

Junos[®] OS

Ansible for Junos OS Developer Guide

Published
2021-04-22

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

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Junos[®] OS Ansible for Junos OS Developer Guide
Copyright © 2021 Juniper Networks, Inc. All rights reserved.

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 the Documentation | viii

Documentation and Release Notes | viii

Documentation Conventions | viii

Documentation Feedback | xi

Requesting Technical Support | xi

Self-Help Online Tools and Resources | xii

Creating a Service Request with JTAC | xii

1

Disclaimer

Ansible for Junos OS Disclaimer | 14

2

Ansible Overview

Understanding Ansible for Junos OS | 16

Ansible for Junos OS Overview | 16

Benefits of Ansible and Ansible for Junos OS | 17

Additional Resources | 17

Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19

Understanding Ansible Collections, Roles, and Modules for Managing Devices Running Junos OS | 19

How to Execute Modules on Devices Running Junos OS | 20

Juniper Networks juniper.device Collection | 23

Juniper Networks Juniper.junos Role | 24

Understanding the Ansible Inventory File When Managing Devices Running Junos OS | 26

Creating and Executing Ansible Playbooks to Manage Devices Running Junos OS | 28

Creating a Playbook | 29

Executing the Playbook | 33

3

Installing Ansible for Junos OS

Ansible for Junos OS Server Requirements | 35

Prerequisite Software | 36

Installing the juniper.device Collection and the Juniper.junos Role | 36

Using the Ansible for Junos OS Docker Image | 37

Setting up Ansible for Junos OS Managed Nodes | 38

Enabling NETCONF on Devices Running Junos OS | 39

Satisfying Requirements for SSHv2 Connections | 39

Configuring Telnet Service on Devices Running Junos OS | 40

4

Using Ansible to Connect to Devices Running Junos OS

Connecting to Devices Running Junos OS Using Ansible | 43

Connection Methods Overview | 43

Understanding Local and Persistent Ansible Connections | 45

Connecting to a Device Using SSH | 46

Connecting to a Device Using Telnet | 49

Connecting to a Device Using a Serial Console Connection | 50

Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51

Authentication Overview | 51

Understanding the Default Values for Juniper Networks Modules | 53

How to Define Authentication Parameters in the vars: Section for Local and Persistent Connections | 54

How to Define the provider Parameter in Juniper.junos Modules | 55

How to Authenticate the User Using SSH Keys | 57

Generating and Configuring the SSH Keys | 57

Using SSH Keys in Ansible Playbooks | 58

How to Authenticate the User Using a Playbook or Command-Line Password Prompt | 59

How to Authenticate the User Using an Ansible Vault-Encrypted File | 61

How to Authenticate Through a Console Server | 63

Using Ansible to Manage Device Operations

Using Ansible to Retrieve Facts from Devices Running Junos OS | 66

Using Ansible to Execute Commands and RPCs on Devices Running Junos OS | 70

How to Execute Commands with the Juniper Networks Modules | 71

How to Execute RPCs with the Juniper Networks Modules | 72

Understanding the Module Response | 74

How to Specify the Format for the Command or RPC Output | 75

How to Save the Command or RPC Output to a File | 76

Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS | 79

Module Overview | 80

Understanding Junos PyEZ Tables | 80

How to Use the Juniper Networks Ansible Modules with Junos PyEZ Tables | 81

Specifying RPC Arguments | 84

Using Ansible to Halt, Reboot, or Shut Down Devices Running Junos OS | 86

Using Ansible to Halt, Reboot, or Shut Down Devices | 86

How to Perform a Halt, Reboot, or Shut Down with a Delay or at a Specified Time | 88

How to Specify the Target Routing Engine | 89

How to Reboot a VM Host | 90

Example: Using Ansible to Reboot Devices Running Junos OS | 91

Using Ansible to Install Software on Devices Running Junos OS | 98

Using Ansible to Install Software | 98

How to Specify the Software Image Location | 99

Installation Process Overview | 101

How to Specify Timeout Values | 103

How to Specify Installation Options That Do Not Have an Equivalent Module Argument | 104

How to Perform a VM Host Upgrade | 105

How to Perform a Unified ISSU or NSSU | 105

Example: Using Ansible to Install Software | 107

Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings | 116

How to Use Ansible to Restore the Factory-Default Configuration Settings | 116

Example: Using Ansible to Restore the Factory-Default Configuration Settings | 118

Using Junos Snapshot Administrator in Python (JSNAPy) in Ansible Playbooks | 125

Module Overview | 126

Taking and Comparing Snapshots | 129

Performing Snapcheck Operations | 132

Understanding the jsnapy and juniper_junos_jsnapy Module Output | 133

Enabling the jsnapy Callback Plugin | 135

Example: Using Ansible to Perform a JSNAPy Snapcheck Operation | 136

6

Using Ansible to Manage the Configuration

Using Ansible to Retrieve or Compare Junos OS Configurations | 156

How to Specify the Source Database for the Configuration Data | 157

How to Specify the Scope of the Configuration Data to Return | 158

How to Specify the Format of the Configuration Data to Return | 160

How to Retrieve Configuration Data for Third-Party YANG Data Models | 161

How to Specify Options That Do Not Have an Equivalent Module Argument | 163

How to Save Configuration Data To a File | 164

How to Compare the Active Configuration to a Previous Configuration | 166

Using Ansible to Configure Devices Running Junos OS | 169

Module Overview | 170

How to Specify the Configuration Mode | 171

How to Specify the Load Action | 173

How to Specify the Format of the Configuration Data to Load | 173

How to Load Configuration Data as Strings | 174

How to Load Configuration Data from a Local or Remote File | 176

How to Load Configuration Data Using a Jinja2 Template | 178

How to Load the Rescue Configuration | 181

How to Roll Back the Configuration | 182

How to Commit the Configuration | 183

How to Ignore Warnings When Configuring Devices | 186

Example: Using Ansible to Configure Devices Running Junos OS | 187

Troubleshooting Ansible for Junos OS

Troubleshooting Junos PyEZ (junos-eznc) Install Errors for Ansible Modules | 197

Troubleshooting Ansible Collection, Role, and Module Errors When Managing Devices Running Junos OS | 198

Troubleshooting Ansible Connection Errors When Managing Devices Running Junos OS | 200

Troubleshooting Failed or Invalid Connection Errors | 201

Troubleshooting Unknown Host Errors | 202

Troubleshooting Refused Connection Errors | 202

Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS | 203

Troubleshooting ConnectAuthError Issues | 204

Troubleshooting Attribute conn_type Errors | 204

Troubleshooting Ansible Errors When Configuring Devices Running Junos OS | 206

Troubleshooting Configuration Timeout Errors | 207

Troubleshooting Configuration Lock Errors | 207

Troubleshooting Configuration Load Errors | 208

Troubleshooting Commit Errors | 208

About the Documentation

IN THIS SECTION

- Documentation and Release Notes | viii
- Documentation Conventions | viii
- Documentation Feedback | xi
- Requesting Technical Support | xi

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

Documentation and Release Notes

To obtain the most current version of all Juniper Networks[®] technical documentation, see the product documentation page on the Juniper Networks website at <https://www.juniper.net/documentation/>.

If the information in the latest release notes differs from the information in the documentation, follow the product Release Notes.

Juniper Networks Books publishes books by Juniper Networks engineers and subject matter experts. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration. The current list can be viewed at <https://www.juniper.net/books>.

Documentation Conventions

[Table 1 on page ix](#) defines notice icons used in this guide.

Table 1: Notice Icons







Icon	Meaning	Description
	Informational note	Indicates important features or instructions.
	Caution	Indicates a situation that might result in loss of data or hardware damage.
	Warning	Alerts you to the risk of personal injury or death.
	Laser warning	Alerts you to the risk of personal injury from a laser.
	Tip	Indicates helpful information.
	Best practice	Alerts you to a recommended use or implementation.

Table 2 on page ix defines the text and syntax conventions used in this guide.

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the configure command: user@host> configure
Fixed-width text like this	Represents output that appears on the terminal screen.	user@host> show chassis alarms No alarms currently active
<i>Italic text like this</i>	<ul style="list-style-type: none"> Introduces or emphasizes important new terms. Identifies guide names. Identifies RFC and Internet draft titles. 	<ul style="list-style-type: none"> A policy <i>term</i> is a named structure that defines match conditions and actions. <i>Junos OS CLI User Guide</i> RFC 1997, <i>BGP Communities Attribute</i>

Table 2: Text and Syntax Conventions (*continued*)

Convention	Description	Examples
<i>Italic text like this</i>	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# set system domain-name <i>domain-name</i>
Text like this	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	<ul style="list-style-type: none"> To configure a stub area, include the stub statement at the [edit protocols ospf area area-id] hierarchy level. The console port is labeled CONSOLE.
< > (angle brackets)	Encloses optional keywords or variables.	stub <default-metric <i>metric</i>>;
(pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	broadcast multicast (<i>string1</i> <i>string2</i> <i>string3</i>)
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	rsvp { # Required for dynamic MPLS only
[] (square brackets)	Encloses a variable for which you can substitute one or more values.	community name members [<i>community-ids</i>]
Indentation and braces ({ })	Identifies a level in the configuration hierarchy.	[edit] routing-options { static { route default { nexthop <i>address</i> ; retain; } } }
;(semicolon)	Identifies a leaf statement at a configuration hierarchy level.	

GUI Conventions

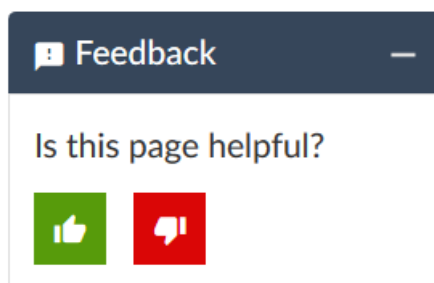
Table 2: Text and Syntax Conventions (*continued*)

Convention	Description	Examples
Bold text like this	Represents graphical user interface (GUI) items you click or select.	<ul style="list-style-type: none"> In the Logical Interfaces box, select All Interfaces. To cancel the configuration, click Cancel.
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select Protocols>Ospf .

Documentation Feedback

We encourage you to provide feedback so that we can improve our documentation. You can use either of the following methods:

- Online feedback system—Click TechLibrary Feedback, on the lower right of any page on the [Juniper Networks TechLibrary](#) site, and do one of the following:



- Click the thumbs-up icon if the information on the page was helpful to you.
- Click the thumbs-down icon if the information on the page was not helpful to you or if you have suggestions for improvement, and use the pop-up form to provide feedback.
- E-mail—Send your comments to techpubs-comments@juniper.net. Include the document or topic name, URL or page number, and software version (if applicable).

Requesting Technical Support

Technical product support is available through the Juniper Networks Technical Assistance Center (JTAC). If you are a customer with an active Juniper Care or Partner Support Services support contract, or are

covered under warranty, and need post-sales technical support, you can access our tools and resources online or open a case with JTAC.

- JTAC policies—For a complete understanding of our JTAC procedures and policies, review the *JTAC User Guide* located at <https://www.juniper.net/us/en/local/pdf/resource-guides/7100059-en.pdf>.
- Product warranties—For product warranty information, visit <https://www.juniper.net/support/warranty/>.
- JTAC hours of operation—The JTAC centers have resources available 24 hours a day, 7 days a week, 365 days a year.

Self-Help Online Tools and Resources

For quick and easy problem resolution, Juniper Networks has designed an online self-service portal called the Customer Support Center (CSC) that provides you with the following features:

- Find CSC offerings: <https://www.juniper.net/customers/support/>
- Search for known bugs: <https://prsearch.juniper.net/>
- Find product documentation: <https://www.juniper.net/documentation/>
- Find solutions and answer questions using our Knowledge Base: <https://kb.juniper.net/>
- Download the latest versions of software and review release notes: <https://www.juniper.net/customers/csc/software/>
- Search technical bulletins for relevant hardware and software notifications: <https://kb.juniper.net/InfoCenter/>
- Join and participate in the Juniper Networks Community Forum: <https://www.juniper.net/company/communities/>
- Create a service request online: <https://myjuniper.juniper.net>

To verify service entitlement by product serial number, use our Serial Number Entitlement (SNE) Tool: <https://entitlementsearch.juniper.net/entitlementsearch/>

Creating a Service Request with JTAC

You can create a service request with JTAC on the Web or by telephone.

- Visit <https://myjuniper.juniper.net>.
- Call 1-888-314-JTAC (1-888-314-5822 toll-free in the USA, Canada, and Mexico).

For international or direct-dial options in countries without toll-free numbers, see <https://support.juniper.net/support/requesting-support/>.

1

CHAPTER

Disclaimer

Ansible for Junos OS Disclaimer | 14

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 | 16

Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19

Understanding the Ansible Inventory File When Managing Devices Running Junos OS | 26

Creating and Executing Ansible Playbooks to Manage Devices Running Junos OS | 28

Understanding Ansible for Junos OS

SUMMARY

You can use Ansible to deploy and manage devices running Junos OS.

IN THIS SECTION

- [Ansible for Junos OS Overview | 16](#)
- [Benefits of Ansible and Ansible for Junos OS | 17](#)
- [Additional Resources | 17](#)

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 devices running Junos OS. You can use Ansible to perform operational and configuration tasks on devices running Junos OS, 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 devices running Junos OS:

- 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 devices running Junos OS. 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 devices running Junos OS.

Ansible typically uses a push model in which the server sends state information to the managed nodes on demand. Ansible modules, which are discrete units of code, 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 devices running Junos OS. The Juniper Networks modules are distributed through collections or roles, depending on the Ansible release. For more information about the available collections, roles, and modules, see [“Understanding the Ansible for Junos OS Collections, Roles, and Modules” on page 19](#).

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

Additional Resources

This documentation assumes that the reader is familiar with the Ansible framework. [Table 3 on page 17](#) provides resources for using Ansible to manage devices running Junos OS.

Table 3: Ansible for Junos OS Resources

Resource	Description	URL
Ansible for Junos OS documentation	Documentation detailing how to use Ansible to manage devices running Junos OS.	https://www.juniper.net/documentation/product/en_US/ansible-for-junos-os

Table 3: Ansible for Junos OS Resources (continued)

Resource	Description	URL
Ansible for Junos OS modules overview	Documentation that outlines the different modules available for managing devices running Junos OS.	“Understanding the Ansible for Junos OS Collections, Roles, and Modules” on page 19
Ansible Galaxy	Ansible Galaxy website and Juniper Networks Ansible Galaxy content	https://galaxy.ansible.com https://galaxy.ansible.com/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/
Google Groups forum	Forum that addresses questions and provides general support for Ansible for Junos OS.	http://groups.google.com/group/junos-python-ez
Juniper Networks juniper.device collection	Download site, API reference documentation, and sample playbooks for the Juniper Networks juniper.device collection.	https://galaxy.ansible.com/juniper/device https://ansible-juniper-collection.readthedocs.io/en/latest/ https://github.com/Juniper/ansible-junos-stdlib/tree/master/Samples
Juniper Networks Juniper.junos role	Download site and API reference documentation for the Juniper Networks Juniper.junos role.	https://galaxy.ansible.com/juniper/junos https://junos-ansible-modules.readthedocs.io/en/latest/

RELATED DOCUMENTATION

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

[Ansible for Junos OS Server Requirements | 35](#)

[Setting up Ansible for Junos OS Managed Nodes | 38](#)

[Creating and Executing Ansible Playbooks to Manage Devices Running Junos OS | 28](#)

Understanding the Ansible for Junos OS Collections, Roles, and Modules

SUMMARY

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

IN THIS SECTION

- [Understanding Ansible Collections, Roles, and Modules for Managing Devices Running Junos OS | 19](#)
- [How to Execute Modules on Devices Running Junos OS | 20](#)
- [Juniper Networks juniper.device Collection | 23](#)
- [Juniper Networks Juniper.junos Role | 24](#)

Understanding Ansible Collections, Roles, and Modules for Managing Devices Running Junos OS

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 devices running Junos OS. The Juniper Networks Ansible modules are grouped and distributed through Ansible roles and collections, which are hosted in the [Ansible Galaxy](#) repository. [Table 4 on page 20](#) outlines the different content sets available for managing devices running Junos OS.

Table 4: Ansible Content Sets for Managing Devices Running Junos OS

Content Set	Description	Ansible Releases
Ansible core modules	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.device collection	Collection for managing devices running Junos OS, which Juniper Networks provides and manages as an open-source project.	Ansible 2.10 and later
Juniper.junos role	Role for managing devices running Junos OS, which Juniper Networks provides, maintains, and supports. This role is superseded by the juniper.device collection.	Ansible 2.1 and later
junipernetworks.junos collection	Collection for managing devices running Junos OS, which Ansible provides, maintains, and supports.	Ansible 2.10 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 devices running Junos OS 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. Although the **Juniper.junos** role can coexist with the **juniper.device** collection and will work in later releases, we recommend that you use the **juniper.device** collection, because new features are only being added to the collection going forward.

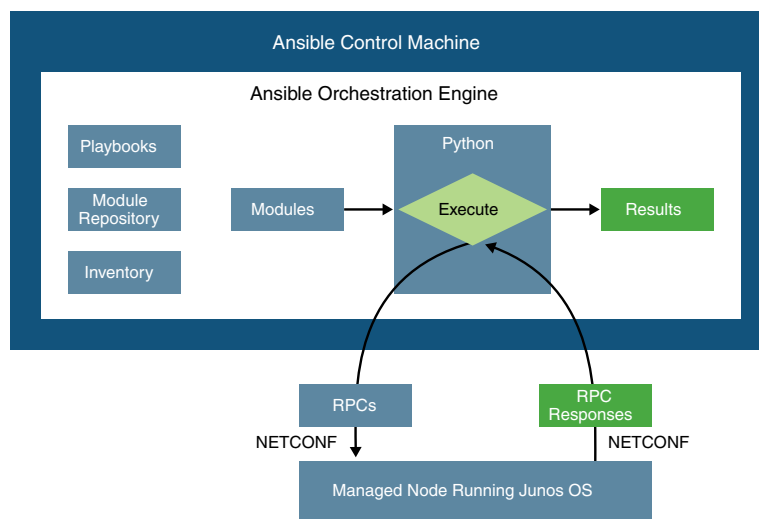
How to Execute Modules on Devices Running Junos OS

To use collections and roles that are hosted in the Ansible Galaxy repository, you must first install Ansible on the control node and then install the collection or role. For more information about installing the **juniper.device** collection or **Juniper.junos** role, see [“Ansible for Junos OS Server Requirements” on page 35](#).

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 and the **Juniper.junos** role, 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 device running Junos OS. [Figure 1 on page 21](#) illustrates the communication between the Ansible control node and a managed device running Junos OS.

Figure 1: Ansible Communication with Devices Running Junos OS



To use the **juniper.device** or **Juniper.junos** modules, the playbook or command must:

- Specify the collection or role—To specify the collection or role, include the **collections** or **roles** key in the play. Alternatively, you can omit the **collections** key and instead reference collection content by its fully qualified collection name (FQCN).
- 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:
 - [Connecting to Devices Running Junos OS Using Ansible on page 43](#)
 - [Authenticating Users Executing Ansible Modules on Devices Running Junos OS on page 51](#)

You can execute Ansible modules using any user account that has access to the managed device running Junos OS. 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Retrieve facts from device running Junos OS
      facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      debug:
        var: junos.version
```

Alternatively, you can also reference collection content by its fully qualified collection name, for example:

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

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

```

- name: Print version
  debug:
    var: junos.version

```

Similarly, the following playbook executes the **Juniper.junos** role's **juniper_junos_facts** module to perform the same operation.

```

---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
  roles:
    - Juniper.junos

  tasks:
    - name: Retrieve facts from device running Junos OS
      juniper_junos_facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      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@host$ 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 and includes Ansible modules that enable you to manage devices running Junos OS.

[Table 5 on page 24](#) 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/en/latest/>.

Table 5: juniper.device Collection Modules

juniper.device Module Name	Description
command	Execute CLI commands on the device running Junos OS and save the output locally.
config	Manage the configuration of devices running Junos OS.
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 from Ansible.
ping	Execute the ping command on devices running Junos OS.
pmtud	Perform path MTU discovery on devices running Junos OS.
rpc	Execute Junos OS RPCs.
software	Install a Junos OS software package and reboot a device running Junos OS.
system	Perform system operations on devices running Junos OS, including resetting, rebooting, or shutting down the device.
srx_cluster	Create an SRX Series chassis cluster for cluster-capable SRX Series Services Gateways running Junos OS.
table	Use Junos PyEZ operational Tables and Views to retrieve operational information from devices running Junos OS.

Juniper Networks Juniper.junos Role

Juniper Networks provides the [Juniper.junos](#) Ansible role, which is hosted on the [Ansible Galaxy](#) website and includes Ansible modules that enable you to manage devices running Junos OS. Starting in Ansible 2.10, the **juniper.device** collection supersedes the **Juniper.junos** role. The modules in the collection have the same functionality and parameters as the modules in the role, with the exception of the **provider** parameter. Although the collection and role can coexist on the same platform and you can continue to use the **Juniper.junos** modules in later releases, we recommend that you use the **juniper.device** collection, because new features are only being added to the collection going forward.

Table 6 on page 25 summarizes the modules in the **Juniper.junos** role. For the most current list, documentation, and usage examples for the modules, see <https://junos-ansible-modules.readthedocs.io/en/latest/>.

NOTE: Starting in **Juniper.junos** Release 2.0.0, the **Juniper.junos** role includes an enhanced set of modules. Each new module replaces the functionality of one or more existing modules. The enhanced modules support a common set of connection and authentication parameters, aliases that enable you to specify the same connection and authentication-related options as the core modules, and the ability to specify the parameters inside a provider dictionary.

Table 6: Juniper.junos Role Modules

Module Name	Description	Deprecated Modules as of Release 2.0.0
juniper_junos_command	Execute CLI commands on the device running Junos OS and save the output locally.	junos_cli
juniper_junos_config	Manage the configuration of devices running Junos OS.	junos_commit junos_get_config junos_install_config junos_rollback
juniper_junos_facts	Retrieve device-specific information from the remote host, including the Junos OS version, serial number, and hardware model number.	junos_get_facts
juniper_junos_jsnapy	Execute Junos Snapshot Administrator in Python (JSNAPy) tests from Ansible.	junos_jsnapy
juniper_junos_ping	Execute the ping command on devices running Junos OS.	junos_ping
juniper_junos_pmtud	Perform path MTU discovery on devices running Junos OS.	junos_pmtud
juniper_junos_rpc	Execute Junos OS RPCs.	junos_rpc
juniper_junos_software	Install a Junos OS software package and reboot a device running Junos OS.	junos_install_os

Table 6: Juniper.junos Role Modules (*continued*)

Module Name	Description	Deprecated Modules as of Release 2.0.0
<code>juniper_junos_system</code>	Perform system operations on devices running Junos OS, including resetting, rebooting, or shutting down the device.	<code>junos_shutdown</code> <code>junos_zeroize</code>
<code>juniper_junos_srx_cluster</code>	Create an SRX Series chassis cluster for cluster-capable SRX Series Services Gateways running Junos OS.	<code>junos_srx_cluster</code>
<code>juniper_junos_table</code>	Use Junos PyEZ operational Tables and Views to retrieve operational information from devices running Junos OS.	<code>junos_get_table</code>

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the Juniper.junos role includes an enhanced set of modules.

RELATED DOCUMENTATION

[Understanding Ansible for Junos OS | 16](#)

[Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51](#)

[Connecting to Devices Running Junos OS Using Ansible | 43](#)

Understanding the Ansible Inventory File When Managing Devices Running Junos OS

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 devices running Junos OS 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 devices running Junos OS 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/user_guide/intro_inventory.html .

RELATED DOCUMENTATION

[Understanding Ansible for Junos OS | 16](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

[Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51](#)

[Creating and Executing Ansible Playbooks to Manage Devices Running Junos OS | 28](#)

[Troubleshooting Ansible Connection Errors When Managing Devices Running Junos OS | 200](#)

Creating and Executing Ansible Playbooks to Manage Devices Running Junos OS

SUMMARY

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

IN THIS SECTION

- [Creating a Playbook | 29](#)
- [Executing the Playbook | 33](#)

Juniper Networks supports using Ansible to manage devices running Junos OS, and Ansible and Juniper Networks provide Ansible modules that enable you to perform operational and configuration tasks on the devices. This topic outlines how to create a simple Ansible playbook to execute Juniper Networks modules on devices running Junos OS.

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 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

The Juniper Networks Ansible modules are distributed through the **juniper.device** collection and the **Juniper.junos** role, which are hosted on [Ansible Galaxy](#). To use the Juniper Networks modules in your playbook, you must install the collection or role on the Ansible control node and include the collection or role in the playbook. For more information about the Juniper Networks collection, role, and modules, see [“Understanding the Ansible for Junos OS Collections, Roles, and Modules” on page 19](#).

The Juniper Networks modules do not require Python on devices running Junos OS, because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on devices running Junos OS, 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, which still executes the modules locally on the Ansible control node but instead 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 device running Junos OS. 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. Because YAML is white-space sensitive and indentation is significant, you should always use spaces rather than tabs when creating playbooks. 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/user_guide/playbooks.html.

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

Creating a Playbook

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

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 devices running Junos OS.

- 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. Include the **juniper.device** collection or the **Juniper.junos** role, as appropriate for your Ansible setup.

- On Ansible control nodes running Ansible 2.10 or later that have the **juniper.device** collection installed, include the collection.

```

---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

```

NOTE: You can omit the **collections** key and instead reference collection content by its fully qualified collection name (FQCN),

- On Ansible control nodes that have the **Juniper.junos** role installed, include the role.

```

---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
  roles:
    - Juniper.junos

```

The remaining steps use the **juniper.device** collection and module names. To use the **Juniper.junos** role, update the final playbook to use the role and role module names.

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
  collections:
    - juniper.device

  tasks:

```

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

```

---
- name: Get Device Facts
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Checking NETCONF connectivity
      wait_for:
        host: "{{ inventory_hostname }}"
        port: 830
        timeout: 5

```

10. Create tasks that use the Juniper Networks modules, and provide any necessary connection or 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
  collections:
    - juniper.device

  tasks:
    - name: Checking NETCONF connectivity
      wait_for:
        host: "{{ inventory_hostname }}"
        port: 830
        timeout: 5

    - name: Retrieving information from devices running Junos OS
      facts:
        savedir: "{{ playbook_dir }}/output"

    - name: Print version

```



```

debug:
  var: junos.version

```

11. (Optional) Define additional plays as needed by repeating steps 3 through 10.

Executing 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 [Checking NETCONF connectivity] *****
ok: [dc1a.example.net]

TASK [Retrieving information from devices running Junos OS] *****
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 | 16](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

[Understanding the Ansible Inventory File When Managing Devices Running Junos OS | 26](#)

[Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51](#)

3

CHAPTER

Installing Ansible for Junos OS

[Ansible for Junos OS Server Requirements](#) | 35

[Setting up Ansible for Junos OS Managed Nodes](#) | 38

Ansible for Junos OS Server Requirements

Juniper Networks supports using Ansible to manage devices running Junos OS and provides Ansible modules that you can use to perform operational and configuration tasks on the devices. The Juniper Networks modules are distributed through the following Ansible collections and roles that are hosted on the [Ansible Galaxy](#) website:

- [juniper.device](#) collection
- [Juniper.junos](#) role

Ansible supports Ansible Content Collections, or collections, starting in Ansible 2.10. With the introduction of the **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. Although the **Juniper.junos** role can coexist with the **juniper.device** collection and will work in later releases, new features are only being added to the **juniper.device** collection going forward.

You install Ansible on a control node with a Unix-like operating system. You can install Ansible and the Juniper Networks collection or role directly on the control node or you can use 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 manually install Ansible, the **juniper.device** collection or **Juniper.junos** role, and any prerequisite software on the Ansible control node, see the following sections:

- [Prerequisite Software on page 36](#)
- [Installing the juniper.device Collection and the Juniper.junos Role on page 36](#)

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

- [Using the Ansible for Junos OS Docker Image on page 37](#)

Prerequisite Software

Before you install the Juniper Networks **juniper.device** collection or **Juniper.junos** role and begin using Ansible to manage devices running Junos OS, ensure that the Ansible control node has the following software installed:

- Python 3.7 or later (Recommended version is Python 3.8)
- Ansible 2.5 or later (Ansible 2.10 or later is required to use Ansible collections)
- 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 [xmldict](#) Python module (required for the **juniper.device** collection).
- Junos Snapshot Administrator in Python (JSNAPy) Release 1.3.6 or later (required to use the **jsnapy** and **juniper_junos_jsnapy** modules).

For installation instructions and current information about JSNAPy, see:

- [JSNAPy GitHub repository](#)
- [Junos Snapshot Administrator in Python Documentation](#)

Installing the juniper.device Collection and the Juniper.junos Role

Ansible control nodes running Ansible 2.9 or earlier only support installing the **Juniper.junos** role. On Ansible control nodes running Ansible 2.10 or later, you can install and use the **juniper.device** collection or the **Juniper.junos** role. However, we recommend migrating playbooks to use the collection.

- To install the **juniper.device** collection from the [Ansible Galaxy](#) website, 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:1.0.0' to
'/home/user/.ansible/collections/ansible_collections/juniper/device'
Downloading https://galaxy.ansible.com/download/juniper-device-1.0.0.tar.gz to
/home/user/.ansible/tmp/ansible-local-23916uzdfbjsk/tmp4nhxnw3v
juniper.device (1.0.0) was installed successfully
```

- To install the **Juniper.junos** role from the [Ansible Galaxy](#) website, issue the **ansible-galaxy install** command and specify the **Juniper.junos** role.

```
user@ansible-cn:~$ sudo ansible-galaxy install Juniper.junos
```

```
- downloading role 'junos', owned by Juniper
- downloading role from
https://github.com/Juniper/ansible-junos-stdlib/archive/2.4.3.tar.gz
- extracting Juniper.junos to /etc/ansible/roles/Juniper.junos
- Juniper.junos (2.4.3) was installed successfully
```

Using 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.junos** role and the **juniper.device** collection (depending on the release), 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](https://github.com/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, 2.0.0.

```
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

[Setting up Ansible for Junos OS Managed Nodes | 38](#)

[Understanding Ansible for Junos OS | 16](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Setting up Ansible for Junos OS Managed Nodes

Juniper Networks supports using Ansible to manage devices running Junos OS 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 devices running Junos OS, 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 device running Junos OS. 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 devices running Junos OS, see the *User Access and Authentication Administration Guide*.

Juniper Networks provides modules that enable you to connect to devices running Junos OS 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 devices running Junos OS when you use Ansible to access the device using the different connection protocols.

1. [Enabling NETCONF on Devices Running Junos OS | 39](#)
2. [Satisfying Requirements for SSHv2 Connections | 39](#)
3. [Configuring Telnet Service on Devices Running Junos OS | 40](#)

Enabling NETCONF on Devices Running Junos OS

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

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
```

Satisfying 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 devices running Junos OS, the most convenient way to access a device is to configure SSH keys.

To establish an SSHv2 connection with a device running Junos OS, 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 device running Junos OS and satisfying the requirements for establishing an SSH session, see the *NETCONF XML Management Protocol Developer Guide*.

Configuring Telnet Service on Devices Running Junos OS

The Juniper Networks Ansible modules can telnet directly to a device running Junos OS. To telnet to a device running Junos OS, 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 | 35](#)

[Understanding Ansible for Junos OS | 16](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

4

CHAPTER

Using Ansible to Connect to Devices Running Junos OS

Connecting to Devices Running Junos OS Using Ansible | 43

Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51

Connecting to Devices Running Junos OS Using Ansible

SUMMARY

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

IN THIS SECTION

- [Connection Methods Overview | 43](#)
- [Understanding Local and Persistent Ansible Connections | 45](#)
- [Connecting to a Device Using SSH | 46](#)
- [Connecting to a Device Using Telnet | 49](#)
- [Connecting to a Device Using a Serial Console Connection | 50](#)

Juniper Networks provides Ansible modules that you can use to manage devices running Junos OS. The Juniper Networks modules are distributed through the [juniper.device](#) collection and the [Juniper.junos](#) role, which are hosted on [Ansible Galaxy](#). The modules can connect to devices running Junos OS using different protocols and Ansible connections, which are described in this document.

Connection Methods Overview

The Juniper Networks Ansible modules enable you to connect to a device running Junos OS 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 device running Junos OS. 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 Networks modules use SSH to connect to a device. To use telnet or a serial console connection, set the module's **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 7 on page 44 summarizes the connection modes, their default values for certain parameters, any required Junos OS configuration, and the **Juniper.junos** release in which support for that connection mode was first introduced. The **juniper.device** modules support all connection modes as of their initial release.

Table 7: Connection Modes for the Juniper Networks Ansible Modules

Connection Mode	Value of mode Argument	Default Port	Required Junos OS Configuration	First Supported Release (Juniper.junos)
NETCONF over SSH (default)	-	830	[edit system services] netconf { ssh; }	1.0.0
Serial console connection	serial	/dev/ttyUSB0	-	2.0.0
SSH through a console server	-	22	-	2.2.0
Telnet to device running Junos OS	telnet	23	[edit system services] telnet;	1.4.0 Default port added in 2.0.0
Telnet through a console server	telnet	23	-	1.4.0

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 Networks modules to manage a device running Junos OS, the remote device must be able to authenticate the Ansible user using credentials appropriate for the given protocol. For more information, see [“Authenticating Users Executing Ansible Modules on Devices Running Junos OS” on page 51.](#)

The Juniper Networks modules support different Ansible connections when connecting to devices running Junos OS, 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 45.](#)

Understanding Local and Persistent Ansible Connections

The Juniper Networks modules do not require Python on devices running Junos OS because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on devices running Junos OS, 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 8 on page 45](#) summarizes the Ansible connections and the content sets that support them.

Table 8: 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
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

NOTE: Ansible has deprecated **connection: local**. Therefore, when you use the **juniper.device** collection modules, we recommend that you use **connection: juniper.device.pyez** in your playbook to avoid issues in the event that Ansible removes support for local connections in a later release.

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 [“Authenticating Users Executing Ansible Modules on Devices Running Junos OS” on page 51](#).

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: dcl
  connection: juniper.device.pyez
  gather_facts: no
  collections:
    - juniper.device

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

  vars_files:
    - vault-vars.yaml

  tasks:
    - name: Retrieve facts from devices running Junos OS
      facts:
        savedir: "{{ playbook_dir }}"

    - name: Get hardware inventory
      command:
        commands: "show chassis hardware"
        dest_dir: "{{ playbook_dir }}"

```

Connecting to a Device Using SSH

The Juniper Networks Ansible modules support using SSH to connect to a device running Junos OS. 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 [“Authenticating Users Executing Ansible Modules on Devices Running Junos OS” on page 51](#). To establish a NETCONF session over SSH, you must also satisfy the requirements outlined in [“Setting up Ansible for Junos OS Managed Nodes” on page 38](#).

The Juniper Networks modules automatically query the default SSH configuration file at `~/.ssh/config`, if one exists, unless the `ssh_config` parameter specifies a different configuration file. When using SSH to connect to a device running Junos OS 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, SSH private keys with empty passphrases are not recommended.

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

```
---
- name: Get Device Facts
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Retrieve facts from devices running Junos OS
      facts:
        savedir: "{{ playbook_dir }}"
    - name: Print version
      debug:
        var: junos.version
```

The Juniper Networks Ansible modules also enable you to connect to a device running Junos OS through an SSH connection to a console server. In this case, you must specify the login credentials for both the device running Junos OS and the console server. Use the **user** and **passwd** parameters to specify the Junos OS login credentials, and 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 device running Junos OS. The module authenticates with the console server and then the device running Junos OS. 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: dcl_con
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  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

tasks:
- name: Retrieve facts from devices running Junos OS
  facts:
    cs_user: "{{ CS_USER }}"
    cs_passwd: "{{ CS_PASSWORD }}"
    user: "{{ JUNOS_USER }}"
    passwd: "{{ JUNOS_PASSWORD }}"
    savedir: "{{ playbook_dir }}"
- name: Print version
  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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

tasks:
- name: Retrieve facts from devices running Junos OS
  facts:
    ssh_config: "/home/admin/.ssh/config_dc"
    savedir: "{{ playbook_dir }}"
- name: Print version

```



```

debug:
  var: junos.version

```

Connecting to a Device Using Telnet

The Juniper Networks modules enable you to connect to a device running Junos OS 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 device running Junos OS using port 7016, retrieves the device facts, and saves them to a file. The module uses the default user and prompts for the login password.

```

---
- name: Get Device Facts
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

  tasks:
    - name: Retrieve facts from devices running Junos OS
      facts:
        passwd: "{{ DEVICE_PASSWORD }}"
        mode: "telnet"
        port: "7016"
        savedir: "{{ playbook_dir }}"
    - name: Print version

```

```

debug:
  var: junos.version

```

Connecting to a Device Using a Serial Console Connection

The Juniper Networks modules enable you to connect to a device running Junos OS 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 device running Junos OS through the **CONSOLE** port. For detailed instructions about connecting to the **CONSOLE** port on a device running Junos OS, see the hardware documentation for your specific device.

To connect to a device running Junos OS 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 device running Junos OS through the **CONSOLE** port and then loads and commits an initial configuration. The module uses the default user and prompts for the login password.

```

---
- name: Load Initial Configuration
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

  tasks:
    - name: Load initial configuration and commit
      config:
        passwd: "{{ DEVICE_PASSWORD }}"
        mode: "serial"
        load: "merge"
        src: "configs/junos.conf"

```

```
register: response
- name: Print the response
  debug:
    var: response
```

RELATED DOCUMENTATION

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

[Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51](#)

Authenticating Users Executing Ansible Modules on Devices Running Junos OS

IN THIS SECTION

- [Authentication Overview | 51](#)
- [Understanding the Default Values for Juniper Networks Modules | 53](#)
- [How to Define Authentication Parameters in the vars: Section for Local and Persistent Connections | 54](#)
- [How to Define the provider Parameter in Juniper.junos Modules | 55](#)
- [How to Authenticate the User Using SSH Keys | 57](#)
- [How to Authenticate the User Using a Playbook or Command-Line Password Prompt | 59](#)
- [How to Authenticate the User Using an Ansible Vault-Encrypted File | 61](#)
- [How to Authenticate Through a Console Server | 63](#)

Authentication Overview

Juniper Networks provides Ansible modules that you can use to manage devices running Junos OS. The Juniper Networks modules are distributed through the **juniper.device** collection and the **Juniper.junos** role, which are hosted on [Ansible Galaxy](#).

The Juniper Networks modules enable you to directly connect to and manage devices running Junos OS 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 devices running Junos OS, 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, SSH private keys with empty passphrases are not recommended. 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 53](#). 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, unless you 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.
- **Module arguments**—The `juniper.device` and `Juniper.junos` modules support specifying connection and authentication-related options for local connections (**connection: local**) as top-level module arguments. Additionally, `Juniper.junos` modules support using a provider dictionary in the module arguments as described in [“How to Define the provider Parameter in Juniper.junos Modules” on page 55](#).
- **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 54](#).

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

Understanding the Default Values for Juniper Networks Modules

You can explicitly define the connection and authentication parameters for modules that manage devices running Junos OS. If you do not define a parameter, the module uses a default value in some cases.

[Table 9 on page 53](#) outlines the default values and variable precedence for common connection parameters for modules in the **juniper.device** collection and **Juniper.junos** role. For information about the arguments accepted for the individual modules, see the API reference documentation for that module.

Table 9: 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.	<code>{{ inventory_hostname }}</code>
passwd	password	The user's password or SSH key passphrase used to authenticate with the managed device.	<ol style="list-style-type: none"> 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.	<code>~/.ssh/config</code>
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.	<ol style="list-style-type: none"> 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.	<ol style="list-style-type: none"> 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 device running Junos OS. 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 9 on page 53](#). 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, 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.

NOTE: Ansible has deprecated **connection: local**. Therefore, when you use the **juniper.device** collection modules, we recommend that you use **connection: juniper.device.pyez** in your playbook to avoid issues in the event that Ansible removes support for local connections in a later release.

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 53](#) apply to both types of connections. However, when you define connection and authentication parameters within a play for persistent connections, you must define the parameters in the **vars:** section as opposed to defining the parameters as top-level module arguments in each task because there is only a single connection, and thus the parameters must apply to all tasks in that play. For local connections, you can define the parameters either in the **vars:** section or as module arguments. If you define the parameters in both places, the module arguments take precedence.

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 devices running Junos OS
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"

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

How to Define the provider Parameter in Juniper.junos Modules

Starting in **Juniper.junos** Release 2.0.0, the **Juniper.junos** modules support the **provider** parameter in addition to supporting individual top-level module arguments for each of the connection and

authentication-related parameters. The **provider** parameter enables you to define the connection and authentication parameters for multiple modules in one place and easily pass those values to the modules that use them. Additionally, if you need to update the parameters later, you only need to make the update in a single location.

NOTE: The **juniper.device** collection modules do not support using the **provider** parameter.

The **provider** argument accepts a dictionary that contains the connection details required to connect to and authenticate with a device. The **host** argument is always required for a connection, but you do not have to explicitly specify a value if the module uses the default value for **host**. The dictionary can optionally define additional parameters required for the connection, including **user**, **passwd**, and **ssh_private_key_file**, among others. For information about the arguments accepted for the individual modules, see the API reference documentation for that module.

In the following example, the **credentials** variable is a dictionary that defines the **host**, **user**, and **passwd** parameters:

```
vars:
  credentials:
    host: "{{ inventory_hostname }}"
    user: "{{ ansible_user }}"
    passwd: "{{ ansible_password }}"
```

The following sample playbook uses the single **provider** argument to pass the connection details to the **juniper_junos_facts** module instead of defining individual module arguments. As you add additional tasks that use the **Juniper.junos** modules, you can then reference the same dictionary for each module.

```
---
- name: Get Device Facts
  hosts: dcl
  connection: local
  gather_facts: no
  roles:
    - Juniper.junos

vars:
  credentials:
    host: "{{ inventory_hostname }}"
    user: "{{ ansible_user }}"
    passwd: "{{ ansible_password }}"
```



```

tasks:
  - name: Retrieve facts from devices running Junos OS
    juniper_junos_facts:
      provider: "{{ credentials }}"
      savedir: "{{ playbook_dir }}"
  - name: Print version
    debug:
      var: junos.version

```

How to Authenticate the User Using SSH Keys

IN THIS SECTION

- [Generating and Configuring the SSH Keys | 57](#)
- [Using SSH Keys in Ansible Playbooks | 58](#)

The Juniper Networks **juniper.device** and **Juniper.junos** modules enable you to use SSH keys to connect to a device running Junos OS 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 device running Junos OS or the console server connected to the device running Junos OS.

Generating and Configuring 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 devices running Junos OS or a console server connected to a device running Junos OS.

The easiest method to configure the public key on a device running Junos OS 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>
```

Using 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, unless you 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 device running Junos OS 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 device running Junos OS 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:

```
juniper.device.facts:
  ssh_private_key_file: "/home/user/.ssh/id_rsa_alternate"
  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 device running Junos OS 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:

```
juniper.device.facts:
  ssh_private_key_file: "/home/user/.ssh/id_rsa_dc"
  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 59 and [“How to Authenticate the User Using an Ansible Vault-Encrypted File”](#) on page 61.

For instructions on using SSH keys to connect to a console server, see [“How to Authenticate Through a Console Server”](#) on page 63.

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.

```
tasks:
  - name: Retrieve facts from devices running Junos OS
    juniper.device.facts:
      user: "{{ USERNAME }}"
      passwd: "{{ DEVICE_PASSWORD }}"
      savedir: "{{ playbook_dir }}"
  - name: Print facts
    debug:
      var: junos.version
```

3. 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 devices running Junos OS
      juniper.device.facts:
        savedir: "{{ playbook_dir }}"
    - name: Print facts
      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
3730326165666565356665343137313161234569336336640a653939633331663935376362376666
65653737653262363235353261626135312345663665396262376339623737366238653436306663
6430376633306339360a343065363331313532633036343866376330623634653538353132314159
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: dcl
  connection: local
  gather_facts: no

  vars_files:
    - vault-vars.yaml

  tasks:
    - name: Retrieve facts from devices running Junos OS
      juniper.device.facts:
        user: "{{ root_username }}"
        passwd: "{{ root_password }}"
        savedir: "{{ playbook_dir }}"
```

```
- name: Print version
  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.

- 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 devices running Junos OS 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 device running Junos OS. You can provide either a device password or a password-protected SSH key file for the console server authentication.

To connect to a device running Junos OS 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: dcl_con
  connection: local
  gather_facts: no

  vars_files:
    - vault-vars.yaml

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

RELATED DOCUMENTATION

[Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS | 203](#)

[Connecting to Devices Running Junos OS Using Ansible | 43](#)

[Understanding Ansible for Junos OS | 16](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

5

CHAPTER

Using Ansible to Manage Device Operations

Using Ansible to Retrieve Facts from Devices Running Junos OS | **66**

Using Ansible to Execute Commands and RPCs on Devices Running Junos OS | **70**

Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS | **79**

Using Ansible to Halt, Reboot, or Shut Down Devices Running Junos OS | **86**

Using Ansible to Install Software on Devices Running Junos OS | **98**

Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings | **116**

Using Junos Snapshot Administrator in Python (JSNAPy) in Ansible Playbooks | **125**

Using Ansible to Retrieve Facts from Devices Running Junos OS

Juniper Networks supports using Ansible to manage devices running Junos OS and provides Ansible modules that enable you to 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 devices running Junos OS, 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 modules that enable you to gather device facts, including the active configuration, from devices running Junos OS. [Table 10 on page 66](#) outlines the available modules. The modules use 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 10: Modules to Gather Facts

Content Set	Module Name
juniper.device collection	facts
Juniper.junos role	juniper_junos_facts

The [facts](#) and [juniper_junos_facts](#) modules return the device facts in the [ansible_facts.junos](#) dictionary. The modules also enable 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 Juniper Networks 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 **facts** module in the **juniper.device** collection 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
  collections:
    - juniper.device

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

By default, the **facts** and **juniper_junos_facts** modules do 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. If the **savendir** option is included, the configuration data is *not* written to the file.

TIP: To use Ansible to retrieve configuration data from a device running Junos OS and save the data to a file, use the **config** or **juniper_junos_config** module instead of the **facts** or **juniper_junos_facts** module. For more information, see [“Using Ansible to Retrieve or Compare Junos OS Configurations” on page 156](#).

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
  collections:
    - juniper.device

  tasks:
    - name: Retrieve device facts and configuration and save facts to file
      facts:
        savendir: "{{ playbook_dir }}"
        config_format: "xml"
        register: result

    - name: Print configuration
      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: [dcla.example.net]

TASK [Print configuration] *****
ok: [dcla.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 *****
dcla.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, Roles, and Modules | 19](#)

[Using Ansible to Execute Commands and RPCs on Devices Running Junos OS | 70](#)

[Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS | 79](#)

Using Ansible to Execute Commands and RPCs on Devices Running Junos OS

SUMMARY

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

IN THIS SECTION

- [How to Execute Commands with the Juniper Networks Modules | 71](#)
- [How to Execute RPCs with the Juniper Networks Modules | 72](#)
- [Understanding the Module Response | 74](#)
- [How to Specify the Format for the Command or RPC Output | 75](#)
- [How to Save the Command or RPC Output to a File | 76](#)

Juniper Networks supports using Ansible to manage devices running Junos OS and provides Ansible modules that enable you to execute operational mode commands and remote procedure calls (RPCs) on the devices. [Table 11 on page 70](#) outlines the modules.

Table 11: Command and RPC Modules

Content Set	Module Name
juniper.device collection	command
	rpc
Juniper.junos role	juniper_junos_command
	juniper_junos_rpc

The following sections discuss how to 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 **table** or **juniper_junos_table** module with custom or predefined Junos PyEZ operational tables. For more information, see [“Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS”](#) on page 79.

How to Execute Commands with the Juniper Networks Modules

The **command** and **juniper_junos_command** modules enable you to execute operational mode commands on devices running Junos OS. The modules require one argument, **commands**, which is a list of one or more Junos OS operational mode commands to execute on the device.

NOTE: Starting in **Juniper.junos** Release 2.0.0, the **juniper_junos_command** module replaces the functionality of the **junos_cli** module.

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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

    - name: Print response
```

```

debug:
  var: junos_result

```

For information about the module's response and output format, see [“Understanding the Module Response” on page 74](#) and [“How to Specify the Format for the Command or RPC Output” on page 75](#).

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 device running Junos OS. The server returns the response using Junos XML elements enclosed within the response tag element.

The `rpc` and `juniper_junos_rpc` modules enable you to execute RPCs on devices running Junos OS. The modules require one argument, `rpcs`, which is a list of one or more Junos OS RPCs to execute on the device.

NOTE: Starting in `Juniper.junos` Release 2.0.0, the `juniper_junos_rpc` module replaces the functionality of the `junos_rpc` module.

The following playbook executes the `get-interface-information` RPC, which is equivalent to the `show interfaces` operational mode command, on each device in the inventory group and displays the module response in standard output. 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
  collections:
    - juniper.device

  tasks:
    - name: Get interface information
      rpc:

```



```

    rpcs: "get-interface-information"
    register: junos_result

- name: Print response
  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 74 and "[How to Specify the Format for the Command or RPC Output](#)" on page 75.

The `rpc` and `juniper_junos_rpc` modules support 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, or it can be 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 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-ldp-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: dcl1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:

```

```

- name: Get device information
  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
  debug:
    var: junos_result

```

Understanding the Module Response

The Juniper Networks command and 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.

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.

How to Specify the Format for the Command or RPC Output

The Juniper Networks 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** and **juniper_junos_command** modules return the command output in text format, and the **rpc** and **juniper_junos_rpc** modules return 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. To request text format, Junos XML elements, or JSON format, use **'text'**, **'xml'**, or **'json'** respectively. The requested format must be supported by the device on which the command or RPC is executed.

The **formats** parameter takes either a string or a list of strings. When you execute multiple commands or RPCs and only specify 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

    - name: Print response
      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
  collections:
    - juniper.device

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

    - name: Print response
      debug:
        var: junos_result
```

How to Save the Command or RPC Output to a File

When you use the Juniper Networks modules to execute a command or RPC on a device, you can save the returned data in 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.

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
  collections:
    - juniper.device

  tasks:
    - name: Get software and uptime information
      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
  collections:
    - juniper.device

  tasks:
    - name: Get software and uptime information
      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`

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
  collections:
    - juniper.device

  tasks:
    - name: Get software and uptime information
      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.

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 devices return a significant amount of data.

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_command module replaces the functionality of the junos_cli module.
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_rpc module replaces the functionality of the junos_rpc module.

RELATED DOCUMENTATION

[Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS | 79](#)

[Using Ansible to Retrieve Facts from Devices Running Junos OS | 66](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Using Ansible with Junos PyEZ Tables to Retrieve Operational Information from Devices Running Junos OS

IN THIS SECTION

- [Module Overview | 80](#)
- [Understanding Junos PyEZ Tables | 80](#)
- [How to Use the Juniper Networks Ansible Modules with Junos PyEZ Tables | 81](#)
- [Specifying RPC Arguments | 84](#)

Module Overview

Junos PyEZ operational (op) Tables provide a simple and efficient way to extract information from complex operational command output. Juniper Networks provides Ansible modules that enable you to leverage Junos PyEZ op Tables from within Ansible playbooks. [Table 12 on page 80](#) outlines the modules.

Table 12: Junos PyEZ Table Modules

Content Set	Module Name
juniper.device collection	table
Juniper.junos role	juniper_junos_table

NOTE: Starting in [Juniper.junos](#) Release 2.0.0, the [juniper_junos_table](#) module replaces the functionality of the [junos_get_table](#) module.

NOTE: The modules do not work with configuration Tables and Views.

Understanding Junos PyEZ Tables

[Junos PyEZ](#) is a microframework for Python that enables you to manage and automate devices running Junos OS. 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 devices running Junos OS.

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 devices running Junos OS, the [table](#) and [juniper_junos_table](#) modules can use Junos PyEZ Tables to retrieve data from a device. The modules can reference the predefined operational Tables and Views that are included with the Junos PyEZ distribution, or they 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](#)

- [Junos PyEZ Predefined Operational Tables and Views](#)

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

The `table` and `juniper_junos_table` modules 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
    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 devices running Junos OS. 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

    - name: Print response
      debug:
        var: result

```

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

```

TASK [Print response] *****
ok: [dcl1.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 devices running Junos OS. 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

    - name: Print response
      debug:
        var: result

```

Specifying 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 command options or pass additional options and arguments to the RPC when calling the **get()** method.

The **table** and **juniper_junos_table** modules also enable you to override the default command 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 and 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 devices running Junos OS. 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 for only the ge-1/0/0 interface.

```
---
- name: Get interface information
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Get interface information for Ethernet interfaces
      table:
        file: "ethport.yml"
        kwargs:
```

```

    interface_name: "ge-1/0/0"
    register: result

- name: Print response
  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 [Using Junos PyEZ Operational Tables and Views](#).

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_table module replaces the functionality of the junos_get_table module.

RELATED DOCUMENTATION

[Using Ansible to Execute Commands and RPCs on Devices Running Junos OS | 70](#)

[Using Ansible to Retrieve Facts from Devices Running Junos OS | 66](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Using Ansible to Halt, Reboot, or Shut Down Devices Running Junos OS

SUMMARY

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

IN THIS SECTION

- [Using Ansible to Halt, Reboot, or Shut Down Devices | 86](#)
- [How to Perform a Halt, Reboot, or Shut Down with a Delay or at a Specified Time | 88](#)
- [How to Specify the Target Routing Engine | 89](#)
- [How to Reboot a VM Host | 90](#)
- [Example: Using Ansible to Reboot Devices Running Junos OS | 91](#)

Using Ansible to Halt, Reboot, or Shut Down Devices

Juniper Networks supports using Ansible to manage devices running Junos OS and provides Ansible modules that enable you to halt, reboot, or shut down a device. [Table 13 on page 86](#) outlines the available modules.

Table 13: Modules to Halt, Reboot, or Shut Down Devices

Content Set	Module Name
juniper.device collection	system
Juniper.junos role	juniper_junos_system

NOTE: Starting in [Juniper.junos](#) Release 2.0.0, the [juniper_junos_system](#) module replaces the functionality of the [junos_shutdown](#) and [junos_zeroize](#) modules.

You can use the modules to request the following operations on devices running Junos OS. By default, the modules immediately execute the requested operation and perform 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** and **juniper_junos_system** modules require one argument, **action**, which defines the action that the module performs. [Table 14 on page 87](#) 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 ["Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings" on page 116](#).

Table 14: system and juniper_junos_system 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 devices running Junos OS
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

```

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

To delay the halt, reboot, or shut down operation by a specified number of minutes, 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 devices running Junos OS
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

To schedule the halt, reboot, or shutdown at a specific 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.
- **yyymmddhhmm**—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
    system:
      action: "shutdown"
      at: "22:30"
```


How to Specify the Target Routing Engine

By default, the `system` and `juniper_junos_system` modules perform the requested operation on all Routing Engines in a dual Routing Engine or Virtual Chassis setup. You can also instruct the modules 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 15 on page 89](#) summarizes the `all_re` and `other_re` values that are required to execute the requested operation on specific Routing Engines.

Table 15: Parameters for Specifying Routing Engines

Affected Routing Engines	<code>all_re</code> Parameter	<code>other_re</code> 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 devices running Junos OS
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Reboot all Routing Engines
      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
    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
    system:
      action: "shutdown"
      other_re: True
```

How to Reboot 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** and **juniper_junos_system** modules support the **vmhost** argument, which enables you to reboot 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.

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
  collections:
    - juniper.device

  tasks:
    - name: Reboot VM host
```

```
system:
  action: "reboot"
  vmhost: True
  all_re: False
```

Example: Using Ansible to Reboot Devices Running Junos OS

IN THIS SECTION

- [Requirements | 91](#)
- [Overview | 91](#)
- [Configuration | 92](#)
- [Executing the Playbook | 95](#)
- [Verification | 97](#)

The **system** module in the **juniper.device** collection enables you to halt, reboot, or shut down a device running Junos OS. This example uses the **system** module to reboot a device running Junos OS.

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
- Device running Junos OS 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 device running Junos OS
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the **system** module to reboot a device running Junos OS. 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 **Checking NETCONF connectivity** task, which utilizes the **wait_for** module to try to establish a NETCONF session with the device running Junos OS 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

Creating and Executing the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the **system** module to reboot a device running Junos OS:

1. Include the boilerplate for the playbook and this play, which executes the modules locally and includes the **juniper.device** collection.

```
---
- name: Reboot devices running Junos OS
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device
```

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: Checking NETCONF connectivity
  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 device running Junos OS
  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
  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
    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 devices running Junos OS
  hosts: dc1

  connection: local
  gather_facts: no
  collections:
    - juniper.device

  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: Checking NETCONF connectivity
    wait_for:
      host: "{{ inventory_hostname }}"
      port: "{{ netconf_port }}"
      timeout: 5

  - name: Reboot all Routing Engines on the device running Junos OS
    system:
      action: "reboot"
      in_min: 2
      all_re: True
    register: result
    notify:
      - pause_for_reboot
      - wait_reboot

  - name: Print response
    debug:
      var: result

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

```

Executing the Playbook

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 devices running Junos OS] *****

TASK [Confirmation check] *****
skipping: [dcl1a.example.net]

TASK [Checking NETCONF connectivity] *****
ok: [dcl1a.example.net]

TASK [Reboot all Routing Engines on the device running Junos OS] *****
changed: [dcl1a.example.net]

TASK [Print response] *****
ok: [dcl1a.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: [dcl1a.example.net]

RUNNING HANDLER [wait_reboot] *****
ok: [dcl1a.example.net]

PLAY RECAP *****
```



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

Verification

Verifying the Reboot

Purpose

Verify that the device running Junos OS 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.

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_system module replaces the functionality of the junos_shutdown and junos_zeroize modules.

RELATED DOCUMENTATION

[Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings | 116](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Using Ansible to Install Software on Devices Running Junos OS

SUMMARY

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

IN THIS SECTION

- [Using Ansible to Install Software | 98](#)
- [How to Specify the Software Image Location | 99](#)
- [Installation Process Overview | 101](#)
- [How to Specify Timeout Values | 103](#)
- [How to Specify Installation Options That Do Not Have an Equivalent Module Argument | 104](#)
- [How to Perform a VM Host Upgrade | 105](#)
- [How to Perform a Unified ISSU or NSSU | 105](#)
- [Example: Using Ansible to Install Software | 107](#)

Using Ansible to Install Software

Juniper Networks supports using Ansible to manage devices running Junos OS and provides modules that enable you to install or upgrade the software image on a device. [Table 16 on page 98](#) outlines the modules.

Table 16: Software Modules

Content Set	Module Name
juniper.device collection	software
Juniper.junos role	juniper_junos_software

NOTE: Starting in [Juniper.junos](#) Release 2.0.0, the [juniper_junos_software](#) module replaces the functionality of the [junos_install_os](#) module.

The following sections discuss how to specify the software image location and the general software installation process and options when using the modules to install software packages on devices running Junos OS. 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 on page 99](#)
- [Installation Process Overview on page 101](#)
- [How to Specify Timeout Values on page 103](#)
- [How to Specify Installation Options That Do Not Have an Equivalent Module Argument on page 104](#)
- [How to Perform a VM Host Upgrade on page 105](#)
- [How to Perform a Unified ISSU or NSSU on page 105](#)

How to Specify the Software Image Location

When you use the **software** or **juniper_junos_software** module to install software on devices running Junos OS, you can download the software package to the Ansible control node, and the modules, by default, copy the package to the target device before performing the installation. For mixed Virtual Chassis environments, the 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 device running Junos OS or resides at a URL that is reachable from the target device.

[Table 17 on page 100](#) 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 control node to the target device.

Table 17: 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 device running Junos OS 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 **local_package** argument to install software on a standalone device running Junos OS or on members in a non-mixed Virtual Chassis, or include the **pkg_set** argument to install software on the members in a mixed Virtual Chassis. The module argument specifies the absolute or relative file path to the software package or packages on the local control node.

The **local_package** argument is a single string specifying the software image path. The **pkg_set** argument contains a list of strings that specify the necessary software image paths, 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 to the **/var/tmp** directory on the target device running Junos OS (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 device running Junos OS, 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 device running Junos OS. 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 device running Junos OS, execute the [software](#) or [juniper_junos_software](#) module, and provide any necessary arguments. For example:

```
---
- name: Perform a Junos OS software upgrade
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Upgrade Junos OS
      software:
        local_package: "software/jinstall-ppc-17.3R1.10-signed.tgz"
        no_copy: false
        validate: True
      register: response
    - name: Print the response
      debug:
        var: response
```

When you execute the **software** or **juniper_junos_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, if the checksum is not already provided in the **checksum** argument. Acceptable **checksum_algorithm** values are **md5**, **sha1**, and **sha256**. The default is **md5**.
 - 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.

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, if a file with the same name and checksum does not already reside in the target location on the device. 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 already exists 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 value 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:

3. Validates the configuration against the new package if the **validate** parameter is set to **true**.

NOTE: By default, the **software** and **juniper_junos_software** modules do 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**.

4. Installs the package on each individual Routing Engine, unless **all_re** is set to **false**.
5. Reboots each upgraded Routing Engine, unless the **reboot** argument is set to **false**.

The **software** and **juniper_junos_software** modules enable 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 Networks software modules perform 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
  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 **software** or **juniper_junos_software** module to install software on a device, the module invokes the appropriate RPC for the given installation arguments, for example, 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 modules support explicit arguments for many of the installation options, for example, the **validate** option. The modules also support 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 modules, see the API reference documentation for that 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: The modules should only include installation options that are supported on the target device running Junos OS.

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
  collections:
    - juniper.device

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



```

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, which upgrades the host OS and compatible Junos OS, requires a VM Host Installation Package (**junos-vmhost-install-x.tgz**) and is performed using the **request vmhost software add** operational mode command, which corresponds to the **<request-vmhost-package-add>** RPC.

The **software** and **juniper_junos_software** modules support 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 devices:

```

---
- name: Upgrade VM Hosts
  hosts: vm_hosts
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

```

How to Perform a Unified ISSU or NSSU

The **software** and **juniper_junos_software** modules support 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** or **juniper_junos_software** module must include the **issu: True** argument. For example:

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

  tasks:
    - name: Perform a unified ISSU
      software:
        local_package: "junos-install-mx-x86-64-17.2R1.13.tgz"
        issu: True
      register: response
    - name: Print the response
      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** or **juniper_junos_software** module must include the **nssu: True** argument. For example:

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

  tasks:
    - name: Perform an NSSU
      software:
        local_package: "jinstall-ex-4300-17.3R1.10-signed.tgz"
        nssu: True
      register: response
```

```
- name: Print the response
  debug:
    var: response
```

Example: Using Ansible to Install Software

IN THIS SECTION

- [Requirements | 107](#)
- [Overview | 107](#)
- [Configuration | 108](#)
- [Executing the Playbook | 111](#)
- [Verification | 113](#)

This example uses the **software** module in the **juniper.device** collection to install a software image on a device running Junos OS.

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
- Device running Junos OS 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 device running Junos OS
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the **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 **Checking NETCONF connectivity** task, which utilizes the **wait_for** module to try to establish a NETCONF session with the device running Junos OS 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 device running Junos OS. 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

Creating the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the **software** module to install a software image on a device running Junos OS:

1. Include the boilerplate for the playbook and this play, which executes the modules locally and includes the **juniper.device** collection.

```
---
- name: Install Junos OS
```

```

hosts: mx1
connection: local
gather_facts: no
collections:
  - juniper.device

```

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: "20.3R1.8"
  OS_package: "junos-install-mx-x86-64-20.3R1.8.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: Checking NETCONF connectivity
    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
  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
  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
  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
  collections:
    - juniper.device

  vars:
    OS_version: "20.3R1.8"
    OS_package: "junos-install-mx-x86-64-20.3R1.8.tgz"
    pkg_dir: "software"
    log_dir: "{{ playbook_dir }}"
    netconf_port: 830
    wait_time: 3600

  tasks:
    - name: Checking NETCONF connectivity
      wait_for:
        host: "{{ inventory_hostname }}"
        port: "{{ netconf_port }}"
        timeout: 5

```

```
- name: Install Junos OS package
  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
  debug:
    var: response

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

Executing the Playbook

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 -v ansible-pb-junos-install-os.yaml
```

```
Using /etc/ansible/ansible.cfg as config file

PLAY [Install Junos OS]
*****

TASK [Checking 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-x86-64-20.3R1.8.tgz successfully installed. Response from device is: \nVerified junos-install-mx-x86-64-20.3R1.8 signed by PackageProductionECP256_2020 method ECDSA256+SHA256\n [...output truncated...] NOTICE: 'pending' set will be activated at next reboot... Reboot successfully initiated. Reboot message: Shutdown NOW! [pid 83918]"}

TASK [Print response]
*****
ok: [mx1a.example.com] => {
  "response": {
    "changed": true,
    "check_mode": false,
    "failed": false,
    "msg": "Package /home/user/ansible/software/junos-install-mx-x86-64-20.3R1.8.tgz successfully installed. Response from device is: \nVerified junos-install-mx-x86-64-20.3R1.8 signed by PackageProductionECP256_2020 method ECDSA256+SHA256\nVerified manifest signed by PackageProductionECP256_2020 method ECDSA256+SHA256\n [...output truncated...] NOTICE: 'pending' set will be activated at next reboot... Reboot successfully initiated. Reboot message: Shutdown NOW! [pid 83918]"
  }
}
```



```

RUNNING HANDLER [wait_reboot]
*****
ok: [mx1a.example.com] => {"changed": false, "elapsed": 209, "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

Verifying 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.

```
user@ansible-cn:~/ansible$ cat software.log
```

```

2020-12-11 00:24:49,478 - paramiko.transport - INFO - Connected (version 2.0,
client OpenSSH_7.5)
2020-12-11 00:24:49,632 - paramiko.transport - INFO - Authentication (publickey)
successful!
2020-12-11 00:24:57,923 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 344145920 / 3441301038 (10%)
2020-12-11 00:25:05,976 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 688275456 / 3441301038 (20%)
2020-12-11 00:25:13,949 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 1032404992 / 3441301038 (30%)
2020-12-11 00:25:22,051 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 1376534528 / 3441301038 (40%)
2020-12-11 00:25:30,357 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 1720664064 / 3441301038 (50%)
2020-12-11 00:25:38,360 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
b'junos-install-mx-x86-64-20.3R1.8.tgz': 2064793600 / 3441301038 (60%)

```

```
2020-12-11 00:25:46,575 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  b'junos-install-mx-x86-64-20.3R1.8.tgz': 2408923136 / 3441301038 (70%)
2020-12-11 00:25:54,983 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  b'junos-install-mx-x86-64-20.3R1.8.tgz': 2753052672 / 3441301038 (80%)
2020-12-11 00:26:03,066 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  b'junos-install-mx-x86-64-20.3R1.8.tgz': 3097182208 / 3441301038 (90%)
2020-12-11 00:26:11,330 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  b'junos-install-mx-x86-64-20.3R1.8.tgz': 3441301038 / 3441301038 (100%)
2020-12-11 00:26:11,331 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  after copy, computing checksum on remote package:
/var/tmp/junos-install-mx-x86-64-20.3R1.8.tgz
...
2020-12-11 00:26:27,623 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  checksum check passed.
2020-12-11 00:26:27,623 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  validating software against current config, please be patient ...
...
2020-12-11 00:30:55,725 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  software validate package-result: 0
Output:
Removing /packages/sets/previous
Verified junos-install-mx-x86-64-20.3R1.8 signed by PackageProductionECP256_2020
method ECDSA256+SHA256
Verified manifest signed by PackageProductionECP256_2020 method ECDSA256+SHA256
Checking PIC combinations
Adding junos-mx-x86-64-20.3R1.8 ...
...
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded

2020-12-11 00:30:55,725 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  installing software on RE0 ... please be patient ...
...
2020-12-11 00:33:56,203 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  software pkgadd package-result: 0
Output:
Verified junos-install-mx-x86-64-20.3R1.8 signed by PackageProductionECP256_2020
method ECDSA256+SHA256
...
2020-12-11 00:33:56,250 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
  installing software on RE1 ... please be patient ...
...
2020-12-11 00:37:18,562 - jnpr.ansible_module.software - INFO - [mx1a.example.com]
```

```

software pkgadd package-result: 0
Output:
Pushing /var/tmp/junos-install-mx-x86-64-20.3R1.8.tgz to
rel:/var/tmp/junos-install-mx-x86-64-20.3R1.8.tgz
Verified junos-install-mx-x86-64-20.3R1.8 signed by PackageProductionECP256_2020
method ECDSA256+SHA256
...
<?xml version="1.0" encoding="UTF-8"?><nc:rpc
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
message-id="urn:uuid:f6f6a59c-fedd-49fc-9cb3-9848f419a5b7">
<request-reboot><both-routing-engines/><in>0</in></request-reboot></nc:rpc>]]>
2020-12-11 00:37:19,880 - ncclient.operations.rpc - INFO - [host mx1a.example.com
session-id 46151]
Requesting 'CloseSession'

```

Meaning

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

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_software module replaces the functionality of the junos_install_os module.

RELATED DOCUMENTATION

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

[Using Ansible to Halt, Reboot, or Shut Down Devices Running Junos OS | 86](#)

[Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings | 116](#)

Using Ansible to Restore a Device Running Junos OS to the Factory-Default Configuration Settings

SUMMARY

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

IN THIS SECTION

- [How to Use Ansible to Restore the Factory-Default Configuration Settings | 116](#)
- [Example: Using Ansible to Restore the Factory-Default Configuration Settings | 118](#)

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

Juniper Networks supports using Ansible to manage devices running Junos OS and provides modules that enable you to restore a device to its factory-default configuration settings. [Table 18 on page 116](#) outlines the modules.

Table 18: Modules to Zeroize Devices

Content Set	Module Name
juniper.device collection	system
Juniper.junos role	juniper_junos_system

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

NOTE: Starting in `Juniper.junos` Release 2.0.0, the `juniper_junos_system` module replaces the functionality of the `junos_shutdown` and `junos_zeroize` modules.

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, and 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 about the **request system zeroize** command, see [request system zeroize](#).

The following Ansible playbook uses the **juniper.device** collection's **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 devices running Junos OS to factory-default configuration
  hosts: dc1
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Restore all Routing Engines to factory-default configuration
      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
    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
    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 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
    system:
      action: "zeroize"
      media: True

```

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

IN THIS SECTION

- [Requirements | 119](#)
- [Overview | 119](#)
- [Configuration | 119](#)
- [Executing the Playbook | 122](#)
- [Verification | 123](#)

This example demonstrates how to use the **system** module in the **juniper.device** collection to restore a device running Junos OS 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
- Device running Junos OS 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 presents an Ansible playbook that uses the **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 task that restores the hosts to the factory-default configuration executes the **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

Creating and Executing the Ansible Playbook

Step-by-Step Procedure

To create a playbook that uses the **system** module to restore a device running Junos OS to its factory-default configuration settings:

1. Include the boilerplate for the playbook and this play, which executes the modules locally and includes the **juniper.device** collection.

```
---
- name: Restore devices running Junos OS to factory-default configuration settings

  hosts: dc1_console
  connection: local
  gather_facts: no
  collections:
    - juniper.device
```

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

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

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

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

4. Create the task that confirms the users intent.

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

5. 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
  system:
    password: "{{ device_password }}"
    mode: "telnet"
    port: 23
    action: "zeroize"
    timeout: 120
    register: result

```

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

```

- name: Print response
  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 devices running Junos OS to factory-default configuration settings
  hosts: dcl_console
  connection: local
  gather_facts: no
  collections:
    - juniper.device

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

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

  tasks:
    - name: Confirmation check

```

```
fail: msg="Playbook run confirmation failed"
when: reset_confirmation != "yes"

- name: Restore all Routing Engines to factory-default configuration
  system:
    password: "{{ device_password }}"
    mode: "telnet"
    port: 23
    action: "zeroize"
    timeout: 120
    register: result

- name: Print response
  debug:
    var: result
```

Executing the Playbook

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.

```
root@ansible-cn:~/ansible# ansible-playbook ansible-pb-junos-zeroize.yaml

Device password:
This playbook resets hosts to factory-default configurations! Enter 'yes' to
continue.
[no]: yes

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

TASK [Confirmation check] *****
skipping: [dcl1-console.example.net]

TASK [Restore all Routing Engines to factory-default configuration] ****
changed: [dcl1-console.example.net]

TASK [Print response] *****
ok: [dcl1-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 *****
dcl1-console.example.net : ok=2    changed=1    unreachable=0    failed=0
skipped=1    rescued=0    ignored=0
```

Verification

Verifying Playbook Execution

Purpose

Verify that the devices running Junos OS 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.

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_system module replaces the functionality of the junos_shutdown and junos_zeroize modules.

RELATED DOCUMENTATION

[Using Ansible to Halt, Reboot, or Shut Down Devices Running Junos OS | 86](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Using 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 devices running Junos OS.

IN THIS SECTION

- [Module Overview | 126](#)
- [Taking and Comparing Snapshots | 129](#)
- [Performing Snapcheck Operations | 132](#)
- [Understanding the jsnapy and juniper_junos_jsnapy Module Output | 133](#)
- [Enabling the jsnapy Callback Plugin | 135](#)
- [Example: Using Ansible to Perform a JSNAPy Snapcheck Operation | 136](#)

Junos[®] Snapshot Administrator in Python (JSNAPy) enables you to capture and audit runtime environment snapshots of your network devices running Junos OS. You can capture and verify the configuration and operational status of a device and verify changes to a device. Juniper Networks provides Ansible modules that enable you to execute JSNAPy tests against devices running Junos OS as part of an Ansible playbook. [Table 13 on page 86](#) outlines the available modules.

Table 19: JSNAPy Modules

Content Set	Module Name
juniper.device collection	jsnapy
Juniper.junos role	juniper_junos_jsnapy

You must install Junos Snapshot Administrator in Python on the Ansible control node in order to use the modules. For installation instructions and information about creating JSNAPy configuration and test files, see the [Junos Snapshot Administrator in Python Documentation](#).

NOTE: Starting in **Juniper.junos** Release 2.0.0, the **juniper_junos_jsnapy** module replaces the functionality of the **junos_jsnapy** module.

The following sections discuss how to use the modules in Ansible playbooks.

Module Overview

The **jsnapy** and **juniper_junos_jsnapy** modules enable you to execute many of the same JSNAPy functions from an Ansible playbook as you can execute using JSNAPy on the command line, including:

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

The modules require specifying the **action** argument and either the **config_file** or the **test_files** argument. The **action** argument specifies the JSNAPy action to perform. [Table 20 on page 126](#) outlines the valid **action** values and the equivalent JSNAPy commands.

Table 20: jsnapy and juniper_junos_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 **hosts** section of the configuration file. In contrast, the Ansible modules execute the requested action on the hosts in the Ansible inventory group defined in the 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 **jsnapy** and **juniper_junos_jsnapy** modules also require either the **config_file** or the **test_files** argument to specify the JSNAPy configuration file or the JSNAPy test files to use for the given action. [Table 21 on page 127](#) outlines the **config_file** and **test_files** arguments.

Table 21: jsnapy and juniper_junos_jsnapy File Arguments

Module argument	Value	Additional Information
config_file	Absolute or relative file path to a JSNAPy configuration file.	<p>If the path is relative, the module checks for the configuration file in the following locations and in the order indicated:</p> <ul style="list-style-type: none"> • Ansible playbook directory • dir argument directory, if provided • /etc/jsnapy/testfiles directory (only if the dir argument is omitted) <p>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.</p>
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.	<p>For each test file that specifies a relative path, the module checks for the file in the following locations and in the order indicated:</p> <ul style="list-style-type: none"> • 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 under the Ansible playbook directory, even if the **dir** argument is present. If the file does not exist under the playbook directory, the module checks under the **dir** argument directory, if it is specified, or under the **/etc/jsnapy/testfiles** directory, if the **dir** argument is omitted. The playbook generates an error message if the file is not found.

The following sample playbook performs the **snap_pre** action using the **configuration_interface_status.yaml** configuration file. If the configuration file does not exist in the playbook directory, the module checks for the file in the user's home directory under the **jsnapy/testfiles** subdirectory.

```
---
- name: Junos Snapshot Administrator tests
```

```

hosts: dcl1
connection: local
gather_facts: no
collections:
  - juniper.device

tasks:
  - name: Take a pre-maintenance snapshot of the interfaces
    jsnapy:
      action: "snap_pre"
      dir: "~/jsnapy/testfiles"
      config_file: "configuration_interface_status.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 **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 **jsnapy** and **juniper_junos_jsnapy** modules 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. The level of information logged in the file is determined by Ansible's verbosity level and debug options. 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 enable the **jsnapy** callback plugin to capture and summarize information for failed JSNAPy tests. To enable the callback plugin, add the **callback_whitelist = jsnapy** statement to the Ansible configuration file. For more information, see [“Enabling the jsnapy Callback Plugin” on page 135](#).

Taking and Comparing Snapshots

JSNAPy enables you to capture runtime environment snapshots of your network devices running Junos OS before and after a change and then compare the snapshots to verify the expected changes or identify unexpected issues. The `jsnapy` and `juniper_junos_jsnapy` Ansible modules enable you to take and compare JSNAPy snapshots as part of an Ansible playbook. The modules save 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` and `juniper_junos_jsnapy` Module Output” on page 133](#).

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 `configuration_interface_status.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
  collections:
    - juniper.device

  tasks:
    - name: Take a pre-maintenance snapshot of the interfaces
      jsnapy:
        action: "snap_pre"
        dir: "~/jsnapy/testfiles"
        config_file: "configuration_interface_status.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 `configuration_interface_status.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
collections:
  - juniper.device

tasks:
  - name: Take a post-maintenance snapshot of the interfaces
    jsnapy:
      action: "snap_post"
      dir: "~/jsnapy/testfiles"
      config_file: "configuration_interface_status.yaml"
      logfile: "jsnapy_tests.log"

```

When the **jsnapy** or **juniper_junos_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, the preexisting **PRE** and **POST** snapshots for each test on each device are compared and evaluated 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 **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 **assert** module to verify that all tests passed on the given device.

```

---
- name: Junos Snapshot Administrator tests
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Compare PRE and POST snapshots
      jsnapy:
        action: "check"
        dir: "~/jsnapy/testfiles"
        config_file: "configuration_interface_status.yaml"
        logfile: "jsnapy_tests.log"

```

```

    register: test_result

- name: Verify JSNAPy tests passed
  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-interface-check.yaml
```

```

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

TASK [Compare PRE and POST snapshots] *****
ok: [dcla.example.net]
ok: [dclb.example.net]

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

PLAY RECAP *****
dclb.example.net  : ok=2    changed=0    unreachable=0    failed=0    skipped=0
                  rescued=0    ignored=0
dcla.example.net : ok=1    changed=0    unreachable=0    failed=1    skipped=0
                  rescued=0    ignored=0

```

Performing 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 **jsnapy** and **juniper_junos_jsnapy** Ansible modules enable 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 **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 assert module to verify that all tests defined in the test files passed on that device.

```
---
- name: Junos Snapshot Administrator tests
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Take a snapshot and immediately evaluate it
      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
      assert:
        that:
          - "test_result.passPercentage == 100"
```

Understanding the jsnapy and juniper_junos_jsnapy Module Output

When the `jsnapy` or `juniper_junos_jsnapy` module performs a `snap_pre`, `snap_post`, or `snapcheck` action, it automatically saves the snapshots in the JSNAPy `snapshots` directory. The modules use the default JSNAPy directories unless you modify the JSNAPy configuration file 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 22 on page 133](#) 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 22: JSNAPy Output Filenames

action value	Output Files
<code>snap_pre</code>	<code>hostname_PRE_hash_command.format</code>
<code>snap_post</code>	<code>hostname_POST_hash_command.format</code>
<code>snapcheck</code>	<code>hostname_snap_temp_hash_command.format</code> or <code>hostname_PRE_hash_command.format</code>

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` and `juniper_junos_jsnapy` modules only differentiate the snapshot filenames for a given action based 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` and `juniper_junos_jsnapy` modules 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.

Enabling the jsnapy Callback Plugin

When you execute JSNAPy tests against devices running Junos OS and one or more tests fail, it can be difficult to identify and extract the failed tests if the output is extensive. 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 not enabled 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"}
```

Example: Using Ansible to Perform a JSNAPy Snapcheck Operation

IN THIS SECTION

- [Requirements | 136](#)
- [Overview | 137](#)
- [Configuration | 138](#)
- [Executing the Playbook | 148](#)
- [Verification | 151](#)
- [Troubleshooting Ansible Playbook Errors | 152](#)

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

Requirements

This example uses the following hardware and software components:

- Ansible control node running:
 - Python 3.7 or later
 - Ansible 2.10 or later with the **juniper.device** collection installed
 - Junos PyEZ Release 2.6.0 or later
 - Junos Snapshot Administrator in Python Release 1.3.6 or later

Before executing the Ansible playbook, be sure you have:

- Devices running Junos OS 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 device running Junos OS
- Existing Ansible inventory file with required hosts defined

Overview

In this example, the Ansible playbook configures BGP peering sessions on three devices running Junos OS 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 device running Junos OS 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 device running Junos OS 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 the handler that pauses playbook execution for a specified amount of time to allow 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 and confirms the commit, provided that all the tests pass. 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 device running Junos OS 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 | 138
- Define the Jinja2 Template and Host Variables | 139
- Create the JSNAPy Test Files | 142
- Create the Ansible Playbook | 143
- Results | 146

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

Step-by-Step Procedure

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;
        }
    }
}
}

```

Step-by-Step Procedure

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

Step-by-Step Procedure

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 and includes the **juniper.device** collection.

```

---
- name: Load and commit BGP configuration
  hosts: bgp_routers
  connection: local
  gather_facts: no
  collections:
    - juniper.device

```

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
  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, and set the pause time to an appropriate value for your environment.

```

handlers:
  - name: Waiting for BGP peers to establish connections
    pause: seconds=60

```

Step-by-Step Procedure

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 and includes the **juniper.device** collection.

```
- name: Verify BGP
  hosts: bgp_routers
  connection: local
  gather_facts: no
  collections:
    - juniper.device
```

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
    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
  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 **assert** module to assert that the JSNAPy tests passed.

```

- name: Verify BGP configuration
  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
  collections:
    - juniper.device

  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
      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
  collections:
    - juniper.device

tasks:
  - name: Execute snapcheck
    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
  config:
    check: true
    commit: false
    diff: false
    logfile: "{{ logfile }}"
  when:
    - config_result.changed
```

```
    - "snapcheck_result.passPercentage == 100"

- name: Verify BGP configuration
  assert:
    that:
      - "snapcheck_result.passPercentage == 100"
      msg: "JSNAPy test on {{ inventory_hostname }} failed"
```

Executing the Playbook

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-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

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 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.

Troubleshooting Ansible Playbook Errors

IN THIS SECTION

- [Troubleshooting Configuration Load Errors | 152](#)
- [Troubleshooting Failed JSNAPy Tests | 153](#)
- [Troubleshooting Failed Commit Confirmations | 154](#)

Troubleshooting 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.

Troubleshooting 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.

Troubleshooting 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 [“Troubleshooting Failed JSNAPy Tests” on page 153](#).

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_jsnapy module replaces the functionality of the junos_jsnapy module.



CHAPTER

Using Ansible to Manage the Configuration

[Using Ansible to Retrieve or Compare Junos OS Configurations](#) | **156**

[Using Ansible to Configure Devices Running Junos OS](#) | **169**

Using Ansible to Retrieve or Compare Junos OS Configurations

SUMMARY

Use the Juniper Networks Ansible modules to retrieve or compare configurations on devices running Junos OS.

IN THIS SECTION

- [How to Specify the Source Database for the Configuration Data | 157](#)
- [How to Specify the Scope of the Configuration Data to Return | 158](#)
- [How to Specify the Format of the Configuration Data to Return | 160](#)
- [How to Retrieve Configuration Data for Third-Party YANG Data Models | 161](#)
- [How to Specify Options That Do Not Have an Equivalent Module Argument | 163](#)
- [How to Save Configuration Data To a File | 164](#)
- [How to Compare the Active Configuration to a Previous Configuration | 166](#)

Juniper Networks provides Ansible modules that enable you to manage the configuration on devices running Junos OS. [Table 23 on page 156](#) outlines the available modules, which enable you to retrieve or compare Junos OS configurations.

Table 23: Modules to Retrieve or Compare Configurations

Content Set	Module Name
juniper.device collection	config
Juniper.junos role	juniper_junos_config

NOTE: Starting in **Juniper.junos** Release 2.0.0, the **juniper_junos_config** module combines and replaces the functionality of the **junos_commit**, **junos_get_config**, **junos_install_config**, and **junos_rollback** modules.

You can use the modules 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 device running Junos OS, execute the **config** or **juniper_junos_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 modules to retrieve or compare Junos OS configurations.

How to Specify the Source Database for the Configuration Data

When you use the **config** or **juniper_junos_config** module to retrieve the configuration, you must include the **retrieve** parameter in the module's argument list and specify the configuration database from which to retrieve the data. You can retrieve data from the committed configuration database or from the candidate configuration database by setting **retrieve** to **'committed'** or **'candidate'**, respectively.

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
  collections:
    - juniper.device

  tasks:
    - name: "Get committed configuration"
      config:
        retrieve: "committed"
```

```

    register: response
- name: "Print result"
  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
  collections:
    - juniper.device

  tasks:
    - name: "Get candidate configuration"
      config:
        retrieve: "candidate"
      register: response
    - name: "Print result"
      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 use the **config** and **juniper_junos_config** modules to retrieve specific portions of the configuration by including the 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. When multiple hierarchies are requested, 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Get selected configuration hierarchies"
      config:
        retrieve: "committed"
        filter: "<configuration><interfaces/><protocols/></configuration>"
        register: response
    - name: "Print result"
      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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Get selected configuration hierarchies"
      config:
        retrieve: "committed"
        filter: "<interfaces><interface>
                <name>ge-1/0/1</name></interface></interfaces>"
        register: response
    - name: "Print result"
      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
  collections:
    - juniper.device

  tasks:
    - name: "Get selected configuration hierarchies"
      config:
        retrieve: "committed"
        filter: "system/services"
        register: response
    - name: "Print result"
      debug:
        var: response

```

How to Specify the Format of the Configuration Data to Return

When you use the `config` or `juniper_junos_config` module to retrieve the configuration, the module invokes the Junos XML protocol `<get-configuration>` operation, which can return Junos OS configuration data as formatted text, Junos XML elements, Junos OS `set` commands, or JavaScript Object Notation (JSON). By default, the module returns configuration data as formatted text, which 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 `format` parameter equal to the desired format. To explicitly request text format, or to request Junos XML elements, Junos OS `set` commands, or JSON format, set the `format` value to `'text'`, `'xml'`, `'set'`, or `'json'`, respectively. 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
  collections:

```



```

- juniper.device

tasks:
- name: "Get configuration in XML format"
  config:
    retrieve: "committed"
    format: "xml"
    register: response
- name: "Print result"
  debug:
    var: response

```

How to Retrieve Configuration Data for Third-Party YANG Data Models

You can load standardized or custom YANG modules on devices running Junos OS 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 **config** or **juniper_junos_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 including the appropriate module arguments. 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 you specify the **ietf** or **openconfig** value for the **model** argument, the module automatically uses the appropriate namespace. If you retrieve data for a custom YANG data model by using **model: "custom"**, 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** or **juniper_junos_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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Retrieve the OpenConfig interfaces configuration"
      config:
        retrieve: "committed"
        model: "openconfig"
        filter: "interfaces"
        format: "xml"
        register: response

    - name: "Print result"
      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"
    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"
    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 `config` or `juniper_junos_config` module to retrieve the configuration, the module invokes the Junos XML protocol `<get-configuration>` operation. The modules support explicit arguments for many of the `<get-configuration>` attributes, for example, the `format` attribute. The modules also support 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 https://www.juniper.net/documentation/en_US/junos/topics/reference/tag-summary/junos-xml-protocol-get-configuration.html.

For example, the modules retrieve 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
  collections:
    - juniper.device

  tasks:
    - name: "Get selected hierarchy from the post-inheritance configuration"
      config:
        retrieve: "committed"
        filter: "system/services"
        options:
          inherit: "inherit"
      register: response
    - name: "Print result"
      debug:
        var: response

```

How to Save Configuration Data To a File

When you use the `config` or `juniper_junos_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 and 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
  collections:

```

```

- juniper.device

tasks:
- name: "Save configuration to a file"
  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 and 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
  collections:
    - juniper.device

  tasks:
    - name: "Get selected configuration hierarchies and save to file"
      config:
        retrieve: "committed"
        filter: "system/services"
        dest: "{{ inventory_hostname }}-system-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 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 devices return a significant amount of configuration data.

How to Compare the Active Configuration to a Previous Configuration

The `config` and `juniper_junos_config` modules enable 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 modules return the `diff` and `diff_lines` keys, which 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.

NOTE: Starting in `Juniper.junos` Release 2.2.0, the `diff` key returns a dictionary instead of a string.

- **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, 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: dcl
connection: local
gather_facts: no
collections:
  - juniper.device

vars_prompt:
  - name: "ROLLBACK"
    prompt: "Rollback ID to compare with active configuration"
    private: no

tasks:
  - name: "Compare active to previous configuration"
    config:
      diff: true
      rollback: "{{ ROLLBACK }}"
      check: false
      commit: false
      diffs_file: "{{ inventory_hostname }}-diff-rollback-{{ ROLLBACK }}"
      register: response
  - name: "Print diff"
    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: [dcl1.example.net]

TASK [Print diff] *****
ok: [dcl1.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, checked, diffed,
closed."
    }
}

PLAY RECAP *****
dcl1a.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;
-   }

```

Release History Table

Release	Description
2.2.0	Starting in Juniper.junos Release 2.2.0, the diff key returns a dictionary instead of a string.
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_config module combines and replaces the functionality of the junos_commit , junos_get_config , junos_install_config , and junos_rollback modules.

RELATED DOCUMENTATION

[Using Ansible to Configure Devices Running Junos OS | 169](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Using Ansible to Configure Devices Running Junos OS

SUMMARY

Use the Juniper Networks Ansible modules to manage the configuration on devices running Junos OS.

IN THIS SECTION

- [Module Overview | 170](#)
- [How to Specify the Configuration Mode | 171](#)
- [How to Specify the Load Action | 173](#)
- [How to Specify the Format of the Configuration Data to Load | 173](#)
- [How to Load Configuration Data as Strings | 174](#)
- [How to Load Configuration Data from a Local or Remote File | 176](#)
- [How to Load Configuration Data Using a Jinja2 Template | 178](#)
- [How to Load the Rescue Configuration | 181](#)
- [How to Roll Back the Configuration | 182](#)
- [How to Commit the Configuration | 183](#)
- [How to Ignore Warnings When Configuring Devices | 186](#)
- [Example: Using Ansible to Configure Devices Running Junos OS | 187](#)

Juniper Networks provides Ansible modules that enable you to configure devices running Junos OS. [Table 24 on page 169](#) outlines the available modules. 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 24: Modules to Manage the Configuration

Content Set	Module Name
juniper.device collection	config
Juniper.junos role	juniper_junos_config

NOTE: Starting in **Juniper.junos** Release 2.0.0, the **juniper_junos_config** module combines and replaces the functionality of the **junos_commit**, **junos_get_config**, **junos_install_config**, and **junos_rollback** modules.

The following sections discuss how to use the modules to modify and commit the configuration on devices running Junos OS.

Module Overview

The **config** and **juniper_junos_config** modules enable you to perform the following operations on devices running Junos OS:

- 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** and **juniper_junos_config** modules make changes to the candidate configuration database using **configure exclusive** mode, which automatically locks and unlocks the candidate global configuration. You can also make changes to a private copy of the candidate configuration. For more information about specifying the configuration mode, see [“How to Specify the Configuration Mode” on page 171](#).

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 candidate configuration. The functions support 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 173](#).
- **Format**—You can configure devices running Junos OS 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 173](#).

- 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 174](#)
 - [How to Load Configuration Data from a Local or Remote File on page 176](#)
 - [How to Load Configuration Data Using a Jinja2 Template on page 178](#)

The **config** and **juniper_junos_config** modules also enable 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 181](#)
- [How to Roll Back the Configuration on page 182](#)

After modifying the configuration, you must commit the configuration to make it the active configuration on the device. By default, the **config** and **juniper_junos_config** modules commit the changes to the configuration. To alter this behavior or supply additional commit options, see [“How to Commit the Configuration” on page 183](#).

By default, when the **config** or **juniper_junos_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 candidate configuration database. By default, the **config** and **juniper_junos_config** modules make 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.

To specify the mode, include the **config_mode** parameter in the module’s argument list. Supported modes include **exclusive** and **private**. Both modes discard any uncommitted changes upon exiting.

The following playbook uses **configure private** mode to modify the configuration:

```

---
- name: "Configure Device"
  hosts: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Configure op script"
      config:
        config_mode: "private"
        load: "set"
        lines:
          - "set system scripts op file bgp.slax"
      register: response
    - name: "Print the config changes"
      debug:
        var: response.diff_lines

```

user@ansible-cn:~/ansible\$ **ansible-playbook configure-script.yaml**

```

PLAY [Configure Device] *****

TASK [Configure op script] *****
changed: [dcl1.example.net]

TASK [Print the config changes] *****

ok: [dcl1.example.net] => {
  "response.diff_lines": [
    "",
    "[edit system scripts op]",
    "+   file bgp.slax;"
  ]
}

PLAY RECAP *****
dcl1.example.net  : ok=2    changed=1    unreachable=0    failed=0    skipped=0
                  rescued=0    ignored=0

```

How to Specify the Load Action

The `config` and `juniper_junos_config` modules support 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 25 on page 173](#) summarizes the parameter settings required for each type of load operation.

Table 25: Parameters for Specifying the Load Operation

Load Operation	load Argument	Description
<code>load merge</code>	<code>load: "merge"</code>	Merge the loaded configuration with the existing configuration.
<code>load override</code>	<code>load: "override"</code>	Replace the entire configuration with the loaded configuration.
<code>load replace</code>	<code>load "replace"</code>	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the <code>replace</code> tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
<code>load patch</code>	<code>load: "patch"</code>	Load configuration data from a patch file.
<code>load set</code>	<code>load: "set"</code>	Load configuration data that is in <code>set</code> format. The configuration data is loaded line by line and can contain configuration mode commands such as <code>set</code> , <code>delete</code> , and <code>deactivate</code> .
<code>load update</code>	<code>load: "update"</code>	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 `config` and `juniper_junos_config` modules enable you to configure devices running Junos OS 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.

NOTE: Starting in Junos OS Release 16.1R1, devices running Junos OS support loading configuration data in JSON format.

The **config** and **juniper_junos_config** modules attempt to auto-detect the format of configuration data supplied as strings using 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 26 on page 174](#) 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 26: 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 **config** and **juniper_junos_config** modules enable 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 modules attempt 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 173](#). If you include the **format** parameter in the module's argument list, it overrides the auto-detect 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Load configuration data using strings and commit"
      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"
      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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Load configuration data using strings and commit"
      config:
        load: "merge"
        lines:
          - |
            system {
              scripts {
                op {
                  file bgp.slax;
```

```

        file bgp-neighbor.slax;
    }
}
register: response
- name: "Print the response"
  debug:
    var: response

```

How to Load Configuration Data from a Local or Remote File

The `config` and `juniper_junos_config` modules enable 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 format of the configuration data in the file and the location of 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 178](#).

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 173](#). 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 27 on page 177](#) outlines the module parameters that you can include to specify the location of the file.

Table 27: 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 a Hypertext Transfer Protocol (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
  collections:
    - juniper.device

  tasks:
    - name: "Load configuration from a local file and commit"
      config:
        load: "merge"
        src: "build_conf/{{ inventory_hostname }}/junos.conf"
      register: response
    - name: "Print the response"
      debug:
        var: response

```

To load configuration data from a file on the managed device running Junos OS, 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
  collections:

```

```

- juniper.device

tasks:
- name: "Load configuration from a remote file and commit"
  config:
    load: "merge"
    url: "/var/tmp/junos.conf"
    register: response
- name: "Print the response"
  debug:
    var: response

```

The value for **url** can be an absolute or relative local file path, an FTP location, or a Hypertext Transfer Protocol (HTTP) URL.

- A local filename can have 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.
- A filename for a file on an FTP server has the following form:

```
ftp://username:password@hostname/path/filename
```

- A filename 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/%2Fpath/filename**.

How to Load Configuration Data Using a Jinja2 Template

The **config** and **juniper_junos_config** modules enable you to render configuration data from a Jinja2 template file on the Ansible control node and load and commit the configuration on a device running Junos OS. 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 modules use 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 templates'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 173](#).

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 **config** or **juniper_junos_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
  collections:
    - juniper.device

  tasks:
    - name: "Load a configuration from a Jinja2 template and commit"
      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"
      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 allows 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 `config` and `juniper_junos_config` modules enable you to revert to an existing rescue configuration on devices running Junos OS. 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: dcl1a
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Load and commit rescue configuration"
      config:
        rollback: "rescue"
      register: response
    - name: "Print response"

```

```

debug:
  var: response

```

How to Roll Back the Configuration

Devices running Junos OS 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 `config` and `juniper_junos_config` modules enable you to roll back to a previously committed configuration on devices running Junos OS. 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
  collections:
    - juniper.device

  vars_prompt:
    - name: "ROLLBACK"
      prompt: "Rollback ID of the configuration to restore"
      private: no

  tasks:
    - name: "Roll back the configuration and commit"
      config:
        rollback: "{{ ROLLBACK }}"
      register: response
    - name: "Print the configuration changes"

```

```

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: [dcl1.example.net]

TASK [Print the configuration changes] *****
ok: [dcl1.example.net] => {
  "response.diff_lines": [
    "",
    "[edit interfaces]",
    "- ge-0/0/0 {",
    "  unit 0 {",
    "    family mpls;",
    "  }",
    "- }"
  ]
}

PLAY RECAP *****
dcl1.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 `config` or `juniper_junos_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 28 on page 184](#) outlines the module arguments that you can use to specify different commit options.

Table 28: Commit Options

Module Argument	Description	Default value for load and rollback operations
check: <i>boolean</i>	Perform a commit check or confirm a previous confirmed commit operation.	True
check_commit_wait: <i>seconds</i>	Wait the specified number of seconds between the commit check and the commit operation.	-
comment: <i>"string"</i>	Log a comment for that commit operation in the system log file and in the device's commit history.	-
commit: <i>boolean</i>	Commit the configuration changes or confirm a previous confirmed commit operation.	True
commit_empty_changes: <i>boolean</i>	Commit the configuration changes even if there are no differences between the candidate configuration and the committed configuration.	False
confirmed: <i>minutes</i>	Require that a commit operation be confirmed within a specified amount of time after the initial commit. Otherwise, roll back to the previously committed configuration. Either the commit: true option or the check: true option must be used to confirm the commit.	-

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.

By default, the **config** and **juniper_junos_config** modules execute 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 this 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.

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.

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** or **juniper_junos_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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: "Load configuration. Wait 10 seconds between check and commit. Confirm
      within 5 min."
      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"
      debug:
        var: response

    - name: "Confirm the commit with a commit check"
      config:
        check: true
        diff: false
        commit: false
      register: response
```

```

- name: "Print the response"
  debug:
    var: response

```

How to Ignore Warnings When Configuring Devices

The `config` and `juniper_junos_config` modules enable you to modify and commit the configuration on devices running Junos OS. 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, thus causing 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` and `juniper_junos_config` modules 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: dcl
  connection: local
  gather_facts: no
  collections:
    - juniper.device

  tasks:
    - name: Configure op script
      config:
        config_mode: "private"
        load: "set"
        lines:
          - "set system scripts op file bgp.slax"
          ignore_warning: true
      register: response
    - name: Print the response
      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
  collections:
    - juniper.device

  tasks:
    - name: Configure device running Junos OS and ignore warnings
      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
      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: Using Ansible to Configure Devices Running Junos OS

IN THIS SECTION

- [Requirements | 188](#)
- [Overview | 188](#)
- [Configuration | 189](#)
- [Executing the Playbook | 191](#)

- Verification | 192
- Troubleshooting Playbook Errors | 193

The **config** module enables you to manage the configuration on devices running Junos OS. This example uses the **config** module to make configuration changes on a device running Junos OS 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
- Device running Junos OS 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 device running Junos OS
- Existing Ansible inventory file with required hosts defined

Overview

This example presents an Ansible playbook that uses the **config** module to enable a new op script in the configuration of the target devices running Junos OS. The configuration data file, **junos-config.conf**, contains the relevant configuration data formatted as text.

The playbook includes the **Checking NETCONF connectivity** task, which utilizes the **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 task to configure the device executes the **config** module provided that the NETCONF check was successful. The **load: "merge"** module 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 | 189](#)
- [Create the Ansible Playbook | 189](#)
- [Results | 190](#)

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 device running Junos OS:

1. Include the playbook boilerplate, which executes the modules locally and includes the **juniper.device** collection.

```
---
- name: Load and commit configuration data on a device running Junos OS
  hosts: dc1
```

```

connection: local
gather_facts: no
collections:
  - juniper.device

```

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

```

tasks:
  - name: Checking NETCONF connectivity
    wait_for:
      host: "{{ inventory_hostname }}"
      port: 830
      timeout: 5

```

3. Create the task to load the configuration onto the device and commit it.

```

  - name: Merge configuration data from a file and commit
    config:
      load: "merge"
      src: "build_conf/{{ inventory_hostname }}/junos-config.conf"
      comment: "Configuring op script with Ansible"
      register: response

```

4. (Optional) Create a task to print the response, which includes the configuration changes in *diff* format.

```

  - name: Print the response
    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 device running Junos OS
  hosts: dc1
  connection: local
  gather_facts: no
  collections:

```

```
- juniper.device

tasks:
- name: Checking NETCONF connectivity
  wait_for:
    host: "{{ inventory_hostname }}"
    port: 830
    timeout: 5

- name: Merge configuration data from a file and commit
  config:
    load: "merge"
    src: "build_conf/{{ inventory_hostname }}/junos-config.conf"
    comment: "Configuring op script with Ansible"
  register: response

- name: Print the response
  debug:
    var: response
```

Executing the Playbook

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-config.yaml
```

```
PLAY [Load and commit configuration data on a device running Junos OS] ****

TASK [Checking NETCONF connectivity] *****
ok: [dcl1a.example.net]

TASK [Merge configuration data from a file and commit] *****
changed: [dcl1a.example.net]

TASK [Print the response] *****
ok: [dcl1a.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/dcl1a.example.net/junos-config.conf",
    "msg": "Configuration has been: opened, loaded, checked, diffed,
committed, closed."
  }
}

PLAY RECAP *****
dcl1a.example.net : ok=3    changed=1    unreachable=0    failed=0    skipped=0
                   rescued=0    ignored=0
```

Verification

Verifying the Configuration

Purpose

Verify that the configuration was correctly updated on the device running Junos OS.

Action

Review the Ansible playbook output to see whether the configuration task succeeded or failed. You can also log in to the device running Junos OS 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
```

Troubleshooting Playbook Errors

IN THIS SECTION

- [Troubleshooting Timeout Errors | 193](#)
- [Troubleshooting Configuration Lock Errors | 194](#)
- [Troubleshooting Configuration Change Errors | 195](#)

Troubleshooting 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.

Troubleshooting 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 device running Junos OS to give the current user the necessary permissions to make the changes.

Troubleshooting 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 device running Junos OS to give the current user the necessary permissions to make the changes.

Release History Table

Release	Description
2.0.0	Starting in Juniper.junos Release 2.0.0, the juniper_junos_config module combines and replaces the functionality of the junos_commit , junos_get_config , junos_install_config , and junos_rollback modules.

RELATED DOCUMENTATION

[Using Ansible to Retrieve or Compare Junos OS Configurations | 156](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

7

CHAPTER

Troubleshooting Ansible for Junos OS

Troubleshooting Junos PyEZ (junos-eznc) Install Errors for Ansible Modules | **197**

Troubleshooting Ansible Collection, Role, and Module Errors When Managing Devices Running Junos OS | **198**

Troubleshooting Ansible Connection Errors When Managing Devices Running Junos OS | **200**

Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS | **203**

Troubleshooting Ansible Errors When Configuring Devices Running Junos OS | **206**

Troubleshooting Junos PyEZ (junos-eznc) Install Errors for Ansible Modules

Problem

Description:

During execution of a **juniper.device** module or a **Juniper.junos** 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 and **Juniper.junos** role use the Junos PyEZ Python library to perform operations on devices running Junos OS. Ansible generates this error if the library is not installed or if Ansible can't locate the library, for example, if you run Ansible in a virtual environment or in a non-standard system location and Ansible is searching in a different 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 inside a virtual environment, you must:

- Install Junos PyEZ inside the virtual environment
- Specify the path to the Python interpreter in the virtual environment—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.

To verify that Junos PyEZ is successfully installed on the control node, launch the Python interactive shell and import the **jnpr.junos** package.

```
user@ansible-cn:~$ 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 the `jnpr.junos` package is successfully imported and there is no error message, then Junos PyEZ is installed on the Ansible control node.

RELATED DOCUMENTATION

[Ansible for Junos OS Server Requirements | 35](#)

[Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS | 203](#)

[Troubleshooting Ansible Collection, Role, and Module Errors When Managing Devices Running Junos OS | 198](#)

Troubleshooting Ansible Collection, Role, and Module Errors When Managing Devices Running Junos OS

Problem

Description:

During execution of an Ansible playbook, the control node generates an error that the `juniper.device` collection or `Juniper.junos` role was not found, that no action was detected in the task, or that the module could not be resolved. For example:

```
ERROR! the role 'Juniper.junos' was not found in
/usr/share/ansible/roles:/etc/ansible/roles ...
```

or

```
ERROR! no action detected in task. This often indicates a misspelled module name,
or incorrect module path.
```

or

```
ERROR! couldn't resolve module/action 'facts'. This often indicates a misspelling,
missing collection, or incorrect module path.
```

Cause

The Ansible control node cannot locate the **juniper.device** collection or the **Juniper.junos** role and associated modules.

Solution

The **juniper.device** collection and **Juniper.junos** role are hosted on the Ansible Galaxy website. In order to use the modules in the collection or role, you must install the collection or role 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:~$ sudo ansible-galaxy collection install juniper.device
```

To install the **Juniper.junos** role on the Ansible control node, execute the **ansible-galaxy install** command, and specify **Juniper.junos**.

```
user@ansible-cn:~$ sudo ansible-galaxy install Juniper.junos
```

NOTE: If you do not install the collection or role in the default location, you might need to define the path to it in your Ansible setup. For more information about installing collections and roles and specifying the path, see the official [Ansible documentation](#).

To enable the playbook to use the **juniper.device** collection modules, either include the collection in the playbook play or use the fully qualified collection name when executing a module. For example:

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

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

To enable the playbook to use the **Juniper.junos** role modules, include the role in the playbook play. For example:

```
---  
- name: Get Device Facts  
  hosts: junos-all  
  connection: local  
  gather_facts: no  
  roles:  
    - Juniper.junos
```

RELATED DOCUMENTATION

[Ansible for Junos OS Server Requirements | 35](#)

[Understanding the Ansible for Junos OS Collections, Roles, and Modules | 19](#)

Troubleshooting Ansible Connection Errors When Managing Devices Running Junos OS

IN THIS SECTION

- [Troubleshooting Failed or Invalid Connection Errors | 201](#)
- [Troubleshooting Unknown Host Errors | 202](#)
- [Troubleshooting Refused Connection Errors | 202](#)

The following sections outline connection errors that you might encounter when using Ansible to manage devices running Junos OS. These sections also present potential causes and solutions for each error.

Troubleshooting Failed or Invalid Connection Errors

Problem

Description:

During execution of a `juniper.device` or `Juniper.junos` module on a device running Junos OS, 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 devices running Junos OS, because they use Junos PyEZ and the Junos XML API over NETCONF to interface with the device. Therefore, to perform operations on devices running Junos OS, 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 device running Junos OS, it generates an error.

Solution

To direct the Ansible control node to run the `juniper.device` or `Juniper.junos` 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
  collections:
    - juniper.device
```

Troubleshooting Unknown Host Errors

Problem

Description:

During execution of a **juniper.device** or **Juniper.junos** module, the Ansible control node generates a **ConnectUnknownHostError** error.

```
"msg": "Unable to make a PyEZ connection: ConnectUnknownHostError(dcl1a.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 Devices Running Junos OS” on page 26](#) as well as the official Ansible documentation at <https://www.ansible.com/>.

Troubleshooting Refused Connection Errors

Problem

Description:

During execution of a **juniper.device** or **Juniper.junos** module, the Ansible control node generates a **ConnectRefusedError** error. For example:

```
"msg": "Unable to make a PyEZ connection: ConnectRefusedError(dcl1a.example.net) "
```

Cause

The most likely cause for a refused connection error is that NETCONF over SSH is not enabled on the device running Junos OS.

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 device running Junos OS.

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

RELATED DOCUMENTATION

[Setting up Ansible for Junos OS Managed Nodes | 38](#)

[Understanding the Ansible Inventory File When Managing Devices Running Junos OS | 26](#)

[Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS | 203](#)

Troubleshooting Ansible Authentication Errors When Managing Devices Running Junos OS

IN THIS SECTION

- [Troubleshooting ConnectAuthError Issues | 204](#)
- [Troubleshooting Attribute conn_type Errors | 204](#)

The following sections outline authentication errors that you might encounter when using Ansible to manage devices running Junos OS. These sections also present potential causes and solutions for each error.

Troubleshooting ConnectAuthError Issues

Problem

Description:

During execution of a **juniper.device** or **Juniper.junos** module, the Ansible control node generates a **ConnectAuthError** error for failed authentication. For example:

```
"msg": "Unable to make a PyEZ connection: ConnectAuthError(dcl1a.example.net)"
```

Cause

The device running Junos OS might fail to authenticate the user for the following reasons:

- The user does not have an account on the device running Junos OS.
- The user has an account with a text-based password configured on the device running Junos OS, but the wrong password or no password is supplied for the user when executing the module.
- The user has an account on the device running Junos OS 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 devices running Junos OS 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 [“Authenticating Users Executing Ansible Modules on Devices Running Junos OS” on page 51](#).

Troubleshooting Attribute conn_type Errors

Problem

Description:

During execution of a **juniper.device** module on a device running Junos OS, the Ansible control node generates the following error:

```
AttributeError: 'JuniperJunosModule' object has no attribute 'conn_type'
```

Cause

Whereas the the Juniper Networks **Juniper.junos** modules support 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: dcl
  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: dcl
  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

[Setting up Ansible for Junos OS Managed Nodes | 38](#)

[Authenticating Users Executing Ansible Modules on Devices Running Junos OS | 51](#)

[Troubleshooting Ansible Connection Errors When Managing Devices Running Junos OS | 200](#)

Troubleshooting Ansible Errors When Configuring Devices Running Junos OS

IN THIS SECTION

- [Troubleshooting Configuration Timeout Errors | 207](#)
- [Troubleshooting Configuration Lock Errors | 207](#)
- [Troubleshooting Configuration Load Errors | 208](#)
- [Troubleshooting Commit Errors | 208](#)

The following sections outline errors that you might encounter when using the `config` module in the `juniper.device` collection or the `juniper_junos_config` module in the `Juniper.junos` role to configure devices running Junos OS. These sections also present potential causes and solutions for each error.

Troubleshooting Configuration Timeout Errors

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.

Troubleshooting Configuration Lock Errors

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 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 device running Junos OS to give the current user the necessary permissions to make the changes.

Troubleshooting Configuration Load Errors

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

This error message is generated when the user executing the Ansible 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 device running Junos OS to give the current user the necessary permissions to make the changes.

Troubleshooting Commit Errors

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 [“Troubleshooting Configuration Lock Errors” on page 207](#). However, a configuration lock failed message might be generated as part of a **CommitError** instead of a **LockError** in the event that the 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"
  config:
    load: "merge"
    format: "text"
    src: "build_conf/{{ inventory_hostname }}/junos.conf"
    check_commit_wait: 5
    comment: "updated using Ansible"
```

RELATED DOCUMENTATION

| [Using Ansible to Configure Devices Running Junos OS | 169](#)