CloudStack Cloud Guide

Introduction

The purpose of this section is to explain how to put Ansible modules together to use Ansible in a CloudStack context. You will find more usage examples in the details section of each module.

Ansible contains a number of extra modules for interacting with CloudStack based clouds. All modules support check mode, are designed to be idempotent, have been created and tested, and are maintained by the community.

> **Note**
>
> Some of the modules will require domain admin or root admin privileges.

Prerequisites

Prerequisites for using the CloudStack modules are minimal. In addition to Ansible itself, all of the modules require the python library `cs` https://pypi.org/project/cs/

You'll need this Python module installed on the execution host, usually your workstation.

```
$ pip install cs
```

Or alternatively starting with Debian 9 and Ubuntu 16.04:

```
$ sudo apt install python-cs
```

> **Note**
>
> cs also includes a command line interface for ad hoc interaction with the CloudStack API, for example `$ cs listVirtualMachines state=Running`.

Limitations and Known Issues

VPC support has been improved since Ansible 2.3 but is still not yet fully implemented. The community is working on the VPC integration.

Credentials File

You can pass credentials and the endpoint of your cloud as module arguments, however in most cases it is a far less work to store your credentials in the cloudstack.ini file.

The python library cs looks for the credentials file in the following order (last one wins):

- A `.cloudstack.ini` (note the dot) file in the home directory.
- A `CLOUDSTACK_CONFIG` environment variable pointing to an .ini file.
- A `cloudstack.ini` (without the dot) file in the current working directory, same directory as your playbooks are located.

The structure of the ini file must look like this:

```
$ cat $HOME/.cloudstack.ini
[cloudstack]
endpoint = https://cloud.example.com/client/api
key = api key
secret = api secret
timeout = 30
```

> **Note**
>
> The section `[cloudstack]` is the default section. `CLOUDSTACK_REGION` environment variable can be used to define the default section.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\scenario_guides\[ansible-devel][docs][docsite][rst][scenario_guides]guide_cloudstack.rst, line 59)**
>
> Unknown directive type "versionadded".

```
    .. versionadded:: 2.4
```

The ENV variables support `CLOUDSTACK_*` as written in the documentation of the library `cs`, like `CLOUDSTACK_TIMEOUT`, `CLOUDSTACK_METHOD`, and so on. has been implemented into Ansible. It is even possible to have some incomplete config in your cloudstack.ini:

```
$ cat $HOME/.cloudstack.ini
[cloudstack]
endpoint = https://cloud.example.com/client/api
timeout = 30
```

and fulfill the missing data by either setting ENV variables or tasks params:

```
---
- name: provision our VMs
  hosts: cloud-vm
  tasks:
    - name: ensure VMs are created and running
      delegate_to: localhost
      cs_instance:
        api_key: your api key
        api_secret: your api secret
        ...
```

Regions

If you use more than one CloudStack region, you can define as many sections as you want and name them as you like, for example:

```
$ cat $HOME/.cloudstack.ini
[exoscale]
endpoint = https://api.exoscale.ch/compute
key = api key
secret = api secret

[example_cloud_one]
endpoint = https://cloud-one.example.com/client/api
key = api key
secret = api secret

[example_cloud_two]
endpoint = https://cloud-two.example.com/client/api
key = api key
secret = api secret
```

> **Hint**
>
> Sections can also be used to for login into the same region using different accounts.

By passing the argument `api_region` with the CloudStack modules, the region wanted will be selected.

```
- name: ensure my ssh public key exists on Exoscale
  cs_sshkeypair:
    name: my-ssh-key
    public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    api_region: exoscale
  delegate_to: localhost
```

Or by looping over a regions list if you want to do the task in every region:

```
- name: ensure my ssh public key exists in all CloudStack regions
  local_action: cs_sshkeypair
    name: my-ssh-key
    public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    api_region: "{{ item }}"
    loop:
      - exoscale
      - example_cloud_one
      - example_cloud_two
```

Environment Variables

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\scenario_guides\[ansible-devel][docs][docsite][rst][scenario_guides]guide_cloudstack.rst, line 136)**

Unknown directive type "versionadded".

```
.. versionadded:: 2.3
```

Since Ansible 2.3 it is possible to use environment variables for domain (`CLOUDSTACK_DOMAIN`), account (`CLOUDSTACK_ACCOUNT`), project (`CLOUDSTACK_PROJECT`), VPC (`CLOUDSTACK_VPC`) and zone (`CLOUDSTACK_ZONE`). This simplifies the tasks by not repeating the arguments for every tasks.

Below you see an example how it can be used in combination with Ansible's block feature:

```
- hosts: cloud-vm
  tasks:
    - block:
        - name: ensure my ssh public key
          cs_sshkeypair:
            name: my-ssh-key
            public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

        - name: ensure my ssh public key
          cs_instance:
              display_name: "{{ inventory_hostname_short }}"
              template: Linux Debian 7 64-bit 20GB Disk
              service_offering: "{{ cs_offering }}"
              ssh_key: my-ssh-key
              state: running

      delegate_to: localhost
      environment:
        CLOUDSTACK_DOMAIN: root/customers
        CLOUDSTACK_PROJECT: web-app
        CLOUDSTACK_ZONE: sf-1
```

Note

You are still able overwrite the environment variables using the module arguments, for example `zone: sf-2`

Note

Unlike `CLOUDSTACK_REGION` these additional environment variables are ignored in the CLI `cs`.

Use Cases

The following should give you some ideas how to use the modules to provision VMs to the cloud. As always, there isn't only one way to do it. But as always: keep it simple for the beginning is always a good start.

Use Case: Provisioning in a Advanced Networking CloudStack setup

Our CloudStack cloud has an advanced networking setup, we would like to provision web servers, which get a static NAT and open firewall ports 80 and 443. Further we provision database servers, to which we do not give any access to. For accessing the VMs by SSH we use a SSH jump host.

This is how our inventory looks like:

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\scenario_guides\[ansible-devel][docs][docsite][rst][scenario_guides]guide_cloudstack.rst`, line 180)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    [cloud-vm:children]
    webserver
    db-server
    jumphost

    [webserver]
    web-01.example.com  public_ip=198.51.100.20
    web-02.example.com  public_ip=198.51.100.21

    [db-server]
    db-01.example.com
    db-02.example.com
```

```
[jumphost]
jump.example.com  public_ip=198.51.100.22
```

As you can see, the public IPs for our web servers and jumphost has been assigned as variable `public_ip` directly in the inventory.

The configure the jumphost, web servers and database servers, we use `group_vars`. The `group_vars` directory contains 4 files for configuration of the groups: cloud-vm, jumphost, webserver and db-server. The cloud-vm is there for specifying the defaults of our cloud infrastructure.

```
# file: group_vars/cloud-vm
---
cs_offering: Small
cs_firewall: []
```

Our database servers should get more CPU and RAM, so we define to use a `Large` offering for them.

```
# file: group_vars/db-server
---
cs_offering: Large
```

The web servers should get a `Small` offering as we would scale them horizontally, which is also our default offering. We also ensure the known web ports are opened for the world.

```
# file: group_vars/webserver
---
cs_firewall:
  - { port: 80 }
  - { port: 443 }
```

Further we provision a jump host which has only port 22 opened for accessing the VMs from our office IPv4 network.

```
# file: group_vars/jumphost
---
cs_firewall:
  - { port: 22, cidr: "17.17.17.0/24" }
```

Now to the fun part. We create a playbook to create our infrastructure we call it `infra.yml`:

```
# file: infra.yaml
---
- name: provision our VMs
  hosts: cloud-vm
  tasks:
    - name: run all enclosed tasks from localhost
      delegate_to: localhost
      block:
        - name: ensure VMs are created and running
          cs_instance:
            name: "{{ inventory_hostname_short }}"
            template: Linux Debian 7 64-bit 20GB Disk
            service_offering: "{{ cs_offering }}"
            state: running

        - name: ensure firewall ports opened
          cs_firewall:
            ip_address: "{{ public_ip }}"
            port: "{{ item.port }}"
            cidr: "{{ item.cidr | default('0.0.0.0/0') }}"
          loop: "{{ cs_firewall }}"
          when: public_ip is defined

        - name: ensure static NATs
          cs_staticnat: vm="{{ inventory_hostname_short }}" ip_address="{{ public_ip }}"
          when: public_ip is defined
```

In the above play we defined 3 tasks and use the group `cloud-vm` as target to handle all VMs in the cloud but instead SSH to these VMs, we use `delegate_to: localhost` to execute the API calls locally from our workstation.

In the first task, we ensure we have a running VM created with the Debian template. If the VM is already created but stopped, it would just start it. If you like to change the offering on an existing VM, you must add `force: yes` to the task, which would stop the VM, change the offering and start the VM again.

In the second task we ensure the ports are opened if we give a public IP to the VM.

In the third task we add static NAT to the VMs having a public IP defined.

> **Note**

The public IP addresses must have been acquired in advance, also see `cs_ip_address`

Note

For some modules, for example `cs_sshkeypair` you usually want this to be executed only once, not for every VM. Therefore you would make a separate play for it targeting localhost. You find an example in the use cases below.

Use Case: Provisioning on a Basic Networking CloudStack setup

A basic networking CloudStack setup is slightly different: Every VM gets a public IP directly assigned and security groups are used for access restriction policy.

This is how our inventory looks like:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\scenario_guides\[ansible-devel][docs][docsite][rst][scenario_guides]guide_cloudstack.rst, line 287)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

    [cloud-vm:children]
    webserver

    [webserver]
    web-01.example.com
    web-02.example.com
```

The default for your VMs looks like this:

```
# file: group_vars/cloud-vm
---
cs_offering: Small
cs_securitygroups: [ 'default']
```

Our webserver will also be in security group `web`:

```
# file: group_vars/webserver
---
cs_securitygroups: [ 'default', 'web' ]
```

The playbook looks like the following:

```
# file: infra.yaml
---
- name: cloud base setup
  hosts: localhost
  tasks:
  - name: upload ssh public key
    cs_sshkeypair:
      name: defaultkey
      public_key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

  - name: ensure security groups exist
    cs_securitygroup:
      name: "{{ item }}"
    loop:
      - default
      - web

  - name: add inbound SSH to security group default
    cs_securitygroup_rule:
      security_group: default
      start_port: "{{ item }}"
      end_port: "{{ item }}"
    loop:
      - 22

  - name: add inbound TCP rules to security group web
    cs_securitygroup_rule:
      security_group: web
      start_port: "{{ item }}"
      end_port: "{{ item }}"
    loop:
      - 80
```

```
      - 443

- name: install VMs in the cloud
  hosts: cloud-vm
  tasks:
  - delegate_to: localhost
    block:
    - name: create and run VMs on CloudStack
      cs_instance:
        name: "{{ inventory_hostname_short }}"
        template: Linux Debian 7 64-bit 20GB Disk
        service_offering: "{{ cs_offering }}"
        security_groups: "{{ cs_securitygroups }}"
        ssh_key: defaultkey
        state: Running
      register: vm

    - name: show VM IP
      debug: msg="VM {{ inventory_hostname }} {{ vm.default_ip }}"

    - name: assign IP to the inventory
      set_fact: ansible_ssh_host={{ vm.default_ip }}

    - name: waiting for SSH to come up
      wait_for: port=22 host={{ vm.default_ip }} delay=5
```

In the first play we setup the security groups, in the second play the VMs will created be assigned to these groups. Further you see, that we assign the public IP returned from the modules to the host inventory. This is needed as we do not know the IPs we will get in advance. In a next step you would configure the DNS servers with these IPs for accessing the VMs with their DNS name.

In the last task we wait for SSH to be accessible, so any later play would be able to access the VM by SSH without failure.