Chapter 7 - Inventories

Earlier in the book, a basic inventory file example was given (see Chapter 1's basic inventory file example). For the simplest of purposes, an inventory file at the default location (`/etc/ansible/hosts`) will suffice to describe to Ansible how to reach the servers you want to manage.

Later, a slightly more involved inventory file was introduced (see Chapter 3's inventory file for multiple servers), which allowed us to tell Ansible about multiple servers, and even group them into role-related groups, so we could run certain playbooks against certain groups.

Let's jump back to a basic inventory file example and build from there:

```
# Inventory file at /etc/ansible/hosts

# Groups are defined using square brackets (e.g. [groupname]).
# Each server in the group is defined on its own line.
[myapp]
www.myapp.com
```

If you want to run an ansible playbook on all the `myapp` servers in this inventory (so far, just one, `www.myapp.com`), you can set up the playbook like so:

```
---
- hosts: myapp

  tasks:
    [...]
```

If you want to run an ad-hoc command against all the `myapp` servers in the inventory, you can run a command like so:

```
# Use ansible to check memory usage on all the myapp servers.
$ ansible myapp -a "free -m"
```

A real-world web application server inventory

The example above might be adequate for single-server services and tiny apps or websites, but most real-world applications require many more servers, and usually separate servers per application concern (database, caching, application, queuing, etc.). Let's take a look at a real-world inventory file for a small web application that monitors server uptime, Server Check.in[76].

```
# Individual Server Check.in servers.
[servercheck-web]
www1.servercheck.in
www2.servercheck.in

[servercheck-web:vars]
ansible_ssh_user=servercheck_svc

[servercheck-db]
db1.servercheck.in

[servercheck-log]
log.servercheck.in

[servercheck-backup]
backup.servercheck.in

[servercheck-nodejs]
atl1.servercheck.in
atl2.servercheck.in
nyc1.servercheck.in
nyc2.servercheck.in
```

[76]https://servercheck.in/

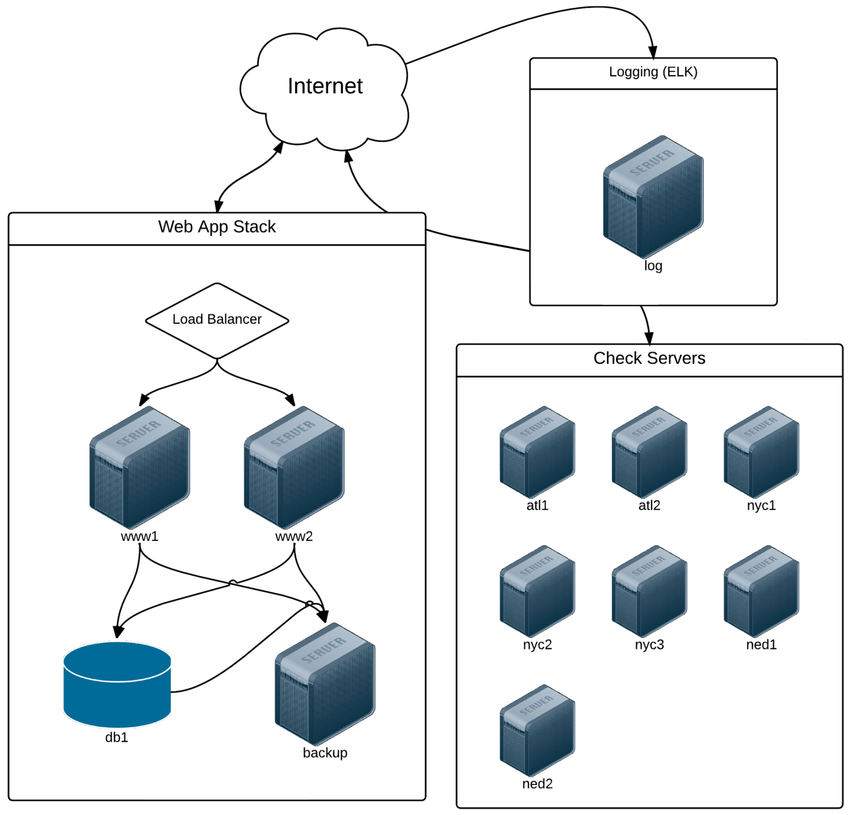
```
nyc3.servercheck.in
ned1.servercheck.in
ned2.servercheck.in

[servercheck-nodejs:vars]
ansible_ssh_user=servercheck_svc
foo=bar

# Server Check.in distribution-based groups.
[centos:children]
servercheck-web
servercheck-db
servercheck-nodejs
servercheck-backup

[ubuntu:children]
servercheck-log
```

This inventory may look a little overwhelming at first, but if you break it apart into simple groupings (web app servers, database servers, logging server, and node.js app servers), it describes a straightforward architecture.



Server Check.in Infrastructure.

Lines 1-29 describe a few groups of servers (some with only one server), so playbooks and `ansible` commands can refer to the group by name. Lines 6-7 and 27-29 set variables that will apply only to the servers in the group (e.g. variables below `[servercheck-nodejs:vars]` will only apply to the servers in the `servercheck-nodejs` group).

Lines 31-39 describe groups of groups (using `groupname:children` to describe 'child' groups) that allow for some helpful abstractions.

Describing infrastructure in such a way affords a lot of flexibility when using Ansible. Consider the task of patching a vulnerability on all your CentOS servers; instead of having to log into each of the servers, or even having to run an `ansible` command against all the groups, using the above structure allows you to easily run an ansible command or playbook against all `centos` servers.

As an example, when the Shellshock[77] vulnerability was disclosed in 2014, patched bash packages were released for all the major distributions within hours. To update all the Server Check.in servers, all that was needed was:

```
$ ansible centos -m yum -a "name=bash state=latest"
```

You could even go further and create a small playbook that would patch the vulnerability, then run tests to make sure the vulnerability was no longer present, as illustrated in this playbook[78]. This would also allow you to run the playbook in check mode or run it through a continuous integration system to verify the fix works in a non-prod environment.

This infrastructure inventory is also nice in that you could create a top-level playbook that runs certain roles or tasks against all your infrastructure, others against all servers of a certain Linux flavor, and another against all servers in your entire infrastructure.

Consider, for example, this example master playbook to completely configure all the servers:

```
---
# Set up basic, standardized components across all servers.
- hosts: all
  become: yes
  roles:
    - security
    - logging
    - firewall

# Configure web application servers.
- hosts: servercheck-web
  roles:
    - nginx
```

[77]https://en.wikipedia.org/wiki/Shellshock_(software_bug)
[78]https://raymii.org/s/articles/Patch_CVE-2014-6271_Shellshock_with_Ansible.html

```
    - php
    - servercheck-web

# Configure database servers.
- hosts: servercheck-db
  roles:
    - pgsql
    - db-tuning

# Configure logging server.
- hosts: servercheck-log
  roles:
    - java
    - elasticsearch
    - logstash
    - kibana

# Configure backup server.
- hosts: servercheck-backup
  roles:
    - backup

# Configure Node.js application servers.
- hosts: servercheck-nodejs
  roles:
    - servercheck-node
```

There are a number of different ways you can structure your infrastructure-management playbooks and roles, and we'll explore some in later chapters, but for a simple infrastructure, something like this is adequate and maintainable.

Non-prod environments, separate inventory files

Using the above playbook and the globally-configured Ansible inventory file is great for your production infrastructure, but what happens when you want to configure a separate but similar infrastructure for, say a development or user certification environment?

In this case, it’s easiest to use individual inventory files, rather than the central Ansible inventory file. For typical team-managed infrastructure, I would recommend including an inventory file for each environment in the same version-controlled repository as your Ansible playbooks, perhaps within an ‘inventories’ directory.

For example, I could take the entire contents of `/etc/ansible/hosts` above, and stash that inside an inventory file named `inventory-prod`, then duplicate it, changing server names where appropriate (e.g. the `[servercheck-web]` group would only have `www-dev1.servercheck.in` for the development environment), and naming the files for the environments:

```
servercheck/
  inventories/
    inventory-prod
    inventory-cert
    inventory-dev
  playbook.yml
```

Now, when running `playbook.yml` to configure the development infrastructure, I would pass in the path to the dev inventory (assuming my current working directory is `servercheck/`):

```
$ ansible-playbook playbook.yml -i inventories/inventory-dev
```

Using inventory variables (which will be explored further), and well-constructed roles and/or tasks that use the variables effectively, you could architect your entire infrastructure, with environment-specific configurations, by changing some things in your inventory files.

Inventory variables

Chapter 5 introduced basic methods of managing variables for individual hosts or groups of hosts through your inventory in the inventory variables section, but it’s worth exploring the different ways of defining and overriding variables through inventory here.

For extremely simple use cases—usually when you need to define one or two connection-related variables (like `ansible_ssh_user` or `ansible_ssh_port`)—you can place variables directly inside an inventory file.

Assuming we have a standalone inventory file for a basic web application, here are some examples of variable definition inside the file:

```
[www]
# You can define host-specific variables inline with the host.
www1.example.com ansible_ssh_user=johndoe
www2.example.com

[db]
db1.example.com
db2.example.com

# You can add a '[group:vars]' heading to create variables that will ap\
ply
# to an entire inventory group.
[db:vars]
ansible_ssh_port=5222
database_performance_mode=true
```

It's usually better to avoid throwing too many variables inside static inventory files, because not only are these variables typically less visible, they are also mixed in with your architecture definition. Especially for host-specific vars (which appear on one long line per host), this is an unmaintainable, low-visibility approach to host and group-specific variables.

Fortunately, Ansible provides a more flexible way of declaring host and group variables.

host_vars

For Hosted Apache Solr[79], different servers in a solr group have different memory requirements. The simplest way to tell Ansible to override a default variable in our Ansible playbook (in this case, the tomcat_xmx variable) is to use a host_vars directory (which can be placed either in the same location as your inventory file, or in a playbook's root directory), and place a YAML file named after the host which needs the overridden variable.

As an illustration of the use of host_vars, we'll assume we have the following directory layout:

```
hostedapachesolr/
  host_vars/
    nyc1.hostedapachesolr.com
  inventory/
    hosts
  main.yml
```

The inventory/hosts file contains a simple definition of all the servers by group:

```
[solr]
nyc1.hostedapachesolr.com
nyc2.hostedapachesolr.com
jap1.hostedapachesolr.com
...

[log]
log.hostedapachesolr.com
```

Ansible will search for a file at either:

```
hostedapachesolr/host_vars/nyc1.hostedapachesolr.com
```

Or:

[79] https://hostedapachesolr.com/

```
hostedapachesolr/inventory/host_vars/nyc1.hostedapachesolr.com
```

If there are any variables defined in the file (in YAML format), those variables will override all other playbook and role variables and gathered facts, *only for the single host.*

The `nyc1.hostedapachesolr.com` host_vars file looks like:

```
---
tomcat_xmx: "1024m"
```

The default for `tomcat_xmx` may normally be `640m`, but when Ansible runs a playbook against nyc1.hostedapachesolr.com, the value of `tomcat_xmx` will be `1024m` instead.

Overriding host variables with `host_vars` is much more maintainable than doing so directly in static inventory files, and also provides greater visibility into what hosts are getting what overrides.

`group_vars`

Much like `host_vars`, Ansible will automatically load any files named after inventory groups in a `group_vars` directory placed inside the playbook or inventory file's location.

Using the same example as above, we'll override one particular variable for an entire group of servers. First, we add a `group_vars` directory with a file named after the group needing the overridden variable:

```
hostedapachesolr/
  group_vars/
    solr
  host_vars/
    nyc1.hostedapachesolr.com
  inventory/
    hosts
  main.yml
```

Then, inside `group_vars/solr`, use YAML to define a list of variables that will be applied to servers in the `solr` group:

```
---
do_something_amazing=true
foo=bar
```

Typically, if your playbook is only being run on one group of hosts, it's easier to define the variables in the playbook via an included vars file. However, in many cases you will be running a playbook or applying a set of roles to multiple inventory groups. In these situations, you may need to use `group_vars` to override specific variables for one or more groups of servers.

Ephemeral infrastructure: Dynamic inventory

In many circumstances, static inventories are adequate for describing your infrastructure. When working on small applications, low-traffic web applications, and individual workstations, it's simple enough to manage an inventory file by hand.

However, in the age of cloud computing and highly scalable application architecture, it's often necessary to add dozens or hundreds of servers to an infrastructure in a short period of time—or to add and remove servers continuously, to scale as traffic grows and subsides. In this circumstance, it would be tedious (if not impossible) to manage a single inventory file by hand, especially if you're using auto-scaling infrastructure new instances are provisioned and need to be configured in minutes or seconds.

Even in the case of container-based infrastructure, new instances need to be configured correctly, with the proper port mappings, application settings, and filesystem configuration.

For these situations, Ansible allows you to define inventory *dynamically*. If you're using one of the larger cloud-based hosting providers, chances are there is already a dynamic inventory script (which Ansible uses to build an inventory) for you to use. Ansible core already includes scripts for Amazon Web Services, Cobbler, DigitalOcean, Linode, OpenStack, and other large providers, and later we'll explore creating our own dynamic inventory script (if you aren't using one of the major hosting providers or cloud management platforms).

Dynamic inventory with DigitalOcean

DigitalOcean is one of the world's top five hosting companies, and has grown rapidly since its founding in 2011. One of the reasons for the extremely rapid growth is the ease of provisioning new 'droplets' (cloud VPS servers), and the value provided; as of this writing, you could get a fairly speedy VPS with 1GB of RAM and a generous portion of fast SSD storage for $5 USD per month.

DigitalOcean's API and developer-friendly philosophy has made it easy for Ansible to interact with DigitalOcean droplets; you can create, manage, and delete droplets with Ansible, as well as use droplets with your playbooks using dynamic inventory.

DigitalOcean account prerequisites

Before you can follow the rest of the examples in this section, you will need:

1. A DigitalOcean account (sign up at www.digitalocean.com).
2. `dopy`, a Python wrapper for DigitalOcean API interaction (you can install it with pip: `sudo pip install dopy`).
3. A DigitalOcean API Personal Access Token. Follow this guide[80] to generate a Personal Access Token for use with Ansible (grant Read and Write access when you create the token).

[80]https://www.digitalocean.com/community/tutorials/how-to-use-the-digitalocean-api-v2

4. An SSH key pair, which will be used to connect to your DigitalOcean servers. Follow this guide[81] to create a key pair and add the public key to your DigitalOcean account.

Once you have these four things set up and ready to go, you should be able to communicate with your DigitalOcean account through Ansible.

Connecting to your DigitalOcean account

There are a few different ways you can specify your DigitalOcean Personal Access Token (including passing it via `api_token` to each DigitalOcean-related task, or exporting it in your local environment as `DO_API_TOKEN` or `DO_API_KEY`). For our example, we'll use environment variables (since these are easy to configure, and work both with Ansible's `digital_ocean` module and the dynamic inventory script). Open up a terminal session, and enter the following command:

```
$ export DO_API_TOKEN=YOUR_API_TOKEN_HERE
```

Before we can use a dynamic inventory script to discover our DigitalOcean droplets, let's use Ansible to provision a new droplet.

Creating cloud instances ('Droplets', in DigitalOcean parlance) will incur minimal charges for the time you use them (currently less than $0.01/hour for the size in this example). For the purposes of this tutorial (and in general, for any testing), make sure you shut down and destroy your instances when you're finished using them, or you will be charged through the next billing cycle! Even so, using low-priced instances (like a $5/month DigitalOcean droplet with hourly billing) means that, even in the worst case, you won't have to pay much. If you create and destroy an instance in a few hours, you'll be charged a few pennies.

Creating a droplet with Ansible

Create a new playbook named `provision.yml`, with the following contents:

[81] https://www.digitalocean.com/community/tutorials/how-to-use-ssh-keys-with-digitalocean-droplets

```
---
- hosts: localhost
  connection: local
  gather_facts: False

  tasks:
    - name: Create new Droplet.
      digital_ocean:
        state: present
        command: droplet
        name: ansible-test
        private_networking: yes
        size_id: s-1vcpu-1gb
        image_id: centos-7-x64
        region_id: nyc3
        # Customize this for your account.
        ssh_key_ids: 138954
        # Required for idempotence/only one droplet creation.
        unique_name: yes
      register: do
```

The `digital_ocean` module lets you create, manage, and delete droplets with ease. You can read the documentation for all the options, but the above is an overview of the main options. `name` sets the hostname for the droplet, `state` can also be set to `deleted` if you want the droplet to be destroyed, and other options tell DigitalOcean where to set up the droplet, and with what OS and configuration.

You can use DigitalOcean's API, along with your Personal Access Token, to get the IDs for `size_id` (the size of the Droplet), `image_id` (the system or distro image to use), `region_id` (the data center in which your droplet will be created), and `ssh_key_ids` (a comma separate list of SSH keys to be included in the root account's `authorized_keys` file).

As an example, to get all the available images, use `curl --silent "https://api.digitalocean.com/v2/images?per_page=999" -H "Authorization: Bearer $DO_API_TOKEN" | python -m json.tool`, and you'll receive a JSON listing of all available values. Browse the DigitalOcean API[82] for information on how to query SSH key information, size information, etc.

We used `register` as part of the `digital_ocean` task so we could immediately start using and configuring the new host if needed. Running the above playbook returns the following output (using `debug: var=do` in an additional task to dump the contents of our registered variable, `do`):

```
$ ansible-playbook provision.yml

PLAY [localhost] *******************************************************\
*********

TASK [Create new Droplet.] *********************************************\
*********
changed: [localhost]

TASK [debug] ***********************************************************\
*********
ok: [localhost] => {
    "do": {
        "changed": true,
        "droplet": {
            "backup_ids": [],
            "created_at": "2017-07-22T00:58:51Z",
```

[82]https://developers.digitalocean.com/

```
            "disk": 20,
            "features": [
                "private_networking",
                "virtio"
            ],
            "id": 20203631,
            "image": {
                ...
            },
            "ip_address": "162.243.20.29",
            "kernel": null,
            "locked": false,
            "memory": 512,
            "name": "ansible-test",
            "networks": {
                ...
            },
            "next_backup_window": null,
            "private_ip_address": "10.1.1.2",
            "region": {
                ...
            },
            "size": {
                ...
            },
            "size_slug": "512mb",
            "snapshot_ids": [],
            "status": "active",
            "tags": [],
            "vcpus": 1,
            "volume_ids": []
        }
    }
}

PLAY RECAP **********************************************************
```

```
localhost              : ok=2    changed=1    unreachable=0    failed=0
```

Since `do` contains the new droplet's IP address (alongside other relevant information), you can place your freshly-created droplet in an existing inventory group using Ansible's `add_host` module. Adding to the playbook we started above, you could set up your playbook to provision an instance and immediately configure it (after waiting for port 22 to become available) with something like:

```
    - name: Add new host to our inventory.
      add_host:
        name: "{{ do.droplet.ip_address }}"
        groups: do
      when: do.droplet is defined
      changed_when: False

- hosts: do
  remote_user: root
  gather_facts: False

  tasks:
    - name: Wait for port 22 to become available.
      local_action: "wait_for port=22 host={{ inventory_hostname }}"

    - name: Install tcpdump.
      yum: name=tcpdump state=present
```

At this point, if you run the playbook (`$ ansible-playbook provision.yml`), it should create a new droplet (if it has not already been created), then add that droplet to the `do` inventory group, and finally, run a new play on all the `do` hosts (including the new droplet). Here are the results:

```
$ ansible-playbook provision.yml

PLAY [localhost] ****************************************************

TASK: [Create new Droplet.] *****************************************
changed: [localhost]

TASK: [Add new host to our inventory.] ******************************
ok: [localhost]

PLAY [do] ***********************************************************

TASK [Wait for port 22 to become available.] *************************\
*********
ok: [162.243.20.29 -> localhost]

TASK: [Install tcpdump.] ********************************************
changed: [162.243.20.29]

PLAY RECAP **********************************************************
162.243.20.29          : ok=2    changed=1    unreachable=0    failed=0
localhost              : ok=2    changed=1    unreachable=0    failed=0
```

If you run the same playbook again, it should report no changes—the entire playbook is idempotent! You might be starting to see just how powerful it is to have a tool as flexible as Ansible at your disposal; not only can you configure servers, you can create them (singly, or dozens at a time), and configure them at once. And even if a ham-fisted sysadmin jumps in and deletes an entire server, you can run the playbook again, and rest assured your server will be recreated and reconfigured exactly as it was when it was first set up.

If you get an error like "Failed to connect to the host via ssh: Host key verification failed.", then you can temporarily disable host key checking. Run the command `export ANSIBLE_HOST_KEY_CHECKING=False` and then run the `provision.yml` playbook again.

You should normally leave `host_key_checking` enabled, but when rapidly building and destroying VMs for testing purposes, it is simplest to disable it temporarily.

DigitalOcean dynamic inventory with `digital_ocean.py`

Once you have some DigitalOcean droplets, you need a way for Ansible to dynamically build an inventory of your servers so you can build playbooks and use the servers in logical groupings (or run playbooks and `ansible` commands directly on all droplets).

There are a few steps to getting DigitalOcean's official dynamic inventory script working:

1. Install `dopy` via pip (the DigitalOcean Python library):

   ```
   $ pip install dopy
   ```

2. Download the DigitalOcean dynamic inventory script[83] from Ansible on GitHub:

   ```
   $ curl -O https://raw.githubusercontent.com/ansible/ansible/devel/\
   contrib/inventory/digital_ocean.py
   ```

3. Make the inventory script executable:

   ```
   $ chmod +x digital_ocean.py
   ```

4. Make sure you have `DO_API_TOKEN` set in your environment.
5. Make sure the script is working by running the script directly (with the command below). After a second or two, you should see all your droplets (likely just the one you created earlier) listed by IP address and dynamic group as JSON.

[83]https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/digital_ocean.py

```
$ ./digital_ocean.py --pretty
```

6. Ping all your DigitalOcean droplets:

```
$ ansible all -m ping -i digital_ocean.py -u root
```

Now that you have all your hosts being loaded through the dynamic inventory script, you can use `add_hosts` to build groups of the Droplets for use in your playbooks. Alternatively, if you want to fork the `digital_ocean.py` inventory script, you can modify it to suit your needs; exclude certain servers, build groups based on certain criteria, etc.

Ansible < 1.9.5 only supported DigitalOcean's legacy v1 API, which is no longer supported. If you need to use Ansible with DigitalOcean, you should use the latest version of Ansible.

Dynamic inventory with AWS

Many of this book's readers are familiar with Amazon Web Services (especially EC2, S3, ElastiCache, and Route53), and likely have managed or currently manage an infrastructure within Amazon's cloud. Ansible has very strong support for managing AWS-based infrastructure, and includes a dynamic inventory script[84] to help you run playbooks on your hosts in a variety of ways.

There are a few excellent guides to using Ansible with AWS, for example:

- Ansible - Amazon Web Services Guide[85]
- Ansible for AWS[86]

I won't be covering dynamic inventory in this chapter, but will mention that the `ec2.py` dynamic inventory script, along with Ansible's extensive support for AWS infrastructure through `ec2_*` modules, makes Ansible the best and most simple tool for managing a broad AWS infrastructure.

In the next chapter, one of the examples will include a guide for provisioning infrastructure on AWS, along with a quick overview of dynamic inventory on AWS.

[84]https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/ec2.py
[85]http://docs.ansible.com/guide_aws.html
[86]https://leanpub.com/ansible-for-aws

Inventory on-the-fly: `add_host` and `group_by`

Sometimes, especially when provisioning new servers, you will need to modify the in-memory inventory during the course of a playbook run. Ansible offers the `add_host` and `group_by` modules to help you manage inventory for these scenarios.

In the DigitalOcean example above, `add_host` was used to add the new droplet to the `do` group:

```
[...]
    - name: Add new host to our inventory.
      add_host:
        name: "{{ do.droplet.ip_address }}"
        groups: do
      when: do.droplet is defined

- hosts: do
  remote_user: root

  tasks:
[...]
```

You could add multiple groups with add_host, and you can also add other variables for the host inline with `add_host`. As an example, let's say you created a VM using an image that exposes SSH on port 2288 and requires an application-specific memory limit specific to this VM:

```
- name: Add new host to our inventory.
  add_host:
    name: "{{ do.droplet.ip_address }}"
    ansible_ssh_port: 2288
    myapp_memory_maximum: "1G"
  when: do.droplet is defined
```

The custom port will be used when Ansible connects to this host, and the `myapp_memory_maximum` will be passed into the playbooks just as any other inventory variable.

The `group_by` module is even simpler, and allows you to create dynamic groups during the course of a playbook run. Usage is extremely simple:

```
- hosts: all
  gather_facts: yes
  tasks:
    - name: Create an inventory group for each architecture.
      group_by: "key=architecture-{{ ansible_machine }}"

    - debug: var=groups
```

After running the above playbook, you'd see all your normal inventory groups, plus groups for `architecture-x86_64`, `i386`, etc. (depending on what kind of server architectures you use).

Multiple inventory sources - mixing static and dynamic inventories

If you need to combine static and dynamic inventory, or even if you wish to use multiple dynamic inventories (for example, if you are managing servers hosted by two different cloud providers), you can pass a directory to `ansible` or `ansible-playbook`, and Ansible will combine the output of all the inventories (both static and dynamic) inside the directory:

```
`ansible-playbook -i path/to/inventories main.yml`
```

One caveat: Ansible ignores `.ini` and backup files in the directory, but will attempt to parse every text file and execute every executable file in the directory—don't leave random files in mixed inventory folders!

Creating custom dynamic inventories

Most infrastructure can be managed with a custom inventory file or an off-the-shelf cloud inventory script, but there are many situations where more control is needed.

Ansible will accept any kind of executable file as an inventory file, so you can build your own dynamic inventory however you like, as long as you can pass it to Ansible as JSON.

You could create an executable binary, a script, or anything else that can be run and will output JSON to stdout, and Ansible will call it with the argument `--list` when you run, as an example, `ansible all -i my-inventory-script -m ping`.

Let's start working our own custom dynamic inventory script by outlining the basic JSON format Ansible expects:

```
{
    "group": {
        "hosts": [
            "192.168.28.71",
            "192.168.28.72"
        ],
        "vars": {
            "ansible_ssh_user": "johndoe",
            "ansible_ssh_private_key_file": "~/.ssh/mykey",
            "example_variable": "value"
        }
    },
    "_meta": {
        "hostvars": {
            "192.168.28.71": {
                "host_specific_var": "bar"
            },
            "192.168.28.72": {
                "host_specific_var": "foo"
            }
        }
    }
}
```

Ansible expects a dictionary of groups (with each group having a list of `hosts`, and group variables in the group's `vars` dictionary), and a `_meta` dictionary that stores host variables for all hosts individually inside a `hostvars` dictionary.

When you return a `_meta` dictionary in your inventory script, Ansible stores that data in its cache and doesn't call your inventory script *N* times for all the hosts in the inventory. You can leave out the `_meta` variables if you'd rather structure your inventory file to return host variables one host at a time (Ansible will call your script with the arguments `--host [hostname]` for each host), but it's often faster and easier to simply return all variables in the first call. In this book, all the examples will use the `_meta` dictionary.

The dynamic inventory script can do anything to get the data (call an external API, pull information from a database or file, etc.), and Ansible will use it as an inventory source, so long as it returns a JSON structure like the one above when the script is called with the `--list`.

Building a Custom Dynamic Inventory in Python

To create a test dynamic inventory script for demonstration purposes, let's set up a quick set of two VMs using Vagrant. Create the following `Vagrantfile` in a new directory:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.ssh.insert_key = false
  config.vm.provider :virtualbox do |v|
    v.memory = 256
    v.linked_clone = true
  end

  # Application server 1.
  config.vm.define "inventory1" do |inventory|
    inventory.vm.hostname = "inventory1.test"
    inventory.vm.box = "geerlingguy/ubuntu1804"
    inventory.vm.network :private_network, ip: "192.168.28.71"
  end

  # Application server 2.
  config.vm.define "inventory2" do |inventory|
    inventory.vm.hostname = "inventory2.test"
    inventory.vm.box = "geerlingguy/ubuntu1804"
    inventory.vm.network :private_network, ip: "192.168.28.72"
  end
end
```

Run `vagrant up` to boot two VMs running Ubuntu, with the IP addresses `192.168.28.71`, and `192.168.28.72`. A simple inventory file could be used to control the VMs with Ansible:

```
[group]
192.168.28.71 host_specific_var=foo
192.168.28.72 host_specific_var=bar

[group:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
example_variable=value
```

However, let's assume the VMs were provisioned by another system, and you need to get the information through a dynamic inventory script. Here's a simple implementation of a dynamic inventory script in Python:

```
#!/usr/bin/env python

'''
Example custom dynamic inventory script for Ansible, in Python.
'''

import os
import sys
import argparse
```

```

try:
    import json
except ImportError:
    import simplejson as json

class ExampleInventory(object):

    def __init__(self):
        self.inventory = {}
        self.read_cli_args()

        # Called with `--list`.
        if self.args.list:
            self.inventory = self.example_inventory()
        # Called with `--host [hostname]`.
        elif self.args.host:
            # Not implemented, since we return _meta info `--list`.
            self.inventory = self.empty_inventory()
        # If no groups or vars are present, return empty inventory.
        else:
            self.inventory = self.empty_inventory()

        print json.dumps(self.inventory);

    # Example inventory for testing.
    def example_inventory(self):
        return {
            'group': {
                'hosts': ['192.168.28.71', '192.168.28.72'],
                'vars': {
                    'ansible_ssh_user': 'vagrant',
                    'ansible_ssh_private_key_file':
                        '~/.vagrant.d/insecure_private_key',
                    'ansible_python_interpreter':
                        '/usr/bin/python3',
                    'example_variable': 'value'
                }
            },
            '_meta': {
                'hostvars': {
                    '192.168.28.71': {
                        'host_specific_var': 'foo'
                    },
                    '192.168.28.72': {
                        'host_specific_var': 'bar'
                    }
                }
            }
        }

    # Empty inventory for testing.
    def empty_inventory(self):
        return {'_meta': {'hostvars': {}}}

    # Read the command line args passed to the script.
    def read_cli_args(self):
        parser = argparse.ArgumentParser()
        parser.add_argument('--list', action = 'store_true')
        parser.add_argument('--host', action = 'store')
        self.args = parser.parse_args()

# Get the inventory.
ExampleInventory()
```

Save the above as `inventory.py` in the same folder as the `Vagrantfile` you created earlier (make sure you booted the two VMs with `vagrant up`), and make the file executable `chmod +x inventory.py`.

Run the inventory script manually to verify it returns the proper JSON response when run with `--list`:

```
$ ./inventory.py --list
{"group": {"hosts": ["192.168.28.71", "192.168.28.72"], "vars":
{"ansible_ssh_user": "vagrant", "ansible_ssh_private_key_file":
"~/.vagrant.d/insecure_private_key", "example_variable": "value
"}}, "_meta": {"hostvars": {"192.168.28.72": {"host_specific_va
r": "bar"}, "192.168.28.71": {"host_specific_var": "foo"}}}}
```

Test Ansible's ability to use the inventory script to contact the two VMs:

```
$ ansible all -i inventory.py -m ping
192.168.28.71 | success >> {
    "changed": false,
    "ping": "pong"
}

192.168.28.72 | success >> {
    "changed": false,
    "ping": "pong"
}
```

Since Ansible can connect, verify the configured host variables (`foo` and `bar`) are set correctly on their respective hosts:

```
$ ansible all -i inventory.py -m debug -a "var=host_specific_var"
192.168.28.71 | success >> {
    "var": {
        "host_specific_var": "foo"
    }
}

192.168.28.72 | success >> {
    "var": {
        "host_specific_var": "bar"
    }
}
```

The only alteration for real-world usage you'd need to make to the above `inventory.py` script would be changing the `example_inventory()` method to something that incorporates the business logic you would need for your own inventory, whether it would be calling an external API with all the server data or pulling in the information from a database or other data store.

Building a Custom Dynamic Inventory in PHP

You can build an inventory script in whatever language you'd like. For example, the Python script from above can be ported to functional PHP as follows:

```php
#!/usr/bin/php
<?php

/**
 * @file
 * Example custom dynamic inventory script for Ansible, in PHP.
 */

/**
 * Example inventory for testing.
 *
 * @return array
 *   An example inventory with two hosts.
 */
function example_inventory() {
  return [
    'group' => [
      'hosts' => ['192.168.28.71', '192.168.28.72'],
      'vars' => [
        'ansible_ssh_user' => 'vagrant',
        'ansible_ssh_private_key_file' => '~/.vagrant.d/\
insecure_private_key',
        'ansible_python_interpreter' => '/usr/bin/python3',
        'example_variable' => 'value',
      ],
    ],
    '_meta' => [
      'hostvars' => [
        '192.168.28.71' => [
          'host_specific_var' => 'foo',
        ],
        '192.168.28.72' => [
          'host_specific_var' => 'bar',
        ],
      ],
    ],
  ];
}

/**
 * Empty inventory for testing.
 *
 * @return array
 *   An empty inventory.
 */
function empty_inventory() {
  return ['_meta' => ['hostvars' => new stdClass()]];
}

/**
 * Get inventory.
 *
 * @param array $argv
 *   Array of command line arguments (as in $_SERVER['argv']).
 *
 * @return array
 *   Inventory of groups or vars, depending on arguments.
 */
function get_inventory($argv = []) {
  $inventory = new stdClass();

  // Called with `--list`.
  if (!empty($argv[1]) && $argv[1] == '--list') {
    $inventory = example_inventory();
  }
  // Called with `--host [hostname]`.
  elseif ((!empty($argv[1]) && $argv[1] == '--host') && \
!empty($argv[2])) {
    // Not implemented, since we return _meta info `--list`.
    $inventory = empty_inventory();
  }
  // If no groups or vars are present, return an empty inventory.
  else {
    $inventory = empty_inventory();
  }

  print json_encode($inventory);
}

// Get the inventory.
get_inventory($_SERVER['argv']);

?>
```

If you were to save the code above into the file `inventory.php`, mark it executable (`chmod +x inventory.php`), and run the same Ansible command as earlier (referencing `inventory.php` instead of `inventory.py`), the command should succeed, just as with the previous Python example.

All the files mentioned in these dynamic inventory examples are available in the Ansible for DevOps GitHub repository[87], in the `dynamic-inventory` folder.

[87]https://github.com/geerlingguy/ansible-for-devops

Managing a PaaS with a Custom Dynamic Inventory

Hosted Apache Solr[88]'s infrastructure is built using a custom dynamic inventory to allow for centrally-controlled server provisioning and configuration. Here's how the server provisioning process works on Hosted Apache Solr:

1. A Drupal website holds a 'Server' content type that stores metadata about each server (e.g. chosen hostname, data center location, choice of OS image, and memory settings).
2. When a new server is added, a remote Jenkins job is triggered, which: 1. Builds a new cloud server on DigitalOcean using an Ansible playbook. 2. Runs a provisioning playbook on the server to initialize the configuration. 3. Adds a new DNS entry for the server. 4. Posts additional server metadata (like the IP address) back to the Drupal website via a private API.
3. When a server is updated, or there is new configuration to be deployed to the server(s), a different Jenkins job is triggered, which: 1. Runs the same provisioning playbook on all the DigitalOcean servers. This playbook uses an inventory script which calls back to an inventory API endpoint that returns all the server information as JSON (the inventory script on the Jenkins server passes the JSON through to stdout). 2. Reports back success or failure of the ansible playbook to the REST API.

The above process transformed the management of the entire Hosted Apache Solr platform. Instead of taking twenty to thirty minutes to build a new server (even when using an Ansible playbook with a few manual steps), the process can be completed in just a few minutes, with no manual intervention.

The security of your server inventory and infrastructure management should be a top priority; Hosted Apache Solr uses HTTPS everywhere, and has a hardened private API for inventory access and server metadata. If you have any automated processes that run over a network, you should take extra care to audit these processes and all the involved systems thoroughly!

[88] https://hostedapachesolr.com/

Summary

From the most basic infrastructure consisting of one server to a multi-tenant, dynamic infrastructure with thousands of them, Ansible offers many options for describing your servers and overriding playbook and role variables for specific hosts or groups. With Ansible's flexible inventory system, you should be able to describe all your servers, however they're managed and wherever they're hosted.

```
 __________________________________
/ A pint of sweat saves a gallon of \
\ blood. (General Patton)           /
 ----------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```