage Service (S3)** bucket or Terraform Cloud—is a crucial best practice for team-based projects. It enables safe collaboration by allowing shared access, state locking, and consistent tracking of infrastructure changes.

We now have the VM in place, so next, you will learn how to leverage Terraform outputs to feed Ansible with the necessary inventory data.

Integrating Terraform outputs with Ansible

A key distinction between Terraform and Ansible is the way they handle infrastructure references. Unlike Ansible, Terraform does not rely on an inventory because it inherently manages dynamic infrastructure. Instead of tracking hosts, Terraform only requires essential parameters such as the account, region, or VPC ID to deploy services.

Ansible, on the other hand, depends on an inventory to define the infrastructure it interacts with. In our example, after provisioning an Ubuntu server on AWS using Terraform, you need to configure it using Ansible. However, before Ansible can manage this server, you must generate an inventory that includes the necessary connection details.

This section explores various approaches to integrating Terraform outputs with an Ansible inventory, ensuring seamless automation between provisioning and configuration.

Let's start with the simplest one: building an Ansible inventory using Terraform state data.

Building an Ansible inventory from the Terraform state

From the previous chapters, you will already be familiar with the Ansible inventory format. So, here, you will create an inventory using the data that Terraform has stored in its state after the `terraform apply` execution:

1. Start by creating an `inventory.ini` file with the group name `aws-ec2`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# echo "[aws-ec2]" >
inventory.ini
```

2. Then, append in the same file the `public_ip` output that was generated by Terraform after provisioning, plus other Ansible variables that you will need to connect: `ansible_user` and `ansible_ssh_private_key_file`. The host IP is extracted from the `public_ip` output from Terraform without formatting:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# echo "$(terraform
output -raw public_ip) ansible_user=ubuntu ansible_ssh_private_key_
file=/tmp/cookbook-key.pem" >> inventory.ini
```

3. That's it. You have the inventory file you need to get started with Ansible:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# cat inventory.ini
[aws-ec2]
54.173.181.113 ansible_user=ubuntu ansible_ssh_private_key_file=/
tmp/cookbook-key.pem
```

4. Validate the inventory using the Ansible ping feature to check that the server is reachable:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# ansible all -i
inventory.ini -m ping
... output omitted ...
54.173.181.113 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.10"
    },
    "changed": false,
    "ping": "pong"
}
```

Executing an Ansible playbook from within Terraform

The second option deeply integrates Ansible into Terraform. Instead of creating an Ansible inventory after Terraform has completed, you will now call Ansible using the command-line inventory option:

1. Update the Terraform plan by editing the `main.tf` file and adding a new null resource that acts as a placeholder to execute provisioners or trigger changes without making actual infrastructure changes:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi main.tf
... omitted output ...
resource "null_resource" "ansible" { (1)
    depends_on = [aws_instance.ubuntu] (2)
```

```

provisioner "local-exec" { (3)
  command = <<EOT
    sleep 10 (4)
    ANSIBLE_HOST_KEY_CHECKING=False ansible all -I
    ${aws_instance.ubuntu.public_ip}, -u Ubuntu --private-key
    /tmp/cookbook-key.pem -m ping (5)
  EOT
}
}

```

The numbered references in the preceding code are explained as follows:

- (1) The `null_resource` variable serves as a placeholder.
 - (2) The `depends_on` variable clearly defines the dependency of this task with the EC2 instance explicitly.
 - (3) The local provisioner runs the command on the local server running Terraform.
 - (4) Add a few waiting seconds to leave time for the Ubuntu server to boot and be ready to accept SSH connections.
 - (5) Use the `-i` option with the public IP, appending a comma at the end to convert it into a list, and pass the access user and SSH key.
2. Because you added a new resource type from the null provider, Terraform has to install it:

```

root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock
  file
- Finding latest version of hashicorp/null...
- Using previously-installed hashicorp/aws v5.90.0
- Installing hashicorp/null v3.2.3...
- Installed hashicorp/null v3.2.3 (signed by HashiCorp)

```

3. Rerun `terraform apply` to see how `local-exec` runs the Ansible `ping` command against the new Ubuntu server:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform apply
... omitted output ...
null_resource.ansible: Creating...
null_resource.ansible: Provisioning with 'local-exec'...
null_resource.ansible (local-exec): Executing: ["/bin/sh" "-c" "
sleep 10\n      ANSIBLE_HOST_KEY_CHECKING=False ansible all -i
100.26.210.76, -u ubuntu --private-key /tmp/cookbook-key.pem -m
ping\n"]
null_resource.ansible: Still creating... [10s elapsed] (1)
null_resource.ansible (local-exec): 100.26.210.76 | SUCCESS => {
null_resource.ansible (local-exec):      "ansible_facts": {
null_resource.ansible (local-exec):          "discovered_interpreter_
python": "/usr/bin/python3.10"
null_resource.ansible (local-exec):      },
null_resource.ansible (local-exec):      "changed": false,
null_resource.ansible (local-exec):      "ping": "pong" (2)
null_resource.ansible (local-exec): }
null_resource.ansible: Creation complete after 14s
[id=4682896565951592942] (3)
Apply complete! Resources: 2 added, 0 changed, 1 destroyed. (4)
```

The numbered references in the preceding code are explained as follows:

- (1) Notice the sleep time that is defined in the command.
- (2) Successful connection!
- (3) The `null` resource gets a Terraform identifier in the state. This means that if you rerun `terraform apply`, it won't recreate/rerun the `null` resource, so the Ansible playbook won't be executed again. If you want to recreate it, you have to explicitly run `terraform taint` for the resource, with `terraform taint null_resource.ansible`, to mark the resource as nonexistent.
- (4) If you apply the Terraform plan after the initial execution, it understands that there is a need for a different EC2 instance.

Using an AWS EC2 Ansible dynamic inventory plugin

Similar to the Nautobot dynamic Ansible inventory covered in *Chapter 11*, there is a dynamic inventory integration for AWS EC2. Here's how you can use it:

1. Create an inventory file to configure the dynamic AWS EC2 Ansible inventory. The inventory file has to end with `aws_ec2.yml` or `aws_ec2.yaml`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi inventory_aws_
ec2.yml
---
plugin: amazon.aws.aws_ec2 (1)
regions:
  - us-east-1 (2)
```

The numbered references in the preceding code are explained as follows:

(1) Define the name of the plugin for the dynamic inventory. It is already installed in Ansible.

(2) Limit the scope of the inventory—in this case, to a specific region. There are many other filters available for you to explore at:

https://docs.ansible.com/ansible/latest/collections/amazon/aws/docsite/aws_ec2_guide.html.

2. Validate the dynamic inventory data with `ansible-inventory`:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# ansible-inventory -i
inventory_aws_ec2.yml --list
{
  "_meta": {
    "hostvars": { (1)
      "ec2-35-171-153-45.compute-1.amazonaws.com": {
        "image_id": "ami-0f9de6e2d2f067fca",
        "instance_id": "i-060b434f223c51433",
        "instance_type": "t2.micro",
        "key_name": "cookbook-key",
        "platform_details": "Linux/UNIX",
        "private_ip_address": "172.31.60.58",
        "public_ip_address": "35.171.153.45",
        ... many outputs omitted ...
      }
    }
  }
```

```
    }
  },
  "all": { (2)
    "children": [
      "ungrouped",
      "aws_ec2"
    ]
  },
  "aws_ec2": { (3)
    "hosts": [
      "ec2-35-171-153-45.compute-1.amazonaws.com"
    ]
  }
}
```

The numbered references in the preceding code are explained as follows:

- (1) The inventory comes with a lot of data related to EC2 instances. In this case, it has been shrunk to show you only a few.
- (2) An all catch-all group is defined by default.
- (3) Without an explicit grouping rule defined in the inventory, all the EC2 instances are added to the `aws_ec2` group.

To ensure that you can leverage the dynamic inventory from Ansible, run a `ping` command to use it:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# ansible all -i
inventory_aws_ec2.yml -u ubuntu --private-key /tmp/cookbook-key.pem
-m ping
... output omitted ...
ec2-35-171-153-45.compute-1.amazonaws.com | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3.10"
  },
  "changed": false,
  "ping": "pong"
}
```

The following table provides a brief comparison of the three options introduced in this section:

Option	Pros	Cons	Best Use Case
Build inventory from Terraform state	- Simple and quick - Transparent workflow	- Manual step - Doesn't scale - Lacks automation	Small, static environments or one-off provisioning
Execute Ansible from Terraform (<code>local-exec</code>)	- Fully automated pipeline - No need to manage inventory separately	- Less flexible for complex playbooks - Harder to debug - Must handle timing/boot delay	Automated end-to-end flows for simple use cases
Use AWS EC2 dynamic inventory plugin	- Scales well - Dynamic and up to date - Rich host metadata	- Requires AWS credentials and plugin setup - May need filtering/grouping logic	Dynamic, multi-host cloud environments with frequent infra changes

So far, you have explored three different options available to integrate Ansible with dynamic infrastructure created with Terraform. Next, completing the recipe, you will build an Ansible playbook that will configure the `iptables` host from a list of blacklisted IPs.

Automating firewall rule provisioning with Ansible

At this point, you are already an expert in creating Ansible playbooks. This is not the main focus of this chapter, but to illustrate an interesting use case, in this section, you will build a playbook that fetches *blacklisted IPs* and loads them in the *INPUT iptables chain* (`iptables` is an administration tool for Linux packet filtering and **network address translation (NAT)**, but these are not covered in this book).



Note

More information about `iptables` is available at: <https://linux.die.net/man/8/iptables>.

Follow the next steps:

1. Create a `pb_blacklist_ips.yml` playbook that fetches a list of IPs, parses it, and finally creates iptables rules:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi pb_blacklist_ips.
yml
---
- name: Block Blacklisted IPs in Ubuntu using iptables
  hosts: all
  become: yes
  tasks:
    - name: Fetch blacklisted IPs from a public URL
      ansible.builtin.uri: (1)
      url: "https://www.spamhaus.org/drop/drop.txt"
      method: GET
      return_content: yes
      register: blacklist_response
    - name: Parse the list of IPs
      ansible.builtin.set_fact:
        blacklisted_ips: "{{ blacklist_response.content |
          regex_findall('^\\d+\\.\\.\\d+\\.\\.\\d+\\.\\.\\d+(?:/\\d+)?',
            multiline=True) }}" (2)
    - name: Block each blacklisted IP using iptables
      ansible.builtin.command: (3)
      cmd: "iptables -A INPUT -s {{ item }} -j DROP"
      loop: "{{ blacklisted_ips }}"
      ignore_errors: yes
    - name: Save iptables rules
      ansible.builtin.command:
        cmd: "iptables-save > /etc/iptables/rules.v4"
```

The numbered references in the preceding code are explained as follows:

- (1) The Ansible `uri` collection allows interaction with web services.
- (2) From the original list, extract the IPv4 addresses using a regex.
- (3) Use the `iptables` tool to register a drop rule for each IP.

2. Before changing anything, check that the Ubuntu server has no iptables rules in the INPUT chain:

```
$ ssh ubuntu@35.171.153.45 -i /tmp/cookbook-key.pem
... omitted output ...
ubuntu@ip-172-31-60-58:~$ sudo iptables -L INPUT -v -n
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target      prot opt in      out     source destination
```

At this point, you have different ways to use this playbook, as covered in the previous section. For example, you could leverage the AWS EC2 dynamic inventory and run it with `ansible-playbook -i inventory_aws_ec2.yml -u ubuntu --private-key /tmp/cookbook-key.pem pb_blacklist_ips.yml`.

The purpose of having an Ansible playbook in this scenario is to run it repeatedly after the initial instance provisioning to continuously update the list of blacklisted IPs. However, for the initial setup, it is more efficient to apply the blacklist as soon as the instance is created, ensuring it starts with the latest updates. This highlights a key difference between Ansible and Terraform—while Terraform provisions the infrastructure, Ansible excels at ongoing configuration management and updates.

3. Update the `main.tf` file with a null resource, like you did for the Ansible ping feature, but now, execute the `pb_blacklist_ips.yml` playbook:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# vi main.tf
... omitted output ...
resource "null_resource" "blacklist_ips" {
  depends_on = [aws_instance.ubuntu]
  provisioner "local-exec" {
    command = <<EOT
      ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i
      ${aws_instance.ubuntu.public_ip}, -u ubuntu --private-key
      /tmp/cookbook-key.pem pb_blacklist_ips.yml
    EOT
  }
}
```

4. Run `terraform apply` to run the new null resource that triggers the Ansible playbook (it takes a while; it's a long list):

```

root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform apply
... omitted output ...
null_resource.blacklist_ips: Creating...
null_resource.blacklist_ips: Provisioning with 'local-exec'...
null_resource.blacklist_ips (local-exec): Executing: ["/bin/sh" "-c"
"ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -i 100.26.210.76,
-u ubuntu --private-key /tmp/cookbook-key.pem pb_blacklist_ips.
yaml\n"]
null_resource.blacklist_ips (local-exec): PLAY [Block Blacklisted
IPs in Ubuntu using iptables] ***** (1)
null_resource.blacklist_ips (local-exec): TASK [Gathering Facts] ***
null_resource.blacklist_ips (local-exec): ok: [100.26.210.76]
null_resource.blacklist_ips (local-exec): TASK [Fetch blacklisted
IPs from a public URL] *****
null_resource.blacklist_ips (local-exec): ok: [100.26.210.76]
null_resource.blacklist_ips (local-exec): TASK [Parse the list of
IPs] *****
null_resource.blacklist_ips (local-exec): ok: [100.26.210.76]
null_resource.blacklist_ips (local-exec): TASK [Block each
blacklisted IP using iptables] *****
null_resource.blacklist_ips: Still creating... [10m30s elapsed]
... omitted output ...
null_resource.blacklist_ips (local-exec): changed: [35.171.153.45]
=> (item=223.254.0.0/16) (2)
null_resource.blacklist_ips (local-exec): PLAY RECAP *****
null_resource.blacklist_ips (local-exec): 35.171.153.45
: ok=4    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
null_resource.blacklist_ips: Creation complete after 10m33s
[id=7216867022650808652]

```

The numbered references in the preceding code are explained as follows:

- (1) It is just an Ansible playbook executed from the Terraform trigger. So, this is output you are already familiar with.
 - (2) A long list of changes for each public IPv4 address blocked.
5. Now, if you check the iptables INPUT chain again, the new filter rules show up:

```
ubuntu@ip-172-31-60-58:~$ sudo iptables -L INPUT -v -n
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target      prot opt in      out     source
destination
    0      0 DROP        all  --  *      *       1.10.16.0/20
0.0.0.0/0
    0      0 DROP        all  --  *      *       1.19.0.0/16
0.0.0.0/0
... omitted output ...
```

Running Ansible on the same server where you run Terraform is just one option. In *Chapter 13*, you learned that Ansible can offer a more sophisticated experience using AWX. In the next section, you will integrate Terraform directly with AWX.

Integrating Terraform with AWX

If you have not already covered *Chapter 13*, now is the time to do so. For this recipe, you need an AWX server up and running as explained there.

On top of the previous AWX chapter, let's configure AWX to implement a recipe to automate firewall rule provisioning integrating with Terraform:

1. Configure the AWS credentials, the access key ID, and secret in AWX by going to **Access Management | Credentials | Create Credential**.

← → ↻ Net Secure cookbook-lab:8383/ui_next/credentials/create

AWX

Templates

Schedules

Projects

Infrastructure

Analytics

Administration

Access Management

Authentication

Organizations

Teams

Users

Roles

Credentials

Credentials > Create Credential

Create Credential

Name * AWS Credentials

Description Enter description

Organization Network

Credential type * Amazon Web Services

Type Details

Access Key * your AWS access key id

Secret Key *

STS Token Enter value

Create credential Cancel

2. Create a new inventory in **Infrastructure** | **Inventories** | **Create Inventory**.

← → ↻ Not Secure cookbook-lab:8383/ui_next/infrastructure/inventories/inventory/create

AWX

Overview

Jobs

Templates

Schedules

Projects

Infrastructure

Topology View

Inventories

Hosts

Instance Groups

Inventories > Create Inventory

Create Inventory

Name * AWS EC2 Inventory

Description Enter description

Organization * Network

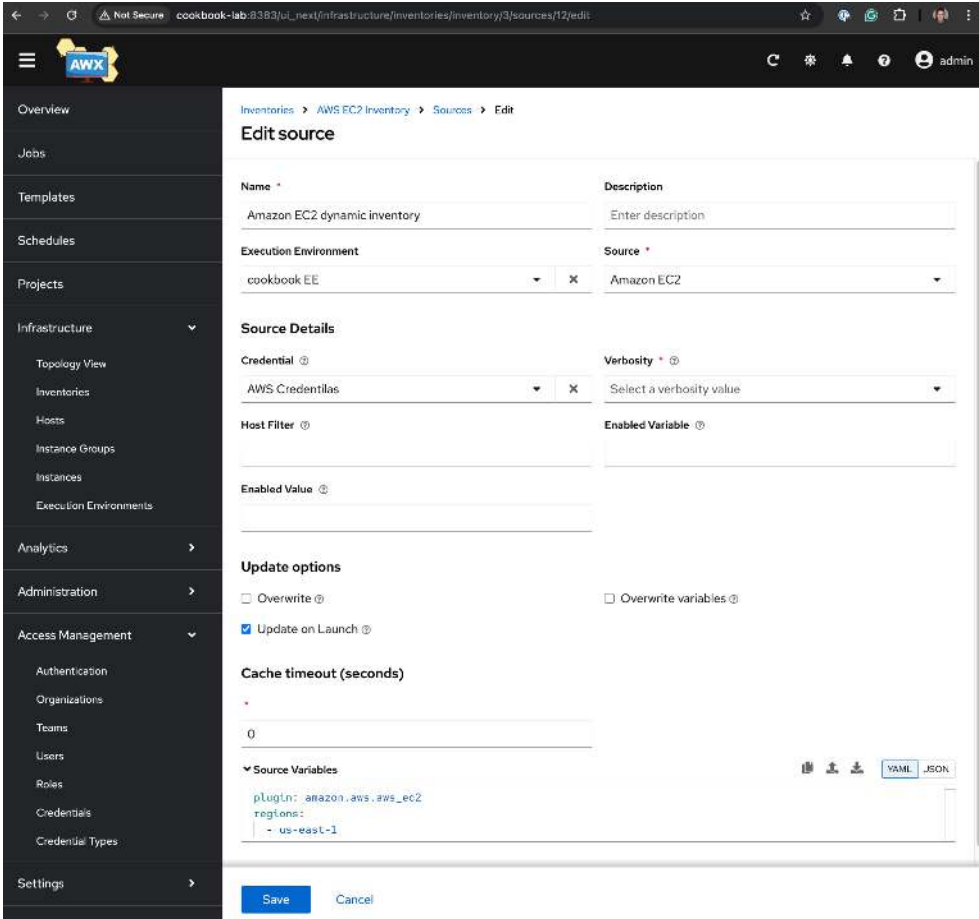
Instance Groups Add instance groups

Labels Select or create labels

Variables Variables

Create inventory Cancel

3. Add a new source that uses the Amazon EC2 plugin inventory (the same as you used before) and leverages the AWS credentials to connect. Also, check the **Update on Launch** option to refresh it every time it is triggered. Finally, in the **Source Variables** section, add the same inventory configuration that narrows the scope.



AWX needs to import playbooks via projects, as explained in *Chapter 13*. Using the same steps as before, you will create a new Git repository and define a playbook there:

4. Create a GitHub repository to contain the project playbooks.

github.com/new

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Repository template

No template

Start your repository with a template repository's contents.

Owner * chadell / **Repository name *** awx_blacklist

awx_blacklist is available.

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

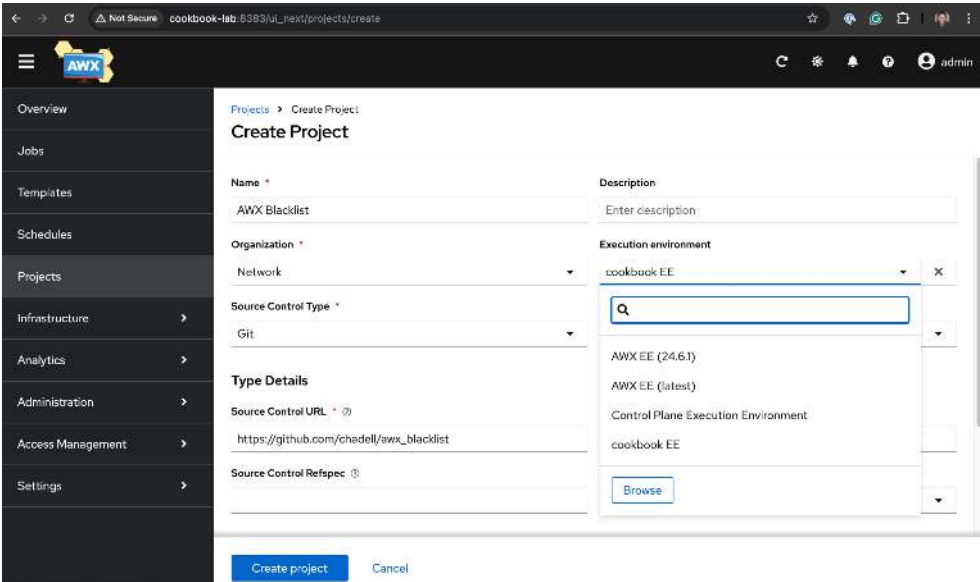
Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

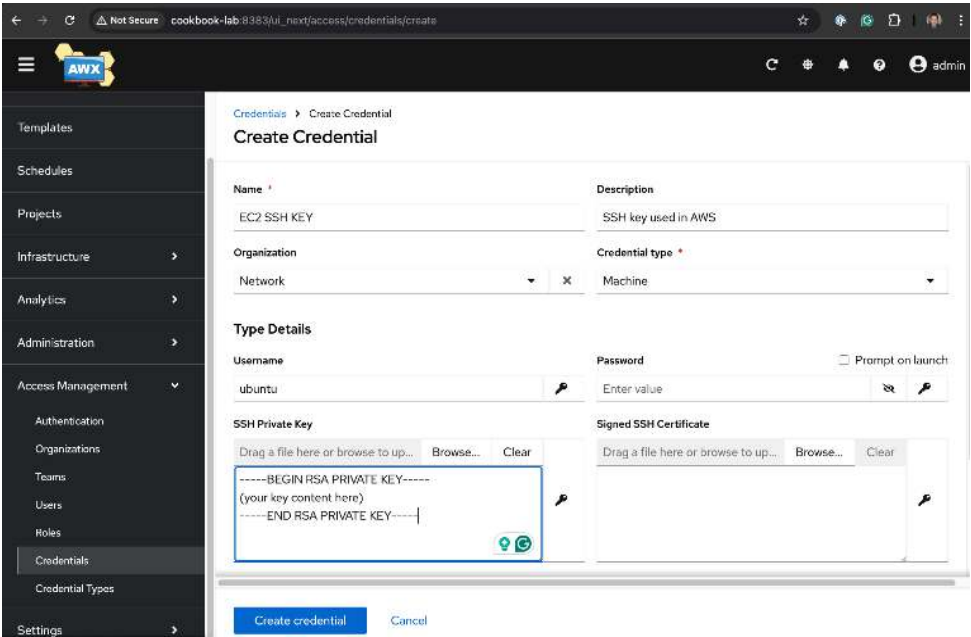
5. Create a new local directory that contains the same pb_blacklist_ips.yml file, and run the suggested commands from GitHub to push them to the remote repository:

```
root@cookbook-lab:~/awx_blacklist# git init
... omitted output ...
Initialized empty Git repository in /root/awx_blacklist/.git/
root@cookbook-lab:~/awx_blacklist# git add .
root@cookbook-lab:~/awx_blacklist# git commit -m "Initial commit"
[master (root-commit) b278df6] Initial commit
... omitted output ...
root@cookbook-lab:~/awx_blacklist# git branch -M main
root@cookbook-lab:~/awx_blacklist# git remote add origin git@github.
com:chadell/awx_blacklist.git
root@cookbook-lab:~/awx_blacklist# git push -u origin main
```

- 6. Create an AWX project that uses the public repository created, and define an execution environment already defined in *Chapter 13*, cookbook EE.



- 7. Upload the SSH key to AWX, creating a new credential of type **Machine**. Paste the content of your SSH private key file (e.g., `cookbook-key.pem`) in the **SSH Private Key** section.



8. Create a job that uses the new AWS EC2 inventory and uses the `pb_blacklist_ips.yml` file that has been imported via Git from the project.

After the initial AWX setup, it's time to create a new Terraform plan that connects with AWX. To simplify it, instead of reusing the previous `main.tf` file, you will use a new one that shares the EC2 instance components but includes the AWX integration:



Note

Terraform aggregates all files in the directory that have the `.tf` extension. If you have the previous `main.tf` file in the same folder, you can easily disable it by adding a suffix of your choice, such as `.old`.

1. Create a new Terraform file, `awx.tf`, that uses a community provider (not officially supported) that offers different resources to interact with AWX:

```
terraform {
  required_providers {
    awx = {
      source = "denouche/awx" (1)
    }
  }
}
```

```
        version = "0.29.1"
      }
    }
  }
  provider "awx" { (2)
    hostname = "http://cookbook-lab:8383"
    username = "admin"
    password = "your AWX secret"
  }
  ... omitted output ... (3)
  resource "awx_job_template_launch" "blacklist_ips" {
    depends_on = [aws_instance.ubuntu] (4)
    job_template_id = 14 (5)
    limit = aws_instance.ubuntu.public_dns (6)
  }
```

The numbered references in the preceding code are explained as follows:

- (1) The non-official Terraform provider, such as denouche/awx (more information is available at <https://registry.terraform.io/providers/denouche/awx/latest/docs>), has to be explicitly defined in the `required_providers` variable.
- (2) Similar to the AWS provider configuration, the AWX provider requires its own configuration—in this case, the AWX URL and the credentials.
- (3) Here, all the VPC, subnet, AMI, and EC2 instance configurations from before are used (no changes).
- (4) As with the `null` resource used before, you have to tell Terraform to wait until the EC2 instance is created to trigger the AWX job.
- (5) As explained in the API AWX integration in *Chapter 13*, each job or workflow in AWX gets an identifier that you specify here. You can check it via the API or through console access. Replace with your actual job template ID.
- (6) The `limit` parameters allow you to specify which one of the EC2 instances in the AWS inventory you want to execute the job playbook against.

2. Because a new provider has been added to the plan, reinitialize Terraform:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform init
Initializing the backend...
Initializing provider plugins...
- Finding denouche/awx versions matching "0.29.1"...
- Reusing previous version of hashicorp/aws from the dependency lock
  file
- Reusing previous version of hashicorp/null from the dependency
  lock file
- Installing denouche/awx v0.29.1...
- Installed denouche/awx v0.29.1 (self-signed, key ID
  026C2C38EA7B6CD5)
- Using previously-installed hashicorp/aws v5.90.0
- Using previously-installed hashicorp/null v3.2.3
Partner and community providers are signed by their developers. (1)
... omitted output ...
Terraform has been successfully initialized!
```

The numbered reference in the preceding code is explained as follows:

(1) Because it is a community provider, this is a warning explaining that it is outputted for informational purposes.

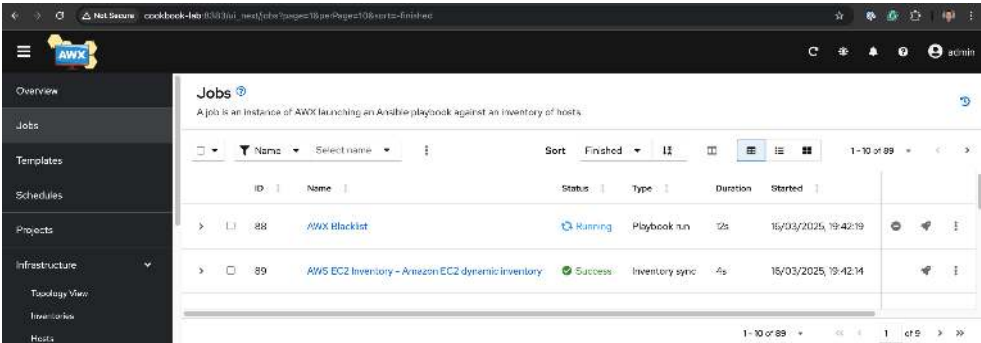
3. Finally, run Terraform to provision the new configuration plan. If you run it from the same folder as you ran the previous provisioning, Terraform uses the same `terraform.tfstate` file, so it will remove the previous resources (e.g., the EC2 instance and null resources) as they are no longer referenced here:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform apply
... omitted output ...
awx_job_template_launch.blacklist_ips: Creating...
awx_job_template_launch.blacklist_ips: Creation complete after 2s
[id=62] (1)
Apply complete! Resources: 2 added, 0 changed, 3 destroyed.
```

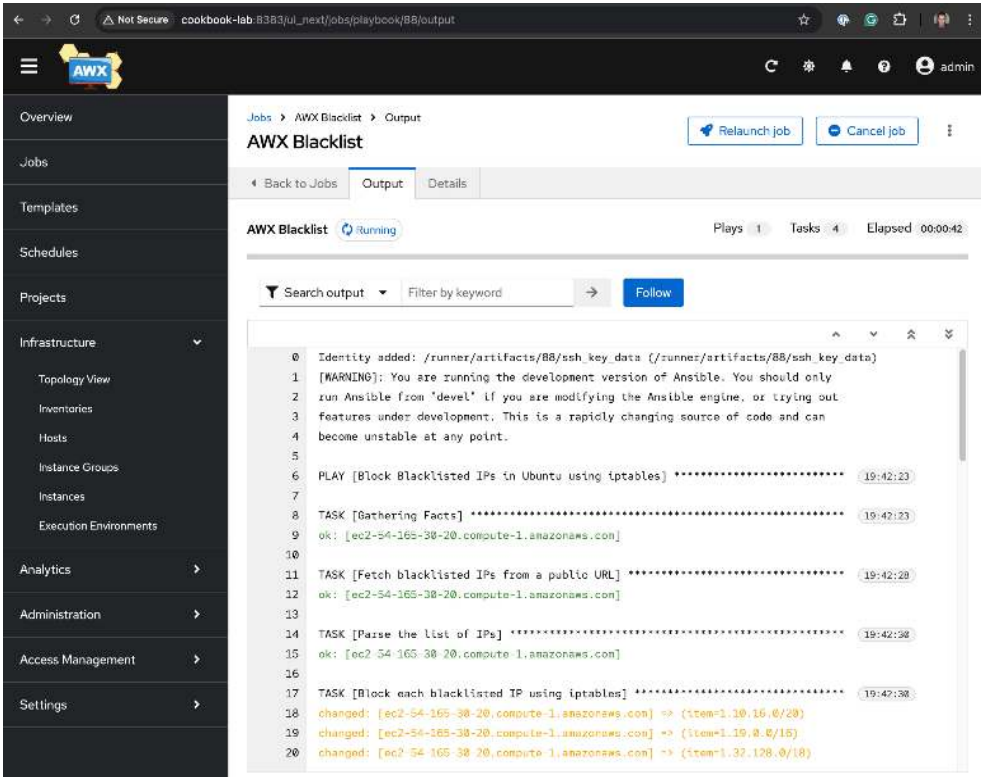
The numbered reference in the preceding code is explained as follows:

(1) Notice that triggering the AWX job template is almost immediate. When running the Ansible playbook as `local-exec`, Terraform waited until completion, but now it only triggered the playbook in the remote AWX instance.

- 4. If you change back to the AWX console, you will see how two new jobs were triggered. The AWS EC2 inventory was refreshed (as defined) to take the recently created EC2 instance, and now the AWX blacklist is in **Running** state.



- 5. Go to **Jobs | AWS Blacklist | Output**, and you can see the Ansible playbook output and see how each IP is changed in the remote server.



After a few minutes, the AWX job will be completed, and all provisioning will be concluded. This leaves the Ubuntu server ready to use but also opens the door for scheduling regular triggers of the AWX job to refresh newly added IP addresses in the blacklist.

Finally, following the best practices in dynamic infrastructure, it's time to clean up the created infrastructure and avoid incurring undesired costs.

Cleaning up dynamic infrastructure with Terraform

Terraform provides an extremely simple method for de-provisioning the infrastructure managed by it. All the resources included in the Terraform state will apply the decommissioning instructions to be deleted simply by running `terraform destroy`:

Run `terraform destroy` and see how all the infrastructure is removed:

```
root@cookbook-lab:~/cookbook-lab/chapters/ch15# terraform destroy
... omitted output including the destroy plan to confirm ...
awx_job_template_launch.blacklist_ips: Destroying... [id=62]
awx_job_template_launch.blacklist_ips: Destruction complete after 1s
aws_instance.ubuntu: Destroying... [id=i-00edf837402e880fb]
aws_instance.ubuntu: Still destroying... [id=i-00edf837402e880fb,
10s elapsed]
aws_instance.ubuntu: Still destroying... [id=i-00edf837402e880fb,
20s elapsed]
aws_instance.ubuntu: Still destroying... [id=i-00edf837402e880fb,
30s elapsed]
aws_instance.ubuntu: Destruction complete after 30s
aws_security_group.ssh_access: Destroying... [id=sg-
0dffaa48a4c6c2414]
aws_security_group.ssh_access: Destruction complete after 1s
Destroy complete! Resources: 3 destroyed.
```

Notice how the order of the destroy actions is the *inverse* of the defined order of creation. You don't need to worry about sequencing the operation; the `depends_on` and implicit dependencies make this a very simple process.

You have seen how these two popular tools, Terraform and Ansible, provide unique functionalities that allow them to dance together.

Summary

Terraform and Ansible take different yet complementary approaches to infrastructure management. Terraform excels in provisioning dynamic infrastructure using a declarative model, where the desired end state is defined, and determines how to achieve it. Terraform's built-in state management ensures consistency and simplifies updates, while its immutable infrastructure philosophy helps prevent configuration drift. Ansible, on the other hand, follows a procedural approach (with some declarative support), executing sequential tasks for configuration management and automation. While both tools can achieve similar results, Terraform is often preferred for cloud resource provisioning, while Ansible is widely used for post-deployment configuration and ongoing management.

By integrating Terraform and Ansible, organizations can optimize their infrastructure workflows. Terraform can provision cloud-based resources such as virtual firewalls or VPN connections, and Ansible can dynamically apply configuration updates based on Terraform's output. This hybrid approach leverages Terraform's strengths in infrastructure lifecycle management while utilizing Ansible's flexibility for operational automation. This chapter explored practical use cases demonstrating this synergy, including dynamic infrastructure deployment, Terraform output integration with Ansible, firewall rule automation, and infrastructure cleanup.

As you reach the final pages of this book, we hope you have enjoyed this practical journey through Ansible and its role in network automation. Throughout these chapters, you explored how Ansible simplifies automation across diverse environments, from data center and service provider networks to multi-cloud deployments. You learned fundamental building blocks, advanced concepts, and real-world implementations with vendors such as Arista, Juniper, and Nokia, as well as how to leverage NAPALM for multi-vendor automation.

Beyond device configuration, we tackled network validation with Batfish, built dynamic inventories with Nautobot, and integrated AWX to streamline workflows. We also explored event-driven automation and examined how Terraform and Ansible can work together for infrastructure provisioning. Whether you're just starting with Ansible or refining an existing automation strategy, we hope this book has equipped you with the knowledge and confidence to build more efficient, scalable, and resilient network automation solutions.

Unlock this book's exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



16

Appendix - The Lab

This lab serves as a controlled hands-on environment to facilitate a consistent reproduction of all the steps and recipes included in the book. We believe that there is no better way to absorb knowledge than doing it, so we have prioritized making this setup efficient rather than complicating scenarios that are not accessible to everyone. Obviously, you can take the same ideas into any kind of network environment you want to automate.

Because we want to give you a complete experience, for Ansible-related management (e.g., install Ansible), we just provide you with a ready-to-go operation system where you can take full responsibility for installing it in the environment.

Thus, the scenario is composed of the following main components:

- **Ubuntu 22.04 Linux server:** This is the operating system where you will run all your Ansible playbooks. (You could use a different OS, but it could come with slightly different behavior).
- **Virtualized network lab with Containerlab:**
- **Nautobot:** A Source of Truth (SoT) that holds our intended network infrastructure data and is used to enrich observability data. Although not the primary focus of the book, Nautobot is featured as an SoT in multiple chapters. For more insights into its role and capabilities, refer to the official Nautobot documentation (<https://nautobot.readthedocs.io/en/latest/>).

In this chapter, we explain more details about the scenario, and how to get it up and running both manually (i.e., step-by-step) or automatically (i.e., using Infrastructure as Code approach).

The lab environment setup

Setting up a lab environment is a powerful way to dive into real-world scenarios without the risk of impacting live systems. It allows you to confidently experiment with Ansible playbooks, troubleshoot, and explore various configurations, all while keeping production systems safe.

This lab is not just a recommendation—it's essential for fully grasping the concepts covered in this book and testing the provided code examples. The environment consists of a single Ubuntu Linux VM, enhanced by Containerlab (<https://containerlab.dev/>) for building container-based network topologies and Docker (<https://www.docker.com/>) for managing additional components.

The diagram below gives you a clear view of the lab setup and how everything fits together. Let's get started on your journey to mastering these skills!

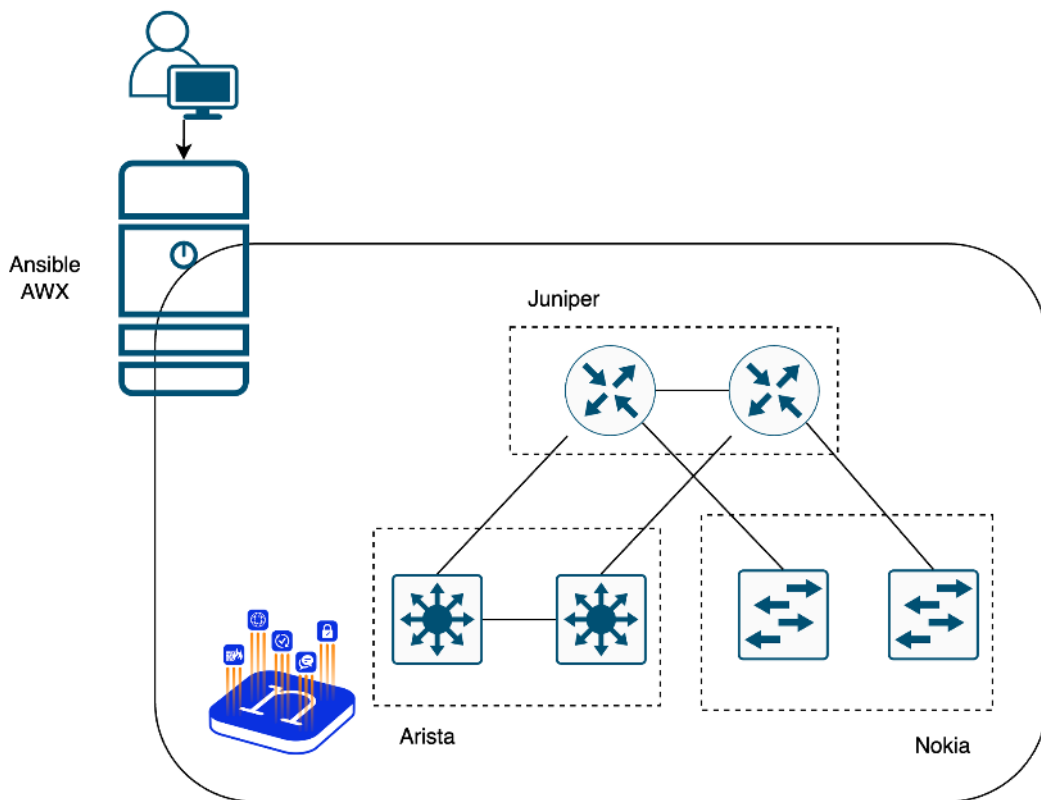


Figure 16.1 - Lab Environment Setup



 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader and a free PDF/ePub copy** of this book are included with your purchase. Scan the QR code OR go to packtpub.com/unlock, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.



Our mission is to offer a reliable, hassle-free way to set up your lab environment. To ensure that the process is both repeatable and tested, we've chosen DigitalOcean as our cloud provider. However, flexibility is key, and you're not limited to this option.

We've provided two detailed guides to help you get started:

- **Manual VM Provisioning:** A step-by-step guide that walks you through the process of setting up the lab environment by hand.
- **Automated VM Provisioning:** An efficient, hands-free setup using Ansible (nice coincidence), streamlining the entire process for you.

While these guides are tailored for **DigitalOcean**, feel free to use your own setup—whether it's on Windows (via WSL2), MacOS, or a Linux-based system like Ubuntu. You're good to go as long as your environment meets the software requirements. Our goal is to give you the flexibility to build an effective lab, regardless of the platform, so you can focus on learning and exploring with ease.

Hardware Requirements

For the best experience running your lab environment, we recommend setting up your virtual machine (VM) on a 64-bit x86 CPU architecture. Ideally, your VM should be equipped with **4 vCPUs and 16 GB of RAM**. While these specs will get you up and running, increasing the resources will noticeably boost performance and speed, making your lab even more efficient.

Whether you’re using a cloud-based VM, such as from Amazon AWS or DigitalOcean, or a self-hosted solution like VMware, be sure that your VM meets or exceeds these specifications for optimal results. More resources mean faster operations and a smoother overall experience.

Software Requirements

Your lab environment VM will leverage Containerlab to create container-based network topologies and Docker to natively construct and manage the lab environment. All the essential configurations, along with step-by-step deployment instructions for both the network topology and the other components, are conveniently stored in our GitHub repository: `cookbook-lab` (https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition).

This repository also includes the `labcli` utility, a tool designed to simplify the deployment and management of your lab environment, ensuring a smooth setup process.

Below is a list of the core dependencies required to run the lab environment on your VM. Make sure these are in place to ensure everything runs smoothly.

Package	Version
containerlab	0.54.2
docker	26.1.1
Python (for the labcli utility)	3.10

Table 16.1 - Main Software Dependencies Versions



Note

The versions listed above are accurate as of this writing. For the most up-to-date information on dependencies, please refer to the GitHub repository.

Additionally, the lab environment requires the container images for the components used in the network and the rest of the stack. Here is the reference for the core components found in the lab scenarios:

Service	Image	Version
Arista cEOS	cEOS	4.31.2F
Juniper cRPD	cRPD	24.2R1.14

Service	Image	Version
Nokia SR Linux	ghcr.io/nokia/srlinux	24.7.2
Nautobot	docker.io/networktocode/nautobot	2.3-py3.10

Table 16.2 - Main Container Dependencies Versions

The ceos image is downloaded from the Arista website and imported into Docker (more on this later in Build Lab Environment).

In the following sections, you'll find a detailed, easy-to-follow guide that walks you through every step needed to deploy and manage your lab environment.

Step 0. Forking the Lab Environment from Git Repository

All the essential tools, configurations, and code examples for the lab environment are available in the network automation cookbook lab GitHub repository (https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition). To begin your journey, simply fork the repository to your own GitHub account. This will allow you to easily customize and experiment with the lab environment as you follow along.

Why Fork?

Forking a repository gives you the freedom to experiment and make changes without affecting the original project. This is incredibly useful for practicing with the code, as you can modify it as much as needed without worrying about impacting the main branch.

When you fork the repository, it creates a copy under your own GitHub account, giving you an independent version to work with. This means you can freely test out code examples, make changes, and even push those changes to your fork to save them. Plus, if you come up with improvements, you can submit pull requests to the original repository and contribute to the project!

If you don't have a GitHub account, no worries—you can still clone the repository directly from the original source. However, keep in mind that any changes you make to the repository won't be saved. Cloning without forking gives you access to the code, but your modifications won't persist once you close or leave the environment.

To fork the repository, you'll first need a GitHub account. If you don't have one yet, you can create it for free at [github.com](https://github.com/join) (<https://github.com/join>).

Once you’re ready, head over to the Cookbook Lab repository (https://github.com/PacktPublishing/Network-Automation-Cookbook_2nd_Edition) and click the **Fork** button in the top right corner. Select your GitHub account as the destination for the fork, and voilà—you now have your own copy to work with!

For more details on how to manage forks, check out GitHub’s official documentation(<https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo>).

After forking, you can clone the repository to your local or lab machine. From there, you’ll be ready to follow the next steps, which cover building your lab machine and deploying the lab environment.

Step 1. Building the Lab Environment

In the next sections, we guide you through building the lab environment using a cloud provider. This method minimizes external dependencies, making it a smooth, hassle-free option. As the authors, we recommend this approach because it’s been rigorously tested, ensuring a reliable and consistent experience when running the examples from the book.

Best of all, there’s no need to pay to get started! You can take advantage of free trials from cloud providers like DigitalOcean or Amazon AWS. Alternatively, you can use a self-hosted solution like VMware, VirtualBox, or even run it natively on your system (e.g., Windows WSL2 or Linux—just note that Docker and Containerlab installations may differ slightly for these platforms).

If you’re going the self-hosted route, check out the table below for links to the Docker and Containerlab installation instructions specific to different operating systems.

Operating System	docker install	containerlab install
Windows WSL2	Docker Desktop on Windows (https://docs.docker.com/desktop/windows/install/)	Containerlab on Windows WSL (https://containerlab.dev/install/#windows-subsystem-linux-wsl)
MacOS	Docker Desktop on Mac (https://docs.docker.com/desktop/mac/install/)	Containerlab on MacOS (https://containerlab.dev/install/#apple-macos)
Ubuntu	Docker Engine on Ubuntu (https://docs.docker.com/engine/install/ubuntu/)	Containerlab with apt package manager (https://containerlab.dev/install/#package-managers)

Table 16.3 - Operating Systems Software Installation Guides

Note: Some container images may not be available for the new ARM Apple platform. All the book examples have been conducted directly over Ubuntu.

As we mentioned above, you explain first the step-by-step installation, and later, we will explain how to perform the same installation in an automated way.

Manual Server Provisioning

The first step in setting up your lab environment is provisioning a VM to host it. In this guide, we'll use DigitalOcean as our cloud provider. Please note that charges may apply, so be sure to keep an eye on your usage.

While we're focusing on DigitalOcean, you can also choose other cloud providers, such as Amazon AWS. Just keep in mind that the steps for provisioning the VM may vary slightly depending on the provider you select. Also, you can skip the first step if you use your local machine.

Create a VM in Digital Ocean

The following steps will guide you to create a VM in DigitalOcean

1. Create a Digital Ocean account
 - Please visit the DigitalOcean website to create an account if you do not already possess one. At the time of this writing, new users can take advantage of a promotional offer, which provides \$200 in credit over 60 days. You may sign up using the following link (<https://cloud.digitalocean.com/registrations/new>).
2. Create an SSH key pair and upload the public key to Digital Ocean.
 - For example on Linux or MacOS, you can use the following command to create an SSH key pair:

```
$ ssh-keygen -t rsa -b 4096 -C "cookbook-lab" -f ~/.ssh/  
id_rsa_do
```

This process will generate an SSH key pair in the `~/.ssh` directory, named `id_rsa_do` for the private key and `id_rsa_do.pub` for the public key. Once created, you can copy and upload the public key to your DigitalOcean account.

Next, navigate to the control panel of your DigitalOcean account. Click on the "Settings" menu, then select "Security." From there, click on "Add SSH Key" and paste the contents of the public key file into the provided field. The name field is optional; however, you may designate it as `cookbook-lab` for clarity. Finally, click on "Add SSH Key" to complete the process.

For further instructions and alternative methods, you may refer to this link (<https://docs.digitalocean.com/products/droplets/how-to/add-ssh-keys/to-account/>)

3. Create a droplet in Digital Ocean.

- You can create a droplet or VM using the DigitalOcean web interface or the command line. You can use this link (<https://docs.digitalocean.com/products/droplets/how-to/create/>) to create a droplet using the web interface.
- When creating your droplet, select the following options:
 - Distribution: Ubuntu 22.04
 - Plan: Standard, 4 vCPUs, 16 GB RAM, 160 GB SSD
 - Datacenter Region: New York 1 (Or closest to your location)
 - Authentication: SSH keys
 - SSH keys: Select the SSH key you created in the previous step
 - Create: Give the Droplet a name and create it

You can monitor the progress of your droplet being built on the DigitalOcean interface. Once the process is complete, you can SSH to it to verify it's ready to go.

```
$ ssh root@<droplet-ip-address> -i ~/.ssh/id_rsa_do
```

For more information on connecting with SSH to the VM/Droplet, you can refer to their official documentation: <https://docs.digitalocean.com/products/droplets/how-to/connect-with-ssh/>

Install Docker and Containerlab

Now we are ready to install the main dependencies for this project: Docker and Containerlab.

1. First, install docker using the following commands:

```
root@cookbook-lab:~# apt-get update -y
root@cookbook-lab:~# curl -fsSL https://get.docker.com -o get-docker.sh
root@cookbook-lab:~# sh get-docker.sh
```

2. Then, install containerlab using the following commands:

```
root@cookbook-lab:~# bash -c "$(curl -sL https://get.containerlab.dev)"
```

For more detailed instructions, visit the official Containerlab installation guide: <https://containerlab.dev/install/>

Now, let's dive in and clone your forked lab environment repository! This is your main hub for all the hands-on exercises in the book. Once you've cloned it, you'll have everything you need right at your fingertips to set up and run the observability tools in your lab. It's your key to unlocking the full experience of the book, so let's get started!

Clone the Git Repository

To set up the lab environment, first, you have to clone the repository from your fork and navigate to the directory:

```
# Clone the repository, replacing <your-github-username> with your GitHub
username.
root@cookbook-lab:~# git clone https://github.com/<your-github-username>/
Network-Automation-Cookbook_2nd_Edition.git
# After cloning the repository, navigate to the directory
root@cookbook-lab:~# cd cookbook-lab/
root@cookbook-lab:~/cookbook-lab#
```

You've made it to the root directory of your cloned repository. This is where the magic happens. It's your main reference point for all the files and configurations throughout the lab exercises.

Next up, you'll need to copy the example environment file (`example.env`) and modify it into a `.env` file. This step is essential for running the lab scenarios. The `.env` file holds key variables used by the components deployed in each lab, along with important settings for the `lacli` utility that help fine-tune its operation. For example:

```
root@cookbook-lab:~/cookbook-lab# cp example.env .env
root@cookbook-lab:~/cookbook-lab# vim .env
```

The `.env` file holds variables like the following:

```
#####
# LabCLI utility
#####
LAB_SCENARIO=batteries-included
LAB_SUDO=false
...
```

It is encouraged to modify the contents of the copied `.env` file to fit your setup, but it is not mandatory. Next, download the network device images and prepare them for Containerlab's usage.

Download and import the network device images

The lab environment uses Arista cEOS, Juniper cRPD, and Nokia SR Linux container images. The Nokia SR Linux is publicly available in a container registry, but the other two require free registration before you can download them from the vendors' websites.

- For Arista cEOS image, go to this link: <https://www.arista.com/en/support/software-download>, and search and download the image under **cEOS-lab > EOS-<version> > cEOS64-lab-<version>.tar.xz**
- For Juniper cRPD image, go to this link: <https://support.juniper.net/support/downloads/>, and search for **cRPD**.

If you have downloaded the images in a different host, you can copy the images to the lab server using scp:

```
# Import cEOS images as Docker images
$ scp -i ~/.ssh/id_rsa \
    ~/Downloads/cEOS-lab-4.30.1F.tar \
    root@<you-server-ip>:/tmp/cEOS-lab-4.30.1F.tar
$ scp -i ~/.ssh/id_rsa \
    ~/Downloads/junos-routing-crpd-docker-amd64-24.2R1.14.tgz \
    root@<your-server-ip>:/tmp/junos-routing-crpd-docker-amd64-
    24.2R1.14.tgz
```

So, at this point, we have both images in the /tmp directory:



If you run into issues downloading any book's repository for the latest instructions of the vendor Docker images, check the *Notes* — they may change over time.

```
root@cookbook-lab:~/cookbook-lab# ls /tmp/
cEOS-lab-4.30.1F.tar
junos-routing-crpd-docker-amd64-24.2R1.14.tgz
```

The next step is to import/load the images as Docker images (it depends on how the image was created):

```
root@cookbook-lab:~# docker import /tmp/cEOS-lab-4.30.1F.tar ceos:image
root@cookbook-lab:~# docker load -i /tmp/junos-routing-crpd-docker-amd64-
    24.2R1.14.tgz
```

The `ceos:image` and `crpd:24.2R1.14` tags (the Juniper is taken from the load) are used in the lab's default configuration.



Note: In the Automated VM Provisioning, the image upload and import are taken care by the `labcli` tool.

In the lab repository, in the `containerlab/` folder, you can find the `lab.yml` file that defines the network topology and uses these Docker images to create the scenario.

Install labcli utility

The `labcli` (<https://github.com/davidban77/labcli>) is a Python utility that offers a simple command line interface to manage the lab scenario.



Note

For the best experience, we recommend installing this in a Python virtual environment, especially if you're setting it up on your local machine. Virtual environments help keep your project dependencies isolated and clean. You can check out the official Python documentation for step-by-step guidance on creating and managing virtual environments at <https://docs.python.org/3/library/venv.html>.

If you prefer, you can also use powerful third-party tools like **conda** from the Anaconda team. It's a great alternative for managing environments, and you can explore more about it at <https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>.

To install `labcli`, just run the following within the repository folder:

```
# If pip is not installed, first install it
root@cookbook-lab:~/cookbook-lab# apt install python3-pip
root@cookbook-lab:~/cookbook-lab# pip install --upgrade pip setuptools
wheel
root@cookbook-lab:~/cookbook-lab# pip install labcli
... omitted output ...
Installing collected packages: labcli
Successfully installed labcli-0.2.0
```


So, you can now interact with the network device from your server via CLI (or other methods):

```
root@cookbook-lab:~# docker exec -it 03fd541602cf cli
root@crpd-02> show configuration
## Last commit: 2024-10-20 14:42:54 UTC by root
version 20240621.103832_builder.r1429411;
system {
    root-authentication {
... omitted output ...
}

root@cookbook-lab:~# docker exec -it 39d74581f7a3 Cli
ceos-02>en
ceos-02#sh run
! Command: show running-config
! device: ceos-02 (cEOSLab, EOS-4.30.1F-32315456.4301F (engineering
build))
!
no aaa root
... omitted output ...
```



Note

For more details about each OS, check it at the Containerlab Kinds page: <https://containerlab.dev/manual/kinds/>.

However, the network scenario is not the only service that the tool has started. Other services used in the book, are also ready to go.

Prepare Nautobot

Nautobot is used throughout the book, and this section will guide you on how to populate Nautobot with data from the network topology used in Containerlab.

1. **Verify Nautobot Service:** Ensure the Nautobot service is up and running by accessing its web interface at `http://<server-ip-address>:8080`. Use the credentials provided in your `.env` file (`NAUTOBOT_SUPERUSER_NAME` and `NAUTOBOT_SUPERUSER_PASSWORD`).

2. Check Nautobot Initialization: If unable to connect, Nautobot might still be initializing. Check its status and logs using `labcli docker logs --services nautobot -t 50 -f` command. Initialization logs should show messages like Applying migration... and will complete with Nautobot initialized!.
3. Empty Nautobot Instance: Once initialized, Nautobot should be running but will be empty of any data. The Nautobot Home Page should be similar to the following figure.

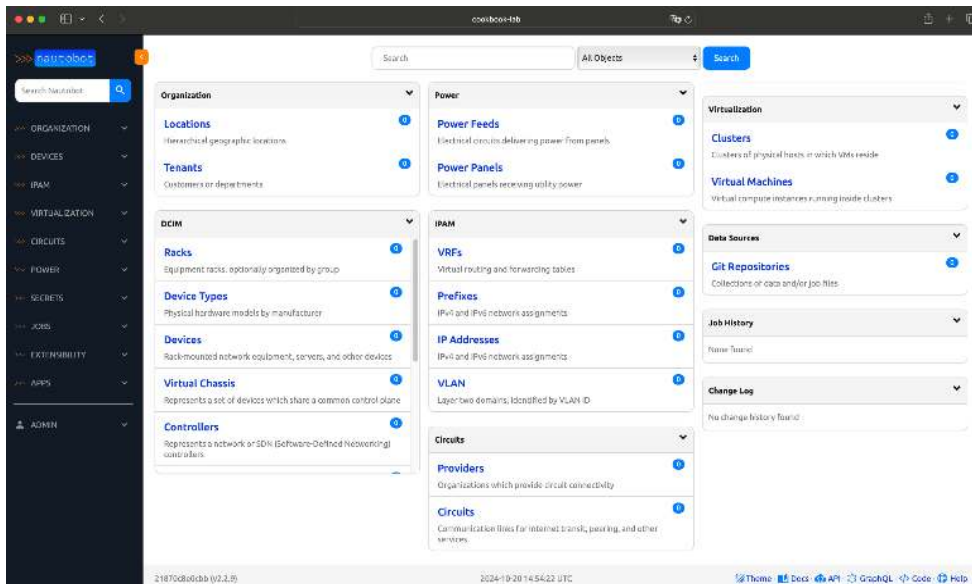


Figure 16.4 - Nautobot Home Page

For now, we don't need to populate Nautobot with data, but you may already infer that the same data that we use in Ansible in host and group vars will end up in Nautobot.

This has been the manual method to get the lab up and ready, but you have a simpler way

Automated Server Provisioning

The `labcli` utility also has Ansible playbooks (we use our own dog food) to set up a Digital Ocean droplet from scratch. We expect you to run the tool from your local machine to control all the server provisioning processes. You can check the last version of these instructions directly in the book repository.

1. Clone the Git Repository and Install the `cookbook-lab` tool (named `lab`), that leverages `labcli`

```
# pip install .
... omitted output ...
```

```
Installing collected packages: cookbook-lab
Successfully installed cookbook-lab-0.1.0
```

This tool is the same one we use inside the server to manage the lab scenario, but in this case, you have to run on your local machine.

2. Setup the environment file

Aside of the setup of the .env file, explained in the Clone Lab Environment section, here you also need a .setup.env file that holds the environment variable for the automated server provisioning. First, let's do a copy of the existing example .setup.env.

```
$ cd cookbook-lab
# Copy the example setup environment file
$ cp example.setup.env .setup.env
```

Then, adjust the data as needed for your case:

```
# Your DigitalOcean API token
DO_API_TOKEN="do_example123"
# The fingerprint of your SSH key registered with DigitalOcean
DO_SSH_
FINGERPRINT="aa:bb:cc:dd:ee:ff:gg:hh:ii:jj:kk:ll:mm:nn:oo:pp:qq"
# The local path to your SSH private key used to connect to the
DigitalOcean droplet
SSH_KEY_PATH=~/.ssh/id_rsa"
# The local path to the network images
CEOS_IMAGE_PATH=~/Downloads/cEOS-lab.tar"
CRPD_IMAGE_PATH=~/Downloads/cRPD.tgz"
# The URL of your forked repository and the branch to be used
# for running the lab scenarios in the DigitalOcean droplet
LABCLI_REPO="https://github.com/<your-user>/ Network-Automation-
Cookbook_2nd_Edition.git"
LABCLI_BRANCH="main"
```

3. Download the network container images not publicly available

Refer to the **Download and import the network device images** section, to get the images into your local machine. Make sure the path matches the previous .setup.env configuration.

4. Create a Digital Ocean API token

You need to create a Personal Access Token (PAT) for DigitalOcean. Follow the instructions in this guide: <https://docs.digitalocean.com/reference/api/create-personal-access-token/> to generate a PAT. Once you have created the PAT, save it in the `DO_API_TOKEN` environment variable in your `.setup.env` file.

5. Create an SSH key pair and get its fingerprint

Follow the same steps from the Manual VM Provisioning section to create an SSH key pair and upload the SSH **public** key to DigitalOcean. This will generate an SSH fingerprint (a representation of the public key) needed for provisioning.

- Navigate to **Settings > Security** in your DigitalOcean account.
- Copy the fingerprint value for the *SSH key* you plan to use.

Save this fingerprint in the `DO_SSH_FINGERPRINT` environment variable in `.setup.env`. This step ensures your SSH key is included in the DigitalOcean droplet build process, allowing you access post-creation.

6. Create the DigitalOcean droplet

You're all set to begin creating your DigitalOcean droplet! To kick things off, simply run the command `lab setup deploy`. This will trigger an Ansible playbook that guides you through the process, asking for key details like droplet size, region, and more. Just follow the prompts, as illustrated in the figure below, and the provisioning process will automatically begin setting up your environment quickly and effortlessly.

```
$ lab setup deploy
[17:42:48] Running command: ansible-playbook setup/create_droplet.
yaml -i setup/inventory/localhost.yaml
Enter the droplet image [ubuntu-22-04-x64]:
Enter the droplet size [s-4vcpu-8gb]:
Enter the droplet region [fra1]:
PLAY [Stand up cookbook-droplet] *****
*****

TASK [Create DigitalOcean Droplets] *****
*****

ok: [localhost]

TASK [Wait for droplets to be ready] *****
*****
```

```
ok: [localhost]
TASK [Show Droplet info] *****
*****

ok: [localhost] => {}
MSG:
Droplet ID is <id>
Public IPv4 is <ip-address>
```

Step 2. Removing the Lab Environment

When you’ve wrapped up your experiments and gained all that valuable knowledge from your lab environment, it’s a good idea to tear it down—especially if you’re running it in a cloud environment where costs can quickly add up. This process, known as *destroying* the environment, wipes the slate clean. It removes all the components you’ve created, frees up the resources, and restores your system back to its pre-lab state. It’s like hitting the reset button, keeping your setup lean and your costs under control!

To destroy the lab scenario you can use the following commands:

```
$ lab setup destroy
```

To remove the remote VM/DigitalOcean droplet, you can run the command `lab setup destroy`. This will delete the droplet from your account, which is useful when you have finished the labs for the day.

Summary

This chapter introduces the hands-on lab environment that serves as the foundation for practicing and applying the concepts covered throughout the book. The lab is designed to be accessible and efficient, allowing readers to focus on learning rather than dealing with complex or resource-heavy setups. It provides a consistent environment for reproducing the steps and examples in the book, while also offering the flexibility to apply these ideas in any network environment.

Unlock this book’s exclusive benefits now

Scan this QR code or go to packtpub.com/unlock, then search this book by name.

Note: Keep your purchase invoice ready before you start.



Index

A

Access Control List (ACL) 9, 113

Address Resolution Protocol (ARP) 211

advanced routing configuration

with template module 153-157, 190-195

advanced YAML syntax

using 60

YAML aliases 64, 65

YAML anchors 64, 65

YAML multiline strings 60-63

AI Code Explainer 218

Amazon Machine Images (AMIs) 486

Amazon Web Services (AWS) 4

Ansible 5, 10, 11, 411

Batfish, integrating with 346-348

characteristics 11, 12

conditionals, using 31, 33

custom Python filters, creating 79-83

execution strategies, controlling 65-67

fact caching, configuring 75-78

IGWs, deploying with 256-258

Jinja2 templates, using with 39-42

NACLs, deploying with 262-264

NAPALM, installing with 217-222

Nautobot, integrating with 371-376

network snapshot, initializing 350-353

reports, building with 234-236

role, in network automation strategy 12

running, in check mode 57-60

subnets, deploying with 254-256

used, for automating firewall rule 494-498

used, for collecting Arista device facts 128-130

used, for collecting Juniper device facts 160-162

used, for collecting Nokia device facts 208, 210

used, for configuring Arista interfaces 103, 105-110

used, for controlling routing within VPC 258-261

used, for decommissioning resources on AWS 266-268

used, for deploying VPC 250-253

used, for deployment validation 264-266

used, for integrating Terraform outputs 488

Ansible Automation Platform (AAP) 12, 412

Ansible community package 10

Ansible Content Collections 53

using 53-56

Ansible control host 140

Ansible Core 10, 54

community package 54

Ansible dynamic inventory

Nautobot, using as 402-405

Ansible Galaxy 10

Ansible inventory

building 244, 246, 274, 305

building, from Terraform state 488, 489

Ansible loops

using 33-36

Ansible network inventory

augmenting 89, 135, 136, 171-173

Ansible network-related filters

using 42-45

Ansible ping module 139**Ansible playbook**

executing, from within Terraform 489-491

Ansible roles 50

using 50-53

Ansible tags

using 45-47

Ansible test (assert) 77**Ansible Vault**

used, for securing secrets 37-39

Ansible Web eXecutable (AWX) 412

execution environment, managing 433-435

installing 413-417

network credentials, managing 426-429

network inventory, creating 421-426

projects, creating 430-433

teams, managing 417-421

templates, creating 435-440

users, managing 417-421

workflow templates, creating 440-444

application programming interfaces (APIs) 5**arbitrary commands**

running, on Arista devices 130

running, on Juniper devices 162, 163

running, on Nokia devices 210-213

Arista cEOS devices 39, 89

reference link 520

Arista device facts

collecting, with Ansible 128-130

Arista devices 133

arbitrary commands, running on 130

configuration, pushing 125-127

connecting to 92-95

generic system options,

configuring on 101, 102

Arista eAPI

used, for migrating connections from SSH to
HTT(S) 95-98

Arista interfaces

configuring, with Ansible 103-110

automated server provisioning 524-526**Autonomous System Number**

(ASN) 111, 154, 190, 204

AWS account

authenticating 246-250

AWS EC2 Ansible dynamic inventory plugin

using 492, 493

AWS Python SDK

installing 243, 244

AWS Virtual Private Clouds (VPCs) 282**AWX 451**

used, for integrating Terraform 498-507

AWX API

automation tasks, running 444, 445

AWX application prerequisites

preparing 463-473

AWX Execution Environment (EE) 433**Azure account**

authenticating 276-280

Azure Python SDK

installing 273, 274

**Azure Resource Manager (ARM) deployment
model 280****Azure resources**

decommissioning 297, 298

Azure Virtual Networks (VNETs) 282**B****bastion host 321****Batfish 341, 342**

high-level architecture 342

- installing 344-346
- integrating, with Ansible 346-348
- network facts, collecting 353, 355
- reference link 342
- URL 341
- working 342

Batfish assertions

- network, validating 356-362

BGP configuration

- finalizing 204-206
- finalizing, to core 123, 124

BGP neighbor facts

- retrieving, from network devices 226-228

Border Gateway Protocol (BGP) 4, 133**business continuity (BC) 7****C****Campus Fabric 169****Centralized Logging 412****Certificate Authority (CA) 97****check mode 57****Chef 5****client-side software development
kit (SDK) 342****cloud networks**

- populating, in Nautobot 394-402

command-line interfaces (CLIs) 4, 411

- migrating, from NETCONF protocol 140-143

command-line interfaces (CLIs)

- migrating, from NETCONF protocol 140-143

conda 521**conditionals 31****configurations**

- generating, with Nautobot 406-408

connections

- populating, in Nautobot 384-387

connections, migrating from SSH to HTTP(S)

- with Arista eAPI 95-98

Containerlab

- reference link 512, 518

Content Collections 10**continuous integration/continuous
deployment (CI/CD) pipelines 5, 343****Contributor role 276****Credential Management 411****cURL command-line tool 444****custom Python filters**

- creating, for Ansible 79-84

D**data center fabric 89****decision environments (DEs) 459**

- creating 459, 460
- reference link 460

deployment validation 294-297

- Ansible, using for 264-266

devices

- populating, in Nautobot 380-384

DigitalOcean 513**Docker 344**

- reference link 512

Domain Name System (DNS)

- servers 147, 179

domain-specific language (DSL) 481**Don't Repeat Yourself (DRY) pattern 60****dynamic infrastructure**

- best practices, with Terraform 507
- concepts 481, 482
- deploying, with Terraform 481-487

E

EDA-Server 451

- installing 452-454
- projects, creating 455, 456
- rulebook, activating 461, 462

EDA-Server application prerequisites

- preparing 463-473

Egress direction 317

Elastic Compute Cloud (EC2) 486

Event-Driven Ansible (EDA) 449

- EDA-Server 450
- standalone 450

event-driven workflow

- executing 473-476

EVPN/VXLAN fabric 111

execution environment (EEs) 413, 459

- managing, in AWX 433-435

execution strategies, in Ansible

- controlling 65, 67
- free strategy 67
- host_pinned strategy 68-70
- serial keyword 70-74

F

fabric configuration, for Arista

Data Center 111

- overlay configuration, generating 117-119
- underlay configuration, generating 111-116
- VXLAN configuration, generating 119-122

fact caching

- configuring 75-78
- reference link 79

firewall rule 317

- automating, with Ansible 494-498

- deploying, in GCP 317-321

folded block 61

forking 515, 516

Fully Qualified Collection Name (FQCN) 54

G

GCP account

- authenticating to 306-309

GCP Python SDK

- installing 303, 305

GCP resources

- decommissioning 333-336

GCP VPC

- networks, creating 310-314
- routing, adjusting within 328, 329
- validating 330-332

GCP VPC networks

- creating 310-314

generic system options

- configuring, on Arista devices 101, 102
- configuring, on Juniper devices 147, 148
- configuring, on Nokia devices 179-181

Google Cloud Platform (GCP) 301

- firewall rules, deploying 317-327
- VMs, deploying 321

Graphical User Interface

(GUI) 12, 330, 411, 413

H

HashiCorp Configuration Language

(HCL) 481

Helm 415

host inventory file 244

hostname 147, 179

I

Identity and Access Management (IAM) 246

Infrahub 365

infrastructure as code (IaC) 5, 365, 481

Ingress direction 317

Integrated Routing Bridge (IRB) 186

interior gateway protocol (IGP) 153

internal Border Gateway Protocol (iBGP) 195

Internet Gateway (IGW)

deploying, with Ansible 256-258

inventory 421

Inventory Management 411

IP addresses

populating, in Nautobot 390-393

IP address management (IPAM) 366, 388

IP prefixes

populating, in Nautobot 388-390

iptables

reference link 494

J

Jinja2 templates

using, with Ansible 39-42

jq Linux utility 444

Juniper core network 133

Juniper cRPD

reference link 520

Juniper devices

arbitrary commands, running on 162, 163

configuration, pushing to 158-160

connecting 136-140

facts, collecting with Ansible 160-162

generic system options, configuring
on 147, 148

Juniper interfaces

configuring 149-152

Junos commands

parsing, with `parse_xml` 163-167

Junos devices 140

K

Kind 413

Kubectrl 414

L

labcli utility

reference link 521

lab environment

automated server provisioning 524-526

building 516, 517

forking, from Git Repository 515

hardware requirements 513

manual server provisioning 517

removing 527

setting up 512, 513

software requirements 514, 515

leafs 171

linear 65

literal block 60

local emulator 413

locations

populating, in Nautobot 377-380

M

manual server provisioning 517

Containerlab, installing 518

Docker, installing 518

Git Repository, cloning 519

labcli utility, installing 521-523

- Nautobot, preparing 523, 524
- network device images, downloading 520
- network device images, importing 520
- VM, creating in Digital Ocean 517, 518

merge key 65

N

NAPALM

- installing, with Ansible 217-222
- network, auditing with 232, 233
- network devices, configuring with 228-231
- network facts, retrieving with 222-226
- network, validating with 232, 233
- reports, building with 234-236

native event sources

- reference link 477

Nautobot 365, 413, 451, 511

- cloud networks, populating 394-402
- configurations, generating 406-408
- connections, populating 384-388
- devices, populating 380-384
- installing 369, 371
- integrating, with Ansible 371-376
- IP addresses, populating 390-393
- IP prefixes, populating 388-390
- locations, populating 377-379
- preparing 523, 524
- reference link 366
- using, as Ansible dynamic inventory 402-405

Nautobot application prerequisites

- preparing 463-473

Nautobot GraphQL API 405

NetBox 365

NETCONF protocol 140

- CLI, migrating from 140-143

NetUtils Python library

- reference link 223

network

- validating, with Batfish assertions 356-362

Network Access Control Lists (NACLs)

- deploying, with Ansible 262

network address translation (NAT) 494

network automation 4

- building blocks 7-9
- drivers 6, 7
- SDN, migrating to NetDevOps 4, 5

network automation architecture

- automation engine 9
- network infrastructure 8
- orchestration 9
- source of truth (SoT) 9
- telemetry and observability 9
- user interaction 8

Network Configuration

(NETCONF) 4, 136, 348, 349

network credentials

- managing, on AWX 426-429

network devices

- BGP neighbor facts, retrieving from 226-228
- configuring, with NAPALM 228-231

network facts

- collecting, from Batfish 353, 355
- retrieving, with NAPALM 222-226

network interface card (NIC) 280

network inventory

- creating, on AWX 421-426

network operating system (NOS) 169

Network Security Groups (NSGs)

- deploying 290-293

network snapshot

- initializing, with Ansible 350-353

network source of truth (NSoT) 365

network tags 318

Nokia Ansible modules 169

Nokia devices

arbitrary commands, running on 210-213

configuration, pushing to 206, 208

connecting to 173-176

facts, collecting with Ansible 208, 210

generic system options, configuring
on 179, 181

interfaces, configuring 181-190

Nokia SR Linux devices 169

Nornir 9

O

Open Shortest Path First (OSPF) 4, 153

Organization 417

out-of-band (OOB) environments 75

overlay BGP EVPN 117

overlay configuration

generating 195-198

P

parse_xml

used, for parsing Junos commands 163-167

password hashing filter 146

playbook

building 31

PostgreSQL database 366, 413

Privacy Enhanced Mail (PEM) 482

projects

creating, in AWX 430-433

pull category 456

Puppet 5

push category 456

pybatfish 342

reference link 342

R

Redis 413

remaining VXLAN configuration

generating 199-204

remote procedure call (RPC) 161

Representational State Transfer (REST) 95

resource groups

creating 280-282

resources, on AWS

decommissioning, with Ansible 266-268

REST API 444

RESTful API 412

role 347

Role-Based Access Control (RBAC) 411

routing

adjusting, within GCP VPC 328, 329

routing, within VPC

controlling, with Ansible 258-261

rulebook 450

creating 456-459

S

Secure Shell (SSH) 92, 136

service account 306

service principal 276

Service Router Linux (SR Linux) 169

Simple Storage Service (S3) 488

single pane of glass (SPOG) 366

software-defined networking (SDN) 3

migrating, to NetDevOps 5

source of truth (SoT) 366

spines 171

SR Linux YANG data model

URL 178

static inventory file 421

strategy 65

subnets

creating 285, 286, 315, 316

deploying, with Ansible 254-256

T

task engine 413

template module

using, for advanced routing

configuration 153-157, 190-195

templates

creating, in AWX 435-440

Terraform

dynamic infrastructure, best practices 507

integrating, with AWX 498-507

outputs, integrating with Ansible 488

reference link 479

used, for deploying dynamic
infrastructure 481-487

U

user-defined routes (UDRs) 287

building 286-289

user interface (UI) 7, 450

users and authentication options

creating 98-100, 144-147, 176-179

managing 98-100, 144-147, 176-179

V

version control systems (VCS) 429

virtual machine (VM) 280, 482

deploying, in GCP 321-327

virtual networks

creating 282-284

Virtual Private Clouds (VPCs) 244

deploying, with Ansible 250-253

**virtual routing and forwarding functions
(VRF functions)** 94, 388

VxLAN Network Identifier (VNI) 200

VXLAN Tunnel Endpoints (VTEPs) 201

W

workflow templates

creating, in AWX 440-444

Y

YAML

aliases 64

anchors 64

multiline strings 60-62

Z

Zero-Touch Provisioning (ZTP) 95



packtpub.com

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

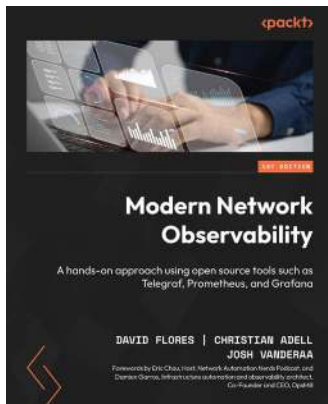
Why subscribe?

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Modern Network Observability

David Flores, Christian Adell, Josh VanDeraa

ISBN: 978-1-83508-106-8

- Collect and normalize data from various sources using Telegraf and Logstash
- Enrich operational data with crucial context from a Source of Truth such as Nautobot
- Visualize data and create insightful dashboards with Grafana
- Automate alerts and responses for your network operations strategy using Prefect
- Understand when to build or buy an observability stack, with tips and best practices
- Explore practical machine learning techniques to enhance observability data value



Zabbix 7 IT Infrastructure Monitoring Cookbook - 3rd Edition

Nathan Liefing, Brian van Baekel

ISBN: 978-1-80107-832-0

- Implement a high-availability Zabbix setup for both server and proxies
- Build templates and explore various monitoring types available in Zabbix 7
- Use Zabbix proxies to scale your environment effectively
- Work with custom integrations and the Zabbix API
- Set up advanced triggers and alerting
- Maintain your Zabbix setup for scaling, backups, and upgrades
- Perform advanced Zabbix database management
- Monitor cloud-based products such as Amazon Web Services (AWS), Azure, and Docker

Packt is searching for authors like you

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

Share your thoughts

Now you've finished *Network Automation Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

