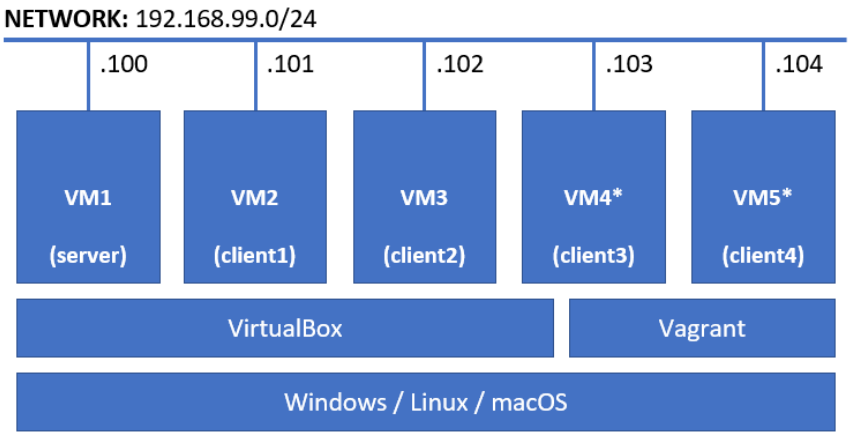
Practice M3: Salt

For this practice, our lab environment will look like this



We are going to use mostly **CentOS Stream 9** boxes and at least one **Debian**-based box

All configurations and supplementary files are provided as a ZIP archive and can be downloaded from the module section in the official site

Part 1

Let us start with the environment

Set up the environment

We will build the first version of the environment by using just a part of the provided **Vagrantfile**

Once done exploring it, we can deploy the infrastructure with

```
vagrant up
```

Install Salt repository

Now, that we have the infrastructure, we can continue with the installation

It is fairly simple, and it is usually a matter of installing a single package

Depending on your choice of distribution for the **Salt** host, follow the appropriate installation instructions

First, establish a session to the **Salt** host *(it is a CentOS-based machine in the provided Vagrantfile)*

```
vagrant ssh server
```

Red Hat/CentOS

Add the repository key

```
sudo rpm --import
https://repo.saltproject.io/salt/py3/redhat/9/x86_64/latest/SALTSTACK-GPG-KEY2.pub
```

Then add the repository

```
curl -fsSL https://repo.saltproject.io/salt/py3/redhat/9/x86_64/latest.repo | sudo tee
/etc/yum.repos.d/salt.repo
```

SUSE/openSUSE

There is no need to do anything special, in terms of preparation, on **SUSE**/**openSUSE**

Debian

On **Debian** (11), we must prepare a directory for the keyring

```
mkdir /etc/apt/keyrings
```

And then, add the key

```
sudo curl -fsSL -o /etc/apt/keyrings/salt-archive-keyring.gpg
https://repo.saltproject.io/salt/py3/debian/11/amd64/latest/salt-archive-keyring.gpg
```

Then create the apt sources list file

```
echo "deb [signed-by=/etc/apt/keyrings/salt-archive-keyring.gpg arch=amd64]
https://repo.saltproject.io/salt/py3/debian/11/amd64/latest bullseye main" | sudo tee
/etc/apt/sources.list.d/salt.list
```

Finally, update package information

```
sudo apt-get update
```

Ubuntu

On **Ubuntu** (22.04), we must add the key

```
sudo curl -fsSL -o /etc/apt/keyrings/salt-archive-keyring.gpg
https://repo.saltproject.io/salt/py3/ubuntu/22.04/amd64/latest/salt-archive-
keyring.gpg
```

And then create apt sources list file

```
echo "deb [signed-by=/etc/apt/keyrings/salt-archive-keyring.gpg arch=amd64]
https://repo.saltproject.io/salt/py3/ubuntu/22.04/amd64/latest jammy main" | sudo tee
/etc/apt/sources.list.d/salt.list
```

Finally, update package information

```
sudo apt-get update
```

First steps (Agentless)

Install the binary

Install the **salt-ssh** binary

Red Hat/CentOS

Execute

```
sudo dnf install salt-ssh
```

SUSE/openSUSE

Execute

```
sudo zypper install salt-ssh
```

Debian/Ubuntu

Execute

```
sudo apt-get install salt-ssh
```

Prepare the roster file

Open the roster file for editing

```
sudo vi /etc/salt/roster
```

And add the following

web1:

host: 192.168.99.101

user: vagrant

passwd: vagrant

sudo: True

web2:

host: 192.168.99.102

user: vagrant

passwd: vagrant

sudo: True

Save and close the file

Remote execution

Now, try to execute a few commands

First, test the communication to the first client with

```
sudo salt-ssh web1 test.ping
```

Nothing happens. Press **Ctrl+C** to stop the execution

Execute this one instead (**-i** will disable host key checking)

```
sudo salt-ssh -i web1 test.ping
```

After a while, it will return result indicating that everything went fine

If you want, check the other station as well

Should we want to address all stations, we can use this instead

```
sudo salt-ssh -i '*' test.ping
```

Add two more stations to the roster file

```
db1:
  host: 192.168.99.103
  user: vagrant
  passwd: vagrant
  sudo: True
```

```
db2:
  host: 192.168.99.104
  user: vagrant
  passwd: vagrant
  sudo: True
```

Now, should we want to address just the two newly added stations, we can execute

```
sudo salt-ssh -i 'db*' test.ping
```

More about this module (**test**) can be found here:

https://docs.saltproject.io/en/latest/ref/modules/all/salt.modules.test.html

We can ask for the list of available or known nodes with

```
sudo salt-ssh -H
```

We should see all the four hosts there

This means that we can omit the -i option and target all the hosts with

```
sudo salt-ssh '*' test.ping
```

Now, let's execute an arbitrary command in all nodes

For this, we will need the **cmdmod** module

For example, let's ask for their host name

```
sudo salt-ssh '*' cmd.run 'hostname'
```

If we execute the command again, we will notice that the order is different

So, the order of execution is not guaranteed

This is because by default **25** simultaneous connections are open to the target stations

We can control this and make the communication serial – one connection at a time

```
sudo salt-ssh --max-procs=1 '*' cmd.run 'hostname'
```

Now the results appear in the order in which the stations are registered in the roster file

We should always be careful what symbols we use to surround the arguments

For example, this

```
sudo salt-ssh '*' cmd.run 'echo $HOSTNAME'
```

Will return different output compared to this

```
sudo salt-ssh '*' cmd.run "echo $HOSTNAME"
```

Execute a few more. For example

```
sudo salt-ssh '*' cmd.run 'uname -a'
sudo salt-ssh '*' cmd.run 'uptime'
sudo salt-ssh '*' cmd.run 'df -h'
```

More about this module (**cmdmod**) can be found here:

https://docs.saltproject.io/en/latest/ref/modules/all/salt.modules.cmdmod.html

First steps (Master + Minions)

Now, let's drop the agentless mode and switch to master and minions

Set up the master

Make sure that you are on the **server** machine

There are multiple ways to install **Salt** server (master)

Two viable options are to use the target system package manager, or to use the bootstrap script, provided by **Salt**

For this practice, we will choose the second option

Download the bootstrap script

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

We can check the available options with

```
sh bootstrap-salt.sh -h
```

Install the latest stable version – just the master (-M), without the minion (-N) part, do not start the daemons (-X):

```
sudo sh bootstrap-salt.sh -M -N -X
```

Now, we must open the firewall ports (if a firewall is present and active)

```
sudo firewall-cmd --permanent --add-port=4505-4506/tcp
sudo firewall-cmd --reload
```

It is time to enable, start the **Salt** master service, and check if everything is okay

```
sudo systemctl enable salt-master
sudo systemctl start salt-master
systemctl status salt-master
```

Okay, we are ready to move forward

Set up the minions

As with the server part, here also exist multiple options, but again, we will stick to the bootstrap way

On all four *(or at least the first two – **web1** and **web2**)* client stations execute the following steps

Download the bootstrap script

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

Install the latest stable version of just the minion part and run it as a daemon:

```
sudo sh bootstrap-salt.sh
```

Now, we must point the minions to the master

Open the **/etc/salt/minion** file

```
sudo vi /etc/salt/minion
```

Uncomment the **master** entry (row 16), and enter the **Salt** server name (**server**)

Save and close the file

And restart the service

```
sudo systemctl restart salt-minion
```

Register the minions

Now return to the **master**, check if there are waiting minions, and accept them eventually

In order to examine the situation with the waiting minions, you must execute

```
sudo salt-key -L
```

Then you can examine a key

```
sudo salt-key -f client-web-1
```

And accept all unauthorized keys

```
sudo salt-key -A
```

If we check again

```
sudo salt-key -L
```

We will see that all waiting minions were accepted

First commands

We should be now ready to start our journey with **Salt**

First, let's experiment with a few command line constructions. Some of them, we already know

Test the communication with all minions

```
sudo salt '*' test.ping
```

We can ask for different information this time, let's ask for the usage of disks and then for their blkids

```
sudo salt '*' disk.usage
```

```
sudo salt '*' disk.blkid
```

We can narrow down the target systems, for example ask only the client #1

```
sudo salt client-web-1 disk.blkid
```

In order to check what functions are supported by particular minion, we can execute

```
sudo salt client-web-1 sys.doc
```

We can do a few more executions

```
sudo salt '*' cmd.run 'cat /etc/os-release'
```

```
sudo salt '*' network.ip_addrs
```

```
sudo salt '*' pkg.install unzip
```

*If nothing returns from the last command, then try with another package. For example, with the **cowsay** package*

Of course, we can limit the targets by applying the known technique

```
sudo salt 'client-db*' cmd.run 'hostname'
```

Targeting nodes with grains

We can filter the target set of minions by utilizing the detailed information available for each one

This set of data in **Salt** is called **Grains** and it is equivalent to the **Facts** used in other ***Configuration Management*** solutions

Let's explore a little bit in this direction

Check what information is available for one of the minions

```
sudo salt client-web-1 grains.items
```

Now narrow down the ping command for example, and apply it only on the **Red Hat** based hosts

```
sudo salt -G 'os_family:RedHat' test.ping
```

Part 2

We continue with the same environment from ***Part 1***

Make sure that all machines are up and running and you are on the master (server) machine

Preparation

Before we continue further, we must configure and initialize the so-called **Salt State Tree**

For this purpose, we must open the master configuration file

```
sudo vi /etc/salt/master
```

Uncomment the lines (685– 687):

```
file_roots:
  base:
    - /srv/salt
```

Save and exit

Restart the **Salt** master

```
sudo systemctl restart salt-master
```

Now, we can go to the **/srv** folder and prepare the structure and files

Create the **/srv/salt** folder

```
sudo mkdir /srv/salt
```

Remote execution and states

As we already know, we can send commands to the minions

For example, we can say to all minions or some part of them to install a package (do not execute it)

```
sudo salt '*' pkg.install cowsay
```

This is not the typical way of using **Salt**

Instead, we use state files and apply them to the minions

This way we are stating the desired state

Let's prepare a simple state file **/srv/salt/test-state.sls** with the following content

```
install_cowsay_package:
  pkg.installed:
    - name: cowsay
```

Save and close the file

First test what would be the outcome of applying the above state

```
sudo salt '*' state.apply test-state test=True
```

It appears that it will work

We can shorten the file to this

```
cowsay:
  pkg.installed
```

We can use the package name as a name for the section

Let's apply the state

```
sudo salt '*' state.apply test-state
```

This way, we applied individual state

Simple top file and state file

In addition, we can have a mapping between states and minions

For this, we should create the top file (**top.sls**)

Create file **top.sls** in **/srv/salt** with the following content

```
base:
  '*':
    - demo
```

Now, create the **/srv/salt/demo.sls** file with the following content

```
create.user:
  user:
    - name: demo
    - fullname: Demo User
    - createhome: True
    - present
install.screen:
  pkg:
    - name: screen
    - installed
```

Let's save the file and apply the state to all minions but in test mode

```
sudo salt '*' state.apply test=True
```

*Note that we omitted the name of the state file (**demo.sls**) because it is referred by the **top.sls** file*

It works

Multiple state files #1

Imagine that our state file is much bigger and contains multiple states

Can we break the state file in smaller parts? Yes, we can

Create two copies of the **demo.sls** file – **demo-usr.sls** and **demo-pkg.sls**

The **/srv/salt/demo-usr.sls** file should contain

```
create.user:
  user:
```

```
    - name: demo
    - fullname: Demo User
    - createhome: True
    - present
```

And the **/srv/salt/demo-pkg.sls** file should contain

```
install.screen:
  pkg:
    - name: screen
    - installed
```

Now, let's change the **/srv/salt/top.sls** file to this

```
base:
  '*':
    - demo-usr
    - demo-pkg
```

Test again with

```
sudo salt '*' state.apply test=True
```

Multiple state files #2

Delete all the files in **/srv/salt**

```
sudo rm -rf /srv/salt/*
```

Let's imagine that we have:

- a package or set of packages that must be installed on all stations
- a package or set of packages that must be installed on just a group of stations

How can we achieve this? Multiple state files? Yes, this is possible

Let's implement it

Create a file **/srv/salt/common-pkg.sls** with the following content

```
install.common.packages:
  pkg.installed:
    - pkgs:
      - zip
      - htop
```

Then create **/srv/salt/specific-pkg.sls** file with the following content

```
install.specific.packages:
  pkg.installed:
    - pkgs:
      - rsync
      - wget
      - curl
```

Now, let's create the **/srv/salt/top.sls** file with the following content

```
base:
  '*':
    - common-pkg
  'client-web*':
    - specific-pkg
```

Test it with

sudo salt '*' state.apply test=True

Filtering (separate states) #1

What if we have to apply a package with one name on some of the stations and with another name on the rest?

There are multiple ways to achieve this

Let's implement one based on what we know so far

Delete again all files from **/srv/salt**

sudo rm -rf /srv/salt/*

Create a file **/srv/salt/apache-redhat.sls** with the following content

```
install.apache.redhat:
  pkg:
    - name: httpd
    - installed
run.apache.redhat:
  service.running:
    - name: httpd
    - require:
      - pkg: httpd
```

And another one, named **/srv/salt/apache-debian.sls** with the following content

```
install.apache.debian:
  pkg:
    - name: apache2
    - installed
run.apache.debian:
  service.running:
    - name: apache2
    - require:
      - pkg: apache2
```

Now, we can use each of these two commands to target different minions

sudo salt 'client-web-1' state.apply apache-redhat test=True

sudo salt 'client-web-2' state.apply apache-debian test=True

Of course, this is not the way to go

Instead, let's create a **/srv/salt/top.sls** file with the following content

```
base:
  'client-web-1':
    - apache-redhat
  'client-web-2':
    - apache-debian
```

Now, we can use this command (to test)

sudo salt '*' state.apply test=True

We will see errors that a few of the clients are not referenced by the top file. Instead, we could execute this:

sudo salt '*web*' state.apply test=True

This works, but what if we have more machines? Will we reference each one or name them appropriately?

It could work, but this is not the way to go

Filtering (using grains) #2

Let's go over the grain information once again

Execute this to check the grains coming from one of the client machines

sudo salt client-web-1 grains.items

We can incorporate this in the top file

Change the **/srv/salt/top.sls** file to match this

```
base:
  'os_family:RedHat':
    - match: grain
    - apache-redhat
  'os_family:Debian':
    - match: grain
    - apache-debian
```

This way, instead of listing individual nodes, we are using the collected information to decide where to send the states

Test the setup with

sudo salt '*' state.apply test=True

Filtering (jinja) #3

We can bring this on the next level

Delete all files in **/srv/salt**

sudo rm -rf /srv/salt/*

Create a new state file named **/srv/salt/apache.sls** with the following content

```
{% if grains['os_family'] == 'RedHat' %}
{%   set vpackage = 'httpd' %}
{% else %}
{%   set vpackage = 'apache2' %}
{% endif %}
install.webserver:
  pkg:
    - name: {{ vpackage }}
    - installed
run.webserver:
  service.running:
    - name: {{ vpackage }}
    - require:
      - pkg: {{ vpackage }}
set.index:
  file.managed:
    - name: /var/www/html/index.html
```

```
    - contents: '<h1>Hello Salt World!</h1><br /><hr /><h5>Running on
{{ grains['os_family'] }}</h5>'
    - require:
      - pkg: {{ vpackage }}
```

Create new **/srv/salt/top.sls** file with the following content

```
base:
  '*':
    - apache
```

Save and execute a dry run

```
sudo salt '*' state.apply test=True
```

If all seems to be okay, then apply the state to both clients (web clients only) (skip it)

```
sudo salt 'client-web*' state.apply
```

Pillars

We can use external data to drive the states

Preparation

While on the master, open the main configuration file for editing

```
sudo vi /etc/salt/master
```

Go to row **850** and uncomment **850** to **852**

Save and close the file

Create the folder to store the data

```
sudo mkdir -p /srv/pillar
```

Restart the master service

```
sudo systemctl restart salt-master
```

See it in action

Imagine that we want to create a set of three files, whose name we want to be able to change easily

Create a file **/srv/pillar/file_data.sls** to hold the data

```
files:
  file1:
    path: /tmp/file1.txt
    content: 'Hello File 1'
  file2:
    path: /file2.txt
    content: 'Hello File 2'
```

Now, create the top file but for the pillar - **/srv/pillar/top.sls**

```
base:
  '*':
    - file_data
```

We can check the pillar data with

```
sudo salt '*' pillar.items
```

If needed, we can ask minions to refresh the pillar data

```
sudo salt '*' saltutil.refresh_pillar
```

Let's create the **/srv/salt/files.sls** state file

```
{% for file_name, file_data in pillar.get('files', {}).items() %}
{{ file_name }}:
  file.managed:
    - name: {{ file_data['path'] }}
    - contents: {{ file_data['content'] }}
{% endfor %}
```

And then the **/srv/salt/top.sls** file

```
base:
  '*':
    - files
```

Now, we can test the whole setup with

```
sudo salt '*' state.apply test=True
```

Should we want, we may apply it as well (by removing the **test=True** part)

Beacons and Reactors

Let's explore how the beacons and reactors work

Preparation

While still on the master (server) create a subfolder in **/srv/salt**

sudo mkdir /srv/salt/files

And create a simple dummy configuration file **/srv/salt/files/dummy.conf** with the following content

 # dummy config file

option=yes

Save and close the file

Now, again on the master, create a state file named **/srv/salt/dummy_conf.sls** with the following content

```
/etc/dummy.conf:
  file.managed:
    - source:
      - salt://files/dummy.conf
    - makedirs: True
```

Create a new **/srv/salt/top.sls** file with the following content

```
base:
  '*':
    - dummy_conf
```

Test and apply the configuration

```
sudo salt '*' state.apply test=True
```

```
sudo salt '*' state.apply
```

Beacons

Add one more state file **/srv/salt/packages.sls** with the following content

```
{% if grains['os_family'] == 'RedHat' %}
{%   set vpackage = 'python3-inotify' %}
{% else %}
{%   set vpackage = 'python3-pyinotify' %}
{% endif %}
install.pyinotify:
  pkg:
    - name: {{ vpackage }}
    - installed
```

Create a new subfolder

sudo mkdir /srv/salt/files/minion.d

And store a **beacons.conf** file there with the following content

```
beacons:
  inotify:
    - files:
        /etc/dummy.conf:
          mask:
            - modify
    - disable_during_state_run: True
```

One more step. Create a state file to deploy the beacon configuration named **/srv/salt/beacons.sls** with the following content

```
/etc/salt/minion.d/beacons.conf:
  file.managed:
    - source:
      - salt://files/minion.d/beacons.conf
    - makedirs: True
```

Now, include the two new state files in the **/srv/salt/top.sls** file. It should look like

```
base:
  '*':
    - dummy_conf
    - packages
    - beacons
```

Apply the configuration to the minions

sudo salt '*' state.apply

As we just changed the configuration for the minions that run on the client machines, we must restart them

Let's use the following command to do this

sudo salt-ssh '*' cmd.run 'sudo systemctl restart salt-minion'

Now, open a new session to the master

vagrant ssh server

And execute

sudo salt-run state.event pretty=True

To start watching for events

Now, open a new session to one of the minions

```
vagrant ssh web1
```

And change the **/etc/dummy.conf file** by executing this

```
echo 'newline=value' | sudo tee -a /etc/dummy.conf
```

Return to the session where we watch for events

A message should appear

Explore the message structure and contents

We will base our next iteration on those 😉

Reactors

We can utilize the refactor mechanism to mitigate a drift registered by beacons

Edit the master's configuration to enable the reactor functionality

```
sudo vi /etc/salt/master
```

Navigate to the reactor section and enter the following

```
reactor:
  - salt/beacon/*/inotify//etc/dummy.conf:
    - /srv/reactor/dummy_file.sls
```

Save and close the file

Create the **/srv/reactor** folder

```
sudo mkdir /srv/reactor
```

And there, create a **/srv/reactor/dummy_file.sls** file with the following content

```
restore dummy.conf:
  local.state.single:
    - tgt: {{ data['id'] }}
    - kwarg:
        fun: file.managed
        name: /etc/dummy.conf
        source: salt://files/dummy.conf
```

Now, restart the master

```
sudo systemctl restart salt-master
```

Or stop it and start it in debug mode

```
sudo systemctl stop salt-master
sudo salt-master -l debug
```

Go to the second session to the master and make sure you are watching for events, if not, start again

sudo salt-run state.event pretty=True

Now, go to the third session – the one established to one of the client machines

Check again the contents of the **dummy.conf** file with

cat /etc/dummy.conf

And then make additional change with

echo 'another-change=value' | sudo tee -a /etc/dummy.conf

Now, return to the session in which we were watching for events

Some new events should appear there. Explore them

Return back on the client and check again the contents of the file

cat /etc/dummy.conf

Everything should be as it was initially

Part 3

Let's create our own, yet simple, module

Own module #1

Create a folder to store the modules

sudo mkdir /srv/salt/_modules

Create there a file named **demo.py** with the following content

```
def msg(text):
    '''
    Display a message with <text> in upper case
    CLI Example:
    .. code-block:: bash
        salt '*' demo.msg 'Hello Salt! :)'
    '''
    return text.upper()
```

Save and close the file

Now, make sure that all minions have this module

sudo salt '*' saltutil.sync_modules

And then execute it

sudo salt '*' demo.msg 'Hello Salt'

We can ask for module's documentation with

sudo salt '*' sys.doc demo.msg

Own module #2

Let's create another one

This one will return current weather data for a city by its coordinates

We will use the **Open Meteo** service - https://open-meteo.com/en

You can explore the API here - https://open-meteo.com/en/docs

Again, in the **/srv/salt/_modules** folder, create a file named **weather.py** with the following content

```
import logging
try:
    import requests
    HAS_REQUESTS = True
except ImportError:
    HAS_REQUESTS = False

log = logging.getLogger(__name__)

__virtual_name__ = 'weather'

def __virtual__():
    '''
    Only load weather if requests is available
    '''
    if HAS_REQUESTS:
        return __virtual_name__
    else:
        return False, 'The weather module cannot be loaded: requests package
unavailable.'


def get(coordinates=None):
    '''
    Gets the Current Weather

    CLI Example::

        salt '*' weather.get lat,long
    '''
    log.debug(coordinates)
    return_value = {}
    coordinates = coordinates.split(',')
    return_value = _make_request(coordinates[0],coordinates[1])
    return return_value

def _make_request(lat,long):
    '''
    The function that makes the request for weather data from the National Weather
Service.
    '''
    request = requests.get('https://api.open-
meteo.com/v1/forecast?latitude='+str(lat)+'&longitude='+str(long)+'&current_weather=tr
ue')
    conditions = {
        "time:": request.json()["current_weather"]["time"],
        "temperature": round(request.json()["current_weather"]["temperature"], 1)
    }
```

```
    return conditions
```

Save and close the file

Make sure that minions have it

```
sudo salt '*' saltutil.sync_modules
```

Check how documentation appears

```
sudo salt '*' sys.doc weather
```

Test it with the coordinates of Sofia

```
sudo salt '*' weather.get 42.6875,23.3125
```

And then using the ones for Varna

```
sudo salt '*' weather.get 43.21667,27.91667
```