


How To Create a Kubernetes Cluster Using Kubectl on CentOS 7

 digitalocean.com/community/tutorials/how-to-create-a-kubernetes-cluster-using-kubectl-on-centos-7



The author selected the [Free and Open Source Fund](#) to receive a donation as part of the [Write for DOnations](#) program.

Introduction

Kubernetes is a container orchestration system that manages containers at scale. Initially developed by Google based on its experience running containers in production, Kubernetes is open source and actively developed by a community around the world.

Note: This tutorial uses version 1.14 of Kubernetes, the official supported version at the time of this article's publication. For up-to-date information on the latest version, please see the [current release notes](#) in the official Kubernetes documentation.

Kubectl automates the installation and configuration of Kubernetes components such as the API server, Controller Manager, and Kube DNS. It does not, however, create users or handle the installation of operating-system-level dependencies and their configuration. For these preliminary tasks, it is possible to use a configuration management tool like [Ansible](#) or [SaltStack](#). Using these tools makes creating additional clusters or recreating existing clusters much simpler and less error-prone.

In this guide, you will set up a Kubernetes cluster from scratch using Ansible and Kubectl, and then deploy a containerized Nginx application to it.

Goals

Your cluster will include the following physical resources:

One master node

The master node (a *node* in Kubernetes refers to a server) is responsible for managing the state of the cluster. It runs Etcd, which stores cluster data among components that schedule workloads to worker nodes.

Two worker nodes

Worker nodes are the servers where your *workloads* (i.e. containerized applications and services) will run. A worker will continue to run your workload once they're assigned to it, even if the master goes down once scheduling is complete. A cluster's capacity can be increased by adding workers.

After completing this guide, you will have a cluster ready to run containerized applications, provided that the servers in the cluster have sufficient CPU and RAM resources for your applications to consume. Almost any traditional Unix application including web applications, databases, daemons, and command line tools can be containerized and made to run on the cluster. The cluster itself will consume around 300-500MB of memory and 10% of CPU on each node.

Once the cluster is set up, you will deploy the web server Nginx to it to ensure that it is running workloads correctly.

Prerequisites

- An SSH key pair on your local Linux/macOS/BSD machine. If you haven't used SSH keys before, you can learn how to set them up by following this explanation of [how to set up SSH keys on your local machine](#).
- Three servers running CentOS 7 with at least 2GB RAM and 2 vCPUs each. You should be able to SSH into each server as the root user with your SSH key pair. Be sure to also add your public key to the **centos** user's account on the master node. If you need guidance on adding an SSH key to a particular user account, see this tutorial on [How To Set Up SSH Keys on CentOS7](#).
- Ansible installed on your local machine. For installation instructions, follow the [official Ansible installation documentation](#).
- Familiarity with Ansible playbooks. For review, check out [Configuration Management 101: Writing Ansible Playbooks](#).
- Knowledge of how to launch a container from a Docker image. Look at "Step 5 — Running a Docker Container" in [How To Install and Use Docker on CentOS 7](#) if you need a refresher.

Step 1 — Setting Up the Workspace Directory and Ansible Inventory File

In this section, you will create a directory on your local machine that will serve as your workspace. You will also configure Ansible locally so that it can communicate with and execute commands on your remote servers. To do this, you will create a `hosts` file containing inventory information such as the IP addresses of your servers and the groups that each server belongs to.

Out of your three servers, one will be the master with an IP displayed as `master_ip`. The other two servers will be workers and will have the IPs `worker_1_ip` and `worker_2_ip`.

Create a directory named `~/kube-cluster` in the home directory of your local machine and `cd` into it:

- `mkdir ~/kube-cluster`
- `cd ~/kube-cluster`

This directory will be your workspace for the rest of the tutorial and will contain all of your Ansible playbooks. It will also be the directory inside which you will run all local commands.

Create a file named `~/kube-cluster/hosts` using `vi` or your favorite text editor:

```
vi ~/kube-cluster/hosts
```

Press `i` to insert the following text to the file, which will specify information about the logical structure of your cluster:

```
~/kube-cluster/hosts
```

```
[masters]
master ansible_host=master_ip ansible_user=root
```

```
[workers]
worker1 ansible_host=worker_1_ip ansible_user=root
worker2 ansible_host=worker_2_ip ansible_user=root
```

When you are finished, press `ESC` followed by `:wq` to write the changes to the file and quit.

You may recall that *inventory files* in Ansible are used to specify server information such as IP addresses, remote users, and groupings of servers to target as a single unit for executing commands. `~/kube-cluster/hosts` will be your inventory file and you've added two Ansible groups (**masters** and **workers**) to it specifying the logical structure of your cluster.

In the **masters** group, there is a server entry named “master” that lists the master node’s IP (`master_ip`) and specifies that Ansible should run remote commands as the root user.

Similarly, in the **workers** group, there are two entries for the worker servers (`worker_1_ip` and `worker_2_ip`) that also specify the `ansible_user` as root.

Having set up the server inventory with groups, let’s move on to installing operating-system-level dependencies and creating configuration settings.

Step 2 — Installing Kubernetes’ Dependencies

In this section, you will install the operating-system-level packages required by Kubernetes with CentOS’s `yum` package manager. These packages are:

- Docker - a container runtime. This is the component that runs your containers. Support for other runtimes such as `rkt` is under active development in Kubernetes.
- `kubeadm` - a CLI tool that will install and configure the various components of a cluster in a standard way.
- `kubelet` - a system service/program that runs on all nodes and handles node-level operations.
- `kubectrl` - a CLI tool used for issuing commands to the cluster through its API Server.

Create a file named `~/kube-cluster/kube-dependencies.yml` in the workspace:

```
vi ~/kube-cluster/kube-dependencies.yml
```

Add the following plays to the file to install these packages to your servers:

`~/kube-cluster/kube-dependencies.yml`

```
- hosts: all
  become: yes
  tasks:
    - name: install Docker
      yum:
        name: docker
        state: present
        update_cache: true

    - name: start Docker
      service:
        name: docker
        state: started

    - name: disable SELinux
      command: setenforce 0
```

- name: disable SELinux on reboot
 - selinux:
 - state: disabled

- name: ensure net.bridge.bridge-nf-call-ip6tables is set to 1
 - sysctl:
 - name: net.bridge.bridge-nf-call-ip6tables
 - value: 1
 - state: present

- name: ensure net.bridge.bridge-nf-call-iptables is set to 1
 - sysctl:
 - name: net.bridge.bridge-nf-call-iptables
 - value: 1
 - state: present

- name: add Kubernetes' YUM repository
 - yum_repository:
 - name: Kubernetes
 - description: Kubernetes YUM repository
 - baseurl: https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
 - gpgkey: <https://packages.cloud.google.com/yum/doc/yum-key.gpg>
 - <https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg>
 - gpgcheck: yes

- name: install kubelet
 - yum:
 - name: kubelet-1.14.0
 - state: present
 - update_cache: true

- name: install kubeadm
 - yum:
 - name: kubeadm-1.14.0
 - state: present

- name: start kubelet
 - service:
 - name: kubelet
 - enabled: yes
 - state: started

- hosts: master
 - become: yes
 - tasks:
 - name: install kubectl
 - yum:
 - name: kubectl-1.14.0
 - state: present
 - allow_downgrade: yes

The first play in the playbook does the following:

- Installs Docker, the container runtime.
- Starts the Docker service.
- Disables SELinux since it is not fully supported by Kubernetes yet.
- Sets a few netfilter-related `sysctl` values required for networking. This will allow Kubernetes to set iptables rules for receiving bridged IPv4 and IPv6 network traffic on the nodes.
- Adds the Kubernetes YUM repository to your remote servers' repository lists.
- Installs `kubelet` and `kubeadm` .

The second play consists of a single task that installs `kubectl` on your master node.

Note: While the Kubernetes documentation recommends you use the latest stable release of Kubernetes for your environment, this tutorial uses a specific version. This will ensure that you can follow the steps successfully, as Kubernetes changes rapidly and the latest version may not work with this tutorial.

Save and close the file when you are finished.

Next, execute the playbook:

```
ansible-playbook -i hosts ~/kube-cluster/kube-dependencies.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [all] ****
```

```
TASK [Gathering Facts] ****
```

```
ok: [worker1]
```

```
ok: [worker2]
```

```
ok: [master]
```

```
TASK [install Docker] ****
```

```
changed: [master]
```

```
changed: [worker1]
```

```
changed: [worker2]
```

```
TASK [disable SELinux] ****
```

```
changed: [master]
```

```
changed: [worker1]
```

```
changed: [worker2]
```

```
TASK [disable SELinux on reboot] ****
```

```
changed: [master]
```

```
changed: [worker1]
```

changed: [worker2]

TASK [ensure net.bridge.bridge-nf-call-ip6tables is set to 1] ****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [ensure net.bridge.bridge-nf-call-iptables is set to 1] ****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [start Docker] ****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [add Kubernetes' YUM repository] *****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [install kubelet] *****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [install kubeadm] *****

changed: [master]

changed: [worker1]

changed: [worker2]

TASK [start kubelet] ****

changed: [master]

changed: [worker1]

changed: [worker2]

PLAY [master] *****

TASK [Gathering Facts] *****

ok: [master]

TASK [install kubectl] *****

ok: [master]

PLAY RECAP ****

master	: ok=9	changed=5	unreachable=0	failed=0
worker1	: ok=7	changed=5	unreachable=0	failed=0
worker2	: ok=7	changed=5	unreachable=0	failed=0

After execution, Docker, `kubeadm`, and `kubelet` will be installed on all of the remote servers. `kubectl` is not a required component and is only needed for executing cluster commands. Installing it only on the master node makes sense in this context, since you

will run `kubectl` commands only from the master. Note, however, that `kubectl` commands can be run from any of the worker nodes or from any machine where it can be installed and configured to point to a cluster.

All system dependencies are now installed. Let's set up the master node and initialize the cluster.

Step 4 — Setting Up the Master Node

In this section, you will set up the master node. Before creating any playbooks, however, it's worth covering a few concepts such as *Pods* and *Pod Network Plugins*, since your cluster will include both.

A pod is an atomic unit that runs one or more containers. These containers share resources such as file volumes and network interfaces in common. Pods are the basic unit of scheduling in Kubernetes: all containers in a pod are guaranteed to run on the same node that the pod is scheduled on.

Each pod has its own IP address, and a pod on one node should be able to access a pod on another node using the pod's IP. Containers on a single node can communicate easily through a local interface. Communication between pods is more complicated, however, and requires a separate networking component that can transparently route traffic from a pod on one node to a pod on another.

This functionality is provided by pod network plugins. For this cluster, you will use Flannel, a stable and performant option.

Create an Ansible playbook named `master.yml` on your local machine:

```
vi ~/kube-cluster/master.yml
```

Add the following play to the file to initialize the cluster and install Flannel:

```
~/kube-cluster/master.yml
```



```

- hosts: master
  become: yes
  tasks:
    - name: initialize the cluster
      shell: kubeadm init --pod-network-cidr=10.244.0.0/16 >> cluster_initialized.txt
      args:
        chdir: $HOME
        creates: cluster_initialized.txt

    - name: create .kube directory
      become: yes
      become_user: centos
      file:
        path: $HOME/.kube
        state: directory
        mode: 0755

    - name: copy admin.conf to user's kube config
      copy:
        src: /etc/kubernetes/admin.conf
        dest: /home/centos/.kube/config
        remote_src: yes
        owner: centos

    - name: install Pod network
      become: yes
      become_user: centos
      shell: kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/a70459be0084506e4ec919aa1c114638878db11b/Dcflannel.yml >> pod_network_setup.txt
      args:
        chdir: $HOME
        creates: pod_network_setup.txt

```

Here's a breakdown of this play:

- The first task initializes the cluster by running `kubeadm init`. Passing the argument `--pod-network-cidr=10.244.0.0/16` specifies the private subnet that the pod IPs will be assigned from. Flannel uses the above subnet by default; we're telling `kubeadm` to use the same subnet.
- The second task creates a `.kube` directory at `/home/centos`. This directory will hold configuration information such as the admin key files, which are required to connect to the cluster, and the cluster's API address.
- The third task copies the `/etc/kubernetes/admin.conf` file that was generated from `kubeadm init` to your non-root **centos** user's home directory. This will allow you to use `kubectl` to access the newly-created cluster.

- The last task runs `kubectl apply` to install `Flannel` . `kubectl apply -f descriptor.[yaml|json]` is the syntax for telling `kubectl` to create the objects described in the `descriptor.[yaml|json]` file. The `kube-flannel.yml` file contains the descriptions of objects required for setting up `Flannel` in the cluster.

Save and close the file when you are finished.

Execute the playbook:

```
ansible-playbook -i hosts ~/kube-cluster/master.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [master] ****
```

```
TASK [Gathering Facts] ****
```

```
ok: [master]
```

```
TASK [initialize the cluster] ****
```

```
changed: [master]
```

```
TASK [create .kube directory] ****
```

```
changed: [master]
```

```
TASK [copy admin.conf to user's kube config] *****
```

```
changed: [master]
```

```
TASK [install Pod network] *****
```

```
changed: [master]
```

```
PLAY RECAP ****
```

```
master          : ok=5   changed=4   unreachable=0   failed=0
```

To check the status of the master node, SSH into it with the following command:

```
ssh centos@master_ip
```

Once inside the master node, execute:

```
kubectl get nodes
```

You will now see the following output:

Output

NAME	STATUS	ROLES	AGE	VERSION
master	Ready	master	1d	v1.14.0

The output states that the **master** node has completed all initialization tasks and is in a **Ready** state from which it can start accepting worker nodes and executing tasks sent to the API Server. You can now add the workers from your local machine.

Step 5 — Setting Up the Worker Nodes

Adding workers to the cluster involves executing a single command on each. This command includes the necessary cluster information, such as the IP address and port of the master's API Server, and a secure token. Only nodes that pass in the secure token will be able to join the cluster.

Navigate back to your workspace and create a playbook named **workers.yml** :

```
vi ~/kube-cluster/workers.yml
```

Add the following text to the file to add the workers to the cluster:

```
~/kube-cluster/workers.yml
```

```
- hosts: master
  become: yes
  gather_facts: false
  tasks:
    - name: get join command
      shell: kubeadm token create --print-join-command
      register: join_command_raw

    - name: set join command
      set_fact:
        join_command: "{{ join_command_raw.stdout_lines[0] }}"

- hosts: workers
  become: yes
  tasks:
    - name: join cluster
      shell: "{{ hostvars['master'].join_command }}" --ignore-preflight-errors all >> node_joined.txt"
      args:
        chdir: $HOME
        creates: node_joined.txt
```

Here's what the playbook does:

- The first play gets the join command that needs to be run on the worker nodes. This command will be in the following format: **kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash>** . Once it gets the actual command with the proper **token** and **hash** values, the task sets it as a fact so that the next play will be able to access that info.

- The second play has a single task that runs the join command on all worker nodes. On completion of this task, the two worker nodes will be part of the cluster.

Save and close the file when you are finished.

Execute the playbook:

```
ansible-playbook -i hosts ~/kube-cluster/workers.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [master] ****
```

```
TASK [get join command] ****
```

```
changed: [master]
```

```
TASK [set join command] *****
```

```
ok: [master]
```

```
PLAY [workers] *****
```

```
TASK [Gathering Facts] *****
```

```
ok: [worker1]
```

```
ok: [worker2]
```

```
TASK [join cluster] *****
```

```
changed: [worker1]
```

```
changed: [worker2]
```

```
PLAY RECAP *****
```

```
master          : ok=2   changed=1   unreachable=0   failed=0
worker1         : ok=2   changed=1   unreachable=0   failed=0
worker2         : ok=2   changed=1   unreachable=0   failed=0
```

With the addition of the worker nodes, your cluster is now fully set up and functional, with workers ready to run workloads. Before scheduling applications, let's verify that the cluster is working as intended.

Step 6 — Verifying the Cluster

A cluster can sometimes fail during setup because a node is down or network connectivity between the master and worker is not working correctly. Let's verify the cluster and ensure that the nodes are operating correctly.

You will need to check the current state of the cluster from the master node to ensure that the nodes are ready. If you disconnected from the master node, you can SSH back into it with the following command:

```
ssh centos@master_ip
```

Then execute the following command to get the status of the cluster:

```
kubectl get nodes
```

You will see output similar to the following:

Output

NAME	STATUS	ROLES	AGE	VERSION
master	Ready	master	1d	v1.14.0
worker1	Ready	<none>	1d	v1.14.0
worker2	Ready	<none>	1d	v1.14.0

If all of your nodes have the value `Ready` for `STATUS`, it means that they're part of the cluster and ready to run workloads.

If, however, a few of the nodes have `NotReady` as the `STATUS`, it could mean that the worker nodes haven't finished their setup yet. Wait for around five to ten minutes before re-running `kubectl get node` and inspecting the new output. If a few nodes still have `NotReady` as the status, you might have to verify and re-run the commands in the previous steps.

Now that your cluster is verified successfully, let's schedule an example Nginx application on the cluster.

Step 7 — Running An Application on the Cluster

You can now deploy any containerized application to your cluster. To keep things familiar, let's deploy Nginx using *Deployments* and *Services* to see how this application can be deployed to the cluster. You can use the commands below for other containerized applications as well, provided you change the Docker image name and any relevant flags (such as `ports` and `volumes`).

Still within the master node, execute the following command to create a deployment named `nginx`:

```
kubectl create deployment nginx --image=nginx
```

A deployment is a type of Kubernetes object that ensures there's always a specified number of pods running based on a defined template, even if the pod crashes during the cluster's lifetime. The above deployment will create a pod with one container from the Docker registry's [Nginx Docker Image](#).

Next, run the following command to create a service named `nginx` that will expose the app publicly. It will do so through a *NodePort*, a scheme that will make the pod accessible through an arbitrary port opened on each node of the cluster:

```
kubectl expose deploy nginx --port 80 --target-port 80 --type NodePort
```

Services are another type of Kubernetes object that expose cluster internal services to clients, both internal and external. They are also capable of load balancing requests to multiple pods, and are an integral component in Kubernetes, frequently interacting with other components.

Run the following command:

```
kubectl get services
```

This will output text similar to the following:

Output

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	1d
nginx	NodePort	10.109.228.209	<none>	80:nginx_port/TCP	40m

From the third line of the above output, you can retrieve the port that Nginx is running on. Kubernetes will assign a random port that is greater than `30000` automatically, while ensuring that the port is not already bound by another service.

To test that everything is working, visit `http://worker_1_ip:nginx_port` or `http://worker_2_ip:nginx_port` through a browser on your local machine. You will see Nginx's familiar welcome page.

If you would like to remove the Nginx application, first delete the `nginx` service from the master node:

```
kubectl delete service nginx
```

Run the following to ensure that the service has been deleted:

```
kubectl get services
```

You will see the following output:

Output

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	1d

Then delete the deployment:

```
kubectl delete deployment nginx
```

Run the following to confirm that this worked:

```
kubectl get deployments
```

Output

No resources found.

Conclusion

In this guide, you've successfully set up a Kubernetes cluster on CentOS 7 using Kubeadm and Ansible for automation.

If you're wondering what to do with the cluster now that it's set up, a good next step would be to get comfortable deploying your own applications and services onto the cluster. Here's a list of links with further information that can guide you in the process:

- [Pod Overview](#) - describes in detail how Pods work and their relationship with other Kubernetes objects. Pods are ubiquitous in Kubernetes, so understanding them will facilitate your work.
- [Deployments Overview](#) - this provides an overview of deployments. It is useful to understand how controllers such as deployments work since they are used frequently in stateless applications for scaling and the automated healing of unhealthy applications.
- [Services Overview](#) - this covers services, another frequently used object in Kubernetes clusters. Understanding the types of services and the options they have is essential for running both stateless and stateful applications.

Other important concepts that you can look into are [Volumes](#), [Ingresses](#) and [Secrets](#), all of which come in handy when deploying production applications.

Kubernetes has a lot of functionality and features to offer. [The Kubernetes Official Documentation](#) is the best place to learn about concepts, find task-specific guides, and look up API references for various objects.