Automation Tools

No discussion of network automation would be complete without evaluating the role that automation tools—tools like Ansible, Chef, Puppet, Salt, Stacktorm, and Terraform—play in a network automation context.

Traditionally, these tools have been more focused on the server automation use case. This was an understandable focus given that most, if not all, of these tools had their origins in automating server operating systems and managing operating system (OS) and/or application configuration. In recent years, though, there has been a great deal of effort by a number of companies to enhance the network automation functionality of their products. These enhancements make these products much more useful and powerful in a network automation use case.

Aside from automated configuration management, the primary case in the beginning, with the advent of dynamic infrastructure services offered mainly by public cloud providers, tooling has evolved and new actors have entered the playground in order to enable the Infrastructure as Code (IaaS) paradigm.

In this chapter, we'll discuss how to use some major automation tools in the context of network automation. The tools we'll cover in this chapter are:

- Salt
- Stackstorm

Before we get into the details and examples of how to use these tools for network automation, let's first take a quick look at an overview of the various tools we're going to discuss.

Reviewing Automation Tools

While all of these tools are focused on automation, each tool has its own architecture and approaches automation in a slightly different way. This gives each tool its own set of strengths and weaknesses. In this section, we'd like to quickly review each of the tools so that you can begin to see how these tools might be used in their environment.

At a high level, some of the major architectural/conceptual differences between the tools include:

Configuration Management vs Infrastructure Provisioning

Infrastructure provisioning is the process used to create infrastructure - network services, virtual machines, databases, etc. - and configuration management is the process of automating the installation of software components and performing configuration management tasks. So, you could understand infrastructure provisioning as a Day 0 activity, and configuration management as a Day 1 activity. In spite of most automation tools being able to achieve both, this chapter will help you to understand where each one shines.

Agent-based versus agentless

Some tools require an agent—a piece of software—to be running on the system or device being managed. In a network automation use case, this could prove to be a problem, as not every network operating system (NOS) supports running agents on a network device. In situations

where the NOS doesn't support running an agent natively on the device, there are sometimes workarounds involving a "proxy agent." Agentless tools, obviously, don't require an agent, and may be more applicable in network automation use cases.

Centralized versus decentralized

Agent-based architectures often also require a centralized "master server." Some agentless products also leverage a master server, but most agentless products are decentralized.

Custom protocol versus standards-based protocol

Some tools have a custom protocol they use; this is often tied to agent-based architectures. Other tools leverage SSH as the transport protocol. Given the ubiquity of SSH in network devices, tools leveraging SSH as their transport protocol may be better suited to network automation use cases.

Domain-specific language (DSL) versus standards-based data formats and general-purpose languages

Some tools have their own DSL; to use this tool, users must create the appropriate files in that DSL to be consumed by the automation tool. A DSL is a language purpose-built for a specific domain (or tool). For organizations that aren't already familiar with the DSL, this might create an additional learning curve. Other tools leverage YAML, which is considered a general-purpose language in this context. Remember, we discussed YAML in [dataformats].

Declarative versus Imperative

There are tools that use a declarative approach to define the final state of the infrastructure, so independently on the definition order, the proper dependencies and tasks will be inferred and executed to enforce the target state. On the other side, other tools use the imperative approach where each step is a procedure to be performed, and the order defines the execution order.

Extensibility

Most of these automation tools support the ability to add or extend functionality using high-level scripting languages. Some tools use Ruby as their language of choice for extending functionality; others leverage Python or Go.

Push versus pull versus event-driven

Some automation tools operate in a "push" model; that is, information is pushed from one place out to the devices or systems being managed. Others operate in a pull model, typically pulling configuration information or instructions (often on some sort of scheduled basis). Finally, there are also event-driven tools, which perform an action in response to some other event or trigger.

Mutable versus Immutable

Traditionally, when you needed to change the configuration of the infrastructure you simply changed its state from the current state by applying some changes, *mutating* its state. This approach is known as the mutable approach, and is how configuration management tools work; On the other side, an *immutable* approach is where you need to *replace*/*restart* the infrastructure with a new one to change its state. So, even the smallest change, such as changing the hostname of a server, would require you to re-provision the server and start from scratch. Depending on the focus, some tools are more suited than others for each case.

State Management

Configuration management tools don't manage the lifecycle of the remote infrastructure. The state is something you can gather at each step, but it's not implicitly tracked. Contrarily, the tools based on the immutable pattern, that need to decide when it's necessary to recreate an infrastructure component, usually keep the state of the remote infrastructure that was provisioned by them.

With this high-level set of architectural differences in mind, let's take a quick look at the three tools we're going to discuss in this chapter.

Salt

Salt can use either an agent-based architecture or an agentless architecture. In an agent-based architecture, Salt agents communicate with the Salt master over a message bus; in the agentless architecture, they communicate via SSH or other third-party libraries such as NAPALM (covered later). Salt is built using Python, and can be extended with Python. Jinja provides default templating functionality. Salt started out as a tool for remote server management, much like Ansible, and has since gained idempotent configuration management via Salt States, which are written in YAML. One distinction to make with Salt is that it is also a platform for event-driven automation beyond general configuration management.

StackStorm

StackStorm takes a dramatically different approach than the other tools listed here. StackStorm focuses solely on event-driven automation; that is, tasks are performed in response to events. StackStorm leverages Python to build sensors that emit events or actions that perform a task. StackStorm uses YAML in several places to provide metadata for sensors or actions, or to define a workflow.

Now, let's dive a bit deeper into each of the products we're going to discuss and take a more in-depth look at how each product can be used for network automation. We've arranged the in-depth discussion of the products in alphabetical order, so we'll start with Ansible.

Automating with Salt

Salt is a robust framework designed as an extremely fast and lightweight communication bus that offers capabilities such as automated configuration management, cloud provisioning, network automation, and event-driven automation, allowing you to achieve modern network operations of your infrastructure.

Salt is very flexible, and allows you to automate a wide variety of network devices and components, as do other tools in this chapter. Despite its somewhat perceived complexity, Salt can be set up in minutes so you can start automating network devices.

Similar to what we did in the previous section on Ansible, our goal is to provide a jump start with enough information so you can use Salt to start automating common network tasks immediately. In order to do this, we've divided this section into five major areas:

- Understanding the Salt architecture
- Getting familiar with Salt

- Using Salt to collect network status
- Managing network configurations with Salt
- Executing Salt functions remotely
- Diving into Salt's event-driven infrastructure

Understanding the Salt Architecture

From an architectural perspective, Salt is designed as a simple core with pluggable interfaces. As you will see throughout this section, *everything* in Salt is pluggable and extensible, including the creation of new device drivers to automate network devices that use different APIs. To that end, Salt can be used to automate any type of network device.

Salt was initially developed to be an agent-based architecture, which wasn't well suited for network automation because, as we know, it's not easy to load software agents on all types of network devices. In fact, it's very hard or even impossible on traditional network equipment. Due to the demand for agentless automation solutions, Salt updated their architecture to offer both agentless and agent-based solutions. Additionally, in either deployment option, Salt facilitates event-driven network automation, which is covered later in this section.

At its core and default setup, Salt is a hub-and-spoke architecture. The hub, or central server, is referred to as the *Salt master* (running software called `salt-master`) and manages the spokes, which are referred to as *Salt minions* (running software called `salt-minion`), which in essence are the nodes being automated. The Salt master has the ability to manage thousands of *minions*. The communication between the master and minions is persistent and uses lightweight protocols to enable real-time communication—this approach allows Salt to scale and manage more than 30,000 minions using a single master server. For even larger designs, it's possible to distribute the minions to multiple master servers, which are eventually managed by a higher-level master.

This is how Salt operates quite commonly when automating servers. To understand how Salt operates when automating network devices, we need to review how Salt operates in an agentless architecture.

Using Salt in an agentless architecture with salt-ssh

The Salt architecture was extended to operate in an agentless mode of operation. In this mode of operation, the target nodes being automated do not have the `salt-minion` software package installed. Rather, another package called `salt-ssh` is used instead and can be installed directly on the master, or distributed on other nodes, as Salt provides a communication bus between all Salt-related processes.

In this design, the master connects to the target device using SSH, which is why this architecture is sometimes compared to Ansible. It's also worth noting that when using `salt-ssh`, you are still able to leverage the full functionality of Salt when automating your infrastructure.

NOTE	The `salt-ssh` subsystem is just another process used within the Salt architecture and can be installed on the master or another system.

Even the agentless mode of operation with `salt-ssh`, however, it hasn't particularly helped yet with

automating network devices due to various transport types, APIs, and network operating systems. This is largely due to the lack of SSH-based integrations that have been built thus far for Salt.

This leads us to the next option that is most applicable to automating network devices, which is using Salt *proxy minions*.

Using Salt in an agentless architecture with proxy minions

Another approach Salt uses for agentless automation uses the concept of a Salt *proxy minion*. A proxy minion is a superset of the minion, thus offering all the features of the regular minions. For all intents and purposes, it is a virtual minion. This virtual minion is not installed on the devices you are automating—they simply proxy access to the devices you are automating. Proxy minions are extensible, offering you the ability to create (or choose) the preferred communication channel from a given proxy minion to the target devices being automated. This is how network automation is performed today with Salt.

NOTE

A device managed, or minion, has a proxy process associated with it on the proxy minion, each consuming about 40 MB RAM. Using the proxy architecture, each proxy minion is capable of managing 100 devices from a proxy machine having only 4 GB RAM available. These characteristics make the proxy minion a solid choice for network automation. The proxy processes are controlled by the master, and very often, they run on the same physical server, but can also be placed in a distributed architecture, improving Salt's scaling capabilities for managing network devices. For example, Salt can automate a network consisting of 10,000 nodes by distributing the proxy minions on 10 machines, with each server running 1,000 proxy minion processes, thus managing 1,000 nodes each.

Automating network devices with Salt

Salt supports network automation through the use proxy minions. Some proxy minions exist specifically for networking. They include:

Netmiko

This natively offers multivendor network automation using Netmiko open source Python library, which we cover in [apis].

NAPALM

Similar to Netmiko, but with the extra functionalities implemented by Napalm library, also covered in [napalm].

Cisco Network Services Orchestrator (NSO)

A commercial solution from Cisco that offers multivendor model-driven network automation primarily using NETCONF.

Juniper

Used to manage Juniper Junos devices and developed by Juniper.

Cisco NX-OS

Used to manage Cisco NXOS devices and developed by SaltStack.

For all of our examples going forward in this chapter, we're going to be strictly focused on using the NAPALM proxy minion to interact with various devices, including Cisco IOS, Cisco NXOS, Arista EOS, and Juniper Junos devices. This was our choice as it's open source, multivendor, and actively being developed, and it offers a structured output for getters.

NOTE	Each one of these modules has a different module identifier, and several functions. For example, you could run a `cli` command using Netmiko (`netmiko.send_command`), Napalm (`net.cli`) or Juniper (`junos.cli`).

Remember, we also used the devices and topology shown in Network topology diagram (repeated from [ansible-network-topology]) for the examples within this section (in addition to the Ansible section).

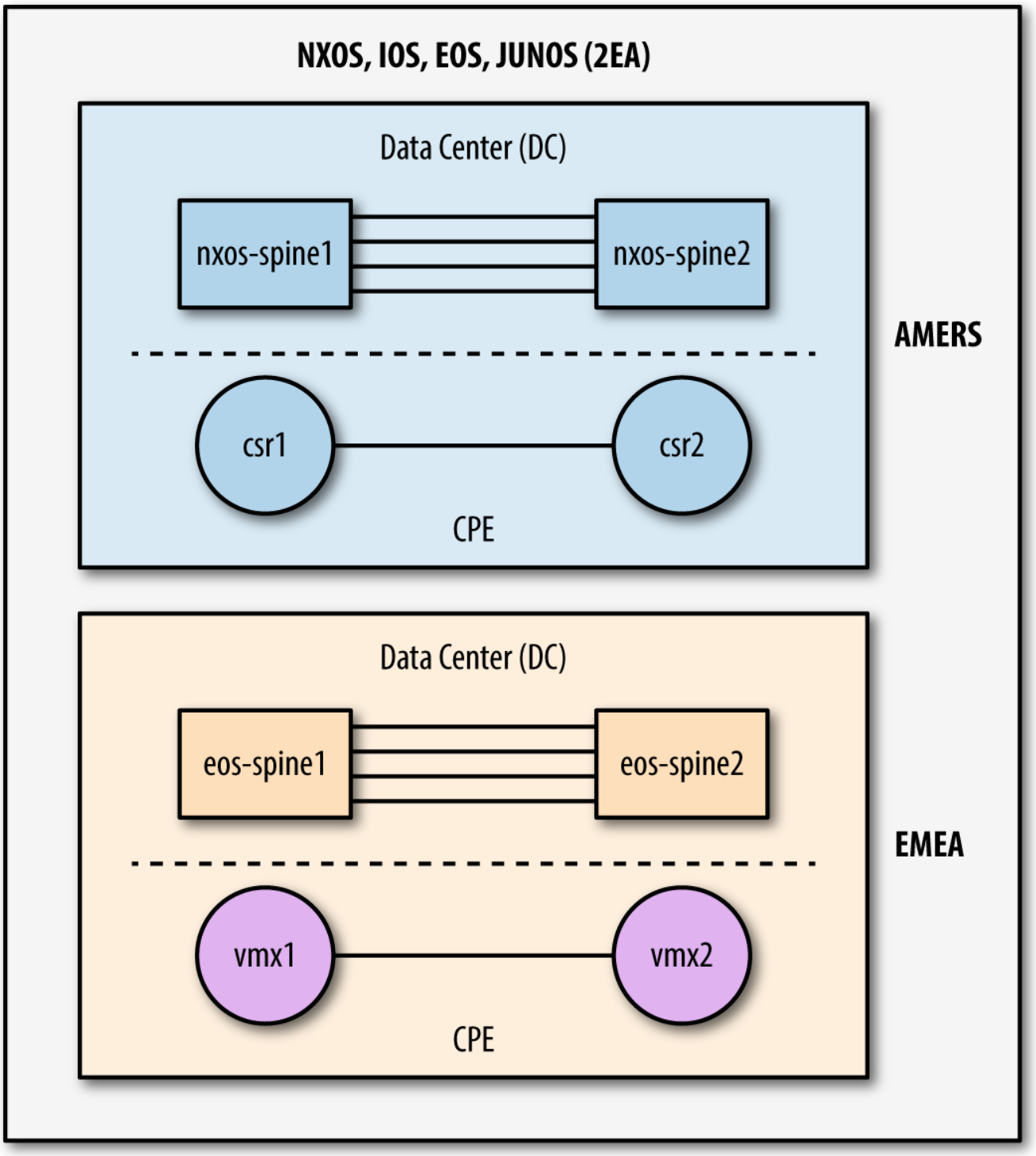

Figure 1. Network topology diagram

NOTE	Again, Salt is very extensible and custom proxy minions can be written for different devices that have unique APIs or legacy interfaces, such as SNMP or Telnet.

Getting Familiar with Salt

There are many terms you should be aware of in order to start using Salt. In order to understand and use the system, you must have an idea of what these concepts are and how they fit into the overall Salt framework. We'll walk through a few of them (pillars, top file, grains, states). First, we'll look at the SLS file format.

Understanding the SLS file format

Throughout this book, you've learned about Jinja templates and YAML files. In both cases, Jinja is just one type of templating language and YAML is just one way to structure data in a very human-readable format. Imagine using a single file that understands Jinja (and other templating languages) and YAML (and other data formats) in order to create different sets of data (the process of inserting data into the template). This is exactly what SLS files are.

SLS is a Salt-specific file format, and it stands for SaLt State. It is a mixture of data representation and templating languages that can be used within the same file.

By default, an SLS file is YAML + Jinja. However, due to the flexibility of Salt and SLS, it can be easily switched to a different combination. This is an example of Salt's pluggability—you are not limited to Jinja and YAML only, but you're able to choose from a variety of options. For example, for data representation you can pick one of the following: YAML, YAMLEX, JSON, JSON5, HJSON, or even pure Python, and for templating, you can pick one of the following: Jinja, Mako, Genshi, Cheetah, Wempy, or, again, pure Python. The list of options can also be extended based on your requirements and preferences.

One case for supporting different types of data representations and templating engines is that it eases migrations from other tools. For example, if you had an internal (or custom) tool using Mako templates, another Python-based templating engine, you could easily use them with Salt—not being forced to use Jinja, as an example.

YAML SLS file is a very basic SLS file that can be written as a pure YAML data file:

YAML SLS file

```
ntp_peers:
  - 10.10.10.1
  - 10.10.10.2
  - 10.10.10.3
```

You can add a Jinja for loop inside the same file to make it more dynamic:

```
ntp_peers:
  {%- for peer_id in range(1, 4) %}
  - 10.10.10.{{ peer_id }}
  {%- endfor %}
```

Showing the power of using SLS data files and other data and template types, the following is an example of using an HJSON data format with a Mako template.

```
#!hjson|mako
ntp_peers: [
  % for peer_id in range(1, 4):
  ${peer_id},
  % endfor
```

```
]
```

These three examples represent exactly the same data—a list of three NTP peers. The last one is a combination of HJSON and Mako—note the shebang at the top of the file specifying this. HJSON is a syntax extension to JSON, making it potentially more human-readable and less error-prone.

As stated earlier, you can also create SLS data files in pure Python. Here is another example that represents the same data:

```
def run():
    return [
      f'10.10.10.{peer_id}' for peer_id in range(1, 4)
    ]
```

While all of these examples are SLS files, they are *data* files. They contain data that we'll eventually want to use to perform network automation tasks such as creating configuration files and configuring devices.

Please note that all of the SLS files noted here would be saved with a *.sls* extension.

Next we'll take a look at what *pillars* are and how they map back to SLS data files.

Understanding pillars

A pillar is a data *resource* that can either be a file that is an SLS file or data pulled from an external service such as a CMDB or another network management platform.

NOTE	When working with pillar files, keep in mind that the Salt master configuration file, which we cover in the next section, needs to have the proper paths defined for where you will store your pillars.

Within pillar files, you store all data required to manage network devices. This includes any common information such as authentication credentials, but also includes the actual configuration data for anything you wish to configure on the device, from interface configuration and protocols configuration to the BGP or NTP configuration.

Pillar file using SLS format is an example pillar file using the SLS file format:

Example 1. Pillar file using SLS format

```
---
proxy:
  proxytype: napalm
  driver: ios
  host: csr1
  username: ntc
  password: ntc123
hostname: csr1
```

```
'openconfig-bgp':
  bgp:
    global:
      config:
        as: 65001
        router_id: 172.17.17.1
```

NOTE	For our deployment, this pillar is added to the Salt master. It is then distributed to all proxy minions, and in our case, we only have one proxy minion that is installed directly on the master server.

In the preceding pillar file, there are three keys defined. The first is called `proxy`, which is a special Salt keyword that requires key-value pairs that map to the specific proxy minion being used. The other keys, `hostname` and `openconfig-bgp`, are arbitrary user-defined keys that we're defining as they contain data values we want to configure and send to the network device.

In our example, this pillar was saved as */srv/pillar/csr1_pillar.sls*. This particular pillar is device-specific, but as we'll see later in this section, they can also be broader for storing data used across a set of devices.

NOTE	To avoid exposing sensitive data, you can encrypt the data using GPG and Salt will decrypt it during runtime, or you can store it in a secured external pillar (for example, Hashicorp Vault).

For large deployments, you may want to retrieve data from some external system that already exists internal to your organization rather than manage large quantities of pillar files. For these use cases, it's possible to have external pillars. External pillars can be any external services including, but not limited to, databases, Git repositories, HTTP APIs, or even Excel files. The complete reference can be found at Salt external pillars documentation.

A common use case for network automation is fetching the data from an IP address management (IPAM) solution. Considering that most IPAM solutions expose data through an HTTP-based API, the next three lines could also be added to a pillar file:

```
ext_pillar:
  - http_yaml:
      url: https://my-ipam.org/api/<node>
```

In this case, all data returned from the IPAM can be data leveraged in some fashion when executing a Salt task such as rendering data into a template that'll be used to generate configurations.

Understanding the top file

We're now aware of the SLS file format and pillar data files that leverage the SLS file format. Another type of file in Salt that uses the SLS file format is called the *top file*.

The top file, often referred to simply as *the top*, defines the mapping between a minion or groups of

minions and the data (through the use of pillars) that should be applied to them. To a certain extent, you can look at the top file as being similar to an Ansible inventory file, which we covered in the last section, but there are in fact many differences that you'll see.

Within the top file, you have the ability to specify which pillar(s) are assigned to which device(s). When new devices are added to Salt management, they are identified by a unique minion ID—this ID is assigned by you, the user. You can then reference this ID and map specific pillars (data) to the new device, or create broader groups based on device type, site, or region.

NOTE	The top file is commonly defined as *top.sls*. Our file was saved as */srv/pillar/top.sls*.

Basic Salt Top File is a basic example of a top file that uses exact matches based on the minion ID and matches each device to a pillar data file.

Example 2. Basic Salt Top File

```
---
base:
  csr1:  # minion id
    - csr1_pillar  # pillar mapped to csr1
  vmx1:
    - vmx1_pillar
  nxos-spine1:
    - nxos_spine1_pillar
  eos-spine1:
    - eos_spine1_pillar
```

In this basic example, the minion with the ID `nxos-spine1` uses the *nxos_spine1_pillar.sls* pillar.

NOTE	Take notice of the `base` keyword as the root key in the top file. In Salt, `base` is a reserved keyword indicating that this is the *default* environment being managed by this Salt system. Thus, as you can imagine, you can manage different environments (prod, test, DR, QA) with Salt and reference them using different keys in your top file. We are using the default, or "base," environment for our examples.

As we alluded to, you may want to map a single pillar data file that contains certain configuration inputs for a certain device type to more than one device. In this case, you don't use the minion ID. You can use more advanced methods such as shell-like globbing and regular expressions, or even use device characteristics including grains, which we cover in an upcoming section, or pillar data.

Let's take a look at a few more examples of more realistic and advanced top files that leverage shell-like globbing and regular expressions.

The following example maps pillars to devices using characteristics about a device, called Salt grains, including vendor and OS version:

```
---
```

```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
```

In this example, the *junos.sls* pillar is loaded only for devices that are identified as manufactured by Juniper using the `vendor` characteristic. Again, these characteristics are called *grains*, which we cover in the next section. For now, you can see the `G@`, which indicates grains are being used. Similarly, the *ios_16.sls* pillar is mapped and loaded for all devices that are IOS and are version 16.X. Finally, you can also see in the last example, the `spine` pillar is loaded for any spine device (e.g., `nxos-spine1` or `eos-spine2`). In this example, regular expressions are being used—note the `E@` (expression). The minion ID must contain any characters (`.*`), followed by `-spine`, then followed by a single digit (`\d`) to match the spine devices in our topology.

You can also have default pillars that you want to apply to all devices. For example, to load a pillar, which we defined previously, called *ntp_peers.sls*, you can add the following to the top file:

```
  '*':
    - ntp_peers
```

In this case, you can ensure that the entire network uses the same set of NTP peers.

You can also define custom groups based on your own business logic. To map a custom group of devices identified by a user-defined name (and a little more analogous to what's defined with Ansible), we need to use the `nodegroups` key in the Salt master configuration file—ours is stored at */etc/salt/master*:

```
---
nodegroups:
  amers:
    - 'csr*'
    - 'or'
    - 'nxos-spine*'
  emea:
    - 'vmx* and G@os:junos'
    - 'or'
    - 'eos-spine*'
```

NOTE Don't worry, we cover the Salt master configuration file in more detail in an upcoming section too.

There are now groups defined called `amers` and `emea`, such that the `amers` groups all devices whose minion ID starts with *csr* or *nxos-spine*, while `emea` groups devices whose ID starts with *eos-spine*, or running Junos and their ID starts with *vmx*.

Once these groups are defined in the master configuration file, they can be referenced in the top file. In the next example, pay attention to the two new keys called `N@emea` and `N@amers`. They are referencing the node groups (`N@`) that were just defined in the master configuration file.

```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
  'N@emea':
    - communities_emea
  'N@amers':
    - communities_amers
```

This assumes two pillars for BGP communities were created, *communities_amers.sls* and *communities_emea.sls*.

TIP

Don't forget that the top file is still SLS, thus Jinja + YAML by default, which can be leveraged to generate dynamic mappings. For example, if we have a longer list of regions, the last example could be written like this:

```
base:
  'G@vendor:juniper':
    - junos
  'G@os:ios and G@version:16*'
    - ios_16
  'E@(.*)-spine(\d)':
    - spine
  {% for region in ['emea', 'amers', 'apac'] -%}
  'N@{{ region }}':
    - communities_{{ region }}
  {% endfor -%}
```

While our focus is on getting started with Salt, you should be aware that you can integrate Salt to use external systems that offer more *dynamic tops*. Rather than a top file, you'd use an external service. This is helpful if you already have inventory and groupings in some other internal system or tool.

Understanding grains

We've already alluded to grains, but now we'll cover them in a little more detail. Remember that we've already defined pillars in SLS files. Thus, pillars are data provided by the user. In contrast, grains represent data gathered by Salt.

Grains are information that Salt collects about a given device such as device vendor, model, serial

number, OS version, kernel, DNS, disks, GPUs, and uptime. You don't need to do anything with this data, but you should be aware that this data exists because it has *many* uses. For example, we've already shown how you can leverage grains in top files. You can also use this data in templates, conditional statements, and reports.

NOTE	*Grains* is a Salt-specific term, but in other tools, this type of data is often referred to as *facts*. However, please note that they are not quite equivalent—grains are purely static data and they are cached. Dynamic details (such as interfaces details, BGP configuration, LLDP neighbors) is retrieved on runtime, via Salt *execution modules*.

Additionally, you have the ability to create your own grains either by using custom Salt integrations in the form of execution modules or by statically defining them in files. One option is to statically store grains data in the proxy minion configuration file as shown here:

```
grains:
  role: spine
  production: true
```

TIP	Before adding a new grain, it is recommended that you evaluate how dynamic the information is. Grains are more suitable for data very unlikely to change; otherwise, storing the data in a pillar is the preferred option.

Understanding states, state SLS files, and state modules

Salt States are modules used to manage, maintain, and enforce configuration. They are a declarative or imperative representation of a system configuration. Having the source of truth in the pillar, the state compares it with the current configuration, then decides what is required to be removed and what has to be added. Given modern network devices able to apply atomic configurations, it is even easier. In that case, we only need to generate the expected configuration and let the device compute the difference.

In cases where the device does not have such capabilities, or it's more optimal to determine the difference ourselves, we need one additional step, as we illustrate in Creating Jinja network configuration templates.

Understanding the state SLS

The state SLS is a descriptor that defines which states will be executed when the state is applied. Each state is identified by a unique `state_name` that you define, which invokes a state function (built into Salt) passing a list of arguments.

```
<state_name>:
  <state_function>:
    - list of state arguments
```

Remember the following when you start working with state SLS files.

- `state_name` is an arbitrary name assigned.
- `state_function` is the state function we want to execute.

NOTE	Do not conflate the state SLS with the state module: the latter is a Python module that processes the arguments, executes the code, and produces the result, while the state SLS invokes one or more state functions.

After the introduction of the basic Salt concepts, you are ready to start playing with Salt.

Using Salt to Collect Network Status

In this subsection, we will bring a local Salt environment to live, with all the necessary configuration, to collect data from network devices. The first step is the master Salt configuration file.

Updating the master configuration file

We've made reference to various files that are used within Salt, such as pillars and templates. These types of files need to be stored in particular locations on your master server. You define these locations within the master configuration file.

The master configuration file is a YAML file that is preconfigured with default options as soon as Salt is installed. The default path for the master configuration file is either */etc/salt/master* or */srv/master*.

The list of options that can be configured in the master configuration file is long, but two of the most important are configuring `file_roots` and `pillar_roots`. These are *keys* in the config file—remember the file is YAML based.

Within `file_roots` you specify the paths local to the master server, where different files are stored, such as templates, states, pillars, and extension modules. The structure used is also flexible enough to allow you to have different environments on the same machine (e.g., production, test, and DR).

Here is a snippet from a configuration file that configures the `file_roots`:

```
file_roots:
  base:
    - /srv/salt
    - /etc/salt/templates
    - /etc/salt/states
    - /etc/salt/reactors
```

Note `base`, the special keyword in Salt we mentioned earlier. When used, it designates the respective paths that map to the *default* environment (since you can define multiple environments each as a different key). For example, if you wanted to define an environment called `dev` used for development only, the structure would be the following:

```
file_roots:
```

```
  dev:
    - /home/ntc/pillar
    - /home/ntc/states
```

The structure of `pillar_roots` is very similar to that of the `file_roots`, pointing to the directory where the pillar files are stored:

```
pillar_roots:
  base:
    - /srv/pillar
```

Similar to using environments for `file_roots`, you could subsequently update `pillar_roots` to support a development environment too.

NOTE	It is not required to define templates within a flat directory such as */srv/template*. There are designs where the templates are actually stored under each state, grouped more logically by how they're used and what devices are using them.

Once you have a base configuration on the master, the next step will be to perform similar tasks on the proxy minion, specifically when automating network devices.

Updating the minion and proxy minion configuration file

The minion has its own configuration file. The default paths supported are */etc/salt/minion* and */srv/minion*. As the proxy minion is a superset of the regular minion, it inherits all the options (YAML keys) supported by the minion configuration.

The proxy configuration file is stored either at */etc/salt/proxy* or */srv/proxy* (depending on the OS).

The most relevant configuration parameter is the *master* server location. If it can't be resolved, the minion process will fail to start.

Salt dev environment configuration

We already mentioned the distributed nature of Salt's architecture. However, you can also run all the components in one server. In Salt local development environment, you can observe the different processes running: the Salt Master, one minion process to manage the local server, and the multiple proxy minions, to connect to the network devices.

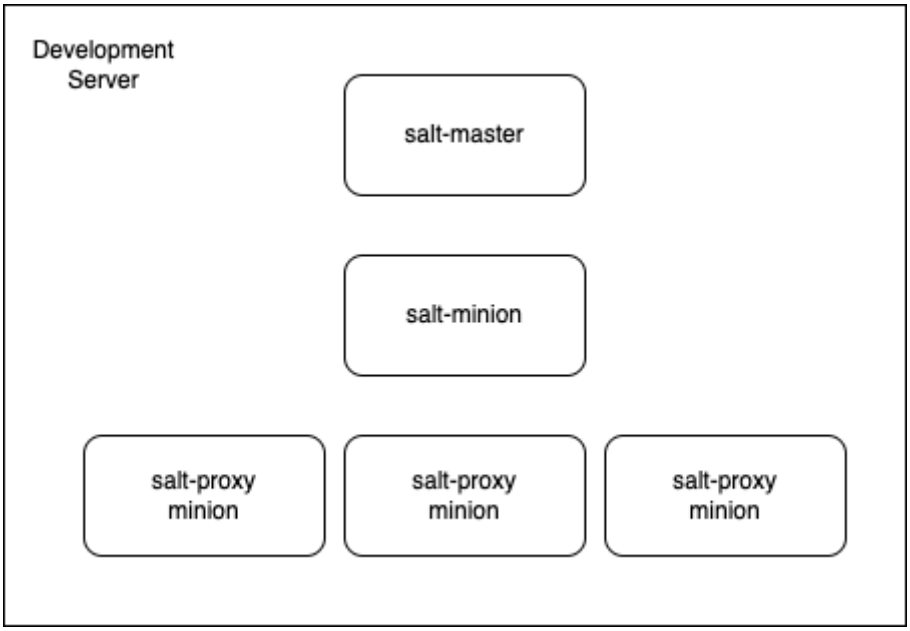

Figure 2. Salt local development environment

When you have installed `salt-master` and `salt-minion` on your computer, you can adjust the configuration files introduced above.

In our local environment, we updated the server minion configuration file (`/etc/salt/minion`) and the proxy one (`/etc/salt/proxy`), to point to master, via `localhost` (`master: localhost`)

Then, we are ready to start the Salt processes, using whatever service management utility you prefer.

```
$ sudo systemctl start salt-master
$ sudo systemctl start salt-minion
```

NOTE Remember to use the proper service manager for your OS platform. This example, with `systemctl`, works for Systemd based platforms (newer Debian, openSUSE, Fedora). On Ubuntu and older Fedora/RHEL using Upstart, you would use the `service salt-master start` syntax.

Like other services, Salt keeps updating logs in `/var/log/salt/` directory. When something is not working as expected, checking the service's logs can give the right information to debug and fix the issue. You have one file for each service:

```
$ tree /var/log/salt/
/var/log/salt/
├── key
├── master
├── minion
└── proxy
```

Hopefully, right now, we have the master salt and the local server minion running. Now, you can start using the `salt` commands to verify the status of the minion.

Verifying minions are up with the test module

In any size deployment, verifying minions are up and functional is a critical step in troubleshooting. You can accomplish this using the `test` module, and more specifically the `test.ping` function.

```
$ sudo salt "*" test.ping
No minions matched the target. No command was sent, no jid was assigned.
ERROR: No return received
```

Why we can't reach the local minion if it is running? Salt implements a key acceptance mechanism, and, by default, the keys are not accepted. We can check the status of the keys with `salt-key --list -all`.

```
$ sudo salt-key --list-all
Accepted Keys:
Denied Keys:
Unaccepted Keys:
ntc
Rejected Keys:
```

We can spot that the local minion key is not accepted by Salt master. We need to explicitly approved them to start communicating with the minion. The same mechanism also applies to the proxy-minions.

Example 3. Approve Salt Keys

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

Once the key is accepted, if we rerun the `test.ping` again, we verify that it is reachable now.

```
$ sudo salt "*" test.ping
ntc:
    True
```

`test.ping` is a simple function that only returns `True`. It is used to check if the minion is up and accepted by the master. Note: this is not an ICMP ping.

Install Napalm in the local minion

We have already introduced Salt States to manage the configuration state. In our case, we decided to use Napalm as the proxy type to interact with the network devices. However, as you may guess, it requires Napalm library to be installed on the minion server hosting the proxy-minions. In our local development environment, this happens in the same server, where Salt Master and Proxy run alonside.

To make installation processes easier, the community has created pre-written Salt States, called *formulas*. To install Napalm, there is already a formula available, napalm-install-formula.

Following the instructions, we clone the repository formula to `/srv/formulas/napalm-install-formula` and, then, update the master configuration file (under `file_roots`) to take this folder into account when looking for Salt States. Remember to restart the process to activate the new configuration.

```
$ cat /etc/salt/master
file_roots:
  base:
      - /srv/salt
      - /srv/formulas/napalm-install-formula
```

Then, you have to create the `sls` files to enforce the installation of this formula into the local minion. You define the state, pointing to the formula, and the pillar information to be used by the state, with the desired Napalm version. These are the all the files related to the Napalm setup:

```
$ cat /srv/salt/top.sls
base:
  ntc:
    - napalm_install

$ cat /srv/pillar/top.sls
base:
  ntc:
    - napalm

$ cat /srv/pillar/napalm.sls
napalm:
  version: 3.4.0
```

In the next section, once Napalm is installed in the minion, we'll look at using execution modules to view grains for one or more devices using the `salt` command.

Start Proxy-Minions

Now, it's time to start our proxy-minions, mapping to each one of the routers, as described in Basic Salt Top File and configure each corresponding pillar with the format from Pillar file using SLS format.

For each router, you start a proxy-minion process, defining its `proxyid`. This `proxyid` should match the key used in the `top.sls` files, to relate to the corresponding pillars and states.

```
$ sudo salt-proxy --proxyid=csr1 -d
$ sudo salt-proxy --proxyid=vmx1 -d
```

Like in Verifying minions are up with the test module, you can verify their reachability with `test.ping`, bub remember that you must approve its connection keys first, as we did in Approve Salt Keys.

Using execution modules

Salt uses execution modules, more commonly referred to simply as modules, in order to retrieve either data that's stored under Salt management or data directly from the device.

NOTE	We only review a handful of modules in this book. For a complete list of modules, please view them at Salt modules documentation.

First, we'll review common modules and associated functions that are used to view both grains and pillar data. We'll show this using the `salt` command, but as you'll see later, you can also leverage these modules directly within SLS files that are used for templates, pillars, and other software artifacts within Salt.

In the first example, we'll simply print the grain called `model` for `csr1` using the `get` function within the `grains` execution module.

```
$ sudo salt csr1 grains.get model
csr1:
    CSR1000V
```

To see the complete list of grains available for a minion, you'd use `grains.items` instead, without passing any arguments.

You can also access specific data from a pillar while using the `salt` CLI command. In the next example, we check to see what the `ntp_peers` value is specifically for `csr1`. Similar to grains, `pillar.get` returns the value of specific pillar data.

```
$ sudo salt csr1 pillar.get ntp_peers
csr1:
    - 10.10.10.1
    - 10.10.10.2
    - 10.10.10.3
```

To retrieve a value from a more complex data structure in a pillar file, you use the `:` delimiter to navigate through key-value hierarchies. Using the previously defined structure from *csr1_pillar.sls* and shown here again, you can print just the BGP ASN for `csr1`:

```
# sample object in a pillar data file: csr1_pillar.sls
'openconfig-bgp':
  bgp:
    global:
      config:
        as: 65001
        router_id: 172.17.17.1
```

```
$ sudo salt csr1 pillar.get openconfig-bgp:bgp:global:config:as
csr1:
    65001
```

TIP

As you start using the `salt` command, be aware of the general syntax:

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

In this case, `target` is used to specify the minions that are going to be automated with the arguments specified.

You can use `salt --help` for added assistance as you continue to use `salt`. We'll continue showing various examples using the `salt` command throughout this section.

Collecting device data using network modules

The previous examples used the grains and pillar modules. Those modules were simply accessing data that was predefined or cached data that was previously collected.

Salt also has over a dozen modules for retrieving feature-specific data from network devices including, but not limited to, NTP configuration, NTP peers, BGP, routes, SNMP, and users. There are even more advanced modules for extracting data from devices and representing it in YAML such that it maps to YANG models.

NOTE

Remember all of this functionality, such as retrieving grains or configuration data from devices, while exposed to Salt, is occurring through the use of the NAPALM Python library.

The following are a few examples using modules on the CLI.

In Collect data using network modules, we retrieve the ARP table from the device with a minion ID of `csr1`.

Example 4. Collect data using network modules

```
$ sudo salt csr1 net.arp
csr1:
```

```
    ----------
    comment:
    out:
        |_
          ----------
          age:
              55.0
          interface:
              GigabitEthernet1
          ip:
              10.0.0.2
          mac:
              52:55:0A:00:00:02
# output trimmed
```

In More examples collecting data from network devices, you can see other examples where specific network functions are being executed within an execution module. In these examples, `net`, `ntp`, and `bgp` are the execution modules and what follows is the function inside the module (e.g., `bgp.neighbors` and `bgp.config`).

Example 5. More examples collecting data from network devices

```
# Retrieve the MAC address table from the device with a minion ID of `vmx1`
$ sudo salt vmx1 net.mac

# Retrieve NTP statistics from the device with a minion ID of `vmx1`
$ sudo salt vmx1 ntp.stats

# Retrieve the active BGP neighbors from the device with a minion ID of `vmx1`
$ sudo salt vmx1 bgp.neighbors

# Retrieve the BGP configuration from the device with a minion ID of `eos-spine1`
$ sudo salt eos-spine1 bgp.config
```

Understanding targeting and compound matching

In the previous examples, we were automating only a single device. Salt offers the ability to use *targeting* to automate more than one device. Targeting can be very simple, but can become as complex as required by the business logic. Let's look at a few examples.

Using the `-L` command line flag, you can explicitly define a list of devices you want to target:

```
$ sudo salt -L csr1,vmx1 net.mac
```

Using what's called *globbing*, you can use expressions such as a wildcard:

```
$ sudo salt 'vmx*' ntp.stats
```

Additionally, you can use grain data with the `-G` flag and target devices based on grains.

```
$ sudo salt -G 'os:junos' bgp.neighbors
```

You can even match devices using their static pillar data using the `-I` flag:

```
$ sudo salt -I 'bgp:as_number:65512' bgp.config
```

For the previous example to work, it would require the BGP configuration data to have the following equivalent YAML structure in a pillar file:

```
bgp:
  as_number: 65512
```

Now that you know how to automate a single minion or a group of minions based on a variety of options, it's worth understanding *compound matching*. Compound matching allows you to perform conditional-like logic, adding more flexibility to target the devices being automated.

In order to retrieve the BGP configuration from minions whose IDs start with vmx, running 18.2x, and have a predefined ASN of 65512 in the pillar file, you can use the following statement:

```
$ sudo salt -C 'vmx* and G@version:18.2* and I@bgp:as_number:65512' bgp.config
```

When using compound matching, you use the `-C` flag and then reference the other flags we previously covered using the flag and the `@` symbol.

As compound matchers can get very complex at times, they can be defined in the master config file under `nodegroups`:

```
nodegroups:
  vmx-18-bgp: 'vmx* and G@os:junos and G@version:18.2* and I@bgp:as_number:65512'
```

You can then access and reference the node group on the CLI:

```
$ sudo salt -N vmx-18-bgp bgp.config
```

Viewing module and function docstrings

When just getting started with Salt, you may not be aware of how a particular function within a module works. In this case, you use the `sys.doc` option in the command being executed. `sys.doc`

without any arguments returns the documentation for all modules.

Optionally, you can specify to return the docstring for a particular module or execution function:

```
$ sudo salt vmx1 sys.doc test.ping
test.ping:

    Used to make sure the minion is up and responding. Not an ICMP ping.

    Returns `True`.

    CLI Example:

        salt '*' test.ping
```

Understanding different output options for modules

```
By default, the structure of a module's output is displayed on the command line, in a
human-readable and colorful format called `nested`:
```

```
$ sudo salt vmx1 ntp.peers
vmx1:
    ----------
    comment:
    out:
        - 1.2.3.4
        - 5.6.7.8
    result:
        True
```

However, the `salt` command permits a significant number of options. One of the most common is `--out`, which returns the output in the format specified. Using this option, we can elect to return the structure in YAML, in JSON, or even as a table:

```
$ sudo salt --out=json vmx1 net.arp
{
    "vmx1": {
        "comment": "",
        "result": true,
        "out": [
            {
                "interface": "fxp0.0",
                "ip": "10.0.0.2",
                "mac": "2C:C2:60:FF:00:5F",
                "age": 1424.0
            },
            {
```

```
                "interface": "em1.0",
                "ip": "128.0.0.16",
                "mac": "2C:C2:60:64:28:01",
                "age": null
            }
        ]
    }
}
```

Next is an example of outputting the data using a table format:

```
$ sudo salt --out=table vmx1 net.arp
vmx1:
----------
    comment:
    ----------
    out:
    ----------
        ------------------------------------------------------
        |  Age  | Interface |     Ip     |        Mac        |
        ------------------------------------------------------
        | 991.0 |   fxp0.0  |  10.0.0.2  | 2C:C2:60:FF:00:5F |
        ------------------------------------------------------
        |  None |   em1.0   | 128.0.0.16 | 2C:C2:60:64:28:01 |
        ------------------------------------------------------
    result:
    ----------
```

There are several output types available and many others can be added. Although the `--out` option is for CLI usage, we are able to take the output in the format displayed on the screen and reuse it as-is in different services, including passing data to an external service.

While data can be passed to an external service, it can also be returned to an external service, which we cover next.

Sending data to external services

As you've seen, data is easily returned and viewed on the command line. However, you may also need to send this data to an external service. Using the `--return` CLI option, you define where to send the data, but you need to specify the name of a *returner*.

The list of available returners is diverse, some of the most usual being Slack, Syslog, Django, Redis, SMS, SMTP, Kafka, MySQL and Postgres.

For example, we can configure the Slack returner by adding in the (proxy) minion or master configuration:

```
slack.channel: Network Automation
slack.api_key: d4735e3a265e16eee03f59718b
```

```
slack.username: salt
```

At this point, we can execute a command with the `--return` flag and the typical stdout is sent to Slack.

Here is an example:

```
$ sudo salt --return slack test.ping
```

NOTE	The output is still displayed on the command line, in the shape we want, while it is also forwarded to the selected service. If you do not want to return anything at all on the CLI, you can use the `--out=quiet` option.

While the outputters make sense only on the command line, returners can be used in other applications. For example, using the Salt scheduler, we can execute a job at specific intervals and its output is then sent to the designated returner. Similarly, when a task is performed as a result of a trigger, we may need to see its result from a monitoring tool.

State modules for network automation

For network automation needs, there are several state modules available, including `net_napalm_yang`, `netconfig`, `netacl`, `netntp`, `netsnmp`, or `netusers`.

One of the most flexible is `netconfig`, which manages and deploys configurations on network devices using arbitrary user-defined templates. Our focus is on `netconfig`.

In this example, we use the `netconfig.managed` state function:

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - debug: true
```

Here is an overview of the various uses of the word state within the previous file:

- The file is a state SLS file.
- `ntp_peers_example` is a state name that we defined.
- `netconfig` is a built-in state module that manages configurations.
- `managed` or `netconfig.managed` is a built-in function that's part of the `netconfig` state module that specifically deploys the configurations onto network devices.

Additionally, remember that *ntp_template.j2* is a template that can leverage SLS functionality, and data inserted into the template could come from pillar data files that are SLS files. The main point is to remember that SLS is a file type.

Next, we'll look at a few workflows that build and deploy network configurations.

Managing Network Configurations with Salt

We've seen how flexible SLS files are and what can be done with the `salt` command, but another unique capability of Salt is that any function available on the CLI can also be executed inside SLS files, including templates.

We're now going to walk through the process of auto-generating configurations through the use of templates.

Accessing data within templates

There are also a number of built-in *identifiers* (almost like variables) available, such as `grains`, `pillar`, `opts` (the dictionary of configuration options) or `env` (provides the environment variables). These can also be used directly inside a template similar to how they're used on the CLI (e.g., `grains.get os`). This adds value as you start templating out network configurations, as you'll see in the next few examples.

NOTE

Salt offers a nice way to manage sensitive data and avoid repeating the same configuration in multiple places. It is called the SDB interface, which is designed to store and retrieve data that, unlike pillars and grains, is not necessarily minion-specific, through small database queries (hence the name SDB) using a compact URI *sdb://<profile>/<args>*. There are a number of SDB modules available, including: SQLite3, CouchDB, Consul, Keyring, Memcached, REST API calls, Hashicorp Vault, Redis or environment variables.

Execution functions can be accessed with the `salt` reserved keyword. While on the CLI we just execute `ntp.peers`; inside the template we only need to prepend `salt`:

```
{%- set configured_peers = salt.ntp.peers()['out'] -%}
```

`configured_peers` is a variable created in the template via the `set` statement in Jinja. `configured_peers` is a Jinja variable and when this template is executed, it'll be assigned the list of active NTP peers configured on the device through the `ntp` execution module.

In Salt, as in many other tools, it's good practice to move the complexity of the Jinja templates into the actual functions such that the data or task can be eventually reused in other applications. This provides major benefits: making templates more readable, opening the gate to reusability, and providing a great way to reintroduce data into the system.

NOTE

Custom execution modules take just a short time to write in Python and are automatically distributed to minions (or proxy minions). They can then be used within templates, other SLS files, or from the command line.

When using the `grains`, `pillar`, and `opts` keywords within a template, you can define high-level business logic and design templates in a vendor-agnostic manner, in such a way that the same template can be executed against different platforms, and it is intelligent enough to identify what configuration changes to load.

Creating Jinja network configuration templates

For example, you can define the Salt template to generate the configuration for the NTP peers, using the input data from the pillar defined earlier, in YAML SLS file, with a Jinja template that looks like NTP Jinja configuration template.

Example 6. NTP Jinja configuration template

```
{%- if grains.os == 'junos' %}
system {
 replace:
 ntp {
   {%- for peer in pillar.ntp_peers %}
   peer {{ peer }};
   {%- endfor %}
 }
}
{%- elif grains.vendor | lower == 'cisco' %}
no ntp
  {%- for peer in pillar.ntp_peers %}
ntp peer {{ peer }}
  {%- endfor %}
{%- endif %}
```

The unique piece here is the use of the `pillar` keyword directly in the template. This allows you to access data defined in the pillar files.

If your goal was to ensure that a specific feature is configured exactly as desired in a declarative manner (with no extra peers still on the device), you can perform a replace operation on the device. On Junos, you do this using the `replace` keyword, while with other more traditional operating systems, command negations ("no" commands) are required.

NOTE The replace keyword here for Junos maps back to the NETCONF replace operation that we covered in [apis] and allows you to *replace* a full hierarchy within a configuration.

In this example, we saved the template as *ntp_template.j2* within the */srv/templates* directory, included in the `file_roots` list, in the master configuration file.

We can then reference this template as `salt://ntp_template.j2` when using it from the command line or from within Salt state files.

At this point, we've simply built out the Jinja template—we haven't yet rendered it with data, or created a configuration file.

To highlight what's possible using of the `salt` directive inside the template, we're able to determine the NTP peers to be added or removed based on retrieving in real time the existing peers using the statement `salt.ntp.peers`.

In NTP Jinja configuration template, removing uncompliant peers, the template creates the configuration for both IOS and Junos for configuring NTP peers. This template has the logic required to ensure only the peers defined in the pillar end up configured on the device, meaning any unwanted peers will be purged from the device (when the configuration is deployed).

Example 7. NTP Jinja configuration template, removing uncompliant peers

```
{%- set configured_peers = salt.ntp.peers()['out'] -%}
{%- set add_peers = pillar.ntp_peers | difference(configured_peers) -%}
{%- set rem_peers = configured_peers | difference(pillar.ntp_peers) -%}
{%- if grains.os == 'junos' -%}
 {%- for peer in rem_peers -%}
delete system ntp peer {{ peer }}
 {% endfor -%}
 {%- for peer in add_peers -%}
set system ntp peer {{ peer }}
 {% endfor -%}
{%- elif grains.vendor | lower == 'cisco' %}
 {%- for peer in rem_peers -%}
no ntp peer {{ peer }}
 {% endfor -%}
 {%- for peer in add_peers -%}
ntp peer {{ peer }}
 {% endfor -%}
{%- endif -%}
```

For devices such as Juniper that provide support for partial configuration replace capabilities, this is quite a nice solution. For others, it could seem tedious due to the logic required to determine which "no" commands are needed to purge the un-wanted peers. However, this is the best way to handle those scenarios where there isn't a native way for *partial* configuration replace operations.

Deploying network configurations with netconfig

Next, we need to define a state that can be executed to insert the data from the NTP pillar(s) into the NTP template to generate the required commands that'll send commands to the devices.

Here is where we'll use the `netconf.managed` state function. This renders the desired configuration and deploys the commands to the network device.

```
ntp_peers_example:
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - debug: true
```

This SLS state file was saved under one of the `file_roots` paths (e.g., */srv/salt/*) as *ntp.sls*.

To execute this SLS state file, we need to call the execution function `state.apply` or `state.sls` with

the name of the state file as an argument:

```
$ sudo salt vmx1 state.apply ntp
vmx1:
----------
          ID: ntp_peers_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 04:25:09.689908
    Duration: 1074.807 ms
     Changes:
              ----------
              diff:
                  [edit system ntp]
                  +    peer 10.10.10.1;
                  +    peer 10.10.10.2;
                  +    peer 10.10.10.3;
                  -    peer 172.16.0.1;
                  -    peer 172.16.0.2;
              loaded_config:

                  system {
                   replace:
                   ntp {
                     peer 10.10.10.1;
                     peer 10.10.10.2;
                     peer 10.10.10.3;
                   }
                  }

Summary for vmx1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   1.075 s
```

Note that in a single execution, the commands were generated in memory and deployed to a network device. This example did not create a config file first.

The format of the output displayed in the previous example on the CLI is called *highstate*, but the object returned is still a Python object (we can verify using `--out=raw`); hence, it can be reused and define complex workflows.

Note the `loaded_config` key returned as we specified `debug: true` in the state SLS, having the configuration generated as required by the business logic.

The execution time is quite fast here given all the steps performed: it retrieved the current

configuration, determined the difference, generated the configuration (all within the template), and subsequently loaded the commands onto the device, generated a diff, and then committed the configuration to memory on the device, all within 1.075 seconds.

We can also run the same exact state file, `ntp.sls`, against `csr1`. The state will process the same template, which knows from the grains that `csr1` is a Cisco IOS device and will generate the appropriate configuration to be loaded on the device:

```
$ sudo salt csr1 state.apply ntp
csr1:
----------
          ID: ntp_peers_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration changed!
     Started: 04:27:01.108327
    Duration: 3899.933 ms
     Changes:
              ----------
              diff:
                  -no ntp
                  +ntp peer 10.10.10.1
                  +ntp peer 10.10.10.2
                  +ntp peer 10.10.10.3
              loaded_config:

                  no ntp
                  ntp peer 10.10.10.1
                  ntp peer 10.10.10.2
                  ntp peer 10.10.10.3

Summary for csr1
------------
Succeeded: 1 (changed=1)
Failed:    0
------------
Total states run:     1
Total run time:   3.900 s
```

Using State dependencies

Another important feature you can leverage within state files is creating state dependencies.

When you need to apply several states that depend on each other, you will find *state requisites* very helpful. For example, if you need the `ntp_peers_example` state to be executed only if another state (such as `bgp_neighbors_example`) has been successfully executed, you only need to add two more lines:

```
ntp_peers_example:
```

```
  netconfig.managed:
    - template_name: salt://ntp_template.j2
    - require:
      - bgp_neighbors_example
```

Generating network configuration files

We also have the ability to decouple the configuration generation and deployment into separate steps—similar to what we showed in the Ansible section too. This is often helpful if you want to version or view the commands before you try doing any deployments.

In order to accomplish this, we'll use the `file.managed` state function. As arguments, we'll specify the template type and location of the template. Once the data is rendered in the template, we'll save it as *ntp_generated.conf* using the `name` key:

```
build_config:
  file.managed:
    - name: /home/ntc/ntp_generated.conf
    - source: salt://ntp_template.j2
    - template: jinja
```

If we saved this as *build-ntp.sls*, in */srv/salt*, we could just build the configuration file using a defined state as follows:

```
$ sudo salt vmx1 state.apply build-ntp
# output omitted

$ cat /home/ntc/ntp_generated.conf

system {
 replace:
 ntp {
   peer 10.10.10.1;
   peer 10.10.10.2;
   peer 10.10.10.3;
 }
}
```

Generating and deploying network configurations from files

Another option if you did want to build the config and deploy within a single workflow, but still wanted to generate a config file on the server first, would be to have both of these states in the same SLS file, as we do in Extending the State to deploy configuration file.

Example 8. Extending the State to deploy configuration file

```
generate_config:
```

```
    file.managed:
      - name: /home/ntc/ntp_generated.conf
      - source: salt://ntp_template.j2
      - template: jinja
  ntp_peers_example:
    netconfig.managed:
      - template_name: /home/ntc/ntp_generated.conf
      - require:
        - file: /home/ntc/ntp_generated.conf
```

If we saved Extending the State to deploy configuration file as *ntp-build-deploy.sls*, and executed it, we'd see the following output:

```
$ sudo salt vmx1 state.sls ntp-build-deploy
vmx1:
----------
          ID: generate_config
    Function: file.managed
        Name: /home/ntc/ntp_generated.conf
      Result: True
     Comment: File /home/ntc/ntp_generated.conf is in the correct state
     Started: 04:40:23.118581
    Duration: 26.408 ms
     Changes:
              ----------
              diff:
                  New file
              mode:
                  0644
----------
          ID: ntp_peers_example
    Function: netconfig.managed
      Result: True
     Comment: Already configured.
     Started: 04:40:23.145850
    Duration: 543.863 ms
     Changes:
              ----------
              diff:
                  [edit system ntp]
                  +    peer 10.10.10.1;
                  +    peer 10.10.10.2;
                  +    peer 10.10.10.3;
                  -    peer 172.16.0.1;
                  -    peer 172.16.0.2;

Summary for vmx1
------------
Succeeded: 2 (changed=2)
```

```
Failed:    0
------------
Total states run:     2
Total run time:   4.331 s
```

Parameterizing configuration filenames

In both previous examples when config files were generated, the filename used was */home/ntc/ntp_generated.conf*. This is not scalable, as the filename is static. To avoid hardcoding the filename, but generate the name depending on the device or minion ID, we can specify this using the `id` field from the `opts` SLS special variable:

```
generate_config:
  file.managed:
    - name: /home/ntc/{{ opts.id }}_ntp_generated.conf
    - source: salt://ntp_template.j2
    - template: jinja
```

The state above generates a file called *home/ntc/vmx1_ntp_generate.conf* for the `vmx1` minion, *home/ntc/csr1_ntp_generate.conf* for `csr1`, and so on.

Scheduling state execution

In Salt, it is very important to distinguish between jobs executed on the master, and jobs executed on the minion. While the minions run execution functions, the master executes runners, covered in the next paragraphs. This is significantly important when we are scheduling jobs: if we want to schedule an execution function, we add the instructions in the (proxy) minion configuration file, while we schedule a runner by adding the options in the master configuration file. In both cases, the syntax is the same. For example, if we need to schedule the preceding state to be applied every Monday at 11 a.m., we'd only need the following lines in the (proxy) minion configuration file:

```
schedule:
  ntp_state_weekly:
    function: state.sls
    args:
      - ntp
    kwargs:
      test: true
    ret: smtp
    when:
      - Monday 11:00am
```

Under `kwargs` we configured `test: true`, which means the state is going to execute a dry run, but it will return the configuration difference. Moreover, we have subtly introduced another feature with the field `ret: smtp`. This tells Salt to take the output of the state and forward it to the *returner* called `smtp`. This returner takes the data from the output of the state and sends an email with the configuration diff.

Generating reports

Generating reports is even more useful when they are also consumed by a process or a human. For this, the *returners* are very handy and easy to use. In the previous example, the NTP state is executed, and then its output is processed via the SMTP returner—this is basically sending the execution report as email.

To send the email with the content as-is, we only need to configure the following options on the minion:

```
smtp.from: ping@mirceaulinic.net
smtp.to: jason@networktocode.com
smtp.host: localhost
smtp.subject: NTP state report
smtp.template: salt://ntp_state_report.j2
```

Where *ntp_state_report.j2* is a template found under the */srv/templates* directory, that customizes the subject body as this:

```
NTP consistency check
---------------------

Running on {{ id }}, which is a {{ grains.vendor }} {{ grains.model }} device,
running {{ grains.os }} {{ grains.version }}:

{{ result }}
```

When the scheduler is executed, it will send an email with the following body:

```
NTP consistency check
---------------------

Running on vmx1, which is a Juniper VMX device,
running junos 18.2R1.9:

vmx1:
----------
          ID: ntp_peers_example
    Function: netconfig.managed
      Result: True
     Comment: Configuration discarded.

              Loaded config:


              system {
               replace:
               ntp {
```

```
                  peer 10.10.10.1;
                  peer 10.10.10.2;
                  peer 10.10.10.3;
                }
               }
     Started: 09:15:30.808802
    Duration: 563.741 ms
     Changes:

Summary for vmx1
------------
Succeeded: 1
Failed:    0
------------
Total states run:     1
Total run time: 563.741 ms
```

With this setup, we can ensure that Salt periodically executes the NTP state in test mode, then generates and sends an email with the report.

From the CLI, we could achieve this by manually executing:

```
$ sudo salt vmx1 state.sls ntp test=True --return smtp
```

In a very similar way, we can set this up to send reports with the result from multiple devices at a time, using a runner instead of an execution function. While the execution function is run by the minion process, a runner function is executed by the master process, which gives visibility over the entire network. In Python language, the result is a dictionary whose keys are the minion IDs matched, while the values are the actual result of each device.

Available from both CLI and scheduled process, *returners* are a very powerful tool for post-processing and data transformation. Later, we will see they can be reused when reacting to events, or to monitor the entire Salt activity.

Executing Salt Functions Remotely

We've covered quite a bit thus far on Salt, but one of the most important components to understand is the architecture employed by Salt for network devices. However, as you've learned a lot in this section, be aware that Salt offers two primary ways you can interact with Salt, and execute any command or tasks remotely from another machine.

Using the Salt API

This RESTful API is included with Salt, using the `salt-api` daemon, and can be used to perform any operation you can when using the `salt` command-line programs within the Linux shell.

A core feature of the RESTful API is that it allows you to pick one of three web servers supported *out of the box*. They include CherryPy, uWSGI, or Tornado.

The following is how you'd enable CherryPy by editing the master configuration file:

```
rest_cherrypy:
  port: 8001
  ssl_crt: /etc/nginx/ssl/my_certificate.pem
  ssl_key: /etc/nginx/ssl/my_key.key
```

This configures the server to listen on port `8001` and use the certificate and the key for secured requests.

Afterward, you can start executing Salt functions remotely through the use of custom scripts, Postman, or cURL. The following example shows the use of cURL to retrieve the ARP table for `vmx1`.

```
$ curl -sSk https://salt-master-ns-or-ip:8001/run \
    -H 'Content-type: application/json' \
    -d '[{
        "client": "local",
        "tgt": "vmx1",
        "fun": "net.arp",
        "username": "ntc",
        "password": "ntc123",
        "eauth": "pam"
    }]'
```

For configuration-related requests, the function, `fun`, is then replaced by `state.sls` or `state.apply`, and the name of the state is specified in the `args` field:

```
$ curl -sSk https://salt-master-ns-or-ip:8001/run \
    -H 'Content-type: application/json' \
    -d '[{
        "client": "local",
        "tgt": "vmx1",
        "fun": "state.sls",
        "args": ["ntp"],
        "username": "ntc",
        "password": "ntc123",
        "eauth": "pam"
    }]'
```

This example, when executed, would run the NTP state that was defined earlier in the chapter.

NOTE

Another option for interacting with the Salt API is the `salt-pepper` Python library, in which you can execute CLI commands from a personal machine, directly on the Salt master server. Pepper comes with a command-line binary (`pepper`) that can be used exactly like the master `salt` command. For example, we can execute NTP state from *our* machine and it'll run directly on the master:

```
$ pepper 'vmx1' state.sls ntp
```

Diving into Salt's Event-Driven Infrastructure

Salt is built around an event bus, which is an open system, based on ZeroMQ, used to notify Salt and other systems about operations. ZeroMQ is a cross-platform high-performance asynchronous messaging toolkit that focuses on handling tasks very efficiently, without additional overheads.

To watch the events in real time, we execute the following command on the master:

```
$ sudo salt-run state.event pretty=True
```

If we looked at using a module with the `salt` command, we'd see that there are three individual events that take place when the command is executed. For example, a command such as `$ sudo salt -G os:ios test.ping` executes the following three events.

First, there is a Job ID, a way to uniquely reference any given event that is mapped to target minions. The event is described, and shown as follows:

```
20220521092719071436    {
    "_stamp": "2022-05-21T09:27:19.071694",
    "minions": [
        "csr1"
    ]
}
```

Next, the job is executed on the appropriate minions. This event is described and shown as follows:

```
salt/job/20220521092719071436/new       {
    "_stamp": "2022-05-21T09:27:19.072185",
    "arg": [],
    "fun": "test.ping",
    "jid": "20220521092719071436",
    "minions": [
        "csr1"
    ],
    "missing": [],
    "tgt": "os:ios",
    "tgt_type": "grain",
    "user": "root"
}
```

The final event for this command is the response and status for each minion that was in the target scope.

```
salt/job/20220521092719071436/ret/csr1  {
    "_stamp": "2022-05-21T09:27:19.130946",
    "cmd": "_return",
    "fun": "test.ping",
    "fun_args": [],
    "id": "csr1",
    "jid": "20220521092719071436",
    "retcode": 0,
    "return": true,
    "success": true
}
```

Note that in the final event there is a unique tag pattern for each minion. As you can see, the preceding example is showing the tag of `salt/job/20220521092719071436/ret/csr1`.

Next, we'll take a look at several items that have very specific meaning within Salt for event-driven network automation.

Watching external processes with beacons

In Salt, beacons are used to watch external processes that are not related to Salt and to import and return events onto the Salt bus.

For example, the `inotify` beacon is used to monitor when a file is changed. If we want to monitor when the *ntp_peers.sls* file is updated, the lines in Beacons configuration in minion configuration need to be added in the (proxy) minion configuration:

Example 9. Beacons configuration in minion configuration

```
beacons:
  inotify:
    - files:
        /srv/pillar/ntp_peers.sls:
          mask:
            - modify
      disable_during_state_run: True
```

NOTE The inotify beacon only works on OSes that have inotify kernel support and requires Pyinotify installed on the minion.

This instructs Salt to start monitoring the file */srv/pillar/ntp_peers.sls* and push events onto the bus. Modifying the contents, we will see events with the following structure:

```
salt/beacon/vmx1/inotify/srv/pillar/ntp_peers.sls {
    "_stamp": "2022-05-21T09:50:18.330556",
    "change": "IN_IGNORED",
```

```
    "id": "vmx1",
    "path": "/srvpillar/ntp_peers.sls"
}
```

This may be valuable for you to track data as it changes in the system. Since you can use modules within pillars (as an example), the data is dynamic, often getting pulled from the devices in real time. You'd be able to see this data change in real time using beacons.

Forwarding events with engines

Engines are another subsystem interfacing with the event bus. While beacons only listen to external processes and transform them into Salt events, engines can be bidirectional. Although their main scope is the forwarding of events, there are also engines able to inject messages on the bus. And that is the main difference between beacons and engines: *beacons* poll the service at specific intervals (default: 1 second), while the *engines* can fire and forward events on immediate occurrence.

A very good application could be logging Salt events to a syslog server such as Logstash, using the *http-logstash* engine. This would be defined on the master like so:

```
engines:
  - http_logstash:
      url: https://logstash.elastic.co/salt
      tags:
        - salt/job/*/new
        - salt/job/*/ret/*
```

The YAML configuration on the master configures the master to send events to Logstash. However, it's configured to send the events only matching the tags `salt/job//new` and `salt/job//ret/*`. For reference, if `tags` is not configured or empty, the engine would forward all events.

Listening to the salt bus with reactors

The reactor system listens to the event bus and executes an action when an event occurs. The reactors are configured on the master, the global syntax being:

```
reactor:
  # <tag match> describes the pattern to be matched against the event tag
  - 'salt/beacon/*/inotify//srv/pillar/ntp_peers.sls':
    # <list of SLS descriptors to execute>
    - salt://run_ntp_state_on_pillar_update.sls
```

This example instructs Salt to execute the *run_ntp_state_on_pillar_update.sls* data file when the inotify beacon injects the corresponding event on the bus, on file update.

The reactor SLS, *run_ntp_state_on_pillar_update.sls*, can have any structure you'd like. For our example, we're using the configuration from Reactor SLS example.

Example 10. Reactor SLS example

```
run_ntp_state:
  local.state.sls:
    - tgt: {{ data['id'] }}
    - arg:
      - ntp
    - ret: mongo
```

This executes the execution function `state.sls` with the argument `ntp` against the minion whose ID is extracted from the event body, under the field `id`.

The following events are what you'd see on the event bus.

```
salt/beacon/vmx1/inotify//srv/pillar/ntp_peers.sls {
    "_stamp": "2022-05-21T10:57:24.651644",
    "change": "IN_IGNORED",
    "id": "vmx1",
    "path": "/srv/pillar/ntp_peers.sls"
}
20220521105724736722  {
    "_stamp": "2022-05-21T10:57:24.737525",
    "minions": [
        "vmx1"
    ]
}
salt/job/20220521105724736722/new {
    "_stamp": "2022-05-21T10:57:24.737804",
    "arg": [
        "ntp"
    ],
    "fun": "state.sls",
    "jid": "20220521105724736722",
    "minions": [
        "vmx1"
    ],
    "tgt": "vmx1",
    "tgt_type": "glob",
    "user": "sudo_admin"
}
# followed also by the result of the state execution, omitted here due to size
# limits further output omitted
```

First, the pillar file, *ntp_peers.sls*, is changed, and it is fired by the `inotify` beacon; then, the reactor kicks in and creates a new job and identifies the minions, then sends the task to the minions (only `vmx1` in this case).

Note the `ret` field in Reactor SLS example: the mongo *returner* is invoked, which means Salt will forward the state results into MongoDB. This statement is optional and not required.

Suppose we have the pillar files maintained in Git. Configuring the local clone to track the remote origin server, the previous example is an excellent orchestration example: a pull request merged triggers automatic configuration deployment of the NTP peers for the entire network, without any manual work. Note also the difference between configuration management only and event-driven automation: beacon, reactor setup, and SLS—15 lines in total, and the results are sent into a structured database service. Moreover, we maintain vendor-agnostic entities of data, not pseudo-formatted files.

> The reactor has the limitation that we are able to trigger actions only to individual events. Thorium is the next step: it is a complex system that can define business logic based on aggregate data and multiple events.
>
> **NOTE**
>
> You can do quite a lot with Salt without event-driven network automation. Our recommendation is to first start using the `salt` command and start creating relevant SLS data files in the form of pillars and templates. Once you've mastered the basics, you'll be ready to start exploring the event-driven capabilities of Salt and have a much greater grasp on what it offers.

Extending Salt

As we hope we have emphasized in this section, every Salt component is pluggable. The extension modules can be placed in a directory, with subdirectories for each of Salt's module types, such as *modules*, *states*, *returners*, *output*, and *runners*. The naming convention for subdirectories is to prepend a `_` to the module type, so execution modules are defined under `_modules`, runners under `_runners`, and output under `_output`. The parent directory can be specified with the option `extension_modules`, or `module_dirs`—which accepts a list of paths. Alternatively, we can also include it as one of the `file_roots` paths.

For example, a new execution module called *example.py* can be placed under */srv/salt/_modules*:

```
def test():
    return {
      'network_programming_with_salt': True
    }
```

To make Salt aware of the new module, we need to resynchronize the modules using the `saltutil.sync_all` execution function:

```
$ sudo salt vmx1 saltutil.sync_all
vmx1:
    ----------
    beacons:
    clouds:
    engines:
    executors:
```

```
    grains:
    log_handlers:
    matchers:
    modules:
        - modules.example
    output:
    proxymodules:
    renderers:
    returners:
    sdb:
    serializers:
    states:
    thorium:
    utils:
```

Here you can see `modules.example`, which is telling us that the new execution module `example` has been synchronized, and is available to be invoked:

```
$ sudo salt vmx1 example.test
vmx1:
    ----------
    network_programming_with_salt:
        True
```

Remember, modules can be used from the CLI or directly within SLS files, such as templates:

```
{%- set successful = salt.example.test()['network_programming_with_salt'] -%}
```

Salt Summary

In this section, we covered some of the most important topics to be aware of when just getting started with Salt for network automation. One of the greatest attributes of Salt that we covered was the use of the SLS file. Remember, you have complete control of how to write SLS data files, from using Jinja and YAML (as the defaults), to using Mako and HJSON, to adding in a new or custom templating language or even data format. This allows you to maintain the use of Salt and extend its capabilities according to the environmental requirements, without depending on the official codebase. Another major benefit of Salt is the use of proxy minions. With Salt, you have a natively built-in ability to distribute load between proxy minions that make it a great choice for large and distributed networks.

Event-Driven Network Automation with StackStorm

StackStorm is an open source software project for providing flexible event-driven automation. It is often lumped together with some of the other tools in this chapter, but StackStorm actually wasn't built to replace existing configuration management tools. For instance, many popular workflows in StackStorm actually leverage tools like Ansible for performing configuration management tasks.

The best way to think about StackStorm is that it fits in the sweet spot between configuration management (or general automation) and monitoring. It aims to provide a set of primitives for allowing the user to describe the tasks that should take place in response to certain events. For this reason, StackStorm can be thought of as the IFTTT (if-this-then-that) of IT infrastructure.

In many areas of IT, we usually respond to problems or outages manually. One very popular use case for StackStorm is the concept of auto-remediation, which is the idea of attempting to resolve issues without any human intervention. Naturally, if you're just getting started, there's not a magical button you can press to automatically fix problems. However, auto-remediation is the idea that after you've fixed a problem manually, you should commit that same procedure as some sort of automated workflow, reducing the number of manual tasks over time. StackStorm aims to allow you to do just that.

NOTE

The design of StackStorm is meant to work well with infrastructure-as-code practices. These practices take the approach that your infrastructure can be described using text files (such as YAML files and Jinja templates we've used in many other places in this book), and that these files can and should be managed in the same way that software developers manage source code for applications.

In StackStorm, nearly everything is described using YAML files. As we go through the following examples, keep this paradigm in mind, and remember that it's always a good idea to use version control and automated testing for the files we use to manage our infrastructure.

In the following section, we'll explore some of the basic StackStorm concepts you'll need to understand in order to get started. You'll find that many of the concepts within StackStorm embrace infrastructure-as-code—everything in StackStorm is defined using plain-text files. So you can (and should) manage everything in StackStorm using the concepts we learned in [sourcecontrol].

StackStorm Concepts

There are a number of concepts we'll need to familiarize ourselves with in order to use StackStorm, as shown in StackStorm concepts. Together, they form a set of tools that make event-driven automation possible.

First, *actions* are conceptually closest to the functionality offered by some of the other tools discussed in this chapter. They're where the actual work gets done. They're logically distinct bits of code that perform tasks like making API calls and executing scripts. They're the atomic building blocks of the "automation" part of event-driven automation.

From there, we want to talk about *workflows*. Workflows are a way to stitch actions together coherently to accomplish your business logic. You may start a workflow by running a few actions to gather some additional data from your infrastructure, then use that data to make decisions about which actions you should run in order to fix a problem. Workflows are available in StackStorm in two forms. The simplest format is ActionChains, which are a rudimentary way to "chain" actions together. The second format is Mistral, which is an OpenStack project with its own workflow definition that comes bundled with StackStorm, and it is much more flexible.

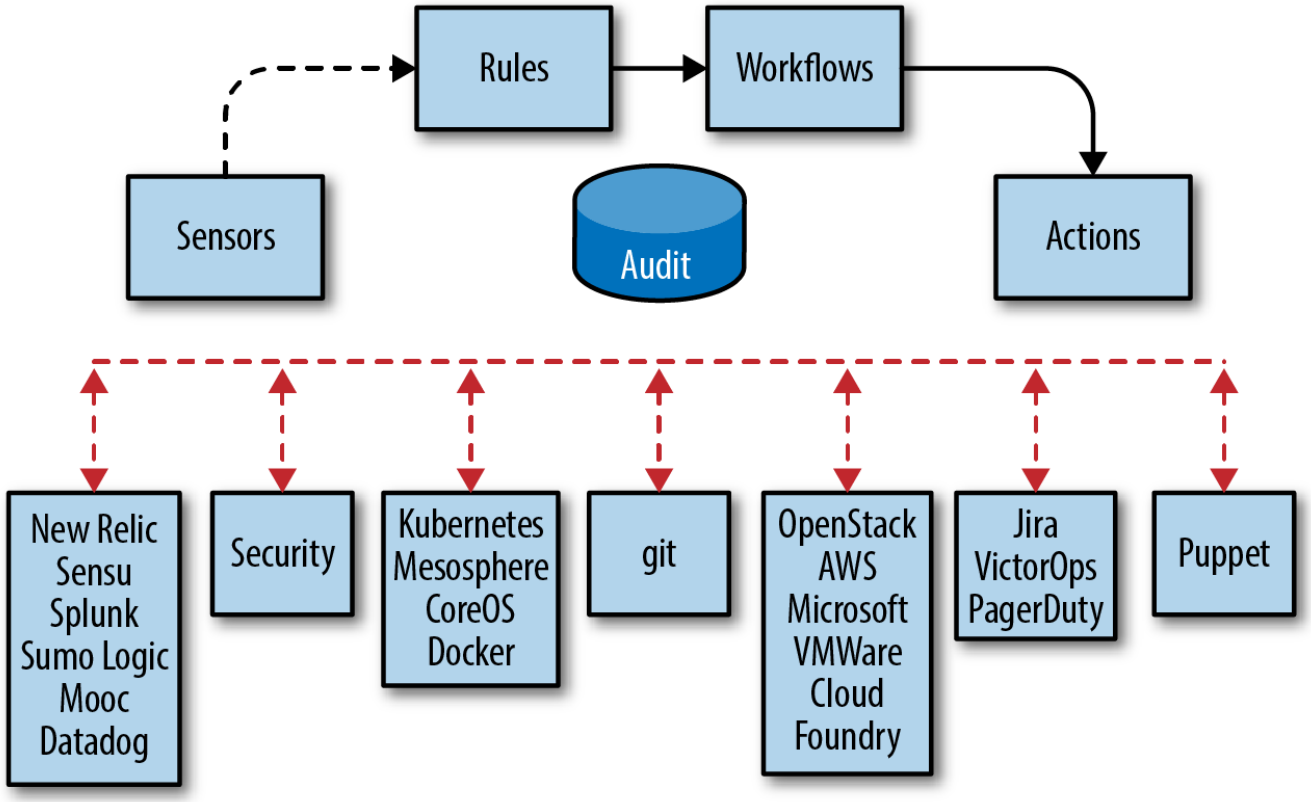

Figure 3. StackStorm concepts

Now that we have actions and workflows to do the work on our behalf, we'll want to bring these concepts into the world of "event-driven" automation. For this, we use *sensors* and *triggers*. You can think of sensors as little bits of Python code whose sole purpose is to gather data about your infrastructure. Note that these aren't agents that you deploy to endpoints like servers or network devices; rather, sensors run within StackStorm itself, and usually connect programmatically to external entities like monitoring or management systems, or in some cases, discrete nodes like virtual machines or network devices, in order to gather the information they need. However, this data is meaningless unless we have a way of recognizing what an "event" is. For that, sensors can also define triggers, which fire when something meaningful has occurred as indicated by the data being brought in by the sensor. For instance, the `napalm.LLDPNeighborDecrease` trigger notifies StackStorm when a given network device experiences a reduction in LLDP neighbors.

The crux of event-driven automation with StackStorm comes in the form of *rules*. Rules are the StackStorm equivalent of "if-this-then-that"—they're a way to connect incoming triggers to actions or workflows that respond to events. You may want to execute a workflow that pushes a new configuration to a network device when the `napalm.LLDPNeighborDecrease` trigger fires; this automated response would be defined in a rule.

Finally, it's worth mentioning that all of these concepts are distributed in StackStorm via *packs*. Packs are the atomic unit of distribution for all manner of extensibility in StackStorm—sensors, rules, workflows, and actions are all defined with text files, and those files are placed in a pack so they can be referenced. For instance, the `napalm.LLDPNeighborDecrease` sensor is in the `napalm` pack. Not all packs are installed by default, but there are CLI commands for easily installing new packs from the StackStorm Exchange. For instance, to download the `napalm` pack, you would simply run:

```
st2 pack install napalm
```

This will make all of the actions, workflows, sensors, or rules present in the `napalm` pack available to you in your instance of StackStorm.

Now that we have the basic concepts in mind, let's talk about how StackStorm works under the covers.

StackStorm Architecture

StackStorm is not really one component, but several microservices that work together to make event-driven automation possible (see StackStorm components). The distributed nature of StackStorm was designed so that each component could scale independently as needs change. This also allows each function to be more resilient in a failure scenario.

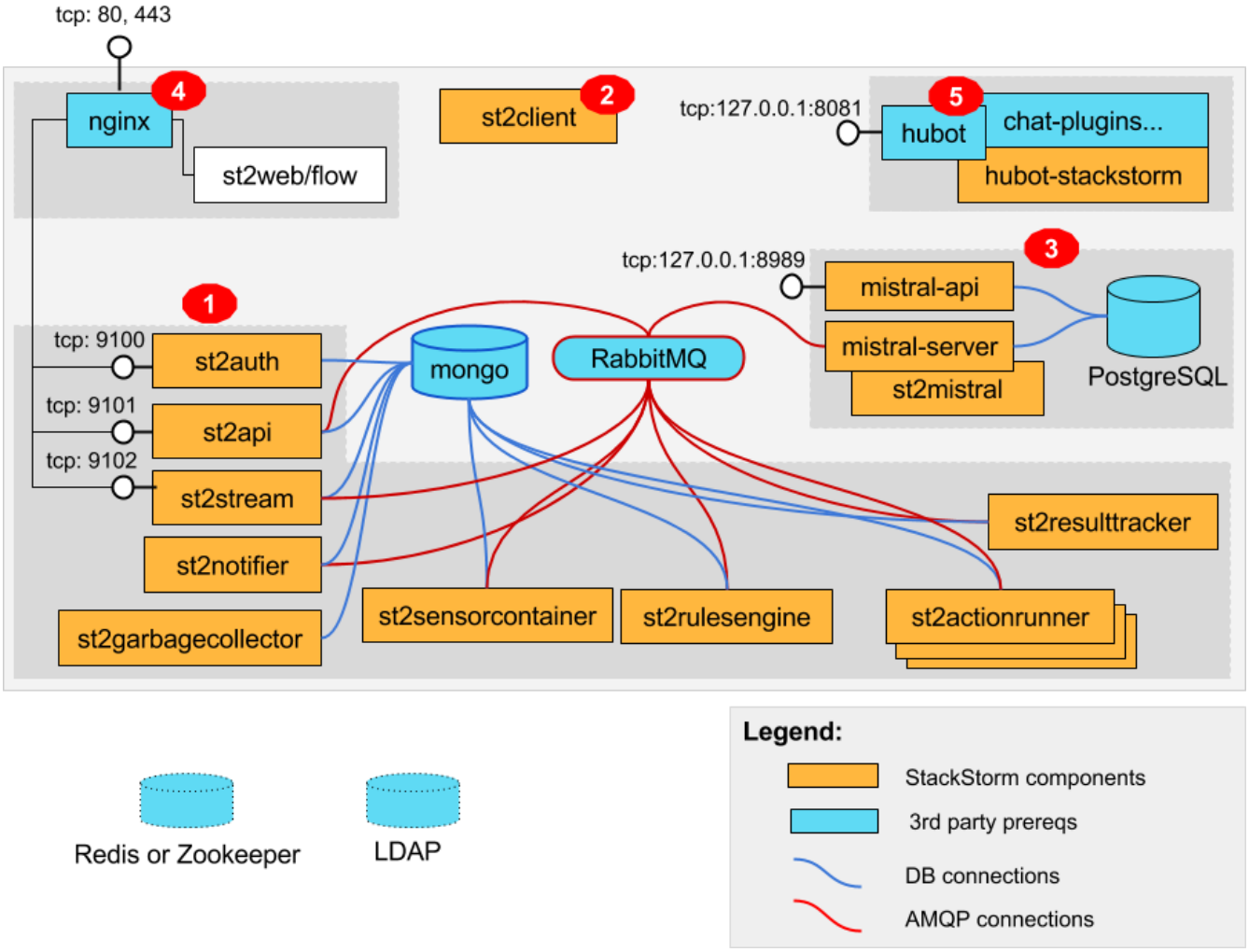

Figure 4. StackStorm components

For instance, if you know you will require a lot of horsepower for your workflows, but won't be watching for that many events in order to kick them off, you may want to spin up some additional `st2actionrunner` components to handle the load. If you wanted to handle a large number of events you would scale the `st2sensorcontainer` component in the same way.

Note that none of these components are "agents"; this does not mean you need to install any of

these components on the rest of your infrastructure. Everything you see in StackStorm components is still self-contained within the servers or instances running StackStorm.

One component in particular is worth talking about—`st2web`. This is just a handy name for the web UI that comes with StackStorm (see StackStorm web UI).

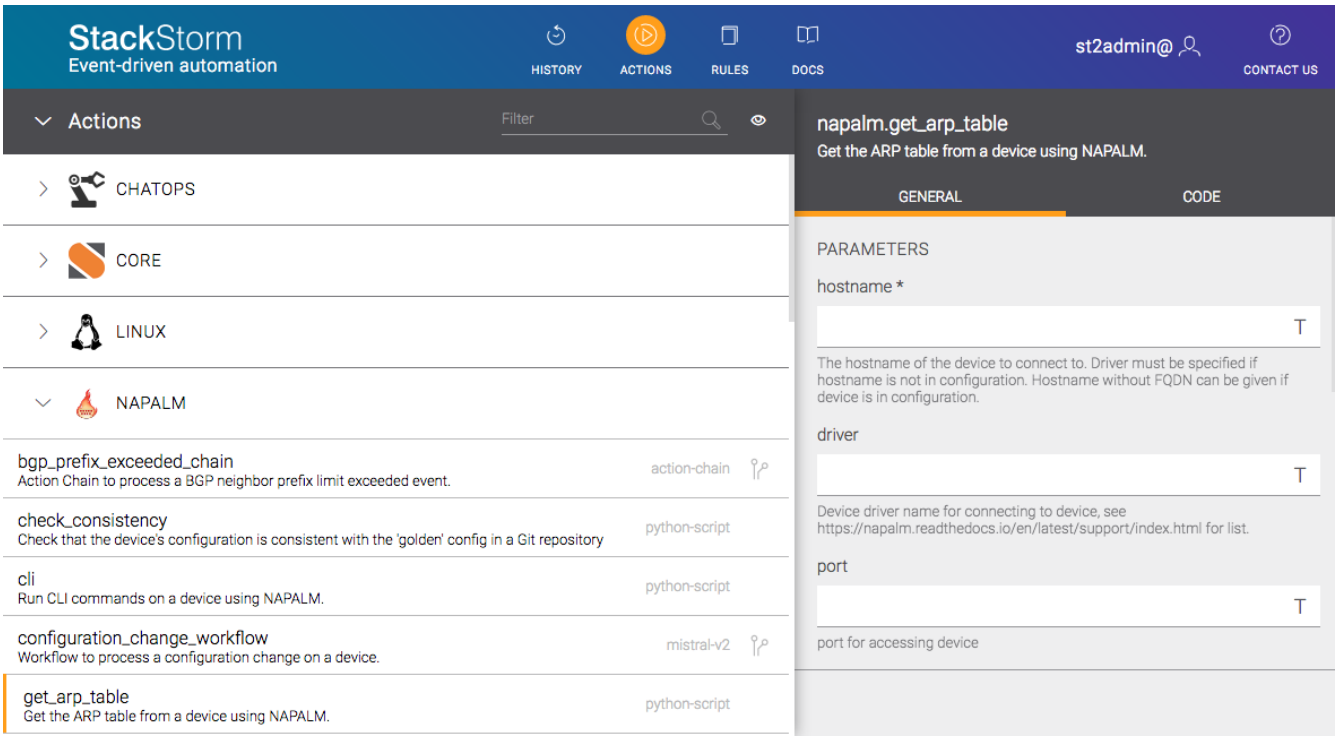

Figure 5. StackStorm web UI

You can use this web UI to do almost anything in StackStorm. So, if you're just getting used to the idea of working in the bash shell, this may be a more friendly option. The examples we'll use in this chapter will use the StackStorm CLI, as it is a bit clearer for our purposes.

Actions and Workflows

Now that we have the concepts and high-level architecture down, we should move into some practical examples of everything we've learned thus far.

NOTE

StackStorm is very powerful, and as a result, there's a lot to discuss. Rather than cram this section full of examples for everything you could possibly want to know, we'll give an overview of the important details. For everything else, bookmark the StackStorm documentation and refer to that for much more detailed explanations, code snippets, and more.

In addition, StackStorm has a very active Slack community (sign up for free at https://stackstorm.com/?#community), and it's the best place to get questions or concerns answered.

You may wish to follow along—and to that end, you should check out the "st2vagrant" repository. There, you'll find a Vagrantfile and some scripts for easily spinning up a single-instance deployment of StackStorm.

NOTE	Vagrant is described in detail in the next chapter, [cicd].

Let's start with something simple, like running a single `echo` command to print "Hello World." For this we'll use the `core.local` action, which allows us to execute any command we'd otherwise run directly in bash.

NOTE	A variety of backends can be leveraged within actions to actually perform the work. For instance, `core.local` happens to simply pass a value to the bash shell, but an action may use a Python or bash script to drive more complicated logic.

StackStorm comes with its own command-line interface: the `st2` command. One subcommand available to use is the `st2 run` command, which allows us to directly run actions or workflows without having to mess with sensors or rules.

We can run `st2 run core.local -h` to see what parameters are required by that action without actually running it:

```
vagrant@st2vagrant:~$ st2 run core.local echo -h

Action that executes an arbitrary Linux command on the localhost.

Required Parameters:
    cmd
        Arbitrary Linux command to be executed on the local host.
        Type: string

Optional Parameters:
    cwd
        Working directory where the command will be executed in
        Type: string

    env
        Environment variables which will be available to the command(e.g.
        key1=val1,key2=val2)
        Type: object

    kwarg_op
        Operator to use in front of keyword args i.e. "--" or "-".
        Type: string
        Default: --

    timeout
        Action timeout in seconds. Action will get killed if it doesn't finish
        in timeout seconds.
        Type: integer
        Default: 60
```

As shown in the previous example, the `core.local` action requires a single positional parameter—namely, the command you wish the action to execute (which in our case is the `echo`

command):

```
vagrant@st2vagrant:~$ st2 run core.local echo "Hello World!"
.
id: 59598d2bc4da5f0506c24981
status: succeeded
parameters:
  cmd: echo Hello World!
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: Hello World!
  succeeded: true
```

The output shows that our command succeeded, and that `stdout` contains the string we passed to `echo`.

Now that we're comfortable with the command line, we can look at something a bit more relevant to network automation. Assuming we've installed the `napalm` pack using the `st2 pack install napalm` command shown previously, we can use `st2 actions list` to view all the actions available to us in this pack:

```
vagrant@st2vagrant:~$ st2 action list --pack=napalm
+-------------------------------------+
| ref                                 |
+-------------------------------------+
| napalm.bgp_prefix_exceeded_chain    |
| napalm.check_consistency            |
| napalm.cli                          |
| napalm.configuration_change_workflow |
| napalm.get_arp_table                |
| napalm.get_bgp_config               |
| napalm.get_bgp_neighbors            |
| napalm.get_bgp_neighbors_detail     |
| napalm.get_config                   |
| napalm.get_environment              |
| napalm.get_facts                    |
| napalm.get_firewall_policies        |
| napalm.get_interfaces               |
| napalm.get_lldp_neighbors           |
| napalm.get_log                      |
| napalm.get_mac_address_table        |
| napalm.get_network_instances        |
| napalm.get_ntp                      |
| napalm.get_optics                   |
| napalm.get_probes_config            |
| napalm.get_probes_results           |
| napalm.get_route_to                 |
```

```
| napalm.get_snmp_information          |
| napalm.interface_down_workflow       |
| napalm.loadconfig                    |
| napalm.ping                          |
| napalm.traceroute                    |
+--------------------------------------+
```

In order to use this pack, we need to configure it so that the pack is able to understand how to reach our network devices and how to authenticate to them. All packs are configured with YAML files located at */opt/stackstorm/configs/*, so this pack's configuration will be located at */opt/stackstorm/configs/napalm.yaml*. A minimal configuration is shown here:

```
---
html_table_class: napalm
config_repo: https://github.com/StackStorm/vsrx-configs.git

credentials:
  local:
    username: root
    password: Juniper

devices:
- hostname: vsrx01
  driver: junos
  credentials: local
```

In this configuration, we have a single device with a hostname `vsrx01`, using the `local` credentials, resulting in a username `root` and a password `Juniper`.

When we make changes to the configuration, it's important to ensure StackStorm is aware of those changes by reloading them. The `st2ctl` utility is useful for doing this:

```
vagrant@st2vagrant:~$ st2ctl reload --register-configs
Registering content...[flags = --config-file /etc/st2/st2.conf --register-configs]
2017-07-03 01:49:44,941 INFO [-] Connecting to database "st2" @ "0.0.0.0:27017" as
user "stackstorm".
2017-07-03 01:49:45,056 INFO [-] ================================================
2017-07-03 01:49:45,056 INFO [-] ############### Registering configs ###############
2017-07-03 01:49:45,056 INFO [-] ================================================
2017-07-03 01:49:45,181 INFO [-] Registered 1 configs.
##### st2 components status #####
st2actionrunner PID: 1007
st2actionrunner PID: 1053
st2api PID: 891
st2api PID: 1286
st2stream PID: 893
st2stream PID: 1288
st2auth PID: 874
```

```
st2auth PID: 1287
st2garbagecollector PID: 872
st2notifier PID: 884
st2resultstracker PID: 879
st2rulesengine PID: 888
st2sensorcontainer PID: 864
st2chatops PID: 881
mistral-server PID: 1032
mistral-api PID: 987
mistral-api PID: 2703
mistral-api PID: 2706
```

NOTE At the time of this writing all actions in the `napalm` pack require at least one argument, namely `hostname`. This lets the action know which device in the pack configuration you intend to work with.

Now, we should be able to run NAPALM actions. The `napalm.get_facts` action will retrieve facts about our network device:

```
vagrant@st2vagrant:~$ st2 run napalm.get_facts hostname=vsrx01
..
id: 5959a495c4da5f0506c2498a
status: succeeded
parameters:
  hostname: vsrx01
result:
  exit_code: 0
  result:
    raw:
      fqdn: vsrx01
      hostname: vsrx01
      interface_list:
      - ge-0/0/0
      - ge-0/0/1
      - ge-0/0/2
      - ge-0/0/3
      model: FIREFLY-PERIMETER
      os_version: 12.1X47-D15.4
      serial_number: da12e84e2e72
      uptime: 240
      vendor: Juniper
  stderr: ''
  stdout: ''
```

Here, we're able to see some useful information about our device, like the vendor and the list of interfaces present.

Actions by design are really intended to perform a single task well. However, in the real world, our

day-to-day tasks in infrastructure rarely take the form of a single task. Usually, the work we do is accomplished over several discrete tasks, and includes some decision making along the way.

For instance, if we notice a router has gone offline, we might want to gather information from its peers. We might want to perform cable checks. For other issues, there may be an entirely different set of tasks required. As discussed previously, workflows allow us to use actions in more interesting ways by allowing us to commit all of these complicated decisions into a text file as if it were source code.

For the sake of brevity, we'll only cover one of the workflow options in StackStorm—specifically, Mistral. Mistral is an OpenStack project that provides two main things:

- A standardized YAML-based language for defining workflows
- Open source software for receiving and processing workflow execution requests

NOTE	When you install StackStorm using the instructions in the documentation, Mistral is installed and runs alongside the other StackStorm processes.

Let's look at a simple example of a Mistral workflow so you can become familiar with how it works.

```
---
version: '2.0'

examples.mistral-basic:
    description: A basic workflow that runs an arbitrary linux command.
    type: direct
    input:
        - cmd
    output:
        stddout: "{{ _.cmd }}"
    tasks:
        task1:
            action: core.local cmd="{{ _.cmd }}"
            publish:
                stdout: "{{ task('task1').result.stdout }}"
```

This workflow has a few important focus points:

- `input` is where we declare the parameters for the workflow. These are published within the workflow as named variables, which can be used in other tasks.
- `output` controls which values are published from the workflow when it finishes.
- `tasks` contains a list of tasks. In this simple example, we have only one: `task1`. Note that this task not only references the action we want to run in that task (namely `core.local`) but also contains a key, `publish`, which assigns the values of `stdout` and `stderr` to similarly named variables (you may have noticed it's using small Jinja snippets to do this). Note also that `stdout` is being passed to `output` so we can see the result when the workflow finishes.

Workflows are executed in the same way we executed our actions earlier. As mentioned previously,

the actual work being done by actions can take the form of a bash or Python script, a simple shell command, or in this case, a Mistral workflow. We can use `st2 run` once more, taking care to pass in the required `cmd` parameter, which the workflow passes into the familiar `core.local` action:

```
vagrant@st2vagrant:~$ st2 run examples.mistral-basic cmd="echo Hello, Mistral!"
.
id: 595ad9c3c4da5f0521ea906c
action.ref: examples.mistral-basic
parameters:
  cmd: echo Hello, Mistral!
status: succeeded
result_task: task1
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: Hello, Mistral!
  succeeded: true
start_timestamp: 2017-07-03T23:56:51.069066Z
end_timestamp: 2017-07-03T23:56:52.442155Z
+--------------------------+------------------------+-------+------------+
| id                       | status                 | task  | action     |
+--------------------------+------------------------+-------+------------+
| 595ad9c3c4da5f0521ea906f | succeeded (0s elapsed) | task1 | core.local |
+--------------------------+------------------------+-------+------------+
```

This output is a bit different from our last example. Any time you run an action, StackStorm produces an action "execution." This execution represents everything about how that action ran. You'll notice in this output as well as the previous examples that each time we call `st2 run`, an execution ID is produced. In the output for our Mistral workflow, this is true, but we also see a table of child executions. These child executions are the tasks in our workflow.

For a more network-centric example, let's look at one of the Mistral workflows in the `napalm` pack (modified for brevity):

```
---
version: '2.0'

napalm.interface_down_workflow:

  input:
    - hostname
    - interface

  type: direct

  tasks:

    show_interface:
```

```
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
      on-success: "show_interface_counters"

    show_interface_counters:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
        counters: true
      on-success: "show_log"

    show_log:
      action: "napalm.get_log"
      input:
        hostname: "{{ _.hostname }}"
        lastlines: 10
```

You'll notice the first two tasks use the `on-success` keyword to control which task runs next (if the statement's enclosing task results in a successful status, that is).

Running this workflow results in some familiar output, showing three executions, one for each of the tasks in our workflow:

```
vagrant@st2vagrant:~$ st2 run napalm.interface_down_workflow hostname=vsrx01
interface="ge-0/0/1"

id: 595adf58c4da5f0521ea90a0
action.ref: napalm.interface_down_workflow
parameters:
  hostname: vsrx01
  interface: ge-0/0/1
status: succeeded
start_timestamp: 2017-07-04T00:20:40.895706Z
end_timestamp: 2017-07-04T00:20:52.735536Z
+--------------------------+-----------+-------------------------
+-----------------------+
| id                       | status    | task                    | action
|
+--------------------------+-----------+-------------------------
+-----------------------+
| 595adf59c4da5f0521ea90a3 | succeeded | show_interface          |
napalm.get_interfaces |
| 595adf5cc4da5f0521ea90a5 | succeeded | show_interface_counters |
napalm.get_interfaces |
| 595adf5fc4da5f0521ea90a7 | succeeded | show_log                | napalm.get_log
|
+--------------------------+-----------+-------------------------
```

```
+----------------------+
```

We can use st2 execution get <*id*> to view the result of one of these executions:

```
vagrant@st2vagrant:~$ st2 execution get 595adf5cc4da5f0521ea90a5
id: 595adf5cc4da5f0521ea90a5
status: succeeded (3s elapsed)
parameters:
  counters: true
  hostname: vsrx01
  interface: ge-0/0/1
result:
  exit_code: 0
  result:
    raw:
      name: ge-0/0/1
      rx_broadcast_packets: 0
      rx_discards: 0
      rx_errors: 0
      rx_multicast_packets: 0
      rx_octets: 0
      rx_unicast_packets: 0
      tx_broadcast_packets: 0
      tx_discards: 0
      tx_errors: 0
      tx_multicast_packets: 0
      tx_octets: 0
      tx_unicast_packets: 0
  stderr: ''
  stdout: ''
```

Finally, we can use some basic branching logic in Mistral to make some more advanced decisions within the workflow. The following example is a slightly modified version of the previous example, but with a new task added at the beginning:

```
---
version: '2.0'

napalm.interface_down_workflow:

  input:
    - hostname
    - interface
    - skip_show_interface

  type: direct

  tasks:
```

```
    decide_task:
      action: "core.noop"
      on-success:
      - show_interface: "{{ _.skip_show_interface != True }}"
      - show_interface_counters: "{{ _.skip_show_interface == True }}"

    show_interface:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
      on-success: "show_interface_counters"

    show_interface_counters:
      action: "napalm.get_interfaces"
      input:
        hostname: "{{ _.hostname }}"
        interface: "{{ _.interface }}"
        counters: true
      on-success: "show_log"

    show_log:
      action: "napalm.get_log"
      input:
        hostname: "{{ _.hostname }}"
        lastlines: 10
```

The `core.noop` action essentially does nothing. This is a common way of making early decisions in a Mistral workflow. The value of the `on-success` key for this task is a list instead of a simple string indicating the next task. In this case, the conditions listed in each list item will determine the next task to run. We can pass `skip_show_interface` into the workflow, and see that `show_interface` does not run.

```
vagrant@st2vagrant:~$ st2 run napalm.interface_down_workflow hostname=vsrx01
interface="ge-0/0/1" skip_show_interface=True

id: 595b4a7ec4da5f0521ea90b4
action.ref: napalm.interface_down_workflow
parameters:
  hostname: vsrx01
  interface: ge-0/0/1
  skip_show_interface: true
status: succeeded
start_timestamp: 2017-07-04T07:57:50.606796Z
end_timestamp: 2017-07-04T07:57:58.032456Z
+--------------------------+-----------+--------------------------
+-----------------------+
| id                       | status    | task                     | action
|
```

```
+--------------------------+----------+------------------------
+----------------------+
| 595b4a7ec4da5f0521ea90b7 | succeeded | decide_task             | core.noop
|
| 595b4a7fc4da5f0521ea90b9 | succeeded | show_interface_counters |
napalm.get_interfaces |
| 595b4a81c4da5f0521ea90bb | succeeded | show_log                | napalm.get_log
|
+--------------------------+----------+------------------------
+----------------------+
```

There's a lot more you can do with actions and workflows, but this will be enough to get you started.

Sensors and Triggers

Actions and workflows are useful in their own right, but in order to enable event-driven automation, we need to gather information about our infrastructure and recognize when actionable events happen. This is accomplished through sensors and triggers. Sensors are little bits of Python code that bring external data into StackStorm—for instance, by periodically pollling REST APIs or subscribing to message queues.

StackStorm also allows you to configure incoming webhooks, which allows external systems to "push" events to StackStorm (sensors provide more of a "pull" model of integration).

NOTE Sensors are the preferred integration method since they offer a more granular and tighter integration, but webhooks offer a simple integration mechanism that allows you to get data into StackStorm quickly. This can be helpful when working with systems that don't yet have a sensor built for them, or that only offer outgoing webhooks as an integration mechanism.

We can see the sensors available on the system using `st2 sensor list` (using the `pack` flag to focus on the `napalm` pack for brevity):

```
vagrant@st2vagrant:~$ st2 sensor list --pack=napalm
+--------------------------+--------+-----------------------------------------
+---------+
| ref                      | pack   | description                             |
enabled |
+--------------------------+--------+-----------------------------------------
+---------+
| napalm.NapalmLLDPSensor | napalm | Sensor that uses NAPALM to retrieve LLDP     |
True    |
|                          |        | information from network devices        |
|
+--------------------------+--------+-----------------------------------------
+---------+
```

This particular sensor periodically queries each of the devices in our configuration file for the LLDP neighbor table. It will keep track of the number of LLDP neighbors active for each device.

As mentioned previously, triggers are the way that StackStorm knows an "event" has occurred. For instance, if the LLDP neighbor count stays the same, we don't need to do anything. However, if that number changes, that's an actionable event. For our example, we have two triggers, one representing a neighbor increase, and the other a neighbor decrease:

```
vagrant@st2vagrant:~$ st2 trigger list --pack=napalm
+----------------------------+--------
+-------------------------------------------------------+
| ref                        | pack   | description
|
+----------------------------+--------
+-------------------------------------------------------+
| napalm.LLDPNeighborDecrease | napalm | Trigger which occurs when a device's LLDP
neighbors   |
|                            |        | decrease
|
| napalm.LLDPNeighborIncrease | napalm | Trigger which occurs when a device's LLDP
neighbors   |
|                            |        | increase
|
+----------------------------+--------
+-------------------------------------------------------+
```

The code that implements our LLDP sensor is responsible for determining when the neighbor count has changed and firing the appropriate trigger.

Let's see if we can cause one of these triggers to fire. We'll log in to our network device and confirm that we can see at least one LLDP neighbor:

```
root@vsrx01> show lldp neighbors
Local Interface    Parent Interface    Chassis Id          Port info          System
Name
ge-0/0/1.0         -                   4c:96:14:10:01:00   ge-0/0/2.0         vsrx02
```

Since we are seeing a neighbor on `ge-0/0/1`, we can shut that interface to clear it from the table.

```
root@vsrx01# set interfaces ge-0/0/1 unit 0 disable

[edit]
root@vsrx01# commit
commit complete
```

This particular sensor periodically provides some useful log messages about the neighbor count.

NOTE	The log file for all sensor activity is usually located at */var/log/st2/st2sensorcontainer.log* but can be located elsewhere via configuration.

```
2017-07-04 23:54:16,134 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 1
2017-07-04 23:54:22,732 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 1
2017-07-04 23:54:28,701 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
went DOWN to 0
2017-07-04 23:54:34,748 139956844022352 INFO lldp_sensor [-] vsrx01 LLDP neighbors
STAYED at 0
```

We can see that the sensor knew that the neighbor count was staying constant at 1, until it decreased to 0, and stayed there.

However, as we've discussed, the trigger is the important part. We can query StackStorm for a list of "trigger instances," which are specific occurrences that a trigger was fired. Each time a neighbor count decreases for a device, the `napalm.LLDPNeighborDecrease` trigger should fire. We can see that this has indeed happened:

```
vagrant@st2vagrant:~$ st2 trigger-instance list --trigger=napalm.LLDPNeighborDecrease
+--------------------------+----------------------------
+------------------------------+
| id                       | trigger                     | occurrence_time  | status
|
+--------------------------+-----------------------------+------------------
+-----------+
| 595c2ab4c4da5f035af38903 | napalm.LLDPNeighborDecrease | < truncated >    |
processed |
+--------------------------+----------------------------
+------------------------------+
```

Finally, in the same way we retrieved details for action executions, we can use the ID for this trigger instance to see the details (payload) of this particular event:

```
vagrant@st2vagrant:~$ st2 trigger-instance get 595c2ab4c4da5f035af38903
+-----------------+-------------------------------------------------+
| Property        | Value                                           |
+-----------------+-------------------------------------------------+
| id              | 595c2ab4c4da5f035af38903                        |
| trigger         | napalm.LLDPNeighborDecrease                     |
| occurrence_time | 2017-07-04T23:54:28.713000Z                     |
| payload         | {                                               |
|                 |     "device": "vsrx01",                         |
|                 |     "timestamp": "2017-07-04 23:54:28.701351",  |
|                 |     "oldpeers": 1,                              |
|                 |     "newpeers": 0                               |
```

```
|                 | }                                                |
| status          | processed                                        |
+-----------------+--------------------------------------------------+
```

This will come in handy as we discuss rules in the next section.

Rules

We finally arrive at the crux of everything we've learned thus far. To recap, actions allow us to perform bread-and-butter automation. Sensors and triggers represent "events" in StackStorm. Now to perform event-driven automation, we need a way to tie events from triggers to actions or workflows. That's where rules come in.

Rules are defined (much like a lot of other things in StackStorm) in YAML files. The plain-English version of a rule might be: "When *X* happens, do *Y*." In our case, *X* is a trigger instance, and *Y* is an action or workflow.

One of the simplest examples of a rule is the ability to run an `echo` every few seconds. There's a special trigger in StackStorm called `core.st2.IntervalTimer` that allows us to treat the passage of a certain amount of time as an actionable event:

```
---
name: sample_rule_with_timer
pack: "examples"
description: Sample rule using an Interval Timer.
enabled: true

trigger:
  parameters:
    delta: 5
    unit: seconds
  type: core.st2.IntervalTimer

criteria: {}

action:
  parameters:
    cmd: echo "{{trigger.executed_at}}"
  ref: core.local
```

There are three main parts to every rule:

`trigger`

: This section specifies the name of the trigger we want to watch for (in this case, `core.st2.IntervalTimer`) as well as any parameters that trigger requires.

`criteria`

: This will further restrict which trigger instances will match this rule. Since nothing is specified

here, all instances of `core.st2.IntervalTimer` will fire this rule.

`action`

Here we specify the action or workflow we want to fire, as well as any parameters it requires. As long as a trigger instance is seen that matches the other two sections, this action or workflow will execute.

```
vagrant@st2vagrant:~$ st2 execution list -a id action.ref context.user status
+--------------------------+--------------------+-------------
+-------------------------+
| id                       | action.ref         | context.user | status
|
+--------------------------+--------------------+-------------
+-------------------------+
|   595c3093c4da5f035af38bb0 | core.local         | stanley      | succeeded (1s
elapsed)  |
|   595c3098c4da5f035af38bb6 | core.local         | stanley      | succeeded (1s
elapsed)  |
|   595c309dc4da5f035af38bbc | core.local         | stanley      | succeeded (1s
elapsed)  |
+--------------------------+--------------------+-------------
+-------------------------+
vagrant@st2vagrant:~$ st2 execution get 595c309dc4da5f035af38bbc
id: 595c309dc4da5f035af38bbc
status: succeeded (1s elapsed)
parameters:
  cmd: echo "2017-07-05 00:19:41.779821+00:00"
result:
  failed: false
  return_code: 0
  stderr: ''
  stdout: '2017-07-05 00:19:41.779821+00:00'
  succeeded: true
```

A dead giveaway that these executions were fired by our rule is that the user that executed them is the built-in user `stanley`, instead of the user we're logged in with (in this case, `st2admin`).

We can use a rule to notify us via Slack when our LLDP neighbor count decreases:

```
---
name: "lldp_notify"
pack: "napalm"
enabled: true
description: "Notify of LLDP Neighbor Decrease"

trigger:
  type: "napalm.LLDPNeighborDecrease"
  parameters: {}
```

```
criteria: {}

action:
  ref: slack.post_message
  parameters:
    message: "WARNING: {{trigger.device}}'s LLDP Neighbors just went DOWN to
      {{trigger.newpeers}} (was {{ trigger.oldpeers }})""
    channel: '#general'
```

In the previous example, we're watching for all instances of the trigger `napalm.LLDPNeighborDecrease`. When one occurs, the `slack.post_message` action is called to post a message to slack (notice that fields in this trigger's payloads like `device` and `newpeers` are referenced to add context to the message).

We could, of course, go further than simple notifications. Let's say the number of LLDP neighbors decreased because someone logged on to our device and accidentally shut one of the interfaces (like we did in the section on triggers). We'd probably push a config like this to bring the interface back online:

```
interfaces {
    ge-0/0/1 {
        unit 0 {
            enable;
        }
    }
}
```

Instead of simply raising a notification, we could react to this event by automatically pushing this configuration using the `napalm` pack's `loadconfig` action:

```
---
name: "lldp_remediate"
pack: "napalm"
enabled: true
description: "Bring interface back up when LLDP neighbor count decreases"

trigger:
  type: "napalm.LLDPNeighborDecrease"
  parameters: {}

criteria: {}

action:
  ref: napalm.loadconfig
  parameters:
    hostname: "{{ trigger.device }}"
    config_file: /vagrant/remediation_config.txt
```

As discussed previously, running workflows is functionally very similar to running actions. You may have a full workflow for doing troubleshooting, auto-remediation, notifications, or a combination of these. Rules can execute full-blown workflows, or simple actions.

StackStorm Summary

There's a lot more to StackStorm, but these core concepts will get you started down the road of auto-remediation and event-driven network automation. The idea behind StackStorm is to give you the tools that allow you to put processes in place that react autonomously to infrastructure events, in the same way that you would manually, but in a more reliable fashion.

For more information on StackStorm, the best place to visit is the project's documentation, as well as the free Slack channel, where there's almost always someone on hand ready to answer any questions.

Summary

In this chapter, we discussed how some automation tools—like Ansible, Salt, and Terraform—can be put to work in a network automation use case. We provided examples for using the products for network automation, and we discussed the advantages and disadvantages of each product along the way.

With all these tools, and other technologies from previous chapters, the next chapter will show you how to apply the software development best practices of Continous Integration to the network automation domain.