Introduction to Saltstack

Our Agenda

- Introduction to DevOps
- What is Saltstack and what does it do?
- Saltstack components
- Installing Saltstack
- Configuring the Salt Master

Introduction to Saltstack

Our Agenda

- Configuring a Salt Minion
- The *salt* and *salt-call* commands
- Salt PKI
- Salt States

Introduction to Saltstack

Our Agenda

- Introduction to YAML
- The structure of a salt state file.
- Salt state declarations
- Salt grains
- Salt pillars

Introduction to Saltstack

Our Agenda

- Introduction to the Jinja2 templating system.
- Salt package management (SPM)
- Salt reactors
- Salt beacons
- Example of third party integration with Nagios.

Introduction to Saltstack

What is DevOps?

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

Introduction to Saltstack

Automate everything

- Automate code testing
- Automate workflows
- Automate infrastructure

Introduction to Saltstack

Constant iteration of development

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

Introduction to Saltstack

Synchronize and Monitor environments

- Identical environments for development, test and production
- Adopt iterative processes to monitor and measure code and operations daily.
- Improve ability to quickly respond to customer concerns and market conditions.

Introduction to Saltstack

Use source code control

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

Introduction to Saltstack

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.

Introduction to Saltstack

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.

Introduction to Saltstack

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

Introduction to Saltstack

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.

Introduction to Saltstack

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.

Introduction to Saltstack

Saltstack components¶

- Salt Master
- Salt Minions
- Execution Modules
- States
- Grains
- Pillars

Introduction to Saltstack

Saltstack components¶

- Top File
- Runners
- Returners
- Reactors
- Salt Cloud
- Salt SSH

Introduction to Saltstack

Salt Master

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

Introduction to Saltstack

Salt Minion

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

Introduction to Saltstack

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.

Introduction to Saltstack

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.

Introduction to Saltstack

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

Introduction to Saltstack

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

Introduction to Saltstack

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.

Introduction to Saltstack

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.

.

Introduction to Saltstack

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.

Introduction to Saltstack

Architectural overview of remote execution

- 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.

Introduction to Saltstack

Architectural overview of remote execution

- 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.

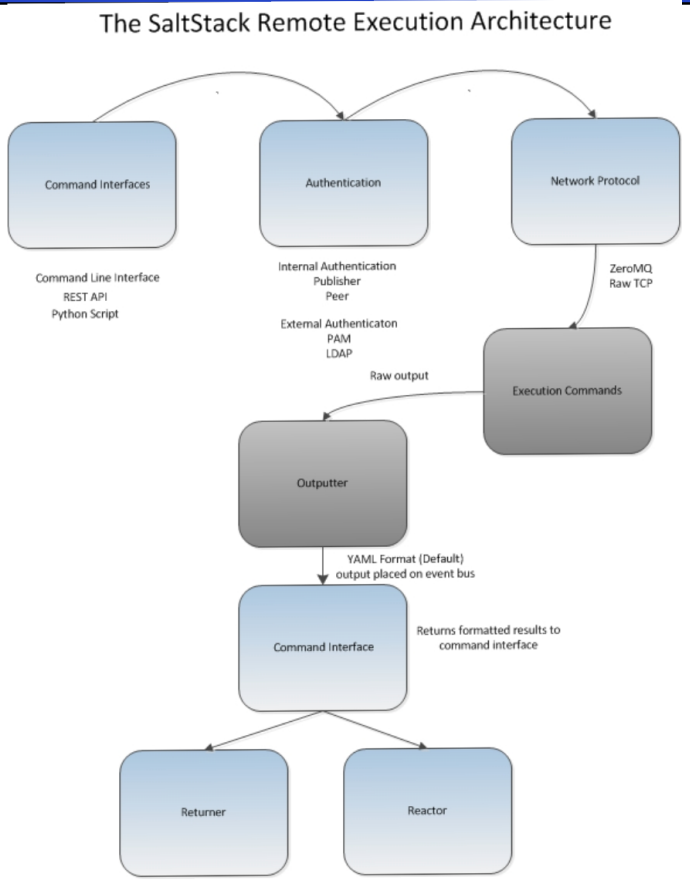
Introduction to Saltstack

Architectural overview of remote execution

- 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.

Introduction to Saltstack



Introduction to Saltstack

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.

Introduction to Saltstack

- 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.

Introduction to Saltstack

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.

Introduction to Saltstack

Public Key Infrastructure for secure authentication.

- 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.

Introduction to Saltstack

Salt Installation and Configuration

- Getting Salt
- Source Installation
- Packaged Installation
- Salt Bootstrap
- Salt Master Network Ports
- Minion Firewall

Introduction to Saltstack

Salt Installation and Configuration

- Basic Minion Configuration
- Salt Security
- Installing salt on Windows minions.
- Lab 1. Installing Salt

Introduction to Saltstack

Saltstack basic commands

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

- /usr/bin/salt
- /usr/bin/salt-run
- /usr/bin/salt-call
- /usr/bin/salt-key

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

Introduction to Saltstack

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.
-

Introduction to Saltstack

The salt command

- 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.

Introduction to Saltstack

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-call 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.

Introduction to Saltstack

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
-
- 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.
-
- Accessing data that is only available on the master.
- salt-run manage.up
-

Introduction to Saltstack

Salt-run. Executing directly on the master.

- 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.

Introduction to Saltstack

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.

Introduction to Saltstack

Salt-key, managing the Salt PKI infrastructure.

- Accepted. The master has accepted the public key of the minion.
- Unaccepted. The master has not yet accepted the public key of the minion.
- 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.

Introduction to Saltstack

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_rejected	Public key of every unaccepted minion

Introduction to Saltstack

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

Introduction to Saltstack

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.

Introduction to Saltstack

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.
-
-

Introduction to Saltstack

Accessing Saltstack documentation.

- 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.

Introduction to Saltstack

Accessing Saltstack documentation.

- 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.

Introduction to Saltstack

Accessing Saltstack documentation.

- 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 <module name>

  salt salt.master sys.argspec <function name>

- 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.

Introduction to Saltstack

Accessing Saltstack documentation.

- 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.

Introduction to Saltstack

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'.

Introduction to Saltstack

Targeting minions

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

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

Introduction to Saltstack

Targeting minions

- 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'.

Introduction to Saltstack

Targeting minions

- 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

Introduction to Saltstack

Targeting minions

- 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 <Regional area>:<City>.

This command will run the network.interfaces method on all minions in which the key is 'Asia' and the value is 'Shanghai'

Introduction to Saltstack

Targeting minions

- 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*

Introduction to Saltstack

Targeting minions

- 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.

Introduction to Saltstack

Targeting minions

- 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

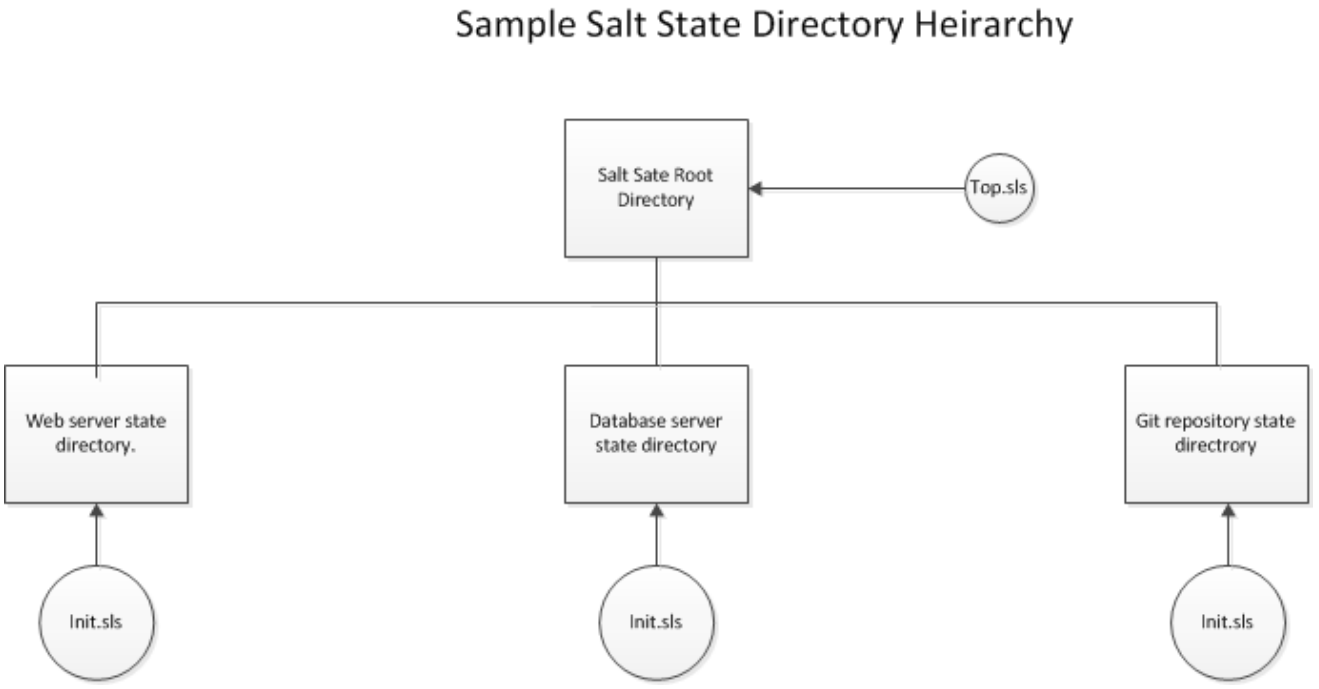
This will return all of the network interfaces from webserver's 001 though 004 and 007 only if the operating system being run by the webserver is Ubuntu.

Introduction to Saltstack

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.
- The following slide is an example of a state tree hierarchy.

Introduction to Saltstack



Introduction to Saltstack

Salt States

- 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 <state module> 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 Saltstack

- 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:

Introduction to Saltstack

- Introduction to YAML

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

        Fuel: "-- 'Petrol' -- 'Diesel' -- 'Electric'"
        Model: C
        Weight: 2250
```

Introduction to Saltstack

- Introduction to YAML

- 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.

Introduction to Saltstack

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.

Introduction to Saltstack

State 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.

Introduction to Saltstack

State 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:

```
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.

Introduction to Saltstack

State files

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

```
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.

Introduction to Saltstack

State files

- 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*.

Introduction to Saltstack

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. The next slide shows four of them.

-

-

-

Introduction to Saltstack

State ordering

- require
- watch
- prereq
- onfail

Introduction to Saltstack

State ordering

- 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.
- Following 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

Introduction to Saltstack

State ordering

```
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
```

Introduction to Saltstack

State ordering

- 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.

Introduction to Saltstack

State ordering

```
  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
```

Introduction to Saltstack

State ordering

```
/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
```

Introduction to Saltstack

State ordering

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

```
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.
```

Introduction to Saltstack

State ordering

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

```
# 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
```

Introduction to Saltstack

State ordering

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

```
# This section controls the nginx defaults file. We'd like to customize this for each web server we set
up.
# To do this, we use a jinja template. We put the .jinja on the end of the filename so we know that it's
a # jinja 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
```

Introduction to Saltstack

State ordering

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

```
# The sites-enabled default file is set up as a symbolic link. Note that we require the source file in
the
# 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
```

Introduction to Saltstack

State ordering.

- 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.

Introduction to Saltstack

State ordering.

```
# 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.

For example, we can do the following:

<table width='750px', align='left'>

graceful-down: cmd.run:

- name: service apache graceful
- prereq:
  - file: site-code

site-code: file.recurse:

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

Introduction to Saltstack

State ordering.

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

```
    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
```

Introduction to Saltstack

State ordering.

- 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

Introduction to Saltstack

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*

Introduction to Saltstack

Salt Grains

- 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*

Introduction to Saltstack

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
```

Introduction to Saltstack

Salt Grains

- And here is an example of storing customized grains in the */etc/salt/grain* file.
- Because the grain is already stored in the grain configuration file, we don't need the top level 'grain' key.

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

Introduction to Saltstack

Salt Grains

- Customized grains can override the core grain objects. When Salt loads grains it has an order of precedence. Custom grains in /etc/salt/minion.
- Custom grain modules in _grains directory, synced to minions.
- Custom grains in /etc/salt/grains.
- Custom grains in /etc/salt/minion.
- 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

Introduction to Saltstack

Salt Grains

- 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

Introduction to Saltstack

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.

Introduction to Saltstack

Salt Pillars

- 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.

Introduction to Saltstack

Salt Pillars

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

  salt '' pillar.items*

Introduction to Saltstack

Salt Pillars

- 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 as shown in the next slide

Introduction to Saltstack

Salt Pillars

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

Introduction to Saltstack

Salt Pillars

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

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

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

Introduction to Saltstack

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.

Introduction to Saltstack

Salt Grains vs Pillars

Differences	Grains	Pillars
Info which	...minion knows about itself	...minion asks Master about
Distributed	Yes	No
Centralized	No	Yes
Computed	Yes (mostly	No

Introduction to Saltstack

Salt Grains vs Pillars

Differences	Grains	Pillars
Assigned manually	No (or kept to a minimum)	Yes
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

Introduction to Saltstack

- Salt Grains/Pillars Use Cases

- Default use case
- In our default case the entire system is 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

Introduction to Saltstack

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:

Introduction to Saltstack

- 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 of the information about Minion should either be assigned to it through Pillars (using Minion id as a key) or evaluated in non-static grains.

Introduction to Saltstack

- 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.

Introduction to Saltstack

- 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.

Introduction to Saltstack

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..

Introduction to Saltstack

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

Introduction to Saltstack

The Jinja2 templating system.

- Jinja2 adds limited programming capabilities to your state, pillar and custom grain files.

Introduction to Saltstack

The Jinja2 templating system.

- 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.
- In the next slide we define a pillar file called /srv/pillar/pkg/init.sls

Introduction to Saltstack

The Jinja2 templating system.

```
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 %}
```

Introduction to Saltstack

The Jinja2 templating system.

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

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

Introduction to Saltstack

The Jinja2 templating system.

- 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

Introduction to Saltstack

The Jinja2 templating system.

- 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

Introduction to Saltstack

The Jinja2 templating system.

- 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.

Introduction to Saltstack

The Jinja2 templating 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
```

Introduction to Saltstack

The Jinja2 templating system.

- .In the previous slide 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.

Introduction to Saltstack

The Jinja2 templating system.

- 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 %}
```

Introduction to Saltstack

The Jinja2 templating system.

- In the previous example, 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.

Introduction to Saltstack

The Jinja2 templating system.

- 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.

Introduction to Saltstack

The Jinja2 templating system.

```
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

Introduction to Saltstack

The Jinja2 templating system.

- 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
```

Introduction to Saltstack

The Jinja2 templating system.

- 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 %}
```

Introduction to Saltstack

The Jinja2 templating system.

- 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.

Introduction to Saltstack

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.

Introduction to Saltstack

The Jinja2 templating system.

- 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
```

Introduction to Saltstack

Salt Package Management

- 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.

Introduction to Saltstack

Salt Package Management

- Release. The release number of this package.
- Summary. A one line description of this package.
- Description. A detailed description of this package.

Introduction to Saltstack

Salt Package Management

- 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

Introduction to Saltstack

Salt Package Management

- 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*

Introduction to Saltstack

Salt Package Management

- 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*

Introduction to Saltstack

Salt Package Management

- 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

Introduction to Saltstack

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.
- Following is an example of using the services module in Salt.

Introduction to Saltstack

The Jinja2 templating system.

```
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.

Introduction to Saltstack

Salt reactors

- 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.

Introduction to Saltstack

Salt reactors

- files end in an .sls extension. The configuration file for the reactor is stored in /etc/salt/master.d/reactor.conf
- 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

Introduction to Saltstack

The Jinja2 templating system.

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

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

Introduction to Saltstack

Salt reactors

- 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.

Introduction to Saltstack

The Jinja2 templating system.

- 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 %}
```

Introduction to Saltstack

Salt reactors

- 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.

Introduction to Saltstack

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
  ```
  ```

Introduction to Saltstack

Testing the reactor

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