Using Jinja with Salt - Salt user guide

18-23 minutes

Salt renderers¶

The Salt State system operates by gathering information from simple data structures. This has the following advantages:

- Interaction is both generic and simple
- SLS files can be written in many formats
- State files with Jinja templates can be translated to YAML

Salt state rendering¶

By default, SLS files are rendered as Jinja templates and then parsed as YAML documents.

Since the state system only processes raw data, the SLS files can be any structured format that is supported by a Python renderer library.

Renderers can be written to support XML files, HTML files, Puppet files, or any format that can be translated into the data structure used by the state system.

Multiple renderers¶

When deploying a State Tree, the `renderer` option selects a default renderer in the master configuration file. Multiple renderers can be used inside the same State Tree.

Currently there is support for:

- Jinja + YAML (default)
- Mako + YAML
- Jinja + JSON
- Mako + JSON
- Wempy + JSON
- Wempy + YAML
- Python

Here is a sample SLS file in YAML:

```
include:
  - python

python-mako:
  pkg.installed
```

One reason to use another renderer is to take advantage of the Python py renderer.

Here is the same SLS file defined in Python:

```
#!py
def run():
    """
    Install the python-mako package
    """
    return {"include": ["python"], "python-mako": {"pkg": ["installed"]}}
```

Using renderer pipes¶

Render pipes allow the output of render engines to be piped into each other, similar to Unix pipes.

To pipe the output of the Jinja renderer into the YAML renderer, place the following shebang on the first line of the SLS file:

When rendering SLS files, Salt checks for the presence of a Salt-specific shebang line. The shebang line syntax was chosen because it is familiar to the target audience. The shebang line directly calls the name of the renderer as it is specified within Salt, which allows for great flexibility in rendering.

Using the Jinja renderer¶

Although SLS files can be written with YAML, Jinja can be used to template SLS files when more flow control is needed.

By default, Salt uses the Jinja templating language to manage programmatic control over the YAML files. Application configuration files can also contain Jinja and are processed when deployed with a state function such as `file.managed`.

Jinja basics¶

The default Jinja delimiters are defined as:

Jinja delimiters	Definition
{% … %}	Define a Jinja statement
{%- … -%}	Define a Jinja statement, but remove whitespace from beginning and end of line
{{ … }}	Print a Jinja expression or call a Salt execution directly at the desired location of the file
{{- … -}}	Jinja expression removing whitespace from beginning and end of line
{# … #}	Jinja comments - not included in output after being rendered
{#- … -#}	Jinja comments removing whitespace from beginning and end of line

Jinja comment tags can span multiple lines. This is a good way to comment blocks of states within a SLS file for testing.

Whitespace removal can be defined for beginning of the line, the end of the line, or both. See Jinja documentation for more details.

All Salt renderers, including the default Jinja + YAML renderer, contain variables that can be used to hold data. Gaining access to this data is one of the main advantages to using Jinja.

Injecting data into Salt state files¶

The state system injects dictionaries for easy accessibility to Salt data. These dictionaries are available through Jinja.

The most commonly used dictionaries are:

- `grains`: all grains for the minion
- `pillar`: all pillar data available to the minion
- `salt`: all available execution modules and functions

Accessing grains with Jinja¶

Salt grains can be accessed using Jinja. Salt grains are exposed to the state system through a `grains` dictionary.

A grain in the `grains` dictionary can be referenced in the following format:

```
{{ grains['name-of-grain'] }}
```

- For example, the `os_family` grain can be referenced using Python syntax:

```
{{ grains['os_family'] }}
```

- Jinja provides conditional `if` statements that enhance states with additional logic.
- Grains are commonly used in conditional statements.

Dictionary access¶

A dictionary can be presented in multiple types of syntax. The traditional Python syntax would look like:

```
# Python notation for dictionary access
push_conf:
  file.managed:
    - name: /etc/named.conf
    # Push either RedHat-named.conf or Debian-named.conf file
    - source: salt://dns/files/{{ grains['os_family'] }}-named.conf
```

This example uses the `os_family` grain to determine the proper file name. Jinja allows for a dotted notation for accessing dictionaries:

```
# Jinja dotted notation for dictionary access
push_conf:
  file.managed:
    - name: /etc/named.conf
    # Push either RedHat-named.conf or Debian-named.conf file
    - source: salt://dns/files/{{ grains.os_family }}-named.conf
```

Note

The type of syntax used is a styling preference, but there may be times when a Python dictionary syntax is needed.

Return data access¶

Using a Salt execution `module.function` for data injection:

```
update_hosts:
  file.append:
    - name: /etc/hosts
    - text: |
        {{ salt['network.interface_ip']('eth0') }} {{ grains['fqdn']}}
```

Jinja statements¶

Jinja statements can be used throughout Salt, in state files as well as configuration files, and include:

- Variable assignment
- Conditional statements
- Iteration

Jinja variable assignment¶

Variables can be set and referenced in Jinja. Jinja variables are declared using the `set` keyword in the following syntax:

```
{% set zone_path = '/etc/named/zones' %}
```

A variable can then be referenced:

```
push_config:
  file.managed:
    - source: salt://dns/files/zones/db.foo.com
    - name: {{ zone_path }}/db.foo.com
```

Jinja variables can also be used to hold return data from a Salt executions:

```
{% set connect_info = salt['network.connect']('www.google.com','80') %}

google_connect:
  test.configurable_test_state:
    - name: "Connect comment: {{ connect_info['comment'] }}"
    - changes: False
    - result: {{ connect_info['result'] }}
```

Jinja variable types¶

Variable assignments can be of many types:

- `"Hello World"`: Everything between two double or single quotes is a string.
- `42` / `42.23`: Integers and floating point numbers. If a decimal point is present, the number is a float.
- `['list', 'of ', 'objects']`: Everything between two brackets is a list.
- `('tuple', 'of ', 'values')`: Tuples are like lists that cannot be modified ("immutable"). If a tuple only has one item, it must be followed by a comma (('1-tuple',)).
- `{'dict': 'of ', 'key': 'and', 'value': 'pairs'}`: A dictionary in Python is a structure that combines keys and values. Keys must be unique and always have exactly one value.
- `True` / `False`: True is always true and False is always false.

Jinja conditional if statements¶

An `if` conditional statement structure in Jinja is followed by a test expression. The following example declares a configuration directory in a variable named `dns_cfg` to be used based on distribution:

/srv/salt/dns/dns_conf.sls¶

```
{% if grains.os_family == 'RedHat' %}
  {% set dns_cfg = '/etc/named.conf' %}
{% elif grains.os_family == 'Debian' %}
  {% set dns_cfg = '/etc/bind/named.conf' %}
{% else %}
  {% set dns_cfg = '/etc/named.conf' %}
{% endif %}
dns_conf:
  file.managed:
    - name: {{ dns_cfg}}
    - source: salt://dns/files/named.conf
```

Note

Spacing of Jinja statements is only for readability. Since Jinja is rendered before YAML, all Jinja formatting is removed when evaluated by the minion.

When rendered, the value is inserted into the proper location:

```
ns01:
    ----------
    dns_conf:
        ----------
        ...
        file:
            |_
                ----------
                name:
                        /etc/named.conf # <-- Rendered
     on RedHat
            |_
                ----------
                source:
                        salt://dns/files/named.conf
            - managed
            ...
```

Using iteration to leverage lists¶

If three users are present on a system as defined in a state, the YAML file looks like:

/srv/salt/users.sls¶

```
create_fred:
  user.present:
    - name: fred

create_bob:
  user.present:
   - name: bob

create_frank:
  user.present:
    - name: frank
```

A list of users can be assigned to a Jinja variable using a `set` statement, which then references each user by using a Jinja `for` loop. The Jinja list is in Python list syntax:

```
# Declare Jinja list
{% set users = ['fred', 'bob', 'frank']%}

# Jinja `for` loop
{% for user in users%}
create_{{ user }}:
  user.present:
    - name: {{ user }}
{% endfor %}
```

Using iteration to leverage dictionaries¶

A Jinja dictionary is defined in the same syntax as Python:

```
{% set users = {
   'leonard': {'uid': 9001, 'shell': '/bin/zsh', 'fullname': 'Leonard
Hofstadter'},
   'sheldon': {'uid': 9002, 'shell': '/bin/sh', 'fullname': 'Sheldon Cooper'},
   'howard': {'uid': 9003, 'shell': '/bin/csh', 'fullname': 'Howard Wolowitz'},
   'raj': {'uid': 9004, 'shell': '/bin/bash', 'fullname': 'Raj Koothrappali'}}
%}

{% for user in users %}
create_user_{{ user }}:
  user.present:
    - name: {{ user}}
    - uid: {{ users[user]['uid']}}
    - shell: {{ users[user]['shell']}}
    - fullname: {{ users[user]['fullname']}}
{% endfor %}
```

More complex iteration¶

Iterations can be used with more complex dictionaries to directly extract `key/value` pairs:

```
{% set servers = {
  'proxy': {
    'host': '203.0.113.18',
    'chassis': {
      'name': 'fx2-1',
      'management_mode': '2'
      'datacenter': 'atl',
      'rack': '1',
      'shelf': '3',
      'servers': {
   'server1': {'idrac_password': 'somethingsecret', 'ipmi_over_lan': True},
   'server2': {'idrac_password': 'supersecret', 'ipmi_over_lan': True},
   'server3': {'idrac_password': 'kindofsecret','ipmi_over_lan': True}}}}%}

{% set details = servers['proxy']['chassis'] %}

standup_step1:
  dellchassis.chassis:
    - name: {{ details['name'] }}
    - location: {{ details['location'] }}
    - mode: {{ details['management_mode'] }}

# Set idrac_passwords for 'servers'.
{% for k, v in details['servers'].iteritems() %}
{{ k }}:
  dellchassis.blade_idrac:
    - idrac_password: {{ v['idrac_password'] }}
{% endfor %}
```

This is a complex example, but it can be simplified by using data from other sources.

Importing data¶

Jinja allows for importing external files and Salt executions. This is useful any time the same data must be made available to more than one SLS file.

- It is quite common for Jinja code to be modularized into separate files.
- Jinja variables can be imported into Salt state files.
- It is recommended to put platform-specific settings in a separate file.

Map files have several benefits:

- Single location for value reuse
- Allows for overrides and sane defaults
- Can be used for platform-specific details
- Can be defined with environment-specific values (dev/prod)

Salt execution `module.functions` allow data to be retrieved from a remote source and injected into the workflow.

YAML map files¶

A YAML map file can be created and managed separately from the state file that consumes it. This allows the data to be managed independently from the function:

/srv/salt/dns/map.yaml¶

```
Debian:
  pkg: bind9
  srv: bind9
RedHat:
  pkg: bind
  srv: named
```

We can now adjust the dns state file to consume the data inside the YAML map file and express the values which are appropriate for the minion’s needs:

/srv/salt/dns/init.sls¶

```
# Import YAML map file
{% import_yaml 'dns/map.yaml' as osmap %}

# Filter the structured data (dictionary) using the 'os_family' grain
{% set dns = salt['grains.filter_by'](osmap) %}

install_dns:
  pkg.installed:
    - name: {{ dns.pkg }}

start_dns:
  service.running:
    - name: {{ dns.srv }}
    - enable: True
```

JSON map files¶

If we convert the previous example from YAML to JSON, an external resource can manage the consumed data inside the map file:

```
{
  'Debian':
    {'pkg': 'bind9', 'srv': 'bind9'},
  'RedHat':
    {'pkg': 'bind', 'srv': 'named'}
}
```

The dns state file can be altered to consume JSON by the `import` line:

/srv/salt/dns/init.sls¶

```
# Import JSON map file
{% import_json 'dns/map.json' as osmap %}

# Filter the structured data (dictionary) using the 'os_family' grain
{% set dns = salt['grains.filter_by'](osmap) %}

install_dns:
  pkg.installed:
    - name: {{ dns.pkg }}

start_dns:
  service.running:
    - name: {{ dns.srv }}
    - enable: True
```

Notice that none of the other logic or syntax needs to be altered.

Jinja map files¶

Another example of using map files is to define the data directly as a dictionary. The main advantage over the other methods is speed of consumption by the minion:

```
{% set osmap = {
  'Debian':
    {'pkg': 'bind9', 'srv': 'bind9'},
    'RedHat':
    {'pkg': 'bind', 'srv': 'named'}
} %}
```

The dns state file is altered as before, except with slightly different syntax:

/srv/salt/dns/init.sls¶

```
# Import Jinja map file - notice "with context"
{% from 'dns/map.json' import as osmap with context %}

# Filter the structured data (dictionary) using the 'os_family' grain
{% set dns = salt['grains.filter_by'](osmap) %}

install_dns:
  pkg.installed:
    - name: {{ dns.pkg }}

start_dns:
  service.running:
    - name: {{ dns.srv }}
    - enable: True
```

Remote execution data¶

Data needed for any workflow may exist external to the Salt infrastructure. Consider the example where data needed for configuration exists via a REST call or a DB query. If the minion can access the remote resource which contains the needed data, it can be used to inject data to any workflow.

Pillar data is another example of an external data store. See the Pillar documentation for more information.

This example makes an `http.query` to a web service to retrieve some structured data and inject it into the workflow:

```
# App server returns data as a list of user data:
# [{'username':'value','uid':'value','shell':'value'}]
{% set user_data = salt['http.query']
('https://example.com/userservice/users','method=GET') %}

{% for user in user_data %}
create_{{ user['username'] }}:
  user.present:
    - name: user['username']
    - uid: user['uid']
    - shell: user['shell']
{% endfor %}
```

Templating application configuration files¶

Files can have Jinja declared to plugin values as they are pushed to minions. Adding `template: jinja` to a `file.managed` state instructs Salt to use Jinja to render the file before it is written to the filesystem.

Consider the following example of map file `/srv/salt/redis/map.json` containing Redis configuration data:

```
{
  'Debian': {
    'pkgs': ['redis-server','python-redis'],
    'service’: 'redis-server',
    'conf': '/etc/redis/redis.conf',
    'bind': '0.0.0.0',
    'port': '6379',
    'user': 'redis',
    'root_dir': '/var/lib/redis'
  },
  'RedHat': {
    'pkgs': ['redis','python-redis'],
    'service’: 'redis',
    'conf': '/etc/redis.conf',
    'bind': '0.0.0.0',
    'port': '6379',
    'user': 'redis',
    'root_dir': '/var/lib/redis'
  }
}
```

Now let’s look at a snippet of the Redis configuration file:

/srv/salt/redis/files/redis.conf¶

```
 daemonize no
 pidfile /var/run/redis/redis.pid

 port {{redis_port}}
 bind {{redis_bind}}
 dir {{redis_dir}}

 tcp-backlog 511
 ...
```

Now, let’s put it all together with a Salt state file:

/srv/salt/redis/init.sls¶

```
{% import_json 'redis/map.json' as osmap %}
{% set redis = salt['grains.filter_by'](osmap) %}
redis_install:
  pkg.latest:
    - pkgs:
    {% for pkg in redis.pkgs %}
      - {{ pkg }}
    {% endfor %}

redis_service:
  service.running:
    - enable: True
    - name: {{ redis.service }}
    - require:
    - pkg: redis_install

redis_conf:
  file.managed:
    - source: salt://redis/files/redis.conf.jinja
    - name: {{ redis.conf }}
    - user: {{ redis.user }}
    - group: root
    - mode: '0644'
    - template: jinja                   # <- Use Jinja to render file
    - redis_bind: {{ redis.bind }}      # <- Pass redis_bind from map value
    - redis_port: {{ redis.port }}      # <- Pass redis_port from map value
    - redis_dir: {{ redis.root_dir }}   # <- Pass redis_dir from map value
    - require:
      - pkg: redis_install
    - watch_in:
      - service: redis_service
```

This example shows us how we can manage the deployment and configuration of an application using external data.

Outputters and parsing return data¶

The output in Salt commands can be configured to present the data in other formats using Salt outputters.

Outputter options¶

The `return data` from Salt minion executions can be formatted by using `--output` as a command line argument. The default format uses the `nested` format. Common formats used are `json`, `pprint` (Python’s pretty print), and `txt` formats. Output Options:

```
--out=OUTPUT, --output=OUTPUT
                    Print the output from the 'salt' command using the specified
                    outputter. The builtins are 'raw', 'compact', 'no_return',
                     'grains', 'overstatestage', 'pprint', 'json', 'nested',
                    'yaml', 'highstate', 'quiet', 'key', 'txt',
                    'newline_values_only', 'virt_query'.

--out-indent=OUTPUT_INDENT, --output-indent=OUTPUT_INDENT
                    Print the output indented by the provided value in spaces.
```

```
                   Negative values disables indentation. Only applicable in
                   outputters that support indentation.

--out-file=OUTPUT_FILE, --output-file=OUTPUT_FILE
                   Write the output to the specified file

--no-color, --no-colour
                   Disable all colored output

--force-color, --force-colour
                   Force colored output
```

The default nested format:

```
$ salt \*redhat status.loadavg --out=nested
```

```
20190218-sosf-lab0-redhat:
  ----------
  1-min:
        0.08
  15-min:
        0.05
  5-min:
        0.05
```

The JSON format:

```
$ salt \*redhat status.loadavg --out=json
```

```
{
    "20190218-sosf-lab0-redhat": {
         "15-min": 0.05,
        "5-min": 0.04,
        "1-min": 0.05
    }
}
```

Parsing return data external to Salt¶

External commands to Salt can parse return data to allow access to subsets of the return data.

The following examples show how to parse JSON formatted output using `jq`:

```
$ salt-call network.interfaces --out=json | jq .
```

```
{
  "local": {
    "lo": {
      "hwaddr": "00:00:00:00:00:00",
      "up": true,
      "inet": [
        {
          "broadcast": null,
          "netmask": "255.0.0.0",
          "address": "127.0.0.1",
          "label": "lo"
        }
      ],
      "inet6": [
        {
          "prefixlen": "128",
          "scope": "host",
          "address": "::1"
        }
      ]
    },
    "eth0": {
      "hwaddr": "00:16:3e:35:b0:85",
      "parent": "if11",
      "up": true,
      "inet": [
        {
          "broadcast": "192.0.2.255",
          "netmask": "255.255.255.0",
          "address": "192.0.2.23",
          "label": "eth0"
        }
      ],
      "inet6": [
        {
          "prefixlen": "64",
          "scope": "global",
          "address": "2001:db8:1ebe:4370:216:3eff:fe35:b085"
        },
        {
          "prefixlen": "64",
          "scope": "link",
          "address": "fe80::216:3eff:fe35:b085"
        }
      ]
    }
  }
}
```

If you only want the IP address of each minion, you can use `jq` to filter the JSON results:

```
$ salt \* network.interfaces --out=json | jq '.[].eth0.inet[].address'
```

```
"192.0.2.23"
"192.0.2.56"
"192.0.2.71"
"192.0.2.125"
"192.0.2.200"
```

This example shows how we can use alternate methods to extract data from a minion for use during a workflow.