ERIKA HEIDI

HOW TO MANAGE REMOTE SERVERS WITH ANSIBLE

ISBN 978-1-7358317-3-2

How To Manage Remote Servers with Ansible

Erika Heidi

DigitalOcean, New York City, New York, USA

2020-11

How To Manage Remote Servers with Ansible

About DigitalOcean

Introduction

About this Book

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, it allows you to control one to hundreds of systems from a central location with either playbooks or ad hoc commands.

This series goes over how to use Ansible to manage remote servers, and how to write your own playbooks to automate your server setup.

An Introduction to Configuration Management with Ansible

Written by Erika Heidi

Introduction

Configuration management is the process of handling changes to a system in a way that assures integrity over time, typically involving tools and processes that facilitate automation and observability. Even though this concept didn't originate in the IT industry, the term is broadly used to refer to **server configuration management**.

In the context of servers, configuration management is also commonly referred to as IT Automation or Server Orchestration. Both terms highlight the practical aspects of configuration management and the ability to control multiple systems from a central server.

This guide will walk you through the benefits of using a configuration management tool to automate your server infrastructure setup, and how one such tool, Ansible, can help you with that.

Benefits of Using a Configuration Management Tool

There are a number of configuration management tools available on the market, with varying levels of complexity and diverse architectural styles. Although each of these tools have their own characteristics and work in slightly different ways, they all provide the same function: make sure a system's state matches the state described by a set of provisioning scripts.

Many of the benefits of configuration management for servers come from the ability to define your infrastructure as code. This enables you to:

- Use a version control system to keep track of any changes in your infrastructure
- Reuse provisioning scripts for multiple server environments, such as development, testing, and production
- Share provisioning scripts between coworkers to facilitate collaboration in a standardised development environment
- Streamline the process of replicating servers, which facilitates recovery from critical errors

Additionally, configuration management tools offer you a way to control one to hundreds of servers from a centralized location, which can dramatically improve efficiency and integrity of your server infrastructure.

Ansible Overview

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.

Users write Ansible provisioning scripts in YAML, a user-friendly data serialization standard that is not tied to any particular programming language. This enables users to create sophisticated provisioning scripts more intuitively compared to similar tools in the same category.

Ansible doesn't require any special software to be installed on the nodes that will be managed with this tool. A control machine is set up with the

Ansible software, which then communicates with the nodes via standard SSH.

As a configuration management tool and automation framework, Ansible encapsulates all of the common features present in other tools of the same category, while still maintaining a strong focus on simplicity and performance:

Idempotent Behavior

Ansible keeps track of the state of resources in managed systems in order to avoid repeating tasks that were executed before. If a package was already installed, it won’t try to install it again. The objective is that after each provisioning execution the system reaches (or keeps) the desired state, even if you run it multiple times. This is what characterizes Ansible and other configuration management tools as having an idempotent behavior. When running a playbook, you’ll see the status of each task being executed and whether or not the task performed a change in the system.

Support to Variables, Conditionals, and Loops

When writing Ansible automation scripts, you can use variables, conditionals, and loops in order to make your automation more versatile and efficient.

System Facts

Ansible collects a series of detailed information about the managed nodes, such as network interfaces and operating system, and provides it as global variables called system facts. Facts can be used within playbooks to make your automation more versatile and adaptive, behaving differently depending on the system being provisioned.

Templating System

Ansible uses the Jinja2 Python templating system to allow for dynamic expressions and access to variables. Templates can be used to facilitate setting up configuration files and services. For instance, you can use a template to set up a new virtual host within Apache, while reusing the same template for multiple server installations.

Support for Extensions and Modules

Ansible comes with hundreds of built-in modules to facilitate writing automation for common systems administration tasks, such as installing packages with `apt` and synchronizing files with `rsync`, and also for dealing with popular software such as database systems (like MySQL, PostgreSQL, MongoDB, and others) and dependency management tools (such as PHP's `composer`, Ruby's `gem`, Node's `npm`, and others). Apart from that, there are various ways in which you can extend Ansible: plugins and modules are good options when you need a custom functionality that is not present by default.

You can also find third-party modules and plugins in the Ansible Galaxy portal.

Getting Familiar with Ansible Concepts

We’ll now have a look at Ansible terminology and concepts to help familiarize you with these terms as they come up throughout this series.

Control Node

A control node is a system where Ansible is installed and set up to connect to your server. You can have multiple control nodes, and any system capable of running Ansible can be set up as a control node, including personal computers or laptops running a Linux or Unix based operating system. For the time being, Ansible can’t be installed on Windows hosts, but you can circumvent this limitation by setting up a virtual machine that runs Linux and running Ansible from there.

Managed Nodes

The systems you control using Ansible are called managed nodes. Ansible requires that managed nodes are reachable via SSH, and have Python 2 (version 2.6 or higher) or Python 3 (version 3.5 or higher) installed.

Ansible supports a variety of operating systems including Windows servers as managed nodes.

Inventory

An inventory file contains a list of the hosts you’ll manage using Ansible. Although Ansible typically creates a default inventory file when installed, you can use per-project inventories to have a better separation of your infrastructure and avoid running commands or playbooks on the wrong server by mistake. Static inventories are usually created as `.ini` files, but you can also use dynamically generated inventories written in any programming language able to return JSON.

Tasks

In Ansible, a task is an individual unit of work to execute on a managed node. Each action to perform is defined as a task. Tasks can be executed as a one-off action via ad-hoc commands, or included in a playbook as part of an automation script.

Playbook

A playbook contains an ordered list of tasks, and a few other directives to indicate which hosts are the target of that automation, whether or not to use a privilege escalation system to run those tasks, and optional sections to define variables or include files. Ansible executes tasks sequentially, and a full playbook execution is called a play. Playbooks are written in YAML format.

Handlers

Handlers are used to perform actions on a service, such as restarting or stopping a service that is actively running on the managed node's system. Handlers are typically triggered by tasks, and their execution happens at the end of a play, after all tasks are finished. This way, if more than one task triggers a restart to a service, for instance, the service will only be restarted once and after all tasks are executed. Although the default handler behavior is more efficient and overall a better practice, it is also possible to force immediate handler execution if that is required by a task.

Roles

A role is a set of playbooks and related files organized into a predefined structure that is known by Ansible. Roles facilitate reusing and repurposing playbooks into shareable packages of granular automation for specific goals, such as installing a web server, installing a PHP environment, or setting up a MySQL server.

Conclusion

Ansible is a minimalist IT automation tool that has a gentle learning curve, thanks in part to its use of YAML for its provisioning scripts. It has a great number of built-in modules that can be used to abstract tasks such as installing packages and working with templates. Its simplified infrastructure requirements and accessible syntax can be a good fit for those who are getting started with configuration management.

In the next part of this series, we'll see how to install and get started with Ansible on an Ubuntu 20.04 server.

How To Install and Configure Ansible on Ubuntu 20.04

Written by Erika Heidi

Introduction

Configuration management systems are designed to streamline the process of controlling large numbers of servers, for administrators and operations teams. They allow you to control many different systems in an automated way from one central location.

While there are many popular configuration management tools available for Linux systems, such as Chef and Puppet, these are often more complex than many people want or need. Ansible is a great alternative to these options because it offers an architecture that doesn't require special software to be installed on nodes, using SSH to execute the automation tasks and YAML files to define provisioning details.

In this guide, we'll discuss how to install Ansible on an Ubuntu 20.04 server and go over some basics of how to use this software. For a more high-level overview of Ansible as configuration management tool, please refer to An Introduction to Configuration Management with Ansible.

Prerequisites

To follow this tutorial, you will need:

- **One Ansible Control Node**: The Ansible control node is the machine we'll use to connect to and control the Ansible hosts over SSH. Your Ansible control node can either be your local machine or a server dedicated to running Ansible, though this guide assumes your control node is an Ubuntu 20.04 system. Make sure the control node has:

    - A non-root user with sudo privileges. To set this up, you can follow **Steps 2 and 3** of our Initial Server Setup Guide for Ubuntu 20.04. However, please note that if you're using a remote server as your Ansible Control node, you should follow **every step** of this guide. Doing so will configure a firewall on the server with `ufw` and enable external access to your non-root user profile, both of which will help keep the remote server secure.
    - An SSH keypair associated with this user. To set this up, you can follow **Step 1** of our guide on How to Set Up SSH Keys on Ubuntu 20.04.

- **One or more Ansible Hosts**: An Ansible host is any machine that your Ansible control node is configured to automate. This guide assumes your Ansible hosts are remote Ubuntu 20.04 servers. Make sure each Ansible host has:

    - The Ansible control node's SSH public key added to the `authorized_keys` of a system user. This user can be either **root** or a regular user with sudo privileges. To set this up, you can follow **Step 2** of How to Set Up SSH Keys on Ubuntu 20.04.

Step 1 — Installing Ansible

To begin using Ansible as a means of managing your server infrastructure, you need to install the Ansible software on the machine that will serve as the Ansible control node. We'll use the default Ubuntu repositories for that.

First, refresh your system's package index with:

```
sudo apt update
```

Following this update, you can install the Ansible software with:

```
sudo apt install ansible
```

Press `Y` when prompted to confirm installation.

Your Ansible control node now has all of the software required to administer your hosts. Next, we'll go over how to set up an inventory file, so that Ansible can communicate with your managed nodes.

Step 2 — Setting Up the Inventory File

The inventory file contains information about the hosts you'll manage with Ansible. You can include anywhere from one to several hundred servers in your inventory file, and hosts can be organized into groups and subgroups. The inventory file is also often used to set variables that will be valid only for specific hosts or groups, in order to be used within playbooks and templates. Some variables can also affect the way a playbook is run, like the `ansible_python_interpreter` variable that we'll see in a moment.

To edit the contents of your default Ansible inventory, open the `/etc/ansible/hosts` file using your text editor of choice, on your Ansible control node:

```
sudo nano /etc/ansible/hosts
```

Note: Although Ansible typically creates a default inventory file at `/etc/ansible/hosts`, you are free to create inventory files in any location that better suits your needs. In this case, you'll need to provide the path to your custom inventory file with the `-i` parameter when running Ansible commands and playbooks. Using per-project inventory files is a good practice to minimize the risk of running a playbook on the wrong group of servers.

The default inventory file provided by the Ansible installation contains a number of examples that you can use as references for setting up your inventory. The following example defines a group named `[servers]` with three different servers in it, each identified by a custom alias: **server1**, **server2**, and **server3**. Be sure to replace the highlighted IPs with the IP addresses of your Ansible hosts.

/etc/ansible/hosts

```
[servers]
server1 ansible_host=203.0.113.111
server2 ansible_host=203.0.113.112
server3 ansible_host=203.0.113.113

[all:vars]
ansible_python_interpreter=/usr/bin/python3
```

The `all:vars` subgroup sets the `ansible_python_interpreter` host parameter that will be valid for all hosts included in this inventory. This parameter makes sure the remote server uses the `/usr/bin/python3` Python 3 executable instead of `/usr/bin/python` (Python 2.7), which is not present on recent Ubuntu versions.

When you're finished, save and close the file by pressing `CTRL+X` then `Y` and `ENTER` to confirm your changes.

Whenever you want to check your inventory, you can run:

```
ansible-inventory --list -y
```

You'll see output similar to this, but containing your own server infrastructure as defined in your inventory file:

```
Output

all:
  children:
    servers:
      hosts:
        server1:
          ansible_host: 203.0.113.111
          ansible_python_interpreter: /usr/bin/python3
        server2:
          ansible_host: 203.0.113.112
          ansible_python_interpreter: /usr/bin/python3
        server3:
          ansible_host: 203.0.113.113
          ansible_python_interpreter: /usr/bin/python3
    ungrouped: {}
```

Now that you've configured your inventory file, you have everything you need to test the connection to your Ansible hosts.

Step 3 — Testing Connection

After setting up the inventory file to include your servers, it's time to check if Ansible is able to connect to these servers and run commands via SSH.

For this guide, we'll be using the Ubuntu **root** account because that's typically the only account available by default on newly created servers. If your Ansible hosts already have a regular sudo user created, you are encouraged to use that account instead.

You can use the `-u` argument to specify the remote system user. When not provided, Ansible will try to connect as your current system user on the control node.

From your local machine or Ansible control node, run:

```
ansible all -m ping -u root
```

This command will use Ansible's built-in `ping` module to run a connectivity test on all nodes from your default inventory, connecting as **root**. The `ping` module will test:

- if hosts are accessible;
- if you have valid SSH credentials;
- if hosts are able to run Ansible modules using Python.

You should get output similar to this:

```
Output

server1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
server2 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
server3 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

If this is the first time you're connecting to these servers via SSH, you'll be asked to confirm the authenticity of the hosts you're connecting to via Ansible. When prompted, type `yes` and then hit `ENTER` to confirm.

Once you get a `"pong"` reply back from a host, it means you're ready to run Ansible commands and playbooks on that server.

Note: If you are unable to get a successful response back from your servers, check our Ansible Cheat Sheet Guide for more information on how to run Ansible commands with different connection options.

Step 4 — Running Ad-Hoc Commands (Optional)

After confirming that your Ansible control node is able to communicate with your hosts, you can start running ad-hoc commands and playbooks on your servers.

Any command that you would normally execute on a remote server over SSH can be run with Ansible on the servers specified in your inventory file. As an example, you can check disk usage on all servers with:

```
ansible all -a "df -h" -u root
```

```
Output

server1 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G     0  3.9G   0% /dev
tmpfs           798M  624K  798M   1% /run
/dev/vda1       155G  2.3G  153G   2% /
tmpfs           3.9G     0  3.9G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/vda15      105M  3.6M  101M   4% /boot/efi
tmpfs           798M     0  798M   0% /run/user/0

server2 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
udev            2.0G     0  2.0G   0% /dev
tmpfs           395M  608K  394M   1% /run
/dev/vda1        78G  2.2G   76G   3% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           2.0G     0  2.0G   0% /sys/fs/cgroup
/dev/vda15      105M  3.6M  101M   4% /boot/efi
tmpfs           395M     0  395M   0% /run/user/0

...
```

The highlighted command `df -h` can be replaced by any command you'd like.

You can also execute Ansible modules via ad-hoc commands, similarly to what we've done before with the `ping` module for testing connection. For example, here's how we can use the `apt` module to install the latest version of `vim` on all the servers in your inventory:

```
ansible all -m apt -a "name=vim state=latest" -u root
```

You can also target individual hosts, as well as groups and subgroups, when running Ansible commands. For instance, this is how you would check the `uptime` of every host in the `servers` group:

```
ansible servers -a "uptime" -u root
```

We can specify multiple hosts by separating them with colons:

```
ansible server1:server2 -m ping -u root
```

For more information on how to use Ansible, including how to execute playbooks to automate server setup, you can check our Ansible Reference Guide.

Conclusion

In this guide, you've installed Ansible and set up an inventory file to execute ad-hoc commands from an Ansible Control Node.

Once you’ve confirmed you’re able to connect and control your infrastructure from a central Ansible controller machine, you can execute any command or playbook you desire on those hosts.

For more information on how to use Ansible, check out our Ansible Cheat Sheet Guide.

How To Set Up Ansible Inventories

Written by Erika Heidi

Introduction

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
```

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
```

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
```

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
```

```
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
```

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.

How To Manage Multiple Servers with Ansible Ad Hoc Commands

Written by Erika Heidi

Introduction

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, it allows you to control one to hundreds of systems from a central location with either playbooks or ad hoc commands.

Unlike playbooks — which consist of collections of tasks that can be reused — ad hoc commands are tasks that you don't perform frequently, such as restarting a service or retrieving information about the remote systems that Ansible manages.

In this cheat sheet guide, you'll learn how to use Ansible ad hoc commands to perform common tasks such as installing packages, copying files, and restarting services on one or more remote servers, from an Ansible control node.

Prerequisites

In order to follow this guide, you'll need:

- **One Ansible control node**. This guide assumes your control node is 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**. An Ansible host is any machine that your Ansible control node is configured to automate. This guide assumes your Ansible hosts are remote Ubuntu 20.04 servers. Make sure each Ansible host has:

  - The Ansible control node's SSH public key added to the `authorized_keys` of a system user. This user can be either root or a regular user with sudo privileges. To set this up, you can follow Step 2 of How to Set Up SSH Keys on Ubuntu 20.04.

- **An inventory file set up on the Ansible control node**. Make sure you have a working inventory file containing all your Ansible hosts. To set this up, please refer to the guide on How To Set Up Ansible Inventories. Then, make sure you're able to connect to your nodes by running the connection test outlined in the section Testing Connection to Ansible Hosts.

Testing Connection to Ansible Hosts

The following command will test connectivity between your Ansible control node and all your Ansible hosts. This command uses the current system user and its corresponding SSH key as the remote login, and includes the `-m` option, which tells Ansible to run the `ping` module. It also features the `-i` flag, which tells Ansible to ping the hosts listed in the specified `inventory` file

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

If this is the first time you're connecting to these servers via SSH, you'll be asked to confirm the authenticity of the hosts you're connecting to via Ansible. When prompted, type `yes` and then hit `ENTER` to confirm.

You should get output similar to this:

```
Output

server1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
server2 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Once you get a `"pong"` reply back from a host, it means the connection is live and you're ready to run Ansible commands on that server.

Adjusting Connection Options

By default, Ansible tries to connect to the nodes as a remote user with the same name as your current system user, using its corresponding SSH keypair.

To connect as a different remote user, append the command with the `-u` flag and the name of the intended user:

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

If you’re using a custom SSH key to connect to the remote servers, you can provide it at execution time with the `--private-key` option:

```
ansible all -i inventory -m ping --private-
key=~/.ssh/custom_id
```

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

Once you’re able to connect using the appropriate options, you can adjust your inventory file to automatically set your remote user and private key, in case they are different from the default values assigned by Ansible. Then, you won’t need to provide those parameters in the command line.

The following example inventory file sets up the `ansible_user` variable only for the `server1` server:

```
~/ansible/inventory

server1 ansible_host=203.0.113.111 ansible_user=sammy
server2 ansible_host=203.0.113.112
```

Ansible will now use **sammy** as the default remote user when connecting to the `server1` server.

To set up a custom SSH key, include the `ansible_ssh_private_key_file` variable as follows:

```
~/ansible/inventory

server1 ansible_host=203.0.113.111
ansible_ssh_private_key_file=/home/sammy/.ssh/custom_id
server2 ansible_host=203.0.113.112
```

In both cases, we have set up custom values only for `server1`. If you want to use the same settings for multiple servers, you can use a child group for that:

```
~/ansible/inventory
[group_a]
203.0.113.111
203.0.113.112

[group_b]
203.0.113.113



[group_a:vars]
ansible_user=sammy
ansible_ssh_private_key_file=/home/sammy/.ssh/custom_id
```

This example configuration will assign a custom user and SSH key only for connecting to the servers listed in `group_a`.

Defining Targets for Command Execution

When running ad hoc commands with Ansible, you can target individual hosts, as well as any combination of groups, hosts and subgroups. For instance, this is how you would check connectivity for every host in a group named `servers`:

```
ansible servers -i inventory -m ping
```

You can also specify multiple hosts and groups by separating them with colons:

```
ansible server1:server2:dbservers -i inventory -m ping
```

To include an exception in a pattern, use an exclamation mark, prefixed by the escape character `\`, as follows. This command will run on all servers from `group1`, except `server2`:

```
ansible group1:\!server2 -i inventory -m ping
```

In case you'd like to run a command only on servers that are part of both `group1` and `group2`, for instance, you should use `&` instead. Don't forget to prefix it with a `\` escape character:

```
ansible group1:\&group2 -i inventory -m ping
```

For more information on how to use patterns when defining targets for command execution, please refer to Step 5 of our guide on How to Set Up Ansible Inventories. ## Running Ansible Modules

Ansible modules are pieces of code that can be invoked from playbooks and also from the command-line to facilitate executing procedures on remote nodes. Examples include the `apt` module, used to manage system packages on Ubuntu, and the `user` module, used to manage system users. The `ping` command used throughout this guide is also a module, typically used to test connection from the control node to the hosts.

Ansible ships with an extensive collection of built-in modules, some of which require the installation of additional software in order to provide full functionality. You can also create your own custom modules using your language of choice.

To execute a module with arguments, include the `-a` flag followed by the appropriate options in double quotes, like this:

```
ansible target -i inventory -m module -a "module options"
```

As an example, this will use the `apt` module to install the package `tree` on `server1`:

```
ansible server1 -i inventory -m apt -a "name=tree"
```

Running Bash Commands

When a module is not provided via the `-m` option, the command module is used by default to execute the specified command on the remote server(s).

This allows you to execute virtually any command that you could normally execute via an SSH terminal, as long as the connecting user has sufficient permissions and there aren't any interactive prompts.

This example executes the `uptime` command on all servers from the specified inventory:

```
ansible all -i inventory -a "uptime"
```

```
Output

server1 | CHANGED | rc=0 >>
 14:12:18 up 55 days,  2:15,  1 user,  load average: 0.03,
0.01, 0.00
server2 | CHANGED | rc=0 >>
 14:12:19 up 10 days,  6:38,  1 user,  load average: 0.01,
0.02, 0.00
```

Using Privilege Escalation to Run Commands with `sudo`

If the command or module you want to execute on remote hosts requires extended system privileges or a different system user, you'll need to use Ansible's privilege escalation module, become. This module is an abstraction for `sudo` as well as other privilege escalation software supported by Ansible on different operating systems.

For instance, if you wanted to run a `tail` command to output the latest log messages from Nginx's error log on a server named `server1` from `inventory`, you would need to include the `--become` option as follows:

```
ansible server1 -i inventory -a "tail
/var/log/nginx/error.log" --become
```

This would be the equivalent of running a `sudo tail /var/log/nginx/error.log` command on the remote host, using the current local system user or the remote user set up within your inventory file.

Privilege escalation systems such as `sudo` often require that you confirm your credentials by prompting you to provide your user's password. That would cause Ansible to fail a command or playbook execution. You can then use the `--ask-become-pass` or `-K` option to make Ansible prompt you for that `sudo` password:

```
ansible server1 -i inventory -a "tail
/var/log/nginx/error.log" --become -K
```

Installing and Removing Packages

The following example uses the `apt` module to install the `nginx` package on all nodes from the provided inventory file:

```
ansible all -i inventory -m apt -a "name=nginx" --become -K
```

To remove a package, include the `state` argument and set it to `absent`:.

```
ansible all -i inventory -m apt -a "name=nginx state=absent" -
-become  -K
```

Copying Files

With the `file` module, you can copy files between the control node and the managed nodes, in either direction. The following command copies a local text file to all remote hosts in the specified inventory file:

```
ansible all -i inventory -m copy -a "src=./file.txt
dest=~/myfile.txt"
```

To copy a file from the remote server to your control node, include the `remote_src` option:

```
ansible all -i inventory -m copy -a "src=~/myfile.txt
remote_src=yes dest=./file.txt"
```

Changing File Permissions

To modify permissions on files and directories on your remote nodes, you can use the `file` module.

The following command will adjust permissions on a file named `file.txt` located at `/var/www` on the remote host. It will set the file's umask to `600`, which will enable read and write permissions only for the current file owner. Additionally, it will set the ownership of that file to a user and a group called `sammy`:

```
ansible all -i inventory -m file -a "dest=/var/www/file.txt
mode=600 owner=sammy group=sammy" --become  -K
```

Because the file is located in a directory typically owned by `root`, we might need `sudo` permissions to modify its properties. That's why we include the `--become` and `-K` options. These will use Ansible's privilege escalation system to run the command with extended privileges, and it will prompt you to provide the `sudo` password for the remote user.

Restarting Services

You can use the `service` module to manage services running on the remote nodes managed by Ansible. This will require extended system privileges, so make sure your remote user has sudo permissions and you include the `--become` option to use Ansible's privilege escalation system. Using `-K` will prompt you to provide the `sudo` password for the connecting user.

To restart the `nginx` service on all hosts in group called `webservers`, for instance, you would run:

```
ansible webservers -i inventory -m service -a "name=nginx
state=restarted" --become  -K
```

Restarting Servers

Although Ansible doesn't have a dedicated module to restart servers, you can issue a bash command that calls the `/sbin/reboot` command on the remote host.

Restarting the server will require extended system privileges, so make sure your remote user has sudo permissions and you include the `--become` option to use Ansible's privilege escalation system. Using `-K` will prompt you to provide the `sudo` password for the connecting user.

Warning: The following command will fully restart the server(s) targeted by Ansible. That might cause temporary disruption to any applications that rely on those servers.

To restart all servers in a `webservers` group, for instance, you would run:

```
ansible webservers -i inventory -a "/sbin/reboot"  --become  -
K
```

Gathering Information About Remote Nodes

The `setup` module returns detailed information about the remote systems managed by Ansible, also known as system facts.

To obtain the system facts for `server1`, run:

```
ansible server1 -i inventory -m setup
```

This will print a large amount of JSON data containing details about the remote server environment. To print only the most relevant information, include the `"gather_subset=min"` argument as follows:

```
ansible server1 -i inventory -m setup -a "gather_subset=min"
```

To print only specific items of the JSON, you can use the `filter` argument. This will accept a wildcard pattern used to match strings, similar to `fnmatch`. For example, to obtain information about both the ipv4 and ipv6 network interfaces, you can use `*ipv*` as filter:

```
ansible server1 -i inventory -m setup -a "filter=*ipv*"
```

Output

```
server1 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "203.0.113.111",
            "10.0.0.1"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a4f5:16ff:fe75:e758"
        ],
        "ansible_default_ipv4": {
            "address": "203.0.113.111",
            "alias": "eth0",
            "broadcast": "203.0.113.111",
            "gateway": "203.0.113.1",
            "interface": "eth0",
            "macaddress": "a6:f5:16:75:e7:58",
            "mtu": 1500,
            "netmask": "255.255.240.0",
            "network": "203.0.113.0",
            "type": "ether"
        },
        "ansible_default_ipv6": {}
    },
    "changed": false
}
```

If you’d like to check disk usage, you can run a Bash command calling the `df` utility, as follows:

```
ansible all -i inventory -a "df -h"
```

```
Output



server1 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G     0  3.9G   0% /dev
tmpfs           798M  624K  798M   1% /run
/dev/vda1       155G  2.3G  153G   2% /
tmpfs           3.9G     0  3.9G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/vda15      105M  3.6M  101M   4% /boot/efi
tmpfs           798M     0  798M   0% /run/user/0

server2 | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
udev            2.0G     0  2.0G   0% /dev
tmpfs           395M  608K  394M   1% /run
/dev/vda1        78G  2.2G   76G   3% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           2.0G     0  2.0G   0% /sys/fs/cgroup
/dev/vda15      105M  3.6M  101M   4% /boot/efi
tmpfs           395M     0  395M   0% /run/user/0
```

Conclusion

In this guide, we demonstrated how to use Ansible ad hoc commands to manage remote servers, including how to execute common tasks such as restarting a service or copying a file from the control node to the remote servers managed by Ansible. We've also seen how to gather information from the remote nodes using limiting and filtering parameters.

As an additional resource, you can check Ansible's official documentation on ad hoc commands.

How To Execute Ansible Playbooks to Automate Server Setup

Written by Erika Heidi

Introduction

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, it allows you to control one to hundreds of systems from a central location with either playbooks or ad hoc commands.

While ad hoc commands allow you to run one-off tasks on servers registered within your inventory file, playbooks are typically used to automate a sequence of tasks for setting up services and deploying applications to remote servers. Playbooks are written in YAML, and can contain one or more plays.

This short guide demonstrates how to execute Ansible playbooks to automate server setup, using an example playbook that sets up an Nginx server with a single static HTML page.

Prerequisites

In order to follow this guide, you'll need:

- **One Ansible control node**. This guide assumes your control node is 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.

- **One or more Ansible hosts**. An Ansible host is any machine that your Ansible control node is configured to automate. This guide assumes your Ansible hosts are remote Ubuntu 20.04 servers. Make sure each Ansible host has:

  - The Ansible control node's SSH public key added to the `authorized_keys` of a system user. This user can be either root or a regular user with sudo privileges. To set this up, you can follow Step 2 of How to Set Up SSH Keys on Ubuntu 20.04.

- **An inventory file set up on the Ansible control node**. Make sure you have a working inventory file containing all your Ansible hosts. To set this up, please refer to the guide on How To Set Up Ansible Inventories.

Once you have met these prerequisites, run a connection test as outlined in our guide on How To Manage Multiple Servers with Ansible Ad Hoc Commands to make sure you're able to connect and execute Ansible instructions on your remote nodes. In case you don't have a playbook already available to you, you can create a testing playbook as described in the next section.

Creating a Test Playbook

To try out the examples described in this guide, you'll need an Ansible playbook. We'll set up a testing playbook that installs Nginx and sets up an `index.html` page on the remote server. This file will be copied from the Ansible control node to the remote nodes in your inventory file.

Create a new file called `playbook.yml` in the same directory as your inventory file. If you followed our guide on how to create inventory files, this should be a folder called `ansible` inside your home directory:

```
cd ~/ansible
nano playbook.yml
```

The following playbook has a single play and runs on all hosts from your inventory file, by default. This is defined by the `hosts: all` directive at the beginning of the file. The `become` directive is then used to indicate that the following tasks must be executed by a super user (`root` by default).

It defines two tasks: one to install required system packages, and the other one to copy an `index.html` file to the remote host, and save it in Nginx's default document root location, `/var/www/html`. Each task has tags, which can be used to control the playbook's execution.

Copy the following content to your `playbook.yml` file:

~/ansible/playbook.yml

```yaml
---
- hosts: all
  become: true
  tasks:
    - name: Install Packages
      apt: name={{ item }} update_cache=yes state=latest
      loop: [ 'nginx', 'vim' ]
      tags: [ 'setup' ]

    - name: Copy index page
      copy:
        src: index.html
        dest: /var/www/html/index.html
        owner: www-data
        group: www-data
        mode: '0644'
      tags: [ 'update', 'sync' ]
```

Save and close the file when you're done. Then, create a new `index.html` file in the same directory, and place the following content in it:

~/ansible/index.html

```
<html>
    <head>
        <title>Testing Ansible Playbooks</title>
    </head>
    <body>
        <h1>Testing Ansible Playbooks</h1>
        <p>This server was set up using an Nginx playbook.</p>
    </body>
</html>
```

Don’t forget to save and close the file.

Executing a Playbook

To execute the testing playbook on all servers listed within your inventory file, which we’ll refer to as `inventory` throughout this guide, you may use the following command:

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

This will use the current system user as remote SSH user, and the current system user’s SSH key to authenticate to the nodes. In case those aren’t the correct credentials to access the server, you’ll need to include a few other parameters in the command, such as `-u` to define the remote user or `--private-key` to define the correct SSH keypair you want to use to connect. If your remote user requires a password for running commands with `sudo`, you’ll need to provide the `-K` option so that Ansible prompts you for the `sudo` password.

More information about connection options is available in our Ansible Cheatsheet guide.

Listing Playbook Tasks

In case you'd like to list all tasks contained in a playbook, without executing any of them, you may use the `--list-tasks` argument:

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

```
Output

playbook: nginx.yml

  play #1 (all): all    TAGS: []
    tasks:
      Install Packages  TAGS: [setup]
      Copy index page   TAGS: [sync, update]
```

Listing Playbook Tags

Tasks often have tags that allow you to have extended control over a playbook's execution. To list current available tags in a playbook, you can use the `--list-tags` argument as follows:

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

```
Output

playbook: nginx.yml

  play #1 (all): all    TAGS: []
      TASK TAGS: [setup, sync, update]
```

Executing Tasks by Tag

To only execute tasks that are marked with specific tags, you can use the `--tags` argument, along with the tags that you want to trigger:

```
ansible-playbook -i inventory playbook.yml --tags=setup
```

Skipping Tasks by Tag

To skip tasks that are marked with certain tags, you may use the `--exclude-tags` argument, along with the names of tags that you want to exclude from execution:

```
ansible-playbook -i inventory playbook.yml --exclude-
tags=setup
```

Starting Execution at Specific Task

Another way to control the execution flow of a playbook is by starting the play at a certain task. This is useful when a playbook execution finishes prematurely, in which case you might want to run a retry.

```
ansible-playbook -i inventory playbook.yml --start-at-
task=Copy index page
```

Limiting Targets for Execution

Many playbooks set up their target as `all` by default, and sometimes you want to limit the group or single server that should be the target for that setup. You can use `-l` (**l**imit) to set up the target group or server in that play:

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

Controlling Output Verbosity

If you run into errors while executing Ansible playbooks, you can increase output verbosity in order to get more information about the problem you're experiencing. You can do that by including the -v option to the command:

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

If you need more detail, you can use `-vv` or `-vvv` instead. If you’re unable to connect to the remote nodes, use `-vvvv` to obtain connection debugging information:

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

Conclusion

In this guide, you’ve learned how to execute Ansible playbooks to automate server setup. We’ve also seen how to obtain information about playbooks, how to manipulate a playbook’s execution flow using tags, and how to adjust output verbosity in order to obtain detailed debugging information in a play.