An Introduction to

SALTSTACK

by Erik Johnson

What is Salt?

- **Remote Execution**
  - Run commands or functions on many hosts at once
  - Receive results asynchronously as each host returns data to the master
  - Uses the ZeroMQ messaging library
    - Communication takes place over persistent connections
    - No need to re-establish connections for each action (reduces TCP overhead)
    - FAST! FAST! FAST!

What is Salt?

- **Configuration Management**
  - Manage installed packages, running services, configuration files, users, groups, and more using an easy-to-read configuration syntax
  - Keep hosts configured the way you want them
  - Changes to hosts which contradict your desired configuration can easily be reverted
  - Provision cloud computing instances (AWS, Linode, OpenStack, Rackspace, Parallels, DigitalOcean, etc.)
  - Fulfills a similar role as projects like Puppet, Cfengine, Chef, etc.

How is Salt Different?

- Remote execution foundation allows for tremendous versatility
- Run one-off commands on hosts for information gathering purposes, or proactively make changes
  - See the sizes and modified times of log files in /var/log
  - Check which version of a given package is installed on all of your hosts
  - See the network information for all interfaces on a given host
  - Install packages, restart services, etc. on many hosts at once
- CM tools like Puppet have remote execution add-ons (MCollective), while remote execution in Salt is built-in
- Amazingly easy to extend

Basic Terminology

- **Master** - The central server from which Salt commands are run and States are applied
- **Minions** - The hosts you are managing, they maintain a connection to the master and await instructions
- **States** - Directives used for configuration management
- **Modules** - Collections of functions which can be run from the Salt CLI (and are also run under the hood by States)
  - Module functions may also be referred to as **commands**

Installation

- http://docs.saltstack.org/en/latest/topics/installation/index.html
  - Platform-specific installation instructions
- A shell script called **salt-bootstrap** is available, and can be used to install salt-minion on most popular distributions
- If necessary, enable the salt-minion daemon so that it starts at boot, as not all distros will do this for you by default

Start Services

- Edit **/etc/salt/master** on the Master, and start the **salt-master** service
- Edit /etc/salt/minion on the Minion, and start the **salt-minion** service
- The Minion will connect to the IP/hostname configured in the minion config file, or will attempt to connect to the hostname **salt** if no master is configured

Accept the Minion Key

- The Master will not allow the Minion to authenticate until the Minion's public key has been accepted
- This is done using the **salt-key** command
  - `salt-key -a hostname`
    - accepts key for specific host
  - `salt-key -A`
    - accepts all pending keys

Targeting Minions

- Several ways to match
  - **Glob (default):** 'web*.domain.com'
  - **PCRE:** 'web0[1-4].(chi|ny).domain.com'
  - **List:** 'foo.domain.com,bar.domain.com'
  - **Grains:** 'os:CentOS', 'os:Arch*'
  - **Grain PCRE:** 'os:(Linux|.+BSD)'
  - **Nodegroup:** (defined in master config file)
  - **Pillar:** 'proxy_ip:10.1.2.3'

Targeting Minions (cont'd)

- Several ways to match
  - **IP/CIDR:** '10.0.0.0/24', '192.168.10.128/25'
  - **Compound Matching**
    - Use multiple match types in more complex expressions
      - 'G@os:RedHat and web*.domain.com'
      - 'G@kernel:Linux or E@db[0-9]+\.domain.com'
      - 'S@10.1.2.0/24 and G@os:Ubuntu'
  - **Range Expressions**
    - https://github.com/grierj/range/wiki/Introduction-to-Range-with-YAML-files

Data Structure Primer

- A basic understanding of data structures will go a long way towards effectively using Salt
- Salt uses lists and dictionaries extensively
  - **list** - pretty much what it sounds like, a list of items
    - Ex. ["foo", "bar", "baz"]
  - **dictionary** - a set of key/value mappings
    - Ex. {"foo": 1, "bar": 2, "baz": 3}
- Dictionaries can be list items, and dictionary values can be lists or even other dictionaries

YAML

- The default data representation format used in Salt is YAML (http://www.yaml.org/)
- Each nested level of data is indented two spaces
- A dictionary key is followed by a colon
- {"a": {"foo": 1, "bar": 2, "baz": 3}, "b": "hello", "c": "world"} would be represented by the following YAML:

```
a:
  foo: 1
  bar: 2
  baz: 3
b: hello
c: world
```

YAML (cont'd)

- Lists items are prepended with a dash and a space, and all items in the list are indented at the same level
- {"foo": [1, 2, 3], "bar": ["a", "b", "c"], "baz": "qux"} would be represented by the following YAML:

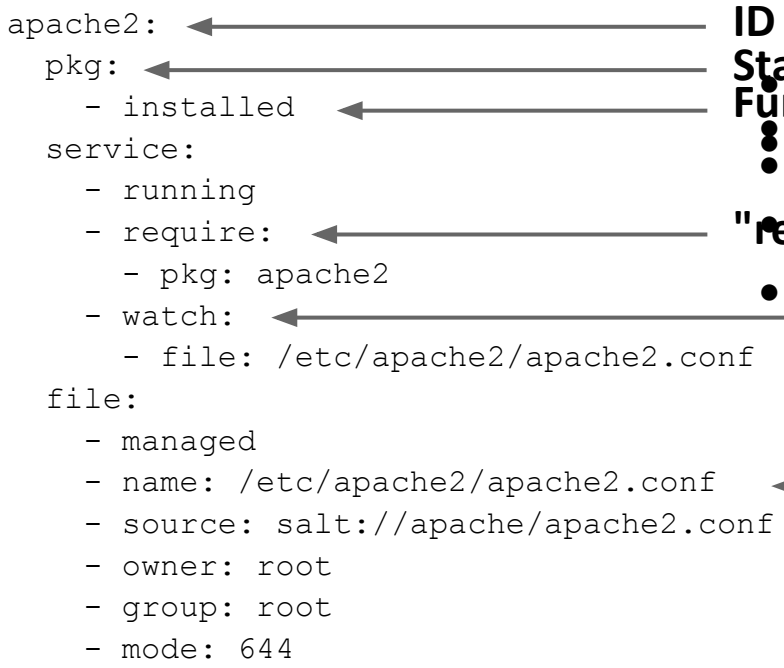
```
foo:
  - 1
  - 2
  - 3
bar:
  - a
  - b
  - c
baz: qux
```

Grains

- Grains are static data that a Minion collects when it first starts
- Similar to ruby's "Facter", which is used by Puppet
  - The major difference between Grains and Facts is that Facts are generated on-the-fly (and thus can change while the Puppet Agent is running)
  - Grains are loaded once when the Minion starts and stay in memory
  - Dynamic information should be retrieved via Module functions
- To view all grains, use the **grains.items** command
  - `sudo salt \* grains.items`
- To view a single grain, use the **grains.item** command
  - `sudo salt \* grains.item os`

Introduction to States

- States are configuration directives which describe the "state" in which you want your hosts to be
- A typical state, represented in YAML, looks like this:



Introduction to States (cont'd)

- When you configure a state, you are really just representing a specific data structure
- This means that your states can be written in any format you wish, so long as you can write a **renderer** that can return the data in the proper structure
- YAML is the default, but Salt provides a JSON renderer, as well as a Python-based Domain Specific Language, and pure Python for even greater control over the data
- You can override the default renderer by setting the **renderer** parameter in the master config file

Using States

- In order to start configuring states, you need to make sure that the **file_roots** parameter is set in the master config file (remember to restart the master when done)
- The respective **file_roots** that you specify will be the root of any **salt://** file paths that you use in your states
- Note that you can have more than one root per environment; if a file is found at the same relative location in more than one root, then the first match wins

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

- If **/srv/salt/foo.conf** and **/home/username/salt/foo.conf** both exist, then **salt://foo.conf** would refer to **/srv/salt/foo.conf**

Using States (cont'd)

- Salt States are kept in SLS files (**S**a**L**t **S**tate Files)
- A simple layout looks like this:

```
top.sls
users.sls
webserver/init.sls
webserver/dev.sls
webserver/files/apache2.conf
```

- In top.sls, you configure which states are applied to which hosts using Salt's targeting system

```
base:
  '*':
    - users
    - webserver
  'dev0[0-9].domain.com':
    - match: pcre
    - webserver.dev
```

Default match type is **glob**, other match types include **pcre**, **list**, **grain**, **grain_pcre**, **pillar**, **nodegroup**, **ipcidr**, **compound**, and **range**.

Using States (cont'd)

- users.sls

```
moe:
  user:
    - present
    - shell: /bin/zsh

larry:
  user:
    - present

curly:
  user:
    - present
```

- If you have a lot of users, there will be a lot of repetition here
- To reduce the amount of SLS code that you need to write, Salt supports templating engines
  - **jinja (default):** http://jinja.pocoo.org/
  - **mako:** http://www.makotemplates.org/
  - **wempy:** http://pypi.python.org/pypi/wempy
- Templating engines are just renderers
- More than one can be used by setting the **renderer** variable in the master config, using a "pipe" syntax
  - `renderer: jinja|mako|yaml`

Using States (cont'd)

- users.sls

```
moe:
  user:
    - present
    - shell: /bin/zsh

larry:
  user:
    - present

curly:
  user:
    - present
```

- An example of this file using a jinja template:

```
{% for username in 'moe', 'larry', 'curly' %}
{{ username }}:
  user:
    - present
{% if username == 'moe' %}
    - shell: /bin/zsh
{% endif %}
{% endfor %}
```

Using States (cont'd)

- Applying states can be done in two ways
  - One or more SLS files at a time, using the **state.sls** command
    - `sudo salt \* state.sls users`
  - Apply all SLS files configured in top.sls, using the **state.highstate** command (recommended)
    - `sudo salt \* state.highstate`
- `test=True` can be appended to the end of either command to see what changes the command would make (but not actually perform them)

Pillar

- Pillar data are user-defined variables
- Dynamic, unlike Grains; can be modified without restarting the minion
- Applied with the same targeting logic and file layout used for States
- Separate file root and top.sls
- Set the **pillar_roots** variable in the master config file (don't forget to restart the master)
- Here is a simple example top.sls for Pillar

```
base:
  '*':
    - users
```

Pillar (cont'd)

```
userdata:
  moe:
    fullname: OhMay
    uid: 1101
    password: $1$TL/F8XPx$Ylxr0TZalM3LnNmBtka8V0
    shell: /bin/zsh

  larry:
    fullname: ArryLay
    uid: 1102
    password: $1$J9Jy3.ke$FOHwZ7nzf6BxEkP9nu.R..

  curly:
    fullname: Curly Cue!
    uid: 1103
    password: $1$V.ciXdRZ$haT79D5N2tgU7I5PkC9aJ0
```

- Going back to our user states from before, we can use Pillar to make them even more flexible by creating a users.sls with more detailed user information
- NOTE: The password hashes at the left are unsalted MD5. Do not use this for passwords!
  - They're only used here so they'll fit in the slide :)

Pillar (cont'd)

```
userdata:
  moe:
    fullname: OhMay
    uid: 1101
    password: $1$TL/F8XPx$Ylxr0TZalM3LnNmBtka8V0
    shell: /bin/zsh

  larry:
    fullname: ArryLay
    uid: 1102
    password: $1$J9Jy3.ke$FOHwZ7nzf6BxEkP9nu.R..

  curly:
    fullname: Curly Cue!
    uid: 1103
    password: $1$V.ciXdRZ$haT79D5N2tgU7I5PkC9aJ0
```

- The templated SLS would now look like this:

```
{% for username, params in
   pillar['userdata'].iteritems() %}
{{ username }}:
  user:
    - present
{% for key, value in
   params.iteritems() %}
    - {{ key }}: {{ value }}
{% endfor %}
{% endfor %}
```

Pillar (cont'd)

Pillar data

```
{% if grains['os'] == 'Ubuntu' %}
apache: apache2
{% elif grains['os_family'] == 'RedHat' %}
apache: httpd
{% endif %}
```

Pkg state

```
{{ salt['pillar.get']('apache', 'apache') }}:
  pkg:
    - installed
  service:
    - running
    - enable: True
```

- Pillar is also useful for values that differ between platforms, such as package names
- Note that the jinja conditional in the pillar SLS could have been placed in the state SLS
  - If you did it this way, you would not need a pillar variable
- **pillar.get** is new in salt 0.14, allowing you to specify a default if the specified pillar variable does not exist
- The normal way of specifying this pillar would be: **{{ pillar['apache'] }}**

Templating Managed Files

- Managed files are files that are deployed using the **file.managed** state
- The same template engines available in SLS are available to managed files
  - Grains and Pillar data are also available, as they can be referenced in templates
- Templating can help you avoid needing to maintain several different copies of a config file for an application if only certain things differ between instances / hosts / physical sites

Templating Managed Files (cont'd)

Config file template

```
[main]
hostname={{ grains['fqdn'] }}
type=web
port={{ pillar['port'] }}
os={{ os }}
somevar={{ somevar }}
```

File state

```
/path/to/config/file:
  file:
    - managed
    - source: salt://config.template.ini
    - user: root
{% if grains['os'] == 'Ubuntu' %}
    - group: sudo
{% elif grains['os_family'] == 'RedHat' %}
    - group: wheel
{% endif %}
    - mode: 644
    - template: jinja
    - context:
      - os: {{ grains['os'] }}
      - somevar: foo
```

- Variables defined in the **context** param will be passed through to the template
- Again, multiple template engines can be used, by setting the **template** param using the "pipe" syntax
  - `template: jinja|mako`

Miscellaneous

- You can include SLS files in other SLS files, allowing "common" SLS code to be written once and re-used in more than one SLS file
  - This is done with an **include** statement at the top of the SLS file

```
include:
  - webserver.common
```

- In addition to Grains and Pillar, Salt Module functions are also available within template code
  - Ex. Retrieving the MAC address for eth0

```
{{ salt['network.hwaddr']('eth0') }}
```

Miscellaneous (cont'd)

- You can override the renderer for a given SLS file by using a "shebang"-like entry at the top of the file
  - Ex: `#jinja|json` or `#py`
- Providers for the service, pkg, etc. states can be overridden from the defaults detected during minion startup
  - https://salt.readthedocs.org/en/latest/ref/states/providers.html
- Each environment defined in the **file_roots** section of the master config can have its own top.sls
  - Defining states for an environment in the **base** environment's top.sls will override the top.sls in any other environment
  - In other words, the **base** top.sls is authoritative

Extending Salt

- Many aspects of Salt are extendable
  - **Modules:** http://docs.saltstack.org/en/latest/ref/modules/index.html
  - **States:** http://docs.saltstack.org/en/latest/ref/states/writing.html
  - **Grains:** http://docs.saltstack.org/en/latest/topics/targeting/grains.html#writing-grains
  - **Renderers:** http://docs.saltstack.org/en/latest/ref/renderers/index.html#writing-renderers
- When designing States/Modules, keep in mind that Modules should do the actual work
- States should check to see if the desired state is already achieved, and (if necessary) invoke Module functions to achieve the desired state
- There are other aspects of Salt, such as **returners**, **outputters**, and **runners**, which can be extended

Get Involved!

- Fork Salt on GitHub and submit pull requests, bug reports, and feature requests
  - https://github.com/saltstack/salt/
- Join the Mailing List
  - https://groups.google.com/group/salt-users
- Chat on IRC (**#salt** on irc.freenode.net)
  - http://webchat.freenode.net/?channels=salt

More Official Salt Stack Projects

- **salt-cloud** - Provision minions on various cloud providers
  - https://github.com/saltstack/salt-cloud
- **salty-vagrant** - Provision Vagrant boxes using Salt
  - https://github.com/saltstack/salty-vagrant
- **salt-api** - Exposes certain aspects of Salt via REST, etc.
  - https://github.com/saltstack/salt-api
- **salt-vim** - Vim plugins to make editing YAML SLS files easier
  - https://github.com/saltstack/salt-vim
- **salt-ui** - Pre-alpha web UI for Salt which uses **salt-api**
  - https://github.com/saltstack/salt-ui

Additional Videos/Demos

- Keep in mind that these (aside from Salt Air) are older videos, and might be outdated as Salt is a *very* actively-developed project
- Intro to Salt Stack (UTOSC 2012)
  - http://youtu.be/q-6v275Kno4
- Managing Web Applications with Salt (UTOSC 2012)
  - http://youtu.be/osGLqv0zPI0
- Remote Execution Demo
  - http://blip.tv/saltstack/salt-installation-configuration-and-remote-execution-5713423
- Thomas Hatch Interviewed on FLOSS Weekly
  - http://twit.tv/show/floss-weekly/191
- Salt Air - Community news, new features, demos, etc.
  - https://www.youtube.com/SaltStack

The End!

- **My Name:** Erik Johnson
- **How to find me:**
  - On Freenode, GitHub, and Twitter under the username **terminalmage**
- These slides available at: http://goo.gl/T8SVz