

# **Configuration Management**

## **What Is Configuration Management?**

Configuration and resource management is an automated method for maintaining computer systems and software in a known, consistent state.

There are several components in a configuration management system. Managed systems can include servers, storage, networking, and software. These are the targets of the configuration management system. The goal is to maintain these systems in known, determined states. Another aspect of a configuration management system is the description of the desired state for the systems. The third major aspect of a configuration management system is automation software, which is responsible for making sure that the target systems and software are maintained in the desired state.

## **What are the Benefits of Configuration Management?**

The primary benefit of configuration management is consistency of systems and software. With configuration management, you no longer guess or hope that a configuration is current. It is correct because the configuration management system ensures that it is correct.

When combined with automation, configuration management can improve efficiency because manual configuration processes are replaced with automated processes. This also makes it possible to manage more targets with the same or even fewer resources.

## **Why Is Configuration Management Important?**

Configuration management is important because it enables the ability to scale infrastructure and software systems without having to correspondingly scale administrative staff to manage those systems. This can make it possible to scale where it previously wasn't feasible to do so.

## **Configuration Management Tools**

It is common for configuration management tools to include automation too. Popular tools are:

- Ansible
- Chef
- Puppet

## **Introduction to Ansible**

**Ansible** is simple open source IT engine which automates application deployment, intra service orchestration, cloud provisioning and many other IT tools.

Ansible is easy to deploy because it does not use any agents or custom security infrastructure.

Ansible uses playbook to describe automation jobs, and playbook uses very simple language i.e. **YAML** (It's a human-readable data serialization language & is commonly used for configuration files, but could be used in many applications where data is being stored) which is very easy for humans to understand, read and write. Hence the advantage is that even the IT infrastructure support guys can read and understand the playbook and debug if needed (YAML – It is in human readable form).

Ansible is designed for multi-tier deployment. Ansible does not manage one system at time, it models IT infrastructure by describing all of your systems are interrelated. Ansible is completely agentless which means Ansible works by connecting your nodes through ssh(by default). But if you want other method for connection like Kerberos, Ansible gives that option to you.

After connecting to your nodes, Ansible pushes small programs called as “Ansible Modules”. Ansible runs that modules on your nodes and removes them when finished. Ansible manages your inventory in simple text files (These are the hosts file). Ansible uses the hosts file where one can group the hosts and can control the actions on a specific group in the playbooks.

## **What is Configuration Management**

Configuration management in terms of Ansible means that it maintains configuration of the product performance by keeping a record and updating detailed information which describes an enterprise's hardware and software.

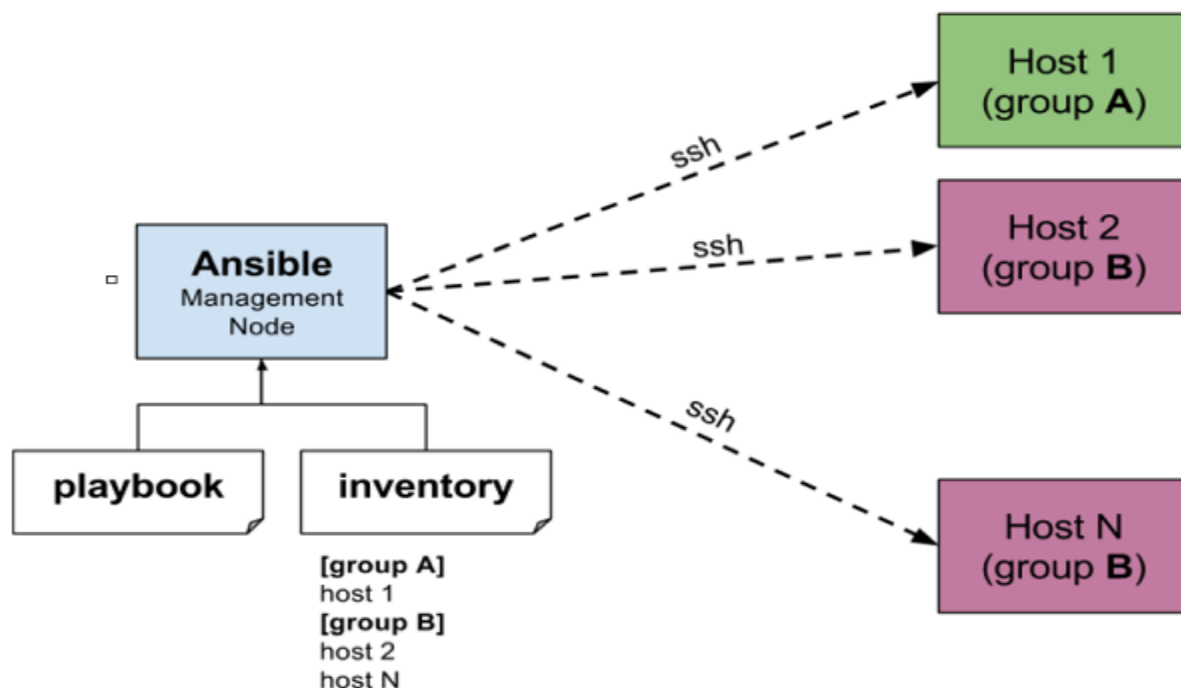
Such information typically includes the exact versions and updates that have been applied to installed software packages and the locations and network addresses of hardware devices. For e.g. If you want to install the new version of **WebLogic/WebSphere** server on all of the machines present in your enterprise, it is not feasible for you to manually go and update each and every machine.

You can install WebLogic/WebSphere in one go on all of your machines with Ansible playbooks and inventory written in the most simple way. All you have to do is list out the IP addresses of your nodes in the inventory and write a playbook to install WebLogic/WebSphere. Run the playbook from your control machine & it will be installed on all your nodes.

### How Ansible Works?

The picture given below shows the working of Ansible.

**Ansible works** by connecting to your nodes and pushing out small programs, called "**Ansible modules**" to them. **Ansible** then executes these modules (over SSH by default), and removes them when finished. Your library of modules can reside on any machine, and there are no servers, daemons, or databases required.



The management node in the above picture is the controlling node (managing node) which controls the entire execution of the playbook. It's the node from which you are running the installation. The inventory file provides the list of hosts where the Ansible modules need to be run and the management node does a SSH connection and executes the small modules on the hosts machine and installs the product/software.

Beauty of Ansible is that it removes the modules once those are installed so effectively it connects to host machine , executes the instructions and if it's successfully installed removes the code which was copied on the host machine which was executed.

## **Ansible - Playbooks**

Playbooks are the files where Ansible code is written. Playbooks are written in YAML format. YAML stands for Yet Another Markup Language. **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.

Playbooks contain the steps which the user wants to execute on a particular machine. Playbooks are run sequentially. Playbooks are the building blocks for all the use cases of Ansible.

### **Playbook Structure**

Each playbook is an aggregation of one or more plays in it. Playbooks are structured using Plays. There can be more than one play inside a playbook.

The function of a play is to map a set of instructions defined against a particular host.

YAML is a strict typed language; so, extra care needs to be taken while writing the YAML files. There are different YAML editors but we will prefer to use a simple editor like notepad++. Just open notepad++ and copy and paste the below yaml and change the language to YAML (Language → YAML).

A YAML starts with --- (3 hyphens)

### **Create a Playbook**

Let us start by writing a sample YAML file. We will walk through each section written in a yaml file.

```
---
name: install and configure DB
hosts: testServer
become: yes
```

```
vars:
  oracle_db_port_value : 1521

tasks:
  -name: Install the Oracle DB
    yum: <code to install the DB>

  -name: Ensure the installed service is enabled and running
    service:
      name: <your service name>
```

The above is a sample Playbook where we are trying to cover the basic syntax of a playbook. Save the above content in a file as **test.yml**. A YAML syntax needs to follow the correct indentation and one needs to be a little careful while writing the syntax.

## The Different YAML Tags

Let us now go through the different YAML tags. The different tags are described below –

### name

This tag specifies the name of the Ansible playbook. As in what this playbook will be doing. Any logical name can be given to the playbook.

### hosts

This tag specifies the lists of hosts or host group against which we want to run the task. The hosts field/tag is mandatory. It tells Ansible on which hosts to run the listed tasks. The tasks can be run on the same machine or on a remote machine. One can run the tasks on multiple machines and hence hosts tag can have a group of hosts' entry as well.

### vars

Vars tag lets you define the variables which you can use in your playbook. Usage is similar to variables in any programming language.

## tasks

All playbooks should contain tasks or a list of tasks to be executed. Tasks are a list of actions one needs to perform. A tasks field contains the name of the task. This works as the help text for the user. It is not mandatory but proves useful in debugging the playbook. Each task internally links to a piece of code called a module. A module that should be executed, and arguments that are required for the module you want to execute.

## Ansible Roles

Roles provide a framework for fully independent or interdependent collections of files, tasks, templates, variables, and modules.

The role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing **complex playbooks** and makes them easier to reuse. The breaking of the playbook allows you to break the playbook into reusable components.

Each role is limited to a particular functionality or desired output, with all the necessary steps to provide that result either within the same role itself or in other roles listed as dependencies.

Roles are not playbooks. Roles are small functionality that can be used within the playbooks independently. Roles have no specific setting for which hosts the role will apply.

Top-level playbooks are the bridge holding the hosts from your inventory file to roles that should be applied to those hosts.

## Creating a Role

The directory structure for roles is essential to creating a new role, such as:

### Role Structure

The roles have a structured layout on the file system. You can change the default structured of the roles as well.

**For example,** let us stick to the default structure of the roles. Each role is a directory tree in itself. So the role name is the directory name within the /roles directory.

```
$ ansible-galaxy -h
```

### Usage

```
ansible-galaxy [delete|import|info|init|install|list|login|remove|search|setup] [--help] [options] ...
```

### Options

- -h: (help) it shows this help message and exit.
- -v: (verbose) Verbose mode (-vvv for more, -vvvv to enable connection debugging).
- --version: it shows program version number and exit.

Roles are stored in separate directories and have a particular directory structure

```
[root@ansible-server test2]# tree
```

```
.
|-- role1
|   |-- defaults
|   |   `-- main.yml
|   |-- handlers
|   |   `-- main.yml
|   |-- meta
|   |   `-- main.yml
|   |-- README.md
|   |-- tasks
|   |   `-- main.yml
|   |-- tests
|   |   |-- inventory
|   |   `-- test.yml
|   `-- vars
```

```
`-- main.yml
```

### Explanation

- The YAML file in the default directory contains a list of default variables that are to be used along with the playbook.
- The handler's directory is used to store handlers.
- The meta-directory is supposed to have information about the author and role dependencies.
- The tasks directory is the main YAML file for the role.
- The tests directory contains a sample YAML playbook file and a sample inventory file and is mostly used for testing purposes before creating the actual role.
- The vars directory contains the YAML file in which all the variables used by the role will be defined. The directory templates and the directory files should contain files and templates that will be used by the tasks in the role.

### Templating (Jinja2)

Ansible uses Jinja2 templating to enable dynamic expressions and access to [variables](#) and [facts](#). You can use templating with the [template module](#). For example, you can create a template for a configuration file, then deploy that configuration file to multiple environments and supply the correct data (IP address, hostname, version) for each environment. You can also use templating in playbooks directly, by templating task names and more. You can use all the [standard filters and tests](#) included in Jinja2. Ansible includes additional specialized filters for selecting and transforming data, tests for evaluating template expressions, and [Lookup plugins](#) for retrieving data from external sources such as files, APIs, and databases for use in templating.

### How to Use Jinja2 Template in Ansible Playbook

[Jinja2](#) is a powerful and easy to use python-based templating engine that comes in handy in an IT environment with multiple servers where configurations vary every other time. Creating static configuration files for each of these nodes is tedious and may not be a viable option since it will consume more time and energy. And this is where templating comes in.



Jinja2 templates are simple template files that store variables that can change from time to time. When Playbooks are executed, these variables get replaced by actual values defined in Ansible Playbooks. This way, templating offers an efficient and flexible solution to create or alter configuration file with ease.

### *Template architecture*

A Jinja2 template file is a text file that contains variables that get evaluated and replaced by actual values upon runtime or code execution. In a Jinja2 template file, you will find the following tags:

- `{{ }}` : These double curly braces are the widely used tags in a template file and they are used for embedding variables and ultimately printing their value during code execution. For example, a simple syntax using the double curly braces is as shown: The `{{ webserver }}` is running on `{{ nginx-version }}`
- `{% %}` : These are mostly used for control statements such as loops and if-else statements.
- `{# #}` : These denote comments that describe a task.

In most cases, Jinja2 template files are used for creating files or replacing configuration files on servers. Apart from that, you can perform conditional statements such as **loops** and **if-else** statements, and transform the data using filters and so much more.

Template files bear the `.j2` extension, implying that Jinja2 templating is in use.

### *Creating template files*

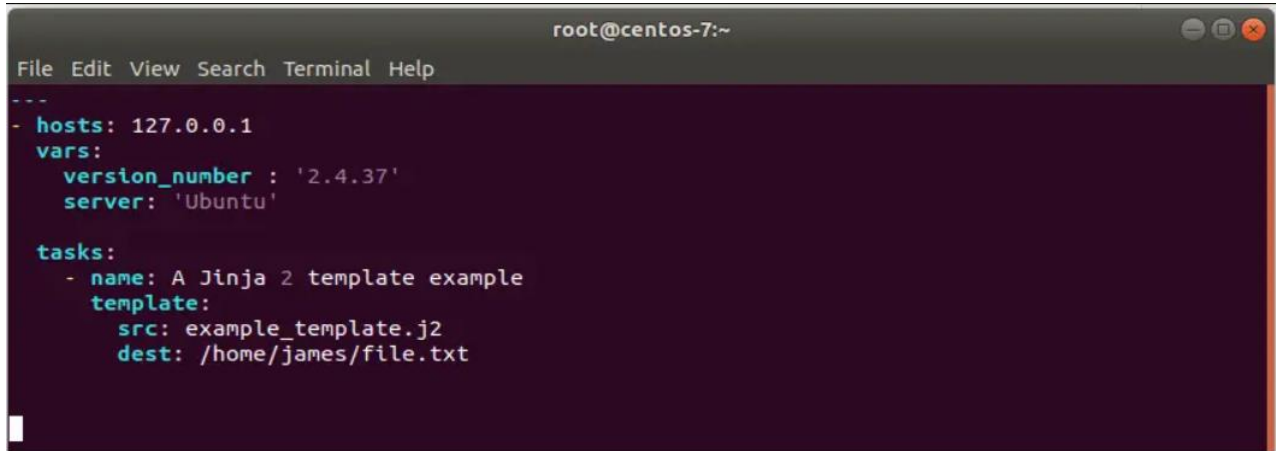
Here's an example of a Jinja2 template file `example_template.j2` which we shall use to create a new file with the variables shown

```
Hey guys!  
Apache webserver {{ version_number }} is running on {{ server }}
```

Enjoy!

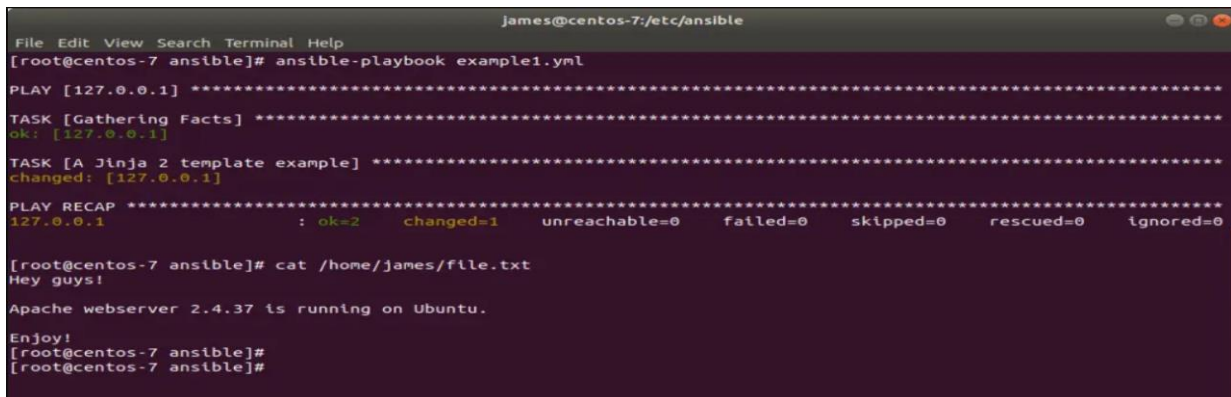
Here, the variables are `{{ version_number }}` & `{{ server }}`

These variables are defined in a playbook and will be replaced by actual values in the playbook YAML file `example1.yml` below.



```
root@centos-7:~  
File Edit View Search Terminal Help  
---  
- hosts: 127.0.0.1  
  vars:  
    version_number: '2.4.37'  
    server: 'Ubuntu'  
  
  tasks:  
    - name: A Jinja 2 template example  
      template:  
        src: example_template.j2  
        dest: /home/james/file.txt
```

When the playbook is executed, the variables in the template file get replaced by the actual values and a new file is either created or replaces an already existing `file.txt` in the destination path.



```
james@centos-7:/etc/ansible  
File Edit View Search Terminal Help  
[root@centos-7 ansible]# ansible-playbook example1.yml  
PLAY [127.0.0.1] *****  
TASK [Gathering Facts] *****  
ok: [127.0.0.1]  
TASK [A Jinja 2 template example] *****  
changed: [127.0.0.1]  
PLAY RECAP *****  
127.0.0.1 : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0  
  
[root@centos-7 ansible]# cat /home/james/file.txt  
Hey guys!  
Apache webserver 2.4.37 is running on Ubuntu.  
Enjoy!  
[root@centos-7 ansible]#  
[root@centos-7 ansible]#
```

From the playbook execution, view the destination and notice that the variables have been replaced by the values defined in the Ansible playbook file.

## Kubernetes – Namespaces

Kubernetes Namespace is a mechanism that enables you to organize resources. It is like a virtual cluster inside the cluster. A namespace isolates the resources from the resources of other namespaces. For example, You need to have different names for deployments/services in a namespace but you can have the same name for deployment in two different namespaces.

By default, Kubernetes has 4 namespaces

```
ishan301@G3-3500:~$ kubectl get namespaces
NAME                STATUS    AGE
default             Active    3h11m
kube-node-lease     Active    3h11m
kube-public         Active    3h11m
kube-system         Active    3h11m
```

- **kube-system:** System processes like Master and kubectl processes are deployed in this namespace; thus, it is advised NOT TO CREATE OR MODIFY ANYTHING namespace.
- **kube-public:** This namespace contains publicly accessible data like a configMap containing cluster information.
- **kube-node-lease:** This namespace is the heartbeat of nodes. Each node has its associated lease object. It determines the availability of a node.
- **default:** This is the namespace that you use to create your resources by default.

Although whatever resources you create will be created in the default namespace but you can also create your own new namespace and create resources there.

**Note:** Avoid creating namespaces with the prefix kube-, since it is reserved for Kubernetes system namespaces and also you should not try to modify these namespaces.

### **Creating a Namespace:**

You can create your namespace by using the command

```
$ kubectl create namespace your-namespace
```

```
ishan301@G3-3500:~$ kubectl create namespace gfg
namespace/gfg created
ishan301@G3-3500:~$ kubectl get namespace
NAME                STATUS    AGE
default             Active    3h30m
gfg                 Active    6s
kube-node-lease     Active    3h30m
kube-public         Active    3h30m
kube-system         Active    3h30m
ishan301@G3-3500:~$
```

As you can see we have successfully created namespace **gfg**.

### Creating Component in a Namespace:

To create a component in a namespace you can either give the `--namespace` flag or specify the namespace in the configuration file.

#### Method 1: Using `--namespace` flag

```
$ kubectl apply -f your_config.yaml --namespace=your-namespace
```

then you can check resources in your namespace using **kubectl get** and specify namespace using **-n**

```
ishan301@G3-3500:~/play around$ kubectl apply -f nginx.yaml --namespace=gfg
deployment.apps/nginx created
ishan301@G3-3500:~/play around$ kubectl get deployment -n gfg
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
nginx     1/1     1            1           36s
ishan301@G3-3500:~/play around$
```

#### Method 2: Adding namespace in the configuration file

Instead of specifying a namespace using the `--namespace` flag you can specify your namespace initially in your config file only.

```
! nginx.yaml •
! nginx.yaml > {} metadata > namespace
io.k8s.api.apps.v1.Deployment (v1@deployment.json)
1  apiVersion: apps/v1
2  kind: Deployment
3
4  metadata:
5    name: nginx
6    namespace: gfg
7  spec:
8    selector:
9      matchLabels:
10     app: nginx
11   template:
12     metadata:
13       labels:
14         app: nginx
15     spec:
16       containers:
17       - name: nginx
18         image: nginx
19         resources:
20           limits:
21             memory: "128Mi"
22             cpu: "500m"
23         ports:
24         - containerPort: 800
25
```

and then use the command.

\$ kubectl apply -f your\_config\_file.yaml

```
ishan301@G3-3500:~/play around$ kubectl apply -f nginx.yaml
deployment.apps/nginx created
ishan301@G3-3500:~/play around$ kubectl get all -n gfg
NAME                                READY    STATUS    RESTARTS   AGE
pod/nginx-6d99f84b48-vb4ts          1/1      Running   0           11s

NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/nginx                1/1      1              1            11s

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/nginx-6d99f84b48     1          1          1        11s
ishan301@G3-3500:~/play around$
```

Hence we have successfully created a component in the desired namespace.

