Aymen El Amri - 2022

Every book has a story, this story has a book

I wanted to resign from my job, my suggestions about working on creating a centralized configuration system have not been considered by the team leader. Worse than that, it was not something he really understood. Even my colleagues at that time didn't understand what is a configuration management system. At that time, "DevOps" wasn't a buzz word and folks in software engineering were not conscious about the power of what was coming next.

Like many of us, I'm lazy when it comes to repeating tasks manually but passionate when I automate them. Automation = Power.

I love automation, and in that previous job, there are hundreds of configuration files and thousands of variables to copy from text files then to adjust in some platforms, a huge number of poorly-configured servers, and hundreds of servers to manage by a very small team.. I wanted to work on the automation of some weekly procedures. I was aware that this is a good solution, but it was not the priority neither for the manager nor for the client. I'm talking about a role within a team of 14 people working on the integration of several heavy applications (mainly Java/Oracle, PHP/Mysql, Nginx, Python/Jython) with a complex architecture, tens of versions/environments to manage, and a critical infrastructure covering all the European continent. You can imagine how the word "change" can be seen as a horror movie for team leaders and decision-makers.

In the beginning, I was obliged to follow my boss's guidelines and the work methods my team has adopted, which have one goal: satisfying as fast as possible the unceasing demands of the client.

No, but .. wait, this is not satisfying for me!

I spent almost two weeks searching and working on some solutions before I convinced my boss to give me the time to set up an application prototype that will ease the heavy load, accelerate daily procedures and reduce human errors.

First, I created a configuration management tool using Python/Sqlite3, I spent weeks developing it and it was funny. Some weeks later, I discovered that a tool called Salt exists and found it then a good solution to replace my homemade platform. It meets the expectations of the integration process, or rather, the continuous integration, deployments, and automatic tests.

I hesitated between several alternatives: Ansible, Puppet, CFEngine, Chef .. etc. The choice was made based on several criteria (I was looking for robust, fast configuration management and remote execution tool that everybody can use without learning a new programming language) .. I have never regretted my choice. I found some difficulties. Let me admit: **many** difficulties. And honestly, when I started learning Salt, the official documentation was not as complete and good as now. I even still think the actual documentation can be better.

This book is the fruit of long hours of work and self-learning.

Well, in the beginning, I wanted to resign from my job, just a few days after discovering Salt, I was in love with it, and with what I was doing and learning. I tried Salt the first time when I saw my team taking more than 3 days (sometimes more) to configure hosted platforms at each deployment (we had more than 10 environments per application). After setting it up, the same procedure was taking less than 1/2 hour.

Through this book, it's your turn to discover SaltStack, and I will be your guide.

I wish you a pleasant reading.

To whom is this book addressed?

To developers, system administrators, and anyone working in one of these teams in collaboration with the other or simply in an environment that requires knowledge in development, software integration, and system engineering.

Developers without knowledge in system engineering, can't imagine how stability is important to system administration.
System administrators without software development knowledge, can't imagine how bad and non-productive it is to be limited by system administration constraints and limits.

This is an ironic way to describe the conflicts between developers and system administrators, but in many cases, it's true.

Moreover, within the same company, there is generally some tension between the two teams: System administrators accuse developers of writing code that consumes memory, does not meet system security standards, or is not adapted to available machines configurations. Developers accuse system administrators of being lazy, of lacking innovation, and of being seriously uncool!

No more mutual accusations, now with the evolution in software development, infrastructure, and adopted methodologies (such as Scrum, XP, Kanban), the rise of the DevOps culture is a natural result. The software must be produced at a frenetic pace while at the same time, the development in cascade seems to have reached its limits. DevOps seeks closer collaboration and a combination of different roles and skills involved in software development, such as the role of developer, responsible for operations, and responsible for quality assurance.

- If you are a fan of configuration management, automation, and the DevOps culture
- If you are a system administrator working on DevOps, SysOps, CloudOps, ResearchOps .. well, Ops in general
- If you are a developer seeking to join the new movement

If you are new to infrastructure as code, configuration management, remote execution, and orchestration, this book is also addressed to beginners.

How to properly enjoy this guide

This guide contains technical explanations and shows in each case an example of a command or a configuration to follow.

The explanation gives you a general idea, and the code that follows gives you convenience and helps you to practice what you are reading. Preferably, you should always look at both parts for a maximum of understanding.

Like any new tool or programming language you learned, it is normal to find difficulties and confusion in the beginning, perhaps even after.

If you are a total beginner, do not worry; everyone has passed at least once by this kind of situation.

In the beginning, try to do a diagonal reading while focusing on the basic concepts, then try the first practical manipulation on your server or just using your laptop/desktop and occasionally come back to this book for further reading about a specific subject or concept.

In this book, I've been using Digital Ocean as it's cheap and fast. You can follow my referral link to create an account and get $25 credits, which is quite enough to learn and test all examples in this book.

This book is neither an encyclopedia nor documentation, but it sets out the most important parts to learn and master, if you find words or concepts that you are not comfortable with, take your time and do your online research.

Learning can be a serial process, so understanding a topic requires the understanding of another one.

Through this course, you will learn how to install, configure, and use SaltStack. You will be guided by hands-on labs and practical examples.

Through these chapters, try to showcase your acquired understanding, and it will not hurt to go back to previous chapters if you are unsure or in doubt.

Finally, try to be pragmatic and have an open mind if you encounter a problem.

The resolution begins by asking the right questions.

About The Author

Aymen is a Cloud Architect, entrepreneur, founder, and CEO of eralabs, founder and CEO of FAUN.

Aymen helps companies and startups in Europe and the US develop modern applications, build multi-tenant cloud infrastructures, scalable applications, distributed systems, and service-oriented architectures (microservices & PaaS).

You can find Aymen on Twitter and Linkedin.

You can read other books from the same author here.

Overview and requirements

Overview

Salt (often called SaltStack) is a Python-based open-source tool. It can be used for data-driven orchestration, remote execution for any infrastructure, configuration management for any app stack, and much more.

After its acquisition by VMWare, Salt was called Salt Project and is officially definined as "an approach to infrastructure management built on a dynamic communication bus".

Definitions do not give an idea about the power of the tool, but I believe Salt is one of the most useful DevOps tools at a single condition: You know how to use it for your advantage. This is the goal of this course.

In this course, we are going to learn the essentials and the must-know skills to start using SaltStack and employ it to manage your cloud infrastructure, applications, and more.

We are going to answer one of the most popular questions when it comes to configuration management and orchestration tools: What are the differences between Salt, Ansible, Puppet, and Chef.

Besides the architecture and the terminology of Salt, we are going to see how to install and configure it. You will understand how Salt works and how to use Salt states and states' ecosystem to manage your enterprise IT assets. To do this, we are going to see how to create, manage, and test states with Salt, then how to create and use Salt pillars.

We are also going to discover how to execute commands remotely on Salt minions and other important features that Salt offers, like interfacing with your infrastructure to derive information about the underlying system and configuration.

We are going to see how to collect arbitrary data from Salt minions and store it on the Salt master and how to make it available to all Salt minions. We are going to understand how to use Salt ssh to manage hosts and bootstrap minions and some strategies for scaling your infrastructure. We are going to see how Salt can help you in monitoring your system.

We are going to understand how to make your infrastructure state changes after following a system event or a custom event that you can create and we are going to see how to make the infrastructure and its configurations reactive to changes.

We are going to understand the best practices when using Salt and how to troubleshoot it to solve your problems faster. We are going to discover many other topics and features, so fasten your seat belts.

Finally, we will see how orchestration work.

Requirements

For this course, learning Salt does not need many requirements, but this is what you are going to need: a GNU/Linux distribution like Ubuntu or CentOS with Python and Salt installed. You can set up a virtual machine or use a cloud computing provider. It is possible to follow all of this course instructions using your laptop/desktop.

You will also need a passion for automation!

Let's discover some basics about Python, YAML and Jinja2 together as they are used by Salt.

Python

Salt is written in Python, but you do not really need advanced knowledge about this programming language unless you want to do advanced things. Although the difficulty is subjective, most people who tried coding with Python, found it easy and intuitive.

Python is simple and has an easy-to-understand syntax; it is open-source and has large community support. Python code is easy to read, and in most cases, if you have any programming experience, you can understand it just by looking at it. This is a "hello world" Python code:

```
print("hello world")
```

Nothing more. Compared to other programming languages, Python "Hello world" is one of the shortest, simplest and easiest.

Creating a function in Python is also easy and short:

```
def hello_world():
    print("hello world")
```

To learn Python, check this official list of tutorials.

YAML

We are also going to use YAML to write Salt states and Jinja2 as a templating language. Both YAML and Jinja2 are easy-to-learn.

YAML stands for *"YAML Ain't Markup Language"*, and it is used extensively for configuration files with many software like Salt, Ansible, Kubernetes, Docker Compose..etc. As described in the official documentation of Salt, follow these three rules to understand how Salt works with YAML rapidly:

Indentation:

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

Colons:

Python dictionaries are represented in YAML as strings terminated by a trailing colon.

Example:

Python:

```
{'my_key': 'my_value'}
```

YAML:

```
my_key: my_value
```

Example

Python:

```
{
    'first_level_dict_key': {
        'second_level_dict_key': 'value_in_second_level_dict'
    }
}
```

YAML:

```
first_level_dict_key:
  second_level_dict_key: value_in_second_level_dict
```

Dashes

A Python list item is represented by a single dash followed by a space then the value.

```
- value
```

Multiple items, with the same indentation level, are part of the same list:

```
- value1
- value2
- value3
```

Example

Python:

```
{'my_dictionary': ['list_value_one', 'list_value_two', 'list_value_three']}
```

YAML:

```
my_dictionary:
  - list_value_one
  - list_value_two
  - list_value_three
```

FAUN community members wrote many tutorials about YAML, you can check them here. Many of these tutorials are beginner-friendly.

Jinja2

Jinja is the default Salt templating language used in SLS files (Salt States). We use Jinja2 to create dynamic templates of YAML files with conditional blocks, loops and other templating features. This is a basic example:

```
{% if var == value %}
key_one:
    key_two:
        - value_one
{% elif var < value %}
key_three:
    key_four:
        - value_two
```

```
{% else %}
key_five:
    key_six:
        - value_three
{% endif %}
```

The above Jinja2/YAML code will generate a YAML file in function of the value of "var", so if var is equal to "value", we will get:

```
key_one:
    key_two:
        - value_one
```

and if var < value:

```
key_three:
    key_four:
        - value_two
```

Otherwise, the code will return a file containing:

```
key_five:
    key_six:
        - value_three
```

The official documentation is well done, you can learn more about Jinja here.

SaltStack VS. Puppet VS. Chef VS. Ansible

Choosing a configuration management tool is important for the reason that it will be amongst the pillars to build your DevOps pipelines, configuration automation, and remote execution. Choosing a tool over another is also investing your time and your organization resources to learn new paradigms and implement a specific infrastructure and requirements.

SaltStack, Puppet, Chef, and Ansible are industry-wide tools, and they all serve the same purpose: automating and managing your IT infrastructure and systems. However, these tools have initially been built for slightly different purposes, solve the same problem differently, and use different terminologies. In the following parts of this course, we are going to discover the most important differences between them.

For the sake of simplicity, we are going to use the following comparison table:

	SaltStack	Ansible	Puppet	Chef
Why (Motivation)	Thomas S. Hatch, the author of Salt, had solved problems of systems management at scale by creating tools for companies and teams but found these and other open source solutions to be lacking some important features	Ansible was created to provide an agent-less configuration and remote execution tool	Luke Kanies founded Puppet and Puppet Labs in 2005 out of fear and desperation, with the goal of producing better operations tools and changing how we manage systems	Chef was created by Adam Jacob as a tool for his company Opscode to simplify building end-to-end server/deployment tools. The tool turned out to have more potential
How (Terminology)	Salt uses States as directives scripted in SLS (Salt States) Formulas. It has a master that we call Master and children called Minions	Ansible uses Tasks as directives scripted in Playbooks. It has a master that we call the Control Machine, and children called Hosts	Puppet uses Resources as directives scripted in Manifests. It has a master that we call Master and children called Agents	Chef works with three core components (1) A Chef server: As the center of operations, (2) Chef Workstations: VMs where the configuration code for Chef is created, tested, and changed. (3) Chef Nodes: Nodes are the servers Chef pushes changes to.
Who (Companies Using this Technology)	CloudFlare, Paypal	Cisco, NASA	Wallmart, Salesforce	IMDB, Splunk
License	Apache License v2	GNU Public License v3	Apache License v2	Apache License v2
Github Stars	+12k	+51k	+6k	+6k
Programming Language	Python	Python	C++, Clojure	Ruby, Erlang
Configuration Language	YAML	YAML, JSON	Proprietary	Ruby
Database	N/A	N/A	PuppetDB	PosgtreSQL
Deployments (1) (Push/Pull)	Push	Push	Pull	Pull
Transport	ZeroMQ	SSH	Mcollective	RabbitMQ
Agent (2)	Yes (Optional)	No	Yes	Yes
Agentless (3)	Yes	Yes	No	No
Public Clouds (AWS, Azure, GCP)	Yes (Salt Cloud)	Yes	Yes	Yes

	SaltStack	Ansible	Puppet	Chef
Entreprise GUI	SaltStack Entreprise	Ansible Tower	Puppet Entpreprise	Opscode Manage
Open Source GUI	Yes. e.g.: SaltPad	Yes. e.g.:Semaphore	Yes. e.g.:Foreman	Yes. e.g.:Chef Manage
High Availability	Yes	Yes	Yes	Yes
Interoperability	High	High	High	High
Child Nodes Bootstrapping (4)	Yes (Using salt-ssh)	Yes	No	No
Remote Execution	Built-in	Built-in	Mcollective	Knife
Default Architecture	Client Only	Client Only	Client/Server	Client/Server
Language Type (5) (Procedural/Declarative)	Declarative	Procedural	Declarative	Procedural
Mutability (6)	Mutable	Mutable	Mutable	Mutable

Additional notes:

- (1): Salt uses a Salt Master on the master node and a Salt minion on the slave node. When the user set up a configuration, the Salt minion daemon, receives commands from a remote Salt master.
- (2): Salt uses Salt minion on the remote node to control it
- (3): The minion can act as a masterless node if the user setup the right configurations.
- (4): Salt ssh is a Salt module that allows executing Salt commands and states over ssh without installing a salt-minion. The same module can help the user to install and configure a remote Salt minion.
- (5): Declarative programming is where the developer says what he/she wants without having to say how to do it. However, in procedural programming, the developer has to specify the exact steps to get the result.
- (6): Immutability in infrastructure is an approach wherein components are replaced rather than changed.

Installation and configuration

For this chapter, you will need two VMs with hostnames: master01 and minion01, respectively for the Salt master and the Salt minion running on CentOS 8.

How Salt versioning works

Salt uses a simple system for versioning based on major and patch version numbers. Version numbers are in the format MAJOR.PATCH.

Examples:

- Salt 3000
- Salt 3000.1
- Salt 3000.2
- Salt 3000.4

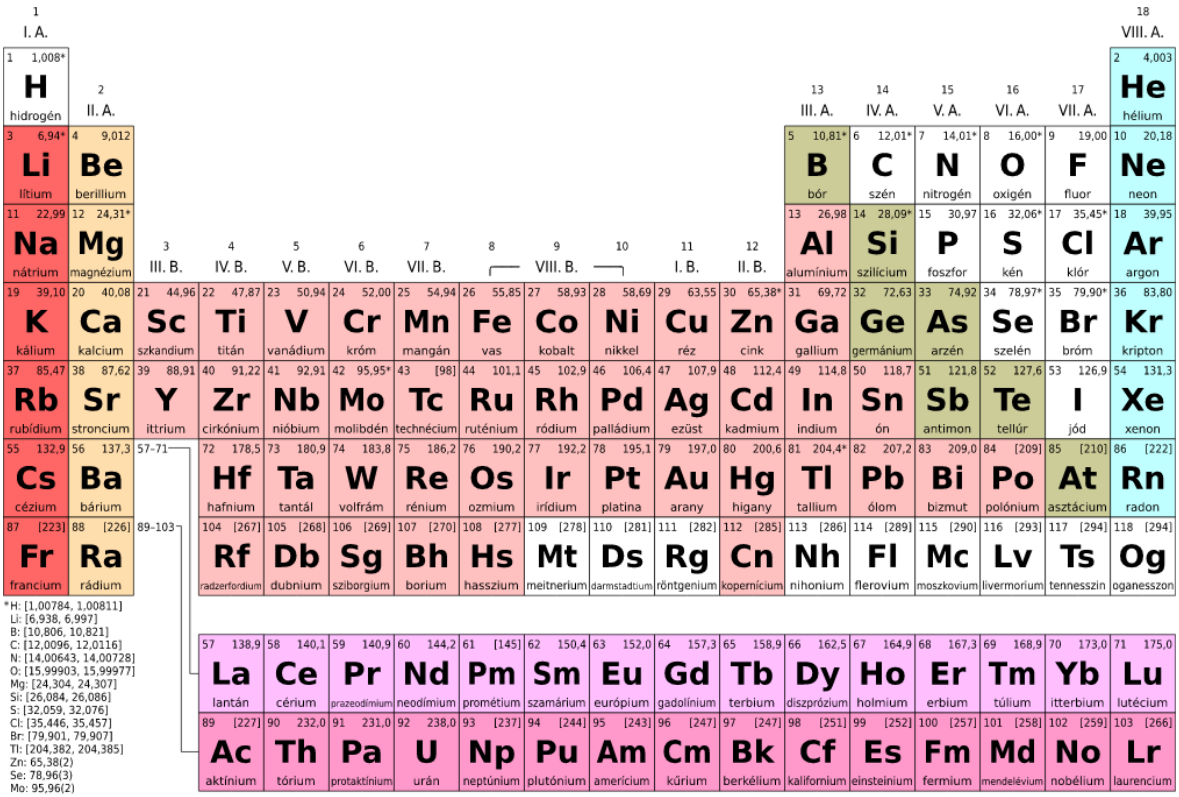
Prior to the 3000 release, Salt used to use another format based on date. This format is using the format YYYY.MM.R where YYYY is the year and MM is the month when the release was created. The bugfix release number (R) increments within that feature release.

Examples:

- Salt 2019.2.8
- Salt 2019.2.7
- Salt 2019.2.6
- Salt 2019.2.5
- Salt 2019.2.4
- Salt 2019.2.3
- Salt 2019.2.2
- Salt 2019.2.1

Each version major version has a codename. For example, Salt 3004 has the codename Silicon, before that, version 3003 has the codename Aluminium, the next version, 3005, will have the codename Phosphorus and version 3006 will have the codename Sulfur.

In other words, each new release will go to the next atomic number.



We are going to use the Salt 3004 Codename Silicon for this course.

Supported OSs

Note: In the following installation parts of this course, we are going to create two VMs, one will have to be the Salt master and the other one will be the minion. For the sake of simplicity, hostnames should be respectively "master" and "minion".

Salt is supported on many *nix systems as long as you have the requirements (e.g.Python).

This is the official list of the supported systems:

- Arch Linux
- Debian GNU/Linux / Raspbian
- Fedora
- FreeBSD
- Gentoo
- OpenBSD
- macOS
- RHEL / CentOS / Scientific Linux / Amazon Linux / Oracle Linux
- Solaris
- Ubuntu
- SUSE

All of the above systems can be a Salt master or a Salt minion. However, some OSs do not support the master and can only be minions:

- Arista EOS
- Cisco Nexus
- Windows

Installing Salt

The official documentation describes in detail every step you need to install Salt on the above systems. We are not going through all of the OSs, but we selected two popular ones: Ubuntu and CentOS.

We are also considering the installation in two different servers, the Salt master on a server and the Salt minion on a different server.

Ubuntu Focal (20)

Packages for Ubuntu Focal (20.04), Ubuntu Bionic (18.04), and Ubuntu Xenial (16) are available in the SaltStack repository.

Let's start by installing the Salt master. Start by running the following commands to import the SaltStack repository key, and to create `/etc/apt/sources.list.d/salt.list`.

Download the key:

```
sudo curl -fsSL -o /usr/share/keyrings/salt-archive-keyring.gpg
https://repo.saltproject.io/py3/ubuntu/20.04/amd64/latest/salt-archive-
keyring.gpg
```

Create the apt source list file:

```
echo "deb [signed-by=/usr/share/keyrings/salt-archive-keyring.gpg arch=amd64]
https://repo.saltproject.io/py3/ubuntu/20.04/amd64/latest focal main" | sudo tee
/etc/apt/sources.list.d/salt.list
```

Update the apt sources:

```
sudo apt-get update
```

Install the master:

```
sudo apt-get install salt-master
```

You can check if the service is running using:

```
systemctl status salt-master
```

On the minion server, follow the same instructions and change the package name:

```
sudo curl -fsSL -o /usr/share/keyrings/salt-archive-keyring.gpg
https://repo.saltproject.io/py3/ubuntu/20.04/amd64/latest/salt-archive-
keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/salt-archive-keyring.gpg arch=amd64]
https://repo.saltproject.io/py3/ubuntu/20.04/amd64/latest focal main" | sudo tee
/etc/apt/sources.list.d/salt.list
sudo apt-get update
sudo apt-get install salt-minion
```

You can check if the service is running using:

```
systemctl status salt-minion
```

Redhat / CentOS 8 PY3

On the master server, start by adding the repository and the necessary key:

```
sudo rpm --import
https://repo.saltproject.io/py3/redhat/8/x86_64/latest/SALTSTACK-GPG-KEY.pub
curl -fsSL https://repo.saltproject.io/py3/redhat/8/x86_64/latest.repo | sudo tee
/etc/yum.repos.d/salt.repo
```

Run:

```
sudo yum clean expire-cache
```

Install the master:

```
sudo yum install salt-master
```

When the installation is completed, start the master service:

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

You can check if the service is running using:

```
systemctl status salt-master
```

On the Salt minion server, execute the same commands while changing the `salt-master` by `salt-minion`:

```
sudo rpm --import
https://repo.saltproject.io/py3/redhat/8/x86_64/latest/SALTSTACK-GPG-KEY.pub
curl -fsSL https://repo.saltproject.io/py3/redhat/8/x86_64/latest.repo | sudo tee
/etc/yum.repos.d/salt.repo

sudo yum clean expire-cache

sudo yum install salt-minion
```

Enable the corresponding services:

```
sudo systemctl enable salt-minion && sudo systemctl start salt-minion
```

You can check if the service is running using:

```
systemctl status salt-minion
```

Understanding the communication model

It's obvious that the master and the minion should be able to communicate without problems.

But before this, since each component is installed on a different server, we need to check if our two servers are able to communicate. In other words, both servers need to be on the same physical or virtual network. To do this, you will need to execute simple Linux commands.

Let's start by pinging each server from the other one.

As an example, the minion address is `10.135.0.3` and the master one is `10.135.0.2`.

On the master export the minion address to a variable:

```
export MINION_IP=10.135.0.3
```

Change 10.135.0.3 by your minion IP address.

On the minion machine, export the master IP address:

```
export MASTER_IP=10.135.0.2
```

From the master, ping the minion:

```
ping -c 4 $MINION_IP
```

On the minion, ping the master:

```
ping -c 4 $MASTER_IP
```

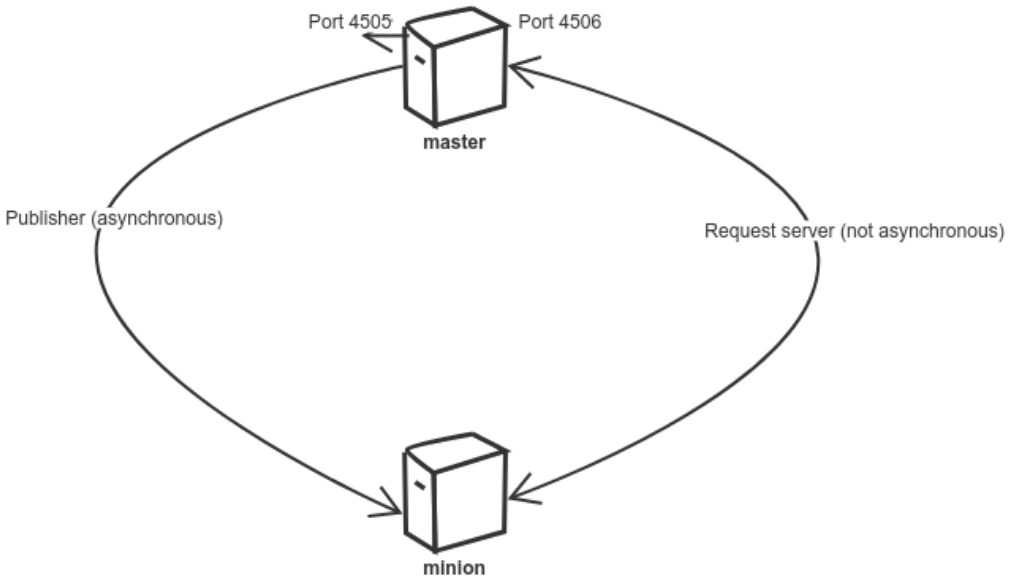
This is the first level of connectivity, but we need more. By default, the Salt master listens on ports 4505 and 4506 on 0.0.0.0 (all interfaces). We need to open these ports on the master machine if they're not.

Before proceeding, it's important to understand how the Salt communication model works.

All Salt communications are managed by Salt itself, all we need to do is apply some configurations. When the minion and the master are configured without any problem, a publish-subscribe communication system will be established.

Connections are initiated by the minion. This means that you do not need to open any incoming ports on those machines. However, the Salt master uses ports 4505 and 4506. These ports should be opened to accept the incoming connections.

The minion creates a persistent connection to the publisher port on the master (4505). The publisher port is typically used to send commands asynchronously.
The minion establishes also another connection to the master machine on the request server (port 4506) to send execution results or to request files and data (including minion-specific data values). This connection, unlike the one done on the publisher port, is not asynchronous.



Let's now make sure that both components of Salt are able to communicate. Since the minion is the connection establisher, we need to check if the remote ports of the master can be reached from the minion.

On the minion machine, execute the following:

```
export PORT_1=4505
export PORT_2=4506

nmap $MASTER_IP -p $PORT_1
nmap $MASTER_IP -p $PORT_2
```

You should be able to see that both ports are not closed, they are likely to be "filtered" in our case.

In the next parts, we are going to effectively connect both master and minion.

Post installation configurations

Salt minion is usually configured using the `/etc/salt/minion` file. This file is by default commented and has no effect after a fresh installation, it only contains sample configurations.

Let's back up it:

```
mv /etc/salt/minion /etc/salt/minion.orig
```

Now let's insert the first configuration:

```
cat >> /etc/salt/minion << EOF
master: $MASTER_IP
EOF
```

This will "inform" the minion about the IP of the master. You can use IP addresses or names if it's possible for you.

This is how the minion file looks in my case:

```
cat /etc/salt/minion
---
master: 10.135.0.2
```

Restart the minion:

```
systemctl restart salt-minion
```

It should look different for you depending on the IP address you used previously (`export MASTER_IP=...`).

On the master machine, we should configure the master interface, which is the interface the master uses to listen to minions.

Let's backup the configuration file:

```
mv /etc/salt/master /etc/salt/master.orig
```

Let's use `0.0.0.0` as an interface. This means that the master will listen on all interfaces.

```
cat >> /etc/salt/master << EOF
interface: 0.0.0.0
EOF
```

Then restart the master:

```
systemctl restart salt-master
```

Salt master and minion identities

Before starting any exchange between the master and the minion, the master should be sure about the minion's identity. This will avoid minions to connect to the wrong masters on the same network, it will also avoid any man-in-the-middle attack.

The master has a public and a private key. To see them, use the following command on the master node:

```
salt-key -F master
```

Copy the pub key (that you can also get using `salt-key -F master | grep master.pub | awk '{ print $2 }'`) and then set it as a value to `master_firngerprint` configuration on the minion configuration file.

In my case, the pub ley is `ee:f5:4d:9d:51:74:7b:81:cd:6d:cc:c3:4b:99:3b:92:14:87:83:7a:29:a1:46:7c:19:a1:aa:46:01:0a:77:66`.

```
export
MASTER_KEY=ee:f5:4d:9d:51:74:7b:81:cd:6d:cc:c3:4b:99:3b:92:14:87:83:7a:29:a1:46:7
c:19:a1:aa:46:01:0a:77:66
cat >> /etc/salt/minion << EOF
master_finger: $MASTER_KEY
EOF
```

Then restart the minion `systemctl restart salt-minion`.

On the minion, local keys can be accessed using `salt-call --local key.finger`

Salt uses AES encryption to send and receive data between the master and the minion, this needs a system of keys available on both machines. Encrypted messages can not be tampered with, but before that, we must accept the minion key on the master.

If there are no networking or configuration issues, you will be able to see the Salt minion request to join the master. Type the following command on the master machine:

```
salt-key -L
```

The command above will list the keys known to the master and those that are not accepted yet. Get the minion ID from the output of the list above (for me it's the minion machine name `minion`), then execute:

```
export MINION_ID=minion01
salt-key --finger $MINION_ID
```

Compare this value to the value that is displayed when you ran `salt-call --local key.finger` on the minion. If the values are the same, we are sure that the minion we will accept to join the master is the same minion we run and not a hacker.

Now, let's accept the minion using:

```
salt-key -a minion
```

You can use `salt-key -A` to accept all minions in a single command. This is useful if you have a long list of minions to accept.

Let's test if the connection is established by executing the following command from the master:

```
salt $MINION_ID test.version
```

Note: The minion id is a globally unique identifier for a minion. It is usually the hostname, unless declared explicitly using `id: <id>` in the Salt minion config file. Remember that a host can run multiple minions. Since Salt uses detached ids it is possible to run multiple minions on the same machine but with different ids.

Salt modules

For this chapter, you will need two VMs with hostnames: master01 and minion01, respectively for the Salt master and the Salt minion running on CentOS 8.

Introduction

Modules are what make Salt scalable and extensible. They help us customize nearly every component in Salt.

Salt is written in Python. If we dive deep into its internals, we'll find that Salt modules are regular Python (or Cython) modules that are used to extend the various subsystems within Salt. We can see some examples here.

Every module contains a list of functions.

For example when we execute `salt '*' test.ping` we are using the function "ping" from module "test" which is an execution module (we are going to understand what are execution models later).

The "test" function is also represented in this way `salt.modules.test.ping()` like it is the case in Salt documentation.

The official documentation of Salt categorizes these modules into more than 20 categories:

- auth modules
- beacon modules
- cache modules
- cloud modules
- engine modules
- executors modules
- fileserver modules
- grains modules
- execution modules
- netapi modules
- output modules
- pillar modules
- proxy modules
- queue modules
- renderer modules
- returner modules
- roster modules
- runner modules
- sdb modules
- serializer modules
- state modules
- thorium modules
- token modules
- master tops modules
- wheel modules

Every module category contains a list of various modules with different functionalities and goals.

For example, Salt execution modules are Python module including different functions that runs on a Salt minion. Using the Salt communication model, the minion performs tasks and returns data to the master.

Executors modules (or simply executors) are used to modify functions behavior, do any pre-execution steps, or execute in a specific way like sudo executor.

State modules control the state system on the minion. Output modules (or outputters) are useful for displaying the results of state runs.

Returners modules allow the redirection of results data to targets other than the Salt master.

With practice, you will get used to all of these modules, or at least the most common ones. Defining what each module is doing exactly, will not help us at this stage, so let's move to hands-on practice.

In the next parts of this course, we are going to focus on 2 main modules:

- Execution modules
- State modules

Salt execution modules

Simply put, Salt execution modules are Python modules with functions that run on a minion. These functions called by the `salt` command.

There are hundreds of Salt execution modules. Let's discover some of them. As a first example, we are going to use the cmdmod. The cmdmod is helpful if you want to execute shell commands on the Salt minion.

Let's test some commands. Go to the master node and execute the following commands:

```
salt '*' cmd.run 'df -kh'
salt '*' cmd.run 'ls -lrth /var/log'
```

As you can see, from the master, we can use the cmdmod module to run any command on all minion machines (`'*'`). Even if we are going to understand targeting later, let's now understand that the `*` tells the master node to target any minion node that it has control over. In order to limit access to a single VM, we're simply going to use the minion id. In our case, it is "minion01".

```
salt minion01 cmd.run 'echo "ok" > /tmp/test.txt'
```

In this case, we ran `echo "this is a test" > /tmp/test.txt` on a single VM (minion01).

Using the same module, let's check if the file is already there:

```
salt minion01 cmd.run 'cat /tmp/test.txt'
```

You should see:

```
minion01:
    ok
```

We can run more advanced commands such as the following one where we use environment variables:

```
salt minion01 cmd.run 'echo "$VAR" > /tmp/test.txt' env='{"VAR": "ok"}'
salt minion01 cmd.run 'cat /tmp/test.txt'
```

A more detailed returned data can be obtained using `run_all`:

```
salt minion01 cmd.run_all 'echo "$VAR" > /tmp/test.txt' env='{"VAR": "ok"}'
```

It is also possible to choose which user is executing the command (whether you are using `run` or `run_all`):

```
salt minion01 cmd.run_all 'echo "$VAR" > /tmp/test.txt' env='{"VAR": "ok"}'
runas=root
```

Remember that errors can be fatal. The `cmdmod` is useful but it should be used very carefully as you are accessing all of your infrastructure VMs using a remote shell session. This is why using other modules instead of having direct access to the shell is more secure. For example, instead of running `salt minion01 cmd.run 'cat cat /proc/cpuinfo'`, you can run:

```
salt minion01 status.cpuinfo
```

Thanks to the status module, we can run different other functions like `all_status`, `cpuinfo`, `cpustats`, `custom`, `diskstats`, `diskusage`, `loadavg`, `master`, `meminfo`, `netdev`, `netstats`, `nproc`, `pid`, `ping_master`, `procs`, `proxy_reconnect`, `time_`, `uptime`, `version`, `vmstats` and `w`.

Let's try some of them:

```
salt minion01 status.uptime
salt minion01 status.cpustats
salt minion01 status.w
salt minion01 status.meminfo
salt minion01 status.diskusage
salt minion01 status.netstats
```

These are some useful commands that help us access stats about the system health including users, memory, disk, CPU and network.

Some other execution modules are even more helpful in other contexts. For example, if you are looking for more details network stats, it's better to use the network module:

```
salt minion01 network.active_tcp
salt minion01 network.default_route
salt minion01 network.netstat
salt minion01 network.routes
```

If you want to check how each function works, you should examine the documentation because there are always possible customizations. For example:

```
salt minion01 network.ping google.com
salt minion01 network.ping google.com return_boolean=True
salt minion01 network.ping google.com timeout=5
```

There are also different ways to do the same thing depending on your use case:

```
salt minion01 network.connect google.com 80
salt minion01 network.connect google.com 80 timeout=3
salt minion01 network.connect google.com 80 timeout=3 family=ipv4
salt minion01 network.connect google.com port=80 proto=tcp timeout=3
```

Asynchronous execution

When you execute a command or apply a state on/to the minion, you do not need to wait for its entire execution to get the terminal back. It is possible to use the async mode. This is how to do it:

```
salt '<minions>' <module>.<function> --async
```

Let's see an example of using the execution module in the async mode:

```
salt minion01 test.version --async
```

After executing this command, Salt will print the job id on the screen.

```
Executed command with job ID: <id>
```

Now, to get the execution result, we can execute the following command:

```
salt-run jobs.lookup_jid <job_id>
```

You can also list recent jobs using the command:

```
salt-run jobs.list_jobs
```

Then like we did before, use the job id (jid) and print the result `salt-run jobs.lookup_jid <job_id>`.

If a job is longer than just printing Salt version on the minion, you can see its id using the following command:

```
salt-run jobs.active
```

You can combine the execution and getting the job id in a single line:

```
salt minion01 pkg.install vim --async && salt-run jobs.active
```

Creating your custom modules

`pkg`, `network`, `cmd`, and `status` are called Salt modules (Salt execution modules).

We only used some of these modules, but as said, there are hundreds of other modules.. but what if you need to do something that is not supported yet by these modules?

Salt is Open Source, you can contribute to the project hosted on Github, and you can also create your own modules. You can also check the source code of a module before using it. If you plan to develop a new module, you may base your development on an existing one.

A Salt execution module is a Python or Cython module placed in a directory called `_modules/` at the root of the Salt fileserver. `_modules/` directory would be located in `/srv/salt/_modules` on most systems by default.

Creating custom modules is easy; you can read about it in [the official documentation](https://docs.saltstack.com/en/latest/ref/modules/index.html.

Let's create a simple module that will echo "ok".

```
mkdir -p /srv/salt/_modules/
```

Add this simple Python function to `/srv/salt/_modules/echo.py`:

```
cat >> /srv/salt/_modules/echo.py << EOF
def ok():
    return "ok"
EOF
```

Now before executing this new module, we need to sync the minion:

```
salt minion01 saltutil.sync_modules
```

To test the module, simply run:

```
salt minion01 echo.ok
```

You should be able to see:

```
minion01:
    ok
```

This module is just an example but to write complete Salt modules, there's a special way recommended by Salt that we can use. Even if the module we are going to write is absolutely not useful, it summarizes how to write a module. We are going to create a module that connects to a joke API, grab some random joke and send it back to the master. The module will of course run on the minion, results are sent back to the master.

It's recommended that you start your module with this useful information:

```
# -*- coding: utf-8 -*-
"""
:maintainer:    Aymen El Amri <hello@aymenelamri.com>
:maturity:      new
:depends:       python
:platform:      all
"""
```

Next, we will import `absolute_import` to provide backward compatibility with Python 2. This is recommended and as Python 2 is no longer supported it's no longer useful in most cases. We are also going to import the logging module and initialize a logger.

```
from __future__ import absolute_import
import logging

logger = logging.getLogger(__name__)
```

Salt allows us to choose a name for the module:

```
__virtualname__ = 'tell_me_a_joke'
```

The `__virtualname__` defines a custom name for the module. If we don't use the `__virtualname__` definition, the module name will be the module file name without the ".py".

Next, we are going to implement the `__virtual_` function:

```
def __virtual__(**kwargs):
    """
    When Python requests library is not installed, we don't need to run the
module and we will return an error.
    """
    try:
        import requests
        return __virtualname__
    except ImportError:
        return False, 'Error: Install Python requests package.'
```

The `__virtual__` function will either return the module's virtual name and continue loading the rest of the module or return False with an error. This function will always run first to check dependencies, system state, or any execution requirement.

We are also going to implement a private function. The private function is a regular Python function except that Salt doesn't have access to execute it. A private function always starts with an underscore `_`.

```
def _private(**kwargs):
    import requests
    request = requests.get('https://v2.jokeapi.dev/joke/Programming?type=single')
    joke = request.json()["joke"]
    ret = {"joke": joke}
    return ret
```

Now let's create another function that we can call when loading the module:

```
def get():
    try:
        logger.info("Sending the joke!")
        ret = _private()
        return ret
    except Exception as e:
        return fallback(e)

def fallback(e):
    return 'Error: %s' % e
```

Let's put everything together: