Network Automation Cookbook

Proven and actionable recipes to automate and manage network devices using Ansible

Karim Okasha

Packt

www.packt.com

Network Automation Cookbook

Proven and actionable recipes to automate and manage network devices using Ansible

Karim Okasha

Packt

BIRMINGHAM - MUMBAI

Network Automation Cookbook

Commissioning Editor: Vijin Boricha
Acquisition Editor: Rohit Rajkumar
Content Development Editor: Ronn Kurien
Senior Editor: Richard Brookes-Bland
Technical Editor: Dinesh Pawar
Copy Editor: Safis Editing
Project Coordinator: Neil Dmello
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Nilesh Mohite

First published: April 2020

Production reference: 1170420

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

ISBN 978-1-78995-648-1

`www.packt.com`

Contributors

About the author

Karim Okasha is a network consultant with over 15 years of experience in the ICT industry. He is specialized 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 in Red Hat as a network automation consultant, helping large telecom and service providers to 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.

I would like to thank my wife and kids for providing me with the freedom and understanding needed to focus on this dream; without their support, this book wouldn't be possible.

I would like to thank the Packt Publishing team and my technical reviewers, for making my dream of writing this book a reality.

Finally, I would like to thank my mentor and best friend, Mohammed Mahmoud, for all his support and encouragement during all these years.

About the reviewers

Mohamed Radwan is a senior network architect with 20 years of experience in designing solutions for telecommunications, global service providers, data centers, the cloud, governments, and Fortune 500 companies in Europe, the Middle East, and the Asia-Pacific. He is the author of *CCDE: The Practical Guide*, he is an award-winning network designer, and he holds bachelor's degree in engineering – computers and systems, in addition to many expert-level certificates. He currently lives in Sydney, Australia, working within the Cisco Advanced Services team. Before that, he worked with Orange S.A, Saudi Telecom Company, Qatar Foundation, and Vodafone.

Bassem Aly is a senior SDN/NFV solution consultant at Juniper Networks and has been working in the telecom industry for the last 10 years. He is focused on designing and implementing next-generation networks by leveraging different automation and DevOps frameworks. Also, he has extensive experience in architecting and deploying telecom applications over OpenStack. Bassem also conducts corporate training on network automation and network programmability using Python and Ansible. Finally, he's an active blogger on different technology areas and is the author of *Hands-On Enterprise Automation with Python*, published by Packt.

Table of Contents

Preface

Network Automation Cookbook provides an overview of the various topics of network automation and how to use software development practices in order to design and operate different networking solutions. We use Ansible as our framework to introduce the topic of network automation and how to manage different vendor equipment using Ansible. In the first section, we outline how to install and configure Ansible specifically for the purpose of network automation. We will explore how we can use Ansible to manage traditional network solutions from various vendors such as Cisco, Juniper, Arista, and F5. Next, we continue to explore how to utilize Ansible to build and scale network solutions from major cloud providers such as AWS, Azure, and **Google Cloud Platform** (**GCP**). Finally, we outline different supporting open source projects in network automation, such as NetBox, Batfish, and AWX. We outline how to integrate all these tools with Ansible in order to build a complete framework for network automation.

By the end of this book, you will have a solid foundation on how to integrate Ansible with different vendor equipment and how to build a network automation solution based on Ansible. Further, you will understand how to use various open source projects and how to integrate all these solutions with Ansible to build a robust and scalable network automation framework.

Who this book is for

This book is ideal for IT professionals and network engineers who are responsible for the design and operation of network devices within an organization and would like to expand their knowledge on using Ansible to automate their network infrastructure. Basic knowledge of networking and Linux is recommended.

What this book covers

`Chapter 1`, *Building Blocks of Ansible*, focuses on how to install Ansible and describes the main building blocks of Ansible and how to utilize them to build advanced Ansible playbooks.

`Chapter 2`, *Managing Cisco IOS Devices Using Ansible*, focuses on how to integrate Ansible with Cisco IOS devices and how to use Ansible to configure Cisco IOS devices. We will explore the core Ansible modules developed to interact with Cisco IOS devices. Finally, we will explore how to use the Cisco PyATS library and how to integrate it with Ansible in order to validate the network state on Cisco IOS and Cisco IOS-XE devices.

`Chapter 3`, *Automating Juniper Devices in the Service Providers Using Ansible*, describes how to integrate Ansible with Juniper devices in **Service Provider** (**SP**) environments and how to manage the configuration of Juniper devices using Ansible. We will explore how to use the core Ansible modules developed to manage Juniper devices. Furthermore, we will explore the PyEZ library, which is used by Juniper custom Ansible modules to extend Ansible functionality in managing Juniper devices.

`Chapter 4`, *Building Data Center Networks with Arista and Ansible*, outlines how to integrate Ansible with Arista devices to build data center fabrics using EVPN/VXLANs. We will explore how to use the core Ansible modules developed to manage Arista devices and how to use these modules to configure and validate the network state on Arista switches.

`Chapter 5`, *Automating Application Delivery with F5 LTM and Ansible*, focuses on how to integrate Ansible with F5 BIG-IP LTM devices to onboard new BIG-IP LTM devices and how to set up the BIG-IP system as a reverse proxy for application delivery.

`Chapter 6`, *Administering Multi-Vendor Network with NAPALM and Ansible*, introduces the NAPALM library and outlines how to integrate this library with Ansible. We will explore how to utilize Ansible and NAPALM to simplify the management of multi-vendor environments.

`Chapter 7`, *Deploying and Operating AWS Networking Resources with Ansible*, outlines how to integrate Ansible with your AWS environment and how to describe your AWS infrastructure using Ansible. We explore how to utilize the core Ansible AWS modules to manage networking resources in AWS in order to build your AWS network infrastructure using Ansible.

`Chapter 8`, *Deploying and Operating Azure Networking Resources with Ansible*, outlines how to integrate Ansible with your Azure environment and how to describe your Azure infrastructure using Ansible. We will explore how to utilize the core Ansible Azure modules to manage networking resources in Azure in order to build Azure network solutions using Ansible.

`Chapter 9`, *Deploying and Operating GCP Networking Resources with Ansible*, describes how to integrate Ansible with your GCP environment and how to describe your GCP infrastructure using Ansible. We explore how to utilize the core Ansible GCP modules to manage networking resources in GCP in order to build GCP network solutions using Ansible.

`Chapter 10`, *Network Validation with Batfish and Ansible*, introduces the Batfish framework for offline network validation and how to integrate this framework with Ansible in order to perform offline network validation using both Ansible and Batfish.

`Chapter 11`, *Building a Network Inventory with Ansible and NetBox*, introduces NetBox, which is a complete inventory system to document and describe any network. We outline how to integrate Ansible with NetBox and how to use NetBox data to build Ansible dynamic inventories.

`Chapter 12`, *Simplifying Automation with AWX and Ansible*, introduces the AWX project, which extends Ansible and provides a powerful GUI and API on top of Ansible to simplify running automation tasks within an organization. We outline the extra features provided by AWX and how to use it to manage network automation within an organization.

`Chapter 13`, *Advanced Techniques and Best Practices for Ansible*, describes various best practices and advanced techniques that can be used for more advanced playbooks.

To get the most out of this book

Basic knowledge regarding different networking concepts, such as **Open Shortest Path First** (**OSPF**) and **Border Gateway Protocol** (**BGP**), is assumed.

Basic knowledge of Linux is assumed, including knowledge of how to create files and folders and install software on Linux machines.

Software/hardware covered in the book	**OS requirements**
Ansible 2.9	CentOS 7
Python 3.6.8	

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

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

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/9781789956481_ColorImages.pdf`.

Code in Action

The code in action videos are based on Ansible version 2.8.5. The code has also been tested on version 2.9.2 and works fine.

Visit the following link to check out videos of the code being run:
`https://bit.ly/34JooNp`

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
$ cat ansible.cfg

[defaults]
 inventory=hosts
 retry_files_enabled=False
 gathering=explicit
 host_key_checking=False
```

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

```
- name: Configure ACL on IOS-XR
  hosts: all
  serial: 1
  tags: deploy
  tasks:
    - name: Backup Config
      iosxr_config:
        backup:
      when: not ansible_check_mode
    - name: Deploy ACLs
      iosxr_config:
        src: acl_conf.cfg
        match: line
      when: not ansible_check_mode
```

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

```
$ python3 -m venv dev
$ source dev/bin/activate
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

Reviews

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

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

1
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 different infrastructure nodes and translates the business logic into well-defined procedures in order to implement this business logic. Ansible is written in Python and it mainly relies on SSH to communicate with infrastructure nodes to execute instructions on them. It started support for networking devices beginning with Ansible 1.9, and with Ansible 2.9, its current support for network devices has grown extensively. It can interact with network devices using either SSH or via API if the network vendors support APIs on their equipment. It also provides multiple advantages, including the following:

- **An easy learning curve:** Writing Ansible playbooks requires knowledge of YAML and Jinja2 templates, which are easy to learn, and its descriptive language is easy to understand.
- **Agentless:** It doesn't require an agent to be installed on the remotely managed device in order to control this device.
- **Extensible:** Ansible comes equipped with multiple modules to execute a variety of tasks on the managed nodes. It also supports writing custom modules and plugins to extend Ansible's core functionality.
- **Idempotent:** Ansible will not change the state of the device unless it needs to in order to change its setting to reach the desired state. Once it is in this desired state, running Ansible Playbooks against the device will not alter its configurations.

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 main recipes that will be covered:

- Installing Ansible
- Building Ansible's inventory
- Using Ansible's variables
- Building Ansible's playbook
- Using Ansible's conditionals
- Using Ansible's loops
- Securing secrets with Ansible Vault
- Using Jinja2 with Ansible
- Using Ansible's filters
- Using Ansible Tags
- Customizing Ansible's settings
- Using Ansible Roles

The purpose of this chapter is to have a basic understanding of the different Ansible components that we will utilize throughout this book in order to interact with the networking device. Consequently, all the examples in this chapter are not focused on managing networking devices. Instead, we will focus on understanding the different components in Ansible in order to use them effectively in the next chapters.

Technical requirements

Here are the requirements for installing Ansible and running all of our Ansible playbooks:

- A Linux **Virtual Machine** (**VM**) with either of the following distributions:
    - Ubuntu 18.04 or higher
    - CentOS 7.0 or higher
- Internet connectivity for the VM

Setting up the Linux machine is outside the scope of this recipe. However, the easiest approach to setting up a Linux VM with any OS version is by using *Vagrant* to create and set up the Ansible VM.

Installing Ansible

The machine on which we install Ansible (this is known as the Ansible control machine) should be running on any Linux distribution. In this recipe, we will outline how to install Ansible on both an Ubuntu Linux machine or a CentOS machine.

Getting ready

To install Ansible, we need a Linux VM using either an Ubuntu 18.04+ OS or CentoS 7+ OS. Furthermore, this machine needs to have internet access for Ansible to be installed on it.

How to do it...

Ansible is written in Python and all its modules need Python to be installed on the Ansible control machine. Our first task is to ensure that Python is installed on the Ansible control machine, as outlined in the following steps.

1. Most Linux distributions have Python installed by default. However, if Python is not installed, here are the steps for installing it on Linux:

    - On an Ubuntu OS, execute the following command:

    ```
    # Install python3
    $sudo apt-get install python3

    # validate python is installed
    $python3 --version
    Python 3.6.9
    ```

    - On a CentOS OS, execute the following command:

    ```
    # Install python
    $sudo yum install pytho3

    # validate python is installed
    $python3 --version
    Python 3.6.8
    ```

2. After we have validated that Python is installed, we can start to install Ansible:

    - On an Ubuntu OS, execute the following command:

    ```
    # We need to use ansible repository to install the latest
    version of Ansible
    $ sudo apt-add-repository ppa:ansible/ansible

    # Update the repo cache to use the new repo added
    $ sudo apt-get update

    # We install Ansible
    $ sudo apt-get install ansible
    ```

    - On a CentOS OS, execute the following command:

    ```
    # We need to use latest epel repository to get the latest
    ansible
    $ sudo yum install epel-release

    # We install Ansible
    $ sudo yum install ansible
    ```

How it works..

The easiest way to install Ansible is by using the package manager specific to our Linux distribution. We just need to make sure that we have enabled the required repositories to install the latest version of Ansible. In both Ubuntu and CentOS, we need to enable extra repositories that provide the latest version for Ansible. In CentOS, we need to install and enable the **Extra Packages for Enterprise Linux Repository** (**EPEL repo**), which provides extra software packages and has the latest Ansible packages for CentOS.

Using this method, we will install Ansible and all the requisite system packages needed to run the Ansible modules. In both Ubuntu and CentOS, this method will also install Python 2 and run Ansible using Python 2. We can validate the fact that Ansible is installed and which version is used by running the following command:

```
$ ansible --version
ansible 2.9
  config file = /etc/ansible/ansible.cfg
  configured module search path =
[u'/home/vagrant/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
```

```
    python version = 2.7.5 (default, Aug 7 2019, 00:51:29) [GCC 4.8.5
20150623 (Red Hat 4.8.5-39)]
```

Also, we can check that Ansible is working as expected by trying to connect to the local machine using the `ping` module as shown:

```
$ ansible -m ping localhost

localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Using this method, we can see that it has the following issues:

- It uses Python 2 as the execution environment, but we want to use Python 3.
- It updates the Python packages installed on the system, which might not be desirable.
- It doesn't provide us with the granularity needed in order to select which version of Ansible to use. Using this method, we will always install the latest version of Ansible, which might not be what we need.

How it works...

In order to install Ansible in a Python 3 environment and to have more control over the version of Ansible deployed, we are going to use the pip Python program to install Ansible as shown here:

- Install Python 3 if it is not present, as follows:

```
# Ubuntu
$ sudo apt-get install python3

# CentOS
sudo yum install python3
```

- Install the `python3-pip` package:

```
# Ubuntu
$ sudo apt-get install python3-pip

# CentOS
$ sudo yum install python3-pip
```

- Install Ansible:

```
# Ubuntu and CentOS
# This will install ansible for the current user ONLY
$ pip3 install ansible==2.9 --user

# We Can install ansible on the System Level
$ sudo pip3 install ansible==2.9
```

- We can verify that Ansible has been installed successfully, as shown here:

```
$$ ansible --version
ansible 2.9
 config file = None
 configured module search path =
['/home/vagrant/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
 ansible python module location =
/home/vagrant/.local/lib/python3.6/site-packages/ansible
 executable location = /home/vagrant/.local/bin/ansible
 python version = 3.6.8 (default, Aug 7 2019, 17:28:10) [GCC 4.8.5
20150623 (Red Hat 4.8.5-39)]
```

Installing Ansible using this method ensures that we are using Python 3 as our execution environment and allows us to control which version of Ansible to install, as outlined in the example shown.

We are going to use this method as our Ansible installation method and all the subsequent chapters will be based on this installation procedure.

In `Chapter 13`, *Advanced Techniques and Best Practices for Ansible*, we will outline yet another method for installing Ansible using Python virtual environments.

See also...

For more information regarding the installation of Ansible, please check the following URL:

`https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html`

Building Ansible's inventory

After installing Ansible, we need to define Ansible's inventory, which is a text file that defines the nodes that Ansible will manage. In this recipe, we will outline how to create and structure Ansible's inventory file.

Getting ready

We need to create a folder that will contain all the code that we will outline in this chapter. We create a folder called `ch1_ansible`, as shown here:

```
$ mkdir ch1_ansible
$ cd ch1_ansible
```

How to do it...

Perform the following steps to create the inventory file:

1. Create a file named `hosts`:

```
$ touch hosts
```

2. Using any text editor, open the file and add the following content:

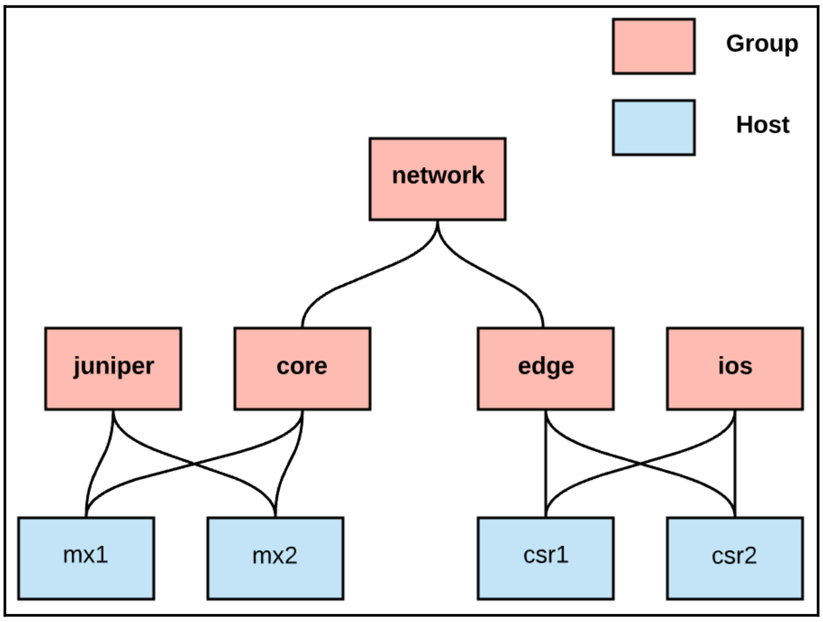
```
$ cat hosts

[cisco]
csr1 ansible_host=172.10.1.2
csr2 ansible_host=172.10.1.3

[juniper]
mx1 ansible_host=172.20.1.2
mx2 ansible_host=172.20.1.3

[core]
mx1
mx2

[edge]
csr[1:2]

[network:children]
core
edge
```

The Ansible inventory file can have any name. However, as a best practice, we will use the name `hosts` to describe the devices in our inventory.

How it works...

The Ansible inventory files define the hosts that will be managed by Ansible (in the preceding example, this is `csr1-2` and `mx1-2`) and how to group these devices into custom-defined groups based on different criteria. The groups are defined with `[]`. This grouping helps us to define the variables and simplify the segregation between the devices and how Ansible interacts with them. How we group the devices is based on our use case, so we can group them as per the vendor (Juniper and IOS) or function (core and edge).

We can also build hierarchies for the groups using the children, which is outlined in the inventory file. The following diagram shows how the hosts are grouped and how the group hierarchy is built:



Using Ansible's variables

Ansible stores the information for the nodes that it manages using Ansible variables. Ansible variables can be declared in multiple locations. However, in observing the best practices for Ansible, we will outline the two main parts where Ansible looks for variables for the nodes that are declared in the inventory file.

Getting ready

In order to follow along with this recipe, an Ansible inventory file must be already defined as outlined in the previous recipes.

How to do it...

In the inventory file, we define hosts and we group the hosts into groups. We now define two directories that Ansible searches for group variables and host variables:

1. Create two folders, `group_vars` and `host_vars`:

   ```
   $ cd ch1_ansible
   $ mkdir group_vars host_vars
   ```

2. Create `ios.yml` and `junos.yml` files inside `group_vars`:

   ```
   $ touch group_vars/cisco.yml group_vars/juniper.yml
   ```

3. Create `mx1.yml` and `csr1.yml` inside `host_vars`:

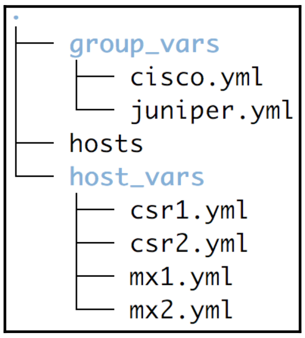
   ```
   $ touch host_vars/csr1.yml host_vars/mx1.yml
   ```

4. Populate variables in all the files, as shown here:

   ```
   $echo 'hostname: core-mx1' >> host_vars/mx1.yml
   $echo 'hostname: core-mx2' >> host_vars/mx2.yml
   $echo 'hostname: edge-csr1' >> host_vars/csr1.yml
   $echo 'hostname: edge-csr2' >> host_vars/csr2.yml
   $echo 'os: ios' >> group_vars/cisco.yml
   $echo 'os: junos' >> group_vars/juniper.yml
   ```

How it works...

We created the following structure of directories and files to host our variables, as shown in the following diagram:



```
.
├── group_vars
│   ├── cisco.yml
│   └── juniper.yml
├── hosts
└── host_vars
    ├── csr1.yml
    ├── csr2.yml
    ├── mx1.yml
    └── mx2.yml
```

All files inside the `group_vars` directory contain the group variables for the groups that we have defined in our inventory and they apply to all the hosts within this group. As for the files within `host_vars`, they contain variables for each host. Using this structure, we can group variables from multiple hosts into a specific group file and variables that are host-specific will be placed in a separate file specific to this host.

There's more...

In addition to `host_vars` and `group_vars`, Ansible supports the definition of variables using other techniques, including the following:

- Using the `vars` keyword within the play to specify multiple variables
- Using `vars_files` to define variables in a file and having Ansible read these variables from this file while running the playbook
- Specifying variables at the command line using the `--e` option

In addition to the user-defined variables that we can specify, Ansible has some default variables that it builds dynamically for its inventory. The following table captures some of the most frequently used variables:

`inventory_hostname`	The name of the hosts as defined in the inventory (for example, `csr1` and `mx1`)
`play_hosts`	A list of all the hosts included in the play
`group_names`	A list of all the groups that a specific host is a part of (for example, for `csr1` this will be [edge, Cisco, network])

Building Ansible's playbook

An Ansible playbook is the fundamental element in Ansible that declares what actions we would like to perform on our managed hosts (specified in the inventory). An Ansible playbook is a YAML-formatted file that defines a list of tasks that will be executed on our managed devices. In this recipe, we will outline how to write an Ansible playbook and how to define the hosts that will be targeted by this playbook.

Getting ready

In order to follow along with this recipe, an Ansible inventory file must already be defined, along with all the group- and host-specific variable files created in accordance with previous recipes.

How to do it...

1. Create a new file called `playbook.yml` inside the `ch1_ansible` folder and incorporate the following lines in this file:

```
$  cat playbook.yml

---
  - name: Initial Playbook
    hosts: all
    gather_facts: no
    tasks:
      - name: Display Hostname
        debug:
          msg: "Router name is {{ hostname }}"
      - name: Display OS
```

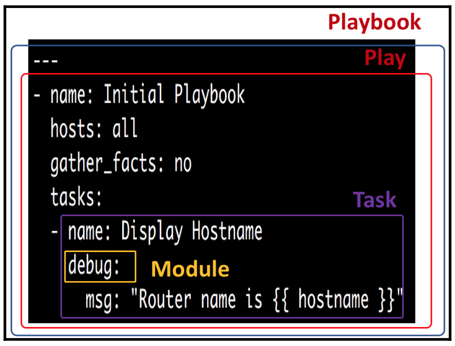
```
                debug:
                  msg: "{{ hostname }} is running {{ os }}"
```

2. Run the playbook as shown here:

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

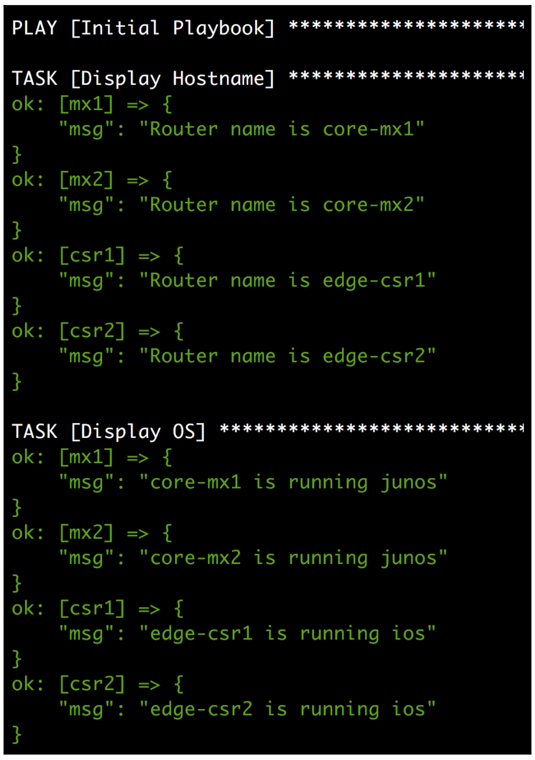
How it works...

The Ansible playbook is structured as a list of plays and each play targets a specific group of hosts (defined in the inventory file). Each play can have one or more tasks to execute against the hosts in this play. Each task runs a specific Ansible module that has a number of arguments. The general structure of the playbook is outlined in the following screenshot:



```
---
- name: Initial Playbook
  hosts: all
  gather_facts: no
  tasks:
  - name: Display Hostname
    debug:
      msg: "Router name is {{ hostname }}"
```

In the preceding playbook, we reference the variables that we defined in the previous recipe inside the `{{ }}` brackets. Ansible reads these variables from either `group_vars` or `host_vars`, and the module that we used in this playbook is the `debug` module, which displays as a custom message specified in the `msg` parameter to the Terminal output. The playbook run is shown here:



```
PLAY [Initial Playbook] *********************

TASK [Display Hostname] *********************
ok: [mx1] => {
    "msg": "Router name is core-mx1"
}
ok: [mx2] => {
    "msg": "Router name is core-mx2"
}
ok: [csr1] => {
    "msg": "Router name is edge-csr1"
}
ok: [csr2] => {
    "msg": "Router name is edge-csr2"
}

TASK [Display OS] ***************************
ok: [mx1] => {
    "msg": "core-mx1 is running junos"
}
ok: [mx2] => {
    "msg": "core-mx2 is running junos"
}
ok: [csr1] => {
    "msg": "edge-csr1 is running ios"
}
ok: [csr2] => {
    "msg": "edge-csr2 is running ios"
}
```

We use the `-i` option in the `ansible-playbook` command in order to point to the Ansible inventory file, which we will use as our source to construct our inventory.

TIP

In this playbook, I have used the `all` keyword to specify all the hosts within the inventory. This is a well-known group name that Ansible dynamically constructs for all hosts within the inventory.

Using Ansible's conditionals

One of the core features of Ansible is conditional task execution. This provides us with the ability to control which tasks to run on a given host based on a condition/test that we specify. In this recipe, we will outline how to configure conditional task execution.

Getting ready

In order to follow along with this recipe, an Ansible inventory file must be present and configured as outlined in the previous recipes. Furthermore, the Ansible variables for all our hosts should be defined as outlined in the previous recipes.

How to do it...

1. Create a new playbook called `ansible_cond.yml` inside the `ch1_ansible` folder.
2. Place the following content in the new playbook as shown here:

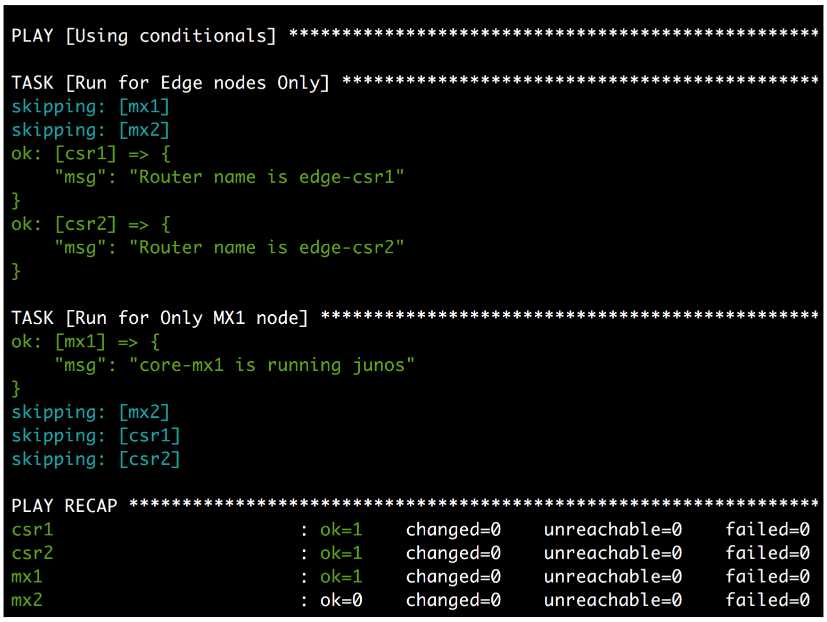
```
---
 - name: Using conditionals
   hosts: all
   gather_facts: no
   tasks:
     - name: Run for Edge nodes Only
       debug:
         msg: "Router name is {{ hostname }}"
       when: "'edge' in group_names"

     - name: Run for Only MX1 node
       debug:
         msg: "{{ hostname }} is running {{ os }}"
       when:
         - inventory_hostname == 'mx1'
```

3. Run the playbook as shown here:

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

How it works...

Ansible uses the `when` statement to provide conditional execution for the tasks. The `when` statement is applied at the task level and if the condition in the `when` statement evaluates to `true`, the task is executed for the given host. If `false`, the task is skipped for this host. The output as a result of running the preceding playbook is shown here:



```
PLAY [Using conditionals] ***************************************************

TASK [Run for Edge nodes Only] **********************************************
skipping: [mx1]
skipping: [mx2]
ok: [csr1] => {
    "msg": "Router name is edge-csr1"
}
ok: [csr2] => {
    "msg": "Router name is edge-csr2"
}

TASK [Run for Only MX1 node] ************************************************
ok: [mx1] => {
    "msg": "core-mx1 is running junos"
}
skipping: [mx2]
skipping: [csr1]
skipping: [csr2]

PLAY RECAP ******************************************************************
csr1                       : ok=1    changed=0    unreachable=0    failed=0
csr2                       : ok=1    changed=0    unreachable=0    failed=0
mx1                        : ok=1    changed=0    unreachable=0    failed=0
mx2                        : ok=0    changed=0    unreachable=0    failed=0
```

The `when` statement can take a single condition as seen in the first task, or can take a list of conditions as seen in the second task. If `when` is a list of conditions, all the conditions need to be true in order for the task to be executed.

In the first task, the `when` statement is enclosed in "" since the statement starts with a string. However, in the second statement, we use a normal `when` statement with no "" since the `when` statement starts with a variable name.

See also...

For more information regarding Ansible's conditionals, please check the following URL:

`https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html`

Using Ansible's loops

In some cases, we need to run a task inside an Ansible playbook to loop over some data. Ansible's loops allow us to loop over a variable (a dictionary or a list) multiple times in order to achieve this behavior. In this recipe, we will outline how to use Ansible's loops.

Getting ready

In order to follow along with this recipe, an Ansible inventory file must be present and configured, as outlined in the previous recipes.

How to do it...

1. Create a new playbook called `ansible_loops.yml` inside the `ch1_ansible` folder.
2. Inside the `group_vars/cisco.yml` file, incorporate the following content:

```
snmp_servers:
  - 10.1.1.1
  - 10.2.1.1
```

3. Inside the `group_vars/juniper.yml` file, incorporate the following content:

```
users:
  admin: admin123
  oper: oper123
```

4. Inside the `ansible_loops.yml` file, incorporate the following content:

```
---
  - name: Ansible Loop over a List
    hosts: cisco
    gather_facts: no
    tasks:
      - name: Loop over SNMP Servers
        debug:
          msg: "Router {{ hostname }} with snmp server {{ item }}"
        loop: "{{ snmp_servers}}"

  - name: Ansible Loop over a Dictionary
    hosts: juniper
    gather_facts: no
    tasks:
      - name: Loop over Username and Passowrds
        debug:
          msg: "Router {{ hostname }} with user {{ item.key }}
password {{ item.value }}"
        with_dict: "{{ users}}"
```

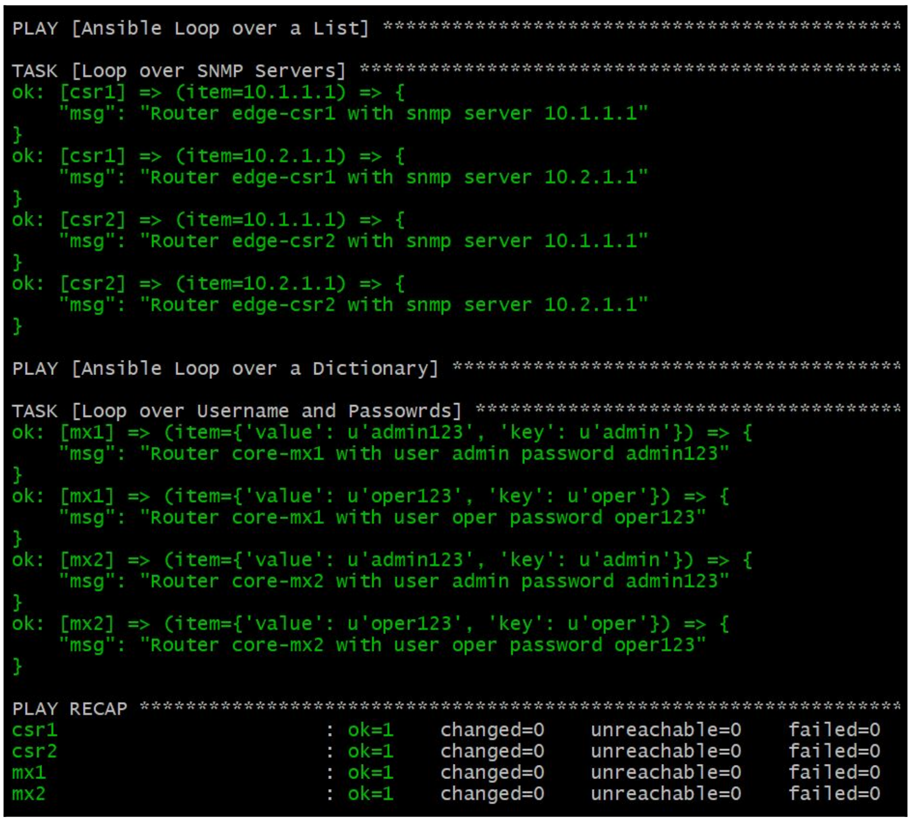
5. Run the playbook as shown here:

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

How it works..

Ansible supports looping over two main iterable data structures: lists and dictionaries. We use the `loops` keyword when we need to iterate over lists (`snmp_servers` is a list data structure) and we use `with_dicts` when we loop over a dictionary (`users` is a dictionary data structure where the username is the key and the passwords are the values). In both cases, we use the `item` keyword to specify the value in the current iteration. In the case of `with_dicts`, we get the key using `item.key` and we get the value using `item.value`.

The output of the preceding playbook run is shown here:



```
PLAY [Ansible Loop over a List] ************************************************

TASK [Loop over SNMP Servers] **************************************************
ok: [csr1] => (item=10.1.1.1) => {
    "msg": "Router edge-csr1 with snmp server 10.1.1.1"
}
ok: [csr1] => (item=10.2.1.1) => {
    "msg": "Router edge-csr1 with snmp server 10.2.1.1"
}
ok: [csr2] => (item=10.1.1.1) => {
    "msg": "Router edge-csr2 with snmp server 10.1.1.1"
}
ok: [csr2] => (item=10.2.1.1) => {
    "msg": "Router edge-csr2 with snmp server 10.2.1.1"
}

PLAY [Ansible Loop over a Dictionary] ******************************************

TASK [Loop over Username and Passowrds] ****************************************
ok: [mx1] => (item={'value': u'admin123', 'key': u'admin'}) => {
    "msg": "Router core-mx1 with user admin password admin123"
}
ok: [mx1] => (item={'value': u'oper123', 'key': u'oper'}) => {
    "msg": "Router core-mx1 with user oper password oper123"
}
ok: [mx2] => (item={'value': u'admin123', 'key': u'admin'}) => {
    "msg": "Router core-mx2 with user admin password admin123"
}
ok: [mx2] => (item={'value': u'oper123', 'key': u'oper'}) => {
    "msg": "Router core-mx2 with user oper password oper123"
}

PLAY RECAP *********************************************************************
csr1                       : ok=1    changed=0    unreachable=0    failed=0
csr2                       : ok=1    changed=0    unreachable=0    failed=0
mx1                        : ok=1    changed=0    unreachable=0    failed=0
mx2                        : ok=1    changed=0    unreachable=0    failed=0
```

See also...

For more information regarding the different Ansible *looping constructs,* please consult the following URL:

`https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html`

Securing secrets with Ansible Vault

When we are dealing with sensitive material that we need to reference in our Ansible playbooks, such as passwords, we shouldn't save this data in plain text. Ansible Vault provides a method to encrypt this data and therefore be safely decrypted and accessed while the playbook is running. In this recipe, we will outline how to use Ansible Vault in order to secure sensitive information in Ansible.

How to do it...

1. Create a new file called `decrypt_passwd` as shown:

```
$ echo 'strong_password' > decrypt_passwd
```

2. Using `ansible-vault` creates a new file called `secrets`, as shown here:

```
$ ansible-vault create --vault-id=decrypt_passwd secrets
```

3. Add the following variables to this new `secrets` file:

```
ospf_password: ospf_P@ssw0rD
bgp_password: BGP_p@ssw0rd
```

4. Create a new playbook called `ansible_vault.yml`, as shown here:

```
---
  - name: Using Ansible vault
    hosts: all
    gather_facts: no
    vars_files:
      - secrets
    tasks:
      - name: Output OSPF passowrd
        debug:
          msg: "Router {{ hostname }} ospf Password {{
ospf_password }}"
```

```
          when: inventory_hostname == 'csr1'

        - name: Output BGP passowrd
          debug:
            msg: "Router {{ hostname }} BGP Password {{ bgp_password
}}"
          when: inventory_hostname == 'mx1'
```

5. Run the playbook as shown here:

```
$ ansible-playbook --vault-id=decrypt_passwd ansible_vault.yml -i
hosts
```

How it works..

We use the `ansible-vault` command to create a new file that is encrypted using a key specified by `-- vault-id`. We place this key/password in another file (which is called `decrypt_passwd` in our example) and we pass this file as an argument to `vault-id`. Inside this file, we can place as many variables as we need. Finally, we include this file as a variable file in the playbook using `vars_files`. The following is the content of the secret file in case we try to read it without decryption:

```
$ cat secrets
$ANSIBLE_VAULT;1.1;AES256
613832643263633733363838396438343866613436303939656561356663363837633439383
13963
353837623061353432383335623766353266636362646264 0a663839396230646634353839 6
26461
313364613863616162613365346631373262653632616265366635646237646638616237356
33865
303335653639363132 0a643561623635363830653236633833383531366166326566623139 6
33838
326333356166636237613136306131346366356638653635633665643133653764313334616
23232
346338383338363638653132383639663034663730653565613536383637316161353861643
73263
666530653334643133383239633237653034
```

In order for Ansible to decrypt this file, we must supply the decryption password (stored in a `decrypt_passwd` file in this example) via the `--vault-id` option. When we run `ansible-playbook`, we must supply this decryption password, otherwise the `ansible-playbook` fails, as shown here:

```
### Running the Ansible playbook without --vault-id
$ansible-playbook ansible_vault.yml -i hosts
ERROR! Attempting to decrypt but no vault secrets found
```

There's more...

In case we don't want to specify the encryption/decryption password in the text file, we can use `--ask-vault-pass` with the `ansible-playbook` command in order to input the password while running the playbook, as shown here:

```
### Running the Ansible playbook with --ask-vault-pass
$ansible-playbook ansible_vault.yml -i hosts --ask-vault-pass
Vault password:
```

Using Jinja2 with Ansible

Jinja2 is a powerful templating engine for Python and is supported by Ansible. It is also used to generate any text-based files, such as HTML, CSV, or YAML. We can utilize Jinja2 with Ansible variables in order to generate custom configuration files for network devices. In this recipe, we will outline how to use Jinja2 templates with Ansible.

Getting ready

In order to follow along with this recipe, an Ansible inventory file must be present and configured as outlined in the previous recipes.

How to do it...

1. Create a new file inside the `group_vars` directory called `network.yml`:

```
$ cat group_vars/network.yml

---
ntp_servers:
    - 172.20.1.1
    - 172.20.2.1
```

2. Create a new `templates` directory and create a new `ios_basic.j2` file with the following content:

```
$ cat templates/ios_basic.j2
hostname {{ hostname }}
!
{% for server in ntp_servers %}
ntp {{ server }}
{% endfor %}
!
```

3. Create a new `junos_basic.j2` file within the `templates` directory with the following content:

```
$ cat templates/junos_basic.j2
set system host-name {{ hostname }}
{% for server in ntp_servers %}
set system ntp server {{ server }}
{% endfor %}
```

4. Create a new playbook called `ansible_jinja2.yml` with the following content:

```
---
  - name: Generate Cisco config from Jinja2
    hosts: localhost
    gather_facts: no
    tasks:
      - name: Create Configs Directory
        file: path=configs state=directory

  - name: Generate Cisco config from Jinja2
    hosts: cisco
    gather_facts: no
    tasks:
      - name: Generate Cisco Basic Config
        template:
          src: "templates/ios_basic.j2"
          dest: "configs/{{inventory_hostname}}.cfg"
        delegate_to: localhost

  - name: Generate Juniper config from Jinja2
    hosts: juniper
    gather_facts: no
    tasks:
      - name: Generate Juniper Basic Config
```

```
            template:
              src: "templates/junos_basic.j2"
              dest: "configs/{{inventory_hostname}}.cfg"
            delegate_to: localhost
```

5. Run the Ansible playbook as shown here:

```
$ ansible-playbook -i hosts ansible_jinja2.yml
```

How it works...

We created the `network.yml` file in order to group all the variables that will apply to all devices under this group. After that, we create two Jinja2 files, one for Cisco IOS devices, and the other for Juniper devices. Inside each Jinja2 template, we reference the Ansible variables using `{{}}`. We also use the `for` loop construct, `{% for server in ntp_servers %}` , supported by the Jinja2 templating engine in order to loop over the `ntp_servers` variable (which is a list) to access each item within this list.

Ansible provides the `template` module that takes two parameters:

- `src`: This references the Jinja2 template file.
- `dest`: This specifies the output file that will be generated.

In our case, we use the `{{inventory_hostname}}` variable in order to make the output configuration file unique for each router in our inventory.

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

The first play in the playbook creates the `configs` directory to store the configuration files for the network devices. The second play runs the template module on Cisco devices, and the third play runs the `template` task on Juniper devices.

The following is the configuration file for one of the Cisco devices:

```
$ cat configs/csr1.cfg
hostname edge-csr1
!
ntp 172.20.1.1
ntp 172.20.2.1
!
```

This is the configuration file for one of the Juniper devices:

```
$ cat configs/mx1.cfg
set system host-name core-mx1
set system ntp server 172.20.1.1
set system ntp server 172.20.2.1
```

See also...

For more information regarding the Ansible template module, please consult the following URL:

`https://docs.ansible.com/ansible/latest/modules/template_module.html`

Using Ansible's 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.

How to do it...

1. Install `python3-pip` and Python's `netaddr` library, since we will be using the Ansible IP filter, which requires Python's `netaddr` library:

```
# On ubuntu
$ sudo apt-get install python3-pip

# On CentOS
$ sudo yum install python3-pip

$ pip3 install netaddr
```

2. Create a new Ansible playbook called `ansible_filters.yml`, as shown here:

```
---
 - name: Ansible Filters
   hosts: csr1
   vars:
          interfaces:
            - { port: FastEthernet0/0, prefix: 10.1.1.0/24 }
            - { port: FastEthernet1/0, prefix: 10.1.2.0/24 }
        tasks:
          - name: Generate Interface Config
            blockinfile:
              block: |
                hostname {{ hostname | upper }}
                {% for intf in interfaces %}
                !
                interface {{ intf.port }}
                  ip address {{intf.prefix | ipv4(1) | ipv4('address')
}} {{intf.prefix | ipv4('netmask') }}
                !
                {% endfor %}
              dest: "configs/csr1_interfaces.cfg"
              create: yes
            delegate_to: localhost
```

How it works...

First of all, we are using the `blockinfile` module to create a new configuration file on the Ansible control machine. This module is very similar to the `template` module. However, we can write the Jinja2 expressions directly in the module in the `block` option. We define a new variable called `interfaces` using the `vars` parameter in the playbook. This variable is a list data structure where each item in the list is a dictionary data structure. This nested data structure specifies the IP prefix used on each interface.

In the Jinja2 expressions, we can see that we have used a number of filters as shown here:

- `{{ hostname | upper}}`: `upper` is a Jinja2 filter that transforms all the letters of the input string into uppercase. In this way, we pass the value of the hostname variable to this filter and the output will be the uppercase version of this value.
- `{{intf.prefix | ipv4(1) | ipv4('address') }}`: Here, we use the Ansible IP address filter twice. `ipv4(1)` takes an input IP prefix and outputs the first IP address in this prefix. We then use another IP address filter, `ipv4('address')`, in order to only get the IP address part of an IP prefix. So in our case, we take `10.1.1.0/24` and we output `10.1.1.1` for the first interface.
- `{{intf.prefix | ipv4('netmask') }}`: Here, we use the Ansible IP address filter to get the netmask of the IP address prefix, so in our case, we get the `/24` subnet and transform it to `255.255.255.0`.

The output file for the `csr1` router after this playbook run is shown here:

```
$ cat configs/csr1_interfaces.cfg
# BEGIN ANSIBLE MANAGED BLOCK
hostname EDGE-CSR1
!
interface FastEthernet0/0
  ip address 10.1.1.1 255.255.255.0
!
!
interface FastEthernet1/0
  ip address 10.1.2.1 255.255.255.0
!
# END ANSIBLE MANAGED BLOCK
```

Using Ansible Tags

Ansible Tags is a powerful tool that allows us to tag specific tasks within a large Ansible playbook and provides us with the flexibility to choose which tasks will run within a given playbook based on the tags we specify. In this recipe, we will outline how to configure and use Ansible Tags.

How to do it...

1. Create a new Ansible playbook called `ansible_tags.yml`, as shown here:

```
---
  - name: Using Ansible Tags
    hosts: cisco
    gather_facts: no
    tasks:
      - name: Print OSPF
        debug:
          msg: "Router {{ hostname }} will Run OSPF"
        tags: [ospf, routing]

     - name: Print BGP
       debug:
         msg: "Router {{ hostname }} will Run BGP"
       tags:
         - bgp
         - routing
    - name: Print NTP
      debug:
        msg: "Router {{ hostname }} will run NTP"
      tags: ntp
```

2. Run the playbook as shown here:

```
$ ansible-playbook ansible_tags.yml -i hosts --tags routing
```

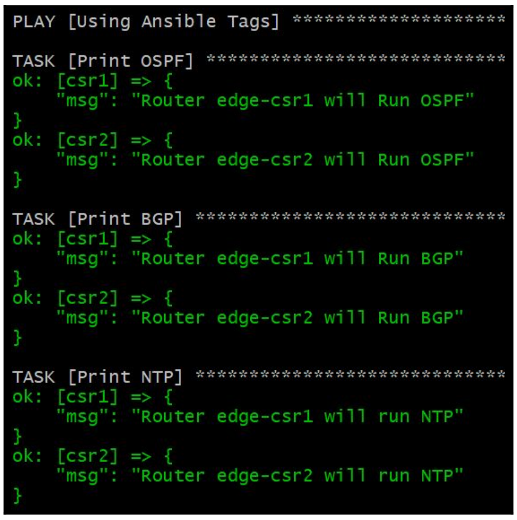
3. Run the playbook again, this time using tags, as shown here:

```
$ ansible-playbook ansible_tags.yml -i hosts --tags ospf

$ ansible-playbook ansible_tags.yml -i hosts --tags routing
```

How it works...

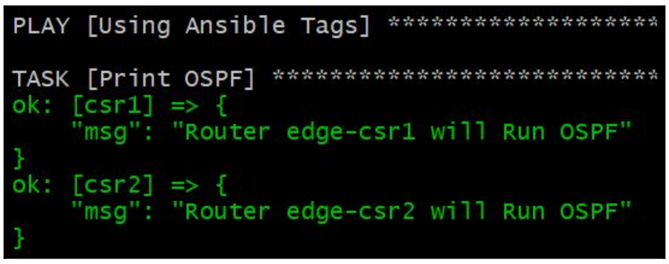
We can use tags to mark both tasks and plays with a given tag in order to use it to control which tasks or plays get executed. This gives us more control when developing playbooks to allow us to run the same playbook. However, with each run, we can control what we are deploying. In the example playbook in this recipe, we have tagged the tasks as OSPF, BGP, or NTP and have applied the `routing` tag to both the OSPF and BGP tasks. This allows us to selectively run the tasks within our playbook as shown here:

- With no tags specified, this will run all the tasks in the playbook with no change in the behavior, as shown in the following screenshot:



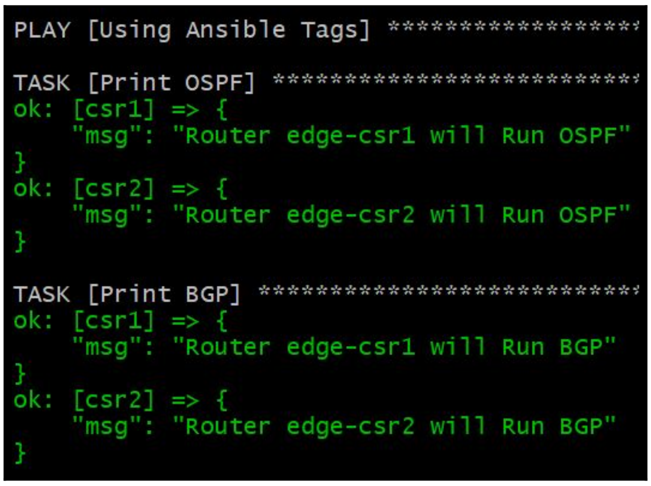
```
PLAY [Using Ansible Tags] *********************

TASK [Print OSPF] *****************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will Run OSPF"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will Run OSPF"
}

TASK [Print BGP] ******************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will Run BGP"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will Run BGP"
}

TASK [Print NTP] ******************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will run NTP"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will run NTP"
}
```

- Using the `ospf` tag, we will only run any task marked with this tag, as shown here:



```
PLAY [Using Ansible Tags] ********************

TASK [Print OSPF] ****************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will Run OSPF"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will Run OSPF"
}
```

- Using the `routing` tag, we will run all tasks marked with this tag, as shown here:



```
PLAY [Using Ansible Tags] ******************

TASK [Print OSPF] **************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will Run OSPF"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will Run OSPF"
}

TASK [Print BGP] ***************************
ok: [csr1] => {
    "msg": "Router edge-csr1 will Run BGP"
}
ok: [csr2] => {
    "msg": "Router edge-csr2 will Run BGP"
}
```

See also...

For more information regarding Ansible Tags, please consult the following URL:

`https://docs.ansible.com/ansible/latest/user_guide/playbooks_tags.html`

Customizing Ansible's settings

Ansible has many setting that can be adjusted and controlled using a configuration file called `ansible.cfg`. This file has multiple options that control many aspects of Ansible, including how Ansible looks and how it connects to managed devices. In this recipe, we will outline how to adjust some of these default settings.

How to do it...

1. Create a new file called `ansible.cfg`, as shown here:

```
[defaults]
inventory=hosts
vault_password_file=decryption_password
gathering=explicit
```

How it works...

By default, Ansible's settings are controlled by the `ansible.cfg` file located in the `/etc/ansible` directory. This is the default configuration file for Ansible that controls how Ansible interacts with managed nodes. We can edit this file directly. However, this will impact any playbook that we will use on the Ansible control machine, as well as any user on this machine. A more flexible and customized option is to include a file named `ansible.cfg` in the project directory and this includes all the options that you need to modify from their default parameters. In the preceding example, we outline only a small subset of these options, as shown here:

- `inventory`*:* This option modifies the default inventory file that Ansible searches in order to find its inventory (by default, this is `/etc/ansible/hosts`). We adjust this option in order to let Ansible use our inventory file and stop using the `-i` operator to specify our inventory during each playbook run.
- `vault_password_file`: This option sets the file that has the secret password for encrypting and decrypting `ansible-vault` secrets. This option removes the need to run Ansible playbooks with the `--vault-id` operator when using `ansible-vault`-encrypted variables.

- `gathering = explicit`: By default, Ansible runs a setup module to gather facts regarding the managed nodes while the playbook is running. This setup module is not compatible with network nodes since this module requires a Python interpreter on the managed nodes. By setting fact gathering to `explicit`, we disable this default behavior.

See also...

For more information regarding Ansible's configuration settings, please consult the following URL:

`https://docs.ansible.com/ansible/latest/reference_appendices/config.html#ansible-configuration-settings`

Using Ansible Roles

Ansible Roles promotes code reusability and provides a simple method for packaging Ansible code in a simple way that can be shared and consumed. An Ansible role is a collection of all the required Ansible tasks, 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. In this recipe, we will outline how to create an Ansible role and how to use it in our playbooks.

How to do it...

1. Inside the `ch1_ansible` folder, create a new folder called `roles` and create a new role called `basic_config`, as shown here:

```
$ mkdir roles
$ cd roles
$ ansible-galaxy init basic_config
```

2. Update the `basic_config/vars/main.yml` file with the following variable:

```
$ cat roles/basic_config/vars/main.yml

---
config_dir: basic_config
```

3. Update the `basic_config/tasks/main.yml` file with the following tasks:

```
$ cat roles/basic_config/tasks/main.yml

---
  - name: Create Configs Directory
    file:
      path: "{{ config_dir }}"
      state: directory
    run_once: yes

  - name: Generate Cisco Basic Config
    template:
      src: "{{os}}.j2"
      dest: "{{config_dir}}/{{inventory_hostname}}.cfg"
```

4. Inside the `basic_config/templates` folder, create the following structure:

```
$ tree roles/basic_config/templates/

roles/basic_config/templates/
├── ios.j2
└── junos.j2

$ cat roles/basic_config/templates/ios.j2
hostname {{ hostname }}
!
{% for server in ntp_servers %}
ntp {{ server }}
{% endfor %}
```

5. Create a new playbook, `pb_ansible_role.yml`, with the following content to use our role:

```
$ cat pb_ansible_role.yml
---
  - name: Build Basic Config Using Roles
    hosts: all
    connection: local
    roles:
      - basic_config
```

How it works...

In this recipe, we start by creating the `roles` directory within our main folder. By default, when using roles, Ansible will look for roles in the following location in this order:

- The `roles` folder within the current working directory
- `/etc/ansible/roles`

Consequently, we create the `roles` folder within our current working directory (`ch1_ansible`) in order to host all the roles that we will create in this folder. We create the role using the `ansible-galaxy` command with the `init` option and the role name (`basic_config`), which will create the following role structure inside our `roles` folder:

```
$ tree roles/
roles/
└── basic_config
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── README.md
    ├── tasks
    │   └── main.yml
    ├── templates
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml
```

As can be seen from the preceding output, this folder structure is created using the `ansible-galaxy` command and this command builds the role in keeping with the best practice role layout. Not all these folders need to have a functional role that we can use, and the following list outlines the main folders that are commonly used:

- The `tasks` folder: This contains the `main.yml` file, which lists all the tasks that should be executed when we use this role.
- The `templates` folder: This contains all the Jinja2 templates that we will use as part of this role.

- The `vars` folder: This 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.
- The `handlers` folder: This contains the `main.yml` file, which includes all the handlers that should run as part of this role.

The role that we created has a single purpose, which is to build the basic configuration for our devices. In order to accomplish this task, we need to define some Ansible tasks as well as use a number of Jinja2 templates in order to generate the basic configuration for the devices. We list all the tasks that we need to run in the `tasks/main.yml` file and we include all the necessary Jinja2 templates in the `templates` folder. We define any requisite variable that we will use in our role in the `vars` folder.

We create a new playbook that will use our new role in order to generate the configuration for the devices. We call all the roles that we want to run as part of our playbook in the `roles` parameter. In our case, we have a single role that we want to run, which is the `basic_config` role.

Once we run our playbook, we can see that a new directory called `basic_config` is created with the following content:

```
$ tree basic_config/
basic_config/
├── csr1.cfg
├── csr2.cfg
├── mx1.cfg
└── mx2.cfg
```

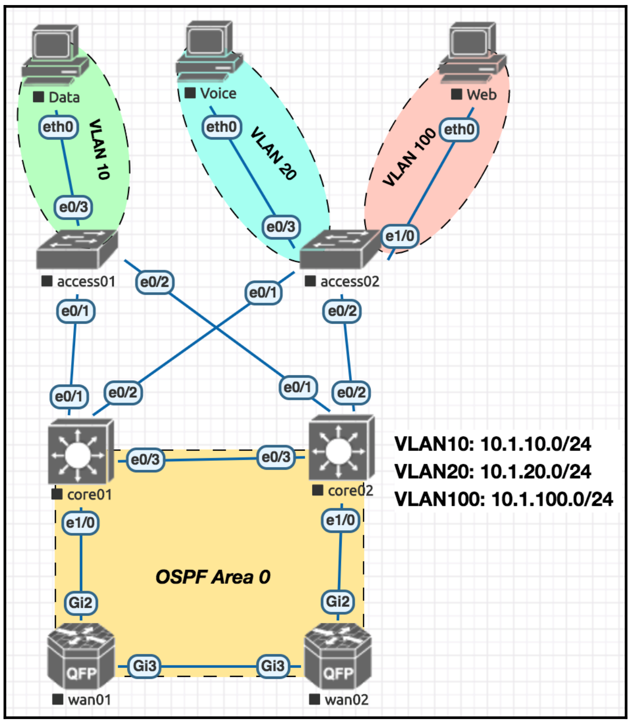
See also

For more information regarding Ansible Roles, please consult the following URL:

`https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html`

2
Managing Cisco IOS Devices Using Ansible

In this chapter, we will outline how to automate Cisco IOS-based devices using Ansible. We will explore the different modules available in Ansible to automate configuration and collect network information from Cisco IOS devices. This chapter will be based on the following sample network diagram, and we will walk through how we can implement this network design using Ansible:



The following table outlines the management IP addresses on the Cisco nodes, which Ansible will use to connect to the devices:

Device	Role	Vendor	MGMT Port	MGMT IP
access01	Access switch	Cisco IOS 15.1	Ethernet0/0	172.20.1.18
access02	Access switch	Cisco IOS 15.1	Ethernet0/0	172.20.1.19
core01	Core switch	Cisco IOS 15.1	Ethernet0/0	172.20.1.20
core02	Core switch	Cisco IOS 15.1	Ethernet0/0	172.20.1.21
wan01	WAN router	Cisco IOS–XE 16.6.1	GigabitEthernet1	172.20.1.22
wan02	WAN router	Cisco IOS–XE 16.6.1	GigabitEthernet1	172.20.1.23

The main recipes covered in this chapter are as follows:

- Building an Ansible network inventory
- Connecting to Cisco IOS devices
- Configuring basic system information
- Configuring interfaces on IOS devices
- Configuring L2 VLANS on IOS devices
- Configuring trunk and access interfaces
- Configuring interface IP addresses
- Configuring OSPF on IOS devices
- Collecting IOS device facts
- Validating network reachability on IOS devices
- Retrieving operational data from IOS devices
- Validating network states with pyATS and Ansible

Technical requirements

The code files for this chapter can be found here:

`https://github.com/PacktPublishing/Network-Automation-Cookbook/tree/master/ch2_ios`

The software releases that this chapter is based on are as follows:

- Cisco IOS 15.1
- Cisco IOS–XE 16.6.1
- Ansible 2.9
- Python 3.6.8

Check out the following video to see the Code in Action:
`https://bit.ly/34F8xPW`

Building an Ansible network inventory

In this recipe, we will outline how to build and structure the Ansible inventory to describe the network setup outlined in the previous section.

Getting ready

Make sure that Ansible is already installed on the control machine.

How to do it...

1. Create a new directory with the following name: `ch2_ios`.
2. Inside this new folder, create the `hosts` file with the following content:

```
$ cat hosts
 [access]
 access01 Ansible_host=172.20.1.18
 access02 Ansible_host=172.20.1.19

[core]
 core01 Ansible_host=172.20.1.20
 core02 Ansible_host=172.20.1.21

[wan]
 wan01 Ansible_host=172.20.1.22
 wan02 Ansible_host=172.20.1.23

[lan:children]
 access
 core

[network:children]
 lan
 wan
```

3. Create the `Ansible.cfg` file with the following content:

```
$ cat Ansible.cfg

[defaults]
 inventory=hosts
 retry_files_enabled=False
 gathering=explicit
```

How it works...

We built the Ansible inventory using the `hosts` file, and we defined multiple groups in order to group the different devices in our topology in the following manner:

- We created the `access` group, which has both access switches (`access01` and `access02`) in our topology.
- We created the `core` group, which groups all core switches that will act as the L3 termination for all the VLANs on the access switches.
- We created the `wan` group, which groups all our Cisco IOS–XE routes, which will act as our wan routers.
- We created another group called `lan`, which groups both access and core groups.
- We created the `network` group, which groups both `lan` and `wan` groups.

Finally, we created the `Ansible.cfg` file and configured it to point to our `hosts` file to be used as an Ansible inventory file. We disabled the setup module, which is not required when running Ansible against network nodes.

Connecting to Cisco IOS devices

In this recipe, we will outline how to connect to Cisco IOS devices from Ansible via SSH in order to start managing devices from Ansible.

Getting ready

In order to follow along with this recipe, an Ansible inventory file should be constructed as per the previous recipe. IP reachability between the Ansible control machine and all the devices in the network must be configured.

How to do it...

1. Inside the `ch2_ios` directory, create the `groups_vars` folder.
2. Inside the `group_vars` folder, create the `network.yml` file with the following content:

```
$cat network.yml
Ansible_network_os: ios
Ansible_connection: network_cli
Ansible_user: lab
Ansible_password: lab123
Ansible_become: yes
Ansible_become_password: admin123
Ansible_become_method: enable
```

3. On all IOS devices, ensure that the following is configured to set up SSH access:

```
!
 hostname <device_hostname>
 !
 ip domain name <domain_name>
 !
 username lab secret 5 <password_for_lab_user>.
 !
 enable secret 5 <enable_password>.
 !
 line vty 0 4
 login local
 transport input SSH
 !
```

4. Generate SSH keys on the Cisco IOS devices from the config mode, as shown in the following code snippet:

```
(config)#crypto key generate rsa
 Choose the size of the key modulus in the range of 360 to 4096 for
your
 General Purpose Keys. Choosing a key modulus greater than 512 may
take
 a few minutes.
How many bits in the modulus [512]: 2048
 % Generating 2048 bit RSA keys, keys will be non-exportable...
 [OK] (elapsed time was 0 seconds)
```

5. Update the `Ansible.cfg` file with the following highlighted parameters:

```
$ cat Ansible.cfg
[defaults]
 host_key_checking=False
```

How it works...

In our sample network, we will use SSH to set up the connection between Ansible and our Cisco devices. In this setup, Ansible will use SSH in order to establish the connection to our Cisco devices with a view to start managing it. We will use username/password authentication in order to authenticate our Ansible control node with our Cisco devices.

On the Cisco devices, we must ensure that SSH keys are present in order to have a functional SSH server on the Cisco devices. The following code snippet outlines the status of the SSH server on the Cisco device prior to generating the SSH keys:

```
wan01#show ip SSH
SSH Disabled - version 2.0
%Please create RSA keys to enable SSH (and of atleast 768 bits for SSH v2).
Authentication methods:publickey,keyboard-interactive,password
Authentication Publickey Algorithms:x509v3-SSH-rsa,SSH-rsa
Hostkey Algorithms:x509v3-SSH-rsa,SSH-rsa
Encryption Algorithms:aes128-ctr,aes192-ctr,aes256-ctr
MAC Algorithms:hmac-sha2-256,hmac-sha2-512,hmac-sha1,hmac-sha1-96
KEX Algorithms:diffie-hellman-group-exchange-sha1,diffie-hellman-group14-
sha1
Authentication timeout: 120 secs; Authentication retries: 3
Minimum expected Diffie Hellman key size : 2048 bits
IOS Keys in SECSH format(SSH-rsa, base64 encoded): NONE
```

Once we create the SSH keys, the SSH server on the Cisco device is operational, and is ready to accept an SSH connection from the Ansible control node.

On the Ansible machine, we include all the variables required to establish the SSH connection to the managed devices in the `network.yml` file. As per our inventory file, the network group includes all the devices within our topology, and so all the attributes that we configure in this file will apply to all the devices in our inventory. The following is a breakdown of the attributes that we included in the file:

- `Ansible_connection`: This establishes how Ansible connects to the device. In this scenario, we set it to `network_cli` to indicate that we will use SSH to connect to a network device.

- `Ansible_network_os`: When using `network_cli` as the connection plugin to connect to the network device, we must indicate which network OS Ansible will be connecting to, so as to use the correct SSH parameters with the devices. In this scenario, we will set it to `ios`, since all the devices in our topology are IOS-based devices.
- `Ansible_user`: This parameter specifies the username that Ansible will use to establish the SSH session with the network device.
- `Ansible_password`: This parameter specifies the password that Ansible will use to establish the SSH session with the network device.
- `Ansible_become`: This instructs Ansible to use the `enable` command to enter privileged mode when configuring or executing `show` commands on the managed device. We set this to `yes` in our context, since we will require privileged mode to configure the devices.
- `Ansible_become_password`: This specifies the `enable` password to use in order to enter privileged mode on the managed IOS device.
- `Ansible_become_method`: This option specifies the method to use in order to enter privileged mode. In our scenario, this is the `enable` command on IOS devices.

In this recipe, I have defined the SSH password and the `enable` passwords as plain text just for simplicity; however, this is highly discouraged. We should use `Ansible-vault` to secure the passwords, as outlined in the *Ansible Vault* recipe in the previous chapter.

On the Cisco devices, we set up the required username and password so that Ansible can open an SSH connection to the managed Cisco IOS devices. We also configure an `enable` password to be able to enter privileged mode, and to make configuration changes. Once we apply all of these configurations to the devices, we are ready to set up Ansible.

In any SSH connection, when an SSH client (Ansible control node in our case) connects to an SSH server (Cisco devices in our case), the server sends a copy of its public key to the client before the client logs in. This is used to establish the secure channel between the client and the server, and to authenticate the server to the client in order to prevent any man-in-the-middle attacks. So, at the start of a new SSH session involving a new device, we see the following prompt:

```
$SSH lab@172.20.1.18
The authenticity of host '172.20.1.18 (172.20.1.18)' can't be established.
RSA key fingerprint is SHA256:KnWOalnENZfPokYYdIG3Ogm9HDnXIwjh/it3cqdiRRQ.
RSA key fingerprint is MD5:af:18:4b:4e:84:19:a6:8d:82:17:51:d5:ee:eb:16:8d.
Are you sure you want to continue connecting (yes/no)?
```

When the SSH client initiates the SSH connection to the client, the SSH server sends its public key to the client in order to authenticate itself to the client. The client searches for the public key in its local known `hosts` files (in the `~/.SSH/known_hosts` or `/etc/SSH/SSH_known_hosts` files). In the event that it does not find the public key for this machine in its local known `hosts` file, it will prompt the user to add this new key to its local database, and this is the prompt that we see when we initiate the SSH connection.

In order to simplify the SSH connection setup between the Ansible control node and its remotely managed `hosts`, we can disable this host checking. We can do this by telling Ansible to ignore host keys and not to add them to the known `hosts` files by setting `host_key_checking` to `False` in the `Ansible.cfg` configuration file.

Disabling host key checking is not a best practice, and we are only showing it as it is a lab setup. In the next section, we will outline an alternative method to establish the SSH connection between Ansible and its remote managed devices.

There's more...

If we need to verify the identity of the SSH `hosts` that we will connect to, and thereby enable `host_key_checking`, we can automate the addition of the SSH public key of the remote managed `hosts` to the `~/.SSH/known_hosts` file using Ansible. We create a new Ansible playbook that will run on the Ansible control machine to connect to the remote devices using the `ssk-keyscan` command. We then collect the SSH public keys for the remote machines and add them to the `~/.SSH/known_hosts` file. The method is outlined here:

1. Create a new `playbook pb_gather_SSH_keys.yml` file and add the following play:

```
- name: "Play2: Record Keys in Known Hosts file"
  hosts: localhost
  vars:
  - hosts_file: "~/.SSH/known_hosts"
tasks:
  - name: create know hosts file
  file:
  path: "{{ hosts_file }}"
  state: file
  changed_when: false
```

2. Update the playbook and add another play within the same playbook to save and store the SSH public keys for the remote managed nodes:

```
- name: "Play2: Record Keys in Known Hosts file"
  hosts: localhost
  vars:
    - hosts_file: "~/.SSH/known_hosts"
  tasks:
    - name: create know hosts file
      file:
        path: "{{ hosts_file }}"
        state: file
      changed_when: false
    - name: Populate the known_hosts file
      blockinfile:
        block: |
          {% for host in groups['all'] if
hostvars[host].SSH_keys.stdout != ''
%}
          {{ hostvars[host].SSH_keys.stdout}}
          {% endfor %}
        path: "{{ hosts_file }}"
        create: yes
```

In our new playbook, we have a play that targets all our managed devices by setting the `hosts` parameter to `all`. In this play, we have a single task, which we run on the Ansible control node (using the `delegate_to` localhost) to issue the `SSH-keyscan` command, which returns the SSH public key for the remote device, as shown in the following code:

```
$ SSH-keyscan 172.20.1.22

# 172.20.1.22:22 SSH-2.0-Cisco-1.25
 172.20.1.22 SSH-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDTwrH4phzRnW/RsC8eXMh/accIErRfkgffDWBGSdEX0r9
EwAa6p2uFMWj8dq6kvrREuhqpgFyMoWmpgdx5Cr+10kEonr8So5yHhOhqG1SJO9RyzAb93H0P0r
o5DXFK8A/Ww+m++avyZ9dShuWGxKj9CDM6dxFLg9ZU/9vlzkwtyKF/+mdWNGoSiCbcBg7LrOgZ7
Id7oxnhEhkrVIa+IxxGa5Pwc73eR45Uf7QyYZXPC0RTOm6aH2f9+8oj+vQMsAzXmeudpRgAu151
qUH3nEG9HIgUxwhvmi4MaTC+psmsGg2x26PKTOeX9eLs4RHquVS3nySwv4arqVzDqWf6aruJ
```

In this task, we are using `delegate_to` as being equal to `localhost,` as Ansible will try to connect to the remote devices and issue the command on the remote device by default. In our case, this is not what we need; we need to issue this command from the Ansible control node. So, we use `delegate_to` as being equal to `localhost` in order to enforce this behavior.

We run the second play on the Ansible control host by setting `hosts` to `localhost,` and we execute tasks to create the known hosts file (if not already present) and to populate this file with the data that we captured in the first play using the `SSH_keys` variable. We run this playbook on the Ansible control machine to store the SSH keys from the remotely managed nodes prior to running any of our playbooks.

Configuring basic system information

In this recipe, we will outline how we can configure basic system parameters on Cisco IOS devices, such as setting the hostname, DNS server, and NTP servers. Following the network setup that we outlined at the start of this chapter, we will configure the following information on all the Cisco IOS devices:

- DNS servers 172.20.1.250 and 172.20.1.251
- NTP server 172.20.1.17

Getting ready

An Ansible inventory file must be present, as well as the configuration for Ansible to connect to the Cisco IOS devices via SSH.

How to do it...

1. To the `group_vars/network.yml` file, add the following system parameters:

```
$ cat group_vars/network.yml
<-- Output Trimmed for brevity ------>
name_servers:
 - 172.20.1.250
 - 172.20.1.251
ntp_server: 172.20.1.17
```

2. Create a new playbook called `pb_build_network.yml` with the following information:

```
$ cat pb_build_network.yml
---
- name: "PLAY 1: Configure All Lan Switches"
  hosts: lan
  tags: lan
  tasks:
  - name: "Configure Hostname and Domain Name"
    ios_system:
      hostname: "{{ inventory_hostname }}"
      domain_name: "{{ domain_name }}"
      lookup_enabled: no
      name_servers: "{{ name_servers }}"
  - name: "Configure NTP"
    ios_ntp:
      server: "{{ ntp_server }}"
      logging: true
      state: present
```

How it works...

In the `network.yml` file, we define the `name_servers` variable as a list of DNS servers, and we also define the `ntp_servers` variable, which defines the NTP servers that we want to configure on the IOS devices. Defining these parameters in the `network.yml` file applies these variables to all the devices within the network group.

We create a playbook and the first play targets all the `hosts` in the `lan` group (this includes both access and core devices) and, within this play, we reference two tasks:

- `ios_system`: This sets the hostname and the DNS servers on the devices.
- `ios_ntp`: This configures the NTP on the IOS devices and enables logging for NTP events.

Both these modules are declarative Ansible modules in which we just identify the state pertaining to our infrastructure. Ansible converts this declaration into the necessary IOS commands. The modules retrieve the configuration of the devices and compare the current state with our intended state (to have DNS and NTP configured on them) and then, if the current state does not correspond to the intended state defined by these modules, Ansible will apply the needed configuration to the devices.

When we run these tasks on all the LAN devices, the following configuration is pushed to the devices:

```
!
 ip name-server 172.20.1.250 172.20.1.251
 no ip domain lookup
 ip domain name lab.net
 !
 ntp logging
 ntp server 172.20.1.17
 !
```

See also...

For more information regarding the `ios_system` and `ios_ntp` modules, as well as the different parameters supported by these modules, please consult the following URLs:

- `https://docs.Ansible.com/Ansible/latest/modules/ios_system_module.html`
- `https://docs.Ansible.com/Ansible/latest/modules/ios_ntp_module.html`

Configuring interfaces on IOS devices

In this recipe, we will outline how to configure the basic interface properties on Cisco IOS-based devices, such as setting the interface description, the interface **maximum transmission unit** (**MTU**), and enabling `interfaces`. We will configure all the links within our topology as having a link MTU of 1,500 and to be fully duplex.

Getting ready

To follow along with this recipe, an Ansible inventory is assumed to be already set up, as is IP reachability between the Ansible control node with the Cisco devices in place.

How to do it...

1. In the `group_vars/network.yml` file, add the following content to define the generic interface parameters:

```
$ cat group_vars/network.yml
<-- Output Trimmed for brevity ------>
```

```
intf_duplex: full
intf_mtu: 1500
```

2. Create a new file, `lan.yml`, under the `group_vars` folder, with the following data to define the `interfaces` on our Cisco devices:

```
$ cat group_vars/lan.yaml

interfaces:
  core01:
    - name: Ethernet0/1
      description: access01_e0/1
      mode: trunk
    - name: Ethernet0/2
      description: access02_e0/1
      mode: trunk
    - name: Ethernet0/3
      description: core01_e0/3
      mode: trunk
 <--   Output Trimmed for brevity ------>
  access01:
    - name: Ethernet0/1
      description: core01_e0/1
      mode: trunk
    - name: Ethernet0/2
      description: core02_e0/1
      mode: trunk
    - name: Ethernet0/3
      description: Data_vlan
      mode: access
      vlan: 10
```

3. Update the `pb_build_network.yml` playbook file with the following task to set up the `interfaces`:

```
    - name: "P1T3: Configure Interfaces"
      ios_interface:
        name: "{{ item.name }}"
        description: "{{ item.description }}"
        duplex: "{{ intf_duplex }}"
        mtu: "{{ intf_mtu }}"
        state: up
      loop: "{{ interfaces[inventory_hostname] }}"
      register: ios_intf
```

How it works...

In this recipe, we outline how to configure the physical interfaces on IOS devices. We first declare the generic parameters (interface duplex and MTU) that apply to all the interfaces. These parameters are defined under the `network.yml` file. Next, we define all the interface-specific parameters for all our LAN devices under the `lan.yml` file to be applied to all devices. All these parameters are declared in the `interfaces` dictionary data structure.

We update our playbook with a new task to configure the physical parameters for all of our LAN devices in our network. We use the `ios_interface` module to provision all the `interface` parameters, and we loop over all the `interfaces` in each node using the `interfaces` data structure. We set the state to `up` in order to indicate that the `interface` should be present and operational.

See also...

For more information regarding the `ios_interface` module, and the different parameters supported by these modules, please consult the following URL: `https://docs.Ansible.com/Ansible/latest/modules/ios_interface_module.html`

Configuring L2 VLANs on IOS devices

In this recipe, we will outline how to configure L2 VLANs on Cisco IOS devices, as per the network topology discussed in the introduction to this chapter. We will outline how to declare VLANs as Ansible variables, and how to use suitable Ansible modules to provision these VLANs on the network.

Getting ready

We will be building on the previous recipes discussed in this chapter to continue to configure the L2 VLANs on all the LAN devices within our sample topology.

How to do it...

1. Update the `group_vars/lan.yml` file with the VLAN definition, as outlined in the following code:

```
$ cat group_vars/lan.yaml

vlans:
  - name: Data
    vlan_id: 10
  - name: Voice
    vlan_id: 20
  - name: Web
    vlan_id: 100
```

2. Update the `pb_build.yml` playbook with the following task to provision the VLANs:

```
  - name: "P1T4: Create L2 VLANs"
    ios_vlan:
      vlan_id: "{{ item.vlan_id }}"
      name: "{{ item.name  }}"
    loop: "{{ vlans }}"
    tags: vlan
```

How it works...

In the `group_vars/lan.yml` file, we define a `vlans` list data structure that holds the VLAN definition that we need to apply to all our core and access switches. This variable will be available for all the core and access switches, and Ansible will use this variable in order to provision the required VLANs on the remote devices.

We use another declarative module, `ios_vlan`, which takes the VLAN definition (its name and the VLAN ID) and configures these VLANs on the remote managed device. It pulls the existing configuration from the device and compares it with the list of devices that need to be present, while only pushing the delta.

We use the `loop` construct to go through all the items in the `vlans` list, and configure all the respective VLANs on all the devices.

After running this task on the devices, the following is the output from one of the access switches:

```
access01#sh vlan
VLAN Name                             Status    Ports
---- -------------------------------- --------- ---------------------------
----
1    default                          active    Et1/0, Et1/1, Et1/2, Et1/3
10   Data                             active    Et0/3
20   Voice                            active
100  Web                              active
```

Configuring trunk and access interfaces

In this recipe, we will show how to configure access and trunk interfaces on Cisco IOS-based devices, and how to map interfaces to an access VLAN, as well as how to allow specific VLANs on the trunks.

Getting ready

Following our sample topology, we will configure the interfaces on the devices. As shown in this table, we are only showing the VLANs for `access01` and `core01`— the other devices are exact replicas:

Device	Interface	Mode	VLANs
Core01	Ethernet0/1	Trunk	10,20,100
Core01	Ethernet0/2	Trunk	10,20,100
Core01	Ethernet0/3	Trunk	10,20,100,200
Access01	Ethernet0/1	Trunk	10,20,100
Access01	Ethernet0/2	Trunk	10,20,100
Access01	Ethernet0/3	Access	10

How to do it...

1. Create a new `core.yml` file under `group_vars` and include the following `core_vlans` definition:

```
core_vlans:
  - name: l3_core_vlan
    vlan_id: 200
    interface: Ethernet0/3
```

2. Update the `pb_build_network.yml` playbook with the following tasks to configure all trunk ports:

```
  - name: "Configure L2 Trunks"
    ios_l2_interface:
      name: "{{ item.name }}"
      mode: "{{ item.mode }}"
      trunk_allowed_vlans: "{{ vlans | map(attribute='vlan_id') |
join(',') }}"
      state: present
    loop: "{{ interfaces[inventory_hostname] |
selectattr('mode','equalto','trunk') | list }}"
  - name: "Enable dot1q Trunks"
    ios_config:
      lines:
        - switchport trunk encapsulation dot1q
      parents: interface {{item.name}}
    loop: "{{ interfaces[inventory_hostname] |
selectattr('mode','equalto','trunk') | list }}"
    tags: dot1q
```

3. Update the playbook with the following task to configure all access ports:

```
  - name: "Configure Access Ports"
    ios_l2_interface:
      name: "{{ item.name }}"
      mode: "{{ item.mode}}"
      access_vlan: "{{ item.vlan }}"
      state: present
    loop: "{{ interfaces[inventory_hostname] |
selectattr('mode','equalto','access') | list }}"
```

How it works...

We are using the same data structure in the `lan.yml` file that defines all the interfaces within the LAN network and describes their type (access/trunk). In the case of access ports, we define which access interface is part of which VLAN. We will reference this list data structure to configure the access and trunk ports on all the devices within the `lan` group. The interfaces within our `layer2` network are one of the following two options:

Access:

- We use `ios_l2_interface` with the `access_vlan` parameter to configure the correct access VLAN on the interface.
- We select only the access interfaces for each device using the `selectattr jinja2` filter, and we match only one interface with a mode equal to `access`, and we loop over this list for each device.

Trunk:

- We use `ios_l2_interface` with the `trunk_allowed_vlans` parameter to add all the VLANs to the trunk ports, on both access and core switches.
- We create the permitted VLAN list using the Jinja2 `map` and `join` filters and we apply this filter to the `vlans` list data structure. This outputs a string similar to the following: `10,20,100`.
- We select only the trunk ports using the `selectattr` Jinja2 filter from the interface's data structure per node.
- We need to configure these trunks as `dot1q` ports; however, this attribute is still not enabled on `ios_l2_interface`. Hence, we use another module, `ios_config`, to send the required Cisco IOS command to set up the `dot1q` trunks.

The following output outlines the configuration applied to the `access01` device as an example for both access and trunk ports:

```
!
interface Ethernet0/3   >> Access Port
 description Data_vlan
 switchport access vlan 10
 switchport mode access

 !
interface Ethernet0/1    >> Trunk Port
 description core01_e0/1
 switchport trunk encapsulation dot1q
```

```
    switchport trunk allowed vlan 10,20,100
    switchport mode trunk
```

See also...

For more information regarding `ios_l2_interface` and the different parameters supported by these modules, please consult the following URL:

`https://docs.Ansible.com/Ansible/latest/modules/ios_l2_interface_module.html`

Configuring interface IP addresses

In this recipe, we will explore how to configure the interface IP address on Cisco IOS devices. We will use the sample topology to configure the VLAN interfaces on both the core switches. We will outline how to configure VRRP between the core switches for all the VLAN interfaces. We will configure the following IP addresses:

Interface	Prefix	VRRP IP address
VLAN10	`10.1.10.0/24`	`10.1.10.254`
VLAN20	`10.1.20.0/24`	`10.1.20.254`
VLAN100	`10.1.100.0/24`	`10.1.100.254`

Getting ready

This recipe assumes that the interface and VLANs are configured as per the previous recipes in this chapter.

How to do it...

1. Update the `group_vars/core.yml` file with following data to define the SVI interfaces:

```
$ cat group_vars/core.yml
<-- Output Trimmed for brevity ------>
svi_interfaces:
  - name: Vlan10
    ipv4: 10.1.10.0/24
    vrrp: yes
    ospf: passive
 -  name: Vlan20
```

```
      ipv4: 10.1.20.0/24
      vrrp: yes
      ospf: passive
   -  name: Vlan100
      ipv4: 10.1.100.0/24
      vrrp: yes
      ospf: passive
```

2. Create `core01.yml` and `core02.yml` files under the `host_vars` folder and add the following content:

```
$ cat host_vars/core01.yml
 hst_svi_id: 1
 hst_vrrp_priority: 100
$ cat host_vars/core02.yml
 hst_svi_id: 2
 hst_vrrp_priority: 50
```

3. Update the `pb_build_network.yml` playbook with the following tasks to create and enable the L3 SVI interfaces:

```
- name: "PLAY 2: Configure Core Switches"
  hosts: core
  tags: l3_core
  tasks:
<-- Output Trimmed for brevity ------>
    - name: "Create L3 VLAN Interfaces"
      ios_l3_interface:
        name: "{{item.name }}"
        ipv4: "{{item.ipv4 | ipv4(hst_svi_id)}}"
      loop: "{{svi_interfaces}}"
      tags: l3_svi
    - name: "Enable the VLAN Interfaces"
      ios_interface:
        name: "{{ item.name }}"
        state: up
      loop: "{{ svi_interfaces }}"
```

4. Update the playbook with the following task to set up VRRP configuration on the SVI interfaces:

```
    - name: "Create VRRP Configs"
      ios_config:
        parents: interface {{ item.name }}
        lines:
          - vrrp {{item.name.split('Vlan')[1]}} priority {{
hst_vrrp_priority }}
          - vrrp {{item.name.split('Vlan')[1]}} ip {{item.ipv4 |
ipv4(254)|ipaddr('address')}}
      loop: "{{svi_interfaces | selectattr('vrrp','equalto',true) |
list }}"
```

How it works...

In this section, we are configuring the IP addresses for the L3 VLAN interfaces on the core switches, as well as configuring VRRP on all the L3 VLAN interfaces to provide L3 redundancy.

We are using a new list data structure called `svi_interfaces`, which describes all the SVI interfaces with L3 IP addresses, and also some added parameters to control both the VRRP and OSPF configured on these interfaces. We also set up two new variables on each core router, `hst_svi_id` and `hst_vrrp_priority`, which we will use in the playbook to control the IP address on each core switch, as well as the VRPP priority.

We use the `ios_l3_interface` Ansible module to set the IPv4 addresses on the VLAN interfaces. On each core switch, we loop over the `svi_interfaces` data structure, and for each VLAN we configure the IPv4 address on the corresponding VLAN interface. We determine which IP address is configured on each router using the Ansible `ipaddr` filter, along with the `hst_svi_id` parameter `{{item.ipv4 | ipv4(hst_svi_id)}}`. So, for example, for VLAN10, we will assign `10.1.10.1/24` for `core01` and `10.1.10.2/24` for `core02`.

When first creating the VLAN interface on Cisco IOS devices, they are in a state of shutdown, so we need to enable them. We use the `ios_interface` module to enable the interfaces.

For the VRRP part, we return to using the `ios_config` module to set up the VRRP configuration on all the VLAN interfaces, and we use `hst_vrrp_priority` to correctly set up `core01` as the master VRRP for all VLANs.

The following is a sample of the configuration that is pushed on the devices after running the playbook:

```
Core01
 ========
 !
 interface Vlan10
 ip address 10.1.10.1 255.255.255.0
 vrrp 10 ip 10.1.10.254
 !
```

```
Core02
 =======
 !
 interface Vlan10
 ip address 10.1.10.2 255.255.255.0
 vrrp 10 ip 10.1.10.254
 vrrp 10 priority 50
```

See also...

For more information regarding `ios_l3_interface` and the different parameters supported by these modules, please consult the following URL:

`https://docs.Ansible.com/Ansible/latest/modules/ios_l3_interface_module.html`

Configuring OSPF on IOS devices

In this recipe, we will outline how to configure OSPF on Cisco IOS devices with Ansible. Using our sample network topology, we will set up OSPF between core switches and WAN routers, as well as advertising the SVI interface via OSPF.

Getting ready

This recipe assumes that all the interfaces are already configured with the correct IP addresses and are following the same procedures outlined in previous recipes.

How to do it...

1. Update the `group_vars/core.yml` file with the following data to define core links between core switches and WAN routers:

```
core_l3_links:
  core01:
    - name: Ethernet1/0
      description: wan01_Gi2
      ipv4: 10.3.1.0/30
      ospf: yes
      ospf_metric: 100
      peer: wan01
  core02:
        - name: Ethernet1/0
          description: wan02_Gi2
          ipv4: 10.3.1.4/30
          ospf: yes
          ospf_metric: 200
          peer: wan02
```

2. Update the `pb_build_network.yml` playbook with the following tasks to set up OSPF:

```
- name: "PLAY 2: Configure Core Switches"
  hosts: core
  tags: l3_core
  tasks:
< -------- Snippet -------- >
    - name: "P2T9: Configure OSPF On Interfaces"
      ios_config:
        parents: interface {{ item.name }}
        lines:
          - ip ospf {{ ospf_process }} area {{ ospf_area }}
          - ip ospf network point-to-point
          - ip ospf cost {{item.ospf_metric |
default(ospf_metric)}}
      loop: "{{ (svi_interfaces +
core_l3_links[inventory_hostname]) | selectattr('ospf') | list }}"
    - name: "P2T10: Configure OSPF Passive Interfaces"
      ios_config:
        parents: router ospf {{ ospf_process }}
        lines: passive-interface {{item.name}}
      loop: "{{ (svi_interfaces +
core_l3_links[inventory_hostname]) |
selectattr('ospf','equalto','passive') | list }}"
```

How it works...

We created another dictionary data structure in the `core.yml` file that describes the L3 links between the core switches and the WAN routers. We specified whether they will run OSPF and what the OSPF metric is on these links.

Currently, Ansible doesn't provide a declarative module to manage OSPF configuration on IOS-based devices. Therefore, we need to push the required configuration using the `ios_config` module. We created two separate tasks using `ios_config` in order to push the OSPF-related configuration on each device. In the first task, we configured the interface-related parameters under each interface, and we looped over both the `svi_interface` and `core_l3_interfaces` data structures to enable OSPF on all the OSPF-enabled interfaces. We used the Jinja2 `selectattr` filter to select all the interfaces that have the OSPF attribute set to `yes`/`true`.

In the last task, we applied the passive interface configuration to all the interfaces that have the passive flag enabled on them. We used the Jinja2 `selectattr` filter to select only those interfaces with the passive parameter set to `yes`/`true`.

Collecting IOS device facts

In this recipe, we will outline how to collect device facts from Cisco devices with Ansible. This information includes the serial number, IOS version, and all the interfaces on the devices. Ansible executes several commands on managed IOS devices in order to collect this information.

Getting ready

The Ansible controller must have IP connectivity with the managed network devices, and SSH must be enabled on the IOS devices.

How to do it...

1. Create a new playbook called `pb_collect_facts.yml` in the same `ch2_ios` folder with the following information:

```
---
- name: "PLAY 1: Collect Device Facts"
  hosts: core,wan
  tasks:
    - name: "P1T1: Gather Device Facts"
      ios_facts:
      register: device_facts
    - debug: var=device_facts
```

How it works...

We run this new playbook against all nodes within the `core` and `wan` group, and we use the `ios_facts` module to collect the information from the managed IOS devices. In this recipe, we use the debug module to print out the information that was collected from the `ios_facts` module. The following is a subset of the information that was discovered:

```
ok: [core01 -> localhost] => {
  "Ansible_facts": {
    "net_all_ipv4_addresses": [
      "172.20.1.20",
< ---------- Snippet ------------ >
      "10.1.100.1"
    ],
    "net_hostname": "core01",
    "net_interfaces": {
 < ---------- Snippet ------------ >
      "Vlan10": {
        "bandwidth": 1000000,
        "description": null,
        "duplex": null,
        "ipv4": [
          {
            "address": "10.1.10.1",
            "subnet": "24"
          }
        ],
        "lineprotocol": "up",
        "macaddress": "aabb.cc80.e000",
        "mediatype": null,
        "mtu": 1500,
        "operstatus": "up",
        "type": "Ethernet SVI"
      },

    },
    "net_iostype": "IOS",
    "net_serialnum": "67109088",
    "net_system": "ios",
    "net_version": "15.1",
  }
 < ------------ Snippet ------------ >
 }
```

From the preceding output, we can see some of the main facts that the `ios_facts` module has captured from the devices, including the following:

- `net_all_ipv4_addresses`: This list data structure contains all the IPv4 addresses that are configured on all the `interfaces` on the IOS device.
- `net_interfaces`: This dictionary data structure captures the status of all of the `interfaces` on this device and their operational state, as well as other important information, such as a description and their operational state.
- `net_serialnum`: This captures the serial number of the device.
- `net_version`: This captures the IOS version running on this device.

There's more...

Using the information that is collected from the `ios_facts` module, we can generate structured reports for the current state of the network and use these reports in further tasks. In this section, we will outline how to modify our playbook to build this report.

Add a new task to the `pb_collect_facts.yml` playbook, as shown in the following code:

```
- name: "P1T2: Write Device Facts"
  blockinfile:
    path: ./facts.yml
    create: yes
    block: |
      device_facts:
      {% for host in play_hosts %}
      {% set node = hostvars[host] %}
        {{ node.Ansible_net_hostname }}:
          serial_number: {{ node.Ansible_net_serialnum }}
          ios_version: {{ node.Ansible_net_version }}
      {% endfor %}
      all_loopbacks:
      {% for host in play_hosts %}
      {% set node = hostvars[host] %}
      {% if node.Ansible_net_interfaces is defined %}
      {% if node.Ansible_net_interfaces.Loopback0 is defined %}
        - {{ node.Ansible_net_interfaces.Loopback0.ipv4[0].address }}
      {% endif %}
      {% endif %}
      {% endfor %}
  run_once: yes
  delegate_to: localhost
```

We use the blockinfile module to build a YAML file called facts.yml. We use Jinja2 expressions within the blockinfile module to customize and select the information we want to capture from the Ansible facts that were captured from the ios_facts task. When we run the pb_collect_facts.yml playbook, we generate the facts.yml file, which has the following data:

```
device_facts:
  wan01:
    serial_number: 90L4XVVPL7V
    ios_version: 16.06.01
  wan02:
    serial_number: 9UOFOO7FH19
    ios_version: 16.06.01
  core01:
    serial_number: 67109088
    ios_version: 15.1
  core02:
    serial_number: 67109104
    ios_version: 15.1
all_loopbacks:
  - 10.100.1.3
  - 10.100.1.4
  - 10.100.1.1
  - 10.100.1.2
```

See also...

For more information regarding ios_facts and the different parameters supported by these modules, please consult the following URL:

https://docs.Ansible.com/Ansible/latest/modules/ios_facts_module.html

Validating network reachability on IOS devices

In this recipe, we will outline how to validate network reachability via ping using Ansible. ICMP allows us to validate proper forwarding across our network. Using Ansible to perform this task provides us with a robust tool to validate proper traffic forwarding, since we can perform this task from each node simultaneously and collect all the results for further inspection.

Getting ready

This recipe is built based on the network setup that was outlined in the chapter introduction, and I am assuming that the network has already been built in accordance with all the previous recipes in this chapter.

How to do it...

1. Create a new playbook called `pb_net_validate.yml` and add the following task to store all SVI IP addresses:

```
---
  - name: "PLay 1: Validate Network Reachability"
    hosts: core,wan
    vars:
      host_id: 10
      packet_count: 10
    tasks:
      - name: "Get all SVI Prefixes"
        set_fact:
          all_svi_prefixes: "{{ svi_interfaces | selectattr('vrrp')
|
                                      map(attribute='ipv4') | list }}"
        run_once: yes
        delegate_to: localhost
        tags: svi
```

2. Update the `pb_net_validate.yml` playbook with the following task to ping all the SVI `interfaces`:

```
      - name: "Ping Hosts in all VLANs"
        ios_ping:
          dest: "{{ item | ipaddr(10) | ipaddr('address') }}"
        loop: "{{ all_svi_prefixes }}"
        ignore_errors: yes
        tags: svi
```

How it works...

In this playbook, we are using the `ios_ping` module, which logs into each node defined in our Ansible inventory, and pings the destination specified by the `dest` attribute. In this sample playbook, we would like to validate network reachability to a single host within the data, voice, and web VLANs, and we choose the tenth host in all these VLANs (just as an example). In order to build all the VLAN prefixes we set in the first task, we add a new variable called `all_svi_prefixes` and use multiple `jinja2` filters to collect only those prefixes that are running VRRP (so as to remove any core VLANs). We get only the IPv4 attributes for these SVI `interfaces`. The following are the contents of this new variable after running the first task:

```
ok: [core01 -> localhost] => {
  "all_svi_prefixes": [
    "10.1.10.0/24",
    "10.1.20.0/24",
    "10.1.100.0/24"
  ]
}
```

We supply this new list data structure to the `ios_ping` module and we specify that we need to ping the tenth host within each subnet. As long as the ping succeeds, the task will succeed. However, if there is a connectivity problem from the router/switch to this host, the task will fail. We are using the `ignore_errors` parameter in order to ignore any failure that might occur owing to the fact that the host is unreachable/down, and to run any subsequent tasks. The following code snippet outlines the successful run:

```
TASK [P1T2: Ping Hosts in all VLANs] *****************************
 ok: [core01] => (item=10.1.10.0/24)
 ok: [core02] => (item=10.1.10.0/24)
 ok: [wan01] => (item=10.1.10.0/24)
 ok: [wan02] => (item=10.1.10.0/24)
 ok: [core01] => (item=10.1.20.0/24)
 ok: [core02] => (item=10.1.20.0/24)
 ok: [core01] => (item=10.1.100.0/24)
 ok: [wan01] => (item=10.1.20.0/24)
 ok: [wan02] => (item=10.1.20.0/24)
 ok: [core02] => (item=10.1.100.0/24)
 ok: [wan01] => (item=10.1.100.0/24)
 ok: [wan02] => (item=10.1.100.0/24)
```

Retrieving operational data from IOS devices

In this recipe, we will outline how to execute operational commands on IOS devices and store these outputs to text files for further processing. This allows us to capture any operational commands from IOS devices during pre- or post-validation after we perform any deployment so that we can compare the results.

Getting ready

In order to follow along with this recipe, an Ansible inventory file should be in place and the network should already be set up as per the previous recipes.

How to do it...

1. Create a new playbook called `pb_op_cmds.yml` and populate it with the following tasks to create the directory structure to save the output from the devices:

```
---
- name: "Play 1: Execute Operational Commands"
  hosts: network
  vars:
    config_folder: "configs"
    op_folder: "op_data"
    op_cmds:
      - show ip ospf neighbor
      - show ip route
  tasks:
    - name: "P1T1: Build Directories to Store Data"
      block:
        - name: "Create folder to store Device config"
          file:
            path: "{{ config_folder }}"
            state: directory
        - name: "Create Folder to store operational commands"
          file:
            path: "{{ op_folder }}"
            state: directory
      run_once: yes
      delegate_to: localhost
```

2. Update the `pb_op_cmds.yml` playbook and populate it with the following tasks to retrieve the running configuration from the devices:

```
        - name: "P1T2: Get Running configs from Devices"
          ios_command:
            commands: show running-config
          register: show_run
        - name: "P1T3: Save Running Config per Device"
          copy:
            content: "{{ show_run.stdout[0] }}"
            dest: "{{ config_folder }}/{{ inventory_hostname }}.cfg"
```

3. Update the playbook and populate it with the following tasks to retrieve the operational commands from the devices and save it:

```
        - name: "P1T4: Create Folder per Device"
          file:
            path: "{{ op_folder}}/{{ inventory_hostname }}"
            state: directory
          delegate_to: localhost
        - name: "P1T5: Get Operational Data from Devices"
          ios_command:
            commands: "{{ item }}"
          register: op_output
          loop: "{{ op_cmds }}"
        - name: "P1T6: Save output per each node"
          copy:
            content: "{{ item.stdout[0] }}"
            dest: "{{ op_folder}}/{{ inventory_hostname
}}/{{item.item | replace(' ', '_')}}.txt"
          loop: "{{ op_output.results }}"
```

How it works...

In this recipe, we are using the `ios_command` module in order to execute operational commands on the IOS devices, and saving them to text files. In order to achieve this goal, we perform the following steps:

- We create the folders that we will store the output to, and we create a folder called `configs` to store the running config of all the devices. We also create an `op_data` file to store the output of the operational commands that we will get from the devices.

- We then execute the `show running` command on all the IOS devices in our inventory and we register the output in a new variable called `show_run`.
- We use the copy module to save the output from the previous task to a file for each device. The output from the command run is saved in the `stdout` variable. As we executed a single command, the `stdout` variable only has a single item (`stdout[0]`).

Once we execute this task, we can see that the `configs` folder is populated as shown in the following output:

```
$ tree configs/
 configs/
 ├── access01.cfg
 ├── access02.cfg
 ├── core01.cfg
 ├── core02.cfg
 ├── isp01.cfg
 ├── wan01.cfg
 └── wan02.cfg
```

For the next part, we create a folder for each node to store the output from the multiple `show` commands that we will execute on the IOS devices.

We use the `ios_command` module to execute the `show` commands on the devices, and save all the output in a new variable called `op_output`. We use the copy execute command, `show ip route`, and we create a file for the output of this command with the name `show_ip_route.txt`.

After running this task, we can see that this is the current structure of the `op_data` folder:

```
$ tree op_data/
 op_data/
 ├── access01
 │   ├── show_ip_ospf_neighbor.txt
 │   └── show_ip_route.txt
 ├── access02
 │   ├── show_ip_ospf_neighbor.txt
 │   └── show_ip_route.txt
 ├── core01
 │   ├── show_ip_ospf_neighbor.txt
 │   └── show_ip_route.txt
 ├── core02
 │   ├── show_ip_ospf_neighbor.txt
 │   └── show_ip_route.txt
 ├── isp01
 │   ├── show_ip_ospf_neighbor.txt
│  └─── show_ip_route.txt
├─── wan01
│  ├─── show_ip_ospf_neighbor.txt
│  └─── show_ip_route.txt
└─── wan02
├─── show_ip_ospf_neighbor.txt
└─── show_ip_route.txt
```

We can check the content of one of the files to confirm that all the data has been stored:

```
$ head op_data/core01/show_ip_ospf_neighbor.txt

Neighbor ID     Pri   State           Dead Time   Address         Interface
10.100.1.3        0   FULL/  -        00:00:37    10.3.1.2
Ethernet1/0
10.100.1.2        0   FULL/  -        00:00:36    10.1.200.2      Vlan200
```

Validating network states with pyATS and Ansible

In this recipe, we will outline how to use Ansible and the Cisco pyATS Python library to execute and parse operational commands on Cisco devices. Using these parsed commands, we can validate various aspects of the network.

Getting ready

This recipe assumes that the network has already been built and configured as outlined in all the previous recipes.

How to do it...

1. Install the Python libraries needed for pyATS:

```
$ sudo pip3 install pyats genie
```

2. Create the `roles` directory and then create the `requirements.yml` file with the following data:

```
 $ cat roles/requirements.yml
- src: https://github.com/CiscoDevNet/Ansible-pyats
  scm: git
  name: Ansible-pyats
```

3. Install the `Ansible-pyats` role as shown in the following code:

```
 $ Ansible-galaxy install -r requirements.yml
```

4. Create a new playbook called `pb_validate_pyats.yml` and populate it with the following task to collect the `ospf neighbor` from the `wan` devices.

```
---
  - name: Network Validation with pyATS
    hosts: wan
    roles:
      - Ansible-pyats
    vars:
      Ansible_connection: local
    tasks:
      - pyats_parse_command:
          command: show ip ospf neighbor
        register: ospf_output
        vars:
          Ansible_connection: network_cli
```

5. Update the playbook with the following tasks to extract the data for OSPF peer information:

```
      - name: "FACT >> Pyats OSPF Info"
        set_fact:
          pyats_ospf_data: "{{ ospf_output.structured.interfaces
}}"

      - name: " FACT >> Set OSPF peers"
        set_fact:
          OSPF_PEERS: "{{ wan_l3_links[inventory_hostname] |
selectattr('ospf','equalto',true) | list }}"
```

6. Update the playbook with the following tasks to validate OSPF peers and the OSPF peer state:

```
        - name: Validate Number of OSPF Peers
          assert:
            that:
              - pyats_ospf_data | length == OSPF_PEERS | length
          loop: "{{ OSPF_PEERS }}"

        - name: Validate All Peers are in Full State
          assert:
            that:
              - pyats_ospf_data[item.name] |
json_query('neighbors.*.state') | first == 'FULL/ -'
          loop: "{{ OSPF_PEERS }}"
```

How it works...

In this recipe, we are exploring how to use the `pyATS` framework to perform network validation. `pyATS` is an open source Python library developed by Cisco as a testing framework for network testing. `Genie` is another Python library that provides parsing capabilities for transforming CLI-based output to Python data structures that we can consume in our automation scripts. Cisco released an Ansible role that uses the pyATS and Genie libraries. Within this role, there are multiple modules that we can use in order to build more robust Ansible validation playbooks to validate the network state. In order to start working with this role, we need to perform the following steps:

1. Install `pyats` and `enie` Python packages using `python-pip`.
2. Install the `Ansible-pyats` role using Ansible-galaxy.

In this recipe, we are using one of the modules within the `Ansible-pyats` role, which is `pyats_parse_command`. This module executes an operational command on the remote managed device and returns both the CLI output for this command and the parsed structured output for this command. The following code snippet outlines the structured data returned by this module for `ip ospf neigbor` on the `wan01` device:

```
"structured": {
  "interfaces": {
    "GigabitEthernet2": {
      "neighbors": {
        "10.100.1.1": {
          "address": "10.3.1.1",
          "dead_time": "00:00:37",
            "priority": 0,
            "state": "FULL/ -"
          }
        }
      }
    }
  }
```

We save the data returned by this module to the `ospf_output` variable and we use the `set_fact` module to capture the structured data returned by this module, before saving it to a new variable – `pyats_ospf_data`. Then, we use the `set_fact` module to filter the links defined in `wan_l3_interfaces` to just the ports that are enabled for OSPF.

Using the structured data returned by `pyats_parse_command`, we can validate this data and compare it with our OSPF peer definition using the `assert` module so as to validate the correct number of OSPF peers and their states.

To extract the OSPF peer state, we use the `json_query` filter to filter the returned data and provide just the OSPF state for each neighbor.

We are setting `Ansible_connection` to `local` on the play level, and setting it to `network_cli` on the `pyats_parse_command` task level, since we only need to connect to the device in this task. All the other tasks can run locally on the Ansible machine.

See also...

For more information regarding the PyATS and Genie libraries and how to use them for network testing, please consult the following URL:

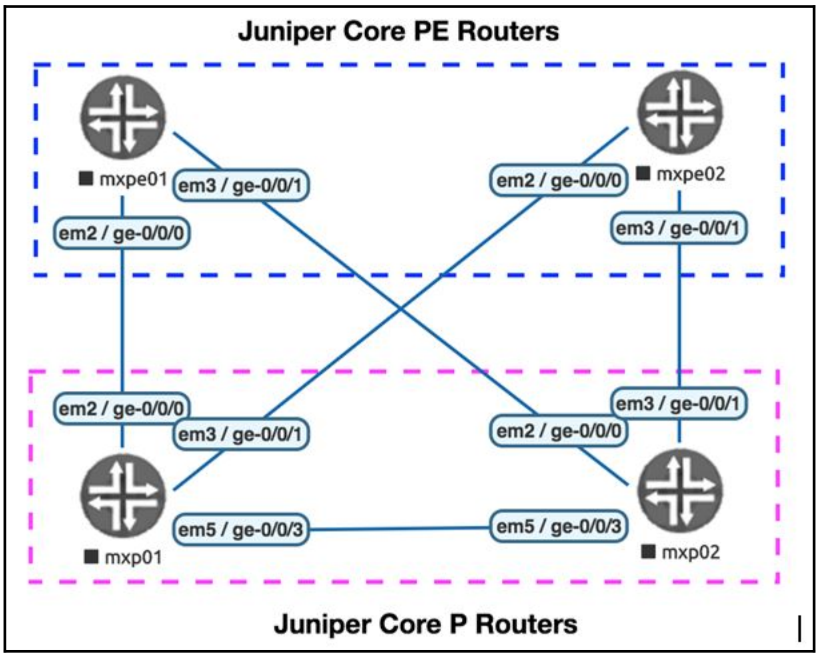
`https://developer.cisco.com/docs/pyats/#!introduction/pyats-genie`

For more information regarding `json_query` and its syntax, please consult the following URLs:

```
https://docs.Ansible.com/Ansible/latest/user_guide/playbooks_filters.html#json-query-filter
http://jmespath.org/tutorial.html
```

3 Automating Juniper Devices in the Service Providers Using Ansible

In this chapter, we will outline how to automate Juniper devices running the Junos OS software in a typical **service provider** (**SP**) environment. We will explore how to interact with Juniper devices using Ansible, and how to provision different services and protocols on Juniper devices using various Ansible modules. We will base our illustration on the following sample network diagram of a basic SP network:



The following table outlines the devices in our sample topology and their respective management **Internet Protocols** (**IPs**):

Device	**Role**	**Vendor**	**Management (MGMT) Port**	**MGMT IP**
`mxp01`	P Router	Juniper vMX 14.1	`fxp0`	`172.20.1.2`
`mxp02`	P Router	Juniper vMX 14.1	`fxp0`	`172.20.1.3`
`mxpe01`	PE Router	Juniper vMX 14.1	`fxp0`	`172.20.1.4`
`mxpe02`	PE Router	Juniper vMX 17.1	`fxp0`	`172.20.1.5`

The main recipes covered in this chapter are as follows:

- Building the network inventory
- Connecting and authenticating to Juniper devices
- Enabling the **Network Configuration Protocol** (**NETCONF**) on Junos OS devices
- Configuring generic system options on Juniper devices
- Configuring interfaces on Juniper devices
- Configuring **Open Shortest Path First** (**OSPF**) on Juniper devices
- Configuring **Multiprotocol Label Switching** (**MPLS**) on Juniper devices
- Configuring the **Border Gate Protocol** (**BGP**) on Juniper devices
- Deploying configuration on Juniper devices
- Configuring the **Layer 3 virtual private network** (**L3VPN**) service on Juniper devices
- Gathering Juniper device facts using Ansible
- Validating network reachability on Juniper devices
- Retrieving operational data from Juniper devices
- Validating the network state using PyEZ operational tables

Technical requirements

The code files for this chapter can be found here: `https://github.com/PacktPublishing/Network-Automation-Cookbook/tree/master/ch3_junos`.

The following are the software releases on which this chapter is based:

- Ansible machine running CentOS 7
- Ansible 2.9
- Juniper **Virtual MX** (**vMX**) running Junos OS 14.1R8 and Junos OS 17.1R1 release

Check out the following video to see the Code in Action:
`https://bit.ly/3ajF4Mp`

Building the network inventory

In this recipe, we will outline how to build and structure the Ansible inventory to describe the sample SP network setup outlined previously. The Ansible inventory is a pivotal part in Ansible, as it defines and groups devices that should be managed by Ansible.

Getting ready

We create a new folder that will host all the files that we will create in this chapter. The new folder is named `ch3_junos`.

How to do it...

1. Inside the new folder, `ch3_junos`, we create a `hosts` file with the following content:

```
$ cat hosts

[pe]
mxpe01 Ansible_host=172.20.1.3
mxpe02 Ansible_host=172.20.1.4

[p]
mxp01 Ansible_host=172.20.1.2
mxp02 Ansible_host=172.20.1.6

[junos]
mxpe[01:02]
mxp[01:02]

[core:children]
pe
p
```

2. Create an `Ansible.cfg` file, as shown in the following code:

```
$ cat Ansible.cfg

 [defaults]
 inventory=hosts
 retry_files_enabled=False
 gathering=explicit
 host_key_checking=False
```

How it works...

We build the Ansible inventory using the `hosts` file and we define multiple groups in order to group the different devices in our network infrastructure, as follows:

- We create the `PE` group, which references all the MPLS **Provider Edge** (**PE**) nodes in our topology.
- We create the `P` group, which references all the MPLS **Provider** (**P**) nodes in our topology.
- We create the `junos` group, which references all the devices running Junos OS as the OS.
- We create the `core parent` group, which references both the `PE` and `P` groups.

Finally, we create the `Ansible.cfg` file and configure it to point to our `hosts` file, to be used as the Ansible inventory file. We set the `gathering` to `explicit` in order to disable the setup module, which runs by default to discover facts for the managed hosts. Disabling the setup module is mandatory since the setup module will fail when run against network devices.

We can validate that our Ansible inventory is structured and written correctly by typing the following command:

```
$ Ansible-inventory --list

     "all": {
        "children": [
            "core",
            "junos",
            "ungrouped"
        ]
    },
    "core": {
        "children": [
                "p",
                "pe"
            ]
        },
        "junos": {
            "hosts": [
                "mxp01",
                "mxp02",
                "mxpe01",
                "mxpe02"
            ]
        },
        "p": {
            "hosts": [
                "mxp01",
                "mxp02"
            ]
        },
        "pe": {
            "hosts": [
                "mxpe01",
                "mxpe02"
            ]
        }
```

Connecting and authenticating to Juniper devices

In this recipe, we will outline how to connect and authenticate to Juniper devices from Ansible via **Secure Shell** (**SSH**), in order to start managing the Juniper devices. We are going to outline how to use SSH keys as the authentication method to establish communication between Ansible and the Juniper devices.

Getting ready

In order to follow along with this recipe, an Ansible inventory file should be constructed as per the previous recipe. IP reachability between the Ansible control machine and all the devices in the network must be configured.

How to do it...

1. On the Ansible machine, create the private and public SSH keys in our `ch3_junos` working directory, as shown in the following code:

```
$ SSH-keygen -t rsa -b 2048 -f Ansible_SSH_key

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:aCqgMYKAWIkv3nVz/q9cYp+2n3doD9jpgw/jeWWcVWI
Ansible@centos7.localdomain
```

2. Capture the public key that was created in the previous step, as follows:

```
$ cat Ansible_SSH_key.pub
 SSH-rsa SSH-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0/wvdC5ycAanRorlfMYDMAv5OTcYAALlE2bd
boajsQPQNEw1Li3N0J50OJBWXX+FFQuF7JKpM32vNQjQN7BgyaBWQGxv+Nj0ViVP+8X
8Wuif0m6bFxBYSaPbIbGogDjPu4qU90Iv48NGOZpcPLqZthtuN7yZKPshX/0YJtXd2q
uUsVhzVpJnncXZMb4DZQeOin7+JVRRrDz6KP6meIylf35mhG3CV5VqpoMjYTzkDiHwI
rFWVMydd4C77RQu27N2HozUtZgJy9KD8qIJYVdP6skzvp49IdInwhjOA+CugFQuhYhH
SoQxRxpws5RZlvrN7/0h0Ahc3OwHaUWD+P7lz Ansible@centos7.localdomain
```

3. On the Juniper devices, add a new user called `admin` and designate that we will use SSH keys for authentication for this user. Copy the public SSH key that was created on the Ansible machine to the device, as shown in the following code:

```
[edit system login]
Ansible@mxpe01# show
user admin {
  uid 2001;
  class super-user;
  authentication {
    SSH-rsa " SSH-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0/wvdC5ycAanRorlfMYDMAv5OTcYAALlE2bd
boajsQPQNEw1Li3N0J50OJBWXX+FFQuF7JKpM32vNQjQN7BgyaBWQGxv+Nj0ViVP+8X
8Wuif0m6bFxBYSaPbIbGogDjPu4qU90Iv48NGOZpcPLqZthtuN7yZKPshX/0YJtXd2q
uUsVhzVpJnncXZMb4DZQeOin7+JVRRrDz6KP6meIylf35mhG3CV5VqpoMjYTzkDiHwI
rFWVMydd4C77RQu27N2HozUtZgJy9KD8qIJYVdP6skzvp49IdInwhjOA+CugFQuhYhH
SoQxRxpws5RZlvrN7/0h0Ahc3OwHaUWD+P7lz Ansible@centos7.localdomain";
## SECRET-DATA
  }
}
```

How it works...

We start by creating the public and private SSH keys on the Ansible control machine, using the `SSH-keygen` command and specifying the following options:

- We specify the encryption algorithm with the `-t` option, and we set it to `rsa`.
- We specify the size of the encryption key using the `-b` option, and we set the size to `2048` bits.
- We specify the location to save the private and public keys using the `-f` option, and we specify the name for the public and private key that will be generated, which will be `Ansible_SSH_key`.

Once we run the command, we will see that the following two files (the private and public SSH keys) are generated, as shown here:

```
$ ls -la | grep Ansible_SSH_key
 -rw------- 1 Ansible Ansible 1679 Dec 31 23:41 Ansible_SSH_key
 -rw-r--r-- 1 Ansible Ansible 409 Dec 31 23:41 Ansible_SSH_key.pub
```

On all the Juniper devices in our inventory, we create the `admin` user and we specify that we will use SSH keys for authentication. We paste the contents of the public key that we have created on the Ansible control machine under the `authentication` stanza for this new user. With this configuration, any host who has the corresponding private key can authenticate and log in to the Juniper devices as the `admin` user.

In order to test and validate that we have successfully logged in to the Junos OS devices from the compute nodes, we can test this using the Ansible command shown in the following code:

```
$ Ansible all -m ping -u admin --private-key Ansible_SSH_key -c network_cli

mxp02 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
mxpe02 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
mxpe01 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
mxp01 | SUCCESS => {
  "changed": false,
    "ping": "pong"
}
```

We specify the username to connect to the devices using the `-u` option and we specify the private SSH key using the –`private-key` option. Finally, we use the `-c` option in order to specify the connection plugin used to connect to the managed devices, and, in this case, we use the `network_cli` connection plugin to open an SSH session with the managed Juniper devices.

There's more...

In order to use the SSH keys that we have generated in our playbooks, we can specify the username and the SSH private key file that we will use to authenticate to our Juniper devices as host or group variables in Ansible. In our case, we will set these variables as group variables for the `junos` group. We create the `group_vars` directory, and we create the `junos.yml` file, and we specify the variables as shown in the following code:

```
$ cat group_vars/junos.yml

Ansible_user: admin
 Ansible_SSH_private_key_file: Ansible_SSH_key
```

We test the connection between Ansible and our devices again using the `Ansible` command; however, this time, without specifying any parameters, as shown in the following code:

```
$ Ansible all –m ping –c network_cli

mxp02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
mxpe02 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
mxpe01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
mxp01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Enabling NETCONF on Junos OS devices

In this recipe, we will outline how to enable the NETCONF protocol on Junos OS devices. This task is critical since we will use the NETCONF API in all the future recipes to manage the Juniper devices. The NETCONF API provides several advantages compared to the traditional SSH access method, and that is why we will use it in all our interactions with the Junos OS devices.

Getting ready

As a prerequisite for this recipe, an Ansible inventory file must be present, as well as the SSH authentication being deployed and working, as per the previous recipe.

How to do it...

1. Create a new playbook called `pb_jnpr_net_build.yml`, as shown in the following code:

```
$ cat pb_jnpr_net_build.yml

- name: Build Juniper SP Network
  hosts: junos
  tasks:
    - name: "Enable NETCONF"
      junos_netconf:
        netconf_port: 830
        state: present
      vars:
        Ansible_connection: network_cli
      tags: netconf
```

2. Update the `group_vars/junos.yml` file with the connection details, as shown in the following code:

```
$ cat group_vars/junos.yml

Ansible_network_os: junos
Ansible_connection: netconf
```

How it works...

In order to start interacting with the Junos OS devices via NETCONF, we need to enable it first, therefore we need to SSH into the device initially and enable NETCONF. That is why we are using the `network_cli` Ansible connection in order to connect with the Junos OS devices via traditional SSH. In order to use the `network_cli` connection plugin, we need to set `Ansible_network_os` as `junos`.

Since we are going to use the NETCONF API in all interactions with Juniper devices in all coming recipes, we enabled the `network_cli` plugin only for the `junos_netconf` task in this playbook via the `vars` attribute. However, for all future tasks that we will add in this playbook, we will use the `netconf` connection specified in the `Ansible_connection` attribute in the `group_vars/junos.yml` file.

We create a new playbook called `pb_jnpr_net_build.yml`, and in the first task, we use the `junos_netconf` module to enable the NETCONF protocol on the remote Junos OS devices. We state the NETCONF port that will be used (by default, it is `830`), and we outline that this configuration must be present on the remote devices via the `state: present` directive.

Once we run the playbook, we will see that all the Junos OS devices are configured with NETCONF, as shown in the following code:

```
admin@mxpe01# show system services
SSH;
netconf {
  SSH {
    port 830;
  }
}
```

Configuring generic system options on Juniper devices

In this recipe, we will outline how to configure some generic system options such as hostname and **Domain Name System** (**DNS**) servers, and provision users on Juniper devices.

Getting ready

To follow along with this recipe, an Ansible inventory is assumed to be already set up, and NETCONF is enabled on all Juniper devices, as per the previous recipe.

How to do it...

1. Update the `group_vars/all.yml` file with the following parameters to define the various system-level parameters such as `dns` and system users, as shown in the following code:

```
$ cat group_vars/all.yml
tmp_dir: ./tmp
config_dir: ./configs
global:
  dns:
  - 172.20.1.1
  - 172.20.1.15
  root_pwd: $1$ciI4raxU$XfCVzABJKdALim0aWVMql0
  users:
  -   role: super-user
      SSH_key: Ansible_SSH_key.pub
      username: admin
  -   hash: $1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1
      passwd: 14161C180506262E757A60
      role: super-user
      username: Ansible
```

2. Create a new playbook called `pb_jnpr_basic_config.yml` with the following tasks, to set up `dns`, `hostname` and system users on Juniper devices:

```
$ cat pb_jnpr_basic_config.yml
---
- name: Configure Juniper Devices
  hosts: junos
  tasks:
    - name: "Conifgure Basic System config"
      junos_system:
        hostname: "{{ inventory_hostname }}"
        name_servers: "{{ global.dns }}"
        state: present
    - name: "Configure Users"
      junos_user:
        name: "{{ item.username }}"
```

```
                role: "{{ item.role }}"
                SSHkey: "{{ lookup ('file', item.SSH_key) }}"
                state: present
              with_items: "{{ global.users |
        selectattr('SSH_key','defined') | list }}"
```

How it works...

Ansible provides declarative modules to configure various system-level parameters on Juniper devices. The `junos_system` Ansible module enables us to set up the hostname and the DNS servers on the Juniper devices. The `junos_user` module provides us with the ability to set up the basic parameters for the system users on a Juniper device. In this example, we set up all the users who have SSH keys as their authentication method, and we loop over the `users` data structure and select only the users with the `SSH_key` option defined.

Once we run this playbook, we can see that the configuration on the devices is updated, as shown in the following code block:

```
$ Ansible@mxpe01# show system
host-name mxpe01;
}
name-server {
    172.20.1.1;
    172.20.1.15;
}
login {
    user admin {
        uid 2001;
        class super-user;
        authentication {
            SSH-rsa "SSH-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0/wvdC5ycAanRorlfMYDMAv5OTcYAALlE2bdboajsQPQ
NEw1Li3N0J50OJBWXX+FFQuF7JKpM32vNQjQN7BgyaBWQGxv+Nj0ViVP+8X8Wuif0m6bFxBYSaP
bIbGogDjPu4qU90Iv48NGOZpcPLqZthtuN7yZKPshX/0YJtXd2quUsVhzVpJnncXZMb4DZQeOin
7+JVRRrDz6KP6meIylf35mhG3CV5VqpoMjYTzkDiHwIrFWVMydd4C77RQu27N2HozUtZgJy9KD8
qIJYVdP6skzvp49IdInwhjOA+CugFQuhYhHSoQxRxpws5RZlvrN7/0h0Ahc3OwHaUWD+P7lz
Ansible@centos7.localdomain"; ## SECRET-DATA
        }
    }
```

There's more...

The declarative Ansible modules that we have outlined in this section provide a simple way to configure the basic system-level parameters for Juniper devices. However, they might not cover all the parameters that we need to set up on a Juniper device. In order to have more control and flexibility to configure the system-level parameters on a Juniper device, we can use Jinja2 templates along with the Ansible `template` module to generate the specific system-level configuration needed for our deployment. In this section, we will outline this method in order to achieve this goal, and this is the method that we will use in subsequent recipes to generate the configuration for the other devices.

We are going to reuse this method to generate the configuration for our Juniper devices for different sections, such as system, interfaces, OSPF, and MPLS. We are going to create an Ansible role in order to include all the Jinja2 templates and tasks required to generate the final configuration that we will push to our devices. The following procedures outline the steps needed to create the role and to use this role to generate the configuration:

1. Create a new `roles` directory and add a new role called `build_router_config` with the following directory structure:

```
$ tree roles/
 roles/
  └──── build_router_config
  ├──── tasks
  └──── templates
```

2. Under the `tasks` folder, create a `build_config_dir.yml` YAML file to create the required folders to store the configuration that will be generated, as follows:

```
$ cat roles/build_router_config/tasks/build_config_dir.yml

---
- name: Create Config Directory
  file: path={{config_dir}} state=directory
  run_once: yes

- name: Create Temp Directory per Node
  file: path={{tmp_dir}}/{{inventory_hostname}} state=directory

- name: SET FACT >> Build Directory
  set_fact:
  build_dir: "{{tmp_dir}}/{{inventory_hostname}}"
```

3. Under the `templates` folder, create a new folder called `junos`, and within this folder, create a new Jinja2 template called `mgmt.j2`, with the following content:

```
$ cat roles/build_router_config/templates/junos/mgmt.j2

system {
    host-name {{inventory_hostname}};
    no-redirects;
{%  if global.dns is defined %}
    name-server {
{%      for dns_server in global.dns %}
        {{dns_server}};
{%      endfor %}
    }
{%  endif %}
    root-authentication {
        encrypted-password "{{ global.root_pwd}}"; ## SECRET-DATA
    }
    login {
{%      for user in global.users if user.hash is defined %}
        user {{ user.username }} {
            class super-user;
            authentication {
                encrypted-password "{{user.hash}}"; ## SECRET-DATA
            }
        }
{%      endfor %}
{%      for user in global.users if user.SSH_key is defined %}
        user {{ user.username }} {
            class {{ user.role }};
            authentication {
                SSH-rsa "{{lookup('file',user.SSH_key)}}"; ##
SECRET-DATA
            }
        }
{%      endfor %}
    }
}
```

4. Under the `tasks` folder, create a new YAML file called `build_device_config.yml`, with the following task to create the system configuration:

```
$ cat roles/build_router_config/tasks/build_device_config.yml

---
- name: "System Configuration"
```

```
    template:
      src: "{{Ansible_network_os}}/mgmt.j2"
      dest: "{{build_dir}}/00_mgmt.cfg"
    tags: mgmt
```

5. Create a `main.yml` file under the `tasks` folder, with the following tasks:

```
$ cat roles/build_router_config/tasks/main.yml

---
- name: Build Required Directories
  import_tasks: build_config_dir.yml

- name: Build Device Configuration
  import_tasks: build_device_config.yml

- name: "Remove Old Assembled Config"
  file:
    path: "{{config_dir}}/{{ inventory_hostname }}.cfg"
    state: absent

- name: Build Final Device Configuration
  assemble:
    src: "{{ build_dir }}"
    dest: "{{config_dir}}/{{ inventory_hostname }}.cfg"

- name: Remove Build Directory
  file: path={{ tmp_dir }} state=absent
  run_once: yes
```

6. Update the `pb_jnpr_net_build.yml` playbook with the following task to generate the configuration for all Juniper devices in our inventory:

```
$ cat pb_jnpr_net_build.yml

- name: Build Device Configuration
  import_role:
    name: build_router_config
  vars:
    Ansible_connection: local
  tags: build
```

In this method, we create a role called `build_router_config` and we create a new Jinja2 template called `mgmt.j2`, which includes the template for Junos OS system-level configuration. We use the Ansible `template` module in order to render the Jinja2 template with the Ansible variables defined under the `group_vars/all.yml` file. In order to save the configuration for each device, we create the `configs` folder directory, which stores the final configuration for each device.

Since we will use this approach in order to generate the configuration for each section (MGMT, OSPF, MPLS, and so on), we will segment each section into a separate Jinja2 template, and we will generate each section in a separate file. We use the `assemble` module in order to group all these different sections into a single configuration file, which we will store in the `configs` directory. This is the `final` and `assembled` configuration file for each device. We store the temporary configuration snippets for each section in a temporary folder for each device, and we delete this temporary folder at the end of the playbook run. This is because we assembled the final configuration for the device, and we don't require these configuration snippets anymore.

In this playbook, we set the `Ansible_connection` to `local` as we don't need to connect to the devices in order to run any of the tasks within our role. We are only generating the configuration on the Ansible control machine, therefore all the tasks need to run locally on the Ansible control machine. Therefore, there is no need to connect to the remotely managed nodes.

Once we run the playbook, we can see that the following configuration files are created inside the `configs` directory:

```
$ tree configs/
 configs/
  ├── mxp01.cfg
  ├── mxp02.cfg
  ├── mxpe01.cfg
  └── mxpe02.cfg
```

We can see the configuration generated for the `mxpe01` device as an example, as follows:

```
$ cat configs/mxpe01.cfg
system {
    host-name mxpe01;
    no-redirects;
    name-server {
        172.20.1.1;
        172.20.1.15;
        }
        root-authentication {
            encrypted-password "$1$ciI4raxU$XfCVzABJKdALim0aWVMql0"; ## SECRET-
    DATA
        }
        login {
            user Ansible {
                class super-user;
                authentication {
                    encrypted-password "$1$mR940Z9C$ipX9sLKTRDeljQXvWFfJm1"; ##
    SECRET-DATA
                }
            }
            user admin {
                class super-user;
                authentication {
                    SSH-rsa "SSH-rsa
    AAAAB3NzaC1yc2EAAAADAQABAAABAQC0/wvdC5ycAanRorlfMYDMAv5OTcYAALlE2bdboajsQPQ
    NEw1Li3N0J50OJBWXX+FFQuF7JKpM32vNQjQN7BgyaBWQGxv+Nj0ViVP+8X8Wuif0m6bFxBYSaP
    bIbGogDjPu4qU90Iv48NGOZpcPLqZthtuN7yZKPshX/0YJtXd2quUsVhzVpJnncXZMb4DZQeOin
    7+JVRRrDz6KP6meIylf35mhG3CV5VqpoMjYTzkDiHwIrFWVMydd4C77RQu27N2HozUtZgJy9KD8
    qIJYVdP6skzvp49IdInwhjOA+CugFQuhYhHSoQxRxpws5RZlvrN7/0h0Ahc3OwHaUWD+P7lz
    Ansible@centos7.localdomain"; ## SECRET-DATA
                }
            }
        }
    }
```

In subsequent recipes, we will outline how to push the generated configuration into the Juniper devices using another Ansible module.

See also...

For more information regarding the Ansible `template` module and the different parameters supported by this module, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/template_module.html`.

For more information regarding the Ansible `assemble` module and the different parameters supported by this module, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/assemble_module.html`.

Configuring interfaces on Juniper devices

In this recipe, we will outline how to manage interfaces on a Juniper device. This allows us to set different parameters for our interfaces, such as the **maximum transition unit** (**MTU**) and the IP addresses on Juniper devices.

Getting ready

To follow along with this recipe, an Ansible inventory is assumed to be already set up, and NETCONF is enabled on all Juniper devices, as per the previous recipe.

How to do it...

1. Update the `group_vars/all.yml` YAML file to include the following data for all the **point-to-point** (**P2P**) and loopback interfaces in our sample network topology:

```
p2p_ip:
  mxp01:
    - {port: ge-0/0/0, ip: 10.1.1.2 , peer: mxpe01, pport:
ge-0/0/0, peer_ip: 10.1.1.3}
    - {port: ge-0/0/1, ip: 10.1.1.4 , peer: mxpe02, pport:
ge-0/0/0, peer_ip: 10.1.1.5}
    - {port: ge-0/0/3, ip: 10.1.1.0 , peer: mxp02, pport: ge-0/0/3,
peer_ip: 10.1.1.1}
  mxp02:
 <-- Output Trimmed for brevity ------>
  mxpe01:
 <-- Output Trimmed for brevity ------>
  mxpe02:
 <-- Output Trimmed for brevity ------>
  xrpe03:
 <-- Output Trimmed for brevity ------>
lo_ip:
  mxp01: 10.100.1.254/32
  mxp02: 10.100.1.253/32
  mxpe01: 10.100.1.1/32
 mxpe02: 10.100.1.2/32
 xrpe03: 10.100.1.3/32
```

2. Update the `pb_jnpr_basic_config.yml` playbook with the following tasks to set up the interfaces on our Juniper devices:

```
- name: "Configure the Physical Interfaces"
  junos_interface:
    name: "{{ item.port }}"
    enabled: true
    description: "peer:{{item.peer}} remote_port:{{item.pport }}"
    mtu: "{{ global.mtu | default(1500) }}"
  with_items: "{{p2p_ip[inventory_hostname]}}"
  tags: intf

- name: "Configure IP Addresses"
  junos_l3_interface:
    name: "{{ item.port }}"
    ipv4: "{{ item.ip }}/{{ global.p2p_prefix }}"
    state: present
  with_items: "{{ p2p_ip[inventory_hostname] }}"
  tags: intf
```

How it works...

We define all the data for all the interfaces in our sample network topology under two main data structures in the `group_vars/all.yml` file. We use the `p2p_ip` dictionary to model all the P2P IP addresses in our sample network, and we use the `lo_ip` dictionary to specify the loopback IP addresses for our nodes.

We use the `junos_interface` Ansible module to enable the interfaces and set the basic parameters for the interfaces, such as MTU and description. We loop over the `p2p_ip` data structure for each device, and we set the correct parameters for each interface on all the devices in our network inventory. We use the `junos_l3_interface` Ansible module to set the correct IPv4 address on all the interfaces in our sample network topology across all the devices.

Once we run the playbook, we can see that the interfaces are configured as required, as shown on the `mxpe01` device:

```
Ansible@mxpe01# show interfaces
ge-0/0/0 {
  description "peer:mxp01 remote_port:ge-0/0/0";
  mtu 1500;
  unit 0 {
    family inet {
      address 10.1.1.3/31;
```

```
        }
      }
    }
    ge-0/0/1 {
      description "peer:mxp02 remote_port:ge-0/0/0";
      mtu 1500;
      unit 0 {
        family inet {
          address 10.1.1.9/31;
        }
      }
    }
```

There's more...

In case we need to have more control over the interface configuration, and to set parameters that are not covered by the declarative Ansible modules that we have outlined in this section, we can use Jinja2 templates to achieve this goal. Using the exact same approach that we outlined in the previous recipe for system configuration, we can generate the interface configuration needed for our Juniper devices.

Using the same Ansible role that we have created in the previous recipe, we can extend it to generate the interface configuration for our Juniper devices. We use the following steps in order to accomplish this task:

1. Create a new Jinja2 template file called `intf.j2` in the `templates` folder, with the following data:

```
$ cat roles/build_router_config/templates/junos/intf.j2

interfaces {
{% for intf in p2p_ip[inventory_hostname] | sort(attribute='port')
%}
  {{ intf.port.split('.')[0] }} {
    description "peer:{{intf.peer}} -- peer_port: {{intf.pport}}"
    unit 0 {
      family inet {
        address {{intf.ip}}/{{global.p2p_prefix}};
      }
      family mpls;
    }
  }
  {% endif %}
  {% endfor %}
  lo0 {
```

```
              unit 0 {
                family inet {
                  address {{lo_ip[inventory_hostname]}};
                }
              }
            }
```

2. Update the `build_device_config.yml` file under the `tasks` directory with the new task to generate the interface configuration, as follows:

```
$ cat roles/build_router_config/tasks/build_device_config.yml

<-- Output Trimmed for brevity ------>

- name: "Interface Configuration"
  template:
    src: "{{Ansible_network_os}}/intf.j2"
    dest: "{{build_dir}}/01_intf.cfg"
  tags: intf
```

This is the generated interface configuration for the `mxp02` device after running the playbook:

```
interfaces {
    ge-0/0/0 {
        description "peer:mxpe01 -- peer_port: ge-0/0/1"
        unit 0 {
            family inet {
                address 10.1.1.8/31;
            }
            family mpls;
        }
    }
    ge-0/0/1 {
        description "peer:mxpe02 -- peer_port: ge-0/0/1"
        unit 0 {
            family inet {
                address 10.1.1.10/31;
            }
            family mpls;
        }
    }
<--   Output Trimmed for brevity ------>
    lo0 {
        unit 0 {
            family inet {
                address 10.100.1.253/32;
            }
            }
        }
```

Configuring OSPF on Juniper devices

In this recipe, we will outline how to configure OSPF on Juniper devices as the **interior gateway protocol** (**IGP**) in our sample network topology, along with different OSPF parameters such as OSPF link type and OSPF interface cost.

How to do it...

1. Create a new Jinja2 file, `ospf.j2`, in the `templates/junos` directory, with the following data:

```
$ cat roles/build_router_config/templates/junos/ospf.j2

 protocols {
    ospf {
        area {{global.ospf_area}} {
{%          for intf in
p2p_ip[inventory_hostname]|sort(attribute='port') %}
            interface {{ intf.port }} {
                interface-type p2p;
                metric {{intf.cost | default(100)}};
            }
{%          endfor %}
            interface lo0.0 {
                passive;
            }
        }
    }
}
```

2. In the `junos_build_config.yml` file inside the `tasks` folder, add the following task:

```
$ cat roles/build_router_config/tasks/build_device_config.yml

<-- Output Trimmed for brevity ------>

- name: "OSPF Configuration"
  template:
```

```
                src: "{{Ansible_network_os}}/ospf.j2"
                dest: "{{config_dir}}/{{ inventory_hostname }}/02_ospf.cfg"
```

How it works...

We use the same interface data that was declared in the `p2p_ip` data structure in the `all.yml` file, in order to provision the OSPF configuration on the network devices in our sample network. We use a new Jinja2 template defined in the `ospf.j2` file under the `templates/junos` directory to capture the OSPF configuration parameters (OSPF cost, OSPF interface type, and so on) that need to be implemented on the Juniper devices.

Under the `tasks/Juniper_build_config.yml` file, we add a new task that uses the `ospf.j2` Jinja2 template to render the Jinja2 template, and output the OSPF configuration section for each device outlined in our Ansible inventory.

The following snippet outlines the OSPF configuration generated for the `mxpe01` device after running the playbook with the new task:

```
$ cat configs/mxpe01.cfg

 <--   Output Trimmed for brevity ------>

protocols {
    ospf {
        area 0 {
            interface ge-0/0/0 {
                interface-type p2p;
                metric 100;
            }
            interface ge-0/0/1 {
                interface-type p2p;
                metric 100;
            }
            interface lo0.0 {
                passive;
            }
        }
    }
}
```

Configuring MPLS on Juniper devices

In this recipe, we will outline how to configure MPLS and some of the related protocols such as the **Label Distribution Protocol** (**LDP**) and the **Resource Reservation Protocol** (**RSVP**) on Juniper devices. We will outline how to generate the required MPLS configuration using Ansible and Jinja2 templates.

How to do it...

1. Create a new Jinja2 file, `mpls.j2`, under the `templates/junos` directory with the following data:

```
$ cat roles/build_router_config/templates/junos/mpls.j2

 protocols {
    ldp {
{%      for intf in
p2p_ip[inventory_hostname]|sort(attribute='port') %}
        interface {{intf.port}}.{{intf.vlan|default('0')}};
{%      endfor %}
        interface lo0.0;
    }
    rsvp {
{%      for intf in
p2p_ip[inventory_hostname]|sort(attribute='port') %}
        interface {{intf.port}}.{{intf.vlan|default('0')}};
{%      endfor %}
    }
    mpls {
{%      for intf in
p2p_ip[inventory_hostname]|sort(attribute='port') %}
        interface {{intf.port}}.{{intf.vlan|default('0')}};
{%      endfor %}
    }
}
```

2. In the `build_device_config.yml` file inside the `tasks` folder, add the following task:

```
$ cat roles/build_router_config/tasks/build_device_config.yml

<-- Output Trimmed for brevity ------>

- name: "MPLS Configuration"
  template:
    src: "{{Ansible_network_os}}/mpls.j2"
    dest: "{{config_dir}}/{{ inventory_hostname }}/03_mpls.cfg"
```

How it works...

We use the same methodology as used to configure the interfaces and OSPF, by using a Jinja2 template to generate the needed MPLS configuration for the Juniper devices in our inventory, and the following is a sample of the MPLS configuration for the `mxpe02` router:

```
protocols {
    ldp {
        interface ge-0/0/0.0;
        interface ge-0/0/1.0;
        interface lo0.0;
    }
    rsvp {
        interface ge-0/0/0.0;
        interface ge-0/0/1.0;
    }
    mpls {
        interface ge-0/0/0.0;
        interface ge-0/0/1.0;
    }
}
```

Configuring BGP on Juniper devices

In this recipe, we will outline how to configure BGP on Juniper devices. We will outline how to set up BGP and BGP **Route Reflectors** (**RR**) as part of our sample topology, along with all the required BGP address families to support **virtual private network** (**VPN**) services.

How to do it...

1. Update the `group_vars/all.yml` file with the following BGP information:

```
bgp_topo:
  rr: mxp01
  af:
  - inet
  - inet-vpn
```

2. For each node within our Ansible inventory, we create a file called `bgp.yml` under the `host_vars` directory. This file holds the BGP information and BGP peers for each node. This is the example for the `mxpe01` device:

```
$ cat host_vars/mxpe01/bgp.yml

bgp_asn: 65400

bgp_peers:
  - local_as: 65400
    peer: 10.100.1.254
    remote_as: 65400
```

3. Create a new Jinja2 file, `bgp.j2`, under the `templates/junos` directory, with the following data:

```
$ cat roles/build_router_config/templates/junos/bgp.j2

 protocols {
{%  if bgp_peers is defined %}
    bgp {
        group Core {
            type internal;
            local-address {{ lo_ip[inventory_hostname] |
ipaddr('address')}};
{%          if bgp_topo.rr == inventory_hostname %}
            cluster {{ lo_ip[inventory_hostname].split('/')[0] }};
{%          endif %}
{%          for af in bgp_topo.af %}
{%          if af == 'inet' %}
            family inet {
                unicast;
            }
{%          endif %}
{%          if af == 'inet-vpn' %}
            family inet-vpn {
                unicast;
```

```
              }
{%          endif %}
<--   Output Trimmed for brevity ------>
{%          endfor %}
{%          for p in bgp_peers %}
            neighbor {{ p.peer}};
{%          endfor %}
        }
    }
{%  endif %}
}
```

4. In the `build_device_config.yml` file inside the `tasks` folder, add the following highlighted task:

```
$ cat roles/build_router_config/tasks/build_device_config.yml

<-- Output Trimmed for brevity ------>

- name: "BGP Configuration"
  template:
    src: "{{Ansible_network_os}}/bgp.j2"
    dest: "{{config_dir}}/{{ inventory_hostname }}/04_bgp.cfg"
```

How it works...

Using a similar approach to all the previous recipes, we use a Jinja2 template to generate the BGP configuration for the Juniper devices. However, in this section, we declare the BGP parameters in two different places, which are the `group_vars` and `host_vars` directories. In the `group_vars/all.yml` file, we declare the overall parameters for our BGP topology, such as the RR that we will use, and which address families we will configure. For each node in our inventory, we create a directory in the `host_vars` directory, and inside this directory, we create a `bgp.yml` file. This new YAML file holds the BGP peers for each node in our inventory. We use the data defined in these two locations to render the BGP configuration for each device.

This is a sample of the BGP configuration for the `mxp01` router, which is the RR in our topology:

```
protocols {
    bgp {
        group Core {
            type internal;
            local-address 10.100.1.254;
```

```
                cluster 10.100.1.254;
                family inet {
                    unicast;
                }
                family inet-vpn {
                    unicast;
                }
                neighbor 10.100.1.1;
                neighbor 10.100.1.2;
                neighbor 10.100.1.3;
            }
        }
    }
```

Deploying configuration on Juniper devices

In this recipe, we will outline how to push the configuration that we have generated via Jinja2 templates in all the previous sections on Juniper devices using Ansible. This provides us with the capability to push any custom configuration that we create to our Juniper devices.

Getting ready

This recipe requires NETCONF to be enabled on the Juniper devices.

How to do it...

1. In the `pb_junos_push_con` file, add the following task:

```
$ cat pb_jnpr_net_build.yml

<-- Output Trimmed for brevity ------>

- name: "Deploy Configuration"
  junos_config:
    src: "{{config_dir}}/{{ inventory_hostname }}.cfg"
```

How it works...

In the previous recipe, we generated different sections of the configuration for Juniper devices such as interfaces, OSPF, MPLS, and BGP. We have used the `assemble` module in order to group all these sections per each node in a single configuration file. This file is stored in the `configs` folder for each device.

We use the `junos_config` module in order to push this configuration file that we have generated to each device in our network inventory. We can use the `update` parameter in order to control how the configuration that we 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 option 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.

We can use the `check` mode to run our playbook in dry-run mode. In this case, we will push the configuration to the devices without committing to the configuration. This enables us to check the changes that will be pushed to the devices. This can be accomplished as follows:

```
$ Ansible-playbook pb_jnpr_net_build.yml -l mxpe01 --check -diff
```

We use the –`check` option to run the playbook in check mode (dry-run), and the –`diff` option in order to output the changes that will be pushed to our devices.

There's more...

The `junos_config` module also supports the rollback feature supported by Junos OS, therefore we can add another task to roll back the configuration and control how it is run, as follows:

```
$ cat pb_jnpr_net_build.yml

<-- Output Trimmed for brevity ------>

- name: "Rollback config"
  junos_config:
    rollback: "{{ rollback | default('1') | int }}"
  tags: rollback, never
```

In the preceding playbook, we roll back to the last version of the configuration. However, by changing the number in the `rollback` attribute, we can control the version of the configuration to which we want to roll back. Also, we are using the tags in order to only execute this task when we specify the `rollback` tag during the playbook run, as shown in the following code snippet:

```
$ Ansible-playbook pb_jnpr_net_build.yml --tags rollback -l mxpe01
```

We can specify another rollback point, as follows:

```
$ Ansible-playbook pb_jnpr_net_build.yml --tags rollback -l mxpe01 -e
rollback=2
```

See also...

For more information regarding the `junos_config` module and the different parameters supported by this module, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/junos_config_module.html`.

Configuring the L3VPN service on Juniper devices

In this recipe, we will outline how to model and configure L3VPNs on Juniper devices using various Ansible modules. This enables us to model our services using **Infrastructure as Code** (**IaC**) practices, and utilize Ansible to deploy and push the required configuration to have the L3VPN deployed on Juniper devices.

Getting ready

NETCONF must be enabled on the Juniper devices so as to use the Ansible modules in this recipe.

How to do it...

1. Create a new file called `l3vpn.yml` with the following content:

```
---
l3vpns:
  vpna:
    state: present
    rt: "target:{{bgp_asn}}:10"
    rd: "1:10"
    sites:
      - node: mxpe01
        port: ge-0/0/3.10
        ip: 172.10.1.1/24
     - node: mxpe02
       port: ge-0/0/3.10
       ip: 172.10.2.1/24
  vpnb:
    state: present
    rt: "target:{{bgp_asn}}:20"
    rd: "1:20"
    sites:
      - node: mxpe01
        port: ge-0/0/3.20
        ip: 172.20.1.1/24
      - node: mxpe02
        port: ge-0/0/3.20
        ip: 172.20.2.1/24
```

2. Create a new playbook called `pb_junos_l3vpn.yml` with the following tasks to configure the PE-**Customer Edge** (**CE**) links:

```
---
- name: "Deploy L3VPNs on Juniper Devices"
  hosts: pe
  vars_files:
    - "l3vpn.yml"
  tasks:
    - name: "Set VPN Interfaces"
      set_fact:
        l3vpn_intfs: "{{ l3vpn_intfs|default([]) +
          l3vpns[item.key].sites |
 selectattr('node','equalto',inventory_hostname) | list}}"
      with_dict: "{{l3vpns}}"
      delegate_to: localhost

    - name: "Configure Interfaces for L3VPN Sites"
```

```
          junos_config:
            lines:
              - set interfaces {{ item.port.split('.')[0]}} vlan-
tagging
              - set interfaces {{ item.port}} vlan-id {{
item.port.split('.')[1] }}
          loop: "{{ l3vpn_intfs }}"
```

3. Add the following tasks in `pb_junos_l3vpn.yml` to set up the P2P IP address on the PE-CE links:

```
- name: "Configure IP address for L3VPN Interfaces"
  junos_l3_interface:
    name: "{{ item.port.split('.')[0]}}"
    ipv4: "{{ item.ip }}"
    unit: "{{ item.port.split('.')[1] }}"
  loop: "{{l3vpn_intfs}}"
  tags: intf_ip
```

4. Add the following task in `pb_junos_l3vpn.yml` to configure the **virtual routings and forwardings** (**VRFs**) on the PE nodes:

```
- name: "Configure L3VPNs"
  junos_vrf:
    name: "{{ item.key }}"
    rd: "{{item.value.rd}}"
    target: "{{ item.value.rt }}"
    interfaces: "{{ l3vpns[item.key].sites |
                    map(attribute='port') | list }}"
    state: "{{ item.value.state }}"
  with_dict: "{{l3vpns}}"
  when: inventory_hostname in (l3vpns[item.key].sites |
map(attribute='node') | list)
  tags: l3vpn
```

How it works...

We create a new YAML file called `l3vpn.yml` that describes and models the L3VPN topology and data that we want to implement on all the Juniper devices on our topology. We include this file in the new playbook that we create in order to provision the L3VPNs on our network devices.

In the `pb_junos_l3vpn.yml` playbook, we use the data from the `l3vpn.yml` file to capture the data required to provision the L3VPN.

In the first task within our playbook, we create a new variable called `l3vpn_intfs` that captures all the L3VPN interfaces on each PE device, across all the VPNs that we have defined in our `l3vpn.yml` file. We loop over all the L3VPNs in this file, and we create a new list data structure for all the interfaces that belong to a specific node. The following snippet outlines the new data structure `l3vpn_intfs` for `mxpe01`:

```
ok: [mxpe01 -> localhost] => {
    "l3vpn_intfs": [
        {
            "ip": "172.10.1.1/24",
            "node": "mxpe01",
            "port": "ge-0/0/3.10"
        },
        {
            "ip": "172.20.1.1/24",
            "node": "mxpe01",
            "port": "ge-0/0/3.20"
        }
    ]
}
```

Next, in our playbook, we divide the provisioning of our L3VPN service to multiple tasks:

- We use the `junos_config` module to configure all the interfaces that are part of the L3VPNs to be ready to configure **virtual LANs** (**VLANs**) on these interfaces.
- We use the `junos_l3_interface` module to apply the IPv4 addresses on all these interfaces that are part of our L3VPN model.
- We use the `junos_vrf` module to configure the correct routing instances on the nodes, as per our L3VPN data model.

The following outlines the L3VPN configuration that is applied on `mxpe01` after running this playbook:

```
Ansible@mxpe01> show configuration routing-instances
vpna {
   instance-type vrf;
   interface ge-0/0/3.10;
   route-distinguisher 1:10;
   vrf-target target:65400:10;
   vrf-table-label;
}
vpnb {
   instance-type vrf;
   interface ge-0/0/3.20;
   route-distinguisher 1:20;
   vrf-target target:65400:20;
```

```
    vrf-table-label;
}
```

See also...

For more information regarding the `junos_vrf` module and the different parameters supported by this module to provision L3VPNs on Juniper devices, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/junos_vrf_module.html#junos-vrf-module`.

Gathering Juniper device facts using Ansible

In this recipe, we will retrieve the basis system facts collected by Ansible for a Juniper device. These basic system facts provide us with a basic health check regarding our Juniper devices, which we can use to validate its operational state.

Getting ready

NETCONF must be enabled on the Juniper devices so as to use the Ansible modules in this recipe.

How to do it...

1. Create a new playbook, `pb_jnpr_facts.yml`, with the following task to collect the facts:

```
$ cat pb_jnpr_facts.yml

---
- name: Collect and Validate Juniper Facts
  hosts: junos
  tasks:
    - name: Collect Juniper Facts
      junos_facts:
```

2. Update the `pb_jnpr_facts.yml` playbook with the following tasks to create a facts report for each node in our inventory:

```
- name: Create Facts Folder
  file: path=device_facts state=directory
  run_once: yes

- name: Create Basic Device Facts Report
  blockinfile:
    path: "device_facts/{{ inventory_hostname }}.txt"
    block: |
      device_name: {{ Ansible_net_hostname }}
      model: {{ Ansible_net_system }} {{ Ansible_net_model }}
      os_version: {{ Ansible_net_version }}
      serial_number: {{ Ansible_net_serialnum }}
    create: yes
```

3. Update the playbook with the following task to validate the operational state for the core interfaces:

```
- name: Validate all Core Interface are Operational
  assert:
    that:
      - Ansible_net_interfaces[item.port]['oper-status'] ==
'up'
    fail_msg: "Interface {{item.port}} is not Operational "
  loop: "{{ p2p_ip[inventory_hostname] }}"
```

How it works...

Ansible provides a fact-gathering module to collect the basic system properties for Juniper devices and returns these facts in a consistent and structured data structure. We can use the facts collected by this module in order to validate the basic properties and operational state of our devices, and we can use this data to build simple reports that capture the state of our devices.

In this recipe, we use the `junos_facts` module to collect the device facts for all our Juniper devices. This module returns the basic facts collected by Ansible for each device in multiple variables, as follows:

```
"Ansible_net_serialnum": "VM5D112EFB39",
"Ansible_net_system": "junos",
"Ansible_net_version": "17.1R1.8",
"Ansible_network_os": "junos",
```

We use this data in order to build a fact report for each device using the `blockinfile` module, and we use this data to validate the operational state of the core interfaces of each device using the `assert` module.

Once we run our playbook, we can see that a facts report for each device is generated, as follows:

```
$ tree device_facts/

device_facts/
  ├─── mxp01.txt
  ├─── mxp02.txt
  ├─── mxpe01.txt
  └─── mxpe02.txt


 $ cat device_facts/mxp01.txt

device_name: mxp01
 model: junos vmx
 os_version: 14.1R4.8
 serial_number: VM5701F131C6
```

In the final task, we use the `assert` module in order to validate that all the core interfaces on all the Juniper devices are operational. Ansible stores all the interfaces' operational status for the device under `Ansible_net_interfaces`. We use the data in this data structure to validate that the operational state is up. In the case that all the core interfaces are operational, the task will succeed—otherwise, the task will fail.

See also...

For more information regarding the `junos_facts` module and the different parameters supported by this module, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/junos_facts_module.html.`

Validating network reachability on Juniper devices

In this recipe, we will outline how to validate network reachability via `ping`, using Ansible on Juniper devices. This will enable us to validate network reachability and traffic forwarding across our sample network topology.

Getting ready

This recipe assumes that the network is already built and configured, as outlined in all the previous recipes.

How to do it...

1. Create a new playbook called `pb_junos_ping.yml` with the following task, to ping all core loopbacks within our sample network:

```
---
- name: "Validate Core Reachability"
  hosts: junos
  tasks:
    - name: "Ping Across All Loopback Interfaces"
      junos_ping:
        dest: "{{ item.value.split('/')[0] }}"
        interface: lo0.0
        size: 512
      with_dict: "{{lo_ip}}"
      vars:
        Ansible_connection: network_cli
      register: ping_rst
      ignore_errors: yes
```

2. Update the `pb_junos_ping.yml` playbook with the following task to create a custom report to capture the ping results:

```
    - name: Create Ping Report
      blockinfile:
        block: |
          Node | Destination | Packet Loss | Delay |
          -----| ------------| ------------| ------|
          {% for node in play_hosts %}
          {% for result in hostvars[node].ping_rst.results %}
          {% if result.rtt is defined %}
          {{ node }} | {{ result.item.value }} | {{
result.packet_loss }} | {{ result.rtt.avg }}
          {% else %}
          {{ node }} | {{ result.item.value }} | {{
result.packet_loss }} | 'N/A'
          {% endif %}
          {% endfor %}
          {% endfor %}
      path: ./ping_report.md
```

```
                create: yes
           run_once: yes
```

How it works...

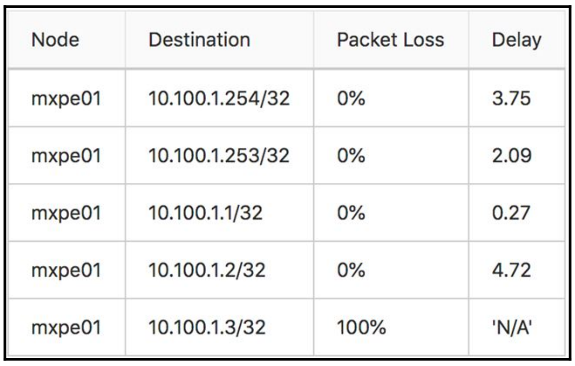
We use the `junos_ping` module in order to ping from all the nodes in our network inventory to all the loopback interfaces defined in the `lo_ip` data structure, which is defined in the `group_vars/all.yml` file. This module connects to each device and executes ping to all the destinations, and validates that ping packets are reaching their intended destination. This module requires the use of the `network_cli` connection plugin, therefore we supply this parameter as a task variable in order to override the group-level NETCONF connection plugin defined at the group level.

We register the output of the module in order to use this data to generate the ping report. Finally, we set `ignore_errors` to `yes` in order to ignore any failed ping task that we might encounter, and ensure that we will run the subsequent tasks to create the report.

We use the `blockinfile` module in order to create a custom report in Markdown. We use a table layout in order to capture the ping results and display a table that captures these ping results. The following snippet captures the table generated for the `mxpe01` ping test report:

```
$ cat ping_report.md

# BEGIN ANSIBLE MANAGED BLOCK
 Node | Destination | Packet Loss | Delay |
 -----| ------------| ------------| ------|
 mxpe01 | 10.100.1.254/32 | 0% | 3.75
 mxpe01 | 10.100.1.253/32 | 0% | 2.09
 mxpe01 | 10.100.1.1/32 | 0% | 0.27
 mxpe01 | 10.100.1.2/32 | 0% | 4.72
 mxpe01 | 10.100.1.3/32 | 100% | 'N/A'
 # END ANSIBLE MANAGED BLOCK
```

Here is the rendered Markdown table for the ping result:



Node	Destination	Packet Loss	Delay
mxpe01	10.100.1.254/32	0%	3.75
mxpe01	10.100.1.253/32	0%	2.09
mxpe01	10.100.1.1/32	0%	0.27
mxpe01	10.100.1.2/32	0%	4.72
mxpe01	10.100.1.3/32	100%	'N/A'

See also...

For more information regarding the `junos_ping` module and the different parameters supported by this module, please consult the following URL: `https://docs.ansible.com/ansible/latest/modules/junos_ping_module.html`.

Retrieving operational data from Juniper devices

In this recipe, we will outline how to execute operational commands on Juniper devices and store these outputs in text files for further processing.

Getting ready

NETCONF must be enabled on the Juniper devices in order to follow along with this recipe.

How to do it...

1. Install the `jxmlease` Python package, as follows:

```
$ pip3 install jxmlease
```

2. Create a new playbook called `pb_get_ospf_peers.yml` and populate it with the following task to extract OSPF peering information:

```
---
- name: "Get OSPF Status"
  hosts: junos
  tasks:
    - name: "Get OSPF Neighbours Data"
      junos_command:
        commands: show ospf neighbor
        display: xml
      register: ospf_output

    - name: "Extract OSPF Neighbour Data"
      set_fact:
        ospf_peers: "{{ ospf_output.output[0]['rpc-reply']\
                        ['ospf-neighbor-information']['ospf-
neighbor'] }}"
```

3. Update the `pb_get_ospf_peers.yml` playbook with the following task to validate that all OSPF peerings across all nodes are in a `Full` state:

```
    - name: "Validate All OSPF Peers are in Full State"
      assert:
        that: item['ospf-neighbor-state'] == 'Full'
        fail_msg: "Peer on Interface {{item['interface-name']}} is
Down"
        success_msg: "Peer on Interface {{item['interface-name']}}
is UP"
      loop: "{{ospf_peers}}"
```

How it works...

One of the advantages of using the NETCONF API to interact with Juniper devices is that we can get a structured output for all the operational commands that we execute on the Juniper devices. The output that the device returns to us over the NETCONF session is in XML, and Ansible uses a Python library called `jxmlease` to decode this XML and transform it to JSON for better representation. That is why our first task was to install the `jxmlease` Python package.

We use the `junos_command` module to send operational commands to a Juniper device, and we specify that we need XML as the output format that gets returned from the node. This XML data structure is transformed to JSON using the `jxmlease` package by Ansible. We save this data using the `register` keyword to a new variable called `ospf_output`. Here is a sample of the JSON data that is returned from this command:

```
"msg": [
    {
        "rpc-reply": {
            "ospf-neighbor-information": {
                "ospf-neighbor": [
                    {
                        "activity-timer": "34",
                        "interface-name": "ge-0/0/0.0",
                        "neighbor-address": "10.1.1.2",
                        "neighbor-id": "10.100.1.254",
                        "neighbor-priority": "128",
                        "ospf-neighbor-state": "Full"
                    },
                    {
                        "activity-timer": "37",
                        "interface-name": "ge-0/0/1.0",
                        "neighbor-address": "10.1.1.8",
                        "neighbor-id": "10.100.1.253",
                        "neighbor-priority": "128",
                        "ospf-neighbor-state": "Full"
                    }
                ]
            }
        }
    }
]
```

All this data structure is contained in the `ospf_output.output[0]` variable, and we use the `set_fact` module to capture the `ospf-neigbour` data. After that, we use the `assert` module to loop through all the OSPF peers in this data structure and validate that the OSPF neighbor state is equal to `Full`. If all the OSPF peers are in a `Full` state, the task will succeed. However, if the OSPF state is in any other state, the task will fail.

There's more...

If we need to get the operational data from Juniper devices in text format for log collection, we can use the `junos_command` module without the `xml` display option, as shown in this new playbook:

```
$ cat pb_collect_output.yml

---
- name: Collect Network Logs
  hosts: junos
  vars:
    log_folder: "logs"
    op_cmds:
      - show ospf neighbor
  tasks:
    - name: "P1T1: Build Directories to Store Data"
      block:
        - name: "Create folder to store Device config"
          file:
          path: "{{ log_folder }}"
          state: directory
      run_once: yes
      delegate_to: localhost

    - name: "P1T2: Get Running configs from Devices"
      junos_command:
        commands: "{{ item }}"
      loop: "{{ op_cmds }}"
      register: logs_output

    - name: "P1T3: Save Running Config per Device"
      copy:
        content: "{{ item.stdout[0] }}"
        dest: "{{ log_folder }}/{{inventory_hostname}}_{{ item.item |
regex_replace(' ','_') }}.txt"
      loop: "{{ logs_output.results }}"
      delegate_to: localhost
```

This playbook will collect the `show ospf neigbor` command from all the devices, and store them in a new folder called `logs`. Here is the content of the `logs` folder after running the playbook:

```
$ tree logs
 logs
  ├── mxp01_show_ospf_neighbor.txt
  ├── mxp02_show_ospf_neighbor.txt
```

```
    ├─── mxpe01_show_ospf_neighbor.txt
    └─── mxpe02_show_ospf_neighbor.txt
```

We can check the content of one of the files to confirm that the required output is captured:

```
 $ cat logs/mxpe01_show_ospf_neighbor.txt

Address Interface State ID Pri Dead
 10.1.1.2 ge-0/0/0.0 Full 10.100.1.254 128 35
 10.1.1.8 ge-0/0/1.0 Full 10.100.1.253 128 37
```

Validating the network state using PyEZ operational tables

In this recipe, we will outline how to use Juniper custom Ansible modules to validate the network state. We are going to use the Juniper PyEZ Python library and PyEZ operational tables and views to validate the operational state for Junos OS devices.

Getting ready

NETCONF must be enabled on the Juniper devices in order to follow along with this recipe.

How to do it...

1. Install the `junos-eznc` Python package, as follows:

   ```
   $ pip3 install junos-eznc
   ```

2. Install the `Juniper.junos` Ansible role using `Ansible-galaxy`, as follows:

   ```
   $ Ansible-galaxy install Juniper.junos
   ```

3. Create a new playbook called `pb_jnpr_pyez_table.yml`, and populate it with the following task to extract BGP peering information using PyEZ tables:

   ```
   $ cat pb_jnpr_pyez_table.yml

   ---
   - name: Validate BGP State using PyEZ Tables
   ```

```
hosts: junos
roles:
  - Juniper.junos
tasks:
  - name: Retrieve BGP Neighbor Information Using PyEZ Table
    Juniper_junos_table:
      file: "bgp.yml"
    register: jnpr_pyez_bgp
```

4. Update the playbook with the following task to validate that all BGP peering across all nodes is operational:

```
      - name: Validate all BGP Peers are operational
        assert:
          that:
            - item.peer in jnpr_pyez_bgp.resource |
map(attribute='peer_id') | list
          fail_msg: " BGP Peer {{ item.peer }} is Not Operational"
        loop: "{{ bgp_peers }}"
```

How it works...

In addition to the built-in Juniper modules that come pre-installed with Ansible that we have outlined in all our previous recipes, there are additional Ansible modules that are maintained by Juniper and are not part of the Ansible release. These modules are packaged in an Ansible role that is maintained in Ansible Galaxy, and all these modules are based on the Juniper PyEZ Python library that is also developed and maintained by Juniper.

The Juniper PyEZ Python library provides a simple and robust API in order to interact with Juniper devices and simplifies how to manage Juniper devices using Python. The Ansible modules maintained by Juniper are all dependent on the PyEZ Python library, and therefore the first task we need to perform is to ensure that PyEZ (`junos-eznc`) is installed on our Ansible control machine.

The Ansible modules maintained and developed by Juniper are packaged as an Ansible role, and they provide multiple modules with extra capabilities compared to the built-in Juniper modules that come as part of the Ansible release. We install this role using Ansible Galaxy in order to start to utilize these extra modules. The following snippet outlines the extra modules that are part of this role:

```
$ tree ~/.Ansible/roles/Juniper.junos/library/

/home/Ansible/.Ansible/roles/Juniper.junos/library/
  ├── Juniper_junos_command.py
```

```
    ├─── Juniper_junos_config.py
    ├─── Juniper_junos_facts.py
    ├─── Juniper_junos_jsnapy.py
    ├─── Juniper_junos_ping.py
    ├─── Juniper_junos_pmtud.py
    ├─── Juniper_junos_rpc.py
    ├─── Juniper_junos_software.py
    ├─── Juniper_junos_srx_cluster.py
    ├─── Juniper_junos_system.py
    └─── Juniper_junos_table.py
```

In this recipe, we outline how to use the `Juniper_junos_table` Ansible module, which uses the PyEZ tables and views to execute operational commands on Juniper devices and extract specific information from the Juniper device. It also parses this information into a consistent data structure, which we can utilize in our automation scripts. In our playbook, our first task is to use the `Juniper_junos_table` module using the `bgp.yml` table definition (which is present as part of the `junos-eznc` installation). We do this to get the BGP peers on a device and return the relevant information in a consistent data structure. The following snippet outlines the BGP data returned by the `Juniper_junos_table` for the BGP information on `mxpe01`:

```
ok: [mxpe01] => {
    "jnpr_pyez_bgp": {
        "changed": false,
        "failed": false,
        "msg": "Successfully retrieved 1 items from bgpTable.",
        "resource": [
            {
                "local_address": "10.100.1.1+179",
                "local_as": "65400",
                "local_id": "10.100.1.1",
                "peer_as": "65400",
                "peer_id": "10.100.1.254",
                "route_received": [
                    "0",
                    "2",
                    "1",
                    "1"
                ]
            }
        ],
    }
}
```

The last task in our playbook is using the `assert` module in order to validate that all our BGP peers (defined under the `host_vars)` directory) are present in the returned data structure in the BGP table, which indicates that all the BGP peers are operational.

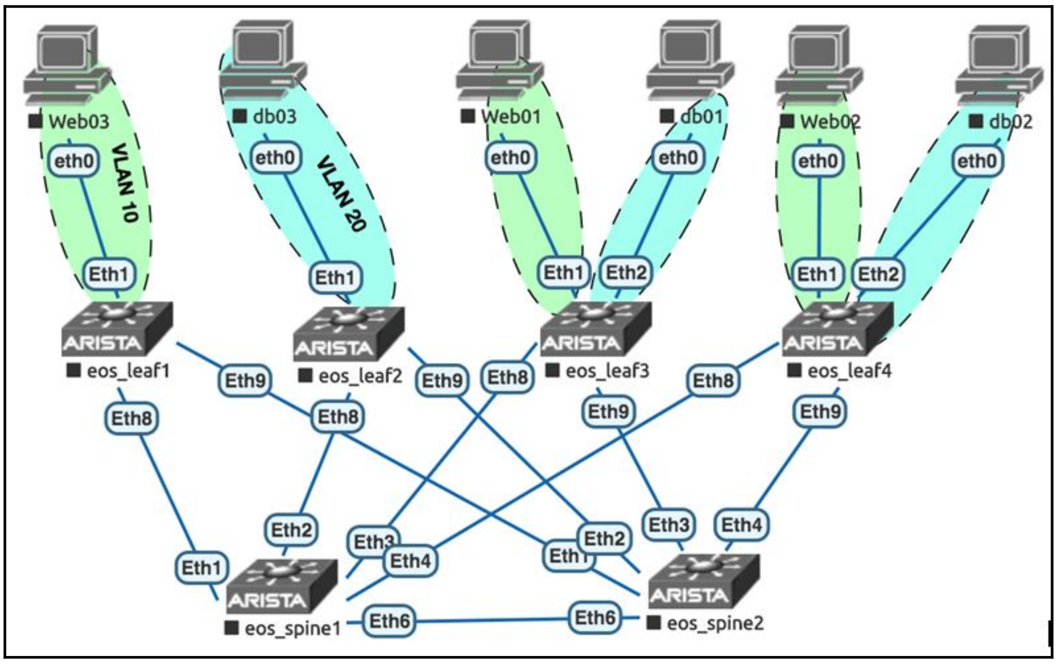
See also...

For more information regarding the Juniper Ansible modules maintained by Juniper, please consult the following URL: `https://www.juniper.net/documentation/en_US/junos-ansible/topics/reference/general/junos-ansible-modules-overview.html.`

For more information regarding PyEZ tables and views, please consult the following URL: `https://www.Juniper.net/documentation/en_US/junos-pyez/topics/concept/junos-pyez-tables-and-views-overview.html.`

4
Building Data Center Networks with Arista and Ansible

In this chapter, we will outline how to automate Arista switches in a typical data center environment in a leaf-spine architecture. We will explore how to interact with Arista devices using Ansible, and how to deploy **virtual local area networks** (**VLANs**) and **virtual extensible LANs** (**VXLANs**) in a **Border Gateway Protocol/Ethernet virtual private network** (**BGP/EVPN)** setup on the Arista switches using various Ansible modules. We will base our illustration on the following sample network diagram of a basic leaf-spine **data center network** (**DCN**):



The following table outlines the devices in our sample topology and their respective management **internet protocols** (**IPs**):

Device	Role	Vendor	Management (MGMT) Port	MGMT IP
Spine01	Spine Switch	Arista vEOS 4.20	Management1	`172.20.1.35`
Spine02	Spine Switch	Arista vEOS 4.20	Management1	`172.20.1.36`
Leaf01	Leaf Switch	Arista vEOS 4.20	Management1	`172.20.1.41`
Leaf02	Leaf Switch	Arista vEOS 4.20	Management1	`172.20.1.42`
Leaf03	Leaf Switch	Arista vEOS 4.20	Management1	`172.20.1.43`
Leaf04	Leaf Switch	Arista vEOS 4.20	Management1	`172.20.1.44`

The main recipes covered in this chapter are as follows:

- Building the Ansible network inventory
- Connecting to and authenticating Arista devices from Ansible
- Enabling **extensible operating system** (**EOS**) **API** (**eAPI**) on Arista devices
- Configuring generic system options on Arista devices
- Configuring interfaces on Arista devices
- Configuring the underlay BGP on Arista devices
- Configuring the overlay BGP/EVPN on Arista devices
- Deploying the configuration on Arista devices
- Configuring VLANs on Arista devices
- Configuring VXLAN tunnels on Arista devices
- Gathering Arista device facts
- Retrieving operational data from Arista devices

Technical requirements

The code for all the recipes in this chapter can be found on the following GitHub repo: `https://github.com/PacktPublishing/Network-Automation-Cookbook/tree/master/ch4_arista`.

This chapter is based on the following software releases:

- Ansible machine running CentOS 7
- Ansible 2.9
- Arista **virtualized EOS** (**vEOS**) running EOS 4.20.1F

Check out the following video to see the Code in Action:
`https://bit.ly/3coydTp`

Building the Ansible network inventory

In this recipe, we will outline how to build and structure the Ansible inventory to describe our sample leaf-spine **direct current** (**DC**) network. The Ansible inventory is a pivotal part of Ansible as it outlines and groups devices that should be managed by Ansible.

Getting ready

We need to create a new folder that will host all the files that we will create in this chapter. The new folder should be named `ch4_arista`.

How to do it...

1. Inside the new folder (`ch4_arista`), we create a `hosts` file with the following content:

```
$ cat hosts

[leaf]
 leaf01 ansible_host=172.20.1.41
 leaf02 ansible_host=172.20.1.42
 leaf03 ansible_host=172.20.1.43
 leaf04 ansible_host=172.20.1.44

[spine]
 spine01 ansible_host=172.20.1.35
 spine02 ansible_host=172.20.1.36

[arista:children]
 leaf
 spine
```

2. Create an `ansible.cfg` file, as shown in the following code block:

```
$ cat ansible.cfg

[defaults]
 inventory=hosts
```

```
retry_files_enabled=False
gathering=explicit
host_key_checking=False
```

How it works...

Defining an Ansible inventory is mandatory, in order to describe and classify the devices in our network that should be managed by Ansible. In the Ansible inventory, we also specify the IP addresses through which Ansible will communicate with these managed devices, using the `ansible_host` parameter.

We built the Ansible inventory using the `hosts` file and we defined multiple groups in order to group the different devices in our topology. These groups are as follows:

- We created the `leaf` group, which references all the `leaf` switches in our topology.
- We created the `spine` group, which references all the `spine` switches in our topology.
- We created the `arista` group, which references both the `leaf` and `spine` groups.

Finally, we created the `ansible.cfg` file and configured it to point to our `hosts` file, to be used as the Ansible inventory file. Further, we disabled the `setup` module (by setting `gathering` to `explicit)`, which is not needed when running Ansible against network nodes.

Connecting to and authenticating Arista devices from Ansible

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. We are going to use a username and password to authenticate the Arista devices in our topology.

Getting ready

In order to follow along with this recipe, an Ansible inventory file should be constructed as per the previous recipe. IP reachability between the Ansible control machine and all the devices in the network must also be implemented.