3. Configuration Management with Salt

Salt (also known as SaltStack) is a remote execution tool and a configuration management system. In this chapter, we will explore both functionalities of Salt using the command line interface.

Chapter Details	
Chapter Goal	Get familiar with Salt Stack
Chapter Sections	*2.1. Explore the Environment* *2.2. Salt Command Line Interface* *2.3. Salt Grains* *2.4. Salt Files* *2.5. Salt Pillars* *2.6. Salt States* *2.7. Salt State Tree* *2.7.1. Manage a state using top file* *2.7.2. Manage an application stack* *2.8. Jinja Templates* *2.8.1. Jinja configuration templating* *2.8.2. Jinja Maps* *2.9. Salt Formulas*

3.1. Explore the Environment

In your environment, there are 3 virtual machines that we are going to use in this chapter:

Host name	IP address	User	Password
master	10.0.0.2	stack	b00tcamp
node1	10.0.0.11	stack	b00tcamp
node2	10.0.0.12	stack	b00tcamp

Verify that you can access **master** using the specified IP address, user name and password:

```
stack@lab:~$ ssh master
stack@master's password: b00tcamp
...
stack@master:~$
```

Then log out from **master** and verify access to **node1** and **node2**:

```
stack@lab:~$ ssh node1
stack@node1's password: b00tcamp
...
stack@node1:~$ exit

stack@lab:~$ ssh node2
stack@node2's password: b00tcamp
```

```
...
stack@node2:~$ exit
```

3.2. Salt Command Line Interface

Salt command line client communicates with the Salt master using Salt client API. Root users can easily send commands simultaneously to all minions through the master node.

Step 1 Log-in to the **master** node as *stack* user

```
stack@lab:~$ ssh master
stack@master's password: b00tcamp
stack@master:~$
```

Step 2 Switch to root user privilege

```
stack@master:~$ sudo su
root@master:/home/stack#
```

> **Important:**
>
> Salt commands require sudo privileges to function because it must have access to certain root directories.

Step 3 Execute the *salt-key* command to view the list of minion public keys reported to master:

```
root@master:/home/stack# salt-key
Accepted Keys:
Denied Keys:
Unaccepted Keys:
master
node1
node2
Rejected Keys:
```

In this example, there are 3 minion keys that are not accepted by master yet (the master itself is also a minion). To accept all minions, execute the following command:

```
root@master:/home/stack# salt-key -y -A
The following keys are going to be accepted:
Unaccepted Keys:
master
node1
node2
Key for minion master accepted.
Key for minion node1 accepted.
Key for minion node2 accepted.
```

Verify that master has accepted the minions:

```
root@master:/home/stack# salt-key
Accepted Keys:
master
node1
node2
Denied Keys:
Unaccepted Keys:
Rejected Keys:
```

Step 4 Test the availability of the minions

```
root@master:/home/stack# salt \* test.ping
node2:
    True
node1:
    True
master:
    True
```

The `salt` command line tool takes two parameters:

- *Target*: Denoted by `\*` which targets all minions (Alternatively, `'*'`)
- *Module*: Denoted by `test.ping` which runs the *ping* function in the *test* module

Step 5 Lets try to execute a more complex command

```
root@master:/home/stack# salt \* network.interfaces
master:
----------
...
eth1:
    ----------
    hwaddr:
        ...
    inet:
          ----------
          address:
              10.0.0.2
          broadcast:
              10.0.0.255
          label:
              eth1
          netmask:
              255.255.255.0
...
```

This command will execute function *interfaces* from the module called *network*. As a result you will get an output that describes all interfaces on all minions.

Step 6 Target a specific minion to check the disk usage

```
root@master:/home/stack# salt 'node1' disk.usage
node1:
----------
/:
    ----------
    1K-blocks:
        8662024
    available:
        7292980
    capacity:
        12%
    filesystem:
        /dev/mapper/primary-root
    used:
        905992
...
```

Using the minion name for the target parameter allows for a specific minion to run the command and return its output.

Step 7 Use the offline system documentation to check module availability and usage

```
root@master:/home/stack# salt \* sys.doc disk
...
```

This command will come in handy for quickly referencing the built-in Salt modules. Lastly, if there are no default modules that fit your needs, it is possible to write your own.

> **Reference:**
>
> For more information on built-in Salt execution modules, visit:
> https://docs.saltstack.com/en/latest/ref/modules/all/index.html#all-salt-modules

3.3. Salt Grains

Grains are key-value pairs that represent static information about the minions. This may include OS family, kernel, CPU, and network information. **Core Grains** are loaded by default with commonly used system information and you do not need to define your own Grains.

Step 1 List the grains on all minions

```
root@master:/home/stack# salt \* grains.items
master:
    ----------
    SSDs:
    biosreleasedate:
        01/01/2011
    biosversion:
        Bochs
    cpu_flags:
        - fpu
        - vme
...
```

> **Notes:**
>
> Due to the static nature of its information, Grains are commonly used to target a specific minion or a set of minions. For example, run a command only on minions with Debian systems.

Step 2 List the grains on all minions with the key *os_family*

```
root@master:/home/stack# salt \* grains.item os_family
node2:
    ----------
    os_family:
        Debian
node1:
    ----------
    os_family:
        Debian
master:
    ----------
    os_family:
        Debian
```

Step 3 Use the `-G` option to target minions matching the `os:ubuntu` grain. Get the memory information for the targetted minions.

```
root@master:/home/stack# salt -G 'os:ubuntu' status.meminfo
master:
    ----------
    Active:
        ----------
        unit:
            kB
        value:
            986788
...
```

3.4. Salt Files

The Salt Master maintains a lightweight file server to distribute regular files and Salt States. This server utilizes Salt transport system and it is not generally intended to be used to transfer large files.

The main reason for having developed its own filesystem is to allow the Salt master server to send out small chunks of files to its minions asynchronously.

Step 1 View the Salt master configuration `/etc/salt/master` and search for the `file_roots` key:

```
root@master:/home/stack# cat /etc/salt/master | grep 'file_roots:' -A 2
# file_roots:
#   base:
#     - /srv/salt/
--
#file_roots:
#  base:
#    - /srv/salt
```

As we see, the `file_roots` key is not set (commented out). The Salt file server is environment aware. This means specific directories will be served for specific environments depending on the configuration. In the `/etc/salt/master` file, we can define the environments under the `file_roots` option. By default, `base` is defined with the path `/srv/salt/`.

Let's create a sample file and see the file server in action.

Step 2 Create default `/srv/salt` directory to serve the files

```
root@master:/home/stack# mkdir /srv/salt
```

Step 3 Create sample file `/srv/salt/sample.cfg` with a sample string

```
root@master:/home/stack# echo 'sample config' > /srv/salt/sample.cfg
```

The `sample.cfg` is located in the `/srv/salt` directory, and this root filesystem is defined in the Salt master configuration file, so that file can be referenced by all minions with the URI *salt://sample.cfg*.

Step 4 Using the `cp` module, request the file string by referencing its URI

```
root@master:/home/stack# salt \* cp.get_file_str salt://sample.cfg
node1:
    sample config
node2:
    sample config
master:
    sample config
```

The output indicates that all minion nodes can reference the `/srv/salt/sample.cfg` file by its URI. The `cp.get_file_str` module downloaded `sample.cfg` file to the minion cache directory and returned its contents as output.

3.5. Salt Pillars

Pillar is an interface for Salt designed to offer global values that can be distributed to minions. Pillar data is compiled on the master. Additionally, pillar data for a given minion is only accessible by the minion for which it is targeted in the pillar configuration. This makes pillar useful for storing sensitive data specific to a particular minion.

The Salt Master server maintains a *pillar_roots* configuration that matches the structure of the *file_roots* used in the Salt file server.

Step 1 View the master configuration file and search for *pillar_roots* option

```
root@master:/home/stack# cat /etc/salt/master | grep 'pillar_roots' -A 3
# pillar_roots:
#   base:
#     - /srv/pillar
```

The default configuration specifies `/srv/pillar` directory as the base environment. Pillar files in this directory can be referenced by the *base* environment by its name.

Step 2 Create the pillar directory

```
root@master:/home/stack# mkdir /srv/pillar
```

A `top.sls` file in the pillar directory declares minion access to pillar files.

Step 3 Create a file named `top.sls` in the `/srv/pillar/` directory and populate the file with the following content:

```
base:
  '*':
    - common
  node1:
    - web
  node2:
    - db
```

In this example, all minions can access the `/src/pillar/common.sls` file. The only minion *node1* can access `/srv/pillar/web.sls` and the only minion *node2* can access `/srv/pillar/db.sls`

> **Notes:**
>
> Take care to use ‘space’ characters when editing YAML files. Tabs will not be accepted.

Pillar data are serialized structures like strings, lists, and dictionaries.

Step 4 Create a new `common.sls` pillar file in the `/srv/pillar/` directory with the following content:

```
log_dir: /var/log
log_level: debug
```

Step 5 Create a new `db.sls` pillar file in the `/srv/pillar/` directory with the following content:

```
database:
  bind: 127.0.0.1
  port: 3306
  user: ubuntu
```

Step 6 Create a new `web.sls` pillar file in the `/srv/pillar/` directory with the following content:

```
keep_alive_timeout: 40
```

To distribute the new pillar information from the Master to appropriate minions, it is recommended to run a utility command to refresh pillars.

Step 7 Use the Salt utility module to refresh the pillars on all minions

```
root@master:/home/stack# salt \* saltutil.refresh_pillar
```

Step 8 View pillar data on *node1*:

```
root@master:/home/stack# salt node1 pillar.items
node1:
    ----------
    keep_alive_timeout:
        40
    log_dir:
        /var/log
    log_level:
        debug
```

Step 9 View pillar data on *node2*

```
root@master:/home/stack# salt node2 pillar.items
node2:
    ----------
    database:
        ----------
        bind:
            127.0.0.1
        port:
            3306
        user:
            ubuntu
    log_dir:
        /var/log
    log_level:
        debug
```

Step 10 Query a specific pillar key-value on *node2*

```
root@master:/home/stack# salt node2 pillar.item database:user
node2:
----------
database:user:
    ubuntu
```

3.6. Salt States

Salt state is an expression of desired state on a host written in human readable configuration file format called SaLt State file (SLS). States are a core part of the configuration management capabilities of Salt. It is built on top of remote execution modules which we explored in the previous sections.

In the following exercise, we will create simple state files and apply the changes to the environment.

Step 1 Create a new `edit.sls` state file in the `/srv/salt/` directory with the following content:

```
install_vim:                # Arbitrary ID declaration (will be referenced in top file)
  pkg.installed:            # state module call
    - name: vim             # function argument
```

The state file called `edit.sls` that we created ensures that the package *vim* is installed on the target system.

Step 2 Apply the state `edit` on `node1`:

```
root@master:/home/stack# salt node1 state.apply edit
```

Salt will verify that vim package presents in the system, if it’s not - it will install it. Suppose that this state file will install text editors (hence called `edit.sls`). Let’s add another package for the state.

Step 3 Modify the `/srv/salt/edit.sls` file and add another declaration to install *nano*:

```
install_vim:
  pkg.installed:
    - name: vim

install_nano:
  pkg.installed:
    - name: nano
```

> **Notes:**
>
> As your states become more complex, it is a good idea to do sanity checks before applying the state.

Step 4 Render the SLS file `edit`:

```
root@master:/home/stack# salt node1 state.show_sls edit
node1:
----------
install_vim:
```

```
    ----------
    __env__:
        base
    __sls__:
        edit
    pkg:
        |_
          ----------
          name:
              vim
        - installed
        |_
          ----------
          order:
              10000
install_nano:
    ----------
    __env__:
        base
    __sls__:
        edit
    pkg:
        |_
          ----------
          name:
              nano
        - installed
        |_
          ----------
          order:
              10001
```

Here you can observe all the functions to be called, the parameters, and the order of execution

Step 5 Test your state (dry run)

```
root@master:/home/stack# salt node1 state.sls edit test=True
```

This will check all conditions and report what will be changed without executing functions. Note that some tests will return failure notices since functions may depend on previous sequence of commands to execute on the environment.

Step 6 Apply the `edit` state

```
root@master:/home/stack# salt node1 state.apply edit
node1:
----------
          ID: install_vim
    Function: pkg.installed
        Name: vim
      Result: True
     Comment: Package vim is already installed
     Started: 18:26:00.839505
    Duration: 301.31 ms
     Changes:
----------
          ID: install_nano
    Function: pkg.installed
        Name: nano
      Result: True
     Comment: The following packages were installed/updated: nano
     Started: 18:26:01.140990
    Duration: 4.695 ms
     Changes:
```

```
Summary for node1
------------
Succeeded: 2
Failed:    0
------------
Total states run:     2
Total run time: 306.005 ms
```

In this example, the packages specified in the state file were already installed. If the package needs to be installed, it may take a few minutes for the command to return after the minions finish installing the packages.

3.7. Salt State Tree

Salt State Tree is a collection of state files organized in the environment directory. You can move beyond using single SLS files and orchestrate host systems utilizing the State Tree. The following is a typical directory hierarchy of a state tree:

```
/srv
|
+-- /salt                # File roots specified in /etc/salt/master
     |
     +-- top.sls         # Mapping of groups of machines and its configuration
     |
     +-- /<app_dir>
     |    |
     |    +-- /files     # Config files used by this application
     |    +-- /resources
     |    +-- init.sls   # Default state of app
     |
     +-- /<app_dir>
     ...
```

In the next few steps, we will create several applications and utilize the State Tree to orchestrate the environments.

3.7.1. Manage a state using top file

Step 1 Create a new directory for Apache web server:

```
root@master:/home/stack# mkdir /srv/salt/apache2
```

Step 2 Create a new file `init.sls` file in the `/srv/salt/apache2/` directory and declare the state of Apache:

```
apache2:
  pkg:
    - installed
  service:
    - running
    - reload: True
    - watch:
      - file: /etc/apache2/apache2.conf

/etc/apache2/apache2.conf:
  file.managed:
    - source: salt://apache2/files/apache2.conf
```

The state ensures that Apache2 package is installed into the system, the service is running and if there are any changes applied to `/etc/apache2/apache2.conf` file the service will restart.

> **References:**
>
> For a full list of state modules, visit SaltStack Documentation

Step 3 Create a new file `top.sls` file in the `/srv/salt/` directory:

```
base:
  'node1':
    - apache2
```

The file defines one environment called `base` and `node1` target to apply `apache2` state.

Step 4 Apply the top state as a dry run and observe the output

```
root@master:/home/stack# salt node1 state.highstate test=True
...
node1:
----------
          ID: apache2
    Function: pkg.installed
      Result: None
     Comment: The following packages would be installed/updated: apache2
     Started: 01:43:14.160336
    Duration: 313.908 ms
     Changes:
----------
          ID: /etc/apache2/apache2.conf
    Function: file.managed
      Result: False
     Comment: Source file salt://apache2/files/apache2.conf not found
     Started: 01:43:14.476147
    Duration: 17.408 ms
     Changes:
----------
          ID: apache2
    Function: service.running
      Result: False
     Comment: One or more requisite failed: apache2./etc/apache2/apache2.conf
     Changes:
...
```

The highstate dry run output indicates that we are missing a configuration file called `apache2.conf`. Let's copy over the prepared configuration file to the Salt directory.

Step 5 Create the `/srv/salt/apache2/files/` directory:

```
root@master:/home/stack# mkdir /srv/salt/apache2/files
```

Step 6 Download `apache2.conf` into the `/srv/salt/apache2/files` directory:

```
root@master:/home/stack# wget https://www.linode.com/docs/assets/apache2.conf
```

Step 7 Copy `apache2.conf` from `/home/stack/` to `/srv/salt/apache2/files/`:

```
root@master:/home/stack# cp /home/stack/apache2.conf /srv/salt/apache2/files/
```

Step 8 Run the highstate with the test option and observe the output

```
root@master:/home/stack# salt node1 state.highstate test=True
...
node1:
----------
          ID: apache2
    Function: pkg.installed
      Result: None
     Comment: The following packages would be installed/updated: apache2
     Started: 01:49:01.139980
    Duration: 340.046 ms
     Changes:
----------
          ID: /etc/apache2/apache2.conf
    Function: file.managed
      Result: None
     Comment: The file /etc/apache2/apache2.conf is set to be changed
     Started: 01:49:01.482094
    Duration: 18.529 ms
     Changes:
----------
          ID: apache2
    Function: service.running
      Result: False
     Comment: The named service apache2 is not available
     Started: 01:49:01.500850
    Duration: 21.064 ms
     Changes:

Summary for node1
------------
Succeeded: 2 (unchanged=2)
Failed:    1
```

There is 1 failure due to `apache2` service not being available. This is expected since Apache was not installed yet on the node.

Step 9 Run the highstate without the test option

```
root@master:/home/stack# salt node1 state.highstate

node1:
----------
          ID: apache2
    Function: pkg.installed
      Result: True
     Comment: The following packages were installed/updated: apache2
     Started: 20:19:07.303969
    Duration: 11658.733 ms
     Changes:
...
Summary for node1
------------
Succeeded: 3 (changed=3)
Failed:    0
------------
Total states run:     3
Total run time:  11.867 s
```

Step 10 Test the apache webserver by executing an HTTP request on node1

```
root@master:/home/stack# salt node1 cmd.run 'curl http://localhost'
# HTML for apache default page goes here
```

3.7.2. Manage an application stack

We will apply the same concepts from the previous section to deploy a full application stack. The following is known as a *LAMP* stack:

- `Linux`
- `Apache HTTP Server`
- `MySQL Database`
- `PHP Programming Language`

As you may have guessed, we will need to create additional state files which install and manage the LAMP stack components (the minions are already running Linux, so you do not need to create a state to install this).

Step 1 Create a new directory for `PHP` states:

```
root@master:/home/stack# mkdir /srv/salt/php
```

Step 2 Create a new file `init.sls` in the directory `/srv/salt/php/`:

```
php:
  pkg:
    - installed
    - names:
      - php
      - php-mysql
```

The state declares two packages to be installed: `php` and `php-mysql`.

Step 3 Edit `/srv/salt/apache2/init.sls` and include mods for running `PHP`:

```
apache2:
  pkg:
    - installed
    - names:
       - apache2
       - libapache2-mod-php
    - require:
      - pkg: php

  service:
    - running
    - reload: True
    - watch:
      - file: /etc/apache2/apache2.conf

/etc/apache2/apache2.conf:
  file.managed:
    - source: salt://apache2/files/apache2.conf
```

We added the *libapache2-mod-php* package to enable php in Apache and specified that this state requires PHP package installed.

Step 4 Create a new directory for `MySQL` states:

```
root@master:/home/stack# mkdir /srv/salt/mysql
```

Step 5 Create a new file `init.sls` in the `/srv/salt/mysql/` directory with the following content:

```
mysql-server:
  pkg:
    - installed
  service:
    - name: mysql
    - running
```

The state installs *mysql-server* package and ensures that service is running.

Step 6 Edit `/srv/salt/top.sls` and assign states to desired nodes:

```
base:
  '*':
    - edit
  node1:
    - apache2
    - php
  node2:
    - mysql
```

Step 7 Run the highstate to apply changes:

```
root@master:/home/stack# salt \* state.highstate
# Summary for all nodes should be successful
```

The summary of the output is the following:

- `edit` state is applied on all nodes (*master*, *node1*, *node2*)
- `apache2` and `php` states are applied on *node1*
- `mysql` state is applied on *node2*

3.8. Jinja Templates

Jinja is a template engine for Python. For Salt, Jinja is the default templating language in SLS files. One of the common use of Jinja is to allow for embedding conditional statements within the State files. Furthermore, *Grains* and *Pillars* can be accessed as variables.

Here is an example of Jinja templating in action in a Salt State file.

```
{% set variables = ['foo', 'bar'] %}
install_variables:
  pkg.installed:
  {% for var in variables %}
    {% if var != '' %}
    - name: {{ var }}
    {% else %}
    - name: foo
    {% endif %}
  {% endfor %}
```

References:

For more information on Jinja, visit http://jinja.pocoo.org

In this exercise, we will go over how to leverage Jinja to make our SLS files configurable.

3.8.1. Jinja configuration templating

Step 1 Rename the `apache2.conf` file in `/srv/salt/apache2/files/` directory with a Jinja suffix:

```
root@master:/home/stack# mv /srv/salt/apache2/files/apache2.conf /srv/salt/apache2/file
```

Step 2 Edit `/srv/salt/apache2/files/apache2.conf.jinja` and replace the `KeepAliveTimeout` value with a Jinja template variable:

```
...
KeepAliveTimeout {{keep_alive_timeout}}
...
```

Step 3 To enable Jinja processing of the configuration file in the `init.sls` state, edit `/srv/salt/apache2/init.sls`:

```
...
/etc/apache2/apache2.conf:
  file.managed:
    - source: salt://apache2/files/apache2.conf.jinja
    - template: jinja
    - keep_alive_timeout: {{salt['pillar.get']('keep_alive_timeout')}}
```

Keep in mind that the following are parameters being passed into the `file.managed` module:

- `source` now references the renamed `apache2.conf.jinja` file
- `template` specifies `jinja` templating to be used
- `keep_alive_timeout` arbitrary parameter interpreted by `file.managed` module
  in this case, Salt Pillar is referenced to set this variable

> **Notes:**
>
> The Salt Pillar keep_alive_timeout was set in the /srv/pillar/web.sls created in an earlier chapter

Step 4 Apply the highstate:

```
root@master:/home/stack# salt \* state.highstate
...
          ID: /etc/apache2/apache2.conf
    Function: file.managed
      Result: True
     Comment: File /etc/apache2/apache2.conf updated
     Started: 00:42:49.210318
    Duration: 53.069 ms
     Changes:
              ----------
              diff:
                  ---
                  +++
                  @@ -102,7 +102,7 @@
                  ...
                   # KeepAliveTimeout: Number of seconds to wait for the next request from
```

```
                 # same client on the same connection.
                 #
                -KeepAliveTimeout 5
                +KeepAliveTimeout 40
                ...
 ...
```

The diff shows that the entire file has changed because we renamed it. After applying the highstate the changes are reflected on the nodes in `/etc/apache2/apache2.conf`. Since the apache service also watches the configuration file, it is restarted as well.

Step 5 Check the `apache2.conf` file on a minion to see if the variable has been set to **40**

```
root@master:/home/stack# salt node1 cmd.run 'cat /etc/apache2/apache2.conf | grep KeepA
node1:
 # KeepAliveTimeout: Number of seconds to wait for the next request from the
 KeepAliveTimeout 40
```

You have successfully parametrized apache configuration. The minions now use a pillar value `keep_alive_time` to configure its file. Since this file is being managed by the Salt module, re-applying the highstate will ensure that the pillar value will update the configuration file.

Step 6 Edit the `/srv/pillar/web.sls` file and change the `keep_alive_timeout` value:

```
keep_alive_timeout: 60
```

Step 7 Re-apply the highstate

```
root@master:/home/stack# salt \* state.highstate
...
      ID: /etc/apache2/apache2.conf
Function: file.managed
  Result: True
 Comment: File /etc/apache2/apache2.conf updated
 Started: 00:55:38.730885
Duration: 37.396 ms
 Changes:
          ----------
          diff:
              ---
              +++
              @@ -102,7 +102,7 @@
               # KeepAliveTimeout: Number of seconds to wait for the next request from
               # same client on the same connection.
               #
              -KeepAliveTimeout 40
              +KeepAliveTimeout 60


               # These need to be set in /etc/apache2/envvars
----------
          ID: apache2
    Function: service.running
      Result: True
     Comment: Service reloaded
...
```

Notice that the value has properly changed and service was restarted.

3.8.2. Jinja Maps

The best practices as outlined by *SaltStack* documentation is to use a **map.jinja** file for mapping the differences between software distributions. For example, our configuration file works with Debian, but on a different platform such as RedHat we will need to introduce package names specific to that platform.

> **References:**
>
> For more information on Salt best practices, see the Salt documentation

Step 1 Create a new file `map.jinja` file in the `/srv/salt/apache2` directory with the following content:

```
{% set apache = salt['grains.filter_by']({
    'Debian': {
        'server': 'apache2',
        'service': 'apache2',
        'conf': '/etc/apache2/apache2.conf',
    },
    'RedHat': {
        'server': 'httpd',
        'service': 'httpd',
        'conf': '/etc/httpd/httpd.conf',
    },
}, merge=salt['pillar.get']('apache:lookup')) %}
```

Step 2 Modify `/srv/salt/apache2/init.sls` to utilize `map.jinja` file:

```
{% from "apache2/map.jinja" import apache with context %}

{{ apache.service }}:
  pkg:
    - installed
    - names:
       - {{ apache.server }}
       - libapache2-mod-php
    - require:
      - pkg: php

  service:
    - running
    - reload: True
    - watch:
      - file: {{ apache.conf}}

{{ apache.conf}}:
  file.managed:
    - source: salt://apache2/files/apache2.conf.jinja
    - template: jinja
    - keep_alive_timeout: {{salt['pillar.get']('keep_alive_timeout')}}
```

Step 3 Apply the high state

```
root@master:/home/stack# salt \* state.highstate
```

Now this deployment supports both Debian and RedHat

3.9. Salt Formulas

Salt Formulas are pre-defined Salt States that perform a collection of common tasks. Each formula is located in an individual git repository which can be found on github: https://github.com/saltstack-formulas

The idea of formula is to provide a “building block” for your environment. A formula may be a particular service or system setup task that is immediately usable with default parameters.

In this exercise, we will install the NTP (Network Time Protocol) formula to all nodes.

Step 1 Create a new directory `/srv/formulas`:

```
root@master:/home/stack# mkdir /srv/formulas
```

Step 2 Go to the formulas directory and clone the NTP formula from the public Git repository:

```
root@master:/home/stack# cd /srv/formulas
root@master:/srv/formulas# git clone https://github.com/saltstack-formulas/ntp-formula
```

Step 3 Create a new file `master.conf` in the `/etc/salt/master.d/` directory and insert the following content:

```
file_roots:
  base:
    - /srv/salt
    - /srv/formulas/ntp-formula
```

Step 4 Restart Salt master to take effect:

```
root@master:/home/stack# service salt-master restart
```

Step 5 Add `ntp` to all minions in `/srv/salt/top.sls`:

```
base:
  '*':
    - edit
    - ntp
  node1:
    - apache2
    - php
  node2:
    - mysql
```

`ntp` is recognized by the top file because we specified its origin in the `file_roots` variable and the directory of the formula is `/srv/formulas/ntp-formula/ntp/`

Step 6 Last but not least, apply the highstate:

```
root@master:/home/stack# salt \* state.highstate
```

Notice that ntp service, dependencies, and configuration files have been installed.

Congratulations, you have finished the *2. Configuration Management with Salt* chapter!

Checkpoint:

- Operate the Salt command line client
- Knowledge of Salt components and their functions
- Managing Salt states using the State Tree
- Familiarity with Jinja templates
- Familiarity with Salt formulas and its use