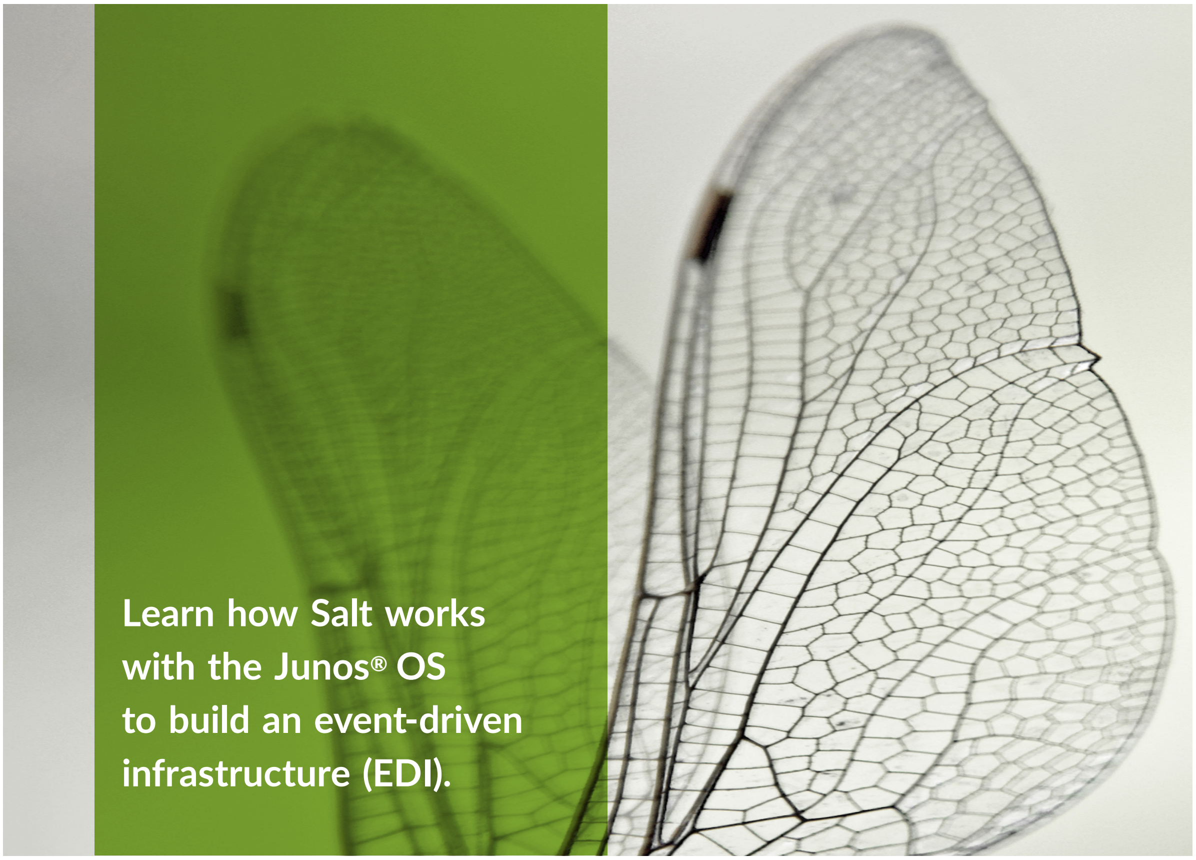
Juniper Networks | Engineering Simplicity

DAY ONE: AUTOMATING JUNOS WITH SALT



By Peter Klimai

DAY ONE: AUTOMATING JUNOS WITH SALT

Network automation is a hot topic. Currently there are several popular configuration management systems that can be used for networking – Puppet, Chef, Ansible, etc. Salt is one of them, but it is different because it is event-driven and allows users to build event-driven infrastructure (EDI). *Day One: Automating Junos with Salt* is a complete book for Junos® OS network engineers unfamiliar with programming, templating, or data structures. From fundamental concepts to working tutorials, this *Day One* book will have you working with Salt in no time at all.

"Peter Klimai has written an engaging crash course about using Salt to manage Junos devices. If you are using Salt and want to include Junos devices in your Salt operations, read this book. If you want to build an event-driven automation environment around your Junos devices, read this book."

Sean Sawtell, Senior Network Engineer, Juniper Networks,
author of Day One: Automating Junos with Ansible, 2nd Ed.

"The quality is top-notch. It is well-written and concise without leaving anything important out. Even though I had never done anything in Salt, I was able to write a few states for our network in a day or two."

Said van de Klundert, Network Engineer for IBM Cloud,
Juniper Ambassador, JNCIE-SP, JNCIE-DC

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

- Install Salt and its basic settings to make it work with the Junos OS.
- Execute Junos commands remotely.
- Provision Junos device configurations with Salt.
- Integrate Junos devices in Salt's event-driven infrastructure.
- Create custom Salt modules with Junos PyEZ.
- Perform network verifications with Salt.
- Use NAPALM Salt modules to manage multivendor networks.

ISBN 978-1-941441-75-6
53500
9 781941 441756

DAY ONE: AUTOMATING JUNOS WITH SALT

Klimai

Day One: Automating Junos® with Salt

by Peter Klimai

Published by Juniper Networks Books
Author: Peter Klimai
Technical Reviewers: Khelil Sator, Nitin Kr, Dwarakanath Yadavalli, Piyush Rai, Diogo Montagner, Sean Sawtell, Said van de Klundert, Clay Haynes
Editor in Chief: Patrick Ames
Copyeditor: Nancy Koerbel
Illustrator: Karen Joice

ISBN: 978-1-941441-75-6 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-941441-76-3 (ebook)

Version History: v1, October 2018
2 3 4 5 6 7 8 9 10

http://www.juniper.net/dayone

About the Author
Dr. Peter Klimai is a Juniper Ambassador working as Lead Engineer and Instructor at Poplar Systems (Juniper partner in Russia since 2001). He is certified JNCIE-SEC #98, JNCIE-ENT #393, JNCIE-SP #2253, JNCIP-DC, and JNCI, and has several years of experience supporting Juniper equipment for small and large companies. Peter teaches Juniper classes on routing, security, automation and troubleshooting and is especially enthusiastic about network automation using various tools, as well as network function virtualization.

Author's Acknowledgments
While I was working on this book, many people helped me with advice, reviews, and pushing me in the right direction – without them this book would have been impossible to complete. I would especially like to thank Patrick Ames, Julie Wider, Nitin Kr, Khelil Sator, Diogo Montagner, Dwarakanath Yadavalli, Leonid Mirenkov, Said Klundert, Sean Sawtell, Alex Tarkhov, and Clay Haynes.

Feedback? Comments? Error reports? Email them to dayone@juniper.net.

Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Networks Books.

Day One books cover the Junos OS and Juniper Networks networking essentials with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow. You can obtain the books from various sources:

- Download a free PDF edition at http://www.juniper.net/dayone.
- Many of the library's books are available on the Juniper app: Junos Genius.
- Get the ebook edition for iPhones and iPads from the iBooks Store. Search for *Juniper Networks Books* or the title of this book.
- Get the ebook edition for any device that runs the Kindle app (Android, Kindle, iPad, PC, or Mac) by opening your device's Kindle app and going to the Amazon Kindle Store. Search for *Juniper Networks Books* or the title of this book.
- Purchase the paper edition at Vervante Corporation (www.vervante.com) for between $15-$40, depending on page length.
- Note that most mobile devices can also view PDF files.

Target Audience

This book is written for network administrators and network engineers who are starting to build and use network automation to make their jobs easier, and is focused on how to use the SaltStack (Salt) automation platform to configure and manage Junos-based devices. However, once you've learned how to use Salt, you can also leverage that knowledge to automate the administration of servers or network gear from other vendors.

This Book's GitHub Site

Go to: https://github.com/pklimai/day-one-junos-salt.

What You Need to Know Before Reading This Book

The author has made a few assumptions while writing this book:

- You are a network engineer having some experience with managing computer networks.
- You understand typical network management tasks and willing to see possible benefits of their automation.
- You are familiar with Junos OS CLI.
- You have basic understanding of programming languages.
- This book is based on Salt 2018.3 (Oxygen) version. It generally applies to all Junos-based platforms running Junos® OS version 11.4 or later.

What You Will Learn by Reading This Book

This book will help you get started automating Junos devices with Salt, the powerful configuration management and remote execution software platform from SaltStack. You will learn relevant Salt concepts by practicing them, and we will go step-by-step, to practice the following:

- Salt installation and basic settings to make it work with Junos.
- Executing Junos commands remotely.
- Provisioning Junos device configurations with Salt.
- Integrating Junos devices in Salt's event-driven infrastructure.
- Creating custom Salt modules with Junos PyEZ.
- Performing network verifications with Salt.
- Using NAPALM Salt modules to manage multivendor networks.

Chapter 1

Introduction to Junos Automation with Salt

This chapter briefly reviews Junos and Junos Automation solutions, discusses network automation a bit more generally, and then introduces the Salt platform. Use the links at the end of each section to get the most recent news and updates.

Junos Automation

Junos is a powerful network operating system (OS) that was first released in 1998 and has gained the love and respect of network engineers all over the world ever since.

Among the many reasons Junos is so pleasant to work with are its consistent features among different physical and virtual platforms, stability, high performance, and its convenient and powerful CLI. And although many of us first fell in love with Junos after looking at its extremely appealing CLI, it is important to remember that a network OS is not the same as its CLI – it is much more.

In fact, there are many reasons why performing day-to-day network management tasks using device CLI is not that good of an idea:

- Monitoring by executing a series of CLI commands on multiple devices and looking at their output is time consuming and not scalable. Clearly, the task can, and should, be automated.
- Provisioning configuration by typing in set-commands is time consuming and does not guarantee consistency – this leads to future problems with troubleshooting, migrations, etc.

- Reacting to events manually, as in an emergency, is also not scalable, and assumes you will be called at 4 a.m. just because you are the person who was able to troubleshoot a similar issue last time.

So you decide to automate both operational and configuration tasks on your Junos network devices, be they physical or virtual. To do so, you need to choose a proper toolkit, and, generally, to decide how far you are willing to go.

Let's briefly review the ways you can perform Junos automation. As Junos automation has a long history, there are multiple options with a use case for each option.

Here is an *incomplete* list of features that were introduced, over time, to automate Junos:

- You can execute scripts written in XSLT, SLAX, or Python programming languages "on-box", that is, on the device itself. Junos supports operational scripts (they work as custom op-mode commands), event scripts (reacting to different events), and commit scripts (customizing configuration processing), as well as SNMP scripts (scripts executed by calling custom SNMP OID). On-box scripts can do a lot of things but they are tied to a single device – so they allow you to automate, but, generally, not to orchestrate tasks on multiple devices.
- You can connect to a Junos device using the NETCONF protocol, which is an XML-based protocol typically using SSH transport. Then, you can execute remote procedure calls (RPCs), which are analogs of Junos CLI commands, remotely, to query operational state and change configuration as needed. NETCONF is currently standardized in RFC 6241 and supported by multiple vendors (although initially NETCONF appeared as improved version of Juniper's JUNOScript protocol). There are multiple libraries for various programming languages that implement the NETCONF protocol.
- If you can make use of the Junos PyEZ library by using Python programming language, it significantly simplifies your life by abstracting out details of working with NETCONF sessions (for example, opening and closing the connection, forming XML RPC documents, and parsing responses, etc.). With the PyEZ library and the flexibility of Python, you can approach virtually any automation task – except, perhaps, for some extreme cases (see the bullet on JET below), but that will require some Python coding, of course.

NOTE Multiple PyEZ use cases are covered in *Day One: Junos PyEZ Cookbook*, https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/junos-pyez-cookbook/.

- Junos also supports REST API, which is a stateless HTTP(S)-based management interface that you can use as an alternative to NETCONF, if you prefer. REST API is currently available for M, MX, T, PTX, and SRX Series Junos-based platforms.
- Junos Extension Toolkit (JET) is additional API available to program Junos control plane. Generally, JET is rather low-level and gives you access to the APIs of internal Junos daemons. With JET, you can modify device configuration hundreds of times a second, as well as doing other mind-blowing things; shooting yourself in the foot is also possible. JET is beyond the scope of this book.
- Junos Advanced Forwarding Toolkit (AFT) provides the packet forwarding engine (PFE) APIs that can be used by the control plane. This is a rather new feature, please see review here: https://forums.juniper.net/t5/Industry-Solutions-and-Trends/Juniper-Forwarding-Interface/ba-p/310823
- Junos supports integration with automation management systems including Ansible, Puppet, Chef, and Salt. Initially these systems were widely used for managing the compute (server) infrastructure, and later they were successfully adapted to managing network equipment. Generally, use of automation management systems allows automation without reinventing the wheel (having to code everything yourself); for more complex tasks not foreseen by the developers, you can still code additional modules as needed. As you should already know, this book covers Salt, or the SaltStack system, which has some unique features reviewed in the next sections.

NOTE *Day One: Automating Junos with Ansible, 2nd Edition* available at https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/automating-junos-ansible/, covers integration of Ansible with Junos in many details.

Network Automation

Now let’s briefly discuss how much automation you want. This is not an official classification, but for the purpose of this chapter, I suggest you consider three automation “stages”:

- *Automate some things* – at this level, you start creating simple scripts to automate basic tasks; you also might be using an automation management system but your workflows are not yet fully integrated –you still do a lot of work manually.
- *Automate most things* – this is the level where you apply DevOps or Network Reliability Engineering (NRE) practices, with all changes going through a review process, version control system, automated testing, and generally a Continuous Deployment (or DevNetOps) pipeline.

NOTE You can learn more on NRE and DevNetOps at the following URLs: https://www.juniper.net/us/en/products-services/what-is/nre/ and https://www.juniper.net/us/en/products-services/what-is/devnetops/ .

- *Automate all things* – ideally, you automate your network up to a level where it is almost completely self-driving (https://www.juniper.net/us/en/products-services/what-is/self-driving-network/). This is not to say that in the end you will just be sitting back and looking at your finished network. Similar to what happens in software development industry, where there is never a final software version – the network automation system itself will need to be regularly fine-tuned, upgraded, and extended, which guarantees you never stay idle.

Using the above classification, in this introductory book we'll be somewhere between Levels 1 and 2. The idea is to start simple and gradually increase the coverage of things that are automated: monitoring, troubleshooting, configuration deployment, migrations, the list goes on.

NOTE For an alternative five-step classification of network automation stages, please consult: https://www.juniper.net/us/en/solutions/automation/the-automation-journey/.

Salt

Salt (https://saltstack.com/) is one of the most powerful, scalable, and flexible platforms that allows you to automate key operational and configuration tasks in your network. Salt is open source and it comes packaged with modules supporting Junos OS right out of the box.

One special advantage of Salt, compared to most other automation management systems, is its native support for event-driven infrastructure (EDI). With Salt, you can automatically react on certain events (such as Junos events) in certain ways, as you see fit.

This book covers Salt installation, basic architecture, and the configuration that must be performed to make Salt and Junos work together. It also addresses monitoring and configuration provisioning, and some EDI examples are discussed.

MORE? This book is intended to get you started with Salt for Junos. SaltStack has extensive documentation available at https://docs.saltstack.com, and many additional resources are cited in the *Appendix* at the end of this book.

Chapter 2

Basic Salt Architecture and Installation

This chapter explains the basics of Salt architecture and shows you how to install Salt. The terminology used by Salt, the basics of its architecture, and how it applies to managing Junos devices, are explained in-line with practical instructions telling you how to perform each task.

Salt Installation

Let's start with some of the terminology involved: *Salt Master* is Salt's main control server. *Salt Minion* is a system managed by Salt. Salt typically uses agent-based architecture, so devices managed by Salt need to run the Salt Minion process. As you will see in Chapter 3, devices that can't run the minion process for some reason can still be managed by using proxy minions.

NOTE Generally, Salt is very flexible and every component is customizable and replaceable. Salt setup may differ greatly depending on the use case – for example, a masterless setup is possible. Additionally, the `salt-ssh` package allows Salt to work in *agentless* mode (no minion process required). This option is not currently used for Junos device management, so it will not be discussed here.

Setting up Salt and configuring it to work with Junos is quite easy if you follow the steps in this chapter.

The initial lab setup used here is shown in Figure 2.1. The setup includes a Salt Master server (`master`) and a Salt Minion server (`minion1`). Also shown is a schematic of Salt's *Event Bus* using (by default) ZeroMQ distributed messaging software for all communication between master and minions. All events that happen in the system are published onto the event bus and the subscribers listen for published events, reacting accordingly. Based on this, event-driven infrastructure (EDI) can be built, as discussed in later chapters.

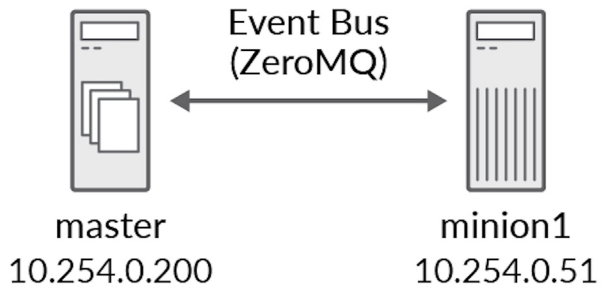


Figure 2.1 Basic Salt Setup Diagram Used in This Chapter

There are two Linux Servers in the initial setup of Figure 2.1, specifically running Ubuntu 16.04.4 LTS, (`master` and `minion1`). Other **nix* flavors should work as well. In our setup, servers run as virtual machines (VMs) using VMware ESXi hypervisor but you can use any other option, such as KVM or VirtualBox, or public cloud-based deployment.

At this point let's assume you have both Linux servers (VMs) ready for Salt installation. When provisioning Linux, it is enough to install standard file system utilities and OpenSSH server.

The easiest way to set up Salt, then, is through a bootstrap script (https://docs.saltstack.com/en/latest/topics/tutorials/salt_bootstrap.html). To install Salt on the `master` server, issue the following commands:

```
lab@master:~$ curl -o bootstrap_salt.sh -L https://bootstrap.saltstack.com
lab@master:~$ sudo sh bootstrap_salt.sh -M
```

Output is omitted for brevity. The installation will take a minute or so. When it completes, you can check the Salt version by using the following command:

```
lab@master:~$ salt --version
salt 2018.3.2 (Oxygen)
```

On the `minion1` server, install Salt by performing similar steps, but for this time do not use the `-M` key:

```
lab@minion1:~$ curl -o bootstrap_salt.sh -L https://bootstrap.saltstack.com
lab@minion1:~$ sudo sh bootstrap_salt.sh

... OMITTED ...
```

```
lab@minion1:~$ salt-minion --version
salt-minion 2018.3.2 (Oxygen)
```

NOTE By default, the bootstrap script installs the latest stable Salt version, so it may differ from the 2018.3.2 that is used for this chapter. This will typically not be a problem. However, you can enforce the installation of a specific Salt version, if you wish, by modifying the second command as follows: `sudo sh bootstrap_salt.sh -M git v2018.3.2` (this is for master; for minion, omit the `-M` key).

Performing Basic Salt Configuration and Verification

There's one thing you definitely want to configure on the minion: that's to tell it where the master is (the default is actually to look for host named "salt", which does not readily meet the needs of our lab setup).

So, start configuring the minion process on the `minion1` server by editing the minion configuration file:

```
lab@minion1:~$ sudo vi /etc/salt/minion
```

NOTE Although the examples show you using the `vi` text editor, any text editor of your choice will work.

Add the following line (in bold) to the file:

```
...
# Set the location of the salt master server. If the master server cannot be
# resolved, then the minion will fail to start.
#master: salt
master: 10.254.0.200
...
```

Here, the IP address is the `master` server's address in the lab setup. Replace it with the address in your lab. Remember that lines starting with a hash are treated as comments. Do not forget to save the changes.

Finally, restart the `salt-minion` process so it re-reads the configuration:

```
lab@minion1:~$ sudo service salt-minion restart
```

Now, switch to the open terminal session with the master and issue the following command to view the minion's public key status:

```
lab@master:~$ sudo salt-key --list-all
Accepted Keys:
Denied Keys:
Unaccepted Keys:
minion1.edu.example.com
Rejected Keys:
```

Note the ID of the minion with an unaccepted key. The security system of Salt will not allow communication until you accept minion's key on `master` – so let's do that now:

```
lab@master:~$ sudo salt-key --accept=minion1.edu.example.com
The following keys are going to be accepted:
Unaccepted Keys:
minion1.edu.example.com
Proceed? [n/Y] y
Key for minion minion1.edu.example.com accepted.
```

Now let's execute some of the Salt commands, using its remote execution capabilities. Note we are testing basic Salt features here – no Junos at all is involved at this point.

First, let's ping our minion:

```
lab@master:~$ sudo salt '*' test.ping
minion1.edu.example.com:
  True
```

Here, you called the `test.ping` *execution function* that is used to make sure the minion is up and responding. The communication happens over Salt's ZeroMQ message bus (so this is not an ICMP ping). The argument `'*'` means that you want to execute on all minions, it just happens that, so far, we only have one.

Generally speaking, various Salt *execution modules* allow you to perform (execute) some specific tasks on minions: `test` is just one of such modules.

Let's now try the `cmd.run` function – it allows the running of arbitrary commands on minions. Let's check minion's Python version, just for an example:

```
lab@master:~$ sudo salt minion* cmd.run 'python -V'
minion1.edu.example.com:
  Python 2.7.12
```

You can run any other commands on the minions the same way. Check out the full Salt module index at https://docs.saltstack.com/en/latest/salt-modindex.html for more examples. In Chapter 3, you will add Junos devices to the setup.

Chapter 3

Using Junos Proxy Minions

This chapter explains how Salt makes use of proxy minions to work with Junos devices.

Adding Proxy Minions to Manage Junos Devices

Chapter 2 explained how to start managing a Linux server with Salt. However, managing Junos devices with Salt is a bit different. You do not run a Salt minion on-box with Junos devices; instead, a *proxy minion* process is used. The proxy minion process may run either on the Salt master server or on some other server. The latter option allows for load distribution in scaled setups and is a bit easier to understand for educational purposes, so let's use it here: proxy minions will run on the `minion1` server (see Figure 3.1, which shows an extension of the setup used previously in Figure 2.1).

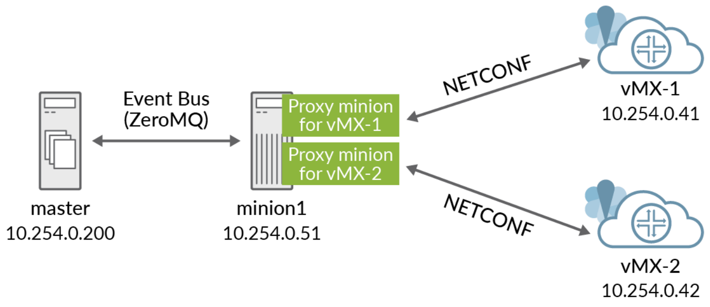


Figure 3.1 *Salt Setup for Managing Two Junos Devices*

You can see from Figure 3.1 that we added two Juniper vMX devices to the setup, named `vMX-1` and `vMX-2`. Their management IP addresses configured on `fxp0` interfaces are also shown.

NOTE *Day One: vMX Up and Running* by Matt Dinham explains the architecture and installation of vMX for KVM hypervisor: https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/vmx-up-running/. The following blog post by Clay Haynes shows how to run the vMX on VMware Fusion: https://alostrealist.com/2018/04/16/running-the-vmx-on-vmware-fusion/.

NOTE This book will not use any vMX-specific functionality, so if you replace the vMXs with any other Junos devices, such as physical MX/EX/SRX/QFX Series, or a vSRX, everything will typically work the same way for the examples discussed here.

As a prerequisite step, NETCONF over SSH must be enabled on both Junos devices, as follows (this shows configuration for one device – make sure you perform this on the other one as well):

```
lab@vMX-1> configure
Entering configuration mode

[edit]
lab@vMX-1# set system services netconf ssh

[edit]
lab@vMX-1# commit
commit complete
```

You can also see from Figure 3.1 that the minion server (`minion1`) will now also run two Junos proxy minions, one for each of the two vMX devices in the topology. Proxy minions are just software processes (daemons) used to manage, in this case, a networking device. You need one proxy process per device, and for Junos one such process requires about 100MB of RAM, so plan your system accordingly.

Refer again to Figure 3.1. From one “side” the proxy minion connects to Salt master using the ZeroMQ bus, while from the other “side” it is connected to the Junos device using NETCONF protocol (Junos PyEZ library is used under the hood).

To start configuring Salt proxy, edit the `/etc/salt/proxy` file on the `minion1` server:

```
lab@minion1:~$ sudo vi /etc/salt/proxy
```

Add the `master` setting to it, indicating where the Salt master is:

```
# Set the location of the salt master server. If the master server cannot be
# resolved, then the minion will fail to start.
#master: salt
master: 10.254.0.200
```

Before proceeding, it's time to get familiar with one more Salt concept: *pillar*. The Salt's pillar system provides various data associated with minions. In the simplest case, pillar files will be YAML files with defined variable values, but pillar data can also be stored in a database such as SQL, obtained via REST API from some external system, etc.

The location where pillar files are stored can vary. By default, it is in the `/srv/pillar` directory of the master server (this is defined by `pillar_roots` parameter in the `/etc/salt/master` configuration file on the Salt master). Let's just use the default directory – to do so, you will have to create it first:

```
lab@master:~$ sudo mkdir /srv/pillar
```

In this directory, create the `/srv/pillar/proxy-1.sls` file with the following content (just replace host IP, username, and password with values matching your setup):

```
lab@master:~$ cat /srv/pillar/proxy-1.sls
proxy:
  proxytype: junos
  host: 10.254.0.41
  username: lab
  password: lab123
  port: 830
```

NOTE Salt certainly has ways to better secure your passwords, but that is beyond the scope of this chapter. Please consult the following URLs for details: https://docs.saltstack.com/en/latest/topics/best_practices.html#storing-secure-data and https://docs.saltstack.com/en/latest/ref/renderers/all/salt.renderers.gpg.html

Similarly, create the `/srv/pillar/proxy-2.sls` file:

```
lab@master:~$ cat /srv/pillar/proxy-2.sls
proxy:
  proxytype: junos
  host: 10.254.0.42
  username: lab
  password: lab123
  port: 830
```

The two files that you just created essentially contain some mappings (pairs of keys and corresponding values). For example, the key `username` maps to value `lab`, etc. The key `proxy` has a nested mapping as a value, which is shown by indentation. The format that you just used for these files is YAML (http://yaml.org/), while the file extension is SLS (SaLt State).

Generally, SLS files can be in various formats: in the simplest case it is YAML, or it can be YAML+Jinja (where Jinja is a templates format – see http://jinja.pocoo.org/), or something else if properly customized (maybe even Python code – Salt is very flexible). The next chapters will contain some additional examples and explanations of the YAML+Jinja syntax.

Now let's create the *pillar top* file. This file will define which minions have access to which pillar data. In our case, the content will be as follows:

```
lab@master:~$ cat /srv/pillar/top.sls
base:
  'vMX-1':
    - proxy-1
  'vMX-2':
    - proxy-2
```

Here, `base` is the name of what is called *environment* in Salt. For example, you can have testing/staging/production environments – here we will just use the default `base` environment. Note also that the .sls extension for the proxy-1.sls and proxy-2.sls file names must be omitted.

Now it's time to perform settings on the minion side. Switch to the `minion1` server. Remember, this server will host two Junos proxy minion processes. For Junos proxy to successfully communicate with Junos devices, a couple of Python packages are needed – namely, `Junos PyEZ` and `jxmlease` (and their dependencies as well). To install those libraries, first install the Python PIP tool, and then the packages themselves (output omitted for brevity):

```
lab@minion1:~$ sudo apt-get install python-pip

lab@minion1:~$ sudo pip install junos-eznc

lab@minion1:~$ sudo python -m easy_install --upgrade pyOpenSSL

lab@minion1:~$ sudo pip install jxmlease
```

NOTE Upgrade the `pyopenssl` package *before* installing `jxmlease`. The explicit upgrade is used here as a workaround, otherwise you may see an error message like this: `AttributeError: 'module' object has no attribute 'SSL_ST_INIT'`.

Okay, it's time to launch the Junos Salt proxy processes:

```
lab@minion1:~$ sudo salt-proxy --proxyid=vMX-1 -d
lab@minion1:~$ sudo salt-proxy --proxyid=vMX-2 -d
```

NOTE The `-d` option makes `salt-proxy` run in the daemon mode. If you want the program to just run in the terminal window, omit that key. In this case you will be able to see some of the salt-proxy log messages in real time. You can modify the level of logging using `-l` key, for example: `sudo salt-proxy --proxyid=vMX-1 -l debug`. This approach can be useful for troubleshooting.

And, on the master, accept the minion keys, just as you did before:

```
lab@master:~$ sudo salt-key -a vMX-1
The following keys are going to be accepted:
Unaccepted Keys:
vMX-1
Proceed? [n/Y] y
Key for minion vMX-1 accepted.
lab@master:~$ sudo salt-key -a vMX-2
The following keys are going to be accepted:
Unaccepted Keys:
vMX-2
Proceed? [n/Y] y
Key for minion vMX-2 accepted.
```

How many minions do you think you have now? Let's check with `test.ping`:

```
lab@master:~$ sudo salt '*' test.ping
minion1.edu.example.com:
    True
vMX-1:
    True
vMX-2:
    True
```

So, you have two more minions (proxies) in addition to `minion1`. In Chapter 4, you will continue using the setup and see how to execute Junos commands using Salt.

Chapter 4

Executing Junos Commands with Salt

In this chapter, you will see how various Junos commands can be executed using Salt's Junos execution module.

Junos Execution Module Overview

The Salt execution module for Junos, named `salt.modules.junos`, includes several useful execution functions that allow you to perform different tasks on a Junos device. This includes executing arbitrary CLI commands, loading and committing the configuration, installing software, and more.

NOTE This chapter demonstrates a few examples of how the execution functions can be used. The complete documentation for the Junos execution module, listing all supported functions, and their parameters, is available at https://docs.saltstack.com/en/latest/ref/modules/all/salt.modules.junos.html.

The next sections of this chapter provide some examples on how you can use the Junos execution module's capabilities.

Collecting and Printing Device Facts

You should remember that Salt execution modules contain execution functions that you can reference using the `<module>.<function>` syntax.

Let's first use the `junos.facts` execution function to collect a basic information bundle from our managed vMX devices:

```
lab@master:~$ sudo salt vMX* junos.facts
vMX-2:
    ----------
    facts:
        ----------
        2RE:
            False
        HOME:
            /var/home/lab
        RE0:
            ----------
            last_reboot_reason:
                Router rebooted after a normal shutdown.
...
vMX-1:
    ----------
    facts:
        ----------
        2RE:
            False
        HOME:
            /var/home/lab
        RE0:
            ----------
            last_reboot_reason:
                Router rebooted after a normal shutdown.
...
```

Even when output is abbreviated, you can see a bunch of information is collected and displayed for each device. Needless to say, the tasks on different devices are performed in parallel by Salt.

It's helpful to note that the facts collected by `junos.facts` are stored in Salt grains. *Grains* generally contain data about the managed system, and you can use that data in various places while working with Salt. For example, you can use grains for filtering minions when running execution functions, as in the following command:

```
lab@master:~$ sudo salt -G 'os_family:junos' junos.facts
```

The output was intentionally omitted – it is the same as the previous example. But in this example, the CLI command was only executed on devices that had the `os_family` grain equal to `junos`, namely `vMX-1` and `vMX-2` (without that filter Salt would also try to execute the `junos.facts` module on `minion1` – that operation would be unsuccessful).

Executing CLI Commands

The `junos.cli` execution function invokes the Junos CLI commands and returns the output in the specified format (default is text, but you can request XML output as well).

For example:

```
lab@master:~$ sudo salt -G 'os_family:junos' junos.cli "show interfaces fxp0.0 terse"
vMX-2:
    ----------
    message:

        Interface               Admin Link Proto    Local                 Remote
        fxp0.0                  up    up   inet     10.254.0.42/24
    out:
        True
vMX-1:
    ----------
    message:

        Interface               Admin Link Proto    Local                 Remote
        fxp0.0                  up    up   inet     10.254.0.41/24
    out:
        True
```

Executing Junos RPCs

You can use the `junos.rpc` module to execute Junos *remote procedure calls* (RPCs). Don't worry if you are unaware of the RPC term – this is just an official name for Junos commands when executed using the application programming interface (API). In particular, this happens every time you are working via the NETCONF protocol. Remember, Junos proxy minion connects to a Junos device using NETCONF.

Most Junos commands have associated RPCs, and you can easily figure out the RPC corresponding to a command using the `| display xml rpc` command modifier, for example in the `show interfaces terse fxp0` command:

```
lab@vMX-1> show interfaces terse fxp0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
    <rpc>
        <get-interface-information>
                <terse/>
                <interface-name>fxp0</interface-name>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Now that you know the RPC, which is contained inside the `<rpc>` tags in the above output, you can execute it using the `junos.rpc` module:

```
lab@master:~$ sudo salt vMX* junos.rpc get-interface-information interface_name
='fxp0' terse=True --out=json
{
    "vMX-2": {
        "rpc_reply": {
            "interface-information": {
                "physical-interface": {
                    "oper-status": "up",
                    "logical-interface": {
                        "oper-status": "up",
                        "address-family": {
                            "address-family-name": "inet",
                            "interface-address": {
                                "ifa-local": "10.254.0.42/24"
                            }
                        },
                        "admin-status": "up",
                        "name": "fxp0.0",
                        "filter-information": ""
                    },
                    "admin-status": "up",
                    "name": "fxp0"
                }
            }
        },
        "out": true
    }
}
{
    "vMX-1": {
        "rpc_reply": {
            "interface-information": {
                "physical-interface": {
                    "oper-status": "up",
                    "logical-interface": {
                        "oper-status": "up",
                        "address-family": {
                            "address-family-name": "inet",
                            "interface-address": {
                                "ifa-local": "10.254.0.41/24"
                            }
                        },
                        "admin-status": "up",
                        "name": "fxp0.0",
                        "filter-information": ""
                    },
                    "admin-status": "up",
                    "name": "fxp0"
                }
            }
        },
        "out": true
    }
}
```

Note that the XML RPC parameters (the nested `<terse>` and `<interface-name>` elements in the original RPC request) must be transformed to command line arguments, as shown. Because the `terse` parameter does not have a value, you just assign a value of `True` to it (a convention used by the Junos PyEZ library that is working under the hood).

You have also instructed Salt to provide output as JSON using the `--out` key. The supported output formats (processed by *outputters*, or output modules of Salt) include `highstate`, `json`, `pprint`, `raw`, `txt`, `yaml`, and more.

Pinging Hosts

You can perform basic reachability checks using the `junos.ping` execution function, as follows:

```
lab@master:~$ sudo salt vMX* junos.ping "10.254.0.1" count=2
vMX-2:
    ----------
    message:
...
            probe-results-summary:
                ----------
                packet-loss:
                    0
                probes-sent:
                    2
                responses-received:
                    2
                rtt-average:
                    483
                rtt-maximum:
                    494
                rtt-minimum:
                    472
                rtt-stddev:
                    11
            target-host:
                10.254.0.1
            target-ip:
                10.254.0.1
    out:
        True
vMX-1:
    ----------
    message:
...
            probe-results-summary:
                ----------
                packet-loss:
                    0
                probes-sent:
                    2
                responses-received:
                    2
                rtt-average:
                    368
                rtt-maximum:
                    434
                rtt-minimum:
                    303
                rtt-stddev:
```

```
                65
        target-host:
            10.254.0.1
        target-ip:
            10.254.0.1
    out:
        True
```

From the output (which was abbreviated to save space), you can see that the ICMP ping from a Junos device to a target host, in this case 10.254.0.1, worked fine.

Working with Junos Configurations

If needed, several execution functions are in the Junos execution module so that you can modify the Junos device configuration. Note that you will typically use the Junos state module and state files covered in subsequent chapters to perform configuration related tasks, rather than these Junos execution functions. Still, in some cases, you will find working with configurations via execution module beneficial, so let's get acquainted.

Let's say you want to enable the REST API service, and the REST API explorer, on both vMX devices in the setup. The following configuration lines will do that (as you see, this is placed in `/srv/salt/myconfig.set` file (more on Salt file locations in later chapters):

```
lab@master:~$ cat /srv/salt/myconfig.set
set system services rest http
set system services rest enable-explorer
```

The following sequence of commands demonstrates how you can lock configuration, load that piece of configuration from the myconfig.set file, print the configuration diff, commit, and then unlock:

```
lab@master:~$ sudo salt vMX* junos.lock
vMX-1:
    ----------
    message:
        Successfully locked the configuration.
    out:
        True
vMX-2:
    ----------
    message:
        Successfully locked the configuration.
    out:
        True

lab@master:~$ sudo salt vMX* junos.load 'salt://myconfig.set' replace='True'
vMX-2:
    ----------
    message:
        Successfully loaded the configuration.
    out:
        True
```

```
vMX-1:
    ----------
    message:
        Successfully loaded the configuration.
    out:
        True

lab@master:~$ sudo salt vMX* junos.diff
vMX-1:
    ----------
    message:

        [edit system services]
        +    rest {
        +        http;
        +        enable-explorer;
        +    }
    out:
        True
vMX-2:
    ----------
    message:

        [edit system services]
        +    rest {
        +        http;
        +        enable-explorer;
        +    }
    out:
        True

lab@master:~$ sudo salt vMX* junos.commit_check
vMX-2:
    ----------
    message:
        Commit check succeeded.
    out:
        True
vMX-1:
    ----------
    message:
        Commit check succeeded.
    out:
        True

lab@master:~$ sudo salt vMX* junos.commit
vMX-2:
    ----------
    message:
        Commit Successful.
    out:
        True
vMX-1:
    ----------
    message:
        Commit Successful.
    out:
        True
```

```
lab@master:~$ sudo salt vMX* junos.unlock
vMX-2:
    ----------
    message:
        Successfully unlocked the configuration.
    out:
        True
vMX-1:
    ----------
    message:
        Successfully unlocked the configuration.
    out:
        True
```

NOTE It is a good practice for the automation systems or scripts to obtain exclusive configuration lock before doing any changes. This way you are making sure you are not interfering with someone else's changes that are possibly being performed at the same time.

Alternatively, you could use the `junos.install_config` function to do everything with one command (it uses the exclusive configuration mode by default, and commits the change if the commit check succeeds):

```
lab@master:~$ sudo salt vMX* junos.install_config 'salt://myconfig.set'
vMX-2:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
vMX-1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

Interestingly, if you try to apply the same configuration again, the following happens:

```
lab@master:~$ sudo salt vMX* junos.install_config 'salt://myconfig.set'
vMX-1:
    ----------
    message:
        Configuration already applied!
    out:
        True
vMX-2:
    ----------
    message:
        Configuration already applied!
    out:
        True
```

For completeness, here is how the output will look like if you try to push an incorrect Junos configuration (in this case, a typo was purposefully injected into `myconfig.set` file):

```
lab@master:~$ sudo salt vMX* junos.install_config 'salt://myconfig.set'
vMX-1:
    ----------
    format:
        set
    message:
        Could not load configuration due to : "ConfigLoadError(severity: error, bad_
element: 1system, message: error: syntax error
        error: syntax error)"
    out:
        False
vMX-2:
    ----------
    format:
        set
    message:
        Could not load configuration due to : "ConfigLoadError(severity: error, bad_
element: 1system, message: error: syntax error
        error: syntax error)"
    out:
        False
```

Copying Files

The `junos.file_copy` execution function copies the file from the local server to the Junos device. You need to provide two parameters: the source path where the file is kept, and the destination path of the file that will be copied. For example:

```
lab@master:~$ sudo salt vMX* junos.file_copy /home/lab/hello.slax /var/db/scripts/op
vMX-2:
    ----------
    message:
        Successfully copied file from /home/lab/hello.slax to /var/db/scripts/op
    out:
        True
vMX-1:
    ----------
    message:
        Successfully copied file from /home/lab/hello.slax to /var/db/scripts/op
    out:
        True
```

Here, the file named `hello.slax` was copied from the `/home/lab` directory of the *minion1* server to the `/var/db/scripts/op` directory of both managed vMX devices.

Installing Software Packages

Finally, the `junos.install_os` function can be used to install the given software image on the Junos OS device. The below example demonstrates installation of the OpenConfig Junos package:

```
lab@master:~$ sudo salt vMX* junos.install_os salt:///junos-openconfig-x86-32-0.0.0.9.tgz
vMX-2:
    ----------
    message:
        Installed the os.
    out:
        True
vMX-1:
    ----------
    message:
        Installed the os.
    out:
        True
```

Here, the file `junos-openconfig-x86-32-0.0.0.9.tgz` is located on the *master* server, in the file roots directory (default is /srv/salt/). It is copied to the devices automatically and then installed, which can be checked afterwards on either of the vMX devices:

```
lab@vMX-1> show version | match openc
JUNOS Openconfig [0.0.0.9]
```

In this case, no reboot was needed after the software package installation. In cases when it is needed, the `reboot=True` parameter will additionally reboot the device after software installation. Alternatively, the `junos.shutdown` function could be used.

NOTE This chapter gave you an idea of what you can do with Salt's Junos execution module. Remember, we did not cover all the available modules and options – consult the above cited SaltStack documentation pages for that.

Chapter 5

Provisioning Junos Configurations with Salt State Module

In this chapter, you will start working with Salt's state system that allows you to define in what state the managed devices must be in. In particular, you will work with Junos state module, `salt.states.junos`, to apply Junos device configuration changes. You will also start working with Jinja configuration templates, which are a powerful way to automate your network device configurations.

Junos Configuration Management with Salt

Now it's time to apply some Junos device configurations. Let's say you want to configure general infrastructure services on your vMX devices – namely, DNS and NTP. One thing you want to take advantage of with automation systems like Salt is configuration templating. That is, the network device feature configuration must be separated from variable data like IP addresses, VLAN numbers, etc.

With Salt, the variable data is naturally stored in the pillar system. Let's create a separate file `infrastructure_data.sls` in the pillar root directory, containing lists of all the NTP and DNS servers that you use:

```
lab@master:~$ cat /srv/pillar/infrastructure_data.sls
ntp_servers:
 - 192.168.0.250
 - 192.168.0.251
dns_servers:
 - 192.168.0.253
 - 192.168.0.254
```

This SLS file uses common YAML syntax whereby colons represent mappings between keys and corresponding values, and dashes represent sequence or list elements (a sequence or a list is just a number of ordered values, such as IP addresses in this example). Generally, indentation is important as it shows structure (nesting of objects) in YAML.

Then, you want to allow your Junos proxy minions to use the data from the `infrastructure_data.sls` file. To do that, edit the pillar's top file as follows (new lines that you need to add are in bold; please realize there are many other ways you can achieve the same result):

```
lab@master:~$ cat /srv/pillar/top.sls
base:
  'vMX-1':
    - proxy-1
  'vMX-2':
    - proxy-2
  'vMX*':
    - infrastructure_data
```

And also refresh the pillar data , so that your minions can see the new pillar data, as follows:

```
lab@master:~$ sudo salt vMX* saltutil.refresh_pillar
vMX-1:
    True
vMX-2:
    True
```

Now let's create a configuration template – but you need to figure out where to place it first. Salt has the concept of a *file roots* directory (actually, it could be a list of directories). It's configured as a `file_roots` parameter in the `/etc/salt/master` configuration file on the Salt master, and this location is `/srv/salt` by default, so let's just use it for now. Create the directory as follows, if you haven't done it already:

```
lab@master:~$ sudo mkdir /srv/salt
```

The important thing about file roots is that Salt runs a lightweight file server over the ZeroMQ bus, and minions have access to the files. So, placing template files in file roots directories automatically allows minions to read them, which is what you want.

Now back to the template. You have multiple options of how you can create the template – Junos text configuration, XML, or Junos set commands. In this example let's create a text configuration template as follows:

```
lab@master:~$ cat /srv/salt/infrastructure_config.conf
system {
  replace: name-server {
{%- for dns_server in pillar.dns_servers %}
  {{ dns_server }};
{%- endfor %}
  }
  replace: ntp {
{%- for ntp_server in pillar.ntp_servers %}
    server {{ ntp_server }};
{%- endfor %}
  }
}
```

And let's discuss a few aspects of the template here:

- The file is placed in the file roots directory on the master. It will be downloaded by minions as needed.
- The file extension is `conf`, which will tell the Junos state module that the configuration should be treated as text format.
- Because of the `replace:` tag, you are removing the previously existing DNS and NTP configurations (if any) from the devices. This approach to configuration can be called "declarative" because this way you unambiguously define exactly what will be in those configuration stanzas after change is applied. Alternatively, you could do `merge load`, so other DNS or NTP servers configured previously would remain in configuration.
- You use Jinja syntax for loops (`for – endfor` keywords) and variable value substitution (double curly braces `{{ }}` around a variable).
- Variables – namely, lists of servers – that are stored in pillar files, are accessed as `pillar.<var_name>`.

The next step is to create a *state SLS* file. This file will describe what state you want your network devices and their configurations to be in. It will reference the Junos *state module* (named `salt.states.junos`), in particular the *state function* `install_config` defined in it, to provision the configuration template.

NOTE Functions of the execution modules, as demonstrated in Chapter 4, could also be used to load Junos device configurations, but the approach with using Salt states is much more powerful.

NOTE For those familiar with Ansible, you can start thinking of Salt states as something similar to Ansible playbooks.

Now let's create the state SLS file in the files root directory on the master server as follows:

```
lab@master:~$ cat /srv/salt/provision_infrastructure.sls
Install the infrastructure services config:
  junos.install_config:
   - name: salt:///infrastructure_config.conf
   - replace: True
   - timeout: 100
```

In this file:

- `Install the infrastructure services config` is a state name (essentially, an arbitrary string describing what this state is going to do).
- `junos.install_config` is a state function from a `salt.states.junos` state module; the job of this function is to load and commit configurations on the Junos devices.

- `name` refers to the path where the configuration (template) file is located.
- `replace: True` specifies that the configuration file uses `replace:` statements.
- `timeout` is NETCONF RPC timeout in seconds. It is especially relevant for commands that take a while to execute.

MORE? The documentation on Salt state module for Junos is available at: https://docs.saltstack.com/en/latest/ref/states/all/salt.states.junos.html

NOTE The set of state functions may look similar to the set of execution functions discussed in Chapter 4, but remember they are not the same. State functions will be commonly invoked by state SLS files and they enforce some specific state on the managed device. Execution functions do something not related to enforcing a specific state. They are typically either executed from Salt CLI or called from within SLS files to return some value – stay tuned for examples of this use in the next few chapters.

Finally, to apply the configuration state described in the SLS file that you just created, you need to execute a `state.apply` function (in this example, vMX devices initially have no NTP or DNS configuration):

```
lab@master:~$ sudo salt vMX* state.apply provision_infrastructure
vMX-2:
----------
          ID: Install the infrastructure services config
    Function: junos.install_config
        Name: salt:///infrastructure_config.conf
      Result: True
     Comment:
     Started: 14:28:38.504545
    Duration: 2483.845 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   2.484 s
vMX-1:
----------
          ID: Install the infrastructure services config
    Function: junos.install_config
        Name: salt:///infrastructure_config.conf
      Result: True
     Comment:
```

```
     Started: 14:28:38.498893
    Duration: 2775.099 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   2.775 s
```

For both vMX devices you can see reports that both configurations were successfully loaded and committed – let's also check the configuration directly, right from one of them:

```
lab@vMX-1> show configuration | compare rollback 1
[edit system]
+  name-server {
+      192.168.0.253;
+      192.168.0.254;
+  }
+  ntp {
+      server 192.168.0.250;
+      server 192.168.0.251;
+  }
```

So far, looks good!

And if you applied the same state again, the following would happen:

```
lab@master:~$ sudo salt vMX* state.apply provision_infrastructure
vMX-1:
----------
          ID: Install the infrastructure services config
    Function: junos.install_config
        Name: salt:///infrastructure_config.conf
      Result: True
     Comment:
     Started: 14:29:54.309258
    Duration: 1389.967 ms
     Changes:
              ----------
              message:
                  Configuration already applied!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
```

```
Failed:    0
------------
Total states run:     1
Total run time:   1.390 s
vMX-2:
----------
          ID: Install the infrastructure services config
    Function: junos.install_config
        Name: salt:///infrastructure_config.conf
      Result: True
     Comment:
     Started: 14:29:55.325782
    Duration: 550.128 ms
     Changes:
              ----------
              message:
                  Configuration already applied!
              out:
                  True

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 550.128 ms
```

As nothing needed to be changed on the devices, no configuration change happened, and no commit was performed, which is natural: the state function has to enforce a certain state, and does nothing if the device is already in that state.

Chapter 6

Provisioning Junos Configurations – Advanced Example

Provisioning network services can be painful if performed manually. Automation of this process improves speed of service delivery and reduces possible errors. This chapter shows how L3VPN services can be configured on MPLS PE routers using Salt.

Lab Setup

This chapter assumes you have a set of provider edge (PE) routers and each PE router generally has multiple connected L3VPN customers. You want to provision the corresponding configuration automatically using Salt.

The example topology used in this chapter is shown in Figure 6.1. As before, you have two vMX devices acting, this time, as MPLS PE routers. The Salt *master* server, as well as *minion1* server running the two Junos proxy minions, are not shown. Consult previous chapters for details on how to set up Salt for managing Junos devices.

In this example, the IP/MPLS backbone is contained of ge-0/0/0 and ge-0/0/1 links connecting vMX-1 and vMX-2 back-to-back. It is pre-configured and not managed by Salt.

More specifically, the initial configuration on PE vMX devices includes:

- Full configuration of core-facing interfaces (family inet and MPLS);
- Standard OSPF, LDP, and IBGP (with `family inet-vpn unicast`) configuration for the IP/MPLS backbone;

- Only physical parameters for customer-facing interfaces (ge-0/0/2) are configured – namely, `flexible-vlan-tagging` and `encapsulation flexible-ethernet-services` are configured. No units are configured on these interfaces – Salt must do that;
- No VRF (L3VPN) instances are configured for the customers – again, Salt must do that;
- The `route-distinguisher-id` is configured in `routing-options` hierarchy on both PEs, so manual configuration for route-distinguisher in VRFs is not needed.

Edge customer-facing logical interfaces and L3VPN VRF instances must be provisioned automatically using Salt, according to the Jinja templates and data specified in pillar YAML files. The goal of this chapter is to show you how to create the required templates and pillar files.

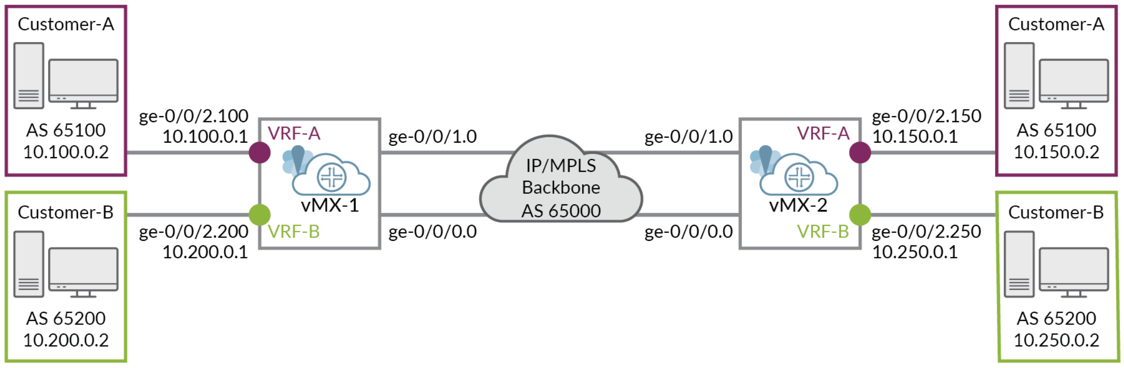


Figure 6.1 The Network Topology Used in This Chapter

Additional points to consider:

- The configuration used for each customer must be standardized using a configuration template. You will use a Jinja template engine for that.
- All variable parameters such as customer AS numbers and IP addresses, as well as mappings of customers to PE devices, must be separated from the template and stored in the pillar YAML files.
- For simplicity, assume that each customer has no more than one Layer 3 connection to each of the PEs.
- The setup must allow for easily adding and removing customers from PE devices, as well as modifying any service parameters.

Steps to Solve the Task

If you carefully read Chapter 5, you will recognize the same approach to solving the configuration provisioning task. Only the template will be a bit more complex, and you will have more variable data to substitute in it.

As part of the solution, the following must be created or updated in Salt:

- A Jinja configuration template.
- Pillar YAML files with variable parameters, describing customers connected to each of the PE devices.
- A pillar top file to properly map pillar data to proxy minions.
- A State SLS file to provision configurations.

Generally speaking, the configuration template is only created once, and is not supposed to be modified unless you are implementing some new functionality, for example, adding configuration options to the template. On the other hand, the pillar YAML files will be changed every time you want to add or remove or modify services for any of the customers. Regardless of the frequency with which you modify the files, you will generally want to keep them all in a version control system (VCS) repository such as Git.

NOTE Detailed coverage of version control systems is beyond the scope of this book. You can find a lot of information on the Internet. A concise practical introduction to Git is contained in the Appendix of *Day One: Automating Junos with Ansible, 2nd Edition* by Sean Sawtell. https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/automating-junos-ansible/.

Configuration Template

To easily create a configuration template, the most straightforward and effective approach is to start with a configuration piece that you want to get from that template. In this case, you want to get something like the following, but for each of the customers on each PE:

```
interfaces {
    ge-0/0/2 {
        unit 100 {
            vlan-id 100;
            family inet {
                address 10.100.0.1/24;
            }
        }
    }
}
routing-instances {
    Cust_A {
        instance-type vrf;
        interface ge-0/0/2.100;
        vrf-target target:65000:1;
        vrf-table-label;
        protocols {
            bgp {
                group EBGP-Cust_A {
                    family inet {
                        unicast {
                            prefix-limit {
                                maximum 10;
                                teardown;
                            }
                        }
                    }
                    peer-as 65100;
                    as-override;
                    neighbor 10.100.0.2;
                }
            }
        }
    }
```

All the parameters here – such as interface names, AS numbers, and more, must be replaced with variables passed to the template from the outside. So the Jinja2 configuration template will include logical interfaces and VRF configurations for your connected customers, but it will also have some operators (such as `if` and `for`) and references to these variables. Using the above example, you should come up with something akin to the following template:

```
lab@master:/srv/salt$ cat l3vpn.conf
groups {
    replace:
    L3VPN-SALT {
        {% if pillar.L3VPN_data %}
        interfaces {
        {% for VPN_entry in pillar.L3VPN_data %}
            {{ VPN_entry.interface_name }} {
                unit {{ VPN_entry.unit }} {
                    vlan-id {{ VPN_entry.vlan_id }};
                    family inet {
                        address {{ VPN_entry.ip_mask }};
                    }
                }
            }
        {% endfor %}
        }
        routing-instances {
        {% for VPN_entry in pillar.L3VPN_data %}
            {{ VPN_entry.customer_id }} {
                instance-type vrf;
                vrf-table-label;
                interface {{ VPN_entry.interface_name }}.{{ VPN_entry.unit }};
                vrf-target {{ pillar.customers[VPN_entry.customer_id].vrf_target }};
                protocols {
                    bgp {
                        group EBGP-{{ VPN_entry.customer_id }} {
                            family inet {
```

```
                                        unicast {
                                            prefix-limit {
                                                maximum {{ VPN_entry.prefix_limit }};
                                                teardown;
                                            }
                                        }
                                    }
                                    peer-as {{ pillar.customers[VPN_entry.customer_id].AS }};
                                    as-override;
                                    neighbor {{ VPN_entry.customer_ip }};
                                }
                            }
                        }
                    }
                {% endfor %}
                }
                {% endif %}
            }
        }
apply-groups L3VPN-SALT;
```

Several points to stress in regard to this template:

- All template configuration is put inside the Junos configuration group named `L3VPN-SALT` so that it is easy to see what part of the configuration was actually uploaded by Salt. It also simplifies configuration modification or removal, as you will see later.
- The `replace:` tag ensures that previous content of `L3VPN-SALT` group is overwritten. The tag is only applied to that group so other groups will be left untouched.
- The `if` operator is checking if `pillar.VPN_data` is empty or not. If it is empty, only the empty configuration group is created.
- The `for` operators are used to loop over multiple entries, corresponding to each of the customers connected to a given PE.
- Note how customer data is put inside the `pillar.customers` data structure. You do not iterate over it in the template – but query the dictionary as needed while iterating over the `VPN_entry`.

MORE? For more information on the Jinja template engine and its available operators visit: http://jinja.pocoo.org.

The Pillar YAML Files

You decide exactly how you will distribute information in pillar files. In this example, a subdirectory named `l3vpn` was created in the pillar root directory, with the following three files:

```
lab@master:/srv/pillar/l3vpn$ ls
customers.sls  vMX-1.sls  vMX-2.sls
```

```
lab@master:/srv/pillar/l3vpn$ cat customers.sls
---
customers:
  Cust_A:
    vrf_target: "target:65000:1"
    AS: 65100
  Cust_B:
    vrf_target: "target:65000:2"
    AS: 65200


lab@master:/srv/pillar/l3vpn$ cat vMX-1.sls
L3VPN_data:
  - customer_id: Cust_A
    interface_name: ge-0/0/2
    unit: 100
    vlan_id: 100
    ip_mask: 10.100.0.1/24
    customer_ip: 10.100.0.2
    prefix_limit: 10
  - customer_id: Cust_B
    interface_name: ge-0/0/2
    unit: 200
    vlan_id: 200
    ip_mask: 10.200.0.1/24
    customer_ip: 10.200.0.2
    prefix_limit: 15

lab@master:/srv/pillar/l3vpn$ cat vMX-2.sls
L3VPN_data:
  - customer_id: Cust_A
    interface_name: ge-0/0/2
    unit: 150
    vlan_id: 150
    ip_mask: 10.150.0.1/24
    customer_ip: 10.150.0.2
    prefix_limit: 10
  - customer_id: Cust_B
    interface_name: ge-0/0/2
    unit: 250
    vlan_id: 250
    ip_mask: 10.250.0.1/24
    customer_ip: 10.250.0.2
    prefix_limit: 15
```

As you can see, the `customers.sls` file defines the basic parameters for each customer, while the PE device-specific files, `vMX-1.sls` and `vMX-2.sls`, define the actual customer connections. These connections match what's seen in Figure 6.1.

You also need to modify the pillar top file to make sure your minions have access to the pillar data you just entered (new lines that must be added are here shown in bold):

```
lab@master:~$ cat /srv/pillar/top.sls
base:
  'vMX-1':
    - proxy-1
    - l3vpn/vMX-1
  'vMX-2':
    - proxy-2
    - l3vpn/vMX-2
  'vMX*':
    - infrastructure_data
    - l3vpn/customers
```

Now let's create the state SLS file:

```
lab@master:/srv/salt$ cat provision_l3vpn.sls
Install the L3 VPN config:
  junos.install_config:
   - name: salt:///l3vpn.conf
   - replace: True
   - timeout: 100
   - diffs_file: /home/lab/diff-{{ grains.id }}.log
```

This file is similar to one used in a previous chapter, it just referenced a different template file this time. Also you provided the `diffs_file` parameter where the `show | compare` output will be stored. Note how the `grains.id` variable, storing the minion ID, is referenced so that the output is stored to a device-specific file.

Before proceeding, don't forget to refresh the pillar data, so that your minions can see the new pillar data, as follows:

```
lab@master:~$ sudo salt vMX* saltutil.refresh_pillar
vMX-2:
    True
vMX-1:
    True
```

Provisioning the Configurations

Let's now apply the state defined in the `provision_l3vpn.sls` file:

```
lab@master:~$ sudo salt vMX* state.apply provision_l3vpn
vMX-2:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:19:00.351538
    Duration: 6099.638 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-2
------------
```

```
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   6.100 s
vMX-1:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:19:00.354764
    Duration: 6898.965 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   6.899 s
```

The messages show that the operation succeeded. Let's confirm by checking the diffs files. Note that diff files are stored in the server running the proxy minions (*minion1* in our case), not the *master* server:

```
lab@minion1:~$ cat diff-vMX-1.log

[edit]
+ groups {
+     L3VPN-SALT {
+         interfaces {
+             ge-0/0/2 {
+                 unit 100 {
+                     vlan-id 100;
+                     family inet {
+                         address 10.100.0.1/24;
+                     }
+                 }
+                 unit 200 {
+                     vlan-id 200;
+                     family inet {
+                         address 10.200.0.1/24;
+                     }
+                 }
+             }
+         }
+         routing-instances {
+             Cust_A {
+                 instance-type vrf;
+                 interface ge-0/0/2.100;
+                 vrf-target target:65000:1;
+                 vrf-table-label;
+                 protocols {
+                     bgp {
+                         group EBGP-Cust_A {
+                             family inet {
+                                 unicast {
+                                     prefix-limit {
+                                         maximum 10;
+                                         teardown;
+                                     }
+                                 }
+                             }
+                             peer-as 65100;
+                             as-override;
+                             neighbor 10.100.0.2;
+                         }
+                     }
+                 }
+             }
+             Cust_B {
+                 instance-type vrf;
+                 interface ge-0/0/2.200;
+                 vrf-target target:65000:2;
+                 vrf-table-label;
+                 protocols {
+                     bgp {
+                         group EBGP-Cust_B {
+                             family inet {
+                                 unicast {
+                                     prefix-limit {
+                                         maximum 15;
+                                         teardown;
+                                     }
+                                 }
+                             }
+                             peer-as 65200;
+                             as-override;
+                             neighbor 10.200.0.2;
+                         }
+                     }
+                 }
+             }
+         }
+     }
+ }
+ apply-groups L3VPN-SALT;

lab@minion1:~$ cat diff-vMX-2.log

[edit]
+ groups {
+     L3VPN-SALT {
+         interfaces {
+             ge-0/0/2 {
+                 unit 150 {
+                     vlan-id 150;
+                     family inet {
+                         address 10.150.0.1/24;
+                     }
+                 }
+                 unit 250 {
+                     vlan-id 250;
+                     family inet {
+                         address 10.250.0.1/24;
+                     }
+                 }
+             }
+         }
+         routing-instances {
+             Cust_A {
+                 instance-type vrf;
+                 interface ge-0/0/2.150;
+                 vrf-target target:65000:1;
+                 vrf-table-label;
+                 protocols {
+                     bgp {
+                         group EBGP-Cust_A {
+                             family inet {
+                                 unicast {
+                                     prefix-limit {
+                                         maximum 10;
+                                         teardown;
+                                     }
+                                 }
+                             }
+                             peer-as 65100;
+                             as-override;
+                             neighbor 10.150.0.2;
+                         }
+                     }
+                 }
+             }
+             Cust_B {
+                 instance-type vrf;
+                 interface ge-0/0/2.250;
+                 vrf-target target:65000:2;
+                 vrf-table-label;
+                 protocols {
+                     bgp {
+                         group EBGP-Cust_B {
+                             family inet {
+                                 unicast {
+                                     prefix-limit {
+                                         maximum 15;
+                                         teardown;
+                                     }
+                                 }
+                             }
+                             peer-as 65200;
+                             as-override;
+                             neighbor 10.250.0.2;
+                         }
+                     }
+                 }
+              }
+          }
+      }
+ }
+ apply-groups L3VPN-SALT;
```

Verifying the configuration on vMX-1 and vMX-2 also shows the configuration was provisioned properly by Salt (this is not shown for the sake of brevity). Let's, however, verify the routes in the Cust_A VRF instance on vMX-1:

```
lab@vMX-1> show route table Cust_A.inet.0

Cust_A.inet.0: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.100.0.0/24      *[Direct/0] 00:06:50
                    > via ge-0/0/2.100
10.100.0.1/32      *[Local/0] 00:06:50
                      Local via ge-0/0/2.100
10.150.0.0/24      *[BGP/170] 00:06:50, localpref 100, from 192.168.0.2
                      AS path: I, validation-state: unverified
                      to 10.0.0.222 via ge-0/0/0.0, Push 16
                    > to 10.0.1.222 via ge-0/0/1.0, Push 16
```

The route to the remote network is there. You can also check to see if a ping between remote Customer-A instances works as it should. The `vr-A` here is a *virtual router instance* created manually on the vMX-1 just for testing purposes (emulating the Customer-A's CE device):

```
lab@vMX-1> ping 10.150.0.2 routing-instance vr-A source 10.100.0.2 rapid
PING 10.150.0.2 (10.150.0.2): 56 data bytes
!!!!!
--- 10.150.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 2.156/2.371/2.679/0.173 ms
```

Modifying Service Configurations

Before you try to modify service configurations, let's see what happens if you just try to apply the same state again:

```
lab@master:~$ sudo salt vMX* state.apply provision_l3vpn
vMX-1:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:48:31.575527
    Duration: 1408.678 ms
     Changes:
              ----------
              message:
                    Configuration already applied!
                out:
                    True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   1.409 s
vMX-2:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:48:32.595480
    Duration: 557.786 ms
     Changes:
              ----------
              message:
                  Configuration already applied!
              out:
                  True

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 557.786 ms
```

As you can see, Salt detects that no change on the devices is needed, and does not perform a commit.

Now it is simple to modify the service parameters – just modify the YAML file and re-run the Salt `state.apply` function. For example, let's say you want to change Customer B's AS number from 65200 to 65300. You only need to change this in `customers.sls` file (the changed value is in bold):

```
lab@master:~$ cat /srv/pillar/l3vpn/customers.sls
customers:
  Cust_A:
    vrf_target: "target:65000:1"
    AS: 65100
  Cust_B:
    vrf_target: "target:65000:2"
    AS: 65300
```

Apply the state:

```
lab@master:~$ sudo salt vMX* state.apply provision_l3vpn
vMX-2:
----------
          ID: Install the L3 VPN config
```

```
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:58:48.782514
    Duration: 6916.208 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   6.916 s
vMX-1:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 09:58:48.787701
    Duration: 7058.338 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   7.058 s
```

And check the diff files:

```
lab@minion1:~$ cat diff-vMX-1.log

[edit groups L3VPN-SALT routing-instances Cust_B protocols bgp group EBGP-Cust_B]
-       peer-as 65200;
+       peer-as 65300;

lab@minion1:~$ cat diff-vMX-2.log

[edit groups L3VPN-SALT routing-instances Cust_B protocols bgp group EBGP-Cust_B]
-       peer-as 65200;
+       peer-as 65300;
```

As another example, assume you want to remove Customer B's service from the vMX-1 device. Just remove the corresponding section from `/srv/pillar/l3vpn/vMX-1.sls` and re-apply the state:

```
lab@master:~$ sudo salt vMX* state.apply provision_l3vpn
vMX-2:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 10:03:20.069114
    Duration: 560.277 ms
     Changes:
              ----------
              message:
                  Configuration already applied!
              out:
                  True

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time: 560.277 ms
vMX-1:
----------
          ID: Install the L3 VPN config
    Function: junos.install_config
        Name: salt:///l3vpn.conf
      Result: True
     Comment:
     Started: 10:03:19.866889
    Duration: 3110.515 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.111 s

lab@minion1:~$ cat diff-vMX-1.log

[edit groups L3VPN-SALT interfaces ge-0/0/2]
-     unit 200 {
-         vlan-id 200;
-         family inet {
-             address 10.200.0.1/24;
-         }
-     }
[edit groups L3VPN-SALT routing-instances]
-    Cust_B {
-        instance-type vrf;
-        interface ge-0/0/2.200;
-        vrf-target target:65000:2;
-        vrf-table-label;
-        protocols {
-            bgp {
-                group EBGP-Cust_B {
-                    family inet {
-                        unicast {
-                            prefix-limit {
-                                maximum 15;
-                                teardown;
-                            }
-                        }
-                    }
-                    peer-as 65300;
-                    as-override;
-                    neighbor 10.200.0.2;
-                }
-            }
-        }
-    }
```

Summary

This chapter has shown you how to provision more advanced configurations using Salt. Once again, the basic idea is to separate the configuration template from variable data, storing all variable data in the pillar system. Once data describing your system changes, the configs are regenerated as needed, and applied to the managed devices.

As you saw in this chapter, all configurations provisioned by Salt were put in a separate configuration group. This method has several advantages, including consistency and visibility into what the automation system is doing. Changing and removing of services with such an approach is really easy. Be aware that configuration groups are a unique feature of Junos, so use them with caution when automating in multivendor environments. Be sure to read Chapter 12 and its coverage of NAPALM modules for Salt that you may want to use in such a situation.

This chapter treated the deployment of L3VPN services – but of course, multiple other services and configurations can be provisioned similarly, as long as you can come up with a standardized template for them.

As a final note, the examples demonstrated here could be modified to implement a Create, Read, Update, Delete (CRUD) model for services, with or without using configuration groups.

Chapter 7

Junos Syslog Engine and Salt Reactors

This chapter dives into the Salt event-driven infrastructure (EDI). You will become familiar with Salt engines using the Junos syslog engine as an example. Then you will start working with Salt reactors, learning how to apply actions based on certain events. By the end of the chapter, you'll also see how to post Slack messages using Salt – again, this will be done based on the specific Junos event. Let's get going.

Salt Engines and Reactors

Salt *engines* are long-running processes managed by Salt. They are used for various purposes and one of the common use cases is importing or exporting Salt events to or from a ZeroMQ event bus. In this case, you can think of an engine as a "proxy" transforming the event data to or from native Salt format.

Junos events are sent from devices as syslog messages, and it's easy to see why *Junos syslog engine* is needed if you want to integrate Junos devices into Salt EDI: the syslog engine will receive the syslog messages and translate them into data that can be published to the native Salt event bus.

NOTE The official Junos syslog engine documentation is available at https://docs.saltstack.com/en/latest/ref/engines/all/salt.engines.junos_syslog.html

The Salt *reactor* system is something that matches events seen on the event bus and applies actions based on the configured rules. Essentially, with reactors your automation system becomes a closed feedback loop system.

To start working with EDI, you'll enable the Junos syslog engine on the *master* server (note however, that it can work on the minion as well). You will then configure Junos devices to send syslog messages to the syslog engine. Finally, some reactor rules will be configured and tested. The setup is summarized in Figure 7.1.

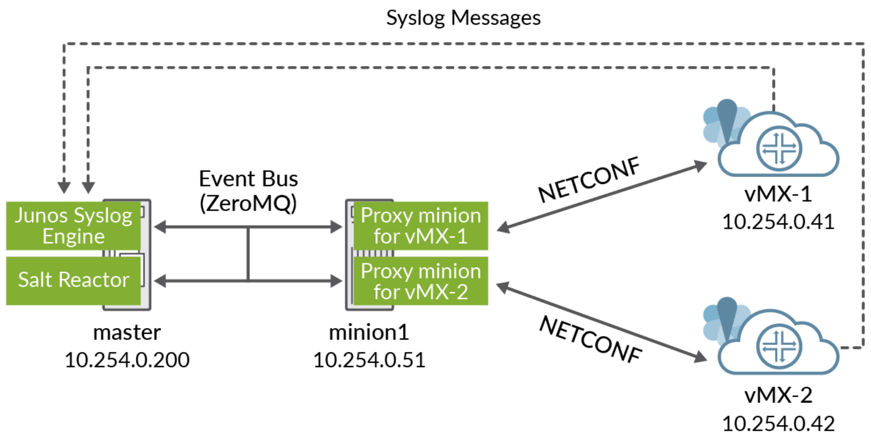


Figure 7.1 Salt Setup for Managing Two Junos Devices, Including the Junos Syslog Engine and Salt Reactor.

Enabling Junos Syslog Engine

To enable Junos syslog engine, you will first need to install a couple of required Python libraries: `pyparsing` and `twisted`. In this case, you do it on the *master* server:

```
lab@master:~$ sudo apt-get install python-pip
lab@master:~$ sudo pip install pyparsing
lab@master:~$ sudo pip install twisted
```

Here, output was omitted for brevity; pip installation in the first line is only necessary if this tool was not installed before.

Now edit the Salt master configuration file, adding the following new lines to it:

```
lab@master:~$ sudo vi /etc/salt/master
engines:
  - junos_syslog:
      port: 10514
```

Here, you have chosen to use UDP port 10514 for syslog messages. You also need to restart the Salt master process to apply new configuration, for example:

```
lab@master:~$ sudo killall salt-master
lab@master:~$ sudo salt-master -d
```

The Junos devices must send their logs to the master server address using the specified port, so the following configuration has to be provisioned on our Junos devices (assuming that you want to send all facility and all severity events to Salt):

```
# set system syslog host 10.254.0.200 any any
# set system syslog host 10.254.0.200 port 10514
```

Now you can provision this configuration exactly as you want. Just as an exercise, here, let's load this configuration from a file, using the Salt execution function *junos.install_config* (discussed in a previous chapter):

```
lab@master:~$ cat /srv/salt/myconfig.set
set system syslog host 10.254.0.200 any any
set system syslog host 10.254.0.200 port 10514

lab@master:~$ sudo salt vMX* junos.install_config salt:///myconfig.set
vMX-2:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
vMX-1:
    ----------
    message:
        Successfully loaded and committed!
    out:
        True
```

Let's now check events on the bus using this command:

```
lab@master:~$ sudo salt-run state.event pretty=True

...

jnpr/syslog/vMX-1/UI_CMDLINE_READ_LINE  {
    "_stamp": "2018-08-21T13:37:29.148658",
    "daemon": "mgd",
    "event": "UI_CMDLINE_READ_LINE",
    "facility": 23,
    "hostip": "10.254.0.41",
    "hostname": "vMX-1",
    "message": "User 'lab', command 'configure '",
    "pid": "77339",
    "priority": 190,
    "raw": "<190>Aug 21 15:30:47 vMX-1 mgd[77339]: UI_CMDLINE_READ_
LINE: User 'lab', command 'configure '",
    "severity": 6,
    "timestamp": "2018-08-21 06:37:29"
}
jnpr/syslog/vMX-1/UI_DBASE_LOGIN_EVENT  {
    "_stamp": "2018-08-21T13:37:29.155585",
    "daemon": "mgd",
    "event": "UI_DBASE_LOGIN_EVENT",
    "facility": 23,
    "hostip": "10.254.0.41",
    "hostname": "vMX-1",
    "message": "User 'lab' entering configuration mode",
    "pid": "77339",
    "priority": 189,
    "raw": "<189>Aug 21 15:30:47 vMX-1 mgd[77339]: UI_DBASE_LOGIN_
EVENT: User 'lab' entering configuration mode",
    "severity": 5,
    "timestamp": "2018-08-21 06:37:29"
}
...
^C
```

Note, here we're using the `salt-run` command that is executing Salt *runners*. Salt runners are modules used to execute functions on the master server (rather than minions). Clearly, the `state.event` function is used to output the events seen on the bus. Because all communication happens over the bus, you will likely see many other messages in addition to the ones shown in the above output. The above two messages are actually messages from the vMX-1 device that were generated at the moment the user entered the CLI configuration mode.

Also note the line at the beginning of each message, such as `jnpr/syslog/vMX-1/UI_CMDLINE_READ_LINE`. It is called the event *tag* (or *topic*). The exact view of the topic generated by Junos syslog engine is configurable, but we will use the default one here, which is `jnpr/syslog/<hostname>/<event-id>`.

Configuring the Reactor

Now let's enable the reactor. This is something you do in the Salt master configuration file. Let's say you want to react on the `UI_COMMIT_COMPLETED` Junos event, which indicates that the configuration commit was just performed. Modify the master configuration as follows (new lines are in bold), and then restart Salt master as you did before:

```
lab@master:~$ sudo vi /etc/salt/master

engines:
  - junos_syslog:
      port: 10514

reactor:
  - 'jnpr/syslog/*/UI_COMMIT_COMPLETED':
    - salt://reactor/react_to_commit.sls
```

Here, the event tag is mapped to a reactor SLS file, content of which will be reviewed shortly. The * in the event tag indicates that any host sending a `UI_COMMIT_COMPLETED` message will match.

NOTE If you are familiar with Junos OS event policies and event scripts, the reactor settings may remind you of Junos event policies, while reactor files take the place of event scripts. The big difference, though, is that Junos event policies and scripts are tied to a single box, while with Salt you are orchestrating the network as a single system.

The reactor SLS file for this example is as follows (also, you create a separate directory for your reactor files):

```
lab@master:~$ sudo mkdir /srv/salt/reactor
lab@master:~$ sudo vi /srv/salt/reactor/react_to_commit.sls
Run task based on event:
  local.state.apply:
    - tgt: {{ data['hostname'] }}
    - arg:
      - post_to_slack
```

In this SLS file:

- "Run task based on event" is the human-readable ID.
- The `local` prefix is rather counter-intuitive – it actually indicates that a remote execution function (in this case, `state.apply`) will be run on targeted minions.
- The `tgt` parameter defines the target, in this case the hostname extracted from event data (which is equal to the minion ID in our case). Note that event data is available in reactor SLS files via a special `data` variable.
- The `arg` parameter defines function arguments, in this case you have `post_to_slack` state file name provided as a parameter.

Overall, the result of executing the `react_to_commit.sls` file is similar to executing this Salt command: `salt 'minion-id' state.apply post_to_slack`, with `minion-id` equal to the hostname contained in the syslog event data.

Configuring Slack Connection

To keep this EDI chapter as simple as possible, the only action you take based on the event is posting a Slack message. So, let's define the `post_to_slack.sls` file. Here's its content:

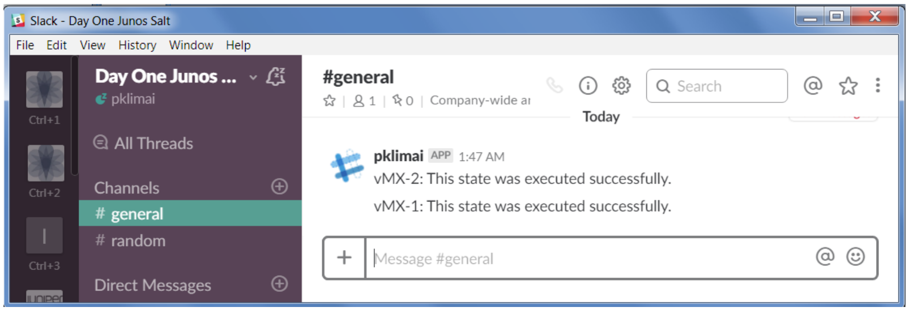
```
lab@master:~$ cat /srv/salt/post_to_slack.sls
slack-message:
  slack.post_message:
    - channel: '#general'
    - from_name: pklimai
    - message: '{{ grains.id }}: This state was executed successfully.'
    - api_key: XXXX-XXXX
```

NOTE You can obtain the Slack API key for your channel here: https://api.slack.com/custom-integrations/legacy-tokens.

As you can see, this state file executes a single state function `slack.post_message`, passing channel and user names, message and API key as parameters. You can also see how the minion ID, using grains data, is substituted into the message string.

Testing the Reactor

If you have done everything right, after every commit, on either of the vMX devices, you should now be getting a Slack message similar to what is shown in Figure 7.2. This is cool, but do not rest on your laurels too soon. Proceed to Chapter 8 and you will dive deeper into the Salt EDI!



pklimai APP 1:47 AM
vMX-2: This state was executed successfully.
vMX-1: This state was executed successfully.

Figure 7.2 Slack Chat Window

Chapter 8

Basics of Building Event-Driven Infrastructure

This chapter continues the discussion started in Chapter 7, explaining how you can integrate Junos devices into Salt's event-driven infrastructure (EDI).

Reacting on Events

Multiple events requiring a proper reaction may happen in your network at any time. For example:

- Interfaces and/or protocol sessions may flap or go down;
- Devices and circuits may become over utilized;
- Some services may become unreachable;
- Wrong configurations may be applied by the user or automation system, and so on.

With Salt, you can react as you see fit, based on such events. Among other things, you can:

- Change a network device's configuration (not necessarily the same device that reported the event);
- Bounce interfaces, as well as software or hardware components;
- Collect data and store it in the file system or commit to a version control system such as Git;
- Automatically create or update the support tickets on the ticketing server;
- Send messages to various channels, such as email, SMS, Slack, HipChat, etc.

With Salt, you can react any way you like to most anything that happens in your network. But be sure not to overcomplicate the reactor system. Remember, actions that you apply based on events may lead to changes that will cause some other events – so you always need to analyze the possible consequences, and keep it simple and clear.

As all networks are different, there is no universal advice on what exactly to react to, and how exactly to react. Many examples are available, however, so you can see what can be done. In particular, Chapter 7 contained a simple example of posting a Slack message based on a commit event. The next sections of this chapter present an example that is a bit more involved, where, based on the event, device configuration is checked and modified, if needed. Chapter 11 contains yet another example where JSNAPy test is run, also based on some event. In addition, many more examples are available, created by Juniper engineers and the community, so please consult the Appendix at the end of this book for some additional inspiration.

SLS File to Check and Enforce Configuration

Now let's discuss another EDI example, where the workflow is as follows: when a commit event for some device is received, that device must be checked for the presence of administratively disabled interfaces. If any of the interfaces (one or more) are disabled, Salt must re-enable them immediately.

Let's first look at the following outputs from a Junos device:

```
lab@vMX-1> show interfaces terse | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
    <rpc>
        <get-interface-information>
                <terse/>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

lab@vMX-1> show interfaces terse | display json
{
    "interface-information" : [
    {
        "attributes" : {"xmlns" : "http://xml.juniper.net/junos/17.1R2/junos-interface",
                        "junos:style" : "terse"
                       },
        "physical-interface" : [
        {
            "name" : [
            {
                "data" : "ge-0/0/0"
            }
            ],
            "admin-status" : [
            {
                "data" : "up"
            }
            ],
            "oper-status" : [
            {
                "data" : "up"
            }
            ],
... output trimmed ...
```

The first command here shows that the RPC corresponding to `show interface terse` command is `<get-interface-information>`, with `<terse>` option (provided as a nested XML element).

The second command's output shows that once you get the RPC reply, the list containing interface data is at the `['interface-information'][0]['physical-interface']` level, and when iterating for each of the interfaces, the interface name and admin status are correspondingly at `['name'][0]['data']` and `['admin-status'][0]['data']`.

Now you can create a state SLS file that checks for interface administrative status and applies the proper configuration if the admin state is down:

```
lab@master:~$ cat /srv/salt/check_interfaces.sls
{% set intf_info = salt['junos.rpc']('get-interface-information', '', 'json', terse=True) %}
{% for interface in intf_info['rpc_reply']['interface-information'][0]['physical-interface'] %}
{% if interface['admin-status'][0]['data'] == 'down' %}
Enable interface {{ interface['name'][0]['data'] }}:
  junos.install_config:
   - name: salt://enable_interface.set
   - template_vars:
       interface_name: {{ interface['name'][0]['data'] }}
{% endif %}
{% endfor %}
```

And the referenced configuration file is as simple as:

```
lab@master:~$ cat /srv/salt/enable_interface.set
delete interfaces {{ template_vars['interface_name'] }} disable
```

In this SLS file:

- You first call the `junos.rpc` execution function, providing RPC and output format (JSON) as parameters, and store the output to `intf_info` Jinja variable.
- You iterate over each interface using for loop, and check the admin status. If it is “down”, you apply `junos.install_config` state, providing the template file name and template variable equal to the interface name.

Note the ID of the state includes the interface name, so it will be different for different interfaces. This part is critical for a situation when you have more than one interface disabled, as the ID must be unique for different states inside the SLS file.

Before inserting the SLS file into the reactor system, you can easily test the SLS file, as follows (two interfaces were manually disabled on vMX-1 in advance):

```
lab@master:~$ sudo salt vMX-1 state.apply check_interfaces
vMX-1:
----------
          ID: Enable interface ge-0/0/0
    Function: junos.install_config
        Name: salt://enable_interface.set
      Result: True
     Comment:
     Started: 08:10:12.372581
    Duration: 3751.626 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True
----------
          ID: Enable interface ge-0/0/1
    Function: junos.install_config
        Name: salt://enable_interface.set
      Result: True
     Comment:
     Started: 08:10:16.124440
    Duration: 2378.168 ms
     Changes:
              ----------
              message:
                  Successfully loaded and committed!
              out:
                  True

Summary for vMX-1
------------
Succeeded: 2 (changed=2)
Failed:    0
------------
Total states run:     2
Total run time:   6.130 s
```

Not yet event-driven, but this SLS file works.

React Based on Event

Now let's make the Salt system perform the same check-and-fix-if-needed every time a commit is performed. To do so, proper Salt engine and reactor configuration is required on the *master* server (most of this was configured in Chapter 7, but let's review everything again):

```
lab@master:~$ cat /etc/salt/master

engines:
  - junos_syslog:
      port: 10514
```

```
reactor:
  - 'jnpr/syslog/*/UI_COMMIT_COMPLETED':
    - salt://reactor/react_to_commit.sls

... output trimmed ...


lab@master:~$ sudo cat /srv/salt/reactor/react_to_commit.sls

Run task based on event:
  local.state.apply:
    - tgt: {{ data['hostname'] }}
    - arg:
      - check_interfaces
```

Also, Junos devices must be instructed to send syslog messages to a master server IP address.

Let's perform a quick check on a device:

```
[edit]
lab@vMX-1# set interfaces ge-0/0/0 disable

[edit]
lab@vMX-1# set interfaces ge-0/0/1 disable

[edit]
lab@vMX-1# commit
commit complete

[edit]
lab@vMX-1# show interfaces
ge-0/0/0 {
    unit 0 {
        family inet {
            address 10.0.0.1/24;
        }
    }
}
ge-0/0/1 {
    unit 0 {
        family inet {
            address 10.1.0.1/24;
        }
    }
}
fxp0 {
    unit 0 {
        family inet {
            address 10.254.0.41/24;
        }
    }
}


[edit]
lab@vMX-1# run show system commit
0   2018-09-22 17:19:18 UTC by lab via netconf
```

```
1   2018-09-22 17:19:16 UTC by lab via netconf
2   2018-09-22 17:19:13 UTC by lab via cli
... output trimmed ...
```

Although your configuration change was applied initially, you can see that almost immediately afterward Salt again enabled both interfaces (in this case, by two different commits performed via NETCONF).

Summary

In the approach demonstrated in this chapter, you first created and tested the state file, and only after that used it from within the reactor. This is a good approach as the Salt reactor system may be rather challenging to troubleshoot. If you have to do it, the most direct way is to stop the Salt master daemon and then start it in the debug mode, using the `-l debug` key. Then, force the triggering event to happen and see where it goes wrong. When finished, do not forget to restart the daemon as normal.

The Salt reactor system is good if you want to perform simple "if this then that" behavior logic – and this is something you want to start with. For something more complicated, such as matching on multiple events happening in certain succession, the Thorium "complex reactor" component of Salt can be used. Consult the documentation at https://docs.saltstack.com/en/latest/topics/thorium/index.html for more details.

Chapter 9

Creating Custom Modules for Salt with Junos PyEZ

In this chapter you'll extend Salt capabilities with custom Python modules. When developing such modules for working with Junos you can use the simple but powerful Junos PyEZ library.

This chapter includes some Python code examples but they will be thoroughly explained. First, you will develop a very simple execution function just to get the idea. Next, a more complicated function solving a realistic task will be demonstrated.

Custom Salt Modules Review

Although a large number of professionally developed, well-tested modules already exist for Salt, sometimes you may find yourself in the situation where some functionality is not easily achievable with the existing ones. Salt allows you to easily create and use custom modules.

NOTE This chapter will discuss creating execution modules. State modules can also be created rather simply. Consult the official SaltStack documentation on creating custom execution and state modules for more details: https://docs.saltstack.com/en/latest/ref/modules/ and https://docs.saltstack.com/en/latest/ref/states/writing.html.

Here is a quick overview on creating execution modules for Salt:

- You create modules using the Python programming language (alternatively, Cython can be used).
- The Python module is just a `.py` file including functions (defined with the `def` keyword) that can be called by Salt.
- The modules are put in the `_modules` directory (note the underscore in the beginning) at the root of the Salt fileserver so, with default settings, the directory will be `/srv/salt/_modules`.
- Modules have access to pillar and grains data using special dictionaries named `__pillar__` and `__grains__`. They also have access to all of the Salt functions via the `__salt__` variable.

Creating a Simple Execution Module

Let's create a very basic Salt module, named `dayonejunos`, including a function named `hello`, as follows:

```
lab@master:~$ cat /srv/salt/_modules/dayonejunos.py

def hello(*args, **kwargs):
    ret = {}
    ret['pillar'] = __pillar__
    ret['grain'] = __grains__
    ret['rpc_result'] = __salt__['junos.rpc']('get-interface-information')
    return ret
```

In the first line of the `hello` function, an empty dictionary named `ret` is created. Then it is filled with the Salt pillar and grain data using special variables mentioned above. Finally, the `junos.rpc` execution function is called via the `__salt__` dictionary, providing RPC name (`'get-interface-information'`) as a parameter. The output is again stored to `ret` dictionary, and this dictionary is returned from a function in the last line.

Also, as you can see, the function `hello` may accept some positional (`args`) and named (`kwargs`) arguments, but they are just not used in our case.

Now you need to synchronize the modules, copying them from the master file server to the (proxy) minions. You can do it like this:

```
lab@master:~$ sudo salt vMX-* saltutil.sync_modules
vMX-1:
    - modules.dayonejunos
vMX-2:
    - modules.dayonejunos
```

And now you can execute your custom function the usual Salt way (note the output of the command is abbreviated, but you can see parts of grain and pillar data, as well as RPC output for the command you requested):

```
lab@master:~$ sudo salt vMX-* dayonejunos.hello
vMX-1:
    ----------
    grain:
        ----------
        cpuarch:
            x86_64
        dns:
            ----------
            domain:
            ip4_nameservers:
                - 8.8.8.8
            ip6_nameservers:
            nameservers:
                - 8.8.8.8
...
    pillar:
        ----------
        L3VPN_data:
            |_
              ----------
              customer_id:
                  Cust_A
              customer_ip:
                  10.100.0.2
              interface_name:
                  ge-0/0/2
              ip_mask:
                  10.100.0.1/24
              prefix_limit:
                  10
              unit:
                  100
              vlan_id:
                  100
        customers:
            ----------
            Cust_A:
...
    rpc_result:
        ----------
        out:
            True
        rpc_reply:
            ----------
            interface-information:
                ----------
                physical-interface:
                    |_
                      ----------
                      active-alarms:
                          ----------
                          interface-alarms:
                              ----------
                              alarm-not-present:
                      active-defects:
                          ----------
                          interface-alarms:
...
```

```
vMX-2:
    ----------
    grain:
        ----------
        cpuarch:
            x86_64
        dns:
            ----------
            domain:
            ip4_nameservers:
                - 8.8.8.8
            ip6_nameservers:
            nameservers:
                - 8.8.8.8
...
```

So, it works. Time to build something perhaps more useful.

Solving a Practical Problem with Execution Function

Let's add a new function to the `dayonejunos` module. The task is as follows:

- Name the module function `check_traceoptions`.
- This function must check your Junos device configuration for possibly enabled traceoptions (a Junos equivalent of debug). To reduce the load on the devices, all traceoptions must normally be disabled so you want to warn users if this is not the case.
- As traceoptions may be enabled at various levels of configuration, the function must take this into account.
- Also take into account that traceoptions can be deactivated, in this case do not warn the user.
- The result of running the function must include the list of hierarchies at which traceoptions are enabled for a device.

Before proceeding to the function source code, let's look at the Junos configuration example that has some traceoptions enabled:

```
[edit]
lab@vMX-1# show protocols
bgp {
    inactive: traceoptions {
        file bgp;
        flag nsr-synchronization;
    }
    group IBGP {
        type internal;
        local-address 192.168.0.1;
        family inet {
            unicast;
        }
        family inet-vpn {
            unicast;
        }
        neighbor 192.168.0.2;
    }
}
ospf {
    traceoptions {
        file ospf;
        flag nsr-synchronization;
    }
    area 0.0.0.0 {
        interface ge-0/0/0.0 {
            interface-type p2p;
        }
        interface ge-0/0/1.0 {
            interface-type p2p;
        }
        interface lo0.0;
    }
}
```

Here, both OSPF and BGP traceoptions are in the configuration, but the BGP ones have been deactivated. Now you might be thinking that the task of analyzing such configurations is hard, because you never know in advance where traceoptions knobs will be located – and you may be right, analyzing configuration in text form is not very pleasant. Let's instead look at the same configuration in XML:

```
[edit]
lab@vMX-1# show protocols | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
    <configuration junos:changed-seconds="1535125177" junos:changed-
localtime="2018-08-24 15:39:37 UTC">
            <protocols>
                <bgp>
                    <traceoptions inactive="inactive">
                        <file>
                            <filename>bgp</filename>
                        </file>
                        <flag>
                            <name>nsr-synchronization</name>
                        </flag>
                    </traceoptions>
                    <group>
                        <name>IBGP</name>
                        <type>internal</type>
                        <local-address>192.168.0.1</local-address>
                        <family>
                            <inet>
                                <unicast>
                                </unicast>
                            </inet>
                            <inet-vpn>
                                <unicast>
                                </unicast>
                            </inet-vpn>
                        </family>
                        <neighbor>
```

```
                            <name>192.168.0.2</name>
                        </neighbor>
                    </group>
                </bgp>
                <ospf>
                    <traceoptions>
                        <file>
                            <filename>ospf</filename>
                        </file>
                        <flag>
                            <name>nsr-synchronization</name>
                        </flag>
                    </traceoptions>
                    <area>
                        <name>0.0.0.0</name>
                        <interface>
                            <name>ge-0/0/0.0</name>
                            <interface-type>p2p</interface-type>
                        </interface>
                        <interface>
                            <name>ge-0/0/1.0</name>
                            <interface-type>p2p</interface-type>
                        </interface>
                        <interface>
                            <name>lo0.0</name>
                        </interface>
                    </area>
                </ospf>
                <ldp>
                    <interface>
                        <name>all</name>
                    </interface>
                    <interface>
                        <name>fxp0.0</name>
                        <disable/>
                    </interface>
                </ldp>
            </protocols>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

Analyzing this XML configuration, it turns out that you just need to locate the `<traceoptions>` tag, making sure it does not have the `inactive="inactive"` attribute applied. To do this, use the power of XPath, which is a standard query language used for selecting nodes from an XML document. With XPath, the task is solved with a few lines of code – you'll see how in a moment.

NOTE Detailed coverage of XML and XPath is beyond the scope of this book. An XML tutorial is available at https://www.w3schools.com/xml/. An XPath introduction can be found at https://www.w3schools.com/xml/xpath_intro.asp.

The source code of the new function is as follows – place it in the `dayonejunos.py` file:

```
# (1)  Perform the required imports
from lxml import etree

# (2)  Execution function definition
def check_traceoptions(*args, **kwargs):
    '''
    Execution function to check if non-deactivated traceoptions
    are enabled in Junos device configuration
    '''

    # (3)  Get reference to the Device instance
    conn = __proxy__['junos.conn']()

    # (4)  Create an empty dictionary
    ret = {}

    # (5)  Fill in values corresponding to 'result' and 'out' keys
    ret['result'] = True
    ret['out'] = True

    # (6)  Try executing the <get-config> Junos RPC
    try:
        conf = conn.rpc.get_config()
    # (7)  In case of an error, stop
    except Exception as exception:
        ret['message'] = 'RPC execution failed due to "{0}"'.format(exception)
        ret['out'] = False
        ret['result'] = None
        return ret

    # (8)  Build XML Element Tree
    conf_tree = etree.ElementTree(conf)

    # (9)  Perform hierarchical search for <traceoptions> tag
    traceoptions_list = conf_tree.xpath(".//traceoptions")

    # (10)  The hierarchies list will store configuration stanzas
    # where non-deactivated traceoptions were found
    hierarchies = []

    # (11)  Iterate over the traceoptions_list
    for traceopt in traceoptions_list:
        # Only append configuration hierarchy to the list if
        # traceoptions are active (no 'inactive' attribute )
        # at this level
        if traceopt.xpath("./@inactive") != ["inactive"]:
            hierarchies.append(conf_tree.getpath(traceopt))

    # (12)  Update the result dictionary with found hierarchies
    ret['conf_stanzas'] = hierarchies

    # (13)  Set message in the result dictionary
    if len(hierarchies) != 0:
        ret['result'] = False
        ret['message'] = "Enabled traceoptions were detected!"
    else:
        ret['message'] = "No traceoptions for this device"

    # (14)  Return the result from execution function
    return ret
```

In addition to the comments in the Python code, here are some extra explanations on what is happening in the function:

1. Import `etree` module from `lxml` library. This is needed to use `ElementTree` class in the below code. Note that although lxml is a third-party library, it is one of the requirements of Junos PyEZ, so if you have PyEZ installed, you already have lxml.
2. Define the new function using `def` keyword. After that line, you have a documentation comment enclosed by three quotes, explaining the function purpose.
3. Use special `__proxy__` dictionary to obtain a reference to a Junos device connection (essentially, an instance of PyEZ `Device` class). Remember the module will be executed on the proxy minion that already has a NETCONF connection to the managed device.
4. Create a new empty dictionary named `ret`. Later, it will be filled in with some data and returned from a function.
5. Fill in the initial `'result'` and `'out'` fields in the dictionary. The `'result'` value will be `True` if there are no traceoptions enabled, and `False` otherwise. A common convention for `'out'` is for it to be set to `True` if the device was able to call RPC successfully.
6. Start the `try/except` block, which is used to process possible exceptional situations in Python. In that block, execute the `get-config` Junos RPC using PyEZ `rpc` object nested inside the device instance. Store the result to `conf` variable.
7. The `except` block will only execute if an exception happened. In this case, return from function, setting `'out'` value to `False` and `'result'` value to `None` (as neither `True` nor `False` seems suitable in such a situation).
8. Build the XML Element Tree from the device XML configuration. For more details on lxml, consult https://lxml.de/tutorial.html.
9. Use XPath to search for `traceoptions` element node in the configuration. The trick here is to do a recursive search at all levels, which is accomplished by using a double slash syntax (//). The result of XPath application is a list, stored to a variable `traceoptions_list`.
10. The `hierarchies` variable is a list that will store configuration stanzas where non-deactivated traceoptions were found. Assign an empty list to this variable initially.
11. Iterate over the `traceoptions_list`. At each pass, the `traceopt` variable will take a value of an element that matched the XPath (essentially, traceoptions are enabled at some level). Check if traceoptions are deactivated, and if not, update the `hierarchies` list with the path to the new element.

12. Update the `ret` dictionary with a `hierarchies` list.
13. Set `'result'` and `'message'` keys in the `result` dictionary, depending whether active traceoptions were found in the configuration.
14. Return the result from the execution function.

DISCUSSION At this point you may be wondering why a PyEZ RPC call in Step 6 was needed at all? Couldn't we have just retrieved the configuration using something like a `__salt__['junos.rpc']('get-config')` call to the execution function? Well, although the `junos.rpc` function is able to return data in multiple formats, it does not really return XML data that can be easily processed with XPath. Instead, a Python object created using the jxmlease library is returned from that function if you specify XML output format. So, to be able to work with XPath, this approach was required.

Running the check_traceoptions Function

Finally, let's sync the modules and run the execution function:

```
lab@master:~$ sudo salt vMX-* saltutil.sync_modules
vMX-1:
    - modules.dayonejunos
vMX-2:
    - modules.dayonejunos

lab@master:~$ sudo salt vMX-* dayonejunos.check_traceoptions
vMX-2:
    ----------
    conf_stanzas:
    message:
        No traceoptions for this device
    out:
        True
    result:
        True
vMX-1:
    ----------
    conf_stanzas:
        - /configuration/interfaces/traceoptions
        - /configuration/protocols/ospf/traceoptions
    message:
        Enabled traceoptions were detected!
    out:
        True
    result:
        False
```

You can see that for the vMX-2 device, no traceoptions are enabled, while a couple of configuration stanzas on vMX-1 do include them.

Summary

Note that if you have a larger configuration part (such as a routing protocol) deactivated, and that larger configuration includes traceoptions, the above discussed `check_traceoptions` function will treat these traceoptions as enabled if there is no explicit `inactive` attribute exactly at the traceoptions level. The idea of this approach is to make sure that traceoptions are explicitly disabled and will not become activated accidentally, when a larger configuration part is activated.

Using the approach demonstrated in this chapter, you can use Python and PyEZ to create many other useful Salt modules for managing Junos devices. As an exercise, you might want to extend the example used in this chapter, building an execution function that not only detects, but also disables the traceoptions in the Junos device configuration!

MORE? You can find lots of script ideas for Junos PyEZ in *Day One: Junos PyEZ Cookbook* at https://www.juniper.net/dayone.

Chapter 10

Validating Operational States of a Junos Device

It's time to validate and test the status of your network devices with Salt. This approach uses state SLS files and Salt's ability to call execution functions from the state files. Additionally, you'll use Python to define your SLS files (instead of the default combination of Jinja and YAML).

A Need to Test

You always want to make sure your network functionality does not degrade after changing configurations, upgrading software, or changing the traffic load. Even during normal operations there is a need to continuously check how your network works with the timely collection of feedback from its elements.

Salt has a few ways to perform network testing. The approach discussed in this chapter is based on the fact that Salt execution functions can be called from SLS state files and the resulting structured output can be analyzed. Then, depending on the results of the analysis, various actions can be performed. In the simplest case, this can be just outputting a corresponding message to the administrator, and in a bit more complex scenario, self-healing can be attempted. Let's go validate some states.

Example Scenario

Figure 10.1 illustrates a basic example scenario used in this chapter. You have two routers connected using two links (any resemblance to actual production network topologies is purely coincidental). The OSPF protocol using area 0.0.0.0 runs on both links. You want to make sure that the OSPF neighbors are in the Full State, as in this example output from vMX-1:

```
lab@vMX-1> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
10.0.0.222       ge-0/0/0.0             Full      192.168.0.2      128    37
10.0.1.222       ge-0/0/1.0             Full      192.168.0.2      128    36
```

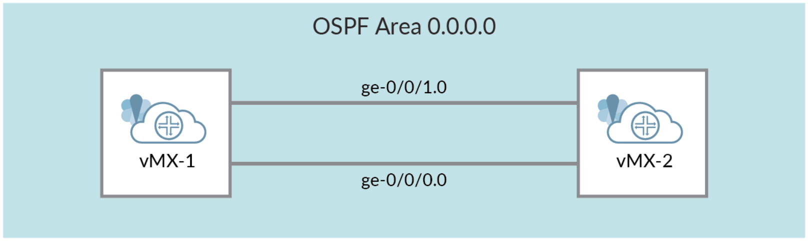


Figure 10.1 Network Topology Used for This Chapter.

As you will see next, the SLS state file will use the `junos.rpc` execution module to get this data in JSON so it can be parsed for the value of the `State` field. So before we start looking at an actual SLS file performing a test, let's look at the JSON output corresponding to the same command:

```
lab@vMX-1> show ospf neighbor | display json
{
    "ospf-neighbor-information" : [
    {
        "attributes" : {"xmlns" : "http://xml.juniper.net/junos/17.1R2/junos-routing"},
        "ospf-neighbor" : [
        {
            "neighbor-address" : [
            {
                "data" : "10.0.0.222"
            }
            ],
            "interface-name" : [
            {
                "data" : "ge-0/0/0.0"
            }
            ],
            "ospf-neighbor-state" : [
            {
                "data" : "Full"
            }
            ],
            "neighbor-id" : [
            {
                "data" : "192.168.0.2"
            }
            ],
            "neighbor-priority" : [
            {
                "data" : "128"
            }
            ],
            "activity-timer" : [
            {
                "data" : "31"
            }
            ]
        },
        {
            "neighbor-address" : [
            {
                "data" : "10.0.1.222"
            }
            ],
            "interface-name" : [
            {
                "data" : "ge-0/0/1.0"
            }
            ],
            "ospf-neighbor-state" : [
            {
                "data" : "Full"
            }
            ],
            "neighbor-id" : [
            {
                "data" : "192.168.0.2"
            }
            ],
            "neighbor-priority" : [
            {
                "data" : "128"
            }
            ],
            "activity-timer" : [
            {
                "data" : "39"
            }
            ]
        }
        ]
    }
    ]
}
```

From this output, the list of OSPF neighbors is located at the `['ospf-neighbor-information'][0]['ospf-neighbor']` level. Once you are processing the particular neighbor (for example, in a for-loop), the actual state is located at the `['ospf-neighbor-state'][0]['data']` level. Note how Junos OS commonly uses lists with strictly one element in JSON representations of RPC outputs, so you have to obtain only this list element with `[0]` indexing – this may be a bit confusing at first.

Additionally, the Junos RPC corresponding to the `show ospf neighbor` command is easily obtained using the Junos CLI:

```
lab@vMX-1> show ospf neighbor | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R2/junos">
    <rpc>
        <get-ospf-neighbor-information>
        </get-ospf-neighbor-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

So, you can see that the RPC name that you need is `get-ospf-neighbor-information`.

Now let's create the state SLS file that will perform the test – in this case you are testing for the presence of exactly two OSPF neighbors in the 'Full' state:

```
lab@master:/srv/salt$ cat test-ospf.sls

{% set ospf_neighbors = salt['junos.rpc']('get-ospf-neighbor-information', '', 'json') %}

{% set full_neighbors_count = {'count': 0} %}
{% set expected_neighbors_count = 2 %}

{% for neighbor in ospf_neighbors['rpc_reply']['ospf-neighbor-information'][0]['ospf-neighbor'] %}
  {% if neighbor['ospf-neighbor-state'][0]['data'] == 'Full' %}
     {% if full_neighbors_count.update({'count': full_neighbors_count.count + 1}) %}
     {% endif %}
  {% endif %}
{% endfor %}

Print results of OSPF neighbor status test:
  module.run:
    - name: test.echo
      text:

{% if full_neighbors_count.count == expected_neighbors_count %}
        - OSPF neighbor test passed with {{ full_neighbors_count.count }} full neighbors
{% else %}
        - OSPF neighbor test failed with {{ full_neighbors_count.count }} full neighbors
{% endif %}
```

As discussed previously, Salt state files are, by default, a combination of Jinja (template engine) and YAML (data serialization language). The following is a basic description of how this SLS state file is processed:

1. You call the `junos.rpc` execution function using `salt[<module-name>.<function-name>]` syntax. You pass parameters including RPC name and the desired output format (JSON). The result is stored to the Jinja variable named `ospf_neighbors`.
2. You create a variable named `full_neighbors_count`, which is a dictionary containing a single key named `'count'` with an initial value of 0. You need to use a dictionary because you will want to modify the counter in a for-loop.

3. The `expected_neighbors_count` variable will contain the expected number of Full OSPF neighbors (in this case, two).
4. Perform iteration (for-loop) over the list of OSPF neighbors extracted using RPC call. In the body of a loop, update `full_neighbors_count` counter if the state is `Full`.
5. Use the `module.run` state module that allows using execution functions in state files. In this case you just execute the `test.echo` module that prints a text message.
6. A different text message is printed depending on the value of `full_neighbors_count.count` variable – either the test passed or not.

Let's now execute the state:

```
lab@master:/srv/salt$ sudo salt vMX* state.apply test-ospf
vMX-1:
----------
          ID: Print results of OSPF neighbor status test
    Function: module.run
        Name: test.echo
      Result: True
     Comment: Module function test.echo executed
     Started: 05:57:28.657801
    Duration: 5.904 ms
     Changes:
              ----------
              ret:
                  - OSPF neighbor test passed with 2 full neighbors

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   5.904 ms
vMX-2:
----------
          ID: Print results of OSPF neighbor status test
    Function: module.run
        Name: test.echo
      Result: True
     Comment: Module function test.echo executed
     Started: 05:57:29.702003
    Duration: 6.652 ms
     Changes:
              ----------
              ret:
                  - OSPF neighbor test passed with 2 full neighbors
```

```
Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   6.652 ms
```

In this case, everything was fine in our small network so the tests passed. If you had OSPF neighbor states other than 'Full', then a test failure message would be shown.

A Variation in Python

The SLS file from a previous section was created as a Jinja+YAML file, which is Salt's default. However, you can use other template engines (for example, Mako) and other data formatting languages (for example, JSON). You can even create your SLS file as pure Python code, and that is what you will do in this section. So if you felt like too much logic was put in the Jinja template in the previous section, this way you can move all logic into Python code (alternatively, you could put Python code in a custom module, similar to what you did in Chapter 9) .

This next file, called `test-ospf-py.sls`, is actually doing the same thing as `test-ospf.sls` in the previous section, but it is now rewritten as Python code:

```
lab@master:/srv/salt$ cat test-ospf-py.sls
#! py

def run():
    full_neighbors_count = 0
    expected_neighbors_count = 2
    ospf_neighbors = __salt__['junos.rpc']('get-ospf-neighbor-information', '', 'json')
    for neighbor in ospf_neighbors['rpc_reply']['ospf-neighbor-information'][0]['ospf-neighbor']:
        if neighbor['ospf-neighbor-state'][0]['data'] == 'Full':
            full_neighbors_count += 1
    if full_neighbors_count == expected_neighbors_count:
        txt_result = "OSPF neighbor test passed with %s full neighbors " % full_neighbors_count
    else:
        txt_result = "OSPF neighbor test failed with %s full neighbors " % full_neighbors_count
    res = {
        "Print results of OSPF neighbor status test" :  {
            "module.run": [ { "name": "test.echo", "text": [ txt_result ] } ]
        }
    }
    return res
```

The explanation of how this SLS state file works:

1. The first line (`#!py`) instructs Salt that this is a Python SLS file so it must be processed accordingly.
2. The `run()` function is defined – it is a convention that SLS files written in Python must follow. The function must return the generated state content as a Python object.

3. Create variables named `full_neighbors_count` (counter for the number of OSPF neighbors in the 'Full' state) and `expected_neighbors_count` (the expected number of 'Full' OSPF neighbors - in this case, two).

4. You call the `junos.rpc` execution function using the `__salt__` special object. You pass parameters including RPC name and the desired output format (JSON). The result is stored to the Python variable named `ospf_neighbors`.

5. Perform iteration (for loop) over the list of OSPF neighbors extracted using RPC call. Update `full_neighbors_count` counter if a state is 'Full'.

6. Depending on the equality between the expected and actual number of OSPF neighbors, assign the value to `txt_results` variable (it basically contains the result of our test).

7. Create a `res` variable (Python dictionary, including some nested dictionaries and lists) matching closely the structure of the SLS file from the previous section (the YAML part of it), putting `txt_results` inside it.

8. Return `res`.

Now let's run the state file (to show a difference, one of the OSPF adjacencies was broken before the command was executed):

```
lab@master:/srv/salt$ sudo salt vMX* state.apply test-ospf-py
vMX-1:
----------
          ID: Print results of OSPF neighbor status test
    Function: module.run
        Name: test.echo
      Result: True
     Comment: Module function test.echo executed
     Started: 10:25:06.594210
    Duration: 7.697 ms
     Changes:
              ----------
              ret:
                  - OSPF neighbor test failed with 1 full neighbors

Summary for vMX-1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   7.697 ms
vMX-2:
----------
          ID: Print results of OSPF neighbor status test
    Function: module.run
        Name: test.echo
      Result: True
     Comment: Module function test.echo executed
     Started: 10:25:06.685827
    Duration: 5.817 ms
```

```
     Changes:
              ----------
              ret:
                  - OSPF neighbor test failed with 1 full neighbors

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   5.817 ms
```

Overall, the state file written in Python behaves the same way as the more common Jinja+YAML version.

As you might guess, it is possible to perform more complicated tests using the approach demonstrated in this chapter. Regardless of what testing methodology or framework you use, the more of your network's functionality is covered with tests, the better.

The next chapter covers another way of network testing, via integration of Salt with the JSNAPy tool.

Chapter 11

Automated Network Verifications with Salt and JSNAPy

This chapter explains how you can integrate Salt with JSNAPy, an open source tool automating network verifications for Junos.

Overview of JSNAPy

JSNAPy (Junos SNAPshot Administrator in Python) allows you to automatically collect and verify operational and configuration data from your Junos devices.

MORE? This chapter only reviews JSNAPy very briefly, and then covers integrating it with Salt. JSNAPy is explained in details in *Day One: Enabling Automated Network Verifications with JSNAPy* available at: https://www.juniper.net/dayone. Source code for this tool, with some additional examples and documentation, is available at https://github.com/Juniper/jsnapy.

To connect to devices, JSNAPy uses the NETCONF protocol (exploiting Junos PyEZ library under the hood). The data is collected in the form of *snapshots*, outputs of given commands or RPCs. Snapshots can be stored as files or records in a SQLite database. The four main operations of JSNAPy are:

- *Snap* – take a snapshot.
- *Check* – compare the two snapshots taken at different times (e.g. before and after maintenance) based on given test conditions. This is sometimes referred to as “snap-snap-check workflow”.
- *Snapcheck* – take a snapshot and compare it against predefined test conditions (“snapcheck workflow”).
- *Diff* – show the difference between two snapshots, without specifying test conditions.

For this chapter, a *snapcheck* workflow will be used.

At the time of this writing, JSNAPy is available in three forms: CLI tool, Ansible module, or Python module. The Junos execution and state modules for Salt do not have out-of-the-box JSNAPy integration, but this does not in any way prevent you from using JSNAPy with Salt. Really, with the flexibility of Salt, integration of this and possibly other tools (for example, some custom PyEZ scripts) is not hard to do.

Example Scenario

Let's show you how this integration can be done. The basic setup is as follows:

- The Salt reactor system, in response to Junos events pushed by the Junos syslog engine to the Salt event bus, invokes a reactor SLS file.
- The reactor SLS file references Salt *runner* (Python code executed on the master server), passing the required parameters (such as the device hostname for this event) to it.
- Salt runner imports JSNAPy Python library and calls JSNAPy methods as needed. This assumes the master server has PyEZ, JSNAPy, and the jxmlease libraries installed.

The setup is illustrated in Figure 11.1. Although not shown, JSNAPy, when invoked, actually connects to the Junos devices using NETCONF sessions initiated from the *master* server.

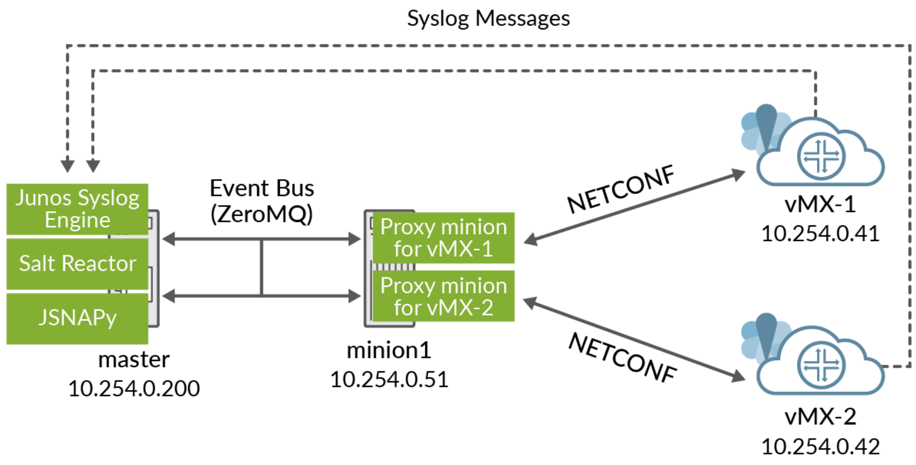


Figure 11.1 The Example Lab Setup for This Chapter.

In the example scenario that we consider, the system will react to commit messages (`UI_COMMIT_COMPLETED`) from vMX-1 and vMX-2 devices, and will perform a basic configuration audit check (for this example we will check that no Telnet service is enabled in the setup, as this service is insecure). The vMX device configurations are similar to ones in Chapter 7 (the Salt *master* server's address is configured as a destination for syslog messages).

The results of the test will be sent to the network administrator's email addresses using JSNAPy's built-in capabilities.

Install and Configure JSNAPy

To implement the outlined plan, first install the required packages (PyEZ, JSNAPy and jxmlease) on the master server:

```
lab@master:~$ sudo pip install junos-eznc
lab@master:~$ sudo python -m easy_install --upgrade pyOpenSSL
lab@master:~$ sudo pip install jxmlease
lab@master:~$ sudo pip install jsnapy
```

The output is omitted to save space. Note that the same PyEZ and jxmlease packages were installed and used in the previous chapters on the *minion1* server running the proxies – and only now are they needed on the master because we are going to use Salt runner, and runners are executed on the master server.

Now that JSNAPy is installed, edit its configuration file to make sure it looks like this:

```
lab@master:~$ sudo vi /etc/jsnapy/jsnapy.cfg

[DEFAULT]
config_file_path = /etc/jsnapy
snapshot_path = /etc/jsnapy/snapshots
test_file_path = /etc/jsnapy/testfiles
```

Also, in the directory `/etc/jsnapy/testfiles` containing the tests, put the test file with the following content:

```
lab@master:~$ cat /etc/jsnapy/testfiles/test_telnet.yml
tests_include:
  - test_telnet_config

test_telnet_config:
  - rpc: get-config
  - kwargs:
      filter_xml: configuration/system
  - item:
      xpath: system/services
      tests:
       - not-exists: telnet
         err: "Test Failed! Disallowed telnet service is configured."
         info: "Test passed."
```

This is a simple JSNAPy test checking for the presence of telnet service configuration in the `[edit system services]` hierarchy on a Junos device. As you can see, JSNAPy's test files are created in YAML and for the most part they are self-explanatory. In this case you are asking for `get-config` RPC output (also filtering out only the `system` configuration) and checking if the `telnet` element exists at the `system/services` level. Please consult the above cited book, or JSNAPy documentation, for more details and available options.

Also, in this chapter you will use JSNAPy's ability to notify you on the results of a test using email. So, create the following file and instead of XXXXX, enter you email parameters:

```
lab@master:~$ cat /etc/jsnapy/send_mail.yml
from: XXXXX
to: XXXXX
sub: JSNAPy results
recipient_name: Admin
sender_name: JSNAPy
server: XXXXX
passwd: XXXXX
```

This finishes our simple setup of JSNAPy. It will be called from a Python module, shown next.

Configure Runners and Reactors

Now, make sure you have the following in your master server configuration (`/etc/salt/master`):

```
engines:
  - junos_syslog:
      port: 10514

reactor:
  - 'jnpr/syslog/*/UI_COMMIT_COMPLETED':
    - salt://reactor/react_to_commit.sls

runner_dirs:
  - /srv/salt/runners
```

Here, the `engines` and `reactor` sections are the same as you had previously, in Chapters 7 and 8. Consult those chapters if you need a refresher. Generally, the reactor triggers an SLS file based on events. As it did previously, the system will be reacting to a Junos commit event, calling the `react_to_commit.sls` file.

The `runner_dirs` is a new section that you must add to the master configuration to be able to execute runners, so do not forget to do so, and then restart the Salt master. Also create the `/srv/salt/runners` directory which, in this example, will contain your runners.

The content of `react_to_commit.sls` file is as follows:

```
lab@master:~$ cat /srv/salt/reactor/react_to_commit.sls
Execute JSNAPy tests:
  runner.run_jsnapy.audit_config:
    - args:
        device: {{ data['hostname'] }}
```

Here, the SLS file calls a runner named `run_jsnapy`, in particular, a function named `audit_config` defined in it. The function will receive the `device` argument, whose value is extracted from the syslog message received on the event bus (the `data` variable).

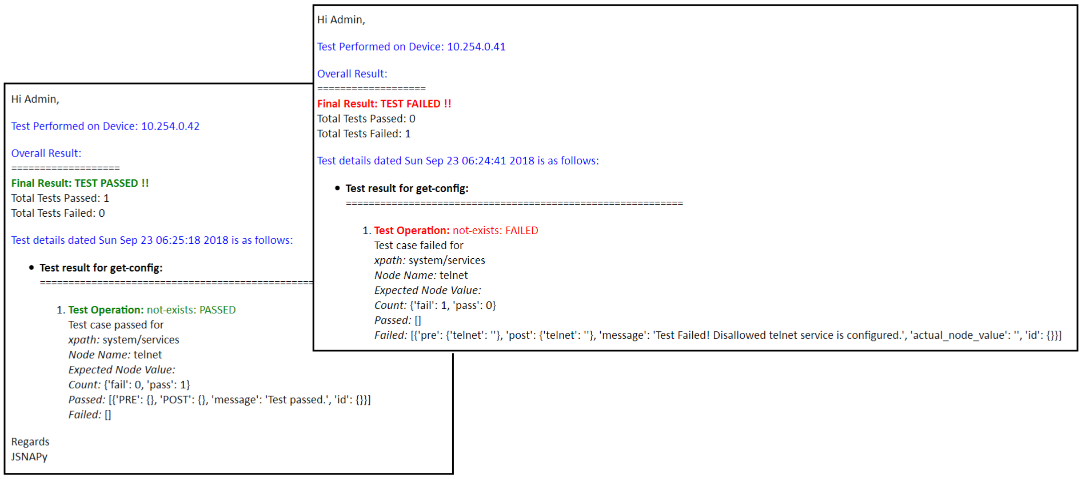
Now you can create a runner file:

```
lab@master:~$ cat /srv/salt/runners/run_jsnapy.py
# (1)
import salt.runner
from jnpr.jsnapy import SnapAdmin

# (2)
def audit_config(*args, **kvargs):

    # (3)
    CONF_DATA = """
tests:
  - test_telnet.yml

mail: /etc/jsnapy/send_mail.yml

hosts:
"""

    vMX_1_DATA = """
  - device: 10.254.0.41
    username: lab
    passwd: lab123
"""

    vMX_2_DATA = """
  - device: 10.254.0.42
    username: lab
    passwd: lab123
"""

    # (4)
    if "device" in kvargs:
        if kvargs["device"] == "vMX-1":
            CONF_DATA += vMX_1_DATA
        if kvargs["device"] == "vMX-2":
            CONF_DATA += vMX_2_DATA
    else:  # possibly called by salt-run
        CONF_DATA += (vMX_1_DATA + vMX_2_DATA)

    # (5)
    snapadmin = SnapAdmin()
    result = snapadmin.snapcheck(CONF_DATA, "pre")
```

And the explanation of how this Python code is processed:

1. Perform the necessary imports, in particular `SnapAdmin` class from `jnpr.jsnapy` library. This class allows calling JSNAPy from Python code.
2. Define `audit_config` function.
3. Create three string variables - CONF_DATA, vMX_1_DATA, vMX_2_DATA. The ultimate goal is to build a JSNAPy YAML configuration string depending on what devices we need to run tests on.
4. Depending on the "device" parameter passed to the function, either append vMX-1 or vMX-2 data to CONF_DATA, or append both (if no "device" is provided).
5. Create an instance of `SnapAdmin` class named `snapadmin` – it will be used to call JSNAPy from the Python code. Then, call `snapcheck` method passing CONF_DATA as a parameter (note that JSNAPy workflow used in this runner is "snapcheck"). Another parameter is a snapshot name (in this case "pre" – the name does not matter much in this particular scenario).

To test the setup, you perform a commit operation on vMX-1 device, and, if everything was done right, you receive an email with test results! Next, commit on vMX-2 gives you another email (see Figure 11.2). In this case, vMX-1 had Telnet enabled so the test failed. For vMX-2, the test passed as no telnet was in configuration.



```
Hi Admin,

Test Performed on Device: 10.254.0.41

Overall Result:
==================
Final Result: TEST FAILED !!
Total Tests Passed: 0
Total Tests Failed: 1

Test details dated Sun Sep 23 06:24:41 2018 is as follows:

  • Test result for get-config:
    ========================================================

     1. Test Operation: not-exists: FAILED
        Test case failed for
        xpath: system/services
        Node Name: telnet
        Expected Node Value:
        Count: {'fail': 1, 'pass': 0}
        Passed: []
        Failed: [{'pre': {'telnet': ''}, 'post': {'telnet': ''}, 'message': 'Test Failed! Disallowed telnet service is configured.', 'actual_node_value': '', 'id': {}}]
```

```
Hi Admin,

Test Performed on Device: 10.254.0.42

Overall Result:
==================
Final Result: TEST PASSED !!
Total Tests Passed: 1
Total Tests Failed: 0

Test details dated Sun Sep 23 06:25:18 2018 is as follows:

  • Test result for get-config:
    ==================================================

     1. Test Operation: not-exists: PASSED
        Test case passed for
        xpath: system/services
        Node Name: telnet
        Expected Node Value:
        Count: {'fail': 0, 'pass': 1}
        Passed: [{'PRE': {}, 'POST': {}, 'message': 'Test passed.', 'id': {}}]
        Failed: []

Regards
JSNAPy
```

Fig. 11.2 Example Emails From JSNAPy.

Alternatively, you can execute a test (on both devices) manually using the `salt-run` command as follows to get the same result (and yes, emails are also sent out):

```
lab@master:~$ sudo salt-run run_jsnapy.audit_config
Connecting to device 10.254.0.42 ................
Taking snapshot of RPC: get-config
**************************** Device: 10.254.0.42 ****************************
Tests Included: test_telnet_config
*************************RPC is get-config*************************
PASS | All "telnet" do not exists at xpath "system/services" [ 1 matched ]
------------------------------- Final Result!! -------------------------------
test_telnet_config : Passed
Total No of tests passed: 1
Total No of tests failed: 0
Overall Tests passed!!!
Connecting to device 10.254.0.41 ................
Taking snapshot of RPC: get-config
**************************** Device: 10.254.0.41 ****************************
Tests Included: test_telnet_config
*************************RPC is get-config*************************
Test Failed! Disallowed telnet service is configured.
FAIL |  "telnet" exists at xpath "system/services" [ 0 matched / 1 failed ]
------------------------------- Final Result!! -------------------------------
test_telnet_config : Failed
Total No of tests passed: 0
Total No of tests failed: 1
Overall Tests failed!!!
None
```

Summary

In this example, the results of the test were simply sent to the administrator's email for review. Performing other types of reactions, based on the test results, is certainly possible. This includes sending other types of notifications as well as trying some Junos device configuration self-healing.

We will not be demonstrating it here, but the basic outline for self-healing scenario could be as follows:

- The runner executing JSNAPy snapcheck analyzes the test result and based on that generates a custom Salt event.
- Salt reactor matches on that event and applies SLS file that performs the required configuration change.

Basically, anything can be done – but remember to keep your workflows as simple and clear as possible.

Chapter 12

Junos Automation with Salt and NAPALM

The previous chapters of this book showed you how Salt works with the Junos OS, using specialized execution and state modules that give you all the control and flexibility you need when managing Junos-based network devices.

Salt also includes NAPALM modules support for multivendor network device management, now discussed in this chapter.

NAPALM Review

NAPALM stands for Network Automation and Programmability Abstraction Layer with Multivendor support and allows you to manage Junos as well as some other vendors' equipment using an uniform interface.

NAPALM is open-source Python library, with source code available at https://github.com/napalm-automation/napalm and documentation at http://napalm.readthedocs.io/en/latest/.

Let's get our hands dirty with just a bit of Python NAPALM code before proceeding to Salt. On the *minion1* server, install NAPALM library using the pip tool:

```
lab@minion1:~$ sudo pip install napalm
```

This chapter also assumes that hostnames vMX-1 and vMX-2 are resolvable to corresponding IP addresses. You can achieve this by setting up DNS or just adding these entries to hosts file (do it on both *master* and *minion1*):

```
$ cat /etc/hosts
10.254.0.41 vMX-1
10.254.0.42 vMX-2
...
```

Now start an interactive Python session and load a NAPALM network driver for Junos (although NAPALM is cross-vendor, let's use Junos for the example here):

```
lab@minion1:~$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from napalm import get_network_driver
>>> driver = get_network_driver('junos')
```

Now create a device instance and open the connection to it:

```
>>> device = driver('vMX-1', 'lab', 'lab123')
>>> device.open()
```

Check the available object attributes with the `dir()` function – you can see you have a lot of options:

```
>>> dir(device)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__enter__', '__exit__', '__
format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__
reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__
weakref__', '_canonical_int', '_detect_config_format', '_get_address_family', '_is_json_format', '_
load_candidate', '_lock', '_netmiko_close', '_netmiko_open', '_parse_route_stats', '_parse_
value', '_rpc', '_unlock', 'cli', 'close', 'commit_config', 'compare_config', 'compliance_
report', 'config_lock', 'config_replace', 'connection_tests', 'device', 'discard_config', 'get_arp_
table', 'get_bgp_config', 'get_bgp_neighbors', 'get_bgp_neighbors_detail', 'get_config', 'get_
environment', 'get_facts', 'get_firewall_policies', 'get_interfaces', 'get_interfaces_
counters', 'get_interfaces_ip', 'get_ipv6_neighbors_table', 'get_lldp_neighbors', 'get_lldp_
neighbors_detail', 'get_mac_address_table', 'get_network_instances', 'get_ntp_peers', 'get_ntp_
servers', 'get_ntp_stats', 'get_optics', 'get_probes_config', 'get_probes_results', 'get_route_
to', 'get_snmp_information', 'get_users', 'hostname', 'ignore_warning', 'is_
alive', 'keepalive', 'key_file', 'load_merge_candidate', 'load_replace_candidate', 'load_
template', 'locked', 'open', 'password', 'ping', 'port', 'post_connection_tests', 'pre_connection_
tests', 'profile', 'rollback', 'ssh_config_file', 'timeout', 'traceroute', 'username']
```

Let's, for example, check the route to the 10.254.0.1 host:

```
>>> device.get_route_to("10.254.0.1")
{u'10.254.0.0/24': [{'protocol': u'Direct', 'last_active': True, 'outgoing_
interface': u'fxp0.0', 'current_active': True, 'routing_table': u'inet.0', 'next_
hop': None, 'selected_next_hop': True, 'preference': 0, 'inactive_
reason': u'', 'age': 4822293, u'protocol_attributes': {}}]}
```

The output is a Python object providing the same information that you could get from the Junos CLI as follows:

```
lab@vMX-1> show route 10.254.0.1

inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.254.0.0/24      *[Direct/0] 7w6d 19:40:31
                    > via fxp0.0
```

However, the major difference here is that NAPALM will present routing information in a uniform way for any supported network device, whether it runs Junos or something else.

Configuring NAPALM Proxy for Junos

To make Junos devices work with Salt via NAPALM modules, you will need to enable a NAPALM proxy in much the same way as you enabled Junos proxy minions in Chapter 3. The only difference is in the pillar configuration, which now will be like this:

```
lab@master:~$ cat /srv/pillar/proxy-1.sls
proxy:
  proxytype: napalm
  driver: junos
  fqdn: vMX-1
  username: lab
  password: lab123

lab@master:~$ cat /srv/pillar/proxy-2.sls
proxy:
  proxytype: napalm
  driver: junos
  fqdn: vMX-2
  username: lab
  password: lab123
```

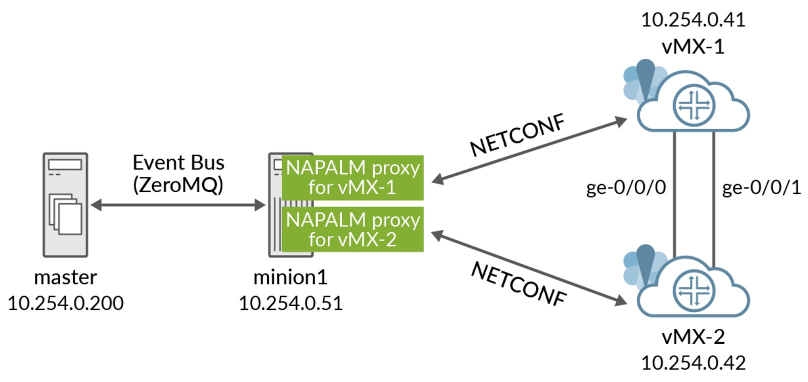


Figure 12.1 Lab Setup Used in This Chapter

See Fig. 12.1 for an illustration. Similar to what you did before, you will also need to start the Salt proxy processes on the *minion1* server, for example:

```
lab@minion1:~$ sudo killall salt-proxy
lab@minion1:~$ sudo salt-proxy --proxyid=vMX-1 -d
lab@minion1:~$ sudo salt-proxy --proxyid=vMX-2 -d
```

Perform the basic connectivity check to make sure everything was set up properly:

```
lab@master:~$ sudo salt vMX* test.ping
vMX-1:
    True
vMX-2:
    True
```

Salt uses multiple modules that exploit NAPALM – this includes the NET execution module for basic functions such as configuration load and commit, the routes execution module, the BGP execution module, the Netconfig state module, and many more. NAPALM also works with YANG models, including OpenConfig, allowing you to create and apply vendor-neutral network device configurations.

MORE? For a complete list of modules you can use for network automation with Salt, see: https://docs.saltstack.com/en/latest/topics/network_automation/index.html.

Now use the NAPALM route module to get a route to 10.254.0.1, similar to what you did with Python in the previous section:

```
lab@master:~$ sudo salt vMX-1 route.show 10.254.0.1 --out json
{
    "vMX-1": {
        "comment": "",
        "result": true,
        "out": {
            "10.254.0.0/24": [
                {
                    "protocol": "Direct",
                    "last_active": true,
                    "current_active": true,
                    "age": 4825895,
                    "routing_table": "inet.0",
                    "next_hop": null,
                    "outgoing_interface": "fxp0.0",
                    "preference": 0,
                    "selected_next_hop": true,
                    "protocol_attributes": {},
                    "inactive_reason": ""
                }
            ]
        }
    }
}
```

Uploading Device Configurations

Now let's load some basic configuration – in this example you want to enable LLDP on two interfaces for both vMX devices in the topology (needless to say, you could also use some advanced templating here – for example, Jinja syntax using pillar data, as demonstrated in previous chapters, but we will limit ourselves to a very simple example):

```
lab@master:~$ sudo cat /srv/salt/lldp.conf
protocols {
    lldp {
        interface ge-0/0/0;
        interface ge-0/0/1;
    }
}
```

Now create a state file as follows:

```
lab@master:~$ cat /srv/salt/lldp.sls
Enable LLDP:
  netconfig.managed:
    - template_name: salt://lldp.conf
```

Here, “Enable LLDP” is a state name and you are referencing the NAPALM `netconfig.managed` state function, providing the path to template (configuration) file as an argument.

Now you can apply the state:

```
lab@master:~$ sudo salt vMX* state.apply lldp
vMX-2:
----------
          ID: Enable LLDP
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 05:39:36.002885
    Duration: 3114.969 ms
     Changes:
              ----------
              diff:
                  [edit protocols]
                  +   lldp {
                  +       interface ge-0/0/0;
                  +       interface ge-0/0/1;
                  +   }

Summary for vMX-2
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.115 s
vMX-1:
----------
          ID: Enable LLDP
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 05:39:35.997927
    Duration: 4450.342 ms
     Changes:
              ----------
              diff:
                  [edit protocols]
                  +   lldp {
                  +       interface ge-0/0/0;
                  +       interface ge-0/0/1;
                  +   }

Summary for vMX-1
------------
```

```
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   4.450 s
```

You can see the configuration was applied successfully. You can also get the state of the now-configured LLDP protocol using the LLDP NAPALM execution module, as follows:

```
lab@master:~$ sudo salt vMX* net.lldp --out json
{
    "vMX-2": {
        "comment": "",
        "result": true,
        "out": {
            "ge-0/0/1": [
                {
                    "remote_port_description": "ge-0/0/1",
                    "remote_port": "519",
                    "remote_system_
description": "Juniper Networks, Inc. vmx internet router, kernel JUNOS 17.1R2.7, Build date: 2017-
06-17 09:46:28 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.",
                    "remote_chassis_id": "00:05:86:1B:0C:C0",
                    "remote_system_name": "vMX-1",
                    "parent_interface": "-",
                    "remote_system_capab": "Bridge Router",
                    "remote_system_enable_capab": "Bridge Router"
                }
            ],
            "ge-0/0/0": [
                {
                    "remote_port_description": "ge-0/0/0",
                    "remote_port": "518",
                    "remote_system_
description": "Juniper Networks, Inc. vmx internet router, kernel JUNOS 17.1R2.7, Build date: 2017-
06-17 09:46:28 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.",
                    "remote_chassis_id": "00:05:86:1B:0C:C0",
                    "remote_system_name": "vMX-1",
                    "parent_interface": "-",
                    "remote_system_capab": "Bridge Router",
                    "remote_system_enable_capab": "Bridge Router"
                }
            ]
        }
    }
}
{
    "vMX-1": {
        "comment": "",
        "result": true,
        "out": {
            "ge-0/0/1": [
                {
                    "remote_port_description": "ge-0/0/1",
                    "remote_port": "519",
                    "remote_system_
description": "Juniper Networks, Inc. vmx internet router, kernel JUNOS 17.1R2.7, Build date: 2017-
06-17 09:46:28 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.",
                    "remote_chassis_id": "00:05:86:C9:68:C0",
                    "remote_system_name": "vMX-2",
                    "parent_interface": "-",
                    "remote_system_capab": "Bridge Router",
                    "remote_system_enable_capab": "Bridge Router"
                }
            ],
            "ge-0/0/0": [
                {
                    "remote_port_description": "ge-0/0/0",
                    "remote_port": "518",
                    "remote_system_
description": "Juniper Networks, Inc. vmx internet router, kernel JUNOS 17.1R2.7, Build date: 2017-
06-17 09:46:28 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.",
                    "remote_chassis_id": "00:05:86:C9:68:C0",
                    "remote_system_name": "vMX-2",
                    "parent_interface": "-",
                    "remote_system_capab": "Bridge Router",
                    "remote_system_enable_capab": "Bridge Router"
                }
            ]
        }
    }
}
```

Note that both the way you applied the configuration using `netconfig` module, and the way you checked LLDP status, were completely vendor- and OS-independent.

Summary

As normally happens when abstraction layers are added, with NAPALM your work becomes more high-level. You don't have to think of fine details, such as the vendor-specific ways of loading configurations, and operational data you get from different devices is now uniform.

Compared to working with specialized vendor modules there may be some drawbacks as well, as you may lose some flexibility; just choose the approach that best fits your needs.

NOTE The principles discussed in the previous chapters of this book, including remote execution, configuration management, and the basics of EDI, remain the same if you choose to use the Salt NAPALM modules.

Appendix

References

For your convenience this Appendix is an entire list of references to resources that were used during preparation of the book, many of which were cited within the text, that you can consult for additional examples and explanations.

This Book's Git Repository

https://github.com/pklimai/day-one-junos-salt

SaltStack Official Documentation

Main documentation page: https://docs.saltstack.com/en/latest/

Junos execution modules for Salt: https://docs.saltstack.com/en/latest/ref/modules/all/salt.modules.junos.html

Junos state modules for Salt: https://docs.saltstack.com/en/latest/ref/states/all/salt.states.junos.html

Junos proxy for Salt: https://docs.saltstack.com/en/latest/ref/proxy/all/salt.proxy.junos.html#module-salt.proxy.junos

Junos Syslog engine for Salt: https://docs.saltstack.com/en/latest/ref/engines/all/salt.engines.junos_syslog.html

Network automation with Salt documentation page: https://docs.saltstack.com/en/latest/topics/network_automation/index.html

YouTube Videos on Junos Automation with Salt

Part 1 - Introduction to SaltStack: https://www.youtube.com/watch?v=JK7z6xnj1k0

Part 2 - Junos specific Salt components (Junos proxy, execution modules and state modules): https://www.youtube.com/watch?v=QE1l8OMwjQU

Part 3 - Junos Syslog Engine and reactors: https://www.youtube.com/watch?v=QFU6RzCgG4I

Useful GitHub Repositories

Hands on Labs around Junos Event Driven automation: https://github.com/ksator/automation_summit_Q3_2018

Junos automation with Salt (presentations, FAQ and Wiki): https://github.com/ksator/junos-automation-with-saltstack

Automated Tickets management with Junos Syslog using Salt: https://github.com/JNPRAutomate/automated_tickets_management_with_syslog_saltstack_RT

Junos Automation with Juniper's AppFormix and Salt: https://github.com/JNPRAutomate/automated_junos_configuration_changes_with_appformix_saltstack

Junos OS show command output collection using Salt: https://github.com/JNPRAutomate/automated_junos_show_commands_collection_with_syslog_saltstack

Junos OS Configuration Continuous Backup with Salt: https://github.com/JNPRAutomate/automated_junos_configuration_backup_with_syslog_saltstack

Junos OS automation demo with Appformix, SaltStack and GitLab: https://github.com/JNPRAutomate automated_junos_show_commands_collection_with_appformix_saltstack
Junos OS automation demo with Appformix, SaltStack and Northstar: https://github.com/JNPRAutomate/network_anomalies_automated_remediation_with_appformix_northstar_saltstack

Juniper Day One Books Library

https://www.juniper.net/dayone

To learn more about Junos PyEZ, check out *Day One: Junos PyEZ Cookbook*.

In addition to covering Ansible, *Day One: Automating Junos with Ansible, 2nd Edition*, introduces you to such topics as XML, JSON, YAML, and the Git version control system.

Day One: Enabling Automated Network Verifications with JSNAPy covers the JSNAPy tool.