Salt - Conductor Orchestration hooks

Overview

Within the Conductor (orchestration runner extension for Saltstack providing end to end provisioning deployment and configuration), we have implemented hooks. These hooks provide the access points to inject salt state configuration to alter the standard orchestration workflow that takes place when provisioning and deploying systems to AWS using the conductor tool.

When you think about orchestration, think about sequence of events that take place between a start point and end point. The start being user invocation or some automated trigger, the end being ready state of system.

Traditional installers such as MSI packages, Setup.exe, Python eggs, RPM's, DEB packages all have a sequence of actions that take place. These actions change based on conditions.

With configuration management framework and deployment there are a set of default, 'built-in' actions, that perform all the steps needed to go from invocation to complete (systems running and excepting traffic). The hooks provided in the Saltstack orchestration runner allow configuration owners to change the flow based on conditions.

When using an enterprise wide end to end CM systems, consumers of that system have different requirements based on product being delivered and their team/group specific use cases to go from invoke to complete. Consumers could be users, groups, teams, divisions or organizations.

Hooks are used as injection points to provide the flexibility of each consumers needs and requirements. They are also used to provide common "out of band" actions during a sequenced orchestration that may be required no matter who the consumer is invoking and what is being delivered all with consistent tooling.

This page describes the use of orchestration hooks in a Saltstack Automation and CM framework, how to configure and use them to perform various injected tasks when provisioning and deploying products, services and systems within a shared CM automation framework.

The Hooks described in the main sections are in the context of the 'create' Conductor runner action. The Conductor has four action types, create, destroy, upsize and downsize. At the time this document was written, hooks were not implemented in any action type other than 'create'. As support for orchestration hooks are implemented in the other 3 Conductor action types, new addendum will be added to the bottom of this page. Most of what's documented in the main section will also apply to the hooks in other action types with any differences noted in the addendum.

In this document there are references to a specific configuration model that is being used and will have it's own set of documentation. In short, when you see something that mentions roles, products, teams, groups systems, or any Saltstack specific terminology consider these descriptor definitions: Configuration Management Terms and Definitions

End to End Delivery Orchestration

End to end delivery orchestration is referred to as the push button process of going from configuration data to fully running systems. Every organization and team/product group may have different requirements when it comes to end to end delivery of products and services to the cloud.

However, there is a common set of things that need to happen regardless of any team/group or organization specifics.

The Salt Conductor runner's main purpose is to facilitate these common tasks with automation, but driven by data. As such, there is some 'baked in' functionality as well as hooks to provide insertion point for organization and/or team/group specific functionality.

Most all delivery automation needs the following common actions:

- **read data to learn what's needed**. *This can be input command line arguments, instruction file input or from data, I.E. configuration. The conductor is set of Saltstack runner extension modules, therefore using Salt's native configuration (pillar) tree (The Pillar Tree) is the primary source of data.*
- **generate/retrieve configuration**. *The conductor auto-generated all aws cloud configuration based on the pillar data model. The cloud configuration files are dynamic, therefore they are ephemeral.*
- **provision create cloud objects**. *virtual machines, networking components etc... required to host software being delivered.*
- **bootstrap new cloud instances**. *This could be installing a management client like salt, puppet, func, chef etc..., or some other custom bootstrap to get instance ready for configuration.*
- **apply configuration**. *Install software, start services or any other action needed to get new instances ready to perform it's role.*

These are just the high level steps needed in pretty much any cloud delivery scenario.

Of course more is needed to provide assurance that everything went as expected. So things like pre and post logging and notifications, post configuration system verification (tests etc...).

Looking at things in more detail, we also need logic to avoid conflicts within the provisioning, like duplicate names, ways to identify systems within large environment and a whole host of things that could be problematic. We also want as much dynamic behavior in automation as possible. I.E. we don't hard code instance names, endpoints etc... This is where it becomes more and more important to model out a system that can scale and provides all the ability for automation to change or adapt during the end to end orchestration, which is why hooks are available.

Taking the above common actions, and adding hooks that all for specific changes to the common flow based on environment, productgroup/team or machine roles, you can see we have much more flexibility:

- read data to learn what's needed
- generate/retrieve configuration
- **pre-provision hook (common)**. Environment scoped insertion point.
- **pre-provision hook (group)**. Product Group/Team scoped insertion point.
- **pre-provision hook (role)**. Role/Machine type scoped insertion point.
- provision create cloud objects
- bootstrap new cloud instances
- **pre-configuration hook (common)**. Environment scoped insertion point.
- **pre-configuration hook (group)**. Product Group/Team scoped insertion point.
- **pre-configuration hook (role)**. Role/Machine type scoped insertion point.
- apply configuration
- **post-configuration hook (common)**. Environment scoped insertion point.
- **post-configuration hook (group)**. Product Group/Team scoped insertion point.
- **post-configuration hook (role)**. Role/Machine type scoped insertion point.

There are other baked in tasks that will occur in the Conductor regardless of whether any hook is enabled or not. One example is if you provision a cluster (a role) of instances, the automation will automatically create a salt grain on each member of the cluster listing all the cluster members. In addition, once the cloud provisioning of the instance is done, the conductor will discover all the ip addresses and create another salt grain on each with the name:ip address of all members or the same cluster. This is very useful in that immediately configuration can be applied to all members that may have a requirement to know all other members, such as creating a local configuration file on each that is used for inner cluster communications. This is a very common behavior in clustered technologies such as Consul, mongodb and many others.

Three Tier Approach to Hooks

As noted through this document, hooks have been provided following a three tier configuration approach. Each of the the tiers also has 3 locations (pre-prov, pre-config, post-config). So it's 3x3 =9 possible hooks can be enable, each with an open ended possibility of actions that it can perform.

- common
- group
- role

Common - these are the environment scoped hooks. If enabled, they would run each time anything is provisioned in the environment

Group - these are product group/team scoped hooks. If enabled, they would run each time anything is provisioned for the corresponding product group

Role - these are product/role scoped hooks. If enabled, they would run each time any role is provisioned for the corresponding product group and role

See next section for example use cases for each hook.

Hook usage examples

Within the provisioning and deployment orchestration, hooks are available at three locations as noted above. We may implement more locations in the future, but for now the hooks have been implemented in the 3 most common places during a sequenced end to end delivery of product and/or services.

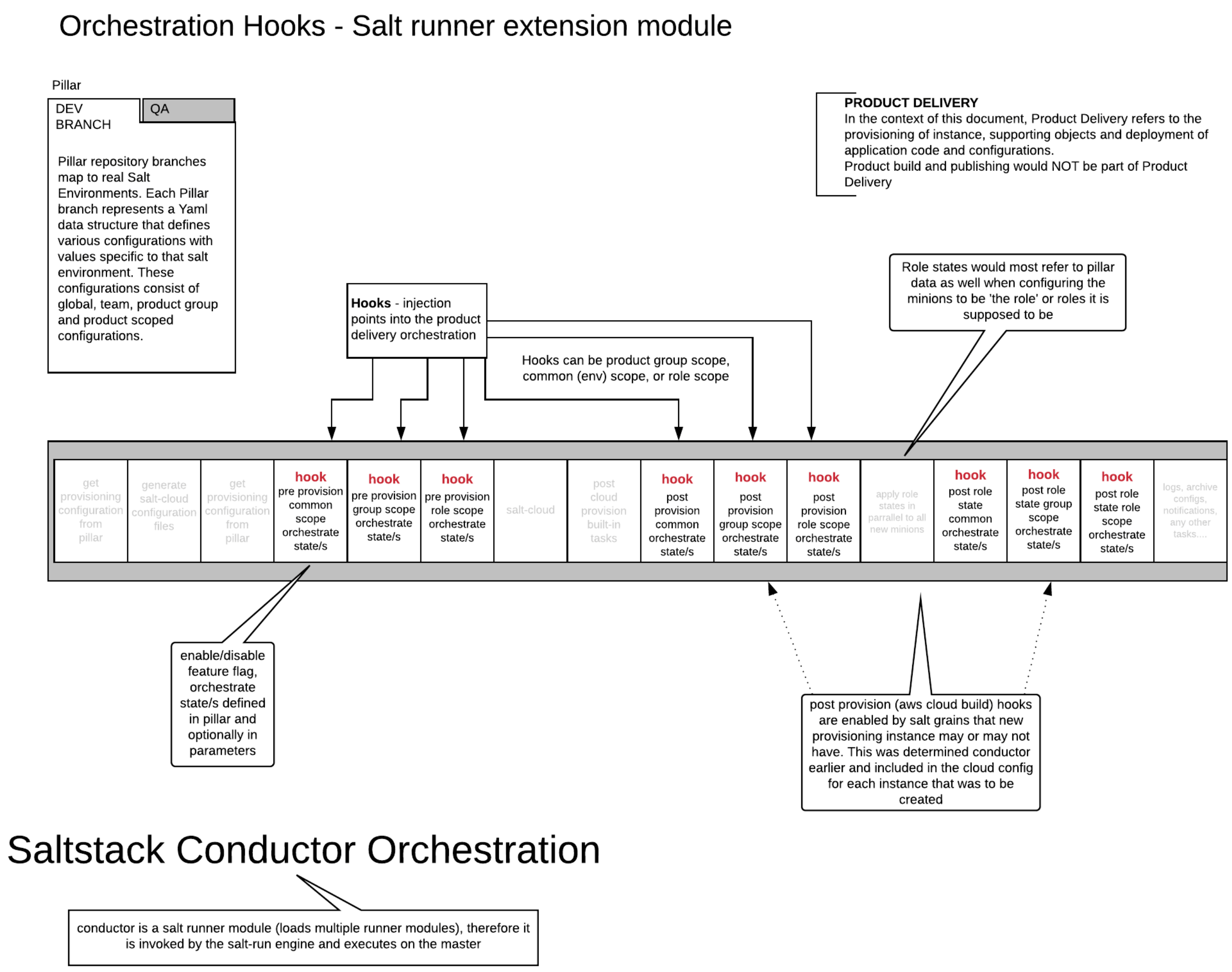
Each hook has a toggle enable/disabled flag in the Pillar configuration model. Depending on the location of the flag in the model, different people may be responsible for enabling and disabling as needed.

Here's some hypothetical use cases where each hook may be needed:

- **pre-provision hook (common)**. Environment scoped insertion point
  - send environment wide administrative alert to slack channel, email, logs etc..
  - trigger some environment wide event, maybe blocking event etc... while new machine are provisioned
  - change management
- **pre-provision hook (group)**. Product Group/Team scoped insertion point.
  - send group/team wide administrative alert to slack channel, email, logs etc..
  - trigger some team wide event, blocking event etc... such as code checkin while new machine are provisioned
  - change management
- **pre-provision hook (role)**. Role/Machine type scoped insertion point.
  - any of the above example could apply
  - update a current status webpage, service discovery
- **pre-configuration hook (common)**. Environment scoped insertion point, post provision.
  - update/retrieve from service discovery
  - status notifications
  - add new instance to DNS
- **pre-configuration hook (group)**. Product Group/Team scoped insertion point, post provision.
  - any of the above would also be examples
  - team specific status update page
- **pre-configuration hook (role)**. Role/Machine type scoped insertion point, post provision.
  - any of the above would apply
  - run some cluster preparation configuration
  - any role scope pre configuration actions needed such as adding additional salt grains etc..
- **post-configuration hook (common)**. Environment scoped insertion point.
  - change management
  - status updates and notifications
  - log archiving
  - service discovery
- **post-configuration hook (group)**. Product Group/Team scoped insertion point.
  - any of the above could also apply
  - product/team status page updates
  - add to monitoring
- **post-configuration hook (role)**. Role/Machine type scoped insertion point.
  - any of the above could apply
  - add to load balancer
  - enable traffic/services

Visual Representation of Hooks

The following diagram shows the locations and types of hooks available to anything using the Salt Conductor


Orchestration Hooks - Salt runner extension module
Pillar
DEV BRANCH
QA
Pillar repository branches map to real Salt Environments. Each Pillar branch represents a Yaml data structure that defines various configurations with values specific to that salt environment. These configurations consist of global, team, product group and product scoped configurations.
PRODUCT DELIVERY
In the context of this document, Product Delivery refers to the provisioning of instance, supporting objects and deployment of application code and configurations.
Product build and publishing would NOT be part of Product Delivery
Hooks - injection points into the product delivery orchestration
Hooks can be product group scope, common (env) scope, or role scope
Role states would most refer to pillar data as well when configuring the minions to be 'the role' or roles it is supposed to be
get provisioning configuration from pillar
generate salt-cloud configuration files
get provisioning configuration from pillar
hook pre provision common scope orchestrate state/s
hook pre provision group scope orchestrate state/s
hook pre provision role scope orchestrate state/s
salt-cloud
post cloud provision built-in tasks
hook post provision common orchestrate state/s
hook post provision group scope orchestrate state/s
hook post provision role scope orchestrate state/s
apply role states in parrallel to all new minions
hook post role state common orchestrate state/s
hook post role state group scope orchestrate state/s
hook post role state role scope orchestrate state/s
logs, archive configs, notifications, any other tasks....
enable/disable feature flag, orchestrate state/s defined in pillar and optionally in parameters
post provision (aws cloud build) hooks are enabled by salt grains that new provisioning instance may or may not have. This was determined conductor earlier and included in the cloud config for each instance that was to be created
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

Pillar data model representation of hooks

Here is the pillar configuration layout for hooks using TeamA as example. TeamB would follow the same pattern as TeamA, but in it's own pillar namespace. Common is environment scope and applies to all Teams.

The file path to the file holding the pillar hook configuration is in bold. The actual pillar path (how salt realizes the key value pair), is italicized in red. In real scenario, there would be a lot more configuration files in the pillar tree, we are only showing the ones related to hooks in this example.

Pillar Tree is 'the available' pillars with the entire hierarchy of the repository (or file system if that is where you are storing Pillar data). To make a pillar 'available', Saltstack uses the top.sls statefile. This is out of scope for this document. More Saltstack specific documentation/training will be needed to fully understand Pillar, but for this topic, just consider the Pillar tree to be the entire git repository or repositories being used to store our pillar data configuration.

Pillar Root is simply the top of the pillar tree. Again, this can changed and not be the first directory in the git repos, but that's for another more detailed Saltstack Pillar discussion.

The 'state:' key in the hook yaml documented below can be a list, or single state.

So in theory each of the 9 hooks available can actually represent an unlimited number of configuration file (salt state file) execution.

'/' = pillar root

Common hooks

A salt environment maps to one branch in pillar repo

```
/config/common.sls

config.common:
  hooks:
    pre-provision-orchestration:
      state: ['orch.common.pre-provision'] <---- any state file that 
would implement actions, example located in state tree 
/orch/common/pre-provision.sls
      enable: False
    pre-startup-orchestration:
      state: ['orch.common.pre-startup'] <---- any state file that would 
implement actions, example located in state tree 
/orch/common/pre-startup.sls
      enable: False
    post-startup-orchestration:
      state: ['orch.common.post-startup'] <---- any state file that 
would implement actions, example located in state tree 
/orch/common/post-startup.sls
      enable: True
```

link to example state file
orch.common.pre-provision

orch.common.pre-startup

orch.common.post-startup

In salt states files, jinja maps as well as the conductor code we would realize the hook pillar items using the following syntax

config.common:hooks:pre-provision-orchestration:state

config.common:hooks:pre-provision-orchestration:enable

config.common:hooks:pre-startup-orchestration:state

config.common:hooks:pre-startup-orchestration:enable

config.common:hooks:post-startup-orchestration:state

config.common:hooks:post-startup-orchestration:enable

Product Group/TeamA hooks

A salt environment maps to one branch in pillar repo

```
/config/teamA/init.sls

teamA.role:
  hooks:
    pre-provision-orchestration:
      state: ['orch.teamA.pre-provision'] <---- any state file that
would implement actions, example located in state tree
/orch/teamA/pre-provision.sls
      enable: False
    pre-startup-orchestration:
      state: ['orch.teamA.pre-startup'] <---- any state file that would
implement actions, example located in state tree
/orch/teamA/pre-startup.sls
      enable: False
    post-startup-orchestration:
      state: ['orch.teamA.post-startup'] <---- any state file that would
implement actions, example located in state tree
/orch/teamA/post-startup.sls
      enable: True
```

link to example state file
orch.teamA.pre-provision

orch.teamA.pre-startup

orch.teamA.post-startup

In salt states files, jinja maps as well as the conductor code we would realize pillar items using the following syntax

config.teamA.role:hooks:pre-provision-orchestration:state

config.teamA.role:hooks:pre-provision-orchestration:enable

config.teamA.role:hooks:pre-startup-orchestration:state

config.teamA.role:hooks:pre-startup-orchestration:enable

config.teamA.role:hooks:post-startup-orchestration:state

config.teamA.role:hooks:post-startup-orchestration:enable

Role scoped hooks (these are specific to each product group)

A salt environment maps to one branch in pillar repo

```
/config/teamA/init.sls

teamA.role:
  productX:      <---- this represents a team/product group defined role
    hooks:
      pre-provision-orchestration:
        state: ['orch.teamA.pre-prov.productX'] <---- any state file
that would implement actions, example located in state tree
/orch/teamA/pre-prov/productX.sls
        enable: False
      pre-startup-orchestration:
        state: ['orch.teamA.pre.productX'] <---- any state file that
would implement actions, example located in state tree
/orch/teamA/pre/productX.sls
        enable: False
      post-startup-orchestration:
        state: ['orch.teamA.post.productX'] <---- any state file that
would implement actions, example located in state tree
/orch/teamA/post/productX.sls
        enable: True
```

link to example state file
orch.teamA.pre-prov.productX

orch.teamA.pre.productX

orch.teamA.post.productX

In salt states files, jinja maps as well as the conductor code we would realize role scoped pillar items using the following syntax, where productX represents a role.

config.teamA.role:productX:hooks:pre-provision-orchestration:state

config.teamA.role:productX:hooks:pre-provision-orchestration:enable

config.teamA.role:productX:hooks:pre-startup-orchestration:state

config.teamA.role:productX:hooks:pre-startup-orchestration:enable

config.teamA.role:productX:hooks:post-startup-orchestration:state

config.teamA.role:productX:hooks:post-startup-orchestration:enable

Addendum - Destroy action hooks

The implementation of Conductor orchestration hooks for the 'destroy' action type is slightly different than that of 'create' action described in the main sections of this document.

Refer to the Conductor action types HERE.

Destroy action hooks are available as a 2x3 injection point design. In short, the same 3 different scoped hooks are supported. Environment, product group and role. However there are only 2 points in the workflow of 'destroy' where these three hooks are available, thus 2x3. The placement is as follows:

- user invoke destroy action in Conductor specifying either by role, by node, or by one of the support grains. Refer to the details of 'destroy' or terminate instance in Conductor HERE.
- build list of matching instances
- **pre-destroy hook (common)**. Environment scoped insertion point.
- **pre-destroy hook (group)**. Product Group/Team scoped insertion point.
- **pre-destroy hook (role)**. Role/product type scoped insertion point.
- destroy instances and wait for terminated status on all
- **post-destroy hook (common)**. Environment scoped insertion point.
- **post-destroy hook (group)**. Product Group/Team scoped insertion point.
- **post-destroy hook (role)**. Role/product scoped insertion point.
- exit Conductor

Pillar tree model representation of destroy hooks

In salt states files, jinja maps as well as the conductor code we would realize the hook pillar items using the following syntax (all 3 scoped hooks shown environment, group and role)

ENVIROMENT SCOPE

config.common:hooks:pre-destroy-orchestration:state

config.common:hooks:pre-destroy-orchestration:enable

config.common:hooks:post-destroy-orchestration:enable

config.common:hooks:post-destroy-orchestration:state

PRODUCT GROUP SCOPE

config.PRODUCTGROUP.role:hooks:pre-destroy-orchestration:enable

config.PRODUCTGROUP.role:hooks:pre-destroy-orchestration:state

config.PRODUCTGROUP.role:hooks:post-destroy-orchestration:enable

config.PRODUCTGROUP.role:hooks:post-destroy-orchestration:state

ROLE SCOPE

config.PRODUCTGROUP.role:PRODUCT:hooks:pre-destroy-orchestration:enable

config.PRODUCTGROUP.role:PRODUCT:hooks:pre-destroy-orchestration:state

config.PRODUCTGROUP.role:PRODUCT:hooks:post-destroy-orchestration:enable

config.PRODUCTGROUP.role:PRODUCT:hooks:post-destroy-orchestration:state

State tree model representation of destroy hooks

Keep in mind that the Saltstack design we are implementing will follow a Model

Pillar tree (model based) file system

```
ENVIRONMENT SCOPED HOOKS

/config/common.sls
config.common:
  hooks:
    pre-destroy-orchestration:
      state: ['orch.common.pre-destroy'] <---- any state file that would
implement actions
      enable: False
    post-destroy-orchestration:
      state: ['orch.common.post-destroy'] <---- any state file that
would implement actions
      enable: True


PRODUCT GROUP SCOPED HOOKS (EXAMPLE PRODUCT GROUP)

/config/PRODUCTGROUP/init.sls
PRODUCTGROUP.role:
  hooks:
    pre-destroy-orchestration:
      state: ['orch.PRODUCTGROUP.pre-destroy']
      enable: False
    post-startup-orchestration:
      state: ['orch.PRODUCTGROUP.post-destroy']
      enable: True

ROLE SCOPED HOOKS (EXAMPLE PRODUCT GROUP AND ROLE)

/config/PRODUCTGROUP/init.sls

PRODUCTGROUP.role:
  PRODUCTGROUP:
    hooks:
      pre-destroy-orchestration:
        state: ['orch.PRODUCTGROUP.pre.destroy.PRODUCT']
        enable: False
      post-destroy-orchestration:
        state: ['orch.PRODUCTGROUP.post.destroy.PRODUCT']
        enable: True
```

Addendum - upsize and down size action hooks

...coming soon