DigitalOcean

How To Create a Kubernetes 1.10 Cluster Using Kubeadm on Ubuntu 16.04

15

Posted April 25, 2018   👁 34.2k   KUBERNETES   DOCKER   ANSIBLE   NGINX   UBUNTU 16.04

By: bsder

Not using **Ubuntu 16.04**? Choose a different version:

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.

Kubeadm 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 Kubeadm, 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/Mac OS/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 Ubuntu 16.04 with at least 1GB RAM. You should be able to SSH into each server as the root user with your SSH key pair.

- Ansible installed on your local machine. If you're running Ubuntu 16.04 as your OS, follow the "Step 1 - Installing Ansible" section in How to Install and Configure Ansible on Ubuntu 16.04 to install Ansible. For installation instructions on other platforms like Mac OS X or CentOS, 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 Ubuntu 16.04 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 `nano` or your favorite text editor:

```
$ nano ~/kube-cluster/hosts
```

Add 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

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

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.

The last line of the file tells Ansible to use the remote servers' Python 3 interpreters for its management operations.

Save and close the file after you've added the text.

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

Step 2 — Creating a Non-Root User on All Remote Servers

In this section you will create a non-root user with sudo privileges on all servers so that you can SSH into them manually as an unprivileged user. This can be useful if, for example, you would like to see system information with commands such as `top/htop`, view a list of running containers, or change configuration files owned by root. These operations are routinely performed during the maintenance of a cluster, and using a non-root user for such tasks minimizes the risk of modifying or deleting important files or unintentionally performing other dangerous operations.

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

```
$ nano ~/kube-cluster/initial.yml
```

Next, add the following *play* to the file to create a non-root user with sudo privileges on all of the servers. A play in Ansible is a collection of steps to be performed that target specific servers and groups. The following play will create a non-root sudo user:

~/kube-cluster/initial.yml

```
- hosts: all
  become: yes
  tasks:
    - name: create the 'ubuntu' user
      user: name=ubuntu append=yes state=present createhome=yes shell=/bin/bash

    - name: allow 'ubuntu' to have passwordless sudo
      lineinfile:
        dest: /etc/sudoers
        line: 'ubuntu ALL=(ALL) NOPASSWD: ALL'
        validate: 'visudo -cf %s'

    - name: set up authorized keys for the ubuntu user
      authorized_key: user=ubuntu key="{{item}}"
      with_file:
        - ~/.ssh/id_rsa.pub
```

Here's a breakdown of what this playbook does:

- Creates the non-root user `ubuntu`.

- Configures the `sudoers` file to allow the `ubuntu` user to run `sudo` commands without a password prompt.

- Adds the public key in your local machine (usually `~/.ssh/id_rsa.pub`) to the remote `ubuntu` user's authorized key list. This will allow you to SSH into each server as the `ubuntu` user.

Save and close the file after you've added the text.

Next, execute the playbook by locally running:

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

The command will complete within two to five minutes. On completion, you will see output similar to the following:

Output

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

TASK [Gathering Facts] ****
ok: [master]
ok: [worker1]
ok: [worker2]

TASK [create the 'ubuntu' user] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [allow 'ubuntu' user to have passwordless sudo] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [set up authorized keys for the ubuntu user] ****
changed: [worker1] => (item=ssh-rsa AAAAB3...
changed: [worker2] => (item=ssh-rsa AAAAB3...
changed: [master] => (item=ssh-rsa AAAAB3...

PLAY RECAP ****
master                      : ok=5    changed=4    unreachable=0    failed=0
worker1                     : ok=5    changed=4    unreachable=0    failed=0
worker2                     : ok=5    changed=4    unreachable=0    failed=0
```

Now that the preliminary setup is complete, you can move on to installing Kubernetes-specific dependencies.

Step 3 — Installing Kubernetes' Dependencies

In this section, you will install the operating system level packages required by Kubernetes with Ubuntu's package manager. These packages are:

- Docker - a container runtime. It 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.

- `kubectl` - 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:

```
$ nano ~/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
     apt:
       name: docker.io
       state: present
       update_cache: true

   - name: install APT Transport HTTPS
     apt:
       name: apt-transport-https
       state: present

   - name: add Kubernetes apt-key
     apt_key:
       url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
       state: present

   - name: add Kubernetes' APT repository
     apt_repository:
      repo: deb http://apt.kubernetes.io/ kubernetes-xenial main
      state: present
      filename: 'kubernetes'

   - name: install kubelet
     apt:
       name: kubelet
       state: present
       update_cache: true

   - name: install kubeadm
     apt:
       name: kubeadm
       state: present

- hosts: master
  become: yes
  tasks:
   - name: install kubectl
```

```
    apt:
      name: kubectl
      state: present
```

The first play in the playbook does the following:

- Installs Docker, the container runtime.

- Installs `apt-transport-https`, allowing you to add external HTTPS sources to your APT sources list.

- Adds the Kubernetes APT repository's apt-key for key verification.

- Adds the Kubernetes APT repository to your remote servers' APT sources list.

- Installs `kubelet` and `kubeadm`.

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

Save and close the file when you are finished.

Next, execute the playbook by locally running:

```
$ 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 [install APT Transport HTTPS] *****
ok: [master]
ok: [worker1]
changed: [worker2]

TASK [add Kubernetes apt-key] *****
```

```
TASK [add Kubernetes apt key]
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [add Kubernetes' APT 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]

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:

```
$ nano ~/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: ubuntu
      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/ubuntu/.kube/config
```

```
      remote_src: yes
      owner: ubuntu


  - name: install Pod network
    become: yes
    become_user: ubuntu
    shell: kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Docum
    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/ubuntu`. 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 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.[yml|json]` is the syntax for telling `kubectl` to create the objects described in the `descriptor.[yml|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 locally by running:

```
$ 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 ubuntu@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        v1.10.1
```

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 join the cluster.

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

```
$ nano ~/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 }} >> 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 by locally running:

```
$ 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 ubuntu@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        v1.10.1
worker1    Ready      <none>    1d        v1.10.1
worker2    Ready      <none>    1d        v1.10.1
```

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 run nginx --image=nginx --port 80
```

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 Ubuntu 16.04 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:

- Dockerizing applications - lists examples that detail how to containerize applications using Docker.

- 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.

By: bsder

♡ Upvote (15)   ⊏⁺ Subscribe

Editor:
Kathleen Juell

Introducing Projects on DigitalOcean

Organize your resources according to
how you work.

READ MORE

Related Tutorials

How to Use Traefik as a Reverse Proxy for Docker Containers on Ubuntu 18.04

An Introduction to the Kubernetes DNS Service

How to Deploy a Symfony 4 Application to Production with LEMP on Ubuntu 18.04

How To Test Ansible Roles with Molecule on Ubuntu 18.04

How To Provision and Manage Remote Docker Hosts with Docker Machine on Ubuntu 18.04

24 Comments

Leave a comment...

Log In to Comment

⌃ [valkyriestudios](#) *April 29, 2018*
♡
0 Brillian tutorial ! Thanks for this :D !

For anyone following the tutorial, some items were missing I feel ? In case you run into issues, locally I've made the following adjustments base on the official kubeadm docs (https://kubernetes.io/docs/setup/independent/install-kubeadm/):

initial.yml should also include an SSH restart near the end:

```
  - name: 'Restart SSH daemon'
    service:
      name: sshd
      state: restarted
```

dependencies.yml : locally I moved the 'apt-transport-https' to before the docker.io install:

```
    - name: 'Install docker and role dependencies'
      become: yes
      apt:
        name: "{{item}}"
        state: present
        install_recommends: false
      with_items:
        - "apt-transport-https"
        - "ca-certificates"
        - "software-properties-common"
        - "cron"
```

master.yml : Should include a check for cgroup drivers and should also reload system daemon and restart kubelet service before initializing the kubeadm:

```
- name: 'Configure cgroup driver used by kubelet on Master node'
  become: yes
  replace:
    path: /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
    regexp: 'cgroup-driver=systemd'
    replace: 'cgroup-driver="cgroupfs"'
    backup: yes

- name: 'Reload configs and restart kubelet'
  become: yes
  systemd:
    state: restarted
    daemon_reload: yes
    name: kubelet
```

Those are the changes I've made locally :), nice tutorial guys!

---

bsder *May 4, 2018*

1 Glad you found the article helpful, thank you for your comment. An `sshd` restart isn't strictly required unless you're already logged into the remote servers, in which case, terminating the existing session and starting another one should work fine.

The default Docker version in 16.04 should work without any tweaks to the `kubeadm.conf` file, I've verified this in a recent cluster. Can you let me know what error you get without the modification if possible?

---

valkyriestudios *May 5, 2018*

0 In regards to the cgroup adjustment and kubelet restart, you're right that it's not fully possible :) I was just following the kubeadm official docs to see if there was anything missing with this setup.

In regards to the sshd restart, I did have an issue with logging in afterwards, stating that there was an issue in regards to permissions. I'll try to reconstruct this tonight when I get home :)

---

valkyriestudios *May 5, 2018*

0 Ok it seems like it was the setup that I was using locally that was causing some issues. I'm running these playbooks together with a terraform setup that creates the droplets, and in some cases the droplet isn't fully booted up yet (giving issues with SSH)

valkyriestudios *April 30, 2018*

0 There is also an issue in the `master.yml` file. The shell command for the join pod network should pipe to a different output file to make sure ansible doesn't rerun this :

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documenta
```

should be

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documenta
```

---

bsder *May 4, 2018*

1 Thanks for the fix, the article will be updated soon.

---

ctal80 *May 16, 2018*

0 while running the command : "kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/v0.9.1/Documentation/kube-flannel.yml"

I'm getting the error "Unable to connect to the server: EOF " any idea what is broken or how I can debug it?

Thanks
Tal

---

valkyriestudios *May 17, 2018*

0 @bsder Just for completeness, there's one extra thing wrong with the setup (if you could adjust the blog post so other people don't fall into this rabbit hole). The fact that the blog uses ubuntu is an issue for kube-dns since it builds on the dns of the host system.

Ubuntu runs a dnsmasq internally, meaning that from the nodes inside of the cluster you won't be able to reach the outside world.

To fix this you need to add an extra configmap for kube-dns to specify upstream nameservers :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
```

```
      namespace: kube-system
  data:
    upstreamNameservers: |
      ["8.8.8.8", "8.8.4.4"]
```

For more information regarding this, here's some documentation
(https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/#configure-stub-
domain-and-upstream-dns-servers) and here's an issue on github talking about this (
https://github.com/coreos/flannel/issues/983).

<img> jstrat38 *June 13, 2018*

0 Hi, can you elaborate? How do I use the ConfigMap that you have listed above? I have read
   through the links you provided and was hoping you could shorten my time to figure this out.

   thanks!

<img> valkyriestudios *June 16, 2018*

0 Hi @jstrat38 glad to be of assistance :) in regards to this article the way you could do it is
   save the configmap to a file on local disk named something like `kube_dns.yml` and then
   right after the ansible task that says 'Copy admin.conf to users kube config' and before the
   'Install pod network', you add 2 tasks that 1) copy the local file to the droplet, 2) Install the
   file in kubernetes through kubectl. This looks like so :

```
   - name: 'Copy dns yml to master node'
     copy:
       src: ./assets/kube_dns.yml
       dest: $HOME/kube_dns.yml
       owner: ubuntu
       mode: 0755
       force: no

   - name: 'Install dns yml'
     become: yes
     become_user: ubuntu
     shell: kubectl create configmap kube-dns --from-file=$HOME/kube_dns.yml --
     args:
       chdir: $HOME
       creates: $HOME/kube_dns_setup.txt
```

This will setup 'upstream nameservers' that will then be usable by each pod :)

Please replace the './**assets/kube_dns.yml**' to the path on your local filesystem where you stored the file :)

Hope it helps !

___

jstrat38 *June 18, 2018*

0 I followed your instructions and everything worked!

   thanks,

___

lucasTiber *July 10, 2018*

0 Following this guide, kubelet isn't starting up and has a status of 255.

Also: `failed to load Kubelet config file /var/lib/kubelet/config.yaml, error failed to read kubelet config file "/var/lib/kubelet/config.yaml", error: open /var/lib/kubelet/config.yaml: no such file`

is in the logs for the systemd service

___

seanturner026 *July 12, 2018*

0 Hi,

I am wondering how to configure persistent volumes on the worker nodes. Basically, I get the following errors when I try and create my kubernetes cluster with `kubectl create -f app.yaml`:

```
kubectl describe pods
```

```
  Conditions:
    Type           Status
    PodScheduled   False
  Volumes:
    tmdir:
      Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in t
      ClaimName:  tmdir-bftx-0
      ReadOnly:   false
    appdir:
      Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in t
      ClaimName:  appdir-bftx-0
      ReadOnly:   false
    tmconfigdir:
      Type:       ConfigMap (a volume populated by a ConfigMap)
      Name:       bftx-config
```

```
    Optional:  false
  appconfigdir:
    Type:       ConfigMap (a volume populated by a ConfigMap)
    Name:       app-config
    Optional:  false
  socksdir:
    Type:       EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
  default-token-rb5hr:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-rb5hr
    Optional:    false
QoS Class:        BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s

Events:
  Type     Reason           Age              From              Message
  ----     ------           ----             ----              -------
  Warning  FailedScheduling  5s (x11 over 32s)  default-scheduler  pod has unbound
```

kubectl describe pvc

```
  Name:          appdir-bftx-0
  Namespace:     default
  StorageClass:
  Status:        Pending
  Volume:
  Labels:        app=bftx
  Annotations:   volume.alpha.kubernetes.io/storage-class=anything
  Finalizers:    [kubernetes.io/pvc-protection]
  Capacity:
  Access Modes:
  Events:
    Type    Reason          Age              From                        Message
    ----    ------          ----             ----                        -------
    Normal  FailedBinding   2m (x103 over 27m)  persistentvolume-controller  no persi


  Name:          tmdir-bftx-0
  Namespace:     default
  StorageClass:
  Status:        Pending
  Volume:
  Labels:        app=bftx
```

```
Annotations:      volume.alpha.kubernetes.io/storage-class=anything
Finalizers:       [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
Events:
  Type    Reason         Age               From                        Message
  ----    ------         ----              ----                        -------
  Normal  FailedBinding  2m (x103 over 27m)  persistentvolume-controller  no persi
```

I'm looking at using something called rexray, but I'm not so sure I'll be able to get the cluster using it. Ideally, the cluster would be able to tap into the ssd storage that is bundled with each node.

HarisF *July 17, 2018*

0 Thanks for the brilliantly authored tutorial. It really helped me learning and setting up the k8s environment and deployment of services. However, I have a setup of two Ubuntu VMs (1-Master, 1-Worker) and setting up a communication via SSH between VMs and altering the configs in k8x was a real pain. But anyway it helped a lot.

---

chicken231 *July 27, 2018*

0 Hi folks.
I created a GitLab Repo that contains the Ansible mentioned in the tutorial, as well as some of the tweaks mentioned in the comments. Feel free to help improve it:
https://gitlab.com/chicken231/ubuntu-kubernetes-ansible

---

erikbotta *August 31, 2018*

0

```
TASK [copy admin.conf to user's kube config] ********************************
fatal: [master]: FAILED! => {"changed": false, "msg": "Source /etc/kubernetes/admi
    to retry, use: --limit @/Users/erik/Documents/repos/kube-cluster/master.retry
```

---

emanymton *September 20, 2018*

0 hello,

i'm pretty sure this tutorial is the one i need but im facing issues due to my lack of experience.

do the servers (=workers) can be VM guests ?
if so, i can't succeed to find the fine ip addresses for the worker nodes.

thanks in advance

△
♡ **shadow2017** *September 20, 2018*
0   Super, I follow your instructions to setup a cluster with one master and two worker nodes. But
    when running .Step 4 — Setting Up the Master Node, there are some fatal errors as showing
    below . Did I do something wrong?

    TASK [initialize the cluster]
    ****

    fatal: [master]: FAILED! => {"changed": true, "cmd": "kubeadm init --pod-network-
    cidr=10.244.0.0/16 >> cluster*initialized.txt", "delta": "0:00:00.741279", "end": "2018-09-20
    17:58:44.206018", "msg": "non-zero return code", "rc": 2, "start": "2018-09-20 17:58:43.464739",
    "stderr": "\t[WARNING SystemVerification]: docker version is greater than the most recently
    validated version. Docker version: 17.03.2-ce. Max validated version: 18.06\n[preflight] Some
    fatal errors occurred:\n\t[ERROR KubeletVersion]: the kubelet version is higher than the control
    plane version. This is not a supported version skew and may lead to a malfunctional cluster.
    Kubelet version: \"1.12.0-rc.1\" Control plane version: \"1.11.3\"\n[preflight] If you know what you
    are doing, you can make a check non-fatal with* `--ignore-preflight-errors=...` *",
    "stderr*lines": ["\t[WARNING SystemVerification]: docker version is greater than the most recently
    validated version. Docker version: 17.03.2-ce. Max validated version: 18.06", "[preflight] Some
    fatal errors occurred:", "\t[ERROR KubeletVersion]: the kubelet version is higher than the control
    plane version. This is not a supported version skew and may lead to a malfunctional cluster.
    Kubelet version: \"1.12.0-rc.1\" Control plane version: \"1.11.3\"", "[preflight] If you know what you
    are doing, you can make a check non-fatal with` --ignore-preflight-errors=... `"], "stdout":
    "", "stdout_lines": []}

△
♡ **XaeroDegreaz** *October 5, 2018*
0   For some reason, my master node stays in a NotReady state. It's been like 20 minutes and it's still
    stuck. I basically copy and pasted all of the configs from this tutorial and ran them in order.

    Things I've tried:

    1. Applied @valkyriestudios DNS suggestions while grasping for straws.

    2. Changed the Flannel version to 0.10.0

    3. I've tried using @chicken231 setup from the repository they provided

    I've used `kubectl describe nodes` and can see a message in the `Unschedulable`
    `Conditions` table at the top of the output:

    ```
    runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady
    message:docker: network plugin is not ready: cni config uninitialized
    ```

There's also a `Starting kube-proxy` message in the events table at the bottom.

Really not sure what else to try here :(

OS: Ubuntu 16.04.4 x64
Kube: v1.12.0

Thanks in advance!

---

fumblehool *October 12, 2018*

0   @XaeroDegreaz You can try installing `weave-kube`
Command - `kubectl apply --filename https://git.io/weave-kube-1.6`
It works for me :)

---

chanoch *October 22, 2018*

0   I had to apply a patch to make networking work as there is a problem with 1.12.

kubectl -n kube-system apply -f
https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd264
flannel.yml

This patches kubectl to 1.12.1 and make it work for me.

Also note that if you follow the prereqs properly and enable ufw, you will need to allow
incoming traffic to port 6443 on the manager as by default only ssh is allowed

sudo ufw allow from {worker_ip} to any port 6443

This will allow the worker to access port 6443 on any of the manager's interfaces. You will
need to do this for each of the workers by their ip. You can open 6443 to any traffic to be able
to join new workers more easily but that feels unsafe.

◄   ►

XaeroDegreaz *October 22, 2018*

0   Thanks, I will try these suggested things. Also, I did not bother with UFW -- I can just setup
the firewall stuff using Digital Ocean's firewall system and use the Droplet tags for ingress
port rules.

Sambho *October 11, 2018*

0   Step #4 has got failed saying -
fatal: [master]: FAILED! => {"changed": false, "msg": "Source /etc/kubernetes/admin.conf not found"}

Checked the cluster_initialized.txt file generated in the master. It has just the following log -

cat cluster_initialized.txt
[init] using Kubernetes version: v1.12.1
[preflight] running pre-flight checks

But..if I run the command, `kubeadm init --pod-network-cidr=10.244.0.0/16 >> cluster_initialized.txt` manually in the master under /root; and after that if I executed step #4 ansible command, it is succeeded.
Hmm. May I know what was the problem with the ansible command earlier? I wanted to know as the manual steps should be avoided for the better automation using ansible....

---

rootshell2 *October 22, 2018*

Thanks for sharing!
I did use this tutorial to make my cluster running. The instances have been created by Terraform, check it out:

https://github.com/rootshell2/kubernetes-cluster

Copyright © 2018 DigitalOcean™ Inc.

Community   Tutorials   Questions   Projects   Tags   Newsletter   RSS

Distros & One-Click Apps   Terms, Privacy, & Copyright   Security   Report a Bug   Write for DOnations   Shop