O'REILLY®

Compliments of CLOUDFLARE®

Network Automation at Scale

Mircea Ulinic & Seth House

Network Automation at Scale

Mircea Ulinic and Seth House

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Network Automation at Scale
by Mircea Ulinic and Seth House

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Courtney Allen and Jeff Bleiel
Production Editor: Kristen Brown
Copyeditor: Jasmine Kwityn
Interior Designer: David Futato
Cover Designer: Karen Montgomery

October 2017: First Edition

Revision History for the First Edition
2017-10-10: First Release

978-1-491-99249-4

[LSI]

Table of Contents

CHAPTER 1
Introduction

Network automation is a continuous process of generation and deployment of configuration changes, management, and operations of network devices. It often implies faster configuration changes across a significant amount of devices, but is not limited to only large infrastructures. It is equally important when managing smaller deployments to ensure consistency with other devices and reduce the human-error factor. Automation is more than just configuration management; it is a broad area that also includes data collection from the devices, automatic troubleshooting, and self-resilience—the network can become smart enough to remediate the problems by itself, depending on internal or external factors.

When speaking about network automation, there are two important classes of data to consider: configuration and operational. *Configuration data* refers to the actual state of the device, either the entire configuration or the configuration of a certain feature (e.g., the configuration of the NTP peers, interfaces, BGP neighbors, MPLS etc.). On the other hand, *operational data* exposes information and statistics regarding the result of the configuration—for example, synchronization of the NTP peers, the state of a BGP session, the MPLS LSP labels generated, and so on. Although most vendors expose this information, their representation is different (sometimes even between platforms produced by the same vendor).

In addition to these multivendor challenges, there are others to be considered. Traditionally, a network device does not allow running custom software; most of the time, we are only able to configure and

use the equipment. For this reason, in general, network devices can only be managed remotely. However, there are also vendors producing whitebox devices (e.g., Arista, Cumulus, etc.), or others that allow containers (e.g., Cisco IOS-XR, Cisco NX-OS in the latest versions).

Regardless of the diversity of the environment and number of platforms supported, each network has a common set of issues: configuration generation and deployment, equipment replacement (which becomes very problematic when migrating between different operating systems), human errors and unmonitored events (e.g., BGP neighbor torn down due to high number of receiving prefixes, NTP unsynchronized, flapping interfaces, etc.). In addition, there is the responsibility of implicitly *reacting* to these issues and applying the appropriate configuration changes, searching for important details, and carrying out many other related tasks.

Large networks bring these challenges to an even higher complexity level: the tools need to be able to scale enough to manage the entire device fleet, while the network teams are bigger and the engineers need to access the resources concurrently. At the same time, everything needs to be accessible for everyone, inclusively for network engineers that do not have extensive software skills. The tooling basis must be easily configurable and customizable, in such a way that it adapts depending on the environment. Large enterprise networks are heterogeneous in that they are built from various vendors, so being able to apply the same methodologies in a cross-platform way is equally important.

Network automation is currently implemented using various frameworks, including Salt, Ansible, Chef, and Puppet. In this book we will focus on Salt, due to its unique capabilities, flexibility, and scalability. Salt includes a variety of features out of the box, such as a REST API, real-time jobs, high availability, native encryption, the ability to use external data even at runtime, job scheduling, selective caching, and many others. Beyond these capabilities, Salt is perhaps the most scalable framework—there are well-known deployments in companies such as LinkedIn that manage many tens of thousands of devices using Salt.

Another particularity of network environments is dynamicity—there are many events continuously happening due to internal or external causes. For example, an NTP server might become

unreachable, causing the device to become unsynchronized, a BGP neighbor to be torn down, and an interface optical transceiver unable to receive light; in turn, a BGP neighbor could leak routes, leaving the device vulnerable to an attacker's attempt to log in and cause harm—the list of examples can go on and on. When unmonitored, these events can sometimes lead to disastrous consequences. Salt is an excellent option for event-driven network automation and orchestration: all the network events can be imported into Salt, interpreted, and eventually trigger configuration changes as the business logic imposes. Unsurprisingly, large-scale networks can generate many millions of important events per hour, which is why scalability is even more important.

The vendor-agnostic capabilities of Salt are leveraged through a third-party library called NAPALM, a community-maintained network automation platform. We will briefly present NAPALM and review its characteristics in "Introducing NAPALM" on page 6.

Automating networks using Salt and NAPALM requires no special software development knowledge. We will use YAML as the data representation language and Jinja as the template language (there are six simple rules—three YAML, three Jinja—as we will discuss in "Brief Introduction to Jinja and YAML" on page 8). In addition, there are some details are Salt-specific configuration details, covered step by step in the following chapters so that you can start from scratch and set up a complex, event-driven automation environment.

Salt and SaltStack

Salt is an open source (Apache 2 licensed), general-purpose automation tool that is used for managing systems and devices. Out of the box, it ships with a number of capabilities: Salt can run arbitrary commands, bring systems up to a desired configuration, schedule jobs, react in real time to events across an infrastructure, integrate with hundreds of third-party programs and services across dozens of operating systems, coordinate complex multisystem orchestrations, feed data from an infrastructure into a data store, extract data from a data store to distribute across an infrastructure, transfer files securely, and even more.

SaltStack is the company started by the creator of Salt to foster development and help ensure the longevity of Salt, which is heavily

used by very large companies around the globe. SaltStack provides commercial support, professional services and consulting, and an enterprise-grade product that makes use of Salt to present a higher-level graphical interface and API for viewing and managing an infrastructure, particularly in team environments.

Speed is a top priority for SaltStack. As the company writes on its website:

> In SaltStack, speed isn't a byproduct, it is a design goal. SaltStack was created as an extremely fast, lightweight communication bus to provide the foundation for a remote execution engine.

Exploring the Architecture of Salt

The core of Salt is the encrypted, high-speed communication bus referenced in the quote above as well as a deeply integrated plug-in interface. The bulk of Salt is the vast ecosystem of plug-in modules that are used to perform a wide variety of actions, including remote execution and configuration management tasks, authentication, system monitoring, event processing, and data import/export.

Salt can be configured many ways, but the most common is using a high-speed networking library, ZeroMQ, to establish an encrypted, always-on connection between servers or devices across an infrastructure and a central control point called the Salt master. Massive scalability was one design goal of Salt and a single master on moderate hardware can be expected to easily scale to several thousand nodes (and up to tens of thousands of nodes with some tuning). It is also easy to set up with few steps and good default settings; first-time users often get a working installation in less than an hour.

Salt minions are servers or devices running the Salt daemon. They connect to the Salt master, which makes deployment a breeze since only the master must expose open ports and no special network access need be given to the minions. The master can be configured for high availability (HA) via Salt's multimaster mode, or in a tiered topology for geographic or logical separation via the Syndic system. There is also an optional SSH-based transport and a REST API.

Once a minion is connected to a master and the master has accepted the public key for that minion the two can freely communicate over an encrypted channel. The master will broadcast commands to minions and minions will deliver the result of those commands back to the master. In addition, minions can request files from the master

and can continually send arbitrary events such as system health information, logs, status checks, or system events, to name just a few.

Diving into the Salt Proxy Minion

As mentioned earlier, one of the challenges when managing network equipment is installing and executing custom software. Whitebox devices or those operating systems allowing containers could potentially allow installing the `salt-minion` package directly. But a traditional device can only be controlled remotely, via an API or SSH.

Introduced in Salt 2015.8 (codename *Beryllium*), proxy minions leverage the capabilities of the regular minions (with particular configuration) and make it possible to control devices such as network gear, devices with limited CPU or memory, or others. They are basically a virtual minion, a process capable of running anywhere in order to control devices remotely via SSH, HTTP, or other transport mechanism.

To avoid confusion caused by nomenclature similarities with other frameworks, a proxy minion is not another machine, it is just one process associated with the device managed, thus one process per device. It is usually lightweight, consuming about 60 MB RAM.

An intrinsic property of the proxy minions is that the connection with the remote device is always kept alive. However, they can also be designed to establish the connection only when necessary, or even let the user decide what best fits their needs (depending on how dynamic the environment is).

Because the list of device types that can be controlled through the proxy minions can be nearly infinite, each having their own properties and interface characteristics, a module (and sometimes a third-party library) is required.

Beginning with 2016.11 (*Carbon*), there are several proxy modules included, four of them aiming to manage network gear:

- NAPALM (covered briefly in "Introducing NAPALM" on page 6)
- Junos (provided by Juniper, to manage devices running Junos)
- Cisco NXOS (for Cisco Nexus switches)

- Cisco NSO (interface with Cisco Network Service Orchestrator)

Installing Salt: The Easy Way

SaltStack supports and maintains a shell script called Salt Bootstrap that eases the installation of the Salt master and minion on a variety of platforms. The script determines the operating system and version, then executes the necessary steps to install the Salt binaries in the best way for that system.

Therefore, the installation becomes as easy as:

```
wget -O bootstrap-salt.sh https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh
```

This fetches the *bootstrap-salt.sh* script from *https://bootstrap.saltstack.com*, then installs the Salt minion.

If you want to also install the Salt master, you only need to append the `-M` option:

```
wget -O bootstrap-salt.sh https://bootstrap.saltstack.com
sudo sh bootstrap-salt.sh -M
```

Introducing NAPALM

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is an open source Python library that accommodates a set of methodologies for configuration management and operational data retrieval, uniformly, covering several network operating systems, including Junos, Cisco IOS-XR, Cisco IOS, Cisco NX-OS, and Arista EOS.

There are other community-driven projects for Cumulus Linux, FortiOS, PAN-OS, MikoTik RouterOS, Pluribus Netvisor, and many others can be provided or adapted in the user's own environment.

The operational data is represented in cross-vendor format. For instance, retrieving the BGP neighbors from a device running Junos, the output is a Python dictionary with the format shown in Example 1-1.

Example 1-1. NAPALM output sample from a Junos device

```
{
  'global': {
    'peers': {
      '192.168.0.2': {
        'address_family': {
          'ipv4': {
            'accepted_prefixes': 142,
            'received_prefixes': 142,
            'sent_prefixes': 0
          }
        },
        'description': 'Amazon',
        'is_enabled': True,
        'is_up': True,
        'local_as': 13335,
        'remote_as': 16509,
        'remote_id': '10.10.10.1',
        'uptime': 8816095
      }
    }
  }
}
```

The output has exactly the same structure if retrieving the BGP neighbors, using NAPALM, from a Cisco IOS-XR router or Arista switch. The same characteristics are available for the rest of the NAPALM features whose structure is available in the NAPALM documentation.

Similarly, configuration management is also cross-vendor and although the configuration loaded depends on the network OS, the methodology is the same for all platforms. For example, when applying manual configuration changes using the CLI of Cisco IOS, the changes will be reflected directly into the running-config. But using NAPALM, the changes are stored in a buffered candidate configuration and transferred into the running config only when an explicit *commit* is performed. The following methods are defined for configuration management:

Method name	Method description
`load_merge_candidate`	Populate the candidate config, either from file or text.
`load_replace_candidate`	Similar to `load_merge_candidate`, but instead of a merge, the existing configuration will be entirely replaced with the content of the file, or the configuration loaded as text.

Method name	Method description
`compare_config`	Return the difference between the running configuration and the candidate.
`discard_config`	Discards the changes loaded into the candidate configuration.
`commit_config`	Commit the changes loaded using `load_merge_candidate` or `load_replace_candidate`.
`rollback`	Revert the running configuration to the previous state.

NAPALM is distributed via *PyPI* (Python Package Index), which is the official repository for third-party Python libraries. The installation is usually as simple as running `$ pip install napalm`; however, the user might need to consider several system dependencies. For Salt users, the process is simplified through the napalm install formula, which performs the required steps to install the underlying packages.

The NAPALM Proxy

Beginning with Salt 2016.11 (Carbon) the cross-vendor capabilities of NAPALM have been integrated in Salt, allowing the network engineers to introduce the DevOps methodologies without worrying about the multivendor issues.

The initial implementation was based exclusively on the NAPALM proxy minion module. In 2017.7.0 (Nitrogen) the capabilities have been extended, allowing the NAPALM modules to run under a regular minion as well. In other words, if the device operating system permits, the `salt-minion` package can be installed directly on the device and then leverage the network automation methodologies through NAPALM such as controlling network devices like servers. For example, there is a SWIX extension for Arista devices to facilitate the installation of the minion directly on the switch; see the SaltStack docs.

Brief Introduction to Jinja and YAML

Before diving into Salt-specific details, let's have a look at two of the most widely adopted template and data representation languages: Jinja and YAML. Salt uses them by default, so it's important that you understand the basics.

The Three Rules of YAML

Yet Another Markup Language (YAML) is a human-readable data representation language. Three easy rules are enough to get started, but for more in-depth details we encourage you to explore the YAML documentation as well as the YAML troubleshooting tips.

Rule #1: Indentation

YAML uses a fixed indentation scheme to represent relationships between data layers. Salt requires that the indentation for each level consists of *exactly two spaces*. Do not use tabs.

Rule #2: Colons

Colons are used in YAML to represent hashes, or associative arrays—in other words, one-to-one mappings between a key and a value. For example, to assign the value `xe-0/0/0` to the `interface_name` field:

```
interface_name: xe-0/0/0
```

The same rule can be extended to a higher level and use nested key-value pairs where we can notice the usage of the indentation:

```
interface:
  name: xe-0/0/0
  shutdown: false
  subinterfaces:
    xe-0/0/0.0:
      ipv4:
        address: 172.17.17.1/24
```

Rule #3: Dashes

Dashes are used to represent a list of items. For example:

```
interfaces:
  - fa1/0/0
  - fa4/0/0
  - fa5/0/0
```

Note the single space following the hyphen.

The Three Rules of Jinja

Jinja is a widely used templating language for Python. Like any template engine, it uses abstract models and data to generate

documents. While Jinja can be quite complex, three simple rules will suffice to get started with using it.

Rule #1: Double curly braces

Double curly braces means the replacement of a variable with its value. For instance, the template in Example 1-2 will generate the result in Example 1-3 when the variable `interface_name` has the value `xe-0/0/0`.

Example 1-2. Example of double curly braces

```
interface {{ interface_name }}
```

Example 1-3. Rendering result of Jinja curly braces

```
interface xe-0/0/0
```

We will see later how you can send the variables to the template. For the moment the most important thing to note is that the output is plain text where the `{{ interface_name }}` has been replaced with the value of the `interface_name` variable.

Rule #2: Conditional tests

Conditional operators can be used to make decisions and generate different documents or parts of a document. The syntax of an if-elif-else conditional test is as follows:

```
{% if interface_name == 'xe-0/0/0' %}
The interface is 10-Gigabit Ethernet.
{% elif interface_name == 'ge-0/0/0' %}
The interface is Gigabit Ethernet.
{% else %}
Different type.
{% endif %}
```

In this example, the template will generate the output "The interface is 10-Gigabit Ethernet." when the variable `interface_name` is `xe-0/0/0`, or "The interface is Gigabit Ethernet." when the variable `interface_name` has the value `ge-0/0/0`, respectively.

Note that the `endif` keyword at the end of the block is mandatory. The `{%` marks the beginning of a Jinja instruction. This will also insert an additional blank line. To avoid this it can be written as

{%- instead. Similarly, to avoid a new line at the end the instruction can be written as -%}.

Rule #3: Loops

Looping through a list of values has the format shown in Example 1-4, which generates the text in Example 1-5 when the variable `interfaces` is an array containing the values `['fa1/0/0', 'fa4/0/0', 'fa5/0/0']`.

Example 1-4. Example of Jinja template loop

```
{% for interface_name in interfaces -%}
interface {{ interface_name }}
  no shut
{% endfor -%}
```

Example 1-5. Rendering result of Jinja loop

```
interface fa1/0/0
  no shut
interface fa4/0/0
  no shut
interface fa5/0/0
  no shut
```

For more advanced topics, consult the Jinja documetnation. The following chapters will cover some other Salt-specific advanced templating methodologies.

Extensible and Scalable Configuration Files: SLS

One of the most important characteristics of Salt is that data is key, not the representation of that data. `SLS` (SaLt State) is the file format used by Salt. By default it is a mixture of Jinja and YAML (i.e., YAML generated from a Jinja template), but flexible enough to allow other combinations of template and data representation languages.

The SLS files can be equally complex Jinja templates that translate down to YAML or they can just be plain and simple YAML files. For instance, let's see how we would declare a list of interfaces for a Juniper device in an SLS file (Example 1-6).

Example 1-6. Sample SLS file: Plain YAML

```
interfaces:
  - xe-0/0/0
  - xe-0/0/1
  - xe-0/0/2
  - xe-0/0/3
  - xe-0/0/4
```

The same list can be generated dynamically using Jinja and YAML (Example 1-7).

Example 1-7. Sample SLS file: Jinja and YAML

```
interfaces:
{% for index in range(5) -%}
  - xe-0/0/{{ index }}
{% endfor -%}
```

Both of these representations are interpreted by Salt in the same way, but the usage of Jinja together with YAML makes the code in Example 1-7 more flexible. Although the list shown here is very short, this methodology proves really helpful when generating dynamic content, as it saves you from having to manually write a long file.

The user can choose between the following template languages: Jinja (default), Mako, Cheetah, Genshi, Wempy, or Py (which is the pure Python renderer). Similarly, there is a variety of data representation languages that can be used: YAML (default), YAMLEX, JSON, JSON5, HJSON, or Py (pure Python). Even more, the user can always extend the capabilities and define a custom renderer in their private environment—and eventually open source it.

NOTE

The Salt rendering pipeline processes template rendering first to produce the data representation, which is then given to Salt's State compiler. By default the SLS first renders the Jinja content followed by translating the YAML into a Python object.

Alternative renderers can be enabled by adding a hashbang at the top of the SLS file. For example, using the hashbang `#!mako|json` will instruct Salt to interpret the SLS file using Mako and JSON. In that case, the SLS file would be written as shown in Example 1-8.

Example 1-8. Sample SLS file: Mako and JSON

```
#!mako|json
{
  "interfaces": [
    % for index in range(5):
    "xe-0/0/${index}",
    % endfor
  ]
}
```

Without moving the focus to Mako, the most important detail to note is the flexibility of the SLS file. Moreover, if the user has even more specific needs the good news is that the renderers are one of the many pluggable interfaces of Salt, hence a new renderer can be added very easily.

The `#!jinja|yaml` header is implicit.

Sensitive data can be natively encrypted using GPG and Salt will decrypt it during runtime.

When inserting GPG-encrypted data it is necessary to explicitly use the hashbang with the appropriate template and data representation languages. For example, even if we work with the default Jinja/YAML combination, the header needs to be `#!jinja|yaml|gpg`.

The GPG renderer has more specific configuration requirements, in particular on the master, but they are beyond the scope of this book. For more information, consult the setup notes.

Remarkably, the SLS file can even be written in pure Python. For instance, the list of interfaces from before could be rewritten as shown in Example 1-9.

Example 1-9. Sample SLS file: Pure Python

```
#!py

def run():
    return [ 'xe-0/0/{}'.format(index)
             for index in range(5) ]
```

The pure Python renderer is extremely powerful; it is basically limited only by Python itself. The only constraint is to define the `run` function, which returns a JSON-serializable object that constitutes our data. We can go even further and design a very intelligent SLS file that builds its content based on external services or sources.

For instance, as shown in Example 1-10, we can build the list of interfaces dynamically by retrieving the data from a REST API found at the URL *https://interfaces-api*.

Example 1-10. The flexiblity of the pure Python renderer

```
#!py

import requests

def run():
    ret = requests.get('https://interfaces-api')
    return ret.json()
```

We will see later how the SLS files can be consumed and how we can leverage their power to help us with automating.

CHAPTER 2

Preparing the Salt Environment

This chapter focuses on the essential steps for preparing the environment to start automating using Salt. We will first present some of the most important Salt-specific keywords and their meaning. Following that, we'll take a look at the main configuration files used to control the behavior of Salt's processes. Finally, we'll review the processes startup, which implies the completion of the environment setup.

Salt Nomenclature

Salt comes with a particular nomenclature that requires a careful review of the documentation to fully understand. In general the documentation is very good and complete, very often providing usage examples, however much of it is written for an audience that already knows Salt basics and only needs to know how a particular module or interface is configured or called.

Pillar

Pillar is free-form data that can be used to organize configuration values or manage sensitive data. It is an entity of data that can be either stored locally using the filesystem, or using external systems such as databases, Vault, Amazon S3, Git, and many other resources (see "Using External Pillar" on page 20). Simple examples of pillar data include a list of NTP peers, interface details, and BGP configuration.

When defined as SLS files they follow the methodologies described under "Extensible and Scalable Configuration Files: SLS" on page 11, and the data type is therefore a Jinja/YAML combination, by default, but different formats can be used if desired. For very complex use cases, we even have the option of using a pure Python renderer.

include, exclude, extend

In order to avoid rewriting the same content repeatedly when working with SLS files, three keywords can be used: `include`, `exclude`, and `extend`. Here we will cover only the `include` statement. The other two work in a similar way and they can be further explored by consulting the SaltStack documentation.

The content of a different SLS file can be included by specifying the name, *without* the *.sls* extension. For example, if we organize *ntp_config.sls*, *router1.sls*, and *router2.sls* all in the same directory then we can include the contents of *npt_config.sls* into the *router1.sls* and *router2.sls* pillar with the syntax shown in Example 2-1.

Example 2-1. Sample pillar SLS include file (device1_pillar.sls)

```
include:
  - ntp_config
```

Note that `include` accepts a list, so we are able to include the content from multiple SLS files.

The inclusion can also be relative using the common `.` (dot) and `..` syntax seen in navigating through a directory hierarchy.

Configuring the NAPALM Pillar

To set up a NAPALM proxy, the pillar should contain the following information:

`driver`
: The name of the NAPALM driver

`host`
: FQDN or IP address to use when connecting to the device (alternatively, this value can be specified using the `fqdn`, `ip`, or `hostname` fields)

`username`
: Username to be used when connecting to the device

`password`
: Password required to establish the connection (if the driver permits, the authentication is established using a SSH key and this field can be blank)

Additionally, we can also specify other parameters such as `port`, `enable_password`, and so on using the `optional_args` field. Refer to the NAPALM documentation for the complete list of optional arguments.

In Examples 2-2 and 2-3, we provide two sample pillar files to manage different operating systems, Junos and EOS. The same format can be used to manage Cisco IOS devices. Note the usage of the `proxytype` field, which must be configured as `napalm` to tell Salt that the NAPALM proxy modules are going to be used.

Example 2-2. Sample file for Juniper router managed using NAPALM (device1_pillar.sls)

```
proxy:
  proxytype: napalm
  driver: junos
  fqdn: r1.bbone.as1234.net
  username: napalm
  password: Napalm123
```

Example 2-3. Sample file for Arista switch managed using NAPALM (device2_pillar.sls)

```
proxy:
  proxytype: napalm
  driver: eos
  fqdn: sw1.bbone.as1234.net
  username: napalm
  password: Napalm123
```

If you are authenticating using SSH keys, and if the driver supports key-based authentication, the `password` field is not mandatory or can be empty (i.e., `password: ''`).

Example 2-4. Sample file for Cisco IOS router using SSH key for authentication (device3_pillar.sls)

```
proxy:
  proxytype: napalm
  driver: ios
  fqdn: r2.bbone.as1234.net
  username: napalm
  optional_args:
    secret: s3kr3t
```

Grains

Grains represent static data collected from the device. The user does not need to do anything but to be aware this information already exists and it is available. Grains are typically handy for targeting minions and for complex and cross-vendor templating, but not limited to these uses. They are directly available when working with the CLI and also inside templates.

Grains must be understood as purely static data or information very unlikely to change, or at least data that does not change often.

NAPALM grains

When a device is managed through NAPALM the following grains are collected:

Grain name	Grain description	Example
`vendor`	Name of the vendor	`Cisco`
`model`	Chassis physical model	`MX960`
`serial`	Chassis serial number	`FOXW00F001`
`os`	The operating system name	`iosxr`
`version`	The operating system version	`13.3R6.5`
`uptime`	The uptime in seconds	`2344`
`host`	Host (FQDN) of the device	`r1.bbone.as1234.net`
`interfaces`	List of interfaces	`Ethernet1`, `Ethernet49/1`, `Loopback0`
`username`	The username used for connection	`napalm`

Configuring static grains

You also have the option of configuring grains statically inside the proxy minion configuration file. This is a good idea when you need

to configure device-specific data that can be used to uniquely identify a device or a class of devices (e.g., `role` can be set as `spine`, `leaf`, etc.).

Custom grain modules

Very often, we will need additional grains to be collected dynamically from the device to determine and cache particular characteristics. Writing *grain modules* is beyond the scope of this book, but is discussed in the documentation.

Master Configuration

The Salt system is very flexible and easy to configure. For network automation our components are the `salt-master`, configured via the *master configuration file* (typically */etc/salt/master* or */srv/master*); and the `salt-proxy`, configured via the *proxy configuration file* (in general */etc/salt/proxy*, */srv/proxy* or *C:\salt\conf\proxy*, depending on the platform). Their location depends on the environment and the operating system. They are structured as YAML files, usually simple key–value pairs, where the key is the *option* name. See the documentation for the complete list of options.

For our network automation needs there are not any particular options to be configured, but `file_roots` and `pillar_roots` are very important to understand.

File Roots

Salt runs a lightweight file server that uses the existing, encrypted transport to deliver files to minions. Under the `file_roots` option one can structure the environment beautifully, having a hierarchy that is also easy to understand. In addition, it allows running different environments on the same server without any overlap between them (see Example 2-5).

Example 2-5. Sample file_roots, using two environments

```
file_roots:
  base:
    - /etc/salt/
    - /etc/salt/states
  sandbox:
```

```
    - /home/mircea/
    - /home/mircea/states
```

In Example 2-5, we have two environments, `base` and `sandbox`, which are custom environment names. When executing various commands the user can specify what is the targeted environment (defaulting to `base` when not explicitly specified).

Pillar Roots

The `pillar_roots` option sets the environments and the directories used to hold the pillar SLS data. Its structure is very similar to `file_roots`, as you can see in Example 2-6.

Example 2-6. pillar_roots sample, using the environments defined under file_roots

```
pillar_roots:
  base:
    - /etc/salt/pillar
  sandbox:
    - /home/mircea/pillar
```

Note that under each environment we are able to specify multiple directories, thus we can define the SLS pillar under more than one single directory.

NOTE

All subsequent examples in this book will use the `file_roots` and `pillar_roots` configuration shown in Examples 2-5 and 2-6.

Using External Pillar

As mentioned in "Pillar" on page 15, the pillar data can also be loaded from external services as described in the documentation. There are plenty of already integrated services that can be used straightaway.

External pillar example: Vault

For security reasons a very common practice is using the HashiCorp Vault to store sensitive information. Setup is a breeze—it only

requires a couple of lines to be appended to the master configuration (see Example 2-7).

Example 2-7. Vault external pillar configuration sample

```
ext_pillar:
  - vault:
      vault.host: 127.0.0.1
      vault.port: 8200
      # The scheme is optional, and defaults to https
      vault.scheme: http
      # The token is required, unless set in environment
      vault.token: 012356789abcdef
```

The data retrieved from the Vault can be used inside other pillar SLS files as well, but it requires the `ext_pillar_first` option to be set as `true` in the master configuration.

Proxy Configuration

As the proxy minion is a subset of the regular minion, it inherits the same configuration options, as discussed in the minion configuration documentation. In addition, there are few other more specific values discussed in the proxy minion documentation.

A notable option required for some NAPALM drivers to work properly is `multiprocessing` set as `false`, which prevents Salt from starting a sub-process per command; it instead starts a new thread and the command is executed therein (see Example 2-8). This is necessary for SSH-based proxies, as the initialization happens in a different process; after forking the child instance it actually talks to a duplicated file descriptor pointing to a socket, whereas the parent process is still alive and might even be doing side-effecting background tasks. If the parent is not suspended, you could end up with two processes reading and writing to the same socket file descriptors. This is why the socket needs to be handled in the same process, each task being served in a separate thread. It is essential as some network devices are managed through SSH-based channels (e.g., Junos, Cisco IOS, Cisco IOS-XR, etc.). However, it can be re-enabled for devices using HTTP-based APIs (e.g., Arista or Cisco Nexus).

Example 2-8. Proxy configuration sample file

```
master: localhost
pki_dir: /etc/salt/pki/proxy
cachedir: /var/cache/salt/proxy
multiprocessing: False
```

The Pillar Top File

Each device has an associated unique identifier, called the *minion ID*. The *top file* creates the mapping between the minion ID and the corresponding SLS pillar file(s).

As we can have multiple environments defined under the `file_roots` master option, the Top File is also flexible enough to create different bindings between the minion ID and the SLS file, depending on the environment.

The top file is another SLS file named *top.sls* found under one of the paths defined in the `file_roots`.

In this book, we will use a simple *top file* structure, having a one-to-one mapping between the minion ID and the associated pillar SLS (see Example 2-9).

Example 2-9. /etc/salt/pillar/top.sls

```
base:
  'device1':  # minion ID
    - device1_pillar
    # minion ID 'device1' loads 'device1_pillar.sls'
  'device2':  # minion ID
    - device2_pillar
    # minion id 'device2' loads 'device2_pillar.sls'
  'device3':  # minion ID
    - device3_pillar
    # minion id 'device3' loads 'device3_pillar.sls'
```

In this file, where `device1_pillar`, `device2_pillar`, and `device3_pillar` represent the name of the SLS pillar files defined in Examples 2-2, 2-3, and 2-4.

When referencing a SLS file in the top file mapping, *do not* include the *.sls* extension.

The mapping can be more complex and we can select multiple minions that use a certain pillar SLS. *Targeting* minions can be based on the grains, regular expressions matched on the minion ID, a list of IDs, static defined groups, or even external pillar data. We will expand on this momentarily, but meanwhile, Example 2-10 illustrates how an SLS file called *bbone_rtr.sls* can be used by a group of minions whose IDs begin with *router*.

Example 2-10. Sample top file

```
# Apply SLS files from the directory root
# for the 'base' environment
base:
  # All minions with a minion ID that
  # begins with 'router'
  'router*':
    # Apply the SLS file named 'bbone_rtr.sls'
    - bbone_rtr
```

Remember that the top file leverages the SLS principles, therefore the mapping from Example 2-9 can also be dynamically created as:

```
base:
  {% for index in range(1, 4) -%}
  'device{{ index }}':
    - device{{ index }}_pillar
  {% endfor -%}
```

The top file can equally be defined using the Python renderer, as complex and dynamic as the business logic requires.

Starting the Processes

Long-running daemons on both the Salt master and Salt minion or proxy minion will maintain the always-on and high-speed communication channel. There are ways to make use of Salt without running these daemons, but those have specific use cases and specific limitations. This section will focus on the common configuration using daemons.

Starting the Master Process

For the scope of this book, `file_roots` and `pillar_roots` are enough to start the master process. The only requirement is that the file respects a valid YAML structure.

The master process can be started in daemon mode by running `salt-master -d`. On a Linux system the process can be controlled using `systemd`, as the Salt packages also contain the service configuration file: `systemctl start salt-master`. On BSD systems, there are also startup scripts available.

Starting the Proxy Processes

While the master requires one single service to be started, in order to control network gear we need to start one separate process per device, each consuming approximately 60 MB RAM.

A very good practice to start with and check for misconfiguration is to run the proxy in debug mode: `salt-proxy --proxyid <MINION ID> -l debug`. When everything looks good we are able to start the process as daemon: `salt-proxy --proxyid <MINION ID> -d` or use `systemd`: `systemctl start salt-proxy@<MINION ID>`.

NOTE

Using `systemd`, one is able to start multiple devices at a time using shell globbing (e.g., `systemctl start salt-proxy@device*`). Additionally, it presents the advantage that `systemd` manages the process startup and restart.

In case we want to avoid using `systemd`, Salt facilitates the management of the proxy processes using a *beacon*, and we have illustrated this particular use case in Chapter 7.

For example, with the environment setup exemplified in the previous paragraphs we can start the process as: `salt-proxy --proxyid device1 -d` and `salt-proxy --proxyid device2 -d` or `systemct start salt-proxy@device1` and `systemct start salt-proxy@device2`.

Each proxy process manages one individual network device, so for large-scale networks this leads to managing thousands of proxy processes. This is easier to manage when controlling the server that is running the proxy processes using the regular Salt minion—the orchestration of the proxies is effectively reduced to just managing a few configuration files!

Accepting the minion key

When a new minion is started, it generates a private and public key pair. For security reasons the minion public key is not accepted automatically by the master. The `auto_accept` configuration option on the master can be turned on, but this is highly discouraged in production environments. The CLI command `salt-key -L` will display the list of accepted and unaccepted minions.

For this reason, we need to accept the minion key (see Example 2-11).

Example 2-11. Accept the proxy minion key, using salt-key

```
$ sudo salt-key -a device1
The following keys are going to be accepted:
Unaccepted Keys:
device1
Proceed? [n/Y] y
Key for minion device1 accepted
```

The `salt-key` program can also accept shell globbing to accept multiple minions at once. For example:

```
salt-key -a device* -Y
```

There are also good ways to automate acceptance of minion keys using the Salt reactor or the REST API.

In this chapter we have presented several of the most important Salt-specific keywords—they all are very important and constitute the foundation moving forward. Very briefly, we have also covered the processes startup. Now that you have the environment set up, go ahead and start automating with Salt.

CHAPTER 3

Understanding the Salt CLI Syntax

It is very important to understand how Salt works when running commands from the CLI. The same methodologies can be used when instructing the system to execute commands periodically via Salt's scheduler or when listening to events and via the reactor, only the syntax is different.

The Salt CLI syntax has the following pattern:

```
$ sudo salt [<options>] <target> <function> [<arguments>]
```

One of the simplest uses is shown in Example 3-1, which executes the `test.ping` function on the device identified using the minion ID `device1`, without any options or arguments.

Example 3-1. Basic CLI execution invoking the test.ping execution function

```
$ sudo salt device1 test.ping
# output omitted
```

As we move forward, we'll analyze each of the components shown here.

Functions and Arguments

Salt is structured as a very simple and pluggable core with many features able to interact between them. A very important functionlity is represented by the *execution modules*. They are the main entry point into the Salt world. The execution modules are Python modules,

and are very easy to read (and eventually write) by anyone with basic Python programming knowledge. Everything is linear, which makes them flexible and easy to understand; in general, they consist only of simple functions.

In Example 3-1 we executed `test.ping`: `test` is the name of the execution module, while `ping` is the *execution function* inside this module.

There are many module types in Salt and some even have overlapping names, so it is important to always qualify the type of the module. For example, the file state module and the file execution module are different modules and used in different contexts.

This function only returns the logical value `True` when the minion is up. `test.ping` is an excellent way to test that the minion has been configured correctly and its key is accepted by the master. It doesn't mean, however, that the connection to the network device succeeded. For this particular usage we can execute the `net.connected` module. Repeating the exercise: the Python function `connected` is defined under the Salt execution module `net`. While the `test` execution module corresponds to a Python module called `test.py`, `net` corresponds to `napalm_network.py`. This is because some modules have a virtual name associated and they are loaded under a cross-platform consistent name.

There are many thousands of functions available. For the complete module reference, consult the Salt Module Index.

An execution function must return JSON-serializable data structures. Thanks to this approach the output can be manipulated, ported, and reused in different modules, displayed in the shape we require or transform it and load in a certain service. A good example is on the CLI: the output is transformed into a more human-readable and colorful format. There are many possibilities to adjust the output and this can be pluggable: you even have the option of customizing the way the output is processed and presented.

In Example 3-2, after `test.ping` was executed Salt provides the response from `device1` as `True`.

Example 3-2. Sample CLI output

```
$ sudo salt device1 test.ping
device1:
    True
```

Another useful function is `grains.items`, which provides the complete list of grains, and `grains.get`, which returns the value of a given grain identified by name, as shown in Example 3-3.

Example 3-3. Retrieve the serial grain value

```
$ sudo salt device1 grains.get serial
device1:
    VM58737E84CF
```

In a similar way, `pillar.items` provides the pillar data available on the minion, while `pillar.get` returns a specific value (see Example 3-4).

Example 3-4. Retrieve the ntp.servers grain value

```
$ sudo salt device1 pillar.get ntp.servers
device1:
    - 172.17.17.1
    - 172.17.17.2
```

Here, the pillar key `ntp.servers` has the value of a list of NTP servers. This also introduces the *arguments* to Example 3-4. `ntp.servers` is an argument passed to the `get` execution function from the `pillar` execution module.

To check what modules are loaded, identified by their virtual name, we can run `sys.list_modules`. Similarly, to display the complete list of available functions we can run `sys.list_functions`.

For network automation needs there are many execution functions natively embedded in Salt. For example, `net.arp` retrieves the ARP tables (see Example 3-5).

Example 3-5. Sample CLI output: net.arp

```
$ sudo salt device1 net.arp
device1:
    ----------
    out:
        |_
          ----------
          age:
              129.0
          interface:
              ae2.100
          ip:
              10.0.0.1
          mac:
              00:0f:53:36:e4:50
        |_
          ----------
          age:
              1101.0
          interface:
              xe-0/0/3.0
          ip:
              10.0.0.2
          mac:
              00:1d:70:83:40:c0
```

In Example 3-5 we executed the `net.arp` function, which retrieves the ARP table from `device1`. One can easily notice that the output is not a Python object, though. The reasoning is that although the `arp` execution function returned JSON-serializable data, Salt transformed it in a more human-readable and colorful format when displaying on the CLI. This format comes from an outputter module called *nested*. The user can choose to display the data formatted by a different outputter if they prefer.

The output is a list of ARP entries, each having the following details: `age`, `interface`, `ip`, and `mac`. It is vendor-agnostic, thanks to NAPALM capabilities, so executing `net.arp` against a device running a different supported platform will return exactly the same structure!

Several other samples are provided in Examples 3-6 through 3-8.

Example 3-6. net.arp using arguments

```
$ sudo salt device3 net.arp interface=TenGigE0/0/0/0
# output omitted
```

Note that although `net.arp` does not require an argument in Example 3-5, it can be passed. However, in Example 3-6, we retrieve the ARP table from `device3` but only on the interface `TenGigE0/0/0/0`.

Example 3-7. The ntp.stats execution function

```
$ sudo salt device2 ntp.stats 172.17.17.1
# output omitted
```

Example 3-7 returns the NTP synchronization statistics with the server `172.17.17.1` from `device2`. The `ntp.stats` execution function can be equally executed without an explicit argument, in that case it returns the synchronization details with all the NTP peers.

Example 3-8. The net.ping execution function

```
$ sudo salt device1 net.ping 8.8.8.8 vrf=CUSTOMER1
# output omitted
```

In Example 3-8 we execute a ping to `8.8.8.8` from the `CUSTOMER1` VRF.

You can learn more about these functions in the documentation, but sometimes it's easier to check it directly from the CLI by executing `sys.doc` followed by the function or module name (e.g., `salt device1 sys.doc net.arp`). To check which arguments are mandatory and which ones are the default values, you can run `sys.argspec` in a similar way (e.g., `salt device1 sys.argspec net.arp`).

Targeting Devices

Targeting is used on the CLI but also in scheduled actions or when reacting to events. It is used to select a group of minions that need to execute a function. Targeting is a task sent by the master in a payload to all minions and the minions decide independently if the target matched their characteristics. This is yet another optimization

that makes Salt extremely scalable. In the next sections, we will be executing the `test.ping` function.

Targeting Using the Minion ID

This is the first targeting method we are exposed to. It's very easy to understand because it executes the function on a single minion, identified by its ID.

As we have already seen, the command shown in Example 3-9 executes `test.ping` only on `device1`.

Example 3-9. Global targeting command

```
$ sudo salt device1 test.ping
# output omitted
```

Targeting Using a List of Minion IDs

As shown in Example 3-10, using the `-L` option we can tell the `salt` executable to call the function on a list of minions, their IDs being specified as comma-separated values. (It is best to avoid spaces in-between each ID since that format isn't universally supported throughout Salt.)

Example 3-10. List targeting

```
$ sudo salt -L device1,device2 grains.get vendor
device2:
    Juniper
device1:
    Arista
```

Example 3-10 is going to execute the `grains.get` function on `device1` and `device2` using the `vendor` argument. The reply provides the value of the `vendor` grain for each device.

NOTE

The reply order from each device is not necessarily the order we have requested. Salt works asynchronously and the output is displayed as soon as it is received from the minion.

Targeting Using Shell-Like Globbing

The minions can be selected via their minion ID using shell-style globbing:

Pattern	Explanation
`*`	Matches everything
`?`	Matches any single character
`[seq]`	Matches any character in the sequence
`[!seq]`	Matches any character that is not in the sequence

The `device*` expression from Example 3-11 matches `device1`, `device2`, and `device3`, as these are the Minions authenticated to this Salt master whose ID starts with "device". If you have thousands of minions with the ID beginning with `device`, they all would be matched so you don't need to write a static list.

Example 3-11. Shell-style globbing to target minions whose ID starts with device

```
$ sudo salt 'device*' grains.get model
device1:
    VMX
device2:
    vEOS
device3:
    ASR9001
```

Targeting Using Regular Expressions

Using Perl-compatible regular expressions (PCRE), we can use an even more flexible way to select groups of minions using their ID (see Example 3-12).

Example 3-12. Targeting using regular expressions

```
$ sudo salt -E '(device|edge)\d' test.ping
# output omitted
```

In this example, the command would be executed on minions starting with either `device` or `edge`, followed by a digit. Note the usage of the `-E` option.

Targeting Using Grains

Grains can be also used to target minions using their characteristics, as shown in Examples 3-13 through 3-15 (if you need a refresher on the topic, refer back to "Grains" on page 18).

Example 3-13. Target devices running Junos

```
$ sudo salt -G 'os:junos' test.ping
# output omitted
```

Example 3-14. Target devices running Junos 15.1F7.3

```
$ sudo salt -C 'G@os:junos and G@version:15.1F7.3' test.ping
# output omitted
```

Example 3-15. Target Cisco ASR routers having TenGigE0/0/0/30 in the list of interfaces

```
$ sudo salt -C 'G@vendor:cisco
and G@model:ASR*
and G@interfaces:TenGigE0/0/0/30' test.ping
# output omitted
```

NOTE

Example 3-15 shows that grain matching also accepts globbing (`model:ASR*` matches any ASR model). To match grains using regular expressions use the `-P` option instead.

Targeting Using Pillar Data

In a very similar way it is also possible to target using nested pillar values. For example, we could match the minions that have the hostname ending with `as1234.net` (Example 3-16).

Example 3-16. Target using the host field under the proxy key in pillar

```
$ sudo salt -I 'proxy:host:*as1234.net' test.ping
# output omitted
```

The nesting levels are separated using the `:` character. However, this can also be changed and select a different delimiter, using the `--delimiter` option: `salt --delimiter='/' -I 'proxy/host/*as1234.net' test.ping`.

NOTE

This option also allows globbing. To match using regular expressions, use the `-J` option.

Compound Target Matching

Everything we just covered can be combined and form a very complex target expression. The command option in that case becomes `-C` and the previous options must be specified in the body of the match string using the `@` sign (see Examples 3-17 and 3-18).

Example 3-17. Target minions whose ID starts with device that are running IOS-XR 6.x

```
$ sudo salt -C 'device* and G@os:iosxr and G@version:6.*' test.ping
# output omitted
```

Example 3-18. Target MX960 and MX480 routers having the hostname respecting a regular expression

```
$ sudo salt -C '
P@model:(MX960|MX480)
and J@proxy:host:^(.*bbone\.as\d+\.net)$ ' test.ping
# output omitted
```

Defining and Targeting Using Nodegroups

For very complex examples such as the one shown in Example 3-18, we can add their definition in the master configuration file along with a name and then use it directly rather than typing the same matching expression each time. These are called nodegroups, and they serve mostly as shortcuts (see Example 3-19).

Example 3-19. Nodegroups definition in the master config file

```
nodegroups:
  bbone-mxs: |
```

```
    P@model:(MX960|MX480)
    and J@proxy:host:^(.*bbone\.as\d+\.net)$
  iosxr6: |
    device*
    and G@os:iosxr and G@version:6.*
```

Which can be used through the `-N` CLI option (Example 3-20):

Example 3-20. Nodegroups usage

```
$ salt -N bbone-mxs test.ping
# output omitted
$ salt -N iosxr6 test.ping
# output omitted
```

Targeting Inside the Top File

The matching techniques presented can also be used in the *top file*, where the default match is the *compound* matcher. We can exploit this flexibility to create smart mappings between the device characteristics and the data entities to be provided.

For example, consider the pillar top file shown in Example 3-21.

Example 3-21. Pillar top file mappings using advanced targeting

```
base:
  'device* and G@os:iosxr and G@version:6.*':
    - pillar_for_iosxr6
  'N@bbone-mxs':
    - mx_routers_bbone
```

This top file includes the *pillar_for_iosxr6.sls* pillar on minions managing Cisco IOS-XR 6.x devices, and equally the *mx_routers_bbone.sls* pillar for minions matching the `bbone-mxs` nodegroup defined earlier in Example 3-19.

Options

The `salt` command has a very long list of options, which you can retrieve by executing `salt --help`. In "Targeting Devices" on page 31 we have listing some of the most usual, the *target options*. We highly encourage you to explore the full list of options, as they prove very handy in many situations. Due to size limitations in this book, we will present only a few of the most common.

Outputters

As we discussed earlier, the output from Salt functions is JSON serializable. This output can be transformed in different shapes depending on the application or personal preferences. One can choose using the `--out` option between: `json`, `yaml`, `nested` (default), `table`, `raw` (displays the exact Python object), or `pprint` (displays the Python object in a more human-readable form).

Example 3-22. net.arp using the JSON outputter

```
$ sudo salt --out=json device1 net.arp
[
  {
    "interface": "ae2.100",
    "ip": "10.0.0.1",
    "mac": "00:0f:53:36:e4:50",
    "age": 129.0
  },
  {
    "interface": "xe-0/0/3.0",
    "ip": "10.0.0.2",
    "mac": "00:1d:70:83:40:c0",
    "age": 1101.0
  }
]
```

NOTE The outputter only displays the Python object on the CLI in a more readable shape. It does not change its structure!

CHAPTER 4

Configuration Management: Introduction

The previous chapters covered the basics: you learned how to install and configure the tools, and began working with the Salt CLI. Now that you have an understanding of those fundamentals, you can start diving into the configuration management and advanced templating capabilities of Salt.

Configuration management is among the most important tasks in the automation process, Salt's built-in features simplify this process dramatically. It is essential that you have thoroughly reviewed the material covered in previous chpaters before proceeding with this one, as the concepts we've discussed previously play an important role in the configuration management methodologies discussed here. For the moment we will mainly use the CLI to apply simple configuration changes: it is important to understand and get comfortable with advanced Salt templating.

Loading Static Configuration

Let's suppose we are at a point where our large-scale network does not have consistent configuration. Loading a static configuration change can come in very handy for such cases (see Example 4-1).

Example 4-1. Load static configuration changes

```
$ sudo salt -G 'vendor:arista' \
   net.load_config \
   text='ntp server 172.17.17.1'
device2:
    ----------
    already_configured:
        False
    comment:
    diff:
        @@ -42,6 +42,7 @@
         ntp server 10.10.10.1
         ntp server 10.10.10.2
         ntp server 10.10.10.3
        +ntp server 172.17.17.1
         ntp serve all
         !
    result:
        True
```

By executing `net.load_config`, you can load a simple configuration change. Targeting using the *grain matcher*, each device that is an Arista switch replies back with a configuration difference (the equivalent of `show | compare` in Junos terms). It also informs us that the device was not already configured and the changes succeeded. The `result` field is `True`, as you can see in Example 4-2.

Example 4-2. Loading static configuration changes: dry run

```
$ sudo salt -G os:eos \
  net.load_config \
  text='ntp server 172.17.17.1' \
  test=True
device2:
    ----------
    already_configured:
        False
    comment:
        Configuration discarded.
    diff:
        @@ -42,6 +42,7 @@
         ntp server 10.10.10.1
         ntp server 10.10.10.2
         ntp server 10.10.10.3
        +ntp server 172.17.17.1
         ntp serve all
         !
```

```
    result:
        True
```

Executing the same command now, but appending the `test=True` option, Salt will load the configuration changes, determine the configuration difference, discard the changes and return the same output as before—the difference is that the `comment` informs us that the changes made into the candidate configuration have been revoked.

NOTE

When executing in test mode (dry run), we *do not* apply any changes in the running configuration of the device. There are no risks: the changes are always loaded into the candidate configuration and transferred into the running configuration only when an explicit commit is invoked. During a dry run (`test=True`) we do not commit.

If you need more changes, you can store them in a file and reference it using the absolute path (see Example 4-3).

Example 4-3. Loading static configuration from the file

```
$ sudo salt -G 'vendor:arista' net.load_config /path/to/file.cfg
# output omitted
```

Loading Dynamic Changes

Loading static changes cannot be enough; very often you will need to configuredifferent properties depending on the device and its characteristics. The methodologies presented here are very important and will be referenced often in this book. At this point, you should focus mainly on learning the substance rather than the CLI usage.

For dynamic changes we will use a template engine, such as Jinja (discussed in "The Three Rules of Jinja" on page 9); see Example 4-4.

Example 4-4. Loading dynamic changes using a very basic template

```
$ sudo salt -G os:eos \
  net.load_template \
  template_source='hostname {{ host }}' \
  host='arista.lab'
device2:
```

```
    ----------
    already_configured:
        False
    comment:
    diff:
        @@ -35,7 +35,7 @@
         logging console emergencies
         logging host 192.168.0.1
         !
        -hostname edge01.bjm01
        +hostname arista.lab
         !
    result:
        True
```

Observe the name of the function is `net.load_template`. Inside the `template_source` argument we have the Jinja template defined inline; `host` is the variable used inside the template.

One of the most important features in Salt is that you are able to use the `grains`, `pillar`, and the configuration options (`opts`) inside the template (see Example 4-5).

Example 4-5. Using grains inside the inline template

```
$ sudo salt -G os:eos \
  net.load_template \
  template_source='hostname {{ grains.model }}'
device2:
    ----------
    already_configured:
        False
    comment:
    diff:
        @@ -35,7 +35,7 @@
         logging console emergencies
         logging host 192.168.0.1
         !
        -hostname edge01.bjm01
        +hostname DCS-7280SR-48C6-M-R
         !
    result:
        True
```

Referencing grains data is very easy, we only need to use the reserved variable `grains` followed by the grain name. Example 4-5 uses the `model` grain, which provides the physical chassis model of the network device.

Grains prove extremely useful inside templates: they allow you to define one single template and use it across your entire network, regardless of the vendor. Example 4-6 shows a sample template.

Example 4-6. Cross-vendor Jinja template

```
{%- set router_vendor = grains.vendor -%}
{%- set hostname = pillar.proxy.fqdn.replace('as1234.net', '') -%}
{%- if router_vendor|lower == 'juniper' %}
system {
  host-name {{hostname}}lab;
}
{%- elif router_vendor|lower in ['cisco', 'arista'] %}
hostname {{hostname}}lab
{%- endif %}
```

Using the `vendor`, `os`, and `version` grains, we can determine the platform characteristics and generate the configuration accordingly, from one single template. Note that we also introduced the usage of another Salt property: `pillar`. Using this we can access data from the pillar. In this example we configure the hostname of the device based on the `fqdn` field from the proxy pillar, by removing the `as1234.net` part.

Saving the contents from Example 4-6 to */etc/salt/templates/hostname.jinja*, you can then execute the configuration load against all your devices and the template is smart enough to know what configuration to generate (see Example 4-7).

Example 4-7. Execute cross-vendor template

```
$ sudo salt device1 \
  net.load_template \
  /etc/salt/templates/hostname.jinja
device1:
    ----------
    already_configured:
        False
    comment:
    diff:
        [edit system]
        -  host-name edge01.flw01;
        +  host-name r1.bbone.lab;
    result:
        True
```

Having `/etc/salt/templates` configured as one of the paths under `file_roots`, we are able to render the template and load the generated configuration on the device, by executing: `salt '*' net.load_template salt://templates/hostname.jinja`. Not only is this syntax easier to remember, but it is also a very good practice as we don't rely on a specific environment setup.

We can even use remote templates: besides `salt://`, we can equally use `ftp://`, `http://`, `https://`, `s3://`, or `swift://`. The templates will be retrieved from the corresponding location, then rendered.

Another very useful feature is the *debug mode*. When working with more complex templates, we can see the result of the template rendering by using the `debug` flag, as shown in Example 4-8.

Example 4-8. Using the debug mode

```
$ sudo salt device1 net.load_template \
  salt://templates/hostname.jinja debug=True
device1:
    ----------
    already_configured:
        False
    comment:
    diff:
        [edit system]
        -  host-name edge01.flw01;
        +  host-name r1.bbone.lab;
    loaded_config:
        system {
          host-name r1.bbone.lab;
        }
    result:
        True
```

Under the `loaded_config` we can see the exact result of the template rendering, which is not necessarily identical to the configuration diff.

For debugging purposes, in Salt we can even use logging inside our templates. For example, the following line will log the message in the proxy log file (typically under */var/log/salt/proxy*, unless configured elsewhere):

```
{%- do salt.log.debug('Get salted') -%}
```

One of the most important features in Salt templating is reusability. We've seen the `grain` and the `pillar` variables, now let's introduce the `salt` variable. This allows you to call *any* execution function. While on the CLI we would execute `salt device2 net.arp`, inside the template we can have `{%- set arp_table = salt.net.arp() -%}` to load the output of the `net.arp` execution function into the `arp_table` Jinja variable, then manipulate it as needed.

Example 4-9. Cross-vendor template reusing Salt functions

```
{%- set route_output = salt.route.show('0.0.0.0/0', 'static') -%}
{%- set default_routes = route_output['out'] -%}

{%- if not default_routes -%}
{# if no default route found in the table #}
  {%- if grains.vendor|lower == 'juniper' -%}
routing-options {
    static {
        route 0.0.0.0/0 next-hop {{ .def_nh }};
    }
}
  {%- elif grains.os|lower == 'iosxr' -%}
  {%- set def_nh = pillar.def_nh %}
  router static address-family ipv4 unicast 0.0.0.0/0 {{ def_nh }}
  {%- endif %}
{%- endif -%}
```

The Jinja template from Example 4-9 retrieves the default static routes from the RIB using the `route.show` execution function. If the result is empty (no static routes found), it will generate the configuration for a static route to `0.0.0.0/0` using as next hop the value of the `def_nh` field from the Pillar. And we can achieve this with just a couple of lines, covering two different types of platforms: Junos and IOS-XR.

Using the Salt advanced templating capabilities, we can write beautiful and more readable templates, by moving the complexity into the execution modules. There's a tutorial showing how in the SaltStack documentation.

CHAPTER 5

Salt States: Advanced Configuration Management

The state subsystem facilitates the management of a device to keep it in a predetermined state. On the server side it is used to install packages, start or restart services, and configure files or other data entities. The same methodologies can be applied on whitebox devices that allow custom software installation, otherwise the state system is an excellent way to manage the configuration of traditional network gear.

We will rely heavily on the advanced templating methodologies covered in Chapter 4, so we can use the `pillar`, `salt`, `grains`, or `opts` reserved keywords presented earlier to access data from the corresponding entities. In other words, we can access the data from databases, Excel files, Git, or REST APIs directly and the state does not rely on the mechanism used to retrieve the data—Salt provides a clear separation of the automation logic and data.

The Salt states are SLS files containing information about how to configure the devices managed. Their structure is very simple, based on key-value pairs and lists. As discussed in Chapter 1, SLS is by default YAML and Jinja, very easy and flexible to design. It preserves the SLS capabilities so that you can switch to a different combination of data representation plus template language, or even pure Python, when required. Inside the state SLS we invoke state functions defined in the state modules.

The State Top File

As any Salt subsystem, the state has its own *top file*, which defines the mapping between the groups of minions that can execute a certain state (Example 5-1).

Example 5-1. Sample state top file (/etc/salt/states/top.sls)

```
base:
  '*':
    - ntp
    - users
    - lldp
  'router* or G@role:router':
    - bgp
    - mpls
  'sw* or G@role:switch':
    - stp
```

In Example 5-1, any minion can execute the `ntp.sls`, `users.sls`, and `lldp.sls` states, while `bgp.sls` and `mpls.sls` can only be executed by the minion ID that starts with *router* or having the `role` grain configured as *router*; `stp.sls` can only be executed by the switches, identified using the minion ID or their `role` grain. Note that the `role` grain is not globally available; it must be defined by the user according to the business requirements.

NetConfig

NetConfig is the most flexible state module for network automation. It does not have any particular dependency, but it requires the user to write their own templates. The cross-vendor templating methodologies presented in Chapter 4 remain the same, with the advantage that the separation between data and instructions becomes obvious. We'll now analyze a few simple examples serving real-world needs.

Automating the Configuration of the NTP Servers

Let's automate the NTP configuration of a large-scale network, to ensure that only the servers `172.17.17.1` and `172.17.17.2` are used for synchronization. The network has devices produced by Cisco, Juniper, and Arista.

The first step is placing the list of NTP servers in the pillar. In this example, the `pillar_roots` option on the master is set as `/etc/salt/pillar`. The NTP servers are defined as a list in an SLS file called *ntp_servers.sls*, shown in Example 5-2.

Example 5-2. The ntp_servers pillar file, /etc/salt/pillar/ntp_server.sls

```
ntp.servers:
  - 172.17.17.1
  - 172.17.17.2
```

Using the `include`, `exclude`, and `extend` directives, we can include the structure shown in Example 5-2 for each device very granularly, or we can simply include it for all devices using the *top file* matching strategies, as shown in Example 5-3.

Example 5-3. The pillar top file, /etc/salt/pillar/top.sls

```
base:
  '*':
    - ntp_servers
  'device1':
    - device1_pillar
  'device2':
    - device2_pillar
  'device3':
    - device3_pillar
```

`device1`, `device2`, and `device3`, as well as the corresponding pillar SLS, were defined in "Configuring the NAPALM Pillar" on page 16. The `'*'` tells Salt to provide the content from *ntp_servers.sls* to all minions.

The next step is refreshing the pillar data, executing `salt 'device*' saltutil.refresh_pillar`.

We can use the `pillar.get` execution function to check that the data has been loaded correctly (Example 5-4).

Example 5-4. Pillar get ntp.servers

```
$ sudo salt 'device*' pillar.get ntp.servers
device1:
    - 172.17.17.1
    - 172.17.17.2
device2:
```

```
    - 172.17.17.1
    - 172.17.17.2
device3:
    - 172.17.17.1
    - 172.17.17.2
```

Now, as the data is available on all devices without much effort, we can define the template (Example 5-5).

Example 5-5. /etc/salt/states/ntp/templates/ntp.jinja

```
{%- if grains.vendor|lower in ['cisco', 'arista'] %}
  no ntp
  {%- for server in servers %}
  ntp server {{ server }}
  {%- endfor %}
{%- elif grains.os|lower == 'junos' %}
  system {
    replace:
    ntp {
      {%- for server in servers %}
      server {{ server }};
      {%- endfor %}
    }
  }
{%- endif %}
```

The template checks the `vendor` and `os` grains and generates the configuration for the NTP servers depending on the platform. Cisco and Arista are grouped together as the syntax for the NTP configuration is very similar.

The variable `servers` will be sent from the state SLS, but it could equally be directly accessed as `pillar['ntp.servers']` or `salt.pillar.get('ntp.peers')`. For flexibility reasons, it is preferred to send the data from the state SLS.

The template is defined under */etc/salt/states/ntp/templates/ntp.jinja*: `/etc/salt/states` is the path to the Salt file server for the states. As defined under `file_roots`, `ntp` is the name of the state, which can be a hierarchical directory where the templates are defined in a dedicated directory. This is a good practice to remember when defining complex states.

In the `netconfig` state, the configuration enforcement behavior requires the user to explicitly use the configuration replace capabilities of the network operating system.

If the device does not have replace capabilities, the workaround is a supplementary execution function that retrieves the current state of the feature that can be executed inside the template using the `salt` directive and determine what needs to be removed and added. Although this requires one additional step and a slightly more complex template, this is a unique feature of Salt, permitting the configuration management for network devices having this drawback.

The last step is defining the SLS file under `file_roots`—we will use the `/etc/salt/states` path. A good practice is grouping the state SLS into directories depending on their role (Example 5-6).

Example 5-6. /etc/salt/states/ntp/init.sls

```
ntp_servers_recipe:
  netconfig.managed:
    - template_name: salt://ntp/templates/ntp.jinja
    - servers: {{ salt.pillar.get('ntp.servers') | json }}
```

`ntp_servers_recipe` is a name assigned to the state and it tells to execute the `managed` function from the `netconfig` state module, using the template `ntp.jinja` defined under the Salt file system and passing the variable `servers` that takes its value from the pillar key `ntp.servers`.

The state SLS is defined as `/etc/salt/states/ntp/init.sls`: `ntp` is the name of the state, while `init.sls` is a reserved name that allows the execution simply by specifying the name of the directory—that is, `ntp`. If we would define the state SLS under `/etc/salt/states/ntp/example.sls`, the state would be executed using: `ntp.example`.

Note the `json` Jinja filter from Example 5-6. This is not mandatory, but almost always a very good practice when passing objects; otherwise, values will be interpreted by the YAML parser, which has some surprising type-casting behaviors.

There are a few handy fields that can be specified in the SLS:

`debug`
: Return also the result of template rendering. The state returns the configuration difference, but that's not necessarily equivalent to the changes loaded.

`template_engine`
: When the user prefers a template engine other than Jinja.

`replace`
: Replace the entire configuration with the generated contents.

The `state.sls` execution function invokes the `ntp` state, defined under */etc/salt/states/ntp/init.sls*, as shown in Example 5-7.

Example 5-7. NTP state execution

```
$ sudo salt 'device2' state.sls ntp
device2:
----------
          ID: ntp_servers_recipe
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 13:15:07.608236
    Duration: 8954.756 ms
     Changes:
              ----------
              diff:
                  @@ -55,6 +55,7 @@
                  !
                  ntp source Loopback0
                 -ntp server 1.2.3.4
                 -ntp server 5.6.7.8
                 +ntp server 172.17.17.1
                  ntp serve all
                  !
Summary for device2
------------
Succeeded: 1 (changed=1)
Failed:    0
```

```
------------
Total states run:     1
Total run time:  8.955 s
```

The code shown in Example 5-8 changes the SLS to use the `debug` field.

Example 5-8. /etc/salt/states/ntp/init.sls

```
ntp_servers_recipe:
  netconfig.managed:
    - template_name: salt://ntp/templates/ntp.jinja
    - servers: {{ salt.pillar.get('ntp.servers') | json }}
    - debug: true
```

We can execute and the state will also provide the configuration rendered and loaded on the device (see Example 5-9).

Example 5-9. NTP state execution: test and debug mode

```
$ sudo salt 'device1' state.sls ntp test=True
device1:
----------
          ID: ntp_servers_recipe
    Function: netconfig.managed
      Result: None
     Comment: Configuration discarded.

              Configuration diff:

              [edit system ntp]
              -    peer 1.2.3.4;
              [edit system ntp]
              +    server 172.17.17.1;
              +    server 172.17.17.2;

              Loaded config:

              system {
                replace:
                ntp {
                  server 172.17.17.1;
                  server 172.17.17.2;
                }
              }
     Started: 13:07:09.983598
    Duration: 8566.857 ms
     Changes:
```

```
Summary for device1
------------
Succeeded: 1 (unchanged=1)
Failed:    0
------------
Total states run:     1
Total run time:   8.567 s
```

This state is a very good choice for production environments because it's easy to check the correctness of the template and if the changes are indeed as expected. The data is clearly decoupled and changes are now applied according to the Pillar, whose structure is vendor-agnostic and human-readable. In our recipe, to update the list of NTP servers in a large-scale network only becomes as simple as updating the */etc/salt/pillar/ntp_servers.sls* file, followed by the execution of the state.

Automating the Interfaces Configuration of a Multivendor Network

We don't have any specific constraint so we can structure the Pillar data at our will (see Example 5-10).

Example 5-10. /etc/salt/pillar/device1.sls

```
openconfig-interfaces:
  interfaces:
    interface:
      xe-0/0/0:
        config:
          mtu: 9000
          description: Interface1
        subinterfaces:
          subinterface:
            0:
              config:
                description: Subinterface1
              ipv4:
                addresses:
                  address:
                    1.2.3.4:
                      config:
                        ip: 1.2.3.4
                        prefix_length: 24
```

Based on the model exemplified earlier, we can start building the skeleton for the interfaces template (Example 5-11).

Example 5-11. /etc/salt/states/interfaces/templates/init.jinja

```
{%- if grains.os|lower == 'junos' %}
replace:
interfaces {
  {%- for if_name, if_details in interfaces.interface.items() %}
  {{ if_name }} {
    mtu {{ if_details.config.mtu }};
    description {{ if_details.config.description }};
    {%- set subif =
    if_details.subinterfaces.subinterface %}
    {%- for subif_id, subif_details in subif.items() %}
    unit {{ subif_id }} {
      description "{{ subif_details.config.description }}";
      {%- if subif_details.ipv4 %}
      family inet {
        {%- set subif_addrs =
        subif_details.ipv4.addresses.address %}
        {%- for _, addr in subif_addrs.items() %}
        address {{ addr.config.ip }}/{{ addr.config.prefix_length }};
        {%- endfor %}
      }{%- endif %}
    }{%- endfor %}
  }{%- endfor %}
}
{%- endif %}
```

The state SLS is defined in a similar way (Example 5-12).

Example 5-12. /etc/salt/states/interfaces/init.sls

```
interfaces_recipe:
  netconfig.managed:
    - template_name: salt://interfaces/templates/init.jinja
    - {{ salt.pillar.get('openconfig-interfaces') | json }}
```

And we can simply execute the state (Example 5-13).

Example 5-13. Interfaces state execution

```
$ sudo salt 'device1' state.sls interfaces
device1:
----------
          ID: interfaces_recipe
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 16:49:45.827128
    Duration: 7973.572 ms
```

```
    Changes:
              ----------
              diff:
                  [edit]
                  +  interfaces {
                  +      xe-0/0/0 {
                  +          description Interface1;
                  +          mtu 9000;
                  +          unit 0 {
                  +              description Subinterface1;
                  +              family inet {
                  +                  address 1.2.3.4/24;
                  +              }
                  +          }
                  +      }
                  +  }
Summary for device1
------------
Succeeded: 1
Failed:    0
------------
Total states run:     1
Total run time:  7.974 s
```

NetYANG

While the NetConfig state is very flexible, it requires you to define an environment-specific template, and implicitly to decide the structure of the pillar. Also, finding a common representation of the structure of the pillar that makes sense and covers several platforms turns out to be difficult sometimes. The structure from Example 5-10 may look overly complicated when generating the configuration, but as we'll soon see, this is a good pattern in order to cover the complexity and differences between various vendors. Fortunately, structures as in Example 5-10 have been standardized being modeled using *YANG*. YANG (Yet Another Next Generation) has been introduced in RFC6020 in October 2010 and is a *data modeling* language. Several organizations such as IETF have concentrated the efforts to standardize the modelation of structured entities of information for networking applications. Network vendors also focused on writing YANG models, but unfortunately this creates divergence when working in multivendor environments.

We will not focus on this topic here, as our main goal is an unified framework. A very important standardization group is OpenConfig, which has provided a significant number of YANG models already.

It consists exclusively of network operators whose goal is providing vendor-neutral representation of configuration and operational data based on production requirements.

> **NOTE**
>
> Do not conflate YANG with a transport protocol, nor a data representation language. YANG is a modelation language that defines the structure of the documents, regardless of their data representation language. These documents can be JSON, XML, or YAML, having the hierarchy tree according to the YANG model.

In Salt we have leveraged the modelation capabilities of YANG in such a way that the user is not required to write the templates, but only to structure the pillars following the YANG models. As the efforts toward programmable infrastructure are still at the very beginning so too are the tools involved: the only limitation of the NetYANG Salt state is the capabilities of its dependency, *napalm-yang*. One of the latest libraries of the NAPALM suite, `napalm-yang` is a community effort to translate vendor-specific data representation into structured documents, as per the YANG models. There are models already well covered, but there are many others waiting to be implemented. Without going into further details, we want to emphasize that, although writing templates in their own environment might be very tempting and feel more straightforward, a public contribution scales much better in the long term and it gives a lot of help back to the community.

A good starting point to visualize the hierarchy of the OpenConfig models can be found on the OpenConfig site.

Writing the Pillar Corresponding to the openconfig-lldp YANG Model

Navigating through the YANG model using the tool referenced above, in particular openconfig-lldp.html, and looking for the `config` containers, we can define the YAML structure shown in Example 5-14.

Example 5-14. YAML pillar structure based on the openconfig-lldp YANG model

```
openconfig-lldp:
  lldp:
    config:
      enabled: true
      hello-timer: 20  # seconds
      system-name: r1.bbone
      chassis-id-type: MAC_ADDRESS
    interfaces:
      interface:
        xe-0/0/0:
          config:
            enabled: true
        xe-0/0/1:
          config:
            enabled: true
        xe-0/0/2:
          config:
            enabled: true
```

Automating the Interfaces Configuration of a Multivendor Network, Using the NetYANG State

The pillar structure from Example 5-10 has been intentionally exemplified, anticipating the OpenConfig models, so we can reuse it here. With that said, we only need to define the SLS and execute as shown in Example 5-15.

Example 5-15. State SLS for the NetYANG state (/etc/salt/states/oc_interfaces/init.sls)

```
interfaces_oc_config:
  napalm_yang.configured:
    - data: {{ salt.pillar.get('openconfig-interfaces') | json }}
    - models:
      - models.openconfig_interfaces
```

Without having to write an environment-specific template, we have the possibility to execute the state and deploy the changes on the device, which creates exactly the same configuration to be loaded on the device, but with much less effort. See Example 5-16.

Example 5-16. OpenConfig interfaces state execution

```
$ sudo salt 'device1' state.sls oc_interfaces
device1:
----------
          ID: interfaces_oc_config
    Function: napalm_yang.configured
      Result: True
     Comment: Configuration changed!
     Started: 16:46:59.262230
    Duration: 7612.234 ms
     Changes:
              ----------
              diff:
                  [edit]
                  +  interfaces {
                  +      xe-0/0/0 {
                  +          description Interface1;
                  +          mtu 9000;
                  +          unit 0 {
                  +              description Subinterface1;
                  +              family inet {
                  +                  address 1.2.3.4/24;
                  +              }
                  +          }
                  +      }
                  +  }
Summary for device1
------------
Succeeded: 1
Failed:    0
------------
Total states run:     1
Total run time:   7.612 s
```

Capirca and the NetACL Salt State Module

Capirca is a mature open source library for multiplatform ACL generation, developed by Google. It simplifies the generation and maintenance of complex filters for more than ten operating systems, including the most common: Cisco IOS, IOS-XR, NX-OS, Arista, Juniper, Palo Alto, and Brocade.

The library requires a configuration file with a specific format, but in Salt the default interpreter is bypassed and we are able to define the data in the format preferred for the pillar, or external pillars in an external service.

The NetACL Salt state requires Capirca to be installed. For the firewall configuration of network devices, this is yet another alternative for quick development without much effort, but it requires a careful read of the documentation in order to understand the caveats. Again, if you discover limitations, consider contributing to Capirca as this is the beauty of open source.

Let's assume we need to automate the configuration of a firewall filter that allows `TCP` traffic from `1.2.3.4` on port `1717`, then counts, and finally rejects anything else.

The pillar structure and the fields are very naturally defined, but there are also platform-specific options such as *counter* or *policer* and we highly recommend consulting the wiki page.

Example 5-17. NetACL pillar structure sample

```
acl:
  - FILTER-EXAMPLE:
      terms:
        - ALLOW-PORT:
            source_address: 1.2.3.4
            protocol: tcp
            port: 1717
            action: accept
            counter: ACCEPTED-PORT
        - DENY-ALL:
            counter: DENY-ALL
            action: deny
```

The state SLS file is again very simple (Example 5-18).

Example 5-18. State SLS salt://firewall/init.sls (/etc/salt/states/firewall/init.sls)

```
state_filter_example:
  netacl.filter:
    - filter_name: FILTER-EXAMPLE
    - pillar_key: acl
```

`netacl.filter` is the name of the state function. The `netacl` state module has three main functions available: `filter`, to manage the configuration of a specific firewall filter; `term`, for the management of a certain term inside a filter; and `managed`, for the entire configuration of the firewall. The `filter_name` field specifies the name of the filter (`FILTER-EXAMPLE`, in this case), as configured in the pillar,

and the `pillar_key` field specifies the Pillar key where the firewall configuration is specified (`acl`, in this case).

Example 5-19 shows everything required for executing the state.

Example 5-19. Execute the NetACL state

```
$ sudo salt device1 state.sls firewall
device1:
----------
          ID: state_filter_example
    Function: netacl.filter
      Result: True
     Comment: Configuration changed!
     Started: 11:58:44.709472
    Duration: 11879.601 ms
     Changes:
              ----------
              diff:
                  [edit firewall family inet]
                  +     /*
                  +      ** $Id: state_filter_example $
                  +      ** $Date: 2017/06/15 $
                  +      **
                  +      */
                  +     filter FILTER-EXAMPLE {
                  +         interface-specific;
                  +         term ALLOW-PORT {
                  +             from {
                  +                 source-address {
                  +                     1.2.3.4/32;
                  +                 }
                  +                 protocol tcp;
                  +                 port 1717;
                  +             }
                  +             then {
                  +                 count ACCEPTED-PORT;
                  +                 accept;
                  +             }
                  +         }
                  +         term DENY-ALL {
                  +             then {
                  +                 count DENY-ALL;
                  +                 discard;
                  +             }
                  +         }
                  +     }

Summary for device1
------------
```

```
Succeeded: 1
Failed:    0
------------
Total states run:     1
Total run time:  11.880 s
```

Moving forward, let's enhance the `ALLOW-PORT` term to allow also `UDP` traffic over the `1719` port.

For flexibility reasons most of the fields can be either a single value or a list of values. With that said, we only need to transform the fields `port` and `protocol` to a list of values (Example 5-20).

Example 5-20. NetACL pillar structure sample: list of values

```
acl:
  - FILTER-EXAMPLE:
      terms:
        - ALLOW-PORT:
            source_address: 1.2.3.4
            protocol:
              - tcp
              - udp
            port:
              - 1717
              - 1719
            action: accept
            counter: ACCEPTED-PORT
        - DENY-ALL:
            counter: DENY-ALL
            action: deny
```

NOTE

The order of the terms is important! The configuration generated and loaded on the device reflects the order defined in the pillar.

In this chapter we have presented three of the most important Salt states for network automation. We encourage you to consider each of them and decide which one is the most suitable to fulfill the needs of your particular environment.

The Salt community also provides pre-written states under the name of *Salt formulas*, which can be downloaded and executed. The user only needs to populate the data into the Pillar. Formulas are also a good resource to learn best practices for maintainable states.

Examples of such formulas include napalm-interfaces for interfaces configuration management on network devices, or napalm-install to install NAPALM and the underlying system dependencies, and so on.

CHAPTER 6

The Salt Event Bus

Salt is a heavily asynchronous tool that comes with all the simplicity, reliability, and difficulty inherent in asynchronicity. Minions can request data and files from the master, send job returns or other data, refresh authentication, and more and all of this can happen at any time. Sometimes this is in response to user actions on either the minion or master; other times it is a scheduled job or routine maintenance.

Salt makes use of an event bus internally to send messages from minion to master and from one process to another. As with most things in Salt, this event bus is also exposed to users to extend, read from, and write to.

Event Tags and Data

A Salt event consists of two things: an event tag and event data. Event tags are a slash-delimited string where each slash provides a level of namespacing. Event data is a dictionary that contains a timestamp plus any arbitrary data useful for that type of event.

For example, when a minion finishes processing a job and sends the job return data back to the master, an event is fired that looks similar to the one shown in Example 6-1.

Example 6-1. salt/job/20170706170156951774/ret/device1

```
{
  "_stamp": "2017-07-06T23:01:57.098368",
```

```
  "...snip...": null
}
```

All Salt events start with `salt/`, all Salt events start with `salt/job/`, and so on, with each slash delimiting a new level of namespacing. The event bus in Salt can get quite busy so event tags are often matched-on to filter unwanted event types. (Event tags that do not contain slashes are legacy tags and should be ignored.)

Any user-generated events should define a unique namespace specific to that user or specific to a particular team or workflow.

Consume Salt Events

Salt events are easily viewed and processed from a variety of sources.

Reactor

Most common is to use Salt's reactor system to match certain events and initiate another Salt action in response. This is described in detail in Chapter 9.

HTTP Stream

`salt-api` via the `rest_cherrypy` module can expose Salt's event bus as an HTTP stream for easy access from external tools or even external systems.

Raw ZeroMQ

And finally, for the adventurous, Salt's event bus is simply a ZeroMQ pub/sub socket that can be consumed by the ZeroMQ bindings for any supported programming language.

Event Types

There are many types of events for different parts of Salt or for various actions. In addition, custom event types are fully supported.

Job Events

A new job event is created when the master generates a job ID and before it broadcasts the new job to listening minions (Example 6-2).

Example 6-2. salt/job/20170706170156951774/new

```
{
  "_stamp": "2017-07-06T23:01:56.953760",
  "arg": [],
  "fun": "test.ping",
  "jid": "20170706170156951774",
  "minions": [
    "device1"
  ],
  "tgt": "*",
  "tgt_type": "glob",
  "user": "shouse"
}
```

Notable fields include the following:

`minions`
: An array of minion IDs that the master expects to receive returns from for the target specified in this job. This is only an educated guess produced by matching the target against minion grains that have been cached on the master—the minions themselves perform the final matching.

`user`
: The person who initiated the command. This value will be the local username or eauth username, and will show if the command was run via sudo. Very useful for keeping long-term audit records of who ran what, where, and when.

Separate job return events are created for each minion when each minion completes and job and delivers the result back to the master (Example 6-3).

Example 6-3. salt/job/20170706170156951774/ret/device1

```
{
  "_stamp": "2017-07-06T23:01:57.098368",
  "cmd": "_return",
  "fun": "test.ping",
  "fun_args": [],
  "id": "device1",
  "jid": "20170706170156951774",
  "retcode": 0,
  "return": true,
  "success": true
}
```

Notable fields include the following:

`success`
: A Boolean value for whether Salt detected any internal errors while running the job. Consult along with "retcode" to determine overall success.

`retcode`
: A return code describing any module-level or system-level errors while running the job. Any non-zero number indicates failure; consult along with "success" to determine overall success.

Authentication Events

Authentication events occur when a minion performs an authentication handshake with the master, or when an unauthenticated minion polls the master to learn if its key has been accepted yet. The master will periodically rotate the AES key it uses to encrypt broadcasts which also triggers a re-auth handshake (Example 6-4).

Example 6-4. salt/auth

```
{
"_stamp": "2017-07-06T23:00:33.076164",
"act": "accept",
"id": "device1",
"pub": "-----BEGIN PUBLIC KEY-----
[...snip...]-----END PUBLIC KEY-----",
"result": true
}
```

The `act` field gives the current status of this minion's key. Possible values are `accept` if the key has been accepted, `pend` if the key has not yet been accepted, and `reject` if the key has been rejected. Useful to automate accepting the key for a new minion, perhaps along with some custom validation logic.

Minion Start Events

Minion start events are triggered each time a minion establishes a connection with a master. The minion is ready to receive jobs from the master once this event is fired, so it is a good event to use to per-

form an initialization action such as running a highstate to bring the minion up to the latest configured state (Example 6-5).

Example 6-5. salt/minion/device1/start

```
{
  "_stamp": "2017-07-06T23:00:54.252873",
  "cmd": "_minion_event",
  "data": "Minion device1 started at Thu Jul  6 17:00:54 2017",
  "id": "device1",
  "pretag": null,
  "tag": "salt/minion/device1/start"
}
```

Key Events

Key events are triggered in response to changes to minion keys via the `salt-key` CLI utility as well as functions in the `key Wheel mod ule`.

Authentication events and minion start events are better choices for most operations that send a command to the minion, because a minion may not be available to respond at the time this event is fired (Example 6-6).

Example 6-6. salt/key

```
{
  "_stamp": "2017-07-07T01:27:44.656330",
  "act": "accept",
  "id": "device2",
  "result": true
}
```

The `act` field gives the action taken for the given key, such as `accept` if it was newly accepted or `delete` if it was deleted.

Presence Events

Presence is a system to determine which minions are currently connected to a master by querying only the local system and not sending executions to minions (such as with `test.ping`).

It works by consulting the `/proc/net/tcp` table and cross-referencing any IPs connected to Salt's publish port with the IP addresses in the local cache of minion grains. It is useful as a light-

weight check only and not for sensitive operations since it does not make use of Salt's authentication system. And it requires that the minion connect to the master using the same IP address it sees locally (i.e., not through a NAT).

The master can generate events that contain minion headcount information. This system can be enabled by adding `presence_events: True` to the master config file.

Example 6-7 generates a list of all currently connected minions, generated every 30 seconds.

Example 6-7. salt/presence/present

```
{
  "_stamp": "2017-07-07T01:38:41.569652",
  "present": [
    "device1"
  ]
}
```

Example 6-8 generates two lists of minion IDs that show changes in the connected minion headcount since the last poll interval. This is useful for detecting and responding to accidentally disconnected minions within a small window of time.

Example 6-8. salt/presence/change

```
{
  "_stamp": "2017-07-07T01:36:43.357588",
  "new": [],
  "lost": [
    "device1"
  ]
}
```

State Events

State events cause each minion to emit progress events during a state run as it completes each individual state function in the state tree, as seen in Example 6-9. They are another opt-in event type and can be enabled by adding `state_events: True` to the Salt master config file.

Example 6-9. salt/job/20170706200340461543/prog/device1/0


```
{
  "_stamp": "2017-07-07T02:03:40.892345",
  "cmd": "_minion_event",
  "data": {
    "len": 4,
    "ret": {
      "__id__": "stage_one",
      "__run_num__": 0,
      "changes": {},
      "comment": "Success!",
      "duration": 0.75,
      "name": "stage_one",
      "result": true,
      "start_time": "20:03:40.877168"
    }
  },
  "id": "device1",
  "jid": "20170706200340461543",
  "tag": "salt/job/20170706200340461543/prog/device1/0"
}
```


Notable fields include the following:

`data.len`
: The total number of states in the current state run.

`data.ret.__run_num__`
: A counter (zero-indexed) that keeps track of the order that each state function was run. Progress through the entire state run can be calculated as a percentage via `(data.ret.__run_num__ + 1) / data.len * 100`.

This chapter introduced the Salt event bus. This constitutes the basic knoweledge for the event-driven infrastructure. In the following chapters you will learn how to interact directly with the bus, how to generate events, and how to consume them and trigger actions.

CHAPTER 7

Beacons

By default, beacon modules run on a fast, one-second interval and emit Salt events. Their main function is to import external events on the Salt bus. For this reason, they pair quite well with the Salt reactor (introduced in Chapter 9). For example, the `inotify` beacon can watch a file or directory and emit an event if any files are modified, then the reactor can kick off a state run to put the file back into the desired state.

There are many beacon modules that cover a wide variety of tasks. To name just a few: `btmp` and `wtmp` watch user logins; `sh` watches user shell activity; `diskusage`, `load`, `memusage`, and `network_info` regularly poll relevant system stats and emit events if a threshold has been crossed; `status` can emit regular events containing current system stats; `log` and `journald` can watch log files for certain patterns and emit an event with matching log entries; `haproxy` and `service` can monitor services and emit events if the service is down or overloaded. Like other Salt modules, new beacon modules are easy to write if you have a custom need.

Configuration

Beacon configuration lives in the Salt minion configuration file under a `beacons` key. As usual, make sure any dependencies for the module are also installed on the minion, and restart the minion daemon after making changes to the config file.

Beacons can be equally used to ensure that processes are alive, and restart them otherwise. Considering that a number of proxy minion processes are executed on a server which is managed using the regular Minion, we can use the `salt_proxy` beacon to keep them alive.

Remember: the proxy minions manage the network devices, while the regular minion manages the server where the proxy processes run.

Consider the following beacon configuration, which mantains the alive status of the proxy processes managing our devices from previous examples in just a few simple lines:

```
beacons:
  salt_proxy:
    - device1: {}
    - device2: {}
    - device3: {}
```

After restarting the minion process, we can observe events with the following structure on the Salt bus:

```
salt/beacon/minion1/salt_proxy/  {
    "_stamp": "2017-08-25T10:17:20.227887",
    "id": "minion1",
    "device1": "Proxy device1 is already running"
}
```

`minion1` is the ID of the minion that manages the server where the proxy processes are executed.

In case a proxy process dies, the `salt_proxy` beacon will restart it, as seen from the event bus:

```
salt/beacon/minion1/salt_proxy/  {
    "_stamp": "2017-08-25T10:17:31.503653",
    "id": "minion1",
    "device1": "Proxy device1 was started"
}
salt/minion/device1/start  {
    "_stamp": "2017-08-25T10:17:42.676464",
    "cmd": "_minion_event",
    "data": "Minion device1 started at [...]",
    "id": "device1",
    "pretag": null,
    "tag": "salt/minion/device1/start"
}
```

A good approach to monitor the health of the minion server is using the `status` beacon, which will emit the system load average every 10 seconds:

```
beacons:
  status:
    - interval: 10  # seconds
    - loadavg:
      - all
```

The event takes the following format:

```
salt/beacon/minion1/status/2017-08-11T09:28:28.233194      {
    "_stamp": "2017-08-11T09:28:28.240186",
    "data": {
        "loadavg": {
            "1-min": 0.01,
            "15-min": 0.05,
            "5-min": 0.03
        }
    },
    "id": "minion1"
}
```

The `status` beacon—together with others such as `diskusage`, `memusage`, `network_info` or `network_settings`—can also be enabled when managing network gear that permits installing the Salt minion directly on the platform, in order to monitor their health from Salt, and eventually automate reactions.

See the documentation for each beacon module for how to configure it and when and what it will emit. The events can be seen on the master using the `state.event` Runner, and the reactor (see Chapter 9) can be configured to match on beacon event tags.

Troubleshooting

The best way to troubleshoot beacon modules is to start the minion daemon in the foreground with trace-level logging: `salt-minion -l trace`. Look for log entries to see if the module is loaded successfully, and then watch for log entries that appear for each interval tick to make sure the beacon is running.

CHAPTER 8
Engines

Engines are another interface that interacts directly with the event bus. While beacons are typically used to import events from external source, engines can be designed bidirectionally. That means they can both import external events and translate them into structured data that can be interpreted by Salt, or export Salt events into different services.

Engines Are Easy to Configure

As most Salt subsystems, engines can be configured on the master or the Minion side depending on the application requirements.

They are configured via a top-level section in the master or (proxy) minion configuration. The following example is an excellent way to monitor the entire Salt activity in real time by pushing the events into *Logstsh*, via HTTP(S):

```
engine:
  - http_logstash:
      url: https://logstash.s.as1234.net/salt
```

Under the `engine` section we can define a list of Engines, each having its particular settings. In this example, for the `http_logstash` engine we have only configured the `url` of the Logstash instance where to log the Salt events.

There are several engines by default embedded into Salt, any of them having a potential to be used in the network automation environment, directly or indirectly, for various services, including Docker,

Logstash, or Redis. Engines can be equally used to facilitate "ChatOps", where they forward the requests between a common chat application, such as HipChat or Slack, and the Salt master.

napalm-logs and the napalm-syslog Engine

For event-driven, multivendor network automation needs, beginning with the release codename *Nitrogen* (`2017.7`), Salt includes an engine called *napalm-syslog*. It is based on *napalm-logs*, which is a third-party library provided by the NAPALM Automation community.

The napalm-logs Library and Daemon

Although written and maintained by the NAPALM Automation community, `napalm-logs` has a radically different approach than the rest of the libraries provided by the same community. While the main goal of the main NAPALM library is to ease the connectivity to various network platforms, `napalm-logs` is a process running continuously and listening to syslog messages from network devices. The inbound messages can be directly received from the network device, via UDP or TCP, either retrieved from other applications including Apache Kafka, ZeroMQ, Google Datastore, etc. The interface ingesting the raw syslog messages is called *listener* and is pluggable, so the user can extend the default capabilities by adding another method to receive the messages. `napalm-logs` processes the textual syslog messages and transforms them into structured objects, in a vendor-agnostic shape. The output objects are JSON serializable, whose structure follows the OpenConfig and IETF YANG models.

For example, the syslog message shown in Example 8-1 is sent by a Juniper device when a NTP server becomes unreachable.

Example 8-1. Raw syslog message from Junos

```
<99>Jul 13 22:53:14 device1 xntpd[16015]: NTP Server
172.17.17.1 is Unreachable
```

A similar message, presenting the same notification, sent by a device running IOS-XR, looks like Example 8-2.

Example 8-2. Raw syslog message from IOS-XR

```
<99>2647599: device3 RP/0/RSP0/CPU0:Aug 21 09:39:14.747 UTC:
ntpd[262]: %IP-IP_NTP-5-SYNC_LOSS : Synchronization lost :
172.17.17.1 : The association was removed
```

The messages examplified here have a totally different structure, although they present the same information. That means, in multi-vendor networks, we would need to apply different methodologies per platform type to process them.

But using `napalm-logs`, their representation would be the same, regardless of the platform, as in Example 8-3.

Example 8-3. Structured napalm-logs message example

```
{
  "error": "NTP_SERVER_UNREACHABLE",
  "facility": 12,
  "host": "device1",
  "ip": "127.0.0.1",
  "os": "junos",
  "severity": 4,
  "timestamp": 1499986394,
  "yang_message": {
      "system": {
          "ntp": {
              "servers": {
                  "server": {
                      "172.17.17.1": {
                          "state": {
                              "stratum": 16,
                              "association-type": "SERVER"
                          }
                      }
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-system"
}
```

The object under the `yang_message` key from Example 8-3 respects the tree hierarchy standardized in the `openconfig-system` YANG model.

Each message published by `napalm-logs` has a unique identification name, specified under the `error` field which is platform-independent.

`yang_model` references the name of the YANG model used to map the data from the original syslog message into the structured object.

These output objects are then published over different channels, including ZeroMQ (default), Kafka, TCP, etc. Similar to the *listener* interface, the *publisher* is also pluggable.

By default, all the messages published by `napalm-logs` are encrypted and signed, however this behavior can be disabled—though doing so is *highly* discouraged.

Due to its flexibility, `napalm-logs` can be used in various topologies. For example, you might opt for one daemon running in every datacenter, securely publishing the messages to a central collector. Another approach is to simply configure the network devices to send the syslog messages to a `napalm-logs` process running centrally, where multiple clients can connect to consume the structured messages. But many other possibilities exist beyond these two examples—there are no design constraints!

The napalm-syslog Salt Engine

The `napalm-syslog` Salt Engine is a simple consumer of the `napalm-logs` output objects: it connects to the publisher interface, constructs the Salt event tag, and injects the event into the Salt bus. The *data* of the event is exactly the message received from `napalm-logs`, while the *tag* contains the napalm-logs error name, the network operating system name and the hostname of the device that sent the notification (Example 8-4).

Example 8-4. Salt event imported from napalm-logs

```
napalm/syslog/junos/NTP_SERVER_UNREACHABLE/device1 {
  "yang_message": {
... snip ...
  }
}
```

CHAPTER 9

Salt Reactor

The reactor is an engine module that listens to Salt's event bus and matches incoming event tags with commands that should be run in response. It is useful to automatically trigger actions in immediate response to things happening across an infrastructure.

For example, a file changed event from the `inotify` beacon could trigger a state run to restore the correct version of that file, and a custom event pattern could post info/warning/error notifications to a Slack or IRC channel.

The reactor is an engine module (see Chapter 8) and uses the event bus (Chapter 6) so it will be helpful to read those chapters before this one. In addition, the reactor is often used to respond to events generated by beacon (Chapter 7) or engine modules.

Getting Started

Salt's reactor adheres to the workflow: match an event tag; invoke a function. It is best suited to invoking simple actions, as we'll see in "Best Practices" on page 83. The configuration is placed in the master config file and so adding and removing a reaction configuration will require restarting the `salt-master` daemon.

To start we will create a reaction that listens for an event and then initiates a highstate run on that minion (Example 6-5). The end result is the same as with the `startup_states` setting except that the

master will trigger the state run rather than the minion. Add the code shown in Example 9-1 to your master config.

Example 9-1. /etc/salt/master

```
reactor:
  - 'napalm/syslog/*/NTP_SERVER_UNREACHABLE/*':
    - salt://reactor/exec_ntp_state.sls
```

As is evident from the data structure, we can listen for an arbitrary list of event types and in response trigger an arbitrary list of SLS files. The configuration from Example 9-1 instructs the reactor to invoke the *salt://reactor/exec_ntp_state.sls* reactor SLS file, whenever there is an event on the bus matching `napalm/syslog/*/NTP_SERVER_UNREACHABLE/*`, the asterisk meaning that it can match anything. For example, this pattern would match the tag from Example 8-4—that is, `napalm/syslog/junos/NTP_SERVER_UNREACHABLE/device1`. In other words, whenever there is a `NTP_SERVER_UNREACHABLE` notification, from any platform, from any device, the reactor system would invoke the `salt://reactor/exec_ntp_state.sls` SLS.

The reactor SLS respects all the characteristics presented in "Extensible and Scalable Configuration Files: SLS" on page 11, with the particularity that there are two more special variables available: `tag`, which constitutes the *tag* of the event that triggers the action, and `data`, which is the *data* of the event. Next, we will create the reactor file, shown in Example 9-2 (you'll also need to create any necessary directories).

Example 9-2. salt://reactor/exec_ntp_state.sls

```
triggered_ntp_state: ❶
  cmd.state.sls: ❷
    - tgt: {{ data.host }} ❸ ❹
    - arg:
      - ntp
```

Let's unpack that example, line by line:

❶ The ID declaration (i.e., `triggered_ntp_state`) is best used as a human-friendly description of the intent of the state.

❷ You'll notice that the function declaration differs from Salt states in that it is prefixed by an additional segment, `cmd`. This denotes the `state.sls` function will be broadcast to minion(s) exactly like the `salt` CLI program. Other values are `runner` and `wheel` for master-local invocations.

❸ The `tgt` argument is the same value the `salt` CLI program expects. So is `arg`. In fact, this reactor file is *exactly* equivalent to this CLI (Jinja variables replaced): `salt --async device1 state.sls ntp`. Both the CLI and the reactor call to Salt's Python API to perform the same task; the only difference is syntax.

❹ In this case, `host` is the field from the event `data`, which can be used to target the minion, when the ID is the same value as the hostname configured on the device. There can be many match possibilities, depending on the pattern the user chooses to define the minion IDs.

Looking at the entire setup, when the `napalm-syslog` engine is started, in combination with the configuration bits from Examples 9-1 and 9-2, we instruct Salt to automatically run the `ntp` state when the device complains that a NTP server is unreachable. This is a genuine example of event-driven network automation.

Best Practices

The reactor is a simple thing: match an event tag, invoke a function. Avoid anything more complicated than that. For example, even invoking two functions is probably too much. This is for two reasons: debugging the reactor involves many, heavy steps; and the reactor is limited in functionality *by design*.

The best place to encapsulate running complex workflows from the Salt master is, of course, in a Salt orchestrate file—and the reactor can, of course, invoke an orchstrate file via `runner.state.orch`. Once again this is exactly equivalent to the CLI command `salt-run state.orch my_orchestrate_file pillar='{param1: foo}'`:

```
something_complex:
  runner.state.orch:
    - mods: my_orchestrate_file
```

```
    - pillar:
        param1: foo
```

Invoking orchestrate from the reactor has two primary benefits:

- The complex functionality can be tested directly from the CLI using `salt-run` without having to wait for an event to be triggered. And the results can be seen directly on the CLI without having to look through the master log files. Once it is working, just call it from the reactor verbatim.
- This functionality can be invoked not only by the reactor but by anything else in the Salt ecosystem that can invoke a Runner, including other orchestrate runs. It becomes *reusable*.

For very complex workflows where the action is triggered as a result of multiple events or aggregate data, we recommend using the Thorium complex reactor.

Debugging

The easiest way to debug the reactor is to stop the `salt-master` daemon, and then to start it again in the foreground with debug-level logging enabled: `salt-master -l debug`. The only useful in-development logging the reactor performs is at the debug level. There are primarily two log entries to search for:

- `Compiling reactions for tag <tag here>`
- `Rendered data from file: <file path here>`

The first log entry will tell you whether the incoming event tag actually matched the configured event tag. Typos are common and don't forget to restart the `salt-master` daemon after making any changes. If you don't see this log message troubleshoot that before moving on.

The second log entry will contain the *rendered* output of the SLS file. Read it carefully to be sure that the file Jinja produced is valid YAML, and is in the correct format and will call the function you want using the arguments you want.

That's all there is to debugging the reactor, although it can be harder than it sounds. Remember to keep your reactor files simple! Once

you have things working, stop the `salt-master` daemon and then start it again using the init system as normal.

Acknowledgments

From Mircea

To the many people I am constantly learning from, including my Cloudflare teammates and the network and Salt communities. Furthermore, I would like to extend my gratitude to Jerome Fleury, Andre Schiper, and many others who believed in me, and taught me about self-discipline and motivation.

From Seth

Thanks to my coworkers at SaltStack and to the Salt community. Both have been a constant source of interesting and fascinating discussions and inspiration over the last (nearly) seven years. The Salt community is one of the most welcoming that I have been a part of and it has been a joy.

Also a sincere thank you to our technical reviewers, Akhil Behl and Eric Chou. Your suggestions and feedback were very helpful.

About the Authors

Mircea Ulinic works as a network engineer for Cloudflare, spending most of his time writing code for network automation. Sometimes he talks about the tools he's working on and how automation really helps to maintain reliable, stable, and self-resilient networks. Previously, he was involved in research and later worked for EPFL in Switzerland and a European service provider based in France. In addition to networking, he has a strong passion for radio communications (especially mobile networks), mathematics, and physics. He can be found on LinkedIn, Twitter as @mirceaulinic, and at his website.

Seth House has been involved in the Salt community for six years and has worked at SaltStack for five years. He wrote the salt-api and also contributed to many core parts of Salt. He has collaborated with the Salt community and started the Salt Formulas organization. Seth has given over 30 introductions, presentations, and training sessions at user groups and conferences and created tutorials on Salt for companies. He has designed and helped fine-tune Salt deployments at companies all across the United States.