SEE TEXT ONLY

A PRIMER

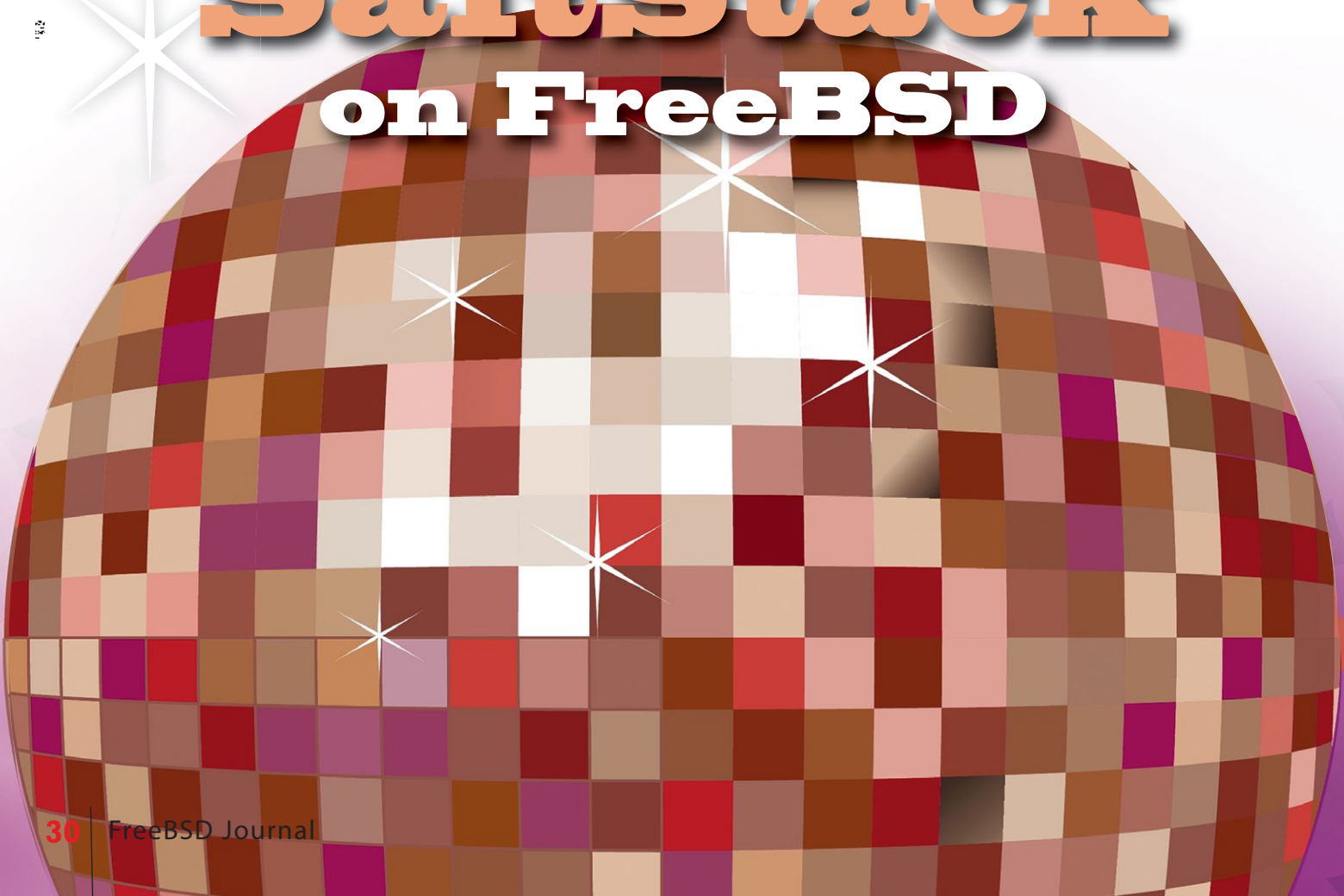
By Peter Wright

SaltStack on FreeBSD

Configuration management comes in many flavors and philosophies. From simple Makefile-based automation tasks to fully orchestrated deployment and administrative environments, there are a multitude of approaches one can take.

In this article, we will take a look at SaltStack, a Python-based configuration management solution that is feature rich and has a reputation for ease of use and scalability. SaltStack also does a very good job in abstracting various OS-specific implementation details from administrator; for example, the same syntax is used to install native OS packages on GNU/Linux as well as on FreeBSD. This makes the task of managing heterogeneous environments much easier, while also encouraging code reuse when possible, regardless of the platform being targeted.

SaltStack is also designed to support large-scale installations. This is primarily due to the architectural decision of using a publish-subscribe messaging model where the Master does not have to craft a configuration directive for each node managed; rather, it can broadcast directives on a channel that clients (Minions in Salt parlance) subscribe to, and, accordingly if the directive applies to them. Communication via the pub/sub method is also encrypted, with user-definable key rotation.



Another feature that SaltStack has implemented that is beneficial for larger installations is what is called event-driven infrastructure (EDI). This allows Salt clients to react to changes in local system state, or events external to SaltStack itself. This is a popular feature for cloud-based infrastructure where capacity is managed dynamically in relation to load on the overall system.

This article is intended as a primer and will take a quick tour of the SaltStack architecture, illustrating that the barrier of entry to start using it is actually quite low. We will also walk through a real-world example of configuring a FreeBSD-based Redis server, then finally highlight some interesting features that I encourage readers to investigate on their own.

Architecture Overview

The architecture of SaltStack is relatively straightforward, yet, due to several design decisions, it is actually quite scalable and easy to customize to suit the specific needs of a given environment. The primary node in a SaltStack cluster is called a Master, and client systems are referred to as Minions. The Master is responsible for publishing messages for the Minions. The Master also manages who is a valid member of a given cluster by accepting an initial key from a Minion, then keeping track of new keys from Minions as they are automatically rotated. In regard to Master/Minion communication, SaltStack uses ZeroMQ as the foundation for its pub/sub message queue.

SaltStack also has the concept of a Grain, akin to Facter in Puppet, which stores system variables such as the name of your OS, the amount of cores available, and what a given Minion's primary IP address is. This is very helpful when developing templates, and is something that will be illustrated below. SaltStack also has the concept of a Pillar that is used to store user-defined variables such as file paths or even passwords. Templates in SaltStack are achieved by using Jinja, and can be used to insert logic into your SaltStack configurations. It is common for them to leverage both Grains and Pillars, allowing for greater flexibility and more succinct configuration directives.

Real World Example

With this background, let's use a real world example of setting up the Redis key/value datastore on a FreeBSD server using SaltStack. First we will set up a SaltStack Master and Minion; then we will walk through a simple SaltStack "statefile" which will deploy a basic Redis configuration.

Preparing a Host to Use SaltStack

Installing SaltStack is straightforward using pkg(7); in fact, the same package is used for installing a SaltStack Master or Minion.

```
% sudo pkg install py27-salt
```

We will skip configuring the salt_master and salt_minion daemons, as that is covered quite well by the SaltStack documentation. After configuring both systems and successfully starting the salt_master daemon, you can then start the salt_minion, which will connect to the Master sending over its initial public key for acceptance. You can view and accept the key using the "salt-key" command on your Master like so:

```
$ sudo salt-key
Accepted Keys:
Denied Keys:
Unaccepted Keys:
test0.com.puter
Rejected Keys:
```

Above, we can see that we have a pending key from "teset0.com.puter". Before accepting the key, you can run "salt-key -f test0.com.puter" on the Master to view and confirm the fingerprint of this key matches the fingerprint of the Minion. Once you have verified the key, you can now accept it like so:

Getting Comfortable with SaltStack Commands

```
$ sudo salt-key -a test0.com.puter
The following keys are going to be accepted:
Unaccepted Keys:
test0.com.puter
Proceed? [n/Y] y
Key for minion test0.com.puter accepted.
```

At this point, we now have a Minion securely paired with our Master, and we can verify this by using a function built into SaltStack that allows for arbitrary commands to be executed. Here we instruct the Master to publish a message stating all hosts matching the test* regular expression to run the "hostname" command. Any systems matching the regex will execute the argument to cmd.run and will post its output back on the message queue for the Master to consume:

```
$ sudo salt 'test*' cmd.run 'hostname'
test0.com.puter:
    test0.com.puter


----------
Summary
----------
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
----------
```

This example actually tells us several other interesting things about SaltStack. Firstly, the Salt command accepts regular expressions as inputs for hosts to target commands against. So, if I were to have a cluster of systems with a common element in their hostnames, I could succinctly target all of them via the Salt CLI tool. Secondly, Salt supports running ad-hoc commands against registered Minions. For example, using this functionality, I can execute an ad-hoc query to verify all my web-servers have the appropriate Apache 2.4 package installed. But let's take a look at a more traditional use of SaltStack, applying a configuration state (called a SaltState) to our new node.

Using SaltState Files to Manage Minions

SaltStack configuration directives are stored in what are called statefiles denoted with the .sls extension. Each SaltStack installation has what is called a "top file" that defines the overall structure and association of SaltStates to hosts under management. For our purposes we will start with a very simple top file that associates two states to our node like so:

```
1  base:
2     '*':
3     - motd
4
5  dev:
6     'test*':
7     - redis_demo
8
```

SaltStack statefiles follow standard YAML syntax and hierarchy rules. In our example above, lines 1–3 define a base class that matches all systems associating a salt state file named "motd". The motd statefile looks like this:

```
1 {% if grains['os'] == 'FreeBSD' %}
2 /etc/motd:
3   file.managed:
4   - source:salt://global_files/motd_freebsd
5   - user: root
6   - group: wheel
7   - mode: 644
8 {% endif %}
```

This example statefile actually has embedded some Jinja templating code inside it, which, in our case, determines if the host we are running on is FreeBSD, and if so, we apply the "motd_freebsd" file to the host with the defined ownership and permissions. The Minion executing the statefile will search its grain inventory checking to see if the value of the "os" key matches FreeBSD. If it matches, we copy over our FreeBSD motd file to /etc/motd, ensuring its ownership and mode.

You can also interact with Grains using the SaltStack CLI like so:

```
$ sudo salt 'test*'  grains.get os
test0.iad0.tribdev.com:
      FreeBSD

----------
Summary
----------
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
----------
$
```

The above command, when invoked on the Master, targets all configured Minions whose name matches the 'test*' regular expression. You can view all available key/value pairs for a given node by substituting "grains.get" with "grains.items".

Being able to embed templating and logic inside a state configuration file is powerful; but SaltStack also allows you to use templating for file resources, which we'll cover next.

Detour into Environment Separation

This is a good time to take a quick tour of how SaltStack organizes files on disk. On my Master, my directory structure of statefiles and file resources looks like so: (next page)

```
$ cd $SALT_ROOT
$ find .
./states/top.sls
./states/global_files/motd_freebsd
./states/dev/states/redis_demo.sls
./states/dev/states/dev_files/redis_demo.conf
./states/dev/states/dev_files/redis_rc
```

One thing you should notice is that our top.sls file is in a separate location as the redis_demo.sls file; redis_demo.sls is a child of the dev/states directory hierarchy. What I have done here is create a separation of environments using this structure. Specifically, the root level contains the top.sls file and our motd_freebsd file. States in this location are all hosts as they have a global scope. Then I have created a "dev" directory structure that contains our redis statefile and supporting file resources. This dev environment is reflected in the previous top file via lines 5–7, and as you'll see in line 6, we are more restrictive in the hosts that are matched via our regular expression. You can also have an arbitrary number of environments in SaltStack, which is very handy. For example, I have several development environments in addition to a pre-production and production environment at my larger SaltStack installation. This allows me to federate who has access to which environments while also helping me ensure changes done in one environment do not affect other environments.

A More Complex Statefile

Now that we've broken down environments and SaltStack filesystem layouts, let's take a look at our redis_demo.sls file that installs several packages and references a templatized file resource:

```
1  /usr/local/etc/redis.conf:
2    file.managed:
3    - require:
4    - pkg: redis_pkgs
5    - source: salt://dev_files/redis_demo.template
6    - template: jinja
7    - user: root
8    - group: wheel
9    - mode: 644
10
11 /var/db/redis/:
12   file.directory:
13   - user: redis
14   - group: redis
15   - mode: 755
16
17 /etc/rc.conf.d/redis:
18   file.managed:
19   - source: salt://dev_files/redis_rc
20   - user: root
21   - group: wheel
22   - mode: 644
23
24  redis_pkgs:
25   pkg.installed:
26   - pkgs:
27         - redis
```

Lines 1–9 define the characteristics of a file deployed to /usr/local/etc/redis.conf. Lines 3–4 create a dependency on the packages defined in lines 24–27 to be installed before this file resource can be deployed. Next, we reference a Jinja template that is our main Redis configuration file at line 5. Finally, through lines 11–22, we ensure a directory exists, with correct permissions for Redis to checkpoint data to disk; and we also ensure the Redis daemon is enabled via rc.

Let's take a look at the redis_demo.template, as that will further illustrate how you can leverage SaltStack Grains and templating to write one configuration file that can be deployed on multiple systems:

```
1 {% set pri_ipv4 = grains['ipv4'][0] %}
2
3 protected-mode no
4 port 6379
5 tcp-backlog 511
6 bind {{ pri_ipv4 }}
7 timeout 0
8
9 tcp-keepalive 300
```

The above snippet represents the first nine lines of a Redis configuration file; it illustrates how Grains and templates can be combined in a pretty powerful manner.

The first line defines a variable "pri_ipv4" and populates it with the first value present in the ipv4 key as reported by the SaltStack Grains system. To view what that Grain will look like in a shell you can execute the following:

```
$ sudo salt 'test*' grains.get ipv4
test0.iad0.tribdev.com:
      - 10.3.16.51
      - 127.0.0.1

----------
Summary
----------
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
----------
```

SaltStack returns these values as a list, and, as such, we have to specify the element in the list that represents the value we would like to use. So, when the Minion evaluates this salt stage, it scans the grain system for a key named "ipv4" and sets the pri_ipv4 variable with the public IPv4 address that is stored. This data is then inserted into the local configuration file on the Minion as per line 6, where "{{ pri_ipv4 }}" is substituted with the IP addresses.

This is a very simple example, but the power should be pretty evident. For example, imagine you manage a fleet of systems spanning several sub-domains. Using Grains and templates, you can abstract many of the per-domain and system configurations in your template and even use logic to determine which parts of a template are rendered on a given Minion. SaltStack also makes it trivial to extend the default Minions by writing simple Python classes.

Applying SaltState Configurations

Now that we have gone through our statefiles, it's time to apply them to our Minion. We will do this by executing the following command from the Master:

```
$ sudo salt 'test*' state.apply
```

This command applies all relevant states—as defined in our top.sls file—to systems whose hostnames begin with test. The output looks like so: (next pages)

```
$ sudo salt   'test*' state.apply
test0.iad0.tribdev.com:
----------
      ID: /etc/motd
      Function: file.managed
      Result: True
      Comment: File /etc/motd is in the correct state
      Started: 23:01:06.397619
      Duration: 581.967 ms
      Changes:
----------
      ID: redis_pkgs
      Function: pkg.installed
      Result: True
      Comment: The following packages were installed/updated: redis
      Started: 23:01:07.450629
      Duration: 1107.676 ms
      Changes:
      ----------
      redis:
      ----------
      new:
      3.2.6
      old:
----------
      ID: /usr/local/etc/redis.conf
      Function: file.managed
      Result: True
      Comment: File /usr/local/etc/redis.conf updated
      Started: 23:01:08.560307
      Duration: 535.906 ms
      Changes:
      ----------
      diff:
      ---
      +++
      @@ -1,1052 +1,82 @@
      -# Redis configuration file example.

< omit multi-page diff of redis.conf>

----------
      ID: /var/db/redis/
      Function: file.directory
      Result: True
      Comment: Directory /var/db/redis is in the correct state
      Started: 23:01:09.096296
      Duration: 0.803 ms
      Changes:
----------
      ID: /etc/rc.conf.d/redis
      Function: file.managed
      Result: True
      Comment: File /etc/rc.conf.d/redis updated
      Started: 23:01:09.097172
      Duration: 521.206 ms
      Changes:
      ----------
```

CONTINUES NEXT PAGE

```
      diff:
      New file
      mode:
      0644

Summary for test0.iad0.tribdev.com
------------
Succeeded: 5 (changed=3)
Failed:    0
------------
Total states run:       5
Total run time:   2.748 s


--------------------------------------------
Summary
--------------------------------------------
# of minions targeted: 1
# of minions returned: 1
# of minions that did not return: 0
# of minions with errors: 0
--------------------------------------------
```

Success! We installed the Redis binary, deployed our custom configuration files for redis and also ensured /etc/motd was deployed and up-to-date. I removed a very long diff that SaltStack reported when the default redis.conf was overwritten. Let's take a look at that configuration though, and ensure that our template and Grain data was applied correctly:

```
$ head -n 10 /usr/local/etc/redis.conf
protected-mode no
port 6379
tcp-backlog 511
bind 10.3.16.51
timeout 0

tcp-keepalive 300
$ ifconfig xn0 | grep inet
      inet 10.3.16.51 netmask 0xffffff00 broadcast 10.3.16.255
$
```

And there we have it! The 0th value of the ipv4 grain has been substituted in our configuration file and it matches the IPv4 address that is associated with xn0 on this host.

Conclusion

This article has barely scratched the surface of SaltStack's capabilities as a configuration management engine. Despite that, hopefully it has given a good overview of its capabilities and demonstrated the flexibility of this tool. In practice, there are several key concepts and components that I have not covered here that the user will want to explore. We'll touch on them briefly, giving you a chance to read up on the SaltStack documentation yourself.

The first concept I completely ignored was nodegroups, which is a method for grouping nodes of similar function together. For example, if I have 10 Minions who act as Apache servers, I could create a nodegroup in my Master configuration named 'webservers'. I would then be able to reference this nodegroup in state files, thus not constraining me to hostname regular expressions as my previous example demonstrated. Here is an example of referencing a nodegroup in my top.sls.

```
dev:
      web-servers
      - match: nodegroup
      - dev_base
```

The above example will apply the "dev_base" salt state to all Minions I've put in my "webservers" nodegroup.

The second feature I bypassed in my demo above was Salt Pillars. They are very similar to Salt Grains, in that they are a key/value registry that

can be queried via statefiles or templates. What sets Pillars apart from Grains is the fact that they are used for user-defined variables. One common use-case would be to store DB credentials as a Pillar key/value pair, which is then referenced by a configuration template on a Minion. Furthermore, SaltStack has a GPG renderer that takes GPG-encrypted input and renders it in un-encrypted form in the Pillar for use by Minions. This method allows administrators to store credentials in a SCCS, yet ensures the encrypted payload itself is protected. ●

PETER WRIGHT is a systems architect currently working at Tronc Inc. helping build scalable and secure systems based on FreeBSD, SaltStack and AWS for the publishing industry. A longtime member of NYCBUG, despite living in Santa Monica, he is always keen to introduce people to FreeBSD. If he's not hacking Unix, you'll probably find him at his favorite beach surfing with his son.

Links

Getting Started with SaltStack (official tutorial):
https://docs.saltstack.com/en/getstarted/

SaltStack Documentation:
https://docs.saltstack.com/en/latest/contents.html

SaltStack Nodegroups:
https://docs.saltstack.com/en/latest/topics/targeting/nodegroups.html

SaltStack GPG Renderer:
https://docs.saltstack.com/en/latest/ref/renderers/all/salt.renderers.gpg.html