Salt - Conductor State Design

Saltstack States

A State is a first class citizen and the most fundamental component in the Saltstack ecosystem. The 'State' is a file with extension.sls, it resides in a 'state tree' which can be a source code repository or backend file system. The state file is where you define action and configurations to occur on a minion. The sls file in salt does not execute like a script or program. It is rendered into a structure yaml (or other rendering) data, then compiled by the salt engines and processed accordingly based on targets. This rendered data in a processed state file is a declarative sequence of steps needed to apply 'a state' to a minion. A minion is simply an instance/vm/managed node in Saltstack and when targeted a state executes on the minion side.

The state file (or simply the state) can consist of unlimited definitions that can perform any task including installing software, modifying, creating removing files, services etc... Basically any configuration that can happen manually can be defined in a salt state. The state file can contain Jinja or other template based rendering language. Templating states is how you achieve re-usability, parameterization and other programmatic behaviors.

The action contained in a state file can be calls to 'state modules, 'execution modules' or other internal saltstack functions.

State module are the outer edge more human friendly descriptions of actions that are coded at a lower layer in execution modules. Saltstack comes with many built-in state and execution modules.

As a consumer of Saltstack, you will create your own states that implement the details of your specific server/instance needs. These states can be thought of as 'Role states' or 'instance/machine states'.

Within these 'role' states, you can define tasks that can include other states, call into state modules or execution modules each either packaged with Saltstack or custom state and execution modules you develop and make available in the Salt 'state tree'. There are other logical categorizations of Salt states in a large enterprise wide system.

State actions (tasks) are all compiled together into a python dictionary whenever a Saltstack engine is invoked and refers to the Salt Tree. Saltstack is built on python and thus all tasks must be unique in through out the entire state tree (for the most part). Salt can be configured to use multiple sources for state and be configured to use a variety of merge strategies. So it's possible to have multiple state task id's that are the same. But it's cleaner and more understandable to keep all tasks unique by using a namespace or other similar naming convention. Salt uses a dotted directory namespace method in it's own methods.

For example: A state file call foo.sls resides in a salt state repository salt://org/product/foo.sls.

In this example 'org' is at the root of the salt 'state tree' and to apply that state to a minion, you would specify org.product.foo in the full salt command. A task defined in foo could be yaml:

install the foo app:

```
pkg.installed:

     - name: foo
```

To prevent conflicts if another team writes a salt state of their own, you could do this:

```
install.org.product.foo:

  pkg.installed:

    - name: foo
```

The leftmost statement in the yaml is considered the 'task' or the 'task ID' in the salt subsystems and would be the key in the python dictionary construct when rendered. As with any python dictionary, you cannot have two dict entries with the same key.

Salt has another built-in state types called, Orchestration states. These states run on the Salt Master and not on the minion. They are invoked using a different engine, the salt-run engine, instead of the salt engine.

These are very useful as orchestration states provide a way to orchestrate a logical sequence of events (states being applied) to a group of minions with the ability to coordinate the entire process, I.E orchestration.

The orchestration state syntax is basically the same as any other state, however, since they execute on the master, they do have modules available to them that regular states do not. These modules allow states to be applied to targeted minions from the orchestration state runtime.

In summary, Saltstack State is a file with .sls extension that implements any number of tasks when applied to targeted minions.

In designing a scalable Saltstack enterprise management system, the better you classify your states, the better off and more long lived the design will be.

As an example here are some state types that can be implemented and used to build scalable and reusable state trees within a Saltstack environment:

Common State - a state file containing non-role specific actions. Typically things that you may want to apply to all targeted minions either directly or as includes in other states, like security patched company policybtype stuff, common tools and utilities etc...

Shared State - Similar to common state, but are more technology specific and get applied to targeted minions mainly as includes in other states like Role states. These are fully generic and should be re-usable by any product group or team and should not be applied directly. These states typically have default values that can be overwritten when included in an implemented state. See implemented states.

Implemented states - these states are typically product group/team specific 'implementations' of shared states. Shared states would be defined as includes, and will have default config values that can be overwritten as needed by the product group/team that is 'consuming' the shared state. Implemented state can simply consume a shared state and nothing else, or it can contain other product group/team specific actions that are required when applying that shared state.

For example, we have thirdparty_AppA shared salt state.

Team x needs to rollout AppA as part of their suite with maybe some override values, nothing else. So thier 'implementation' of AppA is just a basic include in thier own state file, salt://org/TeamX/AppA.sls state file.

Team y needs to rollout AppA as part of thier suite with maybe some override values and additional activeMQ on the minion getting AppA state. Team y implementation of AppA is a state file, salt://org/TeamY/AppA.sls that contains the include of AppA and the tasks (another include) to also install ActiveMq.

To apply the TeamX state: salt 'miniontarget' state.sls org.TeamX.AppA

To apply the TeamX state: salt 'miniontarget' state.sls org.TeamY.AppA

Off topic, but grains can be used (and should be used) in targeting minions. Grains are metadata that exposes a ton of information to the Saltstack ecosystem and are to be configured on the minion. So in the above example, if you have to apply these states to many minions and you either didn't know the full minion name, or the list was too big, you could use grains to create a compound target and be much more precise with less information at hand.

To apply the TeamX AppA state to only thier minions: salt -C '* and G:role:TeamX.AppA' state.sls org.TeamX.AppA

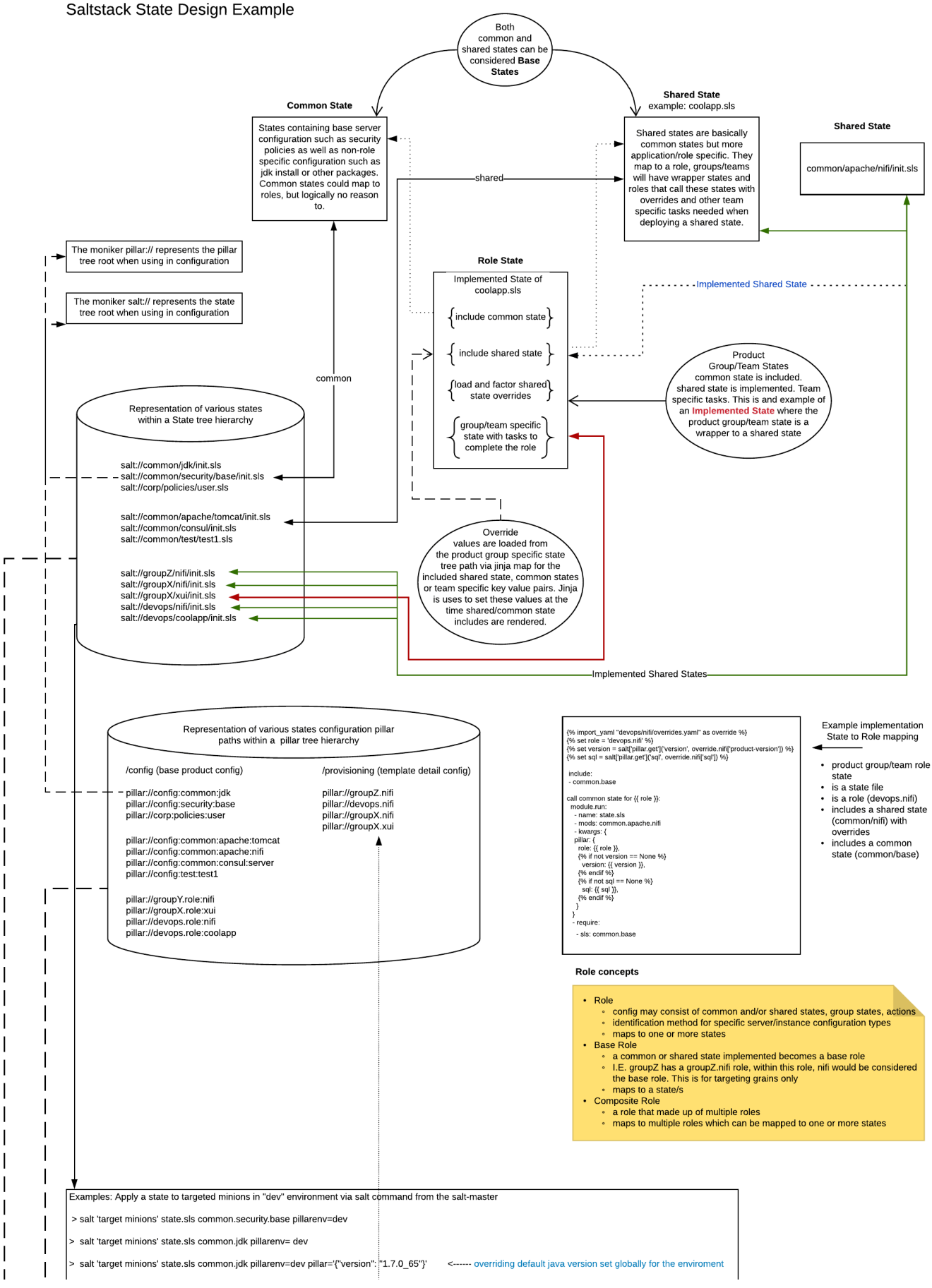
To apply the TeamY AppA state to only thier minions: salt -C '* and G:role:TeamY.AppA' state.sls org.TeamY.AppA

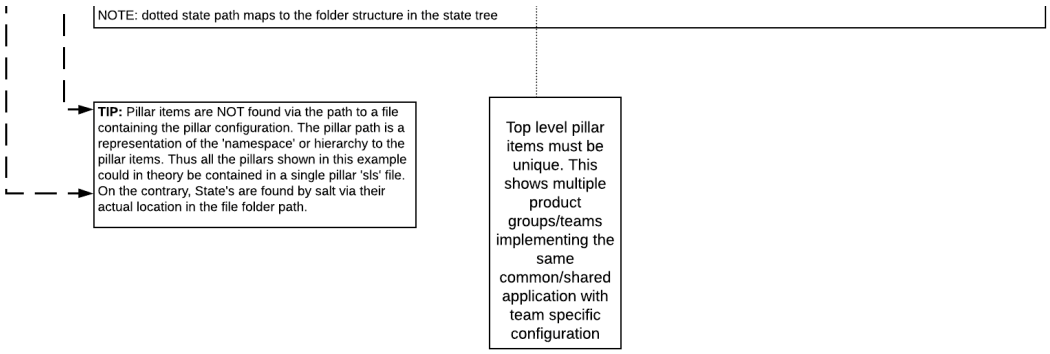
Writing and consuming states in Salt Conductor Framework

Below is a diagram describing the different "types" of states and concepts within the Salt delivery automation framework.

Refer to the Terms and Definitions page for a detailed description of the various state types, roles and how the two are connected.

Saltstack State Design Example





Example of different State implementations

As noted in the above sections there are some differences to how salt states are implemented. In terms of Saltstack, a state is a state except for the Orchestration states which execute on the salt master.

In this or other implementations of Salt, a common practice is to design your strategy around salt states to be scalable, re-usable and easily distinguishable from other states. The Conductor framework makes full use of these concepts, and thus we have different state types (or use cases in other words).

In addition to the diagram above which highlights the different state implementations, below will show examples of some of the more common use cases for salt states.

Example role state

A role state in the context of Conductor framework isn't anything more that a state in a location in the state tree under the product group hierarchy. Role states are thus developed on a per product group basis and meant to map to a server function. A server function could be an application, a webapp hosted, a service, or any thing that represents a distinguishable configuration. For example a simply file configuration such as hosts file deployed to all new instance would NOT represent a role state. That would simply be a utility state or a product state. When decided what should be consider as "role state", think in terms of "can this state configuration a server/instance that can perform a function in and of itself". If yes, then that should be a role state. Another example of something that would not be a role state is a java/jdk install state. By itself, java would not perform any useful function on a server/instance. However when you install a web application on that server that needs java, it would most likely have some java settings it needs. Therefore the web application would have a role state that would have the specific java configuration options it requires. This could all be in one state file, or broken out depending on what makes more sense.

If a role state is simply including a shared common role, it's still a role state.

This example shows a salt state that is a role state for the 'salty' product group and is developed to assure its tasks only run on a minion that is meant to have that role (using salt grains). This particular state is also including two other states to install base common configuration and the shared state. This state does not pass parameters (aka dynamic pillar) to either included state, but does specify a require to assure one state applies before the other.

Example role state designed to be safe by validating role grains

```
{% set role = 'salty.nifi' %}
{% set iscomposite = salt['grains.get']('composite.role', None) %}
{% if (grains['role'] == role) or ((not iscomposite == None) and (role
in grains['composite.role'])) %}
{{ role }} information:
  cmd.run:
    - names:
      - echo deploying {{role}}
include:
  - common.base
  - common.apache.nifi

apply common state for {{ role }}:
  module.run:
    - name: state.sls
    - mods: common.apache.activemq
    - require:
      - sls: common.base
{% endif %}
```

Example role state implementing a shared/common product

This example shows a salt state that is a role state for the 'devops' product group but includes a common shared state. This particular state is also including another state to install base common configuration. Jinja is used for multiple actions such as getting pillar configuration, passing parameters (aka dynamic pillar) to the common state.

```
{% import_yaml "devops/activemq/overrides.yaml" as override %}
{% set role = 'devops.activemq' %}
{% set version = salt['pillar.get']('version',
override.activemq['product-version']) %}

include:
  - common.base

call common state for {{ role }}:
  module.run:
    - name: state.sls
    - mods: common.apache.activemq
    - kwargs: {
          pillar: {
            role: {{ role }},
{% if not version == None %}
            version: {{ version }},
{% endif %}
          }
      }
    - require:
      - sls: common.base
```

Example utility/product state

Below is are basic examples of a utility and product state. Utility as well a product state are generic terms used to describe states that do not map to role and/or server function

utility state example

```
{% set foo = salt['pillar.get']('foo', 'nothing passed') %}

create the file:
  file.touch
    - name: /var/log/foo.empty
    - unless: test -f /var/log/foo.empty

test update that file now:
  file.append:
    - name: /var/log/foo.empty
    - text: |
        writing dynamic pillar to file {{ foo }}
    - onlyif: test -f /var/log/foo.empty
```

The above will create an empty file if it doesn't exist then write a pillar parameter value to it if it exists

The output of running this state twice on a minion would look like this

more advanced utility state

```
{% set service = salt['pillar.get']('service', None) %}
{% set bootstart = salt['pillar.get']('bootstart', None) %}
{% if service == None %}
exception_MISSING_SERVICE_PARAMETER:
  module.run:
    - name: test.exception
    - message: service is a dynamic pillar parameter for this state
{% else %}
  {% if 'managed-services' in salt['pillar.get']('global') %}
    {% set managed_services =
salt['pillar.get']('global:managed-services', {}) %}
  {% else %}
    {% set managed_services = {} %}
  {% endif %}
  {% for k,v in managed_services.iteritems() %}
    {% if k == service %}
start {{v}} service:
      {% if '/' in v %}
  cmd.run:
    - name: |
        {{v}} start
      {% else %}
  service.running:
    - name: {{v}}
    - sig: {{k}}
    - enable: {{bootstart}}
      {% endif %}
    {% endif %}
  {% endfor %}
  # IF NOT MANAGED IN PILLAR
  {% if not service in salt['pillar.get']('global:managed-services') %}
start {{service}} service:
    {% if '/' in service %}
  cmd.run:
    - name: |
        {{service}} start
    {% else %}
  service.running:
    - name: {{service}}
    - sig: {{service}}
    - enable: {{bootstart}}
    {% endif %}
  {% endif %}
{% endif %}
```

The above example is a generic service management state. It uses some expected pillar data, but also has a default behavior if not found.

Another thing to note about salt states. In the above example, we are putting some error handling around one of the tasks. This is a good way to fail a state prematurely. Without handling around a task like this, salt behavior will be to continue through the entire state and report results either all success, all failed or mixed results. It's convenient to not use this default behavior when there is critical dependency actions, or you want to have better accuracy in the output for parsing.

product state example

```
# ##############################
# GENERIC JAVA JDK INSTALL STATE
# ##############################

{% from "common/map.jinja" import common with context %}

{% set product = 'java' %}
{% set version = salt['pillar.get']('version',
common.oracle.java['product-version']) %}
{% if version == None %}
exception_no_version_{{product}}:
  module.run:
    - name: test.exception
    - message: version not found in defaults or pillar for {{product}}
{% endif %}

{% if common.oracle.java.srcpath == None %}
exception_null_srcpath_{{product}}:
  module.run:
    - name: test.exception
    - message: default srcpath not found in defaults for {{product}}
{% endif %}

{% set supportedversions = salt['pillar.get']('global:oracle:' + product
+ ':supported-versions', {}) %}

{% for k,v in supportedversions.iteritems() %}
  {% if version == k %}
    {% set thepackage = v['package'] %}
    {% set version_dir = '/opt/java/' + version %}
    {% set uri = 'https://' + common.artifactory.user + ':' +
common.artifactory.token + '@' + common.artifactory.host +
common.oracle.java.srcpath + version + '/' + thepackage %}

fetch {{product}}:
  cmd.run:
    - name: |
        mkdir -p /opt/java/jdk{{ version }}
        curl {{ uri }} -o /opt/java/jdk{{ version }}/{{ thepackage }}
        cd /opt/java/jdk{{ version }}
        rpm -ivh {{ thepackage }}
        ln -s /opt/java/jdk{{ version }}/ /opt/java/latest
    - unless:
      - test -d /opt/java/jdk{{ version }}
      - ls /opt/java/latest

  {% endif %}
{% endfor %}
```

The example above will evaluate some pillar either from dynamic parameter or pillar source, establish a uri to the install artifact, then proceed to get the artifact and install java jdk if it doesn't already exist. There are also a couple conductor framework specific things going on such as the check in pillar for a list of supported version of that product. This can be configured to assure only known support or good versions are deployed throughout the environment.

Example orchestration state

orchestration state example

```
# pre startup environment wide orchestration state. gets run when a new
instance is brought online
# REQUIRED PARAMETERS:
# target-minion - should be a valid minion name glob, could be in the
compound form as well. example: 'server1 or server2 or server3'
# cpid - cloud provision id

{% set minion_target = salt['pillar.get']('target-minion', None) %}
{% set cpid = salt['pillar.get']('cpid', None) %}

{% if not minion_target == None %}

pre startup common orchestration state hook message:
  salt.function:
    - name: cmd.run
    - tgt: 'saltmaster'
    - arg:
      - echo test pre startup common orchestrate state, nodes
{{minion_target}}

pre startup common orchestration state hook:
  {% set target = "'" + minion_target + "'" %}
  salt.state:
    - tgt: {{ target }}
    - tgt_type: list
    - sls:
      - common.test.ping
    - pillar: {
        target-minion: {{target}}
      }
{% endif %}
```

Above is a basic example of an orchestration state. *As noted, these run on the salt master only*.

This one in particular is taken from the example orchestration hooks that are implemented in the conductor tool.

> The important thing to remember about orchestration states is that they run on the salt master and have only two module options. "salt.state" and "salt.function". These two modules are the wrappers around applying states and execution modules to salt minions.

Using Jinja in states

Since states are files that are templates for the most part. They are rendered. In Salt, the default renderer is Jinja. Jinja is a python templating language and is used to make states parameterized and variable.

States are unlike scripts due to the rendering required. You may be tempted to think about a procedural script when developing salt states, but due to the rendering, you have to think of things slightly different in addition to the Jinja language not be an exact implementation of python. Methods and functions and such are a bit different when used in Jinja in a salt state.

> Jinja or templating is NOT used in Salt Pillar. Jinja is implemented in States only.

This is not meant to be a full tutorial on Jinja, but there are a couple things to point out.

When adding conditional logic to salt states with jinja, and especially looping logic, keep in mind that the output of of the processed state is a rendered document. The rendered document is yaml in Salt's case. And as with yaml formatting in salt state and pillar tree, you cannot have multiple identifying config of the same name.

To highlight this point, see the following two examples.

Example showing Jinja with invalid rendering

invalid jinja loop in salt state

```
{% set roles = [] %}
{% set getroles = salt['pillar.get']('config.common:roles') %}

{% for therole in getroles %}
  {% do roles.append(therole) %}
{% endfor %}

{% set ctr = 0 %}

{% for role in roles %}
  {% set ctr = ctr|int + 1 %}
found role in configuration:                        <--- ERROR this is
where the problem is. This yaml block will be written to the rendered
document once for
  cmd.run:                                                role in the roles
List. Rendered yaml cannot have multiple ID's of the same. Saltstack
compiles into
    - name: |                                             a dictionary, so
this will through a salt exception.
        echo role found is {{ role }}
{% endfor %}

total roles found in configuration:
  cmd.run:
    - name: |
        echo Total roles found {{ roles|length }}
```

Example showing Jinja with valid rendering

valid jinja loop in salt state

```
{% set roles = [] %}
{% set getroles = salt['pillar.get']('config.common:roles') %}

{% for therole in getroles %}
  {% do roles.append(therole) %}
{% endfor %}

{% set ctr = 0 %}

{% for role in roles %}
  {% set ctr = ctr|int + 1 %}
found role {{ role }} in configuration:         <-- The problem is fixed
here. Since the role is a jinja variable in the for loop, we can use
that in the
  cmd.run:                                          yaml task ID name so
as to avoid salt python dictionary exceptions.
    - name: |
        echo role found is {{ role }}
{% endfor %}

total roles found in configuration:
  cmd.run:
    - name: |
        echo Total roles found {{ roles|length }}
```

Jinja is extremely powerful in salt state development. State re-use and parameterization are key concepts to include in the state design when implementing an enterprise scaled solution.