Salt - Conductor Provisioning types

Overview

Conductor is a custom built Saltstack runner. Its purpose is to provide unified, flexible and consistent orchestration framework around the process of provisioning, deployment and configuration of cloud based vm's and cloud components using Saltstack automation. Like all runners, Conductor execution on the salt-master, not the minion (client).

The Salt conductor orchestration runner has 3 top level provisioning types

- instance/s
- cluster/s
- system

These are three very distinct entities that will be described in detail throughout this document.

Additionally, Conductor has 4 top level actions when it comes to managing cloud provider vm's. More actions may be introduced in the future through other submodules

They current supported actions are:

- create
- destroy
- upsize
- downsize

When given a 'create' action command, Conductor will require a variable list of parameters based on the create action request being made. The mix of available and required parameters determines what Conductor is being told to create, or Provision, and thus the provisioning type.

For this document, all topics and command examples will be in the context of the 'create' action unless otherwise noted. Keep in mind that other Conductor actions like destroy, upsize and downsize may need to understand the provisioning types from the model perspective only. For example, the 'destroy' action can be used to destroy instances based on system or cluster identifiers, each of which is a provision type.

Provisioning types

As noted, there are three 'types' of things to provision and configure using Conductor. These are detailed below with some command line examples. Take note that there are specific salt grains that will be created in some types (clusters and systems) that will be used for targeting for state configuration for more than a single instance. Although these grains are not only used for teardown/destroy action, they are detailed in that document HERE.

Instance/s (role based)

The 'instance' provisioning type is the most basic type. The cli parameter role=xxxx tells Conductor to provision and configure a single instance of a particular Role. There is an optional 2nd parameter, 'count=n' that tells Conductor to build n numbers of instance with the role xxxx.

> In the pillar configuration model for the framework, there is a default n number of nodes (instances) for each role to build whenever Conductor is told to create a role based instance. This default 'count' can be overwritten at runtime in the cli. See examples below.

In short a Role is synonymous to a server type, profile, or other name used throughout the industry to identify a function or set of functions that a particular server (machine, vm, instance etc...) will perform in the environment.

Roles are a basic component of the Automation Delivery Framework. They are defined in Pillar configuration, there configuration properties may be derived from common/shared roles (aka base roles, see diagram), or completely unique to a product group. Roles are implemented as a subset of Product Group, so as to allow for re-use of common/shared role configuration as well as to identify by a namespace. In our Salt automation framework, salt states are applied to minions specifically by role in most cases and especially when it comes to state that configures applications and services. Salt grains are used to identify minions' roles along with many other things. Roles usually map 1:1 to a product. This doesn't have to be the case, but logically it makes sense in 95% of the use cases.

Example of a shared role... TeamA (product group) manages a deployed system that includes a rabbitmq cluster of nodes, TeamB also deploys a rabbitmq cluster as part of there application suite (product group). The framework configuration for rabbitmq is common configuration and thus is

shared/re-usable. Each team would have a role that "implements" the common configuration role of rabbitmq with group specific overrides. The available overrides are specific to each technology and our business strategy for supporting it (versions, install options etc...). The two team specific roles would be TeamA.rabbitmq and TeamB.rabbitmq respectively. This would be used for identification in grains, states etc.... along with a built-in grain that is always set, product.group. Together these guarantee a unique method of identifying minions that are using common roles.

There is also the concept of a base role. This is not really a built-in role type in the framework, but more a logical identification that will also be set as a salt grain (base.role). In the event we need to target all salt minions in a given environment that are running a particular product such as rabbitmq regardless of the product group that deployed and is managing it, we could use base.role if desired.

One last role to discuss is a 'composite role'. A composite role is an actual role. It will be set as a grain and it is configured in the Pillar model. In short a composite role is a list type role of other roles.

The use case for composite role is when you have a single instance that must perform many functions that are configured as roles in the model. For example in dev and qa environments it is very common to multi-home servers in order to save resources and do simple function testing of a particular feature. In this case, you probably don't need to spin up and entire set of machines to verify the basic functionality you are testing. This is where composite roles come into play.

Composite roles also provide flexibility in how your vm footprint will look based on traffic and capacity. Maybe you have an application that has a bunch of restful api's and each api is implemented as a separate role (which is the proper way to configure it), and in some environments or regions of the world, you want all rest api's to run on the same machine because you get very little traffic, and other environments you have a very high traffic flow. This is a use case for composite role in the low traffic region.

The composite role is set as the salt grain composite.role. Salt states should be developed to not only verify the role on the machine it is about to run configuration, but also verify the composite.role list grains if it exists. This is how we can assure that no role salt state will get applied to a salt minion that is not supposed to have that role regardless of it being a composite role or standard role.

> In all command line examples, it's assumed the state configuration that implements the role specified in the state tree and contains all setup steps needed with product group implementation specific overrides.
>
> The salt state tree is the state that is available at any given time based on the minions being targeted in the environment specified. This is configured on salt-master and can be a single backend source repository or multiple. It could also be a simple file system backend.

examples of using salt conductor to create a role based instance

```
# create a single instance (assumes default node count is 1 in pillar)
in qa salt environment in a specified aws region/zone configured with
apache activemq role for product group teamA.
> salt-run conductor.create group=teamA role=activemq pillarenv=qa
region=us-east-1a

# create the same as above but create 3 separate instances (assumes
default is not 3) running activemq message broker
> salt-run conductor.create group=teamA role=activemq count=3
pillarenv=qa region=us-east-1a

Constraints:
Role must be a valid role defined in Pillar for the product group
Role must have a configuration provisioning template in Pillar with the
minimum key value pairs configured. See template readme.
Role must be configured in Pillar as a cluster if it spans multiple
instances (vm's)
Member parameter valid only for roles that are configured in Pillar as
cluster
Count parameter is valid only for non-cluster roles
```

very special case

```
The role parameter can also be set to all, i.e. role=all. This is a
special case and will be . documented at a later time. It remains an
undocumented feature until it can be fully understood.
```

Cluster/s (role based)

A cluster is a special type of role. The big difference is a cluster role is a role definition that defined a 'set' of instances that can be identified as one thing. Thus a cluster is a role that when implemented (aka provisioned deployed), will span multiple instances. A cluster can be a single instance of course if configured that way. But the point of the cluster configuration in our Pillar model that Conductor uses, is to provide a single area of configuration to defined a logical set of servers all providing a function in the environment. I.E. a database cluster such as mongodb.

The Conductor command line parameters for creating cluster roles are role=xxxx, and optionally members=n. Just as in a standard non-cluster role, role=xxxx must be a valid role configured in the Pillar tree. And instead of count=n option, you would use members=n option. The members parameter allows you to specify a non-default number of members in the cluster.

Without going into details about the Pillar configuration model used in the framework, just know that there is a separate dictionary key in a role that is a cluster, cluster-config: in yaml. Reference a sample cluster configuration template HERE. This is a complex configuration because it's used as a format and syntax example.

Within the cluster-config section, we specify the 'internal role' for each node type that is in the cluster. It could only be one if the technology does not require differentiation between cluster members. technology like Consul, RabbitMQ, Cassandra (I think) and other all have a leader election process for creating the cluster internally, so no real need to define different internal roles in the cluster-config in Pillar.

However, other technology such as MongoDB, nifi and others do have a function specific internal role in the cluster. For example, mongodb has a primary, a secondary and an arbiter role designation on each cluster member. In these cases, you would configure each internal cluster role separately. The minimum configuration is 3 configuration keys/value items if using the upstream config that is under the top level role configuration in the example template. To summarize, internal role is a salt grain that will be applied to all member nodes when creating a cluster. The grains is set as internal.role and it's purpose is to identify specific members of the cluster that need to perform specific function in the cluster, thus requiring different configuration than other members beyond the common (install) configuration that is defined in the common/sharable state.

Deep diving into the provisioning template configuration is beyond the scope of this document, so we will say no more for now.

examples of using salt conductor to create a cluster role set of instances

```
# create a mongodb cluster (assumes default role is 1 cluster) for teamB
in dev environment (note the cli format is the same for instances and
clusters. clusters are roles)
> salt-run conductor.create group=teamB role=mongodb pillarenv=qa
region=us-east-1a

# create a mongodb cluster and specify a non-default member count (we
will always get one primary in the case the product needs a cluster
manager type node)
> salt-run conductor.create group=teamA role=mongodb members=4
pillarenv=qa region=us-east-1a
```

Systems

A 'system' in Conductor automation framework terms is defined HERE. In short, a system is a logical grouping or collection of various instances running various roles including clusters. In Conductor code terms , it is a high level wrapper around the process of defining and creating multiple instance and/or clusters simultaneously in one Conductor execution. To create a system using Conductor, you specify the system=xxxx

parameter. The system parameter argument must be a valid system configuration available in Pillar for the specified product group and environment. Different than creating role based provision types, the member=n and count=n parameters are ignored if passed in on the command line.

The only optional command parameter for system is sysid=n where sysid is an integer that you wish to designate to the system you are about to create. The sysid is set as a salt grain on ALL instance that are created and configured as part of the create 'system' action in Conductor. If sysid is not specified on the command line, or is already allocated within the salt environment you are targeting, one will be auto-created based on 'next system id' logic built-into Conductor. See section below Salt grains associated with provision types.

The Conductor uses the same code flow to create the cloud configs, and perform all the pre and post orchestration tasks and hooks that is uses when creating role based instance as in the instance/s and cluster/s descriptions above. The big difference is that in code it puts collective logic around whenever it needs to do for role based commands and holds off on invoking salt-cloud until all needed roles and clusters defined in the system configuration in Pillar have been configured in the cloud configuration files. Once provisioned the Conductor will follow the same post provision process of applying states etc...

So the 'system' provisioning type supported in Conductor is merely a high level wrapper around the process of defining and creating a collection of different role based instances using Conductor.

The system configuration in Pillar is very simple and does not have much configuration keys. This is because the product specific (role of cluster role) detailed configuration is still located in the role specific configuration as used when provisioning the other two types. ***System configuration in Pillar is product group specific and there is NO common system configuration!***

Here is a sample of system configuration for the product group called 'salty' (a salt development test product group)

As you can see the format is extremely simple

```
system.PRODUCTGROUP:
  PRODUCTGROUP.system.SYSTEMNAME:
    ROLECLUSTER:
      members: [default | n]
    ROLE:
      count: [default | n]


# REAL EXAMPLE

system.salty:
  salty.system.small_test:
    nifi:
      members: default
    activemq:
      count: 1
  salty.system.big_test:
    mongodb:
      members: 6
    liquibase:
      count: default
```

You specify the top yaml as system.PRODUCTGROUP namespace. The next level down in the yaml is the system name, then dictionary entries for each role to create as part of the system, then in each role either default (which uses the default in the provisioning template Pillar), or count=n or members=n depending on cluster role or not. Note the options here are the same as the command line parameter options for overriding the Pillar default count and members count.

examples of using salt conductor to create full systems

```
# create a system called simple_test for product group salty in the qa
salt environment
>salt-run conductor.create group=salty system=simple_test pillarenv=qa
region=us-east-1a

* you can run this command again, and Conductor will create another full
system with its own unique cloud, system, product group and role
specific id grains ad so forth.
*** you can run a second, third etc.. command at the same time in
parallel from other sessions on salt-master of from some exposed
interface like Rundeck  or Jenkins Jobs, the Saltstack Enterprise web UI
or other custom web UI. For each one you will get a system of instance
all with their own unique set of salt grains used for identification and
targeting when applying state or operational use cases.
```

Salt grains associated with Conductor provision types

Below is a list of some salt grains that are created dynamically and set on instances depending on the provisioning type being provisioned by Conductor. A full list of all the grains both static and dynamic with description that Conductor will create and set by default on new provisioned instances will be available HERE at a later time. The targeted teardown document HERE is also a good reference for what grains we are creating and how we use them within the Saltstack automation.

Below is a brief list of grains that would be set on salt minions created and configured by Conductor that mentioned throughout this document.

grain	description
cpid	cloud provision ID, a unique random 32 bit ID auto-generated at each execution of Salt runner Conductor regardless of provisioning type
cloud.system.id	environment scope unique 'system' provision type ID, auto-generated at each execution of 'system' provisioning type, based on 'next available' logic according to ID's already allocated
productgroup.system.id	product group scoped in the specified salt environment, auto-generated at each execution of 'system' provisioning type, based on 'next available' logic according to ID's already allocated for the product group (team, product suite)
productgroup.system.name	product group specific system name that instance was created as part of
role	product group and function or product specific grain. Value is in the format productgroup.product. I.E role: devops.rabbitmq
cloud.*product(role)*.cluster.id	environment scope unique product (role) cluster ID, auto-generated at each execution of 'Cluster' provisioning type, based on 'next available' logic according to ID's already allocated
productgroup.product(role).cluster.id	product group scope unique product (role) cluster ID in the specified salt environment, auto-generated at each execution of 'Cluster' provisioning type, based on 'next available' logic according to ID's already allocated
product.group	the salt grain that identifies what product group (team | application suite) created the instance, cluster or system. Value derived from Conductor group=xxxx parameter
pillar.environment	derived from Conductor pillarenv=xxx parameter. Another way to target minions based on environment without relying on the minion local config which is set at create time as well.

role.id	pretty much the same as role, but some configuration could be set to define different role ID and role name. Its not recommended to have different values for these, and may not be supported in all functionality of Conductor. In which case this would just be added by Conductor regardless of the role Pillar config having it defined or not.
role.base	set when a role is a shared or common role. In shared role the format of role is productgroup.product. The role.base grain is set to just *product* regardless of the product group
internal.role	a cluster role type would have this set on each member of the cluster depending on the Pillar configuration internal-role. Usually used within product/role specific salt states for technologies that require identifying members of the cluster differently. Mongodb is a good example.
cluster.members	this grain is a List type and is set on all members of a cluster role. This is figured out by Conductor before provisioning and each member of the cluster gets this grain. This is usually used within product/role specific salt states for technologies that require all members to know about all other members. This is very common for local config files the software would use etc... This grain is a List of the members 'names'. This is also the same what the hostname will be set to (Conductor does this) and the same as the 'Name' tag in aws ec2. The built-in salt 'id' will also be set to this once the Conductor changes the hostname form the default cloud provider generated name to the name we assign it based on our Pillar model and cloud configurations Conductor generates for salt-cloud. ***** as a side note, after vm's are created and salt bootstrapping is done, Conductor will resolve all the new hostnames to the dynamic IP given to the instance by the cloud provider and create another list grain on all cluster members in each cluster created. This is the cluster.members.ip grain. This is also very useful in salt role configuration states.***

As noted above this matrix, a full list of all the salt grains Conductor is creating and using will be available in another document.