Inventory File

Ansible is a modern configuration management tool that facilitates the task of setting up and maintaining remote servers, with a minimalist design intended to get users up and running quickly. Ansible uses an **inventory file** to keep track of which hosts are part of your infrastructure, and how to reach them for running commands and playbooks.

There are multiple ways in which you can set up your Ansible inventory file, depending on your environment and project needs. In this guide, we'll demonstrate how to create inventory files and organize servers into groups and subgroups, how to set up host variables, and how to use patterns to control the execution of Ansible commands and playbooks per host and per group.

Prerequisites

In order follow this guide, you'll need:

- **One Ansible control node**: an Ubuntu 20.04 machine with Ansible installed and configured to connect to your Ansible hosts using SSH keys. Make sure the control node has a regular user with sudo permissions and a firewall enabled, as explained in our Initial Server Setup guide. To set up Ansible, please follow our guide on How to Install and Configure Ansible on Ubuntu 20.04.
- **Two or more Ansible Hosts**: two or more remote Ubuntu 20.04 servers.

Step 1 — Creating a Custom Inventory File

Upon installation, Ansible creates an inventory file that is typically located at `/etc/ansible/hosts`. This is the default location used by Ansible when a custom inventory file is not provided with the `-i` option, during a playbook or command execution.

Even though you can use this file without problems, using per-project inventory files is a good practice to avoid mixing servers when executing commands and playbooks. Having per-project inventory files will also facilitate sharing your provisioning setup with collaborators, given you include the inventory file within the project's code repository.

To get started, access your home folder and create a new directory to hold your Ansible files:

```
cd ~
mkdir ansible
```

Copy

Move to that directory and open a new inventory file using your text editor of choice. Here, we'll use `nano`:

```
cd ansible
nano inventory
```

Copy

A list of your nodes, with one server per line, is enough for setting up a functional inventory file. Hostnames and IP addresses are interchangeable:

~/ansible/inventory

```
203.0.113.111
203.0.113.112
203.0.113.113
server_hostname
```

Copy

Once you have an inventory file set up, you can use the `ansible-inventory` command to validate and obtain information about your Ansible inventory:

```
ansible-inventory -i inventory --list
```

Copy

```
Output{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "ungrouped"
        ]
    },
    "ungrouped": {
        "hosts": [
            "203.0.113.111",
            "203.0.113.112",
            "203.0.113.113",
            "server_hostname"
        ]
    }
}
```

Even though we haven't set up any groups within our inventory, the output shows 2 distinct groups that are automatically inferred by Ansible: `all` and `ungrouped`. As the name suggests, `all` is used to refer to all servers from your inventory file, no matter how they are organized. The `ungrouped` group is used to refer to servers that aren't listed within a group.

Running Commands and Playbooks with Custom Inventories

To run Ansible commands with a custom inventory file, use the `-i` option as follows:

```
ansible all -i inventory -m ping
```

This would execute the `ping` module on **all** hosts listed in your custom inventory file.

Similarly, this is how you execute Ansible playbooks with a custom inventory file:

```
ansible-playbook -i inventory playbook.yml
```

Note: For more information on how to connect to nodes, please refer to our How to Use Ansible guide, as it demonstrates more connection options.

So far, we've seen how to create a basic inventory and how to use it for running commands and playbooks. In the next step, we'll see how to organize nodes into groups and subgroups.

Step 2 — Organizing Servers Into Groups and Subgroups

Within the inventory file, you can organize your servers into different groups and subgroups. Beyond helping to keep your hosts in order, this practice will enable you to use **group variables**, a feature that can greatly facilitate managing multiple staging environments with Ansible.

A host can be part of multiple groups. The following inventory file in INI format demonstrates a setup with four groups: `webservers`, `dbservers`, `development`, and `production`. You'll notice that the servers are grouped by two different qualities: their purpose (web and database), and how they're being used (development and production).

~/ansible/inventory

```
[webservers]
203.0.113.111
203.0.113.112

[dbservers]
203.0.113.113
server_hostname

[development]
203.0.113.111
203.0.113.113

[production]
203.0.113.112
server_hostname
```

If you were to run the `ansible-inventory` command again with this inventory file, you would see the following arrangement:

```
Output{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": [
            "dbservers",
            "development",
            "production",
            "ungrouped",
            "webservers"
        ]
    },
    "dbservers": {
        "hosts": [
            "203.0.113.113",
            "server_hostname"
        ]
    },
    "development": {
        "hosts": [
            "203.0.113.111",
            "203.0.113.113"
        ]
    },
    "production": {
        "hosts": [
            "203.0.113.112",
            "server_hostname"
        ]
    },
    "webservers": {
        "hosts": [
            "203.0.113.111",
            "203.0.113.112"
        ]
    }
}
```

It is also possible to aggregate multiple groups as *children* under a "parent" group. The "parent" is then called a *metagroup*. The following example demonstrates another way to organize the previous inventory using metagroups to achieve a comparable, yet more granular arrangement:

~/ansible/inventory

```
[web_dev]
203.0.113.111

[web_prod]
203.0.113.112

[db_dev]
203.0.113.113

[db_prod]
server_hostname

[webservers:children]
web_dev
web_prod

[dbservers:children]
db_dev
db_prod

[development:children]
web_dev
db_dev

[production:children]
web_prod
db_prod
```

The more servers you have, the more it makes sense to break groups down or create alternative arrangements so that you can target smaller groups of servers as needed.

Step 3 — Setting Up Host Aliases

You can use aliases to name servers in a way that facilitates referencing those servers later, when running commands and playbooks.

To use an alias, include a variable named `ansible_host` after the alias name, containing the corresponding IP address or hostname of the server that should respond to that alias:

~/ansible/inventory

```
server1 ansible_host=203.0.113.111
server2 ansible_host=203.0.113.112
server3 ansible_host=203.0.113.113
server4 ansible_host=server_hostname
```

If you were to run the `ansible-inventory` command with this inventory file, you would see output similar to this:

```
Output{
    "_meta": {
        "hostvars": {
            "server1": {
                "ansible_host": "203.0.113.111"
            },
            "server2": {
                "ansible_host": "203.0.113.112"
            },
            "server3": {
                "ansible_host": "203.0.113.113"
            },
            "server4": {
                "ansible_host": "server_hostname"
            }
        }
    },
    "all": {
        "children": [
            "ungrouped"
        ]
    },
    "ungrouped": {
        "hosts": [
            "server1",
            "server2",
            "server3",
            "server4"
        ]
    }
}
```

Notice how the servers are now referenced by their aliases instead of their IP addresses or hostnames. This makes it easier for targeting individual servers when running commands and playbooks.

Step 4 — Setting Up Host Variables

It is possible to use the inventory file to set up variables that will change Ansible's default behavior when connecting and executing commands on your nodes. This is in fact what we did in the previous step, when setting up host aliases. The `ansible_host` variable tells Ansible where to find the remote nodes, in case an alias is used to refer to that server.

Inventory variables can be set per host or per group. In addition to customizing Ansible's default settings, these variables are also accessible from your playbooks, which enables further customization for individual hosts and groups.

The following example shows how to define the default remote user when connecting to each of the nodes listed in this inventory file:

~/ansible/inventory

```
server1 ansible_host=203.0.113.111 ansible_user=sammy
server2 ansible_host=203.0.113.112 ansible_user=sammy
server3 ansible_host=203.0.113.113 ansible_user=myuser
server4 ansible_host=server_hostname ansible_user=myuser
```

Copy

You could also create a group to aggregate the hosts with similar settings, and then set up their variables at the group level:

~/ansible/inventory

```
[group_a]
server1 ansible_host=203.0.113.111
server2 ansible_host=203.0.113.112

[group_b]
server3 ansible_host=203.0.113.113
server4 ansible_host=server_hostname

[group_a:vars]
ansible_user=sammy

[group_b:vars]
ansible_user=myuser
```

This inventory arrangement would generate the following output with `ansible-inventory`:

```
Output{
    "_meta": {
        "hostvars": {
            "server1": {
                "ansible_host": "203.0.113.111",
                "ansible_user": "sammy"
            },
            "server2": {
                "ansible_host": "203.0.113.112",
                "ansible_user": "sammy"
            },
            "server3": {
                "ansible_host": "203.0.113.113",
                "ansible_user": "myuser"
            },
            "server4": {
                "ansible_host": "server_hostname",
                "ansible_user": "myuser"
            }
        }
    },
    "all": {
        "children": [
            "group_a",
            "group_b",
            "ungrouped"
        ]
    },
    "group_a": {
        "hosts": [
            "server1",
            "server2"
        ]
    },
    "group_b": {
        "hosts": [
            "server3",
            "server4"
        ]
    }
}
```

Notice that all inventory variables are listed within the `_meta` node in the JSON output produced by `ansible-inventory`.

Step 5 — Using Patterns to Target Execution of Commands and Playbooks

When executing commands and playbooks with Ansible, you must provide a target. *Patterns* allow you to target specific hosts, groups, or subgroups in your inventory file. They're very flexible, supporting regular expressions and wildcards.

Consider the following inventory file:

~/ansible/inventory

```
[webservers]
203.0.113.111
203.0.113.112

[dbservers]
203.0.113.113
server_hostname

[development]
203.0.113.111
203.0.113.113

[production]
203.0.113.112
server_hostname
```

Now imagine you need to execute a command targeting only the database server(s) that are running on production. In this example, there's only `server_hostname` matching that criteria; however, it could be the case that you have a large group of database servers in that group. Instead of individually targeting each server, you could use the following pattern:

```
ansible dbservers:&production -m ping
```

The `&` character represents the logical operation `AND`, meaning that valid targets must be in both groups. Because this is an ad hoc command running on Bash, we must include the escape character in the expression.

The previous example would target only servers that are present both in the `dbservers` as well as in the `production` groups. If you wanted to do the opposite, targeting only servers that are present in the `dbservers` but **not** in the `production` group, you would use the following pattern instead:

```
ansible dbservers:!production -m ping
```

To indicate that a target must **not** be in a certain group, you can use the `!` character. Once again, we include the escape character in the expression to avoid command line errors, since both `&` and `!` are special characters that can be parsed by Bash.

The following table contains a few different examples of common patterns you can use when running commands and playbooks with Ansible:

Pattern	Result Target
`all`	All Hosts from your inventory file
`host1`	A single host (`host1`)
`host1:host2`	Both `host1` and `host2`
`group1`	A single group (`group1`)
`group1:group2`	All servers in `group1` and `group2`
`group1:&group2`	Only servers that are **both** in `group1` and `group2`
`group1:!group2`	Servers in `group1` **except** those also in `group2`

For more advanced pattern options, such as using positional patterns and regex to define targets, please refer to the official Ansible documentation on patterns.

Conclusion

In this guide, we had a detailed look into Ansible inventories. We've seen how to organize nodes into groups and subgroups, how to set up inventory variables, and how to use patterns to target different groups of servers when running commands and playbooks.

In the next part of this series, we'll see how to manage multiple servers with Ansible ad-hoc commands.