SaltStack Administration

Created 2017/02/01 by Braun Brelin

What is DevOps?¶

- All about bringing together developers and operations teams
- Automating the environment
- Measuring application performance

Automate everything¶

- Automate code testing
- Automate workflows
- Automate infrastructure

Constant iteration of development¶

- Write new features and bug fixes in small chunks
- Automatically create, test and monitor new software additions

Synchronize environments¶

- Identical environments for development, test and production

Monitor yourself¶

- Adopt iterative processes to monitor and measure code and operations daily.
- Improve ability to quickly respond to customer concerns and market conditions.

Write code rather than perform manual actions¶

- Teams write code to automate configuration management
- Teams write code to handle triggered events.
- Teams write code to allow them to scale their environment

Use source code control¶

- Manage and document all changes to application code
- Manage and document all changes to configurations and documentation.

Adopt proper discipline¶

- Use devops to develop processes in place to manage your environment and your applications
- Put an end to controlled chaos that is the lot of most IT departments
- Understand the control the impact of unforseen events and disasters.

Reduce time to market¶

- Use devops to reduce the time it takes to bring new features and bug fixes from weeks and months to days and hours.
- Deploy frequently using an Agile like methodology.

What does Saltstack allow us to do?¶

- Describe our infrastructure with code.
- Makes our infrastructure scalable
- Makes our infrastructure reliable.
- Gives us consistent environments.
- Better Security
- Ability to quickly duplicate our environments.
- Simplifies auditing and tracking capabilities

Managing infrastructure as code¶

- Gives us revision control
- Allows use of bug tracking and ticketing systems.
- Ability to do peer review before changes happen.
- Gives us infrastucture design patterns.
- Test infrastucture changes the same way we test application changes.

Monitoring and Metrics¶

- Allows to track every possible resource
- Gives us alerts on all services, availability and response times
- Capture, learn and improve
- Share access and data with everyone on the team
- Plan metrics and monitoring into the application lifecycle.
- Easily integrate third party monitoring and logging tools.

What is SaltStack?¶

- A tool to turn your infrastructure into software.
- Automates packing and provisioning of code to your IT environment.
- A massive finite state machine
- Configuration manger for your operations devops and cloudops environments.

Saltstack components¶

- Salt Master
- Salt Minions
- Execution Modules
- States
- Grains
- Pillars
- Top File
- Runners
- Returners
- Reactors
- Salt Cloud
- Salt SSH

Salt Master¶

- Central Management Server
- Used to send commands and configuration data to salt minions running on managed systems.

Salt Minion¶

- Software that runs on a managed system.
- Receives commands and configuration data from the Salt master

Execution Modules¶

- Ad hoc commands executed from the command line against one or more managed systems.
- perform real-time monitoring, status and inventory
- Run one-off commands or scripts
- Deploying critical updates.

States¶

- A representation of a system configuration. Can be declarative or imperative.

Grains¶

- Static system information about the managed system.
- Examples include the operating system. CPU, Memory, and other properties.
- You can define customized grains (properties) for your systems.

Pillars¶

- User defined properties.
- Stored securely on the salt master.

- Assignable to one or more minions using targets
- Examples include network ports, file paths, passwords, configuration parameters

Top file¶

- Matches states and pillar data to minions

Runners¶

- Modules that execute on the salt master to perform tasks.
- Examples include runners that report job status, connection status, and query minions

Returners¶

- Modules that can return data to various sources, such as a database.
- Returners can run on the Salt minions or the Salt master.

Reactor¶

- Component that can be programmed to trigger reactions to specific events.

Salt Cloud¶

- Allows provisioning of systems on cloud providers or hypervisors
- Immediately brings these systems under salt management.

Salt SSH¶

- Allows us to run Salt commands over SSH for systems that do not have a minion provisioned.

Architectural overview of remote execution¶

Salt is a highly configurable, scalable and robust application that allows one master to manage thousands of minions simultaneously with almost no performance loss. This means that the time required to update many minions is at worst a logarithmic function rather than a linear or exponential function.

All Salt minions receive commands at the same time. Salt can use multiple protocols to achieve this, however the recommended (and default) protocol is the ZeroMQ message queuing system. While Salt can interface with many types of data stores, it's real win is being able to query minions in real time.

While Salt mainstains a master/slave relationship between the master and the minions, communication with the minions is mostly a set of instructions or commands to run. The minion is responsible for doing the heavy lifting and returning the results back to the master.

Salt normalizes commands between different hardware platforms. All commands and states are the same regardless of the underlying operating system being run on the minion.

Salt will run wherever you can install Python. If a minion cannot run Python, then Salt provides a *proxy minion* that can interface with the real minion and issue commands on behalf of the master un the minion's native protocol. Additionally, the proxy parses and returns all output from the minion back to the master.

Salt can perform all of its functions without requiring or using a programming language (Although you will get its best performance if you know some of the Python programming language).

Everything in Salt is extensible. You can even change the underlying network protocol. You can create your own modules as well. Following is an example of the remote execution architecture diagram.

The preceeding diagram shows an example of the execution model. Note that this diagram omits the configuration management portion of SaltStack and focuses on the remote execution subsystem. It shows that when a job runs, many different subsystems of Salt. Looking at this diagram we see the following

- Start jobs by calling the command line interface, a RESTful service or a Python script.
- Output is formatted by default as YAML, however, other formats, such as JSON, plain text and other formats are also supported.
- Results can be stored in many ways, SQL databases, NoSQL databases, flat text files, or any other data store.
- The job returner can be any one of a number of plugins to other applications, or even not exist at all.
- Salt can abstract details of the underlying O/S or applications by using the concept of a *virtual module*. For example the *pkg* module contains support for both the Debian *aptpkg* and the Red Hat *yumpkg* modules, however, we only need to call the *pkg* module directly.

Salt uses a publish/subscribe model (also known as an observer pattern). The Salt minion subscribes to the salt master via a persistent connection. When an even happens, the master publishes messages that the minions can receive and act upon.

Salt uses Public Key Infrastructure for secure authentication.
The Salt minion on starting up for the first time looks for a master named *salt* (The default name). The minion sends its public key to the master. The master must accept the key, either through manual intervention (The salt-key command) or through an automated process. Note that the minion won't be able to run any commands until this key is accepted.
Encryption is achieved through the use of the Advanced Encryption Standard (AES). The AES key is changed everytime the salt-master reboots or a minion key is deleted.
Access Control verification is done through the Publisher ACL (or whatever ACL standard you choose). Every command run on a minion must be verified through this ACL to make sure that the command sender has permission to do so.

Saltstack basic commands¶

The four commands used most often on the command line are:

1. /usr/bin/salt
2. /usr/bin/salt-run
3. /usr/bin/salt-call
4. /usr/bin/salt-key

Of those four commands the one most often used is the salt command.

The salt command¶

The /usr/bin/salt command runs on the salt master and allows salt administrators to execute jobs on salt minions.
The salt command takes as its first argument the *minion target.* This target can be the wild card operator '*', a specific minion or set of minions, or other criteria, such as a regular expression.

An example of the salt command is:

```
salt '*' test.ping
```

The preceeding command will execute the test.ping method on all salt minions managed by this salt master. test is the salt module name, and ping is the method defined inside the test module. This example shows one of the basic functions of Saltstack, remote execution.

Saltstack comes with a large number of these execution modules, some of which will be explored in this course.

Salt-call. Execution on the minion.¶

Salt-call allows us to run a specific command on a minion itself rather than on the master. Usually, this is used for debugging and testing purposes. Note that you can use this command on the salt master itself because the salt master is running a minion as well.

```
salt test.ping
```

Note that in this example, we don't need to specify a minion target as the salt-call command runs only on the localhost.

Salt-call is also used in a salt topology where there is no master, only minions.

Salt-run. Executing directly on the master.¶

Salt-run allows jobs to run directly on the salt master. This is useful for a number of reasons

1. Coordinating tasks across minions. With the salt command, all minions execute the job concurrently. However, there are times when it is desirable to have the minions execute a job sequentially instead.
2. Accessing data that is only available on the master.

```
salt-run manage.up
```

Internally, the manage.up function calls test.ping. It is used to determine whether the minion is up and running and managed by the salt master. As with salt-call, no minions are targeted. This command runs only on the salt master.

Salt-key, managing the Salt PKI infrastructure.¶

Salt communicates between the master and the minions using an encrypted channel for security. Additionally, authentication and authorization must be established between the master and the minion.

The salt master keeps a record of all its minions and whether or not they are trusted. There are three possible states.

1. Accepted. The master has accepted the public key of the minion.
2. Unaccepted. The master has not yet accepted the public key of the minion.
3. Rejected. The master has rejected the public key of the minion.

We use the salt-key command to view the public key infrastructure of Saltstack.
Simply typing in *salt-key* on the command line will print out the trust state of all the minions.

The Saltstack PKI file structure¶

Master Files	Description
/etc/salt/pki/master/master.{pub.pem}	The Salt Master's public and private keys
/etc/salt/pk/master/minions	Public keys of every accepted minion
/etc/salt/pki/master/minions_pre	Public key of every unaccepted minion
/etc/salt/pki/master/minions_rejected	Publc key of every unaccepted minion

Minion files	Description
/etc/salt/pki/minion/minion.{pub,pem}	Minion's public and private key
/etc/salt/pki/minion/minion_master.pub	The public key of the salt master

The Salt publish/subscribe model.¶

Saltstack works using a *publish/subscribe* model. The saltmaster will publish events on the ZeroMQ bus that the salt minions will subscribe to. The minions look at the target in the message to see if this message is relevant to it.

The master sets up two sockets. Port 4505 is the *publish_port*. The minions subscribe to the publish port listening for any commands that it needs to run. Anything that is returned from the execution is sent via port 4506 back to the master. Any data returned is processed by a *returner*.

Accessing Saltstack documentation.¶

Saltstack is written in the Python programming language. One of the features of Python is that of *docstrings*. Docstrings are simply multiline comments that usually accompany modules, classes and functions. Saltstack contains extensive docstrings for its codebase. To access these docstrings, we can use the *sys* execution module.

To print out the documentation from any execution module simply use the sys.doc command. For example:

```
salt salt.master sys.doc test.ping
```

This command will print out the doc string for the ping method from the test execution module.

If you want to see all of the functions in a module, simply pass the module name to the sys.doc method as the argument. For example:

```
salt salt.master sys.doc test
```

Note that if you want to see the documentation for all of the modules in the system, simply run sys.doc with no arguments. However, be aware that it will print a very large amount of text to the screen.

The sys module has a number of other functions that can assist in understanding salt. For example:

```
salt salt.master sys.list_modules
salt salt.master sys.list_functions
salt salt.master sys.argspec
```

The first command lists all of the modules available in the current system.
The second command lists all functions within a given module.
The third command lists all of the arguments that a given function requires when called.

Salt States¶

The core of the Salt system is the *Salt state*. States are kept in .sls files known as Salt State Files. Salt State files are data structures that define the desired state of any minion. These state files are implemented in Python as Python data structures, such as lists, dictionarys, strings and so on.

Salt state files are kept in a heirarchical format known as a state tree with the root of the tree being the *top.sls* file. Beneath the top level directory you can create directories for each type of state that you wish.

Here's an example of a state tree hierarchy.

Salt states are stored in files with an *.sls* extension. These files are in YAML format. YAML is a list of key and value pairs. To look at the documentation for state functions, we can use the sys.state_doc command.

Note that to run a state, we use an execution module called the *state* module. Don't confuse the state module with the concept of a state in Saltstack. The state module is what is used by salt to actually implement the defined states that we define.

Introduction to YAML¶

YAML stands for YAML Ain't a Markup Language. YAML is a text based format for creating structured data that is very readable and understandable. This differs in marked contrast to other types of markup languages such as XML, which is notorious for being difficult to read and expensive to parse.
YAML consists of keys and values, For example let us consider a YAML file which describes an automibile:

```
\# A YAML example
Mercedes:
  Type:
    -
      Hatchback:
        Fuel: "-- 'Petrol' -- 'Diesel'"
        Model: A
        Weight: 1400
    -
      Saloon:
        Fuel: "-- 'Petrol' -- 'Diesel' -- 'Electric'"
        Model: C
        Weight: 2250
```

Note that we have a top level key, 'Mercedes', which maps to a complex value. We have a list called 'Type' with two elements. We also have an internal list 'Fuel' which is part of the value mapped to 'Type'. Notice that YAML uses indentation to resolve heirarchy. 'Type is part of the 'Mercedes' key because it has been indented. If it were on the same indentation level, then it would be a completely separate key/value pair.

YAML Best Practices

- Use UTF-8 encoding when saving files.
- Always use spaces for indentation, never tabs.
- Edit with a text editor, not a WYSIWYG editor such as Microsoft Word.
- Use a monowidth/monospace font such as Courier when viewing YAML files.

You set up the state tree by defining values in the /etc/salt/master configuration file. The *file_roots:* directive is where you would put file paths. For example:

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

This sets the state tree root to /srv/salt, which is the default. Note that if you change this value, you must restart the salt master service in order for the change to take effect.

The first state that we define in the state tree's root is the *top.sls* file.

```
  base:
    '\*':
      - webserver
```

The first directive 'base' refers to the default state environment. The '*' refers to which minions the directive applies. The '*' refers to all minions. Each minion will have a state defined by the 'webserver' directive, which in itself is another state.

Now let's create the webserver state file. This file will be located in the base directory and be called *webserver.sls*.

```
apache: # ID declaration
  pkg: # state declaration
    - installed # function declaration
```

Note that the first line 'apache:' defines the state's ID. The second line declares that we use a salt module called *pkg*. The third line calls a specific function from the pkg module, *installed*.

Effectively what happens is that when this state is applied, Salt will install this specific package on all minions to whom this state applies to, in our case, it will install apache on all minions that Salt controls.

We use the *state.apply* command from salt to apply the defined states to the minions. Note that if we run this command without any arguments, it will start from the top.sls file and work down the state tree heirarchy. Note that this action is known as the *highstate*.

State ordering¶

In a simple configuration, it may be enough that each states runs sequentially, one after another, however, most use cases require that some states run before other states due to some requirement. This is known as *state order*.

There are many salt declaration that we can use to modify our sequential state order. Here are four of them.

- require
- watch
- prereq
- onfail

A require declaration in a state means that in order for that state to run, the state referenced in the requirement statement must be run first.

This is an example of a non-sequential state ordering using require. Here we have a state *apache*. This will manage the web server on the minion. We'd like to make sure that the apache service is starting, however, we need to specify a requirement that the service is installed before we can run it. Additionally, we want to be able to manage the home page, *index.html*. We will store it in our local

repository

```
apache:
  pkg.installed: []
  service.running:
    - require
      - pkg: apache

/var/www/index.html: # ID declaration
  file: # state declaration
    - managed # function
    - source: salt://webserver/index.html # function arg
    - require: # requisite declaration
      - pkg: apache # requisite reference
```

A watch declaration says that if a state has changed (for example a configuration file has been modified) then the state will be re-run via the state.apply method. Note that not all state modules have a watch declaration defined for it. Check the documentation to make sure that watch is defined for that particular state.

Now we have defined our apache service and are managing the home page. Let's take it one step further. Assume we have a configuration file *httpd-vhosts.conf*. Anytime this file changes, we'd like to have salt restart the web service on all the targeted minions. Let's add a line to the apache.sls file to do this.

```
apache:
  pkg.installed: []
  service.running:
    - watch:
      - file: /etc/httpd/extra/httpd-vhosts.conf
    - require
      - pkg: apache

/etc/httpd/extra/httpd-vhosts.conf:
  file.managed:
    - source: salt://webserver/httpd-vhosts.conf

/var/www/index.html: # ID declaration
  file: # state declaration
    - managed # function
    - source: salt://webserver/index.html # function arg
    - require: # requisite declaration
      - pkg: apache # requisite reference
```

Let's look at another example of a state file. This state file manages the *nginx* web server.

Next we have the prereq declaration. The prereq declaration (new in the 0.16 version of Saltstack) allows us to only run a specific state if another state, which hasn't yet executed, is going to run.

```
nginx: # <--- The state identifier
  pkg:
    - installed # < --- This is the pkg.installed fuction. I.e. we want the nginx package installed
  service.running: # <--- This is the service.running function. It will automatically start nginx.
    - watch: # <--- When certain files change, we want to restart the nginx server. Watch tells salt
             # to monitor the files listed below and restart when they change.
      - pkg: nginx
      - file: /etc/nginx/nginx.conf # <--- File to watch for.

# The next two sections describe how we manage the files specified on the monitor watch list.
   /etc/nginx/nginx.conf:
     file.managed:
       - source: salt://nginx/files/etc/nginx/nginx.conf
       - user: root
        - group: root
        - mode: 640

# This section controls the nginx defaults file. We'd like to customize this for each web server we se
# To do this, we use a jinja template. We put the .jinja on the end of the filename so we know that it
template. Also, we tell Salt to use jinja as the template format.
  /etc/nginx/sites-available/default:
     file.managed:
       - source: salt://nginx/files/etc/nginx/sites-available/default.jinja
       - template: jinja
       - user: root
       - group: root
       - mode: 640

# The sites-enabled default file is set up as a symbolic link. Note that we require the source file in
# link to be available before we try and set up the symbolic link.
/etc/nginx/sites-enabled/default:
   file.symlink:
     - target: /etc/nginx/sites-available/default
       - require:
       - file: /etc/nginx/sites-available/default
```

For example, we can do the following:

```
graceful-down: cmd.run:
- name: service apache graceful
- prereq:
  - file: site-code
```

site-code: file.recurse:

```
- name: /opt/site_code
- source: salt://site/code
```

In this example, we will shutdown the apache server, but only if any of the code on the site has changed.

The *onfail* declaration says that state A will only run if state B has failed.
For example:

```
primary_mount: mount.mounted:

- name: /mnt/share
- device: 10.0.0.45:/share
- fstype: nfs


backup_mount: mount.mounted:

- name: /mnt/share
- device: 192.168.40.34:/share
- fstype: nfs
- onfail:
  - mount: primary_mount
```

Here we see an example of using the mount state module to attempt to perform a remote mount via NFS. The backup_mount state will only run if the primary_mount state fails, in other words, if the primary NFS server is offline or, for some reason, the client minion cannot mount the file system from

the primary_mount master.

Once the sls file is written, we need to be able to test it. To do this, run the following command:]

```
salt-call state.apply -l debug
```

Salt Grains¶

What are Grains? Grains are pieces of information, ususally about the minion itself. For example grains may include information about the Operating System distribution and type or the hardware platform, such as CPU or memory.

SaltStack provides many grains called *core* grains immediately. You can also define customized grains at your convenience.

To find out what grains are available by default, you can list them from the grains module by typing in the following command:

```
salt '*' grains.ls
```

You can also list the values in the salt grains by using the command:

```
salt '*' grains.items
```

Customized grains can be stored in different places.

- The minion configuration file.
- The grain configuration file in /etc/salt/grains

Here is an example of storing customized grains in the minion configuration file.

```
grains:
  roles:
     - webserver
     - memcache
  deployment: datacenter4
  cabinet: 13
  cab_u: 14-15
```

And here is an example of storing customized grains in the /etc/salt/grains file

```
 roles:
     - webserver
     - memcache
  deployment: datacenter4
  cabinet: 13
  cab_u: 14-15
```

Because the grain is already stored in the grain configuration file, we don't need the top level 'grain' key.

Customized grains can override the core grain objects. When Salt loads grains it has an order of precedence. Custom grains in /etc/salt/minion.

1. Custom grain modules in _grains directory, synced to minions.
2. Custom grains in /etc/salt/grains.
3. Custom grains in /etc/salt/minion.
4. Core grains.

If it finds the grain in anyone of these areas, it finishes its lookup. I.e. a Custom grain with the same name as the core grain will be found and used instead of the core grain.

When a grain is changed in some way, we need to re-sync it for all or some of the minions who's states depend on it. We can do this one of two ways.

```
saltutil.sync.grains
or
saltutil.sync_all
```

Salt Pillars¶

Pillars are somewhat similar to Grains in that they store data about minions. They're tree-like structures that store data about minions on the salt master. They allow confidential, secure data to be stored and passed on only to the minions who have allowed access to it.

Pillars data is useful for:

- Sensitive data such as passwords.
- Minion configuration
- Variables that are specific to a minion or group of minions and accessible via a template formula.
- Any arbitrary data needed.

The pillar subsystem runs in salt by default. To look at a minion's pillar data, type

```
salt '*' pillar.items
```

The default root for pillars is located in /srv/pillars. This can be changed in the master's configuration file via the key pillar_root:
As we saw with states, we can create a top level file *top.sls*. We can then define a pillar heirarchy underneath it. For example, consider a pillar heirarchy with a *users* pillar underneath the root. We define our top.sls fil e in /srv/pillar like so:

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

Now, underneath /srv/pillars/users, we can create an init.sls file that looks like this:

Later on, we'll see how we can use the Jinja2 templating system to parameterize pillar data and use it within states.

```
users:
  bbrelin: 1000
  bobama: 1001
  hclinton: 1002
  bsanders: 1003
```

Salt Grains vs Pillars¶

Both grains and Pillars define input data to parameterize Salt states.

Depending on the purpose of data, one should make a choice to put it in one place or another.

NOTE:

Here we talk about the practical differences between Grains and Pillars for the default use case only.

Differences	Grains	Pillars	
Info which...	... Minion knows about itself	... Minion asks Master about	
Distributed:	Yes	No	
Centralized:	No	Yes	
Computed automatically:	Yes (mostly should)	No	Default use case
Assigned manually:	No (or kept to minimum)	Yes	In our default case the entire system is
Conceptually intrinsic to...	... individual Minion node	... entire system managed by Master	
Data under revision control:	No (if static values)	Yes	
Defines...	provided resources	required resources	

managed by a single person (authority).

By default Grains are both:

good case: evaluated automatically by Minion software in built-in (like functions in core.py) or custom Grains;
bad case: assigned manually in /etc/salt/minion file (or files it includes) on each Minion individually. By default Pillars are data defined manually in files under /srv/pillars. If not sure, it is almost always good to define required data in pillar (first).

Distributed vs Centralized

Because Grains are distributed, they are disadvantageous for manual management: one has to access Minions individually to configure Grains for specific Minion.

Instead, it is more convenient to define manual data centrally in Pillars.

It is crucial to differentiate between static and non-static custom Grains:

Static grains have to be distributed across Minions (in their configuration files). Non-static grains are effectively centralized on Master and pushed to Minions for evaluation. Probably, the only static Grain one wants to define is Minion id. The rest information about Minion should either be assigned to it through Pillars (using Minion id as a key) or evaluated in non-static grains.

In other words:

static Grains are the bad (impractical) use of Grains; non-static Grains are the good (practical) use of Grains. This also suggests that:

Pillars values are normally under revision control centrally on Master; static Grains values are less convenient to be kept revisioned (or even used); source code (not values) for non-static Grains is also normally under revision control on Master. Manual vs Automatic

Because Grains can be computed automatically, they are meant to collect information which is Minion-specific and unknown in advance.

The only practical cases for custom Grains are non-static ones with automatic evaluation on the minion.

Pillars are manually defined for the required state of the system which Salt is supposed to manage and enforce.

System-wide vs Node-specific

Pillars suit more to define (top-down) entire system of Minions as required: roles, credentials, platforms, policies, etc. Grains suit more to define (bottom-up) environment of individual Minions as provided. Good vs Bad

Defining good as practical and bad as impractical is heavily influenced by managing system with two hands at a time (single person).

In other cases, static grains may actually be very useful to let (multiple) owners of individual hodes manage preferences. This effectively makes distributed nature of static Grains match nature of responsibilities distributed among the owners. This is simply not default use case.

The Jinja2 templating system.¶

So far, all of the examples of salt files, such as states, pillars and grains that we've seen having been using static YAML files. That is, all of the variables have been directly assigned values in YAML. However, many times, we want to be able to assign values to these variables dynamically, for example if I want to perform some action on a minion based on its operating system, then we need a new tool allow this. That tool is *Jinja2*.
Jinja2 is known a templating system, that is, a system that allows us to replace the value of variables dynamically at runtime.
Jinja2 adds limited programming capabilities to your state, pillar and custom grain files.

Let us consider the following problem. When we want to install the packages Apache web server and vim text editor on a minion, the name of the web server varies from distribution to distribution. In a RedHat based distribution, the server is called 'httpd' and vim is called 'vim-enhanced'. In a Debian based distribution, the server is called 'apache' and vim is just called 'vim'. This means that we need to be able to distinguish on the fly what type of Linux distribution is being run on the minion. To do this, let's create a *pkg* pillar.

Here we define a pillar file called */srv/pillar/pkg/init.sls*

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

Now we add this pkg.sls to the pillars top.sls file

```
base:   '*':
    - data
    - users
    - pkg
```

Now, instead of having to write YAML text to cover every possible type of Linux O/S distribution, we can write the following into our /srv/packages/apache/init.sls file:

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

The Jinja template expression here is looking up the pillar.get function inside the main salt dictionary, and then runs it with the following parameters 'pkgs:apache' and 'httpd'.
Saltstack then goes to the the apache init.sls file and runs the Jinja2 expression. From there it goes to the pkg init.sls and evaluates which type of O/S the minion is running which will get returned in to the 'apache' key. If it isn't running either of the two Linux distribution, the default evaluates to 'httpd'.

Notice the use of the Jinja2 templating system in the above examples. Jinja2 statements begin and end with one of the following types.'

- '{%' and '%}' for Jinja statements.
- '{{' and '}}' for Jinja expressions.
- '{#' and '#}' for Jinja comments.

When salt reads the pkgs file prior to executing it, it evaluates the Jinja2 statements and produces a formatted output file in YAML format (the default output) which will be evaluated.

Jinja has some limited control flow capabilities. For example:

```
{% if grains['os'] != 'FreeBSD' %}
tcsh:
    pkg:
        - installed
{% endif %}
```

Here we see an example of the if statement. Note that all if's must end with an 'endif' statement in Jinja. This template code checks the value of the 'os' grain. If the minion is not running FreeBSD, then install the 'tcsh' shell on the system.

Let's see a further example.

```
motd:
  file.managed:
    {% if grains['os'] == 'FreeBSD' %}
    - name: /etc/motd
    {% elif grains['os'] == 'Debian' %}
    - name: /etc/motd.tail
    {% endif %}
    - source: salt://motd
```

In this example we have a compound if statement. Here we check to see if the minion is running FreeBSD. If it is, then the name of the motd file will be set to /etc/mod, otherwise we'll set the file name to motd.tail. We then tell salt that the default location source for this file is in /srv/salt/base/motd/httpd.conf by referring to the salt URI.

We can also use for loops to iterate over collections of data. Continuing our motd example:

```
{% set motd = ['/etc/motd'] %}
{% if grains['os'] == 'Debian' %}
  {% set motd = ['/etc/motd.tail', '/var/run/motd'] %}
{% endif %}

{% for motdfile in motd %}
{{ motdfile }}:
    file.managed:
    - source: salt://motd
{% endfor %}
```

Here, we set a list variable called *motd* to have as it's first element the default location and name of the motd file. We test our os grain and if it's value is set to Debian, then we change the file name and add a second element. '/var/run/motd', to the motd list.

We can then loop over each element of the motd list and run the file.managed method setting each file's source location to /srv/salt/base/motd.

One of the real advantages of the jinja templating system is that allows us to expand state files without having to completely rewrite them. Let's consider the following problem. We have a QA and a development environment. Each environment has a different .vimrc file with different settings to better fit which ever environment needed. How can we deploy this so that QA servers get the .qavimrc and development servers get the .devrimrc files? Let's first look at our naive vim state implementation located in /srv/salt/edit/vim.sls.

```
vim:
  pkg.installed: []

/etc/vimrc:
  file.managed:
    - source: salt://edit/vimrc
    - mode: 644
    - user: root
    - group: root
    - require:
      - pkg: vim
```

This pillar deploys a vimrc file that is the same across all of our environments. So let's change it to be more intelligent by determing which type of vimrc file we want to install.

Here is our new /srv/salt/edit/vim.sls state file.

```
vim:
  pkg.installed:
    - name: {{ pillar['pkgs']['vim'] }}

/etc/vimrc:
  file.managed:
    - source: {{ pillar['vimrc'] }}
    - mode: 644
    - user: root
    - group: root
    - require:
      - pkg: vim
```

And here is our vim pillar in /srv/pillar/edit/vim.sls

```
{% if grains['id'].startswith('dev') %}
vimrc: salt://edit/dev_vimrc
{% elif grains['id'].startswith('qa') %}
vimrc: salt://edit/qa_vimrc
{% else %}
vimrc: salt://edit/vimrc
{% endif %}
```

Note that the pillar file defines the value of the variable vimrc by checking to see if the id of the minion starts with a 'qa' or a 'dev'. This assumes, of course that your organization has a naming convention for their servers such that all development servers start with a 'dev' and all qa servers start with a 'qa'. Thus, the vimrc variable defines the source location for the vimrc file.

Salt Installation and Configuration¶

- Getting Salt
- Source Installation
- Packaged Installation
- Salt Bootstrap
- Salt Master Network Ports
- Minion Firewall
- Basic Minion Configuration
- Salt Security
- Installing salt on Windows minions.
- Lab 1. Installing Salt

Getting Salt¶

- Installing from source on Linux.
- Installing on Ubuntu via apt.

Installing from source¶

Make sure that you have git installed on your system. Then run the following command

```
git clone https://github.com/saltstack/salt
```

Cloning the repository is enough to start working with Salt and contributing to the source code. You may wish, however to fetch additional tags from git. Salt needs to be able to report the correct version for itself. First, we need to add the git repository as an upstream source.

```
git remote add upstream https://github.com/saltstack/salt
```

Then fetch tags with the 'git fetch' utility.

```
git fetch --tags upstream
```

As of this writing, Saltstack only works with Python version 2. However, Python 3 is now in wide deployment. This means that we'll want a virtual environment set up.

We can then create a new virtual environment with virtualenv.

```
virtualenv --system-site-packages -p /path/to/your/python2/installation
/path/to/your/virtualenv
```

Once you have virtualenv installed, run it like so:

```
source /path/to/your/virtualenv/bin/activate
```

A quick note to Arch Linux users, Python 3 is the default environment. Use virtualenv2 rather than virtualenv. If you're using another distribution besides Debian or Ubuntu, and you are installing M2Crypto via pip, then you must make sure that you have the gcc C compiler installed.

Now you can install Salt into your virtual environment.

```
pip install pyzmq PyYAML pycrypto msgpack-python jinja2 psutil futures tornado pip
install -e ./salt # the path to the salt git clone from above
```

Note: Don't install the M2Crypto library from pip if you're using Debian or Ubuntu. They have a patched version of OpenSSL and you need that version before you can use M2Crypto. Instead of using pip, use apt like so:

```
apt-get install python-m2crypto
```

Once you have your virtual environment running, copy the salt master and salt minion configuration files into your virtual environment.

```
mkdir -p /path/to/your/virtualenv/etc/salt
cp ./salt/conf/master ./salt/conf/minion /path/to/your/virtualenv/etc/salt/
```

Now, you'll need to edit your master configuration file.

1. Uncomment and change the user: root value to your own user.
2. Uncomment and change the root_dir: / value to point to /path/to/your/virtualenv.
3. If you are running version 0.11.1 or older, uncomment, and change the pidfile: /var/run/salt-master.pid value to point to /path/to/your/virtualenv/salt-master.pid.
4. If you are also running a non-development version of Salt you will have to change the publish_port and ret_port values as well.

And also edit the minion configuration file.

1. Repeat the edits you made in the master config for the user and root_dir values as well as any port changes.
2. If you are running version 0.11.1 or older, uncomment, and change the pidfile: /var/run/salt-minion.pid value to point to /path/to/your/virtualenv/salt-minion.pid.
3. Uncomment and change the master: salt value to point at localhost.
4. Uncomment and change the id: value to something descriptive like "saltdev". This isn't strictly necessary but it will serve as a reminder of which Salt installation you are working with.
5. If you changed the ret_port value in the master config because you are also running a non-development version of Salt, then you will have to change the master_port value in the minion config to match.

Start up the master and the minion, accept the minion's RSA key and verify that your local Salt installation is working.

```
cd /path/to/your/virtualenv
salt-master -c ./etc/salt -d
salt-minion -c ./etc/salt -d
salt-key -c ./etc/salt -L
salt-key -c ./etc/salt -A
salt -c ./etc/salt '*' test.ping
```

Note that running the salt master and the minion with the *-l debug* option adds debugging output. If you want the output to go to the console rather than the log file, remove the -d option from the run commands.

Installing from Ubuntu with apt¶

Saltstack has a PPA to allow installation of Salt on Ubuntu. Run the following command as root

```
root@saltmaster:~# apt-get --yes -q install python-software properties
root@saltmaster:~# add-apt-repository ppa:saltstack/salt
You are about to add the following PPA to your system:
Salt, the remote execution and configuration management tool.
More info: https://launchpad.net/~saltstack/+archive/salt
Press [ENTER] to continue or ctrl-c to cancel adding it
```

You must press the [Enter] key, otherwise it won't add the repository

Make sure you update Apt's package index.

```
root@saltmaster:~# apt-get --yes -q update
```

Also, install the Python development package.

```
root@saltmaster:~# apt-get -y python-dev
```

Then install the salt-master package.

```
root@saltmaster:~# apt-get --yes -q install salt-master
```

Now we can configure the salt master. Let's do a real example of a simple configuration.

1. Edit the /etc/salt/master configuration file and configure the interface parameter. Change it's value to the IP address of the salt master.
2. Change the base directory for the salt states files. The default is /srv/salt. Change that to /salt/states/base. To do that, uncomment and edit the *file_roots:* parameter with the new value.
3. Create a new development environment in the top.sls file by adding the following:

```
file_roots:
base:
\- /salt/states/base
development:
\- /salt/states/dev
```

4. Set the pillar_roots

Salt now has a module *chocolately* which can be used to call chocolately commands. For example, on a windows minion, if you want to install a specific package you can run

```
salt '\*' chocolately.install
    base:
    \- /salt/pillars/base
```

```
    Don't forget to create the directories specified in the salt master file!
```

Once we have the master configured, we can then create a minion on a remote machine. Note that we'll have to log in to the remote system to set up its configuration. Once that is done, however, you may never need to log into it again.

Start by installing the required python libraries onto the minion.

```
    root@saltminion:~# apt-get --yes -q install python-software-properties
```

```
    Add the repository to apt.
```

```
    root@saltminion:~# add-apt-repository ppa:saltstack/salt
    You are about to add the following PPA to your system:
    Salt, the remote execution and configuration management tool.
    More info: https://launchpad.net/~saltstack/+archive/salt
    Press [ENTER] to continue or ctrl-c to cancel adding it
```

```
    Update apt with apt-update
```

```
    root@saltminion:~# apt-get --yes -q update
```

```
    Finally, install the salt-minion package
```

```
    root@saltminion:~# apt-get --yes -q install salt-minion
```

```
    Repeat this process for every minion you wish to configure.
```

Finally, the last thing we need to do is to configure the salt minion. The only thing we really need to do

Running Saltstack

Salt is primarily controlled by a command line interface run by the root user on the salt master. Let's look at the anatomy of the following salt command.

```
salt '\*' test.ping
```

The first string 'salt' is the salt executable command. There are multiple commands available with saltstack, including salt and salt-call. The second argument is the *target*. This defines the target minion or minions that the salt command will run on. In this case we use the globbing wildcard '*' to specifiy all minions controlled by the master. Finally, we tell salt what execution module and method to run, in this case, we're going to use the test module and run the ping method. The output will be the result of running a *ping* command on all minions controlled by this master.

When you run a salt command from the command line, every single minion targeted by that master will run the command simultaneously. The cmd module allows you to run specific shell commands directly on the minion by invoking the run method of the module. For example.

```
salt '*' cmd.run 'cat /etc/hosts'
```

Here we run the command 'cat /etc/hosts' on all minions since the target is '*'.

While using the cmd module can be useful for running shell commands, you are far better served by running the execution modules available with salt. These modules express the true power and ability of the Salt application. for example, if we want to get the disk space usage ('df') of a minion, instead of doing a cmd.run df, we can do the following:

```
salt '\*' disk.usage
```

This command will call the disk module's usage method and execute it on all targeted minions.

There are many other types of modules, for example, a network module that has a method to list the network interfaces.

```
salt '\*' network.interfaces
```

Or a module to install a specific package.

```
salt '\*' pkg.install vim
```

Another important salt executable is *salt-call*. Unlike salt, which runs its command on the salt master, the salt-call is called from the target minion. This makes it very useful for debugging purposes. For example running

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

on a specific minion will give detailed debugging information on the command line.

Targeting minions¶

It is somewhat rare that you will run a specific execution module that will target all minions. More likely, you will specify some filtering criteria to only target a specific minion or group of minions. For example,

```
salt 'webserver001' network.interfaces
```

will run the network.interfaces execution only on the minion with the ID 'webserver001'.

We can target multiple minions by ordering them into a like so:

```
salt -L 'webserver001, webserver002, webserver003' network.interfaces
```

This, however, is the least of what we can do. We can for example target minions by the value of a grain. Recall that grains are platform computed values, such as memory, cpu, operating system type, etc.

Let's now target all minions that run Red Hat Enterprise Linux.

```
salt -G 'os:RHEL' network.interfaces
```

We see here that the -G option to salt will expect as the next paramter the value of a specific grain. In this case, the grain is the os attribute and we're looking for all os keys that have the value of 'RHEL'.

We can also target minions using regular expressions. For example, if we name our webservers webserver001 through webserver009 and we'd like to run a command targeting webservers 1-4 and 7, we can do the following:

```
salt -E 'webserver00[1-47]' network.interfaces
```

Additionally, we can target minions using pillar data as well.

```
salt -I 'Region:Asia:Shanghai' network.interfaces
```

Here, we have a pillar called 'Region', which contains a key value pair :. This command will run the network.interfaces method on all minions in which the key is 'Asia' and the value is 'Shanghai'

Note that if the values in the pillar changes, you must refresh the pillar cache by doing either:

```
salt '\*' saltutil.refresh_pillar
```

or

```
salt '\*' saltutil.sync_all
```

We can also match on Internet Protocol addresses.

```
salt -S 192.168.100.1 test.ping
salt -S 10.0.0.0/24 test.ping
```

We use the -S option to indicate to salt that we're passing it IP addresses. Note that we can either use the IP address on its own or add a subnet mask as needed.

Saltstack has the ability to do compound matches, i.e. to match on more than one criteria. Let's see an example.

```
salt -C 'E@webserver00[1-47]' and 'G@os:Ubuntu' network.interfaces
```

This will return all of the network interfaces from webserver's 001 though 004 and 007 only if the operating system being run by the webserver is Ubuntu.

Salt Package Management

The salt package manager (SPM) is a resource to allow users to manage salt formulas. In concept, it is similar to the Red Hat Package Manager (RPM). SPM requires a formula, usually pulled from git, and a FORMULA file. The FORMULA file, which must appear in the package root directory is a YAML file that describes some specific attributes.

Here is an example of a FORMULA file from the apache-formula.

```
name: apache
os: RedHat, Debian, Ubuntu, Suse, FreeBSD
os_family: RedHat, Debian, Suse, FreeBSD
version: 201506
release: 2
summary: Formula for installing Apache
description: Formula for installing Apache
```

The FORMULA file has both required and optional fields. The required fields are:

-Name. The name of the package as it appears in the package filename. For example, the apache-formula should be defined as 'apache'

- OS. This sets the OS grain that is supported by this formula.
- OS_FAMILY. This sets the OS_FAMILY grain that is supported by this formula.
- Version. The version of this package.
- Release. The release number of this package.
- Summary. A one line description of this package.
- Description. A detailed description of this package.

There are a couple of optional fields that can be put into a FORMULA file.

- Top_level_dir. This field is optional, but highly recommended. If it is not specified, the package name will be used. Formula repositories typically do not store .sls files in the root of the repository; instead they are stored in a subdirectory. For instance, an apache-formula repository would contain a directory called apache, which would contain an init.sls, plus a number of other related files. In this instance, the top_level_dir should be set to apache.
- Recommended. A list of optional packages that are recommended to be installed with the package. This list is displayed in an informational message when the package is installed to SPM.

Once the FORMULA file has been created and placed into the root of the package directory, *spm build path_to_saltstack_formulas/formula-name* is called to create a package from the formula. By default the directory that these packages are placed in is /srv/spm.

Finally, a repository is created by calling the create_repo option of spm. For example:

```
spm create_repo /srv/spm
```

Once a package has been built and stored in a package repository, it can be managed. We can update the package by calling the following:

```
spm update_repo /srv/spm
```

We can install the package into our salt system by running the install command.

```
spm install packagename
```

Or, we can remove the package from our salt system by running the remove command.

```
spm remove packagename
```

spm supports three different types of packages.

- Formulas. They are stored in /srv/spm/salt.
- Reactors. They are stored in /srv/spm/reactor
- Conf. Salt conf files normally stored in /etc/salt

Running services under Salt¶

A common use case for Salt is to handle the starting and stopping of system and application services on the minions. Salt has a specific API and command line under the states module to allow this.

Here is an example of using the services module in Salt.

```
openvpn:
  service.running:
    - enable: True
```

In this instance, when the state is run, the openvpn service will be started. The enable flag means to start the service on minion bootup.

Here is another example. In this instance, we are running the *redis* package. This package is enabled on minion boot up and we run the reload rather than the restart command when the redis package changes. Note that we use the *watch* keyword to track the package so that if it changes, the reload command takes effect.

```
redis:
  service.running:
    - enable: True
    - reload: True
    - watch:
      - pkg: redis
```

The Salt reactor¶

Up to now, we have been having one way conversations between the salt master and the minions. We send commands to the minions which execute them and send the results (if any) back to the master. However, it is also possible to minions to send messages to the master. These messages are called *events*.

Events are simply occurrences that happen on the minions. When these events happen, the minion will send a message back to the master that the event occurred. The master can react to these events using the salt *reactor*.

An Event is comprised of two parts. A *tag* that identifies the event and a dictionary of key/value pairs which contains information about the event.

We can then use the data to decide how to handle the event, assuming that we want to handle it at all.,

Reactions are stored in reactor SLS files located by default in /srv/reactor. As with states and pillars, reactor files end in an *.sls* extension. The configuration file for the reactor is stored in /etc/salt/master.d/reactor.conf

Let's take a look at a reactor configuration file.

```
reactor:
  'salt/minion/*/start:
    - /srv/reactor/start.sls
```

The first line is the top level 'reactor:' keyword. Next, we define the event tag we want to watch for. In this case the 'start' event. Finally, for that event, we define a state that we want to run when that event is received by the master.

Now we need to define our start.sls file. All of the reactor files are located by default in /srv/reactor. Let's take a look at our start.sls file.

```
{% if 'dbserver' in data['id'] %}
highstate_run:
  cmd.state.highstate:
    -tgt: 'dbserver\*'
{% endif %}
```

We use a jinja template here to indicate as to whether or not the id of the minion (stored in the *data* dictionary) is a dbserver. If so, we will run the command state.highstate on the minion to force it to re-read its state files.

Testing the reactor¶

Salt comes with a python script called eventlisten.py. We can run this command on the salt-master. like so: *python eventlisten.py*.

On the minion run the following command: *salt-call event.fire_master* '{"*id*": "*dbserver*"}' 'start'
Assuming that the event generation is successful, you will see the following on the minion:

```
local:
    True
```

On the master you will see output that displays the Tag and the Data dictionary.
The salt documentation lists all defined salt events.

Some additional notes on reactor files.
Reactor files only support limited access to salt data. For example, reactor files have no access to salt pillars or grains. Additioally, reactor files do not support requisites, conditionals or other constructs from the Salt State system. This is because rendering reactor files must be quick and simple. Events should be handled in real time, otherwise an event queue can build up.

Saltstack Beacons¶

Salt allows us to monitor non-salt processes using *Beacons*. Beacons allow us to hook into different system processes and monitor them. When this process is triggered we can send an event via the event bus to the salt master for processing.

Beacons are configured either through an entry in the minions file under the beacons: key or by putting a file into /etc/salt/minion.d/beacons.conf

Here is an example of a beacon.

Note in the example above we're watching two processes, *inotify* and *load*. Inotify is being watched on a five second interval, load on a ten second interval. If the criteria for the beacon is met, an event is triggered and sent to the salt master.

```
beacons:
  inotify:
    /etc/important_file: {}
     /opt: {}
     interval: 5
     disable_during_state_run: True
   load:
     1m:
     - 0.0
     - 2.0
     5m:
       - 0.0
       - 1.5
     15m:
       - 0.1
       - 1.0
interval: 10
```

Third party integration with SaltStack and Nagios¶

A common tool for integration with Saltstack is Nagios. Nagios is a web based tool for monitoring the state and health of servers on an enterprise network. Here we will see how we can use Nagios and Saltstack together.

Saltstack supports Nagios out of the box with its Nagios execution module. Simply install Nagios on the minion. Here is an example of a command:

```
salt '*' --out=json --static cmd.run_all '/usr/lib/nagios/plugins/check_procs -w 150 -c 200'
```

Note that this command will run the specified Nagios plugin 'check_procs' on all minions and return the output in JSON format.

Additionally, you could also run this command via the Nagios module's *run* command. Like so:

```
salt '*' --out=json --static nagios.run '/usr/lib/nagios/plugins/check_procs -w 150 -c 200'
```