# Rackspace Cloud Guide

## Introduction

> **Note**
>
> Rackspace functionality in Ansible is not maintained and users should consider the OpenStack collection instead.

Ansible contains a number of core modules for interacting with Rackspace Cloud.

The purpose of this section is to explain how to put Ansible modules together (and use inventory scripts) to use Ansible in a Rackspace Cloud context.

Prerequisites for using the rax modules are minimal. In addition to ansible itself, all of the modules require and are tested against pyrax 1.5 or higher. You'll need this Python module installed on the execution host.

`pyrax` is not currently available in many operating system package repositories, so you will likely need to install it via pip:

```
$ pip install pyrax
```

Ansible creates an implicit localhost that executes in the same context as the `ansible-playbook` and the other CLI tools. If for any reason you need or want to have it in your inventory you should do something like the following:

```
[localhost]
localhost ansible_connection=local ansible_python_interpreter=/usr/local/bin/python2
```

For more information see :ref`Implicit Localhost <implicit_localhost>`

> **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_rax.rst`, line 35); *backlink*
>
> Unknown interpreted text role "ref".

In playbook steps, we'll typically be using the following pattern:

```
- hosts: localhost
  gather_facts: False
  tasks:
```

## Credentials File

The *rax.py* inventory script and all *rax* modules support a standard *pyrax* credentials file that looks like:

```
[rackspace_cloud]
username = myraxusername
api_key = d41d8cd98f00b204e9800998ecf8427e
```

Setting the environment parameter `RAX_CREDS_FILE` to the path of this file will help Ansible find how to load this information.

More information about this credentials file can be found at
https://github.com/pycontribs/pyrax/blob/master/docs/getting_started.md#authenticating

### Running from a Python Virtual Environment (Optional)

Most users will not be using virtualenv, but some users, particularly Python developers sometimes like to.

There are special considerations when Ansible is installed to a Python virtualenv, rather than the default of installing at a global scope. Ansible assumes, unless otherwise instructed, that the python binary will live at /usr/bin/python. This is done via the interpreter line in modules, however when instructed by setting the inventory variable 'ansible_python_interpreter', Ansible will use this specified path instead to find Python. This can be a cause of confusion as one may assume that modules running on 'localhost', or perhaps running via 'local_action', are using the virtualenv Python interpreter. By setting this line in the inventory, the modules will execute in the virtualenv interpreter and have available the virtualenv packages, specifically pyrax. If using virtualenv, you may wish to modify your localhost inventory definition to find this location as follows:

```
[localhost]
localhost ansible_connection=local ansible_python_interpreter=/path/to/ansible_venv/bin/python
```

> **Note**
>
> pyrax may be installed in the global Python package scope or in a virtual environment. There are no special considerations to keep in mind when installing pyrax.

## Provisioning

Now for the fun parts.

The 'rax' module provides the ability to provision instances within Rackspace Cloud. Typically the provisioning task will be performed from your Ansible control server (in our example, localhost) against the Rackspace cloud API. This is done for several reasons:

- Avoiding installing the pyrax library on remote nodes
- No need to encrypt and distribute credentials to remote nodes
- Speed and simplicity

> **Note**
>
> Authentication with the Rackspace-related modules is handled by either specifying your username and API key as environment variables or passing them as module arguments, or by specifying the location of a credentials file.

Here is a basic example of provisioning an instance in ad hoc mode:

```
$ ansible localhost -m rax -a "name=awx flavor=4 image=ubuntu-1204-lts-precise-pangolin wait=yes"
```

Here's what it would look like in a playbook, assuming the parameters were defined in variables:

```
tasks:
  - name: Provision a set of instances
    rax:
        name: "{{ rax_name }}"
        flavor: "{{ rax_flavor }}"
        image: "{{ rax_image }}"
        count: "{{ rax_count }}"
        group: "{{ group }}"
        wait: yes
    register: rax
    delegate_to: localhost
```

The rax module returns data about the nodes it creates, like IP addresses, hostnames, and login passwords. By registering the return value of the step, it is possible used this data to dynamically add the resulting hosts to inventory (temporarily, in memory). This facilitates performing configuration actions on the hosts in a follow-on task. In the following example, the servers that were successfully created using the above task are dynamically added to a group called "raxhosts", with each nodes hostname, IP address, and root password being added to the inventory.

```
- name: Add the instances we created (by public IP) to the group 'raxhosts'
  add_host:
      hostname: "{{ item.name }}"
      ansible_host: "{{ item.rax_accessipv4 }}"
      ansible_password: "{{ item.rax_adminpass }}"
      groups: raxhosts
  loop: "{{ rax.success }}"
  when: rax.action == 'create'
```

With the host group now created, the next play in this playbook could now configure servers belonging to the raxhosts group.

```
- name: Configuration play
  hosts: raxhosts
  user: root
  roles:
    - ntp
    - webserver
```

The method above ties the configuration of a host with the provisioning step. This isn't always what you want, and leads us to the next section.

## Host Inventory

Once your nodes are spun up, you'll probably want to talk to them again. The best way to handle this is to use the "rax" inventory plugin, which dynamically queries Rackspace Cloud and tells Ansible what nodes you have to manage. You might want to use this even if you are spinning up cloud instances via other tools, including the Rackspace Cloud user interface. The inventory plugin can be used to group resources by metadata, region, OS, and so on. Utilizing metadata is highly recommended in "rax" and can provide an easy way to sort between host groups and roles. If you don't want to use the `rax.py` dynamic inventory script, you could also still choose to manually manage your INI inventory file, though this is less recommended.

In Ansible it is quite possible to use multiple dynamic inventory plugins along with INI file data. Just put them in a common directory and be sure the scripts are chmod +x, and the INI-based ones are not.

### rax.py

To use the Rackspace dynamic inventory script, copy `rax.py` into your inventory directory and make it executable. You can specify a credentials file for `rax.py` utilizing the `RAX_CREDS_FILE` environment variable.

> **Note**
>
> Dynamic inventory scripts (like `rax.py`) are saved in `/usr/share/ansible/inventory` if Ansible has been installed globally. If installed to a virtualenv, the inventory scripts are installed to `$VIRTUALENV/share/inventory`.

> **Note**
>
> Users of :ref:`ansible_platform` will note that dynamic inventory is natively supported by the controller in the platform, and all you have to do is associate a group with your Rackspace Cloud credentials, and it will easily synchronize without going through these steps:
>
> > **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_rax.rst`, line 170); *backlink*
> >
> > Unknown interpreted text role "ref".
>
> ```
> $ RAX_CREDS_FILE=~/.raxpub ansible all -i rax.py -m setup
> ```

`rax.py` also accepts a `RAX_REGION` environment variable, which can contain an individual region, or a comma separated list of regions.

When using `rax.py`, you will not have a 'localhost' defined in the inventory.

As mentioned previously, you will often be running most of these modules outside of the host loop, and will need 'localhost' defined. The recommended way to do this, would be to create an `inventory` directory, and place both the `rax.py` script and a file containing `localhost` in it.

Executing `ansible` or `ansible-playbook` and specifying the `inventory` directory instead of an individual file, will cause ansible to evaluate each file in that directory for inventory.

Let's test our inventory script to see if it can talk to Rackspace Cloud.

```
$ RAX_CREDS_FILE=~/.raxpub ansible all -i inventory/ -m setup
```

Assuming things are properly configured, the `rax.py` inventory script will output information similar to the following information, which will be utilized for inventory and variables.


```
{
    "ORD": [
        "test"
    ],
    " meta": {
        "hostvars": {
            "test": {
                "ansible_host": "198.51.100.1",
                "rax accessipv4": "198.51.100.1",
                "rax_accessipv6": "2001:DB8::2342",
                "rax_addresses": {
                    "private": [
                        {
                            "addr": "192.0.2.2",
                            "version": 4
                        }
                    ],
                    "public": [
                        {
                            "addr": "198.51.100.1",
                            "version": 4
                        },
                        {
                            "addr": "2001:DB8::2342",
                            "version": 6
                        }
                    ]
                },
                "rax_config_drive": "",
                "rax created": "2013-11-14T20:48:22Z",
                "rax flavor": {
                    "id": "performance1-1",
                    "links": [
                        {
                            "href": "https://ord.servers.api.rackspacecloud.com/111111/flavors/performance1-1",
                            "rel": "bookmark"
                        }
                    ]
                },
                "rax_hostid": "e7b6961a9bd943ee82b13816426f1563bfda6846aad84d52af45a4904660cde0",
                "rax human id": "test",
                "rax_id": "099a447b-a644-471f-87b9-a7f580eb0c2a",
                "rax_image": {
                    "id": "b211c7bf-b5b4-4ede-a8de-a4368750c653",
                    "links": [
                        {
                            "href": "https://ord.servers.api.rackspacecloud.com/111111/images/b211c7bf-b5b4-4ede-a8de
                            "rel": "bookmark"
                        }
                    ]
                },
                "rax key name": null,
                "rax_links": [
                    {
                        "href": "https://ord.servers.api.rackspacecloud.com/v2/111111/servers/099a447b-a644-471f-87b9
                        "rel": "self"
                    },
                    {
                        "href": "https://ord.servers.api.rackspacecloud.com/111111/servers/099a447b-a644-471f-87b9-a7
                        "rel": "bookmark"
                    }
                ],
                "rax metadata": {
                    "foo": "bar"
                },
                "rax_name": "test",
                "rax name attr": "name",
                "rax_networks": {
                    "private": [
                        "192.0.2.2"
                    ],
                    "public": [
                        "198.51.100.1",
                        "2001:DB8::2342"
                    ]
                },
                "rax_os-dcf_diskconfig": "AUTO",
                "rax os-ext-sts power state": 1,
                "rax_os-ext-sts_task_state": null,
                "rax_os-ext-sts_vm_state": "active",
                "rax progress": 100,
                "rax_status": "ACTIVE",
                "rax_tenant_id": "111111",
                "rax_updated": "2013-11-14T20:49:27Z",
                "rax_user_id": "22222"
            }
        }
    }
}
```


### Standard Inventory

When utilizing a standard ini formatted inventory file (as opposed to the inventory plugin), it may still be advantageous to retrieve discoverable hostvar information from the Rackspace API.

This can be achieved with the `rax_facts` module and an inventory file similar to the following:

```
[test_servers]
hostname1 rax_region=ORD
hostname2 rax_region=ORD
```

```
- name: Gather info about servers
  hosts: test_servers
  gather_facts: False
  tasks:
    - name: Get facts about servers
      rax_facts:
        credentials: ~/.raxpub
        name: "{{ inventory_hostname }}"
        region: "{{ rax_region }}"
      delegate_to: localhost
    - name: Map some facts
      set_fact:
        ansible_host: "{{ rax_accessipv4 }}"
```

While you don't need to know how it works, it may be interesting to know what kind of variables are returned.

The `rax_facts` module provides facts as following, which match the `rax.py` inventory script:

```
{
    "ansible_facts": {
        "rax_accessipv4": "198.51.100.1",
        "rax_accessipv6": "2001:DB8::2342",
        "rax_addresses": {
            "private": [
                {
                    "addr": "192.0.2.2",
                    "version": 4
                }
            ],
            "public": [
                {
                    "addr": "198.51.100.1",
                    "version": 4
                },
                {
                    "addr": "2001:DB8::2342",
                    "version": 6
                }
            ]
        },
        "rax_config_drive": "",
        "rax_created": "2013-11-14T20:48:22Z",
        "rax_flavor": {
            "id": "performance1-1",
            "links": [
                {
                    "href": "https://ord.servers.api.rackspacecloud.com/111111/flavors/performance1-1",
                    "rel": "bookmark"
                }
            ]
        },
        "rax_hostid": "e7b6961a9bd943ee82b13816426f1563bfda6846aad84d52af45a4904660cde0",
        "rax_human_id": "test",
        "rax_id": "099a447b-a644-471f-87b9-a7f580eb0c2a",
        "rax_image": {
            "id": "b211c7bf-b5b4-4ede-a8de-a4368750c653",
            "links": [
                {
                    "href": "https://ord.servers.api.rackspacecloud.com/111111/images/b211c7bf-b5b4-4ede-a8de-a436875
                    "rel": "bookmark"
                }
            ]
        },
        "rax_key_name": null,
        "rax_links": [
            {
                "href": "https://ord.servers.api.rackspacecloud.com/v2/111111/servers/099a447b-a644-471f-87b9-a7f580e
                "rel": "self"
            },
            {
                "href": "https://ord.servers.api.rackspacecloud.com/111111/servers/099a447b-a644-471f-87b9-a7f580eb0c
                "rel": "bookmark"
            }
        ],
        "rax_metadata": {
            "foo": "bar"
        },
        "rax_name": "test",
        "rax_name_attr": "name",
        "rax_networks": {
            "private": [
                "192.0.2.2"
            ],
            "public": [
                "198.51.100.1",
                "2001:DB8::2342"
            ]
        },
        "rax_os-dcf_diskconfig": "AUTO",
        "rax_os-ext-sts_power_state": 1,
        "rax_os-ext-sts_task_state": null,
        "rax_os-ext-sts_vm_state": "active",
        "rax_progress": 100,
        "rax_status": "ACTIVE",
        "rax_tenant_id": "111111",
        "rax_updated": "2013-11-14T20:49:27Z",
        "rax_user_id": "22222"
```

```
    },
    "changed": false
}
```

## Use Cases

This section covers some additional usage examples built around a specific use case.

### Network and Server

Create an isolated cloud network and build a server

```
- name: Build Servers on an Isolated Network
  hosts: localhost
  gather_facts: False
  tasks:
    - name: Network create request
      rax_network:
        credentials: ~/.raxpub
        label: my-net
        cidr: 192.168.3.0/24
        region: IAD
        state: present
      delegate_to: localhost

    - name: Server create request
      rax:
        credentials: ~/.raxpub
        name: web%04d.example.org
        flavor: 2
        image: ubuntu-1204-lts-precise-pangolin
        disk config: manual
        networks:
          - public
          - my-net
        region: IAD
        state: present
        count: 5
        exact count: yes
        group: web
        wait: yes
        wait_timeout: 360
      register: rax
      delegate_to: localhost
```

### Complete Environment

Build a complete webserver environment with servers, custom networks and load balancers, install nginx and create a custom index.html

```
---
- name: Build environment
  hosts: localhost
  gather_facts: False
  tasks:
    - name: Load Balancer create request
      rax clb:
        credentials: ~/.raxpub
        name: my-lb
        port: 80
        protocol: HTTP
        algorithm: ROUND_ROBIN
        type: PUBLIC
        timeout: 30
        region: IAD
        wait: yes
        state: present
        meta:
          app: my-cool-app
      register: clb

    - name: Network create request
      rax_network:
        credentials: ~/.raxpub
        label: my-net
        cidr: 192.168.3.0/24
        state: present
        region: IAD
      register: network

    - name: Server create request
      rax:
        credentials: ~/.raxpub
        name: web%04d.example.org
        flavor: performance1-1
        image: ubuntu-1204-lts-precise-pangolin
        disk_config: manual
        networks:
          - public
          - private
          - my-net
        region: IAD
        state: present
        count: 5
        exact_count: yes
        group: web
        wait: yes
      register: rax

    - name: Add servers to web host group
```

```
      add_host:
        hostname: "{{ item.name }}"
        ansible_host: "{{ item.rax_accessipv4 }}"
        ansible_password: "{{ item.rax_adminpass }}"
        ansible_user: root
        groups: web
      loop: "{{ rax.success }}"
      when: rax.action == 'create'

    - name: Add servers to Load balancer
      rax_clb_nodes:
        credentials: ~/.raxpub
        load_balancer_id: "{{ clb.balancer.id }}"
        address: "{{ item.rax_networks.private|first }}"
        port: 80
        condition: enabled
        type: primary
        wait: yes
        region: IAD
      loop: "{{ rax.success }}"
      when: rax.action == 'create'

- name: Configure servers
  hosts: web
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

  tasks:
    - name: Install nginx
      apt: pkg=nginx state=latest update_cache=yes cache_valid_time=86400
      notify:
        - restart nginx

    - name: Ensure nginx starts on boot
      service: name=nginx state=started enabled=yes

    - name: Create custom index.html
      copy: content="{{ inventory_hostname }}" dest=/usr/share/nginx/www/index.html
            owner=root group=root mode=0644
```

### RackConnect and Managed Cloud

When using RackConnect version 2 or Rackspace Managed Cloud there are Rackspace automation tasks that are executed on the servers you create after they are successfully built. If your automation executes before the RackConnect or Managed Cloud automation, you can cause failures and unusable servers.

These examples show creating servers, and ensuring that the Rackspace automation has completed before Ansible continues onwards.

For simplicity, these examples are joined, however both are only needed when using RackConnect. When only using Managed Cloud, the RackConnect portion can be ignored.

The RackConnect portions only apply to RackConnect version 2.

#### Using a Control Machine

```
- name: Create an exact count of servers
  hosts: localhost
  gather_facts: False
  tasks:
    - name: Server build requests
      rax:
        credentials: ~/.raxpub
        name: web%03d.example.org
        flavor: performance1-1
        image: ubuntu-1204-lts-precise-pangolin
        disk_config: manual
        region: DFW
        state: present
        count: 1
        exact_count: yes
        group: web
        wait: yes
      register: rax

    - name: Add servers to in memory groups
      add_host:
        hostname: "{{ item.name }}"
        ansible_host: "{{ item.rax_accessipv4 }}"
        ansible_password: "{{ item.rax_adminpass }}"
        ansible_user: root
        rax_id: "{{ item.rax_id }}"
        groups: web,new_web
      loop: "{{ rax.success }}"
      when: rax.action == 'create'

- name: Wait for rackconnect and managed cloud automation to complete
  hosts: new_web
  gather_facts: false
  tasks:
    - name: ensure we run all tasks from localhost
      delegate_to: localhost
      block:
        - name: Wait for rackconnnect automation to complete
          rax_facts:
            credentials: ~/.raxpub
            id: "{{ rax_id }}"
            region: DFW
          register: rax_facts
          until: rax_facts.ansible_facts['rax_metadata']['rackconnect_automation_status']|default('') == 'DEPLOYED'
          retries: 30
```

```
        delay: 10

      - name: Wait for managed cloud automation to complete
        rax_facts:
          credentials: ~/.raxpub
          id: "{{ rax_id }}"
          region: DFW
        register: rax_facts
        until: rax_facts.ansible_facts['rax_metadata']['rax_service_level_automation']|default('') == 'Complete'
        retries: 30
        delay: 10

- name: Update new_web hosts with IP that RackConnect assigns
  hosts: new_web
  gather_facts: false
  tasks:
    - name: Get facts about servers
      rax_facts:
        name: "{{ inventory_hostname }}"
        region: DFW
      delegate to: localhost
    - name: Map some facts
      set_fact:
        ansible_host: "{{ rax_accessipv4 }}"

- name: Base Configure Servers
  hosts: web
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

    - role: ntp
```

**Using Ansible Pull**

```
---
- name: Ensure Rackconnect and Managed Cloud Automation is complete
  hosts: all
  tasks:
    - name: ensure we run all tasks from localhost
      delegate_to: localhost
      block:
        - name: Check for completed bootstrap
          stat:
            path: /etc/bootstrap_complete
          register: bootstrap

        - name: Get region
          command: xenstore-read vm-data/provider_data/region
          register: rax_region
          when: bootstrap.stat.exists != True

        - name: Wait for rackconnect automation to complete
          uri:
            url: "https://{{ rax_region.stdout|trim }}.api.rackconnect.rackspace.com/v1/automation_status?format=json"
            return content: yes
          register: automation_status
          when: bootstrap.stat.exists != True
          until: automation_status['automation_status']|default('') == 'DEPLOYED'
          retries: 30
          delay: 10

        - name: Wait for managed cloud automation to complete
          wait for:
            path: /tmp/rs_managed_cloud_automation_complete
            delay: 10
          when: bootstrap.stat.exists != True

        - name: Set bootstrap completed
          file:
            path: /etc/bootstrap_complete
            state: touch
            owner: root
            group: root
            mode: 0400

- name: Base Configure Servers
  hosts: all
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

    - role: ntp
```

**Using Ansible Pull with XenStore**

```
---
- name: Ensure Rackconnect and Managed Cloud Automation is complete
  hosts: all
  tasks:
    - name: Check for completed bootstrap
      stat:
        path: /etc/bootstrap_complete
      register: bootstrap

    - name: Wait for rackconnect automation status xenstore key to exist
      command: xenstore-exists vm-data/user-metadata/rackconnect_automation_status
      register: rcas_exists
```

```
      when: bootstrap.stat.exists != True
      failed when: rcas_exists.rc|int > 1
      until: rcas_exists.rc|int == 0
      retries: 30
      delay: 10

    - name: Wait for rackconnect automation to complete
      command: xenstore-read vm-data/user-metadata/rackconnect_automation_status
      register: rcas
      when: bootstrap.stat.exists != True
      until: rcas.stdout|replace('"', '') == 'DEPLOYED'
      retries: 30
      delay: 10

    - name: Wait for rax_service_level_automation xenstore key to exist
      command: xenstore-exists vm-data/user-metadata/rax_service_level_automation
      register: rsla_exists
      when: bootstrap.stat.exists != True
      failed_when: rsla_exists.rc|int > 1
      until: rsla_exists.rc|int == 0
      retries: 30
      delay: 10

    - name: Wait for managed cloud automation to complete
      command: xenstore-read vm-data/user-metadata/rackconnect_automation_status
      register: rsla
      when: bootstrap.stat.exists != True
      until: rsla.stdout|replace('"', '') == 'DEPLOYED'
      retries: 30
      delay: 10

    - name: Set bootstrap completed
      file:
        path: /etc/bootstrap_complete
        state: touch
        owner: root
        group: root
        mode: 0400

- name: Base Configure Servers
  hosts: all
  roles:
    - role: users

    - role: openssh
      opensshd_PermitRootLogin: "no"

    - role: ntp
```

## Advanced Usage

### Autoscaling with AWX or Red Hat Ansible Automation Platform

The GUI component of :ref:`Red Hat Ansible Automation Platform <ansible_tower>` also contains a very nice feature for auto-scaling use cases. In this mode, a simple curl script can call a defined URL and the server will "dial out" to the requester and configure an instance that is spinning up. This can be a great way to reconfigure ephemeral nodes. See the documentation on provisioning callbacks for more details.

> **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_rax.rst`, line 788); *backlink*
>
> Unknown interpreted text role "ref".

A benefit of using the callback approach over pull mode is that job results are still centrally recorded and less information has to be shared with remote hosts.

### Orchestration in the Rackspace Cloud

Ansible is a powerful orchestration tool, and rax modules allow you the opportunity to orchestrate complex tasks, deployments, and configurations. The key here is to automate provisioning of infrastructure, like any other piece of software in an environment. Complex deployments might have previously required manual manipulation of load balancers, or manual provisioning of servers. Utilizing the rax modules included with Ansible, one can make the deployment of additional nodes contingent on the current number of running nodes, or the configuration of a clustered application dependent on the number of nodes with common metadata. One could automate the following scenarios, for example:

- Servers that are removed from a Cloud Load Balancer one-by-one, updated, verified, and returned to the load balancer pool
- Expansion of an already-online environment, where nodes are provisioned, bootstrapped, configured, and software installed
- A procedure where app log files are uploaded to a central location, like Cloud Files, before a node is decommissioned
- Servers and load balancers that have DNS records created and destroyed on creation and decommissioning, respectively