Juniper Networks | Engineering Simplicity

Junos® OS

Ansible for Junos OS Developer Guide

Published
2024-11-15

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Junos® OS Ansible for Junos OS Developer Guide

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at https://support.juniper.net/support/eula/. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

About This Guide

Use this guide to automate the provisioning and management of Junos devices with Ansible software.

RELATED DOCUMENTATION

- Ansible for Junos OS Module Documentation
- Junos PyEZ API Documentation

1
CHAPTER

Disclaimer

Ansible for Junos OS Disclaimer

Use of the Ansible for Junos OS software implies acceptance of the terms of this disclaimer, in addition to any other licenses and terms required by Juniper Networks.

Juniper Networks is willing to make the Ansible for Junos OS software available to you only upon the condition that you accept all of the terms contained in this disclaimer. Please read the terms and conditions of this disclaimer carefully.

The Ansible for Junos OS software is provided *as is*. Juniper Networks makes no warranties of any kind whatsoever with respect to this software. All express or implied conditions, representations and warranties, including any warranty of non-infringement or warranty of merchantability or fitness for a particular purpose, are hereby disclaimed and excluded to the extent allowed by applicable law.

In no event will Juniper Networks be liable for any direct or indirect damages, including but not limited to lost revenue, profit or data, or for direct, special, indirect, consequential, incidental or punitive damages however caused and regardless of the theory of liability arising out of the use of or inability to use the software, even if Juniper Networks has been advised of the possibility of such damages.

2

CHAPTER

Ansible Overview

Understanding Ansible for Junos OS

SUMMARY

You can use Ansible to deploy and manage Junos devices.

IN THIS SECTION

Ansible for Junos OS Overview

Ansible is an IT automation framework that is used for infrastructure configuration management. Ansible supports automating the network infrastructure in addition to the compute and cloud infrastructure, and Juniper Networks supports using Ansible to manage Junos devices. You can use Ansible to perform operational and configuration tasks on Junos devices, including retrieving information, managing the configuration, installing and upgrading Junos OS, and resetting, rebooting, or shutting down managed devices.

Ansible comes in several varieties. You can use the following applications to manage Junos devices:

- Ansible Core—Free, open-source, base version of Ansible
- Red Hat Ansible Tower—Commercial application that is a superset of Ansible Core with additional features such as a visual dashboard, role-based access control, job scheduling, and graphical inventory management
- AWX—Open-source upstream project for Ansible Tower

Ansible uses a client-server architecture. You install the Ansible software on the control node, which is a Unix-like system that performs operations on one or more managed nodes. Ansible uses an agentless architecture and thus does not require installing any Ansible-specific software on the managed devices. Although Ansible typically requires Python on the managed nodes, it is not required to manage Junos devices. Instead, Ansible for Junos OS requires all tasks to run locally on the Ansible control node and uses the Junos XML API over NETCONF to interface with Junos devices.

Ansible typically uses a push model in which the server sends state information to the managed nodes on demand. Ansible modules are discrete units of code that perform the requested functions on a node. The managed node executes the job and returns the result to the server. In general, Ansible modules are

idempotent such that executing the same playbook or operation multiple times yields the same result, because the modules only apply a change if it's required.

Juniper Networks and Ansible provide modules that enable you to manage Junos devices. The Juniper Networks modules are distributed through collections. For more information about the available collections and modules, see "Understanding the Ansible for Junos OS Collections and Modules" on page 7.

Ansible is written in Python, but it uses simple YAML syntax to express automation jobs. Thus, Ansible users can get started quickly, because they do not require extensive knowledge of Python to use Ansible. Ansible also leverages the Jinja2 templating language to enable dynamic expressions and access to variables.

Benefits of Ansible and Ansible for Junos OS

- Use a simple, easy-to-learn syntax
- Accelerate the time to deploy new network devices and applications
- Provide an efficient and scalable solution for managing large numbers of devices
- Increase operational efficiency by automating tasks and reducing the manual configuration and management of devices
- Minimize risk and errors through standardization
- Improve change management processes
- Use an agentless architecture

Additional Resources

This documentation assumes that the reader is familiar with the Ansible framework. Table 1 on page 6 provides resources for using Ansible to manage Junos devices.

Table 1: Ansible for Junos OS Resources

Resource	Description	URL
Ansible for Junos OS documentation	Documentation detailing how to use Ansible to manage Junos devices.	https://www.juniper.net/documentation/product/us/en/ansible-for-junos-os/
Ansible for Junos OS modules overview	Documentation that outlines the different modules available for managing Junos devices.	"Understanding the Ansible for Junos OS Collections and Modules" on page 7
Ansible Galaxy	Ansible Galaxy website and Juniper Networks Ansible Galaxy content.	https://galaxy.ansible.com https://galaxy.ansible.com/ui/namespaces/juniper/
Ansible website and documentation	Official Ansible website and documentation.	https://www.ansible.com https://docs.ansible.com/
GitHub repository	Public repository for the Ansible for Junos OS project. This repository includes the most current source code, installation instructions, and release note summaries for all releases.	https://github.com/Juniper/ansible-junos-stdlib/
Juniper Networks `juniper.device` collection	Download site and API reference documentation for the Juniper Networks `juniper.device` collection.	https://galaxy.ansible.com/ui/repo/published/juniper/device/ https://ansible-juniper-collection.readthedocs.io/

RELATED DOCUMENTATION

Understanding the Ansible for Junos OS Collections and Modules | 7

Ansible for Junos OS Server Requirements | 21

Set up Ansible for Junos OS Managed Nodes | 24

Create and Execute Ansible Playbooks to Manage Junos Devices | 13

Understanding the Ansible for Junos OS Collections and Modules

SUMMARY

Juniper Networks provides Ansible modules that you can use to manage Junos devices.

IN THIS SECTION

Understanding Ansible Collections and Modules for Managing Junos Devices

Ansible is an IT automation framework that is used for infrastructure configuration management. You use Ansible modules, which are discrete units of code, to perform specific functions on a managed node. You can execute individual modules on remote hosts to perform ad-hoc tasks, or you can execute modules through playbooks.

Ansible and Juniper Networks provide Ansible modules that you can use to manage Junos devices. The Juniper Networks Ansible modules are grouped and distributed through Ansible collections, which are hosted in the Ansible Galaxy repository. Table 2 on page 7 outlines the different content sets available for managing Junos devices.

Table 2: Ansible Content Sets for Managing Junos Devices

Content Set	Description	Ansible Releases
`juniper.device` collection	Collection for managing Junos devices, which Juniper Networks provides and manages as an open-source project.	Ansible 2.10 and later

Table 2: Ansible Content Sets for Managing Junos Devices *(Continued)*

Content Set	Description	Ansible Releases
`junipernetworks.junos` collection	Collection for managing Junos devices, which Ansible provides, maintains, and supports.	Ansible 2.10 and later
Ansible core modules (deprecated)	Ansible modules included in the Ansible base installation. In Ansible 2.10, the core modules moved from the base installation into Ansible's `junipernetworks.junos` collection.	Ansible 2.1 through Ansible 2.9
`Juniper.junos` role (deprecated)	Role for managing Junos devices, which Juniper Networks provides, maintains, and supports. This role is superseded by the `juniper.device` collection.	Ansible 2.1 and later

An Ansible role is a set of tasks and supporting variables, files, templates, and modules for configuring a host. Starting in Ansible 2.10, Ansible supports Ansible Content Collections, a format for distributing Ansible content that is not included as part of the Ansible base installation. Ansible collections can include a wider range of content, including modules, playbooks, plugins, and roles. Ansible collections also have their own repositories and can be developed and released independently from the Ansible base installation.

In Ansible 2.9 and earlier, you can manage Junos devices by using the modules provided in the Juniper Networks `Juniper.junos` role or by using the core modules provided as part of the Ansible base installation. Starting in Ansible 2.10, the `Juniper.junos` role and the Ansible core modules are superseded by the corresponding collection. With the introduction of Juniper Networks' `juniper.device` collection, the modules in the `Juniper.junos` role were duplicated under new names in the collection and thus retain the same functionality and parameters as the original modules, with the exception of the `provider` parameter. We recommend that you use the `juniper.device` collection, because new features are only being added to the collection going forward.

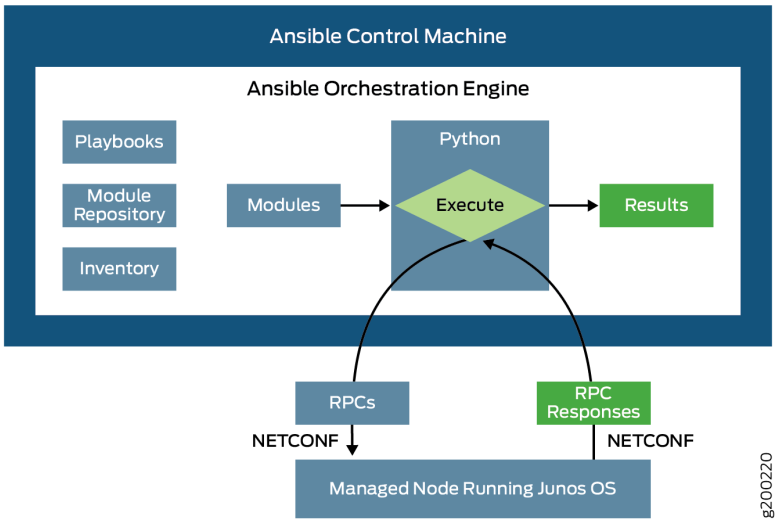
How to Execute Modules on Junos Devices

To use the collections that are hosted in the Ansible Galaxy repository, you must first install Ansible on the control node and then install the collection. For more information about installing the `juniper.device` collection, see "Ansible for Junos OS Server Requirements" on page 21.

Ansible modules can perform operations on a managed node. Typically, the Ansible control node sends a module to a managed node, where it is executed and then removed. In this scenario, the managed node must have the ability to execute the module. Because most Ansible modules are written in Python, Ansible typically requires Python on the managed node.

The Juniper Networks modules in the `juniper.device` collection, however, do not require Python on the managed nodes. In contrast to the typical operation, you execute the modules locally on the Ansible control node, and the modules use Junos PyEZ and the Junos XML API over NETCONF to interface with the managed node. This method of execution enables you to use Ansible to manage any supported Junos device. Figure 1 on page 9 illustrates the communication between the Ansible control node and a managed Junos device.

Figure 1: Ansible Communication with a Junos Device



To use the `juniper.device` collection modules, the playbook or command must:

- Specify the collection or FQCN—To specify the collection, include the `collections` key in the play. Alternatively, you can omit the `collections` key and instead reference collection content by its fully qualified collection name (FQCN), which is the recommended method.
- Execute the modules locally on the control node—To run Ansible modules locally, you define the `connection` parameter as `local`, for example, by including `connection: local` in your playbook or including `--connection local` on the command line.

> **NOTE**: When you use `connection: local`, Ansible establishes a separate connection to the device for each task in the play that requires a connection. The `juniper.device` collection

modules also support using `connection: juniper.device.pyez`, which still executes the modules locally but instead establishes a single, persistent connection to a device for all tasks in a play.

- Provide appropriate connection and authentication information to connect to the managed device—For more information, see:
  - "Connect to Junos Devices Using Ansible" on page 29
  - "Authenticate Users Executing Ansible Modules on Junos Devices" on page 37

You can execute Ansible modules using any user account that has access to the managed Junos device. When you execute Ansible modules, Junos OS user account access privileges are enforced, and the class configured for the Junos OS user account determines the permissions. Thus, if a user executes a module that loads configuration changes onto a device, the user must have permissions to change the relevant portions of the configuration.

The following playbook executes the `juniper.device` collection's `facts` module to retrieve the device facts and save them to a file. The example uses existing SSH keys in the default location to authenticate with the device and thus does not explicitly provide credentials in the playbook.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve facts from a Junos device
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

You can also perform ad-hoc operations on the command line. The following command executes the `juniper.device` collection's `facts` module and retrieves device facts from hosts in inventory group dc1.

```
user@ansible-cn:~$ ansible --connection local -i production dc1 -m juniper.device.facts
```

Juniper Networks juniper.device Collection

Juniper Networks provides the `juniper.device` Ansible Content Collection, which is hosted on the Ansible Galaxy website. The collection includes Ansible modules that enable you to manage Junos devices.

Table 3 on page 11 outlines the modules in the `juniper.device` collection. In the collection's initial release, the collection modules retain the same functionality and parameters as the corresponding module in the `Juniper.junos` role, with the exception of the `provider` parameter, which is not supported for the collection modules.

For the most current list, documentation, and usage examples for the modules, see https://ansible-juniper-collection.readthedocs.io/.

Table 3: juniper.device Collection Modules

juniper.device Module Name	Description
`command`	Execute CLI commands on the Junos device and save the output locally.
`config`	Manage the configuration of Junos devices.
`facts`	Retrieve device-specific information from the remote host, including the Junos OS version, serial number, and hardware model number.
`jsnapy`	Execute Junos Snapshot Administrator in Python (JSNAPy) tests through Ansible.
`file_copy`	Transfer a file between the local Ansible control node and the Junos device.
`ping`	Execute the `ping` command on Junos devices.
`pmtud`	Perform path MTU discovery on Junos devices.
`rpc`	Execute Junos OS RPCs.
`software`	Install a Junos OS software package and reboot a Junos device.

Table 3: juniper.device Collection Modules *(Continued)*

juniper.device Module Name	Description
`system`	Perform system operations on Junos devices, including resetting, rebooting, or shutting down the device.
`srx_cluster`	Create an SRX Series chassis cluster for cluster-capable SRX Series Firewalls.
`table`	Use Junos PyEZ operational Tables and Views to retrieve operational information from Junos devices.

RELATED DOCUMENTATION

- Understanding Ansible for Junos OS | 4
- Authenticate Users Executing Ansible Modules on Junos Devices | 37
- Connect to Junos Devices Using Ansible | 29

Understanding the Ansible Inventory File When Managing Junos Devices

The Ansible inventory file defines the hosts and groups of hosts upon which commands, modules, and tasks in a playbook operate. The file can be in one of many formats depending on your Ansible environment and plugins. Common formats include INI and YAML. The default location for the inventory file is **/etc/ansible/hosts**. You can also create project-specific inventory files in alternate locations.

The inventory file can list individual hosts or user-defined groups of hosts. This enables you to define groups of Junos devices with similar roles upon which to perform the same operational and configuration tasks. For example, if you are managing one or more data centers, you can create Ansible groups for those switches that require the same set of operations, such as upgrading Junos OS and rebooting the device.

In order to manage Junos devices using Ansible, you must have a Junos OS login account with appropriate access privileges on each device where Ansible modules are executed. You must ensure that usernames and passwords or access keys exist for each host in the file.

The following INI-formatted sample inventory file defines an individual host, `host1`, and two groups of hosts, `routers` and `switches`:

```
host1.example.net

[routers]
router1.example.net
router2.example.net

[switches]
switch1.example.net
switch2.example.net
```

For more information about the Ansible inventory file, see the official Ansible documentation at https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html.

RELATED DOCUMENTATION

- Understanding Ansible for Junos OS | 4
- Authenticate Users Executing Ansible Modules on Junos Devices | 37
- Troubleshoot Ansible Connection Errors When Managing Junos Devices | 199

Create and Execute Ansible Playbooks to Manage Junos Devices

SUMMARY

You can create Ansible playbooks that execute Juniper Networks modules to perform operational and configuration tasks on Junos devices.

IN THIS SECTION

Juniper Networks provides Ansible modules that enable you to perform operational and configuration tasks on Junos devices. This topic outlines how to create a simple Ansible playbook to execute Juniper Networks modules on Junos devices.

You create Ansible playbooks to handle more complex management tasks. Playbooks consist of one or more plays, or groups of tasks, that operate on a set of defined hosts. Ansible hosts that are referenced in the playbook must be defined in the Ansible inventory file, which by default resides at **/etc/ansible/hosts**.

Each playbook play must specify:

- The hosts on which the tasks operate
- The list of tasks to execute on each host
- Any required variables or module parameters, including authentication parameters, if these are not defined elsewhere

A playbook executes tasks on a host by calling modules. The Juniper Networks Ansible modules are distributed through the `juniper.device` collection, which is hosted on Ansible Galaxy. To use the Juniper Networks modules in your playbook, you must install the collection on the Ansible control node. For more information about the collection and modules, see "Understanding the Ansible for Junos OS Collections and Modules" on page 7.

The Juniper Networks modules do not require Python on Junos devices because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on Junos devices, you must run modules locally on the Ansible control node, where Python is installed. You can run the modules locally by including `connection: local` in the playbook play. When you use `connection: local`, Ansible establishes a separate connection to the host for each task in the play that requires a connection. The `juniper.device` collection modules also support `connection: juniper.device.pyez`. This connection type still executes the modules locally on the Ansible control node, but it establishes a connection to the host that persists over the execution of all tasks in a play.

By default, Ansible plays automatically gather system facts from the remote host. However, when you execute the plays locally, Ansible gathers the facts from the Ansible control node instead of the remote host. To avoid gathering facts for the control node, include `gather_facts: no` in the playbook.

When you execute the Juniper Networks modules using a NETCONF session over SSH, which is the default, you must have NETCONF enabled on the Junos device. We recommend that you create a simple task in the playbook that explicitly tests whether NETCONF is enabled on each device before executing other tasks. If this task fails for any host, by default, Ansible does not execute the remaining

tasks for this host. Without this test, you might get a generic connection error during playbook execution that does not indicate whether this or another issue is the cause of any failures.

Playbooks are expressed in YAML. YAML is white-space sensitive and indentation is significant. Therefore, your playbooks should always use spaces rather than tabs. In YAML, items preceded by a hyphen (-) are considered list items, and the `key: value` notation represents a hash. For detailed information about creating Ansible playbooks, refer to the official Ansible documentation at https://docs.ansible.com/ansible/latest/playbook_guide/playbooks.html.

The following sections outline the steps for creating and running a simple playbook that executes Ansible modules on a Junos device.

Create a Playbook

To create a simple playbook to perform tasks on Junos devices:

1. In your favorite editor, create a new file with a descriptive playbook name that uses the **.yaml** file extension.
2. Include three dashes to indicate the start of the YAML document.

```
---
```

3. Provide a descriptive name for the play.

```
---
- name: Get Device Facts
```

4. Define a colon-delimited list of the hosts or groups of hosts on which the modules will operate, or specify `all` to indicate all hosts in the inventory file.

   Any hosts or groups referenced in the playbook must be defined in the Ansible inventory file.

```
---
- name: Get Device Facts
  hosts: dc1
```

5. Instruct Ansible to execute the play's tasks locally on the Ansible control node where Python is installed because there is no requirement for Python on Junos devices.

- Include `connection: local` to execute tasks locally but establish a separate connection to the host for each task in the play that requires a connection.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
```

- Include `connection: juniper.device.pyez` to execute tasks locally but establish a persistent connection to the host that persists over the execution of all tasks in the play. This connection type is only supported by the `juniper.device` collection modules.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: juniper.device.pyez
```

The remaining steps use `connection: local`. To use the `juniper.device` collection modules with a persistent connection, update the final playbook to use `connection: juniper.device.pyez`.

6. (Optional) Include `gather_facts: no` to avoid gathering facts for the target host, which for local connections is the Ansible control node.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
```

7. (Optional) Reference the `juniper.device` collection.

   You can define the `collections` key in the playbook and then reference just the module name in the task. However, the recommended method is to omit the `collections` key and instead reference collection content by its fully qualified collection name (FQCN).

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
```

```
  collections:
    - juniper.device
```

This example omits the `collections` key and uses the fully qualified collection name.

8. Define a tasks section, and include one or more tasks as list items.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
```

9. (Optional) As an additional check, create a task to verify NETCONF connectivity for each Junos device.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: 830
        timeout: 5
```

10. Create tasks that use the Juniper Networks modules, and provide any necessary connection and authentication parameters.

    This example uses existing SSH keys in the default location and does not explicitly provide credentials for the `facts` module in the playbook.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
```

```
tasks:
  - name: Check NETCONF connectivity
    ansible.builtin.wait_for:
      host: "{{ inventory_hostname }}"
      port: 830
      timeout: 5

  - name: Retrieve information from Junos devices
    juniper.device.facts:
      savedir: "{{ playbook_dir }}"

  - name: Print version
    ansible.builtin.debug:
      var: junos.version
```

11. (Optional) Define additional tasks or additional plays as needed.

Execute the Playbook

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
user@ansible-cn:~$ ansible-playbook junos-get-facts.yaml

PLAY [Get Device Facts] ********************************************************

TASK [Check NETCONF connectivity] **********************************************
ok: [dc1a.example.net]

TASK [Retrieve information from Junos devices] *********************************
ok: [dc1a.example.net]

TASK [Print version] ***********************************************************
ok: [dc1a.example.net] => {
    "junos.version": "19.4R1.10"
}
```

```
PLAY RECAP *********************************************************************
dc1a.example.net  : ok=3    changed=0    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

RELATED DOCUMENTATION

Understanding Ansible for Junos OS | 4

Understanding the Ansible for Junos OS Collections and Modules | 7

Understanding the Ansible Inventory File When Managing Junos Devices | 12

Authenticate Users Executing Ansible Modules on Junos Devices | 37

CHAPTER 3

Install Ansible for Junos OS

Ansible for Junos OS Server Requirements

IN THIS SECTION

Juniper Networks supports using Ansible to manage Junos devices and provides Ansible modules that you can use to perform operational and configuration tasks on the devices. Ansible supports Ansible Content Collections, or collections, starting in Ansible 2.10. The Juniper Networks modules are distributed through the following Ansible collection, which is hosted on the Ansible Galaxy website:

- `juniper.device` collection

You install Ansible on a control node with a Unix-like operating system. You can install Ansible and the Juniper Networks collection:

- Directly on the configuration management server
- Within a Python virtual environment
- As a Docker container

The Juniper Networks juniper/pyez-ansible Docker image is a lightweight, self-contained system that bundles Ansible, the Juniper Networks modules, and all dependencies into a single portable container. The Docker container enables you to quickly run Ansible in interactive mode or as an executable package on any platform that supports Docker.

To install Ansible and the `juniper.device` collection under the system-installed Python or in Python virtual environment on the control node, see the following sections:

- "Prerequisite Software" on page 22
- "Install the juniper.device Collection" on page 22

To use the Ansible for Junos OS Docker image, see the following section:

- "Use the Ansible for Junos OS Docker Image" on page 23

Prerequisite Software

Before you install the Juniper Networks `juniper.device` collection and begin using Ansible to manage Junos devices, you must install the following prerequisite software in your selected environment on the Ansible control node. You can install the software under the system-installed Python or in a Python virtual environment.

- Python 3.8 or later
- Ansible 2.10 or later
- Junos PyEZ (`junos-eznc`) Release 2.6.0 or later.

  For installation instructions and current information about Junos PyEZ, see:

  - Junos PyEZ documentation
  - Junos PyEZ GitHub repository
- The `jxmlease` Python module
- The `xmltodict` Python module
- Junos Snapshot Administrator in Python (JSNAPy) Release 1.3.6 or later (required to use the `jsnapy` module).

  For installation instructions and current information about JSNAPy, see:

  - JSNAPy GitHub repository
  - Junos Snapshot Administrator in Python Documentation

Install the juniper.device Collection

On Ansible control nodes running Ansible 2.10 or later, you can install the `juniper.device` collection from the Ansible Galaxy website. Install the collection in the same environment where you installed Ansible and the other prerequisite software.

To install the `juniper.device` collection:

- Issue the `ansible-galaxy collection install` command and specify the `juniper.device` collection.

```
user@ansible-cn:~$ ansible-galaxy collection install juniper.device
Starting galaxy collection install process
```

```
Process install dependency map
Starting collection install process
Installing 'juniper.device:1.0.6' to '/home/user/.ansible/collections/ansible_collections/
juniper/device'
Downloading https://galaxy.ansible.com/download/juniper-device-1.0.6.tar.gz to /home/
user/.ansible/tmp/ansible-local-23916uzdfbjsk/tmp4nhxnw3v
juniper.device (1.0.6) was installed successfully
```

Use the Ansible for Junos OS Docker Image

Docker is a software container platform that is used to package and run an application and its dependencies in an isolated container. Juniper Networks provides Docker images, which are automatically built for every new release of the Juniper Networks modules. The Docker image includes Python 3, Ansible, Junos PyEZ, the `juniper.device` collection, and Junos Snapshot Administrator in Python along with any required dependencies. You can run the container in interactive mode or use the container as an executable to run your playbooks.

To use the Ansible for Junos OS Docker image on your Ansible control node:

1. Install Docker.

   See the Docker website at https://www.docker.com for instructions on installing and configuring Docker on your specific operating system.

2. Download the juniper/pyez-ansible Docker image from Docker Hub.

   - To download the latest image, issue the following command:

     ```
     user@host:~$ docker pull juniper/pyez-ansible
     ```

     NOTE: The latest Ansible for Junos OS Docker image is built using the most recently committed code in the Juniper/ansible-junos-stdlib GitHub source repository, which is under active development and might not be stable.

   - To download a specific image, append the appropriate release tag to the image name, for example, v1.0.0-collections.

     ```
     user@host:~$ docker pull juniper/pyez-ansible:tag
     ```

3. Run the container.

For instructions on running the container, see the official usage examples at https://github.com/Juniper/ansible-junos-stdlib/blob/master/README.md#docker.

RELATED DOCUMENTATION

Set up Ansible for Junos OS Managed Nodes | 24

Understanding Ansible for Junos OS | 4

Understanding the Ansible for Junos OS Collections and Modules | 7

Set up Ansible for Junos OS Managed Nodes

IN THIS SECTION

Juniper Networks supports using Ansible to manage Junos devices and provides Ansible modules that you can use to perform operational and configuration tasks on the devices. You do not need to install any client software on the remote nodes in order to use Ansible to manage the devices. Also, Python is not required on the managed Junos devices because the Juniper Networks modules are executed locally on the Ansible control node and use Junos PyEZ and the Junos XML API over NETCONF to perform the corresponding operational and configuration tasks.

You can execute Ansible for Junos OS modules using any user account that has access to the managed Junos device. When you execute Ansible modules, Junos OS user account access privileges are enforced. The class configured for the Junos OS user account determines the permissions. Thus, if a user executes a module that loads configuration changes onto a device, the user must have permissions to change the relevant portions of the configuration. For information about configuring user accounts on Junos devices, see the User Access and Authentication Administration Guide for Junos OS.

Juniper Networks provides modules that enable you to connect to Junos devices using NETCONF over SSH or telnet. To manage devices through a NETCONF session over SSH, you must enable the NETCONF service over SSH on the managed device and ensure that the device meets requirements for

SSHv2 connections. The modules also enable you to telnet to the device's management interface or to a console server that is directly connected to the device's **CONSOLE** port. To use Ansible to telnet directly to the device's management interface, you must configure the Telnet service on the managed device.

The following sections outline the requirements and required configuration on Junos devices when you use Ansible to access the device using the different connection protocols.

Enable NETCONF on Junos Devices

To enable NETCONF over SSH on the default port (830) on a Junos device:

1. Configure the NETCONF-over-SSH service.

```
[edit system services]
user@host# set netconf ssh
```

2. Commit the configuration.

```
[edit system services]
user@host# commit
```

Satisfy Requirements for SSHv2 Connections

The NETCONF server communicates with client applications within the context of a NETCONF session. The server and client explicitly establish a connection and session before exchanging data, and close the session and connection when they are finished. The Ansible for Junos OS modules access the NETCONF server using the SSH protocol and standard SSH authentication mechanisms. When you use Ansible to manage Junos devices, the most convenient way to access a device is to configure SSH keys.

To establish an SSHv2 connection with a Junos device, you must ensure that the following requirements are met:

- The NETCONF service over SSH is enabled on each device where a NETCONF session will be established.
- The client application has a user account and can log in to each device where a NETCONF session will be established.

- The login account used by the client application has an SSH public/private key pair or a text-based password configured.
- The client application can access the public/private keys or text-based password.

For information about enabling NETCONF on a Junos device and satisfying the requirements for establishing an SSH session, see the NETCONF XML Management Protocol Developer Guide.

Configure Telnet Service on Junos Devices

The Juniper Networks Ansible modules can telnet directly to a Junos device. To telnet to a Junos device, you must configure the Telnet service on the device. Configuring Telnet service for a device enables unencrypted, remote access to the device.

> **NOTE**: Because telnet uses clear-text passwords (therefore creating a potential security vulnerability), we recommend that you use SSH.

To enable the Telnet service:

1. Configure the service.

```
[edit system services]
user@host# set telnet
```

2. (Optional) Configure the connection limit, rate limit, and order of authentication, as necessary.

```
[edit system services]
user@host# set telnet connection-limit connection-limit
user@host# set telnet rate-limit rate-limit
user@host# set telnet authentication-order [radius tacplus password]
```

3. Commit the configuration.

```
[edit system services]
user@host# commit
```

RELATED DOCUMENTATION

Ansible for Junos OS Server Requirements | 21

Understanding Ansible for Junos OS | 4

4

CHAPTER

Use Ansible to Connect to Junos Devices

Connect to Junos Devices Using Ansible

SUMMARY

The Juniper Networks Ansible modules enable you to connect to Junos devices using SSH, telnet, or serial console connections.

IN THIS SECTION

Juniper Networks provides Ansible modules that you can use to manage Junos devices. The Juniper Networks modules are distributed through the `juniper.device` collection. The modules can connect to Junos devices using different protocols and Ansible connections, which are described in this document.

Connection Methods Overview

The `juniper.device` collection modules enable you to connect to a Junos device using SSH, telnet, or a serial console connection. You must use a serial console connection when your terminal or laptop is physically connected to the **CONSOLE** port on a Junos device. You can use SSH or telnet to connect to the device's management interface or to a console server that is directly connected to the device's **CONSOLE** port.

New or zeroized devices that have factory-default configurations require access through a console connection. Thus, you can use Ansible to initially configure a device that is not yet configured for remote access by using either a serial console connection when you are directly connected to the device or by using telnet or SSH through a console server that is directly connected to the device.

By default, the `juniper.device` modules use SSH to connect to a device. To use telnet or a serial console connection, set the `mode` parameter to the appropriate value. To telnet to a device, set the `mode` argument equal to `"telnet"`. To use a serial console connection, set the `mode` argument equal to `"serial"`. Table 4 on page 30 summarizes the connection modes, their default values for certain parameters, and any required Junos OS configuration. The `juniper.device` modules support all connection modes as of their initial release.

Table 4: Connection Modes for the Juniper Networks Ansible Modules

Connection Mode	Value of `mode` Argument	Default Port	Required Junos OS Configuration

NETCONF over SSH (default)	–	830	`[edit system services]`
			`netconf {`
			`  ssh;`
			`}`

Serial console connection	`serial`	/dev/ttyUSB0	–
SSH through a console server	–	22	–
Telnet to Junos device	`telnet`	23	`[edit system services]` `telnet;`
Telnet through a console server	`telnet`	23	–

NOTE: Before you can access the management interface using telnet or NETCONF over SSH, you must first enable the appropriate service at the `[edit system services]` hierarchy level. Because telnet uses clear-text passwords (therefore creating a potential security vulnerability), we recommend that you use SSH.

When you execute the `juniper.device` modules to manage a Junos device, the remote device must be able to authenticate the Ansible user using credentials appropriate for the given protocol. For more information, see "Authenticate Users Executing Ansible Modules on Junos Devices" on page 37.

The `juniper.device` modules support different Ansible connections when connecting to Junos devices, including local (per-task) connections and persistent (per-play) connections. The Ansible connection determines whether Ansible establishes a separate connection to the host for each task in the play or whether it establishes a single connection to a host that persists over all tasks in the play. For information about specifying the Ansible connection, see "Understanding Local and Persistent Ansible Connections" on page 31.

Understanding Local and Persistent Ansible Connections

The Juniper Networks Ansible modules do not require Python on Junos devices because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on Junos devices, you must run modules locally on the Ansible control node, where Python is installed. You can run the modules locally by including `connection: local` in the playbook play. When you use `connection: local`, Ansible establishes a separate connection to the host for each task in the play that performs operations on the host.

The `juniper.device` collection modules also support `connection: juniper.device.pyez` for establishing a persistent connection to a host. When you use a persistent connection, Ansible still executes the modules locally on the control node, but it only establishes and maintains a single connection to each host, which persists over the execution of all tasks in the play. Establishing a persistent connection to a host can be more efficient for executing multiple tasks than establishing a separate connection to the host for every task in the play.

Table 5 on page 31 summarizes the Ansible connections and the content sets that support them.

Table 5: Ansible Connections Supported By Juniper Networks Modules

Ansible connection	Description	Content Set Support
`connection: local`	Execute the modules locally on the Ansible control node but establish a separate connection to a host for each task in the play that performs operations on the host.	`juniper.device` collection `Juniper.junos` role (deprecated)
`connection: juniper.device.pyez`	Execute the modules locally on the Ansible control node but establish a persistent connection to a host that persists over the execution of all tasks in the play.	`juniper.device` collection

When you use `connection: local`, Ansible establishes a separate connection to a host for each module, which means you can define module-specific connection and authentication parameters in the module's argument list. By contrast, when you use `connection: juniper.device.pyez`, the connection persists across all tasks in the play, and thus you must define the connection and authentication parameters globally for all modules. You can define the parameters in the `vars:` section of a play, in addition to providing them through other means, for example, in an SSH configuration file or in the Ansible inventory file. For additional details, see "Authenticate Users Executing Ansible Modules on Junos Devices" on page 37.

The following playbook establishes a persistent connection to each host that is used for all tasks in the play. The user's credentials, which are stored in an Ansible vault file, are defined in the play's `vars:` section.

```
---
- name: Get Device Information
  hosts: dc1
  connection: juniper.device.pyez
  gather_facts: no

  vars:
    host: "{{ inventory_hostname }}"
    user: "{{ admin_username }}"
    passwd: "{{ admin_password }}"

  vars_files:
    - vault-vars.yaml

  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"

    - name: Get hardware inventory
      juniper.device.command:
        commands: "show chassis hardware"
        dest_dir: "{{ playbook_dir }}"
```

Connect to a Device Using SSH

The Juniper Networks Ansible modules support using SSH to connect to a Junos device. You can establish a NETCONF session over SSH on the device's management interface or you can establish an SSH connection with a console server that is directly connected to the device's **CONSOLE** port. The SSH server must be able to authenticate the user using standard SSH authentication mechanisms, as described in "Authenticate Users Executing Ansible Modules on Junos Devices" on page 37. To establish a NETCONF session over SSH, you must also satisfy the requirements outlined in "Set up Ansible for Junos OS Managed Nodes" on page 24.

The Juniper Networks modules automatically query the default SSH configuration file at ~/.ssh/config, if one exists. You can also include the `ssh_config` parameter to specify a different configuration file.

When using SSH to connect to a Junos device or to a console server connected to the device, the modules first attempt SSH public key-based authentication and then try password-based authentication. When password-based authentication is used, the supplied password is used as the device password. When SSH keys are in use, the supplied password is used as the passphrase for unlocking the private key. If the SSH private key has an empty passphrase, then a password is not required. However, we do not recommend using SSH private keys with empty passphrases.

The following playbook establishes a NETCONF session over SSH with a Junos device and retrieves the device facts. The playbook uses SSH keys in the default location.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

The Juniper Networks Ansible modules also enable you to connect to a Junos device through an SSH connection to a console server. In this case, you must specify the login credentials for both the Junos device and the console server. You use the `user` and `passwd` parameters to specify the Junos OS login credentials, and you use the `cs_user` and `cs_passwd` parameters to specify the console server credentials. When SSH keys are in use, `cs_passwd` is the passphrase for the private key.

The following playbook prompts for the user's credentials for the console server and the Junos device. The module authenticates with the console server and then the Junos device. If authentication is successful, the playbook then retrieves the device facts from the managed node and prints the Junos OS version.

```
---
- name: Get Device Facts
  hosts: dc1_con
  connection: local
  gather_facts: no

  vars_prompt:
```

```
    - name: "CS_USER"
      prompt: "Console server username"
      private: no
    - name: "CS_PASSWORD"
      prompt: "Console server password"
      private: yes
    - name: "JUNOS_USER"
      prompt: "Junos OS username"
      private: no
    - name: "JUNOS_PASSWORD"
      prompt: "Junos OS password"
      private: yes

  vars:
    cs_user: "{{ CS_USER }}"
    cs_passwd: "{{ CS_PASSWORD }}"
    user: "{{ JUNOS_USER }}"
    passwd: "{{ JUNOS_PASSWORD }}"

  tasks:
    - name: "Retrieve facts from Junos devices"
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

The Juniper Networks modules automatically query the default SSH client configuration file at **~/.ssh/config**, if it exists. You can use a different SSH configuration file by including the `ssh_config` parameter and specifying the location of the configuration file. For example:

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  vars:
    ssh_config: "/home/admin/.ssh/config_dc"

  tasks:
    - name: "Retrieve facts from Junos devices"
```

```
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

Connect to a Device Using Telnet

The Juniper Networks modules enable you to connect to a Junos device using telnet, which provides unencrypted access to the network device. You can telnet to the device's management interface or to a console server that is directly connected to the device's **CONSOLE** port. Accessing the device through a console server enables you to initially configure a new or zeroized device that is not yet configured for remote access. To telnet to the management interface, you must configure the Telnet service at the `[edit system services]` hierarchy level on all devices that require access to the interface.

To telnet to the remote device, set the `mode` parameter to `"telnet"` and optionally include the `port` parameter to specify a port. When you set `mode` to `"telnet"` but omit the `port` parameter, the value for `port` defaults to 23. For persistent connections, define `mode` and `port` under the `vars:` section. For local connections, you can define the parameters either under the `vars:` section or as module arguments.

The following playbook telnets to a Junos device using port 7016, retrieves the device facts, and saves them to a file. The playbook prompts for the username and password.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  vars_prompt:
    - name: "DEVICE_USERNAME"
      prompt: "Device username"
      private: no
    - name: "DEVICE_PASSWORD"
      prompt: "Device password"
      private: yes

  vars:
    user: "{{ DEVICE_USERNAME }}"
    passwd: "{{ DEVICE_PASSWORD }}"
```

```
    mode: "telnet"
    port: "7016"

  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

Connect to a Device Using a Serial Console Connection

The Juniper Networks modules enable you to connect to a Junos device using a serial console connection, which is useful when you must initially configure a new or zeroized device that is not yet configured for remote access. To use this connection method, your terminal or laptop must be physically connected to the Junos device through the **CONSOLE** port. For detailed instructions about connecting to the **CONSOLE** port on a Junos device, see the hardware documentation for your specific device.

To connect to a Junos device through a serial console connection, set the module's `mode` parameter to `"serial"`, and optionally include the `port` parameter to specify a port. When you set `mode` to `"serial"` but omit the `port` parameter, the value for `port` defaults to `/dev/ttyUSB0`. For persistent connections, define `mode` and `port` under the `vars:` section. For local connections, you can define the parameters either under the `vars:` section or as module arguments.

The following playbook connects to a Junos device through the **CONSOLE** port and then loads and commits an initial configuration. The playbook prompts for the username and password.

```
---
- name: Load Initial Configuration
  hosts: dc1
  connection: local
  gather_facts: no

  vars_prompt:
    - name: "DEVICE_USERNAME"
      prompt: "Device username"
      private: no
    - name: "DEVICE_PASSWORD"
      prompt: "Device password"
```

```
    private: yes

tasks:
  - name: Load initial configuration and commit
    juniper.device.config:
      user: "{{ DEVICE_USERNAME }}"
      passwd: "{{ DEVICE_PASSWORD }}"
      mode: "serial"
      load: "merge"
      src: "configs/junos.conf"
    register: response
  - name: Print the response
    ansible.builtin.debug:
      var: response
```

RELATED DOCUMENTATION

Understanding the Ansible for Junos OS Collections and Modules | 7

Authenticate Users Executing Ansible Modules on Junos Devices | 37

Authenticate Users Executing Ansible Modules on Junos Devices

IN THIS SECTION

Authentication Overview

Juniper Networks provides Ansible modules that you can use to manage Junos devices. The Juniper Networks modules are distributed through the `juniper.device` collection, which is hosted on Ansible Galaxy.

The Juniper Networks modules enable you to directly connect to and manage Junos devices using SSH, telnet, or a serial console connection. The modules also support connecting to the device through an SSH or telnet connection to a console server that is connected to the device's `CONSOLE` port. The remote device must be able to authenticate the user using a password or other standard SSH authentication mechanisms, depending on the connection protocol.

When you use Ansible to manage Junos devices, the most convenient way to access the device is to configure SSH keys. SSH keys enable the remote device to identify trusted users. Alternatively, you can provide a username and password when you execute modules and playbooks.

For SSH connections, the Juniper Networks modules first attempt SSH public key-based authentication and then try password-based authentication. When SSH keys are in use, the supplied password is used as the passphrase for unlocking the private SSH key. When password-based authentication is used, the supplied password is used as the device password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase, then a password is not required. However, we do not recommend using SSH private keys with empty passphrases. To retrieve a password for password-based authentication or password-protected SSH keys, you can prompt for the password from the playbook or command-line, or you can create a vault-encrypted data file that securely stores the password.

You can specify connection and authentication parameters for the Juniper Networks modules in the following ways. If you do not explicitly define the values, default values are used in some cases, as described in "Understanding the Default Values for Juniper Networks Modules" on page 39. If you define a parameter's value in multiple places, Ansible selects the value based on variable precedence, as outlined in Understanding variable precedence in the official Ansible docs.

- Ansible variables—You can specify the connection and authentication parameter values by using normal Ansible variables, for example, variables defined in inventory or vault files, in host or group variables, or using command-line options.
- SSH client configuration file—For SSH connections, the Juniper Networks modules automatically query the default SSH configuration file at **~/.ssh/config**, if one exists. You can also include the `ssh_config` option to specify a different configuration file. The modules use any relevant settings in the SSH configuration file for the given connection, unless you explicitly define variables that override the setting.
- Module arguments—The `juniper.device` modules support specifying connection and authentication-related options for local connections (`connection: local`) as top-level module arguments.

- `vars:` section—The `juniper.device` modules support specifying connection and authentication-related options for local and persistent connections in a play's `vars:` section, which is described in "How to Define Authentication Parameters in the vars: Section for Local and Persistent Connections" on page 41.

This document discusses the different aspects of authentication when using the Juniper Networks modules to manage Junos devices.

Understanding the Default Values for Juniper Networks Modules

You can explicitly define the connection and authentication parameters for modules that manage Junos devices. If you do not define a parameter, the module uses a default value in some cases. Table 6 on page 39 outlines the default values and variable precedence for common connection parameters for modules in the `juniper.device` collection. For information about the arguments accepted for the individual modules, see the API reference documentation for that module.

Table 6: Default Values and Variable Precedence for Connection Parameters

Parameter Name	Parameter Aliases	Description	Default Value and Variable Precedence
`host`	`hostname` `ip`	Hostname or IP address of the remote device with which the connection should be established.	`{{ inventory_hostname }}`
`passwd`	`password`	The user's password or SSH key passphrase used to authenticate with the managed device.	1. `ANSIBLE_NET_PASSWORD` environment variable 2. Value specified for `-k` or `--ask-pass` command-line option
`ssh_config`	–	Path to an SSH client configuration file. If you omit this parameter, the modules uses the SSH configuration file in the default location, if one exists.	**~/.ssh/config**

Table 6: Default Values and Variable Precedence for Connection Parameters *(Continued)*

Parameter Name	Parameter Aliases	Description	Default Value and Variable Precedence
ssh_private_key_file	ssh_keyfile	Path to the SSH private key file used to authenticate with the remote device. If you do not explicitly specify the path and no default value is found, then the module uses the SSH private key file specified in the user's SSH configuration or the operating-system-specific default.	1. ANSIBLE_NET_SSH_KEYFILE environment variable 2. Value specified for --private-key or --key-file command-line option 3. none
user	username	Username that is used to authenticate with the managed node.	1. ANSIBLE_NET_USERNAME environment variable 2. remote_user as defined by Ansible 3. USER environment variable

When executing Juniper Networks modules, the `host` argument is always required for a connection. However, you do not have to explicitly specify the host, because it defaults to `{{ inventory_hostname }}`.

You can execute Juniper Networks modules using any user account that has access to the managed Junos device. When you execute the modules, Junos OS user account access privileges are enforced, and the class configured for the Junos OS user account determines the permissions. If you do not specify a user, the user is set according to the algorithm described for `user` in Table 6 on page 39. See the Ansible documentation for the precedence used to define `remote_user`, which can be defined in a number of ways, including:

- `-u` or `--user` command line option
- `ANSIBLE_REMOTE_USER` environment variable
- `remote_user` configuration setting

How to Define Authentication Parameters in the vars: Section for Local and Persistent Connections

You can define connection and authentication parameters for the `juniper.device` modules in the play's `vars:` section. This is in addition to defining them as you normally would through other variables, for example, in the SSH configuration file, in the Ansible inventory file, as command-line arguments, or as module arguments. The `vars:` section enables you to define common connection parameters in a single location that all modules in the play can use to connect to a host. Additionally, certain Ansible connections require using the `vars:` section when you define the parameters within the play, as described here.

The `juniper.device` modules support the following Ansible connections types:

- local connections, which are defined by using `connection: local`
- persistent connections, which are defined by using `connection: juniper.device.pyez`

For both local and persistent connections, Ansible executes modules locally on the control node. When you use `connection: local`, Ansible establishes a separate connection to the host for each task in the play that requires a connection. By contrast, when you use `connection: juniper.device.pyez`, Ansible establishes a single, persistent connection to a host, which persists over the execution of all tasks in the play.

You use the same connection and authentication parameters for persistent connections as you do for local connections, and the default parameter values discussed in "Understanding the Default Values for Juniper Networks Modules" on page 39 apply to both types of connections. For local connections, you can define the connection and authentication parameters either in the `vars:` section or as module arguments. If you define the parameters in both places, the module arguments take precedence. However, when you define parameters within a play for persistent connections, you must define the parameters in the `vars:` section. With persistent connections there is only a single connection for all tasks, and thus you must define the parameters globally so they apply to all tasks in that play.

The following playbook executes two `juniper.device` modules on each host in the inventory group. The play defines the Ansible connection as `juniper.device.pyez`, which establishes a connection to each host that persists over the execution of all tasks in the play. The authentication parameters for the persistent connection are defined within the play's `vars:` section. The `user` and `passwd` values reference variables defined in the `vault-vars.yaml` vault file.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: juniper.device.pyez
  gather_facts: no
```

```
vars:
  user: "{{ admin_username }}"
  passwd: "{{ admin_password }}"

vars_files:
  - vault-vars.yaml

tasks:
  - name: Retrieve facts from Junos devices
    juniper.device.facts:
      savedir: "{{ playbook_dir }}"

  - name: Get hardware inventory
    juniper.device.command:
      commands: "show chassis hardware"
      dest_dir: "{{ playbook_dir }}"
```

How to Authenticate the User Using SSH Keys

IN THIS SECTION

The Juniper Networks `juniper.device` modules enable you to use SSH keys to connect to a Junos device or to a console server that is connected to the device. To authenticate a user using SSH keys, first generate the keys on the Ansible control node and then configure the keys on the device to which the module will connect, either the managed Junos device or the console server connected to the Junos device.

Generate and Configure the SSH Keys

To generate SSH keys on the Ansible control node and configure the public key on the remote device:

1. On the Ansible control node, generate the public and private SSH key pair for the desired user, and provide any required options, for example:

```
user@localhost:~$ cd ~/.ssh
user@localhost:~/.ssh$ ssh-keygen -t rsa -b 4096
Enter file in which to save the key (/home/user/.ssh/id_rsa): id_rsa_dc
Enter passphrase (empty for no passphrase):  *****
Enter same passphrase again: *****
```

2. (Optional) Load the key into the native SSH key agent. For example:

```
user@localhost:~/.ssh$ ssh-add ~/.ssh/id_rsa_dc
```

3. Configure the public key on each device to which the modules will connect, which could include Junos devices or a console server connected to a Junos device.

   The easiest method to configure the public key on a Junos device is to load a file that contains the public key under the appropriate user account.

```
[edit]
user@router# set system login user username authentication load-key-file URL
user@router# commit
```

4. Verify that the key works by logging in to the device using the key.

```
user@localhost:~$ ssh -i ~/.ssh/id_rsa_dc router.example.com
Enter passphrase for key '/home/user/.ssh/id_rsa_dc':
user@router>
```

Use SSH Keys in Ansible Playbooks

After generating the SSH key pair and configuring the public key on the remote device, you can use the key to connect to the device. The Juniper Networks modules automatically query the default SSH configuration file at **~/.ssh/config**, if one exists. You can also define the `ssh_config` option to specify a different configuration file. The modules use any relevant settings in the SSH configuration file for the given connection, unless you explicitly define variables that override the setting. In addition, the modules automatically look for keys in the default location and keys that are actively loaded in an SSH key agent.

To define specific settings for SSH keys, you can include the appropriate arguments in your Ansible playbook. Define the arguments in the location appropriate for your set of modules and Ansible

connection, for example, in the `vars:` section for plays that use the `juniper.device` modules with a persistent connection. The arguments to include are determined by the location of the key, whether the key is actively loaded into an SSH key agent, whether the key is password-protected, and whether the user's SSH configuration file already defines settings for that host.

- To connect to a Junos device using SSH keys that are actively loaded into the native SSH key agent or that are in the default location and do not have password protection, you do not need to define any connection or authentication-related arguments, unless they differ from the default.

```
  juniper.device.facts:
    savedir: "{{ playbook_dir }}"
```

- To connect to a Junos device using SSH keys that are not in the default location and do not have password protection, set the `ssh_private_key_file` argument to the path of the SSH private key file. For example:

```
vars:
  ssh_private_key_file: "/home/user/.ssh/id_rsa_alternate"

tasks:
  - name: Retrieve facts from Junos devices
    juniper.device.facts:
      savedir: "{{ playbook_dir }}"
```

  Alternatively, you can specify the path of the SSH private key by defining it in the SSH configuration file; by setting the `ANSIBLE_NET_SSH_KEYFILE` environment variable; or by defining the `--private-key` or `--key-file` command-line option when you execute the playbook.

- To connect to a Junos device using a password-protected SSH key file, which is the recommended method, you can reference the SSH key file passphrase in the `passwd` argument or provide the password by using normal Ansible variables or command-line options.

  It is the user's responsibility to obtain the SSH key file passphrase in a secure manner appropriate for their environment. It is best practice to either prompt for it during each invocation of the playbook or store the variables using an encrypted vault rather than storing the credentials in an unencrypted format. For example, you can execute the playbook with the `--ask-pass` command-line option and provide the SSH key file passphrase when prompted, as shown here:

```
vars:
  ssh_private_key_file: "/home/user/.ssh/id_rsa_dc"
```

```
  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

```
user@localhost:~$ ansible-playbook playbook.yaml --ask-pass
SSH password:
```

For more information about using a prompt or encrypted vault file for the SSH key passphrase, see "How to Authenticate the User Using a Playbook or Command-Line Password Prompt" on page 45 and "How to Authenticate the User Using an Ansible Vault-Encrypted File" on page 47.

For instructions on using SSH keys to connect to a console server, see "How to Authenticate Through a Console Server" on page 49.

How to Authenticate the User Using a Playbook or Command-Line Password Prompt

To authenticate a user executing Ansible modules, you can prompt for the user's credentials when you execute the playbook. For example, you can define an interactive prompt in the playbook, or you can execute the playbook with the `-k` or `--ask-pass` command-line option to prompt for the password. When SSH keys are in use, the supplied password is used as the passphrase for unlocking the private SSH key. When password-based authentication is used, the supplied password is used as the device password.

To define an interactive prompt in the playbook to obtain the user's password or SSH key passphrase:

1. Include code under `vars_prompt:` that prompts for the user's password or SSH key passphrase (and optionally the username) and stores the value in a variable.

```
---
- name: Get Device Facts
  hosts: all
  connection: local
  gather_facts: no

  vars_prompt:
    - name: "USERNAME"
      prompt: "Username"
      private: no
```

```
    - name: "DEVICE_PASSWORD"
      prompt: "Device password"
      private: yes
```

2. Set the `user` and `passwd` parameters so each references its respective variable.

```
  vars:
    user: "{{ USERNAME }}"
    passwd: "{{ DEVICE_PASSWORD }}"
```

3. Include the tasks to execute.

```
  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print facts
      ansible.builtin.debug:
        var: junos.version
```

4. Execute the playbook, which prompts for the username and password and does not echo the password on the command line because the variable is set to `private: yes`.

```
user@localhost:~$ ansible-playbook playbook.yaml
User name: user
Device password:
```

Alternatively, you can execute a playbook with the `-k` or `--ask-pass` command-line option to prompt for the password or passphrase. Consider the following playbook, which uses the default username:

```
---
- name: Get Device Facts
  hosts: all
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

```
    - name: Print facts
      ansible.builtin.debug:
        var: junos.version
```

Execute the playbook, and include the `-k` or `--ask-pass` command-line option, which prompts for the password and does not echo the password on the command line.

```
user@localhost:~$ ansible-playbook playbook.yaml --ask-pass
SSH password:

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

How to Authenticate the User Using an Ansible Vault-Encrypted File

You can create an Ansible vault that securely stores saved passwords and other sensitive connection and authentication values in an vault-encrypted data file. Your playbook can then reference those variables in the location appropriate for your set of modules and Ansible connection type, for example, in the play's `vars:` section or as module arguments.

To create and use an Ansible vault file containing required variables, including passwords:

1. Create a vault-encrypted data file, and specify the password required to encrypt, decrypt, edit, and use the data file.

   ```
   root@localhost:~# ansible-vault create vault-vars.yaml
   Vault password:
   Confirm Vault password:
   ```

2. Define the required variables in the file and save it.

   ```
   root@localhost:~# ansible-vault edit vault-vars.yaml
   Vault password:

   # Vault variables
   root_username: root
   root_password: password
   ```

3. Verify that the file is encrypted.

```
root@localhost:~# cat vault-vars.yaml
$ANSIBLE_VAULT;1.1;AES256
31415961343966623035373532313264333633663764353763393066643131306565636463326634
37303261656665653566653431373131612345693363366640a65393963333166393537636237666 6
65653737653262363235353261626135312345663665396262376339623737366238653436306663
643037663330633936 0a343065363331313532633036343866376330623634653538353132314159
3835
```

4. In the playbook, include the vault-encrypted variable file, and reference the required variables in the location appropriate for your modules and Ansible connection type.

```
---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no

  vars_files:
    - vault-vars.yaml

  vars:
    user: "{{ root_username }}"
    passwd: "{{ root_password }}"

  tasks:
    - name: Retrieve facts from Junos devices
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      ansible.builtin.debug:
        var: junos.version
```

NOTE: If you instead define the actual `user` and `passwd` variables in the vault, the modules pick them up automatically, and you do not need to explicitly define them in the playbook.

5. Execute the playbook with the `--ask-vault-pass` option, which prompts for the vault password.

```
root@localhost:~# ansible-playbook playbook-name.yaml --ask-vault-pass
Vault password:

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

How to Authenticate Through a Console Server

The Juniper Networks Ansible modules can connect to Junos devices through a console server. For SSH connections through a console server, you need to provide the authentication credentials for both the console server and the Junos device. You can use either a device password or a password-protected SSH key file for the console server authentication.

To connect to a Junos device through a console server, you must provide the following parameters in your playbook, if there is no default value or the default value is not appropriate:

- `host`—Console server hostname or IP address
- `user` and `passwd`—Junos OS login credentials
- `cs_user`—Console server username
- `cs_passwd`—Device password or SSH key file passphrase required to authenticate with the console server

In the following example, the credentials for the Junos OS user and the console server user are defined in an Ansible vault file. The vault variables are then referenced in the playbook. In this case, the `cs_passwd` argument is the passphrase for the SSH key specified in the `ssh_private_key_file` argument.

```
---
- name: Get Device Facts
  hosts: dc1_con
  connection: local
  gather_facts: no

  vars_files:
    - vault-vars.yaml
```

```
vars:
  host: "{{ inventory_hostname }}"
  user: "{{ junos_username }}"
  passwd: "{{ junos_password }}"
  cs_user: "{{ cs_username }}"
  cs_passwd: "{{ cs_key_password }}"
  ssh_private_key_file: "/home/user/.ssh/id_rsa_dc"

tasks:
  - name: Retrieve facts from Junos devices
    juniper.device.facts:
      savedir: "{{ playbook_dir }}"
```

RELATED DOCUMENTATION

Troubleshoot Ansible Authentication Errors When Managing Junos Devices | 203

Connect to Junos Devices Using Ansible | 29

Understanding Ansible for Junos OS | 4

Understanding the Ansible for Junos OS Collections and Modules | 7

5
CHAPTER

Use Ansible to Manage Device Operations

Use Ansible to Retrieve Facts from Junos Devices

Juniper Networks provides Ansible modules that you can use to manage Junos devices and perform operational and configuration tasks on the devices. The modules do not require Python on the managed device because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, when you use Ansible to perform operations on Junos devices, you must execute the Ansible modules locally on the control node. As a result, Ansible defaults to gathering facts from the Ansible control node instead of the managed node.

Juniper Networks provides a module that you can use to gather device facts, including the active configuration, from Junos devices. Table 7 on page 52 outlines the available module. The module uses the Junos PyEZ fact gathering system to retrieve the device facts. For more information about the Junos PyEZ fact gathering system and the complete list of returned dictionary keys, see jnpr.junos.facts.

Table 7: Module to Gather Facts

Content Set	Module Name
`juniper.device` collection	`facts`

The `facts` module returns the device facts in the `ansible_facts.junos` dictionary. The module also enables you to save the returned data in a file on the local Ansible control node. To specify the directory in which to save the retrieved information, include the `savedir` module argument, and define the path to the target directory. When you include the `savedir` argument, the playbook generates the following files for each device, where *hostname* is the value of the `hostname` fact retrieved from the device, which might be different from the hostname passed to the module:

- ***hostname*-facts.json**—Device facts in JSON format
- ***hostname*-inventory.xml**—Device's hardware inventory in XML format

For example, the following playbook retrieves the device facts for each device in the inventory group and saves the data for each device in separate files in the playbook directory on the Ansible control node. Because the playbook runs the `facts` module locally, Ansible defaults to gathering facts from the control node. The playbook includes the `gather_facts: no` argument to prevent Ansible from gathering facts from the control node and instead uses the `juniper.device.facts` module to retrieve the facts from the managed device. To authenticate with the device, the example uses existing SSH keys in the default location and thus does not explicitly provide credentials for the `facts` module in the playbook.

```
---
- name: Get device facts
```

```
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve device facts and save to file
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

By default, the `facts` module does not return the device configuration. To return the active configuration for a device, in addition to the device facts, include the `config_format` option, and specify the format in which to return the configuration. Acceptable format values are `'json'`, `'set'`, `'text'` and `'xml'`. The requested format must be supported by the Junos OS release running on the device.

When you include the `config_format` option, the `ansible_facts.junos` dictionary in the module response includes the `config` key with the configuration in the specified format in a single multi-line string. However, even if the playbook includes the `savedir` option, the configuration data is *not* written to a file.

TIP: To use Ansible to retrieve configuration data from a Junos device and save the data to a file, use the `config` module instead of the `facts` module. For more information, see "Use Ansible to Retrieve or Compare Junos OS Configurations" on page 148.

The playbook in the next example performs the following operations:

- Retrieves the device facts and active configuration for each device in the inventory group
- Saves the facts and hardware inventory for each device in separate files in the playbook directory on the Ansible control node
- Prints the configuration for each device to standard output

```
---
- name: Get device facts and configuration
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve device facts and configuration and save facts to file
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

```
        config_format: "xml"
      register: result

    - name: Print configuration
      ansible.builtin.debug:
        var: result.ansible_facts.junos.config
```

```
user@ansible-cn:~$ ansible-playbook facts.yaml
PLAY [Get device facts and configurations] *********************************

TASK [Retrieve device facts and configuration and save facts to file] *******
ok: [dc1a.example.net]

TASK [Print configuration] *************************************************
ok: [dc1a.example.net] => {
    "result.ansible_facts.junos.config": "<configuration commit-seconds=\"1605564153\" commit-
localtime=\"2020-11-16 14:02:33 PST\" commit-user=\"admin\">\n
<version>20191212.201431_builder.r1074901</version>\n
[...output truncated...]
</configuration>\n"
}

PLAY RECAP *****************************************************************
dc1a.example.net  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

RELATED DOCUMENTATION

- Understanding the Ansible for Junos OS Collections and Modules | 7
- Use Ansible to Execute Commands and RPCs on Junos Devices | 55
- Use Ansible with Junos PyEZ Tables to Retrieve Operational Information from Junos Devices | 69

Use Ansible to Execute Commands and RPCs on Junos Devices

SUMMARY

Use the Juniper Networks Ansible modules to execute operational mode commands and RPCs on Junos devices.

IN THIS SECTION

Juniper Networks provides Ansible modules that you can use to execute operational mode commands and remote procedure calls (RPCs) on Junos devices. Table 8 on page 55 outlines the modules.

Table 8: Command and RPC Modules

Content Set	Module Name
`juniper.device` collection	`command` `rpc`

The following sections discuss how to the use the modules, parse the module response, specify the output format, and save the output to a file.

NOTE: To more easily extract targeted data from operational output, you can also use the `juniper.device.table` module with custom or predefined Junos PyEZ operational tables. For more information, see "Use Ansible with Junos PyEZ Tables to Retrieve Operational Information from Junos Devices" on page 69.

How to Execute Commands with the Juniper Networks Modules

The `juniper.device.command` module enables you to execute operational mode commands on Junos devices. The module requires one argument, `commands`, which is a list of one or more Junos OS operational mode commands to execute on the device.

The following playbook executes two operational mode commands on each device in the inventory group and displays the module response in standard output. In this example, the `command` module authenticates with the device by using SSH keys in the default location.

```
---
- name: Get device information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and uptime information
      juniper.device.command:
        commands:
          - "show version"
          - "show system uptime"
      register: junos_result

    - name: Print response
      ansible.builtin.debug:
        var: junos_result
```

For information about the module's response and output format, see "Understanding the Module Response" on page 58 and "How to Specify the Format for the Command or RPC Output" on page 60.

How to Execute RPCs with the Juniper Networks Modules

The Junos XML API is an XML representation of Junos OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the operational mode commands that you issue in the Junos OS CLI. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element. Request tags are used in remote procedure calls (RPCs) within NETCONF or Junos XML protocol sessions to request information from a Junos device. The server returns the response using Junos XML elements enclosed within the response tag element.

The juniper.device.rpc module enables you to execute RPCs on Junos devices. The modules require one argument, rpcs, which is a list of one or more Junos OS RPCs to execute on the device.

The following playbook executes the get-interface-information RPC on each device in the inventory group and displays the module response in standard output. , The RPC is equivalent to the show interfaces operational mode command. In this example, the rpc module authenticates with the device by using SSH keys in the default location.

```
---
- name: Execute RPC
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get interface information
      juniper.device.rpc:
        rpcs: "get-interface-information"
      register: junos_result

    - name: Print response
      ansible.builtin.debug:
        var: junos_result
```

NOTE: For information about mapping CLI commands to RPC request tags, see the Junos XML API Explorer for operational tags.

For information about the module's response and output format, see "Understanding the Module Response" on page 58 and "How to Specify the Format for the Command or RPC Output" on page 60.

The juniper.device.rpc module supports the kwargs option, which enables you to specify keyword arguments and values for the RPCs. The value of kwargs can be:

- A single dictionary of keywords and values
- A list of dictionaries that supply arguments for multiple RPCs

There must be a one-to-one correspondence between the items in the kwargs list and the RPCs in the rpcs list. If you execute multiple RPCs, and an RPC does not require any arguments, set the corresponding kwargs list item equal to an empty dictionary {}. If an individual RPC argument does not require a value, set its value equal to true.

NOTE: You must use underscores in RPC arguments in place of hyphens, which can cause exceptions or errors in certain circumstances.

The following playbook executes the specified RPCs on each device in the inventory group and displays the module response in standard output. The `get-interface-information` RPC requests terse level output for the lo0.0 interface and the `get-lldp-interface-neighbors` RPC requests information for the ge-0/0/0 interface. The `get-software-information` RPC uses an empty dictionary to execute the RPC with no additional arguments.

```
---
- name: Get Device Information
  hosts: dc1a
  connection: local
  gather_facts: no

  tasks:
    - name: Get device information
      juniper.device.rpc:
        rpcs:
          - "get-interface-information"
          - "get-lldp-interface-neighbors"
          - "get-software-information"
        kwargs:
          - interface_name: "lo0.0"
            terse: true
          - interface_device: "ge-0/0/0"
          - {}
      register: junos_result

    - name: Print response
      ansible.builtin.debug:
        var: junos_result
```

Understanding the Module Response

The `juniper.device.command` and `juniper.device.rpc` modules store the RPC reply from the device within several different keys in the module response. The data for each key is structured as follows:

- `stdout`—RPC reply is a single multi-line string.
- `stdout_lines`—RPC reply is a list of single line strings.
- `parsed_output`—RPC reply is parsed into a JavaScript Object Notation (JSON) data structure. This key is only returned when the format of the data is XML or JSON.

If the module executes a single command or RPC, the module's response places the returned keys at the top level. If the module executes multiple commands or RPCs, the module's response instead includes a `results` key, which is a list of dictionaries. Each element in the list corresponds to a single command or RPC and includes all the keys that would be returned for that command or RPC.

For example, the following response corresponds to executing a single RPC:

```
    "junos_result": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "parsed_output": {
            "interface-information": {
                "physical-interface": [
                [...output omitted...]
            }
        },
        "rpc": "get-interface-information",
        "stdout": "<interface-information style=\"normal\">\n  <physical-interface>\n
            [...output omitted...]  </physical-interface>\n</interface-information>\n"
        "stdout_lines": [
            "<interface-information style=\"normal\">",
            "  <physical-interface>",
                [...output omitted...]
            "  </physical-interface>",
            "</interface-information>"
        ]
    }
```

In some instances, command or RPC output can be extensive, and it might be necessary to suppress the output in the module's response. To omit the output keys in the module's response, include `return_output: false` in that module's argument list.

```
tasks:
  - name: Get interface information
    juniper.device.rpc:
      rpcs: "get-interface-information"
      return_output: false
    register: junos_result
```

How to Specify the Format for the Command or RPC Output

The `command` and `rpc` modules store the RPC reply from the device within several different keys in the module response: `stdout`, `stdout_lines`, and `parsed_output`. The `parsed_output` key, which is only present when the command or RPC output format is XML or JSON, contains data that is parsed into a JSON data structure.

The `stdout` and `stdout_lines` keys contain data in the default format defined for the module. By default, the `command` module returns the command output in text format, and the `rpc` module returns the RPC output in XML format.

To specify a different output format, include the `formats` argument, and set the value equal to the desired format. Supported formats include:

- `json`
- `text`
- `xml`

The `formats` parameter takes either a string or a list of strings. When you execute multiple commands or RPCs and specify only a single format, the output format is the same for all executed commands and RPCs. To specify a different format for the output of each command or RPC, set the `formats` argument to a list of the desired formats. The list must specify the same number of formats as there are commands or RPCs.

The following playbook executes two RPCs on each device in the inventory group and requests text format for the output of all executed RPCs:

```
---
- name: Get device information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and system uptime information
      juniper.device.rpc:
        rpcs:
          - "get-software-information"
          - "get-system-uptime-information"
        formats: "text"
      register: junos_result

    - name: Print response
      ansible.builtin.debug:
        var: junos_result
```

When the playbook is executed, the `stdout` and `stdout_lines` keys in the module response contain the RPC reply in text format.

The following playbook executes two RPCs on each device in the inventory group and requests the output for the first RPC in text format and the output for the second RPC in JSON format:

```
---
- name: Get device information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and system uptime information
      juniper.device.rpc:
        rpcs:
          - "get-software-information"
          - "get-system-uptime-information"
        formats:
          - "text"
```

```
          - "json"
      register: junos_result

    - name: Print response
      ansible.builtin.debug:
        var: junos_result
```

How to Save the Command or RPC Output to a File

When you use the `juniper.device.command` and `juniper.device.rpc` modules to execute a command or RPC on a device, you can save the returned data to a file on the local Ansible control node by including the `dest` or `dest_dir` module arguments. Whereas the `dest_dir` option saves the output for each command or RPC in separate files for a device, the `dest` option saves the output for all commands and RPCs in the same file for a device. If an output file already exists with the target name, the module overwrites the file.

If you are saving the data to a file and do not want to duplicate the command or RPC output in the module's response, you can optionally include `return_output: false` in the module's argument list. Setting `return_output` to `false` causes the module to omit the output keys in the module's response. Doing this might be necessary if the device returns a significant amount of data.

The following sections outline how to use the `dest_dir` and `dest` options.

dest_dir

To specify the directory on the local Ansible control node where the retrieved data is saved, include the `dest_dir` argument, and define the path to the target directory. The module stores the output for each command or RPC executed on a device in a separate file named ***hostname_name.format*** where:

- ***hostname***—Hostname of the device on which the command or RPC is executed.
- ***name***—Name of the command or RPC executed on the managed device. The module replaces spaces in the command name with underscores (_).
- ***format***—Format of the output, which can be **json**, **text**, or **xml**.

The following playbook executes two RPCs on each device in the inventory group and saves the output for each RPC for each device in a separate file in the playbook directory on the Ansible control node:

```
---
- name: Get device information
  hosts: dc1
```

```
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and uptime information
      juniper.device.rpc:
        rpcs:
          - "get-software-information"
          - "get-system-uptime-information"
        dest_dir: "{{ playbook_dir }}"
```

The resulting output files for host dc1a.example.net are:

- **dc1a.example.net_get-software-information.xml**
- **dc1a.example.net_get-system-uptime-information.xml**

Similarly, the following playbook executes the equivalent commands on each device in the inventory group and saves the output for each command for each device in a separate file in the playbook directory on the Ansible control node:

```
---
- name: Get device information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and uptime information
      juniper.device.command:
        commands:
          - "show version"
          - "show system uptime"
        dest_dir: "{{ playbook_dir }}"
```

The resulting output files for host dc1a.example.net are:

- **dc1a.example.net_show_version.text**
- **dc1a.example.net_show_system_uptime.text**

dest

To specify the path and filename to which all command or RPC output for a target node is saved on the local Ansible control node, include the `dest` argument, and define the filename or the full path of the file. If you include the `dest` argument, but omit the directory, the files are saved in the playbook directory. If you execute commands or RPCs on multiple devices, the `dest` argument must include a variable such as `{{ inventory_hostname }}` to differentiate the filename for each device. If you do not differentiate the filenames, the output file for each device will overwrite the output file of the other devices.

The following playbook executes RPCs on each device in the inventory group. The output for all RPCs is stored in a separate file for each device, and the file is placed in the playbook directory on the Ansible control node. Each file is uniquely identified by the device hostname.

```
---
- name: Get device information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get software and uptime information
      juniper.device.rpc:
        rpcs:
          - "get-software-information"
          - "get-system-uptime-information"
        dest: "{{ inventory_hostname }}-system-information.xml"
```

For example, the resulting output file for host dc1a.example.net is **dc1a.example.net-system-information.xml** and contains the output for all RPCs executed on the device.

RELATED DOCUMENTATION

Use Ansible with Junos PyEZ Tables to Retrieve Operational Information from Junos Devices | 69

Use Ansible to Retrieve Facts from Junos Devices | 52

Understanding the Ansible for Junos OS Collections and Modules | 7

Use Ansible to Transfer Files to or from Junos Devices

SUMMARY

Use the Juniper Networks Ansible modules to copy files between the Ansible control node and Junos devices.

IN THIS SECTION

file_copy Module Overview

You can use the `juniper.device.file_copy` Ansible module to transfer files between the Ansible control node and Junos devices. Table 9 on page 65 outlines the module arguments. You must include the `action` argument to specify the direction of transfer. You must also specify the local and remote directories as well as the filename of the file to transfer.

Table 9: file_copy Arguments

Module argument	Description	Default
`action`	Specify whether to copy a file to or from the remote device. Supported values: • `get` • `put`	—
`file`	Filename of the file to copy.	—
`local_dir`	Directory on the local Ansible control node.	—
`remote_dir`	Directory on the remote device.	—

The `file_copy` module reports a successful transfer if the module copies the file to the destination directory and the checksum of the copied file matches the checksum of the original file. In some cases, the transfer is successful but the module reports that the task failed because the checksums do not match. This might happen if the file is corrupted during transfer. It can also happen if you transfer a large log file that is updated frequently. In this case, if the file is updated as it is being transferred, the transferred file can differ slightly from the original file causing a checksum mismatch.

Transfer Files from the Remote Device

You can use the `juniper.device.file_copy` module to copy a file from a Junos device to the Ansible control node. For example, you might want to periodically archive the configuration file or a log file on a device. To transfer a file from the remote device, specify `action: get`.

The following playbook transfers the **messages** log file from each device in the inventory group into a host-specific directory on the Ansible control node.

```
---
- name: Archive the messages log file
  hosts: junos
  connection: local
  gather_facts: false

  vars:
    host_log_dir: "logs/{{ inventory_hostname }}"

  tasks:
    - name: Create the log directory for the host
      ansible.builtin.file:
        path: "{{ host_log_dir }}"
        state: directory

    - name: Copy the log file from remote device
      juniper.device.file_copy:
        action: get
        file: messages
        local_dir: "{{ host_log_dir }}"
        remote_dir: /var/log
```

When you execute the playbook, it creates a **logs/*hostname*** directory for each host in the inventory group. The playbook then copies the **messages** log file from each remote host to the respective

destination directory for that host. Although the copy task appears to fail for all hosts, the file transfer is actually successful. The Junos device constantly updates the **messages** log file. So, in this case, the Junos device is updating the log file as the transfer occurs. As a result, the checksum comparison between the local file and the remote file fails because the original and copied files differ slightly.

```
user@ansible-cn:~$ ansible-playbook ansible-pb-archive-logs.yaml

PLAY [Archive the messages log file] ********************************************

TASK [Create the log directory for the host] ************************************
ok: [r3]
ok: [r2]
ok: [r1]

TASK [Copy the log file from remote device] *************************************
fatal: [r2]: FAILED! => {"changed": false, "msg": "Transfer failed (different MD5 between local
and remote) 0b36d9e93cfd523b79eee5927ed42b68 | 3ea1421a5f68476c853180213df96686"}
fatal: [r3]: FAILED! => {"changed": false, "msg": "Transfer failed (different MD5 between local
and remote) 7bac31566e5ec8da16d2a199dda628a6 | 40acb378a5e2b7aeacda3eb0337b5bec"}
fatal: [r1]: FAILED! => {"changed": false, "msg": "Transfer failed (different MD5 between local
and remote) 7566d1efa73b50081dfee04aa4dbde57 | 2fce0d8a7272fe7e528786dade8cbe1f"}

PLAY RECAP **********************************************************************
r1                         : ok=1    changed=0    unreachable=0    failed=1    skipped=0
rescued=0    ignored=0
r2                         : ok=1    changed=0    unreachable=0    failed=1    skipped=0
rescued=0    ignored=0
r3                         : ok=1    changed=0    unreachable=0    failed=1    skipped=0
rescued=0    ignored=0
```

When you review the **logs** directory, the **messages** log is archived for each host.

```
user@ansible-cn:~$ ls -l logs/*
logs/r1:
total 452
-rw-rw-r-- 1 user admin 459462 Sep 10 21:10 messages

logs/r2:
total 368
-rw-rw-r-- 1 user admin 373848 Sep 10 21:10 messages
```

```
logs/r3:
total 368
-rw-rw-r-- 1 user admin 374433 Sep 10 21:10 messages
```

Transfer Files to the Remote Device

You can use the `juniper.device.file_copy` module to copy a file from the Ansible control node to a Junos device. To transfer the file to the remote device, specify `action: put`.

The following playbook copies the **bgp.slax** script from the Ansible control node to each host in the specified inventory group. The script is copied from the **scripts** directory within the playbook directory to the **/var/db/scripts/op** directory on the Junos device.

```
---
- name: Copy script to the Junos device
  hosts: junos
  connection: local
  gather_facts: false

  tasks:
    - name: Copy a local script to the Junos device
      juniper.device.file_copy:
        action: put
        file: bgp.slax
        local_dir: scripts
        remote_dir: /var/db/scripts/op
```

Use Ansible with Junos PyEZ Tables to Retrieve Operational Information from Junos Devices

SUMMARY

Use Junos PyEZ Tables and Views in your Ansible playbooks to retrieve operational information from Junos devices.

IN THIS SECTION

Module Overview

Junos PyEZ operational (op) Tables provide a simple and efficient way to extract information from complex operational command output. Juniper Networks provides an Ansible module that enables you to leverage Junos PyEZ op Tables from within Ansible playbooks. Table 10 on page 69 outlines the module.

Table 10: Junos PyEZ Table Module

Content Set	Module Name
`juniper.device` collection	`table`

NOTE: The `table` module does not support using configuration Tables and Views.

Understanding Junos PyEZ Tables

Junos PyEZ is a microframework for Python that enables you to manage and automate Junos devices. Junos PyEZ supports using simple YAML definitions, which are referred to as *Tables* and *Views*, to retrieve and filter operational command output and configuration data from Junos devices.

Junos PyEZ operational (op) Tables extract information from the output of operational commands or RPCs. The Junos PyEZ `jnpr.junos.op` modules contain predefined Table and View definitions for some common RPCs. You can also create custom Tables and Views.

When you use Ansible to manage Junos devices, the `table` module can use Junos PyEZ Tables to retrieve data from a device. The module can reference the predefined operational Tables and Views that are included with the Junos PyEZ distribution, or it can reference user-defined Tables and Views that reside on the Ansible control node.

For general information about Junos PyEZ Tables and Views, see the following sections and related documentation in the Junos PyEZ Developer Guide:

- Understanding Junos PyEZ Tables and Views
- Predefined Junos PyEZ Operational Tables (Structured Output)

How to Use the Juniper Networks Ansible Modules with Junos PyEZ Tables

The `juniper.device.table` module can include the following arguments to specify the Table to use:

- `file`—Filename of the YAML file that defines the Junos PyEZ Table and View.
- `path`—(Optional) Path to the directory containing the YAML file with the Table and View definitions. The default file path is the location of the predefined Junos PyEZ op Tables, which reside in the Junos PyEZ install path under the **jnpr/junos/op** directory.
- `table`—(Optional) Name of the Table that will be used to retrieve the data. This option is only required when a file contains multiple Table definitions or the file contains a single Table that does not include "Table" in its name.

For example, the following task retrieves data by using a custom table named `FPCTable`, which is defined in the **fpc.yaml** file located in the playbook directory:

```
tasks:
  - name: Get FPC info
```

```
      juniper.device.table:
        file: "fpc.yaml"
        path: "{{ playbook_dir }}"
        table: "FPCTable"
```

The module's response includes the `resource` key, which contains a list of items returned by the Table. Each list item is a dictionary containing the field names defined by the View and the value extracted from the data for each of the corresponding fields.

Consider the following predefined Table and View, `ArpTable` and `ArpView`, in the **arp.yml** file of the Junos PyEZ distribution. `ArpTable` executes the `<get-arp-table-information>` RPC with the `<no-resolve/>` option, which is equivalent to the `show arp no-resolve` CLI command. The corresponding View extracts the MAC address, IP address, and interface name for each `<arp-table-entry>` item in the response.

```
---
ArpTable:
  rpc: get-arp-table-information
  args:
     no-resolve: true
  item: arp-table-entry
  key: mac-address
  view: ArpView

ArpView:
  fields:
    mac_address: mac-address
    ip_address: ip-address
    interface_name: interface-name
```

The following Ansible playbook executes the `table` module, which uses `ArpTable` to retrieve Address Resolution Protocol (ARP) information from Junos devices. Because `ArpTable` is included with the Junos PyEZ distribution and resides in the default directory for the predefined Junos PyEZ op Tables, the `path` module argument is not required to specify the file location. In addition, because `ArpTable` is the only Table defined in the file and includes "Table" in its name, the `table` argument is not required to specify the Table.

```
---
- name: Get ARP information
  hosts: dc1
  connection: local
  gather_facts: no
```

```
  tasks:
    - name: Get ARP information using Junos PyEZ Table
      juniper.device.table:
        file: "arp.yml"
      register: result

    - name: Print response
      ansible.builtin.debug:
        var: result
```

The playbook output, which is truncated for brevity, includes the corresponding fields, as defined by ArpView, for each <arp-table-entry> item returned by the device.

```
PLAY [Get ARP information] ******************************************************

TASK [Get ARP information using Junos PyEZ Table] ******************************
ok: [dc1a.example.net]

TASK [Print response] ***********************************************************
ok: [dc1a.example.net] => {
    "result": {
        "changed": false,
        "failed": false,
        "msg": "Successfully retrieved 2 items from ArpTable.",
        "resource": [
            {
                "interface_name": "em0.0",
                "ip_address": "10.0.0.5",
                "mac_address": "02:01:00:00:00:05"
            },
            {
                "interface_name": "fxp0.0",
                "ip_address": "198.51.100.10",
                "mac_address": "30:7c:5e:48:4b:40"
            }
        ]
    }
}
```

The following Ansible playbook leverages the predefined Junos PyEZ operational Table, OspfInterfaceTable, to retrieve information about OSPF interfaces on Junos devices. The **ospf.yml** file

defines multiple Tables and Views, so the module call includes the `table` argument to specify which Table to use.

```
---
- name: Get OSPF information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get OSPF interface information
      juniper.device.table:
        file: "ospf.yml"
        table: "OspfInterfaceTable"
      register: result

    - name: Print response
      ansible.builtin.debug:
        var: result
```

Specify RPC Arguments

Junos PyEZ operational Tables have an optional `args` key that defines the default command options and arguments for the RPC executed by that Table. The application executes the RPC with the default options unless the user overrides the defaults. In Junos PyEZ applications, you can override the default options or pass additional options and arguments to the RPC when calling the `get()` method.

The `juniper.device.table` module also enables you to override the default options defined in the Table or pass additional options and arguments to the RPC by using the `kwargs` argument. The `kwargs` value is a dictionary of command options and values, which must be supported by the RPC and the device on which the RPC is executed.

For example, the predefined Junos PyEZ op Table `EthPortTable` in the **ethport.yml** file executes the `<get-interface-information>` RPC with the `media` command option. By default, the RPC returns information for all interfaces that match the given regular expression for the interface name.

```
EthPortTable:
  rpc: get-interface-information
  args:
```

```
    media: true
    interface_name: '[afgxe][et]-*'
  args_key: interface_name
  item: physical-interface
  view: EthPortView
```

The following Ansible playbook uses `EthPortTable` to extract information about the interfaces on Junos devices. The `kwargs` argument includes `interface_name: "ge-1/0/0"`, which overrides the `EthPortTable` default for `interface_name` and instructs the module to retrieve the requested fields only for the ge-1/0/0 interface.

```
---
- name: Get interface information
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Get interface information for Ethernet interfaces
      juniper.device.table:
        file: "ethport.yml"
        kwargs:
          interface_name: "ge-1/0/0"
      register: result

    - name: Print response
      ansible.builtin.debug:
        var: result
```

For more information about the default and user-supplied command options and arguments in Junos PyEZ Tables, see Defining Junos PyEZ Operational Tables and Use Junos PyEZ Operational Tables and Views that Parse Structured Output.

RELATED DOCUMENTATION

Use Ansible to Execute Commands and RPCs on Junos Devices | 55

Use Ansible to Retrieve Facts from Junos Devices | 52

Understanding the Ansible for Junos OS Collections and Modules | 7

Use Ansible to Halt, Reboot, or Shut Down Junos Devices

SUMMARY

Use the Juniper Networks Ansible modules to halt, reboot, or shut down Junos devices.

IN THIS SECTION

Use Ansible to Halt, Reboot, or Shut Down Devices

Juniper Networks provides an Ansible module that you can use to to halt, reboot, or shut down a Junos device. Table 11 on page 75 outlines the available module.

Table 11: Module to Halt, Reboot, or Shut Down Devices

Content Set	Module Name
`juniper.device` collection	`system`

You can use the `juniper.device.system` module to request the following operations on Junos devices. By default, the module immediately executes the requested operation and performs the operation on all Routing Engines in a dual Routing Engine or Virtual Chassis setup.

- An immediate system halt, reboot, or shutdown
- A halt, reboot, or shutdown operation with an optional delay
- A halt, reboot, or shutdown operation scheduled at a specific date and time

The system module requires one argument, action, which defines the action that the module performs. Table 12 on page 76 defines the action parameter value that is required to halt, reboot, or shut down a device and provides a brief description of each action as well as the corresponding CLI command. For information about the "zeroize" action, see "Use Ansible to Restore a Junos Device to the Factory-Default Configuration Settings" on page 107.

Table 12: action Parameter Values

Value of action Parameter	Description	Equivalent CLI Command
"halt"	Gracefully shut down the Junos OS software but maintain system power	request system halt
"reboot"	Reboot the Junos OS software	request system reboot
"shutdown"	Gracefully shut down the Junos OS software and power off the Routing Engines	request system power-off

The following Ansible playbook uses the system module with action: "reboot" to immediately reboot all Routing Engines on the hosts in the specified inventory group.

```
---
- name: Reboot Junos devices
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Reboot all REs on the device
      juniper.device.system:
        action: "reboot"
```

How to Perform a Halt, Reboot, or Shut Down with a Delay or at a Specified Time

You can delay the halt, reboot, or shut down operation by a specified number of minutes. To add a delay, set the optional `in_min` parameter to the number of minutes that the system should wait before executing the operation. The following task requests a reboot of all Routing Engines in 30 minutes:

```
---
- name: Reboot Junos devices
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Reboot all REs in 30 minutes
      juniper.device.system:
        action: "reboot"
        in_min: 30
```

You can also schedule the halt, reboot, or shutdown operation at a specific time. To schedule a time, include the `at` parameter, which takes a string that can be specified in one of the following ways:

- `now`—Immediately initiate the halt, reboot, or shut down of the software.
- `+`*`minutes`*—Number of minutes from now when the requested action is initiated.
- *`yymmddhhmm`*—Absolute time at which to initiate the requested action, specified as year, month, day, hour, and minute.
- *`hh:mm`*—Absolute time on the current day at which to initiate the requested action, specified in 24-hour time.

The following task schedules a system shutdown of all Routing Engines at 22:30 on the current day:

```
  tasks:
    - name: Shut down all REs at 22:30 on the current day
      juniper.device.system:
        action: "shutdown"
        at: "22:30"
```

How to Specify the Target Routing Engine

By default, the system module performs the requested operation on all Routing Engines in a dual Routing Engine or Virtual Chassis setup. You can also instruct the module to perform the operation on only the Routing Engine to which the application is connected or to perform the operation on all Routing Engines except the one to which the application is connected.

To specify the Routing Engines, you use the all_re and other_re parameters. Table 13 on page 78 summarizes the all_re and other_re values that are required to execute the requested operation on specific Routing Engines.

Table 13: Parameters for Specifying Routing Engines

Affected Routing Engines	all_re Parameter	other_re Parameter
All Routing Engines (default)	Omit or set to true	–
Only the connected Routing Engine	Set to false	–
All Routing Engines except the Routing Engine to which the application is connected	–	Set to true

To explicitly indicate that the operation should be performed on all Routing Engines in a dual Routing Engine or Virtual Chassis setup, include the all_re: true argument, which is the default.

```
---
- name: Reboot Junos devices
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Reboot all Routing Engines
      juniper.device.system:
        action: "reboot"
        all_re: true
```

To perform the requested action on only the Routing Engine to which the application is connected, include the `all_re: false` argument.

```
tasks:
  - name: Reboot only the connected Routing Engine
    juniper.device.system:
      action: "reboot"
      all_re: false
```

To perform the requested action on all Routing Engines in the system except for the Routing Engine to which the application is connected, include the `other_re: true` argument.

```
tasks:
  - name: Shut down all other Routing Engines
    juniper.device.system:
      action: "shutdown"
      other_re: true
```

How to Reboot or Shut Down a VM Host

On devices that have Routing Engines with VM host support, Junos OS runs as a virtual machine (VM) over a Linux-based host (VM host). The `system` module supports the `vmhost` argument, which enables you to reboot or shut down a VM Host.

When you include the `action: "reboot"` and `vmhost: true` arguments, the system reboots the host OS and compatible Junos OS on all Routing Engines by executing the `<request-vmhost-reboot>` RPC, which corresponds to the `request vmhost reboot` operational mode command.

Similarly, when you include the `action: "shutdown"` and `vmhost: true` arguments, the system shuts down the host OS and compatible Junos OS on all Routing Engines by executing the `<request-vmhost-poweroff>` RPC, which corresponds to the `request vmhost power-off` operational mode command.

The following playbook performs a VM host reboot, which reboots both the host OS and the guest Junos OS.

```
---
- name: Reboot VM Hosts
  hosts: vm_hosts
  connection: local
```

```
  gather_facts: no

  tasks:
    - name: Reboot VM host
      juniper.device.system:
        action: "reboot"
        vmhost: true
        all_re: false
```

Example: Use Ansible to Reboot Junos Devices

IN THIS SECTION

The `juniper.device.system` module enables you to halt, reboot, or shut down a Junos device. This example uses the `system` module to reboot a Junos device.

Requirements

This example uses the following hardware and software components:

- Configuration management server running Ansible 2.10 or later with the `juniper.device` collection installed
- Junos device with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the Ansible control node and Junos device
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the `juniper.device.system` module to reboot a Junos device. The value of the module's `action` argument defines the operation to execute on the host.

When calling the module from a playbook, we recommend that you use an interactive prompt to confirm that the user does intend to reboot the specified devices. If a user unintentionally runs the playbook and there is no check, it could adversely affect any networks that require the impacted devices. As a precaution, this playbook uses an interactive prompt to verify that the user intends to reboot the devices and requires that the user manually type '**yes**' on the command line to execute the module. If the `Confirmation check` task fails, the Ansible control node skips the other tasks in the play for that device.

This playbook includes the `Check NETCONF connectivity` task, which utilizes the `ansible.builtin.wait_for` module to try to establish a NETCONF session with the Junos device using the default NETCONF port 830. If the control node fails to establish a NETCONF session with the device during playbook execution, then it skips the remaining tasks in the play for that device.

The task that reboots the device executes the `system` module provided that the confirmation and NETCONF checks were successful. The `action` argument is set to the value `"reboot"`, which indicates that the software should be rebooted. The `in_min: 2` argument instructs the module to wait for the specified number of minutes before executing the reboot command. This provides time for any users to log out of the system.

The task stores the module result in the `result` variable and notifies two handlers. The `pause_for_reboot` handler waits a specified amount of time after the reboot operation is initiated to prevent the `wait_reboot` handler from falsely detecting that the device is online before the reboot takes place. The `wait_reboot` handler then tries to establish a session with the device to verify that the device comes back online after the reboot. The `wait_time_after_reboot` variable defines the length of time that the control node attempts to reconnect with the device.

Configuration

IN THIS SECTION

Create and Execute the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the `system` module to reboot a Junos device:

1. Include the boilerplate for the playbook and this play, which executes the modules locally.

```
---
- name: Reboot Junos devices
  hosts: dc1
  connection: local
  gather_facts: no
```

2. Define or import any necessary variables.

```
  vars:
    wait_time_after_reboot: 300
    netconf_port: 830
```

3. Create an interactive prompt to prevent users from accidentally executing the module without first understanding the implications.

```
  vars_prompt:
  - name: "reboot_confirmation"
    prompt: "This playbook reboots devices. Enter 'yes' to continue"
    private: no
```

4. Create the task that confirms the user's intent.

```
  tasks:
    - name: Confirmation check
      fail: msg="Playbook run confirmation failed"
      when: reboot_confirmation != "yes"
```

5. (Optional) Create a task to verify NETCONF connectivity.

```
  - name: Check NETCONF connectivity
    ansible.builtin.wait_for:
      host: "{{ inventory_hostname }}"
      port: "{{ netconf_port }}"
      timeout: 5
```

6. Create the task to reboot the device after a specified number of minutes and then notify the handlers.

```
  - name: Reboot all Routing Engines on the Junos device
    juniper.device.system:
      action: "reboot"
      in_min: 2
      all_re: true
    register: result
    notify:
      - pause_for_reboot
      - wait_reboot
```

7. (Optional) Create a task to print the response.

```
  - name: Print response
    ansible.builtin.debug:
      var: result
```

8. Create the handler that pauses after rebooting and the handler that verifies that the device comes back online after rebooting.

   The handler names should be the same as those referenced in the reboot task.

```
handlers:
  - name: pause_for_reboot
    pause:
      seconds: 180
    when: result.reboot
  - name: wait_reboot
    ansible.builtin.wait_for:
```

```
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port  }}"
        timeout: "{{ wait_time_after_reboot }}"
      when: result.reboot
```

Results

On the Ansible control node, review the completed playbook. If the playbook does not display the intended code, repeat the instructions in this example to correct the playbook.

```
---
- name: Reboot Junos devices
  hosts: dc1
  connection: local
  gather_facts: no

  vars:
    wait_time_after_reboot: 300
    netconf_port: 830

  vars_prompt:
  - name: "reboot_confirmation"
    prompt: "This playbook reboots devices. Enter 'yes' to continue"
    private: no

  tasks:
    - name: Confirmation check
      fail: msg="Playbook run confirmation failed"
      when: reboot_confirmation != "yes"

    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: 5

    - name: Reboot all Routing Engines on the Junos device
      juniper.device.system:
        action: "reboot"
        in_min: 2
        all_re: true
```

```
    register: result
    notify:
      - pause_for_reboot
      - wait_reboot

  - name: Print response
    ansible.builtin.debug:
      var: result

handlers:
  - name: pause_for_reboot
    pause:
      seconds: 180
    when: result.reboot
  - name: wait_reboot
    ansible.builtin.wait_for:
      host: "{{ inventory_hostname }}"
      port: "{{ netconf_port  }}"
      timeout: "{{ wait_time_after_reboot }}"
    when: result.reboot
```

Execute the Playbook

IN THIS SECTION

Procedure

Step-by-Step Procedure

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
user@ansible-cn:~/ansible$ ansible-playbook ansible-pb-junos-reboot.yaml
This playbook reboots devices. Enter 'yes' to continue: yes
```

```
PLAY [Reboot Junos devices] ****************************************************

TASK [Confirmation check] ******************************************************
skipping: [dc1a.example.net]

TASK [Check NETCONF connectivity] **********************************************
ok: [dc1a.example.net]

TASK [Reboot all Routing Engines on the Junos device] *************
changed: [dc1a.example.net]

TASK [Print response] **********************************************************
ok: [dc1a.example.net] => {
    "result": {
        "action": "reboot",
        "all_re": true,
        "changed": true,
        "failed": false,
        "media": false,
        "msg": "reboot successfully initiated. Response got Shutdown at Fri Dec 11 17:36:50
2020. [pid 11595]",
        "other_re": false,
        "reboot": true,
        "vmhost": false
    }
}

RUNNING HANDLER [pause_for_reboot] *********************************************
Pausing for 180 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [dc1a.example.net]

RUNNING HANDLER [wait_reboot] **************************************************
ok: [dc1a.example.net]

PLAY RECAP *********************************************************************
dc1a.example.net   : ok=5    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
```

Verification

IN THIS SECTION

Verify the Reboot

Purpose

Verify that the Junos device successfully rebooted.

Action

When you execute the playbook, review the output of the `wait_reboot` task for each device.

```
RUNNING HANDLER [wait_reboot] ************************************************
ok: [dc1a.example.net]
```

Meaning

The `wait_reboot` result indicates whether the control node successfully established a session with the device after it rebooted. If the result indicates success, the device is online.

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
1.0.3	Starting with `juniper.device` collection Release 1.0.3, the `system` module supports shutting down a VM Host.

Use Ansible to Install Software on Junos Devices

SUMMARY

Use the Juniper Networks Ansible modules to install software on Junos devices.

IN THIS SECTION

Use Ansible to Install Software

Juniper Networks provides an Ansible module that enables you to install or upgrade the software image on a Junos device. Table 14 on page 88 outlines the module.

Table 14: Software Module

Content Set	Module Name
`juniper.device` collection	`software`

The following sections discuss how to specify the software image location and the general software installation process and options when using the Ansible module to install software packages on Junos devices. They also discuss how to perform more specialized upgrade scenarios such as a VM host

upgrade, a unified in-service software upgrade (unified ISSU), or a nonstop software upgrade (NSSU) on devices that support these features.

How to Specify the Software Image Location

When you use the `juniper.device.software` module to install software on Junos devices, you can download the software package to the Ansible control node, and the module, by default, copies the package to the target device before performing the installation. For mixed Virtual Chassis environments, the software packages must reside on the Ansible control node. For standalone devices or non-mixed Virtual Chassis environments, you can also instruct the module to install a software image that already resides on the target Junos device or resides at a URL that is reachable from the target device.

Table 15 on page 90 outlines the module arguments that you must set depending on the software package location. The module must always include either the `local_package`, `pkg_set`, or `remote_package` argument. The `no_copy` argument defaults to `false`, which instructs the module to copy the software package from the specified location on the Ansible control node to the target device.

Table 15: Module Arguments for Software Package Location

Software Package Location	no_copy Parameter	local_package or pkg_set Parameter	remote_package Parameter
Ansible control node	Omit or set to false	For standalone devices or non-mixed Virtual Chassis environments: Set local_package to the file path, including the filename, of the software package on the local control node. File paths are relative to the playbook directory.	(Optional) File path on the target device to which the software package is copied. The default directory is **/var/tmp**. If remote_package includes a filename, it must match the filename specified in local_package.
		For mixed Virtual Chassis environments: Set pkg_set to a list of the file paths, including the filenames, of one or more software packages on the local control node. File paths are relative to the playbook directory.	–
Remote Location	–	–	URL from the perspective of the target Junos device from which the software package is installed.
Target device	Set to true	–	File path on the target device where the software package must already reside. The default directory is **/var/tmp**.

If the software package resides on the Ansible control node, include the argument that is appropriate for your installation:

- local_package—Install software on a standalone Junos device or on members in a non-mixed Virtual Chassis. The argument value is a single string specifying the absolute or relative path to the software image.
- pkg_set—Install software on the members in a mixed Virtual Chassis. The argument value is a list of strings that specify the absolute or relative file paths of the software images, in no particular order, for the various Virtual Chassis members.

For example:

```
pkg_set:
  - 'software/jinstall-qfx-5-13.2X51-D35.3-domestic-signed.tgz'
  - 'software/jinstall-ex-4300-13.2X51-D35.3-domestic-signed.tgz'
```

By default, when you include the `local_package` or `pkg_set` argument, the module copies any software packages from the Ansible control node to the **/var/tmp** directory on the target Junos device (individual device or Virtual Chassis primary device). If you want to copy the `local_package` image to a different directory, define the `remote_package` argument and specify the target directory. If the `remote_package` argument includes a filename, the filenames of the `local_package` and `remote_package` arguments must be identical, or the module generates an error.

If the software package already resides on the target Junos device (individual device or Virtual Chassis primary device), the module must include the `no_copy: true` argument as well the `remote_package` argument, which specifies the file path to an existing software package on the target device. If `remote_package` does not specify a directory, the default is **/var/tmp**.

If the software package resides at a location other than the Ansible control node or target device, the module must include the `remote_package` argument and specify the location of the software package. The value of `remote_package` is a URL from the perspective of the target Junos device. For information about acceptable URL formats, see Format for Specifying Filenames and URLs in Junos OS CLI Commands.

Installation Process Overview

To use Ansible to install a software package on a Junos device, execute the `software` module, and provide any necessary arguments. For example:

```
---
- name: Perform a Junos OS software upgrade
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Upgrade Junos OS
      juniper.device.software:
        local_package: "software/jinstall-ppc-17.3R1.10-signed.tgz"
        no_copy: false
        validate: true
```

```
    register: response
  - name: Print the response
    ansible.builtin.debug:
      var: response
```

When you execute the `software` module, it performs the following operations:

1. Compares the Junos OS version specified in the `version` argument, or in the software package filename if the `version` argument is omitted, to the installed version on the managed device. If the installed and desired versions are identical, the module skips the remaining installation steps and sets `changed` and `failed` to `false`.
2. If the software package is located on the Ansible control node, and the `no_copy` parameter is omitted or set to `false`, the module performs the following operations:
   - Computes the checksum of the local software package or packages using the algorithm specified in the `checksum_algorithm` argument. Acceptable `checksum_algorithm` values are `md5`, `sha1`, and `sha256`. The default is `md5`. Alternatively, you can provide a checksum in the `checksum` argument.
   - Performs a storage cleanup on the target device to create space for the software package, unless the `cleanfs` argument is set to `false`.
   - SCP or FTP copies any packages to the target device, if files with the same names and checksums do not already reside in the target location on the device.

     When the module includes `local_package`, the package is copied to the `remote_package` directory, or if `remote_package` is not specified, to the **/var/tmp** directory. When the module includes `pkg_set`, the packages are always copied to the **/var/tmp** directory on the Virtual Chassis primary device.

     > **NOTE**: If the `cleanfs` argument is omitted or set to `true`, the module copies the software package to the device even if it initially existed in the target location, because the storage cleanup operation removes the existing file. If `cleanfs: false` is present and the file already resides at the target location, the module skips the file copy operation.

   - Computes the checksum of each remote file and compares it to the checksum of the local file.

Once the software package is on the target device, whether downloaded there initially or copied over by the module, the module then performs the following operations:

1. Validates the configuration against the new package when the `validate` argument is set to `true`.

   > **NOTE**: By default, the `software` module does not validate the software package or bundle against the existing configuration as a prerequisite to adding the software package. To ensure

that the active configuration will work with the new software image, set the `validate` argument to `true`.

2. Installs the package on each individual Routing Engine, unless `all_re` is set to `false`.
3. Reboots each upgraded Routing Engine, unless the `reboot` argument is set to `false`.

The `software` module enables you to log the progress of the installation by including the `logfile` module argument. By default, only messages of severity level WARNING or higher are logged. To log messages of severity level INFO or higher, which is required to log messages for the general installation process, execute the playbook with the `-v` or `--verbose` command-line option.

How to Specify Timeout Values

The `juniper.device.software` module performs operations over a NETCONF session. The default time for a NETCONF RPC to time out is 30 seconds. During the installation process, certain operations increase the RPC timeout interval as follows:

- Copying and installing the package on the device—1800 seconds (30 minutes)
- Computing the checksum—300 seconds (5 minutes)
- Performing a storage cleanup—300 seconds (5 minutes)

In some cases, the installation process, checksum calculation, or storage cleanup might exceed these time intervals. You can change the timeout value for these operations by setting the `install_timeout`, `checksum_timeout`, and `cleanfs_timeout` arguments to the required number of seconds in the module's argument list. For example:

```
- name: Upgrade Junos OS
  juniper.device.software:
    local_package: "software/jinstall-ppc-17.3R1.10-signed.tgz"
    validate: true
    install_timeout: 2000
    checksum_timeout: 420
    cleanfs_timeout: 600
```

How to Specify Installation Options That Do Not Have an Equivalent Module Argument

When you use the `juniper.device.software` module to install software on a device, the module invokes the appropriate RPC for the included installation arguments. For example, the module invokes the <request-package-add> RPC for standard Junos OS installations, the <request-vmhost-package-add> RPC for VM host upgrades, the <request-package-in-service-upgrade> RPC for unified ISSU scenarios, and so on.

The module supports explicit arguments for many of the installation options, for example, the `validate` option. The module also supports the `kwargs` argument, which enables you to include any additional options that are supported by the RPC but which do not have an equivalent module argument. The `kwargs` argument takes a dictionary of key/value pairs of additional supported options.

For the current list of options supported by the module, see the API reference documentation for the module. For a list of all available options for a specific RPC, see the documentation for the equivalent command or search for the RPC's request tag in the Junos XML API Explorer.

NOTE: You should only include installation options that are supported on the target Junos device for the given RPC.

In the following playbook, the `software` module installs a new software image on the target hosts. The module includes the `kwargs` argument with `unlink: true`. This argument, which removes the software package from the directory after a successful upgrade, is equivalent to including the <unlink/> option in the <request-package-add> RPC.

```
---
- name: Perform a Junos OS software upgrade
  hosts: router1
  connection: local
  gather_facts: no

  tasks:
    - name: Upgrade Junos OS
      juniper.device.software:
        local_package: "software/jinstall-ppc-17.3R1.10-signed.tgz"
        kwargs:
          unlink: true
      register: response
    - name: Print the response
```

```
    ansible.builtin.debug:
      var: response
```

How to Perform a VM Host Upgrade

On devices that have Routing Engines with VM host support, Junos OS runs as a virtual machine (VM) over a Linux-based host (VM host). A VM host upgrade requires a VM Host installation package (**junos-vmhost-install-*x*.tgz**) and upgrades the host OS and compatible Junos OS. In the CLI, you perform the upgrade using the `request vmhost software add` operational mode command, which corresponds to the `<request-vmhost-package-add>` RPC.

The `juniper.device.software` module supports the `vmhost: true` argument for performing a VM host upgrade. When the argument is present, the module performs the installation using the `<request-vmhost-package-add>` RPC.

The following playbook upgrades and reboots the Junos OS and host OS on the specified devices:

```
---
- name: Upgrade VM Hosts
  hosts: vm_hosts
  connection: local
  gather_facts: no

  tasks:
    - name: Perform a VM host upgrade
      juniper.device.software:
        local_package: "junos-vmhost-install-qfx-x86-64-18.1R1.9.tgz"
        vmhost: true
      register: response
    - name: Print the response
      ansible.builtin.debug:
        var: response
```

How to Perform a Unified ISSU or NSSU

The `juniper.device.software` module supports performing a unified in-service software upgrade (unified ISSU) or a nonstop software upgrade (NSSU) on devices that support the feature and meet the

necessary requirements. For more information about the unified ISSU and NSSU features, see the software documentation for your product.

The unified ISSU feature enables you to upgrade between two different Junos OS releases with no disruption on the control plane and with minimal disruption of traffic. To perform a unified in-service software upgrade, the `software` module must include the `issu: true` argument. For example:

```
---
- name: Perform a Junos OS software upgrade
  hosts: mx1
  connection: local
  gather_facts: no

  tasks:
    - name: Perform a unified ISSU
      juniper.device.software:
        local_package: "junos-install-mx-x86-64-17.2R1.13.tgz"
        issu: true
      register: response
    - name: Print the response
      ansible.builtin.debug:
        var: response
```

The NSSU feature enables you to upgrade the Junos OS software running on a switch or Virtual Chassis with redundant Routing Engines with minimal disruption to network traffic. To perform a nonstop software upgrade, the `software` module must include the `nssu: true` argument. For example:

```
---
- name: Perform a Junos OS software upgrade
  hosts: ex1
  connection: local
  gather_facts: no

  tasks:
    - name: Perform an NSSU
      juniper.device.software:
        local_package: "jinstall-ex-4300-17.3R1.10-signed.tgz"
        nssu: true
      register: response
    - name: Print the response
```

```
      ansible.builtin.debug:
        var: response
```

How to Install Software on an EX Series Virtual Chassis Member

Generally, when you upgrade a non-mixed EX Series Virtual Chassis, you follow the installation process outlined in "Installation Process Overview" on page 91 to upgrade the entire Virtual Chassis. However, there might be times when you need to install software on specific member switches in the Virtual Chassis. Starting with Release 1.0.3 of the `juniper.device` collection, you can install a software package on individual member switches in a non-mixed EX Series Virtual Chassis.

To install software on specific members, include the `member_id` argument and define a list of strings that specify the member IDs. The system installs the software package from the Virtual Chassis primary device onto the specified members.

The following Ansible playbook upgrades the software on member 0 and member 1 in the EX Series Virtual Chassis:

```
---
- name: Upgrade specific EX VC members
  hosts: ex_vc
  connection: local
  gather_facts: no

  vars:
    OS_version: "23.2R1.13"
    OS_package: "junos-install-ex-x86-64-23.2R1.13.tgz"
    pkg_dir: "software"
    log_dir: /var/log/

  tasks:
    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: 830
        timeout: 5

    - name: Install package on EX VC members
      juniper.device.software:
        version: "{{ OS_version }}"
```

```
        local_package: "{{ pkg_dir }}/{{ OS_package }}"
        member_id: ["0","1"]
        logfile: "{{ log_dir }}/software.log"
```

Example: Use Ansible to Install Software

IN THIS SECTION

This example uses the `juniper.device.software` module to install a software image on a Junos device.

Requirements

This example uses the following hardware and software components:

- Configuration management server running Ansible 2.10 or later with the `juniper.device` collection installed
- Junos device with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the Ansible control node and Junos device
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the `juniper.device.software` module to upgrade Junos OS on the hosts in the specified inventory group. In this example, the software image resides on the Ansible control node, and the module copies the image to the target device before installing it. The module does not explicitly define a `host` argument, so the module operates on the default host, which is `{{ inventory_hostname }}`.

This playbook includes the `Check NETCONF connectivity` task, which utilizes the `ansible.builtin.wait_for` module to try to establish a NETCONF session with the Junos device using the default NETCONF port 830. If the control node fails to establish a NETCONF session with a device during playbook execution, then it skips the remaining tasks in the play for that device.

The `Install Junos OS package` task executes the `software` module provided that the NETCONF check was successful. The `version` argument defines the desired Junos OS version as it would be reported by the `show version` command on the Junos device. During playbook execution, the module first checks that the requested version is not already installed on the device. If the requested version is different from the currently installed version, the module installs the requested version.

The `local_package` argument defines the path of the Junos OS software package on the Ansible control node. During the installation, the module performs a storage cleanup operation on the target device, copies the software image to the **/var/tmp** directory on the device, verifies the file's checksum, validates the new software against the active configuration, and then installs the software on each Routing Engine on the target host. By default, the `software` module reboots each Routing Engine after the installation completes; however, this task explicitly sets `reboot: true` for clarity.

The task stores the module result in the `response` variable and notifies one handler. If the user does not execute the playbook using check mode, the `wait_reboot` handler then tries to establish a session with the device to verify that the device is back online. The `wait_time` variable defines the length of time that the control node attempts to reconnect with the device.

This example includes the `logfile` parameter to log the progress of the installation. This is important for debugging purposes should the installation fail as well as for logging the dates and times of installations on the devices. The user executing the playbook must have permissions to write to the specified log file. By default, only messages of severity level WARNING or higher are logged. In this example, the playbook is executed with the `-v` option to log messages of severity level INFO or higher to monitor the installation.

Configuration

IN THIS SECTION

Creating the Ansible Playbook

To create a playbook that uses the `juniper.device.software` module to install a software image on a Junos device:

1. Include the boilerplate for the playbook and this play, which executes the modules locally.

```
---
- name: Install Junos OS
  hosts: mx1
  connection: local
  gather_facts: no
```

2. Define or import any necessary variables, which for this example, includes the desired Junos OS version and the path to the new image, among others.

```
  vars:
    OS_version: "23.4R1.9"
    OS_package: "junos-install-mx-x86-64-23.4R1.9.tgz"
    pkg_dir: "software"
    log_dir: "{{ playbook_dir }}"
    netconf_port: 830
    wait_time: 3600
```

3. (Optional) Create a task to verify NETCONF connectivity.

```
  tasks:
    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: 5
```

4. Create the task to install the Junos OS package on the device and notify the handler.

```
    - name: Install Junos OS package
      juniper.device.software:
        version: "{{ OS_version }}"
        local_package: "{{ pkg_dir }}/{{ OS_package }}"
        reboot: true
        validate: true
        logfile: "{{ log_dir }}/software.log"
      register: response
```

```
      notify:
      - wait_reboot
```

5. (Optional) Create a task to print the module response.

```
    - name: Print response
      ansible.builtin.debug:
        var: response
```

6. Create the handler that verifies that the device comes back online after rebooting.

   The handler name should be the same as that referenced in the installation task.

```
  handlers:
    - name: wait_reboot
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: "{{ wait_time }}"
      when: not response.check_mode
```

Results

On the Ansible control node, review the completed playbook. If the playbook does not display the intended code, repeat the instructions in this example to correct the playbook.

```
---
- name: Install Junos OS
  hosts: mx1
  connection: local
  gather_facts: no

  vars:
    OS_version: "23.4R1.9"
    OS_package: "junos-install-mx-x86-64-23.4R1.9.tgz"
    pkg_dir: "software"
    log_dir: "{{ playbook_dir }}"
    netconf_port: 830
    wait_time: 3600
```

```
  tasks:
    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: 5

    - name: Install Junos OS package
      juniper.device.software:
        version: "{{ OS_version }}"
        local_package: "{{ pkg_dir }}/{{ OS_package }}"
        reboot: true
        validate: true
        logfile: "{{ log_dir }}/software.log"
      register: response
      notify:
      - wait_reboot

    - name: Print response
      ansible.builtin.debug:
        var: response

  handlers:
    - name: wait_reboot
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: "{{ wait_time }}"
      when: not response.check_mode
```

Execute the Playbook

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
user@ansible-cn:~/ansible$ ansible-playbook -v ansible-pb-junos-install-os.yaml
Using /etc/ansible/ansible.cfg as config file

PLAY [Install Junos OS] ********************************************************
```

```
TASK [Check NETCONF connectivity] ****************************************
ok: [mx1a.example.com] => {"changed": false, "elapsed": 0, "match_groupdict": {},
"match_groups": [], "path": null, "port": 830, "search_regex": null, "state": "started"}

TASK [Install Junos OS package] ******************************************
changed: [mx1a.example.com] => {"changed": true, "check_mode": false, "msg": "Package /home/
user/ansible/software/junos-install-mx-x86-64-23.4R1.9.tgz successfully installed. Response
from device is: \nVerified junos-install-mx-x86-64-23.4R1.9 signed by
PackageProductionECP256_2023 method ECDSA256+SHA256\n
[...output truncated...]
NOTICE: 'pending' set will be activated at next reboot... Reboot successfully initiated.
Reboot message: Shutdown NOW! [pid 79385]"}

TASK [Print response] ****************************************************
ok: [mx1a.example.com] => {
    "response": {
        "changed": true,
        "check_mode": false,
        "failed": false,
        "msg": "Package /home/user/ansible/software/junos-install-mx-x86-64-23.4R1.9.tgz
successfully installed. Response from device is: \nVerified junos-install-mx-x86-64-23.4R1.9
signed by PackageProductionECP256_2023 method ECDSA256+SHA256\nVerified auto-snapshot signed
by PackageProductionECP256_2023 method ECDSA256+SHA256\n
[...output truncated...]
NOTICE: 'pending' set will be activated at next reboot... Reboot successfully initiated.
Reboot message: Shutdown NOW! [pid 79385]"
    }
}

RUNNING HANDLER [wait_reboot] ********************************************
ok: [mx1a.example.com] => {"changed": false, "elapsed": 250, "match_groupdict": {},
"match_groups": [], "path": null, "port": 830, "search_regex": null, "state": "started"}

PLAY RECAP ***************************************************************
mx1a.example.com   : ok=4    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Verification

IN THIS SECTION

Verify the Installation

Purpose

Verify that the software installation was successful.

Action

The playbook output should indicate any failed tasks. However, you can also review the contents of the log file defined in the playbook for details about the installation. Sample log file output is shown here. Some output has been omitted for brevity.

```
2024-08-23 22:20:49,455 - ncclient.transport.ssh - INFO - Connected (version 2.0, client
OpenSSH_7.9)
2024-08-23 22:20:52,950 - ncclient.transport.ssh - INFO - Authentication (publickey) successful!
...
2024-08-23 22:21:00,770 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] computing checksum on local package: /home/user/ansible/software/junos-
install-mx-x86-64-23.4R1.9.tgz
2024-08-23 22:21:08,070 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] cleaning filesystem ...
...
2024-08-23 22:21:08,329 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] before copy, computing checksum on remote package: /var/tmp/junos-install-mx-
x86-64-23.4R1.9.tgz
...
2024-08-23 22:21:08,491 - paramiko.transport - INFO - Connected (version 2.0, client OpenSSH_7.9)
2024-08-23 22:21:08,958 - paramiko.transport - INFO - Authentication (publickey) successful!
2024-08-23 22:21:16,846 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 363528192 / 3635202890 (10%)
2024-08-23 22:21:24,405 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 727056384 / 3635202890 (20%)
2024-08-23 22:21:31,966 - jnpr.ansible_module.juniper.device.software - INFO -
```

```
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 1090568192 / 3635202890 (30%)
2024-08-23 22:21:39,652 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 1454096384 / 3635202890 (40%)
2024-08-23 22:21:47,631 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 1817608192 / 3635202890 (50%)
2024-08-23 22:21:55,343 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 2181136384 / 3635202890 (60%)
2024-08-23 22:22:02,878 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 2544648192 / 3635202890 (70%)
2024-08-23 22:22:11,395 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 2908176384 / 3635202890 (80%)
2024-08-23 22:22:19,949 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 3271688192 / 3635202890 (90%)
2024-08-23 22:22:27,522 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] b'junos-install-mx-x86-64-23.4R1.9.tgz': 3635202890 / 3635202890 (100%)
2024-08-23 22:22:27,533 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] after copy, computing checksum on remote package: /var/tmp/junos-install-mx-
x86-64-23.4R1.9.tgz
...
2024-08-23 22:22:44,891 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] checksum check passed.
2024-08-23 22:22:44,892 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] validating software against current config, please be patient ...
...
2024-08-23 22:27:52,538 - ncclient.transport.ssh - INFO - [host mx1a.example.com session-id
27526] Received message from host
2024-08-23 22:27:52,542 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] software validate package-result: 0
Output:
Verified junos-install-mx-x86-64-23.4R1.9 signed by PackageProductionECP256_2023 method
ECDSA256+SHA256
Adding junos-mx-x86-64-23.4R1.9 ...
...
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded

2024-08-23 22:27:52,542 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] installing software on RE0 ... please be patient ...
...
2024-08-23 22:30:57,510 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] software pkgadd package-result: 0
Output:
```

```
Verified junos-install-mx-x86-64-23.4R1.9 signed by PackageProductionECP256_2023 method
ECDSA256+SHA256
...
NOTICE: 'pending' set will be activated at next reboot...

2024-08-23 22:30:57,510 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] installing software on RE1 ... please be patient ...
...
2024-08-23 22:34:30,228 - jnpr.ansible_module.juniper.device.software - INFO -
[mx1a.example.com] software pkgadd package-result: 0
Output:
Pushing /var/tmp/junos-install-mx-x86-64-23.4R1.9.tgz to re1:/var/tmp/junos-install-mx-
x86-64-23.4R1.9.tgz
Verified junos-install-mx-x86-64-23.4R1.9 signed by PackageProductionECP256_2023 method
ECDSA256+SHA256
...
NOTICE: 'pending' set will be activated at next reboot...
...
2024-08-23 22:34:30,732 - ncclient.operations.rpc - INFO - [host mx1a.example.com session-id
27526] Requesting 'CloseSession'
```

Meaning

The log file contents indicate that the image was successfully copied and installed on both Routing Engines on the target device.

RELATED DOCUMENTATION

- Understanding the Ansible for Junos OS Collections and Modules | 7
- Use Ansible to Halt, Reboot, or Shut Down Junos Devices | 75
- Use Ansible to Restore a Junos Device to the Factory-Default Configuration Settings | 107

Use Ansible to Restore a Junos Device to the Factory-Default Configuration Settings

SUMMARY

Use the Juniper Networks Ansible modules to restore a Junos device to its factory-default configuration settings.

IN THIS SECTION

How to Use Ansible to Restore the Factory-Default Configuration Settings

Juniper Networks provides an Ansible module that you can use to restore a Junos device to its factory-default configuration settings. Table 16 on page 107 outlines the module.

Table 16: Module to Zeroize Devices

Content Set	Module Name
`juniper.device` collection	`system`

To use the `juniper.device.system` module to restore a device to its factory-default configuration settings, set the module's `action` argument to `'zeroize'`. After you restore a device to the factory-default configuration settings, you must log in through the console as root in order to access the device.

The `action: "zeroize"` argument causes the module to execute the `request system zeroize` operational command on the target host. This command removes all configuration information on the specified Routing Engines, resets all key values on the device, and then reboots the device and resets it to the factory-default configuration settings. The zeroize operation removes all data files, including customized configuration and log files, by unlinking the files from their directories. It also removes all user-created files from the system including all plain-text passwords, secrets, and private keys for SSH, local encryption, local authentication, IPsec, RADIUS, TACACS+, and SNMP.

For more information, see:

- request system zeroize (Junos OS)
- request system zeroize (Junos OS Evolved)

The following Ansible playbook uses the `juniper.device.system` module with `action: "zeroize"` to reset all Routing Engines on each host in the inventory group to the factory-default configuration settings.

```
---
- name: Restore Junos devices to factory-default configuration
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Restore all Routing Engines to factory-default configuration
      juniper.device.system:
        action: "zeroize"
```

By default, the `action: "zeroize"` operation resets all Routing Engines in a dual Routing Engine or Virtual Chassis setup to the factory-default configuration settings. You can also instruct the module to perform the operation on only the Routing Engine to which the application is connected.

To explicitly indicate that the operation should be performed on all Routing Engines in a dual Routing Engine or Virtual Chassis setup, include the `all_re: True` argument, which is the default.

```
  tasks:
    - name: Restore all Routing Engines to factory-default configuration
      juniper.device.system:
        action: "zeroize"
        all_re: True
```

To perform the requested action on only the Routing Engine to which the application is connected, include the `all_re: False` argument.

```
  tasks:
    - name: Restore connected Routing Engine to factory-default configuration
      juniper.device.system:
        action: "zeroize"
        all_re: False
```

To instruct the module to also scrub all memory and media, in addition to removing all configuration and log files, include the `media: True` argument. Including the `media: True` argument is equivalent to executing the `request system zeroize media` operational mode command. The `media` option scrubs every storage device that is attached to the system, including disks, flash memory devices, removable USBs, and so on. The duration of the scrubbing process is dependent on the size of the media being erased.

```
tasks:
  - name: Restore device to the factory-default configuration and scrub media
    juniper.device.system:
      action: "zeroize"
      media: True
```

Example: Use Ansible to Restore the Factory-Default Configuration Settings

IN THIS SECTION

This example demonstrates how to use the `juniper.device.system` module to restore a Junos device to its factory-default configuration settings. You can execute the module using any type of connection; however, once you reset the device, you can only access it again as root through a console server or the **CONSOLE** port. This example connects to the devices through a console server.

Requirements

This example uses the following hardware and software components:

- Configuration management server running Ansible 2.10 or later with the `juniper.device` collection installed

- Junos device that has access to the console port through a console server and has a user account configured with appropriate permissions
- Existing Ansible inventory file with required hosts defined

Overview

This example creates an Ansible playbook that uses the `juniper.device.system` module to reset each host in the inventory group to its factory-default configuration settings. The value of the module's `action` argument defines the operation to execute on the host. Setting `action` to `"zeroize"` executes the `request system zeroize` command on each host. This command removes all configuration information on the Routing Engines, resets all key values on the device, and then reboots the device and resets it to the factory-default configuration settings.

> **NOTE**: The `request system zeroize` command removes all data files, including customized configuration and log files, by unlinking the files from their directories. The command also removes all user-created files from the system including all plain-text passwords, secrets, and private keys for SSH, local encryption, local authentication, IPsec, RADIUS, TACACS+, and SNMP.

When calling the module from a playbook, we recommend that you use an interactive prompt to confirm that the user does intend to reset the devices. If a user unintentionally runs the playbook and there is no check, it could inadvertently revert devices back to factory-default configurations and disrupt any networks that require those devices. As a precaution, this playbook uses an interactive prompt to verify that the user intends to reset the devices and requires that the user manually type '**yes**' on the command line in order to execute the module. If the `Confirmation check` task fails, the Ansible control node skips the other tasks in the play for that device.

The playbook executes the `juniper.device.system` module provided that the confirmation check was successful. The `mode: "telnet"` and `port: 23` arguments instruct the module to telnet to port 23 of the console server. The `password` parameter is set to the value of the `password` variable, which the playbook prompts for during execution. After the reboot, you must log in through the console as root in order to access the device.

Configuration

IN THIS SECTION

Create and Execute the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the `juniper.device.system` module to restore a Junos device to its factory-default configuration settings:

1. Include the boilerplate for the playbook and this play, which executes the modules locally.

```
---
- name: Restore Junos devices to factory-default configuration settings
  hosts: dc1_console
  connection: local
  gather_facts: no
```

2. Create an interactive prompt to prevent the accidental execution of the module.

```
  vars_prompt:
    - name: reset_confirmation
      prompt: >
        This playbook resets hosts to factory-default configurations!
        Enter 'yes' to continue.
      default: "no"
      private: no
```

3. Create an interactive prompt for the `password` variable, if the user credentials are not already passed in through some other means.

```


    - name: "device_password"
      prompt: "Device password"
      private: yes
```

4. Define the connection parameters.

```
  vars:
    password: "{{ device_password }}"
    mode: "telnet"
    port: 23
```

5. Create the task that confirms the user's intent.

```
  tasks:
    - name: Confirmation check
      fail: msg="Playbook run confirmation failed"
      when: reset_confirmation != "yes"
```

6. Create the task to reset all Routing Engines on the device to the factory-default configuration settings.

```
    - name: Restore all Routing Engines to factory-default configuration
      juniper.device.system:
        action: "zeroize"
        timeout: 120
      register: result
```

7. (Optional) Create a task to print the response.

```
    - name: Print response
      ansible.builtin.debug:
        var: result
```

Results

On the Ansible control node, review the completed playbook. If the playbook does not display the intended code, repeat the instructions in this example to correct the playbook.

```
---
- name: Restore Junos devices to factory-default configuration settings
  hosts: dc1_console
  connection: local
  gather_facts: no

  vars_prompt:
    - name: reset_confirmation
      prompt: >
        This playbook resets hosts to factory-default configurations!
        Enter 'yes' to continue.
      default: "no"
```

```
    private: no

  - name: "device_password"
    prompt: "Device password"
    private: yes

vars:
  password: "{{ device_password }}"
  mode: "telnet"
  port: 23

tasks:
  - name: Confirmation check
    fail: msg="Playbook run confirmation failed"
    when: reset_confirmation != "yes"

  - name: Restore all Routing Engines to factory-default configuration
    juniper.device.system:
      action: "zeroize"
      timeout: 120
    register: result

  - name: Print response
    ansible.builtin.debug:
      var: result
```

Execute the Playbook

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
root@ansible-cn:~/ansible# ansible-playbook ansible-pb-junos-zeroize.yaml
This playbook resets hosts to factory-default configurations!  Enter 'yes' to continue.
 [no]: yes
Device password:

PLAY [Restore Junos devices to factory-default configuration settings]

TASK [Confirmation check] ******************************************
skipping: [dc1a-console.example.net]
```

```
TASK [Restore all Routing Engines to factory-default configuration] ****
changed: [dc1a-console.example.net]

TASK [Print response] ***********************************************
ok: [dc1a-console.example.net] => {
    "result": {
        "action": "zeroize",
        "all_re": true,
        "changed": true,
        "failed": false,
        "media": false,
        "msg": "zeroize successfully initiated.",
        "other_re": false,
        "reboot": false
        "vmhost": false
    }
}

PLAY RECAP ***********************************************************
dc1a-console.example.net : ok=2    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
```

Verification

IN THIS SECTION

Verify Playbook Execution

Purpose

Verify that the Junos devices were successfully reset to the factory-default configuration.

Action

Access the device through the console port as root. The device should now be in Amnesiac state.

```
Amnesiac <ttyd0>

login:
```

Meaning

The `Amnesiac` prompt is indicative of a device that is booting from a factory-default configuration and that does not have a hostname configured.

RELATED DOCUMENTATION

Use Ansible to Halt, Reboot, or Shut Down Junos Devices | 75

Understanding the Ansible for Junos OS Collections and Modules | 7

Use Junos Snapshot Administrator in Python (JSNAPy) in Ansible Playbooks

SUMMARY

Execute JSNAPy tests as part of an Ansible playbook to capture and audit runtime environment snapshots of Junos devices.

IN THIS SECTION

Junos® Snapshot Administrator in Python (JSNAPy) enables you to capture and audit runtime environment snapshots of your Junos devices. You can capture and verify the configuration and the operational status of a device and verify changes to a device. Juniper Networks provides an Ansible module that you can use to execute JSNAPy tests against Junos devices as part of an Ansible playbook. Table 17 on page 116 outlines the available module.

Table 17: JSNAPy Module

Content Set	Module Name
`juniper.device` collection	`jsnapy`

You must install Junos Snapshot Administrator in Python on the Ansible control node in order to use the `juniper.device.jsnapy` module. For installation instructions and information about creating JSNAPy configuration and test files, see the Junos Snapshot Administrator in Python Documentation.

The following sections discuss how to use the `juniper.device.jsnapy` module in Ansible playbooks.

Module Overview

The `juniper.device.jsnapy` module enables you to execute JSNAPy functions from within an Ansible playbook, including:

- capturing and saving a runtime environment snapshot
- comparing two snapshots
- capturing a snapshot and immediately evaluating it

The module requires specifying the `action` argument and either the `config_file` or the `test_files` argument. The `action` argument specifies the JSNAPy action to perform. Table 18 on page 117 outlines the valid `action` values and the equivalent JSNAPy commands.

Table 18: jsnapy action Argument Values

action Value	Description	Equivalent JSNAPy Command
`check`	Compare two existing snapshots based on the given test cases, or if no test cases are supplied, compare the snapshots node by node.	`jsnapy --check`
`snap_post`	Take snapshots for the commands or RPCs specified in the test files after making changes on the given devices.	`jsnapy --snap`
`snap_pre`	Take snapshots for the commands or RPCs specified in the test files prior to making changes on the given devices.	`jsnapy --snap`
`snapcheck`	Take snapshots of the commands or RPCs specified in the test files and immediately evaluate the snapshots against pre-defined criteria in the test cases.	`jsnapy --snapcheck`

When you execute JSNAPy on the command line, JSNAPy performs the requested action on the hosts specified in the configuration file's `hosts` section. In contrast, the Ansible module executes the requested action on the hosts specified in the Ansible playbook. As a result, the module can either reference a configuration file, ignoring the `hosts` section, or it can directly reference one or more test files.

Thus, in addition to the `action` argument, the `juniper.device.jsnapy` module also requires either the `config_file` argument or the `test_files` argument to specify the JSNAPy configuration file or the JSNAPy test files to use for the given action. Table 19 on page 118 outlines the `config_file` and `test_files` arguments.

Table 19: jsnapy File Arguments

Module argument	Value	Additional Information
`config_file`	Absolute or relative file path to a JSNAPy configuration file.	If the path is relative, the module checks for the configuration file in the following locations and in the order indicated: • Ansible playbook directory • `dir` argument directory, if provided • **/etc/jsnapy/testfiles** directory (only if the `dir` argument is omitted) If the configuration file references test files by using a relative file path, the module first checks for the test files in the playbook directory and then checks for the test files in the default `testfiles` directory, which will vary depending on the JSNAPy release and your environment.
`test_files`	Absolute or relative file path to a JSNAPy test file. This can be a single file path or a list of file paths.	For each test file that specifies a relative path, the module checks for the file in the following locations and in the order indicated: • Ansible playbook directory • `dir` argument directory, if provided • **/etc/jsnapy/testfiles** directory (only if the `dir` argument is omitted)

The `config_file` and `test_files` arguments can take an absolute or relative file path. When using a relative file path, you can optionally include the `dir` module argument to specify the directory in which the files reside. If a `config_file` or `test_files` argument uses a relative file path, the module first checks for the file in the Ansible playbook directory, even if the `dir` argument is present. If the file does not exist in the playbook directory, the module checks in the `dir` argument directory, if it is specified, or in the **/etc/jsnapy/testfiles** directory, if the `dir` argument is omitted. The playbook generates an error message if the file is not found.

It is important to note that when you include the `dir` parameter, the module checks that location only for the specified `config_file` or `test_files` argument. Thus, when you specify a configuration file, the module does not check the `dir` directory for the test files that you specify within the configuration file. If the configuration file references relative paths for the test files, the module checks for the test files only in the playbook directory and in the default `testfiles` directory.

Suppose you have the following JSNAPy configuration file, **jsnapy_config_base_tests.yaml**, which resides in the **~/jsnapy/testfiles** directory and references several JSNAPy test files:

```
user@ansible-cn:~/ansible$ cat ~/jsnapy/testfiles/jsnapy_config_base_tests.yaml
tests:
  - system_util_baseline.yaml
  - verify_ldp_neighbors.yaml
  - verify_ospf_neighbors.yaml
  - verify_bgp_neighbors.yaml
  - test_interface_operstate.yaml
```

The following sample playbook performs the `snap_pre` action for each of the test files in the **jsnapy_config_base_tests.yaml** configuration file. If the configuration file does not exist in the playbook directory, the module checks for the file in the `dir` directory, which in this case is **~/jsnapy/testfiles**. The configuration file uses relative paths for the test files. As a result, the module first checks for the test files in the playbook directory and then checks for the test files in the default **testfiles** directory.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1a
  connection: local
  gather_facts: no

  tasks:
    - name: Take a pre-maintenance snapshot
      juniper.device.jsnapy:
        action: "snap_pre"
        dir: "~/jsnapy/testfiles"
        config_file: "jsnapy_config_base_tests.yaml"
```

Alternatively, the `jsnapy` module can use the `test_files` parameter to specify the individual test files to use. The following playbook executes the same tests as in the previous playbook example. In this case, the module first checks for the test files in the playbook directory and then checks for the test files in the `dir` directory.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1a
  connection: local
  gather_facts: no
```

```
tasks:
  - name: Take a pre-maintenance snapshot
    juniper.device.jsnapy:
      action: "snap_pre"
      dir: "~/jsnapy/testfiles"
      test_files:
        - system_util_baseline.yaml
        - verify_ldp_neighbors.yaml
        - verify_ospf_neighbors.yaml
        - verify_bgp_neighbors.yaml
        - test_interface_operstate.yaml
```

NOTE: Starting in Junos Snapshot Administrator in Python Release 1.3.0, the default location for configuration and test files is **~/jsnapy/testfiles**. However, the default location inside a virtual environment or for earlier releases is **/etc/jsnapy/testfiles**.

The module performs the requested action on the hosts specified in the Ansible playbook, even if the module references a configuration file that includes a `hosts` section. The module reports failed if it encounters an error and fails to execute the JSNAPy tests. It does *not* report failed if one or more of the JSNAPy tests fail. To check the JSNAPy test results, register the module's response, and use the `ansible.builtin.assert` module to verify the expected result in the response.

Junos Snapshot Administrator in Python logs information regarding its operations to the **/var/log/jsnapy/jsnapy.log** file by default. The `juniper.device.jsnapy` module can optionally include the `logfile` argument, which specifies the path to a writable file on the Ansible control node where information for the particular task is logged. Ansible's verbosity level and debug options determine the level of information logged in the file. By default, only messages of severity level WARNING or higher are logged. To log messages equal to or higher than severity level INFO or severity level DEBUG, execute the playbook with the `-v` or `-vv` command-line option, respectively.

When you execute JSNAPy tests in an Ansible playbook, you can save or summarize information for failed JSNAPy tests. For more information, see "Review Failed JSNAPy Tests" on page 126.

Take and Compare Snapshots

JSNAPy enables you to capture runtime environment snapshots of your Junos devices before and after a change and then compare the snapshots to verify the expected changes or identify unexpected issues. The `juniper.device.jsnapy` module enables you to take and compare JSNAPy snapshots as part of an

Ansible playbook. The module saves each snapshot for each host in a separate file in the default JSNAPy snapshot directory using a predetermined filename. For more information about the output files, see "Understanding the jsnapy Module Output" on page 124.

To take baseline snapshots of one or more devices prior to making changes, set the module's `action` argument to `snap_pre`, and specify a configuration file or one or more test files.

The following playbook saves **PRE** snapshots for each device in the Ansible inventory group. The task references the **jsnapy_config_base_tests.yaml** configuration file in the **~/jsnapy/testfiles** directory and logs messages to the **jsnapy_tests.log** file in the playbook directory.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Take a pre-maintenance snapshot
      juniper.device.jsnapy:
        action: "snap_pre"
        dir: "~/jsnapy/testfiles"
        config_file: "jsnapy_config_base_tests.yaml"
        logfile: "jsnapy_tests.log"
```

To take a snapshot of one or more devices after performing changes, set the module's `action` argument to `snap_post`, and specify a configuration file or one or more test files.

The following playbook saves **POST** snapshots for each device in the Ansible inventory group. The task references the same **jsnapy_config_base_tests.yaml** configuration file in the **~/jsnapy/testfiles** directory and logs messages to the **jsnapy_tests.log** file in the playbook directory.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Take a post-maintenance snapshot
      juniper.device.jsnapy:
        action: "snap_post"
```

```
        dir: "~/jsnapy/testfiles"
        config_file: "jsnapy_config_base_tests.yaml"
        logfile: "jsnapy_tests.log"
```

When the `jsnapy` module performs a `snap_pre` action or a `snap_post` action, it saves each snapshot for each host in a separate file using auto-generated filenames that contain a 'PRE' or 'POST' tag, respectively. To compare the `PRE` and `POST` snapshots to quickly verify the updates or identify any issues that might have resulted from the changes, set the module's `action` argument to `check`, and specify the same configuration file or test files that were used to take the snapshots.

When the module performs a `check` action, JSNAPy compares the **PRE** and **POST** snapshots for each test on each device and evaluates them against the criteria defined in the `tests:` section of the test files. If the test files do not define any test cases, JSNAPy instead compares the snapshots node by node. To check the test results, register the module's response, and use the `ansible.builtin.assert` module to verify the expected result in the response.

The following playbook compares the snapshots taken for previously executed `snap_pre` and `snap_post` actions for every device in the Ansible inventory group. The results are evaluated using the criteria in the test files that are referenced in the configuration file. The playbook registers the module's response as '`test_result`' and uses the `ansible.builtin.assert` module to verify that all tests passed on the given device.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Compare PRE and POST snapshots
      juniper.device.jsnapy:
        action: "check"
        dir: "~/jsnapy/testfiles"
        config_file: "jsnapy_config_base_tests.yaml"
        logfile: "jsnapy_tests.log"
      register: test_result

    - name: Verify JSNAPy tests passed
      ansible.builtin.assert:
        that:
          - "test_result.passPercentage == 100"
```

When you run the playbook, the assertions quickly identify which devices failed the tests.

```
user@host:~$ ansible-playbook jsnapy-baseline-check.yaml

PLAY [Junos Snapshot Administrator tests] ***********************************

TASK [Compare PRE and POST snapshots] ***************************************
ok: [dc1a.example.net]
ok: [dc1b.example.net]

TASK [Verify JSNAPy tests passed] *******************************************
ok: [dc1b.example.net] => {
    "changed": false,
    "msg": "All assertions passed"
}
fatal: [dc1a.example.net]: FAILED! => {
    "assertion": "test_result.passPercentage == 100",
    "changed": false,
    "evaluated_to": false,
    "msg": "Assertion failed"
}
    to retry, use: --limit @/home/user/jsnapy-baseline-check.retry

PLAY RECAP ******************************************************************
dc1b.example.net  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
dc1a.example.net  : ok=1    changed=0    unreachable=0    failed=1    skipped=0    rescued=0
ignored=0
```

Perform Snapcheck Operations

JSNAPy enables you to take snapshots for the commands or RPCs specified in JSNAPy test files and immediately evaluate the snapshots against pre-defined criteria in the test cases. The `juniper.device.jsnapy` module enables you to perform a JSNAPy snapcheck operation as part of an Ansible playbook.

To take a snapshot and immediately evaluate it based on the pre-defined set of criteria in the `tests:` section of the test files, set the module's `action` argument to `snapcheck`, and specify a configuration file or

one or more test files. To check the test results, register the module's response, and use the `ansible.builtin.assert` module to verify the expected result in the response.

For example, for each device in the Ansible inventory group, the following playbook saves a separate snapshot for each command or RPC in the test files, registers the module's response, and uses the `ansible.builtin.assert` module to verify that all tests defined in the test files passed on that device.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Take a snapshot and immediately evaluate it
      juniper.device.jsnapy:
        action: "snapcheck"
        dir: "~/jsnapy/testfiles/"
        test_files:
          - "test_interface_status.yaml"
          - "test_bgp_neighbor.yaml"
        logfile: "jsnapy_tests.log"
      register: test_result

    - name: Verify JSNAPy tests passed
      ansible.builtin.assert:
        that:
          - "test_result.passPercentage == 100"
```

Understanding the jsnapy Module Output

When the `juniper.device.jsnapy` module performs a `snap_pre`, `snap_post`, or `snapcheck` action, it automatically saves the snapshots in the JSNAPy **snapshots** directory. The module uses the default JSNAPy directories unless you modify the JSNAPy configuration file (**jsnapy.cfg**) to specify a different location. The module creates a separate file for each command or RPC executed on each device in the Ansible inventory group. Table 20 on page 125 outlines the filenames of the snapshot files for each value of the `action` argument.

NOTE: Starting in Junos Snapshot Administrator in Python Release 1.3.0, the default directories for the JSNAPy test files and snapshots are **~/jsnapy/testfiles** and **~/jsnapy/snapshots**, respectively. However, the default directories inside a virtual environment or for earlier releases are **/etc/jsnapy/testfiles** and **/etc/jsnapy/snapshots**.

Table 20: JSNAPy Output Filenames

`action` value	Output Files
`snap_pre`	***hostname*_PRE_*hash_command.format***
`snap_post`	***hostname*_POST_*hash_command.format***
`snapcheck`	***hostname*_snap_temp_*hash_command.format*** or ***hostname*_PRE_*hash_command.format***

where:

- ***hostname***—Hostname of the device on which the command or RPC is executed.
- **(PRE | POST | snap_temp)**—Tag identifying the action. The `snapcheck` operation uses the `PRE` tag in current releases; in earlier releases the operation uses the `snap_temp` tag.
- ***hash***—Hash generated from `kwargs` for test files that include the `rpc` and `kwargs` keys.

  If test files use the same RPC but include different arguments, and the RPCs are executed on the same host, the hash ensures unique output filenames in those cases. If a test file defines the `command` key or if a test file defines the `rpc` key but does not include the `kwargs` key, the hash is omitted.
- ***command***—Command or RPC executed on the managed device. The module replaces whitespace and special characters in the command or RPC name with underscores (_).
- ***format***—Format of the output, for example, **xml**.

NOTE: The `jsnapy` module differentiates the snapshot filenames for a given action based only on hostname and command or RPC. As a result, if the module takes snapshots on the same device

> for the same action using test files that define the same command or RPC, the module will generate snapshots with the same filename, and the new file will overwrite the old file.

For example, if the module includes `action: "snap_pre"` and references test files that execute the `show chassis fpc` and `show interfaces terse` commands on devices dc1a.example.net and dc1b.example.net, the resulting files are:

```
user@ansible-cn:~$ ls jsnapy/snapshots
dc1a.example.net_PRE_show_chassis_fpc.xml
dc1a.example.net_PRE_show_interfaces_terse.xml
dc1b.example.net_PRE_show_chassis_fpc.xml
dc1b.example.net_PRE_show_interfaces_terse.xml
```

If the module includes `action: "snap_post"` and references a test file that executes the `get-interface-information` RPC with `kwargs` item `interface_name: lo0` on device dc1a.example.net, the resulting file is:

```
dc1a.example.net_POST_r1w59I99HXxC3u0VXXshbw==_get_interface_information.xml
```

In addition to generating the snapshot files, the `jsnapy` module can also return the following keys in the module response:

- `action`—JSNAPy action performed by the module.
- `changed`—Indicates if the device's state changed. Since JSNAPy only reports on state, the value is always `false`.
- `failed`—Indicates if the playbook task failed.
- `msg`—JSNAPy test results.

Review Failed JSNAPy Tests

When you execute JSNAPy tests against Junos devices, you can quickly verify if all JSNAPy tests passed by registering the `jsnapy` module's response and using the `ansible.builtin.assert` module to check that the `passPercentage` is 100. However, if one or more tests fail, it can be difficult to identify and extract the failed tests if the output is extensive.

The `juniper.device.jsnapy` module provides the following options to see failed JSNAPy tests:

- jsnapy callback plugin—Print a summary of failed JSNAPy tests after the playbook output.
- dest_dir module argument—Write the failed JSNAPy tests to files in the specified directory.

The jsnapy callback plugin enables you to easily extract and summarize the information for failed JSNAPy tests. When you enable the jsnapy callback plugin and execute a playbook that includes JSNAPy tests, the plugin summarizes the information for the failed JSNAPy tests after the playbook PLAY RECAP.

The jsnapy callback plugin is disabled by default. To enable the jsnapy callback plugin, add the callback_whitelist = jsnapy statement to the Ansible configuration file.

```
[defaults]
callback_whitelist = jsnapy
```

When you enable the jsnapy callback plugin and run a playbook, the plugin summarizes the failed JSNAPy tests in a human-readable format. For example:

```
...
PLAY RECAP *********************************************************************
qfx10002-01                : ok=3    changed=0    unreachable=0    failed=1
qfx10002-02                : ok=3    changed=0    unreachable=0    failed=1
qfx5100-01                 : ok=1    changed=0    unreachable=0    failed=1

JSNAPy Results for: qfx10002-01 ******************************************
Value of 'peer-state' not 'is-equal' at '//bgp-information/bgp-peer' with {"peer-as": "64502",
"peer-state": "Active", "peer-address": "198.51.100.21"}
Value of 'peer-state' not 'is-equal' at '//bgp-information/bgp-peer' with {"peer-as": "64510",
"peer-state": "Idle", "peer-address": "192.168.0.1"}
Value of 'oper-status' not 'is-equal' at '//interface-information/physical-interface[normalize-
space(admin-status)='up' and logical-interface/address-family/address-family-name ]' with {"oper-
status": "down", "name": "et-0/0/18"}

JSNAPy Results for: qfx10002-02 ******************************************
Value of 'peer-state' not 'is-equal' at '//bgp-information/bgp-peer' with {"peer-as": "64502",
"peer-state": "Active", "peer-address": "198.51.100.21"}
```

Starting in juniper.device Release 1.0.6, the juniper.device.jsnapy module also supports the dest_dir argument. You can include the dest_dir argument for check and snapcheck operations that evaluate snapshots against test criteria. When you perform a check or snapcheck operation and include the dest_dir argument, the module writes each failed JSNAPy test for a given host to a file in the specified output directory.

For example, consider the following playbook:

```
---
- name: Verify BGP
  hosts: bgp_routers
  connection: local
  gather_facts: no

  tasks:
    - name: Execute snapcheck
      juniper.device.jsnapy:
        action: "snapcheck"
        dir: "~/jsnapy/testfiles"
        test_files:
          - "jsnapy_test_file_bgp_states.yaml"
          - "jsnapy_test_file_bgp_summary.yaml"
        logfile: "{{ logfile }}"
        dest_dir: "{{ playbook_dir }}/jsnapy_failed_tests"
      register: snapcheck_result
```

When you execute the playbook, the module generates a file in the `dest_dir` directory for each failed test on the given host. For example, the following files were generated for failed `bgp_neighbor` and `bgp_summary` tests on hosts r1 and r3.

```
user@ansible-cn:~/ansible$ ls jsnapy_failed_tests
r1_bgp_neighbor_False.text  r3_bgp_neighbor_False.text
r1_bgp_summary_False.text   r3_bgp_summary_False.text
```

Example: Use Ansible to Perform a JSNAPy Snapcheck Operation

IN THIS SECTION

The `juniper.device.jsnapy` module enables you to execute JSNAPy tests against Junos devices as part of an Ansible playbook. This examples uses the `jsnapy` module to perform a `snapcheck` action to verify the operational state of Junos devices after applying specific configuration changes.

Requirements

This example uses the following hardware and software components:

- Ansible control node running:
  - Python 3.8 or later
  - Ansible 2.10 or later with the `juniper.device` collection installed
  - Junos PyEZ Release 2.7.1 or later
  - Junos Snapshot Administrator in Python Release 1.3.6 or later

Before executing the Ansible playbook, be sure you have:

- Junos devices with NETCONF over SSH enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the Ansible control node and Junos device
- Existing Ansible inventory file with required hosts defined

Overview

In this example, the Ansible playbook configures BGP peering sessions on three Junos devices and uses the `jsnapy` module to verify that the BGP session is established for each neighbor address. If the playbook verifies that the sessions are established on a device, it confirms the commit for the new configuration. If the playbook does not confirm the commit, the Junos device automatically rolls back to the previously committed configuration. The Ansible project defines the group and host variables for the playbook under the `group_vars` and `host_vars` directories, respectively.

The playbook has two plays. The first play, `Load and commit BGP configuration`, generates and assembles the configuration, loads the configuration on the device, and commits it using a commit confirmed operation. If the configuration is updated, one handler is notified. The play executes the following tasks:

Remove build directory
Deletes the existing build directory for the given device, if present.

Create build directory
Creates a new, empty build directory for the given device.

Build BGP configuration
Uses the `template` module with the Jinja2 template and host variables to render the BGP configuration for the given device and save it to a file in the device's build directory.

Assemble configuration parts
Uses the `assemble` module to assemble the device configuration file from the files in that device's build directory.

In this example, only the BGP configuration file will be present, and thus the resulting configuration file is identical to the BGP configuration file rendered in the previous task. If you later add new tasks to generate additional configuration files from other templates, the `assemble` module will combine all files into a single configuration.

Load and commit config, require confirmation
Loads the configuration onto the Junos device and commits the configuration using a `commit confirmed` operation, which requires explicit confirmation for the commit to become permanent. If this task makes a change to the configuration, it also notifies a handler that pauses playbook execution for a specified amount of time. Pausing playbook execution enables the BGP peers to establish connections before the second play is executed.

If the requested configuration is already present on the device, the `config` module does not load and commit the configuration. In this case, the module returns `changed: false`, and thus does not notify the handler.

The second play, `Verify BGP`, performs a JSNAPy `snapcheck` operation on each device using the tests in the JSNAPy test files. If all tests pass, the play also confirms the commit. The play executes the following tasks:

Execute snapcheck
Performs a JSNAPy `snapcheck` operation, which in this case, validates that the BGP session is established for each of the device's neighbors and that there are no down peers.

In this example, the playbook directly references JSNAPy test files by setting the `test_files` argument equal to the list of JSNAPy test files. The `dir` argument specifies the directory where the test files are stored.

`Confirm commit` Executes a commit check operation, which confirms the previous commit operation, provided that the first playbook play updated the configuration and that all of the JSNAPy tests passed. If the playbook updates the configuration but does not confirm the commit, the Junos device automatically rolls the configuration back to the previously committed configuration.

> **NOTE**: You can confirm the previous commit operation with either a `commit check` or `commit` operation on the device, which corresponds to the `check: true` or `commit: true` argument, respectively, in the `config` module.

`Verify BGP configuration` (Optional) Explicitly indicates whether the JSNAPy tests passed or failed on the given device. This task is not specifically required, but it more easily identifies when the JSNAPy tests fail and on which devices.

Configuration

IN THIS SECTION

Define the Group Variables

Step-by-Step Procedure

To define the group variables:

- In the **group_vars/all** file, define variables for the build directory and for the filenames of the configuration and log files.

```
build_dir: "{{ playbook_dir }}/build_conf/{{ inventory_hostname }}"
junos_conf: "{{ build_dir }}/junos.conf"
logfile:  "junos.log"
```

Define the Jinja2 Template and Host Variables

Define the Jinja2 Template

To create the Jinja2 template that is used to generate the BGP configuration:

1. Create a file named **bgp-template.j2** in the project's playbook directory.
2. Add the BGP configuration template to the file.

```
interfaces {
{% for neighbor in neighbors %}
    {{ neighbor.interface }} {
        unit 0 {
            description "{{ neighbor.name }}";
            family inet {
                address {{ neighbor.local_ip }}/30;
            }
        }
    }
{% endfor %}
    lo0 {
        unit 0 {
            family inet {
                address {{ loopback }}/32;
            }
        }
    }
}
protocols {
    bgp {
        group underlay {
            import bgp-in;
            export bgp-out;
```

```
            type external;
            local-as {{ local_asn }};
            multipath multiple-as;
{% for neighbor in neighbors %}
            neighbor {{ neighbor.peer_ip }} {
                peer-as {{ neighbor.asn }};
            }
{% endfor %}
        }
    }
    lldp {
{% for neighbor in neighbors %}
        interface "{{ neighbor.interface }}";
{% endfor %}
    }
}
routing-options {
    router-id {{ loopback }};
    forwarding-table {
        export bgp-ecmp;
    }
}

policy-options {
    policy-statement bgp-ecmp {
        then {
            load-balance per-packet;
        }
    }
    policy-statement bgp-in {
        then accept;
    }
    policy-statement bgp-out {
        then {
            next-hop self;
            accept;
        }
    }
}
```

Define the Host Variables

To define the host variables that are used with the Jinja2 template to generate the BGP configuration:

1. In the project's **host_vars** directory, create a separate file named ***hostname*.yaml** for each host.
2. Define the variables for host r1 in the **r1.yaml** file.

```
---
loopback: 192.168.0.1
local_asn: 64521
neighbors:
   - interface: ge-0/0/0
     name: to-r2
     asn: 64522
     peer_ip: 198.51.100.2
     local_ip: 198.51.100.1
     peer_loopback: 192.168.0.2
   - interface: ge-0/0/1
     name: to-r3
     asn: 64523
     peer_ip: 198.51.100.6
     local_ip: 198.51.100.5
     peer_loopback: 192.168.0.3
```

3. Define the variables for host r2 in the **r2.yaml** file.

```
---
loopback: 192.168.0.2
local_asn: 64522
neighbors:
   - interface: ge-0/0/0
     name: to-r1
     asn: 64521
     peer_ip: 198.51.100.1
     local_ip: 198.51.100.2
     peer_loopback: 192.168.0.1
   - interface: ge-0/0/1
     name: to-r3
     asn: 64523
     peer_ip: 198.51.100.10
```

```
    local_ip: 198.51.100.9
    peer_loopback: 192.168.0.3
```

4. Define the variables for host r3 in the **r3.yaml** file.

```
---
loopback: 192.168.0.3
local_asn: 64523
neighbors:
   - interface: ge-0/0/0
     name: to-r1
     asn: 64521
     peer_ip: 198.51.100.5
     local_ip: 198.51.100.6
     peer_loopback: 192.168.0.1
   - interface: ge-0/0/1
     name: to-r2
     asn: 64522
     peer_ip: 198.51.100.9
     local_ip: 198.51.100.10
     peer_loopback: 192.168.0.2
```

Create the JSNAPy Test Files

Step-by-Step Procedure

The `jsnapy` module references JSNAPy test files in the **~/jsnapy/testfiles** directory. To create the JSNAPy test files:

1. Create the **jsnapy_test_file_bgp_states.yaml** file, which executes the `show bgp neighbor` command and tests that the BGP peer state is established.

```
bgp_neighbor:
  - command: show bgp neighbor
  - ignore-null: true
  - iterate:
      xpath: '//bgp-peer'
      id: './peer-address'
      tests:
          # Check if peers are in the established state
```

```
        - is-equal: peer-state, Established
          err: "Test Failed!! peer <{{post['peer-address']}}> state is not Established, it is
<{{post['peer-states']}}>"
          info: "Test succeeded!! peer <{{post['peer-address']}}> state is <{{post['peer-
state']}}>"
```

2. Create the **jsnapy_test_file_bgp_summary.yaml** file, which executes the `show bgp summary` command and asserts that the BGP down peers count must be 0.

```
bgp_summary:
  - command: show bgp summary
  - item:
      xpath: '/bgp-information'
      tests:
        - is-equal: down-peer-count, 0
          err: "Test Failed!! down-peer-count is not equal to 0. It is equal to <{{post['down-
peer-count']}}>"
          info: "Test succeeded!! down-peer-count is equal to <{{post['down-peer-count']}}>"
```

Create the Ansible Playbook

Define the First Play to Configure the Device

To create the first play, which renders the configuration, loads it on the device, and commits the configuration as a commit confirmed operation:

1. Include the boilerplate for the playbook and the first play, which executes the modules locally.

```
---
- name: Load and commit BGP configuration
  hosts: bgp_routers
  connection: local
  gather_facts: no
```

2. Create the tasks that replace the existing build directory with an empty directory, which will store the new configuration files.

```
  tasks:
    - name: Remove build directory
```

```
    file:
      path: "{{ build_dir }}"
      state: absent

  - name: Create build directory
    file:
      path: "{{ build_dir }}"
      state: directory
```

3. Create the task that renders the BGP configuration from the Jinja2 template file and host variables and stores it in the **bgp.conf** file in the build directory for that host.

```
  - name: Build BGP configuration
    template:
      src: "{{ playbook_dir }}/bgp-template.j2"
      dest: "{{ build_dir }}/bgp.conf"
```

4. Create a task to assemble the configuration files in the build directory into the final **junos.conf** configuration file.

```
  - name: Assemble configuration parts
    assemble:
      src: "{{ build_dir }}"
      dest: "{{ junos_conf }}"
```

5. Create the task that loads the configuration on the device, performs a commit operation that requires confirmation, and notifies the given handler, provided the configuration was changed.

```
  - name: Load and commit config, require confirmation
    juniper.device.config:
      load: "merge"
      format: "text"
      src: "{{ junos_conf }}"
      confirm: 5
      comment: "config by Ansible"
      logfile: "{{ logfile }}"
    register: config_result

    # Notify handler, only if configuration changes.
```

```
      notify:
        - Waiting for BGP peers to establish connections
```

6. Create a handler that pauses playbook execution if the device configuration is updated. Set the pause time to an appropriate value for your environment.

```
  handlers:
    - name: Waiting for BGP peers to establish connections
      pause: seconds=60
```

Define the Second Play to Perform JSNAPy Operations

To create the second play, which performs a JSNAPy snapcheck operation and confirms the committed configuration, provided that the configuration changed and the JSNAPy tests passed:

1. Include the boilerplate for the second play, which executes the modules locally.

```
- name: Verify BGP
  hosts: bgp_routers
  connection: local
  gather_facts: no
```

2. Create a task to perform a JSNAPy snapcheck operation based on the tests in the given JSNAPy test files, and register the module's response.

```
  tasks:
    - name: Execute snapcheck
      juniper.device.jsnapy:
        action: "snapcheck"
        dir: "~/jsnapy/testfiles"
        test_files:
          - "jsnapy_test_file_bgp_states.yaml"
          - "jsnapy_test_file_bgp_summary.yaml"
        logfile: "{{ logfile }}"
      register: snapcheck_result
```

3. Create the task to confirm the commit provided that the given conditions are met.

```
    # Confirm commit only if configuration changed and JSNAPy tests pass
    - name: Confirm commit
      juniper.device.config:
        check: true
        commit: false
        diff: false
        logfile: "{{ logfile }}"
      when:
        - config_result.changed
        - "snapcheck_result.passPercentage == 100"
```

4. (Optional) Create a task that uses the `ansible.builtin.assert` module to assert that the JSNAPy tests passed.

```
    - name: Verify BGP configuration
      ansible.builtin.assert:
        that:
          - "snapcheck_result.passPercentage == 100"
        msg: "JSNAPy test on {{ inventory_hostname }} failed"
```

Results

On the Ansible control node, review the completed playbook. If the playbook does not display the intended code, repeat the instructions in this section to correct the playbook.

```
---
- name: Load and commit BGP configuration
  hosts: bgp_routers
  connection: local
  gather_facts: no

  tasks:
    - name: Remove build directory
      file:
        path: "{{ build_dir }}"
        state: absent
```

```
    - name: Create build directory
      file:
        path: "{{ build_dir }}"
        state: directory

    - name: Build BGP configuration
      template:
        src: "{{ playbook_dir }}/bgp-template.j2"
        dest: "{{ build_dir }}/bgp.conf"

    - name: Assemble configuration parts
      assemble:
        src: "{{ build_dir }}"
        dest: "{{ junos_conf }}"

    - name: Load and commit config, require confirmation
      juniper.device.config:
        load: "merge"
        format: "text"
        src: "{{ junos_conf }}"
        confirm: 5
        comment: "config by Ansible"
        logfile: "{{ logfile }}"
      register: config_result

      # Notify handler, only if configuration changes.
      notify:
        - Waiting for BGP peers to establish connections

  handlers:
    - name: Waiting for BGP peers to establish connections
      pause: seconds=60


- name: Verify BGP
  hosts: bgp_routers
  connection: local
  gather_facts: no

  tasks:
    - name: Execute snapcheck
      juniper.device.jsnapy:
        action: "snapcheck"
```

```
      dir: "~/jsnapy/testfiles"
      test_files:
        - "jsnapy_test_file_bgp_states.yaml"
        - "jsnapy_test_file_bgp_summary.yaml"
      logfile: "{{ logfile }}"
    register: snapcheck_result

  # Confirm commit only if configuration changed and JSNAPy tests pass
  - name: Confirm commit
    juniper.device.config:
      check: true
      commit: false
      diff: false
      logfile: "{{ logfile }}"
    when:
      - config_result.changed
      - "snapcheck_result.passPercentage == 100"

  - name: Verify BGP configuration
    ansible.builtin.assert:
      that:
        - "snapcheck_result.passPercentage == 100"
      msg: "JSNAPy test on {{ inventory_hostname }} failed"
```

Execute the Playbook

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
user@ansible-cn:~/ansible$ ansible-playbook ansible-pb-bgp-configuration.yaml
PLAY [Load and commit BGP configuration] ************************************

TASK [Remove build directory] ***********************************************
changed: [r1]
changed: [r2]
changed: [r3]

TASK [Create build directory] ***********************************************
changed: [r1]
changed: [r2]
```

```
changed: [r3]

TASK [Build BGP configuration] ************************************************
changed: [r2]
changed: [r1]
changed: [r3]

TASK [Assemble configuration parts] *******************************************
changed: [r3]
changed: [r2]
changed: [r1]

TASK [Load and commit config, require confirmation] ***************************
changed: [r2]
changed: [r1]
changed: [r3]

RUNNING HANDLER [Waiting for BGP peers to establish connections] **************
Pausing for 60 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [r3]
```

```
PLAY [Verify BGP] *************************************************************

TASK [Execute snapcheck] ******************************************************
ok: [r2]
ok: [r1]
ok: [r3]

TASK [Confirm commit] *********************************************************
ok: [r2]
ok: [r1]
ok: [r3]

TASK [Verify BGP configuration] ***********************************************
ok: [r1] => {
    "changed": false,
    "msg": "All assertions passed"
}
ok: [r2] => {
    "changed": false,
```

```
    "msg": "All assertions passed"
}
ok: [r3] => {
    "changed": false,
    "msg": "All assertions passed"
}

PLAY RECAP *********************************************************************
r1                         : ok=8    changed=5    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
r2                         : ok=8    changed=5    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
r3                         : ok=9    changed=5    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Verification

IN THIS SECTION

Verify the BGP Neighbors

Purpose

Verify that the BGP session is established for each neighbor address.

The JSNAPy test files test that the BGP session is established for each neighbor address and that the there are no down peers. The `Verify BGP configuration` task output enables you to quickly verify that the given device passed all JSNAPy tests. If the JSNAPy `passPercentage` is equal to 100 percent, the task includes `"msg": "All assertions passed"` in the task output.

Action

Review the `Verify BGP configuration` task output, and verify that each device returns the `All assertions passed` message.

```
TASK [Verify BGP configuration] ********************************************
ok: [r1] => {
    "changed": false,
    "msg": "All assertions passed"
}
ok: [r2] => {
    "changed": false,
    "msg": "All assertions passed"
}
ok: [r3] => {
    "changed": false,
    "msg": "All assertions passed"
}
```

Meaning

The `All assertions passed` message indicates that the BGP sessions are successfully established on the devices.

Troubleshoot Ansible Playbook Errors

IN THIS SECTION

Troubleshoot Configuration Load Errors

Problem

The Ansible playbook generates a `ConfigLoadError` error indicating that it failed to load the configuration on the device because of a syntax error.

```
fatal: [r1]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: error, bad_element: protocol, message: error: syntax error\nerror:
error recovery ignores input until this point)"}
```

Solution

The playbook renders the Junos OS configuration by using the Jinja2 template and the host variables defined for that device in the **host_vars** directory. The playbook generates a syntax error when the Jinja2 template produces an invalid configuration. To correct this error, update the Jinja2 template to correct the element identified by the `bad_element` key in the error message.

Troubleshoot Failed JSNAPy Tests

Problem

The `Verify BGP configuration` task output indicates that the assertion failed, because the JSNAPy `passPercentage` was not equal to 100 percent.

```
TASK [Verify BGP configuration] ************************************************************
fatal: [r1]: FAILED! => {
    "assertion": "snapcheck_result.passPercentage == 100",
    "changed": false,
    "evaluated_to": false,
    "msg": "JSNAPy test on r1 failed"
}
```

The assertion fails when the device has not established the BGP session with its neighbor or the session goes down. If the assertion fails, and the configuration for that device was updated in the first play, the playbook does not confirm the commit for the new configuration on the device, and the device rolls the configuration back to the previously committed configuration.

Solution

The JSNAPy tests might fail if the `snapcheck` operation is taken before the peers can establish the session or because the BGP neighbors are not configured correctly. If the playbook output indicates that the configuration was successfully loaded and committed on the device, try increasing the handler's pause interval to a suitable value for your environment and rerun the playbook.

```
handlers:
  - name: Waiting for BGP peers to establish connections
    pause: seconds=75
```

If the tests still fail, verify that the Jinja2 template and the host variables for each device contain the correct data and that the resulting configuration for each device is correct.

Troubleshoot Failed Commit Confirmations

Problem

The configuration was not confirmed on one or more devices.

```
TASK [Confirm commit] ********************************************************************
skipping: [r2]
skipping: [r2]
skipping: [r3]
```

Solution

The playbook only confirms the configuration if it changed and the JSNAPy tests pass. If the `Load and commit config, require confirmation` task output indicates that the configuration did not change, the playbook does not execute the task to confirm the commit. If the configuration changed but was not confirmed, then the JSNAPy tests failed. The JSNAPy tests might fail if the BGP neighbors are not configured correctly or if the playbook does not provide enough time between the plays for the devices to establish the BGP session. For more information, see "Troubleshoot Failed JSNAPy Tests" on page 145.

CHAPTER 6

Use Ansible to Manage the Configuration

Use Ansible to Retrieve or Compare Junos OS Configurations

SUMMARY

Use the Juniper Networks Ansible modules to retrieve or compare configurations on Junos devices.

IN THIS SECTION

Juniper Networks provides an Ansible module that enables you to manage the configuration on Junos devices. Table 21 on page 148 outlines the available module, which you can use to retrieve or compare Junos device configurations.

Table 21: Module to Retrieve or Compare Configurations

Content Set	Module Name
`juniper.device` collection	`config`

You can use the `juniper.device.config` module to request the complete configuration or selected portions of the configuration for both the native Junos OS configuration as well as for configuration data corresponding to third-party YANG data models that have been added to the device. To retrieve the configuration from a Junos device, execute the `config` module with the `retrieve` parameter. The module's response includes the configuration in text format in the `config` and `config_lines` keys, unless the

return_output option is set to false. You can also compare the active configuration with a previously committed configuration.

The following sections discuss how to use the module to retrieve or compare configurations.

How to Specify the Source Database for the Configuration Data

When you use the juniper.device.config module to retrieve the configuration, you must include the retrieve parameter in the module's argument list. The retrieve parameter specifies the configuration database from which to retrieve the data. You can set retrieve to the following values to return configuration data from the respective database:

- retrieve: 'candidate'—Retrieve data from the candidate configuration.
- retrieve: 'committed'—Retrieve data from the committed configuration.

Committed Configuration Database

The following playbook retrieves the complete committed configuration in text format for each device in the inventory group:

```
---
- name: "Get Junos OS configuration"
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: "Get committed configuration"
      juniper.device.config:
        retrieve: "committed"
      register: response
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

Candidate Configuration Database

The following playbook retrieves the complete candidate configuration in text format for each device in the inventory group. The module returns an error if the database is locked or modified.

```
---
- name: "Get Junos OS configuration"
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: "Get candidate configuration"
      juniper.device.config:
        retrieve: "candidate"
      register: response
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

How to Specify the Scope of the Configuration Data to Return

In addition to retrieving the complete Junos OS configuration, you can retrieve specific portions of the configuration by including the `config` module's `filter` parameter. The `filter` parameter's value is a string containing the subtree filter that selects the configuration statements to return. The subtree filter returns the configuration data that matches the selection criteria. If you request multiple hierarchies, the value of `filter` must represent all levels of the configuration hierarchy starting at the root (represented by the `<configuration>` element) down to each element to display.

The following playbook retrieves and prints the configuration at the `[edit interfaces]` and `[edit protocols]` hierarchy levels in the committed configuration database for each device:

```
---
- name: "Get Junos OS configuration hierarchies"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Get selected configuration hierarchies"
      juniper.device.config:
```

```
        retrieve: "committed"
        filter: "<configuration><interfaces/><protocols/></configuration>"
      register: response
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

The following playbook retrieves and prints the configuration for the ge-1/0/1 interface:

```
---
- name: "Get Junos OS configuration hierarchies"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Get selected configuration hierarchies"
      juniper.device.config:
        retrieve: "committed"
        filter: "<interfaces><interface>
                  <name>ge-1/0/1</name></interface></interfaces>"
      register: response
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

The following playbook retrieves and prints the configuration committed at the `[edit system services]` hierarchy level:

```
---
- name: "Get Junos OS configuration hierarchies."
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Get selected configuration hierarchies"
      juniper.device.config:
        retrieve: "committed"
        filter: "system/services"
      register: response
```

```
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

How to Specify the Format of the Configuration Data to Return

When you use the `juniper.device.config` module to retrieve the configuration, the module invokes the Junos XML protocol `<get-configuration>` operation, which can return configuration data in different formats. By default, the module returns configuration data as formatted text. The text format uses newlines, tabs and other white space, braces, and square brackets to indicate the hierarchical relationships between the statements.

To specify the format in which to return the configuration data, set the `config` module's `format` parameter equal to the desired format. Acceptable values include:

- `'json'`—JavaScript Object Notation (JSON)
- `'set'`—Junos OS `set` commands
- `'text'`—Formatted text (default)
- `'xml'`—Junos XML elements

In the playbook output, the `config` and `config_lines` keys contain the configuration in the requested format. If you request Junos XML or JSON format, the `config_parsed` key contains the equivalent configuration in JSON format.

The following playbook retrieves the complete committed configuration for each device in the inventory group in XML format:

```
---
- name: "Get Junos OS configuration."
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: "Get configuration in XML format"
      juniper.device.config:
        retrieve: "committed"
        format: "xml"
      register: response
```

```
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

How to Retrieve Configuration Data for Third-Party YANG Data Models

You can load standardized or custom YANG modules on Junos devices to add data models that are not natively supported by Junos OS but can be supported by translation. You configure nonnative data models in the candidate configuration using the syntax defined for those models. When you commit the configuration, the data model's translation scripts translate that data and commit the corresponding Junos OS configuration as a transient change in the checkout configuration.

The candidate and active configurations contain the configuration data for nonnative YANG data models in the syntax defined by those models. You can use the `juniper.device.config` module to retrieve configuration data for standard (IETF, OpenConfig) and custom YANG data models in addition to retrieving the native Junos OS configuration. By default, configuration data for third-party YANG data models is not included in the module's reply.

To retrieve configuration data that is defined by a nonnative YANG data model in addition to retrieving the Junos OS configuration, execute the module with the `model` parameter, and include the `namespace` parameter when appropriate. The `model` argument takes one of the following values:

- `custom`—Retrieve configuration data that is defined by custom YANG data models. You must include the `namespace` argument when retrieving data for custom YANG data models.
- `ietf`—Retrieve configuration data that is defined by IETF YANG data models.
- `openconfig`—Retrieve configuration data that is defined by OpenConfig YANG data models.
- `True`—Retrieve all configuration data, including the complete Junos OS configuration and data from any YANG data models.

If the `model` argument specifies `ietf` or `openconfig`, the module automatically uses the appropriate namespace. If you specify `model: "custom"` to retrieve data for a custom YANG data model, you must also include the `namespace` argument with the corresponding namespace.

If you include the `model` argument with the value `custom`, `ietf`, or `openconfig` and also include the `filter` argument to return a specific XML subtree, Junos OS only returns the matching hierarchy from the nonnative data model. If the Junos OS configuration contains a hierarchy of the same name, for example "interfaces", it is not included in the reply. The `filter` option is not supported when using `model: "True"`.

When you use the `config` module to retrieve nonnative configuration data, you can only specify the format of the returned data if you also include the `filter` parameter. If you omit the `filter` parameter, you must specify `format: "xml"`.

The following playbook retrieves the OpenConfig `interfaces` configuration hierarchy from the committed configuration. If you omit the `filter` argument, the RPC returns the complete Junos OS and OpenConfig configurations.

```
---
- name: "Retrieve OpenConfig configuration"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Retrieve the OpenConfig interfaces configuration"
      juniper.device.config:
        retrieve: "committed"
        model: "openconfig"
        filter: "interfaces"
        format: "xml"
      register: response

    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

The following task retrieves the `l2vpn` configuration hierarchy from the committed configuration for a custom YANG data model with the given namespace:

```
  tasks:
    - name: "Retrieve custom configuration"
      juniper.device.config:
        retrieve: "committed"
        model: "custom"
        filter: "l2vpn"
        remove_ns: False
        namespace: "http://yang.juniper.net/customyang/l2vpn"
        format: "xml"
      register: response
```

The following task retrieves the complete Junos OS committed configuration as well as the configuration data for other YANG data models that have been added to the device:

```
tasks:
  - name: "Retrieve Junos OS and all third-party configuration data"
    juniper.device.config:
      retrieve: "committed"
      model: "True"
      format: "xml"
    register: response
```

How to Specify Options That Do Not Have an Equivalent Module Argument

When you use the `juniper.device.config` module to retrieve the configuration, the module invokes the Junos XML protocol `<get-configuration>` operation. The module supports explicit arguments for many of the `<get-configuration>` attributes, for example, the `format` attribute. The module also supports the `options` argument, which enables you to include any additional `<get-configuration>` attributes that do not have an equivalent module argument. The `options` argument takes a dictionary of key/value pairs of any attributes supported by the `<get-configuration>` operation.

For the complete list of attributes supported by the Junos XML protocol `<get-configuration>` operation, see <get-configuration>.

For example, the `config` module retrieves data from the pre-inheritance configuration, in which the `<groups>`, `<apply-groups>`, `<apply-groups-except>`, and `<interface-range>` tags are separate elements in the configuration output. To retrieve data from the post-inheritance configuration, which displays statements that are inherited from user-defined groups and ranges as children of the inheriting statements, you can include the `options` argument with `inherit: "inherit"`.

The following playbook retrieves the configuration data at the `[edit system services]` hierarchy level from the post-inheritance committed configuration. In this case, if the configuration also contains statements configured at the `[edit groups global system services]` hierarchy level, those statements would be inherited under `[edit system services]` in the post-inheritance configuration and returned in the retrieved configuration data.

```
---
- name: "Get Junos OS configuration hierarchies"
  hosts: dc1
```

```
  connection: local
  gather_facts: no

  tasks:
    - name: "Get selected hierarchy from the post-inheritance configuration"
      juniper.device.config:
        retrieve: "committed"
        filter: "system/services"
        options:
          inherit: "inherit"
      register: response
    - name: "Print result"
      ansible.builtin.debug:
        var: response
```

How to Save Configuration Data To a File

When you use the `juniper.device.config` module to retrieve the configuration, you can save the returned configuration data in a file on the local Ansible control node by including the module's `dest_dir` or `dest` parameter. The `dest_dir` option just specifies a directory, and the `dest` option can specify both a path and a filename. If an output file already exists with the target name, the module overwrites the file.

To specify the directory on the local Ansible control node where the retrieved configurations are saved, include the `dest_dir` argument, and define the path to the target directory. The configuration for each device is stored in a separate file named ***hostname*.config**.

The following playbook retrieves the committed configuration from all devices in the inventory group. The playbook saves each device configuration to a separate file in the **configs** directory under the playbook directory on the Ansible control node.

```
---
- name: "Get Junos OS configuration"
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: "Save configuration to a file"
      juniper.device.config:
```

```
        retrieve: "committed"
        dest_dir: "{{ playbook_dir }}/configs"
```

To specify the path and filename for the output files, include the `dest` argument, and define the absolute or relative path of the file. If you include the `dest` argument, but omit the directory, the files are saved in the playbook directory. If you retrieve the configuration for multiple devices, the `dest` argument must include a variable such as `{{ inventory_hostname }}` to differentiate the filename for each device. If you do not differentiate the filenames, the configuration file for each device will overwrite the configuration file of the other devices.

The following playbook retrieves the `[edit system services]` hierarchy from the committed configuration database on all devices in the inventory group. The playbook saves each device configuration to a separate file in the playbook directory on the Ansible control node. Each file is uniquely identified by the device hostname.

```
---
- name: "Get Junos OS configuration"
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: "Get selected configuration hierarchies and save to file"
      juniper.device.config:
        retrieve: "committed"
        filter: "system/services"
        dest: "{{ inventory_hostname }}-system-services-config"
```

If you are saving the configuration data to files and do not want to duplicate the configuration data in the module's response, you can you can optionally include `return_output: false` in the module's argument list. Setting `return_output` to `false` causes the module to omit the `config`, `config_lines`, and `config_parsed` keys in its response. Doing this might be necessary if the device returns a significant amount of configuration data.

How to Compare the Active Configuration to a Previous Configuration

The `juniper.device.config` module enables you to compare the active configuration to a previously committed configuration, or rollback configuration. To compare the active configuration to a previous configuration, include the following module arguments:

```
juniper.device.config:
  diff: true
  rollback: id
  check: false
  commit: false
```

By default, when you include the `rollback: id` argument, the module rolls back the configuration, performs a commit check, and commits the changes. You must include the `commit: false` argument to only compare the configurations and prevent the module from loading and committing the rollback configuration. Including the `check: false` argument prevents the unnecessary commit check operation.

The module returns the `diff` and `diff_lines` keys. The keys contain the configuration differences between the active and previous configuration in *diff* or patch format.

- `diff`— dictionary that contains a single key named `prepared` and its value, which is a single multi-line string containing the differences.
- `diff_lines`—list of single line strings containing the differences.

To save the differences to a file on the local Ansible control node, include the `diffs_file` argument, and define the absolute or relative path of the output file. If you include the `diffs_file` argument but omit the directory, the files are saved in the playbook directory. If you compare the configurations on multiple devices, the `diffs_file` argument must include a variable such as `{{ inventory_hostname }}` to differentiate the filename for each device. If you do not differentiate the filenames, the output file for each device will overwrite the output file of the other devices.

The following playbook prompts for the rollback ID of a previously committed configuration. The playbook then compares the committed configuration to the specified rollback configuration, saves the comparison to a uniquely-named file, and also prints the response to standard output.

```
---
- name: "Compare configurations"
  hosts: dc1
  connection: local
  gather_facts: no
```

```
  vars_prompt:
    - name: "ROLLBACK"
      prompt: "Rollback ID to compare with active configuration"
      private: no

  tasks:
    - name: "Compare active to previous configuration"
      juniper.device.config:
        diff: true
        rollback: "{{ ROLLBACK }}"
        check: false
        commit: false
        diffs_file: "{{ inventory_hostname }}-diff-rollback-{{ ROLLBACK }}"
      register: response
    - name: "Print diff"
      ansible.builtin.debug:
        var: response
```

```
user@ansible-cn:~$ ansible-playbook configuration-compare-to-rollback.yaml
Rollback ID to compare with active configuration: 2

PLAY [Compare configurations] **************************************************

TASK [Compare active to previous configuration] ********************************
changed: [dc1a.example.net]

TASK [Print diff] **************************************************************
ok: [dc1a.example.net] => {
    "response": {
        "changed": true,
        "diff": {
            "prepared": "\n[edit system services]\n-    netconf {\n-        ssh;\n-    }\n"
        },
        "diff_lines": [
            "",
            "[edit system services]",
            "-    netconf {",
            "-        ssh;",
            "-    }"
        ],
        "failed": false,
```

```
        "msg": "Configuration has been: opened, rolled back, diffed, closed."
    }
}

PLAY RECAP *********************************************************************
dc1a.example.net  : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

```
user@ansible-cn:~$ cat dc1a.example.net-diff-rollback-2
[edit system services]
-    netconf {
-        ssh;
-    }
```

RELATED DOCUMENTATION

Use Ansible to Configure Junos Devices | 160

Understanding the Ansible for Junos OS Collections and Modules | 7

Use Ansible to Configure Junos Devices

SUMMARY

Use the Juniper Networks Ansible modules to manage the configuration on Junos devices.

IN THIS SECTION

Juniper Networks provides an Ansible module that enables you to configure Junos devices. Table 22 on page 161 outlines the available module. The user account that is used to make configuration changes must have permissions to change the relevant portions of the configuration on each device.

Table 22: Module to Manage the Configuration

Content Set	Module Name
`juniper.device` collection	`config`

The following sections discuss how to use the module to modify and commit the configuration on Junos devices.

Module Overview

The `juniper.device.config` module enables you to perform the following operations on Junos devices:

- Load configuration data
- Commit the configuration
- Roll back the configuration
- Load the rescue configuration

To modify the configuration, the module's argument list must include either the `load` parameter to load new configuration data or the `rollback` parameter to revert to the rescue configuration or a previously committed configuration. The basic process for making configuration changes is to lock the configuration, load the configuration changes, commit the configuration to make it active, and then unlock the configuration.

By default, the `config` module makes changes to the candidate configuration database using `configure exclusive` mode, which automatically locks and unlocks the candidate global configuration. You can also specify a different configuration mode. For example, you can make changes to a private copy of the candidate configuration or to the ephemeral configuration database. For more information about specifying the configuration mode, see "How to Specify the Configuration Mode" on page 163.

When loading new configuration data, in addition to specifying the configuration mode, you can also specify the load operation and the source and format of the changes.

- Load operation—The load operation determines how the configuration data is loaded into the selected configuration database. The function supports many of the same load operations that are available in the Junos OS CLI. For more information, see "How to Specify the Load Action" on page 165.

- Format—You can configure Junos devices using one of the standard, supported formats. You can provide configuration data or Jinja2 templates as text, Junos XML elements, Junos OS `set` commands, or JSON. For information about specifying the format of the configuration data, see "How to Specify the Format of the Configuration Data to Load" on page 166.

- Configuration data source—You can load configuration data from a list of strings, a file on the local Ansible control node, a Jinja2 template, or a URL reachable from the client device by including the `lines`, `src`, `template`, or `url` parameter, respectively. For more information about specifying the source of the configuration data, see the following sections:

  - "How to Load Configuration Data as Strings" on page 167
  - "How to Load Configuration Data from a Local or Remote File" on page 169
  - "How to Load Configuration Data Using a Jinja2 Template" on page 171

The `config` module also enables you to load and commit the rescue configuration or roll the configuration back to a previously committed configuration. To load the rescue configuration or a previously committed configuration, you must include the `rollback` module argument. For more information, see the following sections:

- "How to Load the Rescue Configuration" on page 174
- "How to Roll Back the Configuration" on page 175

After modifying the configuration, you must commit the configuration to make it the active configuration on the device. By default, the `config` module commits the changes to the configuration. To

alter this behavior or supply additional commit options, see "How to Commit the Configuration" on page 176.

By default, when the `config` module includes the `load` or `rollback` arguments to change the configuration, the module automatically returns the configuration changes in *diff* or patch format in the module's response. The differences are returned in the `diff` and `diff_lines` variables. To prevent the module from calculating and returning the differences, set the `diff` module argument to `false`.

How to Specify the Configuration Mode

You can specify the configuration mode to use when modifying the device configuration. To specify the configuration mode in your task, include the `config` module's `config_mode` parameter. Supported configuration modes include:

- `batch`
- `dynamic`
- `ephemeral`
- `exclusive`
- `private`

By default, the `juniper.device.config` module makes changes to the candidate configuration database using `configure exclusive` mode. Configure exclusive mode locks the candidate global configuration (also known as the *shared configuration database*) for as long as the module requires to make the requested changes to the configuration. Locking the database prevents other users from modifying or committing changes to the database until the lock is released.

The following examples show how to configure a private copy of the candidate configuration and how to configure the ephemeral database.

Example: `config_mode: "private"`

The following playbook uses `private` configuration mode to modify a private copy of the candidate configuration:

```
---
- name: "Configure Device"
  hosts: dc1
  connection: local
```

```
  gather_facts: no

  tasks:
    - name: "Configure op script"
      juniper.device.config:
        config_mode: "private"
        load: "set"
        lines:
          - "set system scripts op file bgp.slax"
      register: response
    - name: "Print the config changes"
      ansible.builtin.debug:
        var: response.diff_lines
```

```
user@ansible-cn:~/ansible$ ansible-playbook configure-script.yaml
PLAY [Configure Device] ***************************************************

TASK [Configure op script] ************************************************
changed: [dc1a.example.net]

TASK [Print the config changes] *******************************************
ok: [dc1a.example.net] => {
    "response.diff_lines": [
        "",
        "[edit system scripts op]",
        "+     file bgp.slax;"
    ]
}

PLAY RECAP ****************************************************************
dc1a.example.net  : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

Configure the Ephemeral Database

You can use the `juniper.device.config` module to update the *ephemeral configuration database* on devices that support this database. The ephemeral database is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on Junos devices.

To open and configure the default instance of the ephemeral configuration database, include the `config_mode: "ephemeral"` argument. For example:

```
---
- name: "Configure ephemeral database"
  hosts: dc1a
  connection: local
  gather_facts: no

  tasks:
    - name: "Configure the default ephemeral database"
      juniper.device.config:
        config_mode: "ephemeral"
        load: "set"
        lines:
          - "set protocols mpls label-switched-path to-hastings to 192.0.2.1"
```

To open and configure an existing user-defined instance of the ephemeral configuration database, include the `config_mode: "ephemeral"` argument, and set the `ephemeral_instance` argument to the name of the instance.

```
  tasks:
    - name: "Configure a user-defined ephemeral instance"
      juniper.device.config:
        config_mode: "ephemeral"
        ephemeral_instance: "eph1"
        load: "set"
        lines:
          - "set protocols mpls label-switched-path to-hastings to 192.0.2.2"
```

How to Specify the Load Action

The `juniper.device.config` module supports loading configuration changes using many of the same load operations supported in the Junos OS CLI. You specify the load operation by including the `load` parameter in the module's argument list and setting it to the value of the corresponding load operation. Table 23 on page 166 summarizes the parameter settings required for each type of load operation.

Table 23: Parameters for Specifying the Load Operation

Load Operation	load Argument	Description
`load merge`	`load: "merge"`	Merge the loaded configuration with the existing configuration.
`load override`	`load: "override"`	Replace the entire configuration with the loaded configuration.
`load patch`	`load: "patch"`	Load configuration data from a patch file.
`load replace`	`load "replace"`	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the `replace:` tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
`load set`	`load: "set"`	Load configuration data that is in `set` format. The configuration data is loaded line by line and can contain configuration mode commands such as `set`, `delete`, and `deactivate`.
`load update`	`load: "update"`	Compare the complete loaded configuration against the existing configuration. Each configuration element that is different in the loaded configuration replaces its corresponding element in the existing configuration. During the commit operation, only system processes that are affected by changed configuration elements parse the new configuration.

How to Specify the Format of the Configuration Data to Load

The `juniper.device.config` module enables you to configure Junos devices using one of the standard, supported formats. You can provide configuration data as strings or files. Files can contain either configuration data or Jinja2 templates. When providing configuration data within a string, file, or Jinja2 template, supported formats for the data include text, Junos XML elements, Junos OS `set` commands, and JSON.

The `config` module attempts to auto-detect the format of configuration data that you supply as strings within the `lines` argument. However, you can explicitly specify the format for strings by including the `format` argument. When you provide configuration data in a file or Jinja2 template, you must specify the

format of the data either by adding the appropriate extension to the file or by including the `format` argument.

Table 24 on page 167 summarizes the supported formats for the configuration data and the corresponding value for the file extension and `format` parameter. If you include the `format` argument, it overrides both the auto-detect format for strings and the format indicated by a file extension.

Table 24: Specifying the Format for Configuration Data

Configuration Data Format	File Extension	format Parameter
CLI configuration statements (text)	**.conf**	`"text"`
JavaScript Object Notation (JSON)	**.json**	`"json"`
Junos OS `set` commands	**.set**	`"set"`
Junos XML elements	**.xml**	`"xml"`

NOTE: When you set the module's `load` argument to `'override'` or `'update'`, you cannot use the Junos OS `set` command format.

How to Load Configuration Data as Strings

The `juniper.device.config` module enables you to load configuration data from a list of strings. To load configuration data as strings, include the appropriate `load` argument and the `lines` argument. The `lines` argument takes a list of strings containing the configuration data to load.

The module attempts to auto-detect the format of the `lines` configuration data. However, you can explicitly specify the format by including the `format` argument. For information about specifying the format, see "How to Specify the Format of the Configuration Data to Load" on page 166. If you include the `format` argument, it overrides the auto-detected format.

The following playbook configures and commits two op scripts. In this case, the `load` argument has the value `'set'`, because the configuration data in `lines` uses Junos OS `set` statement format.

```
---
- name: "Load and commit configuration"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Load configuration data using strings and commit"
      juniper.device.config:
        load: "set"
        lines:
          - "set system scripts op file bgp.slax"
          - "set system scripts op file bgp-neighbor.slax"
      register: response
    - name: "Print the response"
      ansible.builtin.debug:
        var: response
```

The following playbook configures the same statements using `lines` with configuration data in text format. In this case, `load: "merge"` is used.

```
---
- name: "Load and commit configuration"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Load configuration data using strings and commit"
      juniper.device.config:
        load: "merge"
        lines:
          - |
            system {
              scripts {
                op {
                  file bgp.slax;
                  file bgp-neighbor.slax;
```

```
            }
          }
        }
    register: response
  - name: "Print the response"
    ansible.builtin.debug:
      var: response
```

How to Load Configuration Data from a Local or Remote File

The `juniper.device.config` module enables you to load configuration data from a file. The file can reside in one of the following locations:

- Ansible control node
- Client device
- URL that is reachable from the client device

When you load configuration data from a file, you must indicate the location of the file and the format of the configuration data in the file. Supported configuration data formats include text, Junos XML elements, Junos OS `set` commands, and JSON. For information about loading files containing Jinja2 templates, see "How to Load Configuration Data Using a Jinja2 Template" on page 171.

You can specify the format of the configuration data either by explicitly including the `format` parameter in the module's argument list or by adding the appropriate extension to the configuration data file. If you specify the `format` parameter, it overrides the format indicated by the file extension. For information about specifying the format, see "How to Specify the Format of the Configuration Data to Load" on page 166. When the configuration data uses Junos XML format, you must enclose the data in the top-level `<configuration>` tag.

NOTE: You do not need to enclose configuration data that is formatted as ASCII text, Junos OS `set` commands, or JSON in `<configuration-text>`, `<configuration-set>`, or `<configuration-json>` tags as required when configuring the device directly within a NETCONF session.

Table 25 on page 170 outlines the module parameters that you can include to specify the location of the file.

Table 25: Specifying the Location of the Configuration File

Module Parameter	Description
`src`	Absolute or relative path to a file on the Ansible control node. The default directory is the playbook directory.
`url`	Absolute or relative path to a file on the client device, or an FTP location or an HTTP URL. The default directory on the client device is the current working directory, which defaults to the user's home directory.

To load configuration data from a local file on the Ansible control node, set the `src` argument to the absolute or relative path of the file containing the configuration data. For example:

```
---
- name: "Load and commit configuration"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Load configuration from a local file and commit"
      juniper.device.config:
        load: "merge"
        src: "build_conf/{{ inventory_hostname }}/junos.conf"
      register: response
    - name: "Print the response"
      ansible.builtin.debug:
        var: response
```

To load configuration data from a file on the Junos device or from an FTP or HTTP URL, use the `url` parameter and specify the path of the file that contains the configuration data to load. For example:

```
---
- name: "Load and commit configuration"
  hosts: dc1
  connection: local
  gather_facts: no
```

```
tasks:
  - name: "Load configuration from a remote file and commit"
    juniper.device.config:
      load: "merge"
      url: "/var/tmp/junos.conf"
    register: response
  - name: "Print the response"
    ansible.builtin.debug:
      var: response
```

The value for `url` can be an absolute or relative local file path, an FTP location, or an HTTP URL.

- The file path for a local file on the target device has one of the following forms:
  - */path/filename*—File on a mounted file system, either on the local flash disk or on hard disk.
  - **a:***filename* or **a:***path/filename*—File on the local drive. The default path is **/** (the root-level directory). The removable media can be in MS-DOS or UNIX (UFS) format.
- The file path for a file on an FTP server has the following form:

  ftp://*username*:*password*@*hostname*/*path*/*filename*

- The file path for a file on an HTTP server has the following form:

  http://*username*:*password*@*hostname*/*path*/*filename*

In each case, the default value for the ***path*** variable is the home directory for the user. To specify an absolute path, the application starts the path with the characters **%2F**; for example, **ftp://** ***username*:*password*@*hostname*/%2F*path*/*filename***.

How to Load Configuration Data Using a Jinja2 Template

The `juniper.device.config` module enables you to render configuration data from a Jinja2 template file on the Ansible control node and load and commit the configuration on a Junos device. Jinja is a template engine for Python that enables you to generate documents from predefined templates. The templates, which are text files in the desired language, provide flexibility through the use of expressions and variables. You can create Junos OS configuration data using Jinja2 templates in one of the supported configuration formats, which includes ASCII text, Junos XML elements, Junos OS `set` commands, and

JSON. The Ansible module uses the Jinja2 template and a supplied dictionary of variables to render the configuration data.

To load and commit configuration data using a Jinja2 template, include the `template` and `vars` parameters in the module's argument list.

- `template`—Path of the Jinja2 template file
- `vars`—Dictionary of keys and values that are required to render the Jinja2 template

You must also include the `format` parameter when the template's file extension does not indicate the format of the data. For information about specifying the format, see "How to Specify the Format of the Configuration Data to Load" on page 166.

For example, the **interfaces-mpls.j2** file contains the following Jinja2 template:

```
interfaces {
    {% for item in interfaces %}
    {{ item }} {
        description "{{ description }}";
        unit 0 {
            family {{ family }};
        }
    } {% endfor %}
}
protocols {
    mpls {
        {% for item in interfaces %}
        interface {{ item }};
        {% endfor %}
    }
    rsvp {
        {% for item in interfaces %}
        interface {{ item }};
        {% endfor %}
    }

}
```

To use the `juniper.device.config` module to load the Jinja2 template, set the `template` argument to the path of the template file and define the variables required by the template in the `vars` dictionary. The following playbook uses the Jinja2 template and the variables defined in `vars` to render the configuration

data and load and commit it on the target host. The `format` parameter indicates the format of the configuration data in the template file.

```
---
- name: "Load and commit configuration"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Load a configuration from a Jinja2 template and commit"
      juniper.device.config:
        load: "merge"
        template: "build_conf/templates/interfaces-mpls.j2"
        format: "text"
        vars:
          interfaces: ["ge-1/0/1", "ge-1/0/2", "ge-1/0/3"]
          description: "MPLS interface"
          family: "mpls"
      register: response
    - name: "Print the response"
      ansible.builtin.debug:
        var: response
```

The module generates the following configuration data, which is loaded into the candidate configuration on the device and committed:

```
interfaces {
    ge-1/0/1 {
        description "MPLS interface";
        unit 0 {
            family mpls;
        }
    }
    ge-1/0/2 {
        description "MPLS interface";
        unit 0 {
            family mpls;
        }
    }
    ge-1/0/3 {
```

```
        description "MPLS interface";
        unit 0 {
            family mpls;
        }
    }
}
protocols {
    mpls {
        interface ge-1/0/1;
        interface ge-1/0/2;
        interface ge-1/0/3;
    }
    rsvp {
        interface ge-1/0/1;
        interface ge-1/0/2;
        interface ge-1/0/3;
    }
}
```

How to Load the Rescue Configuration

A rescue configuration enables you to define a known working configuration or a configuration with a known state that you can restore at any time. You use the rescue configuration when you need to revert to a known configuration or as a last resort if the device configuration and the backup configuration files become damaged beyond repair. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration.

The `juniper.device.config` module enables you to revert to an existing rescue configuration on Junos devices. To load and commit the rescue configuration on a device, include the module's `rollback: "rescue"` argument. For example:

```
---
- name: "Revert to rescue configuration"
  hosts: dc1a
  connection: local
  gather_facts: no

  tasks:
    - name: "Load and commit rescue configuration"
      juniper.device.config:
```

```
        rollback: "rescue"
      register: response
    - name: "Print response"
      ansible.builtin.debug:
        var: response
```

How to Roll Back the Configuration

Junos devices store a copy of the most recently committed configuration and up to 49 previous configurations, depending on the platform. You can roll back to any of the stored configurations. This is useful when configuration changes cause undesirable results, and you want to revert back to a known working configuration. Rolling back the configuration is similar to the process for making configuration changes on the device, but instead of loading configuration data, you perform a rollback, which replaces the entire candidate configuration with a previously committed configuration.

The `juniper.device.config` module enables you to roll back to a previously committed configuration on Junos devices. To roll back the configuration and commit it, include the module's `rollback` argument, and specify the ID of the rollback configuration. Valid ID values are 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49).

The following playbook prompts for the rollback ID of the configuration to restore, rolls back the configuration and commits it, and then prints the configuration changes to standard output:

```
---
- name: "Roll back the configuration"
  hosts: dc1a
  connection: local
  gather_facts: no

  vars_prompt:
    - name: "ROLLBACK"
      prompt: "Rollback ID of the configuration to restore"
      private: no

  tasks:
    - name: "Roll back the configuration and commit"
      juniper.device.config:
        rollback: "{{ ROLLBACK }}"
      register: response
    - name: "Print the configuration changes"
```

```
      ansible.builtin.debug:
        var: response.diff_lines
```

```
user@ansible-cn:~/ansible$ ansible-playbook configuration-rollback.yaml
Rollback ID of the configuration to restore: 1

PLAY [Roll back the configuration] ********************************************

TASK [Roll back the configuration and commit] *********************************
changed: [dc1a.example.net]

TASK [Print the configuration changes] ****************************************
ok: [dc1a.example.net] => {
    "response.diff_lines": [
        "",
        "[edit interfaces]",
        "-   ge-0/0/0 {",
        "-       unit 0 {",
        "-           family mpls;",
        "-       }",
        "-   }"
    ]
}

PLAY RECAP ********************************************************************
dc1a.example.net  : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

How to Commit the Configuration

By default, when you use the `juniper.device.config` module to modify the configuration using either the `load` or the `rollback` argument, the module automatically performs a commit check and commits the changes. To prevent the module from performing a commit check or from committing the changes, set the `check` or `commit` argument to `false`, respectively.

You can also customize the commit operation with many of the same options that are available in the Junos OS CLI. Table 26 on page 177 outlines the module arguments that you can use to specify different commit options.

Table 26: Commit Options

Module Argument	Description	Default value for `load` and `rollback` operations
`check:` *boolean*	Perform a commit check or confirm a previous confirmed commit operation.	`true`
`check_commit_wait:` *seconds*	Wait the specified number of seconds between the commit check and the commit operation.	–
`comment: "`*string*`"`	Log a comment for that commit operation in the system log file and in the device's commit history.	–
`commit:` *boolean*	Commit the configuration changes or confirm a previous confirmed commit operation.	`true`
`commit_empty_changes:` *boolean*	Commit the configuration changes even if there are no differences between the candidate configuration and the committed configuration.	`false`
`commit_force_sync:` *boolean*	Synchronize and commit the configuration on all Routing Engines, even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine.	`false`
`commit_sync:` *boolean*	Synchronize and commit the configuration on all Routing Engines.	`false`
`confirmed:` *minutes*	Require that a commit operation be confirmed within a specified amount of time after the initial commit. If the commit is not confirmed in the specified time, roll back to the previously committed configuration. Either the `commit: true` option or the `check: true` option must be used to confirm the commit.	–

Table 26: Commit Options *(Continued)*

Module Argument	Description	Default value for `load` and `rollback` operations
`timeout: seconds`	Wait for completion of the operation using the specified value as the timeout.	30 seconds

Commit Comment

When you commit the configuration, you can include a brief comment to describe the purpose of the committed changes. To log a comment describing the changes, include the `comment: "comment string"` argument with the message string.

Commit Check

By default, the `config` module executes both a commit check and a commit operation. The `check_commit_wait` argument defines the number of seconds to wait between the commit check and commit operations. Include this argument when you need to provide sufficient time for the device to complete the commit check operation and release the configuration lock before initiating the commit operation. If you omit the `check_commit_wait` argument, there might be certain circumstances in which a device initiates the commit operation before the commit check operation releases its lock on the configuration, resulting in a `CommitError` and failed commit operation.

Commit Empty Changes

By default, if there are no differences between the candidate configuration and the committed configuration, the module does not commit the changes. To force a commit operation even when there are no differences, include the `commit_empty_changes: true` argument.

Commit Synchronize

If the device has dual Routing Engines, you can synchronize and commit the configuration on both Routing Engines by including the `commit_sync: true` argument. To force the `commit synchronize` operation to succeed even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine, use the `commit_force_sync: true` argument. When you include the `commit_force_sync: true` option, the device terminates any configuration sessions on the other Routing Engine before synchronizing and committing the configuration.

Commit Confirm

To require that a commit operation be confirmed within a specified amount of time after the initial commit, include the `confirmed: minutes` argument. If the commit is not confirmed within the given time limit, the configuration automatically rolls back to the previously committed configuration. The allowed

range is 1 through 65,535 minutes. The confirmed commit operation is useful for verifying that a configuration change works correctly and does not prevent management access to the device. If the change prevents access or causes other errors, the automatic rollback to the previous configuration enables access to the device after the rollback deadline passes. To confirm the commit operation, invoke the `config` module with the `check: true` or `commit: true` argument.

In the following playbook, the first task modifies the configuration, waits 10 seconds between the commit check and the commit operation, and requires that the commit operation be confirmed within 5 minutes. It also logs a comment for the commit. The second task issues a `commit check` operation to confirm the commit. In a real-world scenario, you might perform validation tasks after the initial commit and only execute the commit confirmation if the tasks pass certain validation criteria.

```
---
- name: "Load configuration and confirm within 5 minutes"
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: "Load configuration. Wait 10 seconds between check and commit. Confirm within 5
min."
      juniper.device.config:
        load: "merge"
        format: "text"
        src: "build_conf/{{ inventory_hostname }}/junos.conf"
        check_commit_wait: 10
        confirmed: 5
        comment: "updated using Ansible"
      register: response
    - name: "Print the response"
      ansible.builtin.debug:
        var: response

    - name: "Confirm the commit with a commit check"
      juniper.device.config:
        check: true
        diff: false
        commit: false
      register: response
    - name: "Print the response"
      ansible.builtin.debug:
        var: response
```

How to Ignore Warnings When Configuring Devices

The `juniper.device.config` module enables you to modify and commit the configuration on Junos devices. In some cases, the RPC reply might contain `<rpc-error>` elements with a severity level of warning or higher that cause the module to raise an `RpcError` exception. An `RpcError` exception can cause the load or commit operation to fail.

In certain cases, it might be necessary or desirable to suppress the `RpcError` exceptions that are raised in response to warnings for load and commit operations. You can instruct the `config` module to suppress `RpcError` exceptions that are raised for warnings by including the `ignore_warning` parameter in the module's argument list. The `ignore_warning` argument takes a Boolean, a string, or a list of strings.

To instruct the module to ignore all warnings for load and commit operations performed by the module, include the `ignore_warning: true` argument. The following example ignores all warnings for load and commit operations.

```
---
- name: Configure Device
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Configure op script
      juniper.device.config:
        config_mode: "private"
        load: "set"
        lines:
          - "set system scripts op file bgp.slax"
        ignore_warning: true
      register: response
    - name: Print the response
      ansible.builtin.debug:
        var: response
```

If you include `ignore_warning: true` and all of the `<rpc-error>` elements have a severity level of warning, the application ignores all warnings and does not raise an `RpcError` exception. However, any `<rpc-error>` elements with higher severity levels will still raise exceptions.

To instruct the module to ignore specific warnings, set the `ignore_warning` argument to a string or a list of strings containing the warnings to ignore. The following example ignores two specific warnings:

```
---
- name: Configure Device
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Configure Junos device and ignore warnings
      juniper.device.config:
        config_mode: "private"
        load: "merge"
        src: "build_conf/{{ inventory_hostname }}/junos.conf"
        ignore_warning:
          - "Advertisement-interval is less than four times"
          - "Chassis configuration for network services has been changed."
      register: response
    - name: Print the response
      ansible.builtin.debug:
        var: response
```

The module suppresses `RpcError` exceptions if all of the `<rpc-error>` elements have a severity level of warning and each warning in the response matches one or more of the specified strings.

Example: Use Ansible to Configure Junos Devices

IN THIS SECTION

The juniper.device.config module enables you to manage the configuration on Junos devices. This example uses the config module to make configuration changes on a Junos device through NETCONF over SSH.

Requirements

This example uses the following hardware and software components:

- Configuration management server running Ansible 2.10 or later with the juniper.device collection installed
- Junos device with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the Ansible controller and the Junos device
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the juniper.device.config module to enable a new op script in the configuration of the target Junos devices. The configuration data file, **junos-config.conf**, contains the relevant configuration data formatted as text.

The playbook includes the Check NETCONF connectivity task, which utilizes the ansible.builtin.wait_for Ansible module to try to establish a NETCONF session with the target device using the NETCONF default port (830). If the control node fails to establish a NETCONF session with a target device during playbook execution, then it skips the remaining tasks in the play for that device.

The playbook uses the juniper.device.file_copy module to copy the new op script from the Ansible control node to the Junos device. The module arguments specify the directory and filename of the script on the local device and the destination directory on the remote device.

The task to configure the device executes the juniper.device.config module provided that the NETCONF check was successful. The load: "merge" argument loads the new configuration data into the candidate configuration using a load merge operation. By default, the config module commits configuration data on a device for load and rollback operations. The module arguments include the comment argument, which records a commit comment in the device's system log file and commit history.

Configuration

IN THIS SECTION

Create the Configuration Data File

Step-by-Step Procedure

To create the configuration data file that is used by the module:

1. Create a new file with the appropriate extension based on the format of the configuration data, which in this example is text.
2. Include the desired configuration changes in the file.

```
user@ansible-cn:~/ansible$ cat build_conf/dc1a.example.net/junos-config.conf
system {
    scripts {
        op {
            file bgp.slax;
        }
    }
}
```

Create the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the `config` module to make configuration changes on a Junos device:

1. Include the playbook boilerplate, which executes the modules locally.

```
---
- name: Load and commit configuration data on a Junos device
  hosts: dc1
  connection: local
  gather_facts: no
```

2. (Optional) Create a task to verify NETCONF connectivity.

```
tasks:
  - name: Check NETCONF connectivity
    ansible.builtin.wait_for:
      host: "{{ inventory_hostname }}"
      port: 830
      timeout: 5
```

3. Create a task to copy the new op script to the device.

```
  - name: Copy the op script to the device
    juniper.device.file_copy:
      action: put
      file: bgp.slax
      local_dir: scripts
      remote_dir: /var/db/scripts/op
```

4. Create the task to load the configuration onto the device and commit it.

```
  - name: Merge configuration data from a file and commit
    juniper.device.config:
      load: "merge"
      src: "build_conf/{{ inventory_hostname }}/junos-config.conf"
      comment: "Configuring op script with Ansible"
    register: response
```

5. (Optional) Create a task to print the response, which includes the configuration changes in *diff* format.

```
  - name: Print the response
    ansible.builtin.debug:
      var: response
```

Results

On the Ansible control node, review the completed playbook. If the playbook does not display the intended code, repeat the instructions in this example to correct the playbook.

```
---
- name: Load and commit configuration data on a Junos device
  hosts: dc1
  connection: local
  gather_facts: no

  tasks:
    - name: Check NETCONF connectivity
      ansible.builtin.wait_for:
        host: "{{ inventory_hostname }}"
        port: 830
        timeout: 5

    - name: Copy the op script to the device
      juniper.device.file_copy:
        action: put
        file: bgp.slax
        local_dir: scripts
        remote_dir: /var/db/scripts/op

    - name: Merge configuration data from a file and commit
      juniper.device.config:
        load: "merge"
        src: "build_conf/{{ inventory_hostname }}/junos-config.conf"
        comment: "Configuring op script with Ansible"
      register: response

    - name: Print the response
      ansible.builtin.debug:
        var: response
```

Execute the Playbook

To execute the playbook:

- Issue the `ansible-playbook` command on the control node, and provide the playbook path and any desired options.

```
user@ansible-cn:~/ansible$ ansible-playbook ansible-pb-junos-config.yaml

PLAY [Load and commit configuration data on a Junos device] ***************

TASK [Check NETCONF connectivity] ****************************************
ok: [dc1a.example.net]

TASK [Copy the op script to the device] **********************************
changed: [dc1a.example.net]

TASK [Merge configuration data from a file and commit] *******************
changed: [dc1a.example.net]

TASK [Print the response] ************************************************
ok: [dc1a.example.net] => {
    "response": {
        "changed": true,
        "diff": {
            "prepared": "\n[edit system scripts op]\n+     file bgp.slax;\n"
        },
        "diff_lines": [
            "",
            "[edit system scripts op]",
            "+     file bgp.slax;"
        ],
        "failed": false,
        "file": "build_conf/dc1a.example.net/junos-config.conf",
        "msg": "Configuration has been: opened, loaded, checked, diffed, committed, closed."
    }
}

PLAY RECAP ***************************************************************
dc1a.example.net  : ok=4    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Verification

IN THIS SECTION

- Verify the Configuration | 187

Verify the Configuration

Purpose

Verify that the configuration was correctly updated on the Junos device.

Action

Review the Ansible playbook output to see whether the configuration task succeeded or failed. You can also log in to the Junos device and view the configuration, commit history, and log files to verify the configuration and commit, for example:

```
user@dc1a> show configuration system scripts
op {
    file bgp.slax;
}
```

```
user@dc1a> show system commit
0   2020-12-17 15:33:50 PST by user via netconf
    Configuring op script with Ansible
```

```
user@dc1a> show log messages
Dec 17 15:33:39  dc1a mgd[33444]: UI_COMMIT: User 'user' requested 'commit' operation (comment:
Configuring op script with Ansible)
Dec 17 15:33:57  dc1a mgd[33444]: UI_COMMIT_COMPLETED: commit complete
```

Troubleshoot Playbook Errors

IN THIS SECTION

Troubleshoot Timeout Errors

Problem

The playbook generates a `TimeoutExpiredError` error message and fails to update the device configuration.

```
ncclient.operations.errors.TimeoutExpiredError: ncclient timed out while waiting for an rpc reply
```

The default time for a NETCONF RPC to time out is 30 seconds. Large configuration changes might exceed this value causing the operation to time out before the configuration can be uploaded and committed.

Solution

To accommodate configuration changes that might require a commit time that is longer than the default RPC timeout interval, set the module's `timeout` argument to an appropriate value and re-run the playbook.

Troubleshoot Configuration Lock Errors

Problem

The playbook generates a `LockError` error message indicating that the configuration cannot be locked. For example:

```
FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive mode:
LockError(severity: error, bad_element: None, message: configuration database modified)"}
```

or

```
FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive mode:
LockError(severity: error, bad_element: lock-configuration, message: permission denied)"}
```

A configuration lock error can occur for the following reasons:

- Another user has an exclusive lock on the configuration.
- Another user made changes to the configuration database but has not yet committed the changes.
- The user executing the Ansible module does not have permissions to configure the device.

Solution

The `LockError` message string usually indicates the root cause of the issue. If another user has an exclusive lock on the configuration or has modified the configuration, wait until the lock is released or the changes are committed, and execute the playbook again. If the cause of the issue is that the user does not have permissions to configure the device, either execute the playbook with a user who has the necessary permissions, or if appropriate, configure the Junos device to give the current user the necessary permissions to make the changes.

Troubleshoot Configuration Change Errors

Problem

The playbook generates a `ConfigLoadError` error message indicating that the configuration cannot be modified, because permission is denied.

```
FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: error, bad_element: scripts, message: error: permission denied)"}
```

This error message is generated when the user executing the Ansible module has permission to alter the configuration but does not have permission to alter the requested section of the configuration.

Solution

Either execute the playbook with a user who has the necessary permissions, or if appropriate, configure the Junos device to give the current user the necessary permissions to make the changes.

RELATED DOCUMENTATION

Use Ansible to Retrieve or Compare Junos OS Configurations | 148

Understanding the Ansible for Junos OS Collections and Modules | 7

CHAPTER 7

Troubleshoot Ansible for Junos OS

Ansible for Junos OS Troubleshooting Summary

Table 27 on page 192 lists some common errors that you might encounter when you use Ansible to manage Juniper Networks devices. For each issue, the table provides a reference that has additional information about the error and potential solutions to resolve the issue. However, every environment is unique, and a proposed solution might not work in all cases.

Table 27: Ansible for Junos OS Errors

General Issue	Sample Errors	Reference
Junos PyEZ (`junos-eznc`) library errors	`"msg": "junos-eznc (aka PyEZ) >= 2.6.0 is required for this module. However, junos-eznc does not appear to be currently installed."`	"Troubleshoot Junos PyEZ (junos-eznc) Install Errors for Ansible Modules" on page 194

Ansible collection and module resolution errors	`ERROR! no action detected in task. This often indicates a misspelled module name, or incorrect module path.`
	
	`ERROR! couldn't resolve module/ action 'juniper.device.facts'. This often indicates a misspelling, missing collection, or incorrect module path.`
		"Troubleshoot Ansible Collection and Module Errors When Managing Junos Devices" on page 197

External connection plugins error (`AttributeError: 'Connection'`) (Occurs for Ansible `ansible-core` version >= 2.12.9)	`AttributeError: 'Connection' object has no attribute 'nonetype'`	External connection plugins that do not set _sub_plugin trigger errors

Unreachable host	`UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ", "unreachable": true}`
	
	`unknown command: /bin/sh\r\n`
		"Troubleshoot Failed or Invalid Connection Errors" on page 199

Table 27: Ansible for Junos OS Errors *(Continued)*

General Issue	Sample Errors	Reference
Unknown host (ConnectUnknownHostError)	"msg": "Unable to make a PyEZ connection: ConnectUnknownHostError(dc1a.example.net)"	"Troubleshoot Unknown Host Errors" on page 201
Connection refused (ConnectRefusedError)	"msg": "Unable to make a PyEZ connection: ConnectRefusedError(dc1a.example.net)"	"Troubleshoot Refused Connection Errors" on page 202
Authentication error (ConnectAuthError)	"msg": "Unable to make a PyEZ connection: ConnectAuthError(dc1a.example.net)"	"Troubleshoot ConnectAuthError Issues" on page 204
conn_type error	AttributeError: 'JuniperJunosModule' object has no attribute 'conn_type'	"Troubleshoot Attribute conn_type Errors" on page 205
Configuration timeout error (TimeoutExpiredError)	ncclient.operations.errors.TimeoutExpiredError: ncclient timed out while waiting for an rpc reply	"Troubleshoot Configuration Timeout Errors" on page 207
Configuration lock error (LockError)	"msg": "Unable to open the configuration in exclusive mode: LockError(severity: error, bad_element: None, message: configuration database modified)" "msg": "Unable to open the configuration in exclusive mode: LockError(severity: error, bad_element: lock-configuration, message: permission denied)"	"Troubleshoot Configuration Lock Errors" on page 208

Table 27: Ansible for Junos OS Errors *(Continued)*

General Issue	Sample Errors	Reference
Configuration load error (`ConfigLoadError`)	`FAILED! => {"changed": false, "msg": "Failure loading the configuraton: ConfigLoadError(severity: error, bad_element: scripts, message: error: permission denied)"}`	"Troubleshoot Configuration Load Errors" on page 210
Configuration commit error (`CommitError`)	`FAILED! => {"changed": false, "msg": "Unable to commit configuration: CommitError(edit_path: None, bad_element: None, message: error: remote lock-configuration failed on re0\n\nnote: consider using 'commit synchronize force' to\nterminate remote edit sessions and force the commit)"}`	"Troubleshoot Commit Errors" on page 211

Troubleshoot Junos PyEZ (junos-eznc) Install Errors for Ansible Modules

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module, the Ansible control node generates an error that `junos-eznc` is not installed. For example:

```
"msg": "junos-eznc (aka PyEZ) >= 2.6.0 is required for this module. However, junos-eznc does not
appear to be currently installed. See https://github.com/Juniper/py-junos-eznc#installation for
details on installing junos-eznc."
```

or

```
"msg": "junos-eznc is required but does not appear to be installed. It can be installed using
`pip  install junos-eznc`"
```

Cause

The Juniper Networks Ansible modules in the `juniper.device` collection use the Junos PyEZ Python library to perform operations on Junos devices. Ansible generates this error if the library is not installed or if Ansible can't locate the library. Ansible might fail to locate the library, for example, if you install Ansible in a virtual environment or under a Python installation in a non-standard system location and Ansible is searching for the library in a different location like the default system location.

Solution

Install Junos PyEZ on the Ansible control node and update any necessary environment variables. See https://github.com/Juniper/py-junos-eznc#installation for more information.

If you run Ansible using a Python installation in a virtual environment or a non-standard system location, you must:

- Install Junos PyEZ under the desired Python installation (in the virtual environment or non-standard system location, as appropriate).

- Specify the path to the Python interpreter—for example, by setting the `interpreter_python` variable in the Ansible configuration file or by defining the `ansible_python_interpreter` variable for the appropriate devices in the Ansible inventory file.

```
# file ansible.cfg
[defaults]
interpreter_python = /home/user/MyProjects/Ansible/venv/bin/python
...
```

  For more information, see Interpreter Discovery in the official Ansible documentation.

To verify that Junos PyEZ is successfully installed on the control node, launch the Python interactive shell using the same Python installation that you use for Ansible operations. Then import the `jnpr.junos` package.

```
(venv) user@ansible-cn:~/MyProjects/Ansible$ python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import jnpr.junos
>>> jnpr.junos.__version__
'2.6.0'
```

If you successfully import the `jnpr.junos` package and there is no error message, then Junos PyEZ is installed on the Ansible control node. If you still see the same error message when you execute the Ansible module, make sure you have specified the correct location for the Python interpreter in your Ansible environment.

NOTE: You can verify the Python installation that Ansible uses by executing your Ansible playbook with the `-vvv` option and reviewing the log messages.

RELATED DOCUMENTATION

Ansible for Junos OS Server Requirements | 21

Troubleshoot Ansible Authentication Errors When Managing Junos Devices | 203

Troubleshoot Ansible Collection and Module Errors When Managing Junos Devices | 197

Troubleshoot Ansible Collection and Module Errors When Managing Junos Devices

IN THIS SECTION

Problem

Description

During execution of an Ansible playbook, the control node generates an error that the `juniper.device` collection was not found, that no action was detected in the task, or that the module could not be resolved. For example:

```
ERROR! couldn't resolve module/action 'juniper.device.facts'. This often indicates a
misspelling, missing collection, or incorrect module path.
```

or

```
ERROR! no action detected in task. This often indicates a misspelled module name, or incorrect
module path.
```

Cause

The Ansible control node cannot locate the `juniper.device` collection and associated modules.

Solution

The `juniper.device` collection is hosted on the Ansible Galaxy website. In order to use the modules in the collection, you must install the collection on the Ansible control node and also reference it in your playbook.

To install the `juniper.device` collection on the Ansible control node, execute the `ansible-galaxy collection install` command, and specify `juniper.device`.

```
user@ansible-cn:~$ ansible-galaxy collection install juniper.device
```

NOTE: If you do not install the collection in the default location, you might need to define the path to it in your Ansible setup. For more information about installing collections and specifying the path, see the official Ansible documentation.

To use the `juniper.device` collection modules, reference the fully qualified collection name when you execute a module. For example:

```
---
- name: Get Device Facts
  hosts: junos-all
  connection: local
  gather_facts: no

  tasks:
    - name: Get device facts
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

RELATED DOCUMENTATION

Ansible for Junos OS Server Requirements | 21

Understanding the Ansible for Junos OS Collections and Modules | 7

Troubleshoot Ansible Connection Errors When Managing Junos Devices

IN THIS SECTION

The following sections outline connection errors that you might encounter when using Ansible to manage Junos devices. These sections also present potential causes and solutions for each error.

Troubleshoot Failed or Invalid Connection Errors

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module on a Junos device, the Ansible control node generates an error about a failed SSH connection or an unknown command. For example:

```
UNREACHABLE! => {"changed": false, "msg": "Failed to connect to the host via ssh: ",
"unreachable": true}
```

or

```
unknown command: /bin/sh\r\n
```

Cause

These errors can arise when the module is not run locally on the Ansible control node.

Normally Ansible requires Python on the managed node, and the Ansible control node sends the module to the node, where it is executed and then removed. The Juniper Networks modules do not require Python on Junos devices, because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on Junos devices, you must run the modules locally on the Ansible control node where Python is installed. If Ansible tries to execute a module directly on the Junos device, it generates an error.

Solution

To direct the Ansible control node to run the `juniper.device` modules locally, include `connection: local` in the Ansible playbook, or include the `--connection local` command-line argument when executing individual modules. For example:

```
---
- name: Get Device Facts
  hosts: junos
  connection: local
  gather_facts: no
```

Troubleshoot Unknown Host Errors

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module, the Ansible control node generates a `ConnectUnknownHostError` error.

```
"msg": "Unable to make a PyEZ connection: ConnectUnknownHostError(dc1a.example.net)"
```

Cause

The host is not defined in the Ansible inventory file or the Ansible control node is unable to resolve the hostname.

When executing an Ansible module either directly or from a playbook, any host referenced in the module arguments or the playbook must be defined in the Ansible inventory file. The default location for the inventory file is **/etc/ansible/hosts**. If the inventory file references a hostname, the Ansible control node must be able to resolve the hostname.

Solution

Update the Ansible inventory file to include the missing host, and ensure that DNS resolution is working correctly.

For information about the Ansible inventory file, see "Understanding the Ansible Inventory File When Managing Junos Devices" on page 12 as well as the official Ansible documentation at https://www.ansible.com/.

Troubleshoot Refused Connection Errors

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module, the Ansible control node generates a `ConnectRefusedError` error. For example:

```
"msg": "Unable to make a PyEZ connection: ConnectRefusedError(dc1a.example.net)"
```

Cause

The most likely cause for a refused connection error is that NETCONF over SSH is not enabled on the Junos device.

To quickly test whether NETCONF is enabled, verify that the user account executing the Ansible module can successfully start a NETCONF session with the device.

```
user@ansible-cn:~$ ssh user@dc1a.example.net -p 830 -s netconf
```

If the user can successfully establish a NETCONF session with the device on either the default NETCONF port (830) or a port that is specifically configured for NETCONF on your device, then NETCONF is enabled. Otherwise, you must enable NETCONF over SSH on the device.

Solution

Enable the NETCONF-over-SSH service on the Junos device.

```
[edit]
user@host# set system services netconf ssh
user@host# commit
```

RELATED DOCUMENTATION

- Set up Ansible for Junos OS Managed Nodes | 24
- Understanding the Ansible Inventory File When Managing Junos Devices | 12
- Troubleshoot Ansible Authentication Errors When Managing Junos Devices | 203

Troubleshoot Ansible Authentication Errors When Managing Junos Devices

IN THIS SECTION

The following sections outline authentication errors that you might encounter when using Ansible to manage Junos devices. These sections also present potential causes and solutions for each error.

Troubleshoot ConnectAuthError Issues

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module, the Ansible control node generates a `ConnectAuthError` error for failed authentication. For example:

```
"msg": "Unable to make a PyEZ connection: ConnectAuthError(dc1a.example.net)"
```

Cause

The Junos device might fail to authenticate the user for the following reasons:

- The user does not an have an account on the Junos device.
- The user has an account with a text-based password configured on the Junos device, but the wrong password or no password is supplied for the user when executing the module.
- The user has an account on the Junos device with SSH keys configured, but the SSH keys are inaccessible on either the device or the control node.

Solution

Ensure that the user executing the modules has a Junos OS login account on all target Junos devices and that an SSH public/private key pair or text-based password is configured for the account. If SSH keys are configured, verify that the user can access them. For more information, see "Authenticate Users Executing Ansible Modules on Junos Devices" on page 37.

Troubleshoot Attribute conn_type Errors

IN THIS SECTION

Problem

Description

During execution of a `juniper.device` module on a Junos device, the Ansible control node generates the following error:

```
AttributeError: 'JuniperJunosModule' object has no attribute 'conn_type'
```

Cause

Whereas the older, deprecated `Juniper.junos` modules supported using a `provider` dictionary to define connection and authentication parameters, the `juniper.device` modules do not support using a `provider` dictionary and generate the aforementioned error if one is referenced.

Solution

If you supply connection and authentication parameters in the playbook's play for the `juniper.device` modules, the parameters must be defined in the location appropriate for the Ansible connection. For persistent connections (`connection: juniper.device.pyez`), define the parameters under the `vars:` section. For local connections (`connection: local`), define the parameters either under the `vars:` section or as top-level module arguments. For example:

```
---
- name: Get device facts
  hosts: dc1
  connection: juniper.device.pyez
```

```
  gather_facts: no

  vars_prompt:
    - name: "DEVICE_PASSWORD"
      prompt: "Device password"
      private: yes

  vars:
    passwd: "{{ DEVICE_PASSWORD }}"

  tasks:
    - name: Get device facts
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
```

```
---
- name: Get device facts
  hosts: dc1
  connection: local
  gather_facts: no

  vars_prompt:
    - name: "DEVICE_PASSWORD"
      prompt: "Device password"
      private: yes

  tasks:
    - name: Get device facts
      juniper.device.facts:
        passwd: "{{ DEVICE_PASSWORD }}"
        savedir: "{{ playbook_dir }}"
```

RELATED DOCUMENTATION

Set up Ansible for Junos OS Managed Nodes | **24**

Authenticate Users Executing Ansible Modules on Junos Devices | **37**

Troubleshoot Ansible Connection Errors When Managing Junos Devices | **199**

Troubleshoot Ansible Errors When Configuring Junos Devices

IN THIS SECTION

The following sections outline errors that you might encounter when using the `juniper.device.config` module to configure Junos devices. These sections also present potential causes and solutions for each error.

Troubleshoot Configuration Timeout Errors

IN THIS SECTION

Problem

Description

The module generates a `TimeoutExpiredError` error message and fails to update the device configuration.

```
ncclient.operations.errors.TimeoutExpiredError: ncclient timed out while waiting for an rpc reply
```

Cause

The default time for a NETCONF RPC to time out is 30 seconds. Large configuration changes might exceed this value causing the operation to time out before the configuration can be uploaded and committed.

Solution

To accommodate configuration changes that might require a commit time that is longer than the default RPC timeout interval, set the module's `timeout` argument to an appropriate value and re-run the playbook.

Troubleshoot Configuration Lock Errors

IN THIS SECTION

Problem

Description

The module generates an error message indicating that the configuration database cannot be locked. For example:

```
FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive mode:
LockError(severity: error, bad_element: None, message: configuration database modified)"}
```

or

```
FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive mode:
LockError(severity: error, bad_element: lock-configuration, message: permission denied)"}
```

Cause

A configuration lock error can occur for the following reasons:

- Another user has an exclusive lock on the configuration.
- Another user made changes to the configuration database but has not yet committed the changes.
- The user executing the Ansible module does not have permissions to configure the device.

Solution

The `LockError` message string usually indicates the root cause of the issue. If another user has modified or has an exclusive lock on the configuration, wait until the changes are committed or the lock is released, and execute the playbook again. If the cause is that the user does not have permissions to configure the device, either execute the playbook with a user who has the necessary permissions, or if appropriate, configure the Junos device to give the current user the necessary permissions to make the changes.

Troubleshoot Configuration Load Errors

IN THIS SECTION

Problem

Description

The module generates a `ConfigLoadError` error message indicating that the configuration cannot be modified because permission is denied.

```
FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: error, bad_element: scripts, message: error: permission denied)"}
```

Cause

The Ansible module generates this error message when the user executing the module has permission to modify the configuration but does not have permission to alter the requested section of the configuration.

Solution

To solve this issue, either execute the playbook with a user who has the necessary permissions, or if appropriate, configure the Junos device to give the current user the necessary permissions to make the changes.

Troubleshoot Commit Errors

IN THIS SECTION

Problem

Description

The module generates a `CommitError` error message indicating that the commit operation failed due to a configuration lock error.

```
FAILED! => {"changed": false, "msg": "Unable to commit configuration: CommitError(edit_path:
None, bad_element: None, message: error: remote lock-configuration failed on re0\n\nnote:
consider using 'commit synchronize force' to\nterminate remote edit sessions and force the
commit)"}
```

Cause

A configuration lock error can occur for the reasons described in "Troubleshoot Configuration Lock Errors" on page 208. However, a configuration lock failed message might be generated as part of a `CommitError` instead of a `LockError`. This situation can occur when a task requests a commit check and a commit operation, and the device initiates the commit operation before the commit check operation releases the configuration lock.

Solution

To enable sufficient time for the device to complete the commit check operation and release the configuration lock before initiating the commit operation, set the module's `check_commit_wait` parameter to an appropriate value and re-run the playbook. The `check_commit_wait` value is the number of seconds to wait between the commit check and commit operations.

The following sample task waits five seconds between the commit check and commit operations:

```
- name: "Load configuration. Wait 5 seconds between check and commit"
  juniper.device.config:
    load: "merge"
    format: "text"
    src: "build_conf/{{ inventory_hostname }}/junos.conf"
    check_commit_wait: 5
    comment: "updated using Ansible"
```

RELATED DOCUMENTATION

Use Ansible to Configure Junos Devices | 160