Salt - Conductor Pillar Model

This page will be the starting page for all documentation related to the Pillar configuration model, usage and design of our Saltstack Conductor framework implementation.

Random information here will be organized at a later time.

The Pillar Model

Concepts and structure

The pillar model described here is a design and implementation of Saltstack Pillar. This is not the only way by any means to implement pillar configuration when Saltstack in any organization. It is however a fairly thought out way of architecting pillar data (and state data) in a scalable, safe way throughout an organization. This model is a central piece of a Framework used to implement Saltstack when hosting in AWS (or others in the future). Consider this a Product Delivery Automation Framework. A product can be anything from a configuration, application, service, or system up to a large logical set of systems. A product includes 3rd party technologies as well.

There are three design components to the Framework

- Pillar model
- State Design
- Conductor (automation tool)

The Pillar is consumed by both state and the conductor. Conductor use this pillar to determine what the cloud configuration needs to look like, or for some other function when given a command to do something. Pillar is also used in state files to assure we are re-using common configuration and providing dynamic variables to states.

Targeting

Targeting is big concept in any Saltstack implementation. Targeting is how you tell salt what minions can get what pillar and state data. So you can put a state file in your state source repo/file system and without any targeting, any minion registered to a salt master with access to that state will be able to have that state applied.

Think off targeting like filters. Targeting can be very simple or very complex, see compound matching. Therefore targeting is very flexible. You also have to consider targeting when designing and planning a Salt implementation so as to keep things scalable.

Command

Targeting is done in two ways. Either at the salt command line such as this basic command

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

This command run on the salt master says, "find all minions that are registered with this salt master and run the built-in test module and call function ping.

The '*' is a glob. Targeting can be of many types, list, glob, regex, compound etc...

Top File

Targeting can also be done in the Pillar top.sls and State top.sls. The top.sls file is a special file in salt and used as a top of the tree minion targeting mechanism. See top.sls

> Top file is another huge concept in salt. It has lots of facets, but in short it's a targeting mechanism.

In this Conductor Framework, we do not make complete use of top.sls file in the Pillar source for targeting specific minions. We will use top.sls in the State source for that, but in Pillar, the top.sls is not heavily used currently.

There is nothing to stop us from making more extensive use of top.sls in pillar, and would not require any design or code changes. Because we have the conductor runner as the main user interface for provisioning, configuring and managing minions during the create and destroy phases, the targeting is mostly baked into the runner.

In contrast, in some companies Saltstack used the pillar top.sls to segment environments, for example this tops.sls:

```
'dev':
  '*':
    - do this
'qa':
  '*':
    - do that
'prod':
  '*':
    - do it carefully
```

Can be place in a environment shared pillar source repo. It relies on the top.sls to segment the environment pillars. The problem with this is when you have a huge pillar top.sls in a common shared place, it can become the single point of failure for many environments. This could be very bad.

This single file would get very complex and confusing to parse when used by many folks supporting different instances, platform and products as well each with their own specific targeting requirements. It would be very tough to scale this across an enterprise with many teams, admins and such.

Salt has the ability, and is very common, to use branches in a source repo to segment pillar. We do this in the Conductor framework. In which case, we do not need environment filters in top.sls since we have one in each branch. You can use multiple repos together to make the complete pillar tree for a given environment as well. So many ways to do this!

A simple way to look at top.sls is that the more granular the targeting gets in tops.sls, the more difficult it will be to maintain conflict free scalable solutions in Salt.

Our initial Pillar top.sls will mostly to make provisioning templates available based on product groups and such.

Top file is a deep concept in salt. The conductor framework implementation details of pillar and state tops file will be touched on in one of the knowledge/training sessions.

See Example of the framework pillar top file here.

One very important reason that we do not specify extensive and deeply complex targeting in Pillar top file in the Conductor framework, is due to the nature of what Conductor is.

It is a salt runner that is used to provision, configure and manage cloud vm's in ANY environment for ANY product group or ANY product across and entire organization. ***Therefore the Conductor needs access to an unfiltered (lightly targeted) full Pillar tree.***

Product group is a key concept in the Conductor framework. It allows us to target pillar items and state based on a logical group of products or implementation of products. Since we segment the environment deltas in separate pillar source branches, we really do not need to define much targeting beyond product group related pillar trees.

Model overview and basic layout

There is a certain thought process behind the pillar model used by the framework. In trying to keep it simple, but scalable, the concept of namespace comes into play.

In salt, whenever the Pillar tree or State tree is realized, the underlying salt code compiles the entire yaml structure into a python dictionary. This holds true for any extensions such as custom execution or state module, runner modules as in the Conductor, or any other module of the salt system.

In python a Dictionary is a set of key value pairs. A value can be a nested dictionary, list, string or any other data type. Dictionaries and python is not in scope in this document, but one thing to know is that python dictionaries can not have duplicate keys. Meaning only of each top level key is allowed in a python dictionary.

Since Salt will compile the entire Pillar or State tree into a python dictionary, there can NOT be any duplicate keys (yaml Identifiers) at the top of the tree or any duplicate keys within any subsequent nested dictionary values. The actual file that holds the configuration is not the issue. When salt compiles it's irrelevant what file the pillar was found it. It's all about the yaml information and 'ID' of the configuration.

The file that holds the configuration is more important in the State tree because salt will use the file directory structure in that state source to locate that state it needs.

Details on Salt States will be in another document, but in short, if Salt needs to resolve a state called foobar.test.one, it will look starting at the root of the state source repo for a file in either

salt://foobar/test/one/init.sls

or an explicit file

salt://foobar/test/one.sls

Either is valid, but not both.

init.sls when used, does not need to specified in the salt command.

'salt://' is the moniker for state tree root path

'pillar://' is the moniker to pillar tree root path

On the contrary to how salt resolves state files, salt resolves pillar strictly by the yaml identifiers regardless of the file. For example, if we put the following yaml in a pillar file at the root of the pillar source repo called global.sls.

pillar://global.sls

```
global:
  domain_suffix: ".foobar.com"
  salt-minion:
    version: 2017.7.5
```

Salt will realize the value for salt-minion version as follows regardless of what file we put this in throughout the pillar source repositories.

global:salt-minion:version

If we put the same pillar in a file somewhere else in the pillar source repository, or in another repository entirely that the salt master is configured to refer to, such as

pillar://special/hidden/config/stuff.sls

Salt will still find it by using ***global:salt-minion:version.***

With this behavior, you may start to see why its very important to plan and develop a pillar strategy and put controls such as the Conductor framework around it so as to not have frequent conflicts.

Salt will throw exceptions and fail to compile the pillar or state.

In order to maintain scalable means of defining configuration across an enterprise organization, we need a way to assure no conflicts. Namespace is the most common approach and the approach taken in this Salt framework.

When architecting a CM solution for Saltstack (and other like technologies), the files holding the configuration (pillar or state) should reside in a file tree hierarchy. In the case of pillar, this is simply for organization as outline in above salt pillar resolution.

Additionally in Salt, since targeting is a key concept, using a file system type hierarchy is also good way to keep things from causing conflicts in the event you need to have the same yaml identifiers in multiple files. Although you can have multiple same top level yaml pillar configuration items in the same pillar source and use Pillar top targeting (see above), this Framework does not follow that concept in general.

Pillar Source Layout

In the Salt Conductor framework pillar model, there are only a couple top level directories to know about. However, what teams/groups add below these locations is something that needs to follow the model and should have some automated control to help keep things from conflicting.

Since we segment environments specific pillar configuration via branches in the source repository or repositories, we will just consider one branch in this example.

This is a basic minimal layout of the pillar tree for a salt environment. I.E. a branch in the pillar repo/s. This is pretty much what we have currently in our salt-pillar repository, but since we are still in dev/test mode, there may be differences. Below is how it will look once we start migrating teams/groups into Salt.

```
aws.sls
azure.sls
config
   common.sls
   corp
      init.sls
   groupX
      init.sls
      properties.sls
   groupY
      init.sls
      properties.sls
      systems.sls
   teamA
      init.sls
      systems.sls
   teamB
       init.sls
       systems.sls
global.sls
provisioning
   templates
       groupX
          alt_roles.sls
          roles.sls
       groupY
          alt_roles.sls
          roles.sls
       teamA
          roles.sls
       teamB
          roles.sls
       tests
           test_1.sls
           test_2.sls
           test_3.sls
rackspace.sls
top.sls
```

There could be a lot more pillar files in this tree once migration starts occurring. But the example above shows the two most tops level directories along with a couple global files. The cloud provider specifics for the environment, along with some Organization global configuration such as artifact repository uri, supported versions on applications and default values for common shared products.

As shown in the example, there are two directories at the pillar root

/config

/provisioning

These represent two separate configuration content.

/Config is used for product group and product specific configuration. When salt states are created and need pillar, this is the location is should be resolving it from. The roles that are available in the product group would be defined here along with some details such as whether the role needs a load balancer, java, public vs private security group as well as orchestration hooks. In the above example, teamA, teamB, groupX and groupY all are "Product Groups" in Conductor terms.

Under the /config/productgroup/ directory is where you can also specify systems.sls file to represent group specific 'system' to be provisioned. Refer to conductor provisioning types. Any other files to hold pillar configuration can be created under these group specific locations. It all depends on the product group and product implementation needs as well as how the group wants to separate the pillar data. It doesn't make a difference to the Conductor automation. The yaml id's in each file is what is important to maintain uniqueness or filtered via targeting in the top.sls pillar file as described above.

/Provisioning is used as a logical container to put group specific configuration required for provisioning. The templates directory is created with sub directory for each product group that needs to have provisioning pillar available. It is not a hard requirement, but each group should have a roles.sls that defines the roles that should be available to be provisioned in that environment (pilllar branches). One file per group for all the roles should be fine. However, as shown in the example (see groupX and groupY), additional files with pillar role configuration could be added. This is useful when groups want to try out and provision different configurations of the same product/role. Instead of defining a new role, which is not something we want to do, that group can simply toggle which role configuration template is needs for the environment. This would be done by filtering the file in pillar Top file. In the case above for groupX, the roles.sls and alt_roles.sls files each have the same top level yaml ID (which MUST be the role name), but since targeting can be used in the top file, the group can simply edit top.sls and commit, run the provisioning test, then change top.sls back to the default roles.sls.

This is just one example of an endless way to structure templates for provisioning.

> Environments are cheap in Salt. It's a matter of slicing a new branch. The conductor will simply take the pillarenv parameter value and get that branch when it needs to resolve pillar. So groupX could even have a dev1 and dev2 environment in pillar to test the different configurations. Environments do NOT have to be in different cloud account or regions. It's a basic concept that could simply mean "use a different set of config, and name the instances accordingly".

The Pillar top file for the above example would look like this for the 'dev' environment for aws cloud support (note azure pillar is not in top.sls, I.E. it's not targeted to anything since it's not supported yet)

```
dev:
  '*':
    - global
    - aws
    - config.common
    - config.teamA
    - config.teamA.systems
    - config.teamB
    - config.teamB.systems
    - config.groupX
    - config.groupX.properties
    - config.groupY
    - config.groupY
    - config.groupY.systems
    - config.groupY.properties
    - provisioning.templates.teamA.roles
    - provisioning.templates.teamB.roles
    - provisioning.templates.groupX.alt_roles
    - provisioning.templates.groupY.roles
    - provisioning.tests.test_2
```

> In the above Pillar Top file, groupX would have roles defined in alt_roles.sls available, and only test_2 is being targeted.
>
> I like to think of the Top files used like this as Publishing pillar to minions rather than targeting.

Just to contrast this Top file example, here is another way to product the same pillar configuration to be available to the same minions using another level of targeting with grains.

```
dev:
  '*':
    - global
    - aws
    - config.common
    - provisioning.tests.test_2
    'G@product.group:teamA':
      - config.teamA
      - config.teamA.systems
      - provisioning.templates.teamA.roles
    'G@product.group:teamB':
      - config.teamB
      - config.teamB.systems
      - provisioning.templates.teamB.roles
    'G@product.group:groupX':
      - config.groupX
      - config.groupX.properties
      - provisioning.templates.groupX.alt_roles
    'G@product.group:groupY':
      - config.groupY
      - config.groupY.systems
      - provisioning.templates.groupY.roles
    'G@product.group:groupY and G@role:foobar':
      - config.groupY.properties
```

Important note about the above pillar tree top file configuration

This example above shows how you can target using grains and compound expressions in Top file. However in the case of Conductor needing to provision instances if any role within any product group, this configuration above would NOT work. Since the Conductor is a runner and thus runs on the salt master, the salt master is a minion as well. So if the salt master does not have the grains to match it would not be able to find the correct pillar items it needs to properly generate cloud configuration when provisioning instance, clusters or systems.

For pillar configuration used by salt states, or any other Conductor or salt command after the minions are provisioned, this method is fine to use in Pillar top file.

As noted above the State configuration tree top file targeting is not so much as concern, because states get applied after the minions are provisioned and bootstrapped. It's just the Conductor is using Pillar much earlier in the process to establish the correct cloud configuration files needed to hand off to salt-cloud and thus the cloud providers.

Here are some links to actual Pillar files being used in our Salt framework develop efforts just for reference. The section Configuration Specifics goes into more detail about how to configure role defaults and provisioning detailed configuration.

Example pillar product group provisioning template

Example pillar product group role default configuration

Example pillar product group system configuration

Example use cases

Use case 1

Product group X has a product called foobar and product group Y also has a product called foobar and they both need to deploy to all environments. Consider 'dev' environment for this use case, but it wouldn't matter. Foobar is not a 3rd party application, hence it is not common or shared. Each product group just happened to develop an application that was named foobar. Therefore there is pillar configuration defining and describing provisioning details of foobar for both groupX and groupY within the same organization. Also assume there are no other products for either of these product groups.

Following the model, the minimum source pillar configuration from the root (pillar://) for the two would look like this

```
 config
    groupX
       init.sls   <----- contains role groupX.foobar product details
    groupY
       init.sls   <----- contains role groupY.foobar product details
 provisioning
    templates
        groupX
           roles.sls   <----- contains role groupX.foobar provisioning
specifics (quantity, instance size, volumes etc...)
        groupY
           roles.sls   <----- contains role groupY.foobar provisioning
specifics (quantity, instance size, volumes etc...)
 top.sls
```

For the sake of this example, consider that each product groups implementation is different. For groupsX, foobar product implementation includes enabling some conductor built-in optional hooks (not all), species java 8 install dependency, requires all instances with groupsX.foobar role to be added to a load balancer (conductor will create if not found), and requires jq to be installed as a prerequisite dependency. GroupY implementation of its foobar app is much simpler and doesn't require any of the extra stuff that groupX does.

Role/Product specific configuration

Let's have a look at each product groups ***role configuration file (init.sls)***

pillar://config/groupX/init.sls

```
groupX.role:                                <---- the product group
unique identifier. Most important to be unique throughout the pillar
environment tree.
                                                  For this reason, the
conductor framework requires the PRODUCTGROUP.role syntax.
  product-path: /com/eliza/                 <---- can be used as the
starting path in artifactory to locate artifacts

  # product group scope hooks
  # role based hooks are under the role config
  # common (env) scope hooks are in pillar://config/common.sls

  hooks:                                    <---- Optional product group
hooks enabled means all 'create' conductor actions will run these during
```

```
the process.
    pre-provision-orchestration:              The state config here
is a pointer to the salt orchestration state
salt://orch/groupX/pre-startup.sls.
      state: ['orch.groupX.pre-provision']        Conductor realizes
these and sets pre-provision grains on the minion in the generated
salt-cloud config.
      enable: True                                Once bootstrapped,
Conductor looks for any post provision orchestration hook grains, and
executes the state.
    pre-startup-orchestration:
      state: ['orch.groupX.pre-startup']
      enable: True
    post-startup-orchestration:
      state: ['orch.groupX.post-startup']
      enable: True
    pre-destroy-orchestration:
      state: ['orch.groupX.pre-destroy']
      enable: True
    post-destroy-orchestration:
      state: ['orch.groupX.post-destroy']
      enable: True

  all:                                      <---- A list of all products
(aka roles) being configured and supported by this product group
    - foobar

  foobar:                                   <---- The role specific
configuration defaults
    security-group: private
    hooks:                                  <---- Optional Role based
hooks (see above regarding how these are used)
      pre-startup-orchestration:
        state: ['orch.groupX.pre.foobar']
        enable: True
      post-startup-orchestration:
        state: ['orch.groupX.post.foobar']
        enable: True
      post-destroy-orchestration:
        state: ['orch.groupX.post.destroy.foobar']
        enable: True
    elb:                                    <---- Specifies this role
needs to be put behind an elb
      name: groupX-foobar-dev-webapps

    state: ['groupX.foobar', 'common.jq']   <---- The salt states to
apply to any instance/minion with this role.
                                                  This is a list of any
number of states. This case we also specify salt://common/jq/init.sls
state
```

```
    java:                                   <---- Java is a special case
dependency. This says install java version 8 before running the states
in the list.
      version: 1.8.0_121                          Other product
dependencies could be defined in the salt://groupX/foobar/init.sls
```

```
state, or there is another
      package: jdk-8u121-linux-x64.rpm            configuration not
shown here that allows dynamic dependencies.
```

pillar://config/groupY/init.sls

```
groupY.role:                                <---- the product group
unique identifier. Most important to be unique throughout the pillar
environment tree.
                                                  For this reason, the
conductor framework requires the PRODUCTGROUP.role syntax.
  product-path: /com/eliza/                 <---- can be used as the
starting path in artifactory to locate artifacts

  all:                                      <---- A list of all products
(aka roles) being configured and supported by this product group
    - foobar

  foobar:                                   <---- The role specific
configuration defaults
    security-group: private
    state: ['groupY.foobar']                <---- The salt states to
apply to any instance/minion with this role.
                                                  This is a list of any
number of states.
```

One thing to keep in mind is that there are common configurations in pillar://config/common.sls. So in the case of groupY, they may still need java as a prerequisite install for their foobar application, but instead of overriding the java configuration, groupY is simply relying on the common default java version for this environment. ***Common configurations are environment wide, but are separate per environment in pillar source.***

Provisioning specific configuration

Let's have a look at the provisioning templates for each product group and see the details. GroupY will be ONLY the required provisioning template configuration so as to show in groupX quite a few optional overrides and additional configuration. Also note that in each case foobar is NOT a cluster role. Cluster roles are roles that span multiple instance. See Provisioning Cluster Roles. If foobar in these examples was a cluster, there would be a required configuration under the role ID

pillar://provisioning/templates/groupX/roles.sls

```
groupX.foobar:                  <---- the product group role ID. Most
important to be unique throughout the pillar environment tree.
                                      For this reason, the conductor
framework requires the PRODUCTGROUP.PRODUCT (product is the role in this
config).
  force-delay-state: True       <---- delay the saltstack startup_states
(refer to "delaying startup_states in the Configuration specifics
section).
```

```
  spot_config:                  <---- optionally specify to use spot
instances
    spot_price: 0.10
    tag:
      owner: group X engineering
      contact: somebody
      purpose: pillar walkthrough
  block-volume:                 <---- specify block volume additions and
overrides
    - device-name: /dev/sda1
      volume-size: 30
      volume-type: gp2
      tag:
        owner: group X engineering
        description: used for something
        Contact: someone
    - device-name: /dev/sdb
      volume-size: 15
      volume-type: gp2
      tag:
        owner: groupX
        purpose: use for something else
  root-volume-tags:             <---- not really needed since the root
volume in defined in the block-volume, See below for root and block
volume tagging
    owner: groupX
    purpose: testing
  persist-volumes: False        <---- don't keep volumes on instance
terminate
  role: groupX.foobar           <---- REQUIRED this is the role that
will be used in the role grain on the minion/s
  additional-grains:            <---- add any number of additional
grains or different types
    product.group: groupX       <---- THIS IS NOW BAKED INTO CONDUCTOR,
thus its not needed. But you can specify it anyway.
    test-grain: foobar app
    another:                    <---- list grain
      - one
      - two
      - three
    yetanother:                 <---- dictionary grain
      bob: principal
      sue: quality
  basename: groupX-foobarXX.REGION.ENV <---- REQUIRED set the base name
pattern for defining the hostname/name that all instances of this role
type will get.
                                             XX denotes index number in
the event the nodes: is more than one or if there are other similar in
the env.
                                             REGION and ENV will get
```

```
populated by conductor. The domain suffix will be added. This is defined
in config/common
  nodes: 1                      <---- default quantity each time this
role is provisioned. Can be overwrote on conductor command line
  size: t2.small                <---- instance size for this role in
this environment
  tags:                         <---- instance tags
```

```
    owner: group X engineering
    purpose: pillar consumption testing
    Contact: the boss
```

pillar://provisioning/templates/groupY/roles.sls

```
groupY.foobar:                  <---- the product group role ID. Most
important to be unique throughout the pillar environment tree.
                                      For this reason, the conductor
framework requires the PRODUCTGROUP.PRODUCT (product is the role in this
config).
  force-delay-state: True       <---- delay the saltstack startup_states
(refer to "delaying startup_states in the Configuration specifics
section).
  root-volume-tags:             <---- not 100% needed since the root
volumes will always be tagged with at least Name. This shows some custom
as well.
    owner: groupY
    purpose: testing
  role: groupY.foobar           <---- REQUIRED this is the role that
will be used in the role grain on the minion/s
  additional-grains:            <---- THIS IS NOW BAKED INTO CONDUCTOR,
thus its not needed. But you can specify it anyway.
    product.group: groupX
  basename: groupY-foobarXX.REGION.ENV <---- REQUIRED set the base name
pattern for defining the hostname/name that all instances of this role
type will get.
                                             XX denotes index number in
the event the nodes: is more than one or if there are other similar in
the env.
                                             REGION and ENV will get
populated by conductor. The domain suffix will be added. This is defined
in config/common
  nodes: 1                      <---- default quantity each time this
role is provisioned. Can be overwrote on conductor command line
  size: t2.small                <---- instance size for this role in
this environment
```

The difference is these two provisioning templates is pretty extensive. In short, groupY is the bare minimum needed and doesn't specify any additional or override configuration. GroupX is more detailed and complete. GroupY.foobar role instances will get the default 8gb root drive, one tag Name, and the instance will get only one tag, Name.

Use case 2

Product groupX has systems defined in its pillar configuration. Systems are used to provision logical sets of instances or any role type. See Systems provisioning types.

Below shows the only configuration needed for any product group to define systems. There is nothing else needed. Additionally each system entry has only 2 configurable items.

The 'system' is implemented as a simple grouping of roles and specified quantity. All role configuration is as in the above example, nothing different. The standard file pillar://config/***productgroup***/systems.sls should be used. But as with all pillar data, as log as the yaml ID is unique in the compiled pillar tree, it will work.

The idea of a system is to provide ability to create and destroy entire platforms, or any logical grouping of product instance such as a suite or complete working system.

pillar://config/groupX/systems.sls

```
system.groupX:            <---- top level yaml ID. Must be unique, so we
use product group

  groupX.system.small:    <---- a system with 4 roles being deployed.
Mixed cluster and stand alone applications/services.
    appA:
      members: default    <---- members is use when the role is a
cluster (one primary and n number or secondary internal.role, see
cluster provisioning)
    svcA:
      count: 2            <---- count is used when the role is NOT a
cluster
    appB:
      members: 2
    webappA:
      count: default

  groupX.system.big:
    appA:
      members: 5
    svcA:
      count: default
    appB:
      members: 5
    webappA:
      count: 3

  groupX.system.baseline:
    appA:
      members: default
    svcA:
      count: default
    appB:
      members: default
    webappA:
      count: default
```

The systems shown above in the systems.sls pillar file define 3 separate systems of varying instance quantity.

A system is provisioned using the Create action of the Conductor runner. The command for creating the 'baseline' system in the 'dev' environment would be:

example conductor runner command to create a system for groupX product group

```
salt-run conduct.group create group=groupX system=baseline pillarenv=dev
region=us-east-1a

for the big system

salt-run conduct.group create group=groupX system=big pillarenv=dev
region=us-east-1a
```

Configuration Specifics

This next section describes some Framework specific pillar configuration format, syntax and options

Product Configuration

This section shows the configuration options for products (***aka roles***) specific to the product group they are being deployed as. Common (aka 3rd party products) are shared and consumed by any product group. Common product salt states are consumable with with variable overrides when necessary. In the case of shared products, a product group configuration is considered an implementation if a shared product. The idea is to have re-usable configuration so teams/product groups can have a role that maps to the product and uses the shared state configuration.

Regardless of a product being owned exclusively by the Product Group, or being an implementation of a common/shared product, all product configuration specific to the product group should be configured under the pillar location pillar://config/***PRODUCTGROUP.*** The configuration information for each role could be in init.sls or any other pillar file, as long as the pillar is made available in pillar top. Refer to Use Case 1.

Depending on the desired workflow for the product group, each role could be defined in it's own pillar configuration file.

For these examples, we will assume every role for the team is configured in init.sls and the product group is groupx.

This sample product configuration file defines a few different product/roles with the available options.

Note: The product configuration is where all Pillar data needed by the product/role salt state should reside. When a salt state for a product is created that requires some configuration, especially configuration that may change from environment to environment and group to group, You don't want to put that configuration in the state. That product level configuration should be added to this role configuration so the salt states can realize the data at runtime. More on this in the State Design page.

```
#!yaml|gpg
# ######################################
# CONFIGURATION OF GROUPX PRODUCTS/ROLES
# ######################################

groupX.role:                                 <---- REQUIRED top level ID
must be productgroup.role. '.role' is static and required regardless of
product group

  product-path: /com/groupX/                 <---- could be used to
construct a product group default path to artifacts
```

```
  hooks:                                     <---- OPTIONAL product
groups scope hooks.
    pre-provision-orchestration:
      state: ['orch.groupX.pre-provision']   <---- the state
configuration should refer to a valid salt state. In this case
salt://orch/groupX/pre-provision
      enable: True
    pre-startup-orchestration:
      state: ['orch.groupX.pre-startup']
      enable: True
    post-startup-orchestration:              <---- these hook names are
static and baked into the Conductor.
      state: ['orch.groupX.post-startup'] .        Refer to
Orchestration Hooks document for full list of valid hooks. The states
however are user defined.
      enable: True
    pre-destroy-orchestration:
      state: ['orch.groupX.pre-destroy']
      enable: True                           <---- enable / disable
flag. True | False
    post-destroy-orchestration:
      state: ['orch.groupX.post-destroy']
      enable: True

  all:                                       <---- REQUIRED the 'all:'
block is a list of all the roles that should be available in pillar
    - foobar-core
    - foobar-ui
    - foobar-db-scripts
    - foobar-api                             <---- product group
exclusive as well as shared product implemented roles.
    - dummy
    - cassandra
    - activemq

  foobar-ui:                                 <---- The Role. This is
critical for role ID in provisioning template. I.E. "groupX.foobar-ui"
    source-path: /org/release                <---- can be used to
construct full path to artifact. https://artifactrepo-url/. Replace
'org' with your org.
    product-path: /com/groupX/foobar/1.0.1/  <---- continued artifact
path
    product-name: foobar-ui
    product-version: 1.0.1
    package-name: foobar-ui-1.0.1.war
    mod-package-name: foobar_ui.war          <---- could use something
like this to have a salt state rename the artifact once it's deployed to
instance
    package-type: war
    schema-package: foobar-seed-data.tar.gz  <---- example when salt
```

```
state needs to apply a db schema specific to the product/role
    dependencies:                            <---- A method to depend on
a state without requiring role to be composite role.
      groupX.foobar-api: 1.0.0
      groupX.foobar-core: 2.0.1
    tomcat:                                  <---- example tomcat
multi-homed configuration
      instance: [6,8]                        <---- which tomcat
containers to put this app (assume topcat multi-instance)
      version: 8.0.39
      package: apache-tomcat-8.0.39.tar.gz
      extended-properties:                   <---- used by custom built
tomcat state
        TESTONE: 1
        TESTTWO: 2
      property-override:
        max-heap-size: 2048
        memory-size: 128
        max-memory-size: 356
        spring-profile: uat
      extended-catalina-opts:
        - "-DgroupX.foobar.configDir=$TOMCAT_CONFIG_HOME"
        - "-DgroupX.foobar.configDirTEST1=$TOMCAT_CONFIG_HOME"
        - "-DgroupX.foobar.configDirTEST2=$TOMCAT_CONFIG_HOME"
    java:
      version: 1.8.0_121
      package: jdk-8u121-linux-x64.rpm
    db:
      user: foobar_db
      password: |
        -----BEGIN PGP MESSAGE-----
        Version: GnuPG v2.0.22 (GNU/Linux)

        CIPHER-GENERATED-ON-SALT-MASTER-XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        =XXXX
        -----END PGP MESSAGE-----
    state: ['groupX.foobar-ui']                <---- REQUIRED list of
salt states for this role
    security-group: public                     <---- public | private
default security group membership
    elbv2:                                     <---- needs elb, will be
created if 'FOOBAR-UIV2' doesn't exist
      name: FOOBAR-UIV2
      interval: 10
      timeout: 5
      healthy_threshold: 2
      unhealthy_threshold: 4
      target: /health
      forwarding_port: 80
      scheme: internal

  foobar-db-scripts:
    source-path: /org/release/schemas
    product-name: foobar-db-scripts
    product-version: 3.11.2
    package-name: db-schema-scripts.zip
    package-type: zip
    svc-name:
    state: ['groupX.foobar-db-scripts']
    security-group: private

  foo-api:
    source-path: /org/release
    product-path: /com/groupX/foo-api/2.2.2/
    product-name: foo-api
    product-version: 2.2.2
    package-name: foo-api-2.2.2.war
    mod-package-name: fooapi.war
    package-type: war
    tomcat:                                       <---- app added to a
single tomcat container on multi-homed instance, no additional tc config
      instance: 8
    java:                                         <---- app requires
specific java. Java salt state is a special case in that it is not a
role in itself.
      version: 1.8.0_121                                There is an
environment scope default java version, this not only tell salt to
install the java salt
      package: jdk-8u121-linux-x64.rpm                  salt state, but
overrides the environment default version configured in
pillar://config/common.sls
    state: ['groupX.foo-api']
    security-group: public

  foo-core:
    source-path: /org/release
    product-path: /com/groupX/foo-core/2.2.2/
    product-name: foo-core
    product-version: 2.2.2
    package-name: foo-core-2.2.2.war
    mod-package-name: foocore.war
    package-type: war
    java:
      version: 1.8.0_121
      package: jdk-8u121-linux-x64.rpm
    state: ['groupX.foo-core']
    security-group: public

  dummy:                                          <---- example dummy
test role. simply invokes salt://common/dummy/init.sls. I.E. this role
groupX/foobar
    security-group: public                              doesn't need a
wrapper role that 'implements' dummy. This simply calls common.dummy
state
    state: ['common.dummy']

  activemq:
    security-group: private
    state: ['groupX.activemq']
    hooks:                                        <---- OPTIONAL role
scoped hooks. If the role is an implementation of a shared common role,
the scope
      pre-provision-orchestration:                      is role but
within this product group.
        state: ['orch.groupX.pre-prov.activemq']
        enable: True
      pre-startup-orchestration:
        state: ['orch.groupX.pre.activemq']
        enable: True
      post-startup-orchestration:
        state: ['orch.groupX.post.activemq']
        enable: True
      pre-destroy-orchestration:
        state: ['orch.salty.pre.destroy.activemq']
        enable: True
      post-destroy-orchestration:
        state: ['orch.groupX.post.destroy.activemq']
        enable: True
      pre-upsize-orchestration:
        state: ['orch.groupX.pre.upsize.activemq']
        enable: True
      post-upsize-orchestration:
        state: ['orch.groupX.post.upsize.activemq']
        enable: True
      pre-downsize-orchestration:
        state: ['orch.groupX.pre.downsize.activemq']
        enable: True
      post-downsize-orchestration:
        state: ['orch.groupX.post.downsize.activemq']
        enable: True

  cassandra:                                          <---- A cluster
role. But no cluster specific configuration goes here.
    security-group: private                                 All in
cluster configuration is located
pillar://provisioning/templates/groupX/role.sls
```

```
hooks:
  pre-startup-orchestration:
    state: ['orch.groupX.pre.cassandra']
    enable: False
  post-startup-orchestration:
    state: ['orch.groupX.post.cassandra']
    enable: False
state: ['groupX.cassandra']
java:
  version: 1.8.0_121
```

```
      package: jdk-8u121-linux-x64.rpm
```

Product Provisioning Configuration

This section details the configuration needed for Conductor provisioning. This configuration is not the place to put configuration that is needed by the product/role salt state. (that goes in the Product Configuration as noted in the previous section). The provisioning configuration for all product groups and products/roles defined for each product group goes in pillar://provisioning/templates/productgroup/... This is not a salt requirement, but a housekeeping facet of the Framework model.

These 'product provisioning' configurations are basically templates and are realized and processed by Conductor during the phase of generating the cloud configuration files for salt-cloud. As noted above, any pillar information required in the salt states should be defined in the 'Product Configuration'.

The idea behind these provisioning templates is to provide product groups the ability to create multiple templates of similar or not similar products within the product group. But at the same time isolate environment deltas (variable configuration) from product group and role.product specific configuration in a hierarchy of Pillar configuration to avoid conflicting configuration between product groups within the enterprise.

This Provisioning configuration template shows the various options available to configure non-cluster type role instances. Following the example for Product Configuration, we will use groupX product group and it's associated products. ***Note that many of these configurations are optional and some will be overriding environment scope defaults. The example immediately following this is a basic minimum configuration.***

Non-Cluster Role Provisioning template example

a non-cluster provisioning template example

```
groupX.activemq:                                                <--- the
role ID. Must be at top in yaml, must be unique in the environment
  force-delay-state: True                                       <---
required when needing to delay salt startup_state run
  startup-override: ['groupX.activemq.alternate', 'common.jq']  <---
override the default startup state for all instances, new-instance.sls
  spot_config:                                                  <---
using spot request
    spot_price: 0.10
    tag:                                                        <---
spot request tags, **saltstack feature contribution
      owner: dice
      contact: paul bruno
      purpose: testing
  block-volume:                                                 <---
ordered list of block volume devices, any quantity
    - device-name: /dev/sda1
      volume-size: 45                                           <---
overrides default aws root volume size (8gb)
      volume-type: gp2                                          <---
overrides default aws, its the same actually
      tag:                                                      <---
block volume tagging, ** another saltstack feature contribution. See
Block Device Tagging
```

```
        owner: dice
        description: block device tag testing
        Contact: paul bruno
        Team: dice
        group: groupX
    - device-name: /dev/sdb
      volume-size: 10
      volume-type: gp2
      tag:
        Name: other volume
        description: additional block device
  ebs-optimized: True                                           <--- aws
ebs optimize volume flag
  ami-override:                                                 <---
override environment default ami per region
    us-east-1: ami-9999999
    us-west-2: ami-8888888
  root-volume-tags:                                             <---
root volume tagging, use when root volume is not defined in
block-volumes
    owner: paulbruno
    purpose: salt dev root vol tag testing
    Contact: paul bruno
  persist-volumes: False                                        <--- do
not keep volumes when instance is terminated
  role: groupX.activemq                                         <---
REQUIRED the productgroup.rolename
  additional-grains:                                            <--- add
any other grains to be put on the minion when bootstrapped, list, dict,
string, int
    product.group: groupX                                       <----
THIS IS NOW BAKED INTO CONDUCTOR, thus its not needed. But you can
specify it anyway.
    test-grain: foobar
    another:                                                    <---
List type grains
      - one
      - two
    yetanother:                                                 <---
Dict type grains
      bob: employee1
      sue: employee2
  basename: groupX-mqXX.REGION.ENV                              <---
REQUIRED specify a base name pattern to be used for Name instance tag
and hostname
  nodes: 1                                                      <---
REQUIRED number of default instances created, usually 1, can override on
conductor cmd
  size: t2.small                                                <---
REQUIRED instance type
```

```
  tags:                                                         <---
instance tags
```

```
    owner: paul bruno
    purpose: salt dev testing
    Team: dice
```

Bare minimum provisioning configuration (non-cluster role)

minimum provisioning configuration

```
groupX.activemq:                                                <--- the
role ID. Must be at top in yaml, must be unique in the environment
  role: groupX.activemq                                         <---
REQUIRED the productgroup.rolename
  additional-grains:                                            <--- add
any other grains to be put on the minion when bootstrapped, list, dict,
string, int
    product.group: groupX                                       <----
THIS IS NOW BAKED INTO CONDUCTOR, thus its not needed. But you can
specify it anyway.
  basename: groupX-mqXX.REGION.ENV                              <---
REQUIRED specify a base name pattern to be used for Name instance tag
and hostname
  nodes: 1                                                      <---
REQUIRED number of default instances created, usually 1, can override on
conductor cmd
  size: t2.small                                                <---
REQUIRED instance type
```

Cluster Role Provisioning configuration (basic)

Below we are just showing the configuration options related to cluster type roles. All other configuration options specified in the previous examples are valid as well for cluster roles

This is a very basic cluster with 3 different cluster member functions (called internal roles in the framework). Each member type is using the role default configuration, refer to upstream-config option. Each member instance is created with 1 additional volume of 100G.

basic cluster provisioning template example configuration

```
groupX.nifi:                            <--- REQUIRED role id
  role: groupX.nifi                     <--- REQUIRED role grain
  role.base: nifi                       <--- use when implementing
common/shared products. good for housekeeping
  cluster: True                         <--- REQUIRED when cluster role
type
  nodes: 1                              <--- REQUIRED set to 1 when a
cluster role (use cluster-config to specify member internal role and
quantity)
  size: t2.medium                       <--- REQUIRED
  volume-info:                          <--- OPTIONAL added additional
```

```
volumes to create. This is the base configuration for all cluster
instance.
    data:                                   These can be not including
in the configuration for any member node defined in the cluster-config
      device: /dev/xvdf
      size: 100
      type: gp2
      tags:
        description: all data
  cluster-config:
    primary:                                           <--- primary and
secondary are the 2 available base internal role types for clusters in
framework
      basename: groupX-nifimgr-XX.CLUSTERID.REGION.ENV <--- REQUIRED
base name pattern. Should always use XX.CLUSTERID.REGION.ENV plus
desired prefix
      upstream-config: True                            <--- tells
conductor to use default role configuration, I.E. no internal role
specific configuration
      nodes: 1                                         <--- Always 1
primary regardless of this value
    secondary:                                         <--- primary and
secondary are the 2 available base internal role types for clusters in
framework
      basename: salty-nifi-XX.CLUSTERID.REGION.ENV     <--- REQUIRED
each member type needs to at least define it basename pattern and nodes
      upstream-config: True                            <--- using
default role configuration, secondary internal role has a specific
configuration
      nodes: 2                                         <--- REQUIRED
default is to always create 2 secondaries in each cluster in this
example
    dummy:                                             <--- 'dummy' is a
third internal role type designation for groupX.nifi cluster role
      basename: salty-nifi-dummy-XX.CLUSTERID.REGION.ENV
      upstream-config: True
      nodes: 1
  tags:                                                <--- Instance
tags
    owner: paulbruno
    purpose: salt dev testing
    Contact: paul bruno
    Team: salty
```

Cluster Role Provisioning configuration (complex)

In this example, you see the same cluster role, but a different configuration. This case secondary member types get an entirely different configuration, but dummy member types still use the default role configuration.

basic cluster provisioning template example configuration

```
groupX.nifi:                           <--- REQUIRED role id
  role: groupX.nifi                    <--- REQUIRED role grain, could be different that role ID, but better to keep the same
  role.base: nifi                      <--- use when implementing common/shared products. good for housekeeping
  cluster: True                        <--- REQUIRED when cluster role type
  nodes: 1                             <--- REQUIRED set to 1 when a cluster role (use cluster-config to specify member internal role and quantity)
  size: t2.medium                      <--- REQUIRED
  ebs-optimized: False                 <--- OPTIONAL
  persist-volumes: False               <--- OPTIONAL
  volume-info:                         <--- OPTIONAL added additional volumes to create. This is the base configuration for all cluster instance.
    data:                                   These can be not including in the configuration for any member node defined in the cluster-config
      device: /dev/xvdf
      size: 50
      type: gp2
      tags:
        description: application data
    logs:
      device: /dev/xvdg                <--- *** all configuration data below role id 'groupX.nifi' is the DEFAULT. In cluster-config below, you can define
      size: 10                              which members (different internal roles), will use this configuration (upstream-config), or have it's own defined.
      type: gp2
      tags:
        description: application logs
  cluster-config:
    primary:                                           <--- primary and secondary are the 2 available base internal role types for clusters in framework
```

```
      basename: groupX-nifimgr-XX.CLUSTERID.REGION.ENV <--- REQUIRED
base name pattern. Should always use XX.CLUSTERID.REGION.ENV plus
desired prefix
      upstream-config: True                            <--- tells
conductor to use default role configuration, I.E. no internal role
specific configuration
      nodes: 1                                         <--- Always 1
primary regardless of this value
    secondary:                                         <--- primary and
secondary are the 2 available base internal role types for clusters in
framework
      basename: salty-nifi-XX.CLUSTERID.REGION.ENV
      upstream-config: False                           <--- NOT using
default role configuration, secondary internal role has a specific
configuration
      nodes: 2                                         <--- the default
is to always create 2 secondaries in each cluster
      size: t2.small
      ami-override:                                    <--- secondary is
using special ami
        us-east-1: ami-1234612987
      ebs-optimized: False
      persist-volumes: True                            <--- secondary
need volumes to persist after termination
      volume-info:                                     <--- secondary is
NOT using upstream (default) volume config, it has its own configuration
        other_data:
          device: /dev/xvdf
          size: 100
          type: gp2                                    <--- device,
size, type and tags are the ONLY options in volume-info block.
          tags:
            description: other application data
       logs:
         device: /dev/xvdh
         size: 100
         type: gp2
         tags:
           description: bigger log storage
       user_data:
         device: /dev/xvdi
         size: 100
         type: gp2
         tags:
           description: user data
    dummy:                                             <--- 'dummy' is a
third internal role type designation for groupX.nifi cluster role
      basename: salty-nifi-dummy-XX.CLUSTERID.REGION.ENV
      upstream-config: True                            <--- 'dummy'
internal role is using the default configuration for cluster nodes
```

```
      nodes: 1
  tags:                                                    <--- Instance
tags
    owner: paulbruno
    purpose: salt dev testing
    Contact: paul bruno
    Team: salty
```

Composite Role Provisioning configuration

Below is a simple example of a composite role. It is no different than a standard non-cluster role with the exception of one salt grain, which is a list of the roles these instance will be. The conductor framework looks for this composite.role grain each time it targets instances based on role grain. In order words the role grain is always checked as well as the composite role grain list.

> salt-states should be developed following this same principle. If you have a salt state that installs or does some configuration, you want to make sure the minion that the state is being applied to (in the event a bad target was specified or in pillar) is "supposed" to get this state. By checking the salt grain "role" and "composite.role" you can be sure that no salt state will ever get applied to an incorrect minion.
>
> See Safe Configuration

> **Composite roles CANNOT contains Cluster roles.**

composite role provisioning example

```
groupX.the-kitchen-sink:
  role: groupX.the-kitchen-sink
  composite.role: ['groupX.activemq', 'groupX.dummyrole', 'groupX.app1', 'groupX.svc1']
  additional-grains:
    product.group: groupX
  basename: groupX-kitchensink-XX.REGION.ENV
  nodes: 1
  size: t2.small
  tags:
    purpose: salt testing of composite roles
    contact: paul bruno
    Team: dice
```

Salt-cloud volume tagging

Tagging instance volumes in salt-cloud configuration can take place in more than one configuration block. EBS block volumes (including /dev/sda1 AWS default root volume) can be configured to have tags per device in the block_device_mapping configuration (as of PR-48716 feature add).

ref: Saltstack documentation

ref: In-house example

As shown in the examples, you can configure one of more EBS volume devices in block_device_mapping. This means, you can optionally configure specific properties for the root volume as well. See AWS for a valid list of block devices that can be specified.

Some complexity is introduced because you can optionally configure the root device volume /dev/sda1 or not and either way AWS will create it.

> The root device volume is AWS ec2 is /dev/sda1. AWS seems to always create this as the root regardless of what we configure. For example if we do not configure the /dev/sda1 device in block_device_mappings, AWS will still create it with the default 8gb. The process for changing the root volume is to detach this device and re-attach to a new one. Currently our automation framework doesn't have the use case to require detach and re-attach devices.

If we configure the root device /dev/sda1 in block_device_mappings, we can specify one or more tags, ebs vol size and type etc... as shown in the Salt-cloud documentation. However if we do not configure the root device /dev/sda1 in block_device_mappings, there is no "built-in" salt-cloud way of adding tags to the root device. The solution is root-volume-tags added to our pillar model for provisioning instance with salt-cloud via the conductor runner.

With these two configuration options, we are able to tag any block device including the root device when auto-created by AWS without any given configuration.

The third way to tag volumes is via the built-in salt-cloud volumes configuration shown in the documentation. No custom functionality is needed to tag any number of volumes to create and attach with each instance being provisioned. Additionally, the conductor framework has logic to assure at least one tag, 'Name', is set when creating instances. If the 'Name' tag is not specified in our user salt-cloud configuration, it will be auto-created and set to the hostname of the new instance.

Each volume create and tagging configuration option is outlined in the sub sections.

Block device tagging

Tag one or more EBS block devices, including the ec2 root device using salt-cloud builtin block_device_mappings configuration. The salt conductor will generate the salt-cloud configuration files based on pillar data in our salt-pillar git repository.

Below are Pillar examples as the input, and the output salt-cloud configuration representation. The output and pillar input are snippets of a larger content of configuration that is required in practice.

Example 1

This configuration will create the salt-cloud config for root device being configured in block_device_mappings along with a second block device. Each with user defined tags, but no Name tag defined. Conductor will automatically create the 'Name' tag regardless. This example also sets the root-volume-tags section which is describes later, but note that setting the root device tags in both locations, block-volumes wins. Notice the 'Name' tag is also not set in root-volume-tags. ***Tags are not cumulative when both block-volumes has root device tags and root-volume-tags are configured.***

There is an environment scoped common default available in the Conductor framework for block-volumes. Note that there is NOT a common default for root-volume-tags.

pillar config:

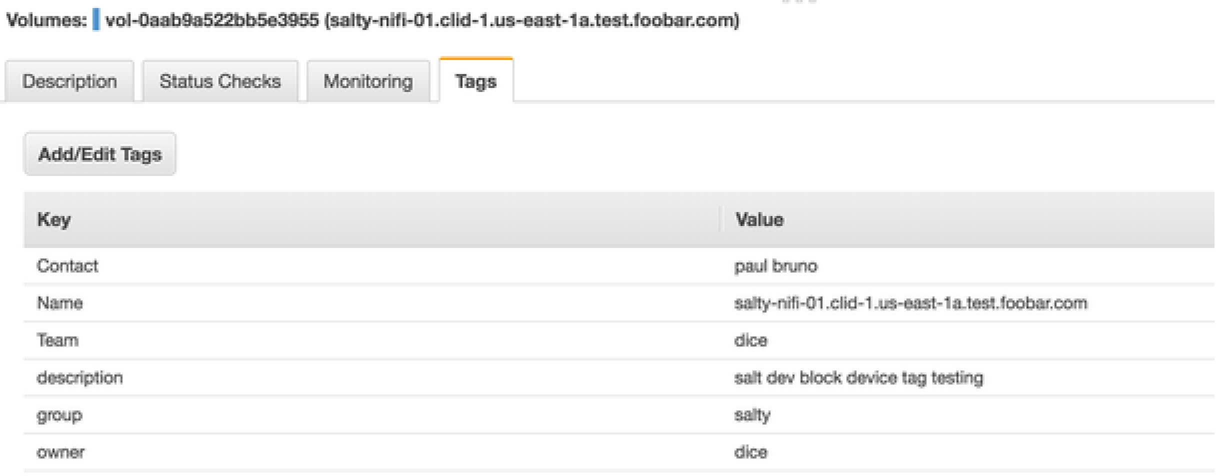
```
block-volume:
    - device-name: /dev/sda1
      volume-size: 45
      volume-type: gp2
      tag:
        owner: dice
        description: salt dev block device tag testing
        Contact: paul bruno
        Team: dice
        group: salty
    - device-name: /dev/sdb
      volume-size: 15
      volume-type: gp2
      tag:
        owner: dice
        description: second block device test
        Contact: paul bruno
        Team: dice
        group: salty
  root-volume-tags:
    owner: paulbruno
    purpose: salty nifi root vol tag testing
    Contact: not me
    Team: dice
```

output salt-cloud config:

```
block_device_mappings:
        - DeviceName: /dev/sda1
          Ebs.VolumeSize: 45
          Ebs.VolumeType: gp2
          tag:
            group: salty
            Name: salty-nifi-01.clid-1.us-east-1a.test.foobar.com
            Contact: paul bruno
            Team: dice
            owner: dice
            description: salt dev block device tag testing
        - DeviceName: /dev/sdb
          Ebs.VolumeSize: 15
          Ebs.VolumeType: gp2
          tag:
            group: salty
            Name: salty-nifi-01.clid-1.us-east-1a.test.foobar.com
            Contact: paul bruno
            Team: dice
            owner: dice
            description: second block device test
```

Notice root-volume-tags configured in pillar information is not in this salt-cloud configuration. It is a custom feature added in conductor framework to support root volume tagging when block_device_mappings (block-volume) is not used.

aws tags set:



Volumes: vol-0aab9a522bb5e3955 (salty-nifi-01.clid-1.us-east-1a.test.foobar.com)

Key	Value
Contact	paul bruno
Name	salty-nifi-01.clid-1.us-east-1a.test.foobar.com
Team	dice
description	salt dev block device tag testing
group	salty
owner	dice

Notice the 'owner' is set to the value configured in block_device_mappings, so when root-volume-tags is also set, the former wins.

Example 2

This configuration will create the salt-cloud config for root device being configured in block_device_mappings along with a second block device but no tags configured here. For the root device volume, the root-volume-tags configuration will be applied to the root device only for the instance. The root-volume-tags configuration DOES specify the 'Name' tag, so Conductor will NOT overwrite this with the instance hostname, but persist this user defined Name instead. However, since the second device has no tags in block-volume, and thus does not specify the default required Name tag, Conductor will automatically create the 'Name' tag for the second device, /dev/sdb, and thus it will end up with one tag only. The root device /dev/sda1 will end up configured with 5 tags.

pillar config:

```
block-volume:
    - device-name: /dev/sda1
      volume-size: 45
      volume-type: gp2
    - device-name: /dev/sdb
      volume-size: 15
      volume-type: gp2
  root-volume-tags:
    Name: foo-bar-server
    owner: paulbruno
    purpose: salty nifi root vol tag testing
    Contact: not me
    Team: dice
```

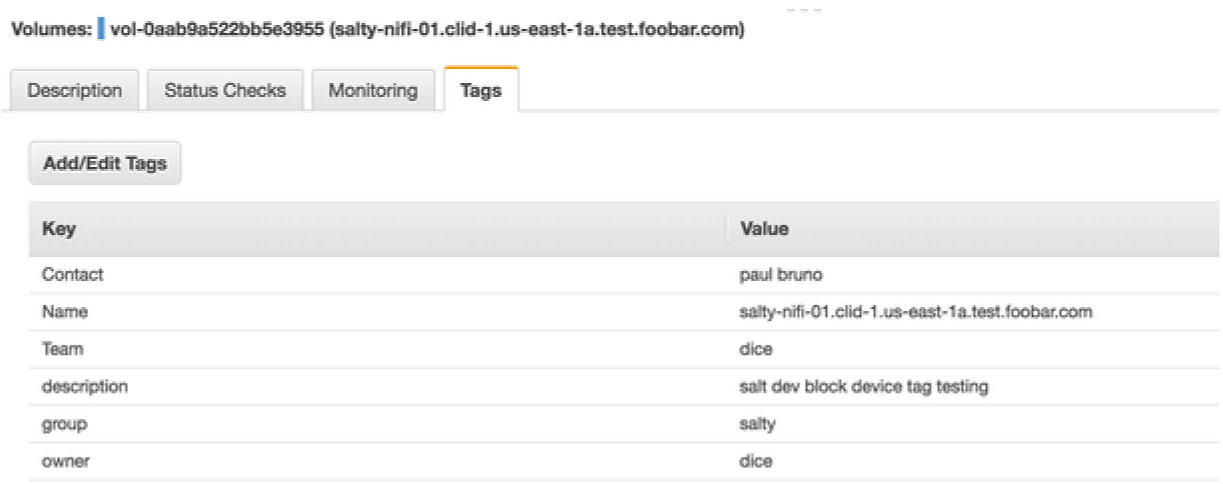
Note: we could create more than two device in the block-volume configuration. Limited to only what AWS limits and using valid block device names. So block-volume is an ordered List containing dictionary entries, an OrderedDict in python terms. root-volume-tags is a dictionary only since there is only one root device.

output salt-cloud config:

```
block_device_mappings:
        - DeviceName: /dev/sda1
          Ebs.VolumeSize: 45
          Ebs.VolumeType: gp2
          tag:
            Name: salty-nifi-01.clid-1.us-east-1a.test.foobar.com
        - DeviceName: /dev/sdb
          Ebs.VolumeSize: 15
          Ebs.VolumeType: gp2
          tag:
            Name: salty-nifi-01.clid-1.us-east-1a.test.foobar.com
```

Note: in this case the 'Name' default tag was created for both by Conductor in the salt-cloud block_device_mappings. The root-volume-tags (custom implementation) would not really be needed in this case in our pillar config since we are configuring the /dev/sda1 block device in block-volumes where would could have added all the user defined tags. However, for demonstration purposes, our pillar config does define root-volume-tags, and thus will be added AFTER the new instance is created. The salt-cloud configuration shown above is generated by Conductor and is processed BEFORE the instance exists. Therefore, the custom Name tag we are specifying in root-volume-tags will be applied post instance create and will overwrite the default 'Name' hostname value that Conductor auto-created in this salt-cloud config. We can see the final tag in the new section.

root device tags (/dev/sda1)

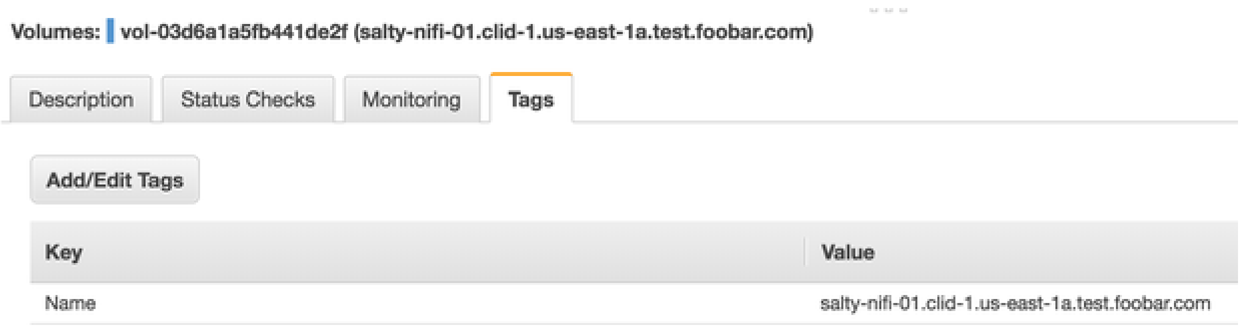


Volumes: vol-0aab9a522bb5e3955 (salty-nifi-01.clid-1.us-east-1a.test.foobar.com)

Key	Value
Contact	paul bruno
Name	salty-nifi-01.clid-1.us-east-1a.test.foobar.com
Team	dice
description	salt dev block device tag testing
group	salty
owner	dice

Notice the 'Name' tag was overwritten from what was configured in the block_device_mappings salt-cloud configuration

second device tags (/dev/sdb):



Volumes: vol-03d6a1a5fb441de2f (salty-nifi-01.clid-1.us-east-1a.test.foobar.com)

Key	Value
Name	salty-nifi-01.clid-1.us-east-1a.test.foobar.com

Root volume tagging (when root is not defined on block-volumes)

This section describes root-volume-tags Pillar configuration. This is a custom piece of functionality in Conductor to fill the gap when root device volume (/dev/sda1) is not configured in block-volumes (which generates into salt-cloud config block_device_mappings. As noted above, if the user defines the root device tags in both locations, block-volumes and root-volume-tags, only block-volumes configured root device tags will be the only ones used. This pillar configuration is only used for root device (/dev/sda1) tag definitions only.

In addition to the above examples which show the relationship and behavior between root-volume-tags and block-volumes in the context of root device, two additional example is shown here.

Example 1

pillar config

```
block-volume:
    - device-name: /dev/sdb
      volume-size: 15
      volume-type: gp2
      tag:
        purpose: second block device test
  root-volume-tags:
    owner: paulbruno
    purpose: salty nifi root vol tag testing
    Contact: someone else but me
    Team: dice
```

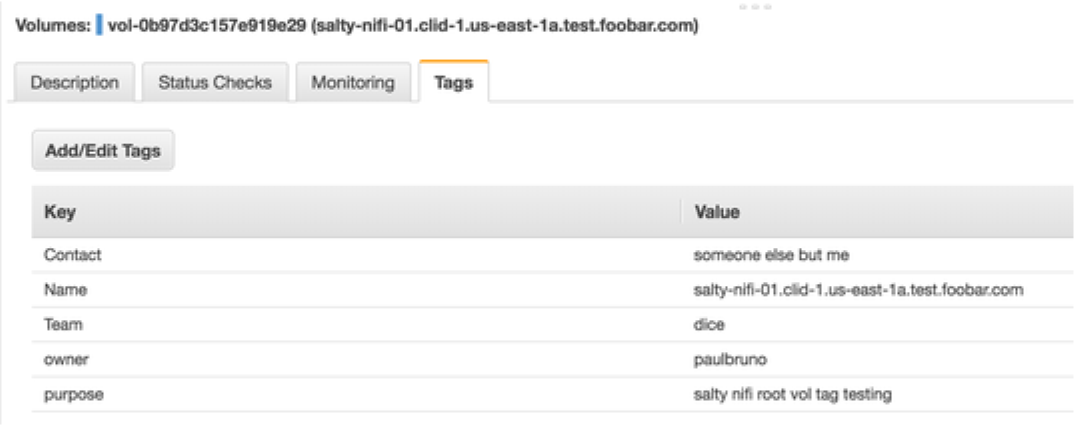
Notice root device volume is NOT configured in block-volumes, it will be created by aws and the tags in root-volume-tags section will be applied in addition to the 'Name' default that is not defined, but will be auto-created by Conductor.

output salt-cloud config:

```
block_device_mappings:
        - DeviceName: /dev/sdb
          Ebs.VolumeSize: 15
          Ebs.VolumeType: gp2
          tag:
            purpose: second block device test
            Name: salty-nifi-dummy-01.clid-1.us-east-1a.test.foobar.com
```

Notice the required default 'Name' tag is configured in the generated salt-cloud config, but was not in the pillar configuration. Conductor will always create this and set it to hostname if it doesn't exist.

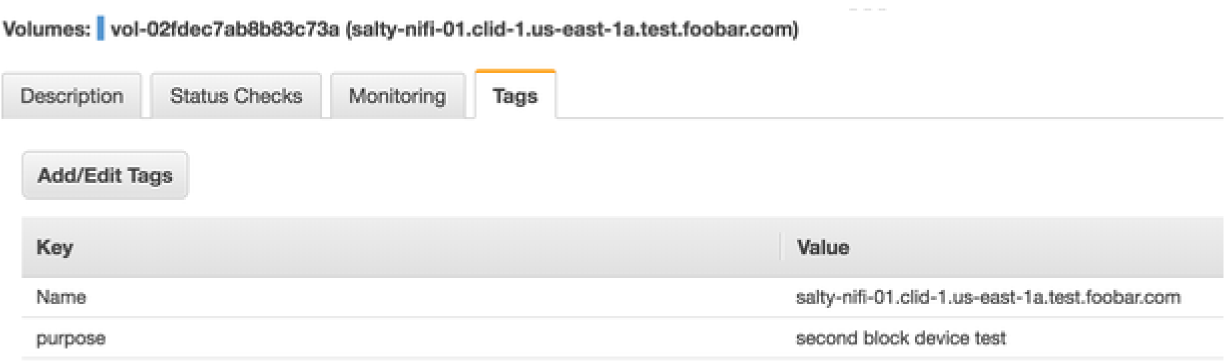
root device tags (/dev/sda1)



Volumes: vol-0b97d3c157e919e29 (salty-nifi-01.clid-1.us-east-1a.test.foobar.com)

Key	Value
Contact	someone else but me
Name	salty-nifi-01.clid-1.us-east-1a.test.foobar.com
Team	dice
owner	paulbruno
purpose	salty nifi root vol tag testing

secondary block device tags (/dev/sdb)



Volumes: vol-02fdec7ab8b83c73a (salty-nifi-01.clid-1.us-east-1a.test.foobar.com)

Key	Value
Name	salty-nifi-01.clid-1.us-east-1a.test.foobar.com
purpose	second block device test

Example 2

In this example, we are not defining the block-volumes at all. root-volume-tags are configured, and the required 'Name' tag is also configured, so Conductor will NOT overwrite this. Also note that in this case, we do not have the environment scope common default block-volume configuration.

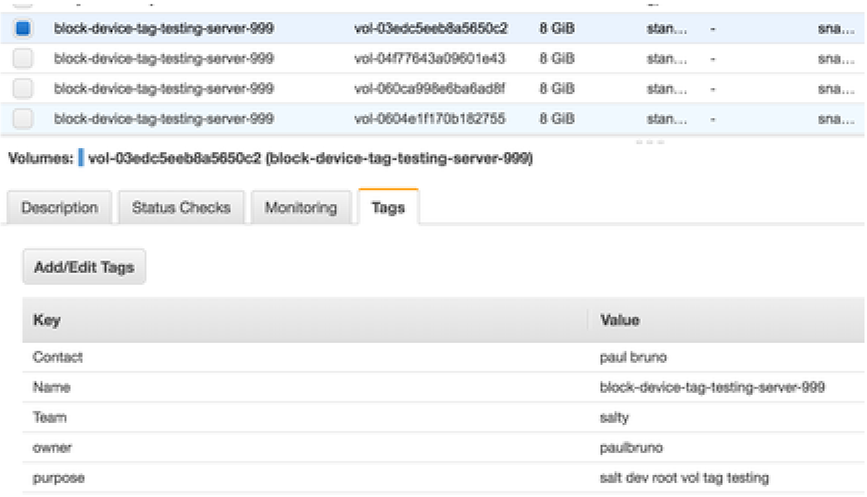
pillar config:

```
root-volume-tags:
    Name: block-device-tag-testing-server-999
    owner: paulbruno
    purpose: salt dev root vol tag testing
    Contact: paul bruno
    Team: salty
```

output salt-cloud config:

There is not salt-cloud configuration generated from the above root-volume-tags configuration since it's not an internal salt-cloud feature. Additionally we are not configuring block-volumes, so no block_device_mappings configuration is generated either. Note however that if we defined the environment scope common default block-volume, we would have seen the block_device_mappings section in the salt-cloud configuration.

root device tags (/dev/sda1)



■	block-device-tag-testing-server-999	vol-03edc5eeb8a5650c2	8 GiB	stan...	-	sna...
□	block-device-tag-testing-server-999	vol-04f77643a09601e43	8 GiB	stan...	-	sna...
□	block-device-tag-testing-server-999	vol-060ca998e6ba6ad8f	8 GiB	stan...	-	sna...
□	block-device-tag-testing-server-999	vol-0604e1f170b182755	8 GiB	stan...	-	sna...

Volumes: vol-03edc5eeb8a5650c2 (block-device-tag-testing-server-999)

Key	Value
Contact	paul bruno
Name	block-device-tag-testing-server-999
Team	salty
owner	paulbruno
purpose	salt dev root vol tag testing

Notice the root device volume is created without defining anything in pillar. Also note the 8 GB default aws volume size for root devices. The 'Name' tag was not auto-set to hostname since we defined it in root-volume-tags.

There is no secondary device configured, so only the aws auto-created root device volume exists /dev/sda1

Example 3

In this example, we do not have any pillar configuration for block-volumes or root-volume-tags. Thus the Conductor generated salt-cloud configuration would have no block_device_mappings.

The created aws instances would have no tags on the root device volume. This should not be the case. The Conductor framework must have either the root-volumes-tags configuration or the block-volumes configuration to enable tagging. Leaving both these out, would give the default aws behavior to not tag the root device volume.

This is why we have the environment scope common default for block-volume in the event a team/individual uses a pillar configuration that has no tag configuration. The common default will be enabled in each pillar environment in practice.

Other volume tagging

About delaying the startup_states

Putting this is the provisioning template tells conductor to remove the startup_states state list entry for each role/minion in the generated salt-cloud configuration and replace with a placeholder state.

This is a work around from a salt bug in previous versions. By doing this conductor holds off executing the startup_states configured for each newly provisioned minion until after the salt bootstrap. The salt bootstrap process will execute any startup_state configuration for the minion.

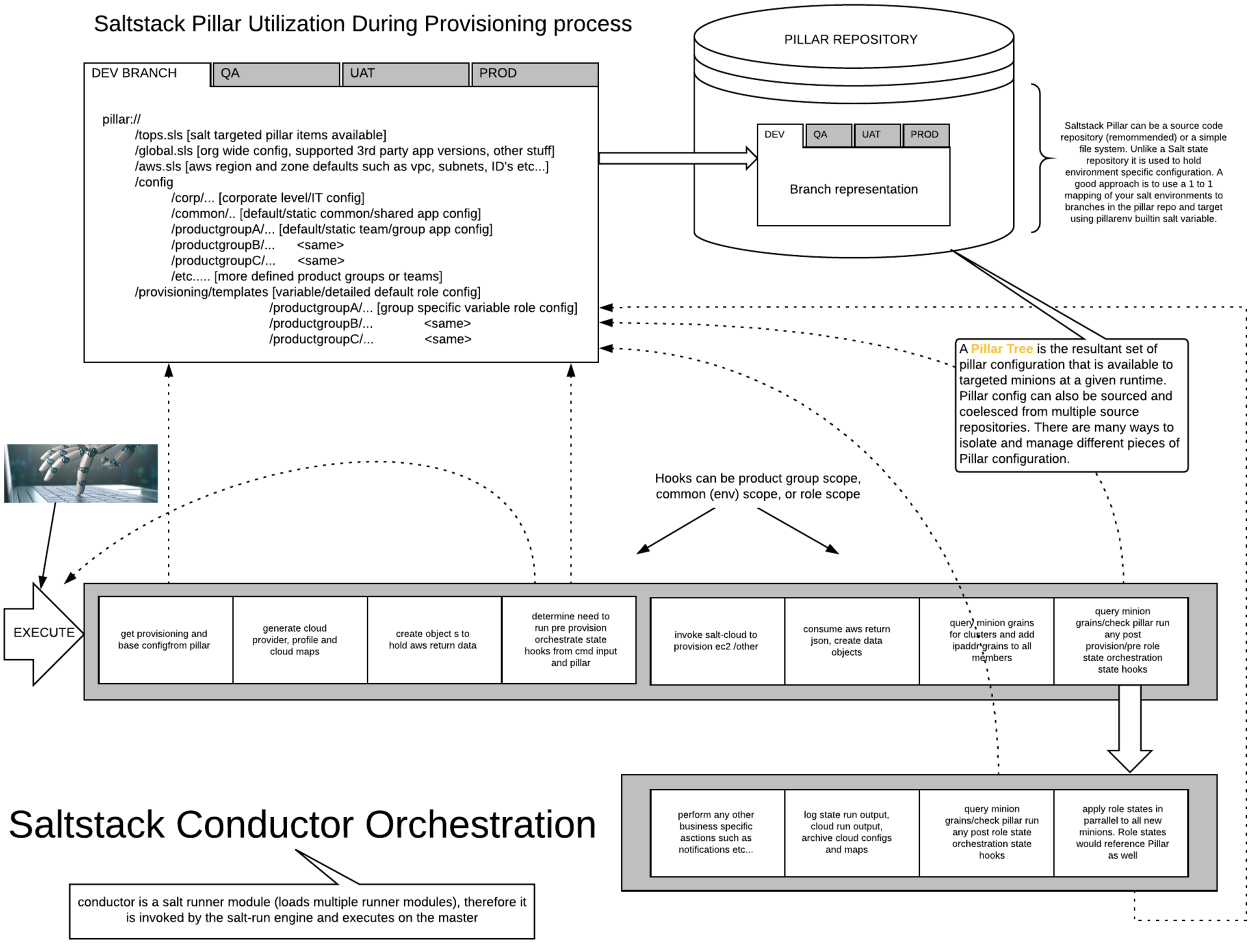
The problem was that if you are using any custom execution modules (which is most likely the case), the salt state tree available immediately to the minion after the instance is created and bootstrapped, will NOT have any custom modules available. The custom modules are located in the salt://_modules directory of you state configuration source.

This was a bug that required running a salt command called sync_modules or sync_all on the new minions. Once this was done, the entire salt state tree was available including any custom execution modules. The bug may have been resolved in these later versions, but it would need full testing.

This solution of force-delay-states is simply shimming in a custom state in replace of the startup_states needed for each minion. So once the salt minion bootstrap is done and AWS/salt-cloud return handle gets back to Conductor, force-delay-states executes which runs sync_modules. Then conductor will go through the dict list of states for each new minion and execute them all in a parallel command.

Diagram

This is a first cut of diagram showing the pillar model used in conjunction with Conductor runner.



Saltstack Pillar Utilization During Provisioning process

DEV BRANCH | QA | UAT | PROD

```
pillar://
        /tops.sls [salt targeted pillar items available]
        /global.sls [org wide config, supported 3rd party app versions, other stuff]
        /aws.sls [aws region and zone defaults such as vpc, subnets, ID's etc...]
        /config
                /corp/... [corporate level/IT config]
                /common/.. [default/static common/shared app config]
                /productgroupA/... [default/static team/group app config]
                /productgroupB/...      <same>
                /productgroupC/...      <same>
                /etc..... [more defined product groups or teams]
        /provisioning/templates [variable/detailed default role config]
                                  /productgroupA/... [group specific variable role config]
                                  /productgroupB/...            <same>
                                  /productgroupC/...            <same>
```

PILLAR REPOSITORY

DEV | QA | UAT | PROD

Branch representation

Saltstack Pillar can be a source code repository (remommended) or a simple file system. Unlike a Salt state repository it is used to hold environment specific configuration. A good approach is to use a 1 to 1 mapping of your salt environments to branches in the pillar repo and target using pillarenv builtin salt variable.

A Pillar Tree is the resultant set of pillar configuration that is available to targeted minions at a given runtime. Pillar config can also be sourced and coelesced from multiple source repositories. There are many ways to isolate and manage different pieces of Pillar configuration.

Hooks can be product group scope, common (env) scope, or role scope

EXECUTE

- get provisioning and base configfrom pillar
- generate cloud provider, profile and cloud maps
- create object s to hold aws return data
- determine need to run pre provision orchestrate state hooks from cmd input and pillar
- invoke salt-cloud to provision ec2 /other
- consume aws return json, create data objects
- query minion grains for clusters and add ipaddr grains to all members
- query minion grains/check pillar run any post provision/pre role state orchestration state hooks

- perform any other business specific asctions such as notifications etc...
- log state run output, cloud run output, archive cloud configs and maps
- query minion grains/check pillar run any post role state orchestration state hooks
- apply role states in parrallel to all new minions. Role states would reference Pillar as well

Saltstack Conductor Orchestration

conductor is a salt runner module (loads multiple runner modules), therefore it is invoked by the salt-run engine and executes on the master