Configuration management with Salt

Ogden Area Linux User Group   Seth House <seth@eseth.com>

2013-06-25

1 Salt states: an introduction
- It's all about the data
- Execution happens on the minions
- Anatomy of the highstate data structure
- In summary

2 Demo: basic states

3 Salt states: diving deeper

4 Salt states: fetching data from the master

5 Salt states: fetching data from other minions

6 Salt states: orchestration

It's all about the data

```
httpd:
  pkg:
    - installed
```

It’s all about the data

Only the data structure matters

It’s all about the data

Only the data structure matters

- YAML
- Jinja
- Mako
- JSON
- Wempy
- Python
- PyDSL

It's all about the data

Only the data structure matters

- YAML
- Jinja
- Mako
- JSON
- Wempy
- Python
- PyDSL
- ???

It's all about the data

YAML

```
httpd:
  pkg:
    - installed
```

Generated data structure

```
{'httpd': {'pkg': ['installed']}}
```

It's all about the data

Declarative / imperative. Full language / templating language. **Your choice.**

```
#!pydsl

apache = state('apache')
apache.pkg.installed()
apache.service.running()
```

Execution happens on the minions

```
hostname:
  cmd:
    - run
```

Execution happens on the minions

Each minion can take local data and local executions into account.

```
apache:
  pkg:
    - installed
    {% if grains['os'] == 'RedHat' %}
    - name: httpd
    {% elif grains['os'] == 'Ubuntu' %}
    - name: apache2
    {% endif %}
```

Execution happens on the minions

Each minion can take local data and local executions into account.

```
{% if salt['file.file_exists']
    ('/tmp/specialfile') %}
dosomething:
  cmd:
    - run
{% endif %}
```

Anatomy of the highstate data structure

A unique identifier (the key in a dictionary).

```
httpd:          # ID declaration
  pkg:
    - installed
```

Anatomy of the highstate data structure

The `pkg` state module.

```
httpd:
  pkg:            # state declaration
    - installed
```

Anatomy of the highstate data structure

The `installed` function in the `pkg` state module.

```
httpd:
  pkg:
    - installed # function declaration
```

Anatomy of the highstate data structure

This data structure maps to the `pkg.installed` function signature.

```
salt.states.pkg.installed(
    name,
    version=None,
    refresh=False,
    fromrepo=None,
    skip_verify=False,
    pkgs=None,
    sources=None,
    **kwargs)
```

Anatomy of the highstate data structure

```
httpd:
  pkg:
    - installed
    - version: 2.2.23 # function arg declaration
```

Anatomy of the highstate data structure

```
httpd:
  pkg:
    - installed
    - version: 2.2.23
    - arbitrary: "key/vals are passed as
                  keyword arguments"
```

Anatomy of the highstate data structure

The first argument to the function is implictly taken from the ID declaration unless specified.

```
myapacheinstall:
  pkg:
    - installed
    - name: httpd
```

Anatomy of the highstate data structure

Multiple state declarations can live under one ID declaration.

```
httpd:
  pkg:
    - installed
  service:
    - running
```

In summary

- Each minion builds it’s own data structure.
- The data structure can be built by any programming language or templating engine.
- All logic happens in that build process.
- The Salt minion runs *deterministic* executions based on that data structure.

1 Salt states: an introduction

2 Demo: basic states
- Installing a package
- Running states
- State config options
- Tying `sls` files together with a top file
- Transfering a file from the master
- Execution happens on the minion
- Templating a YAML file with Jinja
- Creating Jinja macros

3 Salt states: diving deeper

4 Salt states: fetching data from the master

Installing a package

```
httpd:
  pkg:
    - installed
  service:
    - running
```

Running states

- `state.sls`
- `state.highstate`

Running states

- `state.sls`
- `state.highstate`
- `startup_states`

Running states

- `state.sls`
- `state.highstate`
- `startup_states`
- `state.show_highstate`
- `state.show_lowstate`

State config options

- `state_verbose`
- `state_output`
- `failhard`

Tying sls files together with a top file

top.sls

```
'*':
  - base
'app*':
  - web_app
'web*':
  - web_server
'virtual:virtual':
  - match: grain
  - rackspace_stuff
```

Transfering a file from the master

```
/srv/http/index.html:
  file:
    - managed
    - source: salt://index.html
    - user: root
    - group: root
    - mode: 644
```

Execution happens on the minion

```
{% if salt["cmd.run"]
    ("free -m | awk '!/^[A-Z ]/ { print $4 }'")
    > 2000 %}
mem_intensive_op:
  cmd:
    - run
{% endif %}
```

Templating a YAML file with Jinja

```
{% for user in ['fred', 'tom', 'george'] %}
{{ user }}:
  user:
    - present
{% endfor %}
```

Creating Jinja macros

```
{% macro make_user(name) %}
{{ name }}
  user:
    - present
{% endmacro %}

{{ make_user('fred') }}
{{ make_user('tom') }}
{{ make_user('george') }}
```

1 Salt states: an introduction

2 Demo: basic states

3 Salt states: diving deeper
- Special constructs in the highstate data structure
- `names`
- Delay execution until all requirements are met
- React to a change in the dependency tree
- Optionally execute a state based on a test run
- Reuse default args in multiple states
- Spread a state tree across multiple files
- Modify a state in another file
- Cease all execution on failure
- State ordering

Special constructs in the highstate data structure

Salt can alter the data structure at compilation-time if certain constructs are present as well as alter the exection flow.

Special constructs in the highstate data structure

Salt can alter the data structure at compilation-time if certain constructs are present as well as alter the exection flow.
Top level:

- `include`
- `extend`

Special constructs in the highstate data structure

Salt can alter the data structure at compilation-time if certain constructs are present as well as alter the exection flow.

Top level:

- `include`
- `extend`

Declaration level:

- `names`
- `require` / `require_in`
- `watch` / `watch_in`
- `prereq` / `prereq_in`
- `use` / `use_in`
- `failhard`
- `order`

names

`names` will cause the entire dictionary to be duplicated for each item in the list.

```
phpstuff:
  pkg:
    - installed
    - names:
      - php
      - php-mysql
      - drupal7
```

names

```
php:
  pkg:
    - installed

php-mysql:
  pkg:
    - installed

drupal7:
  pkg:
    - installed
```

names

(Actually there's a better option for the `pkg.installed` function.)

```
phpstuff:
  pkg:
    - installed
    - pkgs:
      - php
      - php-mysql
      - php-mbstring
      - php-gd
      - php-xml
      - drupal7
```

Delay execution until all requirements are met

```
httpd:
  pkg:
    - installed

/etc/httpd/httpd.conf:
  file:
    - managed
    - require:
      - pkg: httpd
```

Delay execution until all requirements are met

```
httpd:
  pkg:
    - installed
    - require_in:
      - file: /etc/httpd/httpd.conf

/etc/httpd/httpd.conf:
  file:
    - managed
```

React to a change in the dependency tree

```
httpd:
  pkg:
    - installed
  service:
    - running
    - watch:
      - file: /etc/httpd/httpd.conf

/etc/httpd/httpd.conf:
  file:
    - managed
    - require:
      - pkg: httpd
```

Optionally execute a state based on a test run

```
apachectl graceful:
  cmd:
    - run
    - prereq:
      - git: myapp

myapp:
  git:
    - latest
    - name: git://internal/myapp.git
    - target: /srv/http/mysite
```

Reuse default args in multiple states

```
fred:
  user:
    - present
    - fullname: Fred Jones
    - home: /home/fred
    - shell: /bin/zsh
    - groups:
      - wheel

tom:
  user:
    - present
    - fullname: Tom Smith
    - home: /home/tom
    - use:
      - user: fred
```

Spread a state tree across multiple files

services.sls

```
httpd:
  pkg:
    - installed
  service:
    - running
```

app.sls

```
include:
  - services

php:
  pkg:
    - installed
    - require:
      - pkg: httpd
```

Modify a state in another file

app.sls

```
include:
  - services

extend:
  httpd:
    service:
      - watch:
        - git: myapp

myapp:
  git:
    - latest
    - name: git://internal/myapp.git
    - target: /srv/http/mysite
```

Cease all execution on failure

```
myapp:
  git:
    - latest
    - name: git://internal/myapp.git
    - target: /srv/http/mysite
    - failhard: True
```

State ordering

```
kernel:
  pkg:
    - latest

reboot:
  cmd:
    - run
    - order: last
```

1 Salt states: an introduction

2 Demo: basic states

3 Salt states: diving deeper

4 Salt states: fetching data from the master
- Pillar
- Private data
- Parameterization
- Config values
- Targeting

5 Salt states: fetching data from other minions

6 Salt states: orchestration

Pillar

- Fetch data from the master
  - Flat files on the filesystem
  - Commands that return JSON/YAML
  - Cobbler
  - Hiera
  - libvirt
  - Mongo
  - ldap
  - Puppet

Pillar

- Fetch data from the master
  - Flat files on the filesystem
  - Commands that return JSON/YAML
  - Cobbler
  - Hiera
  - libvirt
  - Mongo
  - ldap
  - Puppet
- Private data
- Parameterization
- Config values
- Targeting

Private data

A pillar top file dictates which minions see what data.

/srv/pillar/top.sls

```
'*':
  - global_stuff
'minion1':
  - private_minion1_stuff
'os:RedHat':
  - match: grain
  - redhat_stuff
```

Parameterization

/srv/pillar/pkg_rosetta.sls

```
pkgs:
  {% if grains['os_family'] == 'RedHat' %}
  apache: httpd
  vim: vim-enhanced
  {% elif grains['os_family'] == 'Debian' %}
  apache: apache2
  vim: vim
  {% endif %}
```

/srv/salt/somesls.sls

```
{{ salt['pillar.get']('pkgs.apache') }}:
  pkg:
    - installed
```

Config values

Minion config **or** in a minion’s pillar:

```
schedule:
  highstate:
    function: state.highstate
    minutes: 60
```

Targeting

```
salt -I 'somekey:specialvalue' test.ping
```

1 Salt states: an introduction

2 Demo: basic states

3 Salt states: diving deeper

4 Salt states: fetching data from the master

5 Salt states: fetching data from other minions
- Live data: peer interface
- Recent data: Salt mine

6 Salt states: orchestration

Live data: peer interface

- Realtime data
- Communication goes through the master
- Whitelist

Live data: peer interface

```
{% for server,ip in
    salt['publish.publish'](
        'web*',
        'network.interfaces',
        ['eth0']).items() %}
server {{ server }} {{ ip[0] }}:80 check
{% endfor %}
```

Recent data: Salt mine

- Recent data (configurable)
- Cached on the master (faster lookup)

```
/etc/salt/{master,minion}:
   mine_functions:
       network.interfaces: [eth0]

   mine_interval: 60
```

Recent data: Salt mine

```
{% for server,ip in
    salt['mine.get'](
        'web-*',
        'network.interfaces',
        ['eth0']).items() %}
server {{ server }} {{ ip[0] }}:80 check
{% endfor %}
```

1 Salt states: an introduction

2 Demo: basic states

3 Salt states: diving deeper

4 Salt states: fetching data from the master

5 Salt states: fetching data from other minions

6 Salt states: orchestration
- Batch execution
- Overstate
- See also

Batch execution

- Execute a command incrementally across minions
  - By `N` at a time
  - By a percentage of all minions

```
salt -G 'os:RedHat' \
    --batch-size 25% service.restart httpd
```

Overstate

Incrementually execute a series of state trees that depend on each other.

```
mysql:
  match: db*
  sls:
    - mysql.server

webservers:
  match: web*
  require:
    - mysql

all:
  match: '*'
  require:
    - mysql
    - webservers
```

See also

- Reacting to live events with Salt's reactor
- Schedule system monitoring with Salt's scheduler