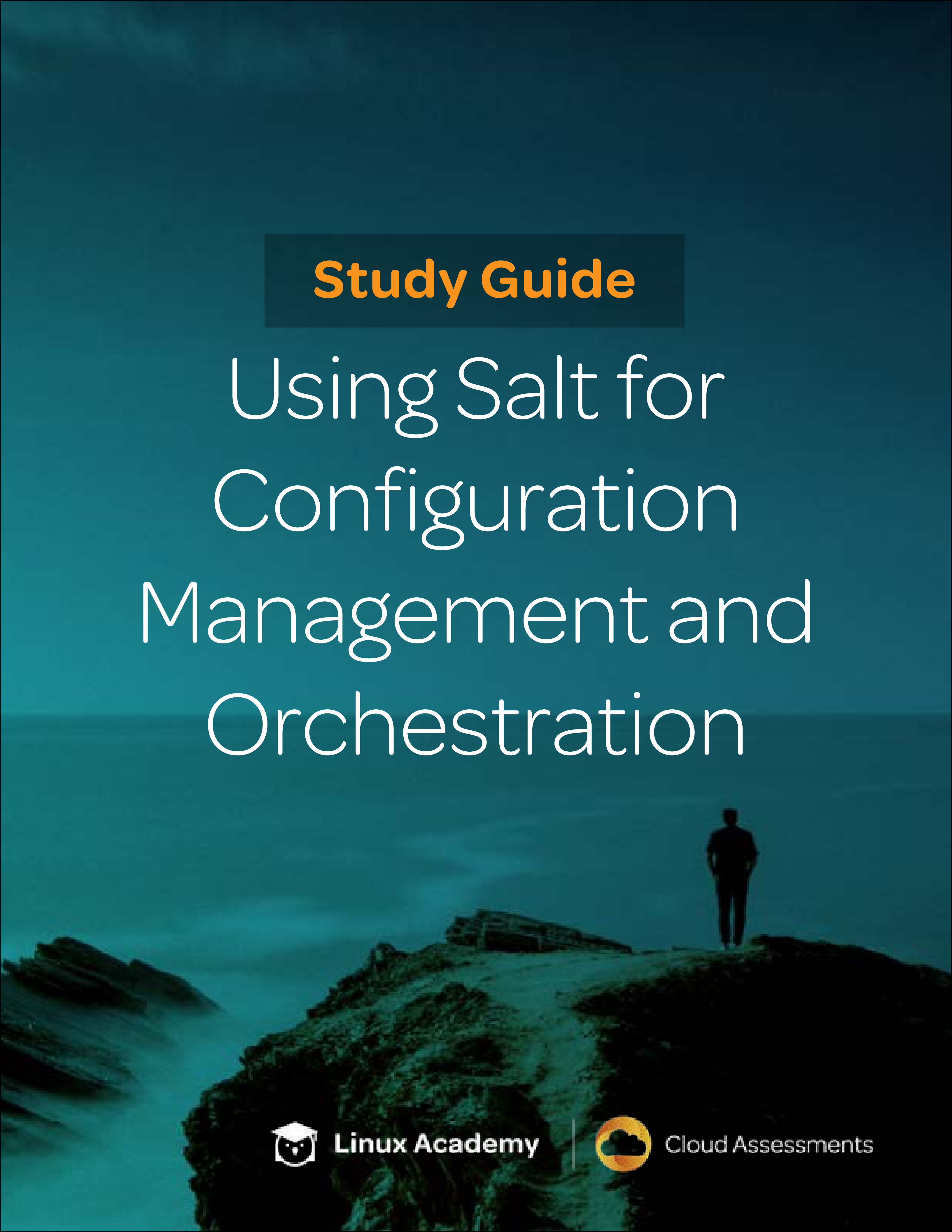
Study Guide

Using Salt for Configuration Management and Orchestration

Linux Academy | Cloud Assessments



Contents

Salt Concepts

Remote Execution and Configuration Management

- **Remote execution**: The process of running command on multiple servers at once; often referred to as "orchestration"
- **Configuration management**: The process of keeping systems consistent within their defined policies
- Historically, when making mass changes to servers, each change had to be done on each server, or the systems administrator would have to create their own script

Imperative Configuration Management

- Tells the system what to do and how to do it
- Done with Salt execution modules
- Example Salt execution module:

```
salt '*' user.add dana home=/home/dana shell=/bin/bash
```

Declarative Configuration Management

- Tells the system how you want it to look AT THE END; "describe" the system
- Done with Salt states
- Example Salt state:

```
dana:
  user.present:
    - home: /home/dana
    - shell: /bin/bash
```

- Notice that the state uses user.present ("is the user there?") while the execution module uses user.add ("we are going to add the user with this command")

Grains (Overview)

- Static data about the system, such as:
    - Operating system
    - Kernel version
    - IP address(es)
    - Networking interfaces
- Can create custom grains
    - Suggested for information like server role, datacenter location, or cabinet number
- Grains are not output in a case-sensitive fashion
    - XFILE = xfile = Xfile
- Grains are refreshed during Salt state runs (“highstating”)
- Can be used in targeting

Pillar (Overview)

- User-defined variables
- Often used for:
    - Setting variable file paths
    - Configuration parameters
    - Key/value pairs
    - Passwords
- Stored on Salt Master, can only be accessed by minions
- Can be used with encrypted items

YAML (Overview)

- Stands for “YAML Ain’t Markup Language”
- A human-readable data serialization format
  - This just means it’s a way of feeding a machine data that human can still easily understand!
- YAML can include:
  - Strings
  - Integers
  - Floating-point decimals
  - Lists
  - Associative arrays
  - Hashes

YAML with Salt

- Used for Salt states
- Can be used alongside Jinja for templating
- Formatting breakdown:
  - Indent two spaces
  - Keys end with colons
  - Values begin with dashes (-) and are indented
  - Lists are also represented with dashes

- Example Salt state using YAML:

```
dana:
  user.present:
    - name: dana
    - fullname: Dana Scully
    - home: /home/dana
    - uid: 2015
    - shell: /bin/bash
```

- Example breakdown:
    - **Line 1** - dana: - Reference ID that other states can use when calling this state
    - **Line 2** - user.present: - The state function, which is an existing Dalt function that can be used across multiple distributions; this one specifically ensures the user is present
    - **Lines 3-7** - A list of configuration options; these options are determined by the state function

Installing and Configuring Salt

Overview

Single-Master Setup

- One master manages all minions
- Concerns:
    - Not highly-available
    - Multiple minions trying to query the master at once can result in a “thundering herd” problem, in which the queries back up
        - This can also be fixed by tweaking the Master’s configuration settings

Multi-Master Setup

- Two options:
  - Redundant setup
    - Multiple masters running “hot” (accepting queries from minions)
    - Any minion can access any master
    - All masters must share the same public key
    - All minions must be added to all masters
    - All masters must be added to the config of all minions
    - file_roots and pillar_roots must be synced
    - Does not sync masters automatically; must be set up separately
  - Failover setup
    - Single master runs “hot”
    - One or more backup masters to take over should “hot” master fail
    - Minions query for master > master fails > minions query next master; process repeats until a working master is found
    - Can be used with PKI-based authentication to check public keys of new master servers

Masterless Setup

- Minion can run states against itself and run salt-call command (without –local parameter)
- Update file_client in /etc/salt/minion to local
- Add file_roots:

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

- Suggested for:
    - Testing before running a state or module against production servers
    - Salting the master itself

Bootstrapping the Master

- Works on multiple platforms
    - Determines target distribution, then automatically selects best install method
    - Supported platforms:
        - Debian and Debian-based
        - CentOS and CentOS-based
        - SUSE
        - Arch Linux
        - Gentoo
        - BSD
        - SmartOS
- Use `curl` to add install script:

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

- Run the script with the `-P` and `-M` flags to install as a master
    - `-P` - Allow `pip` to be used during install
    - `-M` - Install Salt Master

```
sudo sh install_salt.sh -P -M
```

- Update `/etc/hosts` file so `127.0.0.1` is associated with the `salt` hostname:

```
127.0.0.1 salt localhost
```

Bootstrapping a Minion

- Works on same platforms as bootstrapped master
- Use curl to download script:

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

- Run script with -P flag to allow for the use of pip:

```
sudo sh install_salt.sh -P
```

- Update /etc/hosts to map the IP address of master server to salt:

```
\<MASTER IP> salt\
```

Install Salt via Repo (Ubuntu 16.04)

- Add the repository key:

```
wget -O - https://repo.saltstack.com/apt/ubuntu/16.04/amd64/
latest/SALTSTACK-GPG-KEY.pub | sudo apt-key add -
```

- Create the /etc/apt/sources.list.d/saltstack.list file, and add the following line:

```
deb http://repo.saltstack.com/apt/ubuntu/16.04/amd64/latest xenial
main
```

- Run an update:

```
sudo apt-get update
```

- Install any desired components:

```
sudo apt-get install <component-name>
```

- Component options:
    - salt-minion
    - salt-master
    - salt-ssh
    - salt-syndic
    - salt-cloud
    - salt-api

Install Salt via Repo (CentOS/RedHat)

- Add repository and repository key:

```
sudo yum install https://repo.saltstack.com/yum/redhat/salt-repo-
latest-2.el7.noarch.rpm
```

- Reevaluate the cache:

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

- Install any desired components:

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

- Component options:
    - salt-minion
    - salt-master
    - salt-ssh
    - salt-syndic
    - salt-cloud
    - salt-api

Key Management

- Once Salt is installed on your master and minions, we need to accept the minion keys, so that the minions can communicate with the master

- Perform the following for each minion:

- On the master, copy the master.pub key provided in the output of the following command:

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

- On the minion, paste this master.pub key onto the master_finger line found in the /etc/salt/minion configuration file:

```
 master_finger:
'97:65:f8:d6:1e:cd:98:5f:f8:6e:a7:db:e3:3a:12:eb:f9:10:de:4d'
```

- Restart the salt-minion process:

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

- Still on the minion, view the local key:

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

- Return to the master, and view the fingerprints of the available minions:

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

- Compare the minion's fingerprint with the fingerprint given from the minion on the master server (via the salt-key command output)

- If the keys match, accept the minion:

```
sudo salt-key -a <MINIONID>
```

- salt-key command flags:
  - -a - Accept defined key
  - -A - Accept all
  - -d - Delete defined key
  - -D - Delete all keys
  - -F - Show all keys with fingerprints
  - -l [pre|un|unaccepted|acc|accepted|rej|rejected|all] - List unaccepted (pre|un|unaccepted), accepted (acc|accepted), rejected (rej|rejected), or all (all) keys
  - -L - List all keys
  - -r - Reject defined key
  - -R - Reject all keys

Renaming Minions

- On the master, remove the key under the original minion name:

```
sudo salt-key -d <OLDMINIONID>
```

- On the minion, remove the minion's existing keys, located at /etc/salt/pki/minion/minion.pem and /etc/salt/pki/minion/minion.pub (must be done as root):

```
cd /etc/salt/minion/
rm minion.pem minion.pub
```

- Update the /etc/salt/minion_id file with the new minion's name
- Restart the salt-minion service:

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

- Back on the master, accept the key for the minion under its new name (after checking that it has the appropriate fingerprint)

```
sudo salt-key -a <NEWMINIONID>
```

Execution Modules

Introducing Execution Modules

- Execution modules allow you to run commands across multiple servers at once
- The basis for Salt States
- Anatomy:

```
salt '<target>' <function> [arguments]
```

  - salt - The core Salt command
  - - On what servers we want the function to run
  - - The execution module and action
  - [arguments] - Additional information and configurations that are related to the function

Examples

- With no arguments (checking that serves are responding):

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

- With a space-delineated argument (outputs text to console):

```
salt '*' test.echo 'Hello, World!'
```

- With a Keyword=Argument argument (runs the whoami Linux command as user):

```
salt '*' cmd.run "whoami" runas=user
```

- Run multiple functions, including arguments:

```
salt '*' test.ping,test.echo ,"Success!"
```

- Note comma separating functions
- Note comma separating arguments
    - Since test.ping takes no argument, no text is added to the argument, but the comma still denotes that the command should separate nothing (and assign it to test.ping) and the "Success!" message (and assign it to test.echo)

Targeting

- How we denote which module is run on what server
- Minions can be targeted via:
    - Minion ID
    - Grains (-G)
    - Pillar (-I)
    - Regex (-E)
    - Lists (-L)
    - Nodegroups (groups of minions pre-defined in the Master config) (-N)
- Multiple ways of targeting can be used at once ("compound targeting")

Examples

- Minion ID:

```
salt 'web1' user.add melvinf home=/home/melvinf
```

- Grains:

```
salt -G 'os:Ubuntu' user.add johnb home=/home/johnb
```

    - -G - Target grains
    - os: - Define which grain is being used

- Regex:

```
  salt 'minion*' user.add richardl home=/home/richardl
```

- Compound targeting with grains and regex:

```
  salt -C 'G@os:Ubuntu or E@db*' user.add richardl home=/home/
richardl
```

  - -C - Compound target
  - G@ - Define grain targeting
  - E@ - Define regex targeting
  - or - Target minions that match one OR the other targets
  - and - Target minions that match BOTH target properties

Command Modules

- Salt has over 400 modules (as of March 2018)
- These include modules that work with web servers, databases, monitoring tools, containers, Chef and Puppet, Windows, Mac, and cloud/VPS providers
- View a full list of modules at: https://docs.saltstack.com/en/latest/ref/modules/all/index.html
- Salt can also manage its various components using execution modules related to:
  - Salt Beacons
  - Salt Cloud
  - Grains
  - Pillar data
  - Minions
  - Modules

sys Module

- Provides information about modules and their available functions

- View available modules:

```
salt '*' sys.list_modules
```

- View module functions:

```
salt '*' sys.list_functions <module>
```

- View module information:

```
salt '*' sys.doc <module>[.function]
salt '*' sys.doc user.add
```

pkg Module

- Manage packages on the minions

- View available functions:

```
salt '*' sys.list_functions pkg
```

- View documentation:

```
salt '*' sys.doc pkg
```

- Update packages:

```
salt '*' pkg.upgrade
```

- Remove package:

```
salt '*' pkg.remove <package>
```

- Install package:

```
salt '*' pkg.install <package>
```

cmd Modules

- Work with the shell through Salt
- View available functions:

```
salt '*' sys.list_functions cmd
```

- View documentation:

```
salt '*' sys.doc cmd
```

- Locate a command:

```
salt '*' cmd.which <command>
```

- Run custom command:

```
salt '*' cmd.run '<command>'
salt '*' cmd.run 'whoami'
```

- Run custom command as defined user:

```
salt '*' cmd.run '<command>' runas=<user>
salt '*' cmd.run 'whoami' runas=richardl
```

salt-call (Execution Modules)

- salt-call allows you to run execution modules (and Salt states) locally on the minion
- Created to troubleshoot custom execution modules and custom Salt states on a minion
- Use –local flag to designate that we're running the command on the minion:

```
salt-call --local user.add elle home=/home/elle
```

- Used in place on salt command on masterless minions; does not need the –local flag when on a properly-configured masterless minions:

```
salt-call user.add elle home=/home/elle
```

Salt States and Formulas

Introducing Salt States

- Salt states allow you to reuse configuration parameters across multiple servers without having to manually run all the code
- States use **state functions** to configure the server
- A group of states working towards on end goal ("configure PHP") is a formula

Setting the file_roots

- All states are to be saved in a predefined directory; this is the directory defined in our file_roots setting
- The file_roots setting is found in /etc/salt/master
- The directories are set in a per-environment basis
- Default:

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

  - This sets the file root for the base environment at /srv/salt

Salt State Documentation

- Uses the sys execution module, with modified functions
- List available functions for a state (pkg in example):

```
salt '*' sys.list_state_functions pkg
```

- View documentation for a state function (pkg.installed in example):

```
salt '*' sys.state_doc pkg.installed
```

- List arguments (available parameters) for a state function (pkg.insatalled in example):

```
salt '*' sys.state_argspec pkg.installed
```

Anatomy of a Salt State (Review)

```
php_install:
  pkg.installed:
    - name: php
```

- php_install: Reference ID or name declaration
- pkg.installed: State function
- name: An argument or parameter for the state function

The top.sls File

- Allows us to map our states to our minions
- Hierarchy:
  - Set the environment
  - Set the target
    - Targets can be set by minion name, grains, pillar, nodegroups, lists, etc.
  - Set the states
- Example:

```
base:
  'web*':
    - php
    - php.mod-apache
    - php.mod-mysql
```

- Apply states based on top.sls file:

```
salt <target> state.apply
```

or

```
salt <target> state.highstate
```

Managing Files

- Use the file.managed function to add and manage files through Salt
- Files should be saved in the formula directory
- Files managed by Salt should contain a header denoting they are Salt-managed and should not be edited from the server locally
- Use salt:// to reference files in the file root directory:
    - salt:// = /srv/salt, if using the default file_roots
- Example file.managed state:

```
apache_configuration:
  file.managed:
    - name: /etc/apache2/apache2.conf
    - source: salt://apache/config/debian-apache2.conf
```

Salt Requisites

- Requisites allow us to create relationships between states
- Requisites come in two varieties:
    - requisites: Where State A depends on State B (the restart state depends upon the config state to run)
    - requisite_ins: Where State B is depended upon by State A (the config state notes that the restart state should be run after the config state runs)

Requisites

Requisite	Description
require	The targeted state must execute before the dependent state
watch	Dependent state runs if result of targeted state is true
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
onfail	Dependent state runs if targeted state fails
onchanges	Dependent state runs if targeted state makes any changes
use	Dependent state inherits parameters of targeted state (not including requisites)

Templating

Basic Templating

- By default, Salt uses Jinja as its templating agent
- Jinja is evaluated before the YAML when running a state
- Jinja can be used for:
    - Conditionals (if/else)
    - Macros (reusing repeated lines of YAML/text)
    - Filtering (sorting, adding quotations, etc.)
    - Templating files
    - Calling execution modules
        - Two ways of calling execution modules:
            - salt‘module.function’
            - salt.module.function(‘parameters’)

- Jinja is encased in either double curly brackets ({{ }}) for calling variables or a curly bracket/ percent sign combination (\{% %}) for writing the template code (conditionals, writing statements, calling modules, etc.)

- Example: If/else statement

```
apache_install:
  pkg.installed:
    {% if grains['os_family'] == 'Debian' %}
    - name: apache2
    {% elif grains['os_family'] == 'RedHat' %}
    - name: httpd
    {% endif %}
```

  - {% if grains['os_family'] == 'Debian' %}: If (if) the operating system family grain (grains['os_family']) matches Debian (== 'Debian'), run the line below

  - {% elif grains['os_family'] == 'RedHat' %}: If the operating system family grain does DOES NOT match Debian, check to see if (elif or "else if") it matches RedHat (grains['os_family'] == 'RedHat'), then run the line below if it does

  - {% endif %}: Terminates the if/else statement

- Example: Setting variables by calling the grains module:

```
{% set apache = salt['grains.filter_by']({
'Debian': { 'package': 'apache2' },
'RedHat': { 'package': 'httpd' },
}) %}
```

  - {% set apache: Set the name of the group of variables being created

  - salt['grains.filter_by']: Call the grains.filter_by module, which filters minion by operating system family

  - 'Debian':{ 'package': 'apache2' },: For Debian-based minions, set the package variable to apache2; note that the comma after this line is because we providing a list of operating system families and their related variables'RedHat': { 'package': 'httpd' },: Set the package variable to httpd for Red Hat-based systems

  - Note that the two above lines are encased in curly brackets (end of line 1, beginning of line 4) because it is a list

  - To call the package variable, we would use {{ apache.package }}(remember that our set of variables is named \apache)

map.jinja

- Allows us to create a lookup table of values to use in a formula
  - Most often, platform-specific (separated by operating system family)
- Use set to define variable set, with each distribution working as a sub-set
  - See example above
- Example map.jinja file:

```
{% set apache = salt['grains.filter_by']({
   'Debian': {
        'package': 'apache2',
        'service': 'apache2',
        'configfile': '/etc/apache2/apache2.conf',
        'configsource': 'salt://apache/config/debian-apache2.
conf',
    },
    'RedHat': {
        'package': 'httpd',
        'service': 'httpd',
        'configfile': '/etc/httpd/conf/httpd.conf',
        'configsource': 'salt://apache/config/redhat-httpd.conf',
    },
})%}
```

- To use map.jinja from within a Salt state, add:

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

- Calling variables works the same as when defined at the top of the file; e.g., {{ apache.package }}

Templating Files

- Templating files in Salt uses the same process as using Jinja within our Salt states
- Add any variables to the map.jinja file or at the top of the templated file
- If using map.jinja, import the mapping with:

```
{% from "<statename>/map.jinja" import <set> with context %}
```

- Add the template: jinja parameter to your Salt state that manages the file:

```
apache_envvars:
  file.managed:
    - name: /etc/apache2/envvars
    - template: jinja
    - source: salt://apache/config/debian-envvars
    - require:
      - pkg: {{ apache.package }}
```

Best Practices for States

- States should have a clear purpose
  - This includes file names and directory structures
    - These names should related directly to what the state does
  - Remember that state names are used in requisites and directory/file names are used when highstating or adding states to the top.sls file
- Use variables sensibly
  - Use them in places where the value may vary between OS family, etc.
  - Use sane defaults for parameters that are likely to need the same setting across all minions
- States should be modular
  - Do not include all states in a formula in one file
  - Not every minion will need every state
  - Ensure each state includes information about its dependencies and define clear relationships between states

Pillar

- Pillar stores user-provided variables that tend to vary by minion role (not static data like operating system)
  - Virtual hosts on Apache
  - MySQL database information
- Pillar data is stored in /srv/pillar, by default
  - The pillar_roots
  - Contains a top.sls in which pillar data is mapped to minions
  - Pillar data should be saved as .sls files

Example: Basic Pillar

- Add the following dictionary to mysql.sls in your pillar roots

```
mysql:
  server:
    bind: 192.168.50.12
```

- Map mysql to the db1 server in the pillar top file:

```
base:
  'db1':
    - mysql
```

- To call pillar data use {{ pillar[‘key’][‘key’][‘value’] }}

```
bind-address = {{ pillar['mysql']['server']['bind'] }}
```

Encrypting Pillar

- Encrypted data can be saved to pillar and unencrypted via the Salt GPG renderer
- Add the directory /etc/salt/gpgkeys with the mode of 0700
- Generate a master GPG key:

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

  - Ensure your key choices line up with your company policy
  - Do NOT set a password
- Export the key:

```
gpg --homedir /etc/salt/gpgkeys --armor --export saltstack > exported_key.gpg
```

- Import the key to your local keyring:

```
gpg --import exported_key.gpg
```

- Encrypt your value:

```
echo -n "password" | gpg --armor --batch --trust-model always --encrypt -r saltstack
```

- To add the password data to pillar:
  - Add #!yaml|gpg to the top of the pillar file
  - Add copy the encryption block (ensure the | is included):

```
root_pass: |
  -----BEGIN PGP MESSAGE-----
  v1hQA/291jKx54hDAQf/da4mZlPJWzQCrxlVFZJKOQp
  0gBiKNk+eZroGeWZ7NI+tBIsGAGfGQ7eXZBi5yQpTVL
  0gBiKNk+eZroGeWZ7NI+tBIsGAGfGQ7eXZBi5yQpTVL
  0gBiKNk+eZroGeWZ7NI+tBIsGAGfGQ7eXZBi5yQpTVL
  ----END PGP MESSAGE------
```

- Call the pillar data in the state as normal. Additionally, you can see that the message is unencrypted properly by running salt '' pillar.items against the minions using the pillar data.

Using Pillar with Jinja

- We can use for loops in pillar to create multiple changes from one state, such as with users.

- Set the dictionary:

```
mysql:
  users:
    wpuser:
      password: wppass
      host: localhost
      database: wordpress.*
      grants: ALL PRIVILEGES
    richardl:
      password: passpass
      host: localhost
      database: wordpress.*
      grants: ALL PRIVILEGES
```

- In you state file, open the for loop with:

```
  {% for user, arg in salt['pillar.get']('mysql:users', {}).
iteritems() %}
```

  - user and arg are arbitrary values we assign, the first for our dictionary (which will be the user's name), the second for the arguments provided in that dictionary; choose ones that make sense in context.

  - salt'pillar.get'calls our pillar items under \mysql:users in pillar

  - .iteritems iterates through all dictionaries provided under users in pillar

- Call the username with {{ user }}in the state, and the arguments with \{{ arg.keyword }}. For example:

```
  {% for user, arg in salt['pillar.get']('mysql:users', {}).
iteritems() %}

  {{ user }}_mysql_user:
    mysql_user.present:
      - name: {{ user }}
      - password: {{ arg.password }}
      - host: {{ arg.host }}
      - connection_user: root
      - connection_pass: {{ pillar['mysql']['root']['password'] }}
      - connection_charset: utf8

  {{ user }}_user_grants:
    mysql_grants.present:
      - database: {{ arg.database }}
      - user: {{ user }}
      - grant: {{ arg.grants }}
      - host: {{ arg.host }}
      - connection_user: root
      - connection_pass: {{ pillar['mysql']['root']['password'] }}
      - connection_charset: utf8

  {% endfor %}
```