SaltStack Certified Engineer

Chapter 0: Welcome!

Welcome to the SaltStack Certified Engineer (SSCE) prep course from Linux Academy! In this course, we’re going to work with real-world scenarios to learn the core concepts of Salt and all the little details you need to pass the SSCE exam.

Currently, the SSCE is an 80-question, multiple-choice exam. At the time of writing this, the SSCE is still fairly new, and SaltStack still requires some official training to take it (either through Salt’s official courses or through SaltConf). That said, sometimes it’s best to take your time and learn and practice new concepts at your own pace. This course aims to fully prepare you for the SSCE without the need to cram for information.

This course covers all the topics you need to know to pass the SSCE, as well as some additional Salt features and functionality that will take your Salt skills to the next level. By the end of this course, you’ll be able to use Salt for event-driven infrastructure, cloud platform management, and even on servers that don’t have Salt installed at all.

The topics covered in this course include:

- Installation, configuration, and security concerns
- The remote execution system and its inner workings
- The state system and its relationship with remote execution modules
- Events, runners, and using Salt for event-driven infrastructure
- Using Salt SSH to manage servers that do not (or cannot) have Salt installed
- Using Salt Cloud to provision servers on cloud and virtualization platforms

You can do all of this on our three provided Cloud Servers at no cost to you. Of course, you can also use your own homelab if you prefer, but keep in mind that the Salt Cloud lessons will require the master to be accessed from an external source, namely AWS. Because of this, Vagrant (and Vagrant-like) environments may not be ideal. That said, it’s a quick and easy process to set up a Salt master, so if you’d rather use Vagrant or a similar setup for the majority of the course, you can always switch to one of our Cloud Servers for the Salt Cloud lessons.

This course also contains a variety of hands-on learning activities, end-of-chapter quizzes, and flash cards to test your skills and solidify concepts like important execution and state modules. If any of these terms are unfamiliar to you, don’t worry! They’ll all make sense soon.

Finally, this course can be taken in a variety of ways. You can use this book alongside our Cloud Servers (or virtual environment) and Learning Activities without ever watching a video (although I recommend you do). Or if you’d prefer to just watch the videos and follow along, you don’t even have to read the rest of this book. Whichever way you want to learn, I’ve got you covered.

This course also includes a workbook of important points and tutorials. You’ll notice I always have it open in the videos. If you’re the type of person who likes to watch a video or read a chapter and then perform tasks, this workbook can be used as a reference for completing the tasks. You can use it as much or as little as you like. It’s up to you and your learning style.

In creating this course, I really wanted to make something that works for all our students: those who thrive on video tutorials, those who (like me) prefer reading all the little details before ever touching a server, and those who learn best by completing hands-on tasks. I hope you find that this course works for your learning style, whatever it may be.

Good luck, and happy learning!

End State Goals

In this course, we’re going to explore various features of Salt using real-world scenarios. To do this, we’re going to be acting as members of the systems/DevOps team at a fictitious software company:

IdunnaSoft is a small company focused on building project management tools. Recently, their kanban-like product, Eski, has had an uptick in popularity, and the systems team, which you are a part of, has been exploring additional configuration management options to better handle Eski’s infrastructure. After much deliberation, your team has chosen Salt due to its extensibility, robust list of features, and commitment to open source. An inventory of the current infrastructure has since been performed, and the various tasks needed to Salt the infrastructure have been divided between the department’s DevOps engineers. This effort is being referred to internally as Project Golden Apple.

As part of the team-wide effort to prepare for the move to Salt, you have been given the following tasks:

Prepare a Test Environment

Eski is currently hosted entirely on Ubuntu 16.04 servers. However, additional products released by IdunnaSoft use CentOS 7 machines. As such, a test environment that uses both Ubuntu 16.04 and CentOS 7 machines is needed. Your team has requested that this environment contain an Ubuntu 16.04 Salt master and two Salt minions, one using Ubuntu 16.04 and one using CentOS 7. The Salt master should also be a minion itself.

Build Formulas

The systems team has divvied up the parts of the infrastructure that will need formulas and assigned them to different members of the team. You have been tasked with creating two formulas: an Apache formula and a MySQL formula. Both must work across distributions and both must be set up to work with Eksi and the company website. For Apache, this means two virtual hosts files. For MySQL, databases will also need to be set up.

Protect Salt-Managed Files

Using the events system — specifically beacons and reactions — ensure that if any changes are made to any Salt-managed configuration files outside of Salt itself, Salt registers these changes and reverts the file.

Set Up Orchestration

Finally, IdunnaSoft is exploring cloud options and the ability to use Salt not just to enforce configurations on minions, but to fully bring up and provision servers. As such, your team needs to create an orchestrate state that deploys a full LAMP stack, from server creation to configuration provisioning. This should be done with a single command.

By the end of this course, we will have completed these tasks and more to bring Project Golden Apple to fruition and, ideally, use the information we have learned to pass the SSCE.

Chapter 1: Introduction

Salt Overview

Salt, or SaltStack, is a remote execution and configuration management framework created by Thomas Hatch and released in 2011. It was initially released as a remote execution framework only but has since been expanded to contain configuration management, orchestration systems, and more.

At its core, Salt uses a high-speed data bus, ZeroMQ, to establish an encrypted channel between master and minion servers, which can receive and run commands simultaneously. This works because the Salt minions run the commands instead of the master. The master simply feeds the information to the minions.

To do this, the Salt master provides a port (4505) to which the minions bind and wait for commands from the master. After the master publishes the commands, the targeted minions execute the event, then return information about the commands run to the master on port 4506. This is called the “pub/sub” model.

Salt’s commands work across multiple operating systems and distributions, including Linux, MacOS, and Windows. Linux tends to be the most popular use case, so that’s what we’ll focus on in this course. However, the concepts are the same for all operating systems.

Salt Components

Salt is highly extensible and contains over twenty pluggable subsystems. Two of these systems are remote execution and configuration management frameworks. Others include an authentication system, the file server, and the return formatter for normalizing data structures in a Salt output. Although these all contain default behaviors, there is no one “right” way to use these systems.

Let’s take a look at some of the more common subsystems, all of which we will use in the course. Think of this as a terminology rundown before we get into the meat of the course. Don’t worry if you can’t remember it all right now; just familiarize yourself with the terms.

Master

The central server that sends out commands to its subscribed minion servers.

There can be more than one.

Minion

The managed system that takes orders from the master server. There can be more than one.

Execution Modules

Used in remote execution, execution modules provide ad-hoc commands that are run when called from the command line. This is ideal for commands that only need to be run once, for gathering real-time information about the minions, or for performing any task that cannot wait for a state or formula to be crafted.

States and Formulas

Part of the configuration management subsystem, formulas are end-state and “infrastructure as code” configurations that are written in YAML by default. A single task in a formula — such as installing a package — is called a state. A formula is a collection of states that configures one full part of the system (such as Apache). States, or entire formulas, are run in a process called highstating.

Grains

Grains provide information about the underlying system, such as kernel architecture and operating system family. These are made available to other Salt components so they can be used within execution modules or states.

Pillar

The pillar lets you define secure data that is supplied to minions by the master. The pillar is ideal for storing information like configuration parameters and passwords.

Runners

Assistive tasks that are run on the master, such as checking minion connection status, viewing job information, and making web calls.

Returners

Returners send returned Salt data to another system, such as a database, monitoring system, or even a messaging application (like Slack).

Beacons and Reactions

Use beacons to monitor events from non-Salt processes using Salt, then use reactions to perform actions triggered by these events. For example, reset a changed configuration file.

Salt SSH

Use Salt to perform remote execution tasks via SSH on systems without Salt.

Salt Cloud

Use Salt as a provisioning tool to add and manage servers on certain providers or cloud hosts and immediately bring them under Salt’s control on systems that do not have Salt installed.

Chapter 2: Salt Installation and Configuration

Package Install

When installing Salt on Linux using a typical package install, we can choose to install from one of three SaltStack-provided repositories:

- The latest release repository will always install the latest release of Salt, whether it’s a major release or a point release.
- The major version repository allows you to install the latest release, but any updates it receives will only be for that major version.
- The minor release repository installs one specific release of Salt, and updating will ***not*** change the release that is installed.

These distributions all follow the same basic installation steps:

1. Add the repository key and repository.
2. Update the cache for the packing tool.
3. Install the Salt components you want:
   – salt-master
   – salt-minion
   – salt-ssh
   – salt-syndic
   – salt-cloud
   – salt-api

To see how this works, let’s walk through installing Salt on a Debian system.

Install the Latest Salt Release on Debian 9

1. Set the hostname to `salt` and update `/etc/hosts` to use the hostname `salt`:

```
 $ sudo hostnamectl set-hostname salt
$ sudo $EDITOR /etc/hosts
```

Note that by default, Salt looks for a master named `salt`.

2. Import the repository key:

```
  $ wget -O -
https://repo.saltstack.com/apt/debian/9/amd64/archive/201
GPG-KEY.pub | sudo apt-key add -
```

3. Add the `SaltStack` repository to your repository list:

```
$ sudo $EDITOR /etc/apt/sources.list.d/saltstack.list
```

```
deb
http://repo.saltstack.com/apt/debian/9/amd64/archive/2018
stretch main
```

4. Run an update:

```
$ sudo apt-get update
```

5. Install Salt components:

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

Bootstrap Salt

Bootstrapping Salt is a simple process that uses the same series of commands across all available distros. The bootstrap script is available on the following platforms:

- Debian 7, 8:
  - Mint 1
  - Kali Linux 1
- Red Hat 5, 6, 7:
  - Amazon Linux 2012.09, 2013.03, 2013.09, 2014.03, 2014.09
  - CentOS 5, 6, 7
  - Fedora 17, 18, 19, 20, 21, 22
  - Oracle Linux 5, 6, 7
  - Scientific Linux 5, 6, 7
- Ubuntu 10.*, 11.*, 12.*, 13.*, 14.*, 15.*, 16.*:
  - Linaro 12.04
  - Mint 13, 14, 15, 16, 17
  - Trisquel 6
- Arch Linux
- Gentoo
- OpenBSD
- FreeBSD 9, 10, 11
- SmartOS

To bootstrap Salt, we first need to prepare our servers by setting our hostnames and ensuring that any servers acting as minions can access the master via the hostname `salt`.

Next, we need to pull down the installation script using a file retrieval application like `curl` or `wget`.

Then, we need to run the installation script using flags to denote what we want to install. Generally, the installation script looks like this:

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

This script is followed by a flag or series of flags that define the configuration options we want. The most commonly used flags are:

- `-M` — Install `salt-master`
- `-N` — Do ***not*** install `salt-minion`
- `-P` — Allow `pip`-based installations
- `-S` — Install `salt-syndic`
- `-L` — Install `salt-cloud` and the required library (`python-libcloud`)

You can view a full list of flags by running:

```
$ sudo sh bootstrap-salt.sh -h
```

And that's all it takes to bootstrap Salt!

Now let's set up the system we're going to use throughout the bulk of this course.

Bootstrapping the Demo Environment

We'll be using three Linux Academy cloud servers:

- Master: Ubuntu 16.04
- Minion 1: Ubuntu 16.04
- Minion 2: CentOS 7

Master Setup

We'll be using the Ubuntu 16.04 server as our Salt master. Let's get started.

1. Update the hostname and the `/etc/hosts` file so that the server answers to the hostname `salt`:

```
  $ sudo hostnamectl set-hostname salt
$ sudo $EDITOR /etc/hosts
```

```
127.0.0.1 localhost salt
```

2. Download the installation script (I used `curl`):

```
 $ curl -L https://bootstrap.saltstack.com -o bootstrap-salt.sh
```

3. Run the installer. Remember, we want the Salt master to be a minion against itself:

```
 $ sudo sh bootstrap-salt.sh -P -M
```

Minion 1 Setup

Our first minion, which we'll name `minion1`, will be our Ubuntu 16.04 server.

1. Update the hostname and the `/etc/hosts` file so the hostname resolves to `minion1`. We also want to ensure that the hostname `salt` resolves to the local IP address of our master.

```
 $ sudo hostnamectl set-hostname minion1
$ sudo $EDITOR /etc/hosts

<MASTER_PRIVATE_IP> salt
```

2. Download the installation script:

```
 $ curl -L https://bootstrap.saltstack.com -o bootstrap-salt.sh
```

3. Run the installer. Remember, this is a ***minion*** server:

```
 $ sudo sh bootstrap-salt.sh -P
```

Minion 2 Setup

1. Update the hostname and the `/etc/hosts` file so the hostname resolves to `minion2`. We also want to ensure that the hostname `salt` resolves to the local IP address of our master:

```
 $ sudo hostnamectl set-hostname minion2
$ sudo $EDITOR /etc/hosts

<MASTER_PRIVATE_IP> salt
```

2. Download the installation script:

```
 $ curl -L https://bootstrap.saltstack.com -o bootstrap-salt.sh
```

3. Run the installer. Remember, this is a ***minion*** server:

```
$ sudo sh bootstrap-salt.sh -P
```

Now we just need to finish the handshake process between our master and minions so that the connection between the two is established and secure. We'll cover this in the next section.

LEARNING ACTIVITY: INSTALLING SALT

Time to get hands-on! Log in to Linux Academy and complete the Installing Salt Learning Activity.

Key Management

Now that we have Salt installed on our environment, we need to finish the key exchange so that our Salt master can send commands to our minions. But first, let's talk about how this works.

Communication between the master and minions is done through AES encryption; the minions initiate this connection by searching the network for a system named `salt`. The minion then initiates a handshake and sends its public key to the master server. After this, an administrator must accept the keys on the master (although this process can be automated).

When commands are sent to the minions, they are sent through an encrypted channel. The key used for this channel is different for each minion and session.

So how do we go about accepting and managing these keys? This is done through the use of the `salt-key` command. With `salt-key`, we can view and manage our keys with the following flags:

- `-l <arg>, --list=<arg>`: List keys that match the given argument. Arguments include:
  - `pre`, `un`, and `unaccepted` for unaccepted and/or unsigned keys
  - `acc or accepted` for accepted keys
  - `rej` or `rejected` for rejected keys
  - `all` for all keys
- `-L, --list-all`: List all keys
- `-a <keyname>, --accept=<keyname>`: Accept defined key
- `-A, --accept-all`: Accept all pending keys

- `-r <keyname>, --reject=<keyname>`: Reject defined key
- `-R, --reject-all`: Reject all pending keys
- `--include-all`: Accept/reject all keys, including non-pending keys
- `-d <keyname>, --delete=<keyname>`: Delete defined key
- `-D, --delete-all`: Delete all keys
- `-f <keyname>, --finger=<keyname>`: Print defined key fingerprint
- `-F, --finger-all`: Print all key fingerprints
- `-p <keyname>, --print=<keyname>`: Print the defined public key
- `-P, --print-all`: Print all public keys

You can always view a list of commands with:

```
$ sudo salt-key -h
```

Now let's check the legitimacy of our keys and then accept them so we can start working with Salt.

Verify and Accept the Salt Keys

On the Master Server

1. First, let's take a look at what Salt keys we have available:

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

   Here we can see that we have unaccepted keys for `salt`, `minion1`, and `minion2`. Now, it might seem like we should go ahead and run `sudo salt-key -A` to accept all of them, but we want to take some extra steps to stay secure. So, the first thing we want to do is validate our key identity.
2. Let's view the master fingerprint for the server:

   ```
   $ sudo salt-key -f master
   ```

   This outputs our `.pem` and `.pub` files, which contain information about our master keys. We want to copy the `master.pub` fingerprint and add it to our minions' configuration files.

On Minion 1

1. With the `master.pub` fingerprint still copied, open up `/etc/salt/minion` on `minion1`:

   ```
   $ sudo $EDITOR /etc/salt/minion
   ```
2. We want to add the fingerprint to the `master_finger` line. This ensures that our Salt minion will ***only*** connect to the appropriate master

and prevent man-in-the-middle attacks:

```
  master_finger:
'93:c1:0f:d2:55:76:0d:4b:8a:37:26:69:58:44:bd:04:b5:de:89
```

Be sure to uncomment the line.

Save and exit the file.

3. Restart the `salt-minion` service:

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

Now that our Salt minions know which master they should use, we need to ensure that the correct minions are the ones receiving instructions. Notice that on the master, our unaccepted key fingerprints are also viewable using the fingerprint output from earlier. Let’s verify that these match.

4. View the fingerprint for `minion1`:

```
  $ sudo salt-call --local key.finger
```

5. Confirm that this fingerprint matches the one on the master.
6. Switch to `minion2`.

On Minion 2

Now we need to repeat the above process on our second minion.

1. Uncomment and add the `master.pub` fingerprint to the `master_finger` line in `/etc/salt/minion`:

```
 $ sudo $EDITOR /etc/salt/minion

master_finger:
'93:c1:0f:d2:55:76:0d:4b:8a:37:26:69:58:44:bd:04:b5:de:89
```

Save and exit the file.

2. Restart the `salt-minion` service:

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

3. View the fingerprint for `minion2`:

```
  $ sudo salt-call --local key.finger
```

4. Confirm that this fingerprint matches the one on the master.
5. Return to the master.

Back on the Master

Now that we've verified our minions, we want to perform the same process on our master. We need to do this before we can accept our keys.

1. Copy the `master.pub` fingerprint and add it to the `master_finger` line in `/etc/salt/minion`:

```
$ sudo $EDITOR /etc/salt/minion
```

```
master_finger:
'93:c1:0f:d2:55:76:0d:4b:8a:37:26:69:58:44:bd:04:b5:de:89
```

Save and exit.

2. Restart the `salt-minion` service:

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

3. Confirm that the minion fingerprint matches:

```
$ sudo salt-call --local key.finger
```

4. Accept all keys:

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

Now that our working environment is up and running, we can start using Salt. Before we get started, let's talk about some of the different configurations and setups we can use when configuring our Salt environment.

Master Configuration

Knowing the various options available in both the master and minion configuration files is an integral part of the SSCE exam. In this lesson, we're going to open up our `/etc/salt/master` file, review the different configuration options for our master server, and perform some common setup tasks for our new Salt environment.

Note: There are ***many*** options available for this file, and most contain commented instructions within the `/etc/salt/master` file. I could list all of these options, but the best documentation is the file itself. Let's work with some common configuration options and review some common-sense approaches to configuring our master.

If It Exists, We Can Change It

One of the nice things about Salt is that we can configure almost every component of it. In these past few lessons, we've learned that Salt uses ports

4505 and 4506. The configuration file for the master is located at `/etc/salt/master`, and Salt looks for hosts using the name `salt`. However, depending on the Salt setup, none of this might actually be true. Why? Because we can change all of it within our configuration.

Within our master configuration file, we can set where Salt binds, we can enable IPv6, and we can manage the default directories for every part of Salt (configuration files, custom modules, public key storage directories, etc.).

The master configuration file lets us set our file server. We don't have to store the file on our master; we can use something like a Git repository or AWS. The master configuration file is where we can set up Salt components like Salt SSH as well as set the locations for our states, formulas, and pillar data (which we'll discuss later).

Since Salt allows you to use their documentation during the SSCE exam, there's no need to memorize the hundreds of lines and options in the configuration file. Instead, I suggest familiarizing yourself with the general options it covers (as we did above) and then referencing the file or documentation when you need to during the exam. Just search to find the exact configuration attribute you're looking for.

Now, let's go ahead and make some changes.

Set the File Roots

While we can update the master configuration file directly, the file itself is very long, clocking in at over 1,000 lines. If we take a look at the very first configuration option in our configuration file, we can see that the `master.d` directory is set to pull in any configurations saved with the `.conf` extension. Let's go ahead and use this setup for our configurations to keep things clean.

1. Navigate to the `/etc/salt` directory:

```
$ cd /etc/salt
```

2. Open up the `master` configuration file:

```
$ $EDITOR master
```

3. Search for the `file_roots` setting in your editor, and copy the existing configuration. The `file_roots` setting lets us define the location in which we store our states and formulas; we can also define all our

environments within their configuration setting.

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

4. Exit the file.
5. Navigate to the `master.d` directory, then create and open `file-roots.conf`:

```
$ cd master.d
$ sudo $EDITOR file-roots.conf
```

6. Add the configuration property:

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

7. Save and exit the file.
8. Restart the `salt-master` service:

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

Minion Configuration

Much like with our master configuration file, the minion configuration file allows us to configure various options related to our minion and our minions' connection to the master. Located at `/etc/salt/minion`, the minion config is a fairly long file with default settings that tend to work out of the box. However, we also have the option to alter almost all of these settings. Minion configuration files can also be stored under `minion.d` using the `.conf` extension.

The single most important line in the minion configuration file is the `master` setting; this is where we denote the hostname, IP address, or URL of our master server. By default, it looks for a master using the hostname `salt`, which we used when setting up our environment. We've also previously worked in this file during our Key Management lesson, when we updated the `master_finger` line to define the public key of our master in case more than one server uses the name `salt`.

Note that a large portion of this file is unused unless we're working with a masterless Salt minion. We also have options for setting file and pillar roots, defining a file server, and even storing custom modules.

Just like with our master configuration file, instead of trying to memorize all the options within this file, I instead suggest you review it a few times until you feel confident that you can search through the file quickly and efficiently during the exam.

Before we move on, let’s make one change on each of our minions. Since our minion configuration can also contain grain data — remember that grains are data about the server — we’re going to add a grain for each server denoting its role or roles.

Add Grain Information

On the Master

1. Navigate to `/etc/salt/minion.d`:

   ```
   $ cd /etc/salt/minion.d
   ```

2. Create and open a configuration file called `grains.conf`:

   ```
   $ sudo $EDITOR grains.conf
   ```

   Again, we want to define `roles` for our server. In this instance, we’re working on the master, so `master` is the role we’re going to assign.

3. Assign the role:

   ```
   grains:
     roles:
       - master
   ```

4. Save and exit the file.

5. Restart the `salt-minion` service:

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

On Minion 1

1. Navigate to `/etc/salt/minion.d`:

   ```
   $ cd /etc/salt/minion.d
   ```

2. Create and open `grains.conf`:

   ```
   $ sudo $EDITOR grains.conf
   ```

3. Since we know this minion is used in a test — or ***dev*** — environment, we want to give it the role `dev`. We also want to assign it the roles

`webserver` and `database`, since it will be taking on the roles of both:

```
grains:
  roles:
    - dev
    - webserver
    - database
```

Notice how a grain can have more than one value.

4. Restart the `salt-minion` service:

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

On Minion 2

1. Navigate to `/etc/salt/minion.d`:

```
$ cd /etc/salt/minion.d
```

2. Create and open `grains.conf`:

```
$ sudo $EDITOR grains.conf
```

3. As with our previous minion, we want to give it the role `dev`. We also want to assign it the roles `webserver` and `database`, since it will also be performing the roles of both:

```
grains:
  roles:
    - dev
    - webserver
    - database
```

4. Restart the `salt-minion` service:

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

Salt Mine

The last configuration option we're going to set up is our Salt mine. The Salt mine is a completely optional component of Salt that, like grains, queries for and stores arbitrary data on the master. The difference is that mine data is updated more frequently (every 60 minutes, although this can be changed), and we need to define what data is pulled into the mine. Mine data can also be saved within pillar; the setup is also exactly the same, except the file will be saved as an `sls` in your pillar roots.

The Salt mine enables a minion to retrieve data about ***other*** minions without needing to talk to anything but the master it is already keyed to. This data is gathered on each minion, then sent to the master, which stores it. Only the most recent mine data is kept. This means that if we're using a cloud solution with elastic IP addresses, only the most recent IP address will be used, and it will not keep a catalog of any prior addresses.

Although the Linux Academy cloud servers maintain the same private IP until we actually delete them, we're going to set up Salt mine to store data about our IP addresses. You may find this especially beneficial if you're following along on an environment you set up elsewhere.

Add Networking Data to the Mine

On the Master, Minion 1, and Minion 2

1. Navigate to `/etc/salt/minion.d`:

```
$ cd /etc/salt/minion.d
```

2. Create and open `mine.conf`:

```
$ sudo $EDITOR mine.conf
```

3. Set up `mine_functions` so that the mine queries data related to our available network IP addresses. We haven't covered execution modules yet, but the function name is fairly self-explanatory:

```
mine_functions:
  network.ip_addrs: []
```

If we want to narrow our responses, we can add extra parameters within the square brackets. For example, we could add the `interface: eth0` parameter to tell Salt to gather addresses only on the eth0 interface. This would look like:

```
mine_functions:
  network.ip_addrs:
    interface: eth0
```

4. Save and exit the file.

5. Restart the `salt-minion` service:

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

Secure Salt

As we wrap up the configuration section of this course, the last thing we want to do is consider any potential security concerns we may have when using Salt and what steps we should take to stay secure. We’ll discuss security in greater depth as we work through the course — like how we should always store sensitive data in the pillar and how we need to be aware that Salt has full root access to all of the minions — but for now, let’s consider a few basic security concerns.

The single most important thing to remember about Salt is that it has a root view of and root access to any server it manages. Even if we run Salt as non-root (which is possible), Salt will still be able to make major changes to all of our servers. This means we should restrict access to the master, as well as limit the general surface area of the system. For example, when logging in to our Salt master through a public IP address, using a Bastion host and locking down the IP range of SSH connections can prevent outsiders from trying to access our master. In addition, we want to make sure we have the basics covered: no root SSH login, secure SSH keys, strong user passwords that are rotated periodically according to company policy, and a stringent firewall.

We also need to consider how our formulas, states, and pillar data are being pulled into Salt. In this course, we’re going to be writing directly on the master because it makes sense for us; we’re learning, and no one will lose important company data if we make a mistake and accidentally overwrite a configuration file we shouldn’t. In actual practice, however, we want to keep our states, formulas, and pillar in a secure (and preferably private) version-control system like GitHub. We also want to make sure our states and formulas go through code review.

On the minion side, we want to limit any extraneous logins; in fact, we shouldn’t ever have to log in to these systems directly. After all, the whole point of Salt is that we don’t have to deal with the tedious parts of managing a server. Instead, consider the times and circumstances in which you normally would have to log in to these servers, and create beacons for them. That way, when something that may require your attention comes up, it’s sent straight to the Salt event bus for you to deal with. Even better, add a reaction to automatically fix the problem.

We can also make Salt restrict some of its own actions. Remember the dangers of `cmd.run`? Well, we can prevent Salt from using `cmd` at all. Within our minion configuration, we have the option to disable modules. This works exactly like any other configuration setting; we simply uncomment it, then supply a list of modules. For example, here’s how to disable `cmd` and `network`:

```
disable_modules:
  - cmdmod
  - network
```

The `sys` module is the only module that cannot be disabled. It is required for the minion to work.

We can also disable returners with the `disable_returners` option.

Finally, Salt recommends subscribing to their mailing lists to keep track of release information. The more up-to-date Salt is on your system, the more secure it will be:

- salt-users Mailing List
- salt-announce Mailing List

LEARNING ACTIVITY: CONFIGURATION AND KEY MANAGEMENT IN SALT

To get hands-on and practice your skills, log in to Linux Academy and complete the Configuration and Key Management in Salt Learning Activity.

Multi-Master Setup

In addition to the traditional single-master, multi-minion setup we created in the previous sections, Salt also offers alternative setup options, including a multi-master setup. This is exactly what it sounds like: a "normal" Salt environment but with multiple masters to distribute the minion load and create redundancy.

In the simplest setup of a multi-master environment, the masters all run "hot" — that is, they can all send commands to all minions. An alternative setup can also be configured in which the minion is provided a list of masters and connects to one, then only uses another if the first one fails. You can read more about this in the SaltStack documentation page Multi-Master-PKI Tutorial with Failover.

There are a few things we need to consider before setting up a multi-master environment:

- The masters need to share the same private key.
- The minions' configuration files need to be updated to use both masters.
    - This can be done through Salt via the existing master.
- The masters need to be able to share files among themselves.
    - Shared files:

        - Minion keys
        - `file_roots`
        - `pillar_roots`
    - Do not sync minion key files; this is a security risk.
        - Instead, use `salt-key` to accept the keys of all minions on all masters.
    - Use a file server backend like `gitfs` to store states, formulas, and pillar data.
- The configurations of both masters should be the same.

Let's go ahead and set up a multi-master configuration with a single minion. This is going to be outside of our end-state environment, and there's no need to keep the servers for the rest of the course.

Set Up a Multi-Master Environment

Before we begin, bring up a regular Salt master and a single minion. We're going to be using CentOS 7 for all servers. Refer to either the Package Install or Bootstrap Salt section if you need help. To make things easier, I suggest not installing a minion on the masters.

On the New Master

1. Set the hostname. We're going to use `salt2`:

```
 $ sudo hostnamectl set-hostname salt2
$ sudo $EDITOR /etc/hosts

127.0.0.1 localhost salt2
```

2. Install Salt:

```
 $ curl -L https://bootstrap.saltstack.com -o bootstrap-
salt.sh
$ sudo sh bootstrap-salt.sh -P -M -N
```

3. Stop the `salt-master` daemon:

```
  $ sudo systemctl stop salt-master
```

Before we continue setting up the new master, we need to make a few changes to the old one. Open up our original Salt master in another window or tab.

On the Old Master

1. We need to copy the `master.pem` and `master.pub` keys (located in

`/etc/salt/pki/master`) to our second master. This can only be done as the `root` user. Switch to `root`:

```
$ sudo su
```

2. Navigate to `/etc/salt/pki/master`:

```
# cd /etc/salt/pki/master
```

3. Copy the `master.pem` and `master.pub` keys to the new master. In this example, I use `scp`:

```
# scp master.p* user@172.31.30.57:
```

We use the `user` user and not `root` when accessing the new master because root login is disabled on cloud servers.

Switch back to the new master.

Back on the New Master

1. Switch to `root`:

```
$ sudo su
```

2. Navigate to the `/etc/salt/pki/master` directory and remove the current `master.pub` and `master.pem` keys:

```
# cd /etc/salt/pki/master
# rm master.p*
```

3. Move the new keys into this directory:

```
# mv ~user/master.p* .
```

4. Set the permissions:

```
# chmod 0400 master.pem
# chmod 0644 master.pub
```

5. Start the `salt-master` service:

```
# systemctl start salt-master
```

Leave `root`, then confirm that the keys are the same by running `sudo salt-key -F master` on both masters. You may notice that our new master still has no minions. This is because we have one last thing we need to change.

On the Minion

1. Add the hostname information for the new master to the `/etc/hosts` file:

```
$ sudo $EDITOR /etc/hosts

<SALT2_PRIVATE_IP> salt2
```

2. Open the `/etc/salt/minion` file:

```
$ cd /etc/salt
$ sudo $EDITOR minion
```

3. Find the `#master:` value and uncomment. We want to add ***both*** master hostnames here:

```
master:
  - salt
  - salt2
```

4. Save and exit the file.

5. Restart the `salt-minion` service:

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

Back on the Master

View and accept the minion's key:

```
$ sudo salt-key -l unacc
$ sudo salt-key -f minion1
$ sudo salt-key -a minion1
```

Normally at this point, we would go ahead and set up our file backend, but that's a task for another lesson.

Masterless Setup

Now we're going to take one more detour and learn how to set up a masterless Salt minion. The `salt-minion` setup is so powerful, it can use most of Salt's functionality on its own. This is ideal for testing various bits of your Salt infrastructure, from states to custom modules.

When running a masterless Salt minion, we can use `salt-call` to run Salt commands. This takes some configuration, but it's minimal. All we have to do is update the `file_client` line to `local` in our `/etc/salt/minion` configuration file. Additionally, any configurations you would normally use in your master config can be added to the standalone minion's configuration file.

Finally, it's important to note that the `salt-minion` daemon should ***not*** be enabled in a masterless setup. If it is, the daemon will attempt to find a master to connect to.

Now we're going to walk through setting up a masterless minion. Again, this has no bearing on our end-state example, and you don't need to keep the server when you're done. In this example, I'm using a CentOS 7 server.

Set Up a Standalone Minion

1. Install Salt:

```
 $ curl -L https://bootstrap.saltstack.com -o bootstrap-
salt.sh
$ sudo sh bootstrap-salt.sh -P
```

2. Turn off and disable the `salt-minion` daemon:

```
 $ sudo systemctl stop salt-minion
$ sudo systemctl disable salt-minion
```

3. Update the `/etc/salt/minion` configuration and set the `file_client` to `local`:

```
 $ sudo $EDITOR /etc/salt/minion

file_client: local
```

4. Save and exit the file.

Now we can test our masterless minion to see if any commands work. We've used the `network.ip_addrs` command before, so let's use that:

```
$ sudo salt-call network.ip_addrs
```

LEARNING ACTIVITY: SETTING UP SALT MULTI-MASTER

To get hands-on and practice your skills, log in to Linux Academy and complete the Setting Up Salt Multi-Master Learning Activity.

Chapter 3: Remote Execution

Execution Modules

Now that we’ve finished installing and configuring Salt, we can get into the real meat of this course and actually ***use*** Salt. While Salt has various practical applications, its core function is built off its first intended purpose: the remote execution framework.

Salt’s remote execution framework allows us to run commands on multiple machines simultaneously (although we can stagger jobs; we’ll discuss this in a later lesson). It solves a common problem in IT: having to either run a command on multiple servers by hand or by writing scripts to perform the task.

The remote execution framework uses functions known as ***execution modules***. Execution modules are distro-agnostic[^1] and look like this:

```
module.function
```

For example, let’s consider the `pkg`, or ***package***, module. This module allows us to manage which packages we have on our server using a number of provided functions that perform specific tasks. One such function is the `install` function. As you can probably guess, this function allows us to install packages. To use this module function, we would call it with `pkg.install`.

So how would we use this on the command line? Let’s demonstrate by installing something. Let’s try to add `htop`, the process viewer.

```
$ sudo salt '*' pkg.install htop
```

But wait! What are all those ***other*** parts of the command? Let’s break it down:

```
sudo salt <target> <module>.<function> <parameters>
```

In this instance, `salt` is simply the command we use to work with Salt; we will use this command frequently throughout this course. Next, the target (`'*'`) is which server or servers we want to run the command on. Here, we’re using a wildcard to denote that we’ll be running our command on all servers. Next, we call the module and function as we discussed above, then follow it with our parameters. For our example, we provide our package’s name, `htop`. In other instances, we may have more parameters to add or none at all.

Now, what happens when we run this command? First, the user is authenticated and the command is placed in the job queue. Then, following the pub/sub model, the master publishes the command to the bind address, and the waiting minion receives the command. Once received, the execution module is run. Finally, the results are formatted by the outputter, returned to the event bus, and sent back to the master.

Before we start running our execution modules for real, let's talk more about targeting.

[^1] Not all execution modules are available for all distributions. In addition, Windows uses its own set of modules, usually prepended with `win_`.

Targeting

Now that we know how to use our execution modules to direct commands across multiple servers, we need to consider how we determine which servers we want to run our commands on. Previously, we used the `*` wildcard to work with ***all*** of our minions.

Salt provides several ways to target our minions. We can call them via:

- Minion name
- Grains
- Node group
- Pillar data
- Subnet or IP address
- SECO range (for those using SECO tools)

We can also define how many minions we want running the commands at once.

Let's take a look at some examples.

Minion Name

Targeting via minion name is arguably the easiest way to target a minion. It's certainly the clearest — all you have to do is call the minion by name on the command line. However, we can also get more advanced here by using globbing, regular expressions, or lists to call multiple minions at once.

We've already seen globbing in action once. When we use our `*` wildcard to call all minions, we're using globbing. We could also get more specific. Say we only want to target our minions with the prefix `minion`. We could use:

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

Or:

```
$ sudo salt 'minion?' test.ping
```

Or even:

```
$ sudo salt 'minion[1-2]' test.ping
```

Alternatively, we can use regex, which shares many characteristics with general shell globbing. In fact, all of the above examples could also be considered regular expressions.

Finally, we can call minions via a list using the `-L` flag:

```
$ sudo salt -L 'salt,minion2' test.ping
```

Grains

Grains can also be used to target minions. For example, say there's an update being sent out to all Ubuntu servers, and you need to install it. You only want this update run on Ubuntu-based minions, so you can use the `-G` flag and run:

```
$ sudo salt -G 'os:Ubuntu' pkg.upgrade
```

Or what about the `roles` grain we added earlier? Let's target only our web servers:

```
$ sudo salt -G 'roles:webserver' test.ping
```

Node Group

Unlike our previous examples, using node groups requires some up-front configuration. Node groups need to be added to the `master` config (or to the `master.d` directory) and can use existing targeting practices to group multiple servers together. Since our test environment is only three servers, we're fairly limited here, but let's say we wanted to have a group of only CentOS 7 dev servers. Within our master configuration, we would set:

```
nodegroups:
  centos-dev: 'G@os:CentOS and G@roles:dev'
```

The `G@` denotes that we're using grains; these use the same letters as their respective flags. So if we wanted to provide a list instead, we would use `L@` before listing out our minions:

```
nodegroups:
  list-group: 'L@salt,minion1'
```

To use these node groups in practice, we would need to use the `-N` flag with our `salt` command:

```
$ sudo salt -N centos-dev test.ping
```

Pillar Data

Targeting in pillar is remarkably similar to targeting with grains. You provide a key and its expected value, and the commands will only run on the minions that match that pillar data. Pillar is defined with the `-I` flag:

```
$ sudo salt -I 'website:example' test.ping
```

Subnet/IP

Much like targeting via minion name, we can also choose which minions to work with via IP address. For example, if the private IP of my Salt master is `172.31.126.12`, I can target a command on it using:

```
$ sudo salt -S 172.31.126.12 test.ping
```

Additionally, we can target whole subnets:

```
$ sudo salt -S 10.0.1.0/24 test.ping
```

Compound Targeting

We can also combine the above targeting method with globbing by using ***compound targeting***. Let’s target all CentOS servers again, this time with the globbed minion names of `minion*`:

```
$ sudo salt -C 'G@os:CentOS and minion*' test.ping
```

If you think this looks a lot like defining a node group, you’re right! Just like when adding a group, we use the same letter signifiers as the command flag, followed by the `@` symbol. Note that the `-C` flag must be used when calling compound targets.

Batching

Finally, if we have a large infrastructure and are concerned about running tens, hundreds, or thousands of commands at once, we can use the `-b` flag to denote how many servers should run the command simultaneously in a process called ***batching***. We only have three servers in our practice environment, so let's test with a batch size of `2`:

```
$ sudo salt '*' -b 2 test.ping
```

Here we can see that our output provides more information than we usually see. We're given information about the job ID on each server, and we can see that Salt ran the command on two servers initially, then on the final server alone:

```
Executing run on [u'minion2', u'salt']

jid:
    20180424145214769610
retcode:
    0
salt:
    True
jid:
    20180424145214769610
minion2:
    True
retcode:
    0

Executing run on [u'minion1']

jid:
    20180424145214930378
minion1:
    True
retcode:
    0
```

As a final note, SECO range is a very specific use case that you will not be tested on and will not be covered in this course. If you find you need to use SECO range, you can learn more about it in the official SaltStack documentation.

`salt-call`

Everything we’ve just covered regarding execution modules relates to a Salt master dictating commands to its minions. But what about instances where the master isn’t in play or where we’re logged on to a minion? For these instances, we can use `salt-call` to run execution modules on the minion itself.

We’ve used `salt-call` before to view a minion’s key fingerprint. If you’ve been following along with the videos and doing the learning activities, then you’ve used it at least three times already. Remember:

```
$ sudo salt-call --local key.finger
```

In fact, the command itself is excellent at describing how `salt-call` works. We first call the `salt-call` command, then denote that we’re using it locally by adding the `--local` flag (unless configured otherwise). Finally, `key.finger` includes our execution module and function, and we already know what it does — it prints out our minion’s key. If we wanted to use additional arguments, we would simply add them after the module and function as we would in a regular `salt` command.

The `salt-call` module is especially useful for testing. If you’re working on a new Salt execution module, you can run that module (or state, as we’ll address later) directly on the minion to gain access to more information about the changes that are happening, which is helpful for troubleshooting issues.

LEARNING ACTIVITY: TARGETING IN SALT

To get hands-on and practice your skills, log in to Linux Academy and complete the Targeting in Salt Learning Activity.

The `sys` Module

Now that we’ve discussed the basics of working with execution modules in Salt, let’s take a look at some of the commonly used modules that are available to us. Salt has hundreds of execution modules; the full list is available on the Saltstack website.

The first one we want to consider is the `sys` execution module. `sys` brings up information about the various functions available to a minion. Think of `sys` like a manual page or documentation repository for the available execution (and state) modules, but with a few more features.

Let’s use the `pkg` module as an example. We know that `install` is a function of this package. But what other functions does this module have available? To check this, we can use the `list_functions` function:

```
$ sudo salt 'salt' sys.list_functions pkg

salt:
    - pkg.add_repo_key
    - pkg.autoremove
    - pkg.available_version
    - pkg.del_repo
    - pkg.del_repo_key
    - pkg.expand_repo_def
    - pkg.file_dict
    - pkg.file_list
    - pkg.get_repo
    - pkg.get_repo_keys
    - pkg.get_selections
    - pkg.hold
    - pkg.info_installed
    - pkg.install
    - pkg.latest_version
    - pkg.list_pkgs
    - pkg.list_repo_pkgs
    - pkg.list_repos
    - pkg.list_upgrades
    - pkg.mod_repo
    - pkg.owner
    - pkg.purge
    - pkg.refresh_db
    - pkg.remove
    - pkg.set_selections
    - pkg.unhold
    - pkg.upgrade
    - pkg.upgrade_available
    - pkg.version
    - pkg.version_cmp
```

We could even do this with sys itself:

```
$ sudo salt 'salt' sys.list_functions sys

salt:
    - sys.argspec
    - sys.doc
    - sys.list_functions
    - sys.list_modules
    - sys.list_renderers
    - sys.list_returner_functions
    - sys.list_returners
    - sys.list_runner_functions
    - sys.list_runners
    - sys.list_state_functions
    - sys.list_state_modules
    - sys.reload_modules
    - sys.renderer_doc
    - sys.returner_argspec
    - sys.returner_doc
    - sys.runner_argspec
    - sys.runner_doc
    - sys.state_argspec
    - sys.state_doc
    - sys.state_schema
```

This brings us to some other features, including `argspec`, which gives us information about the arguments we can use with a function. Let’s go back to using the `pkg.install` function for this one:

```
$ sudo salt 'salt' sys.argspec pkg.install

alt:
    ----------
    pkg.install:
        ----------
        args:
            - name
            - refresh
            - fromrepo
            - skip_verify
            - debconf
            - pkgs
            - sources
            - reinstall
            - ignore_epoch
        defaults:
            - None
            - False
            - None
            - False
            - None
            - None
            - None
            - False
            - False
        kwargs:
            True
        varargs:
            None
```

Here, we’re given a list of available arguments as well as the defaults for each argument. The `defaults` will always be in the same order as the `args` list, so `name` has a default of `None`, `refresh` has a default of `False`, etc.

We can also use wildcards for any of these. For example, if we want to see the argument specifications for all arguments for the `pkg` module, we can use:

```
$ sudo salt 'salt' sys.argspec pkg.*
```

Now let’s take a look at another `sysmod` function that will be invaluable for your Salt journey: the `doc` function. Like most Salt functions, this does exactly what it sounds like: it returns documentation about an execution module or specific function. Let’s check out the `doc` for `doc` itself:

```
$ sudo salt 'salt' sys.doc sys.doc
sys.doc:

    Return the docstrings for all modules. Optionally,
specify a module or a
    function to narrow the selection.

    The strings are aggregated into a single document on the
master for easy
    reading.

    Multiple modules/functions can be specified.

    CLI Example:
        salt '*' sys.doc
        salt '*' sys.doc sys
        salt '*' sys.doc sys.doc
        salt '*' sys.doc network.traceroute user.info

    Modules can be specified as globs.

    New in version 2015.5.0

        salt '*' sys.doc 'sys.*'
        salt '*' sys.doc 'sys.list_*'
```

As we can see, this also contains examples of how the function is used. We also don’t have to put in a function when using `sys.doc`; we can simply input a module or a partial name using a wildcard.

Remember how earlier in the course we linked to a list of available execution modules? We can also view what’s available to us on the command line with:

```
sudo salt 'salt' sys.list_modules
```

Note that these are just ***available*** modules. Linux servers won’t list Windows-specific modules, and servers without Apache installed won’t list the `apache` module.

`sys` is arguably the most important module to know because if we’re ever confused about any other execution module, we can use `sys` to give us some direction.

`sys` can also provide us with information about state modules, which we’re going to cover later. These are predominantly the same as the ones we addressed above but use the prefix `state_`.

The `test` Module

The `test` module provides us with a number of options for testing our Salt infrastructure. Many of these options have highly specific use cases, and we can view more information about them using the `sys.list_functions` and `sys.doc` modules. But first, let’s take a moment to consider some other useful modules.

The `test.ping` function is especially important. When working on temporary servers, especially ones that are frequently stopped, checking that the master can connect to the minions is a quick and easy way to make sure nothing is wrong with our infrastructure. In fact, before doing any work in Salt, I tend to start up my testing environment and run:

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

This way I know all my minions are functioning before I get too far into my work.

We can also do things like send information to Salt's outputter:

```
$ sudo salt '*' test.outputter "Hello, World!"
```

View the current stack trace:

```
$ sudo salt 'salt' test.stack
```

Or just output the versions of Salt, its dependencies, and the system itself using:

```
$ sudo salt 'salt' test.versions_information
```

Many of these commands are particularly useful for troubleshooting and diagnosing issues, so it's important to be aware of the `test` module.

The `pkg` Module

Now that we're learned the two execution modules that help us understand and troubleshoot Salt, we can work with some execution modules that actually make real changes to our minions. The first of these, the `pkg` module, allows us to work with the packages we have on our server. The `pkg` module is actually a ***virtual*** module, meaning it combines the functionalities of a number of more distro-specific modules so that it works across multiple operating systems. The `pkg` module uses one of the following modules when working with a minion: `aptpkg`, `brew`, `ebuild`, `freebsdpkg`, `openbsdpkg`, `pacman`, `pkgin`, `pkgng`, `pkgutil`, `solarispkg`, `solarisips`, `win_pkg`, `yumpkg`, or `zypper`. If you're familiar with multiple operating systems, you can probably tell which operating system each of these goes with.

The `pkg` module is ideal for quick installs and fixes. For example, say we need to get a better look at our I/O for a single server that's not running as well as the rest. You can install `iotop` on that server with a simple:

```
$ sudo salt 'minion1' pkg.install iotop
```

Of course, if we can install it, we can also remove it:

```
$ sudo salt 'minion1' pkg.remove iotop
```

Or perhaps we want to ensure that all of our packages on our Salt master are up to date:

```
$ sudo salt 'salt' pkg.upgrade
```

Alternatively, we also have the ability to do things like add and remove repository keys, list packages, mark specific packages so that they exhibit a certain behavior during upgrade, and much more. Again, we can rely on the `sys.doc` and `sys.list_functions` commands to find out more when we need to.

The `user` and `group` Modules

It's also helpful to understand how to manage users and groups using execution modules. This lets us quickly add and remove users, add and remove groups, and add and remove users from groups. We can also manage things like a user's home directory and default shell — anything that you would normally do for user management.

Like the `pkg` module, these are virtual modules. `user` combines the ***Nix* **`useradd`***, ***`userdel`***, and ***`usermod`*** features, with*** BSD's `pw` and Mac and Windows user management functionalities (which are traditionally done in a GUI). The `group` module also combines the ***Nix,*** BDS, MacOS, and Windows group functionalities.

Let's walk through adding a new group, `salt`, to which we'll add our default `user` user (if you're not using Linux Academy's cloud servers, substitute in the name of the user on your environment). We'll also add a new user for ourselves in order to get some hands-on experience working with arguments.

Add the `salt` Group

This is as easy as running a single command:

```
$ sudo salt 'salt' group.add salt
```

This might be a little hard to understand, given all the instances of the word `salt`, so let’s take a moment to remember how we break down commands:

```
sudo salt '<target>' group.add <arguments>
```

Note that there are additional arguments we could use — `gid`, `system`, and `root` — but none we need to use in this instance.

Now, let’s go ahead and add our `user` user:

```
$ sudo salt 'salt' group.adduser salt user
```

Notice that the name of the group we’re adding the user to comes before the username.

Add a New User

We also want to add a user for ourselves; after all, we are an employee at IdunnaSoft, too. Unlike most of the commands we’ve used thus far, the `user` command has a number of arguments we generally want to include. Let’s take a look:

```
$ salt 'salt' sys.argspec user.add

salt:
    ----------
    user.add:
        ----------
        args:
            - name
            - uid
            - gid
            - groups
            - home
            - shell
            - unique
            - system
            - fullname
            - roomnumber
            - workphone
            - homephone
            - createhome
            - loginclass
            - root
            - nologinit
```

Let’s add a user with our information. Here’s mine:

```
$ sudo salt 'salt' user.add elle groups=salt,sudo
home=/home/elle shell=/bin/bash fullname="Elle K"
```

I included arguments for our user’s home directory, the default shell, and full name. I also added our new user to our newly created `salt` group and granted superuser privileges.

Whether you use your new user for the rest of this course or stick with the `user` user is up to you! If you want to use your own user, you’ll have to set a password.

Set a Password

1. Hash the password you wish to use:

   ```
   $ sudo salt 'salt' shadow.gen_password 'mypassword'
   ```

2. Use the hash to set a password for your new user:

   ```
   $ sudo salt 'salt' shadow.set_password elle
   '$6$8stAtY6O$9o8prUuAeg94heacqhN2SsOTWQ/Kz7YzZOuByAKSQ9yR
   ```

The `grains` Module

Another module we want to take a look at before we proceed is the `grains` module. We’ve already added grain information through our minion configuration. The `grains` module is an alternative way to set, view, and manage grains for our minions.

First, let’s take a look at the grains that are available on a minion right now:

```
$ sudo salt 'minion1' grains.ls
```

The list is too long to replicate here, but we can see it contains a number of items related to our system, such as system architecture, kernel, CPU module, and more. We can also see that it contains our `roles` grain from earlier. Let’s say we wanted to double-check what values that grain contains:

```
$ sudo salt 'minion1' grains.fetch roles
```

We could have also used the `grains.get` or `grains.item` functions.

What if we wanted to view the grains and values for all our items at once? We would use:

```
$ sudo salt 'minion1' grains.items
```

Of course, we can also add grains. Let’s add one for the websites we’re “hosting” on this webserver:

```
$ sudo salt 'minion1' grains.set 'sites' eski
```

We can also append items to this list:

```
$ sudo salt 'minion1' grains.append sites internal
convert=True
```

The `convert` value converts our grain to a list.

And if we have to remove a value that was added by mistake, we can use:

```
$ sudo salt 'minion1' grains.remove sites internal
```

Note that this is intended to remove a value from a grain with a list of values. To remove a grain entirely, we would use:

```
$ sudo salt 'minion1' grains.delkey sites
```

The `cmd` Module

We’ve discussed a number of useful Salt modules so far, but what about when we need to perform a task that doesn’t have a native Salt module? We have the option to write the execution module ourselves, of course, but that’s a lot of work. While we may want to do that for common or recurring tasks that lack a native execution module, it’s not always feasible for the quick, one-off commands that we sometimes need to run. Luckily, we have the `cmd` module for this.

The `cmd` module allows us to run arbitrary commands across our servers. Note that we have to ensure the appropriate command is used on the appropriate server — if there’s a difference between a CentOS command and an Ubuntu command, for example, we’ll have to take that into account. But otherwise, `cmd` opens up a lot of options for things that Salt may not currently support. Also note that the `cmd` module essentially gives anyone with access to the Salt master root access to all of your servers. Use `cmd` wisely.

The most common function we’ll see with the `cmd` module is the `cmd.run` function. As the name suggests, this runs any command we give it. It also has the following associated commands: `run_all` (returns any output as a dictionary instead of a string), `run_bg` (runs the command as a background process), `run_chroot` (a `cmd.run_all` wrapped in a `chroot`), `run_stderr` (only returns error data), and `run_stdout` (only returns the standard output). There's also `cmd.shell`, which works the same way as `cmd.run`.

Let's try using the `cmd` module. Say we're experiencing some errors with our hostnames, and we want to make sure the `/etc/hosts` files for our minions didn't change:

```
$ sudo salt '*' cmd.run 'cat /etc/hosts'
```

Or maybe we want to add in an API key to one particular person's `.bashrc` file across all servers. To do this, we'd have to use the `runas` argument alongside the `cmd.run` command:

```
$ sudo salt '*' cmd.run 'echo "export API_KEY=KEYDATA" >>
.bashrc' runas=user
```

The `cmd` module also has some practical data-related features. Writing a script and want to make sure you're using the right path for an executable? Use `cmd.which`:

```
$ sudo salt 'minion2' cmd.which echo
```

Essentially, the `cmd` module enables us to work around any limitations Salt has by letting us work as system administrators like we normally would, but on a larger scale. You can simultaneously run commands across multiple servers, even if it means having to run a "normal" command within the module instead of running a Salt execution module.

Again, remember to be cautious. By granting someone access to your Salt master, unless you take other precautions, you're granting them root access to everything and the ability to run just about any command. Anything done as `cmd.run` also doesn't go through the testing that Salt modules do, so make sure you check for typos or test on a non-production server because making big changes.

The `git` Module

If you're familiar with other configuration management platforms, then you probably know that when working with recipes, manifests, or playbooks, you usually upload your work as you go to GitHub, GitLab, or some other source control repository. This is also a good practice for Salt, and along with many other frequently-used applications, Salt contains a module for managing Git. In our next lesson, we’ll start working on a MySQL formula for both of our servers. Let’s use Salt to set up Git on a server so we can push our future MySQL formula to a GitHub repository.

Set Up Git on the Salt Master

Before we begin, create a GitHub (or GitLab) repository for the `mysql` formula. Since this will be under your own account, you can give the repository any name you desire, but I suggest something descriptive. I’ll be using `la-salt-mysql` below.

1. Git is already installed on the Linux Academy cloud servers, so we don’t need to do any prep work there. Instead, we want to use Salt to set our default global configuration for our `user` user:

```
 $ sudo salt 'salt' git.config_set user.name Elle
global=True user=user
$ sudo salt 'salt' git.config_set user.email
elle@linuxacademy.com global=True user=user
```

Note that we have to define the user we want to set the configurations for; otherwise, it will default to the minion’s user, which is `root`. Additionally, if we do not include the `global=True` argument, we have to use the `cwd` parameter to define which directory path the user is being set under.

2. To check that our configuration values are properly set, let’s use the `git.config_get` function. Much like the function above, we’ll need to define our user and the directory we’re working with (or set `global=True`):

```
 $ sudo salt 'salt' git.config_get user.name user=user
global=True
$ sudo salt 'salt' git.config_get user.email user=user
global=True
```

You’ll notice that if we leave out any of the above required arguments, we receive an error:

```
 $ sudo salt 'salt' git.config_get user.email
global=True
salt:
ERROR: Minions returned with non-zero exit code
```

3. Create the `file_roots` directory, where we'll be storing all our Salt states and formulas:

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

At this point, we want to make sure that we can easily add, remove, and edit files in this directory. Remember how we added a `salt` group earlier?

4. Grant users in the `salt` group read, write, and execute privileges for the `file_roots`:

```
$ sudo chgrp salt /srv/salt/
$ sudo chmod 775 /srv/salt/
$ sudo chmod g+s /srv/salt/
```

5. In order for the permission changes to take effect, log out of the server, then log back in. You can also run `bash` to start a new session.

6. Create a directory to store our future MySQL formula:

```
$ mkdir /srv/salt/mysql
```

7. Initialize the directory. Instead of doing this the traditional way, let's use an execution module:

```
$ sudo salt 'salt' git.init cwd=/srv/salt/mysql
```

8. Add the remote origin:

```
$ sudo salt 'salt' git.remote_set /srv/salt/mysql
https://github.com/elle-la/la-salt-mysql.git
```

Now we can `add`, `commit`, `pull`, and `push` changes, either through Salt or Git itself. This exercise may seem a bit unnecessary — why couldn't we just add the Git repository like normal? — but consider instances where you may have to spin up multiple environments all working on the same Git project. Salt's `git` module also contains options for branching, archiving, checking out branches, and cloning — all the common Git functionalities. As usual, you can find more information through the `sys` command.

LEARNING ACTIVITY: USING EXECUTION MODULES

To get hands-on and practice your skills, log in to Linux Academy and complete the Using Execution Modules Learning Activity.

Chapter 4: Salt States and Formulas

Anatomy of a Salt State

At the beginning of this course, we noted that most of Salt was built upon the remote execution framework. One of the features that uses this framework is the state system, which allows us to create "infrastructure as code"-style blueprints for our minions. These let us describe our desired end state for a server, then, when run against a minion, bring the minion into compliance with that state.

Let's take a look at an example state and break down how it works:

```
set_hostname:
  network.system:
    - enabled: True
    - hostname: salt
    - apply_hostname: True
    - retain_settings: True
```

To download this file on your own server:

```
  cd /srv/salt
  curl -O
https://raw.githubusercontent.com/linuxacademy/content-ssce-
files/master/hostname.sls
```

This stanza of code is called a ***state***. It is a single configuration task for our minion(s). In this instance, the file `hostname.sls` sets the hostname for our master server. State files should always end in `.sls`; otherwise, Salt won't know what to run when we call them.

Let's break this state down line by line. We start with `set_hostname`, which is the name declaration of the state. This is something that you, as the person writing the state, set yourself. Name declarations should be specific and, if included in a formula (a collection of states), should have a title that somehow references the formula name. For example, if we're working on a series of states that configure MySQL, when adding MySQL users, `mysql_add_users` is a far better descriptor than `add_users`.

Next, we call the ***state module***. This is the state equivalent of an execution module. You'll notice the language in state modules tends to differ from that of execution modules. When adding a user via execution module, for example, we use `user.add`. In contrast, when adding a user via state, we use `user.present` because we're actually checking to see if a user is present, and then bringing the minion into compliance if the minion does not match the state. In the example above, `network.system` ensures that global networking properties are set.

All the parameters beneath the state module are just that: parameters. In this instance, we're enabling networking, setting the `hostname`, ensuring the hostname is applied, and making sure that the settings are retained[^2].

These states are written in YAML, a human-friendly markup language. When writing YAML for a state, it is important to remember that:

- Two spaces are used to tab.
- Keys should end in a colon (in the above example, the name declaration and state module are keys).
- Lists should begin with dashes (`-`) (in the above example, lists are the parameters defined for the execution module).

Next, let's take a look at how to actually use states. We have a few options for this. We can manually run a state using one of the following execution modules:

- `state.sls`
- `state.apply`

Or we can create a file that maps which states belong to what minions (the `top.sls` file), then apply all states at once either via scheduled runs (called ***highstates***) or by manually running the `state.highstate` execution module.

We have a whole topic dedicated to the `top.sls` file, so let's try running a `state.sls` instead:

```
$ sudo salt 'salt' state.sls hostname test=true
```

Notice that when we call the state, we use the name of the file without the `.sls` suffix. The `test=true` parameter ensures that there are no apparent errors with the state itself and reports back on how many changes would occur if we were to run it for real.

Now, let's write our first basic state.

[^2] Note that when restarting your Linux Academy cloud server, your hostname may be reset.

Create a MySQL Server Installation State

Although we have two servers in our environment working as databases, we're going to start building out our MySQL states with our CentOS server. If you're familiar with Debian or Ubuntu MySQL installs, you'll know that there's some extra configuration involved during the installation process, which we'll get into as we learn about advanced state creation.

If you downloaded the network state, then you're already in the `/srv/salt` folder. This folder is known as the `file_roots` and is the default location for storing states and formulas. This can be changed in the master config.

1. Navigate to the `mysql` directory that we created and initialized as our Git repository:

```
$ cd mysql
```

This is where we'll store all of our MySQL configuration files. Remember that a collection of related states is called a ***formula***.

2. Create and open a file, `server.sls`, where we can write our state:

```
$ $EDITOR server.sls
```

3. We're going to use this file to create a state to download the MySQL server. Let's start by adding a descriptive state declaration:

```
mysql_server_install:
```

4. Now we need to determine which state module to use. Most state modules share names with execution modules; it's the functions that change. This means that when installing packages with the `pkg` module, instead of using `pkg.install`, we use `pkg.installed`:

```
mysql_server_install:
  pkg.installed:
```

5. Finally, we need to set our parameters for the `pkg.installed` module. Remember that we can use the `sys.state_list_functions` function to view a list of functions. For `pkg.installed`, we have to define the name of the package, and we have the option to define a particular version (`version`), reinstall the package that is already installed (`reinstall`), refresh the repository database (`refresh`), and more. Let's set the `name` of the package:

```
mysql_server_install:
      pkg.installed:
        - name: mariadb-server
```

CentOS servers use a community-maintained version of MySQL called MariaDB.

6. Save and exit.
7. We can now test this state:

   ```
   sudo salt 'minion2' state.sls mysql.server test=true
   ```

   Note that, because the state is found under a directory, we called the directory name, then the state, in a format that looks a lot like calling a module.
8. Finally, add and commit this first state to Git:

   ```
   git add .
   git commit -am "MySQL server installaton state added
   (server.sls)"
   git push origin master
   ```

Note that, depending on your GitHub account security settings, you may need to generate a personal access token and use that to log in instead of your usual password.

The Salt File Server

As we know, most tools and systems are not only configured by running a series of commands or simply making sure packages are installed. Instead, we often have to make changes to the configuration files on the hosts themselves. To help with this, Salt includes a stateless ZeroMQ file server that lets us send files to the minions from the master.

Files should be saved in our `file_roots` (`/srv/salt/`), generally within the directory of the formula they are being used for. However, files that are not associated with any formulas or states can still be saved and shared.

For example, let’s work with a simple `vimrc` file that will configure Vim to use two spaces for tabbing. This will be useful if you’re planning on writing your states in Vim.

```
$ cd /srv/salt
$ curl -O
https://raw.githubusercontent.com/linuxacademy/content-ssce-
files/master/vimrc
```

Alternatively, if you prefer nano, feel free to grab the following file and substitute `vimrc` for `nanorc`:

```
    https://raw.githubusercontent.com/linuxacademy/content-
ssce-files/master/nanorc
```

Now, if we want to copy this file to a minion, we can use the `cp` execution module:

```
$ sudo salt 'salt' cp.get_file salt://vimrc ~/.vimrc
runas=user
```

Notice how we call the file from the master file server, `salt://`. This tells Salt to use `file_roots`. So by calling `salt://vimrc`, we're asking for the file located at `/srv/salt/vimrc`, since `/srv/salt` is the location of our `file_roots`.

But Salt's file server generally isn't used with execution modules. Instead, its real functionality comes into play when working with the state system.

Manage the MariaDB Configuration

For CentOS servers, the MariaDB configuration files are located in the `/etc/my.cnf.d` directory, with a primary configuration file located at `/etc/my.cnf`. Since we're working on setting up MariaDB as a server, not a client, we want to make sure our `/etc/my.cnf.d/server/server.cnf` file is managed by Salt.

1. Within the `mysql` directory, create a folder to store our configuration files:

```
  $ cd /srv/salt/mysql
$ mkdir files
$ cd files
```

2. Pull in the configuration file:

```
   $ curl -O
https://raw.githubusercontent.com/linuxacademy/content-
ssce-files/master/server.cnf
```

3. Open the file. When managing a file in Salt, it is best practice to place a comment at the top of the file informing anyone using the server that this file is managed by Salt and should not be edited outside of configuration management:

```
$ $EDITOR server.cnf
```

```
# The file is managed by Salt.
```

If necessary, you can get more specific in this comment to provide more information and discourage eager coworkers from editing the file.

4. Save and exit the file.
5. Now we can create another state to add this file to our CentOS 7 database minion. Let's go back into our MySQL formula folder, then create a `config.sls` file:

```
$ cd ..
$ $EDITOR config.sls
```

6. Next, we want to set our state's name declaration. Remember, we want this to be specific so that anyone running these states later can understand the output:

```
mysql_server-config:
```

7. Next, we'll use the `file` state module to work with our files. Specifically, we want to use the `file.managed` function, which allows us to define a file, its source, its location on the minion, and more:

```
mysql_server_config:
  file.managed:
```

8. Now we need to set the parameters for our managed file. The two parameters we ***must*** include are the `name` parameter, which defines the location of the file on the minion, and the `source` parameter, which defines where on the master the file is located. Additional arguments, such as `user`, `group`, and `source_hash` are also available. All arguments can be found via the `sys.state_argspec` function.

Let's add our file's name and source:

```
mysql_server_config:
  file.managed:
    - name: /etc/my.conf.d/server.cnf
    - source: salt://mysql/files/server.cnf
```

9. Save and exit.
10. Finally, let's ensure our state doesn't contain any errors by running a test state run:

```
    $ sudo salt 'minion2' state.sls mysql.config test=true
```

Requisites

Although we only have two states in our MySQL formula at this time, one thing we want to consider is the order that our states will work in. In general, we want to ensure that packages are installed first, then configurations are set, then any additional work that needs to be done — such as creating a user or database — is carried out.

This is where Salt's requisite system comes in. Salt's requisite system allows us to define the relationships between our states and include any requirements (such as necessary packages) that we need for our state to work.

There are two ways to define requisites: requisites and requisite_ins. In direct requisite relationships, the state defines what it needs to work. In requisite_in relationships, the state defines what ***other*** state depends on it.

For example, our server config requires the `mariadb-server` package to be installed. We can address this in two ways:

```
mysql_server_config:
  file.managed:
    - name: /etc/my.cnf.d/server.cnf
    - source: salt://mysql/files/server.cnf
    - require:
      - pkg: mariadb-server
```

Or:

```
mysql_server_install:
  pkg.installed:
    - name: mariadb-server
    - require_in:
      - file: /etc/my.cnf.d/server.cnf
```

In the first option, we denote that in order for the configuration file state to run, we first need the `mariadb-server` package to be installed. In contrast, the second option will download the `mariadb-server` package regardless; however, it still needs to be run before the config state because the `/etc/my.cnf.d/server.cnf` file depends on it.

Let's go ahead and change our `config.sls` to match the first option, then try to run the state:

```
$ sudo salt 'minion2' state.sls mysql.config test=true

minion2:
----------
          ID: mysql_server_config
    Function: file.managed
        Name: /etc/my.cnf.d/server.cnf
      Result: False
     Comment: The following requisites were not found:
                                 require:
                                     pkg: mariadb-server
     Changes:

Summary for minion2
------------
Succeeded: 0
Failed:    1
------------
Total states run:     1
Total run time:   0.000 ms
```

Unlike our prior attempt to run this state, we receive an error! This is because the required `mariadb-server` package has ***not*** been installed on our CentOS 7 minion; we've only been testing our states thus far, not applying them.

So, what happens if we run this ***with*** our required download state?

```
$ sudo salt 'minion2' state.sls mysql.server,mysql.config
test=true
```

This time, the state succeeds because its required state comes with it.

But `requires` isn't the only requisite we have available to us. In fact, there are several requisites we can use to ensure our states are ordered properly:

Requisite	Action
`require`	Sets required state; the targeted

	state must succeed in order for the dependent state to run
`watch`	Targeted state must run and return `true` in order for the dependent state to run
`prereq`	Dependent state takes action if it determines the targeted state will make any changes; runs `test=true` evaluation, then determines if it should act before any changes are made by the targeted state
`onchanges`	Dependent state only

	runs if the targeted state has made changes
`onfail`	Dependent state only runs if the targeted state fails

All of these states also include an `_in` option, in which the dependent state becomes the targeted state and vice versa.

Finally, let’s take one more look at requisites by creating a `restart.sls` state that will trigger a restart of our `mariadb` service if any changes are made to our `config.sls` server state.

Create a Restart State

1. Navigate to the MySQL formula directory, then create and open `restart.sls`:

```
 $ cd /srv/salt/mysql
$ $EDITOR restart.sls
```

2. Set the name declaration and function. To get this state to work, we’ll use two functions: `service.restart` and `module.wait`. `service.restart` is what restarts our defined service. `module.wait` prevents this state from running every highstate; instead, it waits for the provided requisite to occur.

```
mysql_restart:
  module.wait:
    - name: service.restart
    - m_name: mariadb
```

Here we can see that the `module.wait` function takes precedence; it’s the function we call after our name declaration. Then, the `name` value we provide is the actual function we want to work with. After this, we provide the service name, as we would if we were just using `service.restart`, only instead of setting it as `name`, we use the `m_name` option.

3. Next, we need to set our requisite. We only want this state to run when there are changes to our configuration file, so let’s use the `onchanges` option:

```
mysql_restart:
  module.wait:
    - name: service.restart
    - m_name: mariadb
    - onchanges:
      - mysql_server_config
```

We use the name declaration of our configuration state to denote what state we’re waiting to change.

4. Save and exit.

5. Finally, we can test this:

```
  $ sudo salt 'minion2' state.sls
mysql.server,mysql.config,mysql.restart test=true
```

Note that when we test now, we have to include all of our required states in order for the test (or the highstate itself) to work.

Unfortunately, because we’re not running these states (only testing them), we don’t get to see our requisite in action. So, let’s remove the `test` parameter:

```
  $ sudo salt 'minion2' state.sls
mysql.server,mysql.config,mysql.restart
```

Instead of a comment denoting that there were no changes, our `mysql_restart` state instead reports back as `true`. This means it was run successfully.

What happens if we run it again? Since our configuration file has now been updated, there will be no changes, so this time, we’ll get a comment saying the state was not run.

`init.sls`

We now have the very beginning of a MySQL (or MariaDB) formula. But what good are all these states if we still have to manually assign them to a server and run the commands? While repeatedly running the `state.sls` function is great for practicing writing and testing states in our learning environment, it’s not how we want to work day to day while managing a vast array of servers. Instead, let’s explore how we can map states both at the formula level with the `init.sls` file and at the minion level with `top.sls`, which we’ll cover in the next section.

`init.sls` is a file that should be included with every formula. Which states we include in it depends heavily on how we’re working with the formula, but generally we want to pull in things like installation states and mandatory users or configurations. For example, for our current formula, it would make sense to include our `server.sls` installation state and our `config.sls` state. Let’s add those now:

```
$ cd /srv/salt/mysql
$ $EDITOR init.sls
```

We’ll use `include` to call our state:

```
include:
  - mysql.server
  - mysql.config
```

Notice that we call our state the same way we do when running `state.sls` on the command line: using the formula name (taken from the directory) and the file name for the state.

Save and exit the file. Now let’s see what happens when we try to run the `mysql` formula on its own without defining a state:

```
$ sudo salt 'minion2' state.sls mysql test=true
```

Our server installation and config states were run! This is because when we run a formula using just the formula name, we’re running that formula’s `init.sls`. Notice that we didn’t include our restart state. Because our restart state is so closely related to our configuration state, it would make more sense to include it there. `include` statements aren’t exclusive to the `init.sls` file and can be used in any state:

```
$ $EDITOR config.sls

include:
  - mysql.restart

...
```

Save and exit. Now, let’s run our `mysql` formula again:

```
$ sudo salt 'minion2' state.sls mysql test=true
```

Notice that this time ***all*** our states are included, and we’ve only run a single state: `mysql`. Using `include` and the requisites we discussed in the last lesson can make for very clean, easy-to-use formulas.

There’s also a lot of room for changing things up, if necessary. If keeping things modular makes sense for your use case, `init.sls` files can contain as many or as few states as you want. There may even be times when you want to keep states entirely separate. For example, if we were to include a state for downloading the MySQL client, we might want to keep that out of our `init.sls` and instead reference it by state name when mapping our states and formulas to minions in our `top.sls` file (which we’ll create next).

`top.sls`

Now that we’ve written a basic MySQL formula and set the relationships between our states, we want to think about how to relate these formulas and states back to our minions. This is done with the `top.sls` file.

The `top.sls` file sits in our `file_roots` and lets us map our states and formulas to our minions. We can do this in many of the same ways we target minions with execution modules: through minion names (regular expressions included), grains, pillar data, subnet, data store match, and node group.

Let’s create the `top` file for our environment:

```
$ cd /srv/salt/
$ $EDITOR top.sls
```

First, we need to set our environment. Since Salt is most often run in a single environment called `base` (which is how we are currently running Salt), we’ll define this now:

```
base:
```

Next, we can begin to name our targets in a hierarchy underneath. Our formula is currently only set to work on `minion2`, so we can set that as our first target:

```
base:
  'minion2':
```

Next, we need to create a list of our desired states or formulas. Let’s go ahead and add our `mysql` state:

```
base:
  'minion2':
    - mysql
```

But what about instances where we only want to include a single state? To demonstrate, let’s create the MySQL client state that we discussed in the previous lesson. Specifically, let’s set it so we can install the MySQL client on our Ubuntu minion:

```
$ cd mysql
$ $EDITOR client.sls

mysql_client_install:
  pkg.installed:
    - name: mysql-client
```

Now we can map this to our `minion1` state in our `top` file:

```
$ cd ..
$ $EDITOR top.sls

base:
  'minion1':
    - mysql.client
  'minion2':
    - mysql
```

Notice how this time we call the state the same way we would when running it via the `state.sls` execution module: by the formula name (taken from the directory) then by the state name (taken from the file name).

Save and exit the file. Let’s manually run a highstate to ensure all our servers have their states enforced:

```
$ sudo salt '*' state.highstate test=true
```

Both `minion1` and `minion2` return information about their states, with `minion1` installing only the MySQL client, and `minion2` installing the server and setting its configuration. Our `salt` minion returns an error because we have nothing mapped to it. This is expected behavior.

If we wanted to be more restrictive, we could even rewrite this to ensure that only minions with the proper distribution attempt to run our formula. This uses the same technique as compound targeting (because it ***is*** compound targeting):

```
$ $EDITOR top.sls

base:
  'minion* or G@os:Ubuntu':
    - mysql.client
  'minion* or G@os:CentOS':
    - mysql
```

We can confirm this by running another highstate:

```
$ sudo salt '*' state.highstate test=true
```

Our results are the same as our previous highstate, despite the different targeting method. Again, ignore the error caused by not having states mapped to our `salt` server.

LEARNING ACTIVITY: CREATING AN APACHE FORMULA

To get hands-on and practice your skills, log in to Linux Academy and complete the Creating an Apache Formula Learning Activity.

Jinja

Although we now have a basic MySQL formula, it has a few limitations. Notably, we can only use it to install and configure a MySQL server on CentOS and install a client on Ubuntu.

However, state creation does not end with YAML. Salt lets us use a templating language to expand our states and ensure they work across any distribution or operating system we need them to. By default, this templating language is Jinja, and that’s what we’ll use in this course.

Jinja is a Python-based templating language — remember, Salt is written in Python — and works with Python 2.4 and above. When we use Jinja in a state, it is evaluated before the YAML itself and prior to the state running.

Perhaps the easiest way to identify Jinja is by its opening and closing brackets, or ***delimiters***:

- `{% ... %}` — Statements

- `{{ ... }}` — Expressions (in Salt, this is how we call variables)
- `{# ... #}` — Comments

Most often, we'll use Jinja to parameterize our variables, create `if` statements, and generate `loops`. Let's start with an `if` statement.

If you've ever installed MySQL across distributions, then you might have guessed why we started with CentOS 7 when we created our formulas. Debian-based distributions, such as Ubuntu, require user input when installing the MySQL server, which requires the use of `debconf` in scripts and in Salt.

If we want to expand our formula to work across multiple distributions, we need to add some optional states to our `server.sls` that will only work on machines running Debian-based operating systems.

Add `debconf` to a MySQL Server State

1. Open the `server.sls` file for our MySQL formula:

```
 $ cd /srv/salt/mysql
$ $EDITOR server.sls
```

2. We need to create an `if` statement that includes a state that will only run when the parameters we set are met. In this instance, we want our state to run only on systems of the Debian operating system family:

```
 {% if grains['os_family'] == 'Debian' %}

{% endif %}
```

Note the use of the `{%` delimiters. Since we need to match an operating system family, we call a grain with the `grains` function. We also need to make sure to close our statement.

3. Next, we have to write a state that will install the `debconf` utility. This is the same as if we were to write it outside of our `if` statement:

```
 {% if grains['os_family'] == 'Debian' %}

mysql_debconf_install:
   pkg.installed:
     - name: debconf

{% endif %}
```

4. Finally, let's add our `debconf` settings. For MySQL, this means setting

the root password in two different settings. We can do this through the use of the debconf state module.

```
mysql_debconf_settings:
  debconf.set:
    - name: mysql-server
    - data:
        'mysql-server/root_password': {'type':
'password', 'value': 'temppass'}
        'mysql-server/root_password_again': {'type':
'password', 'value': 'temppass'}
```

As our formula grows, we need to consider our requisites as we write, not as something to be added later. For this particular state, we need the debconf package to be installed. However, this state is also necessary for mysql_server_install to work on Debian-based distros. As such, this is an excellent time to use a require_in — that way, the state will only be required for minions of the Debian family.

```
mysql_debconf_settings:
  debconf.set:
    - name: mysql-server
    - data:
        'mysql-server/root_password': {'type':
'password', 'value': 'temppass'}
        'mysql-server/root_password_again': {'type':
'password', 'value': 'temppass'}
    - require:
      - pkg: debconf
    - require_in:
      - mysql_server_install
```

5. Now let's test what we have so far. Let's change the installation value in our regular server installation package name from mariadb-server to mysql-server to test our state on our Ubuntu minion:

```
mysql_server_install:
  pkg.installed:
    - name: mysql-server
```

6. Finally, let's test and then run our server state against minion1. We only want to run the single mysql.server state, not the entire formula:

```
$ sudo salt 'minion1' state.sls mysql.server test=true
```

```
   $ sudo salt 'minion1' state.sls mysql.server
```

Note that we do ***not*** want to commit any of this to GitHub (this goes for the rest of the section). We now have password information in our states, which we do ***not*** want to share, even if it’s only a temporary password on a demo environment.

Jinja Mapping

Now that our formula is in a half-CentOS and half-Ubuntu state, we need to use Jinja to parameterize things like package names and file locations on a per-distribution basis. This is done through what is commonly called a `map.jinja` file, although the file name does not have to be `map.jinja`.

The `map.jinja` file allows us to map out variables that differ between distribution or distro family. We can then reference these variables in our states, and the appropriate value will be used according to the distribution of the minion. This works because Jinja is rendered before the states themselves are run.

Let’s start by working on that same `server.sls` file we added an `if` statement to earlier:

```
{% if grains['os_family'] == 'Debian' %}

mysql_debconf_install:
  pkg.installed:
    - name: debconf



mysql_debconf_settings:
  debconf.set:
    - name: mysql-server
    - data:
        'mysql-server/root_password': {'type': 'password',
'value': 'temppass'}
        'mysql-server/root_password_again': {'type':
'password', 'value': 'temppass'}
    - require:
      - pkg: debconf
    - require_in:
      - mysql_server_install
```

```
{% endif %}


mysql_server_install:
  pkg.installed:
    - name: mariadb-server
```

We could parameterize the `debconf` install itself. However, since that part is already specific to Debian-based distros, let’s consider the `mysql_server_install` state instead. Specifically, we want to make sure the `name` parameter for the installation module can use the appropriate package on the appropriate distro.

Let’s navigate to our `mysql` formula folder and create a `map.jinja` file:

```
$ cd /srv/salt/mysql
$ $EDITOR map.jinja
```

We’ll start by using the `set` template tag to inform Salt that we’re creating a set of variables we’re going to use. In this line, we’ll also define that we’re filtering our variables by grain:

```
{% set mysql = salt['grains.filter_by']({
```

The name `mysql` here is arbitrary — it could be anything. I tend to just use the formula name.

You may have noticed that we didn’t define which grain we’re filtering with. This is because the `os_family` grain is used by default, and that’s the grain we’re going to be using.

We also want to take this time to close our set’s brackets:

```
})%}
```

Now, within our two sets of brackets, we want to define our first subset. We’ve been working primarily with CentOS, so let’s start there. CentOS is in the Red Hat family, so `RedHat` is the operating system family we’re going to reference:

```
{% set mysql = salt['grains.filter_by']({

  'RedHat': {
    'server': 'mariadb-server',
  },

})%}
```

Here, we define our subset (again, `RedHat`), then create the `server` variable and assign it the value `mariadb-server`, which, as we know, is the name of the package we use in our `server.sls` state.

Note that after we define our key-value pair, we end the line in a comma. We also do this when we end our subset. Don't forget your commas!

We also want to add this same variable for Debian-based distros, like our Ubuntu server. Although it's not required or even considered best practice to keep our subsets in alphabetical order, I prefer to. So I'll add our `Debian` set above `RedHat`:

```
{% set mysql = salt['grains.filter_by']({

  'Debian': {
    'server': 'mysql-server',
  },
  'RedHat': {
    'server': 'mariadb-server',
  },

})%}
```

Now let's save and exit the file, then consider how we're going to take the set we just created and use it without our states.

Once more, open up the `server.sls` file:

```
$ $EDITOR server.sls
```

To call the variable we just added, we need to import our map file. We can do this with a single `from` statement:

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

The `mysql/map.jinja` value references the location of our map file from our `file_roots`, while the `mysql` value comes from our set name (not the formula name). `with context` references a Jinja behavior — `import` statements only retain their context when told to do so.

Finally, we also want to reference the variable we added in our mapping. To call a variable, we use double curly brackets (`{{ ... }}`), then reference the set name, followed by the variable name:

```
    - name: {{ mysql.server }}
```

Save and exit the file. Let's see what happens when we test this:

```
$ sudo salt 'minion*' state.sls mysql.server test=true
```

Notice how the `name` values in our output reference a different package depending on the distro, just like we intended. We can also see that our `debconf` states only work on our Ubuntu server.

Now let's update the rest of our formula to use our `map.jinja` file.

Update the MySQL Formula

The following values need to be added to the `map.jinja` **file: -** `client.sls` **-** `name` **package -** `config.sls` **- Add an Ubuntu server config (found here) -** `name` **value -** `source` **value -** `required` **package value -** `restart.sls` **-** `m_name` **value**

If you're feeling confident, try to update the `map.jinja` file on your own first!

1. Add the Debian-based configuration file to the `files` directory:

```
  $ curl https://github.com/linuxacademy/content-ssce-
files/blob/master/mysqld.cnf -o files/deb_mysqld.cnf
```

 I added the `deb` prefix to keep our files organized. I'm also going to rename the original file so we know it's intended for Red Hat-based servers:

```
  $ mv files/server.cnf files/rh_server.cnf
```

 Now that the prep work is done, we can focus on updating our states.

2. Reopen the `map.jinja` file. You may find it beneficial to have two terminal tabs or panes open: one for our map file and one for the state we're updating. We'll start with the `client.sls` state.

 For this state, there's only one value that we want to add to our mapping: the `name` value. It's currently set to `mysql-client`, which is the value we want to input for our `Debian` set. The name of the client package for Red Hat-based systems, including CentOS, is `mariadb`:

```
  'Debian': {
    'server': 'mysql-server',
    'client': 'mysql-client',
},
'RedHat': {
    'server': 'mariadb-server',
    'client': 'mariadb',
```

We also want to update the state itself:

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

mysql_client_install:
   pkg.installed:
     - name: {{ mysql.client }}
```

3. Now we want to do the same thing to our `config.sls`. We're going to name the variable for our `name` parameter `server_conf` and the `source` parameter `server_conf_source`. The required package should use our `server` variable that we created earlier:

```
  ...

     'server_conf': '/etc/mysql/mysql.conf.d/mysqld.cnf',
     'server_conf_source':
'salt://mysql/files/deb_mysqld.cnf',

...

     'server_conf': '/etc/my.cnf.d/server.cnf',
     'server_conf_source':
'salt:///mysql/files/rh_server.cnf',

...
```

Then update the state:

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

include:
   - mysql.restart

mysql_server_config:
   file.managed:
     - name: {{ mysql.server_conf }}
```

```
    - source: {{ mysql.server_conf_source }}
    - require:
      - pkg: {{ mysql.server }}
```

4. The last state we need to update is the `restart` state. As before, start by adding the required variable to the `map.jinja` file (`service`):

```
 ...

    'service': 'mysql',

...

    'service': 'mariadb',

...
```

Then update the state:

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

mysql_restart:
  module.wait:
    - name: service.restart
    - m_name: {{ mysql.service }}
    - onchanges:
      - mysql_server_config
```

5. Test the formula:

```
$ sudo salt 'minion*' state.sls mysql test=true
$ sudo salt 'minion*' state.sls mysql.client test=true
```

LEARNING ACTIVITY: JINJA MAPPING

To get hands-on and practice your skills, log in to Linux Academy and complete the Jinja Mapping Learning Activity.

Pillar

Now that our MySQL formula works across multiple distros, we have one last component to configure: pillar. As we discussed in the “Concepts” section of this course, pillar is a data store where we can keep user-provided key-value pairs. These pairs can then be referenced in our MySQL states.

Before we get started, let’s consider some of the benefits and use cases of pillar.

Pillar can be used to store variables, minion configurations, any arbitrary data that would be helpful to have in a dictionary, and highly sensitive data, such as passwords.

Pillar is stored outside of our `file_roots` in the aptly-named `pillar_roots`, which, by default, is found at `/srv/pillar`. To set this up, we follow essentially the same steps we took to set up our `file_roots`.

1. Create `/srv/pillar`:

```
$ sudo mkdir /srv/pillar
```

2. Set the `pillar_roots` so anyone in the `salt` group can add and edit files:

```
  $ sudo chgrp salt /srv/pillar/
$ sudo chmod 775 /srv/pillar/
$ sudo chmod g+s /srv/pillar/
```

3. Log out, then log back in, or run `bash` to refresh.

Now, pillar itself is already set up on all of our servers. Although we had to add the default directory, we don’t need to make any additional configurations for pillar to function. Instead, we can jump right in and add pillar to our formula.

Let’s consider our end goal again. We want to create a MySQL formula that will add two databases to our database servers. To do this, we need to provide Salt information regarding the `root` user so it can use that information to log in to our MySQL instance. Additionally, we need to provide Salt with the MySQL Python connector. Let’s first create that state, then `include` it in our `init.sls` file:

```
$ vim /srv/salt/mysql/map.jinja


...
  'Debian': {
...
    'python': 'python-mysqldb',
...

  'RedHat': {
...
    'python': 'MySQL-python',
...
```

```
$ vim /srv/salt/mysql/python.sls

{% from "mysql/map.jinja" import mysql with context %}

mysql_python_install:
  pkg.installed:
    - name: {{ mysql.python }}


$ vim /srv/salt/mysql/init.sls

include:
  - mysql.server
  - mysql.config
  - mysql.python

$ sudo salt 'minion*' state.sls mysql test=true
$ sudo salt 'minion*' state.sls mysql
```

With that done, we can now create the state to ensure our `root` user can be used within our formula:

Add the `root` Connection User

1. Let’s first create the state for our MySQL `root` user:

```
$ vim /srv/salt/mysql/root.sls

mysql_root_user:
  mysql_user.present:
    - name: root
    - password: temppass
    - host: localhost
```

Notice that we use the `mysql_user.present` state module, which we gain access to when MySQL is included on a minion. As you may have guessed, this ensures a MySQL user with the defined parameters is present on the server.

2. Now let’s start building out our pillar data. Pillar works in a way that is remarkably similar to states. We can either include a `mysql.sls` file in the `pillar_roots` itself, or we can add a `mysql` directory and store

specific MySQL-related pillar files below it, referencing them the same way we would otherwise: `mysql.<pillarfile>`. For our purposes, we’ll use a simple `mysql.sls` in our `pillar_roots`:

```
  $ cd /srv/pillar
$ vim mysql.sls
```

3. The first thing we want to do is set the name of this dictionary (since that’s essentially what we’re creating: a dictionary). The name we choose should somehow reference what the pillar data is used for. In our case, it’s `mysql`:

```
mysql:
```

It is entirely possible to use a plain key-value pair, such as `password: passpass` **in this file instead of creating a dictionary. However, this is a rare use case.**

4. Next, we need to define our subdict. It might be tempting to name this something like `users`, but since we’re specifically setting up our connection user, we’re going to keep things simple and separate by using the name `root`:

```
mysql:
  root:
```

5. Now let’s add some key-value pairs to this dictionary. Specifically, let’s define the `root` user and password:

```
mysql:
  root:
    name: root
    password: temppass
```

The `temppass` value is the same root password we set for `debconf` earlier. We’ll cover how to encrypt this in the next lesson.

6. Save and exit the pillar file.

7. Now we need to map our pillar data to the minions that need access to it. This is done with a `top.sls` file, much like when we map states to minions in our `file_roots`:

```
$ vim top.sls
```

```
base:
   'minion*':
      - mysql
```

Note that when mapping pillar files to minions, they are applied in order. If there are conflicting keys, the last value for that key will be used.

When you're finished, save and exit.

8. Let's refresh our pillar data and check to see if Salt can read our pillar items:

```
 $ sudo salt '*' saltutil.refresh_pillar
$ sudo salt '*' pillar.items
```

Pillar data is stored in memory in order to keep costs down — compiling data can get expensive! By default, pillar is refreshed at minion start and during state runs.

9. Now that we have our data in pillar, we can reference it within our states. Let's open our `root.sls` state:

```
 $ cd /srv/salt/mysql
$ vim root.sls
```

To call a value from pillar, we use the format `{{ pillar['key'] }}`. Notice the double curly brackets — just like we would use to call a Jinja variable. Since we have nested keys, we'll use: `{{ pillar['key']['nested-key=1']['nested-key-2'] }}`. Let's go ahead and use this to update our state:

```
  mysql_root_user:
    mysql_user.present:
      - name: {{ pillar['mysql']['root']['name'] }}
      - password: {{ pillar['mysql']['root']['password']
}}
      - host: localhost
```

10. Save and exit, then add the `root` state to our `init` file:

```
 $ vim init.sls

include:
  - mysql.server
  - mysql.config
  - mysql.python
  - mysql.root
```

11. Test and run the formula:

```
 $ sudo salt 'minion*' state.sls mysql test=true
```

Notice we get a CentOS 7 error. This is because, although we have our packages installed, we never added a state to start and enable our service. Let’s quickly add a `service` state, then try again:

```
$ vim service.sls

{% from "mysql/map.jinja" import mysql with context %}

mysql_service_enable:
  service.running:
    - name: {{ mysql.service }}
```

Note that we already have the appropriate variable available from our `restart` state.

```
$ vim init.sls

include:
  - mysql.server
  - mysql.service
  - mysql.config
  - mysql.python
  - mysql.root
```

Retest:

```
$ sudo salt 'minion*' state.sls mysql test=true
```

Run the formula:

```
$ sudo salt 'minion*' state.sls mysql
```

The GPG Renderer

Although getting our passwords out of our Salt states and into pillar is a step in the right direction for securing Salt, we can take this one step further by encrypting our data using Salt’s GPG renderer. This enables Salt to read encrypted data and use it within states.

Before we begin encrypting passwords, however, we need to prepare our master server to use this renderer.

Prepare the GPG Renderer:

1. Create a directory in which we can store our GPG keys:

   ```
   $ sudo mkdir /etc/salt/gpgkeys
   ```

2. Set the permissions:

```
 $ chmod 0700 /etc/salt/gpgkeys
```

3. Chances are that our Salt master does not have enough entropy to create our master key. So let’s first generate some extra entropy:

```
 $ sudo apt-get install rng-tools
$ sudo rngd -r /dev/urandom
```

4. Now let’s create our master key:

```
 $ sudo gpg --gen-key --homedir /etc/salt/gpgkeys
```

Leave the key type, size, and expiry date with the default settings. Generally, when creating a key, you should follow your company’s existing policies regarding GPG keys and expiration.

Set the `full name` to `Saltstack`, and leave the `email` and `comment` prompts blank. Do ***not*** input a password! Salt will not be able to use this key if a password is set.

5. To use this key to encrypt our files, we need to import it to our workstation. In our case, that’s the master itself, although generally it will be whatever environment you’ve set up to create your states.

First, let’s export the key:

```
 $ sudo gpg --homedir /etc/salt/gpgkeys --armor --export
Saltsalt > exported_pubkey.gpg
```

Then import it:

```
 $ gpg --import exported_pubkey.gpg
```

Now we’re ready to create ***secrets*** (encrypted words or phrases) that we can use in pillar. Let’s work on encrypting that `temppass` password we used for our `root` MySQL user.

Encrypt Pillar Data

1. First, we want to encrypt our password:

```
 $ echo -n "tempppass" | gpg --armor --trust-model always
encrypt -r Saltstack
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1

hQEMA+IrJEeFbLG0AQf/dahR/1jvoO4+OV9yIenHkFKHerTNwWiBXgEq3
9KgNLpi5bvD/kv3NN0BhPNHCaqR2Y71t5hjHFqR8Fy8X99HXfXuyY/hvv
76MKUPS4MCyldvcR1IXY+AZdvjM0eM3YkV5YnJpGry9m2ZXJXEEpIQFUq
v4StCGaWnSgXzHCyMWjJ7L3iQi3GfOfUFeQlkRr380j2/PaaH4qJYKJ5F
csIOQmyCQQT/gQSkBODQecIlD+sScZKdP+vMyMZmYcBDVupW/W6uF/xsl
```

```
CTdbcPg5UMl/uR6v8YWtKovwD1VuTJ1X+ZuKIlFEodJEAW7q6N4tTPeVh
ElE7zBSjB1anAtxJ7sv7+fir+lhFYjwtpBn2O6aXZ4irGmOBv8kqcMOIB
Pk3cyG0=
=Brqj
-----END PGP MESSAGE-----
```

Make note of this output. This is what we'll be adding to our pillar file.

2. Now we can open our `mysql.sls` pillar file:

```
$ vim /srv/pillar/mysql.sls
```

3. Next, we want to inform Salt that our file contains a cipher that Salt will need to decrypt. To do this, we add a hashbang to the top of our file, telling Salt which renderers it will need to use:

```
#!yaml|gpg
```

4. Now we can add that PGP block to our `password` value. To do this, we use a pipe (`|`) to tell Salt to look one line down at the provided block of text:

```
mysql:
  root:
    name: root
    password: |
      -----BEGIN PGP MESSAGE-----
      Version: GnuPG v1


hQEMA+IrJEeFbLG0AQf/dahR/1jvoO4+OV9yIenHkFKHerTNwWiBXgEq3

9KgNLpi5bvD/kv3NN0BhPNHCaqR2Y71t5hjHFqR8Fy8X99HXfXuyY/hvv

76MKUPS4MCyldvcR1IXY+AZdvjM0eM3YkV5YnJpGry9m2ZXJXEEpIQFUq

v4StCGaWnSgXzHCyMWjJ7L3iQi3GfOfUFeQlkRr380j2/PaaH4qJYKJ5F

csIOQmyCQQT/gQSkBODQecIlD+sScZKdP+vMyMZmYcBDVupW/W6uF/xsl

CTdbcPg5UMl/uR6v8YWtKovwD1VuTJ1X+ZuKIlFEodJEAW7q6N4tTPeVh

ElE7zBSjB1anAtxJ7sv7+fir+lhFYjwtpBn2O6aXZ4irGmOBv8kqcMOIB
      Pk3cyG0=
      =Brqj
      -----END PGP MESSAGE-----
```

Notice that the message is indented underneath the `password` key, not direc
line with it.

5. Save and exit, then refresh the pillar data:

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

6. Let’s see what happens when we run our MySQL state now:

```
$ sudo salt 'minion*' state.sls mysql
```

Nothing! It looks exactly like it did before. This is because we didn’t actually change anything; we just supplied the password in an encrypted form.

7. Now let’s take that stored password key and use it in our `debconf` states:

```
$ vim /srv/salt/mysql/server.sls
```

```
mysql_debconf_settings:
   debconf.set:
     - name: mysql-server
     - data:
         'mysql-server/root_password': {'type':
'password', 'value': '{{ pillar['mysql']['root']
['password'] }}'}
         'mysql-server/root_password_again': {'type':
'password', 'value': '{{ pillar['mysql']['root']
['password'] }}'}
     - require:
       - pkg: debconf
     - require_in:
       - mysql_server_install
```

Save and exit.

8. Let’s reapply the formula to ensure no changes were made:

```
$ sudo salt 'minion*' state.sls mysql
```

9. We can also check that our password renders correctly by viewing our pillar items:

```
$ sudo salt ‘*’ pillar.items
```

Note that the password is ***unencrypted*** in the resulting output.

10. Finally, now that the passwords have been removed from our state, we can push to GitHub:

```
$ cd /srv/salt/mysql/
$ git add .
$ git commit -am "Added service, root, and python states;
stripped passwords"
$ git push origin master
```

Pillar and Jinja

So far in this section, we have created a MySQL formula that installs and configures MySQL across two distro families and prepares MySQL for management in Salt. The final step is to create two databases for our company's two websites. We'll do this with the `mysql_database.present` state module and the `root` MySQL user connection data we set up previously. Finally, we'll use the ability to loop through pillar data to create more than one database with a single state.

Create a Database State

1. The first thing we want to do is create that MySQL database state. Within the `mysql` directory in our `file_roots`, create and open `database.sls`:

```
$ $EDITOR database.sls
```

2. Next, we're going to write our basic state. The name we give this state will be temporary, but for now let's use `mysql_db_create`. Beneath the name declaration, we also want to include the `mysql_database.present` module:

```
mysql_db_create:
  mysql_database.present:
```

3. Now we need to set our parameters, including the name of our database, its host, and the connection information. In this case, our connection information is taken from our `mysql:root` pillar data:

```
mysql_db_create:
  mysql_database.present:
    - name: eski
    - host: localhost
    - connection_user: {{ pillar['mysql']['root']
['name'] }}
    - connection_pass: {{ pillar['mysql']['root']
['password'] }}
```

```
      - connection_charset: utf8
```

So now we know that when we add database information, we need to include the database name and host information.

4. Let’s take this new knowledge and add it to our pillar file. We know we need two databases — `eksi` and `internal`— both on the `localhost`. But first, we need to add the name of our sub-dictionary, `database`:

```
  $ $EDITOR /srv/pillar/mysql.sls

mysql:
   root:
     name: root
     password: |
       <PGP INFORMATION>
   database:
```

5. We can now add information for each database:

```
   database:
     eski:
       host: localhost
     internal:
       host: localhost
```

You may have noticed that we don’t specifically define the `name` value. This is because when we create our loop, we’ll pull the name from the sub-dictionary name itself.

6. Save and exit.

7. Return to the `database.sls` state file. We want to create a loop that will cycle through our dictionaries. This is done using Jinja and works just like any other programmatic loop. In fact, it’s ***exactly*** the same as creating a loop in Python, with a few added Salt and Jinja features.

```
  $ $EDITOR /srv/salt/mysql/database.sls
```

8. Let’s create our `for` loop. First, I’m going to provide the code line, and then we’ll break it down:

```
  {% for database, arg in salt['pillar.get']
('mysql:database', {}).items() %}
```

– `database, arg` — These are arbitrary values we’re using to reference parts of our pillar. The `database` value references our database name, and the `arg` value references any keys below it.

– `salt['pillar.get']()` — This is a function that simply retrieves our pillar data. In this case, it takes two values: the first is the location of the names of our databases, and the second is a placeholder for the `args`.

– `('mysql:database', {})` — These are the values for this specific instance of our function. We call `mysql:database` for the first value because that's the dictionary that contains our database information. The empty set after this references all the key-value pairs in each individual database set.

– `.items` — This tells Salt to iterate through each sub-dictionary under `database`. In Python 2, this function is called `.iteritems`. Both versions currently work in Salt.

9. Close the `for` loop at the end of the file:

```
{% endfor %}
```

10. Finally, we can call the values in our state. To reference the database name, we use the `database` variable defined in the first part of our loop. To reference an argument, we use `arg.<KEYNAME>`. For example, to reference our host, we would use `arg.host`:

```
 {% for database, arg in salt['pillar.get']
('mysql:database', {}).items() %}

mysql_db_create:
  mysql_database.present:
    - name: {{ database }}
    - host: {{ arg.host }}
    - connection_user: {{ pillar['mysql']['root']['name']
}}
    - connection_pass: {{ pillar['mysql']['root']
['password'] }}
    - connection_charset: utf8

{% endfor %}
```

However, we aren't quite done yet. Although we're looping through twice (once for each database), both of the created states are going to have the same name declaration, which will result in an error. To prevent this, we need to reference our database name in the name declaration itself:

```
 {% for database, arg in salt['pillar.get']
```

```
('mysql:database', {}).items() %}

mysql_{{ database }}_db_create:
  mysql_database.present:
    - name: {{ database }}
    - host: {{ arg.host }}
    - connection_user: {{ pillar['mysql']['root']['name']
}}
    - connection_pass: {{ pillar['mysql']['root']
['password'] }}
    - connection_charset: utf8

{% endfor %}
```

11. Save and exit the file.

12. Finally, let’s test and run our database state to see what happens:

```
$ sudo salt 'minion*' state.sls mysql.database test=true
```

Notice that each minion has ***two*** resulting states in the output: one for each database that we need created!

Now we can run the state for real:

```
$ sudo salt 'minion*' state.sls mysql.database
```

And we’ve done it! If you think back to the goals we set for our MySQL formula at the start of this course, you’ll see that we’ve met all the requirements. And in the process, we’ve learned how to create both simple and advanced states.

For the rest of this course, we’ll explore how to use these states and execution modules in different ways, as well as how to further manage our infrastructure using additional Salt features.

LEARNING ACTIVITY: USING PILLAR IN SALT STATES

To get hands-on and practice your skills, log in to Linux Academy and complete the Using Pillar in Salt States Learning Activity.

Chapter 5: Events

Event System Overview

In addition to its remote execution and state systems, Salt uses a pub/sub system for event management. Salt's event system enables us to set up reactions to watched events and even send Salt events to third-party applications.

Salt's event system, or ***event bus***, is used for inter-process communication via UNIX domain sockets (UDX) and network transport. It consists of two components: the ***event socket***, which publishes events, and the ***event library***, which listens to events and sends them to the Salt system. Each Salt master and minion has its own event bus.

Let's take a look at the event bus in action. Open a second window, tab, or pane and `ssh` into another instance of your Salt master. On the CLI for the instance of your master, input the following to connect to the event bus:

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

Since our test environment is pretty uneventful compared to most production systems, we're not going to see much here. So, on the second instance of our master, try running a command:

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

    salt/job/20180605133538313578/ret/minion2    {
        "_stamp": "2018-06-05T13:35:38.361872",
        "cmd": "_return",
        "fun": "test.ping",
        "fun_args": [],
        "id": "minion2",
        "jid": "20180605133538313578",
        "retcode": 0,
        "return": true,
        "success": true
    }
    salt/job/20180605133538313578/ret/minion1    {
        "_stamp": "2018-06-05T13:35:38.374056",
        "cmd": "_return",
        "fun": "test.ping",
        "fun_args": [],
        "id": "minion1",
        "jid": "20180605133538313578",
        "retcode": 0,
        "return": true,
        "success": true
    }
```

Now watch the event bus. As each individual job is completed, we receive information about the job ID, the command run, the function used, any arguments provided, and return information, including whether or not the job succeeded.

To see what an unsuccessful job looks like, let’s intentionally run a state that will fail. Since we know our `mysql.config` state can’t run without its requisites, we can use this to test:

```
$ sudo salt 'minion1' state.sls mysql.config

salt/job/20180605133622335971/ret/minion1    {
    "_stamp": "2018-06-05T13:36:22.669850",
    "cmd": "_return",
    "fun": "state.sls",
    "fun_args": [
        "mysql.config"
    ],
    "id": "minion1",
    "jid": "20180605133622335971",
    "out": "highstate",
    "retcode": 2,
    "return": {
        "file_|-mysql_server_config_|-
/etc/mysql/mysql.conf.d/mysqld.cnf_|-managed": {
            "__run_num__": 0,
            "__sls__": "mysql.config",
            "changes": {},
            "comment": "The following requisites were not
found:\n                   require:\n
pkg: mysql-server\n",
            "result": false
        },
        "module_|-mysql_restart_|-service.restart_|-wait": {
            "__run_num__": 1,
            "__sls__": "mysql.restart",
            "changes": {},
            "comment": "One or more requisite failed:
mysql.config.mysql_server_config",
            "result": false
        }
    },
    "success": true
}
file.managed    {
    "__run_num__": 0,
    "__sls__": "mysql.config",
    "_stamp": "2018-06-05T13:36:22.670215",
    "changes": {},
    "comment": "The following requisites were not found:\n
require:\n                       pkg: mysql-server\n",
    "result": false,
    "retcode": 2
}
```

Notice that the event bus output contains the same comments and results as the information returned to us when we ran the state itself, only in a more machine-readable format.

Now let’s look at the output.

The ***event tag*** is the line of data that starts each job. It always begins with the word `salt` and looks like this: `salt/job/20180605133622335971/ret/minion1`. The slash (`/`) is used for namespacing, with subsequent lines referencing what is happening (which job is running), the ID of the job (something like `20180605133622335971`), and which minion the job was run on. To see how the output varies for different kinds of jobs, log in to one of your Salt minions and restart the `salt-minion` service:

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

Now watch the event bus. This time we get data about the entire minion connection process: first the `salt/auth` service authenticated the minion, then Salt refreshed the minion’s data cache (`minion/refresh/minion1`). The minion service was then triggered and started (`minion_start` and `salt/minion/minion1/start`).

The rest of the returned data is just that: ***event data***. Event data always contains a timestamp but is otherwise unique to the specific event that generated it. Notice that when we ran our `mysql.config` state, it contained information about each state module that was used, whereas when we restarted the `salt-minion` service, some events contained only a timestamp.

Event Types

Salt contains a number of default events that the event bus watches for and reports on, as well as optional events we can enable in our master configuration file. We can also set up and watch for custom events.

In this section, we’ll take a look at the different kinds of Salt events. Just like the “Components” lesson earlier in the course, this is going to be a conceptual rundown of event types.

salt/minion//start

This event is fired at minion start.

Try it out on a Salt minion:

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

salt/key

This event is fired when `salt-key` accepts, rejects, or deletes a key, or when a minion initiates a handshake.

Try it out on a Salt master with:

```
sudo salt-key -d minion2
```

Then, on `minion2`, restart the `salt-minion` service to initiate a handshake. Reaccept the key on the master.

salt/job//new

This event is fired when a new job is sent to the minions.

Try it out on a Salt master with:

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

salt/job//ret/

This event is fired when there is return data for a job.

Try it out on a Salt master with:

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

salt/job//prog//

This event is fired when each function in a state run has completed execution.

For the event to fire, we have to add the following to the master config:

```
state_events: True
```

Try it out on a Salt master with:

```
sudo salt 'minion*' state.sls mysql
```

salt/run//new

This event is fired when a runner begins.

Try it out on a Salt master with:

```
sudo salt-run pillar.show_pillar minion1
```

salt/run//ret

This event is hired when a runner has return data.

Try it out on a Salt master with:

```
sudo salt-run pillar.show_pillar minion1
```

salt/run//args

This event is fired when the `state.orchestrate` runner is used.

The following presence system events cannot be triggered manually.

salt/presence/present

This event is fired on a regular interval to check for currently connected, newly connected, or recently disconnected minions.

For the event to fire, we have to add the following to the master config:

```
presence_events: True
```

salt/presence/change

This event is fired when the presence system detects new minions or new disconnects.

The following events are for Salt Cloud. We won’t be able to demo these until later.

salt/cloud//creating

This event is fired when `salt-cloud` starts a virtual machine.

salt/cloud//deploying

This event is fired when a virtual machine is available and `salt-cloud` begins deploying Salt.

salt/cloud//requesting

This event is fired when `salt-cloud` sends a request to create a virtual machine.

salt/cloud//querying

This event is fired when `salt-cloud` queries a new instance.

salt/cloud//tagging

This event is fired when `salt-cloud` tags a new instance.

salt/cloud//waiting_for_ssh

This event is fired when a `salt-cloud` deploy is waiting for SSH on a new instance.

salt/cloud//deploy_script

This event is fired when a deploy script is finished.

salt/cloud//created

This event is fired when a new instance has been created.

salt/cloud//destroying

This event is fired when `salt-cloud` initiates the destruction of a minion.

salt/cloud//destroyed

This event is fired when the destruction of an instance is complete.

Beacons

In addition to Salt-related events, Salt can also monitor non-Salt processes using beacons. Beacons allow minions to monitor various processes and send reports to the event bus when activities occur, like when disk usage reaches a certain percentage or when a file is changed on the system. Beacons can ***only*** monitor processes and send events to the event bus. In order to trigger actions from these events, we have to use the reactor system, which we'll cover in the next section.

The beacon system currently contains over twenty modules, which can be used for everything from monitoring system metrics like CPU and disk space, to sending messages to the Telegram messaging application. The full list of modules can be found on Salt's website.

Create a Beacon to Monitor File Changes

To get some hands-on experience using beacons, let's set up the beacon system so that it triggers an alert whenever there are any changes to the MySQL configuration file we have managed by Salt. As with most minion configurations, to do this, we can either update our minion configuration, add a configuration file to the `/etc/salt/minion.d` directory, or save the information in our `pillar_roots`. Since we've made configuration changes to our minion configs more than once, let's add our beacon information to pillar.

1. Before we begin, we need to install the Python inotify connector on our minions:

```
 $ sudo salt 'minion1' pkg.install python-pyinotify
$ sudo salt 'minion2' pkg.install python-inotify
```

2. Navigate to the `pillar_roots` and create a `beacons.sls` file to store our beacon configurations:

```
 $ cd /srv/pillar
$ $EDITOR beacons.sls
```

3. To track a file in pillar, we want to use the `inotify` beacon. This beacon watches files for changes, then translates this activity into a Salt event.

```
beacons:
  inotify:
```

4. Next, we need to define the files we want to watch. We can use Jinja `if` statements to ensure the correct files are monitored on the correct operating system families:

```
  beacons:
    inotify:
      - files:
{% if grains['os_family']=="RedHat" %}
          /etc/my.cnf.d/server.cnf:
{% endif %}
{% if grains['os_family']=="Debian" %}
          /etc/mysql/mysql.conf.d/mysqld.cnf:
{% endif %}
            mask:
              - modify
```

The `modify` mask tells `inotify` to trigger only when the file is ***modified***. We could also change this to any time the file is viewed, deleted, moved, updated, etc.

5. Finally, since we're going to be using this beacon to tell the reactor system to reset the file whenever it's edited, we need to add a line preventing this beacon from triggering whenever a state that updates these files is run:

```
  beacons:
    inotify:
      - files:
{% if grains['os_family']=="RedHat" %}
          /etc/my.cnf.d/server.cnf:
{% endif %}
{% if grains['os_family']=="Debian" %}
          /etc/mysql/mysql.conf.d/mysqld.cnf:
{% endif %}
            mask:
              - modify
      - disable_during_state_run: True
```

6. Save and exit.

7. Map the beacon configuration to our minions:

```
$ $EDITOR top.sls
```

```
base:
   'minion*':
     - mysql
     - apache
     - beacons
```

8. Save and exit, then force a pillar refresh:

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

9. Now let’s check to make sure this works. On the master, open up the event bus:

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

10. In a different tab, log in to `minion1`, then open the `/etc/mysql/mysql.conf.d/mysqld.cnf` file and make an arbitrary change. View the output in the event bus:

```
salt/beacon/minion1/inotify//etc/mysql/mysql.conf.d/mysql
{
     "_stamp": "2018-06-11T12:51:34.552158",
     "change": "IN_IGNORED",
     "id": "minion1",
     "path": "/etc/mysql/mysql.conf.d/mysqld.cnf"
}
```

As expected, we get an event about the changes made to that file, noting the that was changed, the minion where the change took place, and a timestamp. If we repeat this with our other minion, we’ll get a similar result:

```
  salt/beacon/minion2/inotify//etc/my.cnf.d/server.cnf
     "_stamp": "2018-06-11T12:51:08.192400",
     "change": "IN_IGNORED",
     "id": "minion2",
     "path": "/etc/my.cnf.d/server.cnf"
}
```

Now that our beacon is set up, we can apply this trigger to reactor to enforce our states and revert changes.

Reactor

Now we can monitor non-Salt processes with Salt. But what benefits does this give us? Well, we can now use Salt’s reactor system and beacons to perform actions based on events. The reactor system monitors the event bus, looking for certain event tags based on our defined parameters. These tags are then associated with reactor SLS files that define what actions we want Salt to take when events occur.

Currently, there are four types of reactions we can use: `local` or `remote execution`, `runner`, `wheel`, and `caller`.

A `local` (or `remote execution`) reaction lets us use Salt's remote execution system to run execution modules, including highstates and state runs. When using the `local` reaction type, we provide basically the same information as we do when running an execution module: we supply a target, a function, and arguments. For example, the following reactor SLS installs the `vim-enhanced` package:

```
restore_vim:
  local.pkg.install:
    - tgt: 'os_family:RedHat'
    - args:
      - name: vim-enhanced
```

`Runner` reactions let us use a Salt runner to respond to an action. We haven't discussed runners yet, but here's an example of a runner that shows us pillar data for `minion1`:

```
$ sudo salt-run pillar.show_pillar 'minion1'
```

And here it is translated to a reactor SLS:

```
show_pillar:
  runner.pillar.show_pillar:
    - args:
      - minion: minion1
```

Next, `wheel` reactions let us run functions specifically from Salt's `wheel` module, which is used to manage configuration files, generate errors, read `file_roots` and `pillar_roots`, use the `salt-key` system, and view connected minions. For example, we can delete any Salt keys matching the `dev*` ID with:

```
remove_dev_minions:
  wheel.key.delete:
    - match: 'dev*'
```

Finally, `caller` reactions also let us use execution modules, only this time, these modules are run from the reactor engine. This is required for masterless minions. Here's the same `vim-enhanced` example from above, modified for use with a `caller` reaction:

```
restore_vim:
  caller.pkg.install:
    - args:
      - name: vim-enhanced
```

Notice how it doesn't name any targets.

To use reactions, we need to set up a reactor config as part of our master configuration. Reactor SLS files should be added to `/srv/reactor/`.

Create a Reactor Directory

1. Create `/srv/reactor`:

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

2. Set the directory so that all users in the `salt` group can add and edit files:

```
$ sudo chgrp salt /srv/reactor/
$ sudo chmod 775 /srv/reactor/
$ sudo chmod g+s /srv/reactor/
```

3. Log out, then log back in, or run `bash` to refresh the terminal.

Create a Reaction to Reset Config Files

Creating reactions is a two-step process. First, we need to map the reactor SLS file to the event tag that triggers it in our master config, and then we need to create the reactor SLS within our reactor directory. Note that although reactor files are similar to state files, they are not the same.

1. Navigate to the reactor directory:

```
$ cd /srv/reactor
```

2. We can either add a file directly to this directory or store our files under organized sub-directories. Since we're specifically working to restore our MySQL config file, let's create a `mysql` directory, then navigate to it:

```
$ mkdir mysql
$ cd mysql
```

3. Create and open `deb_config.sls`:

```
$ $EDITOR config.sls
```

Although we could create a single configuration file for both of our operating systems, it would involve some advanced regex in the reactor configuration file. To keep things simple, we'll keep them separate.

4. Now we can set up a reactor SLS to re-enforce our MySQL server configuration file on our Debian-based servers. Using the `local` reactor type, let's first define our targeted minions:

```
restore_deb_mysql_server_config:
  local.state.single:
    - tgt: 'E@minion* and G@os_family:Debian'
    - tgt_type: compound
```

Notice that we're using the `state.single` function. This lets us run a single state function against our target.

5. Add the arguments for the `file.managed` function:

```
restore_deb_mysql_server_config:
  local.state.single:
    - tgt: 'E@minion* and G@os_family:Debian'
    - tgt_type: compound
    - args:
      - fun: file.managed
      - name: /etc/mysql/mysql.conf.d/mysqld.cnf
      - source: salt://mysql/files/deb_mysqld.cnf
```

6. Save and exit.

7. Repeat this process for `rh_config.sls` (a RedHat version of the above SLS):

```
restore_rh_mysql_server_config:
  local.state.single:
    - tgt: 'E@minion* and G@os_family:RedHat'
    - tgt_type: compound
    - args:
      - fun: file.managed
      - name: /etc/my.cnf.d/server.cnf
      - source: salt://mysql/files/rh_server.cnf
```

8. Now we need to set up our master config for our reactors. We're going to store this at `/etc/salt/master.d/reactor.conf`:

```
$ $EDITOR /etc/salt/master.d/reactor.conf
```

9. This is where we map out which event tag causes which reaction. In our case, we want our Debian SLS to run when the Debian file is updated, and the Red Hat SLS to run when the Red Hat MySQL server config changes:

```
reactor:
  -
'salt/beacon/*/inotify//etc/mysql/mysql.conf.d/mysqld.cnf
    - /srv/reactor/mysql/deb_config.sls
  - 'salt/beacon/*/inotify//etc/my.cnf.d/server.cnf':
    - /srv/reactor/mysql/rh_config.sls
```

Notice that we replace the minion names with wildcards in the event tag.

10. Save and exit, then restart the `salt-master` daemon:

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

11. Now we want to test that our reactions work. On the master, connect to the event bus:

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

12. In a new terminal, `ssh` into `minion1` and edit the MySQL server config file with an arbitrary change:

```
$ sudo $EDITOR /etc/mysql/mysql.conf.d/mysqld.cnf
```

13. Watch the event bus. After the initial beacon event, a couple of things happen. First, our reaction is triggered, then our `file.managed` job starts running.
14. Reopen the configuration file on `minion1`. Our changes are now gone.
15. Close `minion1`, then repeat the process on `minion2`:

```
$ sudo $EDITOR /etc/my.cnf.d/server.cnf
```

16. View the event bus. Again, reactor has responded.
17. Finally, reopen the `server.cnf` configuration file. Our changes have been reset.

When used together, the beacon and reactor systems can help us create a truly self-healing, event-driven architecture that requires minimal admin intervention for mundane tasks.

LEARNING ACTIVITY: USING BEACONS AND REACTOR

To get hands-on and practice your skills, log in to Linux Academy and complete the Using Beacons and Reactors Learning Activity.

Chapter 6: Runners and Orchestration

Runners

In the last chapter, I briefly mentioned that Salt's event bus works with a component we had not yet discussed: Salt runners. Salt runners are convenience applications; that is, they provide utilities that are helpful for Salt management tasks, like viewing active or recently completed jobs, or even orchestrating Salt runs so that jobs are executed in the desired order. It's easy to tell when we're using Salt runners because they are used alongside the `salt-run` command.

Salt contains a number of runners by default. The list can be viewed on the Salt website and includes utilities that allow us to do things like manage authentication, retrieve cached data, display documentation, manage Salt jobs, work with pillar, and even run execution modules through the Salt master (remember, Salt generally sends information to minions, and the minions do the work themselves). We can also write our own Salt runners, but that is out of scope for this course and not on the SSCE.

You may have noticed that we've already used runners; when we viewed our event bus, we used `salt-run state.event pretty=true`. If we parse this, we can see that runners actually work a lot like execution and state modules. There's the runner itself, `state`, which lets us perform orchestration actions, and the function, `event`. In this case, we know that the `event` function connects us to the event bus.

Just like with execution and state modules, there's an easy way for us to view documentation related to our runners. Instead of an execution module, we simple use the `-d`, `--doc`, or `--documentation` flags. For example:

```
$ sudo salt-run -d state.event
```

Or, to view all our `state` runner functions:

```
$ sudo salt-run -d state
```

The `state` runner is the one we're going to use the most in this course; we have an entire section dedicated to it. We're also going to spend an entire lesson working with the `jobs` runner. But before we get into the runners themselves, let's check out some quick and easy `salt-run` commands we can use.

First, we can view a list of connected minions through Salt's presence detection with:

```
$ sudo salt-run manage.present
```

This doesn't send any commands to our minions; it just informs us they're there.

We can view information from our Salt mine:

```
$ sudo salt-run mine.get 'minion*' network.ip_addrs
```

Or see the `top` file for our pillar data:

```
$ sudo salt-run pillar.show_top
```

We can even add reactions via the `runner` module. Here's how we can add the one we used in our previous lesson:

```
$ sudo salt-run reactor.add
'salt/beacon/*/inotify//etc/my.cnf.d/server.cnf'
reactors='/srv/reactor/mysql/rh_config.sls'
```

Luckily, Salt can tell this reaction already exists, so we get an error.

Next, let's take a deeper look at managing jobs with the `jobs` runner.

Jobs

We know that when we perform an action in Salt, a job occurs. We saw a feed of jobs every time we ran an execution module or state while viewing our event bus. With the `jobs` runner, we can further view and manipulate jobs. We can also use the jobs scheduler to assign highstates, state runs, runner runs, and more at specific times or using specific credentials.

Let's start with the basics. By default, Salt caches jobs every 24 hours. To see a list of recently run jobs, we can use:

```
$ sudo salt-run jobs.list_jobs
```

We could also view currently running jobs, but our test environment doesn't have enough going on for this to report much. However, if you have a larger environment, you can view a list of active jobs by running:

```
$ sudo salt-run jobs.active
```

Finally, we can look up jobs by ID using:

```
$ sudo salt-run jobs.lookup_jid <JOBID>
```

Take one of the IDs from the earlier `jobs.list_jobs` command and plug it in. The lookup returns the same data we got when we ran the command on the minion. Here's what I got for a `test.ping`:

```
$ sudo salt-run jobs.lookup_jid 20180612182718526124
minion1:
    True
minion2:
    True
salt:
    True
```

But we can use the `jobs` runner to do a lot more than simply query for results on the command line. We can also use it to schedule all kinds of jobs, including highstates, execution modules, and even runners themselves.

We can enable scheduling in a number of ways: through the master or minion configuration files (using the `schedule` setting), through pillar, via the `schedule` state module, or through the `schedule` execution module.

Schedule a Highstate

1. To see this in action, let's schedule a highstate. This will highstate our servers once every minute (if that seems like a lot, don't worry; we'll remove it shortly).

   ```
   $ sudo salt 'minion*' schedule.add highstate-1
   function='state.highstate' seconds=60
   ```

   In this instance, `highstate-1` is the name we gave to the job. (The name can be anything we want.)
2. Now we can connect to our event bus and wait:

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

   After a minute, we'll start to receive information about the jobs run during our highstate. As expected, enforcing our states once every minute is a little excessive.
3. Close the event bus by typing CTRL+C and remove the scheduled job:

   ```
   $ sudo salt 'minion*' schedule.delete highstate-1
   ```

This is just one way of setting a schedule. If we wanted to do this using pillar data or a minion config we would use:

```
schedule:
  highstate:
    function: state.highstate
    minutes: 30
```

This time, we scheduled the highstate for a more reasonable 30-minute interval.

When scheduling a highstate in pillar, remember to map it to the appropriate minions:

```
base:
  'minion*':
    - mysql
    - beacons
    - schedule
```

Orchestration

So far, we’ve enforced states and configurations across multiple servers through highstating and the `state.sls` module. However, when working across large systems and provisioning numerous servers, we often need to configure components in a specific order. For example, if the configuration of a web application on our web servers depends on the presence of a MySQL database for setup, that database needs to be configured first. However, aside from running states in batches, we can’t always guarantee that our databases will finish configuration before the web servers do. This is where Salt’s orchestrate runner comes in.

Unlike a highstate or `state.sls`, when we use the orchestrate runner, our state is not run concurrently and independently on each minion. Instead, the master does the work, maintaining a high-level view of what is happening across all minions so that the state requirements are met across all systems. Where a highstate works at a minion level, `state.orchestrate` works on the infrastructure level, ensuring that commands are run in the correct order across all minions or a defined group of minions.

With the orchestrate runner, we can run execution and state modules, highstate our minions, and use runners, beacons, and reactions to complete overarching tasks. But first we need a place to store our orchestrate files. Generally, orchestrate files are stored in the `/srv/` directory or in `file_roots`. We’re going to store our orchestrate files directly in our `file_roots`:

```
$ cd /srv/salt/
$ mkdir orch
$ cd orch
```

Orchestrate files are very similar to some of the other Salt components we've used so far. Like states and reactor SLS files, orchestrate files can use Jinja and pillar, and like reactor SLS files, they often include targets. However, unlike formulas, orchestrate files often contain many states or formulas in a single file. Because orchestrate SLS files have to deal with the infrastructure at a higher level than states and formulas, the inclusion of multiple tasks makes sense.

To get started with orchestration, we first need to ensure our hostnames are set and then install MySQL on `minion2`. When the MySQL installation is complete, we want to install both Apache and the MySQL client on `minion1`. This may seem basic, but don't worry; we'll do some more advanced work with the orchestrate runner when we learn how to use Salt Cloud in a future lesson.

Create an Orchestrate SLS

Before we begin, pull in the Apache formula from the Learning Activities:

```
$ git clone -b final
https://github.com/linuxacademy/content-ssce-apache.git
/srv/salt/apache
$ curl -o /srv/pillar/apache
https://raw.githubusercontent.com/linuxacademy/content-ssce-
files/master/apache.sls
$ $EDITOR /srv/pillar/top.sls

base:
  'minion*':
    - mysql
    - beacons
    - schedule
    - apache
```

1. Create and open a file to store our orchestration work. We'll call it `setup.sls`:

```
 $ $EDITOR setup.sls
```

2. We'll go through some different options for setting up our

infrastructure using Salt orchestration. To start, let's consider how we can set the hostname for `minion1`. Since we don't have our `hostname.sls` file set up for `minion1`, we can write a new one using the `salt.function` module:

```
set_hostname:
  salt.function:
    - name: network.mod_hostname
    - tgt: 'minion1'
    - arg:
      - minion1
```

When we do this, we're essentially using an execution module in the orchestrate file itself, the same way we did when working with reactions. Notice that we provide a function, a target, and arguments, just like we did in previous sections.

3. Next, we want to run a highstate on `minion2`. To do this with our orchestrate file, we'll use the `salt.state` module, but instead of providing state information, we'll use the `highstate` parameter:

```
configure_db_minion:
  salt.state:
    - tgt: 'minion2'
    - highstate: True
```

4. Finally, we can use that same `salt.state` function to list the SLS files we want, like this:

```
configure_web_minion:
  salt.state:
    - tgt: 'minion1'
    - sls:
      - apache
      - mysql.client
```

Again, notice that we always need to provide a target.

5. Save and exit.

6. We can now use our `setup` SLS alongside the `state.orchestrate` (or `state.orch`) runner to use our `setup` orchestration. We call the file as we would any other state in our `file_roots`, using `orch.setup`:

```
$ sudo salt-run state.orch orch.setup
```

Note that this is just ***one*** way of accomplishing our goals. You can probably think of a better way!

LEARNING ACTIVITY: USING SALT ORCHESTRATION

To get hands-on and practice your skills, log in to Linux Academy and complete the Using Salt Orchestration Learning Activity.

Chapter 7: Salt SSH

Salt SSH Setup

Up until this point, we have worked exclusively with a master-minion setup. However, Salt lets us run commands on servers that do not have the `salt-minion` installed. This is done via SSH and allows us to work with unsalted systems through the use of the `salt-ssh` command. As we’ll discover later, using `salt-ssh` isn’t much different than using the `salt` command from a user standpoint; the commands are essentially identical. Rather, it’s the behavior of Salt that changes.

When using Salt SSH, gone is the ZeroMQ data bus. Remember, we don’t install any Salt components on the minions, so all communication is done through SSH only. Although this can slow down state runs and execution modules significantly, it offers an alternative way to manage systems in Salt.

Salt SSH is not installed alongside the `salt-master`, so if we want to use it, we need to install and configure it separately. Salt SSH is also not included in the bootstrap script; however, we can install it via the `apt` command. When we ran the `bootstrap-salt.sh` script, SaltStack’s repositories were added to our repolist, and we can use them:

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

But now we need a Salt-less server to work with. Since we don’t really need two minions anymore, I removed `minion1` from my cloud server and re-created it. You can do the same, or if you have extra cloud servers and don’t mind using more than half of your allotted servers for a single course, you can use one of those. You can also remove `minion2`. However, I wanted to leave a CentOS 7 machine available. My re-created server uses Ubuntu 16.04.

Prepare the Saltless Server

Although we don’t have to install Salt on the minion itself, we do have to do a bit of prep work. We need to log in to our cloud server and set the password when prompted at initial login, then we need to update our sudoers file so that Salt can run commands as `sudo` without issue. Since we can’t log in as `root`, Salt will be using our `user` user.

```
$ sudo su -
$ visudo

user    ALL=(ALL) NOPASSWD:ALL
```

If you can log in as `root` via SSH, you do not need to make these changes. Linux Academy's cloud servers have this disabled. It is also generally considered best practice to turn off SSH `root` login.

Set Up the Salt SSH Roster

So how does Salt know what servers to `ssh` into? We need to use the `/etc/salt/roster` file, which was added after we installed Salt SSH. As with most of our configurations, the file is written in simple YAML.

To add a server to the roster, we first need to give the server an ID. Since our new minion was created specifically to work with Salt SSH, I just used a designation of `ssh_minion` (although this would usually be something more specific):

```
$ sudo $EDITOR /etc/salt/roster

ssh-minion:
```

After this, we need to supply some parameters. At the very least, we always need to provide the host location (be sure to use your server's own private IP):

```
ssh-minion:
  host: <PRIVATE IP>
```

However, there are a number of other options we want to consider, including `user`, `passwd`, `port`, `sudo` (which always runs Salt with `sudo` privileges when using a non-root user), `sudo_user` (gives escalated privileges to a specific user), `tty` (for systems where `sudo` is set to `True` and `require tty` is set), `priv` (to set the private key for login), `timeout` (for establishing the SSH connection), `minion_opts`, `thin_dir` (for Salt-related temporary file storage on the SSH minion), and `cmd_umask` (for instances using `salt-call`).

Since we can't log in via `root`, we want to use the `user` option. We also want to use the `sudo` option so we can run any commands that require escalated privileges.

```
ssh-minion
```

```
  host: <PRIVATE IP>
  user: user
  sudo: True
```

At this point, you may be wondering why we're not supplying any password or private key information. This is because, by default, Salt will generate and supply its own SSH key for use on our SSH minions. However, for this to be generated, we need to attempt to run a `salt-ssh` command. Save and exit the `roster` file, then try out a simple `test.ping` using `salt-ssh`:

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

As expected, we received an error because our SSH public key is not yet on our minion. We can fix this with `ssh-copy-id`. By default, the Salt SSH key is located at `/etc/salt/pki/master/ssh/salt-ssh.rsa.pub`. To use something different, use the `priv` argument from your roster.

```
$ sudo sh-copy-id -i /etc/salt/pki/master/ssh/salt-
ssh.rsa.pub user@<PRIVATE IP OF SSH MINION>
```

With our key added, we can try testing our SSH minion again:

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

Now we can use the Salt SSH minions just like we would any other minions, with a few exceptions.

Using Salt SSH

In our last lesson, we learned that Salt SSH allows us to use our states and modules as though our servers were "proper" Salt minions with the `salt-minion` daemon installed. However, there are some important differences between regular minions and Salt SSH minions. Most notably, Salt SSH doesn't support many of the targeting options we've become accustomed to.

Currently, Salt SSH can only target in two ways: through globbing and through regular expressions. We can't target using grains, nodelists, pillar data, subnet, or SECO range. Of course, we can always target using our assigned minion names, too.

Minion configurations do not persist through reboots in Salt SSH. Salt only stores its data in the `/tmp/` directory of the minion, so it will clear upon minion shutdown. This means that custom grains will need to be re-added upon reboot, along with any mine data or schedules. When possible, try to save these settings to pillar so you can easily reconfigure your Salt SSH minions if you end up using Salt SSH in your infrastructure.

Now let’s get hands-on and make some changes to our `ssh-minion`.

Work with Salt SSH

When we created our SSH minion, you may have noticed that we did almost nothing to it during the configuration stage, including the usual things like setting a hostname. Let’s take this time to get our minion in a more appropriate state.

1. Remember that we can use execution modules alongside Salt SSH, just as we would with any other minion. Let’s set our minion’s hostname:

```
 $ sudo salt-ssh 'ssh-minion' network.mod_hostname ssh-minion
```

2. But what about using our formulas? First, we need to update our pillar `top` file, just like we would with any other minion. Let’s add the `mysql` pillar data:

```
$ $EDITOR /srv/pillar/top.sls

  'sshminion':
    - mysql
```

3. We can also include our `ssh-minion` in our regular `top` file located within our `pillar_roots`:

```
$ $EDITOR /srv/salt/top.sls

  'sshminion':
    - mysql
```

4. Now, let’s run our MySQL formula:

```
 $ sudo salt-ssh ssh-minion state.sls mysql
```

As we can see, this works just like it did when it ran against our `minion*` minions, without any additional work beyond the initial server setup. Salt SSH also had no issues working with the file server (remember, our MySQL formula uses the `salt://` path in our configuration state).

LEARNING ACTIVITY: USING SALT SSH

To get hands-on and practice your skills, log in to Linux Academy and complete the Using Salt SSH Learning Activity.

Chapter 8: Salt Cloud

Salt Cloud Setup

Salt isn’t restricted to managing existing servers. If we plan on working with multiple cloud platforms, we can use Salt to provision additional servers straight from our master, bring them under Salt’s control, then use them just like the minions we’ve spun up by hand.

A full list of supported Salt Cloud providers can be found in the SaltStack docs.

To use Salt Cloud, we first have to install it. Since the Salt bootstrapping process added SaltStack repos to our repolist for us, all we have to do is run:

```
sudo apt-get install salt-cloud
```

For this lesson, we’ll be using AWS as our cloud provider. If you do not already have an AWS account, create one now. Everything we’ll do in this section will use the free tier of AWS.

Prepare AWS

We need to configure the following in AWS:

- A default VPC
  - Created by default.
  - If you’ve removed your default VPC, you can re-add it in the VPC dashboard under Your VPCs > Actions > Create Default VPC.
- A default security group accepting inbound SSH connections
  - Under the VPC dashboard, check Security Groups, click on the default group, then check the Inbound Rules tab.
- An access ID and key
  - To create your access key, go to the IAM dashboard, click on Users, select your user, then click the Security Credentials tab.
- A key pair
  - I named mine `salt-cloud.pem`.

Once you’ve finished setting this up, return to your Salt master.

Provision an EC2 Instance

1. First, we want to make sure Salt has access to the key pair it needs to

connect to any of our Salt servers. Since I’m on a Unix-based machine, I used `scp` to copy it into my home directory. Windows users should use a file client. Once the key is on your Salt master, move it somewhere safe. There is no default directory in which these need to be stored, so I added mine to our `pki` directory with the rest of our Salt-related keys:

```
 $ sudo su -
$ cd /etc/salt/pki/master
$ mv /home/user/salt-cloud.pem .
```

We also need to make sure it has the correct permissions:

```
 $ chmod 400 salt-cloud.pem
$ exit
```

2. Now we can get to work defining our provider file. This file contains information for using EC2: things like our preferred VPC, our access keys, and that key file we added. All provider files ***must*** be saved to `/etc/salt/cloud.providers.d/` and end with `.conf`. We’ll call ours `ec2.sls`:

```
  $ sudo $EDITOR ec2.conf'
```

As usual, we first need to give it a name:

```
  ec2-web:
```

Then, we provide the relevant settings:

```
 ec2-web:
   driver: ec2
   id: '<YOUR ACCESS ID>'
   key: '<YOUR ACCESS KEY>'
   private_key: /etc/salt/pki/master/salt-cloud.pem
   keyname: salt-cloud
   securitygroup: default
```

Note that the `driver`, `private_key`, `keyname`, `id`, and `key` information ***must*** be provided (unless you provide an SSH password instead of the key information). All other settings will automatically use their default settings.
Save and exit when you’re finished.

3. Next, we need to add profile information. Profiles are descriptions of EC2 servers with information such as AMI ID, provider information, SSH username information, and server size. It’s also where we store our minion config information, including the location of our master. Multiple profiles can be saved to one file.

Like provider files, profile files must be saved with the `.conf` file extension in a particular location (this time: `/etc/salt/cloud.profiles.d/`). We'll call this file `ec2.conf`:

```
$ sudo $EDITOR ec2.conf'
```

We're going to call our profile `tiny-server` because it should bring up a tiny server for us to use:

```
tiny-server:
```

Next, we want to associate the profile with a provider file, then define the image, SSH information, and size:

```
tiny-server:
  provider: ec2-web
  image: ami-a4dc46db
  ssh_username: ubuntu
  size: t1.micro
```

The information provided above provisions an Ubuntu server using a free-tier AMI. `ubuntu` is the default username provided in this image.

Finally, we want to add our minion configuration information. Specifically, we want to define our Salt master. We do have a small problem, however. Since we're using Linux Academy cloud servers, we cannot use our private IP address or `salt` as the master name. Instead, we need to use either our public IP address (which resets on reboot) or, more logically, one of our provided hostnames. This can be found on the cloud server page:

```
tiny-server:
  provider: ec2-web
  image: ami-a4dc46db
  ssh_username: ubuntu
  size: t1.micro
  minion:
    master: <PUBLIC HOSTNAME>
```

Save and exit the file.

4. Now we're ready to provision a server. To do so, all we have to do is run `sudo salt-cloud -p <PROFILE> <MINION1NAME> <MINION2NAME> <ETC>`. Let's only provision one server right now:

```
$ sudo salt-cloud -p tiny-server web-test
```

Now we can watch our client go through the provisioning process. Note that this may take several minutes.

5. When it's finished, we can verify that the process worked by checking our Salt keys:

```
 $ sudo salt-key -L

Accepted Keys:
salt
web-test
```

Notice that our new server was automatically pulled under our Salt master.

6. We can also delete our servers with:

```
 $ sudo salt-cloud -d web-test
```

Now we can continue to use Salt as we normally would. The only difference is that we can now manage our cloud servers on the provider end in addition to managing what happens on them.

Final Exam Review Notes

We're almost finished with the course! All that's left is the practice exam. Back when we first started this journey, I mentioned that the exam consists of 80 multiple choice questions; our practice exam is the same length.

While taking the SaltStack Certified Engineer exam, you can use books and the internet. I particularly encourage you to keep the lists of execution and state modules up, as well as versions of the master and minion configurations. However, despite being open-book, you cannot and ***should*** not talk during the exam or ask the people around you questions — as SaltStack says, the exam is not "open-friend."

The exam is one hour long, and you can review the questions after answering them. As such, I encourage you to go through and answer the questions you feel confident about first, give your best answers to the rest, and then take the rest of your time to review the harder questions and the ones you need to spend a little extra time on.

Looking for more practice before you take the exam? Try creating more formulas, adding more beacons and reactions, and using and monitoring the event bus.

Good luck, and study hard!

Congratulations! We've reached the end of the course! It's been a long journey, but you've learned so much. From that first Salt install to the orchestrate state we built to provision a full LAMP stack, we’ve written formulas, run execution modules, used the event system to set up responsive, self-healing infrastructure, and learned the ins and outs of how Salt works. You now have the skills to not only pass the SSCE, but also to manage Salt in the real world.

If you’re interested in further developing your Salt skills, I suggest taking our Python 3 course, Python 3 For System Administrators. As you know, Salt is written in Python, and if you want to venture into more advanced Salt features like custom execution modules, learning Python is a great place to start.

Happy learning!

About the Author

Hello there! I’m Elle, your friendly neighborhood training architect for this course. I’m a former technical writer with a focus on documenting Linux systems and the DevOps toolchain. I’ve previously worked at a cloud hosting company before I joined Linux Academy almost three years ago to produce our written content and ensure our spelling was on point.

I’ve been using Salt for over four years and became certified at SaltConf ’17 in Salt Lake City, Utah (a conference I highly recommend to any DevOps and systems professionals using Salt).

Besides wrangling servers and sentences, I enjoy writing fiction, reading anything that has words, making things (Arduino and dresses are equal passions of mine), and playing video games.

You can contact me on Twitter(@ellejaclyn) or on LinkedIn (/ellejaclyn).