Engineering
Simplicity

Junos® OS

Salt for Junos OS Developer Guide

Published
2025-06-17

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

YEAR 2000 NOTICE

END USER LICENSE AGREEMENT

Table of Contents

About This Guide

Use this guide to include devices running Junos OS in your Salt operations or to use Salt to build an event-driven infrastructure (EDI) for devices running Junos OS.

RELATED DOCUMENTATION

Salt for Junos OS Quick Start Guide

Day One: Automating Junos With Salt

1

CHAPTER

Disclaimer

IN THIS CHAPTER

Salt for Junos OS Disclaimer

2

CHAPTER

Salt for Junos OS Overview

IN THIS CHAPTER

Understanding Salt for Junos OS

SUMMARY

You can use Salt to manage devices running Junos OS and define and enforce the state of a system, execute commands, and monitor, troubleshoot, and resolve issues in real time.

IN THIS SECTION

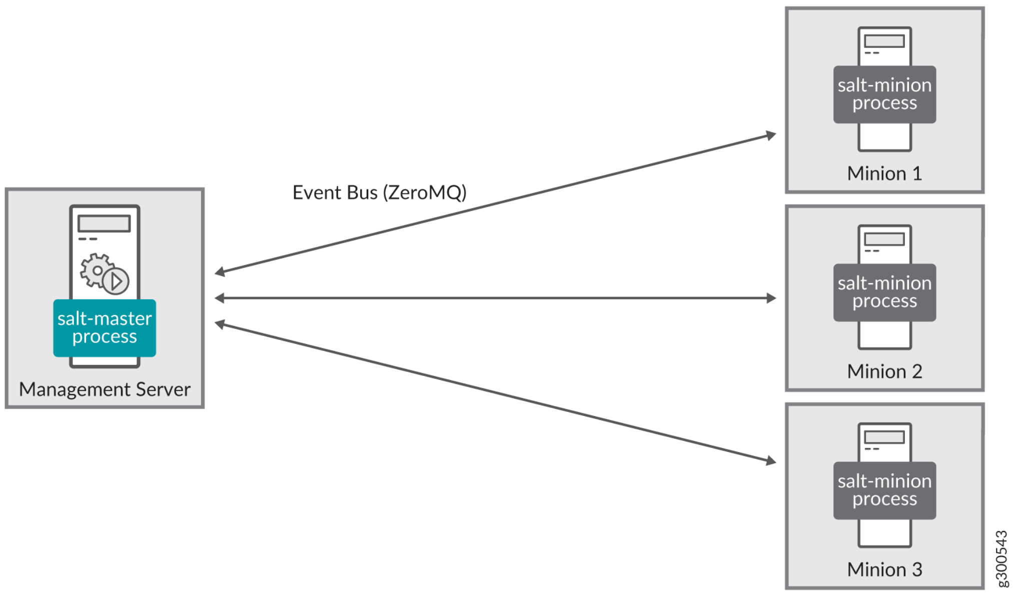
Salt for Junos OS Overview

Salt, or the SaltStack Platform, is a Python-based, open-source remote execution and configuration management tool. Salt enables you to define the state of a system and enforce that state on managed nodes. You can also use Salt to execute ad hoc commands on a device. The distributed design and persistent sessions make it fast, flexible, and highly scalable, with deployments that can manage thousands of devices across multiple vendors. The Salt installation includes modules that enable you to use Salt Open or SaltStack Enterprise to manage devices running Junos OS.

Salt differs from other configuration management tools in that it is event driven. The Salt architecture is centered around an event bus, which uses ZeroMQ as the default message and event transport mechanism for communication between components. Components communicate by sending messages through the bus, and external events can be injected on the bus. This enables you to build an event-driven infrastructure (EDI) that automatically responds to specific events.

The general Salt architecture is a server/client model in which the Salt master manages one or more Salt minions. Salt typically uses an agent-based architecture, which requires the managed nodes to run the salt-minion process, as shown in Figure 1 on page 5. The Salt master issues commands to the minions, and the minions execute the commands and return the results to the Salt master.

Figure 1: Salt Server/Client Architecture



However, Salt setups are flexible and allow for a masterless option, an agentless option, and the use of proxy minions. A proxy minion enables you to manage a device that does not directly run a Salt minion but can be managed through another protocol, for example, the Network Configuration Protocol (NETCONF). A Salt master can manage devices running Junos OS through a Salt proxy minion.

The Salt installation includes the Junos proxy module, which translates Salt operations into Junos OS-specific instructions and enables the Salt system to use a proxy minion to connect to and manage devices running Junos OS. The Junos proxy connects to the Salt master using the ZeroMQ event bus and uses the Juniper Junos PyEZ library to establish a NETCONF session over SSH with the device running Junos OS. The proxy minion process can run on the Salt master as shown in Figure 2 on page 5 or on one or more separate servers as shown in Figure 3 on page 6.

Figure 2: Junos Proxy Minions Same Server Architecture

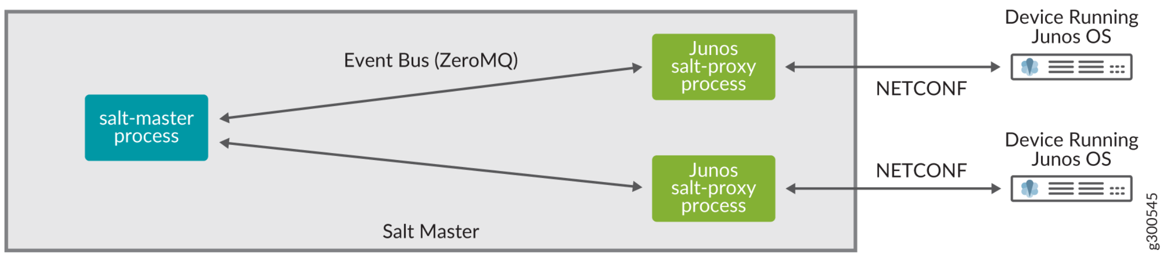
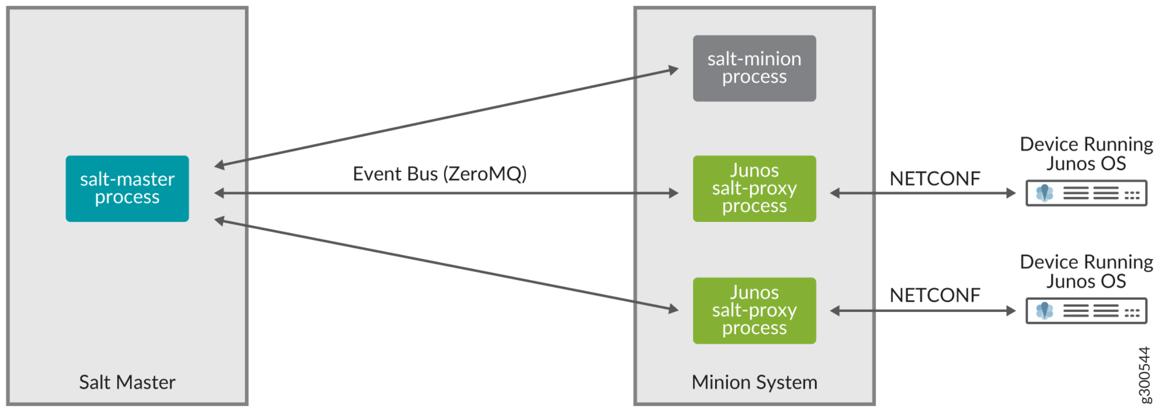


Figure 3: Junos Proxy Minion Separate Server Architecture



The Salt installation also includes the Junos execution and state modules (for Salt). These modules define functions that enable you to execute ad hoc commands or define and enforce particular states on devices running Junos OS. You can use the functions to automate key operational and configuration tasks in your network. For example, you can:

- Manage and provision configurations, including generating, loading and committing, and backing up configurations
- Retrieve and analyze operational information or configuration data
- Execute Junos OS CLI commands and RPCs
- Perform software installations
- Audit and validate operational states

The Salt event system uses a publish-subscribe (pub/sub) model. Events are published on the event bus, and subscribers listen for published events and react as directed. Salt provides methods to publish external events on the bus and to react to events. This enables you to use Salt to monitor and respond to events in real time.

The Salt installation includes the Junos syslog engine (for Salt), which listens for Junos OS system log messages from managed devices and publishes them on the Salt event bus. You can use the Junos syslog engine, in conjunction with Salt reactors, for event-based state management of devices running Junos OS. Salt reactors enable you to take action in response to events on the Salt event bus. When you enable the Salt Reactor system, it monitors the Salt event bus and triggers actions for matching events based on your configured rules. By enabling the Junos syslog engine and configuring custom reactors, you can automatically troubleshoot and correct common network issues.

Benefits of Salt and Salt for Junos OS

- Provide an efficient and scalable solution for managing large numbers of devices.
- Enable automatic enforcement of the correct state of a device.
- Shorten troubleshooting time and speed time to resolution for network issues by automating troubleshooting tasks.
- Improve network reliability and maximize network uptime by automatically responding to specific system events.
- Enable customized event-driven automation of devices running Junos OS. You can define which events to track and write custom reactors to respond to those events.

Additional Resources

This documentation assumes that the reader is familiar with the SaltStack Platform. For more information about Salt, including supported operating systems and installation instructions, see the SaltStack website at https://saltproject.io/.

Table 1 on page 7 describes a number of Salt and Salt for Junos OS resources.

Table 1: Salt for Junos OS Resources

Resource	Description	URL
Day One: Automating Junos with Salt	Day One book that includes fundamental concepts and working tutorials to enable you to use Salt to automate your network devices running Junos OS.	https://www.juniper.net/documentation/en_US/day-one-books/DO_Automating_SALT.pdf
Documentation	Documentation containing detailed information about installing and using Salt to perform operational and configuration tasks on devices running Junos OS.	https://www.juniper.net/documentation/product/us/en/salt-for-junos-os/

Table 1: Salt for Junos OS Resources *(Continued)*

Resource	Description	URL
Examples	GitHub repository and Wiki with additional information and examples for automating devices running Junos OS and performing event-driven automation with Salt.	https://github.com/ksator/junos-automation-with-saltstack/wiki
GitHub repositories	Public repository for the SaltStack platform.	https://github.com/saltstack/salt
Junos execution and state module documentation	Reference documentation for the Junos execution and state modules.	https://docs.saltproject.io/en/latest/ref/modules/all/salt.modules.junos.html https://docs.saltproject.io/en/latest/ref/states/all/salt.states.junos.html
Video tutorials	Short tutorials to help you get started using Salt to manage devices running Junos OS.	SaltStack Part 1: Basics and Installation SaltStack Part 2: Junos Proxy Modules SaltStack Part 3: Junos Syslog Engine

RELATED DOCUMENTATION

Understanding the Salt for Junos OS Components | 9

Understanding the Junos Execution and State Salt Modules | 20

How to Install Salt for Junos OS

Understanding the Salt for Junos OS Components

SUMMARY

Understand the different Salt components in general and as they relate to managing devices running Junos OS.

IN THIS SECTION

Components Overview

You can use Salt to manage devices running Junos OS. Table 2 on page 9 provides a high-level overview of the different Salt components. Each component is described in more detail in the sections that follow.

Table 2: Salt components

Component	Description
Salt master	Main control server that manages one or more minions.
Salt minion	Node managed with Salt.
Salt proxy	Process that runs on the Salt master or a separate server and enables the Salt master to manage devices that do not run a standard salt-minion process directly on the device. The Junos proxy (for Salt) enables the Salt master to manage devices running Junos OS.

Table 2: Salt components *(Continued)*

Component	Description
Grains	Static information or facts about a Salt minion. Salt retrieves facts from a device running Junos OS upon first connecting to the device. Junos OS facts are stored in Salt grains and can be accessed in the same way as other grains.
Pillars	User-defined data that is associated with and accessible by a minion.
State files	Define data structures that are used by the Salt system and can be applied to specific targets. For example, they can be used to define pillar data or the state of a system.
Top file	Maps groups of devices to the Salt state files that should be applied to them within a given environment.
Execution module	Defines execution functions that enable you to perform ad hoc tasks on minions from the Salt master command line. The Junos execution module (for Salt) enables you to execute operational and configuration tasks on devices running Junos OS.
State module	Defines state functions that enable you to declare and enforce the desired state of a minion. The Junos state module (for Salt) provides functions that you can use in Salt state files to declare and enforce a specific state on devices running Junos OS.
Junos syslog engine	Monitors system log messages sent from devices running Junos OS, extracts the event information, and publishes it in Salt format on the Salt event bus.
Reactor	Enables you to define actions to take for certain events received on the Salt event bus.

The following Salt components, which are integrated with the Salt software, are used to manage devices running Junos OS:

- Junos proxy
- Junos execution module

- Junos state module
- Junos syslog engine

Salt Master, Minions, and Proxy Minions

The *Salt master* is the main control server that manages one or more nodes or *Salt minions*. Salt typically uses an agent-based architecture in which the managed node runs the Salt minion process. Some devices, for example networking devices, might not support installing and running the Salt minion process directly on the device. In these cases, Salt supports using a proxy minion to manage a device that cannot directly run a Salt minion but can be managed through another protocol, for example, NETCONF. A minion (proxy or regular) is just a software process used to manage a device, and you can run multiple processes on the same server. A proxy minion process can run on either the Salt master or on a separate server, and by default, it is always connected to the managed node.

The standard Salt installation includes the Junos proxy module, which enables a Salt master to manage devices running Junos OS through a Salt proxy minion. The Junos proxy minion process can run on either the Salt master or a separate server and requires approximately 40 to 100 MB of RAM per process. You must have one proxy minion process for each managed device running Junos OS. The Junos proxy provides execution and state modules that enable you to perform operational and configuration tasks on devices running Junos OS.

The Junos proxy minion server must have the Junos PyEZ library and the `jxmlease` and `yamlordereddictloader` Python modules installed. Junos PyEZ is a microframework for Python that enables you to manage and automate devices running Junos OS. The Junos proxy minion connects to the Salt master using the ZeroMQ event bus and uses Junos PyEZ to establish a NETCONF session over SSH with the device running Junos OS. The `jxmlease` module converts XML to Python data structures and Python data structures to XML.

Grains

Grains are static pieces of data that the Salt master collects about a minion. Grain data includes properties about the system, for example, the model or serial number of the device. When the Salt master executes commands or enforces states, it can use grains to target certain groups of minions. For example, it might apply a state to all minions running a particular operating system or version.

When the Junos proxy minion establishes a connection with a device running Junos OS, it gathers the Junos OS facts and stores them in grains. Junos OS facts can be accessed in the same way as other grains.

To view the grains for a device running Junos OS, which includes the `junos_facts` grain, execute the `grains.items` function.

```
saltuser@salt-master:~$ sudo salt 'router1' grains.items
router1:
    ----------
    cpuarch:
        x86_64
    dns:
        ----------
        domain:
        ip4_nameservers:
            - 198.51.100.252
        ip6_nameservers:
        nameservers:
            - 198.51.100.252
        options:
            - edns0
        search:
            - example.com
        sortlist:
    ...
    junos_facts:
        ----------
        2RE:
            True
        HOME:
            /var/home/saltuser
        RE0:
            ----------
            last_reboot_reason:
                0x1:power cycle/failure
            mastership_state:
                master
            model:
                RE-MX-104
            status:
                OK
            up_time:
                29 days, 2 hours, 35 minutes, 25 seconds
        ...
```

If you want to display just the Junos OS facts, you can request just the `junos_facts` grain item.

```
saltuser@salt-master:~$ sudo salt 'router1' grains.item junos_facts
router1:
    ----------
    junos_facts:
        ----------
        2RE:
            True
        HOME:
            /var/home/saltuser
        RE0:
            ----------
            last_reboot_reason:
                0x1:power cycle/failure
            mastership_state:
                master
            model:
                RE-MX-104
            status:
                OK
            up_time:
                29 days, 2 hours, 37 minutes, 15 seconds
        ...
```

Alternatively, you can execute the `facts` function from the `junos` execution module. The `junos.facts` function returns the same data as the previous command.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.facts
router1:
    ----------
    facts:
        ----------
        2RE:
            True
        HOME:
            /var/home/saltuser
        RE0:
            ----------
            last_reboot_reason:
                0x1:power cycle/failure
            mastership_state:
                master
            model:
                RE-MX-104
            status:
                OK
            up_time:
                29 days, 2 hours, 37 minutes, 15 seconds
        ...
```

The device facts are retrieved and cached when the Junos proxy minion connects to the device running Junos OS. When you display the facts, they are served from the cache. As a result, if the value of an already-cached fact changes, it is not automatically updated in the Salt system. To reload the facts from the device and display them, call the `junos.facts_refresh` function.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.facts_refresh
```

You can use grains to refine the set of minions on which to perform tasks. For example, the following command only executes the `show chassis routing-engine` command on devices running Junos OS that have dual Routing Engines:

```
saltuser@salt-master:~$ sudo salt -G 'junos_facts:2RE:True' junos.cli 'show chassis routing-
engine'
```

Pillars

The Salt pillar system enables you to define data that can be distributed to minions. Salt pillars are treelike structures of user-defined data that are defined on the Salt master and are accessible by minions. Pillar data might define configuration data, user-defined variables, or sensitive information associated with the minion. You can restrict the pillar data that a minion can access by mapping the pillar to the appropriate targets in the pillar top file.

Pillar data can be defined in simple YAML files but can also be stored in and retrieved from a database such as SQL. By default, pillar data files are stored in the Salt master's **/srv/pillar** directory. To change the location of the pillar data, set the `pillar_roots` parameter in the Salt master configuration file to the desired directory.

When managing devices running Junos OS, pillars are used to define the data for the Junos proxy, including the hostname and authentication information for a given device running Junos OS. The following sample pillar file contains the Junos proxy configuration for a device running Junos OS:

```
saltuser@salt-master:~$ cat /srv/pillar/router1-proxy.sls
proxy:
  proxytype: junos
  host: router1.example.com
  username: saltuser
  password: lab123
  port: 830   # NETCONF port
```

The Salt pillar top file, which generally resides at **/srv/pillar/top.sls**, defines the pillar data that a minion can access. When managing devices running Junos OS, the pillar top file maps the Junos proxy minion ID to the pillar file containing the proxy data for the corresponding device running Junos OS. In the following example, the top file maps the proxy minion ID router1 to the **router1-proxy.sls** pillar file in the base environment:

```
saltuser@salt-master:~$ cat /srv/pillar/top.sls
base:
  'router1':          # proxy minion ID
    - router1-proxy   # State file name
  'router2':
    - router2-proxy
```

You can view a minion's pillar data by executing the `pillar.items` function.

```
saltuser@salt-master:~$ sudo salt 'router1' pillar.items

router1:
    ----------
    proxy:
        ----------
        host:
            router1.example.com
        password:
            lab123
        port:
            830
        proxytype:
            junos
```

```
          username:
              saltuser
```

Salt State (SLS) Files

SaLt State (SLS) files define data structures that are used by the Salt system for various purposes. For example, they can be used to define pillar data or the state of a system. By default, Salt state files are written in YAML format and use the **.sls** file extension. State files support using Jinja syntax for added flexibility.

Salt enables you to work with managed systems' configurations in a declarative manner. The Junos state module (for Salt), `salt.states.junos`, provides functions that you can use in state files to declare and enforce certain states on devices running Junos OS. For example, you might want to ensure that the device configuration includes a specific set of statements or that the device reflects a specific operational state.

The general structure for an SLS file that defines the state for a managed device is as follows, although this can vary depending on the module and function:

```
State name:
  module.function:
    - param1: value1
    - param2: value2
```

For example, the following state file uses the `junos.install_config` function to apply the configuration in the given template to the target devices running Junos OS:

```
saltuser@salt-master:~$ cat /srv/salt/junos_bgp_config.sls
Apply BGP configuration:
  junos.install_config:
    - name: salt://configs/junos-config-bgp-template.set
    - comment: Configuring BGP using Salt
    - diffs_file: /var/log/salt/output/{{ grains['id'] }}_junos_bgp_config_diff
```

You can apply an individual state to one or more target devices. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_bgp_config
```

Alternatively, the Salt top file can be used to apply one or more states to a minion during execution of a *highstate*.

```
saltuser@salt-master:~$ cat /srv/salt/top.sls
base:
  router1:
    - junos_bgp_config
    - junos_ospf_config
```

When you execute a highstate on a given target, the Salt system applies all of the states that are configured in the `top.sls` file for that target.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply
```

For more information about the Junos state module, see "Understanding the Junos Execution and State Salt Modules" on page 20.

Top File

When you use Salt to manage devices, you can group minions into different categories, or environments. By default, all minions are in the base environment, but you might place other minions in a development or production environment. A Salt top file maps a device or group of devices to the state files that should be applied to them and to the pillar data that they can access within the given environment. Top files are named **top.sls** by default, and exist at the top of the directory hierarchy that contains Salt state files.

By default, the top file used for states resides at **/srv/salt/top.sls**. It uses the following syntax to map the target devices to their corresponding state files within the given environment. The file must always define the `base` environment.

```
environment1:
  target:
    - state1
    - state2

environment2:
  target:
    - state1
    - state2
```

For example, the following top file applies the **core.sls** and **common_config.sls** state files to all minions in the base environment that have an ID starting with "router":

```
base:
  'router*'
    - core
    - common_config
```

To apply the states, execute a highstate and specify the target devices.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply
```

Similarly the Pillar top file, which generally resides at **/srv/pillar/top.sls**, maps a pillar file to the minions that are allowed to access the data in that file. When managing devices running Junos OS, the Pillar top file maps a Junos proxy minion ID to the state file containing the pillar data for the corresponding device running Junos OS, as described in the "Pillars" on page 14 section.

Execution and State Modules

Salt is a configuration management and remote execution tool that enables you to declare and enforce the state of a system and perform operational and configuration tasks on managed devices. Salt execution modules are Python modules that run on a Salt minion. They define functions that enable you to perform tasks on the minion. You can execute ad hoc commands on the Salt master that instruct the target minions to execute the given function and return the result. Similarly, Salt state modules define functions that manage the application of a state to a minion.

For example, to test connectivity to all minions, you can execute the `test` module's `ping` function and specify the target as `'*'` to indicate all minions.

```
saltuser@master:~$ sudo salt '*' test.ping
router1:
    True
router2:
    True
```

The Salt installation includes the Junos execution and state modules, which define functions that enable you to perform operational and configuration tasks and manage the application of states on devices running Junos OS. For detailed information about using the Junos execution and state modules, see "Understanding the Junos Execution and State Salt Modules" on page 20.

Junos Syslog Engine, Salt Event Bus, and Salt Reactors

The Salt architecture is centered around a high-performance event bus, which uses ZeroMQ as the default message and event transport mechanism for communication between the Salt master and minions. Components communicate by sending messages through the bus, and external events can be injected on the bus. The Salt master and each Salt minion has its own event bus.

The event system uses a publish-subscribe (pub/sub) model. Events are published on the event bus, and subscribers listen for published events and react as directed. For example, when you execute a function on a minion, the Salt master publishes a job event, and the matching minion executes the job and fires a return event, which contains the job results, onto the bus. Salt provides methods to publish external events on the bus and to react to events, which enables you to use Salt to monitor and respond to events in real time.

Salt engines are external system processes that enable you to export or import events on the Salt event bus. The Salt installation includes the Junos syslog engine (for Salt), which can monitor Junos OS system log (syslog) messages that are sent to the Salt server and publish them on the Salt event bus. When enabled, the Junos syslog engine listens on the specified port for syslog events from managed devices running Junos OS. When the Junos syslog engine receives an event, it extracts the event information, translates it to Salt format, and publishes it on the Salt event bus.

The following output is a sample `UI_COMMIT_COMPLETED` event that was sent from a device running Junos OS and published on the Salt event bus by the Junos syslog engine:

```
jnpr/syslog/router1/UI_COMMIT_COMPLETED  {
    "_stamp": "2019-07-24T17:17:30.390374",
    "daemon": "mgd",
    "event": "UI_COMMIT_COMPLETED",
    "facility": 23,
    "hostip": "198.51.100.2",
    "hostname": "router1",
    "message": "commit complete",
    "pid": "5795",
    "priority": 188,
    "raw": "<188>Jul 24 10:17:38 router1 mgd[5795]: UI_COMMIT_COMPLETED: commit complete",
    "severity": 4,
    "timestamp": "2019-07-24 17:17:30"
```

You can use the Junos syslog engine, in conjunction with Salt reactors, for event-based state management of devices running Junos OS. Salt reactors enable you to take action in response to events on the Salt event bus. When you enable the Salt Reactor system, it monitors the Salt event bus and triggers actions for matching events based on your configured rules. To enable the Salt Reactor system, you configure the `reactor` option in the Salt master configuration file, and associate event tags with one or more reactor SLS files. The reactor files define the actions to take when that event occurs.

For more information about the Junos syslog engine and creating reactors for Junos OS events, see:

- "Junos Syslog Engine for Salt" on page 130
- "Configuring Salt Reactors for Junos OS Events" on page 137
- https://docs.saltstack.com/en/latest/ref/engines/all/salt.engines.junos_syslog.html

RELATED DOCUMENTATION

Understanding Salt for Junos OS | 4

Understanding the Junos Execution and State Salt Modules

SUMMARY

The Junos execution and state Salt modules enable you to execute ad hoc commands or define and enforce particular states on devices running Junos OS.

IN THIS SECTION

Junos Execution and State Modules Overview

Salt is a configuration management and remote execution tool that enables you to declare and enforce the state of a system and perform operational and configuration tasks on managed devices. Salt uses Python modules to perform these tasks. The two main types of modules are: execution modules and state modules.

Salt execution modules are Python modules that run on a Salt minion. They define functions that enable you to perform tasks on the minion. You can execute ad hoc commands on the Salt master that instruct the target minions to execute the given function and return the result. Similarly, Salt state modules define functions that are used in Salt state files to manage the application of a state to a minion.

The Salt installation includes the following execution and state modules, which enable you to interact with devices running Junos OS:

- `salt.modules.junos`—The Junos execution module (for Salt) defines functions that enable you to perform operational and configuration tasks on devices running Junos OS. For example, you can execute an operational command or RPC on the device, modify the configuration, or install a new software image.
- `salt.states.junos`—The Junos state module (for Salt) defines functions that you can use in Salt state files to declare and enforce a specific state on devices running Junos OS. For example, you might want to ensure that the configuration includes a specific set of statements or that the device reflects a specific operational state. The Junos state functions invoke the Junos execution function of the same name to perform these tasks.

When Salt manages devices running Junos OS through the Junos proxy (for Salt) and you call a function, the Python module does not run directly on the device. The proxy minion instead uses Junos PyEZ to send remote procedure calls (RPCs) to the NETCONF server on the managed device to perform the requested tasks. When you execute the function on multiple targets, Salt performs the task on the different devices in parallel.

Table 3 on page 21 lists the functions defined in the Junos execution and state modules and provides a brief description of each function. With the exception of the `facts`, `facts_refresh`, and `ping` functions, the same functions are defined in both modules.

Table 3: Junos Execution and State Module Functions

Function Name	salt.modules.junos	salt.states.junos	Description
`cli`	Y	Y	Execute a CLI command and return the command output in the specified format.

Table 3: Junos Execution and State Module Functions *(Continued)*

Function Name	salt.modules.junos	salt.states.junos	Description
`commit`	Y	Y	Commit the changes loaded into the target configuration database.
`commit_check`	Y	Y	Perform a `commit check` operation on the candidate configuration to verify that the syntax is correct.
`diff`	Y	Y	Return the difference between the candidate configuration and the specified rollback configuration.
`facts`	Y	–	Display the device facts that were gathered during the connection phase.
`facts_refresh`	Y	–	Reload the device facts from the target device and refresh the facts stored in the Salt grains.
`file_copy`	Y	Y	Copy a file from the Salt master or the proxy minion server to the device running Junos OS.
`get_table`	Y	Y	Retrieve operational or configuration data from a device running Junos OS using a Junos PyEZ Table and View.
`install_config`	Y	Y	Lock, modify, commit, and unlock the target configuration database.
`install_os`	Y	Y	Install a Junos OS software image on a device.
`load`	Y	Y	Load the configuration data in the specified file into the target configuration database.
`lock`	Y	Y	Request an exclusive lock on the candidate configuration.

Table 3: Junos Execution and State Module Functions *(Continued)*

Function Name	salt.modules.junos	salt.states.junos	Description
`ping`	Y	–	Send an Internet Control Message Protocol (ICMP) ping from a device running Junos OS to the specified target and display the results.
`rollback`	Y	Y	Roll back the configuration to a previously committed configuration and commit it.
`rpc`	Y	Y	Execute the specified RPC on the target device.
`set_hostname`	Y	Y	Configure the hostname of a managed device running Junos OS and commit the change.
`shutdown`	Y	Y	Reboot or shut down a device running Junos OS.
`unlock`	Y	Y	Release the exclusive lock on the candidate configuration.
`zeroize`	Y	Y	Restore the device to the factory-default configuration settings.

How to Use the Junos Execution Functions

Salt execution modules enable you to perform ad hoc tasks on one or more minions. To invoke an execution function from the Salt master command line, use the following syntax:

```
salt options 'target' module.function arguments
```

Where:

- *arguments*—Required or optional function arguments.

- *target*—Minions on which to execute the function. You can specify a single minion, use `'*'` to target all minions, or define an expression that targets a group of minions, for example, `'router*'` to include all minions whose name starts with `router`.
- *module*—Name of the Salt execution module containing the function to execute.
- *function*—Name of the function to execute.

For example, to test connectivity to all minions, execute the `test` module's `ping` function and use `'*'` for the target.

```
saltuser@salt-master:~$ sudo salt '*' test.ping
router1:
    True
router2:
    True
```

To invoke Junos execution functions, specify the `junos` module and the desired function. For example, the following command invokes the `cli` function to execute the `show version` operational mode command on router1:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.cli 'show version'
router1:
    ----------
    message:

        Hostname: router1
        Model: mx104
        Junos: 18.4R1.8
        JUNOS Base OS boot [18.4R1.8]
        JUNOS Base OS Software Suite [18.4R1.8]
        JUNOS Crypto Software Suite [18.4R1.8]
        JUNOS Packet Forwarding Engine Support (TRIO) [18.4R1.8]
        JUNOS Web Management [18.4R1.8]
        JUNOS Online Documentation [18.4R1.8]
        JUNOS SDN Software Suite [18.4R1.8]
        JUNOS Services Application Level Gateways [18.4R1.8]
        JUNOS Services COS [18.4R1.8]
        JUNOS Services Jflow Container package [18.4R1.8]
        JUNOS Services Stateful Firewall [18.4R1.8]
        JUNOS Services NAT [18.4R1.8]
        JUNOS Services RPM [18.4R1.8]
        JUNOS Services SOFTWIRE [18.4R1.8]
        JUNOS Services Captive Portal and Content Delivery Container package [18.4R1.8]
        JUNOS Macsec Software Suite [18.4R1.8]
        JUNOS Services Crypto [18.4R1.8]
        JUNOS Services IPSec [18.4R1.8]
        JUNOS DP Crypto Software Software Suite [18.4R1.8]
        JUNOS py-base-powerpc [18.4R1.8]
        JUNOS py-extensions-powerpc [18.4R1.8]
        JUNOS jsd [powerpc-18.4R1.8-jet-1]
        JUNOS Kernel Software Suite [18.4R1.8]
        JUNOS Routing Software Suite [18.4R1.8]
    out:
        True
```

When you execute commands on the Salt master command line, Salt passes the CLI input through `PyYAML` to ensure it is loaded as a proper Python data type. In some cases, arguments that take an integer or that take a string value that parses to an integer might not be parsed correctly. For those arguments, you can provide the value using one of the following methods:

- Use a backslash (\ ) to escape quotation marks around string values.
- Use double quotation marks to enclose single quotation marks and vice versa.
- Include the `--no-parse=`*`param_name`* option.

For example, the following commands schedule the device to reboot at the given time:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at=\'16:10\'
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True "at='16:10'"
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at="'16:10'"
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at='16:10' --no-parse=at
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at=\'+2\'
```

The following commands request information for the Routing Engine in the specified slot:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-route-engine-information slot=\'0\'
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-route-engine-information slot='0' --no-
parse=slot
```

How to Use the Junos State Functions

You can invoke Junos state module functions from Salt state files. The syntax varies by function. Check the module documentation for the proper syntax before using a function.

Many of the state functions use the following general structure:

```
State name:
  module.function:
    - param1: value1
    - param2: value2
```

For example, the following state file uses the `junos.install_config` function to apply the configuration in the given template to the target devices:

```
saltuser@salt-master:~$ cat /srv/salt/junos_bgp_config.sls
Apply BGP configuration:
  junos.install_config:
    - name: salt://configs/junos-config-bgp-template.set
    - comment: Configuring BGP using Salt
    - diffs_file: /var/log/salt/output/{{ grains['id'] }}_junos_bgp_config_diff
```

However, in some cases, the first line of the structure can supply a function argument. For example, the `rpc` function can supply the RPC either by using the previous syntax or by providing it as the first line of the data structure. The following state files, which are equivalent, invoke the `rpc` function to retrieve information for the ge-1/0/0 interface, but they provide the RPC argument using different methods:

```
saltuser@salt-master:~$ cat /srv/salt/junos-rpc.sls
Retrieve interface information:
  junos.rpc:
    - name: get-interface-information
    - interface_name: ge-1/0/0
    - terse: True
    - dest: /tmp/interface.log
    - format: text
```

```
saltuser@salt-master:~$ cat /srv/salt/junos-rpc2.sls
get-interface-information:
  junos.rpc:
    - interface_name: ge-1/0/0
    - terse: True
    - dest: /tmp/interface.log
    - format: text
```

When you apply the state to the target device, it executes the `get-interface-information` RPC for the given interface and returns the result. Salt displays the terse output on the terminal and also saves the output in text format to the **/tmp/interface.log** file on the proxy minion server.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos-rpc2
router1:
-
          ID: get-interface-information
    Function: junos.rpc
      Result: True
     Comment:
     Started: 20:56:46.669149
    Duration: 871.23 ms
     Changes:
              -
              out:
                  True
              rpc_reply:

                  Interface               Admin Link Proto    Local                 Remote
                  ge-0/0/0                up    up
                  ge-0/0/0.0              up    up   inet     192.0.2.2/24
                                                     multiservice


Summary for router1
-
Succeeded: 1 (changed=1)
Failed:    0
-
Total states run:     1
Total run time: 871.230 ms
```

RELATED DOCUMENTATION

Understanding the Salt for Junos OS Components | **9**

3

CHAPTER

Use Salt to Manage Device Operations

IN THIS CHAPTER

Use Salt to Execute Operational Commands on Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.cli` execution function and the `salt.states.junos.cli` state function enable you to execute operational mode commands on devices running Junos OS to perform operations or retrieve information.

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

junos.cli Function Syntax

The `salt.modules.junos.cli` execution function syntax is:

```
salt 'target' junos.cli 'command' dest=dest format=format
```

The `salt.states.junos.cli` state function supports the following syntaxes:

```
command:
  junos.cli:
    - dest: dest
    - format: format
```

```
id:
  junos.cli:
    - name: command
    - dest: dest
    - format: format
```

where:

- *command*—Operational mode command to execute on the device running Junos OS, for example, `show interfaces fxp0.0 terse`.
- *dest*—(Optional) Path of the destination file on the proxy minion server to which the command output is written. If you do not specify an absolute path on the target device, the path is relative to the top-level root (/) directory.
- *format*—(Optional) Format of the command output. Specify `text` or `xml`.
- *id*—User-defined identifier for the state declaration.

How to Execute Commands with the junos.cli Execution Function

The `salt.modules.junos.cli` execution function enables you to execute a CLI command on a device running Junos OS. The function requires one argument, which is the command to execute. If the operation is successful, the command returns `out: True`, and the `message` key includes the command output.

For example, the following command executes the `show system uptime` operational mode command on the target device and displays the command output in standard output:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.cli 'show system uptime'
router1:
    ----------
    message:

        Current time: 2019-07-23 11:04:18 PDT
        Time Source:  NTP CLOCK
        System booted: 2019-07-01 07:03:56 PDT (3w1d 04:00 ago)
        Protocols started: 2019-07-01 07:07:27 PDT (3w1d 03:56 ago)
        Last configured: 2019-07-18 16:16:33 PDT (4d 18:47 ago) by saltuser
        11:04AM  up 22 days, 4 hrs, 0 users, load averages: 0.02, 0.05, 0.06
    out:
        True
```

How to Execute Commands with the junos.cli State Function

You can use the `salt.states.junos.cli` state function within a Salt state file to execute one or more operational commands on a device running Junos OS. You can define the command as the first line of the data structure, or you can define it within the function's argument list. If you need to execute the same command multiple times within the same state file, you must define the command within the argument list.

The following state file uses the `junos.cli` state function to execute two commands on the target device and save the output for each command in a separate file. In this case, the proxy identifier stored in the `id` grain is included in the destination filename to distinguish the output files when the state is applied to multiple targets.

```
saltuser@salt-master:~$ cat /srv/salt/junos_cli.sls
show system uptime:
  junos.cli:
    - dest: /var/log/salt/output/{{ grains['id'] }}_junos_system_uptime.txt

show version:
  junos.cli:
    - dest: /var/log/salt/output/{{ grains['id'] }}_junos_show_version.txt
```

When you apply the state, the Salt master displays the command output in standard output and also saves the output to the corresponding destination file on the proxy minion server.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_cli
router1:
----------
          ID: show system uptime
    Function: junos.cli
      Result: True
     Comment:
     Started: 21:18:12.130459
    Duration: 911.207 ms
```

```
     Changes:
              ----------
              message:

                  Current time: 2019-07-26 14:18:12 PDT
                  Time Source:  NTP CLOCK
                  System booted: 2019-07-01 07:03:56 PDT (3w4d 07:14 ago)
                  Protocols started: 2019-07-01 07:07:27 PDT (3w4d 07:10 ago)
                  Last configured: 2019-07-24 10:17:34 PDT (2d 04:00 ago) by saltuser
                   2:18PM  up 25 days,  7:14, 1 user, load averages: 0.09, 0.06, 0.07
              out:
                  True
----------
          ID: show version
    Function: junos.cli
      Result: True
     Comment:
     Started: 21:18:13.041796
    Duration: 968.359 ms
     Changes:
              ----------
              message:

                  Hostname: router1
                  Model: mx104
                  Junos: 18.4R1.8
                  JUNOS Base OS boot [18.4R1.8]
                  JUNOS Base OS Software Suite [18.4R1.8]
                  JUNOS Crypto Software Suite [18.4R1.8]
                  JUNOS Packet Forwarding Engine Support (TRIO) [18.4R1.8]
                  JUNOS Web Management [18.4R1.8]
                  JUNOS Online Documentation [18.4R1.8]
                  JUNOS SDN Software Suite [18.4R1.8]
                  JUNOS Services Application Level Gateways [18.4R1.8]
                  JUNOS Services COS [18.4R1.8]
                  JUNOS Services Jflow Container package [18.4R1.8]
                  JUNOS Services Stateful Firewall [18.4R1.8]
                  JUNOS Services NAT [18.4R1.8]
                  JUNOS Services RPM [18.4R1.8]
                  JUNOS Services SOFTWIRE [18.4R1.8]
                  JUNOS Services Captive Portal and Content Delivery Container package [18.4R1.8]
                  JUNOS Macsec Software Suite [18.4R1.8]
                  JUNOS Services Crypto [18.4R1.8]
                  JUNOS Services IPSec [18.4R1.8]
                  JUNOS DP Crypto Software Software Suite [18.4R1.8]
                  JUNOS py-base-powerpc [18.4R1.8]
                  JUNOS py-extensions-powerpc [18.4R1.8]
                  JUNOS jsd [powerpc-18.4R1.8-jet-1]
                  JUNOS Kernel Software Suite [18.4R1.8]
                  JUNOS Routing Software Suite [18.4R1.8]
              out:
                  True

Summary for router1
------------
Succeeded: 2 (changed=2)
Failed:    0
------------
Total states run:     2
Total run time:   1.880 s
```

```
saltuser@minion:~$ ls /var/log/salt/output
router1_junos_show_version.txt  router1_junos_system_uptime.txt
```

If the state is applied to multiple targets, Salt generates different output files for each target on the proxy minion server in this case, because the defined filename references a unique identifier.

```
saltuser@salt-master:~$ sudo salt '*' state.apply junos_cli
router1:
----------
          ID: show system uptime
    Function: junos.cli
    ...
router2:
----------
          ID: show system uptime
    Function: junos.cli
    ...
```

```
saltuser@minion:~$ ls /var/log/salt/output
router1_junos_show_version.txt  router1_junos_system_uptime.txt
router2_junos_show_version.txt  router2_junos_system_uptime.txt
```

State files cannot use the same state identifier multiple times in a file. Therefore, if you want to execute the same command multiple times in a state file, for example, before and after you modify the configuration, you must define the command in the function's argument list. For example:

```
Get OSPF neighbor information:
  junos.cli:
    - name: show ospf neighbor
    - format: text

Install OSPF config :
  junos.install_config:
    - name: salt://configs/junos_ospf_config.conf

Get OSPF neighbor information after configuration changes:
  junos.cli:
    - name: show ospf neighbor
    - format: text
```

How to Save the Command Output to a File

When you execute the `junos.cli` function, you can save the returned data in a file on the proxy minion server by including the `dest` argument and specifying the path of the destination file. If you do not specify an absolute path, the path is relative to the top-level root (/) directory. If an output file already exists with the target name, the new file overwrites the old file.

The `dest` argument does not save the entire Salt command response in the file. It only saves the command output contained within the `message` key.

The following command displays the output from the `show system uptime` command and also saves it to the specified file on the proxy minion server:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.cli 'show system uptime' dest='/home/saltuser/
cli_output.txt'
```

```
saltuser@minion:~$ ls
cli_output.txt
```

How to Specify the Format for the Command Output

By default, the `junos.cli` function returns the command output in text format for both the data displayed in standard output as well as the data saved to the destination file, if specified. You can also return the data in XML format. To specify XML format, include the `format` argument, and set the value equal to `xml`.

The following command executes the `show system uptime` command and returns the data as XML. In standard output, the elements are displayed in a hierarchy without the traditional brackets enclosing each element name.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.cli 'show system uptime' format=xml dest=/home/
saltuser/show-system-uptime.xml
router1:
    ----------
    message:
        ----------
        system-uptime-information:
            ----------
            current-time:
                ----------
                date-time:
                    2019-07-29 15:38:21 PDT
            last-configured-time:
                ----------
                date-time:
                    2019-07-24 10:17:34 PDT
                time-length:
                    5d 05:20
                user:
                    saltuser
            protocols-started-time:
                ----------
                date-time:
                    2019-07-01 07:07:27 PDT
                time-length:
                    4w0d 08:30
            system-booted-time:
                ----------
                date-time:
                    2019-07-01 07:03:56 PDT
                time-length:
                    4w0d 08:34
            time-source:
                NTP CLOCK
            uptime-information:
                ----------
                active-user-count:
                    1
                date-time:
                    3:38PM
                load-average-1:
                    0.04
                load-average-15:
                    0.04
                load-average-5:
                    0.05
                up-time:
                    28 days,  8:34
    out:
        True
```

The output in the destination file on the proxy minion server displays the same Junos XML tags that would be emitted if you execute the `show system uptime | display xml` command in the Junos OS CLI.

```
saltuser@minion:~$ cat /home/saltuser/show-system-uptime.xml
<system-uptime-information><current-time><date-time seconds="1564439901">2019-07-29 15:38:21
PDT</date-time></current-time><time-source>
 NTP CLOCK
</time-source><system-booted-time><date-time seconds="1561989836">2019-07-01 07:03:56 PDT</date-
time><time-length seconds="2450065">4w0d 08:34</time-length></system-booted-time><protocols-
started-time><date-time seconds="1561990047">2019-07-01 07:07:27 PDT</date-time><time-length
seconds="2449854">4w0d 08:30</time-length></protocols-started-time><last-configured-time><date-
time seconds="1563988654">2019-07-24 10:17:34 PDT</date-time><time-length seconds="451247">5d
05:20</time-length><user>saltuser</user></last-configured-time><uptime-information><date-time
seconds="1564439902">
3:38PM
</date-time><up-time seconds="2450096">
28 days,  8:34
</up-time><active-user-count format="1 user">
1
</active-user-count><load-average-1>
0.04
</load-average-1><load-average-5>
0.05
```

```
</load-average-5><load-average-15>
0.04
</load-average-15></uptime-information></system-uptime-information>
```

RELATED DOCUMENTATION

Use Salt to Execute RPCs on Devices Running Junos OS | **38**

Use Salt to Execute RPCs on Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.rpc` execution function and the `salt.states.junos.rpc` state function enable you to execute remote procedure calls (RPCs) on devices running Junos OS to perform operations or retrieve information.

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

Understanding the Junos XML API

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 commands that you issue in CLI operational mode. 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.

All operational commands that have Junos XML counterparts are listed in the Junos XML API Explore. You can also display the Junos XML request tag element for any operational mode command that has a Junos XML counterpart on the CLI by appending the `| display xml rpc` option after the command. The following example displays the request tag for the `show route` command:

```
user@router> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/19.2R1/junos">
    <rpc>
        <get-route-information>
        </get-route-information>
    </rpc>
</rpc-reply>
```

When you use Salt to manage a device running Junos OS, you can use the `junos.rpc` function to execute the RPC on the device and return the response.

junos.rpc Function Syntax

The `salt.modules.junos.rpc` execution function syntax is:

```
salt 'target' junos.rpc rpc dest=dest format=format arg1=arg1-value arg2=arg2-value
```

The `salt.states.junos.rpc` state function supports the following syntaxes:

```
rpc:
  junos.rpc:
    - dest: dest
    - format: format
    - arg1: arg1-value
    - arg2: arg2-value
```

```
id:
  junos.rpc:
    - name: rpc
    - dest: dest
    - format: format
    - arg1: arg1-value
    - arg2: arg2-value
```

where:

- *arg=arg-value*—(Optional) One or more RPC arguments to include and their corresponding values, for example, `interface-name=ge-1/1/1` or `terse=True`.
- *dest*—(Optional) Path of the destination file on the proxy minion server to which the RPC reply is written. If you do not specify an absolute path on the target device, the path is relative to the top-level root (/) directory.
- *id*—User-defined identifier for the state declaration.
- *format*—(Optional) Format of the RPC reply written to the destination file, if specified. Specify `text`, `xml`, or `json`.
- *rpc*—Remote procedure call to execute on the device running Junos OS, for example, `get-system-uptime-information`.

How to Execute RPCs with the junos.rpc Execution Function

The `salt.modules.junos.rpc` execution function enables you to execute an RPC on a device running Junos OS. The function requires one argument, which is the RPC to execute. If the operation is successful, the command returns `out: True`, and the `rpc_reply` key includes the RPC reply.

For example, the following command executes the `get-system-uptime-information` RPC on the target device and displays the response in standard output:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-system-uptime-information
router1:
    ----------
    out:
        True
    rpc_reply:
        ----------
        system-uptime-information:
            ----------
            current-time:
                ----------
                date-time:
                    2019-06-13 17:01:32 PDT
            last-configured-time:
                ----------
                date-time:
                    2019-06-12 18:47:12 PDT
                time-length:
                    22:14:20
                user:
                    saltuser
        ...
```

For information about specifying RPC arguments or the function's output format and location, see:

- "How to Specify RPC Arguments" on page 45
- "How to Specify the Format for the RPC Output" on page 47
- "How to Save the RPC Output to a File" on page 46

How to Execute RPCs with the junos.rpc State Function

You can use the `salt.states.junos.rpc` state function within a Salt state file to execute RPCs on a device running Junos OS. You can define the RPC as the first line of the data structure, or you can define it within the function's argument list. If you need to execute the same RPC multiple times within the same state file, you must define the RPC within the argument list.

The following state file uses the `junos.rpc` state function to execute two RPCs on the target device and save the output to a file. In this case, the proxy identifier stored in the `id` grain is included in the destination filename to distinguish the output files when the state is applied to multiple targets.

```
saltuser@salt-master:~$ cat /srv/salt/junos_rpc.sls
get-system-uptime-information:
  junos.rpc:
    - dest: /var/log/salt/output/{{ grains['id'] }}_junos_rpc_system_uptime.xml

get-interface-information:
  junos.rpc:
    - interface-name: fxp0.0
    - terse: True
    - dest: /var/log/salt/output/{{ grains['id'] }}_fxp0_status.txt
    - format: text
```

When you apply the state, the Salt master displays the RPC reply for each RPC in standard output and also saves the output to the corresponding destination file on the proxy minion server.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_rpc
router1:
----------
          ID: get-system-uptime-information
    Function: junos.rpc
      Result: True
     Comment:
     Started: 23:49:00.633541
    Duration: 180.134 ms
     Changes:
              ----------
              out:
                  True
              rpc_reply:
                  ----------
                  system-uptime-information:
                      ----------
                      current-time:
                          ----------
                          date-time:
                              2019-07-26 16:49:00 PDT
                      last-configured-time:
                          ----------
                          date-time:
                              2019-07-24 10:17:34 PDT
                          time-length:
                              2d 06:31
                          user:
                              saltuser
```

```
                        protocols-started-time:
                            ----------
                            date-time:
                                2019-07-01 07:07:27 PDT
                            time-length:
                                3w4d 09:41
                        system-booted-time:
                            ----------
                            date-time:
                                2019-07-01 07:03:56 PDT
                            time-length:
                                3w4d 09:45
                        time-source:
                            NTP CLOCK
                        uptime-information:
                            ----------
                            active-user-count:
                                1
                            date-time:
                                4:49PM
                            load-average-1:
                                0.01
                            load-average-15:
                                0.01
                            load-average-5:
                                0.04
                            up-time:
                                25 days,  9:45
----------
          ID: get-interface-information
    Function: junos.rpc
      Result: True
     Comment:
     Started: 23:49:00.813806
    Duration: 900.33 ms
     Changes:
              ----------
              out:
                  True
              rpc_reply:

                  Interface               Admin Link Proto    Local                 Remote
                  fxp0.0                  up    up   inet     198.51.100.2/24
```

```
Summary for router1
------------
Succeeded: 2 (changed=2)
Failed:    0
------------
Total states run:     2
Total run time:   1.080 s
```

```
saltuser@minion:~$ ls /var/log/salt/output
router1_fxp0_status.txt              router1_junos_rpc_system_uptime.xml
```

If the state is applied to multiple targets, Salt generates different output files for each target on the proxy minion server in this case, because the defined filename references a unique identifier.

```
saltuser@salt-master:~$ sudo salt '*' state.apply junos_rpc
```

```
saltuser@minion:~$ ls /var/log/salt/output
router1_fxp0_status.txt              router1_junos_rpc_system_uptime.xml
router2_fxp0_status.txt              router2_junos_rpc_system_uptime.xml
```

State files cannot use the same state identifier multiple times in a file. Therefore, if you want to use the same RPC request tag multiple times in a state file, you must define the RPC in the function's argument list. For example:

```
saltuser@salt-master:~$ cat /srv/salt/junos_rpc.sls
Get fxp0 interface information:
  junos.rpc:
    - name: get-interface-information
    - interface-name: fxp0.0
    - terse: True
    - dest: /var/log/salt/output/{{ grains['id'] }}_fxp0_status.txt
    - format: text

Get ge interface information:
  junos.rpc:
    - name: get-interface-information
    - interface-name: ge-1*
    - terse: True
    - dest: /var/log/salt/output/{{ grains['id'] }}_ge_interfaces_status.txt
    - format: text
```

How to Specify RPC Arguments

The `junos.rpc` function supports specifying keyword arguments and values for an RPC. If an argument takes a value, include the argument name and specify the value. If an argument does not require a value, set its value equal to `True`.

For example, the following RPC includes two arguments, one of which takes a value:

```
saltuser@router1> show interfaces ge-1/1/1 terse | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.4R1/junos">
    <rpc>
        <get-interface-information>
                <terse/>
                <interface-name>ge-1/1/1</interface-name>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

To execute the equivalent RPC on the Salt master command line, include the `interface-name='ge-1/1/1'` and `terse=True` arguments.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-interface-information interface-
name='ge-1/1/1' terse=True
```

Similarly, in a Salt state file:

```
saltuser@salt-master:~$ cat /srv/salt/junos_interface_information.sls
get-interface-information:
  junos.rpc:
    - interface-name: ge-1/1/1
    - terse: True
```

NOTE: You can specify Junos OS arguments by using the hyphenated element name or by using underscores in place of any hyphens.

When you execute commands on the Salt master command line, Salt passes the CLI input through `PyYAML` to ensure it is loaded as a proper Python data type. In some cases, arguments that take an integer or that take a string value that parses to an integer might not be parsed correctly. For those arguments, you can provide the value using one of the following methods:

- Use a backslash (\ ) to escape quotation marks around string values.
- Use double quotation marks to enclose single quotation marks and vice versa.
- Include the `--no-parse=`*`param_name`* option.

For example:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-route-engine-information slot=\'0\'
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-route-engine-information slot="'0'"
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-route-engine-information slot='0' --no-
parse=slot
```

How to Save the RPC Output to a File

When you execute the `junos.rpc` function, you can save the returned data in a file on the proxy minion server by including the `dest` argument and specifying the path of the destination file. If you do not specify an absolute path, the path is relative to the top-level root (/) directory. If an output file already exists with the target name, the new file overwrites the old file.

The `dest` argument does not save the entire Salt command response in the file. It only saves the RPC reply contained within the `rpc_reply` key.

The following command displays the output from the `get-system-uptime-information` RPC and saves the `rpc_reply` value to the specified path on the proxy minion server:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-system-uptime-information dest='/home/
saltuser/rpc_output.txt'
```

```
saltuser@minion:~$ ls
rpc_output.txt
```

How to Specify the Format for the RPC Output

By default, the `junos.rpc` function returns the RPC output in XML format for both the data displayed in standard output as well as the data saved to the destination file, if specified. To specify a different output format, include the `format` argument, and set the value equal to the desired format. To request text format or Junos XML elements, use `text` or `xml` respectively. To save the `junos.rpc` output to the destination file in JSON format, specify `json`.

The following command executes the `get-system-uptime-information` RPC and returns the data in text format:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.rpc get-system-uptime-information format=text
dest=/home/saltuser/router1-get-system-uptime-information.txt
router1:
    ----------
    out:
        True
    rpc_reply:

        Current time: 2019-07-29 17:15:03 PDT
        Time Source:  NTP CLOCK
        System booted: 2019-07-01 07:03:56 PDT (4w0d 10:11 ago)
        Protocols started: 2019-07-01 07:07:27 PDT (4w0d 10:07 ago)
        Last configured: 2019-07-24 10:17:34 PDT (5d 06:57 ago) by saltuser
         5:15PM  up 28 days, 10:11, 1 user, load averages: 0.91, 0.30, 0.15
```

The output is also written in the requested format to the destination file on the proxy minion server.

```
saltuser@minion:~$ cat /home/saltuser/router1-get-system-uptime-information.txt
Current time: 2019-07-29 17:15:03 PDT
Time Source:  NTP CLOCK
System booted: 2019-07-01 07:03:56 PDT (4w0d 10:11 ago)
Protocols started: 2019-07-01 07:07:27 PDT (4w0d 10:07 ago)
Last configured: 2019-07-24 10:17:34 PDT (5d 06:57 ago) by saltuser
 5:15PM  up 28 days, 10:11, 1 user, load averages: 0.91, 0.30, 0.15
```

RELATED DOCUMENTATION

Use Salt to Execute Operational Commands on Devices Running Junos OS | 30

Use Salt to Copy Files to Devices Running Junos OS

IN THIS SECTION

Salt can manage a device running Junos OS through a proxy minion that establishes a NETCONF session over SSH with the device. When you use a proxy minion, you can't use standard Salt functions to perform file copy operations on the device, because the device does not directly run the salt-minion process. The `salt.modules.junos.file_copy` execution function and the `salt.states.junos.file_copy` state function enable you to copy files from the Salt master or proxy minion server to the managed device running Junos OS.

junos.file_copy Function Syntax

The `salt.modules.junos.file_copy` execution function syntax is:

```
salt 'target' junos.file_copy src dest
```

The `salt.states.junos.file_copy` state function syntax is:

```
src:
  junos.file_copy:
    - dest: dest
```

where:

- *src*—Source file's path. To specify a path on the Salt master, use **salt://** notation. To specify a path on the proxy minion server, use an absolute path.
- *dest*—Destination file's absolute or relative path on the device running Junos OS. If you do not specify an absolute path on the target device, the path is relative to the current working directory, which is the user's home directory.

How to Copy Files with the junos.file_copy Execution Function

The `junos.file_copy` execution function enables you to quickly copy a file from the Salt master or proxy minion server to a device running Junos OS.

For example, the following command copies the **bgp.slax** file from the Salt master to the target device running Junos OS, router1, and renames the file to **bgp-neighbors.slax**:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.file_copy salt://scripts/op/bgp.slax bgp-
neighbors.slax
router1:
    ----------
    message:
        Successfully copied file from salt://scripts/op/bgp.slax to bgp-neighbors.slax
    out:
        True
```

Because the command does not specify an absolute path for the destination file, the file is copied to the home directory of the user.

```
saltuser@router1> file list ~

/var/home/saltuser:
.ssh/
bgp-neighbors.slax
```

To copy the file to a specific location, specify the absolute path on the target device.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.file_copy salt://scripts/op/bgp.slax /var/db/
scripts/op/bgp-neighbors.slax
router1:
    ----------
    message:
        Successfully copied file from salt://scripts/op/bgp.slax to /var/db/scripts/op/bgp-
neighbors.slax
    out:
        True
```

```
saltuser@router1> file list /var/db/scripts/op/

/var/db/scripts/op:
bgp-neighbors.slax
```

How to Copy Files with the junos.file_copy State Function

You can use the `junos.file_copy` state function within a Salt state file to copy files from the Salt master or proxy minion server to a device running Junos OS. This enables you to store a master set of files in a single central repository and declare a state that defines which files to copy to a given minion. When you enforce or apply the state, Salt copies the necessary files from the repository to the device.

The following state file copies two scripts from **/srv/salt/scripts/op** directory on the Salt master server to the **/var/db/scripts/op** directory on the device running Junos OS:

```
saltuser@salt-master:~$  cat /srv/salt/junos_copy_op_scripts.sls
salt://scripts/op/bgp.slax:
  junos.file_copy:
    - dest: /var/db/scripts/op/bgp.slax

salt://scripts/op/ospf.slax:
  junos.file_copy:
    - dest: /var/db/scripts/op/ospf.slax
```

When you apply the state to the target device, it copies each script to its specified location.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_copy_op_scripts
router1:
----------
          ID: salt://scripts/op/bgp.slax
    Function: junos.file_copy
      Result: True
     Comment:
     Started: 17:37:10.050920
    Duration: 993.378 ms
     Changes:
              ----------
              message:
                  Successfully copied file from salt://scripts/op/bgp.slax to /var/db/scripts/op/
bgp.slax
              out:
                  True
----------
          ID: salt://scripts/op/ospf.slax
    Function: junos.file_copy
      Result: True
     Comment:
     Started: 17:37:11.044443
    Duration: 454.666 ms
     Changes:
              ----------
              message:
                  Successfully copied file from salt://scripts/op/ospf.slax to /var/db/
```

```
scripts/op/ospf.slax
              out:
                  True

Summary for router1
------------
Succeeded: 2 (changed=2)
Failed:    0
------------
Total states run:     2
Total run time:   1.448 s
```

As another example, suppose you have a library of script files in the **/srv/salt/scripts** directory on the Salt master, and you want to ensure specific scripts are copied to the managed device running Junos OS.

```
saltuser@salt-master:/srv/salt/scripts$ ls -R
.:
commit  op

./commit:
bgp-config.slax

./op:
bgp.slax  ospf.slax
```

On the Salt master, you can specify the script information as pillar data and make that accessible to the appropriate minions in the pillar top file, for example:

```
saltuser@salt-master:~$ cat /srv/pillar/junos_scripts.sls
junos_scripts:
  script_path: salt://scripts
  type:
    commit:
      - bgp-config.slax
    op:
      - bgp.slax
      - ospf.slax
```

```
saltuser@salt-master:~$ cat /srv/pillar/top.sls
base:
  'router1':
    - router1-proxy
  'router2':
    - router2-proxy
  'os_family:junos':
    - match: grain
    - junos_scripts
```

You can then create a Salt state file that uses a Jinja template to iterate over the script type and script name and generate the appropriate instructions to copy the scripts to the target devices.

```
saltuser@salt-master:~$  cat /srv/salt/junos_copy_scripts.sls
{% for script_type, scripts in pillar['junos_scripts']['type'].items() %}
{% for script in scripts %}
{{ pillar['junos_scripts']['script_path'] }}/{{ script_type }}/{{ script }}:
  junos.file_copy:
    - dest: /var/db/scripts/{{ script_type }}/{{ script }}
{% endfor %}
{% endfor %}
```

NOTE: Your Jinja syntax might vary depending on the version of Python that Salt uses.

The rendered state file becomes:

```
salt://scripts/commit/bgp-config.slax:
  junos.file_copy:
    - dest: /var/db/scripts/commit/bgp-config.slax

salt://scripts/op/bgp.slax:
  junos.file_copy:
    - dest: /var/db/scripts/op/bgp.slax

salt://scripts/op/ospf.slax:
  junos.file_copy:
    - dest: /var/db/scripts/op/ospf.slax
```

When you apply the state to the target device, it copies each script to the appropriate directory on the device for that script type.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_copy_scripts
router1:
----------
          ID: salt://scripts/commit/bgp-config.slax
    Function: junos.file_copy
      Result: True
     Comment:
     Started: 17:23:19.049243
    Duration: 1110.873 ms
     Changes:
              ----------
              message:
                  Successfully copied file from salt://scripts/commit/bgp-config.slax to /var/db/
scripts/commit/bgp-config.slax
              out:
                  True
----------
          ID: salt://scripts/op/bgp.slax
    Function: junos.file_copy
      Result: True
     Comment:
     Started: 17:23:20.160245
    Duration: 837.084 ms
     Changes:
              ----------
              message:
                  Successfully copied file from salt://scripts/op/bgp.slax to /var/db/scripts/op/
bgp.slax
              out:
                  True
----------
          ID: salt://scripts/op/ospf.slax
    Function: junos.file_copy
      Result: True
     Comment:
     Started: 17:23:20.997516
    Duration: 854.307 ms
     Changes:
              ----------
              message:
                  Successfully copied file from salt://scripts/op/ospf.slax to /var/db/
scripts/op/ospf.slax
              out:
                  True

Summary for router1
------------
Succeeded: 3 (changed=3)
Failed:    0
------------
Total states run:     3
Total run time:   2.802 s
```

Use Salt to Reboot or Shut Down Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.shutdown` execution function and the `salt.states.junos.shutdown` state function enable you to use Salt to reboot or power down a device running Junos OS.

This topic discusses how to use Salt to reboot or power down a device running Junos OS, execute the operation after a delay, or schedule the operation at a specific time.

junos.shutdown Function Overview

You can use the `junos.shutdown` function to request the following operations on devices running Junos OS:

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

By default, the `junos.shutdown` function executes the requested operation on all Routing Engines, if in a dual Routing Engine or Virtual Chassis setup. The `junos.shutdown` function requires one argument, which is the action to perform. You must explicitly include either the `reboot` or the `shutdown` parameter and set it equal to `True`.

The `salt.modules.junos.shutdown` execution function syntax is:

```
salt 'target' junos.shutdown (reboot | shutdown)=True (in_min=minutes | at=\'time\')
```

The `salt.states.junos.shutdown` state function syntax is:

```
id:
  junos.shutdown:
    - (reboot | shutdown): True
    - at: 'time'
    - in_min: minutes
```

where:

- `at=\'time\'`—(Optional) Time at which to schedule the operation.
- *`id`*—User-defined identifier for the state declaration.
- `in_min=`*`minutes`*—(Optional) Number of minutes to delay the reboot or shutdown operation.
- `(reboot | shutdown)=True`—Operation to perform. You must explicitly set either the `reboot` or the `shutdown` parameter to `True` to specify whether to reboot or shut down the device, respectively.

How to Use the junos.shutdown Execution Function

The `salt.modules.junos.shutdown` execution function enables you to reboot or power down one or more devices running Junos OS from the Salt master command line.

For example, the following command immediately reboots all Routing Engines on the target device:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.shutdown reboot=True
router1:
    ----------
    message:
        Successfully powered off/rebooted.
    out:
        True
```

Similarly, the following command powers down the device:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.shutdown shutdown=True
router1:
    ----------
    message:
        Successfully powered off/rebooted.
    out:
        True
```

The following command immediately reboots all devices that match on the `os_family:junos` grain, which is useful when you want to apply the operation to all managed devices running Junos OS:

```
saltuser@salt-master:~$  sudo salt -G 'os_family:junos' junos.shutdown reboot=True
router1:
    ----------
    message:
        Successfully powered off/rebooted.
    out:
        True
router2:
    ----------
    message:
        Successfully powered off/rebooted.
    out:
        True
```

How to Use the junos.shutdown State Function

You can use the `salt.states.junos.shutdown` state function within a Salt state file to reboot or power down one or more devices running Junos OS.

The following state file immediately reboots all Routing Engines on the target devices to which the state is applied:

```
saltuser@salt-master:~$  cat /srv/salt/junos_reboot.sls
Reboot all REs now:
  junos.shutdown:
    - reboot: True
```

When you apply the state to the target, the device immediately performs the operation.

```
saltuser@salt-master:~$  sudo salt 'router1' state.apply junos_reboot.sls
router1:
----------
          ID: Reboot all REs now
    Function: junos.shutdown
      Result: True
     Comment:
     Started: 21:54:51.158521
    Duration: 1141.496 ms
     Changes:
              ----------
              message:
                  Successfully powered off/rebooted.
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
```

```
Total states run:     1
Total run time:   1.141 s
```

Similarly, the following state file powers off the target devices:

```
saltuser@salt-master:~$  cat /srv/salt/junos_shutdown.sls
Shut down device running Junos OS:
  junos.shutdown:
    - shutdown: True
```

How to Perform a Reboot or Shutdown Operation with a Delay

The default behavior of the `junos.shutdown` function is to immediately execute the reboot or shutdown operation. You can also request a delay before the operation executes.

To delay the reboot or shutdown operation by a specified number of minutes, set the optional `in_min` parameter to the number of minutes that the target device running Junos OS should wait before executing the operation. The following command requests a reboot of all Routing Engines in 2 minutes:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.shutdown reboot=True in_min=2
router1:
    ----------
    message:
        Successfully powered off/rebooted.
    out:
        True
```

The target device broadcasts messages about the impending reboot to any users logged into the system. After the specified amount of time has passed, the system reboots.

```
user@router1>
*** System shutdown message from saltuser@router1 ***

System going down in 2 minutes
```

How to Perform a Reboot or Shutdown Operation at a Specified Time

The default behavior of the `junos.shutdown` function is to immediately execute the reboot or shutdown operation. You can also schedule the operation at a particular date and time. To schedule the operation at a specific time, include the `at` parameter, which takes a string that can be specified in one of the following ways:

- `now`—Stop or reboot the software immediately.
- `+`*`minutes`*—Number of minutes from now to perform the operation.
- *`yymmddhhmm`*—Absolute time at which to perform the operation, specified as year, month, day, hour, and minute.
- *`hh:mm`*—Absolute time on the current day at which to perform the operation, specified in 24-hour time.

NOTE: The proxy minion server requires Junos PyEZ Release 2.3.0 or later to use the `at` parameter when you specify `shutdown=True`.

For example, the following Salt state file schedules a system reboot of all Routing Engines at 22:30 on the current day:

```
reboot:
  junos.shutdown:
    - reboot: True
    - at: '22:30'
```

When you apply the state to the target, the device schedules the reboot for the specified time.

```
saltuser@salt-master:~$  sudo salt 'router1' state.apply junos_reboot.sls
router1:
----------
          ID: reboot
    Function: junos.shutdown
      Result: True
     Comment:
     Started: 17:23:27.368341
    Duration: 173.988 ms
     Changes:
              ----------
              message:
                 Successfully powered off/rebooted.
             out:
                 True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 173.988 ms
```

You can view the pending reboot on the device with the `show system reboot` command.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.cli 'show system reboot'
router1:
    ----------
    message:

        reboot requested by saltuser at Wed Jul 31 22:30:00 2019
        [process id 2313]
    out:
        True
```

When you execute the same operation on the Salt master command line, Salt passes the CLI input through `PyYAML` to ensure it is loaded as a proper Python data type. In this case, the `at` value might be parsed incorrectly. To use the `junos.shutdown` execution function with the `at` argument, you can provide the value using one of the following methods:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at=\'22:30\'
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at="'22:30'"
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True "at='22:30'"
saltuser@salt-master:~$ sudo salt 'router1' junos.shutdown reboot=True at="22:30" --no-parse=at
```

RELATED DOCUMENTATION

Use Salt to Install Software on Devices Running Junos OS | 62

Use Salt to Install Software on Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.install_os` execution function and the `salt.states.junos.install_os` state function enable you to install software on devices running Junos OS.

The following sections discuss how to use Salt to install a Junos OS image, including the supported deployment scenarios, the general installation process and options, and specialized upgrade scenarios such as a VM Host upgrade, a unified in-service software upgrade (unified ISSU), and a nonstop software upgrade (NSSU) on devices that support these features.

junos.install_os Function Overview

The `junos.install_os` function enables you to use Salt to update the software on a device running Junos OS in the following scenarios:

- Standalone device with a single Routing Engine
- Standalone device equipped with dual Routing Engines
- EX Series Virtual Chassis in non-mixed-mode configurations
- QFX Series Virtual Chassis in non-mixed-mode configurations

- VM Host upgrade on Routing Engines with VM Host support
- Deployment configuration that has some form of *in-service* feature enabled, such as unified ISSU or NSSU

The basic syntax for the `junos.install_os` execution and state functions is shown here. For a complete list of the available function parameters, see the API documentation for that function.

The `salt.modules.junos.install_os` execution function syntax is:

```
salt 'target' junos.install_os 'path' no_copy=(False | True) remote_path='remote-path'
reboot=(False | True)
```

The `salt.states.junos.install_os` state function syntax is:

```
path:
  junos.install_os:
    - no_copy=(False | True)
    - remote_path: 'remote-path'
    - reboot: (False | True)
```

where:

- `no_copy=(False | True)`—(Optional) Specify whether to copy the software image to the device running Junos OS. The default is False.
- *path*—Path to the software image to install.
- `reboot=(False | True)`—(Optional) Specify whether to reboot the system after installing the software. The default is False.
- `remote_path='`*remote-path*`'`—(Optional) Directory on the target device from which the image is installed. The default is **/var/tmp**.

Table 4 on page 64 outlines the parameter settings required for the different software package locations.

Table 4: junos.install_os Parameter Settings for Different Software Package Locations

Software Package Location	`no_copy` Parameter	`path` Parameter	`remote_path` Parameter
Salt master	Omitted or set to `False`	Path to the software package on the Salt master. The path must use the `salt://` notation.	(Optional) Directory on the target device to which the package is copied. Default is **/var/tmp**.
Proxy minion server	Omitted or set to `False`	Absolute path to the software package on the proxy minion server.	(Optional) Directory on the target device to which the package is copied. Default is **/var/tmp**.
Target device	Set to `True`	Software package filename.	(Optional) Directory on the target device where the installation package must already reside. Default is **/var/tmp**.
URL	–	URL from the perspective of the target device running Junos OS from which the software package is installed.	–

By default, the `junos.install_os` function copies the software image from the path on the Salt master or proxy minion server, which is specified in the `path` parameter, to the **/var/tmp** directory on the device running Junos OS (individual device or primary device in a non-mixed Virtual Chassis). To copy the image to a different directory, include the `remote_path` argument and specify the directory. To ensure that the device has enough storage space to accommodate the image, the `junos.install_os` function automatically performs a storage cleanup on the target device before copying the image, unless you set the `cleanfs` parameter to `False`.

If the software image already resides on the target device, set the `no_copy` argument to `True`, and set the `path` argument to the image filename. The image must reside either in the directory specified by the `remote_path` argument, or if `remote_path` is omitted, in the default **/var/tmp** directory.

To install the software image from a remote URL, set the `path` argument to a URL from the perspective of the target device running Junos OS. The image is copied over and installed from the specified URL, and the `no-copy` and `remote_path` arguments are ignored. For information about specifying the format of the URL, see Format for Specifying Filenames and URLs in Junos OS CLI Commands.

For a complete list of the execution and state function parameters, see:

- salt.modules.junos.install_os
- salt.states.junos.install_os

The junos.install_os execution and state functions use the Junos PyEZ install() function to perform the installation. For more information about the operations executed as part of the installation, see the *Installation Process Overview* section in Use Junos PyEZ to Install Software on Junos Devices.

How to Use the junos.install_os Execution Function

The salt.modules.junos.install_os execution function enables you to install a software image on one or more devices running Junos OS. For example, the following command copies the specified image from the Salt master to the **/var/tmp** directory on the target device. The device installs the package and then reboots.

```
saltuser@salt-master:~$ sudo salt -t 600 'router1' junos.install_os 'salt://images/jinstall-
ppc-19.2R1.8-signed.tgz' reboot=True
router1:
    ----------
    message:
        Successfully installed and rebooted!
    out:
        True
```

Because the installation can take a long time, the command includes the -t *seconds* option to increase the time that Salt waits for a reply from the minion. Otherwise, the command line client might time out before the minion returns a response.

How to Use the junos.install_os State Function

The salt.modules.junos.install_os state function enables you to specify the image that a managed device should run. You can apply the state to one or more targets as required or reference it within the state top file for the appropriate minions.

The following state file ensures that the target minion runs the specified image:

```
saltuser@salt-master:~$ cat /srv/salt/junos_install_os_ppc_19_2R1.sls
salt://images/jinstall-ppc-19.2R1.8-signed.tgz:
  junos.install_os:
    - reboot: True
```

When the state is applied, Salt copies the local image, which in this case resides in the **/srv/salt/images** directory on the Salt master, to the **/var/tmp** directory on the target device. The device installs the image and then reboots.

```
saltuser@salt-master:~$  sudo salt -t 600 'router1' state.apply junos_install_os_ppc_19_2R1
router1:
----------
          ID: salt://images/jinstall-ppc-19.2R1.8-signed.tgz
    Function: junos.install_os
      Result: True
     Comment:
     Started: 21:51:58.488113
    Duration: 441546.724 ms
     Changes:
              ----------
              message:
                  Successfully installed and rebooted!
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 441.547 s
```

Because the installation can take a long time, the command includes the `-t` *seconds* option to increase the time that Salt waits for a reply from the minion. Otherwise, the command line client might time out before the minion returns a response.

How to Specify Timeout Values

The `junos.install_os` function performs operations over a NETCONF session. The default time for a NETCONF RPC to time out is 30 seconds. During the installation process, the function increases the RPC timeout interval to 1800 seconds (30 minutes) when copying and installing the package on the device and to 300 seconds (5 minutes) when computing the checksum. In some cases, the installation process or checksum calculation might exceed these time intervals.

To increase the timeout value for the installation process and the checksum calculation in the call to the `junos.install_os` function, include the `dev_timeout` and `checksum_timeout` parameters, respectively, and set them to appropriate values. For example:

```
salt://images/jinstall-ppc-19.2R1.8-signed.tgz
  junos.install_os:
    - checksum_timeout: 400
    - dev_timeout: 2400
    - reboot: True
```

When the `no_copy` parameter is omitted or set to `False`, the `junos.install_os` function automatically performs a storage cleanup on the target device before copying the software image. Doing this helps to ensure that there is sufficient space for the image. By default, the cleanup operation times out after 300 seconds. To change the default timeout value for this operation, set the `cleanfs_timeout` parameter to the number of seconds to wait for the operation to complete.

```
salt://images/jinstall-ppc-19.2R1.8-signed.tgz
  junos.install_os:
    - cleanfs_timeout: 500
    - reboot: True
```

The default time that the Salt command line client waits for replies from a minion is 5 seconds. However, the installation process can take a considerable amount of time. As a result, the command line client might time out before the minion returns a response. For example:

```
router1:
    Minion did not return. [No response]
    The minions may not have all finished running and any remaining minions will return upon
completion. To look up the return data for this job later, run the following command:

    salt-run jobs.lookup_jid 20200520174528242061
```

If that happens, you can check the result of the installation later. However, if you want to display the results when you execute the command, you can increase the time that Salt waits for replies from the minion by including the `-t` *seconds* option in the command and increasing the timeout value as necessary.

```
saltuser@salt-master:~$  sudo salt -t 600 'router1' junos.install_os 'salt://images/jinstall-
ppc-19.2R1.8-signed.tgz' reboot=True
router1:
    ----------
    message:
        Successfully installed and rebooted!
    out:
        True
```

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 `junos.install_os` function supports the `vmhost=True` argument for performing a VM Host upgrade. When this argument is present, the function performs the installation using the `<request-vmhost-package-add>` RPC.

The following state declaration upgrades both the Junos OS and host OS on the device:

```
saltuser@salt-master:~$  cat /srv/salt/junos_install_os_vm_host.sls
salt://images/junos-vmhost-install-qfx-x86-64-18.1R1.9.tgz:
  junos.install_os:
    - remote_path: /var/tmp
    - vmhost: True
    - reboot: True
```

How to Perform a Unified ISSU or an NSSU

Salt for Junos OS provides support for 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 unified ISSU and NSSU, 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 on devices that support this feature, use the `junos.install_os` function, and set `issu` to `True`.

In the following state declaration, the `junos.install_os` function upgrades Junos OS on both Routing Engines and reboots the new primary Routing Engine (previously the old backup Routing Engine) as part of the installation process. If the installation is successful, the `reboot: True` and `all_re: False` arguments then instruct the device to reboot the connected Routing Engine, which is the new backup Routing Engine (previously the old primary Routing Engine).

```
saltuser@salt-master:~$  cat /srv/salt/junos_install_os_issu.sls
salt://images/junos-install-mx-x86-64-17.2R1.13.tgz:
  junos.install_os:
    - remote_path: /var/tmp
    - issu: True
    - reboot: True
    - all_re: False
```

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 on devices that support this feature, use the `junos.install_os` function, and set `nssu` to `True`. For example:

```
saltuser@salt-master:~$  cat /srv/salt/junos_install_os_nssu.sls
salt://images/jinstall-ex-4300-14.1X53-D44.3-domestic-signed.tgz:
  junos.install_os:
    - remote_path: /var/tmp
    - nssu: True
    - reboot: True
    - all_re: False
```

RELATED DOCUMENTATION

Use Salt to Reboot or Shut Down Devices Running Junos OS | 55

Use Salt to Restore a Device Running Junos OS to the Factory-Default Configuration Settings | 70

Use Salt to Restore a Device Running Junos OS to the Factory-Default Configuration Settings

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.zeroize` execution function and the `salt.states.junos.zeroize` state function enable you to restore a device running Junos OS to its factory-default configuration settings. 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.

junos.zeroize Function Overview

The `junos.zeroize` function executes the `request system zeroize` operational command on the target hosts. The function resets both Routing Engines in a dual Routing Engine system.

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.

The salt.modules.junos.zeroize execution function syntax is:

```
salt 'target' junos.zeroize
```

The salt.states.junos.zeroize state function syntax is:

```
id:
  junos.zeroize
```

where *id* is the user-defined identifier for the state declaration.

For more information about the `request system zeroize` command, see request system zeroize.

How to Use the junos.zeroize Execution and State Functions

The `junos.zeroize` function enables you to easily reset one or more devices running Junos OS to its factory-default configuration settings from the Salt master command line.

For example, the following command resets the target device to its factory default settings:

```
saltuser@salt-master:~$  sudo salt -t 300 'router1' junos.zeroize
```

You can perform the same operation in a Salt state file. For example:

```
saltuser@salt-master:~$  cat /srv/salt/junos_zeroize.sls
Reset device running Junos OS:
  junos.zeroize
```

```
saltuser@salt-master:~$  sudo salt -t 300 'router1' state.apply junos_zeroize
```

When you execute the `junos.zeroize` function, it resets the device configuration, including the hostname and IP address and any configured connection and authentication information. As a result, the proxy minion is unable to maintain the connection to the device.

```
saltuser@salt-master:~$  sudo salt 'router1' test.ping
router1:
    False
```

After the device is restored to its factory-default configuration settings, you must log in through the console as root in order to access the device. The device displays `Amnesiac` in place of a hostname, because the updated configuration no longer specifies a hostname.

```
Amnesiac (ttyu0)

login: root

--- JUNOS 18.4R1.8 built 2018-12-17 03:28:10 UTC
% cli
root>
```

To restore the connection between the Salt proxy minion and the managed device, you must configure the device with the appropriate settings. For example, you must configure the NETCONF-over-SSH service. You must also configure the device with the same connection and authentication settings that are defined in the proxy configuration for that device in the pillar data on the Salt master. The proxy minion will automatically reconnect to the device after the appropriate settings are configured and committed.

```
saltuser@salt-master:~$  sudo salt 'router1' test.ping
router1:
    True
```

RELATED DOCUMENTATION

Use Salt to Reboot or Shut Down Devices Running Junos OS | 55

Use Salt to Install Software on Devices Running Junos OS | 62

Use Salt with Junos PyEZ Tables to Retrieve Operational and Configuration Data from Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.get_table` execution function and the `salt.states.junos.get_table` state function can reference predefined or custom Junos PyEZ Tables to extract operational information or configuration data from a device running Junos OS.

Understanding Junos PyEZ Tables

Junos PyEZ is a microframework for Python that enables you to manage and automate devices running Junos OS. The Junos proxy minion (for Salt) connects to the Salt master using the ZeroMQ event bus and uses the Junos PyEZ library to connect to the device running Junos OS. As a result, Salt can leverage Junos PyEZ features when managing the devices.

Junos PyEZ supports using simple YAML definitions, which are referred to as *Tables* and *Views*, to retrieve and filter operational command output and configuration data from devices running Junos OS. Junos PyEZ operational (op) Tables extract information from the output of operational commands or RPCs, and configuration Tables retrieve specific configuration data. 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 Salt to manage devices running Junos OS, the `junos.get_table` function can use Junos PyEZ Tables to retrieve data from a device. The function can reference the predefined operational Tables and

Views that are included with the Junos PyEZ distribution, or it can reference user-defined operational and configuration Tables and Views that reside on either the Salt master or a separate proxy minion server.

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

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

junos.get_table Function Overview

The `junos.get_table` function can use Junos PyEZ Tables to retrieve data from a device running Junos OS. The basic syntax for the `junos.get_table` execution function and state function is presented here. For a complete list of the available function parameters, see the API documentation for the function.

The `salt.modules.junos.get_table` execution function syntax is:

```
salt 'target' junos.get_table table table_file path
```

The `salt.states.junos.get_table` state function syntax is:

```
id:
  junos.get_table:
    - table: table
    - table_file: table_file
    - path: path
```

where:

- *id*—User-defined identifier for the state declaration.
- `path:` *path*—(Optional) Path to the YAML file that contains the Table and View definitions.
- `table:` *table*—Name of the Junos PyEZ Table.
- `table_file:` *table_file*—Filename of the YAML file that contains the Table and View definitions.

You use the `path` and `table_file` parameters to specify the path and filename of the YAML file containing the Table definition. If you import a Table from the `jnpr.junos.op` module included with the Junos PyEZ distribution, you can omit the `path` parameter. In this case, the `get_table` function automatically checks the appropriate directory in the Junos PyEZ install path for the specified Table file. You can also specify a path to a custom Table. If you use a custom Table, the Table can reside on the Salt master or proxy minion server. Table 5 on page 75 outlines the different Tables you can use, the location of the Tables, and the required value for the `path` parameter.

Table 5: get_table Table Types

Table	Location	`path` Value	Example
jnpr.junos.op module Tables	**jnpr/junos/op** directory under the Junos PyEZ install path on the proxy minion server	–	`junos.get_table:` `- table: ArpTable` `- table_file: arp.yml`
Custom Tables	Salt master	Path (using **salt://** notation) to the YAML file containing the custom Table.	`junos.get_table:` `- table: IFTable` `- table_file: IFTable.yml` `- path: salt://tables`
	Proxy minion server	Absolute path to the YAML file containing the custom Table.	`junos.get_table:` `- table: IFTable` `- table_file: IFTable.yml` `- path: /srv/salt/tables`

How to Use the junos.get_table Function

You can call the `junos.get_table` function to use a Junos PyEZ Table to retrieve operational or configuration data from a device running Junos OS. The following command uses the `ArpTable` Table in the `jnpr.junos.op` module to retrieve Address Resolution Protocol (ARP) entries for the target device. The Table is defined in the **arp.yml** file located in the Junos PyEZ **jnpr/junos/op** directory.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.get_table ArpTable arp.yml
```

The function's response includes the device's hostname, the Table and View, and the `out` key, which returns `True` if the operation is successful. If the operation is successful, the response also includes the `reply` key, which contains the Table items. Each Table item includes the fields defined by the View and the value extracted from the data for each of the corresponding fields.

```
router1:
    ----------
    hostname:
        router1.example.com
    out:
        True
    reply:
        ----------
        00:50:56:93:7c:13:
            ----------
            interface_name:
                fxp0.0
            ip_address:
                198.51.100.10
            mac_address:
                00:50:56:93:7c:13
        f8:c0:01:18:8b:67:
            ----------
            interface_name:
                fxp0.0
            ip_address:
                198.51.100.254
            mac_address:
                f8:c0:01:18:8b:67
        ...
    table:
        ----------
        ArpTable:
            ----------
            args:
                ----------
                no-resolve:
                    True
            item:
                arp-table-entry
            key:
                mac-address
            rpc:
                get-arp-table-information
```

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

As another example, the following state file retrieves interface status information using a custom Junos PyEZ Table named IFTable. The Table and View are defined in the **/srv/salt/tables/IFTable.yml** file on the Salt master.

```
saltuser@salt-master:~$ cat /srv/salt/junos_get_table_interface_status.sls
Get Interface Status Information:
  junos.get_table:
    - table: IFTable
    - table_file: IFTable.yml
    - path: salt://tables
```

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_get_table_interface_status
router1:
----------
          ID: Get Interface Status Information
    Function: junos.get_table
      Result: True
     Comment:
     Started: 22:51:38.947322
    Duration: 373.053 ms
     Changes:
              ----------
              hostname:
                  router1.example.com
              out:
                  True
```

```
                reply:
                    ----------
                    cbp0:
                        ----------
                        adstat:
                            up
                        name:
                            cbp0
                        opstat:
                            up
...
```

How to Define Optional Parameters in Tables and Views

Junos PyEZ Tables and Views can include a number of optional parameters depending on the type of Table. The Table and View definitions can define default values for these parameters. You can also specify or override the value for one or more parameters when you use the Table in an application.

The `junos.get_table` function enables you to define a value or override the default value for many of these same parameters. For example, when you call the `junos.get_table` function, you can include the `key` argument for Tables that support the corresponding parameter. You can also include the `table_args` argument to supply key/value pairs for the `args` parameter in op Tables. For information about the parameters supported by the different types of Tables and Views, see the Junos PyEZ Developer Guide.

Consider the `EthPortTable` Table in the `jnpr.junos.op` module, which executes the `get-interface-information` RPC. The RPC uses the default arguments `media: True` and `interface_name: '[afgxe][et]-*'`.

```
---
EthPortTable:
  rpc: get-interface-information
  args:
    media: True
    interface_name: '[afgxe][et]-*'
  args_key: interface_name
  item: physical-interface
  view: EthPortView
```

In this case, when you call the `junos.get_table` function, you can include optional parameters that are supported by op Tables that execute RPCs and return structured output. For example, you can include the `table_args` parameter to override the default values for the RPC arguments that are defined in `args`.

The following command uses `EthPortTable` to retrieve interface information from the device, and it includes the `table_args` parameter to only return data for 10-Gigabit Ethernet (GbE) interfaces instead of the default set of interfaces defined in the Table. The Table definition displayed in the output reflects the updated arguments under `args`.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.get_table EthPortTable ethport.yml
table_args='{"interface_name" : "xe-*"}'
router1:
    ----------
...
    table:
        ----------
        EthPortTable:
            ----------
            args:
                ----------
                interface_name:
                    xe-*
                media:
                    True
            args_key:
                interface_name
            item:
                physical-interface
            key:
                name
            rpc:
                get-interface-information
            view:
                EthPortView
...
```

The equivalent state file is:

```
saltuser@salt-master:~$ cat /srv/salt/junos_get_table_ethport.sls
Retrieve Ethernet port information:
  junos.get_table:
    - table: EthPortTable
    - table_file: ethport.yml
    - table_args:
        interface_name: 'xe-*'
```

4

CHAPTER

Use Salt to Manage the Configuration

IN THIS CHAPTER

Use Salt to Retrieve a Junos OS Configuration

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to retrieve the configuration from managed devices. You can use the `salt.modules.junos.rpc` execution function or the `salt.states.junos.rpc` state function to execute the `get_config` RPC to retrieve configuration data from a device running Junos OS.

You can use the `get_config` RPC to retrieve the complete configuration or a subset of the configuration from the candidate, committed, or ephemeral configuration databases. You can return data for the pre-inheritance or post-inheritance configuration. In addition, you can return the data in several different formats and save the data to a file on the proxy minion server.

How to Retrieve the Complete Candidate Configuration

To retrieve the complete candidate configuration from a device running Junos OS, call the `junos.rpc` function, and execute the `get_config` RPC.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config
router1:
    ----------
    out:
        True
    rpc_reply:
        ----------
        configuration:
            ----------
            apply-groups:
                - global
                - re0
                - re1
    ...
```

Similarly, you can define a Salt state file that retrieves the configuration. The following sample state file retrieves the complete candidate configuration for the target devices in text format:

```
saltuser@salt-master:~$  cat /srv/salt/junos_get_config.sls
get_config:
  junos.rpc:
    - format: text
```

When you apply the state, it displays the candidate configuration for each target device.

```
saltuser@salt-master:~$  sudo salt 'router1' state.apply junos_get_config
router1:
----------
          ID: get_config
    Function: junos.rpc
      Result: True
     Comment:
     Started: 20:15:20.409910
    Duration: 973.546 ms
     Changes:
              ----------
              out:
                  True
              rpc_reply:

                  ## Last changed: 2019-08-06 11:38:21 PDT
                  version 18.4R1.8;
                  groups {
                      re0 {
                          system {
                              host-name router1;
                          }
                      ...
```

```
Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 973.546 ms
```

How to Specify the Source Database for the Configuration Data

By default, the `get_config` RPC retrieves configuration data from the candidate configuration database. You can also retrieve data from the committed configuration database or the ephemeral configuration database by including the `database` parameter with the appropriate value.

Committed Configuration Database

To retrieve data from the committed configuration database, set the `database` argument equal to `committed`. For example:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config database=committed
router1:
    ----------
    out:
        True
    rpc_reply:
        ----------
        configuration:
            ----------
            apply-groups:
                - global
                - re0
                - re1
    ...
```

Ephemeral Configuration Database

You can also use Salt to retrieve data from the ephemeral configuration database on devices that support this database. When you retrieve configuration data from the shared configuration database, by default, the results do not include data from the ephemeral configuration database.

To retrieve data from the default instance of the ephemeral configuration database, include the `database` argument and set it equal to `ephemeral`.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config database=ephemeral
```

To retrieve data from a user-defined instance of the ephemeral configuration database, set the `database` argument equal to `ephemeral`, and set the `ephemeral-instance` argument to the name of the ephemeral instance.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config database=ephemeral ephemeral-
instance=eph1
router1:
    -
    out:
        True
    rpc_reply:
        -
        configuration:
            -
            protocols:
                -
                mpls:
                    -
                    label-switched-path:
                        -
                        name:
                            to-hastings
                        to:
                            192.0.2.1
```

How to Specify the Scope of the Configuration Data to Return

In addition to retrieving the complete Junos OS configuration, the `get_config` RPC can retrieve a subset of the configuration by using the `filter` parameter. The `filter` parameter takes 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.

To request multiple hierarchies, the `filter` string must include the `<configuration>` root element. Otherwise, the value of `filter` must represent all levels of the configuration hierarchy starting just under the root `<configuration>` element down to the hierarchy to display. To select a subtree, include the empty tag for that hierarchy level. To return a specific object, include a content match node that defines the element and value to match.

The following command retrieves and prints the configuration at the `[edit interfaces]` and `[edit protocols]` hierarchy levels in the candidate configuration:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config
filter='<configuration><interfaces/><protocols/></configuration>'
```

The following command retrieves and prints the configuration at the `[edit system services]` hierarchy level:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config filter='<system><services/></
system>'
```

The following state file retrieves the `<name>` element for each `<interface>` element at the `<interfaces>` hierarchy level in the post-inheritance candidate configuration:

```
saltuser@salt-master:~$  cat /srv/salt/junos_get_config_interface_names.sls
get_config:
  junos.rpc:
    - filter: <interfaces><interface><name/></interface></interfaces>
    - inherit: inherit
```

Applying the state returns the list of interface names.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_get_config_interface_names
router1:
----------
          ID: get_config
    Function: junos.rpc
      Result: True
     Comment:
     Started: 22:11:24.430639
    Duration: 1522.123 ms
     Changes:
              ----------
              out:
                  True
```

```
                rpc_reply:
                    ----------
                    configuration:
                        ----------
                        interfaces:
                            ----------
                            interface:
                                |_
                                  ----------
                                  name:
                                      ge-1/0/0
                                |_
                                  ----------
                                  name:
                                      ge-1/0/2
                                |_
                                  ----------
                                  name:
                                      ge-1/0/3
                                |_
                                  ----------
                                  name:
                                      ge-1/0/4
                                |_
                                  ----------
                                  name:
                                      lo0
                                |_
                                  ----------
                                  name:
                                      fxp0

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   1.522 s
```

Similarly, the following example retrieves the subtree for the ge-1/0/1 interface:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config
filter='<interfaces><interface><name>ge-1/0/1</name></interface></interfaces>' format=text
router1:
    ----------
    out:
        True
    rpc_reply:

        ## Last changed: 2019-08-06 14:58:44 PDT
        interfaces {
            ge-1/0/1 {
                unit 0 {
                    family inet {
                        address 10.1.1.1;
                    }
                }
            }
        }
```

How to Specify the Format of the Configuration Data to Return

By default, the `junos.rpc` function returns the RPC output in XML format for both the data displayed in standard output as well as the data saved to the destination file, if specified. To specify an output format, include the `format` argument, and set the value equal to the desired format. To request text format or Junos XML elements, use `text` or `xml`, respectively. To save the configuration data to a destination file in JSON format, specify `json`.

The following command saves the configuration data to the specified file and also displays it in standard output in text format:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config dest=/var/configs/
router1_config.txt format=text
```

The following command saves the configuration data to the specified file in JSON format:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config dest=/var/configs/
router1_config.xml format=json
```

How to Specify Inheritance and Groups Options

The `get_config` RPC retrieves data from the pre-inheritance configuration, in which the `<groups>`, `<apply-groups>`, `<apply-groups-except>`, and `<interface-range>` tags are separate elements in the configuration output. To retrieve data from the post-inheritance configuration, which displays statements that are inherited from user-defined groups and ranges as children of the inheriting statements, you can include the `inherit=inherit` argument. If you also include the `groups=groups` argument, the text and XML-formatted output will indicate the group from which a statement was inherited.

For example, the following command retrieves the configuration at the `[edit system services]` hierarchy level from the post-inheritance candidate configuration. In this case, if the configuration also contains statements configured at the `[edit groups global system services]` hierarchy level, those statements are inherited at the `[edit system services]` hierarchy level in the post-inheritance configuration and returned in the configuration data.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config filter='<system><services/></
system>' inherit=inherit groups=groups dest=/var/configs/router1-system-services.xml
```

In the destination file, the elements that are inherited from a group include the `group` attribute. The `group` attribute value indicates the group from which the element was inherited.

```
<configuration changed-seconds="1565131770" changed-localtime="2019-08-06 15:49:30
PDT"><system><services>
            <ftp group="global"></ftp>
            <ssh group="global"></ssh>
            <netconf group="global"><ssh group="global">
                </ssh><traceoptions group="global"><file group="global"><filename
group="global">netconf.log</filename><size group="global">10m</size><files group="global">2</
files></file><flag group="global"><name group="global">all</name></flag></traceoptions></netconf>
</services></system></configuration>
```

How to Save Retrieved Configuration Data to a File

When you execute the `junos.rpc` function, you can save the returned data in a file on the proxy minion server by including the `dest` argument and specifying the path of the destination file. If you do not specify an absolute path, the path is relative to the top-level root (/) directory. If the target output file already exists, the new file overwrites the old file.

To retrieve configuration data from a device running Junos OS and save the output to a file for later reference, execute the `get_config` RPC and include the `dest` parameter. The following command retrieves the complete configuration, displays the data in standard output, and also saves the data in the specified file on the proxy minion server:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rpc get_config dest=/var/configs/
router1_config.xml
router1:
    ----------
    out:
        True
    rpc_reply:
        ----------
        configuration:
            ----------
            apply-groups:
                - global
                - re0
                - re1
    ...
```

On the proxy minion server, the configuration data is saved to the specified file.

```
saltuser@minion:~$ ls /var/configs
router1_config.xml
```

Similarly, you can define a Salt state that retrieves configuration data and saves it in a file. The following sample state file retrieves the complete configuration and writes the data to both standard output and to the specified file on the proxy minion server:

```
saltuser@salt-master:~$  cat /srv/salt/junos_save_config.sls
get_config:
  junos.rpc:
    - dest: /var/configs/{{ grains['id'] }}_config.txt
    - format: text
```

In this case, the proxy identifier stored in the `id` grain is included in the destination filename to distinguish the output files when the state is applied to multiple targets. When you apply the state to devices that match the `os_family:junos` grain, it saves the configuration for each device to its own file on the proxy minion server.

```
saltuser@salt-master:~$  sudo salt -G 'os_family:junos' state.apply junos_save_config
router1:
----------
          ID: get_config
    Function: junos.rpc
      Result: True
     Comment:
     Started: 20:58:25.705709
    Duration: 284.552 ms
     Changes:
              ----------
              out:
                  True
              rpc_reply:

                  ## Last changed: 2019-08-05 17:24:18 PDT
...
```

Each device configuration is saved in a separate file on the proxy minion server.

```
saltuser@minion:~$ ls /var/configs
router1_config.txt  router2_config.txt
```

RELATED DOCUMENTATION

Use Salt to Compare the Junos OS Candidate Configuration with a Previously Committed Configuration | 91

Use Salt to Configure Devices Running Junos OS | 94

Use Salt to Compare the Junos OS Candidate Configuration with a Previously Committed Configuration

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to perform operational and configuration tasks on the managed devices. The `salt.modules.junos.diff` execution function and the `salt.states.junos.diff` state function enable you to compare the candidate configuration to a previously committed configuration and return the difference.

NOTE: The ephemeral configuration database only stores the current version of the committed ephemeral configuration data, and as a result, it does not support comparing the current configuration to previously committed configurations.

Devices running Junos OS store a copy of the most recently committed configuration and up to 49 previous configurations. The `junos.diff` function returns the difference between the candidate configuration and a previously committed configuration, which is referenced by the rollback ID parameter. Table 6 on page 91 outlines the rollback ID parameter to use for the `junos.diff` execution and state functions. If the rollback ID parameter is omitted, the rollback ID defaults to 0, which corresponds to the active or most recently committed configuration.

Table 6: Rollback ID Parameter

Function	Rollback ID Parameter
`salt.modules.junos.diff`	`id`
`salt.states.junos.diff`	`d_id`

The configuration differences are returned in patch format. Statements that exist only in the candidate configuration are prefixed with a plus sign (+), and statements that exist only in the comparison configuration and not in the candidate configuration are prefixed with a minus sign (-). If there is no difference between the configurations, the `junos.diff` function prints `None`.

To compare the candidate configuration to the active configuration, execute the `junos.diff` function, and either omit the rollback ID parameter or set it equal to 0. For example:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.diff
router1:
    ----------
    message:

        [edit system scripts op]
        +     file bgp-neighbors.slax;
    out:
        True
```

The previous command is equivalent to issuing the `show | compare` or `show | compare rollback 0` configuration mode command in the Junos OS command-line interface (CLI).

```
saltuser@router1# show | compare
[edit system scripts op]
+     file bgp-neighbors.slax;
```

If you compare the candidate configuration to a previous configuration and there are no differences, the `junos.diff` function returns `None`. In the following example, the `junos.diff` function compares the candidate configuration to the active configuration and returns `None`, which indicates that the there are no uncommitted changes in the candidate configuration:

```
saltuser@salt-master:~$  sudo salt 'router2' junos.diff
router2:
    ----------
    message:
        None
    out:
        True
```

To compare the candidate configuration to a specific rollback configuration, set the `id` parameter to the ID of the rollback configuration. This is equivalent to issuing the `show | compare rollback` *n* configuration mode command in the Junos OS CLI.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.diff id=3
router1:
    ----------
    message:

        [edit system scripts op]
        +     file bgp-neighbors.slax;
        +     file bgp-summary.slax;
    out:
        True
```

You can also define a state that returns the same information.

```
saltuser@salt-master:~$ cat /srv/salt/junos_diff.sls
Retrieve configuration differences:
  junos.diff:
    - d_id: 0
```

When you apply the state to one or more targets, it returns the differences for each target.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_diff

router1:
----------
          ID: Retrieve configuration differences
    Function: junos.diff
      Result: True
     Comment:
     Started: 00:09:55.359546
    Duration: 3110.709 ms
     Changes:
              ----------
              message:

                  [edit system scripts op]
                  +     file bgp-neighbors.slax;
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
```

```
Total states run:     1
Total run time:   3.111 s
```

RELATED DOCUMENTATION

Use Salt to Retrieve a Junos OS Configuration | 81

Use Salt to Configure Devices Running Junos OS | 94

Use Salt to Configure Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS, and the Junos execution and state modules (for Salt) define functions that enable you to manage the Junos OS configuration. This topic discusses how to use Salt to generate, provision, and manage Junos OS configurations.

Overview of Using Salt to Provision Junos OS Configurations

The Junos execution and state modules (for Salt) enable you to use Salt to manage the configuration of a device running Junos OS. The modules define functions to perform the following tasks:

- Lock and unlock the configuration database
- Load configuration data
- Commit the configuration
- Roll back the configuration
- Configure the device hostname and commit the change

Table 7 on page 95 lists the functions you can use to manage the configuration and provides a brief description of each function. You can use the execution functions to perform the operations on the Salt master command line. But more often, you will define the state in which the managed device must be and use the state functions to apply the configuration.

Table 7: Junos Execution and State Functions to Manage the Configuration

Junos Execution Function	Junos State Function	Description
`salt.modules.junos.commit`	`salt.states.junos.commit`	Commit the changes loaded into the target configuration database.
`salt.modules.junos.commit_check`	`salt.states.junos.commit_check`	Perform a `commit check` operation on the candidate configuration to verify that the syntax is correct.
`salt.modules.junos.install_config`	`salt.states.junos.install_config`	Lock, modify, commit, and unlock the target configuration database.
`salt.modules.junos.load`	`salt.states.junos.load`	Load the configuration data in the specified file into the target configuration database.
`salt.modules.junos.lock`	`salt.states.junos.lock`	Request an exclusive lock on the candidate configuration.

Table 7: Junos Execution and State Functions to Manage the Configuration *(Continued)*

Junos Execution Function	Junos State Function	Description
`salt.modules.junos.rollback`	`salt.states.junos.rollback`	Roll back the configuration to a previously committed configuration and commit it.
`salt.modules.junos.set_hostname`	`salt.states.junos.set_hostname`	Configure the hostname of a managed device running Junos OS and commit the change.
`salt.modules.junos.unlock`	`salt.states.junos.unlock`	Release the exclusive lock on the candidate configuration.

The basic process for making configuration changes is to lock the configuration database, load the configuration changes, commit the configuration to make it active, and then unlock the configuration database. You can use the corresponding function to execute each operation individually, or you can use the `junos.install_config` function to execute all of the operations with a single function call.

By default, the `junos.install_config` function makes changes to the candidate configuration database using `configure exclusive` mode, which automatically locks and unlocks the configuration database. The function compares the loaded configuration with the current configuration and only applies the new configuration if there are changes. If the function modifies the configuration, it then performs a `commit check` and `commit` operation.

We recommend using the `junos.install_config` function to make configuration changes, because it handles the entire configuration workflow. The function also enables you to save the configuration differences to a file and use different configuration modes. For example, you can use `configure private` mode to modify a private copy of the candidate configuration, as described in "How to Specify the Configuration Mode" on page 99.

When loading new configuration data with the `junos.load` or `junos.install_config` function, you can 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, including merge, override, replace, and update. For more information, see "How to Specify the Load Operation" on page 100.

- 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 more information, see "How to Specify the Format of the Configuration Data to Load" on page 102.

- Configuration data source—You can load configuration data from a file containing a partial or complete configuration or a Jinja2 template. For details, see "How to Load Configuration Data" on page 102 and "How to Load Configuration Data Using a Jinja2 Template" on page 105.

After modifying the configuration, you must commit the configuration to make it the active configuration on the device. The `junos.install_config`, `junos.rollback`, and `junos.set_hostname` functions automatically commit the changes to the configuration. You can also execute the `junos.commit` function to perform a commit operation. For information about the behavior of the different functions and their supported commit options, see "How to Commit the Configuration" on page 109.

In addition to loading new configuration data, you can use the `junos.rollback` function to roll back the configuration to a previously committed configuration. The function automatically commits the updated configuration. For more information, see "How to Roll Back the Configuration" on page 114.

How to Lock and Unlock the Configuration Database

You can lock the candidate configuration before modifying it to prevent other users or applications from updating it until the lock is released. This is equivalent to the `configure exclusive` command in the CLI. We recommend locking the configuration before making changes, particularly on devices where multiple users are authorized to change the configuration, because a commit operation applies to all changes in the candidate configuration, not just those made by the user or application that requests the commit.

The `junos.install_config` function makes changes to the candidate configuration database using `configure exclusive` mode, which automatically locks the configuration database, loads and commits the changes, and unlocks the database. However if you need to execute the load and commit operations separately, for example, by using the `junos.load` and `junos.commit` functions, you can explicitly lock and unlock the database by using the `junos.lock` and `junos.unlock` execution or state functions.

To explicitly lock the configuration database before modifying it, use the `junos.lock` function. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.lock
router1:
    ----------
    message:
        Successfully locked the configuration.
    out:
        True
```

To unlock the database and discard any uncommitted changes, use the `junos.unlock` function. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.unlock
router1:
    ----------
    message:
        Successfully unlocked the configuration.
    out:
        True
```

If the configuration database has been modified, or if another user already has an exclusive lock on it, the `junos.lock` function returns a `LockError` message, as shown in the following examples:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.lock
router1:
    ----------
    message:
        Could not gain lock due to : "LockError(severity: error, bad_element: None, message: 
configuration database modified)"
    out:
        False
```

```
saltuser@salt-master:~$ sudo salt 'router1' junos.lock
router1:
    ----------
    message:
        Could not gain lock due to : "LockError(severity: error, bad_element: None, message: 
configuration database locked by:
          admin terminal p1 (pid 28508) on since 2019-08-12 12:46:52 PDT
              exclusive [edit])"
    out:
        False
```

How to Specify the Configuration Mode

The `junos.install_config` function enables you to make changes in different configuration modes. By default, the function makes changes to the candidate configuration database using `configure exclusive` mode. The `configure exclusive` mode locks the candidate global configuration (also known as the *shared configuration database*) for as long as the function 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 a different mode, set the `mode` parameter equal to the desired mode. Supported modes include `batch`, `dynamic`, `ephemeral`, `exclusive`, and `private`. For information about the different modes, see the *CLI User Guide*.

For example, the following command makes changes to the configuration using `configure private` mode, which opens a private copy of the candidate configuration:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.install_config 'salt://configs/mpls-
config.set' mode=private
router1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

Similarly, you can include the argument in a Salt state file.

```
saltuser@salt-master:~$ cat /srv/salt/junos_mpls_config.sls
Install MPLS Config:
  junos.install_config:
    - name: salt://configs/mpls-config.set
    - mode: private
```

You can also use the `junos.install_config` function to update the ephemeral configuration database on devices that support this database. The ephemeral database is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS. You must have Salt version 3001 and Junos PyEZ Release 2.1.3 or later to use this feature

NOTE: The ephemeral configuration database is an advanced feature, which if used incorrectly can have a serious negative impact on the operation of the device. For more information, see Understanding the Ephemeral Configuration Database.

To configure the default instance of the ephemeral configuration database, set the `mode` argument equal to `ephemeral`. For example:

```
saltuser@salt-master:~$ cat /srv/salt/junos_ephemeral_config.sls
Install MPLS config in default ephemeral instance:
  junos.install_config:
    - name: salt://configs/mpls-ephemeral-config.set
    - mode: ephemeral
```

To configure a user-defined instance of the ephemeral configuration database, set the `mode` argument equal to `ephemeral`, and set the `ephemeral_instance` argument equal to the name of the instance.

```
saltuser@salt-master:~$ cat /srv/salt/junos_ephemeral_instance_config.sls
Install MPLS config in ephemeral instance:
  junos.install_config:
    - name: salt://configs/mpls-ephemeral-instance-config.set
    - mode: ephemeral
    - ephemeral_instance: eph1
```

How to Specify the Load Operation

You can use the `junos.load` and `junos.install_config` functions to load configuration changes using a `load merge`, `load replace`, `load override`, or `load update` operation. You specify the desired load operation by including or omitting the appropriate arguments for that function. If you do not specify a load operation, the default is `load replace`. Table 8 on page 101 summarizes the arguments required for each type of load operation.

Table 8: Parameters for Specifying the Load Operation

Load Operation	Function Argument	Description
`load merge`	`merge=True`	Merge the loaded configuration with the existing configuration.
`load override`	`overwrite=True`	Replace the entire configuration with the loaded configuration.
`load replace` (Default)	–	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the `replace:` tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
`load update`	`update=True`	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.

The following command replaces the entire configuration on the target device with the specified configuration and commits it. The RPC timeout is increased to provide sufficient time to load and commit the complete configuration.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.install_config overwrite=True 'salt://configs/
junos-complete-config.conf' dev_timeout=60
```

The equivalent state file is:

```
Replace complete configuration:
  junos.install_config:
    - name: salt://configs/junos-complete-config.conf
    - overwrite: True
    - dev_timeout: 60
```

How to Specify the Format of the Configuration Data to Load

The `junos.load` and `junos.install_config` functions enable you to configure devices running Junos OS using one of the standard, supported formats. You can load configuration data from a file containing a partial or complete configuration or a Jinja2 template. The data can be provided as text, Junos XML elements, Junos OS `set` commands, or JSON.

You must specify the format of the configuration data either by adding the appropriate extension to the configuration file or by explicitly including the `format` argument in the function call. Table 9 on page 102 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 the format indicated by the file extension.

Table 9: 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: Devices running Junos OS support loading configuration data in JSON format starting in Junos OS Release 16.1R1.

How to Load Configuration Data

The `junos.load` and `junos.install_config` functions enable you to load configuration data from a file that contains a partial or complete configuration or a Jinja2 template. The file can reside on the Salt master or the proxy minion server. The functions must use the **salt://** notation to specify a path on the Salt master, and they must use an absolute path to specify a path on the proxy minion server. If the file does not use one of the accepted file extensions to indicate the format of the configuration data, then the function call must also include the `format` argument to specify the format of the data.

Table 10 on page 103 describes the configuration data sources that you can use and lists the execution and state function arguments required to specify the location of the file. The execution functions can specify the path as a positional argument or by using the `path` keyword.

Table 10: Configuration Data Sources

Configuration Data Source	Description	Location	Execution Function Argument	State Function Argument
Configuration data file	File containing configuration data formatted as ASCII text, Junos XML elements, Junos OS `set` commands, or JSON.	Salt master—use **salt://** notation Proxy minion server—use an absolute path	`path`	`name`
Jinja2 template file	File containing a Jinja2 template formatted as ASCII text, Junos XML elements, Junos OS `set` commands, or JSON. Include the `template_vars` parameter in the function call to supply a dictionary of any required Jinja2 template variables that are not already defined in the Salt system, for example, as pillar or grain data.	Salt master—use **salt://** notation Proxy minion server—use an absolute path	`path`	`name`

For example, the following file contains Junos OS `set` commands that configure two op scripts. The file's extension indicates the format for the configuration data.

```
saltuser@salt-master:~$ cat /srv/salt/configs/op-scripts.set
set system scripts op file bgp.slax
set system scripts op file ospf.slax
```

The following Salt command uses the `junos.install_config` execution function to load and commit the configuration on the target device. The path is provided as a positional argument.

```
saltuser@salt-master:~$ sudo salt 'router1' junos.install_config 'salt://configs/op-scripts.set'
router1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

The equivalent state file is:

```
saltuser@salt-master:~$ cat /srv/salt/junos-config-op-scripts.sls
Installing op scripts:
  junos.install_config:
    - name: salt://configs/op-scripts.set
    - comment: committed using Salt
```

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos-config-op-scripts
router1:
----------
          ID: Installing op scripts
    Function: junos.install_config
        Name: salt://configs/op-scripts.set
      Result: True
     Comment:
     Started: 04:19:59.819155
    Duration: 14853.384 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:  14.853 s
```

The `junos.install_config` function automatically performs a diff between the candidate configuration and the requested configuration. The function only applies the configuration if there are changes. If you attempt to apply the same configuration a second time, the `junos.install_config` function returns a message that the configuration has already been applied and does not load and commit it again, as shown in the following example:

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos-config-op-scripts
router1:
----------
          ID: Installing op scripts
    Function: junos.install_config
        Name: salt://configs/op-scripts.set
      Result: True
     Comment:
     Started: 06:01:52.365353
    Duration: 778.843 ms
     Changes:
              ----------
              message:
                  Configuration already applied!
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 778.843 ms
```

How to Load Configuration Data Using a Jinja2 Template

The `junos.load` and `junos.install_config` execution and state functions support using Jinja2 templates for Junos OS configuration data. 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 functions use the Jinja2 template and a supplied dictionary of variables to render the configuration data.

Jinja2 templates provide a powerful method to generate configuration data, particularly for similar configuration stanzas. For example, rather than manually adding the same configuration statements for each interface on a device, you can create a template that iterates over a list of interfaces and creates the required configuration statements for each one. In Jinja, blocks are delimited by '{%' and '%}' and variables are enclosed within '{{' and '}}'.

To load a Jinja2 template, you must include the following parameters in the `junos.load` or `junos.install_config` function call:

- Template path—Specify the path to a template file on the Salt master or proxy minion server, as described in "How to Load Configuration Data" on page 102.
- Template format—Set the `format` argument to indicate the format of the configuration data if the template file does not use one of the accepted file extensions to specify the format. For information about specifying the format, see "How to Specify the Format of the Configuration Data to Load" on page 102.
- Template variables—The template can reference Salt internal variables like those defined in pillar or grain data. Any variables that are not already defined within the Salt system must be supplied in the `template_vars` argument. The `template_vars` value is a dictionary of keys and values that are required to render the Jinja2 template.

  NOTE: If your template only includes Salt internal variables like pillar data, grain data, and functions, the `junos.install_config` function might need to define `template_vars: True` in order to render the template.

The following sample Jinja2 template generates configuration data that enables MPLS on logical unit 0 for each interface in a given list and also configures the interface under the MPLS and RSVP protocols.

```
saltuser@salt-master:~$ cat /srv/salt/configs/junos-config-mpls-jinja2-template.conf
interfaces {
    {% for item in template_vars['interfaces'] %}
    {{ item }} {
        description "{{ template_vars['description'] }}";
        unit 0 {
            family {{ template_vars['family'] }};
        }
    } {% endfor %}
}
protocols {
    mpls {
        {% for item in template_vars['interfaces'] %}
        interface {{ item }};
        {% endfor %}
    }
    rsvp {
        {% for item in template_vars['interfaces'] %}
        interface {{ item }};
        {% endfor %}
    }

}
```

The following command uses the Jinja2 template and the variables defined in `template_vars` to render the configuration data, which is then loaded and committed on the target host:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.install_config 'salt://configs/junos-config-
mpls-jinja2-template.conf' template_vars='{ "interfaces" : ["ge-1/0/1", "ge-1/0/2", "ge-1/0/3"],
"description" : "MPLS interface", "family" : "mpls" }'
router1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

The following state file loads the same configuration. The configuration differences are stored in the `diffs_file` file on the proxy minion server.

```
saltuser@salt-master:~$ cat /srv/salt/junos_install_config_mpls.sls
Install Junos OS config:
  junos.install_config:
    - name: 'salt://configs/junos-config-mpls-jinja2-template.conf'
    - comment: committed using Salt
    - diffs_file: /home/saltuser/junos-config-mpls-diff
    - template_vars:
        interfaces: ['ge-1/0/1', 'ge-1/0/2', 'ge-1/0/3']
        description: MPLS interface
        family: mpls
```

When you apply the state, Salt renders the configuration and loads and commits the configuration on the device.

```
saltuser@salt-master:~$ sudo salt 'router1' state.apply junos_install_config_mpls
router1:
----------
          ID: Install Junos OS config
    Function: junos.install_config
        Name: salt://configs/junos-config-mpls-jinja2-template.conf
      Result: True
     Comment: 
     Started: 05:28:51.575045
    Duration: 23675.957 ms
     Changes:   
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for router1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:  23.676 s
```

The function 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-1/0/1.0;
        interface ge-1/0/2.0;
        interface ge-1/0/3.0;
    }
    rsvp {
        interface ge-1/0/1.0;
        interface ge-1/0/2.0;
        interface ge-1/0/3.0;
    }
}
```

How to Commit the Configuration

After modifying the configuration, you must commit the configuration to make it the active configuration on the device. The `junos.install_config`, `junos.rollback`, and `junos.set_hostname` functions load the requested configuration changes and then automatically perform a commit check and commit operation. You can also use the individual `junos.commit_check` and `junos.commit` functions to perform a commit check and commit operation, for example, after using the `junos.load` function to update the configuration.

The Junos OS CLI provides options for the commit operation, such as adding a commit comment or synchronizing the configuration on multiple Routing Engines. Some of these options are supported by the Junos execution and state module functions. Table 11 on page 110 outlines the commit options that are available and the functions that support them. Supported arguments are valid for both the execution function and the state function.

Table 11: Arguments for Commit Options

Function Argument	Description	Functions That Support the Argument	CLI Command Equivalent

`comment=`*`comment`*	Log a comment for that commit operation in the system log file and in the device's commit history.	`junos.commit`
		`junos.install_config`
		`junos.rollback`
		`junos.set_hostname`
			`commit comment "`*`comment`*`"`

`confirm=(True | `*`minutes`*`)`	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.
	
	Set the argument to `True` to use the default time of 10 minutes.
		`junos.commit`
		`junos.install_config`
		`junos.rollback`
		`junos.set_hostname`
			`commit confirmed <`*`minutes`*`>`

`detail=True`	Return detailed information about the commit process.	`junos.commit`	`commit | display detail`

`dev_timeout=`*`seconds`*	Wait for completion of the operation using the specified value as the timeout.	`junos.commit`
		`junos.install_config`
		`junos.rollback`
		`junos.set_hostname`
			–

`force_sync=True`	Synchronize and commit the configuration on both Routing Engines, even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine.	`junos.commit`	`commit synchronize force`
`sync=True`	Synchronize and commit the configuration on both Routing Engines.	`junos.commit`	`commit synchronize`

Commit Comment

When you commit the configuration, you can include a brief comment to describe the purpose of the committed changes. To log a comment describing the changes, include the `comment` argument and a message string. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' junos.commit comment='Committed using Salt'
```

Similarly, in a state file:

```
saltuser@salt-master:~$  cat /srv/salt/junos_install_mx_config.sls
Install Junos OS config:
  junos.install_config:
    - name: salt://configs/mx-config-common.set
    - comment: Committed using Salt
```

The commit comment is included in the system log message for the commit as well as logged in the commit history.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.cli 'show system commit'
router1:
    ----------
    message:

        0   2019-08-05 15:08:01 PDT by saltuser via netconf
            Committed using Salt
        ...
      out:
        True
```

Commit Confirmed

When you commit the candidate configuration, you can require an explicit confirmation for the commit to become permanent. If the commit is not confirmed within the specified amount of time, the device automatically loads and commits (rolls back to) the previously committed configuration. 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 restores access to the device after the rollback deadline passes.

To require that a commit operation be confirmed within a specified amount of time after the initial commit, include the `confirm=minutes` argument. The allowed range is 1 through 65,535 minutes. You can also specify `confirm=True` to use the default time of 10 minutes.

The following command requires that the commit be confirmed within 15 minutes:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.install_config 'salt://configs/mx-config-common.set' confirm=15
router1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

To confirm the commit operation, call either the `junos.commit` or `junos.commit_check` function.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.commit comment='Confirming commit using Salt'
router1:
    ----------
    message:
        Commit Successful.
    out:
        True
```

Commit Detail

When you use the `junos.commit` function to commit the configuration, you can review the details of the entire commit operation by including the `detail=True` argument. When you include this argument, the function returns detailed information about the commit process.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.commit detail=True
router1:
    ----------
    message:
        ----------
        routing-engine:
            ----------
            name:
                re0
            progress-indicator:
```

```
            |_
              ----------
              message:
                  Obtaining lock for commit
              timestamp:
                  2019-08-14 11:17:50 PDT
            |_
              ----------
              message:
                  updating commit revision
              timestamp:
                  2019-08-14 11:17:50 PDT
...
```

Commit Synchronize

When you use the `junos.commit` function to commit the configuration, you can synchronize and commit the configuration on both Routing Engines in a dual Routing Engine system by including the `sync=True` argument. For example:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.commit sync=True
```

When you include the `sync=True` argument, the device copies the candidate configuration stored on the local Routing Engine to the other Routing Engine, verifies the candidate's syntactic correctness, and commits it on both Routing Engines. To force the `commit synchronize` operation to succeed even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine, use the `force_sync=True` argument. Including this argument causes the device to terminate any configuration sessions on the other Routing Engine before synchronizing and committing the configuration.

Commit Timeout

The default time for an 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. To accommodate configuration changes that might require a commit time that is longer than the default timeout interval, include the `dev_timeout=`*`seconds`* argument, and set the timeout interval to an appropriate value. For example:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.install_config 'salt://configs/
junos_mx_config.conf' dev_timeout=60
```

```
saltuser@salt-master:~$ cat /srv/salt/junos_install_mx_config.sls
Install Config:
  junos.install_config:
    - name: salt://configs/junos_mx_config.conf
    - dev_timeout: 60
```

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 `salt.modules.junos.rollback` execution function and the `salt.states.junos.rollback` state function enable you to roll back the configuration to a previously committed configuration on a device running Junos OS. To roll back the configuration and commit it, execute the function, and set the `id` argument to the ID of the desired 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). If you omit the `id` keyword, it defaults to 0.

For example, the following command rolls the configuration back to the previously committed configuration and commits it:

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rollback id=1 comment='Rolling back
configuration using Salt'
router1:
    ----------
    message:
        Rollback successful
    out:
        True
```

To roll back the configuration and log the configuration differences in a file for later reference, include the `diffs_file` argument, and set it to the path of the file on the proxy minion server to which the differences are written.

```
saltuser@salt-master:~$  sudo salt 'router1' junos.rollback id=1 comment='Rolling back
configuration using Salt' diffs_file='/home/saltuser/router1-rollback-diff'
router1:
    ----------
    message:
        Rollback successful
    out:
        True
```

The differences are saved to the specified file on the proxy minion server.

```
saltuser@minion:~$  cat /home/saltuser/router1-rollback-diff
[edit system scripts op]
-     file bgp-neighbors.slax;
```

RELATED DOCUMENTATION

Example: Use Salt to Configure Devices Running Junos OS | 115

Use Salt to Retrieve a Junos OS Configuration | 81

Example: Use Salt to Configure Devices Running Junos OS

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS. This example uses Salt to configure several devices running Junos OS.

Requirements

This example uses the following hardware and software components:

- Three MX Series routers running Junos OS with NETCONF enabled
- Salt master with the following requirements:
  - Salt version 3001 or later
  - Can ping and perform operations on the devices running Junos OS

Overview

This example defines a Salt state that configures BGP peering sessions on the target devices running Junos OS. Table 12 on page 116 outlines the device hostnames, proxy IDs, and device-specific files.

Table 12: Device Information

Hostname	Proxy ID	Proxy Configuration File **(/srv/pillar)**	BGP Data File (/srv/pillar/bgp)
r1	r1	**r1-proxy.sls**	r1.sls
r2	r2	**r2-proxy.sls**	r2.sls

Table 12: Device Information *(Continued)*

Hostname	Proxy ID	Proxy Configuration File **(/srv/pillar)**	BGP Data File (/srv/pillar/bgp)
r3	r3	**r3-proxy.sls**	r3.sls

The example uses the following components:

- Jinja2 configuration template—Defines the BGP configuration as a Jinja template using `set` command format. The template uses the **.set** file extension to indicate the format.
- Pillar files—Define the device-specific configuration data required by the Jinja2 template. The data for each device is stored in a separate file under the **/srv/pillar/bgp** directory on the Salt master.
- Pillar top file—Maps each pillar file to the appropriate proxy minion.
- State file—Defines the state to apply to the target devices. In this case, the state file uses the `junos.install_config` function to apply the BGP configuration to a device.
- State top file—Maps the state file to the devices on which the state should be applied.

In this example, you create a Jinja2 template to generate the BGP configuration that gets loaded and committed on the device. The template substitutes in variables for device-specific configuration data so that it can be reused as needed. The template file is placed under the **/srv/salt/configs** directory on the Salt master.

You then create separate pillar files for each device and define the device-specific configuration data in the file. Each pillar file defines a `BGP_data` key that contains all of the variable parameters that are referenced in the configuration template. The pillar top file maps each pillar file to its respective proxy minion. Storing the data in pillars ensures that each proxy minion can only access the data for that device.

The **junos_bgp_config.sls** state file defines a state that uses the `junos.install_config` function to apply the BGP configuration in the template. When invoked, the function locks the configuration database, loads the configuration specific to the target device, commits the configuration, and then unlocks the database. The function includes the `diffs_file` parameter to store the configuration differences in a file on the proxy minion server. The filename uses the `id` grain to generate device-specific filenames for the output files.

The state top file applies the `junos_bgp_config` state to any minion with a `BGP_data` pillar key with any value. When you run a highstate or apply that state to a device, Salt uses the template and pillar data to generate the configuration specific to the target and then invokes the `junos.install_config` function to load and commit the configuration on the device. If the configuration is already applied, the `junos.install_config` function does not reapply the configuration.

After you configure the devices, you then apply the `junos_bgp_verify_peers` state. This state executes the `get-bgp-neighbor-information` RPC on each device until it returns a `peer-state` value of `Established` for each peer defined in that device's pillar data or times out. This state assumes that BGP is running on the device.

Configuration

IN THIS SECTION

To configure the devices using Salt, perform the tasks included in this section.

Define the Pillar Data

Step-by-Step Procedure

To define the pillar data that is used with the Jinja2 template to generate the BGP configuration:

1. On the Salt master, create a separate file named **/srv/pillar/bgp/*hostname*.sls** for each managed device.

2. Define the data for host r1 in the **r1.sls** file.

```
BGP_data:
  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 data for host r2 in the **r2.sls** file.

```
BGP_data:
  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 data for host r3 in the **r3.sls** file.

```
BGP_data:
  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
```

5. In the pillar top file, map the pillar file with the BGP data under the appropriate device ID to enable the device to access the data.

```
saltuser@salt-master:~$ cat /srv/pillar/top.sls
base:
  'r1':
    - r1-proxy
    - bgp/r1
  'r2':
    - r2-proxy
    - bgp/r2
  'r3':
    - r3-proxy
    - bgp/r3
```

6. Refresh the pillar data.

```
saltuser@salt-master:~$ sudo salt '*' saltutil.refresh_pillar
r3:
    True
r1:
    True
r2:
    True
```

Define the Jinja2 Template

Step-by-Step Procedure

To create the Jinja2 template that is used to generate the BGP configuration:

1. Create a file named **/srv/salt/configs/junos-config-bgp-template.set** on the Salt master.

2. Add the BGP configuration template to the file and save it.

```
saltuser@salt-master:~$ cat /srv/salt/configs/junos-config-bgp-template.set
{% if pillar.BGP_data %}

set interfaces lo0 unit 0 family inet address {{ pillar.BGP_data.loopback }}/32
set policy-options policy-statement bgp-ecmp then load-balance per-packet
set policy-options policy-statement bgp-in then accept
set policy-options policy-statement bgp-out then next-hop self
set policy-options policy-statement bgp-out then accept
set protocols bgp group underlay type external
set protocols bgp group underlay import bgp-in
set protocols bgp group underlay export bgp-out
set protocols bgp group underlay local-as {{ pillar.BGP_data.local_asn }}
set protocols bgp group underlay multipath multiple-as
set routing-options router-id {{ pillar.BGP_data.loopback }}
set routing-options forwarding-table export bgp-ecmp

{% for neighbor in pillar.BGP_data.neighbors %}
set interfaces {{ neighbor.interface }} unit 0 description {{ neighbor.name }}
set interfaces {{ neighbor.interface }} unit 0 family inet address {{ neighbor.local_ip }}/30
set protocols bgp group underlay neighbor {{ neighbor.peer_ip }} peer-as {{ neighbor.asn }}
set protocols lldp interface {{ neighbor.interface }}
{% endfor %}

{% endif %}
```

Define the States

Define the Configuration State File

To define the state file that applies the configuration:

1. Create a file named **/srv/salt/junos_bgp_config.sls** on the Salt master.

2. Define a state that uses the `junos.install_config` function to apply the BGP configuration in the template.

```
saltuser@salt-master:~$ cat /srv/salt/junos_bgp_config.sls
Apply BGP configuration:
  junos.install_config:
    - name: salt://configs/junos-config-bgp-template.set
    - comment: Configuring BGP using Salt
    - diffs_file: /var/log/salt/output/{{ grains['id'] }}_junos_bgp_config_diff
    - template_vars: True
```

NOTE: If your template only includes Salt internal variables like pillar data, grain data, and functions, the `junos.install_config` function might need to define `template_vars: True` in order to render the template.

3. In the top file, define the targets to which the **junos_bgp_config** state applies. The targets comprise all devices that have a `BGP_data` pillar item defined.

```
saltuser@salt-master:~$ cat /srv/salt/top.sls
base:
  BGP_data:*:
    - match: pillar
    - junos_bgp_config
```

Define the BGP Verification State File

To define the state that verifies that the BPG peers have a `peer-state` of `Established`:

1. Create a file named **/srv/salt/junos_bgp_verify_peers.sls** on the Salt master.

2. Define a state that uses the `junos.rpc` function to execute the `get-bgp-neighbor-information` RPC and then checks the RPC reply for a `peer-state` of `Established`.

```
saltuser@salt-master:~$ cat /srv/salt/junos_bgp_verify_peers.sls
{% for peer in pillar['BGP_data']['neighbors'] %}
validate_bgp_session_state_with_{{ peer['peer_ip'] }}:
  loop.until:
    - name: junos.rpc
    - condition: m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established'
    - period: 5
    - timeout: 20
    - m_args:
      - get-bgp-neighbor-information
    - m_kwargs:
        neighbor-address: {{ peer['peer_ip'] }}
{% endfor %}
```

Results

When you execute a highstate, the states in the `top.sls` file are applied to the appropriate target devices.

```
saltuser@salt-master:~$ sudo salt '*' state.apply

r1:
----------
          ID: Apply BGP configuration
    Function: junos.install_config
        Name: salt://configs/junos-config-bgp-template.set
      Result: True
     Comment:
     Started: 07:07:14.449582
    Duration: 3379.914 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for r1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.380 s
r2:
----------
          ID: Apply BGP configuration
    Function: junos.install_config
        Name: salt://configs/junos-config-bgp-template.set
      Result: True
     Comment:
     Started: 07:07:14.485640
    Duration: 3132.677 ms
```

```
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for r2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.133 s
r3:
----------
          ID: Apply BGP configuration
    Function: junos.install_config
        Name: salt://configs/junos-config-bgp-template.set
      Result: True
     Comment:
     Started: 07:07:14.388629
    Duration: 3431.723 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for r3
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.432 s
```

Verification

IN THIS SECTION

Verifying the BGP Configuration

Purpose

Verify that the BGP session is established for each neighbor address.

Action

Apply the `junos_bgp_verify_peers` state to the target devices and review the output.

```
saltuser@salt-master:~$ sudo salt '*' state.apply junos_bgp_verify_peers
r1:
----------
          ID: validate_bgp_session_state_with_198.51.100.2
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:01.825414
    Duration: 125.241 ms
     Changes:
----------
          ID: validate_bgp_session_state_with_198.51.100.6
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:01.950786
    Duration: 148.944 ms
```

```
     Changes:

Summary for r1
------------
Succeeded: 2
Failed:    0
------------
Total states run:     2
Total run time: 274.185 ms
r3:
----------
          ID: validate_bgp_session_state_with_198.51.100.5
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:02.849612
    Duration: 99.527 ms
     Changes:
----------
          ID: validate_bgp_session_state_with_198.51.100.9
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:02.949265
    Duration: 165.041 ms
     Changes:

Summary for r3
------------
Succeeded: 2
Failed:    0
------------
Total states run:     2
Total run time: 264.568 ms
r2:
----------
          ID: validate_bgp_session_state_with_198.51.100.1
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:02.811094
    Duration: 143.335 ms
     Changes:
----------
          ID: validate_bgp_session_state_with_198.51.100.10
    Function: loop.until
        Name: junos.rpc
      Result: True
     Comment: Condition m_ret['rpc_reply']['bgp-information']['bgp-peer']['peer-state'] ==
'Established' was met
     Started: 07:17:02.954551
    Duration: 170.651 ms
     Changes:

Summary for r2
------------
Succeeded: 2
Failed:    0
------------
Total states run:     2
Total run time: 313.986 ms
```

Meaning

The state file requires that the `peer-state` is equal to `Established` for each peer. The output indicates that this condition is met for all peers on each device. Alternatively, you can use the `junos.cli` function to execute the `show bgp summary` command on the devices and review the output to verify that the BGP session is established for each neighbor address.

Troubleshooting

IN THIS SECTION

Troubleshooting Configuration Load Errors

Problem

The Salt master generates a `ConfigLoadError` error indicating that it failed to load the configuration on the device because of a syntax error.

```
message:
    Could not load configuration due to : "ConfigLoadError(severity: error, bad_element:
interface1, message: error: syntax error)"
```

Solution

Salt renders the Junos OS configuration by using the Jinja2 template and the pillar data defined for that device. Salt 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.

RELATED DOCUMENTATION

Use Salt to Configure Devices Running Junos OS | **94**

Use Salt to Retrieve a Junos OS Configuration | **81**

Use Salt to Compare the Junos OS Candidate Configuration with a Previously Committed Configuration | **91**

5
CHAPTER

Junos Syslog Engine for Salt

IN THIS CHAPTER

Junos Syslog Engine for Salt

IN THIS SECTION

Juniper Networks provides support for using Salt to manage devices running Junos OS. The Salt installation includes the Junos syslog engine (for Salt), which listens for Junos OS system log messages from managed devices and publishes them on the Salt event bus. The Junos syslog engine, in conjunction with existing or custom reactors, enables you to use Salt for event-based state management of devices running Junos OS.

The following sections discuss how the Junos syslog engine works, how to configure and enable it, how to configure the events to send and subscribe to, and how to view these events on the event bus.

Understanding the Junos Syslog Engine

Junos OS generates system log messages (also called *syslog* messages) to record events that occur on the device, including events for routine operations, failure and error conditions, and emergency or critical conditions. System log messages can contain the following information:

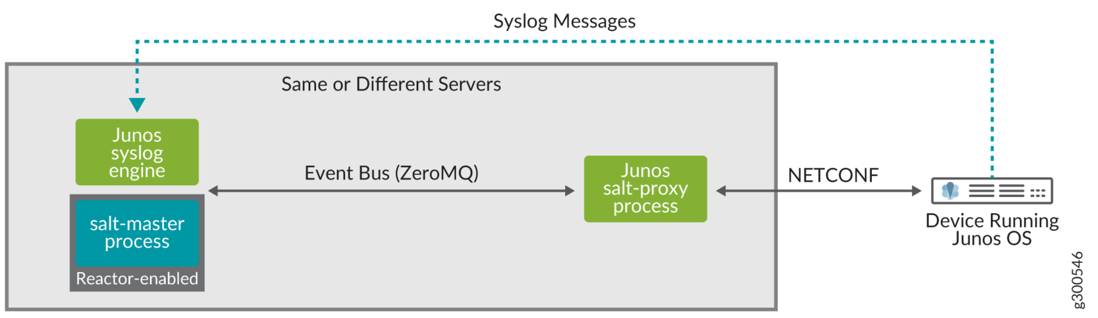
- Junos OS process that generated the message
- Date and time the message was generated
- Severity of the event
- Tag that uniquely identifies the event
- String that provides information about the event

For example, the following syslog message records the successful completion of a commit operation on router1:

```
Sep  3 11:52:22  router1 mgd[83498]: UI_COMMIT_COMPLETED: commit complete
```

Salt engines are external system processes that are monitored by and leverage Salt. Salt engines can export or import events on the Salt event bus. The Salt installation includes the Junos syslog engine, which can monitor Junos OS system log messages that are sent to the Salt server and publish them on the Salt event bus. Figure 4 on page 131 illustrates a Salt setup that includes the Junos syslog engine. When enabled, the Junos syslog engine listens on the specified port for syslog events from managed devices running Junos OS.

Figure 4: Junos Syslog Engine



When the Junos syslog engine receives an event, it extracts the event information, translates it to Salt format, and publishes it on the Salt event bus. The following output represents the same commit complete event as it is displayed on the Salt event bus:

```
jnpr/syslog/router1/UI_COMMIT_COMPLETED {
    "_stamp": "2019-09-03T18:52:11.279878",
    "daemon": "mgd",
    "event": "UI_COMMIT_COMPLETED",
    "facility": 23,
    "hostip": "198.51.100.2",
    "hostname": "router1",
    "message": "commit complete",
    "pid": "83498",
    "priority": 188,
    "raw": "<188>Sep  3 11:52:22 router1 mgd[83498]: UI_COMMIT_COMPLETED: commit complete",
    "severity": 4,
    "timestamp": "2019-09-03 18:52:11"
}
```

Salt events all use the same basic data structure, which consists of an event tag and a body. The event tag is just a high-level description of the event, and the body is a dictionary that contains the event details. In the previous example, the event tag is `jnpr/syslog/router1/UI_COMMIT_COMPLETED`.

How to Enable the Junos Syslog Engine

The Junos syslog engine requires installing the `pyparsing` and `twisted` Python modules on the server where the engine runs. This documentation assumes the Junos syslog engine is running on the Salt master. For detailed instructions on installing the prerequisites and enabling the Junos syslog engine, see *Configuring the Junos Syslog Engine* in the Salt for Junos OS Quick Start Guide.

To use the Junos syslog engine, you must configure the device running Junos OS to send its syslog messages to a designated port on the Salt master. You can configure the device to send all events or only events corresponding to a specific facility or message severity level.

To send all events, configure `any any` to indicate all facilities and all message severity levels.

```
[edit]
saltuser@router1# set system syslog host salt-server port 9999 any any
saltuser@router1# commit
```

To send, for example, only security events with a severity level of info or higher, configure `security info`.

```
[edit]
saltuser@router1# set system syslog host salt-server port 9999 security info
saltuser@router1# commit
```

For more information about system logging facilities and message severity levels, see System Logging Facilities and Message Severity Levels.

To enable the Junos syslog engine, you must configure the engine under the `engines` key in the Salt master configuration file and specify the same port that you configured on the devices running Junos OS.

```
engines:
  - junos_syslog:
      port: 9999
```

To apply the configuration, you must restart the Salt master, which automatically starts the Junos syslog engine process. The Junos syslog engine then listens for Junos OS syslog messages on the specified port, extracts the event information, and publishes it in Salt format on the Salt event bus. To create and configure reactors that automatically take action when specific events occur, see "Configuring Salt Reactors for Junos OS Events" on page 137.

How to Configure the Event Tag

Events published to the Salt event bus have two components: the event tag and the data. Junos OS events use the following default event tag (or event topic) syntax:

```
jnpr/syslog/hostname/event
```

The event tag for Junos OS events must always start with `jnpr/syslog`, but you can customize the remaining fields by defining the `topic` parameter in the engine configuration and specifying the fields to include. For example, the following configuration generates event tags that include the device's IP address, the process that generated the message, and the event:

```
engines:
  - junos_syslog:
      port: 9999
      topic: jnpr/syslog/hostip/daemon/event
```

When you configure this syntax, the tag for a completed commit operation event might be:

```
jnpr/syslog/198.51.100.2/mgd/UI_COMMIT_COMPLETED
```

The event tag fields can include any combination of the following fields after `jnpr/syslog/`:

- `daemon`

- `event`
- `hostip`
- `hostname`
- `message`
- `pid`
- `priority`
- `raw (the raw event data forwarded from the device)`
- `severity`
- `timestamp`

How to Subscribe to Events for Specific Junos OS Processes

As described in "How to Enable the Junos Syslog Engine" on page 132, you can configure which syslog events the device running Junos OS sends to the Junos syslog engine. By default, the Junos syslog engine publishes the event data for all received events on the Salt event bus. You can also customize the Junos syslog engine to only subscribe to certain processes by configuring the `daemon` parameter in the engine configuration. When you configure this parameter, the Junos syslog engine only publishes the events pertaining to those processes on the Salt event bus.

The following Salt engine configuration subscribes the Junos syslog engine to events from the management process (mgd) and the routing protocol process (rpd):

```
engines:
  - junos_syslog:
      port: 9999
      topic: jnpr/syslog/hostname/event
      daemon:
        - mgd
        - rpd
```

After restarting the salt-master process to apply the configuration, the Junos syslog engine only publishes events from these two processes on the Salt event bus.

```
jnpr/syslog/router1/UI_DBASE_LOGIN_EVENT        {
    "_stamp": "2019-08-28T22:16:42.612723",
    "daemon": "mgd",
    "event": "UI_DBASE_LOGIN_EVENT",
    "facility": 23,
    "hostip": "198.51.100.2",
    "hostname": "router1",
    "message": "User 'saltuser' entering configuration mode",
    "pid": "52764",
    "priority": 189,
    "raw": "<189>Aug 28 15:17:00 router1 mgd[52764]: UI_DBASE_LOGIN_EVENT: User 'saltuser' 
entering configuration mode",
    "severity": 5,
    "timestamp": "2019-08-28 22:16:42"
}
```

How to View Events on the Salt Event Bus

Salt runners execute modules on the Salt master rather than on the minions. You execute runners by using the `salt-run` command. To view the events on the Salt master event bus in real time, execute the following command, which displays the events in the terminal:

```
saltuser@salt-master~$ sudo salt-run state.event pretty=True
jnpr/syslog/router1/UI_COMMIT_COMPLETED  {
    "_stamp": "2019-07-24T17:17:30.390374",
    "daemon": "mgd",
    "event": "UI_COMMIT_COMPLETED",
    "facility": 23,
    "hostip": "198.51.100.2",
    "hostname": "router1",
    "message": "commit complete",
    "pid": "5795",
    "priority": 188,
    "raw": "<188>Jul 24 10:17:38 router1 mgd[5795]: UI_COMMIT_COMPLETED: commit complete",
    "severity": 4,
    "timestamp": "2019-07-24 17:17:30"
```

If you want to quickly trigger an event, you can ping the minion as shown in the following example:

```
saltuser@salt-master~$ sudo salt 'router1' test.ping
```

The corresponding event bus output shows the Salt job and the minion’s response.

```
20190626185606864697    {
    "_stamp": "2019-06-26T18:56:06.865972",
    "minions": [
        "router1"
    ]
}
salt/job/20190626185606864697/new       {
    "_stamp": "2019-06-26T18:56:06.867352",
    "arg": [],
    "fun": "test.ping",
    "jid": "20190626185606864697",
    "minions": [
        "router1"
    ],
    "missing": [],
    "tgt": "router1",
    "tgt_type": "glob",
    "user": "sudo_saltuser"
}
salt/job/20190626185606864697/ret/router1        {
    "_stamp": "2019-06-26T18:56:06.968557",
    "cmd": "_return",
    "fun": "test.ping",
    "fun_args": [],
    "id": "router1",
    "jid": "20190626185606864697",
    "retcode": 0,
    "return": true,
    "success": true
}
```

To test the Junos syslog engine configuration, you can execute an operation on the device running Junos OS. The device must be configured to send messages with that operation's facility and that severity level to the Junos syslog engine. In addition, the engine must be subscribed to messages from that process (or all processes, which is the default). When you execute the operation, the Junos syslog engine publishes the event to the Salt event bus.

RELATED DOCUMENTATION

Configuring Salt Reactors for Junos OS Events | 137

Understanding the Salt for Junos OS Components | 9

Configuring Salt Reactors for Junos OS Events

IN THIS SECTION

Salt reactors provide the ability to take action in response to specific events on the Salt event bus. The Salt Reactor system monitors the events on the Salt event bus and triggers actions for matching events based on the configured rules. You can enable the Junos syslog engine (for Salt) to publish events from managed devices running Junos OS to the Salt event bus. You can then configure existing or custom reactors to take action on those events. This enables you to create an event-driven infrastructure (EDI) to achieve closed-loop automation in real time.

How to Configure Salt Reactors

IN THIS SECTION

Before you can use Salt reactors to respond to Junos OS events, you must first enable the Junos syslog engine to publish Junos OS events to the Salt event bus and configure the managed devices running Junos OS to send syslog messages to the server on which the engine is running. For more information, see "Junos Syslog Engine for Salt" on page 130. After enabling the Junos syslog engine, you can configure Salt reactors that are automatically triggered in response to certain Junos OS events.

To configure a reactor that reacts to a specific event:

1. Define the event and corresponding event tag that will trigger the action.
2. Create one or more reactor SLS files on the Salt master and define the actions to take when the particular event occurs.
3. In the Salt master configuration file, define the `reactor` option, and associate each event tag with one or more reactor files.
4. Restart the Salt master to apply the updated configuration.

These steps are described in more detail in the following sections:

How to Define the Event and Event Tag

The Junos syslog engine publishes Junos OS events to the Salt event bus using the following default event tag string. The tag includes the target's hostname and a Junos OS syslog event name.

```
jnpr/syslog/hostname/event
```

To define the event tag to match on from the Salt event bus, specify the hostname and the event that will trigger the reaction. You can use an asterisk (*) to match on the event for all hosts. For detailed information about Junos OS system log messages, see the System Log Explorer.

For example, to match on a `UI_COMMIT_COMPLETED` event from any managed device running Junos OS, you can use the following event tag:

```
jnpr/syslog/*/UI_COMMIT_COMPLETED
```

NOTE: If you customize the event tag by configuring the `topic` parameter in the `engines` configuration, your matching event tag must reflect the same fields.

When you configure the reactor in the Salt master configuration file, you will reference this event tag.

How to Create the Reactor SLS File

Reactor SLS files define the actions to take when a specific event occurs. Reactor SLS files are similar to other Salt SLS files in that they use YAML format and support using Jinja. Generally, the files are stored in a separate directory, for example, **/srv/reactor** or **/srv/salt/reactor**. Reactor files can use data from the event, execute actions on the Salt master or minions, and apply states to a target, among other things. For detailed information about writing reactor files, see the official Salt documentation.

For example, the following reactor file applies the `junos_backup_config` state to the target device. In this case, the target is the proxy ID that matches the `hostname` value in the event data.

```
saltuser@salt-master~$ cat /srv/reactor/junos_backup_on_commit.sls
Backup config for commit complete event:
  local.state.apply:
    - tgt: {{ data['hostname'] }}
    - arg:
        - junos_backup_config
```

How to Configure the Reactor

To configure a reactor, define the `reactor` option in the Salt master configuration file. You define the event that triggers the action and the action to take. For each event that you want to match on and perform an action, associate the event tag for that event with one or more reactor SLS files. When the event is fired on the event bus, it automatically triggers the action defined in the file.

```
reactor:
  - 'event-tag1':
    - reactor-file1
  - 'event-tag2':
    - reactor-file2
    - reactor-file3
```

For example, the following reactor configuration calls the **junos_backup_on_commit.sls** reactor file whenever a `UI_COMMIT_COMPLETED` event occurs for any managed device running Junos OS that is configured to send syslog events to the Junos syslog engine:

```
reactor:
  - 'jnpr/syslog/*/UI_COMMIT_COMPLETED':
    - /srv/reactor/junos_backup_on_commit.sls
```

To apply the reactor configuration, you must restart the Salt master using the commands appropriate for your operating system. For example:

```
saltuser@salt-master~$ sudo killall salt-master
saltuser@salt-master~$ sudo salt-master -d
```

Example: Responding to a Junos OS Commit Event

IN THIS SECTION

This example demonstrates a simple Salt reactor that generates a backup configuration file every time a managed device running Junos OS commits a configuration.

Requirements

This example uses the following hardware and software components:

- Device running Junos OS with NETCONF enabled
- Salt master with the following requirements:
    - Salt version 3001 (Sodium) or later

- Can ping and perform operations on the device running Junos OS

Overview

This example generates a backup configuration file every time a managed device running Junos OS commits a configuration. The example configures the Salt Reactor system to trigger the action when it matches on a `UI_COMMIT_COMPLETED` event. A sample event is shown here for reference:

```
jnpr/syslog/router1/UI_COMMIT_COMPLETED {
    "_stamp": "2019-08-30T00:05:39.293342",
    "daemon": "mgd",
    "event": "UI_COMMIT_COMPLETED",
    "facility": 23,
    "hostip": "198.51.100.2",
    "hostname": "router1",
    "message": "commit complete",
    "pid": "60883",
    "priority": 188,
    "raw": "<188>Aug 29 17:05:57 router1 mgd[60883]: UI_COMMIT_COMPLETED: commit complete",
    "severity": 4,
    "timestamp": "2019-08-30 00:05:39"
```

In this example you first enable the Junos syslog engine on the Salt master and configure the device running Junos OS to send syslog messages to the Salt master server port 9999. After the configuration is applied, the device starts sending events to the Junos syslog engine, which publishes them on the Salt event bus.

You then create a reactor SLS file, **junos_backup_on_commit**, which applies the `junos_backup_config` state to the target identified by the value of the `hostname` key in the event data. The corresponding state file uses the `junos.rpc` function to execute the `get_config` RPC to retrieve the configuration in text format from the target device. It then saves the configuration on the proxy minion server at the file path specified by the `dest` parameter. The `dest` argument references the `id` grain and the current time to generate a unique filename that includes the device's ID and a timestamp.

You then configure the reactor by defining the `reactor` option in the Salt master configuration file, which associates the event tag with the reactor file. After restarting the Salt master to apply the configuration, the Salt Reactor system invokes the appropriate reactor file when the event occurs. Thus, when a commit complete event occurs on the device running Junos OS that is managed by the Salt master, the Junos syslog engine publishes the event on the event bus, and the Salt Reactor system invokes the reactor file.

Configuration

IN THIS SECTION

Enable the Junos Syslog Engine

Step-by-Step Procedure

To enable the Junos syslog engine:

1. Configure the Junos syslog engine in the Salt master configuration file.

```
engines:
  - junos_syslog:
      port: 9999
```

2. Restart the Salt master to apply the configuration, for example:

```
saltuser@salt-master~$ sudo killall salt-master
saltuser@salt-master~$ sudo salt-master -d
```

3. Configure the device running Junos OS to forward syslog events to the same port on the Salt master.

```
[edit]
saltuser@router1# set system syslog host 198.51.100.1 port 9999 any any
saltuser@router1# commit
```

Create the State file

Step-by-Step Procedure

To create the state file that generates a backup configuration file for the target node whenever the state is applied:

1. On the Salt master, create a new state file named **/srv/salt/junos_backup_config.sls**.

2. Define a state that retrieves the configuration and saves it to a uniquely named file.

```
saltuser@salt-master~$ cat /srv/salt/junos_backup_config.sls
{% set curtime = None | strftime("%Y-%m-%d-%H-%M-%S") %}
get_config:
  junos.rpc:
    - dest: /var/log/salt/backup-configs/{{ grains['id'] }}-config-{{ curtime }}.conf
    - format: text
```

NOTE: If the proxy ID and hostname are not identical, you can reference the hostname by substituting `{{ grains['junos_facts']['hostname'] }}` for `{{ grains['id'] }}`.

Create the Reactor File

Step-by-Step Procedure

To create the reactor SLS file:

1. On the Salt master, create a new reactor SLS file named **/srv/reactor/junos_backup_on_commit.sls**.

2. Configure the reactor file to apply the `junos_backup_config` state to the target.

   - If the proxy ID is identical to the device hostname, you can reference the hostname in the event data as the target.

```
saltuser@salt-master~$ cat /srv/reactor/junos_backup_on_commit.sls
Backup config for commit complete event:
  local.state.apply:
    - tgt: {{ data['hostname'] }}
    - arg:
        - junos_backup_config
```

- If the proxy ID is different from the device hostname, you can map the hostname to the proxy ID by referencing the value of the `junos_facts:hostname` grain.

```
saltuser@salt-master~$ cat /srv/reactor/junos_backup_on_commit.sls
Backup config for commit complete event:
  local.state.apply:
    - tgt: junos_facts:hostname:{{ data['hostname'] }}
    - tgt_type: grain
    - arg:
        - junos_backup_config
```

Configure the Reactor

Step-by-Step Procedure

To configure the reactor:

1. In the Salt master configuration file, configure the `reactor` option and specify the event and reactor file:

```
reactor:
  - 'jnpr/syslog/*/UI_COMMIT_COMPLETED':
    - /srv/reactor/junos_backup_on_commit.sls
```

2. Restart the Salt master to apply the configuration, for example:

```
saltuser@salt-master~$ sudo killall salt-master
saltuser@salt-master~$ sudo salt-master -d
```

Results

When you commit the configuration on the device running Junos OS, the Salt Reactor system matches on the event and calls the reactor. The reactor applies the state to the device, which retrieves and saves the configuration.

Verification

IN THIS SECTION

Verify the Salt Reactor Works

Purpose

Verify that the Salt reactor applies the `junos_backup_config` state and saves a backup configuration file to the proxy minion server when the device running Junos OS commits the configuration.

Action

After committing the configuration on the device running Junos OS, review the **/var/log/salt/backup-configs** directory on the proxy minion server to verify that the backup configuration file was generated.

```
saltuser@minion:~$ ls /var/log/salt/backup-configs
router1-config-2019-08-30-19-24-57.conf
```

```
saltuser@minion:~$ cat /var/log/salt/backup-configs/router1-config-2019-08-30-19-24-57.conf
## Last commit: 2019-08-15 09:56:10 PDT by saltuser
version 18.4R1.8;
groups {
    re0 {
        system {
            host-name router1;
        }
...
```

Meaning

The configuration is saved in text format to the specified file on the proxy minion server.

RELATED DOCUMENTATION

Junos Syslog Engine for Salt | **130**

Understanding the Salt for Junos OS Components | **9**

6

CHAPTER

Troubleshoot Salt for Junos OS

IN THIS CHAPTER

Troubleshoot Connection Errors When Using Salt to Manage Devices Running Junos OS

IN THIS SECTION

Problem

Description

After starting the proxy minion process for a device running Junos OS and accepting the key on the Salt master, the Junos proxy (for Salt) fails to connect to the device, and the proxy log file on the proxy minion server includes an error regarding missing `init()` or `shutdown()` functions. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' test.ping
router1:
    Minion did not return. [No response]
```

```
saltuser@minion:~/.ssh$ sudo cat /var/log/salt/proxy
...
KeyError: 'junos.initialized'
2019-06-21 22:23:07,044 [salt.minion                                          :3414][ERROR   ]
[31048] Proxymodule junos is missing an init() or a shutdown() or both. Check your proxymodule.
Salt-proxy aborted.
2019-06-21 22:23:07,044 [salt.cli.daemons                                     :533 ][ERROR   ]
[31048] Proxymodule junos is missing an init() or a shutdown() or both. Check your proxymodule.
Salt-proxy aborted.
```

Cause

The Junos proxy minion server must have the Juniper Network’s Junos PyEZ library (`junos-eznc`) and `jxmlease` Python module as well as the `yamlordereddictloader` Python module installed on the server in order to use Salt to manage devices running Junos OS.

Solution

Install Junos PyEZ and the `jxmlease` and `yamlordereddictloader` Python modules on the Junos proxy minion server under the same version of Python that Salt is installed. For example:

```
saltuser@minion:~$ sudo pip3 install junos-eznc jxmlease yamlordereddictloader
```

After the dependencies are installed, start the proxy minion process for that device on the proxy minion server and accept the new Salt key on the Salt master.

```
saltuser@minion:~$ sudo salt-proxy --proxyid=router1 -d
```

```
saltuser@salt-master:~$ sudo salt-key -a router1
The following keys are going to be accepted:
Unaccepted Keys:
router1
Proceed? [n/Y] y
Key for minion router1 accepted.
```

RELATED DOCUMENTATION

How to Install Salt for Junos OS

Troubleshoot Authentication Errors When Using Salt to Manage Devices Running Junos OS

IN THIS SECTION

Problem

Description

After starting the proxy minion process for a device running Junos OS and accepting the key on the Salt master, the Junos proxy (for Salt) fails to connect to the device, and the proxy log file on the proxy minion server includes an error regarding failed authentication. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' test.ping
router1:
    Minion did not return. [No response]
```

```
saltuser@minion:~/.ssh$ sudo cat /var/log/salt/proxy
...
  File "/usr/lib/python3/dist-packages/salt/minion.py", line 3420, in _post_master_init
    proxy_init_fn(self.opts)
  File "/usr/lib/python3/dist-packages/salt/proxy/junos.py", line 109, in init
    thisproxy['conn'].open()
  File "/usr/local/lib/python3.6/dist-packages/jnpr/junos/device.py", line 1268, in open
    raise EzErrors.ConnectAuthError(self)
jnpr.junos.exception.ConnectAuthError: ConnectAuthError(router1.example.com)
```

Cause

The Salt user might fail to authenticate with a device running Junos OS 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 in the pillar file that defines the proxy configuration for that device.
- The user has an account and authenticates with the device running Junos OS using SSH keys, but the SSH keys are inaccessible on either the device or the proxy minion server.
- The user's SSH configuration file on the proxy minion server, which is automatically queried when the Junos proxy attempts to establish the connection, defines incorrect settings for authenticating with that device.

NOTE: If you do not specify a user in the pillar file that defines the proxy information or in an SSH configuration file, the user defaults to the current user.

Solution

Verify the following configuration items to ensure that the user can authenticate with the managed device:

- The Salt user authenticating with the device running Junos OS has a login account on the device and that a text-based password or SSH public key is configured for the account.
- If SSH keys are configured, verify that the user can access them on the proxy minion server and can successfully connect to the device using the keys.
- The user's SSH configuration file does not contain any settings for that device that will cause the connection to fail, for example, a different username or SSH key file.
- The correct parameters are supplied for the device's proxy configuration on the Salt master, for example:

```
# /srv/pillar/router1-proxy.sls
proxy:
  proxytype: junos
  host: router1.example.com
  username: saltuser
  password: lab123   # SSH password or SSH key file password
  ssh_private_key_file: /home/saltuser/.ssh/id_rsa_salt   # non-default SSH key location on
proxy minion server
  port: 830
```

If you update the pillar file containing the proxy configuration for a given device, you might need to restart the proxy minion process for that device on the proxy minion server and accept the new Salt key on the Salt master for the changes to take effect.

RELATED DOCUMENTATION

How to Authenticate Salt for Junos OS Users

Troubleshoot Junos Execution Module Errors When Using Salt to Manage Devices Running Junos OS

IN THIS SECTION

Problem

Description

After starting the proxy minion process for a device running Junos OS and accepting the key on the Salt master, the Junos proxy (for Salt) connects to the device, but when you execute a function from the Junos execution module, it returns an error. For example:

```
saltuser@salt-master:~$ sudo salt 'router1' test.ping
router1:
    True
```

```
saltuser@salt-master:~$ sudo salt 'router1' junos.cli 'show version'
router1:
    'junos' __virtual__ returned False: The junos or dependent module could not be loaded: junos-
eznc or jxmlease or yamlordereddictloader or proxy could not be loaded.
```

Cause

The Junos proxy minion server must have the Juniper Network's Junos PyEZ library (`junos-eznc`) and `jxmlease` Python module as well as the `yamlordereddictloader` Python module installed on the server in order to use Salt to manage devices running Junos OS.

Solution

Install Junos PyEZ and the `jxmlease` and `yamlordereddictloader` Python modules on the Junos proxy minion server under the same version of Python that Salt is installed. For example:

```
saltuser@minion:~$ sudo pip3 install junos-eznc jxmlease yamlordereddictloader
```

After the dependencies are installed, restart the proxy minion process for that device on the proxy minion server and accept the new Salt key on the Salt master.

RELATED DOCUMENTATION

How to Install Salt for Junos OS

Understanding the Junos Execution and State Salt Modules | **20**