ATS SALTSTACK INTRODUCTORY TRAINING

2014-03-06

Presented by Ken Smith / @Ken_2scientists

INTRODUCTION

SECTION OBJECTIVES

After this section, you should be able to:

- Explain the infrastructure automation problem Salt is designed to address
- Describe the salt master/minion architecture
- Manage minion registration including:
  - Registering a new minion with a master
  - Identify all registered minions
  - Pre-seed a minion with a key
- Execute remote commands on all or some minions
- Gather system information from a minion
- Explain which commands/processes execute on a master versus on the minion

OVERVIEW

Salt is an infrastructure automation tool designed to *scalably* manage (and deploy) systems.

QUICK FACTS

- Salt is written in Python
- Salt is Apache-2.0 licensed
- Salt has been in active development since March 2011
- SaltStack is the commercial company that backs the development of Salt, offering consulting, training, and certification services

INFRASTRUCTURE AUTOMATION

- System administration is typically performed by logging into a server, and using command-line tools or GUIs to alter system configuration
  - This is rarely auditable
  - This is rarely reproducible
  - Systems become works of art
  - The time to recreate a system in case of disaster grows with the complexity
- Infrastructure automation and configuration management tools allow us to use tools like `diff` and source-control to *see* exactly what is being done
- This applies to hardware like switches as well as servers

WHY SALT?

There are a number of other open-source configuration management tools out there designed to manage systems. Why are we supporting the adoption of Salt when it's one of the newer players in the field?

THE COMPETITION

- Ohloh comparison of Salt, Puppet, Chef
- InfoWorld article reviewing Puppet, Chef, Ansible, Salt
- Taste Test: Puppet Chef Salt Ansible

ARCHITECTURE

- Salt is designed as a *centralized* system management tool, using persistent processes running on both the manager and the managed systems.
- The *master* maintains the knowledge of all managed systems. The master is your central control point and must be protected. It can bring up and take down your infrastructure.
- The *minions* are the managed systems. Every minion is identified by a unique key (minion id) that allows you to *target* them. A common convention is to use the key to denote a function such as `projname-database-1`, `projname-appserver-1`, `projname-loadbalancer`

COMMUNICATION

- Salt uses 0mq (a distributed communication library). This allows it to manage large numbers of minions without blocking (synchronous) communication.
- Salt uses AES to encrypt all communication between the master and minions. This requires prior exchange and acceptance of shared keys.

SETTING UP THE SANDBOX

1. Install VirtualBox
2. Install Vagrant
3. Install Git client (e.g. GitHub for Windows, GitHub for Mac)
4. Clone salt-sandbox repo:

```
git clone https://github.com/Ken-2scientists/salt-sandbox
```

5. Download the basebox:

```
vagrant box add precise32-stackstrap \
 https://dl.dropboxusercontent.com/s/ftlm33esku2w8aa/ubuntu1204-i386.b
```

6. Launch the VMs:

```
vagrant up master
vagrant up minion
```

KEY MANAGEMENT W/ `SALT-KEY`

- The minions by default will attempt to check in to a host identified by DNS as '`salt`'. Our Vagrantfile has pre-seeded the private IPs of the master and minion. To override the default, edit `/etc/salt/minion`
- On the master, to see all keys:

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

- To accept all pending keys ('`-a`' to accept single key):

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

- To delete all keys ('`-d`' to reject single key):

```
sudo salt-key -D
```

  Deleting a key is necessary if you want to regenerate a minion with a fixed ID.

PRE-SEEDING KEYS

1. Generate a new keypair:

```
salt-key --gen-keys=[key_name]
```

2. Add the public key to the master's accepted keys:

```
cp key_name.pub /etc/salt/pki/master/minions/[minion_id]
```

3. Get the keypair to your minion securely
4. Place keypair in appropriate location before starting salt-minion daemon:

```
/etc/salt/pki/minion/minion.pem
/etc/salt/pki/minion/minion.pub
```

FIRST EXECUTION

```
sudo salt \* cmd.run date
```

- The first argument is the *target*. Here, we're targeting all minions. Globs are supported, which is where the naming convention becomes useful.
- The second argument is the call function. As we'll see later, these will typically be *states*
- Any remaining arguments are passed as arguments to the call function.

SALT'S PUN-FILLED TERMINOLOGY

minion: a managed server

grains: minion-specific metadata that can be used for conditional configuration

pillar: global data that can be selectively distributed to minions, especially used for sensitive data like connection URLs, passwords, etc

RETRIEVING MINION-SPECIFIC INFORMATION

- Retrieve all possible grain information (keys):

```
salt \* grains.ls
```

- Retrieve all grain information:

```
salt \* grains.items
```

- Retrieve the value for a specific grain:

```
salt \* grains.get os_family
```

WHERE DIFFERENT COMMANDS RUN

- grain data comes from each minion
- pillar data comes from the master, but specific variables/values are provided to each minion, as configured
- The `salt` command gathers grain information from minions, gathers pillar data, and orchestrates execution on minions
- `salt-call` can be run locally on a minion with identical syntax to `salt` except without the target argument
- minions keep a cache of pillar data

SALT STATES

SECTION OBJECTIVES

After this section, you should be able to:

- Organize state (sls) files in a consistent directory structure per conventions
- Create a self-contained state (sls) file to accomplish a goal
- Use the dry-run `test=True` argument to test states
- Use the pkg state to install software on minions
- Use the file state to manage files on minions
- Use the service state to manage services
- Use the '`require`' attribute to ensure ordering of state application
- Use jinja templating to execute conditional logic in a state file
- Use the SaltStack documentation to look up options for a state
- Debug the salt state data structures
- Use a salt-formula to use existing "cookbooks"

WHY STATES?

- When you write a shell or batch script, you're *imperatively* declaring commands to execute, but this has limitations
  - What happens if you run the same command twice?
  - How do you handle commands that need to run in a specific order?
  - How do you repurpose commands?
- Salt encourages (but does not mandate) the use of declarative *states*
  - States describe how a minion *should be*
  - States are backed by execution modules that *make it so*
  - States, ideally, are idempotent
  - States can declare interdependencies

CONFIGURATION AS *DATA*

- In salt, the configuration of a system is treated as a set of data structures
- Salt is written in Python, and Python makes heavy use of the `dict` structure. Python `dicts` and JSON are almost equivalent
- Neat salt trick:

```
sudo salt \* network.interfaces --out=json
sudo salt \* network.interfaces --out=txt
sudo salt \* network.interfaces --out=yaml
sudo salt \* network.interfaces --out=pprint
```

DIRECTORY ORGANIZATION

- The primary file in `/srv/salt` is the `top.sls`. This file determines which states to apply to which minions

```
# /srv/salt/top.sls
base:
  '*':
    - webserver
```

- A top file can specify different configurations for different environments. The default environment is always `base`.
- The `webserver` module can define specific configuration settings in a file `webserver.sls` or `webserver/init.sls`. The latter is preferred as soon as the configuration gets complex.

TARGETING IN `TOP.SLS`

```
# /srv/salt/top.sls
base:
  'projname-web*':
    - webserver
  'projname-db*':
    - database
  'projname-lb*':
    - loadbalancer
```

ANATOMY OF AN SLS FILE

- Here's an example SLS file:

```
# /srv/salt/webserver/init.sls
apache:
  pkg.installed
```

- This is equivalent, making the name argument explicit:

```
# /srv/salt/webserver/init.sls
mywebsite:            # id
  pkg:                # state module
    - installed       # function
    - name: apache    # name argument
```

- Let's look at the `pkg` state module's documentation.

THE HIGH STATE

- Let's look at what's called the `highstate`:

```
sudo salt minion state.show_highstate
```

- The high-state is a data structure compiled for the minion to tell it what it needs to do.
- Let's apply it (but be cautious):

```
sudo salt minion state.highstate test=True
```

- Let's try pulling from the minion instead of pushing from the master:

```
sudo salt-call -l debug state.highstate
```

MANAGING FILES

- The salt-master provides a shared filesystem that can be accessed by minions
- This filesystem can be used to share configuration files or other content with minions
- For example, let's assume we have a custom homepage for our webserver:

```
/var/www/index.html:
  file.managed:
    - source: salt://webserver/index.html
    - require:
      - pkg: apache
```

FILESYSTEMS

- The most common use case is that you'll serve content out of the `salt://` filesystem exposed by the master
- You can also directly retrieve content from an HTTP or FTP site, but only if you ensure that the file signature (hash) matches:

```
playzip:
  file.managed:
    - source: http://downloads.typesafe.com/play/2.2.1/play-2.2.1.zip
    - source_hash: md5=08fad782828df77be24ec42afa1980bb
    - user: vagrant
    - group: vagrant
    - mode: 644
```

OTHER USEFUL FILE FUNCTIONS

- managed - ensures that a file exists and has the correct attributes
- recurse - ensure an entire directory structure exists, with correct attributes
- absent - ensure a file is not present
- symlink - create a symlink
- Check out the `file` state module documentation

TEMPLATING WITH JINJA

- You can imagine a number of scenarios in which you want to customize the configuration based on the server to which you're deploying
- On some distributions, you might want to select a different package to install.
- You might want to customize the `ServerName` variable of the Apache conf to reflect the FQDN of the host.
- Jinja is a templating language that can be used for this purpose:

```
apache:
  pkg.installed:
{% if grains.os == "Ubuntu" %}
    - name: apache2
{% endif %}
  file.managed:
    - source: salt://webserver/default
    - name: /etc/apache2/sites-available/default
    - template: jinja
```

CONTENT REPLACEMENT

```
<VirtualHost *:80>
        ServerName {{ grains.fqdn }}
        ServerAdmin webmaster@{{ grains.fqdn }}

        DocumentRoot /var/www
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        <Directory>
        <Directory /var/www/>
                Options Indexes FollowSymLinks MultiViews
                AllowOverride None
                Order allow,deny
                allow from all
        <Directory>
</VirtualHost>
```

THE SERVICE STATE

- Often, you want to specifically ensure that a service is (or is not) running.
- The Service state module provides this functionality:

```
ssh:
  service.running:
    - enable: True
    - watch:
      - file: /etc/ssh/sshd_config
```

- Let's look at the `service` state module documentation

REQUISITES

- Managing a system often requires steps to be performed in a given order
- Before a service can be started, the package has to be installed. Before the package is installed, a custom package repository may have to be added

require make sure the listed states have succeeded before ensuring this state

require_in make sure that another state is checked after this state

watch if the module implements a `mod_watch` method, call it when a watched state changes

LEARNING HOW TO FISH

- If you find yourself forced to call the `cmd.run` state, chances are you're thinking about the problem imperatively
- There's usually a state for that
- Bookmark and get comfortable skimming the salt documentation: http://docs.saltstack.com/salt-modindex.html

DRY: DON'T REPEAT YOURSELF

- Salt is very powerful at managing systems, but it seems like we might end up with lots of duplication among projects trying to accomplish the same thing
- salt-formulas are an emerging repository of "recipes" for provisioning specific services such as MySQL, PostgreSQL, OpenStack, etc.
- Where possible, we want to use salt-formulas (and contribute back to the community!)

EXAMPLE: INSTALLING MYSQL

- Add a GitFS repository to the salt-master config:

```
fileserver_backend:
  - roots
  - git

gitfs_remotes:
  - git://github.com/saltstack-formulas/mysql-formula.git
```

- Update your top file to install MySQL:

```
base:
  '*':
    - mysql.server
```

SUMMARY

- Salt has a strong community behind it, and the ambition to become *the* infrastructure automation platform of choice
- This introduction has not gone into advanced use cases or all of the options available:
- Advanced topics (potential for future training) include:
  - Pillar for sensitive configuration data
  - salt-virt and salt-cloud for VM management
  - Writing new salt-state-modules (e.g., for bower)
  - The salt reactor system (for creating event-driven automation)

THANKS!