How to work with Ansible?

## About Ansible

* Ansible is an open source tool (with enterprise editions available) developed using Python and runs on Windows, Mac, and UNIX-like systems. You can use Ansible for configuration management, orchestration, provisioning, and deployments, which covers many of the problems that are solved under the broad umbrella of **DevOps.**
* The architecture of Ansible is agentless and works purely on SSH connections. This also means that you can install it only on one system (either a Linux or Mac machine) and you can control your entire infrastructure from that machine.
* It is recommended to have multiple such machines in production, but this was just an example to elucidate the simplicity of Ansible. You could even run some of these machines from where you kick off Ansible scripts in a Demilitarized Zone (DMZ) to deal with your production machines.

**Agentless systems specifications:**

-No specific agent or third-party dependencies are installed on these systems. However, you need an SSH daemon that's up and running, in most cases.

- These systems invoke the run remotely.

- Parallel agent runs might be faster than when all agents are contacting the same machine, but they might be constrained by the number of SSH connections since the runs are being invoked remotely.

- Remote connections can log in as a specific user and with the right level of user support since it's SSH-based.

* Ansible primarily runs in the push mode but you can also run Ansible using **ansible-pull**, where you can install Ansible on each agent, download the playbooks locally, and run them on individual machines.
* To speedup default SSH connections, you can always enable **ControlPersist** and the pipeline mode, which makes Ansible faster and secure. Ansible works like any other UNIX command that doesn't require any daemon process to be running all the time.
* To deploy a package on one or more machines in Ansible, you would need to write a playbook that has a single task, which in turn uses the **package** module that would then go ahead and install the package based on an inventory file that contains a list of these machines. Similar to the **package** module, Ansible comes loaded with more than 200 modules, purely written in Python.

## Installing Ansible

* Installing Ansible is rather quick and simple. You can directly use the source code by cloning it from the GitHub project (<https://github.com/ansible/ansible>), install it using your system's package manager, or use Python's package management tool (**pip**).

Note:

*Note that, as Ansible is developed using Python, you would need Python Version 2.4 or a higher version installed. Python is preinstalled, as specified earlier, on the majority of operating systems*

Installing Ansible using the system's package manager on RHEL and CentOS:

* To install Ansible use the following command:

**$ sudo yum install ansible**

Note:

*On Cent 6 or RHEL 6, you have to run the command****rpm -Uvh***

* You can also install Ansible from an RPM file. You need to use the make rpm command against the git checkout of Ansible, as follows:

**$ git clone git://github.com/ansible/ansible.git**

**$ cd ./ansible**

**$ make rpm**

**$ sudo rpm -Uvh ~/rpmbuild/ansible-\*.noarch.rpm**

Note:

*You should have****rpm-build, make****, and****python2-devel****installed on your system to build an rpm.*

* Once you're done installing Ansible, run command **ansible --version** to verify that it has been installed:

**$ ansible --version**

* **To check if Ansible is working the following are the steps that need to be performed:**
  + 1. Create an Ansible inventory file in **/etc/ansible/**. This can contain one or more groups. Each group is defined within square brackets

In this example the group is named **‘servers’:**

**$ cat inventory**

**[servers]**

**machine1**

**machine2**

* + 1. Run command **ansible --help** to view the available options, as shown below (only copying the subset that we need for this example):

**ansible --help**

**Usage: ansible <host-pattern> [options]**

**Options:**

**-a MODULE\_ARGS, --args=MODULE\_ARGS**

**module arguments**

**-i INVENTORY, --inventory-file=INVENTORY**

**specify inventory host file**

**(default=/etc/ansible/hosts)**

**-m MODULE\_NAME, --module-name=MODULE\_NAME**

**module name to execute**

**(default=command)**

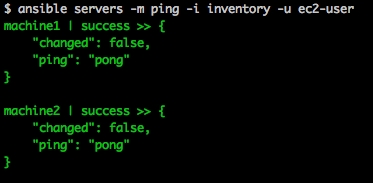
* + 1. Generate **ssh RSA key** with the following command:

**$ssh-keygen**

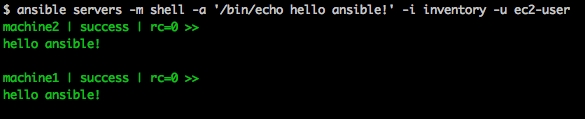
* + 1. Copy ssh rsa key on all machines that will be added **/etc/ansible/hosts:**

**$ssh-copy-id ‘servers name’**

* + 1. Ping the two servers using the Ansible command line, as shown in the following screenshot:



* + 1. Now that we can ping these two servers, let's echo **hello ansible!**, using the command line shown in the following screenshot:



## The Ansible Architecture

A screenshot of a cell phone

Description automatically generated

### Ad-hoc Commands

Ad hoc commands are commands which can be run individually to perform quick functions. These commands need not be performed later.

For example, you have to **reboot** all your company servers. For this, you will run the Ad-hoc commands from **‘/usr/bin/ansible’**.

These **ad-hoc** commands are not used for configuration management and deployment, because these commands are of one time usage.

**Ansible-playbook** is used for configuration management and deployment.

* **Parallelism and Shell Commands**

Reboot your company server in 12 parallel forks at time. For this, we need to set up SSHagent for connection.

**$ ssh-agent bash**

**$ ssh-add ~/.ssh/id\_rsa**

To run reboot for all your company servers in a group, 'abc', in 12 parallel forks −

**$ Ansible abc -a "/sbin/reboot" -f 12**

By default, Ansible will run the above Ad-hoc commands form current user account. If you want to change this behavior, you will have to pass the username in Ad-hoc commands as follows −

**$ Ansible abc -a "/sbin/reboot" -f 12 -u username**

* **File Transfer**

You can use the Ad-hoc commands for doing **SCP** (Secure Copy Protocol) lots of files in parallel on multiple machines.

Transferring file to many servers/machines

**$ Ansible abc -m copy -a "src = /etc/yum.conf dest = /tmp/yum.conf"**

Creating new directory

**$ Ansible abc -m file -a "dest = /path/user1/new mode = 777 owner = user1 group = user1 state** = directory"

Deleting whole directory and files

**$ Ansible abc -m file -a "dest = /path/user1/new state = absent"**

* **Managing Packages**

The Ad-hoc commands are available for yum and apt. Following are some Ad-hoc commands using yum.

The following command checks if yum package is installed or not, but does not update it.

**$ Ansible abc -m yum -a "name = demo-tomcat-1 state = present"**

The following command check the package is not installed.

**$ Ansible abc -m yum -a "name = demo-tomcat-1 state = absent"**

The following command checks the latest version of package is installed.

**$ Ansible abc -m yum -a "name = demo-tomcat-1 state = latest"**

* **Users and groups**

The ‘user’ module allows easy creation and manipulation of existing user accounts, as well as removal of user accounts that may exist:

**$ ansible all -m user -a "name=foo password=<crypted password here>"**

**$ ansible all -m user -a "name=foo state=absent"**

* **Deploying From Source Control**

Deploy your webapp straight from git:

**$ ansible webservers -m git -a "repo=https://foo.example.org/repo.git dest=/srv/myapp version=HEAD"**

Since Ansible modules can notify change handlers it is possible to tell Ansible to run specific tasks when the code is updated, such as deploying Perl/Python/PHP/Ruby directly from git and then restarting apache.

* [**Managing Services**](https://docs.ansible.com/ansible/latest/user_guide/intro_adhoc.html#id13)

Ensure a service is started on all webservers:

**$ ansible webservers -m service -a "name=httpd state=started"**

Alternatively, restart a service on all webservers:

**$ ansible webservers -m service -a "name=httpd state=restarted"**

Ensure a service is stopped**:**

**$ ansible webservers -m service -a "name=httpd state=stopped"**

* **Time Limited Background Operations**

Long running operations can be run in the background, and it is possible to check their status later. For example, to execute long\_running\_operation asynchronously in the background, with a timeout of 3600 seconds (-B), and without polling (-P):

**$ ansible all -B 3600 -P 0 -a "/usr/bin/long\_running\_operation --do-stuff"**

If you do decide you want to check on the job status later, you can use the async\_status module, passing it the job id that was returned when you ran the original job in the background:

**$ ansible web1.example.com -m async\_status -a "jid=488359678239.2844"**

Polling is built-in and looks like this:

**$ ansible all -B 1800 -P 60 -a "/usr/bin/long\_running\_operation --do-stuff"**

The above example says “run for 30 minutes max (-B 30\*60=1800), poll for status (-P) every 60 seconds”.

Poll mode is smart so all jobs will be started before polling will begin on any machine. Be sure to use a high enough --forks value if you want to get all of your jobs started very quickly. After the time limit (in seconds) runs out (-B), the process on the remote nodes will be terminated.

* **Gathering Facts**

Facts can be used for implementing conditional statements in playbook. You can find adhoc information of all your facts through the following Ad-hoc command −

**$ Ansible all -m setup**

### **Host inventory file**

* Ansible can run its tasks against multiple hosts in parallel. To do this, you can directly pass the list of hosts to Ansible using an inventory file. For such parallel execution, Ansible allows you to group your hosts in the inventory file; the file passes the group name to Ansible. Ansible will search for that group in the inventory file and run its tasks against all the hosts listed in that group. It follows the INI format and tells Ansible whether the remote host or hosts provided by the user are genuine or not.

You can pass the inventory file to Ansible using the **-i** or **--inventory-file** option followed by the path to the file. If you do not explicitly specify any inventory file to Ansible, it will take the default path from the **host\_file** parameter of **ansible.cfg**, which defaults to **/etc/ansible/hosts**.

**A basic inventory file example**:

A close up of a logo

Description automatically generated

Ansible can take either a hostname or an IP address within the inventory file. In the preceding example, we specified four servers; Ansible will take these servers and search for the hostname that you provided, to run its tasks. If you want to run your Ansible tasks against **all** of these hosts, then you can pass all to the hosts parameter while running the **ansible-playbook** or to the **ansible** command; this will make Ansible run its tasks against all the hosts listed in an inventory file.

The command that you would run is shown in the following screenshot:

A screenshot of a cell phone

Description automatically generated

In the the screenshot, the Ansible command took all the hosts from an inventory file and ran the ping module against each of them. Similarly, you can use all with the ansible-playbook by passing all to the hosts.

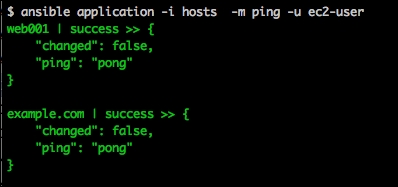
**Groups in an inventory file**:

In the following example, we grouped the inventory file into three groups, that is, **application**, **db**, and **jump**:

A close up of a logo

Description automatically generated

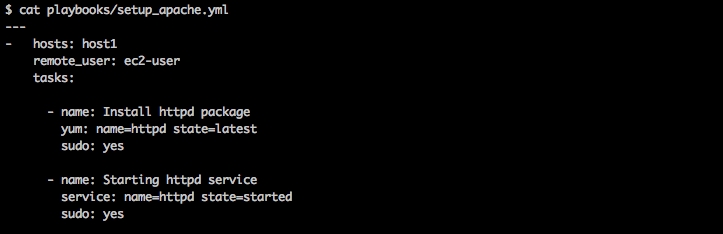
Instead of running Ansible against all the hosts, you can run it against a set of hosts by passing the group name to the ansible-playbook command. When Ansible runs its tasks against a group, it will take all the hosts that fall under that group. To run Ansible against the group, you need to run the following command line :



This time we directly passed the group name instead of running Ansible against all hosts; you can have multiple groups in the inventory file and you can even club similar groups together in one group (we will see how clubbing groups works in the next section). You can use groups using Ansible's **playbook** command as well by passing the group name to hosts.

### Playbooks

* Are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks; each task internally links to a piece of code called a module. Playbooks are simple human-readable YAML files, whereas modules are a piece of code that can be written in any language with the condition that its output should be in the JSON format. You can have multiple tasks listed in a playbook and these tasks would be executed serially by Ansible. Playbooks can have a list of remote hosts, user variables, tasks, handlers (covered later in this chapter), and so on. You can also override most of the configuration settings through a playbook. One of playbook purpose is to ensure that the **httpd** package is installed and the service is started. Consider the following command line to display the available playbooks:



* Playbooks contain the following sections:

Every playbook starts with 3 hyphens ‘—‘

**Host section** – Defines the target machines on which the playbook should run. This is based on the Ansible inventory file.

**Variable section** – This is optional and can declare all the variables needed in the playbook. We will look at some examples as well.

**Tasks section** – This section lists out all the tasks that should be executed on the target machine. It specifies the use of Modules. Every task has a name which is a small description of what the task will do and will be listed while the playbook is run

Consider the following command line to run a playbook for example:



* You can also use the Ansible API to run **scripts**. These are situations where you would have a wrapper script that would then utilize the **API** to run the playbooks as needed. The playbooks are declarative in nature and are written in **YAML Ain't Markup Language (YAML)**. This takes the declarative simplicity of such systems to a different level.
* Ansible can also be used to provision new machines in data centers and/or Cloud, based on your infrastructure and configure them based on the role of the new machine. For such situations, Ansible has the power to execute a certain number of tasks in the local mode, that is, on the command center, and certain tasks on the actual machine, post the machine-boot-up phase.
* In this case, a local action can spawn a new machine using an API of some sort, wait for the machine to come up by checking whether standard ports are up, and then log in to the machine and execute commands. The other aspect to consider is that Ansible can run tasks either serially or **N** threads in parallel. This leads to different permutations and combinations when you're using Ansible for deployment.

### Modules

* Ansible provides more than 200 modules under top-level modules, such as System, Network, Database, and Files, that you can readily use and deal with in your infrastructure. The following modules that are commonly used :

###### **Command modules**

They are used to execute tasks on remote machines:

* The command module **-** This takes the command name along with the arguments. However, shell variables or operations such as **<**, **>**, **|**, and **&** will not work as they will not be processed by the shell. Running the command module is secure and predictable. Also, the command module gives you the following parameters:

**chdir**: This is used to change to a specific directory and execute commands

**creates**: You can specify what file will be created with this option

**removes**: This is used to remove a file

Consider the following command line example:

**- name: Reboot machine**

**command: /sbin/shutdown -r now**

**sudo: yes**

On running the preceding command, we can see the following output:

A screenshot of a cell phone

Description automatically generated

* The script module - This module is used to copy a script remotely to a machine and execute it. It supports the **creates** and **removes** parameters. Let's look at an example where we list down directories within a particular directory. Remember, you don't have to copy the script remotely in this case. The module does it for you as follows:

**- name: List directories in /etc**

**script: list\_number\_of\_directories.sh /etc**

**sudo: yes**

On running the preceding command, we get the following output:



Here, **83** is the number of directories in the **/etc** directory, which can be verified by running the following command:

**$ls -l /etc | egrep '^d' | wc -l**

**83**

* The shell module - The major difference between the **command** and **shell** modules is that the **shell** module uses a shell **(/bin/sh**, by default) to run the commands. You can use shell environment variables and other shell features. Consider the following command line example where we redirect the list of all files in **/tmp** to a directory and, in a subsequent task, concatenate (using **cat**) the file. The tasks are shown as follows:

**- name: List files in /tmp and redirect to a file**

**shell: /bin/ls -l /tmp > /tmp/list**

**sudo: yes**

**- name: Cat /tmp/list**

**shell: /bin/cat /tmp/list**

The output is shown as follows:

A screenshot of a cell phone

Description automatically generated

###### **File modules**

* The shell module - The **file** module allows you to change the attributes of a file. It can touch a file, create or delete recursive directories, and create or delete symlinks.

The following example makes sure that **httpd.conf** has the right permissions and owner:

**- name: Ensure httpd conf has right permissions and owner/group**

**file: path=/etc/httpd/conf/httpd.conf owner=root group=root mode=0644**

**sudo: yes**

On running the preceding command, you should see the following output:



If we check the output on the machine, we will see the following:

A close up of a logo

Description automatically generated

* As shown in the preceding screenshot, there is no change as the file has the expected permissions and ownership.

The next example will create a **symlink** to the **httpd conf** file, as follows:

**- name: Create a symlink in /tmp for httpd.conf**

**file: src=/etc/httpd/conf/httpd.conf dest=/tmp/httpd.conf owner=root group=root state=link**

**sudo: yes**

The output of the preceding task is as follows:



If we check on the machine, we will see the following output:



* Template Module - Ansible uses the **Jinja2** templating language for creating templates, modeled after Django's templates (Django is a popular Python web framework). Templating is also a way to create a file on a machine. Consider the following command line to create a simple template using the following command lines:

**$cat test**

**This is a test file on {{ ansible\_hostname }}**

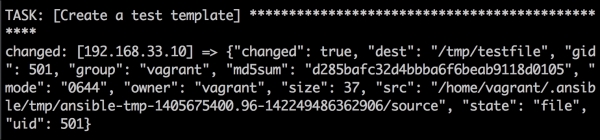
The test file is in the same directory as **example1.yml**.

To reference the template, we'll add the following to the playbook:

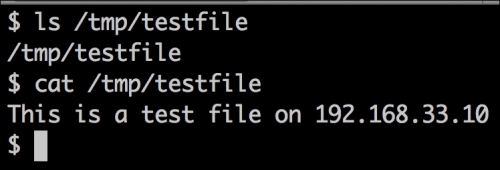
**- name: Create a test template**

**template: src=test dest=/tmp/testfile mode=644**

On running the preceding playbook, we get the following output:



As you can see in the following screenshot, Ansible created **testfile** inside **/tmp** and applied the template to the file.

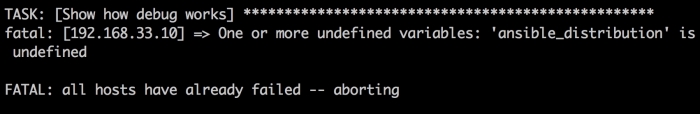


The user running the playbook is **vagrant** in this case and the file created will also be owned by the same user. The **ansible\_hostname** variable is populated during the **gather facts** phase. Also **gather facts** can be **disable** by adding the following to the playbook:

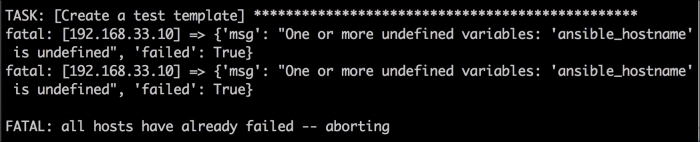
**- hosts: host1**

**gather\_facts: False**

Now, on running the playbook, the debug task fails as follows:



On commenting out the task and running it again, we get an error with the template, as shown in the following screenshot:

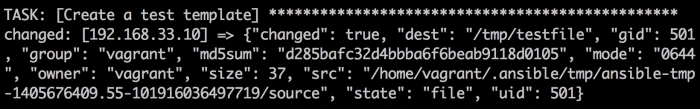


There are several cases where it is no need to gather facts. In such cases, to refer to the host, Ansible provides another useful variable **inventory\_hostname**, which you can use. To modify the template, use the following command line:

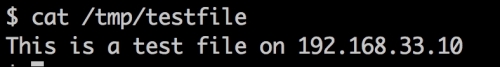
**cat playbooks/test**

**This is a test file on {{ inventory\_hostname }}**

On deleting the test file and rerunning the Ansible playbook, we find the same result as before:



As expected, Ansible created **testfile** and did not fail because we used the **inventory\_hostname** variable this time:



The Ansible template also has a **validate** parameter that allows you to run a command to validate the file before copying it. This is like a hook that Ansible provides to make sure files that might break the service are not written. A classic example is that of the **Apache httpd** configuration. You can verify that the **Apache httpd** configuration files have the right syntax using the **httpd** or **apachectl** command. Since the **validate** command takes an argument, we'll use the **httpd** option. It works as follows:

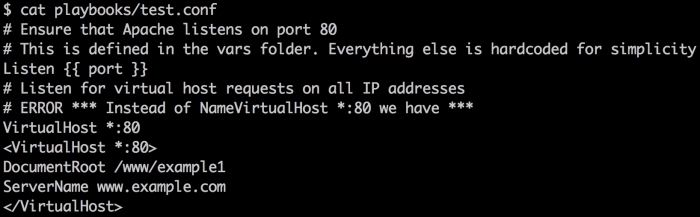
**$httpd -t -f /etc/httpd/conf/httpd.conf**

**Syntax OK**

If an error is introduce by uncommenting the **Virtual** **hosts** line, we get the following output when we rerun the **$httpd -t -f /etc/httpd/conf/httpd.conf** command:

**httpd: Syntax error on line 1003 of /etc/httpd/conf/httpd.conf:/etc/httpd/conf/httpd.conf:1003: <VirtualHost> was not closed**.

Consider the following example to verify that the new virtual host file that must be added has the right syntax. There is an error, as shown in the following screenshot:



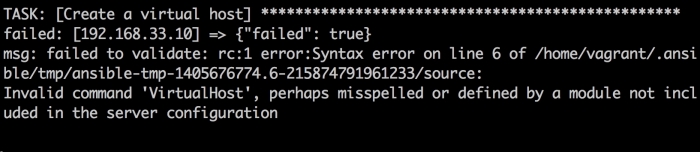
The playbook will have the following template task. The **validate** option takes a **%s** parameter, which is the source file that we're planning to write to the machine:

**- name: Create a virtual host**

**template: src=test.conf dest=/etc/httpd/conf.d/test.conf mode=644 validate='httpd -t -f %s'**

**sudo: yes**

Now, on running the preceding command line, we get the following output:



Ansible points out the error and the file is **not written**. This is a great feature that Ansible templates offer; if there are scripts for different packages/services that can verify the validity of files, then there will be a lot fewer breakages due to incorrect configuration files. It is especially useful to have the **validate** feature when you write your own modules.

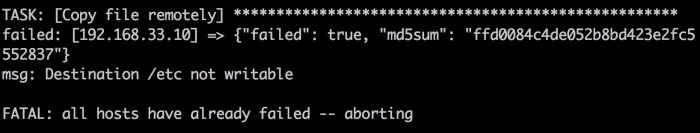
* Copy Module - Using **copy**, you can copy a file from your local location to remote machines. This is another way to create a remote file with predetermined content (the template being the first).

Consider the following command line exemple:

**- name: Copy file remotely**

**copy: src=test2.conf dest=/etc/test2.conf owner=root group=root mode=0644**

On running the preceding command, we see the following output:



The error is encountered because **sudo: true** is not part of the module invocation.

Once it is added, the run goes through without errors, as shown in the following screenshot:

C:\Users\ezmardo\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\FE93A9DF.tmp

The **copy** module also supports the **validate** parameter just like the **template** module.

* Source control module - **git** – It is a very important module, Ansible has modules to support **subversion**, **bzr**, and **hg**, apart from **github\_hooks.**

Consider the following example for installing a **git repository** from **GitHub** using an Ansible task”

First must install **git** with a **yum** task, shown as follows:

**- yum: name=git state=installed**

**sudo: yes**

The Ansible task is shown as follows:

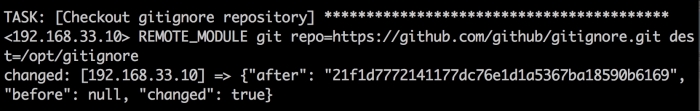
**- name: Checkout gitignore repository**

**git: repo=https://github.com/github/gitignore.git**

**dest=/opt/gitignore**

**sudo: yes**

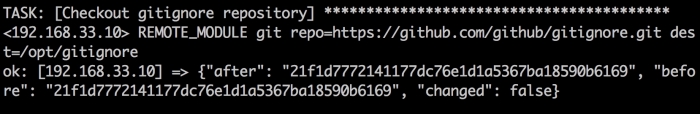
On running the playbook (using **–vv** here), we will see the following output:



If we check the machine, we will see that the repository has been checked out in the expected directory.

C:\Users\ezmardo\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\57ED3D9B.tmp

On rerunning the task, we see the following output:



This basically means that the task is idempotent as it checks the **before** and **after** **SHA** (**Secure Hash Algorithm**) values.

There are several other parameters with the **git** module, such as **depth** and **version** (which version to checkout). Quite often, we need to check out the git repository via the SSH protocol. The public keys have to be added to the repository and post that the checkout can happen. The git module has two other parameters, **accept\_hostkey** and **key\_file**, to help you in the git checkouts. The following is an example of a sample repository in one of our GitHub accounts, which we'll checkout on the remote machine. This example assumes that the private key pair is already present in **~/.ssh**.

Consider the following task for example:

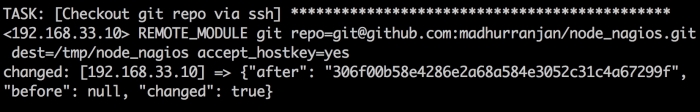
**- name: Checkout git repo via ssh**

**git: repo=git@github.com:madhurranjan/node\_nagios.git**

**dest=/tmp/node\_nagios**

**accept\_hostkey=yes**

The output for the preceding command line is as follows:



## Configuring Ansible

* An Ansible configuration file uses an **INI** format to store its configuration data. In Ansible, you can overwrite nearly all of the configuration settings either through Ansible playbook options or environment variables. While running an Ansible command, the command looks for its configuration file in a predefined order, as follows:

**ANSIBLE\_CONFIG**: Firstly, the Ansible command checks the environment variable, which points to the configuration file

**./ansible.cfg**: Secondly, it checks the file in the current directory

**~/.ansible.cfg**: Thirdly, it checks the file in the user's home directory

**/etc/ansible/ansible.cfg**: Lastly, it checks the file that is automatically generated when installing Ansible via a package manager

* If you have installed Ansible through your **system's package manager** or **pip**, then you should already have a copy of **ansible.cfg** under the **/etc/ansible directory**

### Configuration using environment variables

* You can use most of the configuration parameters directly via environment variables by appending ANSIBLE\_ to the configuration parameter (the parameter name should be in uppercase).

Command line example:

**export ANSIBLE\_SUDO\_USER=root**

*The****ANSIBLE\_SUDO\_USER****variable can then be used as part of the playbooks.*

### Configuration using *ansible.cfg*

* Ansible has many configuration parameters, some of the important parameters are:
* **hostfile**: This parameter indicates the path to the inventory file. The inventory file consists of a list of hosts that Ansible can connect to. We will discuss inventory files in detail later in this chapter. Consider the following command line for example:

**hostfile = /etc/ansible/hosts**

* **library:** Whenever you ask Ansible to perform any action, whether it is a local action or a remote one, it uses a piece of code to perform the action; this piece of code is called a module. The **library** parameter points to the path of the directory where Ansible modules are stored. Consider the following command line for example:

**library = /usr/share/ansible**

* **forks**: This parameter is the default number of processes that you want Ansible to spawn. It defaults to five maximum processes in parallel. Consider the following command line for example:

**forks = 5**

* **sudo\_user**: This parameter specifies the default user that should be used against the issued commands. You can override this parameter from the Ansible playbook as well (this is covered in a later chapter). Consider the following command line for example:

**sudo\_user = root**

* **remote\_port**: This parameter is used to specify the port used for SSH connections, which defaults to **22**. You might never need to change this value unless you are using SSH on a different port. Consider the following command line for example:

**remote\_port = 22**

* **remote\_user:** This is one of the configuration parameters of Ansible (consider, for example, **tom' - remote\_user**) that tells Ansible to use a particular user (in this case, **tom**) while logging into the system.
* **timeout**: This is the default value for the timeout of SSH connection attempts:

**timeout = 60**

* **hosts:** This lists the host or host group against which we want to run the task. The **hosts** field is mandatory and every playbook should have it (except roles). It tells Ansible where to run the listed tasks. When provided with a host group, Ansible will take the host group from the playbook and will try looking for it in an inventory file (covered later in the chapter). If there is no match, Ansible will skip all the tasks for that host group. The **--list-hosts** option along with the playbook (**ansible-playbook <playbook> --list-host**) will exactly tell you against which hosts the playbook will run.
* **host\_key\_checking**: This parameter is used to disable the SSH host key checking; this is set to True by default. Consider the following command line for example:

**remote\_port = 22**

* **host\_key\_checking: T**his parameter is used to disable the SSH host key checking; this is set to True by default. Consider the following command line for example:

**host\_key\_checking = False**

* **log\_path**: By default, Ansible doesn't log anything; if you would like to send the Ansible output to a logfile, then set the value of **log\_path** to the file you would like to store the Ansible logs in. Consider the following command line for example:

**log\_path = /var/log/ansible.log**

* **tasks:** All playbooks should contain tasks. Tasks are a list of actions you want to perform. A **tasks** field contains the name of the task, that is, the help text for the user about the task, a module that should be executed, and arguments that are required for the module. Consider the following example of two tasks that are listed in the playbook:

**tasks:**

**- name: Install httpd package**

**yum: name=httpd state=latest**

**sudo: yes**

**- name: Starting httpd service**

**service: name=httpd state=started**

**sudo: yes**