Ansible Fundamentals

Managing Complex Inventories

Working with Dynamic Inventories

Red Hat

Objectives

- Install and use dynamic inventory scripts.

Red Hat

Limitations of Static Inventories

- Static inventories are easy to write, and convenient for a small number of managed hosts
- Difficult to keep up to date in a large infrastructure
- Also hard to use with short-lived cloud instances
- Most large environments have a "single source of truth" that tracks available hosts already
    - Monitoring systems like Zabbix
    - Directory servers like Active Directory or FreeIPA / Red Hat Identity Management
    - Other Configuration Management Databases (CMDBs)

Dynamic Inventories

- A dynamic inventory script is an executable program that can build an inventory automatically
- Usually used to get information from an external "source of truth" like a CMDB
- Configured in the same way as a static inventory file, but marked with executable permissions

```
chmod a+x inventory-script.py
```

- Can be written in any programming language that provides inventory output in JSON format
- **ansible-inventory -i inventory-file --list** will show you an example of this format for *inventory-file*
- A number of sample dynamic inventory scripts are available from the Ansible GitHub site at https://github.com/ansible/ansible/tree/devel/contrib/inventory

Comparing INI and JSON Format

- **ansible-inventory -i inventory-file --list** was run to convert the file on the left (in INI format) to the output on the right (in JSON format)
- Dynamic inventory uses JSON output because it is easier to parse when inventories are complex

```
workstation1.lab.example.com

[webservers]
web1.lab.example.com
web2.lab.example.com

[databases]
db1.lab.example.com
db2.lab.example.com
```

```
{
      "_meta": {
          "hostvars": {}
      },
      "all": {
          "children": [
             "databases",
             "ungrouped",
             "webservers"
          ]
      },
      "databases": {
          "hosts": [
             "db1.lab.example.com",
             "db2.lab.example.com"
          ]
      },
      "ungrouped": {
          "hosts": [
             "workstation1.lab.example.com"
          ]
      },
      "webservers": {
          "hosts": [
             "web1.lab.example.com",
             "web2.lab.example.com"
          ]
      }
}
```


Red Hat

Using Dynamic Inventories

- It is easiest to simply reuse one of the dynamic inventory scripts provided at GitHub
- Most of them have documentation included that indicates how to configure them
- If you want to write your own dynamic inventory scripts, look at the documentation: https://docs.ansible.com/ansible/dev_guide/developing_inventory.html
- A brief overview of the process of writing a dynamic inventory script follows.

Basics of Dynamic Inventory Scripts

- If the script is written in an interpreted language, start it with an appropriate interpreter line
  - **#!/usr/bin/python**
- The script should have executable permission so that Ansible can run it.
- When passed the **--list** option, the script must print a JSON-encoded hash/dictionary of all the hosts and groups in the inventory.
  - Output as seen at right:

```
{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "databases",
            "ungrouped",
            "webservers"
        ]
    },
    "databases": {
        "hosts": [
            "db1.lab.example.com",
            "db2.lab.example.com"
        ]
    },
    "ungrouped": {
        "hosts": [
            "workstation1.lab.example.com"
        ]
    },
    "webservers": {
        "hosts": [
            "web1.lab.example.com",
            "web2.lab.example.com"
        ]
    }
}
```

Basics of Dynamic Inventory Scripts

- The **_meta** section can be used to provide inventory variables from the external source.
- The example at right provides host variables for server1.example.com
  - **ntpserver**
  - **dnsserver**
- If your script does not provide inventory variables, provide an empty **_meta** section to speed up processing:

```
"_meta": {
    "hostvars": {}
}
```

- Make sure you have commas in the right places between sections and items

```
{
      "_meta": {
          "hostvars": {
            "server1.example.com": {
              "ntpserver": "ntp.example.com",
              "dnsserver": "8.8.8.8"
          }
      },
      "all": {
          "children": [
             "servers",
             "ungrouped"
          ]
      },
      "servers": {
          "hosts": [
             "server1.lab.example.com",
             "server2.lab.example.com"
          ]
      },
      "ungrouped": {
          "hosts": [
             "workstation1.lab.example.com"
          ]
      }
}
```


Red Hat

Summary of Using Dynamic Inventories

Once you have a script:

1. Copy it into place
2. Set it executable: **chmod +x *script***
3. Complete any configuration needed by the script itself
4. Configure Ansible to use it as your inventory in the same way as a static inventory
5. Test it to make sure it works by running **ansible-inventory** and other Ansible commands

Managing Inventory Variables

Red Hat

Objectives

- Construct inventories for different environments, parameters, and use cases.
- Combine variables from multiple inventory sources.

Using Inventories Effectively

- A well-structured inventory provides you with more ways to easily manage hosts

- Be willing to assign hosts to multiple groups, and organize your groups in different ways:
  - By function of the server -- what its purpose is
  - By geographic location -- region, datacenter, row, rack
  - By processor architecture
  - By operating system or operating system version
  - By place in the lifecycle -- development, testing, staging, production

- You can do many of these things with conditionals, but it is often less efficient to do that

- If a play should only target a certain kind of host, having well-defined groups can help you do that

Using Inventories Effectively

- If your hostnames are long or complicated, you can use something simple in your inventory and then use the **ansible_host** variable to point Ansible at the actual server

- For example, put **webserver** in your inventory, but in **host_vars/webserver** use **ansible_host** to set the "real" hostname
  - for example, maybe it is something like **ip-192.0.2.4.us-west-2.compute.internal**)

- Then your plays can reference **webserver** but the inventory will direct Ansible to the actual host

Inventory Variables and Groups

- If a variable is set by multiple groups that apply to the same host, which value wins?
    - If the variable is set as a host variable, that takes priority
    - If the variable is set by a child group and its parent group, the child group's value wins
    - If the variable is set by group *all*, it is overridden by any other group or host values

Inventory Variables and Groups

- If the variable is set by two groups at the same level (for example, two parent groups), they are merged alphabetically by default.
    - So if **testvar** is set by *a_group* and *b_group*, and *host* is a member of both groups, the value set by *b_group* overrides the value set by *a_group* by default
    - It is possible for this to be configured differently: for details, see https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html
- Try to set up your group variables to avoid this from being a possible problem.

Keeping Variables Organized

As you define and manage variables in a project, plan to follow these principles:

- **Keep it simple**.
  - Define variables using only a couple of different methods and in only a few places.

- **Do not repeat yourself**.
  - Set variables for a whole group, not for individual hosts, if they have the same value.

- **Organize variables into small, readable files**.
  - You can use a directory instead of a file for the group or host in **group_vars** or **host_vars**.
  - All files in that directory are automatically used.
  - Split variable definitions into multiple files for large projects.
  - To make it easier to find particular variables, group related variables into the same file and give it a meaningful name.

Multiple Inventory Sources

- You can use multiple inventory files and scripts combined together
  - Point the ansible.cfg **inventory** directive at a directory, not a file.
  - Put your static inventory files and dynamic inventory scripts in that directory.
  - Ansible will automatically combine all of them together.

- The inventory sources are combined in alphabetical order by default
  - Last source in alphabetical order that conflicts wins

- Therefore, if there is a risk of conflicts between the inventory sources, naming of the files and scripts will be important

Inventory Design

- Careful group design can make it easier to write your plays.
- Remember that the **hosts** directive in a play can target a group.
- A group may consist of only one host, or many hosts.

- Therefore, you can write a playbook containing multiple plays, each of which act on a different group or set of groups.

```
- name: Database configuration
  hosts: databases
  tasks:
    - name: First task
```
(rest of the first play in the playbook)

```
- name: Webserver config
  hosts: webservers
  tasks:
    - name: First task
```
(rest of the second play in the playbook)

- You can also set a condition on a task so that it runs only if the current managed host is also in another group by using the special **inventory_hostname** and **groups** variables:

```
- name: This task runs only if the current host is in group testing
  debug:
    msg: "{{ inventory_hostname }} is in group testing"
  when: inventory_hostname in groups['testing']
```

Inventory Design: Production, Testing, Development

- At right is an example of a single inventory that has groups based on server purpose and based on whether the server is in a production or test environment:

- The advantage of this inventory is that you can write plays that target machines in different ways:
  - all database servers
  - all web servers
  - all servers in the testing environment
  - all servers in the production environment
  - all servers everywhere (with group **all**)

```
[databases]
database1.prod-east.example.com
database1.test.example.com
web1.test.example.com
web1.prod-east.example.com

[webservers]
web1.test.example.com
web1.prod-east.example.com

[testing]
database1.test.example.com
web1.test.example.com

[production]
database1.prod-east.example.com
web1.prod-east.example.com
```

Inventory Design: Production, Testing, Development

- An alternative approach is to have two separate inventory files (or sources), one for testing and one for production:
- The advantage of this approach is that if you make a mistake with group **all** using the test environment's inventory, you cannot accidentally affect production hosts
- Some disadvantages of this approach:
  - You cannot easily write a play that affects both testing and production
  - You have to call a different inventory file (possibly with command line options) in different circumstances

inventory-production

```
[databases]
database1.prod-east.example.com
web1.prod-east.example.com

[webservers]
web1.prod-east.example.com

[production]
database1.prod-east.example.com
web1.prod-east.example.com
```

inventory-testing

```
[databases]
database1.test.example.com
web1.test.example.com

[webservers]
web1.test.example.com

[testing]
database1.test.example.com
web1.test.example.com
```


Red Hat

Inventory Design: Group Variables

- One reason you might use separate inventory files: to clarify which group's variables have precedence.
- To understand this, consider:

  If the same variable is set for two different groups in an inventory, and a host is a member of both groups, which group variable has precedence?

- In this case, *the last value loaded has precedence*. By default, files are loaded in alphabetical order.
- In the example at right, given the single inventory, the value of **a_conflict** displayed by the playbook is **from_webservers**

inventory

```
[webservers]
web1.prod-east.example.com

[production]
web1.prod-east.example.com
```

group_vars/production.yml

```
a_conflict: from_production
```

group_vars/webservers.yml

```
a_conflict: from_webservers
```

playbook.yml

```
- name: Test group var precedence
  hosts: production
  tasks:
    - name: Display value of a_conflict
      debug:
        var: a_conflict
```


Red Hat

Inventory Design: Child Groups

- Child groups are one way to try to resolve this.
- Variables set by child groups take precedence over values set by their parent groups.
- In the example at right, the host is still a member of both *webservers* and *production*, but the group *production* is a child of *webservers*:
- Now the value of **a_conflict** reported by the play is **from_production**.
- However, if you write a play that targets the group *webservers*, it will also affect all hosts in group *production*.

inventory

```
[webservers]
web1.prod-east.example.com

[webservers:children]
production

[production]
web1.prod-east.example.com
```

group_vars/production.yml

```
a_conflict: from_production
```

group_vars/webservers.yml

```
a_conflict: from_webservers
```

playbook.yml

```
- name: Test group var precedence
  hosts: production
  tasks:
    - name: Display value of a_conflict
      debug:
        var: a_conflict
```


Red Hat

Inventory Design: Separate Inventories by Environment

- Ideally, you should avoid this sort of conflict between different classes of group (like "type of server", "location", "dev/test/prod env").
- If this does not work for your use case, you can build separate inventories with separate sets of group variables.
- In this case you can run different inventories for different deployment environments:

```
ansible-playbook -i inventories/production/hosts playbook.yml
ansible-playbook -i inventories/testing/hosts playbook.yml
```

- This allows you to have different values for each environment in its **group_vars/webservers.yml** file

```
project
├── inventories
│   ├── production
│   │   ├── group_vars
│   │   │   └── webservers.yml
│   │   └── hosts
│   └── testing
│       ├── group_vars
│       │   └── webservers.yml
│       └── hosts
└── playbook.yml
```

Inventory Design: Conditional Variables

- Another approach is to load some variables in your playbook with vars_files or with the include_vars module.
- This allows you to control the order in which the variables are loaded.
- Both vars_files and include_vars have higher precedence than group variables and will override any values set there.
- In the example at right, if the host is in groups *production* and *webservers*, the value of **a_conflict** set in **vars/production.yml** will be displayed due to include_vars having precedence.

```
project
├── group_vars
│   └── webservers.yml
├── inventory
├── playbook.yml
└── vars
    ├── production.yml
    └── testing.yml
```

```
- name: Example play
  hosts: webservers
  tasks:
   - name: Include production variables
     include_vars:
        file: vars/production.yml
     when: inventory_hostname in groups['production']

   - name: Include testing variables
     include_vars:
       file: vars/testing.yml
     when: inventory_hostname in groups['testing']

   - name: Display aconflict
     debug:
       var: a_conflict
```


Red Hat

Conclusion

Red Hat

Learn More about Red Hat Training and Certification

- Congratulations on completing this course! Want to learn more? Visit the Red Hat Training and Certification page to explore Red Hat courses and certifications.
- Join the Red Hat Learning Community to ask questions and access a collaborative learning environment that enables open source skill development.

Red Hat

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat